Object-oriented Software for Industrial Robots

Dissertation in partial fulfillment of the grade of Doctor of Natural Sciences
of the Faculty of Applied Computer Science
of the University of Augsburg by

Andreas Angerer

March 4, 2014

Erstgutachter:	Prof. Dr. Wolfgang Reif
Zweitgutachter:	Prof. Dr. Bernhard Bauer
Mündliche Prüfung:	28.05.2014

Danksagung

Für die Betreuung dieser Arbeit danke ich herzlich Herrn Prof. Dr. Wolfgang Reif. Er gab mir die Möglichkeit, mich mit diesem faszinierenden Thema an der Universität Augsburg auseinander zu setzen, ließ mir dabei viele Freiheiten und verhinderte gleichzeitig wann immer nötig das Abdriften in die falsche Richtung. Viele der Ergebnisse dieser Arbeit sind gemeinsam in einem hervorragenden Team aus Kollegen und gleichzeitig Freunden entstanden: Alwin Hoffmann, Andreas Schierl, Michael Vistein, Ludwig Nägele und Miroslav Macho danke ich für die offenen, konstruktiven und spannenden Diskussionen und den Zusammenhalt, mit dem wir gemeinsam alle Projekttreffen, Paper-Deadlines und sonstige Herausforderungen gemeistert haben. Für die guten Anregungen in Oberseminaren und auf den Lehrstuhlklausuren danke ich den anderen Kollegen am Lehrstuhl für Softwaretechnik sowie Herrn Prof. Dr. Alexander Knapp.

Mit Markus Bischof und Alexander Chekler von KUKA Laboratories sowie Franz Stegmair und Sebastian Pieniack von MRK Systeme hatte ich im Rahmen des Forschungsprojekts SoftRobot in vielen Projekttreffen die Gelegenheit, über Roboter, Software und alle damit verbundenen Herausforderungen zu diskutieren - danke für alle Ideen und Anregungen! Darüber hinaus danke ich auch Herrn Prof. Dr. Frank Ortmeier, der maßgeblich für die Entstehung des Forschungsprojekts verantwortlich war.

Während der Entstehung der Dissertation durfte ich mit vielen engagierten Studenten zusammenarbeiten, die mit ihrer Arbeit am Lehrstuhl auch meiner Forschung geholfen haben. Besonders danke ich Andreas Bareth, Constantin Wanninger, Alexander Pöppel, Christian Böck und Remi Smirra für ihre Beiträge. Mit ihrer Hilfe sind tolle Anwendungsbeispiele und Software-Tools entstanden.

Meiner Familie und meinen Schwiegereltern danke ich für das viele Verständnis und dafür, daß sie mich immer wieder motiviert haben. Das haben auch meine Freunde Alexander und Bernadette Rieger sowie Florian Häglsperger immer wieder geschafft, sei es durch Nachfragen, Mut zusprechen oder auch mal eine spontane Essenseinladung – danke dafür! Sehr dankbar bin ich außerdem Andreas Schierl, Bernadette Rieger, Florian Häglsperger und ganz besonders Dr. Hella Seebach für das aufmerksame Korrekturlesen dieser Ausarbeitung. Mit ihren Anregungen bekam die Dissertation den letzten Schliff.

Mein größter Dank geht an meine Frau Veronika. Ohne sie hätte ich weder meinen Weg gefunden noch die Kraft gehabt, ihn zu gehen und daran zu wachsen. Danke für deine unendliche Geduld, dein tiefes Verständnis und Dein großes Vertrauen in mich!

Abstract

Industrial robots are very flexible machines that can perform operations with high speed and sub-millimeter accuracy. However, those robots are bound to proprietary programming languages, which often leads to limited reusability of applications and causes high effort for coordinating teams of robots or integrating the robots with other systems.

On the other hand, the area of software engineering has experienced significant progress in the last decades. Besides structured development processes and modeling paradigms, large ecosystems of programming languages, frameworks and all kinds of libraries have been created by communities of developers.

The main contribution of this thesis is the design of an object-oriented framework for industrial robotics applications. This design is language independent, allowing it to be realized based on any modern software platform and programming language. The main challenge in the context of industrial robotics is the need for deterministic execution of operations that control hardware devices, and to guarantee that tight timing bounds are held during execution. The presented design meets these requirements without compromising flexibility and expressiveness.

The software design is applied to several concrete hardware devices and evaluated based on three application examples. The thesis demonstrates that the design is able to fulfill a large set of requirements of current and future industrial robotics applications and thus presents a significant contribution to software engineering for industrial robotics.

Contents

Chapter 1

Industrial Robotics

The first industrial robot that was put into service was a machine called *Unimate*, which was installed in a General Motors plant in 1961 for extracting parts from other machines [1]. The invention of the industrial robot is attributed to George C. Devol, who filed a patent on a "programmed article transfer" [2] in 1954 and later co-founded the company Unimation Inc., which produced the Unimate robots.

However, in recent years an ingenious invention started to attract attention, which was published 16 years before George C. Devol filed his patent. This invention, called the "Robot Gargantua", is to date the first known device that conforms to modern definitions of the term *industrial robot*. The Robot Gargantua was built from parts of the *Meccano* model building system and published in the Meccano Magazine [3] in the year 1938. It is a crane-like device with five controllable degrees of freedom – and it is programmable by a so-called *robot unit*.

The Robot Gargantua and its robot unit are depicted in Fig. 1.1. The robot unit is driven by the same single (!) motor that drives all movements of the crane. Griffith P. Taylor, who constructed the device, describes the mechanism in this way:

> "Its central feature is a roll of paper punched with holes set out on a pre-arranged system. The roll resembles on a miniature scale those used for operating player pianos. It is drawn slowly over a brass drum and there passes under a row of spring brushes, which are connected in separate electric circuits and press lightly on the paper. When a hole passes beneath one of the brushes, this makes contact with the drum, and so completes the electric circuit through it. This current operates a solenoid that is used to move one of the control levers of the crane by means of a special differential drive operated by the crane motor." (Griffith P. Taylor, [3])

To connect the robot unit to the crane, all five mechanical levers that operate the crane's movement are aligned to the base of the crane. Taylor mentions that his robot unit is able

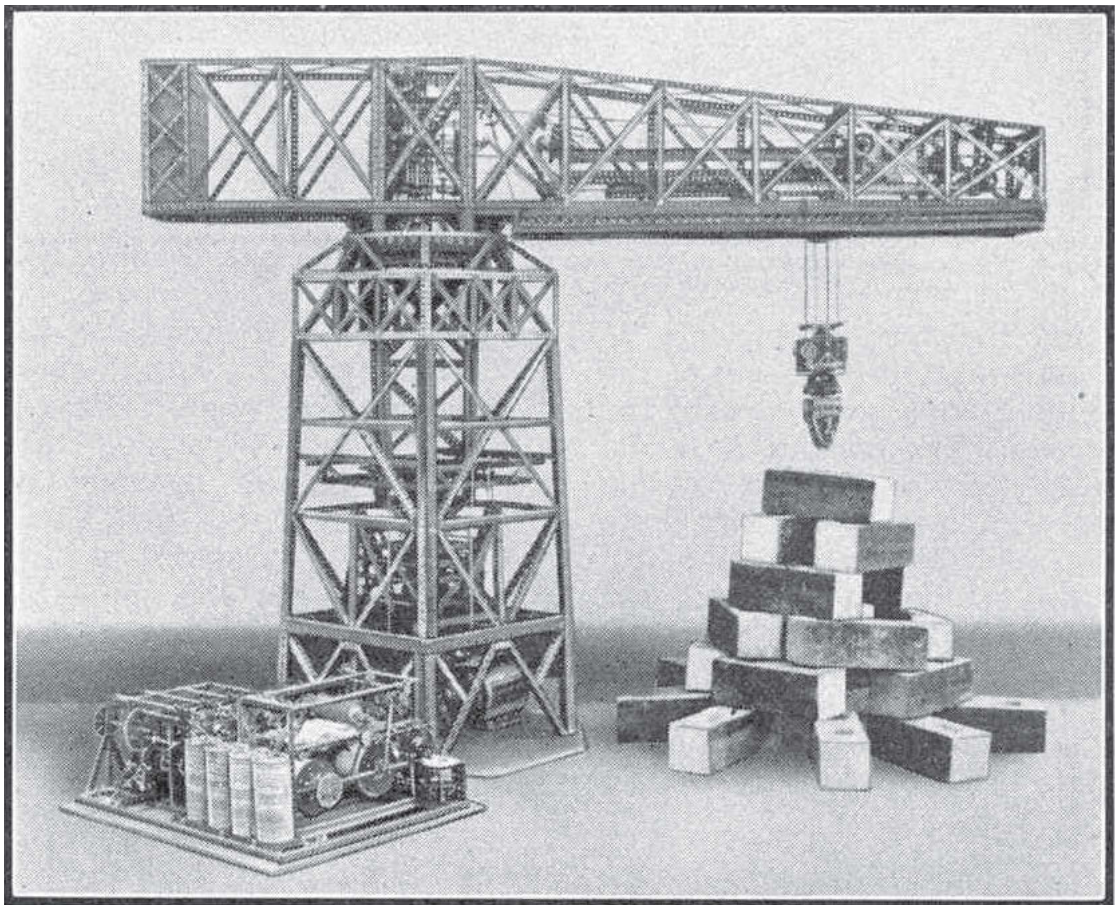

Figure 1.1: The *Robot Gargantua*, a crane with five degrees of freedom, programmable by a *robot unit* (bottom left). Image source: [3]

to control "not only the crane illustrated here, but also any other type of machine, such as an excavator or a dragline, that incorporates not more than five different operations".

Besides various outstanding mechanical features such as mechanical limiters for each movement, levers that are locked in one of three positions (forward, neutral, reverse) and a lifting tackle that is able to grab and rotate itself, the robot unit is a very unique feature that can be considered significantly ahead of its time. Konrad Zuse had just constructed the mechanical Z1 [4], the first binary computer, in 1937. The term *robot* was at that time mostly a concept of science fiction, like in Karel Čapek's drama "R.U.R. – Rossum's Universal Robots" [5] from the year 1921.

Programming the Robot Gargantua is a complicated process. It is described briefly in the original Meccano Magazine article by Griffith P. Taylor and more in detail by Chris Shute [6], who rebuilt the complete mechanism in 1998. For each operation the crane should perform (e.g. moving the gripping device to a certain position in order

to grab an item), the number of motor revolutions required per degree of freedom has to be determined. Holes in the paper tape trigger events like starting and stopping a motion in a certain degree of freedom, which can be done for multiple degrees of freedom in parallel. Thus, holes have to be stamped whenever a certain amount of predetermined motor revolutions has passed. The robot unit supports the operator in this process by providing a mechanical rotation counter. To prevent cumulative errors, the crane's ability to stop at mechanical limits can be employed: the program can move the crane into these mechanical limits deliberately from time to time to perform reference movements. As the whole mechanism of the Robot Gargantua only employs a single motor that drives the crane as well as the robot unit, it is ensured that the program flow is always synchronized to the actual speed of the mechanical manipulator.

The great effort that is put into the elaborate design of the robot unit demonstrates the potential that researchers and inventors have recognized in reprogrammable automatons even at that time. Since then, the progress in computer science has dramatically influenced the characteristics of all kinds of machines, including robots. Industrial robots are used intensively in the manufacturing industry since more than fifty years now. A report by the International Federation of Robotics (IFR) [7] estimates the worldwide stock of operational industrial robots to be in the range of 1.2 million units. When considering the robot density, Germany is among the top-ranked countries with 273 units per 10.000 persons in the manufacturing industry in general. Figures are significantly higher in the automotive industry, where 1.133 robots per 10.000 employes are installed in Germany. In Japan, the density in the automotive industry is even higher with 1.562 robots per 10.000 employes.

Developing programs for today's industrial robots is of course significantly easier compared to the beginnings. Not only is the representation of programs much more convenient (text or graphical notations on computers instead of holes in paper), but also the basic programming principles have become significantly more intuitive. For example, motions can be described by geometric paths in space rather than in terms of motor revolutions. Still, software development for industrial robots is an active research field with an immense potential to help in building new generations of robots that are much more flexible, easier to program and more efficient to use. This thesis presents a significant contribution to the field of software engineering for industrial robotics that can help to realize this 'software push'.

1.1 Trends and Challenges in Industrial Robotics

As the robot density numbers mentioned in the previous chapter have indicated, industrial robots are much more established in the automotive industry (1.133 robots per 10.000 workers) than in all other industries (147 robots per 10.000 workers) in Germany. The quotient of both densities is even higher for other countries in the world. Thus, these other industries are often termed *General Industry*, which includes everything but the automotive industry.

The automotive industry is the dominating market for industrial robots today, as stated above. However, the IFR states in their World Robotics 2013 report [7] that "the potential for robot installations in the non-automotive industries is still tremendous", and Hägele et al. remark that the "use of robots in small and medium-sized manufacturing is still tiny" [1]. Thus, there are obviously factors limiting the wide use of industrial robots in the General Industry. Some of the commonly identified challenges are outlined below.

Besides economic factors like the initial and running costs of a robot system, the potential flexibility of industrial robots should be better exhausted. On the one hand, this includes tighter integration of sensors. Pires states that "external sensing should be used for new types of motions or for handling unknown variation." [8]. Hägele et al. claim that "Measuring devices mounted on robots and increased use of sensor systems and RFID-tagged parts carrying individual information contributes to better dealing with tolerances in automated processes." [9]. Vision and force sensors are considered the most important types in this context by the authors. On the other hand, the means of programming robots today are agreed to be a problematic factor. According to Pires, "the typical teaching method is a tedious and time-consuming task that requires a considerable technical expertise" [8]. Hägele et al. stress the importance of "development of methods and tools for instructing and synchronising the operation of a group of cooperative robots at the shop-floor" [9]. In this context, the composability of subsystems, in particular of software components, is seen as a prerequisite for dividing complex problems into simpler parts [1].

Another general trend is the emerging of *service robots*. According to the World Robotics 2013 report, the sale of service robots for personal and domestic use increased by 20% to about 3 million units in 2012 [7]. The most important types are, according to the report, vacuum and floor cleaning robots, lawn-mowing robots and all kinds of entertainment and leisure robots. Research has enabled more advanced service robots to autonomously fetch objects [10], open doors [11], and even cook pancakes [12]. The robotics industry is following this trend and is envisioning products like e.g. robotic co-workers, logistics robots and inspection robots [13]. In contrast to today's industrial robots, these kinds of service robots require a certain degree of autonomy and thus cognitive capabilities. As of today, most commercial service robots are designed to fulfill a particular task with limited autonomy and few to no reprogrammability.

The future of commercial robots could lie in between the classic, inflexible industrial robots and the autonomous, but mostly single-purpose service robots of today. To be multifunctional, service robots need to be 'programmable' in some way. On the other hand, industrial robots could profit from better sensorial and cognitive abilities, which would potentially lead to less effort for adapting them to new tasks. The key to success of future robots in much more areas – professional as well as domestic – lies in the software that transforms an arrangement of mechanical and electronic elements into a mechatronic device.

The research on this thesis was performed for the most part in the context of traditional industrial robots. Cognitive capabilities are of less importance in this scope. However,

as this thesis will show, it is the software design that makes the difference between an inflexible machine suitable only for mass production tasks, and a flexible, easily reprogrammable manufacturing assistant.

1.2 Motivation and Goal of this Work

Software engineering has experienced significant progress in the last decades. The emerging of software development processes, new modeling paradigms, modeling and programming languages and many tools, libraries and platforms have contributed to the goal of making software development an engineering discipline. The emerging of international standards, like for agreed terms (ISO/IEC/IEEE 24765:2010 [14]) and common knowledge (ISO/IEC TR 19759:2005 [15]) in the context of software engineering are a further push in this direction.

Robotics, "the science and technology of robots" [16], has agreed foundations considering robot mechanics, sensing, planning, and control [16] and has developed "consolidated methodologies and technologies of robot structures (...), sensing and perception (...), manipulation and interfaces" [16]. These foundations enable companies like ABB Robotics and the KUKA Robot Group to build and sell industrial robots since the mid 1970s [17][18].

Software development for industrial robots is considered by far less mature than both general software engineering and the mechanical and control part of robotics. Both disciplines have evolved for the most part separate from each other in the last decade. In particular, most software ecosystems for industrial robots are highly proprietary and are developed by the robot manufacturers themselves, or by a small set of companies that integrate robots in customer-specific automation applications (so-called system integrators). Due to this state of software development for industrial robots, robot customers can in practice hardly profit from tools and methods developed by software engineering and its large community, like object-oriented design, service-oriented architectures, component-based software engineering, testing frameworks, debugging tools, IDE platforms and many more.

The motivation of this work is to push the limits of software engineering in industrial robotics. This thesis presents the design of a novel, object-oriented software framework for industrial robotics applications. The presented design, in a nutshell, has the following unique properties: It is *programming language independent* and thus can be realized based on any object-oriented language. It is *manufacturer independent* and thus usable for various kinds of robots. The design is *modular*, which ensures extensibility and adaptability to fit a vast set of use cases in the industrial robotics domain. It is *flexible* and can address the future challenges of industrial robotics. Finally, it is *hard real-time compliant*, allowing for superior control performance. In sum, the presented software design has the potential for a new software standard in the robotics industry.

The property of programming language independence is of central importance and deserves further explanation. The choice of a programming language is nowadays not

only the choice for a certain programming paradigm or a specific syntax, but also the choice for an *ecosystem* that accompanies the language. This ecosystem comprises many things: from the available compilers, interpreters or virtual machines, over the development environments, the documentation, the debugging and profiling tools to the libraries and frameworks created by the language developer/provider *and* the community. Note that the community can play a very important role, as those developers often provide a vast amount of libraries and documentation that can significantly speed up software development. Finally, the choice of a programming language might also depend on its interoperability with other languages. For example, the Java language integrates the Java Native Interface (JNI) that can be used to call any *native code*, i.e. code that has been compiled from languages like C++ for a certain hardware and operating system. There are further community-driven projects that e.g. achieve compatibility between Java and Python (Jython[1], Py4J[2]), Java and C# (IKVM.NET[3]) and Java and Ruby (JRuby[4], Ruby Java Bridge[5]).

Coming back to software engineering for industrial robotics, the feature of programming language independence is not easy to achieve. On the one hand, execution performance is critical in some parts of robot control software. On the other hand, flexibility is desired, e.g. for programming complex operations that involve sensor feedback. Most of today's industrial robot controllers try to provide an appropriate abstraction by a proprietary, manufacturer-dependent programming language. The controllers' software architecture usually leaves developers no choice of programming language and thus forces them to use the ecosystem provided by the manufacturer. It is obvious that these ecosystems cannot compete in features and tools with those of popular general purpose programming languages and their respective communities.

1.3 Main Contributions

During the work on this thesis, a set of results has been achieved that raise the state of the art in software development for industrial robot systems by a certain degree. This section gives an overview of the research contributions of this work.

In this thesis, an **extensive object-oriented model of robotic devices (Device model)** is introduced. This model allows for integrating all kinds of robotic devices, from single rotational or translational joints to complex systems consisting of multiple kinematic chains. It is on the one hand detailed and flexible enough to express subtle specialties of various devices, and on the other hand generic enough to model a large variety of robotic entities. In this work, devices like single robot joints, robot arms, tools and even robots-of-robots have been modeled using this device model. In other work, the

[1] http://www.jython.org
[2] http://py4j.sourceforge.net
[3] http://www.ikvm.net
[4] http://jruby.org
[5] http://rjb.rubyforge.org/

model has been and is being employed to cover mobile robot platforms, arm-platform systems and also flying robots.

For controlling all kinds of robotic devices, this work presents a **model of hard real-time device operations and their combination to complex commands (Command model)**. The model distinguishes between fine-grained basic operations and complex commands that can be flexibly composed from single operations and other commands. Its generality makes it suitable for controlling all robotic devices that are modeled using the abovementioned device model.

To support reactiveness in controlling devices, a **model of system states and events (State model)** is introduced as well. It allows for capturing important parts of the state of a robot system and react to changes of this state. By integrating such states in device commands, the behavior of robotic devices can be adapted to their environment. The state model is extensible so that new sources of states can be integrated, and it allows for flexibly combining single state entities to express new more general or more specific states.

An important source for determining a system's state are sensors. Therefore, an important contribution of this work is a **model of sensors and sensor data processing (Sensor model)**. This model can be used to gather data from raw, physical sensors as well as combine such sensors to create more complex and meaningful sources of data. The sensor model and the state model are tightly integrated, enabling reactive device control based on sensor data as well.

In robotics applications, a **powerful model for describing spatial relationships between entities (World model)** is of great importance and is presented in this work. On the one hand, this model can be used to capture the relationships between static parts of a robotic system and its environment. On the other hand, the world model also supports dynamic spatial relations that are controlled or that can be measured. Like all other models, the world model is extensible so that new kinds of relations can be defined. Furthermore, the world model integrates into the sensor model, which allows for measuring geometric data, merging it with data from other sensors and react to changes in this data.

To guarantee timing bounds for various calculation aspects of the abovementioned models, an **adapter concept for connecting the device, command, state, sensor and world model to a real-time capable runtime environment (runtime adapter)** is introduced. This allows for using different real-time platforms for interfacing hardware devices and executing control command instances created using the models. Due to the fine-grained, compositional character of the models, adapters for concrete platforms can be designed in a generic and extensible way so that vertical integration of new hardware devices is possible with minor effort.

To ease application development for typical industrial robotics use cases, a **simple and elegant model of typical robot activities (Activity model)** is presented. It employs the flexibility and expressiveness of command, state, sensor and world model to

offer application developers pre-configured, frequently used, yet configurable and combinable programming primitives. Those primitives have a distinct execution semantics that fits typical needs of industrial robot applications well.

A **generic extension mechanism** defines an interface for registering new kinds of devices, sensors, activities and real-time execution platforms. This mechanism provides developers willing to create extensions with the means to provide those extensions to application developers in a uniform way.

By separating application and configuration logic and integrating an elaborate **configuration management and application lifecycle** mechanism, reusability of robotic applications is increased. Transferring applications to new kinds of devices and other real-time platforms is easily possible.

The **development of domain-specific tools and methods** is substantially simplified by employing modern software engineering platforms and paradigms. This work presents case studies of a novel Integrated Development Environment (IDE) for robotic applications, a graphical language for specifying robot commands and an interpreter for a proprietary robot language on top of the proposed software platform.

1.4 Structure of this Work

Chapter 2 starts with an introduction of basic concepts of (industrial) robotics that are used throughout this work. To give an impression of the way industrial robots are programmed today, the character of the KUKA Robot Language and its features are presented in this chapter as well.

During the work on this thesis, a set of applications served as testbed for new concepts and were created to demonstrate the usefulness of the developed approach. Chapter 3 presents three applications that challenge the software development capabilities of existing robot controllers. Together with those examples, the robot hardware that was used for evaluation is presented.

The results presented in this work originate from the SoftRobot project, a joint research effort by industry and academia. Early in the project, a general architectural approach was developed which is referred to as the SoftRobot architecture. This architecture separates robot software in a low-level hardware control layer and an object-oriented layer for application development. Chapter 4 presents the requirements that the SoftRobot architecture has to fulfill, gives a detailed overview of its structure and states the goals of the object-oriented application development layer, which is the main focus of this thesis.

Chapter 5 introduces the structure of the Robotics API, the general object-oriented framework for robotics applications, and clarifies its exact role in the SoftRobot architecture. In contrast to today's robot controllers, which use proprietary programming languages to achieve determinism, the Robotics API can be implemented in any modern general-purpose language.

The Robotics API defines a core component, which provides a generic model of robotic devices and real-time critical operations those devices can perform. The design of this core component, its extension mechanisms and the interface to a low-level hardware control layer is presented in Chapter 6.

In order to describe interactions of robots with the physical world, robotics applications require a model of the world. Chapter 7 introduces the design of the Robotics API's world model component and its relationship to the core component.

To provide an intuitive programming interface to application developers which supports frequently used operation patterns, the Activity model component was developed as part of the Robotics API and is presented in Chapter 8.

The Chapters 9 and 10 demonstrate how the generic concepts of the Robotics API's core, world model and activity model components can be used to model concrete robot tools and robot arms and their operations.

To illustrate the usefulness of the results presented in this work, Chapter 11 demonstrates the realization of the three application examples with the Robotics API. Furthermore, the complexity and performance of the Robotics API's reference implementation and the degree to which the initial requirements could be realized are assessed.

During the work on this thesis, various software tools have been developed that assist in application development with the Robotics API. Three important tools are presented in Chapter 12. They are based on the Eclipse platform, a popular IDE for various general purpose languages, and extend this platform with robotics-specific features.

Chap. 13 concludes the results, gives an outlook and comments on the lessons learned during the work on this thesis.

Chapter 2

Basics

Summary: This chapter introduces the basic concepts of (industrial) robotics that are used throughout this work and gives an overview of how industrial robots are typically programmed today. Based on the KUKA Robot Language (KRL), frequently used programming concepts and the strengths and weaknesses of proprietary robot programming languages are evaluated.

The first part of this chapter defines terms commonly used in robotics in general and industrial robotics in particular, which should be familiar to robotics experts who may thus skip this part. The second part introduces the KUKA Robot Language (KRL) as a prominent example of a proprietary robot programming language. The goal is to demonstrate two aspects: On the one hand, KRL provides special concepts for industrial robot programming, some of which cannot be found in other programming languages. On the other hand, KRL has severe weaknesses that affect maintainability, reusability and composability of robot applications.

2.1 Important Robotics Concepts

This section introduces basic concepts of (industrial) robotics that are a prerequisite for other parts of this thesis. It will not go into detail about the foundations and definitions of all concepts, but rather focuses on creating an understanding of their meaning. The section is based on work by Siciliano et al. [19] and Waldron et al. [20], which is advised for further reading.

A commonly used definition of a *robot* is according to Siciliano et al. [19] this one:

> "A Robot is a reprogrammable multifunctional manipulator designed to move materials, parts, tools or specialized devices through variable programmed motions for the performance of a variety of tasks." ([19], p. 17)

The ISO standard 8373:2012 [21] provides a definition of an industrial robot that separates the system in a *manipulator* and a *controller*. In this work, the term *controller* is used in particular to denote the reprogrammable software part of the physical controller.

Manipulators can have diverse mechanical structures. In general a *robot manipulator* "consists of a sequence of rigid bodies (*links*) interconnected by means of articulations (*joints*)" ([19], p. 4). Joints can be *revolute* or *prismatic*, where the former are able to induce rotational motion between the adjacent links and the latter induce translational motion ([19], p. 4). According to the same source, robot manipulators can be characterized by an *arm*, a *wrist* and an *end-effector*. The arm "ensures mobility" ([19], p. 4) and most robot manipulators use *antropomorphic* geometry for the arm ([19], p. 9). Antropomorphic arms employ three revolute joints, where "the revolute axis of the first joint is orthogonal to the axes of the other two which are parallel" ([19], p. 8). The wrist "confers dexterity" ([19], p. 4), where the most dexterous mechanical wrist construction is the spherical hand with three axes intersecting at a single point ([19], p. 10). In the context of this work, the term *robot arm* will be used most of the time to denote the unit of arm and wrist, as industrial robot arms are usually offered as one unit of both parts. Fig. 2.1 shows a rather small six-axis industrial robot and the rotation axes of its revolute joints. The axes 2 and 3 are parallel (the perspective distorts this relationship), and axes 4-6 intersect in a common point to form a spherical hand. Note that the axes 3 and 4 do not intersect due to a mechanical skew in this robot. However, not all industrial robots are equipped with six joints (3 arm joints + 3 wrist joints). In particular, robots with only four joints are sometimes used for simple pick&place tasks.

An end-effector "performs the task required of the robot" ([19], p. 4). Examples used in this work include grippers, screwdrivers and welding torches. Most industrial robot arms provide a standardized *mounting flange* for mechanically mounting tools, e.g. according to ISO 9409-1 [22]. In this work, this is often simply called *flange*.

The most important property of any robot is its ability to execute *motions*. Intuitively, a motion takes the robot arm from its current configuration to another configuration. The configuration of a robot arm can be described in *joint space* by assigning a certain value to each of the robot's joints, or in the robot's *operational space* by describing the *position* and *orientation* of the end-effector ([19], p. 84). The term *pose* is used to denote a specification consisting of both position and orientation. The *workspace* of a robot is "the region described by the origin of the endeffector frame when all manipulator joints execute all possible motions" ([19], p. 85).

Commonly, the Euclidean space is used as operational space of an articulated robot arm, and its position in this space is described using Cartesian coordinates [20]. A

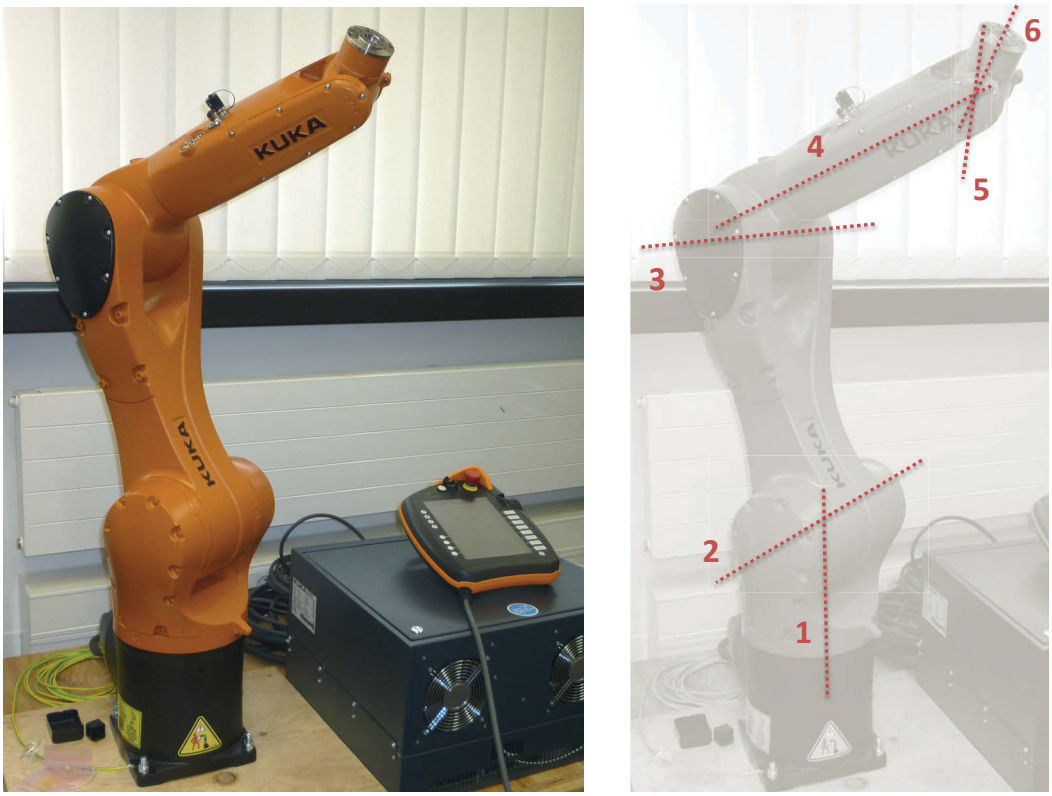

Figure 2.1: An industrial robot arm with controller and teach pendant (left) and the six joint axes of this robot arm (right).

mathematical formalism for expressing these coordinates and performing calculations on them will be presented later in this work. Intuitively, Cartesian coordinates assign the robot arm's flange or end-effector three values that specify its position with respect to a certain reference system. As for most end-effectors of robot arms also their orientation in space is important, this orientation is described by further coordinates which express the rotation of the robot arm's end-effector w.r.t. the same reference system. Often representations of orientation are used that consist of three values as well [20]. A robot's pose in its operational space is then fully described by six values.

In industrial robotics, certain basic types of motions are usually provided by robot controllers. Those motions are classified in *joint space motions* and *Cartesian motions*. The *path* of a motion is "the locus of points in the joint space, or in the operational space, which the manipulator has to follow in the execution of the assigned motion" ([19], p. 161). In contrast, a *trajectory* is not just a pure geometric description of a motion, but "a path on which a timing law is specified, for instance in terms of velocities and/or accelerations at each point" ([19], p. 161). Industrial robot controllers are able to pre-plan the trajectories of different types of motions in a deterministic way and ensure that the motion actually executed by a robot arm will only minimally deviate from the

trajectory. Pre-planning of trajectories is termed as *motion planning* in this work.

For joint-space motions, a trajectory in joint space will be planned. Each joint moves individually, and synchronization is only done e.g. with respect to the total motion time. Thus, the Cartesian path of a joint-space motion is not easily predictable for operators. However, joint-space motions can in general be executed faster than Cartesian motions, as each joint can be moved with its maximum possible acceleration and velocity. For Cartesian motions, a trajectory in Euclidean space will be planned, for example along a straight line.

Joint-space motions do not necessarily have to be specified in joint space by operators. It is often more convenient to specify the target of a joint-space motion using Cartesian coordinates. The robot controller has to convert the Cartesian coordinates to joint-space coordinates before motion planning. For this purpose, the *inverse kinematics function* is used. It performs "the determination of the joint variables corresponding to a given end-effector position and orientation" ([19], p. 90). Conversely, the *direct kinematics function* has the responsibility "to compute the pose of the end-effector as a function of the joint variables" ([19], p. 58).

While the computation of the direct kinematics function is straightforward for open kinematic chains like articulated robot arms, the inverse kinematics function is much more complex to compute and it depends on the manipulator's structure and current configuration if a unique solution can be calculated. One source of complexity is *kinematic redundancy* of manipulators. This redundancy is related to the *degrees of freedom (DOFs)* of a manipulator, which is equal to the number of joints in an open kinematic chain ([19], p. 4). Kinematic redundancy means that the manipulator "has a number of DOFs which is greater than the number of variables to describe a given task" ([19], p. 87). Thus, when a robot arm with six joints as described above should execute a task that is specified by defining all six Cartesian pose variables, there is no redundancy. However, when one or more of the pose variables are not specified, or the robot arm has more than six joints, there is redundancy which has to be resolved. One way to do this is to introduce additional artificial variables that are not required by the task, but are used solely to resolve redundancy.

Further sources of complexity when calculating the inverse kinematics function are the existence of multiple solutions and infinite solutions ([19], p. 91). As described there, for a manipulator with six DOFs without mechanical joint limits, there are in general up to 16 solutions of the inverse kinematics function. To choose an appropriate solution, one way is to extend the specification of a Cartesian target by additional constraints that specify criteria for a preferrable solution. Infinite solutions of the inverse kinematics function can occur in case of kinematic redundancy. According to Siciliano et al., infinite solutions may also exist when the manipulator's structure is at a *singularity* ([19], p. 116). As stated there, "Singularities represent configurations at which the mobility of the structure is reduced (...). In the neighborhood of a singularity, small velocities in the operational space may cause large velocities in the joint space". One class of problematic singularities are *internal singularities*, which are "caused by the alignment of two or more

axes of motion, or else by the attainment of particular end-effector configurations. (...) they can be encountered anywhere in the workspace for a planned path in the operational space." ([19], p. 116). Depending on the kind of operation performed by a robot arm, singularities may be avoided, or otherwise render the operation as such invalid.

The basic robot motions mentioned above are based on pre-planning and interpolating trajectories that are fully defined in Cartesian or joint space. Approaches for *on-line trajectory planning* have evolved as well (e.g. work by Kröger et al. [23]), which "enables systems to react instantaneously to unforeseen and unpredictable (sensor) events at any time instant and in any state of motion" ([23]). Furthermore, it is in some tasks more convenient to specify motions not in terms of paths, but by specifying other parameters like a velocity in a certain geometric direction, or a force to apply to a certain workpiece. This is not supported by most industrial robot controllers, but does play a role in the context of this work. For best performance of such velocity-based or force-based motions, appropriate control loops should be available. For example, *direct force control* enables "controlling the contact force to a desired value, thanks to the closure of a force feedback loop" ([19], p. 364). In contrast, *indirect force control* does not explicitly close such a force feedback control loop, but realizes force control via an ordinary motion control loop that is used for controlling the robot's position ([19], p. 364). *Impedance control* belongs to the category of indirect force control. Its goal is to actively control the dynamic behavior of the robot's end-effector such that it acts for example like a damped spring system with controllable damping and stiffness ([24]). A more advanced form of control structure is *hybrid force/motion control* ([19], p. 396), which simultaneously takes into account force and position or velocity control values.

Irrespective of the type of motion, it is often desirable to explicitly integrate sensors in the specification of motions. The two principles of *sensor guarding* and *sensor guiding* [25] are of particular interest in this work. The idea of sensor guarding is to monitor sensor measurements during motions (or other operations) of a robot and to react to changes by stopping the current robot operation if necessary. Sensor guiding instead allows for using sensor measurements as input for controlling the robot's operation, for example by adapting its velocity or the force it applies to the environment.

2.2 Programming Industrial Robots

This section will give an overview of the KUKA Robot Language (KRL). The following description is not exhaustive, but aims at giving a good overview of the language. Details can be found in [26]. The following explanations are based on the example KRL program shown in Listing 2.1.

General language concepts

KRL is an interpreted, imperative language with some robotics-specific instructions and a distinct, restricted model of parallelism tailored to robot programming. It provides typical control flow statements for branching (IF... THEN... ELSE... ENDIF, see Lst. 2.1,

```
1  DEF example()
2    DECL INT i
3    DECL POS cpos
4    DECL AXIS jpos
5
6    FOR i=1 TO 6
7      $VEL_AXIS[i]=60
8    ENDFOR
9
10   jpos = {AXIS: A1 0,A2 -90,A3 90,A4 0,A5 0,A6 0}
11   PTP jpos
12
13   IF $IN[1] == TRUE THEN
14     cpos = {POS: X 300,Y -100,Z 1500,A 0,B 90,C 0}
15   ELSE
16     cpos = {POS: X 250,Y -200,Z 1300,A 0,B 90,C 0}
17   ENDIF
18
19   INTERRUPT DECL 3 WHEN $IN[2]==FALSE DO stop_robot()
20   INTERRUPT ON 3
21
22   TRIGGER WHEN DISTANCE=0 DELAY=20 DO $OUT[2]=TRUE
23   LIN cpos
24   LIN {POS: X 250,Y -100,Z 1400, A 0,B 90,C 0} C_DIS
25   PTP jpos
26
27   INTERRUPT OFF 3
28 END
29
30 DEF stop_robot()
31   BRAKE
32 END
```

Listing 2.1: Example KRL program, taken from [27]

line 13-17) and for various kinds of loops (e.g. WHILE...ENDWHILE). There also exists a special statement for delaying the program flow until some condition has become true. KRL is statically typed. The type system contains some predefined basic types (e.g., INT, BOOL, CHAR, see Lst. 2.1, line 2), arrays of those, predefined structured data types (e.g. POS for specifying robot poses, see Lst. 2.1, line 3) and the possibility to create user-defined data structures.

KRL programs are split in two files: A *source* file contains the actual program code, whereas a *data* file contains program data, like e.g. taught robot poses. Source files consist of a *declaration* and an *instruction* part, where the latter may contain up to 255 *subroutines* or *functions*. Functions specify a return value, subroutines do not. In the following, the term 'function' is used to denote both. Functions may also be declared globally in a program, or be declared in their own source file. Both ways make functions available to other programs as well.

Besides functions, a second way of structuring programs are the so-called *FOLDs*. A FOLD is a special pair of KRL comments which causes the KRL editor to hide the enclosed content by default. Depending on the type of user that views or edits a program (KRL distinguishes between *Operator*, *Expert* and *Administrator*), the content of FOLDs

can be displayed in the editor.

System variables

A lot of information about the robot's state and the controller's state is provided by various global variables, called *system variables*. Examples are $POS_ACT (the current robot pose) and $VEL.CP (Cartesian velocity). Some of those variables also serve to define configuration properties for robot and controller, like $ALARM_STOP (specifies the controller output which signals an emergency stop to other controllers).

Motion instructions

Motion instructions are built-in functions for controlling the robot's motion. The *PTP* function (see Lst. 2.1, line 11) starts a joint space movement to a given goal, which can be specified either in joint space or in Cartesian space. Built-in Cartesian path motions include *LIN* (linear motion, see Listing. 2.1, line 23 et seq.), *CIRC* (circular motion) and in newer controller versions also *SPL* (spline motion). All motion instructions usually start from the robot's position at the point in time the motion operation is started. Linear motions move the robot to a given goal on a straight line in space. Circular motions calculate a circular path in space that connects the robot's start position, the specified goal position and an auxiliary position. Spline motions consist of multiple segments. Each segment is specified by two points and a segment type. The resulting motion is along a mathematically defined path through the single segments.

Motion blending can be used to achieve continuous movement across successive motion instructions. This technique can be used in all cases where the intermediate point of the two motions (i.e. the target of the first motion, which equals the start point of the second motion) does not have to be reached exactly. Fig. 2.2 shows the resulting motion path when blending from motion A to motion B. On programming language level, blending can be activated by adding special keywords to motion instructions (e.g. *C_DIS*, see Lst. 2.1, line 24). Additionally, parameters for blending (e.g. distance to target at which blending is started) have to be specified in advance.

Motion blending is primarily an optimization technique in robot programs. By moving continuously, the total time required for the motions decreases, as the robot has to perform less braking and accelerating. This also leads to less mechanical stress for the robot hardware.

Program counter and advance run

As stated before, KRL programs are interpreted. The interpreter distinguishes two kind of program counters: The *main program counter* is located at the motion instruction that is currently executed by the robot. The *advance run counter* is located at the motion instruction that is currently being interpreted by the robot control. The robot control will by default interpret a set of motion instructions in advance, as displayed in Fig. 2.3. The number of pre-interpreted motion instructions can be controlled by applications (via

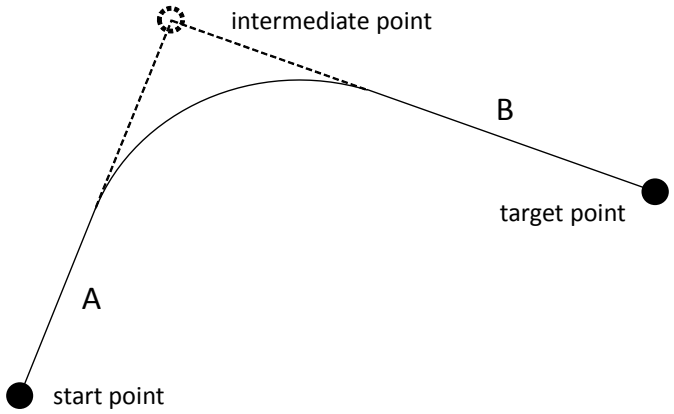

Figure 2.2: Blending between two successive motions A and B.

```
DECL AXIS jpos

$ADVANCE=1

jpos = {AXIS: A1 0,A2 -90,A3 90,A4 0,A5 0,A6 0}
PTP jpos                                    Main program counter

FOR i=1 TO 6
   $VEL_AXIS[i]=60
ENDFOR                                      Position of advance run counter;
                                            $ADVANCE = 1
PTP POINT42

LIN cpos
LIN {POS: X 250,Y -100,Z 1400, A 0,B 90,C 0} C_DIS
PTP jpos
```

Figure 2.3: Program counters in KRL.

the system variable $ADVANCE). All other instructions in the program that precede the next motion instruction will also be interpreted in advance. The advance run mechanism enables the robot control to perform planning of subsequent motions while a motion is being executed. This is also a prerequisite for realizing motion blending, as the robot controller can plan the blending motion in time. However, some kinds of instructions (like I/O accesses, see below) stop the advance run. In these cases, motion blending is not possible.

I/O instructions

Robot controllers often have to communicate with other devices. Such devices may include tools, sensors, other robots or all other kinds of machinery. The communication is performed on input and output channels. Those channels are accessed in KRL programs via system variables ($IN and $OUT, see Lst. 2.1, line 13, 19 and 22). All values of those

variables are read and set in a cyclic manner by the robot control. By default, every access to an input or output stops the advance run. This behavior can be overridden on a per-case basis in applications. The I/O accesses are in this case already performed when they are interpreted by the advance run, which can lead to unwanted interactions.

Trigger

Trigger statements are bound to motion statements and are a way to execute operations depending on the motion progress. Listing 2.1 contains an example of a trigger statement (line 22). A trigger's condition can specify the distance to a point on the motion path and an additional delay in time before or after the specified distance has been reached. The trigger's instruction can be a variable assignment, an operation for an output or a function call. In the latter case, a priority has to be assigned. Those priorities control the order of execution of simultaneously started triggers and interrupts (see below). The main program flow is paused during processing of the trigger.

Interrupts

With *interrupts*, reactions to some kinds of events can be handled in an asynchronous way in a KRL program (see Lst. 2.1, line 19). A maximum of 32 interrupts can be declared in each program. An interrupt declaration assigns to an event a function that can implement appropriate reactions. Events can be based on boolean variables, boolean input channels, or comparisons on numeric variables or input channels. The reaction is triggered each time the specified boolean expression is evaluated to true. Interrupts have to be activated in order to be evaluated, which is allowed for a maximum of 16 interrupts in parallel. Like for triggers that call functions, a priority has to be assigned to each interrupt. Only a single interrupt routine can be executed at the same time. The interpretation of the main program is paused during handling of an interrupt. The robot will, however, finish all motions that have already been pre-planned. When this is done, motion instructions specified in an interrupt routine can be executed. A special BRAKE statement (cf. Lst. 2.1, line 31) can be used in interrupt handlers (and only there) to halt the robot's motion.

Cyclic flags

To mitigate the problem of limited expressiveness of interrupt conditions (no complex boolean expressions are allowed), KRL allows to define a limited number (32) of *cyclic flags*. Cyclic flags can be assigned arbitrarily complex boolean expressions consisting of (system) variables, but not boolean functions. A cyclic flag is evaluated regularly by the robot controller and can, like ordinary boolean variables, be used as event in the definition of interrupts.

Submit program

The KUKA Robot Control allows to define a so-called *submit program* that is interpreted in parallel to KRL programs. Submit programs are programmed in KRL as well, but are very limited in terms of allowed statements (e.g., robot movements are not allowed). Submit programs are intended to be used for monitoring safety properties.

Execution of programs

A property not inherent to KRL itself, but to its interpreters are various ways of executing programs, including *record selection*, *stepwise execution* and *backwards execution*. Record selection allows for specifying motion instructions as entry point for an execution run of a program. Stepwise execution is bound to motion instructions as well, i.e. code between subsequent motion instructions is executed in one step, but execution pauses before each new motion instruction. Both mechanisms are useful for tuning single motion instructions without having to execute a complete workflow. For similar purposes, backwards execution of KRL programs is possible. The details and limits of backwards execution are not entirely clear. However, it can be assumed that at least motion instructions are planned backwards (which can yield slightly different paths) or are recorded and replayed in reverse order, if they have already been executed in forward direction.

Add-on technology packages

Besides the abovementioned concepts that are essential part of KRL and the KRC, KUKA offers additional packages that extend the capabilities of KRL and the underlying real-time control software. Those packages are called *add-on technology packages*. Some add-on technology packages are targeted at particular application domains, while others add support for new general robotics concepts. Two of those packages will be briefly explained in the following. They extend KRL and the KRC to allow for multi-robot programming and sensor-based external control.

The add-on technology package *KUKA.CR* (**C**ooperating **R**obots [28], sometimes also called *RoboTeam*) extends the KRC and KRL by diverse mechanisms for synchronizing the programs of multiple robots. The controllers of all robots have to be connected via a network. The technology *Shared Pendant* allows for configuring multiple controllers with a single KUKA Control Panel. KRL is extended with additional instructions for synchronizing the program flow of multiple robot programs (*Program Cooperation*) and for synchronizing sequences of motions of multiple robots (*Motion Cooperation*). Program Cooperation can be used e.g. to start motions at the same time, or to prevent multiple robots from working in the same physical space concurrently. For tight real-time cooperation, e.g. sharing loads among multiple robots or moving a robot relative to another robot that is moving itself, Motion Cooperation has to be used.

With the add-on technology package *KUKA.RobotSensorInterface* (RSI) [29], real-time capable networks of function blocks can be defined, e.g. for processing sensor values. This processing logic can also be used to influence the robot movement or the flow of a KRL

program. The technology package comes with a set of pre-defined function blocks, which can be combined with dataflow links to create an application-specific processing logic. When combined with the package *KUKA.Ethernet RSI XML* (ERX) [30], it is possible to integrate external systems in a network of RSI function blocks. Special function blocks are introduced for sending values to and receiving feedback from external systems. The ERX package is employed by some system integrators and research institutes to control KUKA robots completely with external control systems (e.g., in the *Remote Robot Control* application developed by MRK Systeme GmbH, described in [31]).

Summary

The KUKA Robot Language is essentially an imperative language with built-in functions for robot motions, communication with tools and other devices and certain mechanisms for reactive actions. An outstanding feature is its execution semantics, which involves an advance run counter to realize pre-planning of motions and blending between motions. From publicly available documentation, it is not clear to which extent KRL code is compiled and to which it is interpreted. However, the execution environment provides features that suggest that many parts are interpreted, for example record selection and backwards execution. These features are also rarely found in other programming languages. Add-on technology packages extend KRL by new functions and UI components. Some technology packages aim to support integration of KUKA robots with other systems by e.g. providing network-based communication functionality.

Major downsides of the KUKA Robot Language are:

Limitedness in terms of motion control concepts. KRL itself provides only access to position-controlled, pre-planned motions. Though there are technology packages available that can be used to create more advanced control concepts, this requires expert knowledge and considerable effort. Sensor-guarded motions can be realized by combining interrupts and cyclic flags, but possible reaction strategies are limited by the general interrupt semantics (e.g., the main program will be interrupted).

One controller per robot. Each KRL program can implicitely control only one robot arm. For programming multiple (cooperating) robots, a separate controller and a separate program is required per robot. To synchronize programs, each single program has to be modified. This approach for multi-robot programming is thus not compositional, and program complexity increases with every additional robot.

Few structuring mechanisms. The only mechanisms provided by KRL for structuring operations and data are functions and structured data types. The language lacks further modularization concepts for larger applications, which makes it difficult to design reusable robot software. There is furthermore no dedicated concept of libraries that could be used to encapsulate reusable functionality.

Limited support of various general programming concepts. KRL is not designed as a general-purpose language, thus it does not provide concepts like file system access, database connectors or graphical user interfaces. Truly parallel execution of program

parts is not possible with KRL, with the exception of submit interpreter programs. However, submit programs may only use a subset of KRL.

Proprietary character. KRL is developed and maintained by KUKA, and developing substantial extensions to the language is usually only possible with permission and support by KUKA. The fact that KRL is usable exclusively for programming robots greatly limits the community of developers. In contrast, general purpose languages are geared at a much wider audience, and often libraries created by experts in one field can be reused by developers in completely different fields. The potential of KRL for this kind of reuse is limited in principle.

Chapter 3

Application Examples

Summary: During the work on this thesis, several robotics applications have been developed, three of which are introduced in this chapter, together with the robot hardware used. These three applications incorporate many of the challenges that were faced during development of a new software architecture for industrial robotics. The challenges are stated and are taken up in later chapters.

To demonstrate the usefulness of research results and to evaluate the state of the reference implementation of those results, multiple robotics applications have been developed in the course of this work. Three of those applications are presented in this chapter:

- the *PortraitBot* application, which employs two KUKA Lightweight Robot arms to sketch portraits of people's faces while compensating disturbances,
- the *Tangible Teleoperation* application, enabling a remote operator to manipulate objects with two robot arms with a multi-modal user interface,
- and the *Assembly Cell* application, in which two robots cooperatively transport and assemble objects from parts using force-based manipulation concepts.

This chapter focuses on the general ideas behind those applications. It also introduces the robots and robot tools used. Those hardware devices are used in various parts of this work to clarify the developed concepts with concrete examples. The last section of this chapter summarizes the challenges incorporated in the presented applications. Chapter 11 comes back to the three applications and demonstrates how the concepts presented throughout the work can be employed to solve all of the applications' challenges.

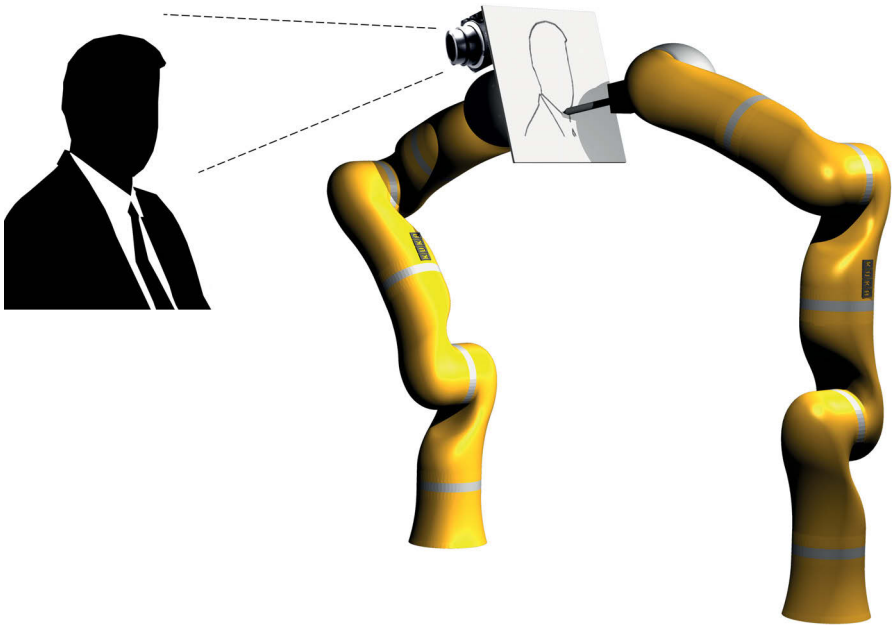

Figure 3.1: The PortraitBot can draw sketches of human faces on a moving canvas.

3.1 The PortraitBot Application

The first part of this section illustrates the concept of the PortraitBot application. In the second part, the KUKA Lightweight Robot is introduced, which is used in the PortraitBot application as well as in the other applications presented in this work.

Concept

The core idea of the project is to demonstrate accurate synchronization of two robots' motions. A pen is mounted to one robot's flange, and the task is to draw a geometric figure on a canvas mounted at a second robot's flange. This robot is allowed to move while the other robot is drawing. Based on known geometric relationships between the robots, the drawing robot should be able to fully compensate the motion of the canvas.

By adding a camera and mounting it to the robot arm that holds the canvas, the 'PortraitBot' is born. The system is able to record pictures of human faces and draw sketches of them on the moving canvas. The portrayed persons can influence the canvas movement by moving their head, as the PortraitBot will move the camera and try to keep a central view on the portrayed face. The concept is depicted in Fig. 3.1.

The KUKA Lightweight Robot

The KUKA Lightweight Robot (LWR) is a serial manipulator with seven revolute joints. Thus, it is by design a redundant manipulator for all tasks in Euclidean space. The hard-

ware concept was developed at the Institute of Robotics and Mechatronics at the German Aerospace Center (DLR) [32]. After several development iterations in cooperation with KUKA [33], the *LWR iiwa* was finally released as official product by KUKA [34].

At the Institute for Software & Systems Engineering at the University of Augsburg, a setup of two Lightweight Robots (see Fig. 3.2) was used as main platform for evaluating the concepts developed in this work. The robots are of type LWR4 (cf. [33]) and thus a predecessor of the LWR iiwa. In the rest of this work, the term *LWR* thus refers to the LWR4 version of this robot.

The LWR4 robot arm has a low mass of 15kg and is officially allowed to lift up to 7kg according to Bischoff et al. [33]. The authors state that the low weight "decisively improves the robot's dynamic performance" and that its seven-axis design "gives the programmer more flexibility in cluttered workspaces". A further important feature of the LWR are torque sensors integrated in each joint. According to [33], "a detailed dynamic model of the robot, state control and a high servo-control cycle rate (3 kHz locally in the joints, 1 kHz overall) (...) enable active damping of vibrations to achieve excellent motion performance" and additionally "makes it possible to achieve a programmable compliance, both axis-specific and Cartesian". The impedance control principles used for programmable compliance are presented by Ott et al. [35]. They allow "the robot to act like a spring-damper system in which the parameters can be set within wide limits" [33].

The LWR4 is delivered with a particular version of the KUKA Robot Controller called *KR C2 lr*. This controller also provides a special version of KRL that has been extended by some LWR-related features. In particular, functions for switching between the LWR's pure position controller, its joint-specific impedance controller and the integrated Cartesian impedance controller are provided, as well as means for setting controller-specific parameters. The KR C2 lr can be extended by an add-on technology package called *Fast Research Interface (FRI)* [36]. With FRI, the LWR can be controlled from external systems by commanding target position values for all joints in a fixed, but configurable rate. Such a control mode is termed *cyclic synchronous position* (CSP), e.g. in the IEC 61800-7-201 standard [37]. In the FRI interface, the cyclic rate is configurable in a range of 1-100 ms.

Figure 3.2: Lightweight Robot setup in ISSE's robot lab.

3.2 The Tangible Teleoperation Application

Recent progress in sensor hardware, sensor information processing and cognititive capabilities led to a significant increase in the autonomy of robot systems. While this progress leads to less involvement of human operators in some areas, there are other cases where human intervention is desirable or even indispensable – e.g. in rescue robotics, where certain decisions should rather be taken by a human than relying on a machine. In these areas, teleoperation systems that support human operators in efficiently controlling robot systems from a remote location play an important role.

During the work on this thesis, a multi-modal, tangible user interface for teleoperation of a two-arm robot system was developed. The system was developed mainly as a case study to evaluate the robustness and usefulness of the developed software architecture, in particular to demonstrate the interoperability between modern programming languages. On the other hand, the combination of touch-based and tangible user interface concepts also constituted a new approach in teleoperation research and was thus published in [38].

The Tangible Teleoperation application was realized with two KUKA Lightweight Robots like in the PortraitBot application. Both robots were equipped with Schunk MEG 50 grippers. The application itself was realized with the Microsoft PixelSense[1] (formerly known as Microsoft Surface) and its SDK, which builds on the .NET platform [39]. In the first part of this section, the concept of the Tangible Teleoperation application is

[1] `http://www.microsoft.com/en-us/pixelsense/default.aspx`

presented and the challenges related to robot control are outlined. The second part introduces the characteristics of the grippers used.

Concept

The Tangible Teleoperation application is focused on a rather simple scenario: A tele-operator should be able to pick workpieces from some location and put those workpieces to any target location. Though the core steps in this task are quite simple, it may involve preliminary steps like deciding a gripping strategy (see [40], pp. 671) or even moving other objects that block access to the workpiece to be transported. A human operator can infer most of those necessary steps quickly from just observing the scene (or the camera picture, respectively), in contrast to complex cognition steps in an automated robotic system. The tele-operator should be able to perform pick and place tasks with intuitive support by the teleoperation system. Similar to previous work in this area ([41], [42], [43], [44]), visual perception of the scene is considered very important and is realized here by a single camera, mounted next to the gripper of one of the robot arms. Additionally, a 3D model of the current poses of the robot arms is displayed to the operator and is updated continuously. In this model, also a visualization of the end-effector forces measured by the KUKA Lightweight Robots is integrated, which serves as an additional aspect for perceiving the scene. Fig. 3.3 shows the laboratory setup of both Lightweight Robots and the operator's control interface. One of the robots is equipped with a simple network camera, which utilizes Wireless LAN for transmitting the video stream to the application.

Figure 3.3: Robot setup (left) and operator interface (right)

The teleoperation system supports two distinct modes of operation: In the *direct control*

mode, the robot arm with the camera mounted on it performs the gripping task. In this mode, the movement of the camera (and thus the scene perceived by it) corresponds directly to the arm movement commanded by the user. It can be considered a "first person" perspective. In the *observer mode*, the robot arm with the camera mounted takes an observer position, while the second robot arm, the manipulator robot, performs the actual task (cf. Fig. 3.4). In both modes, the operator can control the robot arm's linear and angular velocity with a so-called *6D mouse*. The observer robot automatically follows movements of the manipulator robot. The operator can rotate the camera robot around the observed point of interest. Additionally, by moving the observer robot closer to or further away from the point of interest, a zooming functionality can be realized. Thus, the scene can be observed flexibly.

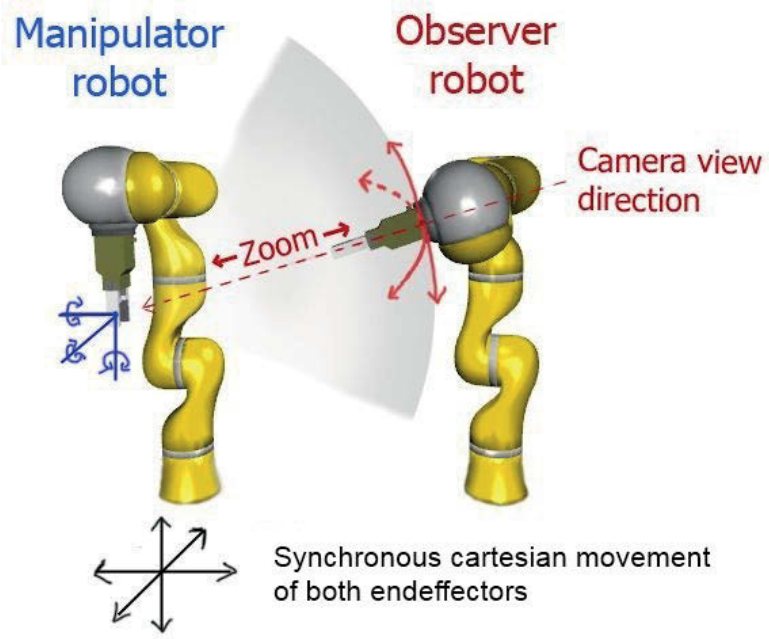

Figure 3.4: Observer mode of the Tangible Teleoperation application.

Schunk MEG 50 gripper

In the Tangible Teleoperation Application as well as the Assembly Cell application, a electric parallel gripper of type *MEG 50* by Schunk is used. This gripper is described by Schunk as "an interchangeable device for machines or systems in the manufacturing, packaging and laboratory industries" which has been "designed for form-fit gripping, secure holding and releasing of workpieces" [45]. It is an electric device with two moving jaws that are parallel to each other (see Fig. 3.5). The jaws can be extended in length by mounting so-called 'fingers' to them. The total movement range between the jaws is 16mm, for which Schunk specifies a repeat accuracy of 0.02mm. It is recommended to grip workpieces with a maximum weight of 0.5kg.

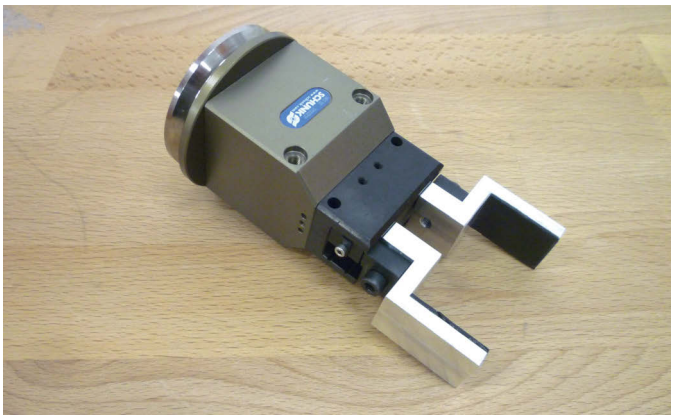

Figure 3.5: The Schunk MEG 50 parallel gripper.

The gripper device itself is called *MEG 50 EC*, while the external controller is called *MEG C 50*. The controller provides a variety of electrical input and output pins which allow to control the movement of the gripper in different ways and get feedback about current movement parameters. The pins are classified as *digital* and *analog* inputs and outputs, respectively. This simple form of inter-device communication, termed *I/O communication*, is closely related to the physical medium used (mostly copper cables) and is based on electric potential differences. Digital signals are sent by creating a defined potential between two electric poles, e.g. 24V. Changes of this potential are detected by the receiver and interpreted as signals if they exceed a certain threshold. Analog signals take into account the exact value of the potential, e.g. in a range of -10..10V. Many simple robot tools available on the market demand to be controlled in this way, and many sensor devices report values in this form.

In the Tangible Teleoperation application, (and also the Assembly Cell application presented in the next section), the MEG 50 gripper was used in the so-called *power movement* mode. In this mode of operation, the gripper is not precisely positioned, but can grip workpieces in the most secure way. Only the controller's digital inputs for opening and closing the gripper are required, whereas the desired moving speed and gripping force can be set via two analog inputs. When one of the digital inputs receives a *high* value (in this case 24V), the gripper will start to open or close with the currently specified speed and force parameters. When the gripper stops movement due to being blocked by its own mechanical design or a workpiece, it indicates this by setting a high value on a certain digital output of the controller. However, the gripper continues to maintain the specified force and will stop this only when the *high* value on the open or close input is reset to a *low* value (0V).

A complete documentation of the gripper's abilities can be found at [45]. This section is mainly intended to create an impression of the way in which many simple robot tools are controlled. Often robot controllers provide means to communicate with such kinds of devices. The LWR's KRC 2 lr controller can be extended to support I/O communication

and was used for controlling robot tools in all applications presented in this work. The FRI protocol that is used to control the LWR remotely (see Sect. 3.1) can be employed to transmit I/O values to and from the KRC 2 lr.

3.3 The Assembly Cell Application

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 according to 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 application, two robot arms and a mobile robot platform work together in part assembling. To achieve that, force-guarded compliant motion, real-time motion synchronization and different coordination patterns are applied.

Concept of Factory 2020

Fig. 3.6 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 inaccurate positioning 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. The motion of both arms has to be real-time synchronized to ensure proper transport without damaging the containers or robots.

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 for 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, inserts and finally tightens it using the screwdriver. These operations are also performed using force-guarded compliant 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 placed 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 Assembly Cell

In this work, the focus is put on a part of the Factory 2020 application, termed the *Assembly Cell*. This includes the following tasks performed by the two Lightweight

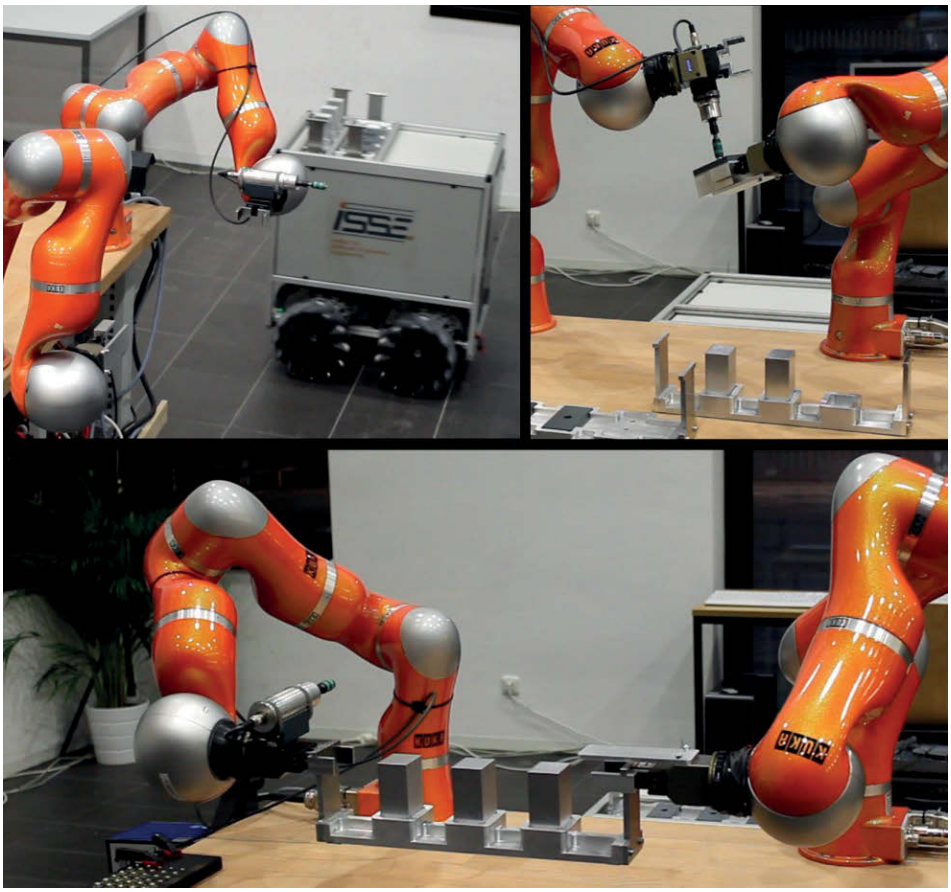

Figure 3.6: In Factory 2020, an autonomous robot platform delivers parts (top left), robot arms cooperate in transporting (bottom) and assembling them (top right).

Robots:

- cooperative workpiece carrier transport,
- coordinated compliant assembling of workpieces from parts,
- compliant fetching, inserting and tightening of screws.

The control of the mobile platform and the overall system architecture is not considered in detail in this work.

Kolver Pluto screwdriver

Both Lightweight Robots in the Assembly Cell application are equipped with Schunk MEG 50 grippers (cf. Sect. 3.2). Additionally, an electric screwdriver is mounted to one of the robot arms, which is of type *Pluto 10 CA/N* by Kolver S.r.l. Italy. A photo

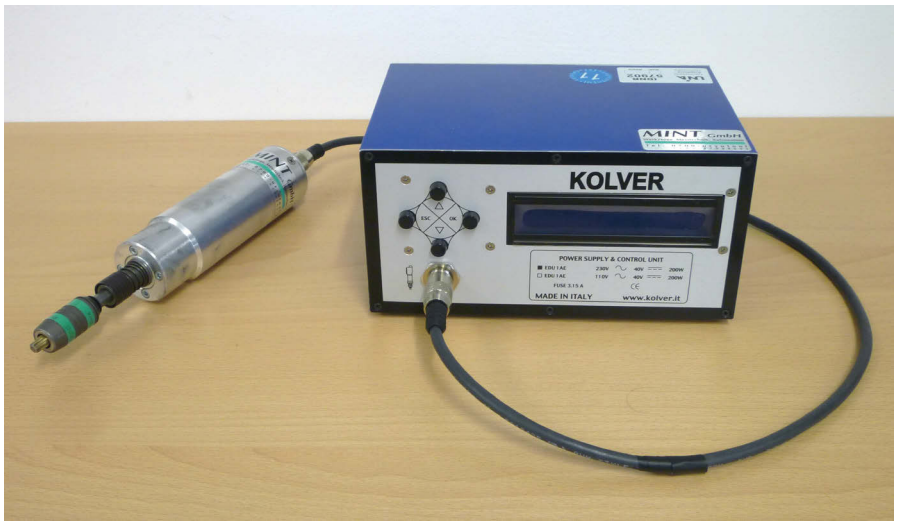

Figure 3.7: The Kolver Pluto screwdriver (left) and its controller (right).

of this device and its controller is shown in Fig. 3.7. The controller provides a set of inputs and outputs that are connected to those of the KR C2 lr in the Assembly Cell setup. With these communication channels, the screwdriver can be started in forward or reverse direction and stopped again. Additionally, the screwdriver's controller provides feedback whether a preconfigured torque level is reached during tightening of a screw. Based on these communication channels, the screwdriver is controlled by the developed automation software.

3.4 Challenges

When realizing the applications presented in this chapter, multiple challenges arise that are difficult to handle with current industrial robot controllers. In the following, those challenges are summarized.

Velocity-based control of robot movements is beneficial in various cases. For devices like the 6D mouse used in the Tangible Teleoperation application, it is a natural kind of control mode. Deflections of the mouse can be mapped to translatory and rotatory robot movements, similar to what is done in 3D simulation environments, which is a typical area of use for such input devices. Furthermore, tasks like the face-tracking employed in the PortraitBot application can also be realized based on velocity control. Velocity control of robots in user programs is not supported e.g. by the KUKA Robot Control and KRL. It can be realized with the RSI add-on technology package (cf. Sect. 2.2). However, in this case an open control loop based on position control has to be realized on an application level. It is desirable to rather offer such functionality to application developers as elementary operation that is easy to use.

Sensor-guarded and sensor-triggered actions are of great importance in the Assembly Cell application. In particular, the torque sensors and the KUKA LWR's ability to estimate the force applied by an end-effector based on the torque sensors are a prerequisite for the tasks that should be realized. Some operations, like joining the workpiece parts, have to be guarded by force sensor measurements to prevent excessive forces. In other cases, the measured force can be used to trigger further operations. For example, when a screw is to be inserted and tightened, it is natural to start the tightening process when a certain force on the screw has been established. Besides sensor-based guarding and triggering, the active compliance provided by the Lightweight Robot is very useful for such assembling tasks. The KRC 2 lr robot controller provides support for changing the internal controller parameters of the LWR and thus control the level of compliance. However, the provided operations are only partially integrated into the KUKA Robot Language. Instead, many commands have to be sent to a so-called 'motion driver' in form of byte-patterns. This approach can lead to problems considering maintainability of programs, as the respective operations are hard to comprehend by non-experts. Similar problems arise when actions should be guarded or triggered based on sensor measurements. KRL provides no means for representing sensor devices due to the limited structuring and encapsulation mechanisms. Instead, merely global variables or I/O channels can be used for retrieving sensor data. Thus, programs become intransparent and hard to comprehend due to hidden information (which device does a certain output control?) and reusability is limited, as the system configuration is implicitely encoded in the program.

Robot motion synchronization is needed in the Tangible Teleoperation application as well as in various aspects of the PortraitBot and Assembly Cell applications. During teleoperation in observer mode, the observer robot should automatically follow all movements of the manipulator robot to keep the manipulated objects in sight of the operator. The PortraitBot should be able to compensate movements of the canvas automatically, which requires some synchronization concept between both robot arms used. In the Assembly Cell application, the motion of two robot arms needs to be synchronized in space and time to realize the desired cooperative transport of workpieces. Various robot manufacturers provide support for synchronization of multiple robots (e.g. *KUKA.CR* for the KUKA Robot Control, cf. Sect. 2.2). The main drawback here is the *one controller per robot* paradigm used: each robot arm is controlled by its own controller that runs its own program. Controllers have to be interconnected with networking hardware to exchange data required for synchronization. On programming level, synchronization usually requires custom statements that have to be added to the programs of all affected robots. This increases the complexity of the single programs, leads to complicated parallel behavior of the system as a whole and negatively affects maintainability, as changes have to be propagated through all programs in the worst case.

System integration here means coupling the proprietary robot control infrastructure with the system that runs the actual application. The need for an external system to run a robotics application is recurring in many applications nowadays (except very basic robot programs with simple, static movement patterns). The reasons mainly lie in the

limitations of the robot programming languages that have been discussed in Sect. 2.2. In the PortraitBot application, image processing is required to detect human faces, extract sketches from them and additionally track their movements. There is a large number of libraries available[2] that are able to solve all of those tasks. However, none of them can be directly integrated in most commercial robot controllers. Such tasks are thus often realized by separate computer systems which run common operating systems and general-purpose programming languages. In case of the Tangible Teleoperation application, the need to use an external system is induced by the PixelSense platform and its SDK, which is itself bound to the PixelSense hardware. However, integrating those systems with traditional robot controllers is a tedious task and is mostly done in an application-specific way over and over again, consuming significant time for development and testing. Examples for this are also published by other robotics researchers, e.g. in [46] and [47]. In the Assembly Cell application, the abovementioned limitations in structuring complex applications with proprietary robot programming languages makes it hard to realize the application logic purely on the robot controller. This is further complicated by the one controller per robot paradigm described above. In sum, if robot controllers provided application developers with an interface in a modern, general purpose language, which is usable on the robot controller itself or from external systems, the problem of system integration could be greatly mitigated.

[2] An overview of over 50 libraries for face detection can be found at `http://blog.mashape.com/post/53379410412/list-of-50-face-detection-recognition-apis`

Chapter 4

The SoftRobot Architecture: A New Generation of Industrial Robot Controllers

Summary: The work presented in this thesis is part of a larger effort shared by a team of researchers in the project *SoftRobot*. The resulting software architecture consists of two main parts with distinct responsibilities. This chapter presents the goals and requirements to the software architecture, illustrates how those requirements are assigned to the architectural parts and clarifies the focus of this work. The general architectural approach has been published in [48] and [49].

The joint research project *SoftRobot* was conducted from October 2007 until March 2012 and was coordinated by the Institute for Software & Systems Engineering (ISSE) at the University of Augsburg. Together with the industrial partners KUKA Laboratories GmbH (KUKA) and MRK Systeme GmbH (MRK), the goal was to create a new kind of robot controller architecture. It should allow for programming challenging robot applications with modern standard programming languages, while at the same time satisfying hard real-time constraints for operations of robots. Additionally, the architecture should not rely on real-time capabilities of the programming language used for two reasons: 1) Developers should not have to care about real-time compliance of the program code they create. 2) Languages with automatic (and therefore usually non-deterministic) memory management should be usable to relieve developers from manual memory management.

A hard real-time constraint imposes a timing deadline to the execution of particular software operations that may under no circumstances be overrun. A general definition of a real-time system goes as follows:

> "A real-time system is a system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment. (...) Hard real-time systems are those where it is absolutely imperative that responses occur within the specified deadline." (Burns et al. [50], p. 2)

Robots and their controllers to some extent clearly have to be hard real-time systems. For example, to achieve deterministic and reliable interpolation of robot motions, the pre-defined interpolation cycles may not be overrun. To capture all hard real-time critical aspects of industrial robot systems, an analysis of a broad variety of typical industrial robot applications (e.g. gluing, welding, palletizing) was performed in the first phase of the SoftRobot project. As input, products of KUKA Robotics and MRK were used, as well as some of their customers' applications. The results of the analysis showed that these applications embody only a relatively small set of hard real-time critical tasks and a limited number of combinations of those tasks. For example, the abovementioned interpolation of a robot movement is a real-time critical task. 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. In the context of this work, the term *real-time* is used synonymously for *hard real-time*, unless stated otherwise.

The following recurring patterns for hard real-time critical robot tasks have been identified.

Pre-planned motions. The ability to perform precise and repeatable motion is a key feature of industrial robots. To achieve this, usually a set of motion primitives is provided which have a clearly defined geometric semantics. Motion statements in robot programs thus have to be interpreted and executed deterministically. Most industrial robot controllers claim repeatability of motions with an accuracy better than 0.1mm.

Geometrically synchronized actions. In many applications, synchronizing the operations of multiple devices according to geometric constraints is of central importance. For example, in welding or cutting applications, the respective tool mounted at a robot has to be controlled relative to the task geometry and thus to the motion of the robot. When a seam is to be welded, the robot's motion is often programmed to accelerate to full velocity before the attached welding torch reaches the start point of the welding seam. The goal is to achieve an equal distribution of the welding intensity. The welding torch has to be activated exactly when it has reached the seam start during the robot motion. Sometimes it is also necessary to combine geometric constraints with timing aspects, e.g., trigger some action at a defined delay before a certain point on a motion path is reached.

Event-based reactive actions. Robot applications need to be able to react to certain events occurring in a workspace. These events might be related to devices like robots and other machines, or to the environment from which they are captured by some kind of sensor. For example, sometimes a search routine is performed to determine the exact position of workpieces before welding. A voltage is applied to the welding torch, and

contact to the workpiece can be measured by monitoring the electric current through the welding torch. As soon as a rise in current is detected, the robot's motion has to be stopped and the position has to be recorded.

Sensor-guided actions. In unstructured and unknown environments, pre-planned motions are often not sufficiently flexible. Instead, sensors are used to guide the operation of robots. A robot that e.g. has to insert a battery into some device is much more flexible when it is able to act according to the forces occuring during the operation, instead of just following a pre-planned insertion strategy. Thus, sensor data has to be processed and used as input for controlling robots. Hard timing constraints apply to the sensor processing operations.

In addition to these patterns, which concern the workflow of applications, industrial robot controllers have to monitor various hard-real time critical safety properties. For example, the maximum velocities and accelerations of all robot joints may not be overrun, otherwise the robot has to be stopped immediately. Such safety properties were not particularly in the focus of the SoftRobot project.

Besides the abovementioned real-time critical patterns, typical robot applications contain a workflow of different tasks, connected by typical control flow instructions like loops and branches. This workflow can in general be considered not hard real-time critical. In sum, the results of the domain analysis led to an important hypothesis: The *workflow of industrial robot applications* can be programmed and executed on top of standard, non-realtime capable programming platforms. *Real-time critical robot tasks* can be executed independently in a separate architectural layer which utilizes a hard real-time capable execution platform.

This hypothesis describes the central assumption that was made in the development of the SoftRobot architecture. The most central research issue concerned the interface between the non-realtime capable *workflow layer* and the *real-time task layer*. The goal was to create an abstraction for real-time critical tasks in the implementation of the workflow layer and provide it to application developers. This approach allows developers to create robot applications including hard real-time critical operations, without the burden of having to write real-time compliant program code, which would imply e.g. to use solely deterministic algorithms, pre-allocate all memory used by the application, call no operating system functions and use no libraries that do not guarantee the same real-time compliant behavior.

4.1 Requirements

Functional requirements

The functional requirements to the SoftRobot architecture were determined based on the abovementioned analysis documents, requirements identified in literature and discussions with all project partners. In these discussions, input from the companies' practical experiences and customer feedback were taken into account. The functional requirements

that have been determined stem on the one hand from the capabilities of today's robot controllers (cf. Sect. 2.2), most of which should be matched by a new architecture, and on the other hand from what has been identified as requirements of intended future robotics applications. The following pages present all functional requirements that the SoftRobot architecture should fulfill.

FR-1: An application should be able to **drive robot arms along pre-defined paths in joint space**. This can be considered one of the most basic tasks, which is required in virtually every robot application. Motions in joint space have the potential to move the robot to a desired goal in the fastest way possible. The SoftRobot architecture should allow for exploiting this potential.

FR-2: **Driving robot arms along different pre-defined paths in Cartesian space** is another basic task required in most robotics applications. In contrast to joint-space motions, the exact robot motion path in 3-dimensional space is determined, at the cost of higher motion time in most cases. The architecture should enable the integration of various motion path types.

FR-3: A prerequisite for Cartesian motion is the ability to **specify an arbitrary number of interdependent points in Cartesian space**, e.g. to use them as goals in motion specifications. It should be possible to express dependencies between points, at least that one point is defined relative to another point and thus changes its absolute position in space accordingly when the other point is repositioned.

FR-4: **Motions should have parameterizable profiles** to adapt their execution according to an application's needs. At least velocity and acceleration should be limitable, if feasible for the respective kind of motion.

FR-5: The SoftRobot architecture should support **blending across multiple motions**. This technique is used in many robotics applications to achieve continuous movement of a robot arm, thus decreasing the time required for movements and causing less mechanical stress for the robot hardware due to less need to accelerate and decelerate.

FR-6: As robotics applications increasingly affect many more devices than just one robot arm, it should be possible to **operate multiple devices with one program**. This should work for devices of arbitrary type, e.g. multiple robot arms, robot tools or any other machinery that has been integrated into the software platform. Today, many commercial robot controllers support real-time robot cooperation (e.g. by KUKA [51] and ABB [52]). However, most controllers follow a 'one program per robot' paradigm, 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 robotics application. The vision of more intensive use of multi-robot systems in future industrial applications is shared by KUKA as well as other researchers (cf. work by Hägele et al. [9])

FR-7: Applications should be able to **synchronize arbitrary device actions based on timing conditions**. Actions should be anything that is executable by some hardware device. Timing conditions should be specified either relative to some clock or relative to the occurrence of some event, e.g. the start or end of another device action.

FR-8: Similar to time-based action synchronization, it should be possible to **trigger arbitrary device actions based on the progress of motions**. Again, triggerable actions should be anything that is executable by some hardware device. The progress of motions should be specified at least based on time criteria.

FR-9: Besides the abovementioned synchronization functionality, **geometrically synchronized actions** should be possible. This should enable actions to be specified relative to a certain geometric reference context. For example, when a robot should move relative to some workpiece, it should not be relevant on application level whether the workpiece itself is moving, as long as the workpiece's movement can be tracked in some way.

FR-10: The architecture should provide means for **querying the measurements of sensors and monitoring them**.

FR-11: Applications should be able to **post-process sensor measurements**, e.g. by filtering data or combining it with data provided by other sensors.

FR-12: Besides time based and motion progress based action triggering, it should be possible to **trigger arbitrary device actions based on sensor measurements**. Raw sensor data should be usable as well as data processed in some way. Based on the measured data, arbitrary trigger conditions should be definable, e.g. that measurements exceed a certain threshold for a given time.

FR-13: The SoftRobot architecture should allow to **guard robot motions using sensors**. In this context, *guard* means to prevent dangerous interaction between the robot and its environment by monitoring measurements of one or more sensors and to modify the execution such that dangerous interactions are prevented. An example is sensor-guarded motion of a robot arm that is stopped when a sensor measures that the robot arm has come too close to an obstacle.

FR-14: Some application scenarios demand **sensor-guided motions**, i.e. motions that are completely determined by measurements of sensors. For example, many assembly tasks can be realized best using velocity or force control, i.e. controlling the force exerted by a robot arm (and measured by appropriate sensors) instead of the robot's position. This has also been stressed by others [1]. Sensor-guided motions should be realizable in the SoftRobot architecture.

Non-functional requirements

Non-functional requirements, i.e. different attributes that describe the *quality* of a system and of the functions it provides, played a very important role in the development of the SoftRobot architecture. In particular, the fact that robot control operations have to satisfy hard real-time requirements has a huge effect on the overall architectural approach. Those hard real-time requirements are reflected in the non-functional requirement categories Performance, Robustness and Safety. However, not all of those categories have been equally considered in the scope of the SoftRobot project: Safety properties of the developed architecture have not been considered, neither were the related requirement categories Compliance (confirmance to law regulations) and Certification (external validation against standards) as well as Availability (part of time the system is ready to be used).

In the following, the non-functional requirements that have particularly been taken into account in the design of the SoftRobot architecture are presented. They are very related to the non-functional requirements part of FURPS by Grady and Caswell [53]. **U**sability and **P**erformance are directly adopted as requirements for the SoftRobot architecture. Robustness is considered as an important aspect of **R**eliability, and **S**upportability particularly includes Extensibility and Testability.

NFR-1: **Usability** is often related to the (graphical) user interface of a software system. However, it can also be interpreted in the context of an application framework like the SoftRobot architecture. In this context, Usability is understood as the learnability and memorability of using the provided framework in applications, as well as the efficiency in creating an application with the framework. In the industrial robotics domain, the understanding of usability also depends on the type of user of an application framework: 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. Vendors of robot hardware require access to the lowest level of a software architecture to extend it with drivers for new types of robots.

NFR-2: In robotic systems, **Performance** is relevant in two different dimensions: The first is execution time of an application, the second is geometrical and timing accuracy of the operations performed by devices. Instead of execution time, the term *cycle time* is often used. Cycle time is related to applications in which a robotic system performs a certain task many times, and one complete execution of the task is referred to as a cycle. Lower time to complete one cycle results in higher throughput and thus more efficiency of the system. Thus, cycle times should be kept as low as possible, which has to be considered in the design of a control software framework. The second performance dimension, geometrical and timing accuracy, is even more important: a major strength of industrial robots is their ability to almost exactly reproduce trained activities. This includes *geometrically exact motions* as well as *exact timing of synchronized actions*, e.g. when executing blended motion sequences

or tool actions that are synchronized with motions. The accuracy of motions is usually measured by different variants of *repeatability*, which can be thought of as how good poses or complete motions can be reproduced. Exact definitions based on the standard ISO 9283 can be found in the Handbook of Industrial Robotics ([54], Chap. 39). To give an example, KUKA claims a repeatability of ± 0.05mm for most of their standard robot arms. To achieve good (guaranteed) repeatability and to hold timing guarantees in executing synchronized motions, a hard real-time capable software platform is inevitable.

NFR-3: **Robustness** describes the "degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [55]. In a robot system, robustness of course heavily depends on the robot hardware itself and the robot's environment (including possible disturbances affecting the robot's task). However, the robot control software can support building robust robot applications by providing means for detecting errors and handling them in an application-/environment-specific way. When handling errors, timing guarantees are again important to prevent damage to the robot system or its environment. For example, when a robot that performs a welding operation stops unexpectedly (e.g. due to an obstacle in the workspace), the welding tool has to be deactivated immediately to prevent damage to the workpiece.

NFR-4: **Maintainability** can be considered an important quality attribute for any kind of software, as it describes the effort needed to make any modification to the software [56]. Such modifications can have all kinds of motivations, from simply correcting a bug over improving performance to adding completely new functionality. Thus, maintainability was considered important also for the SoftRobot architecture. The software design can support maintainability of implementations. In particular, functional *extensibility* is mainly a matter of appropriately defined interfaces. Extensibility has been recognized as important aspect of reusability also in robotics, e.g. in the context of the Player/Stage project by Vaughan [57] or the OROCOS project by Bruyninckx [58]. A high degree of reusability of generic concepts will decrease the effort required to extend the architecture. The SoftRobot architecture should provide interfaces to integrate new devices (e.g. robots or sensors), device operations (e.g., new types of movements, new control algorithms) or even completely new programming interfaces. Furthermore, the architecture should allow for *configurability* of devices in order to decouple applications from the workcell setup to some extent.

NFR-5: **Testability** can in the context of software-intensive systems be interpreted as the possibility to test the functionality of a system or parts of it. Testability is hard to realize for many systems that control hardware devices, as tests with physical actuators are often time-consuming (e.g., robot movements can take considerable time), require complex setup (e.g., placing appropriate workpieces in a robot's workspace) and can be expensive (e.g., a robot may be damaged as a consequence of bugs that lead to collisions). Thus, methods for testing the workflow of applications

offline, i.e. without the physical hardware, should be supported by the SoftRobot architecture. For testing the applications with the final robot hardware, step-by-step execution of device operations (e.g. robot movements) has proven to be efficient and should be supported as well.

4.2 Layers of the SoftRobot Architecture

To achieve an abstraction for real-time critical tasks, a two-tier software architecture was developed (Hoffmann et al. [48]), which is depicted in Fig. 4.1. Its most important aspect is the separation between the *Robot Control Core* tier and the *Robotics Application Framework* tier. The Robot Control Core (RCC) is responsible for low-level, real-time critical hardware communication. The Robotics Application Framework delegates robot commands to the RCC that need to be executed deterministically and real-time compliant, and it receives feedback about sensor values and events that occurred during execution of commands. The Robotics Application Framework is designed as interface for creating complex workflows in robotics applications.

This section will introduce the responsibilities and general design of the Robot Control Core, as well as the responsibilities of the Robotics Application Framework. In the course of the SoftRobot project, a reference implementation of both tiers was created. The *SoftRobotRCC*, which is the reference implementation of a hard real-time compliant Robot Control Core, was implemented in C++ and is usually run on a hard real-time compatible operating system for reliable hardware control. The Robotics Application Framework was implemented in Java and can be run on any operating system for which a Java Virtual Machine is available. Both tiers communicate with IP-based protocols, which allows deploying the two tiers on different physical systems. This reference implementation is mentioned in various parts of this work.

Design of the Robot Control Core

The Robot Control Core is responsible for communicating with hardware devices in a deterministic manner and executes operations issued by the Robotics Application Framework with hard real-time guarantees. The reference implementation of the RCC tier in the SoftRobot project is called *SoftRobotRCC*. The SoftRobotRCC accepts commands specified in terms of the *Realtime Primitives Interface (RPI)*, described in previous work by Vistein et al. [59, 60]. RPI is a dataflow language, consisting of calculation modules (so called *primitives*) with defined input and output ports that may be interconnected by typed *links*. Primitives may use device drivers to send data to or read data from devices. Commands are specified as graphs of interconnected primitives. Such a *primitive net* is executed cyclically at a high rate (typically 1kHz). This execution is performed deterministically by a real-time runtime environment. A simplified example of an RPI primitive net is shown in Fig. 4.2. A *TrajectoryPlanner* primitive provides interpolation values for movement of a point *MCP* along a trajectory in Cartesian space relative to a coordinate system *F*. These values are transformed by a *Transformation* primitive ac-

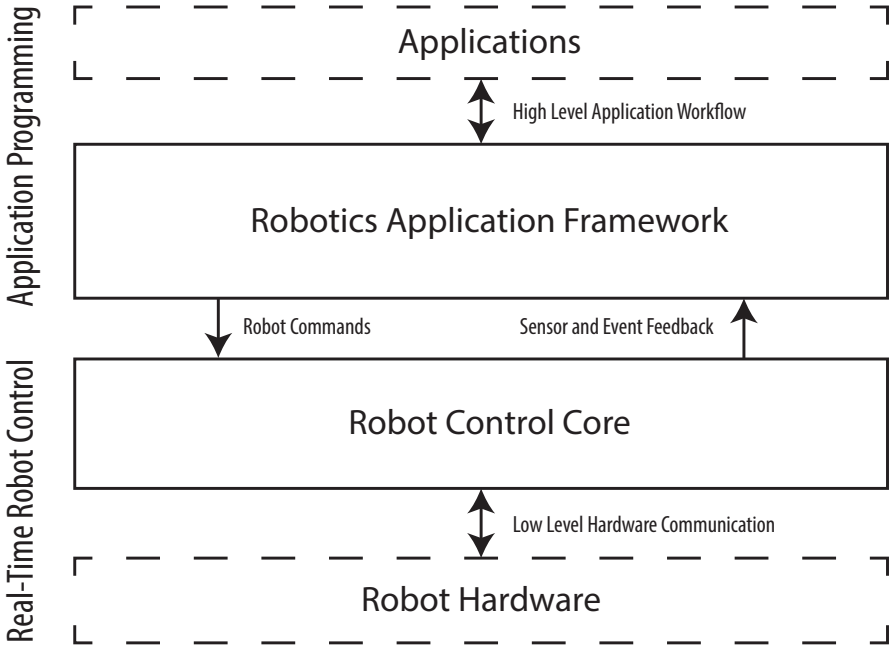

Figure 4.1: The *SoftRobot* architecture, separating real-time concerns from application development in robotics.

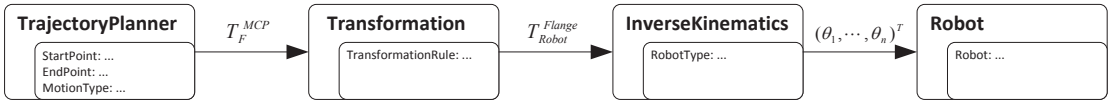

Figure 4.2: RPI primitive net example: a robot follows a Cartesian trajectory (taken from [60]).

cording to some predefined rule such that the resulting values describe the positions of a robot's *Flange* relative to its internal base coordinate system *Robot*. For the resulting values, a primitive *InverseKinematics* calculates a solution of the inverse kinematics function to determine valid joint configurations for each Cartesian interpolation value for a specific type of robot. These values are finally fed to a *Robot* primitive, which internally forwards them to the hardware-specific driver.

The SoftRobotRCC's implementation is based on the Orocos framework (developed by Bruyninckx [58]) and makes use of some scheduling features of Orocos provided through TaskContexts as well as kinematic calculation functions provided by its Kinematics and Dynamics Library (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 primitive nets can also be the basis for synchronisation of multiple devices. The RCC furthermore supports adding new device drivers and calculation modules (even at runtime). It turned out that supporting some of the special concepts of industrial robot control also requires support on the RCC level. Instantaneous switching between the executed primitive nets (see Vistein et al. [61]) can be used to realize e.g. motion blending. Sensor guided motions, in particular force-based manipulation, also profits from this concept. The SoftRobotRCC supports instantaneous command switching with a defined latency between execution of the commands.

Features of the Robotics Application Framework

The Robotics Application Framework defines object-oriented models of devices, operations of such devices and geometric relations. Operations modeled on this level can be flexibly composed based on an event mechanism. Sensors can be integrated in operations to achieve sensor guarded and sensor guided motion. Pre-planned motions, which are a basic element of industrial robotics applications, are supported as well. Accurate motion planning, including dynamic properties, is possible in the Robotics Application Framework. Combined with the RCC's instantaneous command switching mechanism, deterministic motion blending and force-based manipulation can be achieved. The Robotics Application Framework relies on the Robot Control Core for executing all operations with real-time guarantees. For using the SoftRobotRCC, the operations have to be transformable to primitive nets according to the Realtime Primitives Interface.

The notion of multiple robots in one program can be 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, however, requires a particular design of the operation model and the geometric model.

To support extensibility, the Robotics Application Framework should provide a plug-in structure. Robot manufacturers or system integrators may thus provide models of new devices or new operations. Those should be realized by combining existing concepts or adding completely new ones. In this way, also existing parts of the provided Application Programming Interface (API) can be combined to form new specialized APIs for certain application domains.

4.3 Relevance of SoftRobot to this Work

This work will present the design of the Robotics Application Framework tier in the SoftRobot architecture. The features of this tier have been described in the previous section. As the Robotics Application Framework forms the programming interface for application developers, functionality has to be provided in an intuitive way that makes it easy for developers to comprehend the basic concepts, but does not limit functionality required for complex tasks.

The results presented in this work have some overlaps to the ongoing dissertation work of other researchers. The design of the SoftRobotRCC, the Realtime Primitives Interface and the principles of transforming operations of the Robotics Application Framework to RPI primitive nets are subject of Michael Vistein's work. The principles of transforming operations have been investigated by Andreas Schierl as well, who aims to extend the SoftRobot architecture towards mobile robot systems. Alwin Hoffmann is investigating ways of employing the SoftRobot architecture for constructing larger automation systems from reusable components. The correlations will become clear throughout this work and will be made explicit by appropriate references.

4.4 Related Work

The robotics research community has developed many approaches for a software architecture that aims to satisfy the numerous requirements that exist in robotics. The majority of those approaches in the past few years focused on the field of experimental robotics, as industrial robotics was considered a solved problem for some time ([1], p. 983). A major difference between approaches that target industrial robotics and experimental robotics is often the significance of hard real-time constraints during execution. As of today, almost all industrial robot controllers use special operating systems that help to create software which satisfies hard real-time constraints. An example is the operating system VxWorks [62], which is used e.g. in KUKA's and ABB's industrial robot controllers (cf. [63, 64]). By relying on such real-time operating systems and designing a control software that incorporates real-time compliance as an elementary design goal, industrial robots are able to achieve their characteristic precision and reliability. The SoftRobot architecture is targeted at the industrial robotics domain and thus shares this elementary design goal. However, in contrast to existing industrial robot controllers, a modern general-purpose programming language with automatic memory management is used as a high-level programming interface instead of a proprietary language like the KUKA Robot Language (see Sect. 2.2). To achieve this, the programming interface contains a powerful abstraction from real-time critical tasks, which is the separating line between application workflow and real-time compliant robot operations.

The SoftRobot architecture provides an object oriented Robotics Application Framework as an interface to application developers. This programming interface can be used by application developers to directly create applications that have a fixed workflow with limited variation points. It is assumed that a significant number of industrial robotics applications will have this characteristics in the future, and thus this work puts a focus on the careful design of this interface to application developers. However, this does not limit the usefulness of the SoftRobot approach for applications with much more flexible workflows, for robots working in dynamic environments or even to create fully autonomous robots. The SoftRobot architecture was created as an enabling technology: By providing a software design that can be realized in a standard software development platform like e.g. the Java platform, many libraries and tools are available that provide

immediate benefits for developers of robotics applications. Some examples will be given later.

In the 1980s and 90s, there was a major trend to design robot architectures consisting of three interacting tiers, which was denoted as *3T architecture* by Bonasso et al. [65]. The authors describe the bottom layer as "dynamically reprogrammable set of reactive skills coordinated by a skill manager", which was developed by previous research (Yu et al. [66]). The middle layer is formed by "a sequencer that activates and deactivates sets of skills" and is based on the Reactive Action Packages approach by Firby [67]. Finally, the highest layer contains a "deliberative planner that reasons in depth about goals, resources and timing constraints". The work is extended by Simmons et al. [68, 69], which introduce Task Trees as operation model in the middle layer of the 3T architecture. In general, the bottom layer (skills) and the middle layer (skill sequences) are related to the SoftRobot architecture. Conceptually, the Robot Control Core is responsible for executing reactive skills of devices with real-time constraints. The Robotics Application Framework provides application developers means of executing and parameterizing skills according to the application's needs. The separation of those two layers in previous approaches enables encapsulation of real-time matters in the lower layer as well. However, the mechanism for defining such skills and their composition in the Robotics Application Framework is more powerful and will be discussed in detail later in this work.

In recent years, there has been a strong trend to component-based software development in experimental robotics. This approach aims at composing the software for autonomously acting robots from single components with clearly defined interfaces (cf. Brugali et al.'s work [70, 71]). Component-based frameworks for robotics like ORCA [72], RT-Middleware [73], MiRPA [74] and in particular ROS [75] are very popular, in the latter case obviously to some extent because of the good tool support and the large number of available components. Purely component-based systems are not organized in layers, but consist of various components of diverse granularity (e.g., components that compute trajectories, components for planning operation sequences, components that control hardware devices). For structuring components and to increase the cohesion in single components, Radestock et al. [76] suggested to separate the aspects of Coordination, Computation, Configuration and Communication and assign them to different components. This approach was taken up by the robotics community ([71]). Implementing the workflow of an application in a component-based system usually requires multiple configuration and coordination components (cf. Prassler et al. [77]). Coordination components are often implemented using state machine mechanisms (e.g., rFSM [78], ROS SMACH [79, 80]). Most component-based frameworks do not particularly support hard real-time execution of robot tasks distributed across components, which makes it harder to achieve the precision required in industrial applications. However, some frameworks like MiRPA [74] provide hard real-time communication mechanisms even for components distributed across systems.

As described above, the SoftRobot architecture takes a principally different approach to software development for robotics, which provides application developers with an object-oriented programming interface. This is driven by the conviction that work-

flows of industrial robotics applications can be described more intuitive and concise by such an explicit interface than by coordinating a set of component interactions. For larger automation systems that potentially do not use a central coordination instance, there is ongoing work at ISSE that investigates how such systems can be composed from single components that employ the SoftRobot architecture in their implementations. An important research challenge is how independent components can be unified with the tight coupling required to ensure real-time execution constraints. When this challenge is solved, it is possible to employ a multitude of existing tools created by the software engineering community for coordination in such systems. For example, the State Chart XML specification (SCXML, see [81]) defines an executable model of Harel-like statecharts, which is supported by graphical tools (e.g., Apache Commons SCXML [82]). Similarly, approaches exist for modeling executable Petri Nets, including graphical tools (cf. [83, 84, 85]). There is further research by the robotics community (e.g. by Klotzbücher and Bruyninckx [78] or Boren and Cousins [79]) that aims to specify details of executable state charts geared towards particular requirements of robotics, which includes e.g., limited preemptiveness of certain tasks.

Chapter 5

A Modular Robotics Application Programming Interface

Summary: The Robotics Application Programming Interface (Robotics API) is a modular framework for developing robotics applications. The Robotics API is independent of a concrete Robot Control Core. This chapter gives an overview about the most important parts of the Robotics API and the adapter that connects it to the SoftRobotRCC.

The SoftRobot architecture as presented in the previous chapter distinguished two general tiers: The Robot Control Core and the Robotics Application Framework. This chapter will go into detail about the internal structure of the Robotics Application Framework tier and its relationship to the internal structure of the Robot Control Core tier. Fig. 5.1 gives a more detailed overview of those two tiers and their structure in the reference implementation of the SoftRobot architecture. The SoftRobotRCC is an RPI-compatible implementation of a Robot Control Core. It defines communication protocols that can be used to transmit primitive nets according to the RPI specification. Those communication protocols will not be described here, as they are subject of Michael Vistein's dissertation. Rather, the implications for the Robotics Application Framework tier will be discussed.

The Robotics Application Framework is composed from an RCC-independent framework, the *Robotics API*, and an adapter that connects the Robotics API to the SoftRobotRCC. The task of this adapter is to transform (instances of) the Robotics API's models of devices, operations and geometric relations to RPI primitive nets and transmit them to the SoftRobotRCC using its communication protocols. The motivation to introduce a dedicated adapter for this purpose was to decouple this task from the Robotics API and thus allow the development of new adapters to different kinds of Robot Control

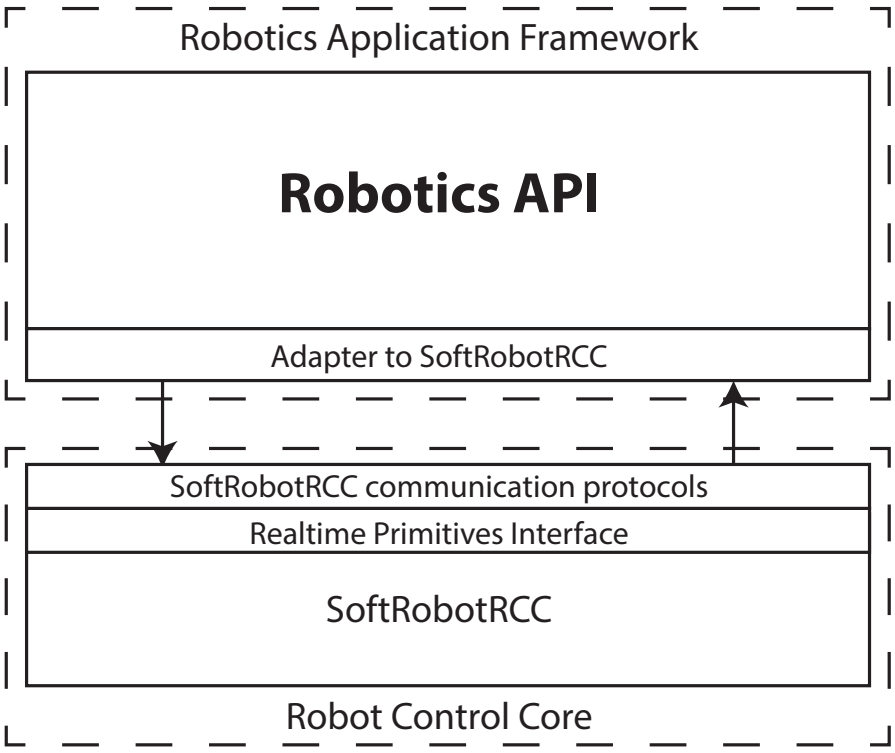

Figure 5.1: The Robotics API and its adapter to the SoftRobotRCC.

Cores. This motivation is similar to those of the Adapter pattern proposed by Gamma et al. [86] (pp. 139). The realization is different, as in the SoftRobot architecture, conceptually different architectural layers have to be bridged, not just different object-oriented interfaces.

In the rest of this chapter, the internal structure of the Robotics API and the basic approach of the adapter to the SoftRobotRCC will be explained. All following parts of this work will focus on the Robotics API's design as an RCC-independent framework for industrial robotics applications.

5.1 The Basic Robotics API Packages

The Robotics API is structured in different packages. This section will illustrate the basic packages and their dependencies using UML package diagrams [87]. The packages *robotics.core*, *robotics.world* and *robotics.activity* are the most basic parts inside the Robotics API. As illustrated in Fig. 5.2, *robotics.activity* and *robotics.world* have an *import relationship* with the *robotics.core* package, or, in other words, both packages *import* the package *robotics.core*. The diagram in Fig. 5.2 is complete w.r.t. import relationships, in the sense that the depicted packages do not import further packages. Thus, *robotics.core*

is completely self-contained, and *robotics.activity* as well as *robotics.world* depend only on *robotics.core*.

The package *robotics.core* defines the basic object-oriented model of the Robotics API. This model introduces the concepts of controllable devices that are able to execute certain actions. Commands that assign actions to devices can be composed to more complex commands, based on an event mechanism. Sensors play an important role as source of events in the Robotics API. Most importantly, the Robotics API core demands the existence of a runtime environment that is able to execute commands, which include devices and sensors, with real-time timing guarantees. The package *robotics.core* contains the definition of an interface to such a runtime environment. This interface defines the contract that all runtime environments have to fulfill, independent of their concrete realization. Details regarding the abovementioned models as well as the contract of the Robotics API runtime environment interface are explained in Chap. 6.

Besides the abovementioned means to operate devices, *robotics.core* provides other important framework concepts. First of all, it defines a mechanism for extending the framework functionality by loading Robotics API extensions in concrete applications. Such extensions may introduce, for example, support for new devices, new kind of operations (for existing or new devices), new runtime environments or all kinds of other functionality. To achieve a modular and fine-grained structure (and thus increase reusability), the Robotics API heavily employs this extension mechanism. This is best illustrated by the fact that neither of the basic Robotics API packages contains any notion of a robot as such. All concepts in *robotics.core*, *robotics.world* and *robotics.activity* are independent of concrete devices or classes of devices. The concept of a robot arm is introduced by the Robotics API Robot Arm extension, whose structure and design is presented in Chap. 10.

Modeling parts of the physical world is an important part of robotics applications. This modeling is usually based on Cartesian coordinates, which are used to describe geometric displacements and rotations between points or objects in space (e.g., the location a robot is mounted and the location of workpieces it should manipulate). In the Robotics API, the package *robotics.world* provides concepts to model entities in Cartesian space in object-oriented applications and to perform calculations based on geometric relationships among the entities. It also connects those concepts to the structure defined in *robotics.core*, e.g. by allowing to relate geometric concepts to certain devices.

A very important contribution of *robotics.world* are sensors that can measure, combine and filter geometric relationships like translational and rotational distances and velocities. In combination with the sensor-event mechanisms of *robotics.core*, geometric sensors can be applied for a variety of typical use cases in robotics. For example, consider some sensor that is measuring the distance to obstacles and is mounted at some point of the robot structure. The robot's motion should be stopped when a certain distance to an obstacle is underrun. The geometric sensors from *robotics.world* can generate events that express such kinds of conditions. Those events can then be used to stop the robot in a defined, hard timeframe. When the robot has a complex structure (e.g., a robot arm

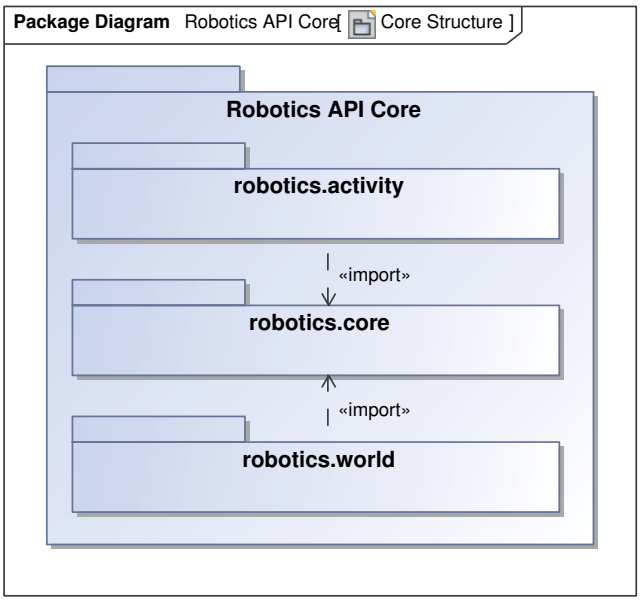

Figure 5.2: Structure of the basic packages forming the Robotics API.

consisting of several joints), it might be necessary to track the distances of all relevant parts of it to capture any dangerous condition. The geometric sensor model provides means to automatically process the sensor data appropriately by deriving the necessary calculation rules from a known geometric structure. The design of *robotics.world* is presented in Chap. 7.

The third basic package of the Robotics API, *robotics.activity*, introduces a model of tasks with a special execution semantics. These tasks are particularly designed to realize typical operations of robotic devices and offer them to developers in a convenient way. To implement the operations, the classes in *robotics.activity* fully rely on concepts of *robotics.core*, in particular its model of composable real-time commands. However, the task model of *robotics.activity* allows to augment operations with meta-data describing their execution. For example, pre-planned motions can provide information about the state of the respective device at a certain point in time. This, in turn, can be employed e.g. for blending over to successive motions. A special task scheduler in *robotics.activity* manages execution of all tasks. It treats devices as shared resources, guaranteeing single tasks exclusive access to the devices they need. Furthermore, the scheduler collects meta-data of tasks and propagates them to successive tasks, which allows those to execute in a context-sensitive way.

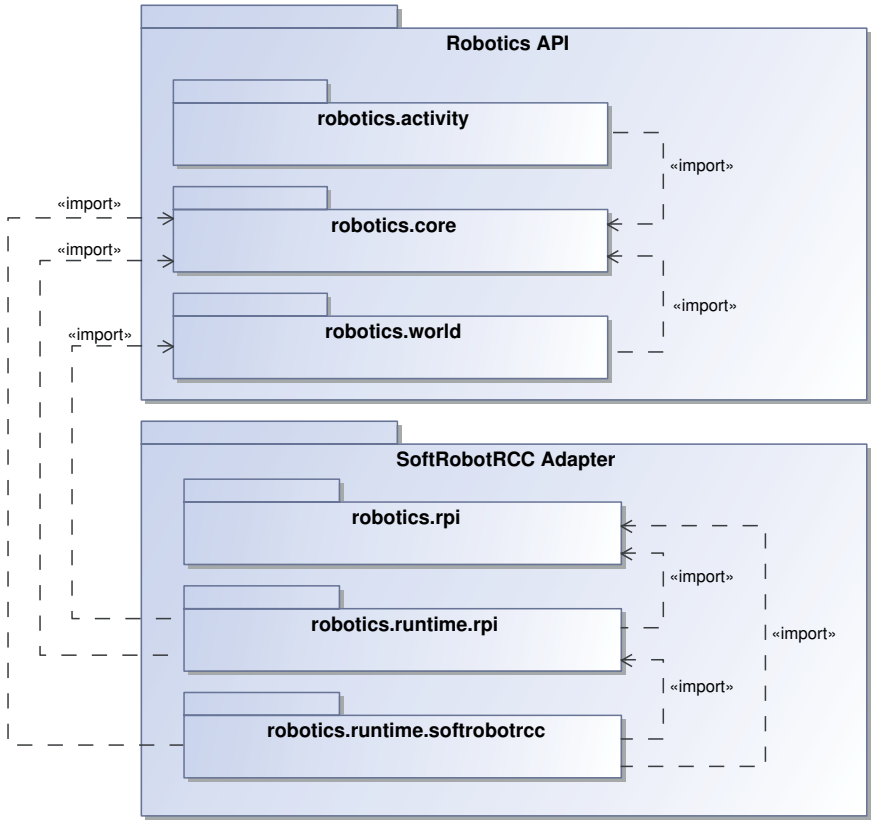

Figure 5.3: Packages of the adapter to the SoftRobotRCC and their relationships to the basic Robotics API packages.

5.2 Packages of the SoftRobotRCC Adapter

This section will introduce the basic packages that are used in the SoftRobotRCC adapter for the Robotics API. Fig. 5.3 introduces those packages and their relationships to the basic packages of the Robotics API:

- *robotics.rpi* includes packages that contain definitions of basic concepts of the Realtime Primitives Interface.
- *robotics.runtime.rpi* defines an extensible algorithm for transforming Robotics API concepts to RPI primitive nets.
- *robotics.runtime.softrobotrcc* provides implementations of the protocols to communicate with the SoftRobotRCC.

The package *robotics.rpi* provides a class library that can be used to create specifications of concrete RPI primitive nets on an object-oriented level. For this purpose, they provide

on the one hand the syntactic composition rules of such nets (e.g., nets contain primitives, which have ports that can be interconnected). On the other hand, the package specifies a set of concrete types of primitives that should be supported by every RPI-compatible RCC. Instances of such primitives in the package *robotics.rpi* can be thought of as functionless *proxy* objects. They can be used to create specifications of RPI primitive nets. When such a specification is loaded to an RCC, all primitive proxies have to be substituted by instantiating the respective implementations in the RCC. The set of primitive types provided by *robotics.rpi* is not complete, as new kinds of devices, sensors or operations provided by Robotics API extensions might also require new kinds of primitive types for hard real-time control. Such extensions thus also have to provide new primitive type definitions, and the Robot Control Core has to be extended by implementations of those primitive types.

Note that *robotics.rpi* does not import other packages. Thus, it can be considered completely self-contained and is not at all bound to the Robotics API. Instead, the connection between the Robotics API and RPI is built by the package *robotics.runtime.rpi*. It contains an algorithm to transform Robotics API concepts to (parts of) RPI primitive nets. Thus, it imports *robotics.core* to have access to the general Robotics API concepts. It furthermore imports *robotics.rpi* which is used to express the created RPI primitive net specifications. The transformation algorithm provides extension points. When new Robotics API concepts are introduced, new transformation rules can be added to the algorithm by using these extension points. The new rules can map new Robotics API concepts to RPI primitive nets that use existing or new RPI primitives.

The packages *robotics.rpi* and *robotics.runtime.rpi* are still independent of concrete RCC implementations (e.g. communication protocols) and can be reused for all RCCs that support RPI. The specific adapter to the SoftRobotRCC is introduced by the package *robotics.runtime.softrobotrcc*. The package imports *robotics.core* to be able to implement the specification of a runtime environment which is defined in *robotics.core*. Furthermore, *robotics.rpi* and *robotics.runtime.rpi* are imported, which enables *robotics.runtime.softrobotrcc* to employ the RPI transformation algorithm and interpret the results correctly.

The internal design of the three packages that are part of the SoftRobotRCC adapter will not be described in detail in this work. Instead, this thesis will focus on the design of the Robotics API itself. However, some examples of how Robotics API concepts are transformed to RPI primitive nets are given in various parts of this work to create a fundamental understanding of the process.

Chapter 6

Software Design of the Robotics API Core

Summary: The Robotics API's core package provides a generic software design that can be used to model and control all kinds of robotic devices. It further defines an interface to a real-time runtime environment, as well as mechanisms for extensibility and configurability. This chapter introduces the Robotics API's general software design principles. The main contributions have been published in [88], [89] and [31].

The central part of the Robotics API framework is the package *robotics.core*. It defines the framework's model of devices and how to control them with hard real-time guarantees, sensors and means to obtain data from them, event-driven behaviour of real-time operations and finally mechanisms for extending the framework core and configuring applications. Extensibility is important in particular, as *robotics.core* mainly consists of interfaces and abstract classes which have to be extended as a prerequisite for creating executable applications. The framework core of the Robotics API is even independent of the concept of a robot.

This chapter introduces the object-oriented structure of *robotics.core*. It starts with an introduction to the model of robotic devices in Sect. 6.1 and continues with the model of sensors in Sect. 6.2. Sect. 6.3 describes the modeling of important system states, which are essential for the model of real-time critical operations, which is defined in Sect. 6.4. A mechanism for composition of device functionality is introduced in Sect. 6.5. A model of exceptions and exception handling in real-time operations is presented in Sect. 6.6. Subsequently, the interface (Sect. 6.7) and the contract w.r.t. execution timing (Sect. 6.8) of a real-time capable runtime environment is defined. An overview of a runtime environment implementation based on the SoftRobotRCC is then given (Sect. 6.9). The

extension mechanism of *robotics.core* is introduced in Sect. 6.10, while Sect. 6.11 illustrates the mechanism for configuring Robotics API applications. The chapter concludes with a discussion of related work in Sect. 6.12.

UML class diagrams are employed in Sects. 6.1-6.7 to illustrate the static structure of the framework. Whenever it is feasible, the class diagrams contain only the interfaces of the relevant concepts, whereas classes are added when appropriate. Furthermore, methods of interfaces and classes as well as class members are omitted when they are not relevant for the scope of the respective section to increase clarity of presentation. In some cases, UML object diagrams are used to give examples of concrete instances of some parts of the object-oriented model. UML sequence diagrams and timing diagrams are additionally used in Sects. 6.8-6.11 to demonstrate the (timing) behavior and interactions between instances of Robotics API concepts during runtime of an application.

6.1 A Generic Model of Robotic Devices

All mechatronic devices that can be interfaced in some way by the Robotics API are abstracted by the interface *Device*. The Robotics API's so called *device model* also includes the interfaces *RoboticsObject*, *OnlineObject*, *DeviceDriver*, *Actuator* and *ActuatorDriver*. This section explains the semantics of each concept as well as their relationships.

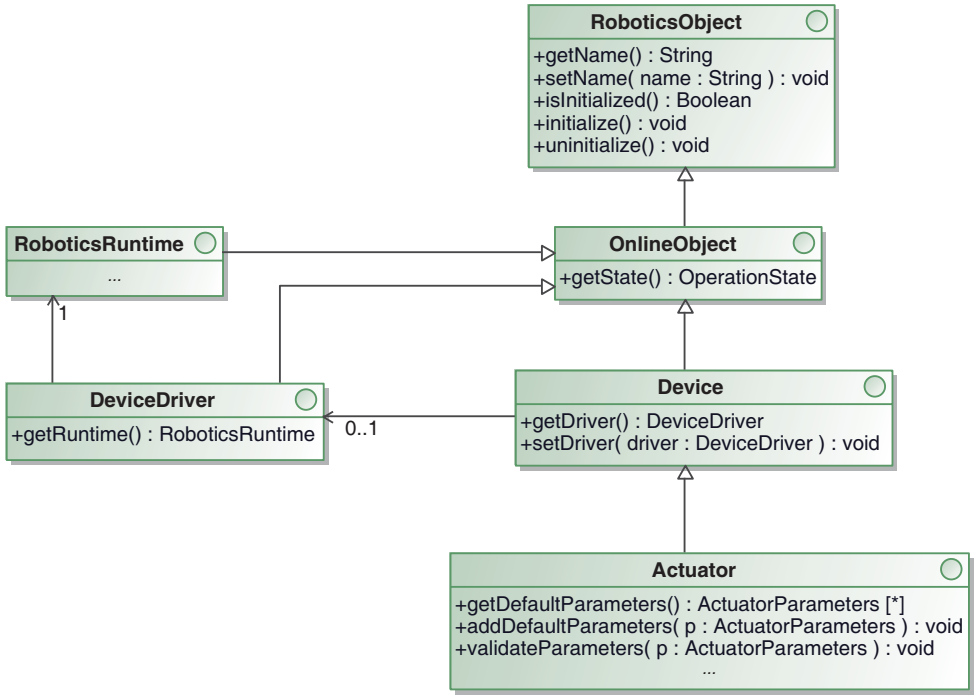

Figure 6.1: Interfaces defining the Robotics API's device model.

Fig. 6.1 gives an overview of the basic concepts of the device model in form of a UML class diagram. In the following, the responsibilities of all those concepts will be defined.

Definition 6.1 (RoboticsObject)**.** A *RoboticsObject* is a named entity that has to be initialized before it can be used. Its name is immutable after initialization, as may be other properties. Uninitializing it restores mutability, but invalidates the RoboticsObject.

The process of initializing and uninitializing establishes a lifecycle for all RoboticsObjects: Before initialization, a RoboticsObject can be configured by setting its properties (e.g. as part of application initialization, see Sect. 6.11). More concrete interfaces or implementations of RoboticsObjects may add more properties, for example the base and flange coordinate systems of a robot arm. When initialized, applications are not allowed to change the configuration of a RoboticsObject. This is important to preserve consistency of the device model during runtime of an application, e.g. by ensuring that some RoboticsObject instance represents the same hardware device as long as it is valid (i.e., initialized).

The *name* property of a RoboticsObject makes it identifyable to a certain extent. The device model as such does not enforce name uniqueness. However, certain aspects of the world model (see Chap. 7) as well as the application startup process of the Robotics API reference implementation (see Sect. 6.11) require unique names for RoboticsObjects.

The next three definitions are closely interrelated:

Definition 6.2 (OnlineObject)**.** An *OnlineObject* is a RoboticsObject that corresponds to some mechatronic device of the real world and can determine the current operational state of this device.

Definition 6.3 (Device)**.** Each device that can interact with the real world is modeled by the interface *Device* in the Robotics API. A Device requires a DeviceDriver for operation.

Definition 6.4 (DeviceDriver)**.** A *DeviceDriver* provides real-time critical access to a physical device by communicating with a real-time capable runtime environment, the RoboticsRuntime.

A complete description of the characteristics and responsibilities of a RoboticsRuntime is given in Sect. 6.4. At this point, a RoboticsRuntime can be thought of as an adapter to a Robot Control Core, which can communicate with hardware devices.

When combining the above definitions, we can state that a Robotics API Device is a named representation of a mechatronic entity, may communicate with this entity via a DeviceDriver and provides information about the entity's operational state. The initialization of a Device can be thought of as "binding" the Device to the respective mechatronic entity.

Being a property of Device, different implementations of DeviceDrivers can be used with the same kind of Device. Thus, Robotics API Extensions can provide support for

accessing Devices with different real-time capable runtime environments. For example, while the Robotics API reference implementation comes with robot arm DeviceDrivers for the SoftRobotRCC, robot manufacturers will provide DeviceDriver implementations that support the real-time execution environments they developed. Note that a Device is not required to be associated to a DeviceDriver, thus implementations of concrete Devices may work without a DeviceDriver. Such Devices are termed *runtime independent*, as they should work with any RoboticsRuntime.

Definition 6.5 (Actuator, ActuatorParameters)**.** A Device that can be actively controlled in one or multiple degrees of freedom is modeled as *Actuator*. Actuators provide a customizable set of *ActuatorParameters* that parameterize all operations they execute.

ActuatorParameters are an important instrument for influencing operation execution. Different types of Actuators support different types of parameters. For example, actuators that can operate in Cartesian space can interpret CartesianParameters, which define among others the Cartesian velocity and acceleration that should be used for motions.

During operation of any Actuator, errors may occur and have to be handled. It is the responsibility of each Actuator to define concrete exceptions which correspond to the different types of errors that may happen. Based on this, appropriate real-time error handling reactions as well as recovery strategies can be defined (cf. Sect. 6.6). The next section introduces the modeling of sensors in the Robotics API as a prerequisite for understanding the command model introduced later.

6.2 Support for Sensors

A very important contribution of the Robotics API is the possibility to access sensors in robot programs, process sensor data and integrate it into real-time control of devices. The Robotics API provides an extensive model of sensors, which will be presented in this section.

The Robotics API *Sensor* class models all kinds of real-time data sources. Such a data source may e.g. originate from a concrete physical sensor (like a robot joint encoder), from processing of other sources' data (like filtering data of a joint encoder) or from any other calculation algorithm (like control values calculated from users' inputs). Robotics API Sensors provide a homogenous, generic way to access all such sources of data and combine them appropriately.

In Fig. 6.2, the relationships between *Sensor*, *SensorListener* and *RoboticsRuntime* are illustrated. The Sensor class itself is a generic class, whose type parameter T specifies the type of data measured by the Sensor. Implementations of Sensor can substitute (*bind*) T by any type that is a subtype of a generic *Object* class. Furthermore, the figure contains some specialized types of Sensors. In the following, those concepts will be clarified.

Definition 6.6 (Sensor)**.** A *Sensor* is a real-time capable source of data which is of fixed type. A Sensor can be restricted to a certain RoboticsRuntime.

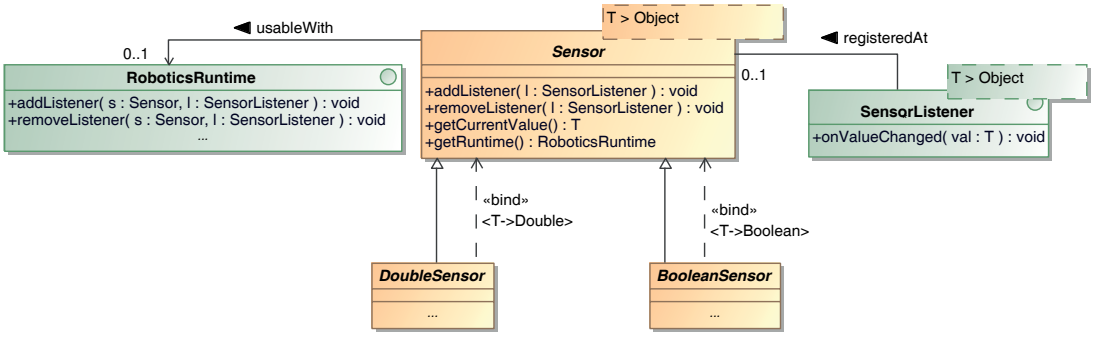

Figure 6.2: Modeling of sensors in the Robotics API, including some basic Sensor types.

When a Sensor is associated to a RoboticsRuntime instance by the *usableWith* association, the Sensor can only be used in the context of this RoboticsRuntime. This mainly affects the use of the Sensor in controlling devices (see Sect. 6.4). *Free* Sensors, i.e. Sensors that are not associated to an instance of RoboticsRuntime, are usable in the context of any RoboticsRuntime.

In a Robotics API application, a Sensor's current value can be obtained anytime by calling its method `getCurrentValue()`. In case the current value is required regularly by an application, a listener can be registered as explained below. In both cases, no real-time guarantees are given w.r.t. latency in delivering new data. Means of employing Sensors as real-time data sources are presented in Sects. 6.3 and 6.4.

Definition 6.7 (SensorListener)**.** A *SensorListener* can be registered with a Sensor and is notified about changes in the data values delivered by the Sensor. This notification is performed without hard timing guarantees.

SensorListeners can be registered with a type-compatible Sensor via the method `addListener()` and unregistered via `removeListener()`. Thus, when compared to the Observer pattern by Gamma et al. ([86], pp. 293), SensorListener has the role of an *Observer*, whereas Sensor is equivalent to an observed *Subject*. Sensors that are restricted to a RoboticsRuntime can delegate the Subject role to this RoboticsRuntime, which also provides methods for adding and removing SensorListeners for a particular Sensor (cf. Sect. 6.7). This allows for implementing appropriate communication channels to observe Sensor values specifically for a particular RoboticsRuntime. The RoboticsRuntime is thus also responsible for calling each SensorListener's method `onValueChanged(...)` when new values are available. If a Sensor is free, it has to implement logic for providing up-to-date values to its SensorListeners.

Figure 6.2 contains two subclasses of Sensors that are defined in the *robotics.core* package. *DoubleSensor* and *BooleanSensor* represent basic Sensor types that can deliver double and boolean values. The core package defines a large variety of concrete implementations of those basic Sensors, which will be discussed in the following. Some

important concrete double- and boolean-type Sensors are displayed in Fig. 6.3. *AbsDoubleSensor*, *AddedDoubleSensor*, *MultipliedDoubleSensor* and *NegatedDoubleSensor* (left part of the figure) are DoubleSensors whose value depends on one or more other DoubleSensors. Their values are the result of calculating the absolute value, the addition, the multiplication and the negation of the values provided by the DoubleSensors they depend on. The DoubleSensor class provides methods `abs()`, `add(...)`, `multiply(...)` and `negate()`, whose implementations create appropriate instances of the abovementioned Sensor types. As those Sensors are DoubleSensors themselves, more complex combinations can be constructed. This principle is applied by some further methods of the DoubleSensor class: For example, the method `subtract(...)` first calls `negate()` on the DoubleSensor that is passed as argument and then passes the result to a call of `add(...)` on the DoubleSensor instance which `subtract(...)` was called on. Similarly, `square()` multiplies the DoubleSensor with itself.

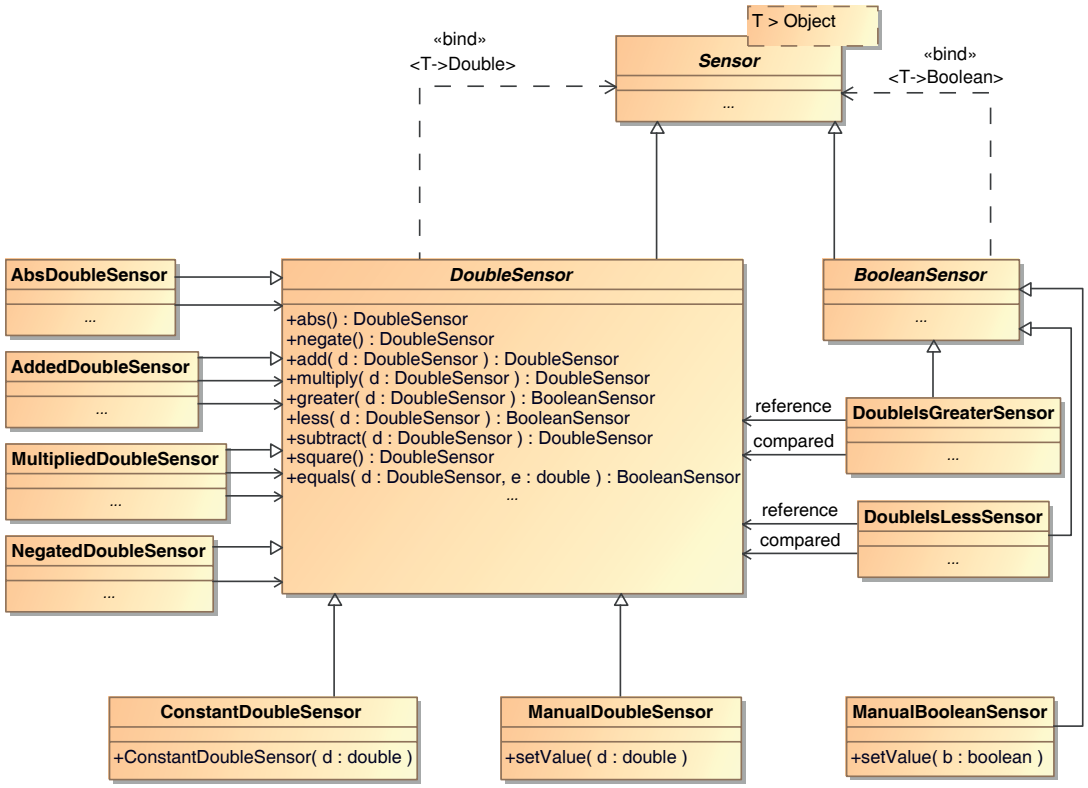

Figure 6.3: Detailed modeling of double-type Sensors in the Robotics API.

The rationale behind creating such explicit, fine-grained Sensor structures is to preserve the information about how a certain Sensor has been constructed. This allows RCC adapters to interpret the single steps in a generic way. For example, the adapter to the SoftRobotRCC is able to transform each type of Sensor to a part of an RPI primitive net and can thus compose primitive nets for arbitrary complex Sensor structures in a

generic way.

In Fig. 6.2, *DoubleIsGreaterSensor* and *DoubleIsLessSensor* are shown (right part of the figure), which are implementations of boolean-type Sensors. They model the result of a comparison operation between two DoubleSensors. These Sensors are used in the implementation of DoubleSensor's methods `greater(...)` and `less(...)`. DoubleSensor's method `equals(...)` uses a combination of NegatedDoubleSensors and DoubleIsLessSensors to construct a BooleanSensor that indicates whether one DoubleSensor's measured value is within an epsilon-range of another DoubleSensor's measured value. BooleanSensor provides a set of methods to combine measurements of boolean-type Sensors with boolean operations, e.g. *or* and *not*. Thus, various BooleanSensors that depend on other BooleanSensors do exist, but will not be presented in detail.

In the bottom part of Fig. 6.3, *ConstantDoubleSensor*, *ManualDoubleSensor* and *ManualBooleanSensor* are depicted. ConstantDoubleSensor models a Sensor that always measures the same constant value, which can be used e.g. to compare the values of a DoubleSensor with a constant value. Similarly, a *ConstantBooleanSensor* exists, but is not shown in the figure. ManualDoubleSensor and ManualBooleanSensor are 'artifical' Sensors, whose measured value can be specified directly by Robotics API applications via the Sensors' methods `setValue(...)`. RCC adapters have to ensure that the values are propagated to the RCC itself, but do not need to give timing guarantees for this process. Those kinds of Sensors give large flexibility to applications, for example to directly influence sensor-based operations. This, however, has to be used with care and with awareness of the missing timing guarantees.

In this section, the Robotics API's Sensor model as such has been introduced as well as basic Sensors defined in the Robotics API's core package. The mechanism for combining Sensors to complex structures has been outlined as well as some particular Sensors with a special semantics. The Sensor model is a very important part of the Robotics API, as it provides the basis for flexible specification of device operations, as well as event-based aspects in controlling devices. Robotics API extensions may provide further types of Sensors with further operations for combining or filtering measured values. Examples are shown in Chapters 9 and 10. The next section introduces means to discretize values measured by Sensors, which is a prerequisite for event-based reactivity in complex device operations.

6.3 States: Capturing Discrete System Conditions

The Robotics API provides the class *State* for capturing interesting parts of a robotic system's state. A State can be *active* or *inactive* and can e.g. express whether measurements of a Sensor exceed a certain treshold, whether a digital field bus input is set or whether a robot operation is already started. In many cases, States can be employed to discretize measurements of Sensors. Like Sensors, States can be monitored in the context of a real-time runtime environment. States can thus be used to trigger real-time critical reactive behavior, e.g. when controlling Devices. In contrast to Sensors, however, States

are only valid in the context of such a control operation and cannot be queried independently (i.e., State does not provide an equivalent to `Sensor#getCurrentValue()`).

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.

The Robotics API core defines various types of States, which are summarized in Fig. 6.4. In the following, the semantics of the various types will be defined. In some cases, concepts of the Command model (see Sect. 6.4) are mentioned and will be briefly explained.

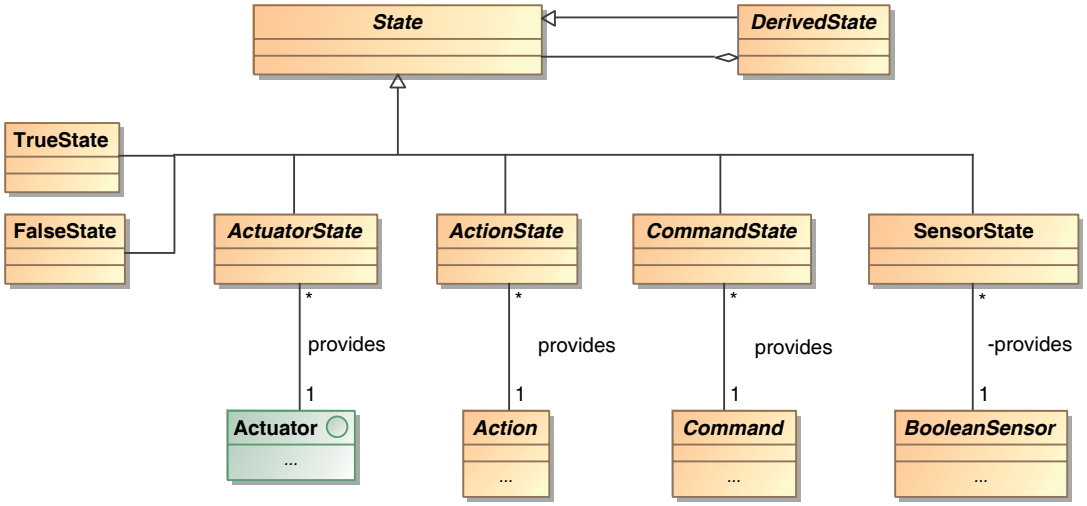

Figure 6.4: The different types of States supported by the Robotics API.

Definition 6.8 (TrueState)**.** *TrueState* is always active.

Definition 6.9 (FalseState)**.** *FalseState* is always inactive.

TrueState and FalseState are useful mainly in defining complex DerivedStates, which will be explained later.

Definition 6.10 (SensorState)**.** Boolean-type Sensors provide *SensorStates* that are active exactly when the Sensor measures the value *true*.

SensorState thus forms the bridge between the Sensor model and the State model. Any BooleanSensor, including those that have been derived from other Sensors that deliver continuous values, can thus be used to derive a discrete State.

The following three generic definitions are tightly related to the Robotics API's Command model, which is explained in detail in Sect. 6.4. At this point, consider an Action

as an elementary operation performed by an Actuator, and a Command as a real-time critical combination of such Actions. Concrete ActuatorStates, ActionStates and CommandStates are introduced in the next Section.

Definition 6.11 (ActuatorState)**.** Actuator-related states that can become active during the execution of a Command are expressed by *ActuatorState*.

Definition 6.12 (ActionState)**.** *ActionState* is the superclass of all Action-related states that can be active when a specific Action is being executed.

Definition 6.13 (CommandState)**.** *CommandState* represents a State that can be active during the execution time of a Command. It models information about the Command's execution status.

The State mechanism becomes much more flexible with the introduction of DerivedStates:

Definition 6.14 (DerivedState)**.** A *DerivedState* is constructed from one or more other States. Its activeness depends on the activeness of the respective other States.

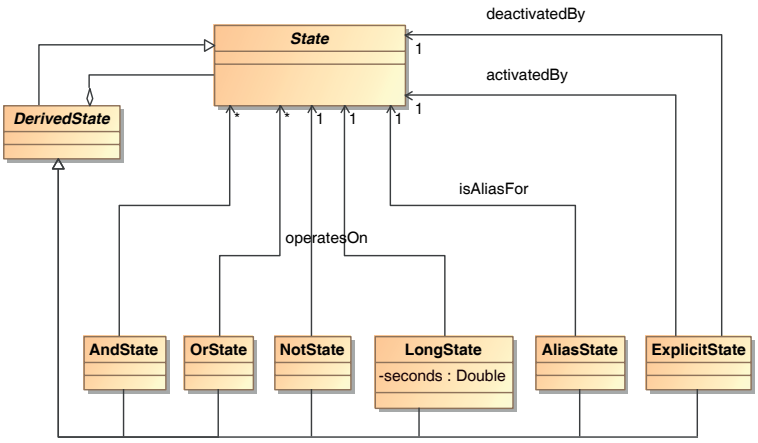

Figure 6.5: Various subclasses of DerivedState can be used to form more complex States.

The exact activeness semantics of a DerivedState is defined individually by concrete subclasses. Fig. 6.5 gives an overview of concrete DerivedStates that have been implemented in the Robotics API. *AndState* and *OrState* represent the straightforward boolean-like combinations of States, where both can build on an arbitrary number of other States. *NotState* is based on exactly one other State and it is active exactly when the other State is not active. *LongState* allows to incorporate timing to define a State: it becomes active as soon as another State has been active at least for a given time. Finally, *ExplicitState* is activated by exactly one other State and deactivated by one other State as well. Its

semantics can be compared with that of an SR flip-flop (see [90], pp. 436), with the activating State in the role of S and the deactivating State in the role of R.

An example of the usage of DerivedStates is given in the instance diagram in Fig. 6.6. It contains some of the Sensors provided by the Robotics API's Lightweight Robot (LWR) implementation. The measurements of two joint torque sensors are monitored. If any of them measures a value above a certain threshold (States `isGreater0` and `isGreater1`) and at the same time the LWR's translational velocity is above some threshold (State `isVelGreater`), the State `critical` becomes active. Note that the diagram has been simplified for clarity: DoubleSensors are in fact not able to directly provide States, but have to be 'converted' to BooleanSensors first by some comparison operation. The BooleanSensors are then able to provide States.

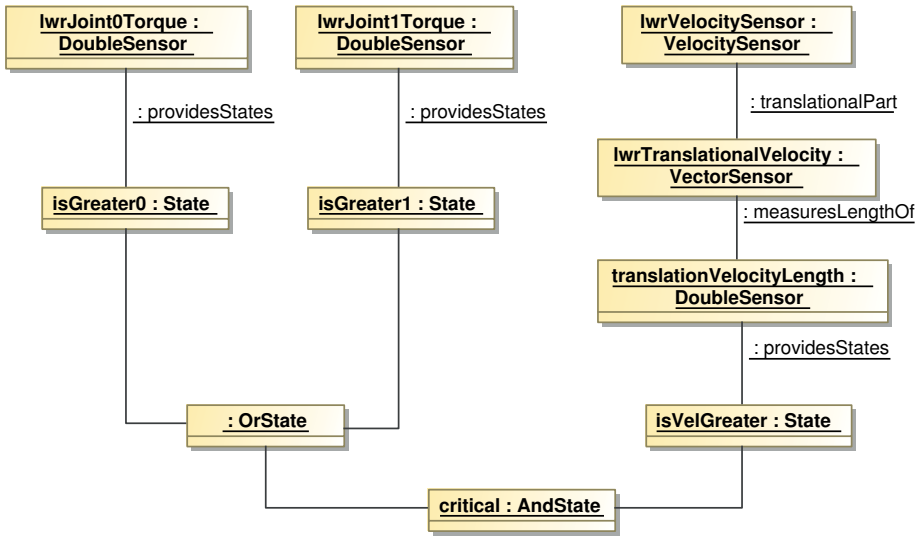

Figure 6.6: Example of combining States provided by an LWR's sensors.

As mentioned before, States are only usable in the context of a device operation. In this case, arbitrary real-time critical reactions based on States can be defined. The next section introduces the Robotics API Command model and will in this context illustrate the power of the State mechanism.

6.4 A Flexible Model of Real-time Operations

For specifying operations that devices should perform, the Robotics API provides the *command model*. This term refers to a set of classes which combine the previously introduced concepts Actuator, Sensor and State to enable a flexible description of what robotic devices should do. This section introduces the relevant classes and illustrates the advantages of the design.

The command model as instance of the Command Pattern

The SoftRobot architecture requires that real-time critical device control is to be executed in the Robot Control Core layer (cf. Sect. 4.2). Thus the challenge in designing a model of operations in the Robotics API is twofold: On the one hand, the requirements (Sect. 4.1) demand programming and control concepts that are much more flexible than the state of the art in industrial robotics. On the other hand, those concepts still need to be compatible with the approach of having a separate real-time layer.

The solution to this challenge proved to be the application of the well-known Command Pattern which was introduced by Gamma et al. [86] (pp. 235), though with some modifications. Fig. 6.7 introduces the basic Command model concepts and relates them to the concepts of Gamma et al.'s Command Pattern. Those are described in the following.

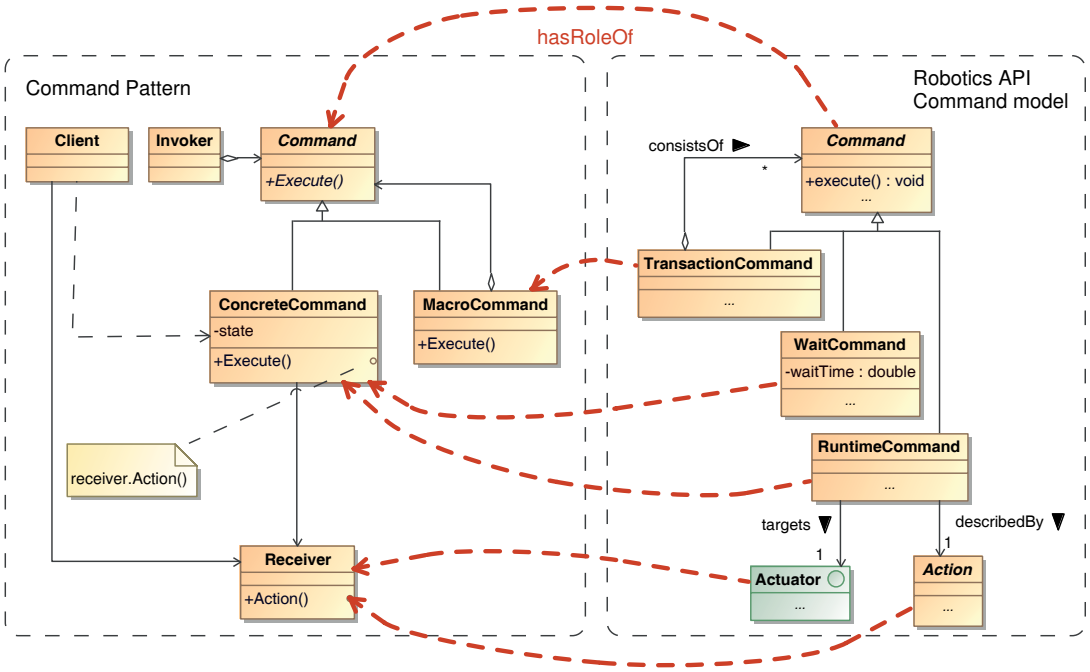

Figure 6.7: The Robotics API Command model (right) as an instance of Gamma et al.'s Command Pattern (left, taken from [86], pp. 235). Curved lines indicate which Robotics API concept corresponds to which part of the Command Pattern.

Definition 6.15 (Command)**.** A Robotics API *Command* is a specification of an operation that is executed atomically in a real-time runtime environment.

The Robotics API's Command directly corresponds to the Command concept of the Command pattern. The Robotics API just adds the guarantee of real-time execution to the contract of Command execution, which is not covered by the Command Pattern as

such. “Executed atomically” as stated in the above definition means that a Command in general cannot be influenced from a Robotics API application once it has been started. However, the Command model allows to flexibly compose Commands from single parts before they are executed. The following definitions provide more detail.

Definition 6.16 (Action)**.** Single actions that Actuators are capable of performing are modeled by the concept *Action*.

Definition 6.17 (RuntimeCommand)**.** *RuntimeCommand* is a Command which specifies that exactly one Actuator should execute exactly one Action. A set of ActuatorParameters specifies Actuator-related execution parameters.

Examples for concrete subclasses of Action 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 target of a linear motion). Actions are not 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 real-time capable runtime environment. Therefore, Actions in the Robotics API can only describe what is to be done, whereas the runtime environment is responsible for implementing real-time compliant execution logic.
- 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.

Note that the set of ActuatorParameters of a RuntimeCommand is not shown in Fig. 6.7. When a RuntimeCommand is created, the set of default ActuatorParameters provided by the Actuator should always be considered. However, when additional ActuatorParameters are specified, those should be preferred over the same kind of ActuatorParameters in the Actuators default parameter set.

Definition 6.18 (WaitCommand)**.** *WaitCommand* is a concrete Command that waits for a certain, defined time, or infinitely.

WaitCommand can be used e.g. to introduce defined execution delays, or to wait for the occurrence of a certain Sensor-based event.

When drawing the comparison to the Command Pattern, the Robotics API's RuntimeCommand and WaitCommand are the equivalents to the ConcreteCommand defined in the Command Pattern. Like ConcreteCommand, RuntimeCommand and WaitCommand store an internal execution state (see Sect. 6.8) and provide a method for executing them. However, the implementation of this method differs: The RuntimeCommand's Actuator (which corresponds to the Receiver in the Command pattern) does not provide methods that implement the action to be performed. Instead, this action is represented by the Robotics API's Action class. For a WaitCommand, the execution logic is defined by its semantics.

Definition 6.19 (TransactionCommand)**.** Multiple Commands can be composed in a *TransactionCommand*. The TransactionCommand forms an atomic context for the execution of all its inner Commands. When marked as *auto-start* Command, an inner Command is automatically started when the TransactionCommand is started.

TransactionCommand provides a method `addCommand(...)` to add a Command to the TransactionCommand's inner Commands. A second method, `addAutoStartCommand(...)`, adds the supplied Command and marks it as auto-start Command. An override of this method allows to additionally specify a BooleanSensor that serves as guard for auto-starting the specified Command: this Command will only be started automatically if the BooleanSensor measures a *true* value in the time instant when the TransactionCommand is started.

Besides the possibility to automatically start inner Commands, a TransactionCommand itself does not provide further means of scheduling the execution of the Commands it is composed of. This task is left to an event-based scheduling mechanism, which will be explained further below.

TransactionCommand takes the role of MacroCommand in the Command Pattern[1]. In contrast to MacroCommand, TransactionCommand does not impose a fixed order of execution of its inner Commands (as noted above).

In contrast to Gamma et al.'s Command Pattern, subclasses of Robotics API Commands usually do not need to provide concrete implementations of the method `execute`. The reason is that all Commands are interpreted by a real-time execution environment according to their type and structure. Thus, the implementation of the method `execute` in the Command class performs two main steps:

- First, the Command is made immutable by a call to its method `seal()`. In this step, a preprocessing of inner events and exceptions is performed, which will be explained later.

[1] MacroCommand is not part of the basic Command Pattern structure, but is discussed as a variant in [86].

- Then, the Command is loaded to a runtime environment for execution, as will be explained in Sect. 6.7.

The Command Pattern specifies two further concepts: Client and Invoker. A Client is responsible for creating and configuring ConcreteCommands, whereas an Invoker calls the Commands' Execute() method. The Robotics API core does not define analogous concepts, but leaves the task of creating, configuring and triggering execution of Commands to the user/the application. This is appropriate for an application framework like the Robotics API.

Reactivity in the command model

As mentioned above, the Robotics API includes an event-based scheduling mechanism to further control Command execution. This has not been included in Fig. 6.7, as it is not covered by the scope of Gamma's Command Pattern. Figure 6.8 gives an overview of the full Command model, including the concepts EventHandler and EventEffect and their relationships.

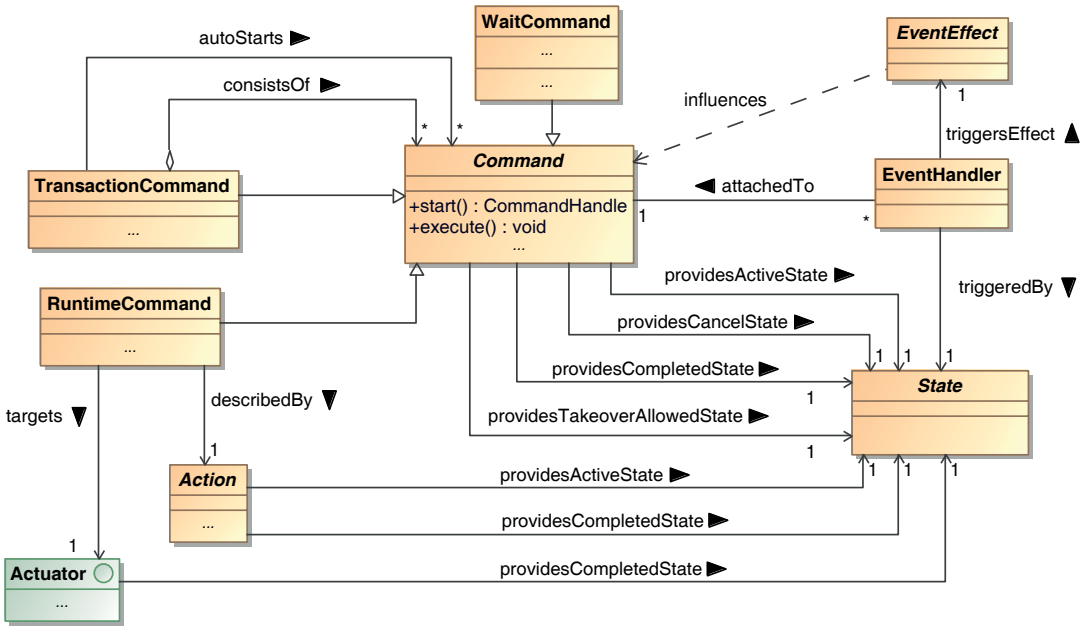

Figure 6.8: EventHandler adds reactive aspects to the Command model.

Definition 6.20 (EventHandler)**.** An *EventHandler* can be attached to a Command, monitors the activeness of a given State and triggers the execution of an EventEffect accordingly.

Definition 6.21 (EventEffect)**.** An *EventEffect* is an executable entity which may affect execution of a running Command.

To each Command, an arbitrary number of *EventHandlers* can be attached. EventHandlers can be used to schedule execution of the inner Commands of a TransactionCommand, as mentioned above. Furthermore, EventHandlers may also influence the execution of RuntimeCommands and WaitCommands. An EventHandler can be parameterized to react to the event that the State it monitors 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 an EventEffect. The Robotics API includes support for the following set of EventEffects:

- *CommandStarter* triggers execution of a particular Command.
- *CommandStopper* forcefully aborts a particular Command.
- *CommandCanceller* gracefully cancels a particular Command.
- *WorkflowEffect* denotes an EventEffect that triggers some logic *outside* of the respective Command, thus leaving the real-time context. WorkflowEffect is parameterized with an instance of the interface *CommandRunnable*, which specifies a single, parameterless method `run()`. Implementations of RCC adapters are expected to call this method in a separate programming language thread.

Note that the Command affected by an EventEffect does in general not have to be the same Command the EventHandler is attached to. The difference between CommandStopper and CommandCanceller 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, CommandCanceller is preferred to give the canceled Command the opportunity to terminate in a controlled way. For example, in case of a robot motion, CommandCanceller should brake the robot until halt, and then terminate the Command. Stopping a TransactionCommand always stops all inner Commands immediately, whereas canceling has to be implemented inside the TransactionCommand.

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

As introduced in Sect. 6.3, States can be provided by Actions, Actuators, Commands and Sensors. Note that, of the Sensors defined in *robotics.core*, only BooleanSensor can provide States. However, further types of Sensors introduced by Robotics API extensions may provide States as well. All Actions provide a State that indicates whether they are currently *active* and if their execution is already *completed*. Note that neither of the two

States is active if execution of the Action has not been started yet. Concrete Actions may provide further States, e.g. to indicate whether a certain via-point of a motion trajectory has been passed. Actuators provide a *completed* State as well. It indicates that the Actuator has fully executed some operation that has been commanded by an Action. Finally, Command provides States that indicate whether it has been *started*, *completed*, has received a *cancel* signal or *allows takeover*.

Cancel signals have to be forwarded to the Action in a RuntimeCommand automatically. In TransactionCommands, however, cancel signals may not be forwarded to any inner Command by default. Instead, each TransactionCommand has to be augmented with appropriate EventHandlers in order to react to a cancel signal (i.e., the activeness of the TransactionCommands *cancel* State) by e.g. canceling a particular inner Command with a CommandCanceller.

A Command can *allow takeover* by activating a particular state. This can be done to signal to the real-time runtime environment that the Command may be stopped and a subsequent Command may be executed as its successor. The subsequent Command has to be completely loaded before by the runtime environment, and has to be declared as the successor to the running Command. The runtime environment is expected to switch between execution of the two Commands instantaneously. This mechanism is explained more in detail in Sect. 6.7 as part of the runtime environment interface definition, and its application to blending between subsequent motions is presented in Chap. 10.

Through the mechanisms presented above, the Robotics API's Command model allows for specifying 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. However, the instantaneous Command switching mechanism mentioned above allows for instantaneous transitions between multiple Command issued by the application workflow. This will be discussed more in detail later in this work.

Command examples

The rest of this Section discusses some examples of concrete Commands that control the KUKA LWR. As a prerequisite, Fig. 6.9 introduces three concrete Actions, two of which are generic robot arm motions. *PTP* is a fast joint-space motion and *LIN* stands for motion along a linear path in Cartesian space. All robot arms, including the LWR, are expected to be able to perform those kind of motions.

The Lightweight Robot provides several control modes, which can in general be combined with any kind of motion. Switching controllers requires execution of a complex protocol which takes considerable time. Thus, an explicit *SwitchController* Action has been designed.

The Command model allows for scheduling control mode switches before executing motion Actions. Fig. 6.10 gives an example of a concrete Command model instance

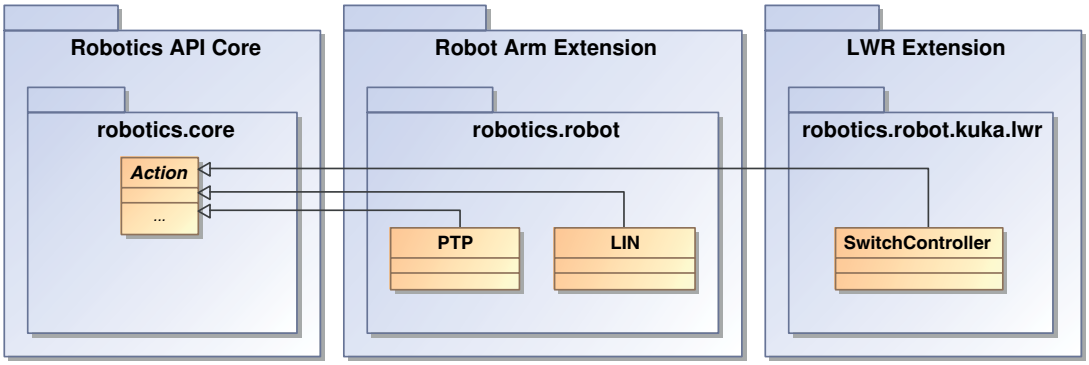

Figure 6.9: Simplified illustration of basic robot motion Actions and LWR specific extensions.

structure. It contains two RuntimeCommands packed in a TransactionCommand. The RuntimeCommand `switchCmd` lets an LWR execute a SwitchController Action and the RuntimeCommand `linCmd` lets the same LWR perform a linear Cartesian motion. `switchCmd` is part of the `autoStarts` association with the TransactionCommand it is contained in and is thus immediately started when the TransactionCommand is started. The EventHandler `startMotion` that has been attached to this TransactionCommand monitors the occurrence of a State indicating that `switchCmd` has been completed. In this case, the EventHandler triggers a CommandStarter which then starts `linCmd`. Cancel signals to the TransactionCommand are forwarded exclusively to `linCmd` by the additional EventHandler `cancelMotion`. This is necessary as the controller switching protocol is uninterruptible, otherwise the robot would be left in an undefined state.

When this Command is executed, the runtime environment guarantees that the RuntimeCommands as well as the interactions incurred by EventHandlers are executed within certain time bounds. In the example, this is not strictly necessary for the transition from `switchCmd` to `linCmd`. It is, however, mandatory for triggering an immediate reaction (i.e. braking) to a cancel signal.

The Command composition mechanism allows for reusing existing Command structures and embedding them in new contexts. This can be employed e.g. for constructing a *LIN to contact* motion as illustrated in Fig. 6.11. This motion operation can be used when an application desires to establish contact between an LWR (or something carried by the robot) and some object in the environment. To evaluate whether contact exists, the operation relies on the LWR's torque sensors, in particular the derived sensor that estimates the resulting end-effector force. In the figure, the `lwrLinCmd` from the previous example is enhanced by an additional EventHandler `contactHandler`. This handler triggers canceling of `lwrLinCmd`. The State monitored by `contactHandler` is an OrState which is active when at least one of the States `xForceInRange`, `yForceInRange` or `zForceInRange` is active. Those States in turn are derived from the respective LWR sensors. Like in a previous example (cf. Fig. 6.6), the construction of the aforementioned

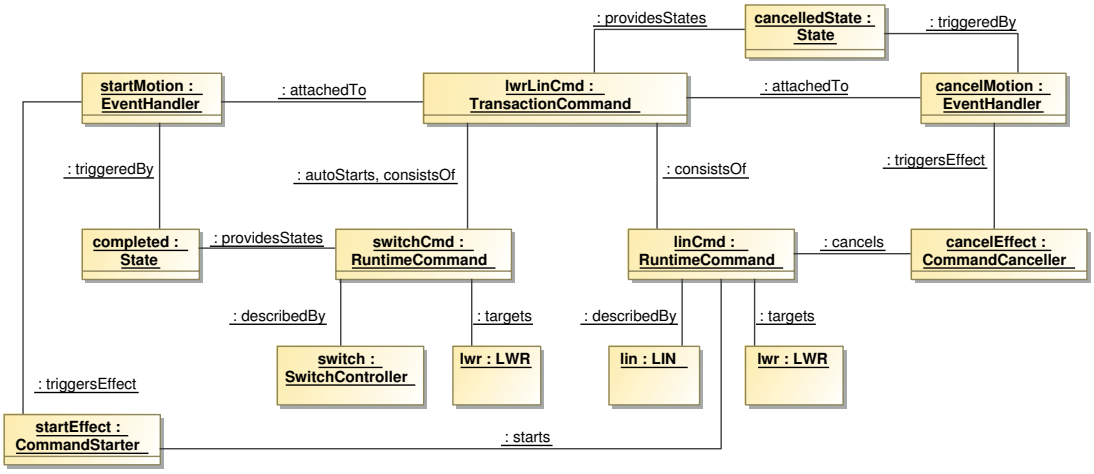

Figure 6.10: Concrete motion Command for an LWR, including controller switching.

States is simplified by leaving away the intermediate BooleanSensors.

This last example demonstrates two aspects of the Robotics API's Command model: On the one hand, any existing Command controlling a certain Actuator can be reused in a black box manner, provided it has been designed carefully. Here, force-based motion guarding logic can be added to a motion Command without knowing its internal structure. However, the Command has to be designed to properly handle canceling, which has to be documented in a guideline for developers. The second aspect demonstrated in the example is the combination of Sensors and States to capture meaningful events in the system.

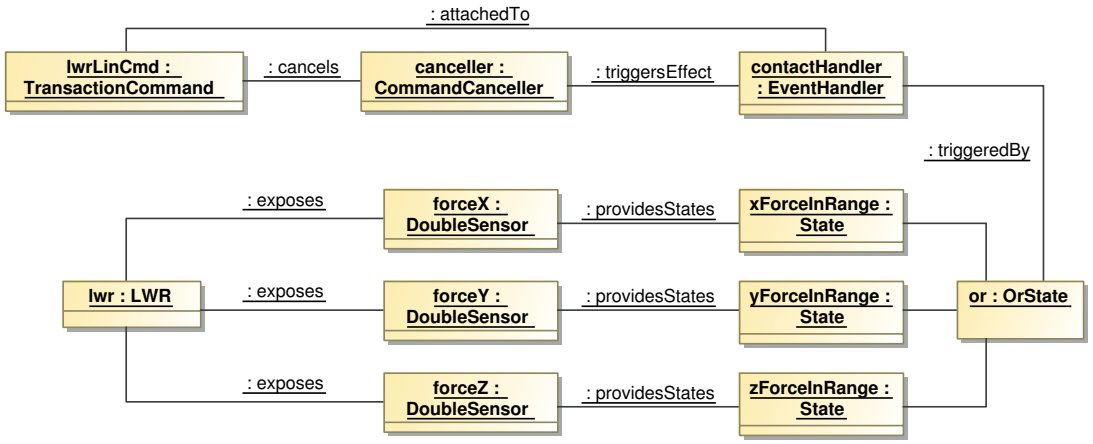

Figure 6.11: Force-guarded motion Command for an LWR.

6.5 Functional Composition by ActuatorInterfaces

The Robotics API's Command model allows to flexibly assign all kinds of operations to Actuators. However, it does not provide any means of expressing compatibility between a certain Action and a certain Actuator. It also does not provide a way of specifying typical operation patterns. Both features are desirable for providing application developers with a set of operations of a concrete Actuator to start with. A straightforward way of realizing this would be to introduce factory methods for Commands in subclasses of Actuator (e.g. a method `ptp(...)` in a subclass of an Actuator that models robot arms). Those methods could be redefined by more concrete types of Actuators to implement them appropriately for those concrete Actuators. The major drawback of this approach is its lack of extensibility, as adding further methods requires adding new subclasses or modifying many existing classes. Consider a subclass of Actuator called RobotArm which is a general model of robot arms consisting of multiple joints. As robot arms are designed to provide as much mechanical flexibility as possible, the set of possible operations is virtually unlimited. Thus, if a new type of operation is developed for a certain use case that is compatible with *all* robot arms, the implementation of the generic RobotArm class has to be modified and an additional method has to be added in order to make the operation usable for all kinds of RobotArms.

To mitigate this problem, the Robotics API's Actuator is designed as a composition of *ActuatorInterfaces* (see Fig. 6.12):

Definition 6.22 (ActuatorInterface)**.** An *ActuatorInterface* is intended to serve as a collection of methods that create operations for a certain class of Actuators. It also defines a scope for ActuatorParameters that are used for these operations.

Each ActuatorInterface has to maintain a collection of ActuatorParameters, which can be retrieved with the method `getDefaultParameters()` and extended with `addDefaultParameters(...)`. The set of parameters should be merged with the Actuator's default set of ActuatorParameters and the result should be used to parameterize operations that are created by the ActuatorInterface. Methods defined in ActuatorInterfaces may also accept additional ActuatorParameters, which should be merged with both of the above sets. All merge operations on those sets should prefer the ActuatorInterface's parameters over the Actuator's parameters, and additional parameters supplied to methods should be preferred over the ActuatorInterface's parameters. In this way, application developers can control parameterization on a fine-grained level and adapt it to each use case appropriately.

To increase the flexibility of parameterization, each ActuatorInterface instance is created *on demand* by an *ActuatorInterfaceFactory* as illustrated in Fig. 6.12. An ActuatorInterfaceFactory can be added to an Actuator with the method `addInterfaceFactory(...)` and increases the set of ActuatorInterfaces the Actuator is able to provide. To access a certain type of ActuatorInterface, the Actuator provides the method `use(...)`. This method iterates through all ActuatorInterfaceFactory instances that it is currently associated with. If a factory is found that can provide an ActuatorInterface of the re-

quested type (indicated by its method `getProvidedInterfaceType()`), the factory's method `createActuatorInterface()` is called. Otherwise, an exception is thrown. Thus, application developers are provided with a new instance of the requested type of ActuatorInterface each time `use(...)` is called. When this ActuatorInterface's set of ActuatorParameters is modified, this will not affect the parameter set of other instances. Thus, each ActuatorInterface defines a unique *scope* for ActuatorParameters.

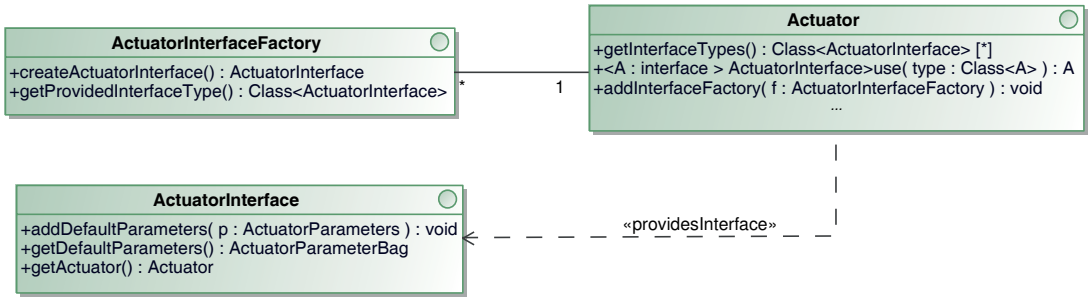

Figure 6.12: Actuators provide a dynamic set of ActuatorInterfaces, which are created by ActuatorInterfaceFactories.

ActuatorInterfaceFactory is an *Abstract Factory* in the sense of the Pattern introduced by Gamma et al. [86] (pp. 87). ActuatorInterfaceFactories can be assigned to Actuators statically by concrete Actuator implementations, or (as often done in the reference implementation) globally to all Actuators of certain type. The Robotics API extension mechanism (see Sect. 6.10) provides means to do so. Coming back to the abovementioned example, all concrete implementations of an abstract RobotArm class can be easily extended with new operations by providing these operations in an ActuatorInterface and adding an appropriate ActuatorInterfaceFactory to all *instances* of RobotArm during startup of a Robotics API application.

In the following, a particular kind of dependency with a stereotype *providesInterface* (cf. Fig. 6.12) will be used in UML diagrams to indicate that an Actuator is associated to an ActuatorInterfaceFactory that can create instances of an ActuatorInterface of a certain type. Effectively, a particular Actuator has a providesInterface dependency to a particular ActuatorInterface *iff* calling this Actuator's method `use(...)` will return an ActuatorInterface instance of that particular type (or subtypes).

More concrete types of ActuatorInterfaces will be presented in later chapters (in particular, in Chaps. 9 and 10). These ActuatorInterfaces will create so-called *Activities*, a concept that will be introduced in Chap. 8. However, it is also possible to use ActuatorInterfaces as factories for Robotics API Commands. The ActuatorInterface concept makes no assumption about the type of objects that are created.

6.6 Real-time Exception Handling

Modern programming languages like Java and C# provide an elaborate exception handling mechanism that allows for separating the main program flow from the flow that has to be performed in error cases. Applications that use the Robotics API can automatically profit from this mechanism. However, the Robotics API core introduces an additional mechanism for handling errors during real-time Command execution. This mechanism is called *real-time exception handling*. In this context, the built-in exception handling mechanism of the programming language used is called *programmming language exception handling* to avoid confusion. The Robotics API's real-time exception handling mechanism ensures that errors during execution of Commands can be handled flexibly by a real-time reaction. Real-time reactions can be used to bring any device in a stable state in case of an error. When the controlled system's stability is restored, the error is propagated to the Robotics API application in form of a programming language exception. Applications can then apply ordinary exception handling mechanisms to recover from the error. For example, consider a welding robot that recognizes an obstacle in its motion path while welding along a seam. Before applying any avoidance or recovery strategy, it should turn off its welding torch in a guaranteed timeframe to avoid damage to the workpiece being welded.

Real-time exception handling is based on attaching EventEffects to a Command. Those EventEffects are executed when certain *CommandRtExceptions* are *thrown* in a running Command. Internally, the Command assigns a Robotics API State to each instance of CommandRtException, which allows for mapping all exception handling logic to the ordinary event model of Commands. Thus, when a CommandRtException is said to be *thrown*, this equals to the underlying State becoming active.

CommandRtExceptions that are thrown, but have no EventEffect attached, will cause the Command to abort execution. This is the main difference to States in a Command, which will just be ignored if they do not have attached EventHandlers. In a hierarchy of TransactionCommands, any CommandRtException that occurred in a child Command and was not handled will be *re-thrown* in the parent TransactionCommand. There, the same mechanism is applied recursively. This is very similar to the exception semantics of programming languages like Java or C# and allows for handling exceptions on any level in a Command hierarchy.

Fig. 6.13 shows the sets of CommandRtExceptions that are part of any Robotics API Command. The sets are defined as follows:

- *inner exceptions* are all CommandRtExceptions inherent to the type of Command. The set of inner exceptions of a RuntimeCommand is the union of its Actuator's exceptions and its Action's exceptions, the set of inner exceptions of a TransactionCommand (also shown in the figure) is the union of the unhandled exceptions (see below) of all its child Commands, and the set of inner exceptions of a WaitCommand is the empty set,

- *declared exceptions* are all CommandRtExceptions that have been declared explicitely for this Command,
- *exceptions* are the union of its inner exceptions and its declared exceptions,
- *handled exceptions* are all exceptions for which an EventEffect has been registered as handler,
- *unhandled exceptions* are the exceptions for which no handling EventEffect has been registered.

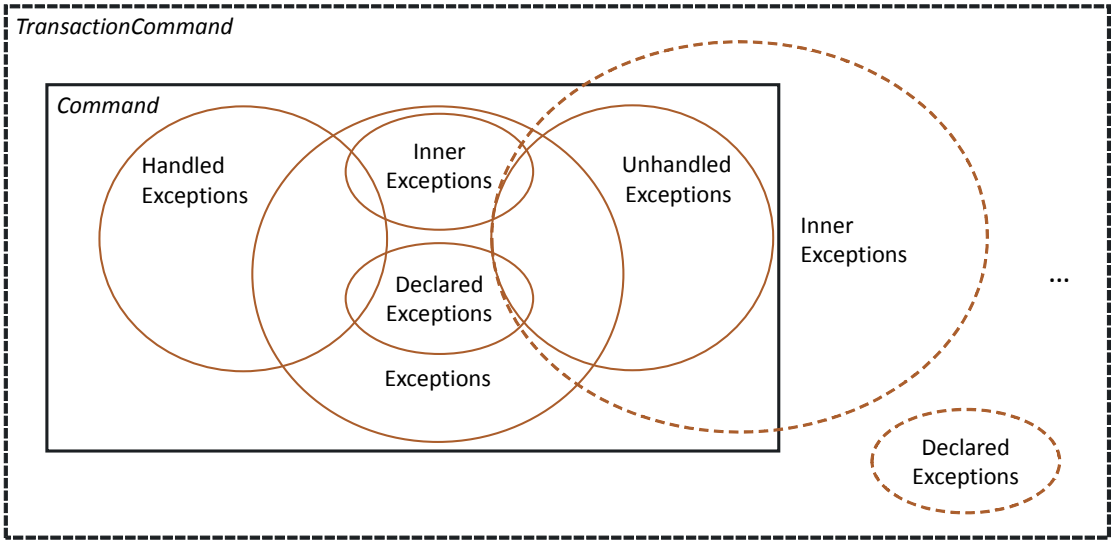

Figure 6.13: Sets of CommandRtExceptions in Robotics API Commands.

The syntax for declaring, throwing and catching real-time exceptions differs from typical programming languages. The main reason for this is the fact that Command structures are composed dynamically when an application is executed. Thus, CommandRtExceptions can not be declared, thrown and caught in a static way. This also implies that the exception sets defined above are highly dynamic and change when operations are performed on the Command. For example, when a new exception is declared at runtime, it is added to the set of declared exceptions. As long as no handler for this exception has been defined, it will also be part of the set of unhandled exceptions. When a handler is defined, the exception will be removed from the set of unhandled exceptions and added to the set of handled exceptions. Thus, the order of operations is important for this exception handling mechanism.

In Fig. 6.14, the hierarchy of exceptions defined by the Robotics API core is displayed, as well as the methods provided by Command related to real-time exception handling. The Robotics API defines *RoboticsException*, which is used as robotics-specific exception in various places. *CommandRtException* is a subclass of RoboticsException and serves

as a base class for all real-time exceptions that can occur when a Command is executed. Each CommandRtException is associated to the Command in whose context it may occur. Furthermore, the Robotics API core defines *ActionRtException* and *ActuatorRtException* that are associated to an Action and an Actuator, respectively, in whose context the exception may be thrown. Each Action and each Actuator are required to provide a list of *exception definitions*, modeled by the concepts *ActionRtExceptionDefinition* and *ActuatorRtExceptionDefinition*. Both define a one-to-one mapping from a State to a particular instance of ActionRtException and ActuatorRtException. In this way, the exceptions that may be thrown when the Action or the Actuator is used in a RuntimeCommand are declared.

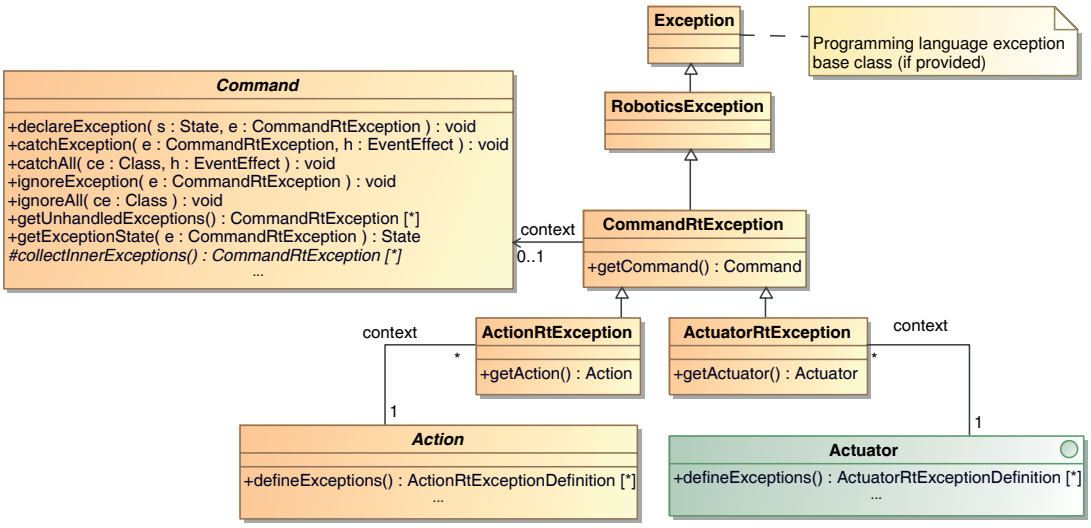

Figure 6.14: Hierarchy of CommandRtException and the Command's real-time exception handling methods.

Command offers the following methods related to real-time exception handling (cf. Fig. 6.14):

- `declareException(s, e)`: Adds the CommandRtException e to the Command's declared exceptions. The exception is thrown when State s becomes active during Command execution.
- `catchException(e, h)`: Registers EventEffect h as handler for CommandRtException e, if e is contained in the Command's set of unhandled exceptions. In this case, the CommandRtException e is added to the Command's set of handled exceptions.
- `catchAll(ce, h)`: Registers EventEffect h as handler for all CommandRtExceptions in the Command's set of unhandled exceptions that are a subclass of the given class ce. Those exceptions are added to the Command's set of handled exceptions.

- `ignoreException(e)`: Adds the CommandRtException e to the Command's handled exceptions without registering an EventEffect for it, if e is contained in the Command's set of unhandled exceptions.
- `ignoreAll(ce)`: Adds all CommandRtExceptions e in the Command's set of unhandled exceptions which are a subclass of the given class ce to the Command's handled exceptions without registering an EventEffect for them.
- `getUnhandledExceptions()`: Gets the set of unhandled exceptions of the Command.
- `getExceptionState(e)`: Retrieves the State that triggers the CommandRtException e, if such exists in the Command's set of exceptions.
- `collectInnerExceptions()`: Collects the set of inner exceptions. The method is abstract in Command and is implemented in each concrete Command with the semantics defined above.

Exceptions can be declared explicitly based on Robotics API States. When declaring an exception, its context is set to the Command it is declared for. If the exception was already scoped to a different context, the declaration is rejected and a programming language exception is thrown. Consider the following sequence of operations on a Command c:

```
c.declareException(s, e);
c.catchException(e, h);
```

The implementation of Command has to guarantee that EventEffect h is executed when State s becomes active and that c's set of unhandled exceptions does not contain e (as it has been caught). This implies that it has to be ensured that the set of exceptions contains no duplicate entries.

Catching as well as ignoring real-time exceptions can be done per exception instance or by specifying a class of exceptions. In the former case, exactly the supplied exception instance is caught when it is thrown. Developers in this case have to be careful to supply the correct instance. When an exception instance that is not part of a Command's unhandled exceptions is supplied to one of the methods for catching and ignoring exceptions, a programming language exception is thrown. Valid instances of real-time exceptions should be retrieved only from the Command in which they are intended to be handled via `getUnhandledExceptions()`. The implementation of this method ensures that all instances of unhandled exceptions from all contexts will be provided. To understand the problem of context, consider the following example: A TransactionCommand that models a series of motions of the same Actuator will contain multiple RuntimeCommands that use the same Actuator. In each RuntimeCommand, all of the Actuator's exceptions can be thrown. Thus, the TransactionCommand's unhandled exceptions may contain multiple ActuatorRtException instances of the same concrete type and with

the same Actuator context, but with different RuntimeCommands as context. It is the responsibility of the TransactionCommand to determine the correct set of exception instances according to its structure.

In some cases, it is not necessary to consider the exact context of a certain type of real-time exception and it is sufficient to handle the occurrence of all instances of this exception in the same way. For this case, Command provides methods for catching or ignoring all instances of a certain class of exceptions. However, the order of operations is important also in this case: Only those instances of the specified class will be captured which are *currently* contained in the Command's set of unhandled exceptions. Exceptions of the same type declared later will not be caught by the same handler.

For the EventEffects that are defined as exception handlers, the same rules like in EventHandlers apply (cf. Sect. 6.4). In particular, CommandStarters, CommandStoppers and CommandCancellers can only be used with a TransactionCommand when the affected Command is one of its child Commands.

During sealing of a Command (method `seal()`, see also Sect. 6.4), remaining unhandled exceptions are handled by a CommandStopper, causing the Command to be aborted on otherwise unhandled exceptions. This is done recursively, beginning with 'leaf' Commands in a Command hierarchy. In contrast to the normal mechanism in `catchException(...)`, the attached CommandStopper is not counted as exception handler, thus each exception remains in the set of unhandled exceptions of this Command. Thus, it is effectively propagated through the complete Command hierarchy. The outmost Command (the one which is currently being started by the application) attaches a *WorkflowEffect* as second EventEffect to all unhandled exceptions. This WorkflowEffect's implementation registers the RtException instance that lead to the abort of the Command's execution in the *CommandHandle* (see Sect. 6.7) that controls execution of the Command. The CommandHandle will throw this exception as programming language exception appropriately.

A further degree of flexibility arises from the possibility to retrieve the State that throws any real-time exception in a Command via the method `getExceptionState(...)` (cf. Fig. 6.14). Note that for this State's activeness, it is irrelevant whether the associated exception is handled or unhandled. Such a State can, like any other State, be combined with further States to create complex conditions matching various requirements. Based on these States, further real-time exceptions can be declared without affecting the handling of other exceptions.

The real-time exception handling mechanism, though part of the Command's interface, can be completely mapped to the event handling mechanism based on States and EventEffects by implementations of Commands. Thus, all real-time runtime environments need not consider any means of throwing or catching such exceptions. Instead, they merely have to activate the appropriate States. Note that these States do not differ from all other States and can thus be handled in the same way.

6.7 Integrating a Real-time Execution Environment

As noted before, execution of Commands needs to be performed with real-time guarantees. The SoftRobot architecture requires a Robot Control Core to take care of such tasks. 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 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 contains a RoboticsRuntime that transforms Robotics API Commands to real-time dataflow graphs according to the Realtime Primitives Interface specification. Details about this concrete RoboticsRuntime implementation will be given in Sect. 6.9. This section will focus on the contract of the RoboticsRuntime interface.

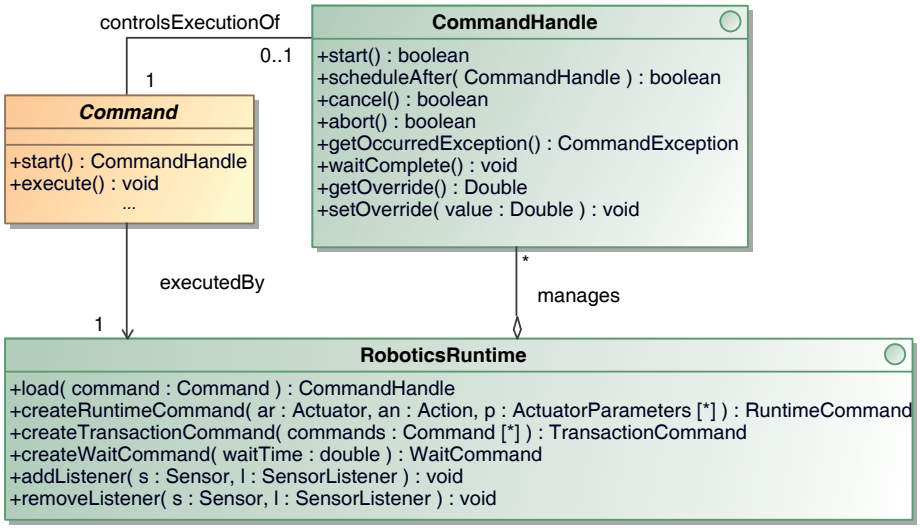

Figure 6.15: A RoboticsRuntime serves as an adapter for executing Commands on a Robot Control Core.

The lifecycle of each Command is controlled by RoboticsRuntime and CommandHandle (see Fig. 6.15). They have the following roles:

Definition 6.23 (RoboticsRuntime)**.** A *RoboticsRuntime* can create Robotics API Commands and load such Commands to a real-time runtime environment. Furthermore, it has to manage SensorListeners and provide them with up-to-date values of the Sensors they observe.

Definition 6.24 (CommandHandle)**.** The execution of a Command is controlled by a *CommandHandle* once the Command has been loaded by a RoboticsRuntime.

RoboticsRuntime is an *Abstract Factory* (cf. Gamma et al. [86], p. 87) for the basic Robotics API Command types. For creating Commands, RoboticsRuntime provides the methods `createRuntimeCommand(...)`, `createTransactionCommand(...)` and `createWaitCommand(...)`. To create a RuntimeCommand, Action, Actuator and a set of ActuatorParameters to use in this RuntimeCommand have to be supplied. The supplied ActuatorParameters should be merged with the Actuator's default parameters during creation of the RuntimeCommand. When creating a TransactionCommand, a list of Commands can be supplied which are added as children to the newly created TransactionCommand. The creation of a WaitCommand requires that the wait duration is supplied. As the RoboticsRuntime serves as factory for Commands, it can perform preliminary steps to loading the respective Commands as soon as they are created, like e.g. opening appropriate communication channels to the real-time runtime environment.

Loading of arbitrary complex Commands is performed by the RoboticsRuntime's method `load(...)`. The method is required to terminate only once the given Command has been loaded completely to the RCC (provided no error has occurred). When it terminates, it returns a CommandHandle which controls the execution of the Command. The methods `start()` and `scheduleAfter(...)` both trigger starting of the Command. A Command-wide override that modifies the execution speed of Actions can be altered via `setOverride(...)`. `cancel()` and `abort()` can end execution preliminarily. The method `waitComplete()` blocks the Robotics API workflow until execution has completed. Once execution is complete, a CommandException which led to termination of the Command can be obtained via `getOccurredException()`, if such exists. The methods `start()` and `execute()` of Command start execution: `start()` can be seen as a macro which first calls the RoboticsRuntime's `load()` method to load the respective Command and then calls the returned CommandHandle's method `start()` to start the Command. The method then returns the CommandHandle if starting was successful. A Command's method `execute()` calls its `start()` method and subsequently `waitComplete()` of the returned CommandHandle to block until Command execution has finished.

The two different methods of CommandHandle to start Command execution need some explanation. The method `start()` immediately starts execution of a Command without preconditions. In contrast, `scheduleAfter(...)` schedules the Command for instantaneous execution right after another Command which has already been loaded and is referenced by the respective CommandHandle. This scheduling mechanism signals the RCC to perform a seamless transition between execution of those Commands. Such a transition is mandatory in all cases where continuous control of an actuator is required across multiple Commands, e.g. when blending across motions or performing a series of operations that maintain force closure between the actuator and its environment. The application of this scheduling mechanism to achieve motion blending is explained in detail later in this work (see Sect. 10.2).

The distinction between a CommandHandle's methods `cancel()` and `abort()` is also worth noting. Both methods can be used to signal a running Command to terminate execution prematurely. The semantics is similar to that of the EventEffects Command-

Canceller and CommandStopper, which have been explained in Sect. 6.4: *Cancel* triggers a 'soft' termination which has to be handled by each Command internally, if possible. *Abort* causes a 'hard', immediate termination of a Command, which may leave actuators in an uncontrolled state. In contrast to the respective EventEffects, the methods `cancel()` and `abort()` do not give guarantees w.r.t. the latency of the termination signal's arrival at the RCC environment. In other words, the CommandHandle's methods may take longer to execute than expected and should thus not be used for safety-critical functionality.

6.8 Atomic Execution of Real-time Operations

From the view of a Robotics API application, Command execution in general is an atomic process that can only be influenced by canceling 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 to modify a motion path) must be known before a Command is started and be integrated in its definition. As has been explained in Sect. 6.4, the Command model provides sufficient flexibility in defining Commands. A Robot Control Core, interfaced by a RoboticsRuntime, is responsible for executing Commands and has to provide certain timing guarantees during execution. In the following, the expected timing guarantees are presented.

The loading and starting process of a Command is not required to hold any hard timing limits. Loading a Command may include certain steps that have to be performed in the RoboticsRuntime implementation and thus inside the (not real-time capable) Robotics API execution environment. Starting a previously loaded Command is triggered from inside Robotics API applications as well and usually requires (inter-process) communication between this application and the RCC, which can hardly be done within strict timing bounds. Thus, the following assumption should guide the design of any Command composed in applications: When a Command terminates its execution, it is guaranteed that none of the Actuators that were controlled by this Command is left in an unstable state in which it can cause harm to the environment.

This assumption in turn also defines a contract for RoboticsRuntime implementations that may transform Commands to the RCC input format, and the RCC that executes the transformed Commands. Command implementations (in particular implementations of Actions) have to ensure that they do not terminate prematurely, leaving some Actuator in an unstable state. The RCC in turn has to take care that Command execution is not interrupted and that certain conditions are fulfilled when an instantaneous transition is performed from a running Command to its scheduled successor. Note that the external aborting of a Command by an application is not covered by the above hypothesis, where *external* means calling a CommandHandle's method `abort`. This user-triggered operation is by definition 'dangerous' in that it can lead to unpredictable behavior.

When a RoboticsRuntime executes a Robotics API Command, its implementation has to guarantee that it can hold several timing bounds:

- When any RuntimeCommand is started, the Action's *Active* State has to be activated within a defined time.
- In any RuntimeCommand, control values calculated by the Action must be provided to the Actuator within a defined time when the Action's *Completed* State is not active.
- For any TransactionCommand, child Commands that are part of the *autoStarts* relationship have to be started at the same time instant at which the TransactionCommand is started, if the BooleanSensor that guards starting measures a *true* value at this time instant.
- The execution time of any WaitCommand must not differ more than a defined time from the wait time specified in its definition.
- For any EventHandler whose EventEffect influences some Command, the effect has to be applied to this Command within a defined time.
- Any State has to be activated and deactivated within a defined time after a change in the respective physical or logical system state has been detected. This also applies to derived States, regardless of their structure.
- When a Command is started, its *Active* State has to be activated within a defined time.
- When a Command is canceled, its *Cancel* State has to be activated within a defined time. When the Command is terminated, this State has to be deactivated within a defined time.
- When a running Command's *Completed* State becomes active, the Command has to be terminated within a defined time.
- When a running Command's *TakeoverAllowed* State becomes active and a successor for this Command has been scheduled, the Command has to be terminated within a defined time and the successor has to be started within a defined time.

Fig. 6.16 illustrates some of the abovementioned guarantees based on the example from Fig. 6.11. The figure demonstrates an example run of a force guarded linear motion performed by an LWR. This motion is modeled by `lwrLinCmd` in the figure. The Command can be in state *New*, *Running*, *Cancelling* and *Terminated*. The transition from Running to Cancelling is triggered by the CommandCanceller `canceller`, which itself reacts to the OrState `or` becoming active. This derived State is composed from several States that become active when the LWR's force sensor measurements exceed some threshold. From those States, only `xForceInRange` is depicted in the figure. At the bottom of the

figure, three timing specifications are annotated. From the timing requirements stated above, we can derive that the total time `t` needed to bring `lwrLinCmd` to state Cancelling after the DoubleSensor `forceX` has measured a force above a treshold is bounded. In contrast, the time needed to start `lwrLinCmd`, which includes loading and starting it on the RCC, is not bounded. The same applies to the time needed to signal Command termination to the Robotics API, which usually involves indeterministic inter-process communication.

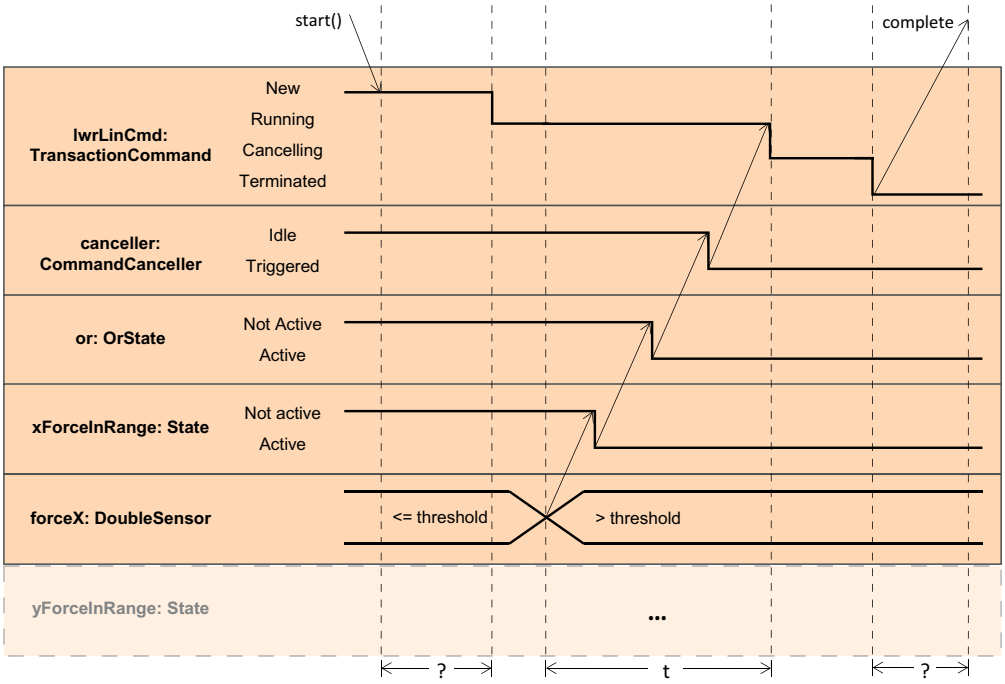

Figure 6.16: UML timing diagram showing the execution of a force-guarded motion Command from the view of a Robotics API Application.

6.9 Overview: Transforming Commands to RPI Primitive Nets

In Section 6.7, the concept *RoboticsRuntime* has been introduced as interface to a hard real-time runtime environment for Robotics API Commands. Implementations of this interface serve as a factory for Commands and have to be able to execute complex Command instances in a deterministic manner, guaranteeing that the timing bounds defined in the previous section are held.

In the SoftRobot project, an implementation of the RoboticsRuntime interface for the SoftRobotRCC has been developed, which is termed *SoftRobotRuntime*. This implementation transforms Robotics API Commands to RPI primitive nets, using a generic

and extensible algorithm. The SoftRobotRCC is able to execute RPI primitive nets deterministically with a fixed and guaranteed cycle rate. Within one execution cycle, the complete RPI primitive net is fully evaluated. It is assumed that all values that are calculated within the same execution cycle are available *instantly*.

When a Robotics API Command has been transformed to a primitive net, this net is serialized in a format specific to the SoftRobotRCC and subsequently transmitted to this RCC. The SoftRobotRCC has to load the primitive net, instantiate implementations of the specified primitive types and is then able to execute the primitive net in a deterministic way.

This section will give an overview of the generic transformation algorithm that creates RPI primitive nets from Robotics API Command instances. Instead of presenting details of the algorithm, the focus is on creating an understanding of the principles of the process. Details can be found in [91].

The basic idea of the algorithm is "to transform the basic building blocks of the [Robotics API] commands into corresponding data-flow [RPI] net fragments with certain responsibilities. These fragments are composed according to given composition rules, leading to a complete data-flow net that can be executed on a Robot Control Core to create the behavior described by the command." [91].

Figure 6.17 gives an overview of the structure of an RPI primitive net that a Robotics API RuntimeCommand is transformed to. All rectangles represent *fragments* of the primitive net. Fragment *ports* are represented by pointed symbols, where in-ports (dataflow sinks) point inside a rectangle and out-ports (dataflow sources) outside of it. In the central part of the figure, fragments with rounded edges are located which represent the transformation results of the Robotics API *Action* and *Actuator* that are part of the RuntimeCommand. For example, Actions that resemble pre-planned motions might be transformed to a single, parameterized trajectory generator primitive (or to a net of primitives) that calculates motion interpolation values. An Actuator that should perform such a motion can be transformed to one or multiple primitives that are able to forward interpolation values to the physical device.

Additionally, a *Converter* may be part of the net, for example when a motion Action has produced Cartesian coordinates as output that have to be converted to a robot Actuator's joint space before they can be forwarded to this Actuator. The full fragment shown in the figure represents the transformation result of the complete RuntimeCommand. This result is also a fragment itself, which has still unconnected in-ports and out-ports. It can either be embedded in a larger primitive net (e.g. the result of transforming a TransactionCommand), or completed by connecting the remaining ports to special start, cancel and termination primitives of the RPI execution environment.

Inside the fragment created from the RuntimeCommand, the lifecycle of the primitive net is implemented by the fragments *Activation*, *Stop* and *Completion*. The Activation fragment has to decide, based on its in-ports, whether the rest of the primitive net should be set active. If so, cancel signals have to be forwarded to the Action and Actuator fragments. The Completion fragment signals that the primitive net is completed once the

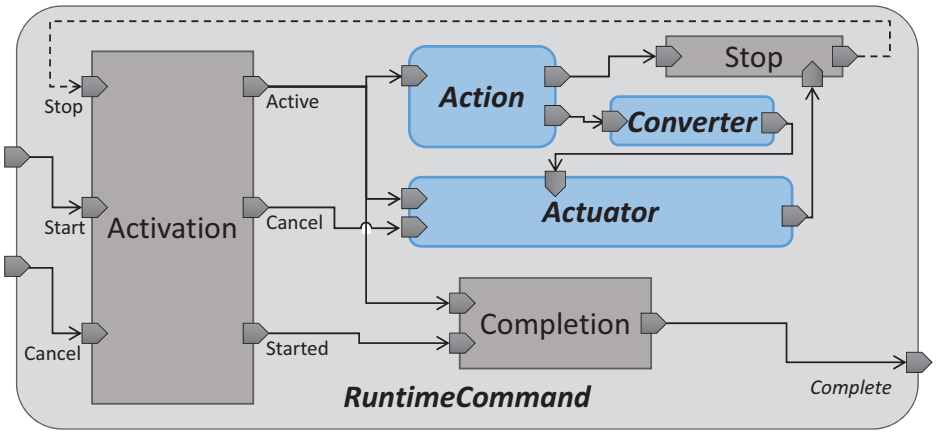

Figure 6.17: Structure of an RPI primitive net generated from a Robotics API RuntimeCommand (adapted from [91]).

execution has been started and the net is no longer active. The Action and Actuator fragments can both signal that they have completed a certain part of their execution. If both fragments do this, the Stop fragment will signal this to Activation. The dotted dataflow connection from Stop to Activation fragment indicates that this dataflow is delayed by one execution cycle. This is necessary in order to prevent a non-terminating dataflow loop (cf. [60]). Note that the Activation, Stop and Completion fragments may be realized as single RPI primitives (for better execution performance) or as true fragments of interconnected primitives (for more flexibility in adapting the transformation algorithm).

TransactionCommands are transformed by recursively transforming all child Commands and adding lifecycle management logic like with RuntimeCommands. Sensors are transformed to fragments whenever required in some Command. States and EventEffects are also transformed as needed, where EventEffects may influence the lifecycle logic of Command fragments. Note that most Commands contain numerous States, in particular due to the exception handling mechanism inside the Commands, which heavily relies on States. In the SoftRobot reference implementation, many different RPI primitives of quite simple type are employed. For example, primitives can calculate unary and binary boolean operations or add, multiply and interpolate numeric values, which is used to compose trajectory generator fragments. Thus, a RuntimeCommand for a point-to-point motion of a robot arm with seven joints results in a net of about 600 primitives.

The transformation algorithm provides various extension points, e.g. for integrating new transformation steps for Actions, Actuators and Sensors. Robotics API extensions that contribute such new entities have to define concrete transformation steps for each of those entities in order to integrate them in RPI primitive nets. If new types of RPI primitives are required for this purpose, implementations of those primitives have to be

integrated in the Robot Control Core used. By designing extensions for certain classes of Actuators, Actions and Sensors, it is possible to provide support for concrete realizations of those entities with minimal extensions to the RPI transformation algorithm. This will be demonstrated in further parts of this work.

6.10 A Mechanism for Extensibility

The last sections introduced the core architecture of the Robotics API. While this architecture is flexible and powerful, many of its concepts are still abstract and need to be extended to be usable. Several examples illustrated this (e.g., LWR as example of a concrete Actuator, PTP and LIN as concrete Action implementations). However, all such concrete Devices, Actions or Sensors are not part of the *robotics.core* component. This design was chosen deliberately in favor of a slim and clearly structured design of this core component. This section will present a plug-in mechanism which is part of *robotics.core*. Based on this mechanism, arbitrary extension components can be developed to extend the Robotics API by concrete hardware support and control functionality.

An extension mechanism should support Robotics API extension developers in several ways:

- Allow for registering new types of RoboticsObjects like RoboticsRuntimes, Devices and DeviceDrivers in the framework,
- extend existing RoboticsObjects, e.g. RoboticsRuntimes, possibly registered by other extensions, with new functionality,
- give developers the possibility to access other parts of the framework at some defined time, e.g. for registering subscribers to publisher objects.

A further independent goal in the development of the Robotics API extension mechanism was to support different application paradigms. A classic application created on top of the Robotics API's reference implementation is based on the Java platform, which implies having one application entry point (a *main* method). In this scenario, a feasible approach is to demand the application to call some framework initialization logic and subsequently provide access to framework functionality e.g. through singleton objects. A second scenario that has been investigated was to compose a Robotics API application from multiple components as defined by the OSGi framework [92]. In this case, each component has its own entry point, which substantially changes the lifecycle of an application. This approach also is hardly compatible to the singleton concept, as classical singleton classes would reside only in the scope of one component and would not be easily accessible by other components. Thus, the extension mechanism has to be flexible in the sense that it should support different ways of integrating extensions into the framework core at application startup. For example, in a pure Robotics API Java application, RoboticsObjects provided by extensions could be registered in some singleton object. In contrast, in an OSGi application, new RoboticsObjects could be registered at some

dedicated OSGi component using appropriate component communication mechanisms. The rest of this section presents a Robotics API extension mechanism which is flexible enough to support both (and possibly further) scenarios. The rest of this work assumes that Robotics API applications are pure Java applications.

The Robotics API Extension mechanism is based on the concepts *Extension* and *ExtensionConsumer*. Both interfaces and their relationship are depicted in Fig. 6.18, as well as three concrete Extension types defined in *robotics.core*. The interface Extension is an empty interface and is solely used as template parameter type in the definition of ExtensionConsumer. ExtensionConsumer defines the methods `addExtension` and `removeExtension`. The former can be called to register an Extension at a matching ExtensionConsumer, whereas the latter unregisters the Extension. Different implementations of ExtensionConsumer can accommodate for different application scenarios, e.g. the abovementioned pure Java and OSGi scenarios.

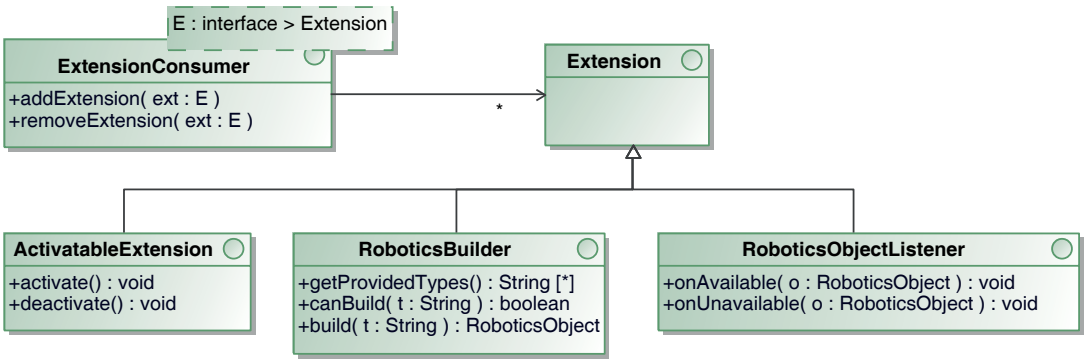

Figure 6.18: The Robotics API extension mechanism is based on the interfaces Extension and ExtensionConsumer.

During application initialization, all available ExtensionConsumers have to be located before any Extension can be loaded. In the next section, an algorithm is introduced which ensures this during startup of a Robotics API Java application.

6.11 Configuration Management and Application Lifecycle

To properly initialize the Robotics API and initialize RoboticsObjects like Devices and RoboticsRuntimes, applications may use the class *RoboticsAppLoader* (see Fig. 6.19). Its singleton instance provides a method `loadConfigFile(f : File)` which can load configurations for all kinds of RoboticsObjects from a configuration file. The Robotics API's reference implementation uses a custom XML schema for specifying configurations. This will not be described here in detail. Instead, the process of initializing the Robotics API from a configuration file will be explained. Conversely, uninitializing the

Robotics API can be performed via the RoboticsAppLoader's method `unload()`. An important concept for configurable RoboticsObjects is the stereotype *ConfigProp*. This stereotype can be applied to methods and indicates that the method is responsible for accepting configuration items of a certain type. For example, implementations of Device that require a DeviceDriver provide a method `setDriver(d : DeviceDriver)` that is annotated with the *ConfigProp* stereotype. Those methods are expected to accept exactly one argument. *ConfigProp* furthermore provides a boolean attribute `optional` which specifies whether this configuration property is mandatory for using the RoboticsObject, and an attribute `name` of type String. This attribute represents the name of the configured property for configuration management. In the Robotics API's reference implementation, the *ConfigProp* stereotype has been implemented by a Java Annotation.

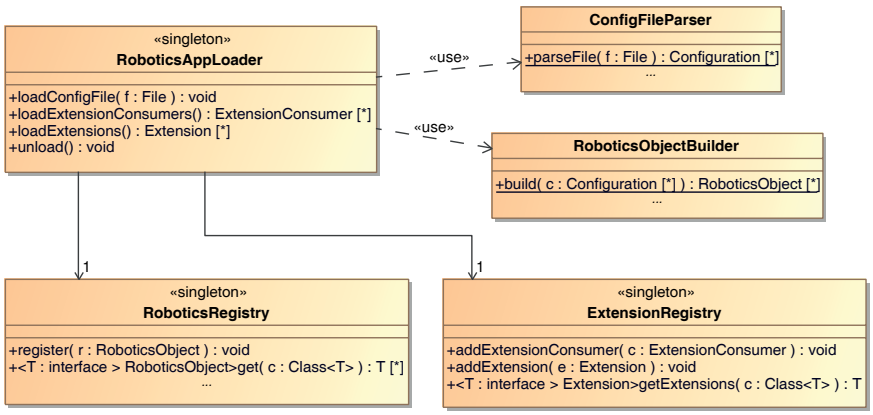

Figure 6.19: Classes that realize a startup mechanism for Robotics API Java applications.

To model persisted configurations of RoboticsObjects, the Robotics API provides the concept *Configuration*. Its characteristics are the following:

- A Configuration specifies a name.
- A Configuration specifies a concrete type whose instances can be configured by this Configuration.
- A Configuration consists of a set of *ConfigurationEntry* instances with these characteristics:
  - Each ConfigurationEntry maps a *name* to a *property*.
  - The property can either be a value or a named reference.

The sequence diagram in Fig. 6.20 shows the simplified flow of interactions when the RoboticsAppLoader receives a `loadConfigFile(...)` message. To properly initialize

the Robotics API, first all ExtensionConsumers are loaded (interaction 2). In the reference implementation, the Java Service Provider Interface is used (cf. [93, 94]) to locate all implementations of the ExtensionConsumer interface in the application's classpath. Each ExtensionConsumer is then registered at the singleton instance of ExtensionRegistry (interaction 3). The same process is performed for all Extensions (interactions 4 and 5). Note that loading the ExtensionConsumers first is the prerequisite for handling all Extensions properly, as explained in Sect. 6.10. Extensions, in turn, might be the prerequisite for correctly interpreting configurations specified in the configuration file and assigning them to appropriate RoboticsObject instances.

The RoboticsAppLoader triggers parsing of the configuration file by the ConfigFileParser (interaction 6), who returns all Configuration objects that could be parsed from the given file. These are then supplied to RoboticsObjectBuilder by the message `build(...)` (interaction 7). The RoboticsObjectBuilder then has to do the following for each Configuration c:

- Create an instance i of the type of object specified in c.
- Assign the name specified in c to i by passing it to i's method `setName(...)`.
- For each ConfigurationEntry e stored in c:
  - Locate those method of i whose *ConfigProp* stereotype matches e, i.e. the names are equal.
  - Assign e's property to i by passing it to the method.
- If all of i's non-optional properties have been set (i.e., all methods have been called that have *ConfigProp* stereotypes whose optional property is false), initialize i by calling its method `initialize()`.

Note that if a ConfigurationEntry's property is a named reference, it is interpreted as the reference to another RoboticsObject which has the name specified in the reference. On the other hand, no order is given among the Configurations. This implies that RoboticsObjectBuilder has to resolve such implicit dependencies between RoboticsObjects and has to detect potential cycles. For creating instances from the types specification in Configurations, RoboticsObjectBuilder may rely on further concepts. For example, the Robotics API reference implementation provides particular kinds of Extensions that are able to interpret type specifications and instantiate appropriate classes. Finally, RoboticsAppLoader registers all RoboticsObject instances that have been configured and initialized appropriately at the RoboticsRegistry. Application developers can use the singleton instance of RoboticsRegistry to access all those RoboticsObjects.

The basic interaction flow shown does not contain error cases (e.g., the object type specified cannot be resolved, no method with a stereotype matching a ConfigurationEntry can be found). The reference implementation handles errors in most cases by discarding

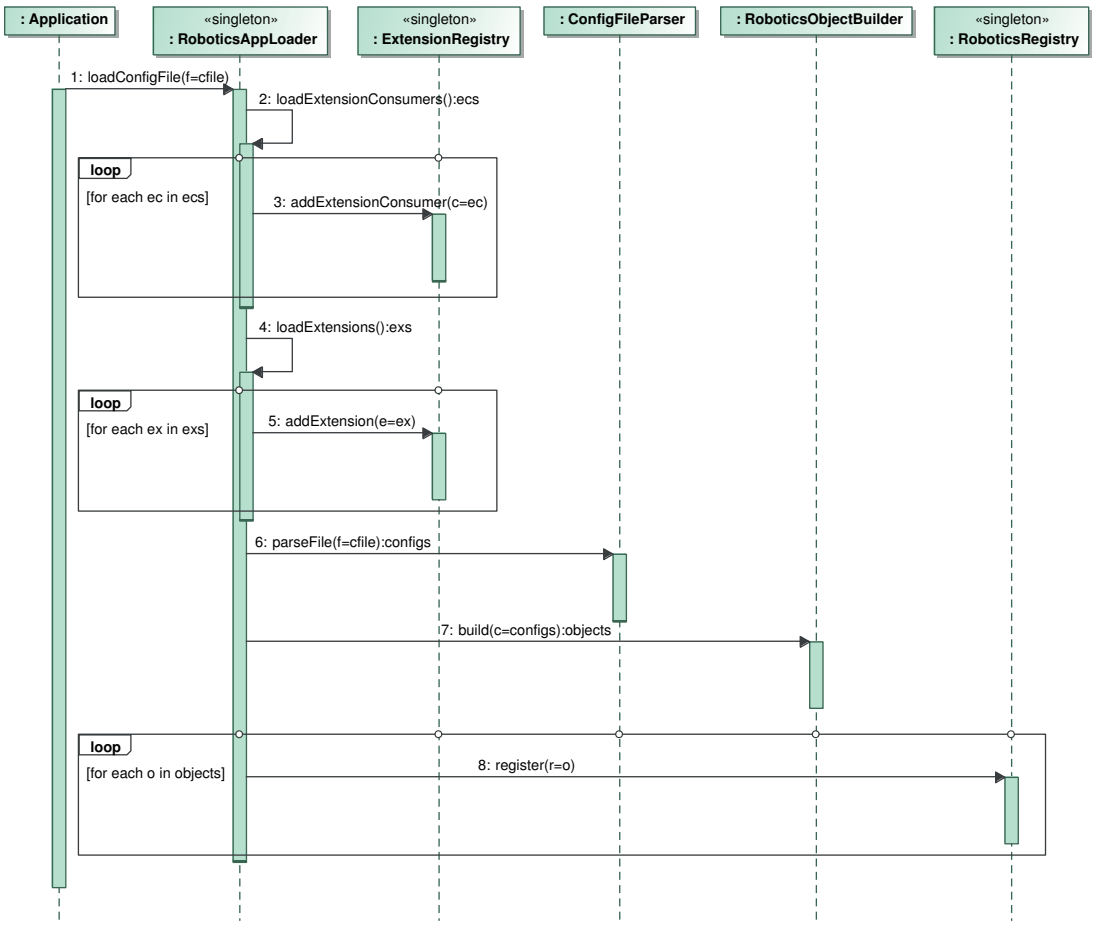

Figure 6.20: The Robotics API core defines a stereotype for configurable properties of RoboticsObjects.

the current, erroneous Configuration and logging the respective error. All subsequent Configuration are still processed.

ConfigFileParser and RoboticsObjectBuilder are state-less concepts that provide static methods for parsing configuration files and creating RoboticsObjects from configurations. In contrast, RoboticsAppLoader, RoboticsRegistry and ExtensionRegistry are modelled as singletons (cf. Gamma et al.'s singleton pattern [86], pp. 127) to provide a globally unique state w.r.t. ExtensionConsumers, Extensions and RoboticsObjects that have been loaded. RoboticsAppLoader prevents application developers from loading a configuration file twice without previous unloading.

The configuration management mechanism presented in this section has two strong points: On the one hand, the procedure for loading and applying configurations to RoboticsObjects is decoupled from the implementation of those RoboticsObjects. Thus,

this procedure can be adapted, e.g. in order to provide a more convenient structure of configuration files or a completely different data format. On the other hand, by mapping the configuration properties to methods (instead of fields) via the *ConfigProp* stereotype, implementations of those methods in concrete RoboticsObjects can inspect the properties and perform specific checks for validity.

In the following chapters, the *ConfigProp* stereotype will be used to indicate the configurable properties of various implementations of RoboticsObjects.

6.12 Related Work

There exist various approaches in the robotics research community that deal with modeling robotic devices, their configurations, the operations they perform, reactivity for handling events and exceptions during those operations, and sensors that provide data required by those operations or by higher-level applications. In the following, some approaches are presented which come close to the way this work solves the abovementioned aspects. The commonalities and differences to this work are discussed.

In the context of Three-Layered Architectures that were introduced previously in this work, approaches for modeling robot task specifications have been developed. One such specification, the so-called Task Trees introduced by Simmons et al. [68], have some commonalities with the Robotics API's Command model. A Task Tree is a tree formed by hierarchical parent/child relationships. Actions in tree nodes can contain user code that may operate actuators and also modify the tree structure itself. Synchronization constraints between nodes of the tree can be used to achieve sequential or parallel execution of actions. Furthermore, exception handlers can be attached to nodes. However, Task Trees do not provide any support for real-time execution. Instead, quality of actuator control is purely a matter of user code. Furthermore, task trees are specified in a custom syntax (a subset of C++) that requires a special compiler. This constrains developers to the language subset supported by the compiler.

Borrely et al. [95] introduced the ORCCAD architecture as a hybrid control system that can express continuous time control laws at its lower levels and event-based discrete actions on a higher-level. Elementary actions of robots are described by the concept ROBOT-TASK, which incorporates the definition of a control law and of various events during its execution. ROBOT-TASKS can be composed into ROBOT-PROCEDURES, which consist of a main program and a set of rules for processing exceptions during execution. For specifying the main program of a ROBOT-PROCEDURE, a custom language called MAESTRO is employed, which has been developed by the same group [96]. ROBOT-TASKS are translated to the Esterel language developed by Berry et al. [97]. Furthermore, from each ROBOT-PROCEDURE, a controller in the Esterel language based on predefined templates is generated as well. Execution of the generated code on the real-time operating system VxWorks is possible by combining it with an additional run-time library. The approach is supported by multiple tools, e.g. for specifying ROBOT-TASKS graphically and for simulating the controller generated from a

ROBOT-PROCEDURE. Furthermore, tools for logical and temporal verification based on theorem proving and real-time model checking methods are provided. The tool support is clearly a strong point of ORCCAD. ROBOT-TASKS are conceptually similar to Robotics API Actions. However, the Robotics API's Command model is more flexible in terms of handling exceptions (ORCCAD only supports three predefined exception types and reaction strategies) and sensor feedback (arbitrary sensors can be used in Robotics API Commands, where ROBOT-PROCEDURES are limited to the events defined by the ROBOT-TASKS). Furthermore, the Robotics API's approach to describe the application workflow with a modern general-purpose language is clearly more powerful and flexible than ORCCAD's ROBOT-PROCEDURES, which binds developers to the proprietary MAESTRO language.

Object-oriented models of robotic devices are used for quite some time in various frameworks. One of the earlier examples is *RIPE* presented by Miller et al. [98]. They distinguish the generic concepts Station, Device and WorkPiece and assume that Stations consist of Devices and WorkPieces and assign geometrical positions to them, and that Devices carry out operations on WorkPieces. Specializations of Device include Tool and Transport, and a Robot is a specialization of Transport. The argumentation is that Robots have the ability to "transport" Tools or WorkPieces and thus are derived from the Transport class. Possible operations are modeled by methods of those generic classes. For example, Robot provides methods to move to a position on a certain path or with force control. Methods for such force-controlled motions are parameterized with an instance of ForceSensor, which is a generic concept derived from Tool. However, to execute a force-controlled motion with real-time guarantees, concrete implementations of the abstract concepts have to be created. They have to be deployed in a real-time capable environment, together with a control program that creates appropriate instances of the concrete device implementations, parameterizes them and executes a workflow based on the provided methods. The Robot and ForceSensor implementations may be distributed to subsystems. In this case, RIPE provides some support for distributed communication. Furthermore, a concept for error handling is introduced which assigns each Device a single ErrorHandler whose implementation has to handle all errors that can occur during operation of that Device. In sum, RIPE does not provide any abstractions for real-time critical, sensor-based operations like provided by the Robotics API's Command and Sensor models. The same is true for handling of errors, which is furthermore not supported per operation, but only globally for each device. Extensibility is also a weakness of RIPE, as all operations are modeled as methods in the generic device classes.

The *Robotic Platform* by Loffler et al. [99] provides an object-oriented model of devices and algorithms. The top-level concept RoboticsObject models a configurable entity that has to indicate its error status and aggregates operating system threads required for its operation. Furthermore, it has to provide components for a modular graphical user interface as well as 'Interactive Commands' that are available in this user interface. Finally, a generic message handler and shutdown logic are incorporated on this level. Generic subclasses are PhysicalObject and FunctionalObject, where the former models all enti-

ties with physical properties like geometric position and relations to other objects. The latter is intended to model algorithms like trajectory generators or servo control loops. Manipulator, which is a generic superclass of e.g. various robots, inherits from both PhysicalObject as well as ControlProgram, which is a concept provided by the QMotor framework [100], which is responsible for real-time control. The default implementation of Manipulator creates a separate thread with high priority that executes the real-time control logic. This introduces some separation between application logic and real-time control. However, errors in the application logic might disturb the control loops and leave devices in an unstable state. Operations provided by PhysicalObjects like Manipulators are modeled by methods of those classes, causing similar problems with extensibility like in RIPE. Configuration of RoboticsObjects in the Robotic Platform is possible via a global configuration file, similar to the Robotics API.

McKee et al. [101] take a different approach with the *MARS* platform. Instead of relying on object oriented concepts integrated in typical programming languages, they introduce a custom model of object-oriented modules whose properties are declared by annotating programming language classes. Classes annotated in this way are called *modules*. Different kinds of relations can be defined among modules. The type of relation implies a certain inheritance of functionality between the modules. For example, if a module has a relationship *is_mounted_on* with another module, it will inherit the motion functionality from the other module, if this module provides such functionality. Further annotations are defined to realize e.g. aggregation of modules and modeling of sensors. When an application has been composed from a set of modules, an analysis stage identifies inheritance conflicts (e.g., motion functionality is inherited from two different parent modules) and a synthesis stage resolves those conflicts by introducing additional special modules according to some pre-defined rules. In sum, the modules and relationships defined by MARS extend general object oriented concepts like inheritance according to requirements of software development for robotics. However, MARS does not provide means of modeling real-time operations of devices, event-based operations and exception handling.

CLARAty by Nesnas et al. [102, 103] contains an object-oriented functional layer whose approach is to "highlight abstract behavior and interface to states of the system while hiding runtime models" [102]. Its object oriented model distinguishes data structure classes, physical classes and functional classes. Physical classes are intended to model devices and can control those by an internal threading model. Whether hard real-time control is possible depends on the runtime architecture chosen. Operations are encapsulated by functional classes, which is less flexible compared to the Robotics API's Command model. Error handling is done by implementations of physical classes.

The Player project [104, 105] originally followed the approach of modeling robotic devices as files, similar to the UNIX model of embedding hardware devices in the file system. Thus, applications accessed devices by *opening* and *closing* them and reading data in form of a character stream. This model was later revised [106] in favor of an easier access to devices via message passing, similar to most component frameworks. The current version of Player provides interesting abstractions for modeling devices that

are in some respect similar to the Robotics API. A central concept is the *interface*, which specifies the allowable messages for a certain type of interaction (e.g. querying sensor data, controlling actuation). Interfaces can be bound to *drivers*, which translate the interface's protocol for a certain type of concrete entity (e.g., a concrete sensor or actuator). A *device* is merely the combination of a certain interface and a specific driver, with an additional fully qualified address. Thus, implementations of interfaces for various entities are hidden by the respective drivers. Player interfaces are conceptually similar to the Robotics API's ActuatorInterfaces. However, ActuatorInterfaces usually provide models of complex, executable operations, while Player interfaces merely define messages with a certain syntax and semantics. Player as such does not provide an extensive model of (hard real-time) operations like the Robotics API. Finally, Devices in the Robotics API are direct representations of physical entities and their properties, while Player devices are merely abstract, addressable entities.

OROCOS is a hard real-time capable component-based framework. In [107], Bruyninckx et al. propose a generic control core that is able to interpret commands and configure and start a control loop appropriately. The idea of a generic command model and generic control loops is similar to the Robotics API's Command model and its mapping to the Robot Control Core. However, the control loop as proposed by Bruyninckx et al. is formed rather statically, compared to the dynamic generation of RPI primitive nets as realized with the SoftRobotRCC, which is more flexible considering command composition. Furthermore, the exact nature of the commands that applications should submit to the OROCOS command interpreter remains unclear. In more recent work [108, 109], the group developed an approach for constraint-based specification of robot tasks. Such a way of specifying tasks could be integrated in the Robotics API in form of special kinds of Actions.

The general goal behind the design of the Robotics API's Command model is to provide a powerful model of real-time critical operations that developers can use to express real-time critical operation sequences. Developers are thus able to encapsulate all real-time critical operations in Commands that are executed atomically. This distinguishes the Robotics API from all approaches that consider real-time support as matter of the runtime platform like the ones presented above. In a way, the Robotics API's Command model can be considered a (simple) Domain Specific Language. Its simplicity compared to a full language is chosen deliberately to allow deterministic execution. Approaches that use full programming languages and provide execution environments that try to ensure hard real-time properties are clearly more expressive, but suffer from the potential indeterminism of user code, as exemplified in work by Klotzbücher et al. [110].

Chapter 7

The Robotics API World Model

Summary: To manipulate the physical world with robots, applications require mechanisms to describe important aspects of the world. In the context of the Robotics API, this description mechanism is termed *world model* and is realized by an object-oriented structure based on an established mathematical formalism, which are both described in this chapter.

7.1 Mathematical Background: Position and Orientation Representation

This chapter will introduce the mathematical formalism that is employed in the Robotics API to represent position and orientation of points in space. This work adopts an established formalism from the Springer Handbook of Robotics ([40], pp. 10). The following paragraphs are therefore largely based on the abovementioned work, which is advised for further reading.

The Robotics API uses *coordinate reference frames* to describe any point in 3-dimensional space. In the following, those are just referenced as *frames*. A frame is described using an origin point O_i and an orthonormal set of three vectors that span an Euclidean space. Those basis vectors are denoted $\hat{\mathbf{x}}_i$, $\hat{\mathbf{y}}_i$ and $\hat{\mathbf{z}}_i$. When any *displacement* relative to a frame is described, this is always done relative to its origin and with respect to its basis vectors.

The displacement of frame j relative to frame i is described by a *translation* and a *rotation* component. The translation component describes the position of O_j relative to O_i, expressed in i's basis vectors. This position is denoted as ${}^i\mathbf{p}_j$ and is defined as a 3x1

vector:

$$ {}^{i}\mathbf{p}_j = \begin{pmatrix} {}^{i}p_j^x \\ {}^{i}p_j^y \\ {}^{i}p_j^z \end{pmatrix} $$

For representing the rotation component, rotation matrices are employed. A rotation matrix ${}^{i}\mathbf{R}_j$ expresses the orientation of the basis vectors $\hat{\mathbf{x}}_j$, $\hat{\mathbf{y}}_j$ and $\hat{\mathbf{z}}_j$ with respect to the basis vectors $\hat{\mathbf{x}}_i$, $\hat{\mathbf{y}}_i$ and $\hat{\mathbf{z}}_i$. A strength of this representation of orientation is the possibility to combine rotation matrices by multiplication. Thus, if ${}^{i}\mathbf{R}_j$ and ${}^{j}\mathbf{R}_k$ are known, ${}^{i}\mathbf{R}_k$ can be constructed like this:

$$ {}^{i}\mathbf{R}_k = {}^{i}\mathbf{R}_j {}^{j}\mathbf{R}_k $$

Note that applying a Rotation ${}^{i}\mathbf{R}_k$ to a vector v expressed in Frame k transforms the vector to its representation in Frame i (not the other way round).

A second strong point of rotation matrices is the possibility to invert them by transposing them (see [40], p. 11). Thus, ${}^{j}\mathbf{R}_i$ can be calculated from ${}^{i}\mathbf{R}_j$ in the following way:

$$ {}^{j}\mathbf{R}_i = {}^{i}\mathbf{R}_j{}^{\top} $$

Rotation matrices are impractical to specify explicitly, as it is difficult to imagine rotations in terms of orthonormal unit vectors. However, there exist various other conventions for specifying rotations that all have rotation matrix equivalents. In the Robotics API, the so called *Z-Y-X Euler angles* ([40], p. 12) are commonly used for specifying rotations. According to this convention, the rotation between frame i and j is expressed by three successive rotations around coordinate axes. The first rotation is performed around the coordinate axis defined by $\hat{\mathbf{z}}_i$. The second rotation is performed around the already rotated axis $\hat{\mathbf{y}}_i$, denoted by $\dot{\hat{\mathbf{y}}}_i$. The third rotation is performed around the twice-rotated axis $\hat{\mathbf{x}}_i$, denoted by $\ddot{\hat{\mathbf{x}}}_i$. The first, second and third rotations are often identified (in this order) by α, β and γ. Another convention used by KUKA and also adopted in the Robotics API names the rotations A, B and C. The equivalent rotation matrix for applying Z-Y-X rotations A, B and C to frame i can be constructed as follows (see [40], p. 11, Table 1.1):

$$ {}^{i}\mathbf{R}_j = \begin{pmatrix} c_A c_B & c_A s_B s_C - s_A c_C & c_A s_B c_C + s_A s_C \\ s_A c_B & s_A s_B s_C + c_A c_C & s_A s_B c_C - c_A s_C \\ -s_B & c_B s_C & c_B c_C \end{pmatrix} $$

where $c_\theta := \cos\theta$ and $s_\theta := \sin\theta$. Conversely, there also exist ways to derive various other representations from a rotation matrix (see [40], p. 12, Table 1.2).

The so-called *homogeneous transformations* ([40], pp. 13) are a way to combine translations and rotations in a single mathematical construct, named *homogeneous transformation matrix*. Given frames i and j, a 4x4 matrix ${}^{i}\mathbf{T}_j$ can be constructed as follows:

$$ {}^{i}\mathbf{T}_j = \begin{pmatrix} {}^{i}\mathbf{R}_j & {}^{i}\mathbf{p}_j \\ \mathbf{0}^{\top} & 1 \end{pmatrix} $$

For any vector ${}^{j}\mathbf{r}$ expressed in frame j, its equivalent ${}^{i}\mathbf{r}$ in frame i can be calculated using ${}^{i}\mathbf{T}_{j}$:

$$\begin{pmatrix} {}^{i}\mathbf{r} \\ 1 \end{pmatrix} = {}^{i}\mathbf{T}_{j} \begin{pmatrix} {}^{j}\mathbf{r} \\ 1 \end{pmatrix}$$

Thus, 3-dimensional vectors as well as 3x3 rotation matrices have to be extended by an 'artificial' fourth dimension to be able to perform homogeneous transformations. While this is not computationally efficient due to additional calculations required in matrix and vector multiplications, it allows for an easy-to-use representation of displacements.

Homogeneous transformation matrices can again be combined by simple multiplication to obtain combined displacements:

$${}^{i}\mathbf{T}_{k} = {}^{i}\mathbf{T}_{j}\,{}^{j}\mathbf{T}_{k}$$

In contrast to rotation matrices, the inverse cannot simply be obtained by transposition. Instead, translation and rotation components of a homogeneous transformation matrix have to be considered independently (see [40], p. 14):

$${}^{i}\mathbf{T}_{j}{}^{-1} = {}^{j}\mathbf{T}_{i} = \begin{pmatrix} {}^{i}\mathbf{R}_{j}{}^{\top} & -{}^{i}\mathbf{R}_{j}{}^{\top}{}^{i}\mathbf{p}_{j} \\ \mathbf{0}^{\top} & 1 \end{pmatrix}$$

The design of the Robotics API's world model is based on describing displacements between frames as translations of their origin and rotation of the basis vectors, as described above. The design is independent of a concrete mathematical representation of displacements, though the abovementioned Z-Y-X Euler convention for representing rotations is used frequently.

Besides representing displacements, a formalism for representing velocities is also desirable. In this work, relative velocities of coordinate reference frames are described by four components:

- *Moving frame* μ: The coordinate reference frame whose velocity is described.
- *Reference frame* ρ: The coordinate reference frame from which μ's movement is observed.
- *Orientation* o: A set of three orthogonal basis vectors in which the velocity is described.
- *Pivot point* π: A 3D point relative to which angular velocities are described (the 'center of rotation').

Together, those four components form a quadruple $\Theta[\mu, \rho, o, \pi]$, called *twist*. A twist's values are encoded by a pair of 3x1 vectors. The first vector specifies the linear velocity $\mathbf{v}$, where the vector can be interpreted as the direction of the velocity, and the vector's length as the magnitude of the linear velocity. The second vector specifies the angular

velocity ω. In this case, the vector can be interpreted as the rotation axis, and the length of the vector as the magnitude of the angular velocity. Both velocities are expressed w.r.t. the basis vectors o and the Pivot point π. In the following, a twist's values are denoted by a tuple $(\mathbf{v}, \omega)$.

With this definition of velocity, it is possible to define operations for inverting it, changing its orientation component or pivot point, and combining two velocities. The operations are based on work by Jain [111]. An inverted twist Θ is denoted Θ^{-1} and is calculated in the following way:

$$\Theta[\mu, \rho, o, \pi]^{-1} = \Theta[\rho, \mu, o, \pi] = (-\mathbf{v}, -\omega)$$

where $(\mathbf{v}, \omega)$ is the value of $\Theta[\mu, \rho, o, \pi]$.

Given a twist $\Theta[\mu, \rho, o_i, \pi_m]$, the orientation can be changed from o_i to o_j and the pivot point from π_m to π_n. To change orientation, a Rotation matrix ${}^i\mathbf{R}_j$ is needed that expresses the rotation of the basis vectors o_j with respect to the basis vectors o_i. Then, a twist with changed orientation can be calculated in the following way:

$$\Theta[\mu, \rho, o_j, \pi] = ({}^i\mathbf{R}_j{}^{-1}\mathbf{v}_i, {}^i\mathbf{R}_j{}^{-1}\omega_i)$$

where $(\mathbf{v}_i, \omega_i)$ is the value of $\Theta[\mu, \rho, o_i, \pi]$.

In a similar way, the pivot point can be changed. In this case, the vector $\mathbf{d}$ expressing the displacement of π_n relative to π_m in basis o is required. A twist with an alternative pivot point can be calculated like this:

$$\Theta[\mu, \rho, o, \pi_n] = (\mathbf{v}_m + \omega_m \times \mathbf{d}, \omega_m)$$

where $(\mathbf{v}_m, \omega_m)$ is the value of $\Theta[\mu, \rho, o, \pi_m]$. Note that the angular velocity component is not affected by changes of the Pivot point.

Two twists Θ_i and Θ_j can be combined by an addition operation, provided the following holds: Both twists must be expressed relative to the same orientations and pivot points. Furthermore, Θ_i's moving frame μ_i must match Θ_j's reference frame ρ_j. In this case, both twists can be added by simply adding the two velocity components:

$$\Theta[\mu_i, \rho, o, \pi] + \Theta[\rho, \rho_j, o, \pi] = \Theta[\mu_i, \rho_j, o, \pi] = (\mathbf{v}_i + \mathbf{v}_j, \omega_i + \omega_j)$$

where $(\mathbf{v}_i, \omega_i)$ is the value of $\Theta[\mu_i, \rho, o, \pi]$ and $(\mathbf{v}_j, \omega_j)$ is the value of $\Theta[\rho, \rho_j, o, \pi]$.

The Robotics API's world model does not only provide means to specify and operate on displacements, but also to perform calculations on the velocities in the model of coordinate frames. The concepts forming this model and their mathematical semantics are presented in the next section.

7.2 An Object-Oriented Model of Frames and their Relationships

To express single geometric relationships, the Robotics API's world model integrates classes that represent mathematical concepts and can perform mathematical operations accordingly. Those basic mathematical concepts are termed *Vector*, *Rotation* and *Transformation*, as depicted in Fig. 7.1. The same figure also contains the classes *Frame*, *WorldOrigin* and *Relation*. Those classes form a simple model for relating sets of coordinate frames. This section will define each concept and its operations.

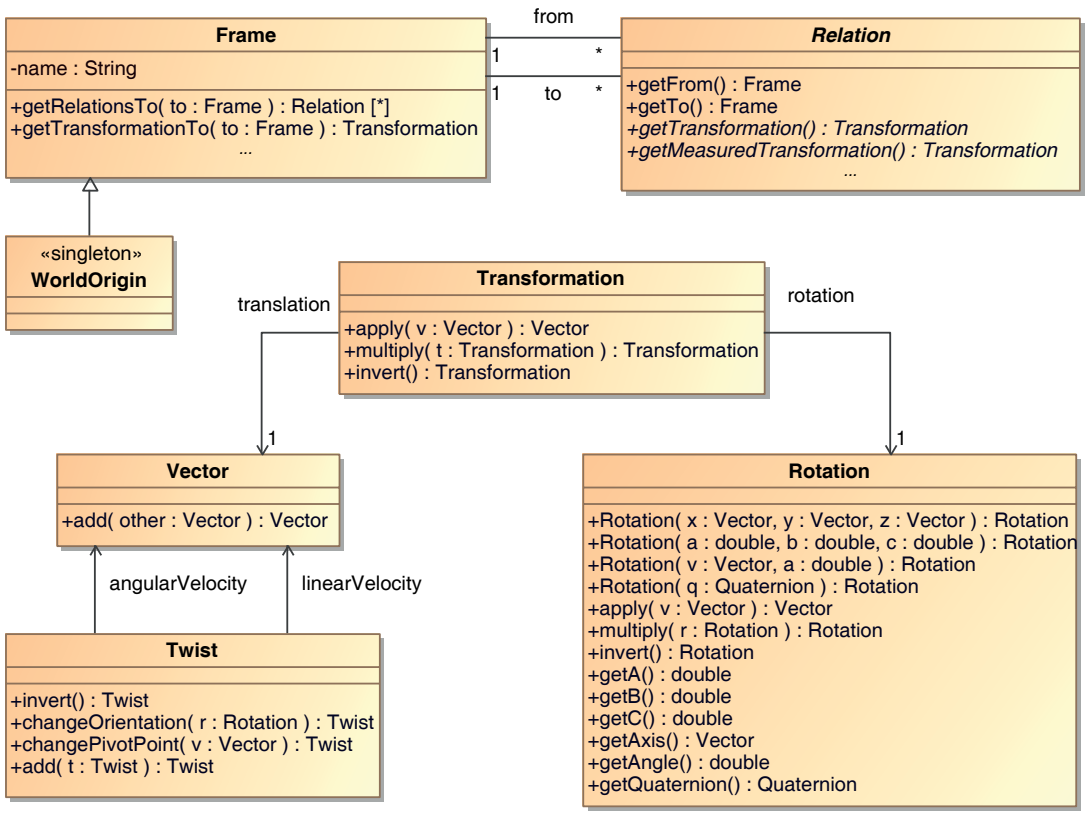

Figure 7.1: Basic classes forming the Robotics API model of Cartesian coordinate frames.

Modeling mathematical concepts

Definition 7.1 (Vector)**.** A Robotics API *Vector* is the representation of a 3x1 vector **v**.

The Vector class provides the method `add(Vector)`, which, when called on a Vector instance v with argument w of type Vector creates a new Vector x that equals the

mathematical addition of v and w: $\mathbf{x} = \mathbf{v} + \mathbf{w}$.

Definition 7.2 (Rotation)**.** *Rotation* is the representation of a 3-dimensional rotation.

The Rotation class is intended to support all kinds of representations of rotation. While the reference implementation stores rotation internally as a 3x3 rotation matrix, other implementations are feasible. However, every implementation has to support conversions to other representations as well as some mathematical operations. The design includes conversions to the Z-Y-X Euler angle representation, the Axis-Angle representation ([40], p. 12) and the Quaternion representation ([40], p. 13). Each Rotation instance can be constructed from those representations (through the respective constructor) and converted to these representations (through the methods `getA()/getB()/getC()`, `getAxis()/getAngle()` and `getQuaternion()`).

In the following, the mathematical operations which have to be implemented by Rotation are explained. It is assumed that all operations are called on a Rotation instance R and that a semantically equal rotation matrix R can be derived from R.

- `apply(v:Vector)` constructs a new Vector r which corresponds to the Vector v rotated by the Rotation R the operation is called on: $\mathbf{r} = R\mathbf{v}$.
- `multiply(S:Rotation)` constructs a new Rotation C which equals rotating a vector first by Rotation S and then by Rotation R: $C = RS$.
- `invert()` constructs a new Rotation I which corresponds to the inverse of R: $I = R^{-1}$.

Based on the concepts Vector and Rotation, the Robotics API's Transformation class is characterized as follows:

Definition 7.3 (Transformation)**.** A geometric displacement is fully described by an instance of *Transformation*. The translational part is defined by an associated Vector, whereas the rotational part is defined by an associated Rotation.

Like Vector and Rotation, a Transformation also supports elementary mathematical operations. In the following, it is assumed that each Transformation T (with Rotation R and Vector v) has an equivalent homogeneous transformation matrix T which can be derived from it.

- `apply(w:Vector)` constructs a new Vector r which corresponds to the Vector w rotated by the Rotation R and translated by Vector v: $\mathbf{r} = R\mathbf{w} + \mathbf{v}$. Note that the rotation is only applied to w and not to v.
- `multiply(U:Transformation)` constructs a new Transformation V which equals displacing a vector first by Transformation U and then by Transformation T: $V = TU$.

- `invert()` constructs a new Transformation I which corresponds to the inverse of T: $I = T^{-1}$

Definition 7.4 (Twist)**.** A Robotics API *Twist* models a mathematical twist. The Vector associated by *linearVelocity* represents the linear velocity. The Vector associated by *angularVelocity* represents the angular velocity.

Elementary operations on Twists are provided by several methods (assume T to be a Twist instance which represents twist Θ_T):

- `invert()` constructs a new inverted twist Θ_{inv}: $\Theta_{inv} = \Theta_T^{-1}$.
- `changeOrientation(R:Rotation)` constructs a new twist Θ_R which is calculated from Θ_T by changing the orientation to o_R, i.e. rotating the orientation vectors by R: $\Theta_R = \Theta_T[\mu, \rho, o_R, \pi]$.
- `changePivotPoint(d:Vector)` constructs a new twist Θ_d which is calculated from Θ_T by changing the pivot point to π_d, i.e. displacing the point by d: $\Theta_d = \Theta_T[\mu, \rho, o, \pi_d]$.
- `add(U:Twist)` constructs a new combined twist Θ_c which is calculated by adding Θ_T and Θ_U: $\Theta_c = \Theta_T[\mu_T, \rho_T, o, \pi] + \Theta_U[\rho_T, \rho_U, o, \pi]$

As stated before, Vector, Rotation, Transformation and Twist are merely implementations of mathematical concepts for expressing displacements and velocities between Frames. The classes *Frame* and *Relation* (see Fig. 7.1) build on these concepts to actually work with arbitrary coordinate frames and describe the relationship between them in a consistent way. Twists are used in particular by VelocitySensors, which will be described in Sect. 7.4.

Definition 7.5 (Frame)**.** A Robotics API *Frame* is a named, oriented point in Cartesian space.

Definition 7.6 (WorldOrigin)**.** The singleton class *WorldOrigin* is a globally unique Frame.

As the Robotics API's Frame concept does not incorporate an origin and a set of basis vectors, it is not directly equivalent to the mathematical concept of a coordinate reference frame. To construct a coordinate reference frame, a Frame has to be augmented by at least one Relation.

Modeling geometric relationships

Definition 7.7 (Relation)**.** A *Relation* defines a geometric connection between exactly two Frames. A Relation has the ability to determine the current displacement between the two Frames.

Relations can be thought of as the *undirected* edges in a multigraph with all existing Frame instances as nodes. Note that though each Relation is associated to a *from* and *to* Frame, the Relation is always navigatable (and meaningful) in both directions. However, the distinction between the separate ends of a Relation plays a role for the semantics of its geometric operations:

- `getTransformation()` determines the commanded displacement of the Relation's *to* Frame relative to its *from* Frame at the current point in time. *Commanded* denotes the ideal, desired displacement, compared to the actual or *measured* displacement as described below.
- `getMeasuredTransformation()` determines the *measured* displacement of the Relation's *to* Frame relative to its *from* Frame at the current point in time. *Measured* denotes the actual, physical displacement as measured e.g. by some sensor.

The behavior of commanded and measured displacement depends on the type of Relation. There exist static types of Relations that are used e.g. to describe motion goals in a robot's workspace. For these kinds of Relations, commanded and measured displacement are usually equal. In contrast, dynamic Relations that e.g. describe the relative position of Frames located on separate joints of a robot may behave differently. For example, during execution of a motion, the commanded displacement is usually specified by the Action controlling the motion. The measured displacement may differ at some points in time due to the inertia of the robot arm and its controllers. This example also illustrates the limits inherent to both methods that determine Transformations: In dynamic environments, Transformations can be invalid shortly after they have been determined. Thus, the Robotics API's world model also provides geometric Sensors that can provide up-to-date values continuously. Those are introduced in Sect. 7.4. As mentioned before, special types of Sensors exist for measuring relative velocities between Frames. In this case, methods to determine the current velocity between the Frames of a Relation were deliberately not specified, as providing snapshot values of velocities seemed to be not worthwile.

Based on the core concepts contained in Fig. 7.1, some additional concepts have been derived and are provided by the Robotics API. Those are called *Direction*, *Point* and *Orientation*. Their relationship to the basic concepts is illustrated in Fig. 7.2.

Definition 7.8 (Point)**.** *Point* represents a non-oriented point in Cartesian space. It is constructed from a Vector and a Frame, where the Vector is interpreted relative to the Frame.

Definition 7.9 (Orientation)**.** *Orientation* is a rotation basis in Cartesian space. It is constructed from a Rotation and a Frame, where the rotation is interpreted relative to the Frame.

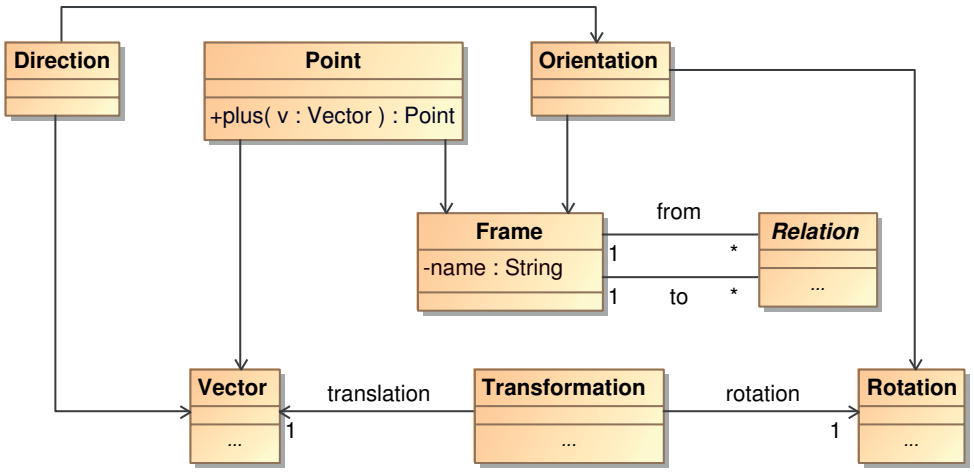

Figure 7.2: Further geometric concepts derived from the basic frame model.

Point and Orientation can be thought of as two 'halves' of a Frame, with Point being the specification of an origin and Orientation being the specification of a rotation basis. It would have been possible to ground the Frame definition on Point and Origin (thus reversing the direction of the associations from Point to Frame and from Orientation to Frame in Fig. 7.2). However, the decision to make Frames 'first level citizens' and Point and Orientation only derived concepts was driven by the insight that complete 6-dimensional cartesian coordinate frames are the most commonly used concepts in robotics applications.

Definition 7.10 (Direction)**.** *Direction* defines a direction axis in space. It is constructed from a Vector and an Orientation, where the vector is interpreted relative to the Orientation.

As mentioned before, the Robotics API provides different types of Relations as depicted in Fig. 7.3. The two basic types are *Connection* and *Placement*. The two types have distinct semantical meanings.

Definition 7.11 (Connection)**.** A *Connection* is a fixed relationship between two Frames that will persist during the runtime of a robotics application.

Connections should be used in all cases where the relationship is induced by some physical context, e.g. when an object is mechanically fixed to another object, or in any other case where the relationship will outlast the application's runtime. A Connection may, however, change over time in the sense that the respective Transformation changes.

Definition 7.12 (Placement)**.** A *Placement* is a loose relationship between two Frames. It can be changed or removed by explicit or implicit operations during the runtime of a robotics application.

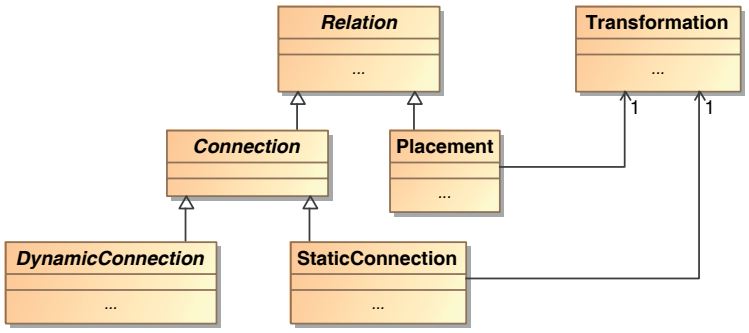

Figure 7.3: Different types of Relations defined by the Robotics API world model.

In contrast to Connections, Placements may be deleted during runtime of an application. Thus, Placements should be used in cases where a relationship between Frames is merely required for some task that is part of the application, e.g. to describe the location of a workpiece relative to some reference system. This Placement will become invalid and has to be deleted as soon as the workpiece is fetched from this location. Deleting can either be done explicitly by a statement in the application's logic, or implicitely by a device operation, e.g. an implementation of a grasp operation of a robot gripper that also manipulates the Frame model of the grasped object accordingly. A Placement can furthermore be assigned a new Transformation anytime to reflect changes in the environment.

For Connections, the Robotics API distinguishes between two more concrete types: *DynamicConnection* and *StaticConnection*.

Definition 7.13 (StaticConnection)**.** A *StaticConnection* is a Connection between two Frames whose Transformation remains constant.

Definition 7.14 (DynamicConnection)**.** A *DynamicConnection* is a Connection between two Frames whose Transformation may change over time.

StaticConnections are intended to model fixed, constant relationships, e.g. between the base of a robot arm and the physical entity (floor, table, ...) the base is mounted to. DynamicConnections can be used to model fixed, but dynamic relationships, e.g. between adjacent links of a robot arm, which are connected by movable joints.

Due to their static nature, Placement as well as StaticConnection are associated to a Transformation which resembles the Relations' displacement. This Transformation may be returned by the methods `getCommandedTransformation()` as well as `getMeasuredTransformation()`, as the semantics of those Relations induces that both kinds of displacements have to be equal.

In contrast, concrete implementations of the abstract class DynamicConnection have to determine the current commanded and measured Transformations according to the physical relationships. For example, DynamicConnections modeling robot joints may rely on

encoders integrated in those joints for determining measured Transformations. Commanded Transformations for those Relations may e.g. be retrieved from a DeviceDriver that communicates directly with the RCC.

The Frame class provides methods for adding and removing Relations. Implementations of this methods have to update a Relation's *from* and *to* associations accordingly. For conveniently creating an auxiliary Frames a that is located with a certain displacement T (a Transformation) relative to an existing Frame r, Frame offers a method `plus(T)`, which creates a, creates a Placement p with Transformation T and connects r and a with this Placement. r has the role of T's *from* Frame and a the role of its *to* Frame. As such auxiliary Frames are frequently needed in many applications, this method was included for convenience.

A second frequently usable convenience method provided by Frame is `snapshot(s)`. When called on a Frame instance a, a new Frame b is constructed that geometrically coincides with a at the current time instance, but is connected to Frame s with a StaticConnection. This implies that, if a is moved relative to s, b and a will no longer coincide, but b will keep its pose relative to s. Intuitively, a 'snapshot' of the position of a at the current time instant is created. The method's implementation determines the current relative transformation between a and s and uses this to parameterize the newly created StaticConnection.

Example: Geometric model of an LWR

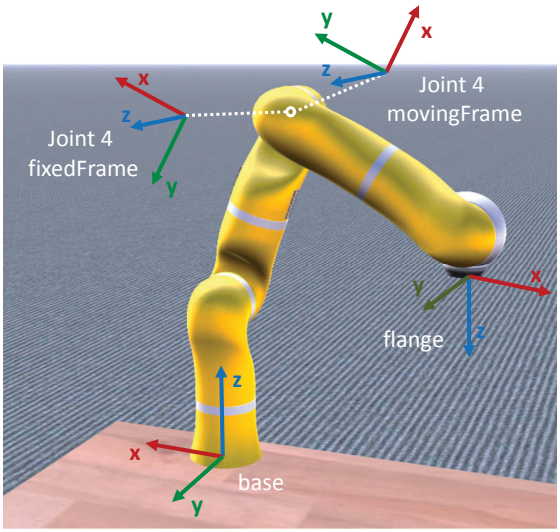

Figure 7.4: Base, flange and a pair of joint frames of a Lightweight Robot.

With the concepts introduced in the previous parts of this section, the complete geometric shape of devices like robot arms can be modeled. For example, each joint of a robot arm can be modeled by two Frames, one representing the fixed part of the joint and one

its moving part. A *RobotArm* device may be composed of several such joints, and may additionally have associated *base* and *flange* Frames. An intuitive visualization of some of an LWR's Frames is given by Fig. 7.4. Frames are visualized by three Vectors that originate from the respective Frame's origin and represent this Frame's basis vectors. Like in the figure, the LWR's base Frame is located at the position where the robot is mounted to a table. The flange is located at the opposite end of the robot structure, where the LWR is equipped with a mounting mechanism for robot tools. Furthermore, the figure shows the two Frames of the robot's 4th joint. Both Frames' origins are located at the same point inside the robot structure, marked with small circle. The basis vectors are depicted separately for clarity. The *fixedFrame* is attached to the link preceding the joint, whereas the *movingFrame* is attached to the link following the joint. When the 4th joint is rotated, the movingFrame rotates relative to the fixedFrame around their common z axis.

The complete Robotics API representation of a LWR is illustrated by the instance diagram in Fig. 7.5. It depicts the LWR, its joints and their associated Frames and Relations. The LWR's base Frame is connected to the fixedFrame of the first joint via a StaticConnection, as is the movingFrame of the last joint to the LWR's flange Frame. The fixedFrame and movingFrame of each joint are connected via a *JointConnection*, which is a concrete implementation of DynamicConnection. Each JointConnection is associated to the respective LWRJoint, which is an implementation of a Robotics API Actuator. The JointConnection employs a Sensor measuring the joint's current position to calculate the current Transformation between fixedFrame and movingFrame. The movingFrame of each joint is connected to the fixedFrame of the subsequent joint with a StaticConnection again, which in this case models the displacement induced by the respective link between the joints.

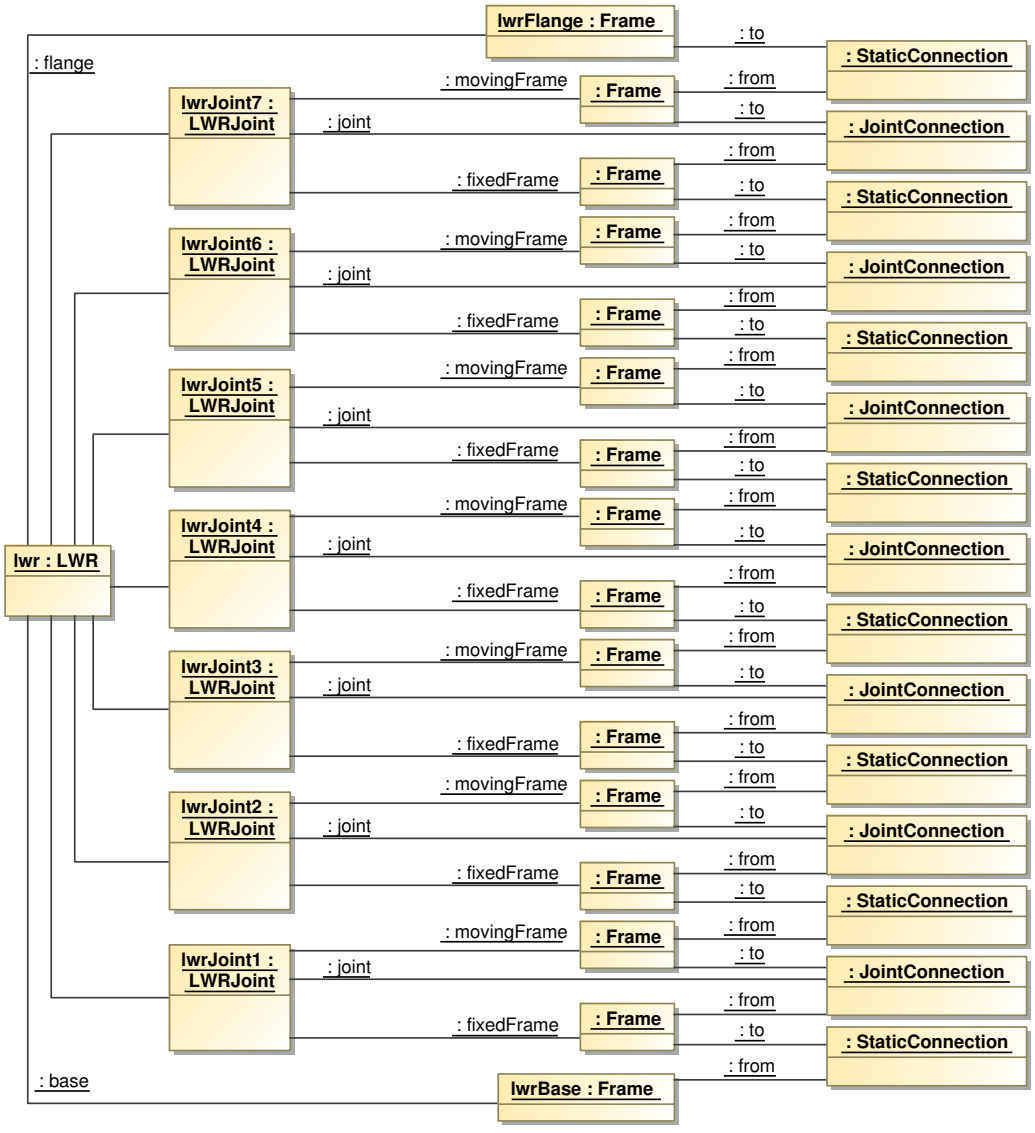

Figure 7.5: Geometric model of a Lightweight Robot in the Robotics API.

7.3 Determining Frame Transformations

In robotics applications, it can be useful to define complex graphs of Frames and Relations to describe robots and parts of their environment that are relevant to the application. Consider a workcell in which a robot loads workpieces from a conveyor into some machine for further processing. The fetching task from the conveyor as well as the loading task into the machine typically consist of multiple steps. In a minimalistic version of the fetching task, usually a pre-fetching position is approached with a rapid (PTP) motion. Then the workpiece is approached with a Cartesian path motion (e.g., a linear motion) to avoid collisions. After the workpiece has been grasped, often a path motion is used to leave the fetching area before the workpiece is transported to the loading area of the machine. Each of the aforementioned steps usually requires one motion goal Frame, which means a total of three Frames. Analogously, a simple loading task will require three Frames as well. To be approachable for the robot arm, the Frames could simply be connected to the arm's base frame, as the inverse kinematics function of the robot can be used to calculate joint configurations for displacements relative to this base frame. However, this approach has two major disadvantages:

- Determining the displacements of the Frames w.r.t. the robot's base Frame is not intuitive for developers and hard to do. Even if the Frames are specified using teaching methods with the robot arm, any changes to the definition (e.g. to compensate for small inaccuracies of the taught displacements) have to be specified relative to the robot's base Frame and not in a Frame more relevant to the actual task.
- The defined Frames are not robust against repositioning of the robot, the conveyor or the machine. When any of those devices is repositioned, multiple Frames have to be adapted to fit the new situation.

To mitigate the first issue, base Frames for the conveyor and the machine can be introduced. Those can be located at characteristic points of the devices (e.g. edges of conveyor fetching area and machine loading area) such that the three Frames required for each of the abovementioned tasks can be easily specified relative to the base Frames. When those base Frames are connected via Relations to the robot's base Frame, a certain robustness w.r.t. repositioning of the devices is introduced. In such cases, only the displacement between the robot's base and the repositioned device's base Frame has to be adapted, not each of the task Frames. To increase robustness, yet another Frame can be introduced that serves as a base Frame for relating the base Frames of both devices and the robot itself. Such a Frame is often called *cell origin* or, more general, *world origin*. In this case, repositioning the robot requires merely adapting the displacement between its base and the world origin Frame.

This example demonstrates the need for complex frame graphs in robotics applications. As illustrated in the previous section, the Robotics API supports building such graphs using Frames and Relations. However, there is also a need to query the model: When the

robot is instructed to move to a Frame on the conveyor, it must resolve the displacement of this Frame relative to its own base Frame to be able to apply its inverse kinematics function. In this case, this means combining three displacements: conveyor base Frame to goal Frame, world origin Frame to conveyor base Frame and robot base Frame to world origin Frame. The Robotics API provides a convenient mechanism for determining displacements between arbitrary Frames that are part of the same Frame-Relation graph. This mechanism is introduced in this section.

Determining the displacement of a Frame f relative to another Frame g in the Robotics API can be done in two steps: First, a *way*, i.e. a series of Relations, has to be found that connects f with g. Second, the Transformations of all Relations in the series can be determined and combined (i.e., multiplied) to one Transformation. This second step can be achieved by simple matrix multiplication (see Sect. 7.1). The first step can be realized using existing graph search algorithms. In the Robotics API reference implementation, a breadth-first search algorithm (see e.g. [112], pp. 531) has been implemented.

Frames provide the following methods that perform queries in the Frame graph:

- `getRelationsTo(to, forbidden)` returns a series of Relations that form the shortest way from this Frame to the Frame `to` which does not use any of the Relations contained in the set `forbidden`, if such a way exists.
- `getTransformationTo(to, allowDynamic, forbidden)` returns the displacement of Frame `to` relative to this Frame, if a way exists between the Frames which does not use any of the Relations contained in the set `forbidden`. Additionally, if the boolean flag `allowDynamic` is false, the way is constrained to consist of Relations that have static characteristics, i.e. who are not of type `DynamicConnection`.

Mathematically, `f.getRelationsTo(g, ...)` computes a series of Relations ${}^{i}\mathcal{R}_{i+1}$ with associated Transformations ${}^{i}T_{i+1}$ connecting Frame f with Frame g. When `f.getTransformationTo(g, ...)` is called, these Transformations are combined by multiplication:

$$ {}^{f}T_{g} = \prod_{i=1..n} {}^{i}T_{i+1} $$

The semantics of `getTransformationTo` is to calculate the commanded displacement between the respective Frames, which implies calling `getTransformation` to determine the displacement for each of the Relations forming the way between the Frames (see Sect. 7.2). In contrast, the syntactically equal method `getMeasuredTransformationTo` of Frame determines the measured displacement between the Frames. Thus, its implementation has to use `Relation#getMeasuredTransformation` to determine the measured Transformation for each Relation.

The methods presented in this Section are of high practical relevance in rather static scenarios with few or no moving objects. Displacements between static objects remain valid for the runtime of the application and can reliably be used with standard robot motions.

7.4 Geometric Sensors

Relations connecting Frames provide methods for retrieving the current displacement between those Frames, as described in Sect. 7.2. However, the same section already mentioned the limits of such methods: When Relations are dynamic and displacements thus change over time, any static displacement information is practically useless. For this reason, Relations are required to provide Sensors which can deliver up-to-date information about displacements. This section introduces the relevant methods and types of geometric Sensors provided by the Robotics API.

The geometric Sensor model is structured similar to the basic Frame model (cf. Fig. 7.1). The concepts *TransformationSensor*, *VectorSensor* and *RotationSensor* shown in Fig. 7.6 are analogons to Transformation, Vector and Rotation. The concept *RelationSensor* is semantically a counterpart to Relation, but is of type TransformationSensor. In the following, those concepts will be introduced in detail. To capture mathematical relationships correctly, we denote the currently measured value of a Sensor s by $\tilde{s}$.

Definition 7.15 (VectorSensor)**.** A *VectorSensor* is a Sensor that measures 3-dimensional displacements. Measurements are provided in form of Robotics API Vectors.

Like Vector, VectorSensor provides a set of methods that resemble mathematical vector operations. In the following explanations, it is assumed that all methods are called on a VectorSensor instance v.

- `add(w:VectorSensor)` creates a new VectorSensor x that measures the mathematical addition of the values provided by v and w. Consequently, the following relationship between the respective Vectors holds: $\tilde{\mathbf{x}} = \tilde{\mathbf{v}} + \tilde{\mathbf{w}}$.
- `scale(s:DoubleSensor)` creates a new VectorSensor x that measures the result of the vector provided by v multiplied with the scalar provided by s: $\tilde{\mathbf{x}} = \tilde{s} \cdot \tilde{\mathbf{v}}$.
- `getLength()` creates a new DoubleSensor x that measures the scalar length of the vector provided by v: $\tilde{x} = \sqrt{\tilde{\mathbf{v}} \cdot \tilde{\mathbf{v}}}$.
- `getX()`, `getY()` and `getZ()` create new DoubleSensors that measure the scalar value of the x, y and z components of $\tilde{v}$, respectively.

Definition 7.16 (RotationSensor)**.** A *RotationSensor* is a Sensor that measures a 3-dimensional rotation. Measurements are provided in form of Robotics API Rotations.

On a RotationSensor instance R, the following methods can be called:

- `apply(v:VectorSensor)` creates a new VectorSensor x that measures the values provided by v, rotated by the values measured by R: $\tilde{\mathbf{x}} = \tilde{R}\tilde{\mathbf{v}}$.

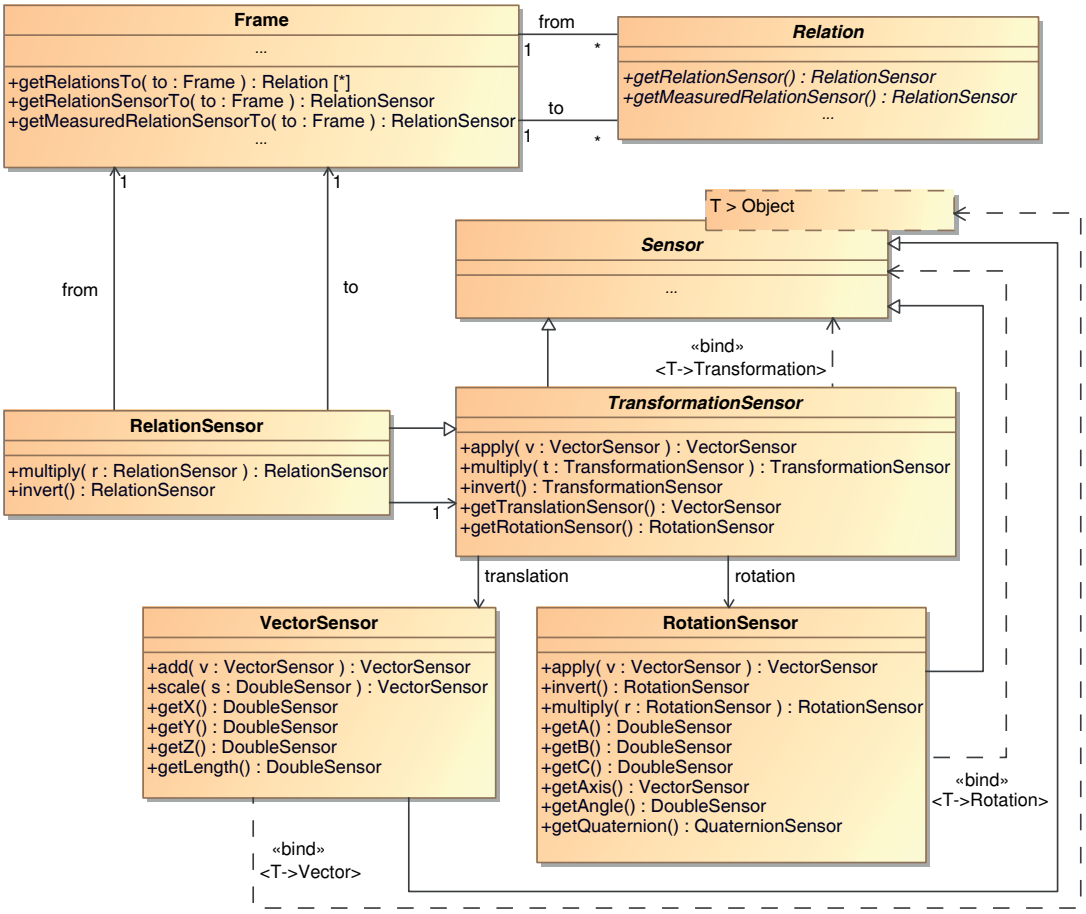

Figure 7.6: Sensors measuring geometric displacements.

- `invert()` creates a new RotationSensor S that measures the inverse of the values provided by R: $\tilde{S} = \tilde{R}^{-1}$.
- `multiply(S:RotationSensor)` creates a new RotationSensor T that measures the multiplication of the Rotations provided by R and S: $\tilde{T} = \tilde{R}\tilde{S}$.
- `getA()`, `getB()` and `getC()` create new DoubleSensors that measure the A, B and C components, respectively, of the Rotation provided by R according to the Z-Y-X Euler angle convention (see Sec. 7.1).
- `getAxis()` and `getAngle()` create a new VectorSensor v and a new DoubleSensor d, respectively, that measure a Vector defining an axis and a scalar value defining a rotation angle around this axis. Thus, $\tilde{v}$ and $\tilde{d}$ define the same rotation like $\tilde{R}$, but as Axis-Angle representation.

- `getQuaternion()` creates a new QuaternionSensor q that measures Quaternions. Those resemble the same rotation like $\tilde{R}$, but in Quaternion representation.

Definition 7.17 (TransformationSensor)**.** A *TransformationSensor* is a Sensor that measures geometric displacements, consisting of a translatory and a rotatory part. Measurements are provided in form of Robotics API Transformations.

Analogously to Transformation, a TransformationSensor T with translatory part v and rotatory part R provides the following mathematical operations:

- `apply(w:VectorSensor)` creates a new VectorSensor r which measures the values provided by w, but displaced by the values provided by T. More precise, $\tilde{\mathbf{r}} = R\tilde{\mathbf{w}}+\tilde{\mathbf{v}}$
- `multiply(U:TransformationSensor)` creates a new TransformationSensor V which measures subsequent displacements by the values provided by U and T, in this order. Mathematically, this operation creates V such that the following equation holds: $\tilde{\mathbf{V}} = \tilde{T}\tilde{U}$.
- `invert()` creates a new TransformationSensor I which measures the inverse values of T: $\tilde{I} = \tilde{T}^{-1}$.

Similar to the basic Sensors in *robotics.core* (cf. Sect. 6.2), the above operations on geometric Sensors create new instances of derived Sensors. For example, the `invert()` method returns an instance of *InvertedTransformationSensor* that is associated to the TransformationSensor the operation was called on. In this way, RoboticsRuntime implementations that provide support for complex, composed geometric Sensors can support those derived Sensors and use the structural information for composing Sensor measurement results.

Definition 7.18 (RelationSensor)**.** A *RelationSensor* is a Sensor that measures geometric displacements between two specific Frames, consisting of a translatory and a rotatory part. Measurements are provided in form of Robotics API Transformations.

RelationSensor is an extended version of TransformationSensor. Besides measuring displacements, it stores information about the pair of Frames whose displacement is measured. Besides this, RelationSensor is semantically equal to TransformationSensor. RelationSensor adds or redefines the following operations (be R an instance):

- `multiply(S:RelationSensor)` creates a new RelationSensor T that measures the displacement of R's *from* Frame to S' *to* Frame under the precondition that R's *to* Frame and S's *from* Frame are identical. Mathematically, T returns the multiplication of the values provided by R and S: $\tilde{T} = \tilde{R}\tilde{S}$.
- `invert()` creates a new RelationSensor S that measures the displacement of R's *from* Frame relative to R's *to* Frame, thus effectively inverting R: $\tilde{S} = \tilde{R}^{-1}$.

Similar to determining the current Transformation between a pair of Frames, a RelationSensor can be constructed that continuously provides up-to-date values of the Transformation between the Frames. Each Frame provides methods `getRelationSensorTo` and `getMeasuredRelationSensorTo` (cf. Fig. 7.6) that determines the way between two Frames and constructs an appropriate RelationSensor, analogously to the algorithm for determining the current Transformation (cf. Sect. 7.3).

Definition 7.19 (VelocitySensor)**.** A *VelocitySensor* measures the relative velocity of a moving Frame w.r.t to a reference Frame, given an orientation (modeled by an associated instance of Orientation) and a pivot point (modeled by an associated instance of Point). Measured values are of type Twist.

Fig. 7.7 shows the relationships between VelocitySensor and other Robotic API concepts. VelocitySensor is a realization of the twist concept introduced in Sect. 7.1, providing operations to create derived and combined VelocitySensors that respect the semantics of all operations on twists. For the following explanation, be V an instance of a VelocitySensor. Furthermore, for each VelocitySensor S, be $\Theta_S[\mu_S, \rho_S, o_S, \pi_S]$ the corresponding twist. The following operations are provided on V:

- `invert()` creates a new VelocitySensor W that measures the inverse values of V:

  $$\tilde{W} = \tilde{V}^{-1}$$

  W's moving Frame is V's reference Frame and vice versa. W's orientation and pivot Point are the same as V's.

- `changeOrientation(o:Orientation)` creates a new VelocitySensor W that measures the values of V when its orientation is changed to o:

  $$\tilde{W} = \Theta_V[\mu_V, \rho_V, o, \pi_V]$$

  W's moving Frame, reference Frame and pivot Point equal those of V, but its orientation is changed to o.

- `changePivotPoint(p:Point)` creates a new VelocitySensor W that measures the values of V when its pivot point is changed to p:

  $$\tilde{W} = \Theta_V[\mu_V, \rho_V, o_V, p]$$

  W's moving Frame, reference Frame and orientation equal those of V, but its pivot Point is changed to p.

- `add(W:VelocitySensor)` creates a new VelocitySensor U that measures the combined velocity of V and W:

  $$\tilde{U} = \tilde{V} + \tilde{W}$$

  The operation is only permitted if W's orientation and pivot Point equal those of V, and V's moving Frame equals W's reference Frame.

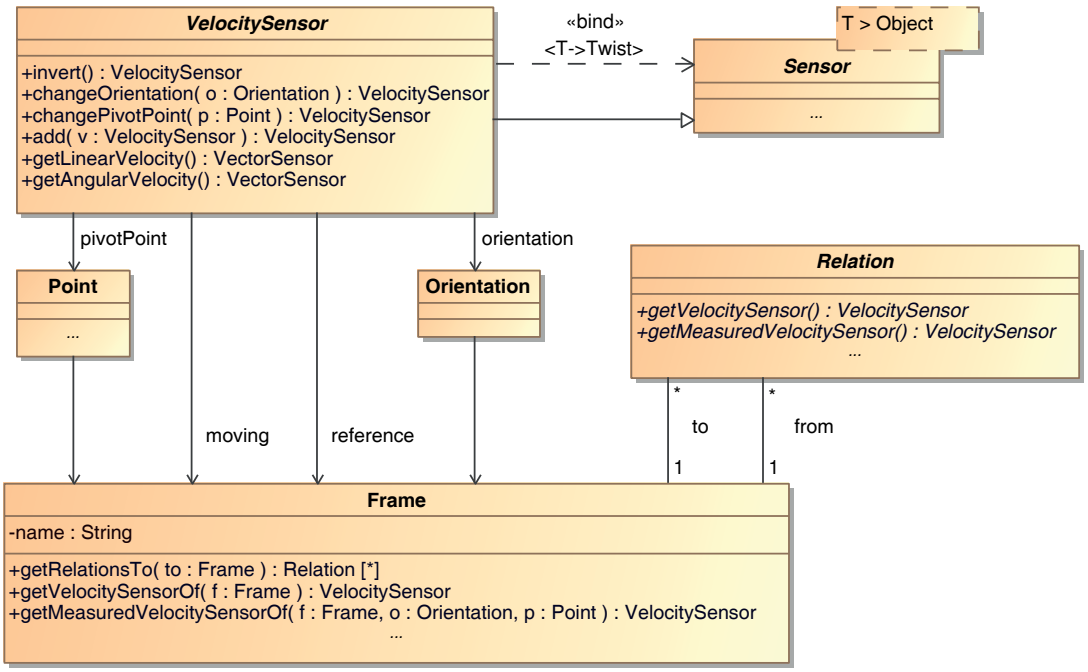

Figure 7.7: Sensors measuring geometric velocities.

Thus, VelocitySensors allow for modeling all operations on twists in a semantically consistent way. Furthermore, VectorSensors can be extracted from a VelocitySensor with the methods `getTranslationVelocity()` and `getRotationVelocity()` that measure the linear and angular velocity components, respectively. Conversely, there exists a derived VelocitySensor called *VelocityFromComponentsSensor*, which allows for constructing a VelocitySensor from two VectorSensors.

As shown in Fig. 7.7, each Relation has to provide VelocitySensors through the methods `getVelocitySensor()` and `getMeasuredVelocitySensor()`. Those Sensors are expected to measure the relative velocity (commanded and measured, respectively) of the Relation's from and to Frames. The Sensors can use arbitrary orientations and pivot Points, as long as those are based on Frames that are connected with from and to.

Frame finally provides methods `getVelocitySensorOf(...)` and `getMeasuredVelocitySensorOf(...)` that are able to provide VelocitySensors that measure the velocity of an arbitrary, connected Frame f relative to the Frame instance the method is called on. Orientation and pivot Point can also be chosen freely, provided that they are based on Frames that are connected to the Frame instance. This is possible by using the method `getRelationsTo(...)` to find all Relations that build a way to the target Frame, and then combine the VelocitySensors of all Relations with the operations provided by the Sensors.

7.5 Related Work: Geometric Modeling in Robotics

The Robotics API's world model as presented in this thesis aims to provide an easy and safe mechanism for describing geometric relationships. Its central concept Frame represents a coordinate frame and is used to describe important parts of devices and other physical objects, as well as to parameterize device operations. A Frame can be seen as a node in an undirected graph, with Relations forming the edges. Relations employ the well established mechanism of coordinate transforms ([40], pp. 10) to describe the geometric relationship between a pair of Frames. In addition, a formalism of Jain [111] is adopted to describe the relative velocity of the Frame pair.

The Robotics API's world model provides a safe mechanism to calculate the relative transformation and velocity between arbitrary pairs of Frames. This is a significant advantage compared to libraries that provide only data types and operations that represent the pure mathematical concepts and leave the task of performing the correct operations to developers, like e.g. Eigen [113] or the Orocos Kinematics and Dynamics Library (KDL, [114]). The authors of KDL have, however, recently developed tools for semantic checking of KDL operations (De Laet et al. [115]). The KUKA Robot Language provides a type-safe approach by defining a language operator that works on geometric data types. This 'geometric operator' can best be compared with the `plus(...)` method provided by each Robotics API Frame. Methods for searching the Frame graph and determining the geometric relationship between arbitrary Frames, which are provided by the Robotics API's world model, are not provided by KRL and many other geometric libraries. An exception is ROS-TF [116], which supports such kinds of queries.

The general graph-based approach is characteristical for Topological Maps, according to a classification by Burgard and Hebert [117]. Most approaches of this class employ directed acyclic graphs, effectively resulting in a tree structure. Examples are ROS-TF [116] and all approaches based on the scene graph concept (Shuey et al. [118]) like the ones presented by Naef et al. [119], Smits ([109], pp. 121) and Blumenthal et al. [120]. In contrast, the Robotics API's world model imposes no direction on edges and allows for cyclic structures. The choice to use undirected edges was taken to not introduce an 'artificial' hierarchy in the graph, as Relations (edges) are intended to model geometric relationships between equal nodes. The roles of 'from' and 'to' Frame of a Relation only have a mathematical semantics (the Relation's Transformation represents ${}^{from}\mathbf{T}_{to}$) and are not meant to introduce a logical hierarchy. Cyclic structures are deliberately allowed in order to support modeling closed kinematic chains. This might be necessary to represent parallel robots, but also for situations where cooperating robot arms form kinematic chains (e.g. two robots holding the same workpiece with their grippers). Such situations are expressible with the Robotics API's world model. The tradeoff is the need to perform semantic checks during runtime to resolve inconsistencies in such situations.

Due to the undirected nature of the graph edges, the Robotics API's world model does not allow for representing part-whole relationships. For example, all Frames that are part of the structure of a robot arm are not distinguishable from any other Frame by just reasoning about the graph structure. An extension of the world model presented in

this thesis, which allows for expressing such topological relationships as well, is subject of Alwin Hoffmann's ongoing work.

Chapter 8

A Context-Aware Model for Typical Operations of Robotic Devices

Summary: The Robotics API core provides powerful and fine-grained software concepts for modeling all kinds of real-time critical operations. However, when controlling robots or other robotic devices, the desired operations often have inherent dependencies on preceding operations and their effect on devices' states. Furthermore, many robotics applications share recurring operation patterns. This chapter presents the Robotics API's Activity package, which introduces a model for context-aware operations. This model is also able to support common operation patterns and to combine them to form new patterns. The design of the Activity model has been published in [31].

The model of Commands introduced by the Robotics API's core package provides a very powerful, flexible basis for defining real-time critical operations. On the one hand, time-based synchronization of multiple Commands that may control multiple devices is easily possible. On the other hand, reactiveness is supported by the flexible State and EventHandler mechanism, which in particular includes reactions to events triggered by sensor measurements. The Command Takeover mechanism even allows for real-time transitions across Commands. In Sect. 6.4, this mechanism was introduced as a key to realize e.g. motion blending across multiple robot-specific Commands. In fact, the Takeover mechanism can be employed in all cases which demand continuous real-time control of actuators. Further possible applications of Takeover include stable force control of actuators (e.g. robots, grippers), velocity control of mobile robots or thrust control of flying robots. In these cases, it is not necessary to encode the complete flow of continuous real-time critical operations in one Robotics API Command. Provided that some of those operations are able to maintain all affected actuators in a stable state, successive operations may be started from a Robotics API application at any time

and can then take control of the actuators seamlessly. However, in such cases it is no longer possible to treat Commands independently of each other. For example, in order to achieve deterministic and repeatable motion blending across two Commands, the second Command has to make certain assumptions about the state the moving Actuator will be in when it takes over control from the first Command. Otherwise, the second Command would not be able to plan a deterministic motion continuation for the actuator. The Command model does not provide any support for modeling such dependencies between separate Commands.

Furthermore, though flexible and powerful, the Command model does not provide an elegant and easy-to-use syntax for application developers. This is particularly noticeable when comparing code for relatively basic tasks to its equivalent in robot languages like KUKA's KRL. This observation is not too surprising, as KRL can be seen as a Domain Specific Language. It was developed solely for the purpose of creating industrial robotics applications, whereas the Robotics API extends 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-centered style of KRL (or similar DSLs).

To overcome both problems mentioned above, 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 dependencies between Commands. Furthermore, the interface to application developers could be designed to be more convenient by reducing its complexity.

2. *Introducing an application development layer on top of the Command Layer.* A separate layer on top of the Command layer could be designed which allows for exchanging information across Commands and introduces 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 with an appropriate execution semantics that is based on Robotics API Commands and respects their dependencies 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 [121]), it seemed logical to choose one of the latter solutions. Additionally, all stated requirements like multi-robot coordination and sensor integration can 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 modern programming language ecosystem (the reference implementation is based on Java) 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 the host language.

To stay in the same programming language ecosystem, this work introduces a separate object-oriented layer on top of the Robotics API. This layer provides an abstraction of the Robotics API Command model that reflects the mechanisms useful for developers of industrial robotics applications. It is realized as a separate package that is decoupled from *robotics.core*. The central concept introduced by this package is called the *Activity*. Consequently, the package is called *robotics.activity*. The Robotics API core can be used completely without the Activity package, and the new package does not require any support by the RoboticsRuntime used, but builds solely on existing concepts in *robotics.core*. This section presents the design of the Activity package and illustrates how it solves the abovementioned challenges.

8.1 Modeling Device Operations by 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 controlled Actuator during or after the execution of the operation. The real-time critical operation logic of an Activity is implemented by a Robotics API Command. Like Commands, Activities can be executed in a synchronous or asynchronous manner. However, in contrast to Commands, Activities are not instantaneously delegated to a RoboticsRuntime for execution. Instead, the execution of Activities is controlled by a special scheduler. This scheduler maintains a history of executed Activities for each Actuator. When a new Activity is scheduled, the scheduler provides the new Activity with all previously scheduled Activities that affected the same Actuators as the newly scheduled Activity. The new Activity in turn provides the scheduler with information about which Actuators it is able to take control of, respecting the previous operations the Actuator has executed or is still executing. The scheduler then ensures two aspects: 1) Multiple Activities that are started in an asynchronous manner are scheduled so that they do not access one or more Actuators

concurrently and 2) if feasible, the Command provided by an Activity is scheduled to instantaneously take over a currently running Command (created by a preceding Activity) on the same RoboticsRuntime.

The class diagram in Fig. 8.1 depicts the static relationships between Activity, the aforementioned scheduler (*ActivityScheduler*) and further related concepts. The ActivityScheduler is designed as a singleton object (see [86], pp. 127) to ensure that all Activities are scheduled by the same scheduler instance. It maintains a history of executed Activities. This *ActivityHistory* assigns to each Actuator the last Activity of all executed Activities which affected this Actuator. If an Actuator was not affected previously, the ActivityScheduler does not provide any ActivityHistory for this Actuator.

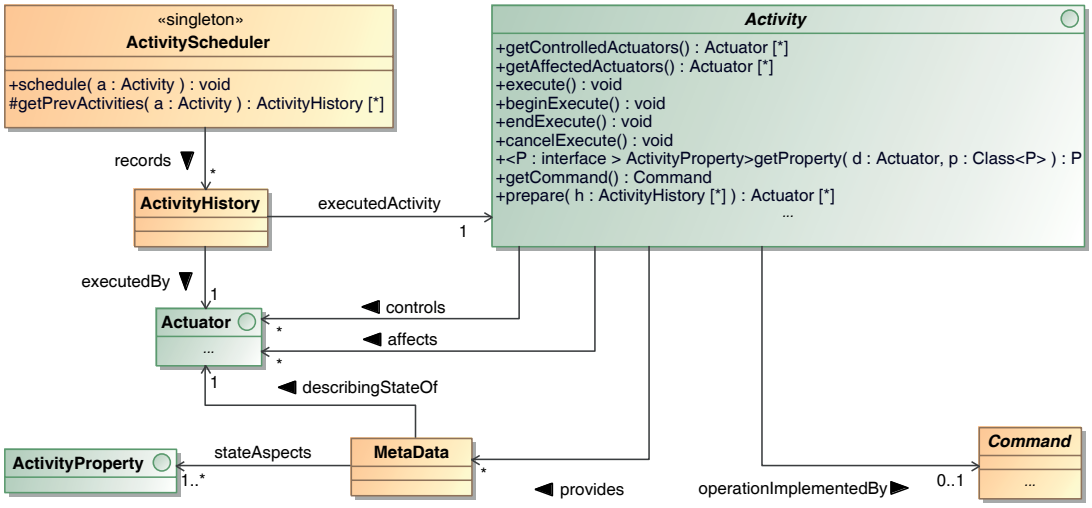

Figure 8.1: Core classes of the Robotics API Activity model.

When an Activity is started by a call to `execute()` or `beginExecute()`, it delegates the task of correct scheduling to the ActivityScheduler by calling its method `schedule(...)`. During scheduling, the ActivityScheduler will call the scheduled Activity's method `prepare(...)` and supply accurate ActivityHistories. The Activity implementation can inspect the Activities in the supplied list and retrieve *ActivityProperties* via the method `getProperty(...)`. Each ActivityProperty describes a certain aspect of a particular state of an Actuator. If the newly scheduled Activity creates its Command such that it is able to take over an Actuator in *all* of the described states, it may indicate this to the ActivityScheduler by including the Actuator in the list returned by `prepare(...)`. The ActivityScheduler can then analyze the situation and decide whether it is feasible to schedule the new Command to instantaneously take over a running Command in the RoboticsRuntime. An important obligation for concrete implementations of the method `prepare(...)` is to call the Activity's method `setCommand(...)` as a last step and pass the created Command as an argument.

As an example, consider an Activity s implementing a robot motion which is being

scheduled and has to prepare a Command. Let another Activity p be part of the history that implements a motion of the same robot, which is currently still being executed. p can provide one type of property that specifies the robot's Cartesian position when the motion was fully executed. Another type of property supplied by p can specify the robot's position, velocity and acceleration at a previously defined motion blending start point. If s constructs a Command that is able to move the robot in a deterministic way in both possible states, it may indicate to the scheduler that it is able to take over control of the robot. The scheduler can then perform Command scheduling on the RoboticsRuntime level. The exact scheduling algorithm used by the ActivityScheduler is discussed in detail in Sect. 8.2.

Through the introduction of the abovementioned scheduling mechanism for Activities, it is possible to consider dependencies to previous real-time operations when planning new ones. As mentioned before, this is one important motivation for the design of the Activity package. The lifecycle of Activities from the perspective of application developers is presented in Sect. 8.3 and further exemplified in later chapters. Additionally, the Activity concept provides mechanisms for supporting common patterns of real-time critical operations in robotics applications:

- *Execution guards* allow for defining conditions that lead to an Activity being gracefully stopped to avoid dangerous situations. Execution guards are introduced in Sect. 8.4.
- *Triggers* can be used to attach additional Activities to any Activity. They are executed when certain real-time events occur. The trigger mechanism is explained in Sect. 8.5.
- *Composition patterns* offer general and powerful ways of creating arbitrary complex real-time critical combinations of Activities. The last section of this chapter (Sect. 8.6) proposes a set of such patterns.

The challenge here is to ensure that these mechanisms do not interfere with each other and the scheduling and take-over concepts outlined above. It has turned out that all those mechanisms can be handled in a generic way in the design of the Activity class and a set of derived classes for common composition patterns. Thus, developers of robotics applications can employ all those mechanisms without having to worry about unforeseen effects on real-time actuator control. System integrators who want to add support for new kinds of actuators may also derive new Activities from two particular implementations of Activity, which are provided by *robotics.activity*:

- *AbstractActivity* is an abstract implementation of the Activity interface that provides implementations of all methods but `prepare`. To implement concrete Activities, this Activity can be employed as base class to reuse its functionality.

- *BasicActivity* is a concrete implementation of the Activity interface that can serve as an *adapter* ([86], pp. 139) for converting a Command to an Activity. It provides a constructor that accepts a Command and a set of controlled Actuators.

These two classes can help to reuse functionality in many cases.

8.2 Context-Aware Scheduling of Activities

As outlined earlier, the execution of each Activity is controlled internally by the ActivityScheduler. The implementations of its methods `execute()` and `beginExecute()` ensure this by submitting the Activity to the ActivityScheduler by calling its method `schedule(...)`. For the following explanation of the ActivityScheduler's scheduling algorithm, it is important to clarify the distinction between *affected* and *controlled* Actuators of an Activity:

- *Controlled Actuators* are those Actuators which are directly controlled by the Command created by this Activity.
- *Affected Actuators* include all controlled Actuators as well as additional Actuators who may not be affected concurrently by other Activities.

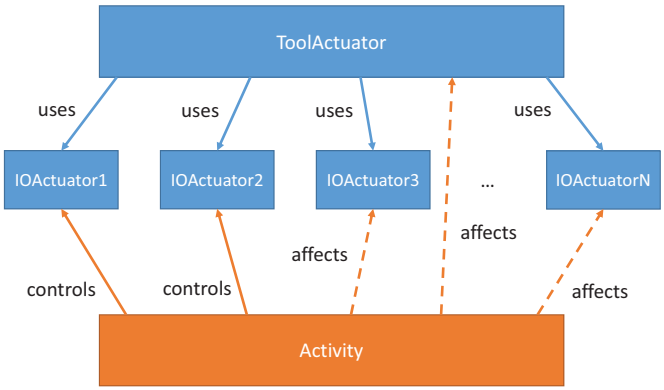

Figure 8.2: Controlled and affected Actuators of an Activity.

Controlled Actuators of an Activity a are effectively all Actuators that are targeted by any RuntimeCommand that is part of the Command created by a. If a subsequent Activity wants to take over a, it must be able to take over control of all of a's controlled Actuators. This can be checked by the ActivityScheduler to decide whether Command scheduling is feasible. The introduction of affected Actuators as a superset of controlled Actuators allows for a more flexible 'locking' of Actuators by the ActivityScheduler. For example, consider an I/O based robot tool that is modeled as an Actuator, but is controlled 'indirectly' via other Actuators and Sensors which model the output and input

channels physically connected to the tool. Fig. 8.2 sketches this tool and an Activity that is operating it. The Activity's set of controlled Actuators only includes those I/O Actuators that are necessary for the current operation. However, it is desirable to 'lock' also other I/O Actuators that are connected to the tool, to ensure that no other operation has the chance to interfere with the current tool operation. Therefore, the remaining I/O Actuators should be added to the set of affected Actuators. The robot tool itself, which cannot be directly controlled due to the fact that it builds solely on the I/O Actuators, might also be added to the set of affected Actuators of all tool-related Activities. In this way, it will be locked for other Activities that affect it, e.g. Activities controlling a tool-changing system. In the following, it is said that an Actuator A is *affected* by an Activity a if A is contained in a's set of affected Actuators, and that A is *controlled* by an Activity a if A is contained in a's set of controlled Actuators.

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. Note also that preceding Activities whose execution terminated with an error are not considered relevant, as their meta data might be inaccurate. 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 Activity 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. 6.7).

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's Command 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 pre-plan its execution and determine all Actuators Ψ that it is able to take control of.
   d) If a succeeds in pre-planning and provides Ψ, do the following:
      i. If Ψ is a subset of all Actuators controlled by p, schedule a's Command after p's Command.
      ii. Otherwise, await execution end of p's Command, then start a's Command.
   e) Otherwise, if a indicates that it is not able to do pre-planning while p is still running, wait for p to terminate and return to step 3c.
4. In any case, if a's Command has been scheduled or started, create a new ActivityHistory for each Actuator affected by a and store a as last executed Activity in these ActivityHistories.

Note again that if an Activity a indicates that it can take control over a set of Actuators Ψ (step 3c), it has to ensure that the created Command is able to control *all Actuators* in Ψ in *all possible states* that are described by the ActivityProperties offered by the preceding, still running Activity. The Activity scheduling mechanism (and the Command scheduling mechanism as well) does not give any guarantees regarding the point in time where the Command to be scheduled will actually have been loaded in the Robot Control Core. Thus, it can only be ensured that the Actuators in Ψ will be in *one of the possible states* described by the Activity's meta data, but not which of them. An Activity is allowed to offer no ActivityProperties at all for Actuators it controls. In this case, a scheduled Activity may only include those Actuators in Ψ which it is able to control in *any* possible physical state. This is feasible e.g. for motion Activities that employ online motion planning and are thus able to take control over Actuators reliably, regardless of their current state.

The Command scheduling mechanism in its current form (see Sect. 6.7) only allows a single Command to be scheduled after exactly one predecessor. This design mainly follows the mechanism offered by the SoftRobotRCC, which at present can also schedule only a single RPI primitive net as successor to exactly one running RPI primitive net. The restrictions in the above Activity scheduling algorithm to schedule one Activity at a time (step 2) and to keep only one Activity running before scheduling a new Activity (step 3b) are direct consequences. In the ongoing dissertation work of Michael Vistein, there are efforts to release this restriction in the SoftRobotRCC by introducing a much more flexible RPI primitive net scheduling mechanism. The scheduling algorithm used

in the ActivityScheduler could be revised in order to release the abovementioned restrictions. However, effects on the workflow in applications also have to be considered. This will be discussed in Sect. 8.3.

The sequence diagram in Fig. 8.3 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 behavior. 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 primitive net. 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. The complete lifecycle of an Activity will be discussed in the next section.

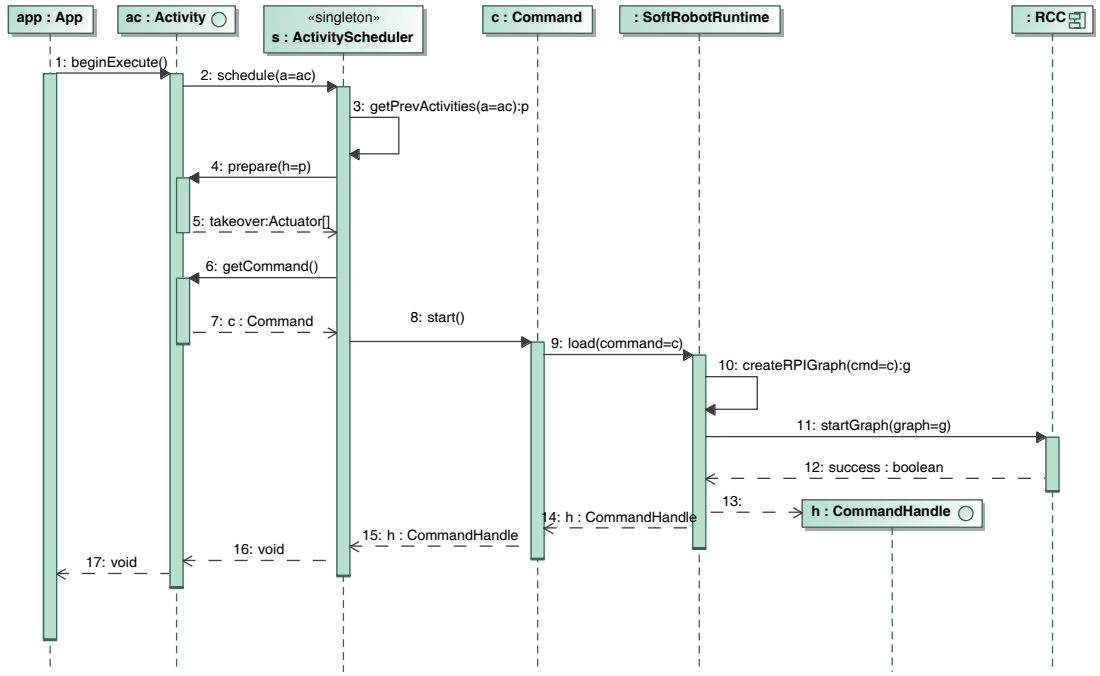

Figure 8.3: Sequence diagram showing the asynchronous execution of an Activity.

8.3 The Lifecycle of an Activity

Activities are stateful entities that track the progress of their execution. Their internal state is modeled by *ActivityStatus*, which can be queried anytime by calling an Activity's method `getStatus()`. This section will present the lifecycle of an Activity, i.e. the different states, their semantics and the possible transitions. The section will also recap the semantics of the various methods for controlling Activity execution, which is based on the current ActivityStatus.

The set of possible ActivityStatus values and valid transitions between those states is displayed in the right part of Fig. 8.4. The ActivityStatus values have the following semantics:

- *New*: The Activity has not been started and can be modified (e.g. by adding triggers, declaring exceptions) in arbitrary ways.
- *Scheduled*: The Activity has been started and is being scheduled by the ActivityScheduler. Once an Activity has entered this state, it may no longer be modified nor started again.
- *Running*: The Activity's Command has been started by the RoboticsRuntime. In particular, if the Command has been scheduled to take over a preceding Command, the takeover process has been executed.
- *Failed*: There has been an error during the starting or the execution of the Activity's Command. This may have happened in any of the states Scheduled, Running or Maintaining.
- *Completed*: The Activity's execution has fully completed. In particular, the Activity's Command has terminated regularly.
- *Maintaining*: The Activity's execution has completed and the Activity is maintaining the controlled Actuators' final states. In particular, the Activity's Command may still be running and actively controlling Actuators.

In the following, an Activity will be called *terminated* when it is in status Completed, Failed or Maintaining.

The left part of Fig. 8.4 indicates in which states certain Activity operations are allowed and how their execution time depends on the ActivityStatus:

- `execute()` is allowed to be called solely when an Activity has status New. This method will return as soon as the Activity has terminated.
- `beginExecute()` is also allowed to be called only in status New. In contrast to `execute()`, this method will return as soon as the Activity has reached status Running.

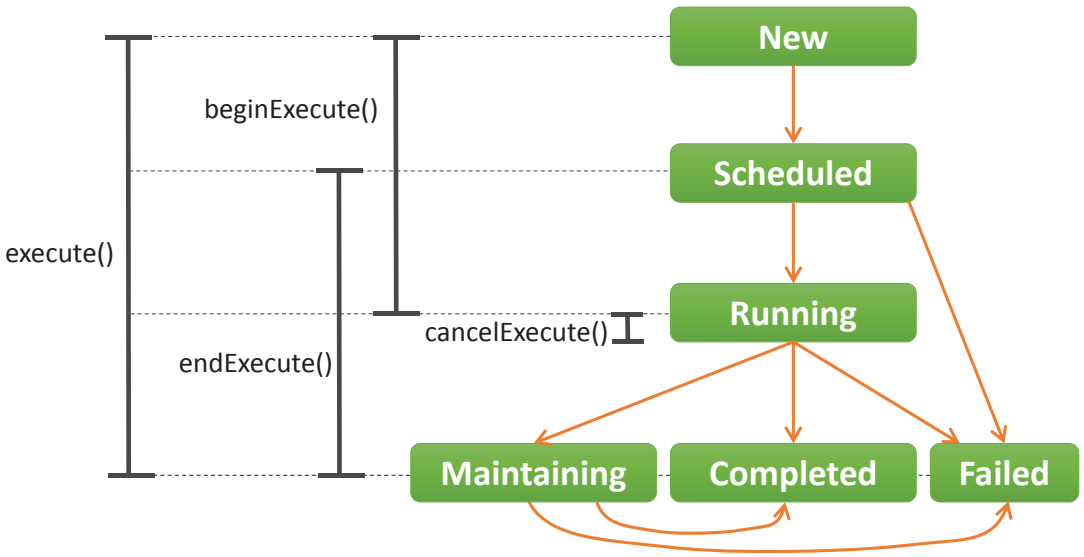

Figure 8.4: The lifecycle of an Activity.

- `endExecute()` is allowed to be called as soon as an Activity has reached Scheduled status. This in particular means that it is allowed to be called when `beginExecute()` has returned. `endExecute()` will return as soon as the the Activity has terminated.
- `cancelExecute()` may be called when the Activity is in status Running. It will send a cancel request to the Activity's running Command and will return immediately. `endExecute()` can be used to synchronize the application to the actual termination of the Activity.

When an operation is called in a status in which calling is not allowed, an exception is thrown. The method `execute()` can be used in applications whenever a strictly sequential flow of operations is intended. Alternatively, the application flow continues when Activities are started via `beginExecute()`, which allows to employ the time it takes an actuator to perform an operation for other tasks, e.g. to plan subsequent operations. In this case, `endExecute()` can be used at any time to synchronize the application flow to the end of a running operation. If an error occurred during the operation, an appropriate exception will be thrown by `endExecute()`.

The Activity scheduling algorithm presented in Sect. 8.2 has a considerable effect on the workflow in Robotics API applications. This in particular regards the case when multiple Activities are subsequently started with `beginExecute()` that affect the same actuator(s). Each call to this method will serve as a synchronization point in the sense that the workflow will continue only as soon as the respective Activity has entered status Running. This implies that the preceding Activity has terminated. Alternatively,

all subsequent Activities that are started via `beginExecute()` could be queued by the ActivityScheduler and their Commands scheduled in the RoboticsRuntime as soon as possible. In this way, a completely asynchronous semantics of `beginExecute()` could be realized. The current design was chosen for two reasons:

1. The application workflow can not be too far ahead of currently running actuator operations, which makes it easier to track the application behavior. Experienced developers can use additional threads for truly asynchronous program parts anytime if needed.

2. Errors that occurred during execution of an Activity can be indicated 'locally' in applications, i.e. when scheduling a subsequent Activity affecting the same Actuator at the latest. `execute()` as well as `endExecute()` will throw appropriate exceptions immediately, but this can obviously not be done by `beginExecute()` for errors that occurr while the Activity is Running. With the current scheduling algorithm, exceptions of an Activity can be thrown when a subsequent Activity is started via `beginExecute()`, as the information about all errors is available at this point.

The rest of this section will discuss the motivation for the particular ActivityStatus Maintaining. As explained above, the Activity's Command may continue running though the Activity counts as terminated. This concept can be employed when an Activity has to perform active control of an actuator to ensure that the intended state of the Actuator – which may be 'promised' by the meta data provided by the Activity – is upheld. For example, consider a robot that is moving to a goal Frame, which is itself moving relative to the robot (e.g. a workpiece on a conveyor). It is desirable to maintain the robot's position relative to the moving goal Frame even when the motion Activity is terminated and the application workflow continues. Another example is force-controlled manipulation: When an actuator executes a series of Activities and it is necessary to maintain a certain force on the environment during the complete series, each Activity in the series may have to actively control the force also after its termination. Both examples can be realized by the Activity maintaining mechanism.

An Activity that needs to maintain control of certain Actuators after termination has to consider this when creating a Command. The created Command (or a part of it) has to be able to perform the necessary control of all relevant Actuators to maintain the desired conditions. Furthermore, some Robotics API State has to be defined whose activeness indicates that the Command is now maintaining actuator control. Finally, the Command has to allow to be taken over during this maintaining. The Activity's implementation of the `prepare(...)` method can then use a variant of the `setCommand(...)` method that takes a State as additional argument. The Activity class interprets the first entering of this State as trigger for changing the ActivityStatus to Maintaining, thus terminating the Activity's execution.

When an Activity is in status Maintaining, a new Activity affecting the same Actuators can only be scheduled if it is able to take control over all Actuators of the maintaining

Activity. Thus, application developers have to mind this when executing subsequent Activities. However, the ActivityScheduler integrates a mechanism to detect unwanted cases, i.e. when a subsequent Activity is scheduled that is not able to take over a maintaining Activity. The ActivityScheduler will throw an exception in this case. Otherwise, a deadlock situation would occur, as the newly scheduled Activity would wait for the maintaining Activity to terminate (which might never occur).

8.4 Guarding Execution of Activities

Activity provides the method `addGuard(s : State [*])`. All States added as guards using this method will lead to the Activity being canceled. The implementation of `setCommand(...)` in Activity ensures this by combining all the specified States in an OrState and adding a CommandCanceller to the Activity's Command that reacts to the first occurrence of the aforementioned OrState. Thus, it is ensured that a cancel signal will be issued at most once to the Command, and it will be issued if either of the guarding States becomes active.

Implementations of concrete Activities have to make sure that the cancel signal is interpreted properly by the created Command. This can be done e.g. by using Actions that react to canceling appropriately, and by composing the Command in such a way that the appropriate part handles canceling. Thus, the cancel signal must be forwarded inside the Command appropriately, as the CommandCanceller added by the Activity will only notify the outmost part of the supplied Command.

Activity implementations furthermore have to ensure the following:

- Either cancelling will bring all Actuators in one of the states described by the set of ActivityProperties of this Activity,
- or the Command throws a CommandRtException when canceling is finished.

In the former case, a canceled Activity behaves like any other Activity in that subsequent Activities can rely on the meta data of the canceled Activity when planning their execution. In the latter case, throwing an exception will lead to the Activity terminating with status Failed. This in turn will cause the ActivityScheduler to ignore the Activity when determining preceding Activities during the scheduling of a new Activity. Thus, the new Activity will not rely on inaccurate meta data when planning its execution.

8.5 Triggering Dependent Activities

A common requirement in industrial robotics applications is to execute operations of robot tools at certain execution states of robot motions. The Activity model provides a convenient mechanism for executing all kinds of *triggered Activities* depending on the execution state of a *main Activity*. This mechanism is realized in a generic way by the Activity class. The goal is to design this mechanism in a way that triggered Activities

do not influence the execution of the main Activity, while at the same time ensuring that the triggered Activities are guaranteed to be started within a short time after the triggering condition has occured. This section will present the realization of triggered Activities and illustrate to which extent the abovementioned goals can be realized.

Each Activity provides the method `addTrigger(t : State, a : Activity)`. This method can be used to define a triggered Activity that should be executed in parallel to the main Activity, as soon as a supplied State is becoming active the first time. An arbitrary number of triggered Activities may be added to an Activity. It is guaranteed that each triggered Activity is started within a defined time after the specified State has become active. The only way to achieve this, based on the Robotics API's Command model, is to encapsulate the Commands of all triggered Activities as well as the Command of the main Activity in one TransactionCommand. This, however, has certain implications also for the lifecycle of the main Activity. Fig. 8.5 shows an Activity with three attached triggered Activities. The triggered Activities a and b do not influence the main Activity's execution, as their execution ends before the main Activity terminates. However, the triggered Activity c terminates at a later point in time than the main Activity. Thus, the execution time of the respective TransactionCommand will be prolonged by triggered Activity c. This affects the main Activity, as a subsequent Activity can only take over in the main Activity's final state[1]. Thus, in some cases triggered Activities might prevent takeover of the main Activity in an intermediate state by subsequent Activities. The only alternative solution to preserve takeoverability of the main Activity would be to allow all triggered Activities to be aborted in this case. This, however, might have unexpected consequences for the Actuators controlled by the triggered Activities and is thus considered not feasible.

Triggered Activities may affect and control other Actuators than the main Activity. By adding the affected Actuators of all triggered Activities to the set of affected Actuators of the main Activity, it can be ensured that all Activities that affect the same Actuators like the triggered Activities will be scheduled after the triggered Activities have terminated. On the other hand, all triggered Activities can be supplied with accurate ActivityHistories, thus they can plan their execution based on ActivityProperties provided by preceding Activities. However, as triggers are started in an event-driven manner, they are in general not able to take over control of Actuators when the main Activity is started. This implies that triggered Activities are also not suited for taking over preceding Activities that are in Maintaining status. This might be possible in the future as the instantaneous Command transition mechanism becomes more flexible.

Dealing with the controlled Actuators of triggered Activities is more difficult. If they were added to the set of controlled Actuators of the main Activity, this would allow takeover of the main Activity only if the subsequent Activity can take over control of all the triggered Activities' Actuators as well. Thus, the requirements to subsequent Activities willing to take over control would be raised significantly by adding triggered

[1] Command takeover currently only works on full Commands, i.e. the TransactionCommand has to be taken over completely

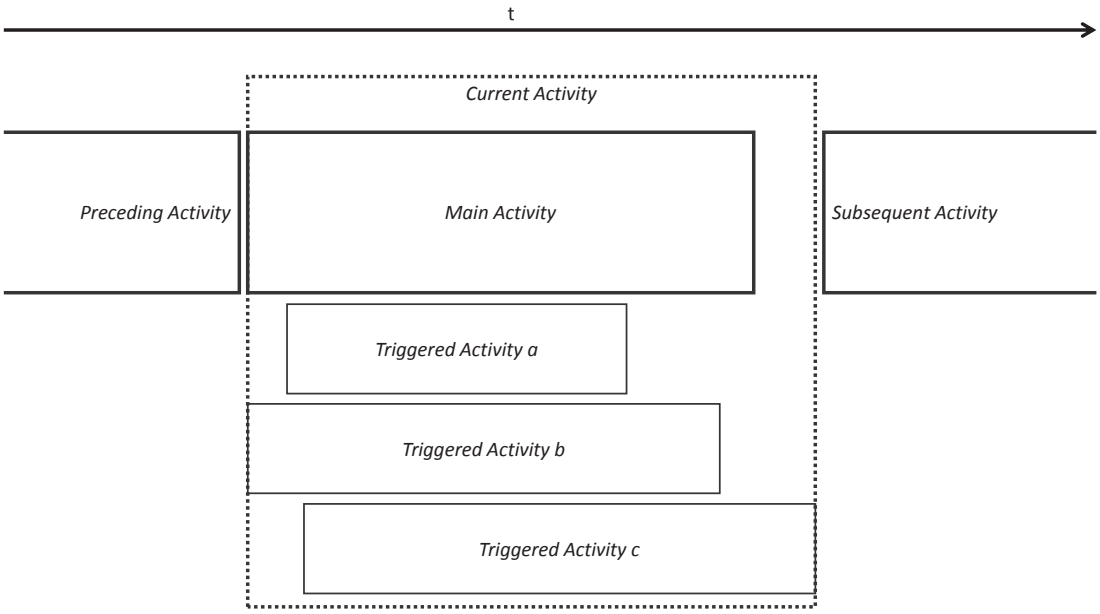

Figure 8.5: Activity with attached triggered Activities.

Activities. It would rather be desirable to preserve the possibility for takeover in all cases where the triggered Activities are already completed when the main Activity can be taken over. Such cases are expected to occur frequently in typical applications, as it is assumed that triggered Activities in many cases model short-running operations that e.g. control robot tools. To achieve this, the following measures are taken:

- The controlled Actuators of triggered Activities are 'hidden' from future Activities by not adding them to the set of controlled Actuators of the main Activity.
- The ActivityProperties specified by triggered Activities are not propagated to the main Activity.
- Activities that specify a maintaining State are not allowed to be used as triggered Activities.
- The main Activity is only allowed to be taken over if none of the triggered Activities is still running.

By hiding the triggered Activities' controlled Actuators, subsequent Activities can take over the main Activity without having to control the Actuators of triggered Activities.

The control of those Actuators will not be interrupted unexpectedly in any case. For each Actuator controlled by triggered Activities, the combination of the above measures creates a similar situation like when the Actuator has not executed any Activity before. When a subsequent Activity controlling such Actuators is started, it is guaranteed that no Activities are currently controlling those Actuators and no ActivityProperties describing the Actuators' states are supplied. Before new Activities can control those Actuators, they have to find out information about the Actuators' current states themselves. This can be done by querying the Actuators about their current state directly, e.g. by calling appropriate methods.

In the following, the construction of an appropriate TransactionCommand for an Activity with attached triggered Activities is presented. Assume an Activity A with a set of triggered Activities τ and triggering States σ, where τ_i is the i-th Activity in τ and σ_i the State triggering this Activity. A's method `prepare(...)` is called in the course of scheduling, and a set of ActivityHistories Θ is supplied. Note that, as mentioned above, this set will also contain ActivityHistories for the Actuators affected by the triggered Activities. The construction of a TransactionCommand T_A is started once the main Activity's method `setCommand(...)` is called, which is to be done by its implementation of `prepare(...)`. C_A is the Command passed to `setCommand(...)`. It represents the concrete implementation of the main Activity's control operation, irrespective of all triggered Activities. C_i is the Command that will be created by τ_i. The task now is to create an appropriate TransactionCommand T_A that combines C_A and all Commands τ_i in an appropriate way. This is done as follows:

1. C_A is added to T_A as auto-started Command.
2. T_A's CancelState is forwarded to C_A by a CommandCanceller.
3. For each τ_i, the following steps are performed:
   a) τ_i's method `prepare(...)` is called, passing the ActivityHistories Θ.
   b) If τ_i specifies a maintaining State (this is only known after `prepare(...)` has been called), an exception is thrown, aborting the scheduling process of A.
   c) C_i (which has been created now) is added to T_A as ordinary (not auto-started) Command.
   d) A CommandStarter is added to T_A that starts C_i when σ_i becomes active the first time.
   e) T_A's CancelState is forwarded to C_i by a CommandCanceller.
4. An OrState α is created that contains the ActiveStates of all Activities τ_i.
5. The negation of α ('no τ_i is active') is added to a new AndState, together with C_A's TakeoverAllowedState. This AndState ('C_A allows takeover and no τ_i is active') is specified as T_A's TakeoverAllowedState.

Note that the default completion logic of TransactionCommands ('no child command active') is also appropriate here.

8.6 Supporting Operation Patterns by Activity Composition

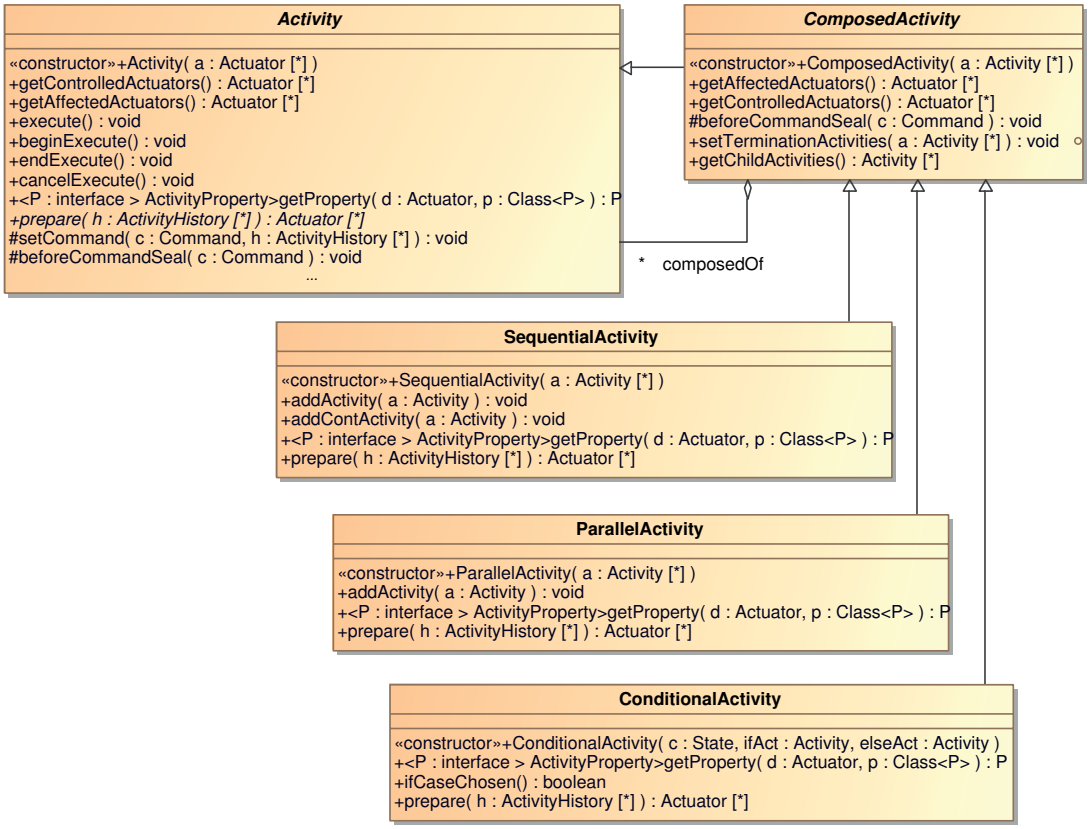

Figure 8.6: Sequential, parallel and conditional composition of Activities.

Arbitrarily complex real-time critical operations can be constructed from Activities with various composition patterns. The following patterns are directly supported by the Activity Extension:

- *Sequential composition.* The specified Activities are executed one after another (optionally with blending) with a bounded delay in between two Activities.
- *Parallel composition.* The specified Activities are started at the same time and run in parallel with a bounded timing mismatch between them.

- *Conditional execution.* Based on a given condition, either one or another Activity is started. The time to evaluate the condition and start the appropriate Activity is bounded.

The patterns are implemented by the classes *SequentialActivity*, *ParallelActivity* and *ConditionalActivity*, which are displayed in Fig. 8.6. The figure also depicts the abstract base class *ComposedActivity*, which implements some aspects in a generic way. In the following, the distribution of functionality will be illustrated.

ComposedActivity is designed as a Composite (cf. [86], pp. 163) containing arbitrary child Activities. It redefines the following operations of the Activity class in a generic way:

- `getAffectedActuators()` collects the affected Actuators of all child Activities and merges them with the affected Actuators of the ComposedActivity itself.
- `getControlledActuators()` in a similar way collects the controlled Actuators of all child Activities and merges them with the controlled Actuators of the ComposedActivity itself.
- `beforeCommandSeal(c:Command)` redefines a 'hook' method of the Activity class to correctly handle termination Activities of a ComposedActivity (see below).

ComposedActivities allow for defining some of their childs as *termination Activities* using the method `setTerminationActivities(...)`. If one of the termination Activities is completed, the ComposedActivity is also defined to be completed. This implies that the Commands created by all other child Activities are aborted as well, which has to be considered when using termination Activities. ComposedActivity provides the method `getChildActivities()` for retrieving child Activities.

Further methods are redefined or added in the concrete composite Activities and are discussed in the following.

SequentialActivity

A sequence of arbitrary Activities can be composed using a SequentialActivity. The implementation of SequentialActivity ensures that the delay between the end of one Activity and the start of the next Activity of the sequence is bounded. Additionally, each Activity in the sequence can also be executed in a *continuous* way. Such Activities are allowed to be taken over by subsequent Activities, just like when those are scheduled continuously by the ActivityScheduler. As SequentialActivities are completely pre-planned and scheduled atomically, it is guaranteed that the take-over process will take place if a subsequent Activity supports this. In contrast, the implementation of the ActivityScheduler does not guarantee that take-over will take place, as the Activity planning and scheduling process is not performed in a real-time capable execution environment.

SequentialActivity provides two methods for adding Activities to the sequence: `addActivity(...)`, which adds an Activity to the end of the sequence which is executed completely, and `addContActivity(...)`, which adds an Activity to the end of the sequence that can be taken over. When a list of Activities is supplied to the constructor of SequentialActivity, this has the same effect like constructing a SequentialActivity with an empty list and adding the single Activities one by one by calling `addActivity(...)`.

To provide ActivityProperties of their child Activities, SequentialActivity redefines the method `getProperty(...)`. To determine an ActivityProperty of the requested kind, the implementation searches all child Activities for matching kinds. If multiple child Activities provide an ActivityProperty of the requested kind, the one of the Activity that is nearest to the end of the sequence is returned. Thus, it is ensured that the returned data is the most current one.

SequentialActivity furthermore redefines the method `prepare(...)` to create an appropriate Command that models a sequential execution of all its child Activities. The basic idea is encapsulating all Commands created by the child Activities in one TransactionCommand and adding appropriate EventHandlers. The first step is calling the child Activities' `prepare(...)` methods to enable them to create their Commands. As the child Activities are executed in a sequential manner, the set of ActivityHistories supplied to each `prepare(...)` method has to be modified accordingly. The first child Activity in the sequence is simply supplied with the set of ActivityHistories that has been passed to the SequentialActivity's `prepare(...)` method. For each of the remaining Activities a_i in the sequence, the set of ActivityHistories is updated as follows: The Actuators affected by Activity a_{i-1}, which is preceding a_i in the sequence, are determined. For each Actuator, a new ActivityHistory h is created, storing a_{i-1} as last Activity of this Actuator. h is added to the set of ActivityHistories and any other ActivityHistory for the same Actuator is removed from the set. Finally, the updated set of ActivityHistories is passed in the call to a_i's method `prepare(...)`.

A second step is concerned with the correct handling of controlled Actuators in the course of the sequence. In particular, the SequentialActivity itself is able to take over control from a preceding Activity if the first child Activity is able to do so. Thus, the set of controllable Actuators returned by the call to the first child Activity's method `prepare(...)` is finally returned by the SequentialActivity's method `prepare(...)`. For each child Activity a_i^c that has been added as continuously executed Activity, it has to be determined whether the subsequent Activity a_{i+1} can take control over all Actuators of a_i^c. Thus, it is checked whether the set of Actuators controlled by a_i^c is a subset of the Actuators returned by a_{i+1}'s method `prepare(...)`.

Finally, the Commands created by all child Activities are collected and added to a common TransactionCommand. Fig. 8.7 illustrates the TransactionCommand created by a SequentialActivity with three child Activities. The first Activity in the sequence is added to the TransactionCommand as auto-started Command. Each further Command C_i, which was created by the i-th Activity in the sequence, is added as a regular Command to the TransactionCommand. For each C_i a CommandStarter is attached to the

TransactionCommand that starts C_i if the preceding Command has terminated (i.e., its CompletedState is active) and the TransactionCommand has not been canceled (i.e., its CancelState is not active). Additionally, if an Activity has been added as continuous Activity to the sequence, it is checked if the subsequent Activity in the sequence is able to take over (which has been determined in the second step as illustrated above). In this case (cf. C_2 in Fig. 8.7), the Command is stopped once it allows takeover (i.e., its TakeoverAllowedState is active) and the subsequent Command is started. To allow for continuous execution of the SequentialActivity as a whole, the TransactionCommand is parameterized to allow takeover iff the last Command in the sequence allows takeover (i.e., its TakeoverAllowedState is active). Finally, the TransactionCommand's CancelState is forwarded to each Command in the sequence by a CommandCanceller. Thus, any running Command can execute specific canceling logic when the SequentialActivity and thus the TransactionCommand is canceled.

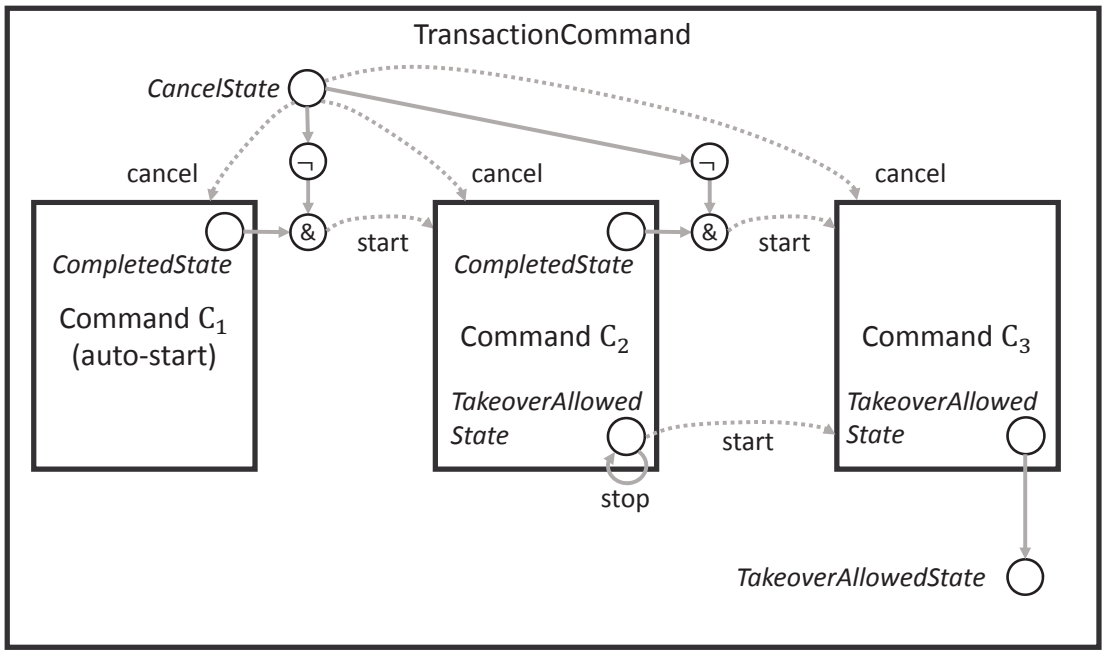

Figure 8.7: Command created by a SequentialActivity with three child Activities.

A SequentialActivity is completed when the last Activity in the sequence has completed execution. Any exception in one of the sequence's Activities causes the sequence to be aborted. Activities that specify a Maintaining phase can be added to the sequence. If they are last in the sequence, the SequentialActivity will perform Maintaining and has to be taken over by subsequent Activities. If an Activity is added to the sequence and the preceding Activity in the sequence has a Maintaining phase, the subsequent Activity has to be able to take over this Activity, otherwise the SequentialActivity as a whole is considered illegal and scheduling is aborted.

Sequential composition proved to be useful in various cases. During the work on this thesis, Activities for the Schunk MEG 50 gripper and the Kolver Pluto screwdriver were implemented in this way, which will be presented in Chap. 9.

ParallelActivity

ParallelActivity has been designed to start and run arbitrary Activities in parallel. Its implementation guarantees that all child Activities are started within a defined delay and that there is no significant time drift between those Activities during their execution. It also allows for taking over arbitrary preceding Activities, if the child Activities in sum are able to take over control of all Actuators controlled by the preceding Activity. If subsequent Activities are able to take over control of all Actuators controlled by the child Activities, ParallelActivity supports this as well.

A ParallelActivity can be created by supplying a list of Activities to its constructor (cf. Fig. 8.6). Child Activities can be added to an existing ParallelActivity using the method `addActivity(...)`. In either case, it is checked whether the sets of affected Actuators of the child Activities have any overlap, which is not allowed by Activities that run in parallel. ParallelActivity's implementation of `getProperty(...)` is straightforward: It can just delegate calls to the single child Activity which affects the specified Actuator. The above check ensures that there exists at most one such Activity.

ParallelActivity redefines the method `prepare(...)`, which has to create a single Command that implements the semantics of real-time parallel execution of all child Activities. The set of ActivityHistories supplied to this method can be forwarded to each child Activity's `prepare(...)` method without modification, as the parallel semantics imposes no ordering among the child Activities like in SequentialActivity. A ParallelActivity is able to take control of any Actuator that can be taken over by any of its child Activities. Thus, the sets of Actuators returned by all the child Activities' `prepare(...)` methods are merged and returned by the ParallelActivity's implementation of `prepare(...)`. ParallelActivity finally adds each of the child Activities' Commands as auto-started Commands to a common TransactionCommand. Fig. 8.8 illustrates the structure of the created Command for a ParallelActivity containing three child Activities. To allow the ParallelActivity to be taken over by subsequent Activities, the Commands of all child Activities must either be finished or allow takeover themselves. Consequently, the TransactionCommand's TakeoverAllowedState is active only when, in *all* child Commands, either the child Command's TakeoverAllowedState is active or its ActiveState is not active.

The ParallelActivity is considered completed when all of its child Activities have completed execution. If some child Activities specify a Maintaining phase, the ParallelActivity will also transit to Maintaining and has to be taken over by subsequent Activities that can take over all active child Activities. Any exception in one of the child Activities will abort the ParallelActivity.

Chapter 10 will demonstrate how parallel composition of Activities can be employed to operate teams of robots in a tightly synchronized manner.

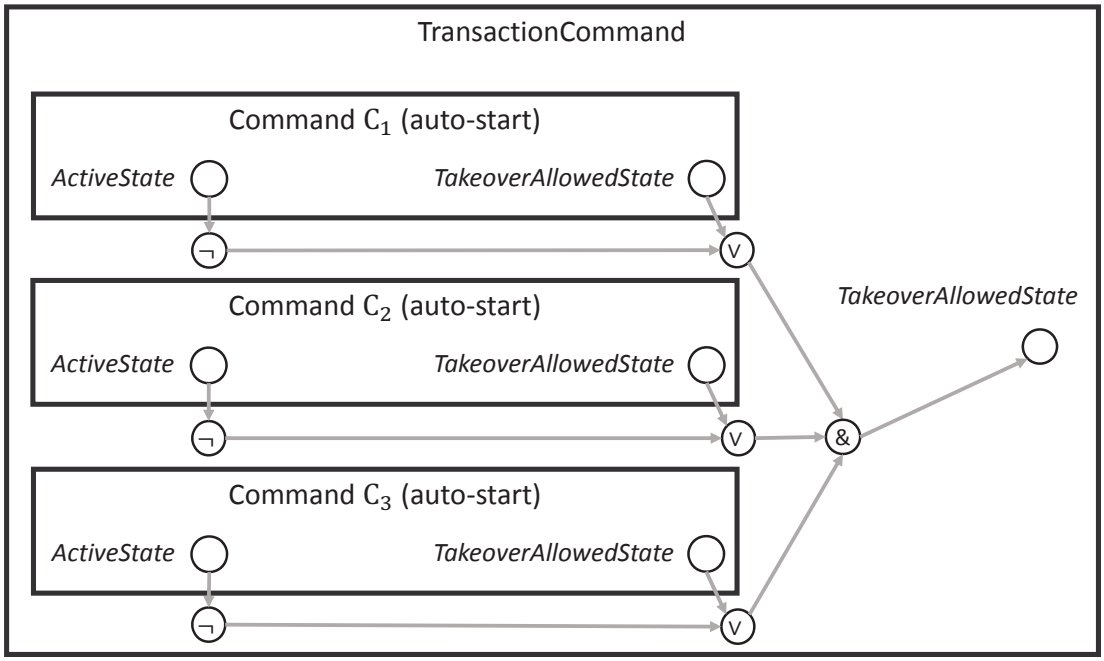

Figure 8.8: Command created by a ParallelActivity with three child Activities.

ConditionalActivity

The Activity concepts allows for creating a continuous workflow of real-time critical tasks. ConditionalActivity adds a lot of flexibility by introducing a mechanism for real-time critical branching of a workflow. While it is also possible to branch a workflow in the host programming language, this has a major disadvantage in the context of continuously executed Activities: The branching decision has to be evaluated *before* planning of subsequent Activities can be started. ConditionalActivity allows for pre-planning alternative execution branches and defer the actual decision to the latest possible point in time.

A ConditionalActivity has to be fully parameterized upon construction. Its constructor (cf. Fig. 8.6) demands one BooleanSensor and two Activities to be specified. The BooleanSensor serves as branching condition: Depending on the value measured by this sensor at the time instant when the ConditionalActivity is started, either the first Activity ('if-case') or the second Activity ('else-case') is started. Note that this is the time instant when the ConditionalActivity (precisely, its Command) is started in the RoboticsRuntime, not when the Activity is being scheduled.

ConditionalActivity supports to continue planning of different subsequent Activities depending on the execution branch taken. This is possible by its method `ifCaseChosen()`. This method blocks callers until the branching decision has been taken, i.e. until one of the alternatives has actually been started in the RoboticsRuntime. It also determines

which of the child Activities is actually executed. In case the first of the specified Activities (the 'if-case') is executed, it returns **true**, otherwise **false**. This allows for employing simple programming language branching mechanisms to continue the application workflow appropriately and still leaves time for planning further Activities.

ConditionalActivity redefines the method `getProperty(...)` with a similar intention: To provide accurate information about the actual execution branch that is executed, the method blocks until the decision has been taken and determines which of the two child Activities has actually been started. Only the ActivityProperties provided by this Activity are returned. From this point on, subsequent Activities can start to plan their execution accordingly. Note that this is completely transparent to those future Activities, as the ConditionalActivity's method `getProperty(...)` behaves in the same way like in other Activities, it just might take more time to deliver the requested ActivityProperties.

To create an appropriate Command, ConditionalActivity redefines the method `prepare(...)`. Both child Activities' `prepare(...)` methods are called and passed the set of ActivityHistories supplied to the ConditionalActivity without modification. This is feasible as the chosen child Activity will be executed immediately when the ConditionalActivity is started. To achieve this, the Commands created by both child Activities are added as auto-started Commands to a common TransactionCommand (cf. Fig. 8.9). TransactionCommand allows for specifying a BooleanSensor as guard for auto-starting child Commands. This mechanism is employed here: The BooleanSensor B that has been specified upon construction of the ConditionalActivity is used directly as guard for autostarting the first child Activity's Command ('if-case'). The second child Activity's Command ('else-case') is guarded by the inverse of B, termed $\neg B$, which is created by BooleanSensor's method `not()`. To be notified about the branch that is actually chosen, `prepare(...)` registers two separate WorkflowEffects (cf. Sect. 6.4) to each of the child Commands' StartedStates. The one WorkflowEffect that is actually run will start a separate programming language thread which stores the branching decision and signals this to a semaphore. Other methods, i.e. `ifCaseChosen()` and `getProperty(...)`, that waited for this semaphore can continue their execution according to the branching decision.

The TransactionCommand is parameterized to allow takeover (i.e. activate its TakeoverAllowedState) if any of the two child Commands allows takeover. The ConditionalActivity can take over preceding Activities only if both execution branches can take over control of the respective Actuators. Thus, the ConditionalActivity's `prepare(...)` method returns the intersection of the sets of Actuators returned by the two child Activities' `prepare(...)` methods. As it is not known a priori which of the two branches will be executed, ConditionalActivity affects and controls the union of the affected and controlled Actuators of the two inner Activities.

A ConditionalActivity terminates when the child Activity that is executed terminates, either due to an exception or because of regular completion. Maintaining of child Activities is supported and will cause the ConditionalActivity to enter Maintaining. For subsequent Activities, the situation is the same as if the child Activity had been executed directly.

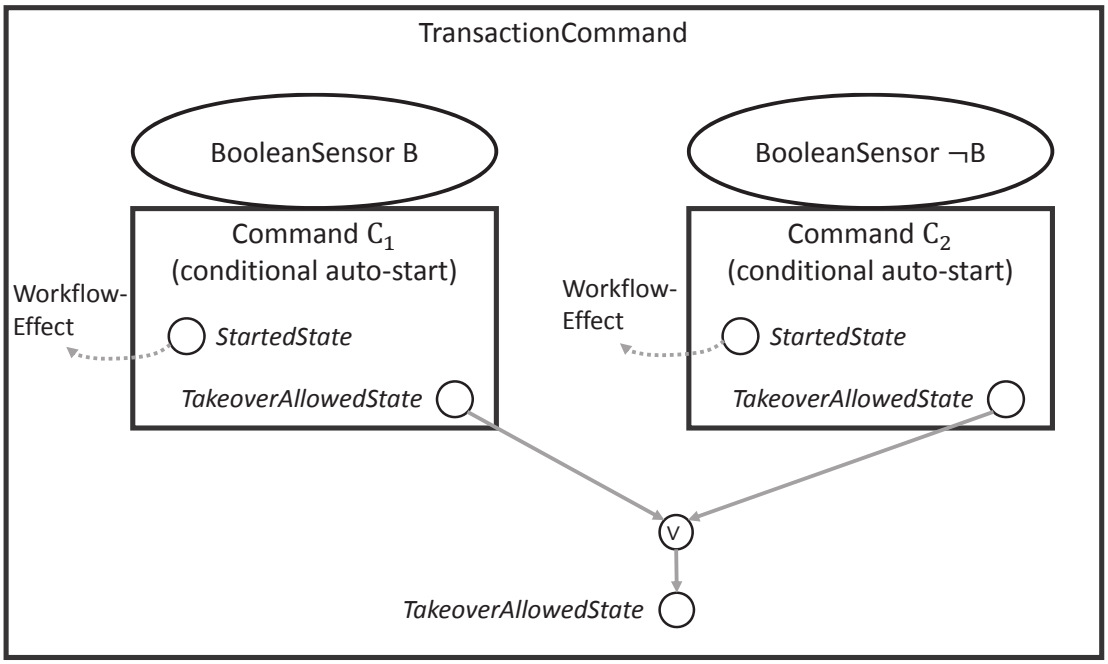

Figure 8.9: Command created by a ConditionalActivity.

Conditional execution is useful e.g. for continuous execution of motions with intermediate branching. Thus, it allows e.g. to solve a particular weakness of the KUKA Robot Language: When sensor inputs are used for a branching decision, KRL will stop its advance run in order to not evaluate the sensor measurements prematurely. Thus, motion blending is not possible in this case. ConditionalActivities can solve such cases and provide deterministic branching at the latest possible point in time.

8.7 Related Work

The Robotics API's Activity model is intended to provide application developers with a convenient way of describing and composing hard real-time critical operations. Some concepts of Activities have equivalents in the KUKA Robot Language. In particular, this regards triggered Activities, which are similar to KRL Triggers. Furthermore, the scheduling concept of Activities to some part mimicks KRL's advance run feature, and motion blending is realized on this basis, similar to KRL. Activity execution guards have no direct equivalent in KRL, though they can be emulated by KRL's interrupts. Other features of Activities by far exceed the mechanisms provided by KRL: All composition patterns of Activities are not supported by KRL. In particular, truly parallel execution of arbitrary operations is not achievable at all with KRL, and sensor-based branching decisions have severe limitations as well (cf. Sect. 8.6).

As discussed in the context of the Robotics API's Command model (cf. Sect. 6.12), most

existing software frameworks in robotics do not provide an explicit model of operations, but instead rely on the 'natural' object-oriented approach of augmenting classes with operations, or rely on message passing. This has limitations when it comes to augmenting operations with e.g. execution guards, and composing operations. An exception are Task Trees, whose limitations have been discussed earlier in Sect. 6.12.

Still, in most existing systems there is at least an implicit notion of a *task* that is executed. Lütkebohle et al. [122] analyzed many existing robotic software frameworks and derived a pattern for describing the execution state of such tasks, the *Task-State Pattern*. They suggest that the modeling and tracking of task states in distributed, component-based systems can be handled separately from the application-specific system structure by a generic toolkit. As the Robotics API provides an explicit, exhaustive model of tasks, it is natural to model the state of such tasks as part of the task specification itself. However, Lütkebohle et al. also suggest a candidate for a general task lifecycle, which is similar to the Robotics API's Activity lifecycle. The Activity states *New*, *Running*, *Failed* and *Completed* have direct equivalents in the Task-State Pattern lifecycle candidate. This candidate does not contain equivalents to the Activity states *Scheduled* and *Maintaining*, which result from the distinct execution semantics of Activities, involving pre-planning and active control beyond the natural end of an Activity. In turn, Activities have no equivalent to two states of the Task-State Pattern lifecycle candidate. The first is the state *CANCEL REQUESTED*, that explicitly models that the operation should be canceled. Tasks can react to this by actually canceling their execution, but may also decide to ignore the request and return to their ordinary execution state. Thus, applications are able to track exactly whether a cancel request was ignored or maybe not even delivered to the task. The second such state is *UPDATE REQUESTED*, which enables tasks to acknowledge changed goals during their execution. While some Robotics API Activities (and Actions) are in fact able to accept new goals during execution, the explicit modeling of such a state was not considered important in the scope of this work. The same applies to the *CANCEL REQUESTED* state discussed above. The possible transitions in both state models are roughly equal, with minor differences that deserve no detailed discussion.

Finkemeyer et al. have developed the concept of Manipulation Primitives and Manipulation Primitive Nets [123, 124, 125]. They state that this approach is particularly suited for sensor-based manipulation tasks and demonstrate this with several examples like inserting a light bulb into a bayonet socket or a battery into a cellphone. The rest of this sections outlines how the Manipulation Primitive approach can be embedded in the software design of the Robotics API.

A Manipulation Primitive (MP) is defined as a triple $\{\mathcal{HM}, \tau, \lambda\}$, where $\mathcal{HM}$ is a hybrid move definition, τ contains tool commands and λ resembles a stop condition which causes execution of the MP to end. The semantics of a hybrid move will be discussed later. The exact definition of a tool command is not specified and is "kept very open and very general" [125]. The stop condition is a boolean expression that maps sensor measurements to a boolean value. The Robotics API's Command model is able to start and stop arbitrary Commands (including tool commands like contained in τ) during the

execution of any other Command. This is controlled by the event mechanism based on Sensors and the State concept, which is able to model boolean expressions like τ as well. The Activity model even provides more direct equivalents of τ and λ in form of triggered Activities and cancel conditions. Triggered Activities are still more general than simple tool commands.

Manipulation Primitive Nets, as described in detail in [124], allow for forming tree structures from single Manipulation Primitives. Depending on disjunctive parts of a MP's stop condition, a subsequent MP is chosen for execution. To achieve a defined control behavior during the transition between MPs, the MP execution engine maintains all control values of the last active MP. In contrast, Robotics API Activities allow for an operation-specific handling of a stable control state during the Activities' Maintaining status. The Command scheduling mechanism as presented in this thesis only allows a single Command to be scheduled as successor. It is, however, possible to integrate alternative execution branches in a subsequent Command and decide which one should be executed upon starting of the subsequent Command. Chapter 10 will introduce an application of this approach to motion blending. In his ongoing dissertation work, Michael Vistein is investigating a novel scheduling algorithm which promises to provide at least the flexibility of the Manipulation Primitive Net approach.

In Manipulation Primitives, the hybrid move $\mathcal{HM}$ is defined as a tuple $\{\mathcal{TF}, \mathcal{D}\}$, where $\mathcal{TF}$ is the so-called Task Frame which is the reference frame in which various set-points are applied by certain control algorithms specified in $\mathcal{D}$. In [124], Finkemeyer et al. state that they extended the Task Frame Formalism by Bruyninckx and De Schutter [126] to allow their Task Frame to be coupled w.r.t. any frame in the workcell. For this purpose, they define a special anchor frame relative to which the Task Frame is fixed in position, velocity and acceleration. An additional frame is specified as reference frame for feedforward compensation, in order to allow hybrid moves to be executed also in dynamic frames transparently. $\mathcal{D}$ is described in [125] as the 'Adaptive Selection Matrix', which is used to specify, for each component of the Task Frame, which control algorithms to use w.r.t. sensor-based precedence rules. For example, $\mathcal{D}$ can specify to use force-control in the translational dimensions of the Task Frame, but switch to position control if sensors indicate that no contact force can be established. Experiments during the work on this thesis indicate that it is possible to implement this hybrid move specification as a Robotics API Action. The Task Frame $\mathcal{TF}$ can be constructed by simply combining appropriate Robotics API TransformationSensors and VelocitySensors. The implementation of an adaptive selection matrix in such an Action can use Robotics API Sensors as well to switch between different control algorithms. It is even possible to reuse existing Actions that implement position, velocity or force control algorithms and compose them in the implementation of a Hybrid Move Action. As discussed above, all other components needed to form Manipulation Primitives and Manipulation Primitive Nets already exist in the Robotics API's Command and Activity models.

Chapter 9

Modeling Robot Tools

Summary: Robots are usually equipped with tools in order to perform their tasks. This chapter presents how robot tools can be modeled and controlled based on the concepts of the Robotics API. Using two concrete examples, the reusability of the Robotics API's core concepts and a set of tool-specific Robotics API extensions is demonstrated.

A large class of robot tools is controlled using basic I/O communication mechanisms as introduced in Sect. 3.2. This chapter thus first introduces the design of a Robotics API extension that provides basic support for such communication (Sect. 9.1). Subsequently, extensions supporting the Schunk MEG 50 gripper (Sect. 9.2) and the Kolver Pluto screwdriver (Sect. 9.3) are presented. Reusable parts of those extensions are highlighted.

This chapter employs class diagrams for illustrating static structure and communication diagrams for describing interactions during runtime. The diagrams in this chapter are not a precise description of implementation details, but rather focus on the underlying concepts and abstract from details.

9.1 Basic I/O Support

To support devices that are controlled via electrical input and output signals, the Robotics API *IO extension* has been created. This extension introduces digital and analog outputs (i.e. channels on which values can be written by applications). Those outputs are modeled as Robotics API Actuators. Furthermore, digital and analog inputs (i.e. channels on which values can be read by applications) are introduced. Inputs are modeled as Robotics API Devices that provide Sensors which measure the values at the input channels. This approach has two advantages: on the one hand, the IO extension can be

used to implement control of other devices, e.g. the abovementioned gripper and screwdriver, and on the other hand, configuration of the communication setup of those other devices can be realized using the configuration management mechanisms of the Robotics API. This section will introduce the structure of the IO extension as prerequisite for the Schunk MEG and Screwdriver extensions presented later. As inputs and outputs are a quite simple form of Robotics API Devices, they are well suited for illustrating the application of the Robotics API's architectural concepts in detail.

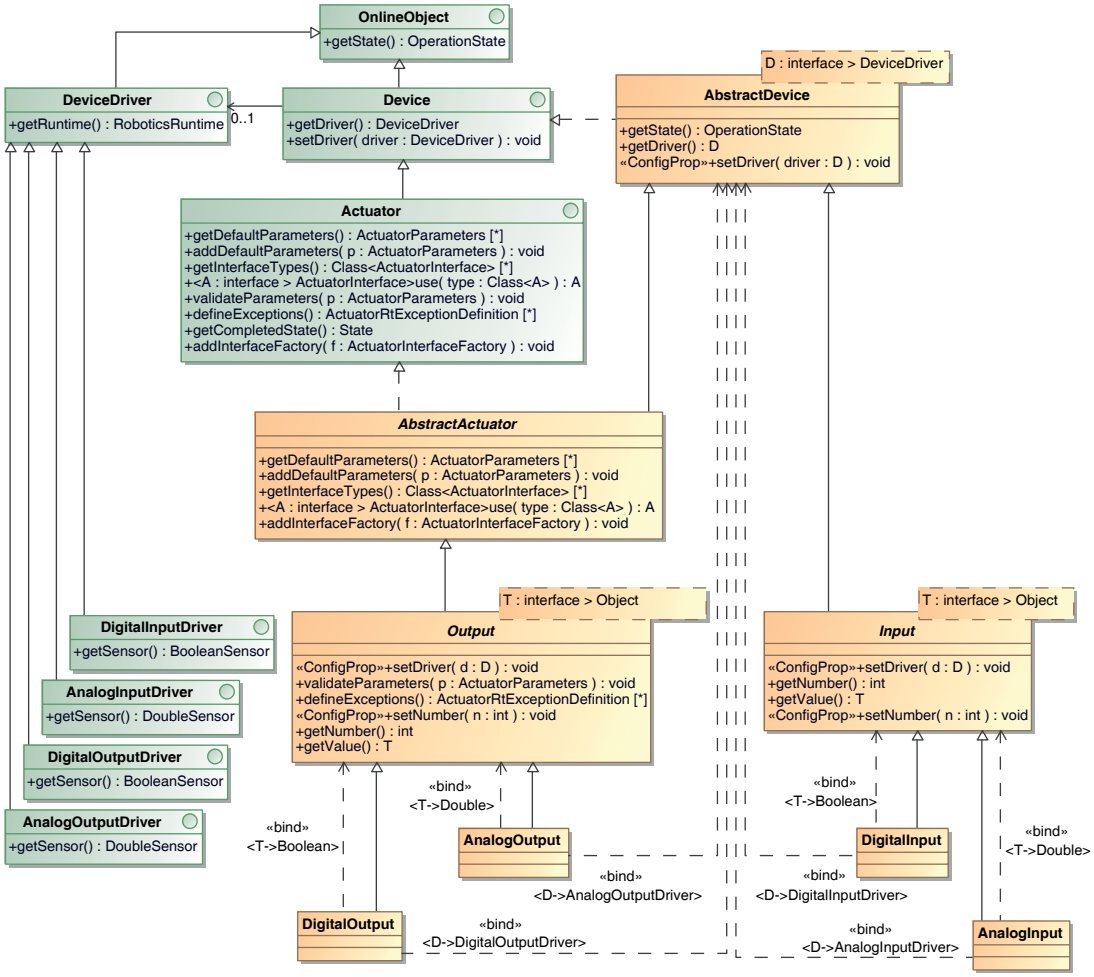

Figure 9.1: Devices and DeviceDrivers to support I/O communication.

The modeling of inputs and outputs as Robotics API Devices is illustrated by the class diagram in Fig. 9.1. Inputs are modeled by the abstract class *Input* and the concrete realizations *DigitalInput* and *AnalogInput*. Input is a Device and reuses functionality from AbstractDevice by inheritance. AbstractDevice does not require a DeviceDriver in order to work properly. Its method `setDriver` is marked with the *ConfigProp* stereotype

to make the DeviceDriver a configurable property. However, the attribute *optional* of the applied *ConfigProp* stereotype is set to *true*, thus allowing for the DeviceDriver property of AbstractDevice to be left unconfigured. Still, AbstractDevice provides a default implementation of the method `getState`, which relies on the OperationState provided by the Device's DeviceDriver, if a DeviceDriver is configured. This is feasible as DeviceDrivers are OnlineObjects as well, and it is sensible because DeviceDrivers can communicate with the specific RoboticsRuntime implementation to determine the actual state of the physical device. If no DeviceDriver is configured, `getState` can return a default OperationState (e.g. `UNKNOWN`).

As Input is a Device, instances of Input can be configured and are initialized during application startup. However, as Input does not implement the Actuator interface, instances cannot be controlled via Actions and RuntimeCommands, cannot be parameterized with ActuatorParameters and do not provide ActuatorInterfaces. The respective drivers (*InputDriver*, *AnalogInputDriver* and *DigitalInputDriver*) are required to provide a Sensor which is able to measure the current value at the physical input channel.

AbstractActuator inherits functionality from AbstractDevice and provides implementations of some methods defined by Actuator. The methods `addDefaultParameters` and `getDefaultParameters` store and retrieve default ActuatorParameters for Actuators. Furthermore, `addInterfaceFactory`, `getInterfaceTypes` and `use` implement handling of ActuatorInterfaces.

Outputs are modeled by the classes *DigitalOutput* and *AnalogOutput*, which are both generalized by the abstract class *Output*. By inheritance, Output reuses functionality from AbstractDevice and AbstractActuator:

- *AbstractDevice* allows for configuring and accessing an optional DeviceDriver and provides a default implementation for determining a Device's operation state.
- The functionality for handling of ActuatorParameters and ActuatorInterfaces is provided by *AbstractActuator*.

The Output class implements further methods defined in interfaces, redefines methods from base classes and also provides new methods:

- `setDriver(...)` redefines `AbstractDevice#setDriver(...)` and changes the stereotype *ConfigProp* by specifying its *optional* attribute to be *false*. This makes a DeviceDriver a mandatory configuration property of each Output.
- `validateParameters(...)` is intended for Actuators to check the validity of ActuatorParameters specified for their operations and throw an exception in case they are not valid. As the IO extension does not define any ActuatorParameters, Output only provides an empty implementation of this method.

- `defineExceptions()` defines exceptions that can be raised during the execution of operations on Outputs. The IO extension provides *CommunicationException* (indicates some problem with the physical connection to the Output) and *ValueOutOfRangeException* (indicates that the voltage to set exceeded the allowed range).
- `setNumber(...)` assigns the logical number of the physical output to control to the Output device. This is a configurable property as well. The number corresponds to specific electrical connections on the I/O controller.
- `getNumber()` gets the logical output number that was configured for the Output.
- `getValue()` gets the value that is currently set at the physical output.

As all Actuators in the Robotics API are stateless objects (cf. Sect. 6.4), `getValue()` has to retrieve the currently set value from the hardware driver inside the RoboticsRuntime associated with the Output. The bridge to this runtime driver is created by *AnalogOutputDriver* (for AnalogOutputs) and *DigitalOutputDriver* (for DigitalOutputs). Both drivers are required to provide a Sensor which is able to measure the value currently set at the physical output. Like all OnlineObjects, both drivers also have to be able to determine the operational state of the output device. This should also be done by querying the respective hardware driver.

For controlling Outputs, Actions are needed. The IO extension provides one Action for each type of Output, named *MirrorAnalogValue* and *MirrorDigitalValue* (see Fig. 9.2). Each Action is parameterized with a Sensor and is required to write all values measured by this Sensor to the Output the Action is executed for. Both Actions allow to specify a boolean flag *continuous*. If this flag is set, the Action's implementation should continuously forward all values measured. Otherwise, it should only forward the first measured value and raise the Action's *Completed* State afterwards. Outputs should expect values between 0 and 1 and their hardware drivers have to map those values to the voltage range of the physical output. In case of SetDigitalValue, the mapping is straightforward (minimum voltage for sensor value 'false' and maximum voltage for value 'true'). The mapping of values during execution of SetAnalogValue is a little bit more complicated, as values outside the range 0..1 can occur. In this case, the real-time driver may ignore those values and indicate an error. For propagating this error to Robotics API applications, the abovementioned ValueOutOfRangeException may be used.

Though the IO extension defines only one Action per type of output, each Action can be employed for various use cases through appropriate Command construction and parameterization. To offer this flexibility to application developers in a convenient way, the IO extension introduces ActuatorInterfaces for AnalogOutput and DigitalOutput as depicted in Fig. 9.3. As both ActuatorInterfaces' methods differ only in the types of their arguments, the following explanation of the methods of *AnalogOutputInterface* can be directly transferred to *DigitalOutputInterface*. The only exception is the method `pulseValue(..)`, which requires an additional argument for analog outputs.

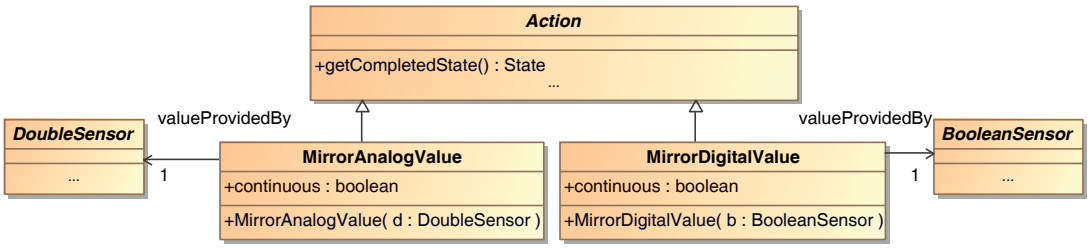

Figure 9.2: Actions for controlling Outputs.

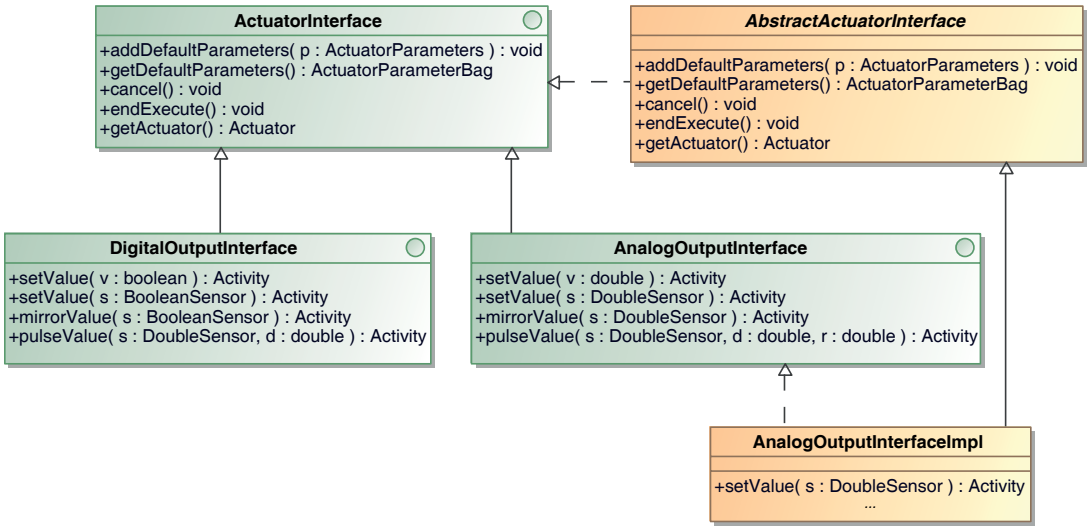

Figure 9.3: ActuatorInterfaces provided by DigitalOutput and AnalogOutput.

- `setValue(v : double)` creates an Activity whose execution sets the given value `v` to the analog output.
- `setValue(s : DoubleSensor)` is similar, but sets the value measured by the Sensor `s` at the moment the Activity's execution is started.
- `mirrorValue(s : DoubleSensor)` creates an Activity that continuously sets all values measured by the Sensor `s` until Activity execution is canceled.
- `pulseValue(s : DoubleSensor, d : double, r : double)` creates an Activity that sets the value measured by the Sensor `s` for duration `d` and afterwards sets the constant value given by `r`. The similar method for digital outputs just sets the opposite of the previously set value before terminating.

The choice to offer three different kinds of Activities (for setting, mirroring and pulsing values at outputs) was guided by the requirements of industrial applications that

have been identified in the analysis phase of the SoftRobot project. To illustrate the realization of those patterns based on one type of Action, the communication diagram in Fig. 9.4 displays the flow of interactions in AnalogOutputInterfaceImpl's `setValue` method. To create a mirrorValue Activity instead, only one additional interaction is necessary. In the following, all interactions will be explained briefly.

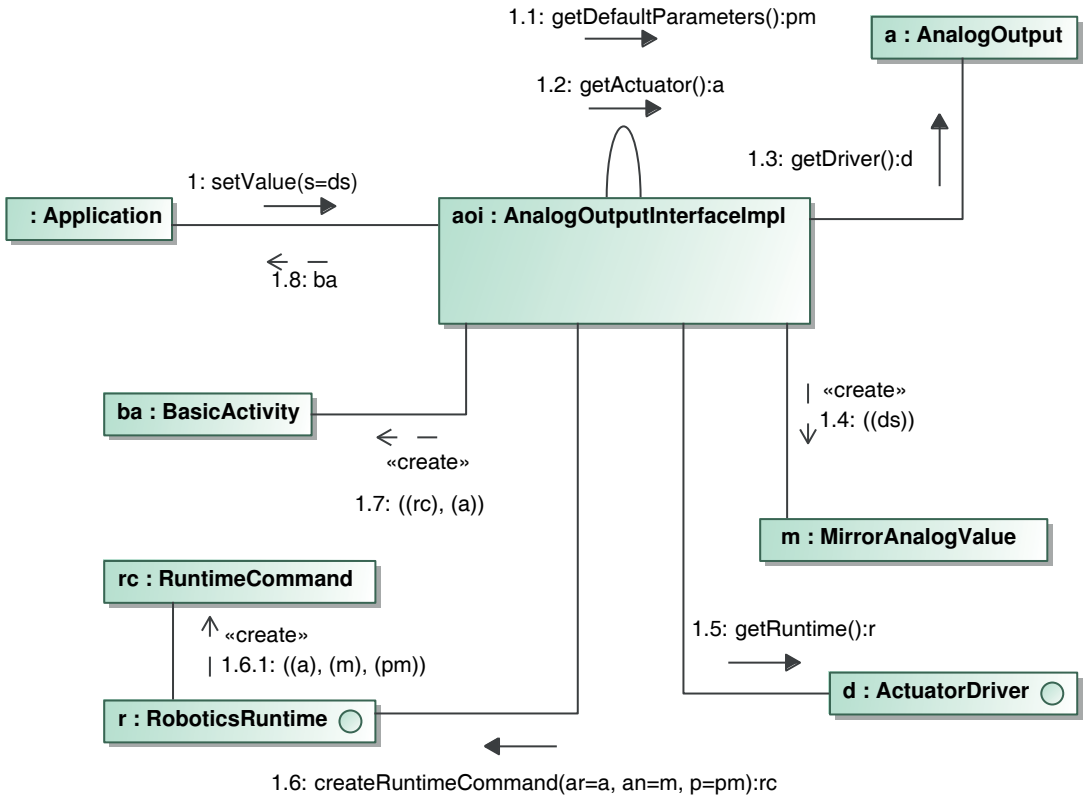

Figure 9.4: Activity construction in AnalogOutputInterfaceImpl's method setValue().

The interaction flow is triggered by a call to `aoi.setValue(ds)`, with argument `ds` of type DoubleSensor. This method first determines the default set of ActuatorParameters `pm` to use for this Activity (no other parameters have been passed in the call to setValue) by calling aoi's method `getDefaultParameters()` (1.1) and the AnalogOutput `a` that is controlled by calling the method `getActuator()` (1.2). The Actuator's ActuatorDriver is also retrieved by calling its method `getDriver()` (1.3). Then, the method creates a MirrorAnalogValue instance `m` and passes `ds` as constructor argument (1.4). Note that `m`'s *continuous* attribute by default has the value **`false`**. Using the aforementioned ActuatorDriver, the appropriate RoboticsRuntime `r` is retrieved via `d.getRuntime()` (1.5). After that, the RoboticsRuntime's role as a factory for Commands is employed to construct a RuntimeCommand with the call `r.createRuntimeCommand(a, m, pm)`. The created RuntimeCommand `rc` is finally used to create a BasicActivity `ba` for the

AnalogOutput `a`. Returning the Activity `ba` to the caller of `aoi.setValue(..)` concludes the interaction flow. The created Activity will mirror the first value measured by the given DoubleSensor to the AnalogOutput.

When setting a value is not adequate and constantly mirroring values is desired instead, the same interaction flow can be used and just one additional interaction has to be added: the *continuous* flag of the created MirrorAnalogValue Action has to be set to *true*. Then, the Action's implementation is expected to forward every value measured by the Sensor until the RuntimeCommand is canceled. When canceled, the implementation of RuntimeCommand will by default forward the cancel signal to the SetAnalogValue action, whose implementation is expected to terminate.

The implementation of `setValue(v:double)` is very simple: It can just create a ConstantDoubleSensor (see Fig. 6.3) that constantly delivers `v` and then rely on `setValue(s:DoubleSensor)`. Realizing the method `pulseValue(..)` is a bit more complicated, as it involves executing a MirrorAnalogValue Action for a certain time and afterwards executing a second MirrorValueAction. However, this can be realized by combining RuntimeCommands, WaitCommands and TransactionCommands appropriately and thus requires no further I/O specific Actions.

The generation of appropriate RPI primitive nets for I/O Actuators and Actions is relatively simple and straightforward. Fig. 9.5 shows a schematic illustration of the RuntimeCommand created when calling `setValue(0.5)`. The RuntimeCommand contains the MirrorAnalogValueAction with additional completion logic, and a ConstantDoubleSensor as described above, and the AnalogOutput with runtime-specific AnalogOutputDriver. In the figure, the representations of those concepts in the generated RPI primitive net are marked. For ConstantDoubleSensors, an RPI primitive instance of type *Core::DoubleValue* is employed, which just provides a constant double value. This value is forwarded to an RPI primitive of type *IO::OutDouble*, which writes the supplied values to the physical analog output and provides a boolean OutPort indicating whether the last write operation is completed. This information is interpreted as *Completed* State of the Actuator. MirrorAnalogValue's completion is calculated by the *Core::BooleanOr* type primitive, which combines an external dataflow indicating canceling of the RuntimeCommand and a completion signal by the Action's implementation itself. This completion signal is implemented using a *Core::BooleanValue* primitive which supplies a *true* value. This value, however, is delivered to the abovementioned *Core::BooleanOr* primitive using a *delayed* dataflow connection (inidicated by a dotted line). In this way, the Action's implementation will forward the first value measured by the Sensor and then indicate completion in the next execution cycle. The RuntimeCommand itself is completed when both the Action and the ActuatorDriver signal completion, thus both dataflows are combined with a *Core::BooleanAnd* primitive.

The implementation of concrete InputDrivers and OutputDrivers for the SoftRobotRuntime are not discussed in detail. However, these drivers need to provide information that is required for creating RPI primitive nets like the one in Fig. 9.5. In this case, for example, the implementation of a SetAnalogValueDriver has to provide values for *deviceId* and *port* that are required for parameterizing the *IO::OutDouble* primitive.

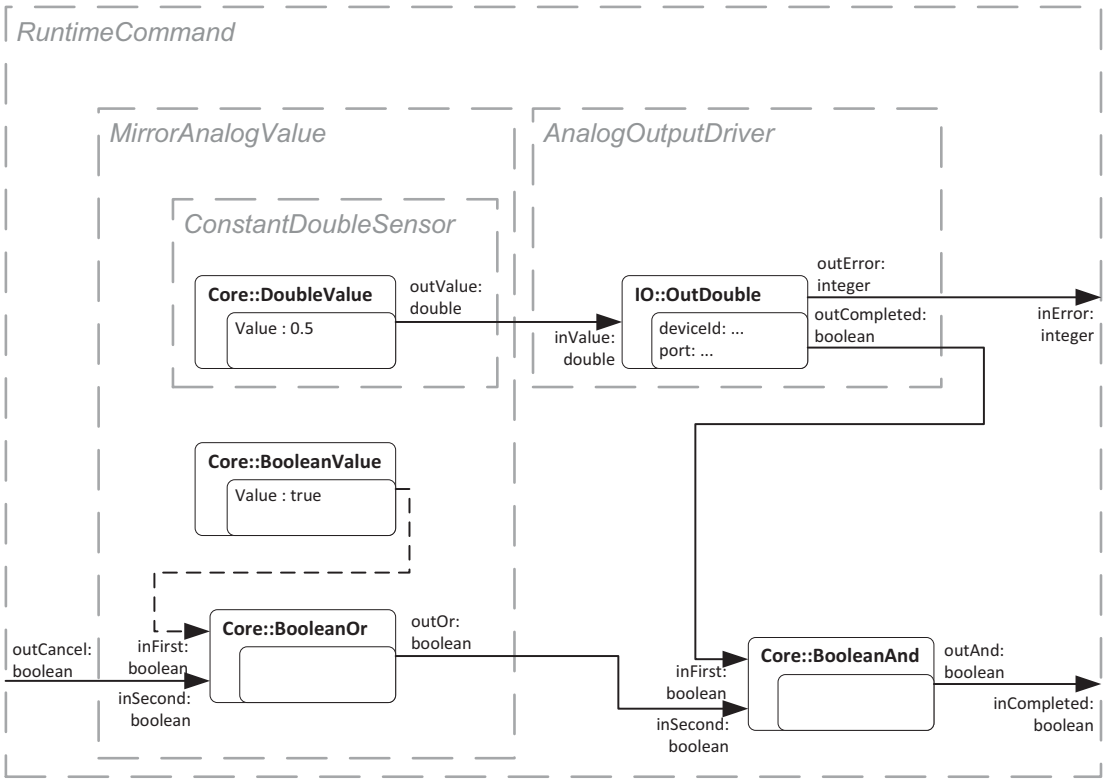

Figure 9.5: Schematic illustration of an RPI primitive net generated for setting an analog output value.

The Robotics API IO extension is a simple, yet comprehensive example for modeling of devices and their operations across all architectural layers. The IO extension furthermore demonstrates that the Robotics API's design allows to offer complex functionality to application developers while still keeping the required support by RoboticsRuntime implementations small. The following I/O specific parts have to be provided by a specific RoboticsRuntime:

- Concrete implementations of the four drivers for analog and digital inputs and outputs, each providing a Sensor that can measure the currently set or read value.
- An implementation of the RPI execution logic for the MirrorAnalogValue and MirrorDigitalValue Actions, respecting cancel signals and indicating errors during execution.

The next section will present two extensions that build on the IO extension to control grippers and screwdrivers based on I/O communication. Interestingly, those extensions require no RoboticsRuntime specific concepts at all.

9.2 An I/O-based Gripper

In the Assembly Cell application, both LWRs are equipped with I/O based tools. The Schunk MEG 50 grippers as well as the Kolver Pluto screwdriver are controlled via digital and analog inputs and provide feedback at some digital and analog outputs. Those devices could be controlled by just including the Robotics API IO extension in applications. However, for intuitive and semantically meaningful control of such devices, additional extensions can be created that provide convenient interfaces to developers. This section presents the design of the Schunk MEG extension and the concrete classes necessary for controlling a Schunk MEG50 device. Some details of the design are omitted in the presented diagrams for clarity of presentation. Thus, this section is not a complete documentation of the Schunk MEG extension, but rather focuses on the way functionality is reused from more generic classes and the combination of I/O control concepts to form more complex gripper control operations.

Structure of the Gripper extension

There exists a huge variety of tools for industrial robots, many of which are developed solely for a concrete automation task. The Robotics API Tool extension, which contributes the package *robotics.tool*, introduces a simple, unified model of robot tools. It is depicted in Fig. 9.6, alongside the Gripper extension, which provides the package *robotics.tool.gripper*. The Gripper extension should serve as a basis for modeling and controlling finger-based robot grippers. Like in the Robot Arm extension (see Chap. 10), *robotics.activity* is employed to provide an interface for the basic operations of such grippers to application developers.

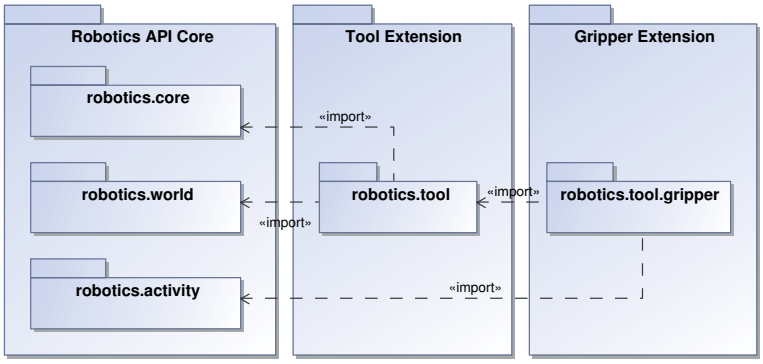

Figure 9.6: Robotics API extensions for robot tools.

Based on the (abstract) Tool and Gripper extensions, support for concrete grippers can be realized. The next section demonstrates this for the Schunk MEG electrical gripper.

Structure of the Schunk MEG extension

For supporting the MEG series of grippers, the Schunk MEG extension has been developed. Its structure is depicted in Fig. 9.7. All MEG grippers rely on control by I/O signals. The Robotics API IO extension already provides support for writing output values and reading input values by the packages *robotics.io* and the respective SoftRobotRCC adapter package *robotics.runtime.softrobotrcc.io*. The Schunk MEG extension contains the package *robotics.tool.gripper.schunk.meg*. This package extends *robotics.tool.gripper* and relies on *robotics.io* for realizing the necessary gripper operations by controlling outputs and inputs in an appropriate way. Thus, the MEG extension itself does not need a runtime environment specific adapter and can be reused with any runtime environment that provides IO support.

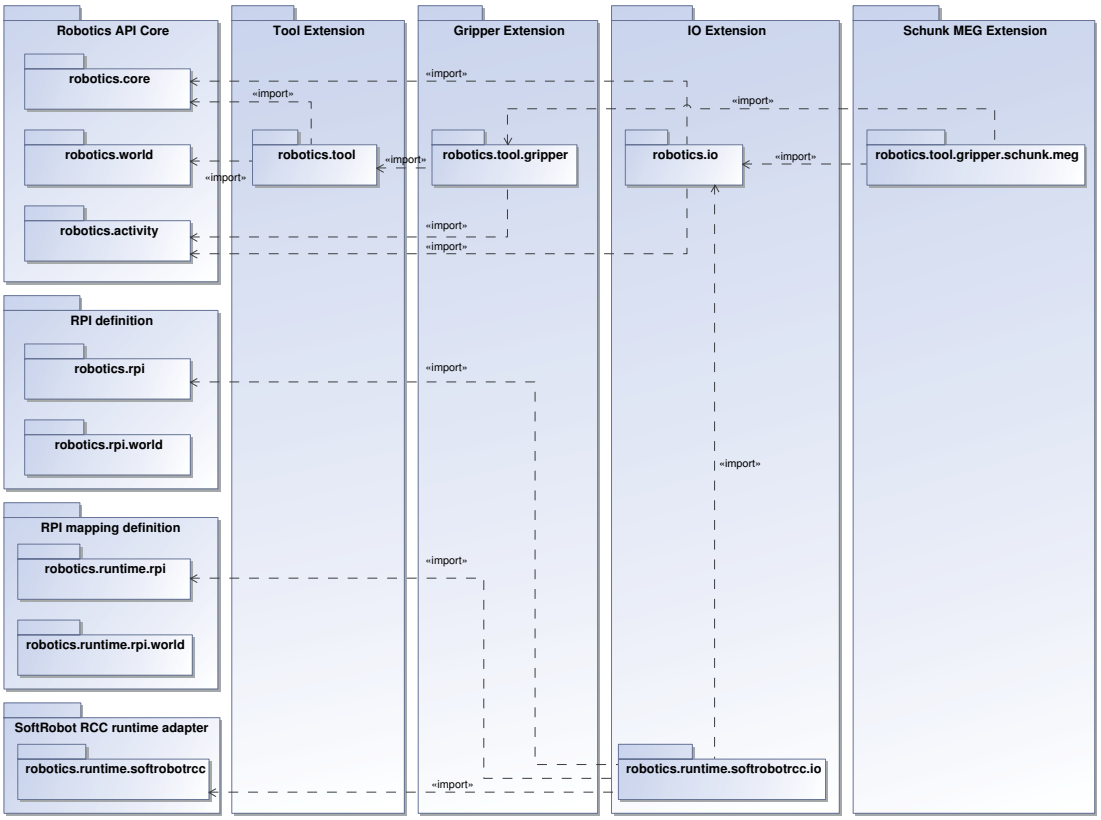

Figure 9.7: Structure of the Robotics API Schunk MEG Gripper extension.

The MEG 50 gripper device

The Schunk MEG 50 gripper device is modeled by the class *MEG50*, which is displayed in Fig. 9.8 along with its relationships to other Robotics API classes. By inheritance, MEG50 reuses functionality from AbstractActuator, AbstractTool and AbstractParal-

lelGripper:

- The functionality for handling of ActuatorParameters and ActuatorInterfaces is provided by *AbstractActuator*.
- *AbstractTool* provides methods and fields for configuring and retrieving characteristic tool Frames (a *base Frame* for modeling the location where a tool is mounted to an actuator and an *effector Frame* that is located at the tool's effector location).
- *AbstractParallelGripper* introduces a general model for describing parallel grippers by fields and methods for providing the maximum recommended workpiece weight, the minimum, current, and maximum opening width of the gripper jaws and further properties (not included in the figure).

The MEG50 class is an example of a Device without a DeviceDriver. As this Device is controlled purely via inputs and outputs, it can rely on the respective I/O Devices and their DeviceDrivers. This implies that the MEG50 implementation (and the implementations of the ActuatorInterfaces it provides) can be used with any RoboticsRuntime, as long as DeviceDrivers for Input and Output Devices are available for this RoboticsRuntime.

The choice to go without DeviceDriver has further implications when it comes to determining the OperationState of a MEG50 instance. The default implementation of the method `getState()` in AbstractDevice relies on the OperationState provided by the Device's DeviceDriver (see Sect. 9.1). For accurately determining the operation state, the MEG50 class redefines `getState()`. The specific implementation of this method calculates the operation state of the MEG device from the OperationStates of all its Input and Output Devices. It takes into account only those inputs and outputs that are mandatory for operating the gripper and determines the gripper's operational state to be the 'weakest' state of the I/O devices. At this point it is important to note that the Robotics API's device model is flexible enough to represent devices that rely on other devices for most of their functionality, and still reuse other aspects of generic device implementations.

Being a concrete class, MEG50 has to implement the methods `validateParameters` and `defineExceptions`. The Gripper extension specifies a set of standard parameters (see Fig. 9.9) which are also applicable for the MEG gripper. The implementation of `validateParameters` validates whether the specified velocity, acceleration and force are within the bounds of the gripper's physical capabilities. The MEG50's implementation of `defineExceptions` merely collects the potential exceptions of all Outputs that are used to control it. The MEG 50 gripper does not provide a particular output that indicates errors. Other grippers with the ability to detect certain errors could be modeled with specific exceptions for those kinds of errors.

Besides the abovementioned methods, the MEG50 class provides further methods for converting process data (e.g. force in Newton) to output signals. This is done by

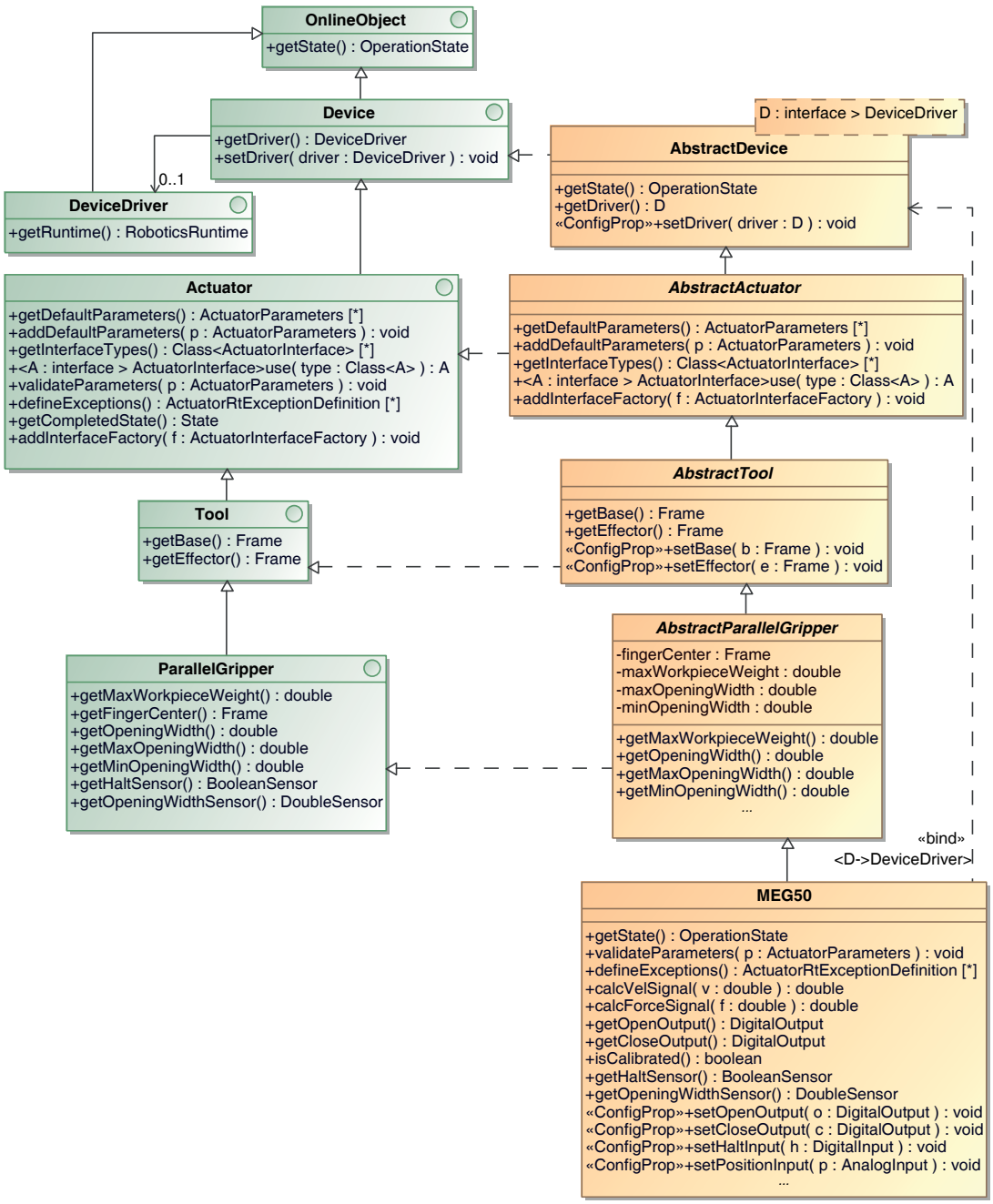

Figure 9.8: Devices and DeviceDrivers modeling the I/O based Schunk MEG50 gripper.

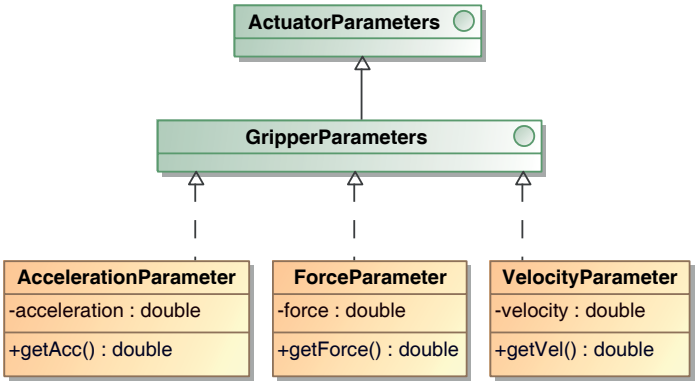

Figure 9.9: ActuatorParameters for use with gripper devices.

the methods `calcVelSignal` and `calcForceSignal`. Other methods retrieve process data (e.g. the opening width of the gripper jaws in meters) from input values, e.g. `getOpeningWidthSensor`. Finally, methods like `setOpenOutput` allow for configuring the Output and Input Devices to use.

ActuatorInterfaces for grippers

For exposing gripper functionality to application developers, a set of ActuatorInterfaces has been designed. Those ActuatorInterfaces are defined in the Gripper extension and are intended to be re-used by various other extensions for concrete gripper models. The Schunk MEG extension provides an appropriate implementation. Fig. 9.10 presents the relevant ActuatorInterfaces and the class *MEGGrippingInterface*, which is the implementation of the interfaces for Schunk MEG grippers. This implementation can be reused for all grippers of the MEG series, as they are controlled in the same way.

The Gripper extension introduces two ActuatorInterfaces: *GrippingInterface* and *StepwiseGrippingInterface*. GrippingInterface is intended to be the simplest possible interface that most gripper devices should support. It provides just the two methods `open()` and `close()`. StepwiseGrippingInterface extends GrippingInterface and adds methods for calibrating and exact positioning of parallel grippers. It is not mandatory that different ActuatorInterface of the same Actuator are connected via an inheritance relationship, but in this case it was semantically adequate: Grippers that support the operations defined by StepwiseGrippingInterface are as well able to support the basic opening and closing operations of GrippingInterface.

The class MEGGrippingInterface implements GrippingInterface as well as StepwiseGrippingInterface for use with Schunk MEG grippers. Depending on the configuration of the gripper Actuator, not all functionality may be usable. For example, if the gripper's analog input for specifying movement steps is not connected, `openStepwise()` and `closeStepwise()` will not work. Implementations of those methods can check if

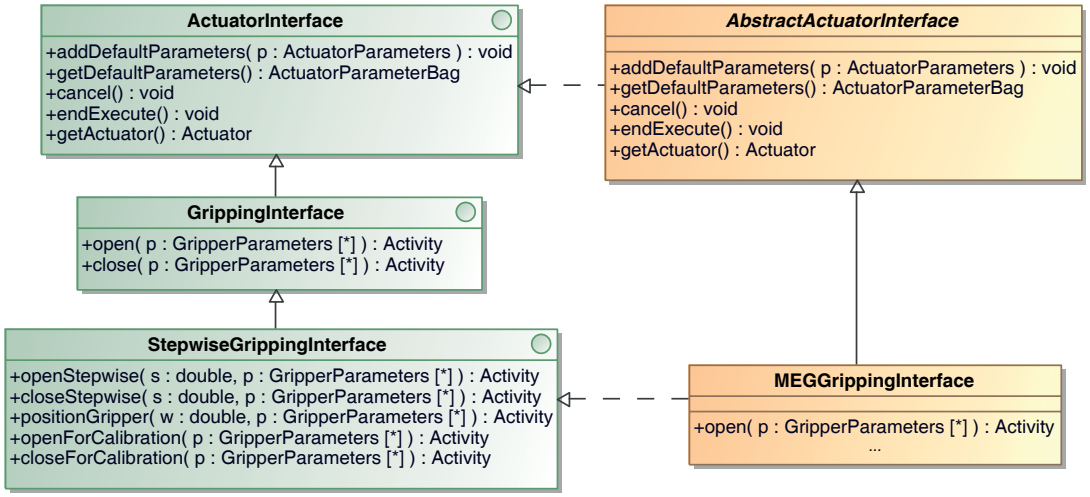

Figure 9.10: ActuatorInterfaces aggregating functionality of parallel grippers like the MEG50.

all required inputs and outputs are connected and otherwise indicate an error e.g. by throwing an exception.

As the MEG50 Actuator is designed to have no ActuatorDriver, MEGGrippingInterface can not rely on a RoboticsRuntime like e.g. AnalogOutputInterfaceImpl did (cf. Fig. 9.4). However, this is not necessary, as the implementation can rely completely on the ActuatorInterfaces of the gripper's inputs and outputs. To demonstrate this, a communication diagram (Fig. 9.11) illustrates the interactions required to create an Activity for closing the gripper. The diagram contains only "first-level" interactions, i.e. further interactions required to process the sent messages are omitted to prevent the figure from becoming too complex. Furthermore, return messages are omitted and the return values are appended to the initial messages with a colon. Finally, some communication partners are assumed to be known a priori. Those are the AnalogOutputInterfaces `velIF` and `forceIF` for communicating goal values for velocity and force to the gripper, and the DigitalOutputInterfaces `closeIF` and `openIF` for triggering closing and opening of the gripper.

The interaction sequence is triggered by the message `close(...)` sent to a MEGGrippingInterface. The implementation first determines the gripper Actuator (for later use) and then the default ActuatorParameterBag `pm`. As no further ActuatorParameters were specified in this case in the initial `close(...)` message, the parameters required for the operation (ForceParameter `f` and VelocityParameter `v`) are determined from `pm`. Both parameters provide values of the force and velocity to use. Those values are process values which have to be converted to device-specific signals before they can be written to outputs. This conversion is performed by sending the messages `calcForceSignal(...)` and `calcVelSignal(...)` to the MEG Actuator. Additionally, the Sensor measuring

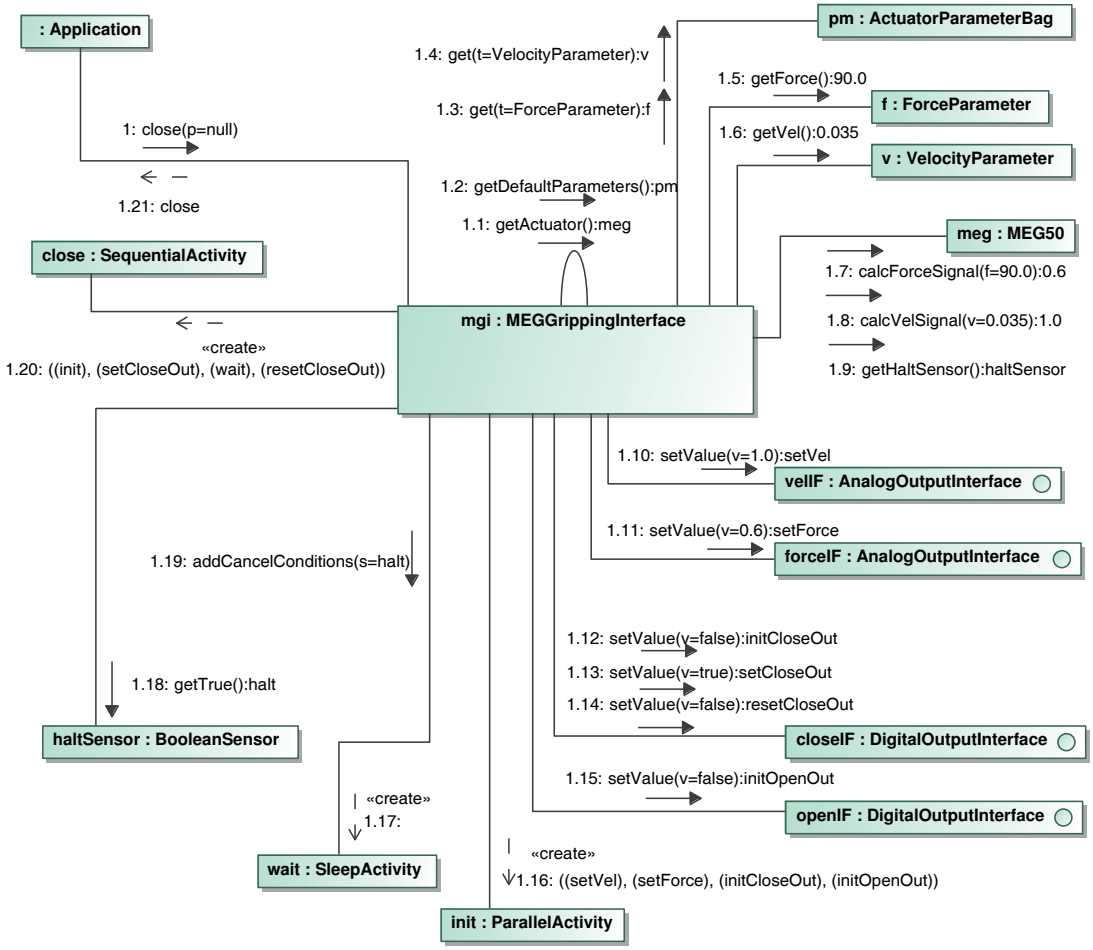

Figure 9.11: Activity construction in MEGGrippingInterface's method close().

gripper halt is retrieved from the gripper Actuator for later use. Subsequently, a series of Activities is created by the gripper's AnalogOutputInterfaces and DigitalOutputInterfaces. The Activities `setVel` and `setForce` communicate the desired velocity and force to the gripper. The Activities `initCloseOut`, `setCloseOut` and `resetCloseOut` set false, true and again false (in this order) to the output connected to the gripper's close input. Combined with the Activity `initOpenOut` (sets false to the output connected to the gripper's open input), the ParallelActivity `init` is then created, which is the first part in a sequence of operations for closing the gripper. The second part is the abovementioned Activity `setCloseOut`, followed by the SleepActivity `wait`. This last Activity is augmented with a canceling condition, which is the State `halt`. This State is derived from the BooleanSensor `haltSensor` that measures the gripper halt. By the message `addCancelCondition(halt)` to the Activity `wait`, this Activity stops waiting when the gripper halts. The final step of the control sequence is the abovementioned

Activity `resetCloseOut`. The sequence is encoded as a SequentialActivity (`close`), which is created and returned as last step of the interaction shown in the figure.

9.3 An I/O-based Screwdriver

This section will give an overview of the Kolver Screwdriver extension which supports the electrical screwdriver used in the Assembly Cell application (cf. Sect. 3.3). As the design is based on the I/O extensions and thus in many ways similar to the Schunk MEG extension, it will not be presented with the same level of detail.

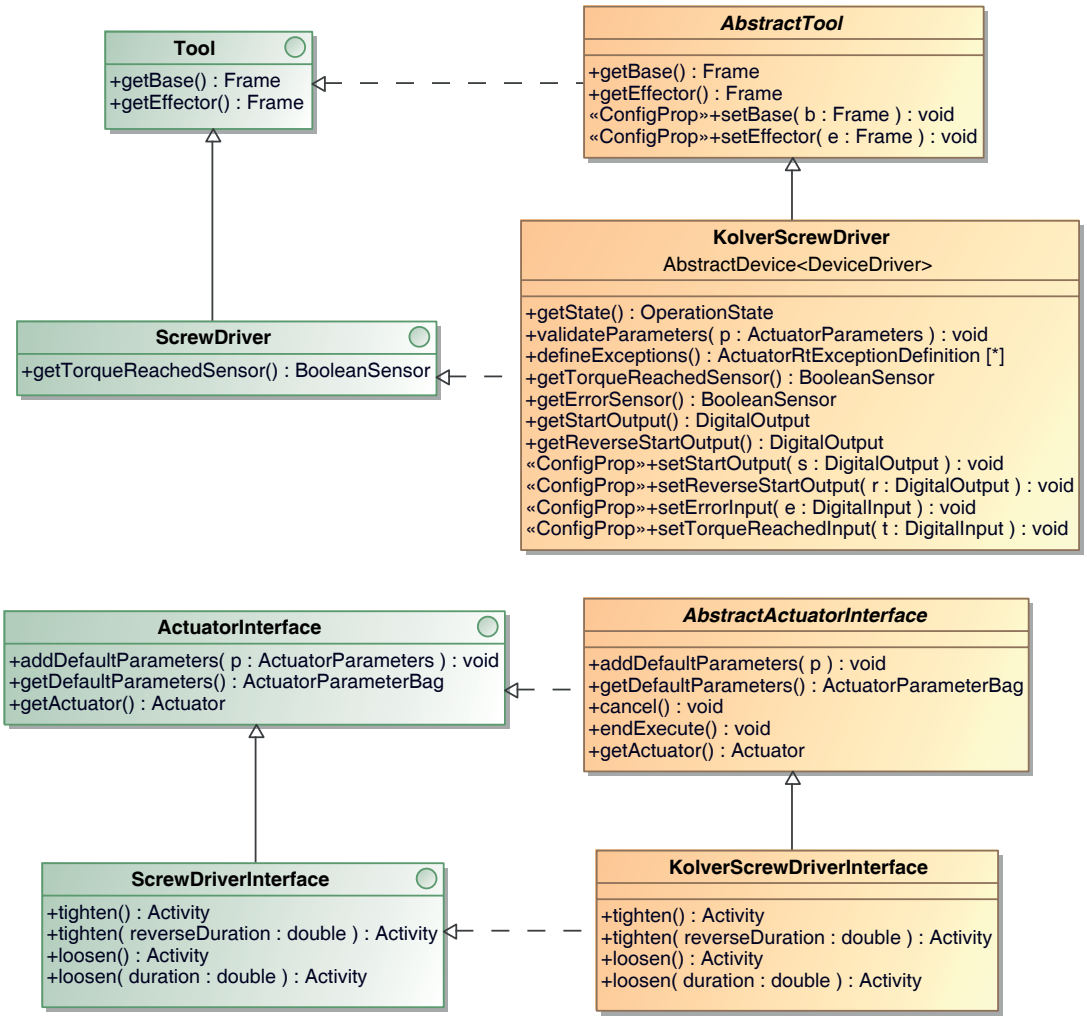

Figure 9.12: Model of the Kolver screwdriver and its ActuatorInterface.

The structure of the Kolver Screwdriver extension is similar to that of the Schunk MEG extension (cf. Fig. 9.7). In the following, the model of the screwdriver device and the

provided ActuatorInterface will be explained. Fig. 9.12 shows the interfaces *ScrewDriver* and *ScrewDriverInterface* that model a generic screwdriver tool and ActuatorInterface, and the concrete classes *KolverScrewDriver* and *KolverScrewDriverInterface*, which are implementations that support the Kolver device. The class KolverScrewDriver is similar to the MEG50 class in that it provides methods for configuring and retrieving the output and input devices that are used to communicate with the screwdriver. Furthermore, it implements abstract methods of its base classes (`getState()`, `validateParameters()` and `defineExceptions()`, which are inherited from superclasses of AbstractTool), similar to MEG50 as well. In contrast to MEG50, the Kolver screwdriver controller provides a dedicated digital output which indicates that an error has occurred. Thus, its implementation of `defineExceptions()` creates an instance of a special *ScrewDriverException* class which indicates a screwdriver-specific error when thrown.

ScrewDriverInterface defines two methods for controlling a screwdriver, `tighten(...)` and `loosen(...)`, both of which are implemented by KolverScrewDriverInterface specifically for the Kolver device. To tighten a screw, the screwdriver's start output is set to a high value. When a predefined timeout has exceeded, or a sensor indicates that the desired torque has been reached (method `getTorqueReachedSensor()`), the start output is set to a low value after a short additional delay. This additional delay is introduced to give the screwdriver time for a short reverse motion. This motion is performed automatically by the screwdriver's controller and is intended to remove restraints between screwdriver and screw in order to allow moving the screwdriver away without problems. The provided variant `tighten(reverseDuration:double)` allows for increasing the delay for reverse motion by the value specified. Loosening a screw works similar. In contrast to tightening, the process is always time-based, as the screwdriver does not provide a sensor that indicates when the screw is fully loosened. The time for loosening can be specified when the method `loosen(duration:double)` is used.

The interfaces ScrewDriver and ScrewDriverInterface are designed to be independent of the concrete screwdriver model to make them reusable for other types of screwdrivers. This enables applications that are developed against these interfaces to be reused with different hardware. Furthermore, implementations of the method `getTorqueReachedSensor()` specified by ScrewDriver may return a null value if no such sensor is available. The KolverScrewDriverInterface is implemented such that it is able to cope with this. Thus, the KolverScrewDriver can also be controlled when the respective input has not been set, and the KolverScrewDriverInterface implementation might be reused for other I/O based screwdrivers even if they do not provide such an input.

9.4 Summary

In this chapter, three Robotics API extensions have been presented. The IO extension provides support for basic inter-device communication via digital and analog inputs and outputs. It defines the Robotics API Devices Input and Output, which require runtime-specific DeviceDrivers in order to be controllable with real-time guarantees. The Devices' implementations reuse various functionality from abstract classes provided

by the Robotics API core packages. The IO extension provides an ActuatorInterface that enables convenient control of outputs according to various patterns. It was demonstrated that those patterns can be realized with a single Action by employing the variation possibilities provided by the Robotics API's Command model. Thus, the number of concepts that have to be implemented in a runtime-specific way can be kept minimal.

The presented Schunk MEG and Kolver Screwdriver extensions build on the IO extension and require no runtime-specific support at all. Instead of delegating responsibility to DeviceDrivers, all functionality can be realized solely based on Input and Output Devices. Despite this, a lot of functionality can still be reused from the same abstract classes employed also in the IO extension. Furthermore, the internal design of the MEG50 and KolverScrewDriver Devices is completely hidden from application developers by the homogenous design of Devices, ActuatorInterfaces and Activities.

Further kinds of robot tools can be integrated in the Robotics API in a similar way. There exists a variety of much more complex I/O based tools, e.g. laser-based cutters, welding guns or glueing pistols (cf. add-on technology packages provided by KUKA for such tools [127, 128, 129]). When using such tools, most operations have to be synchronized to the robot's motion with (guaranteed) accuracies in the range of milliseconds and millimeters. Real-time guarantees are thus inevitable and are ensured by the Robotics API in general and the IO extension in particular.

During the work on this thesis, extensions for further gripper-like devices have been developed. The Schunk WSG extension supports the more advanced WSG series of grippers by Schunk [130], whereas the Schunk SDH extension supports the SDH 3-finger robotic hand by Schunk [131]. Those devices are controlled via proprietary protocols based on the CAN bus [132], and, in case of the SDH, partially by an additional serial RS232 bus. Robotics API Devices that rely on particular DeviceDrivers were created for both grippers. Both Devices provide implementations of GrippingInterface and StepwiseGrippingInterface, besides further ActuatorInterfaces that provide special operations supported by each of the devices. Thus it is even possible to reuse applications that employ solely standard gripping interfaces with such advanced gripping devices.

Chapter 10

Modeling Robot Arms

Summary: Articulated robot arms are the most common kind of industrial robots used. This chapter presents a model of such robot arms and their basic control operations. It furthermore demonstrates how Activities can model complex, context-aware robot operations. The presented concepts are exemplified using the KUKA Lightweight Robot. The design of the robot arm model has been published in [133] and the modeling of robot operations by Activities in [31].

The basic Robotics API packages presented in Sect. 5.1 as well as the adapter to the SoftRobotRCC (Sect. 5.2) form the core structure of the SoftRobot architecture. This structure is very generic and does not even contain the notion of a robot itself. The task to introduce the concept of a robot and its capabilities is left to Robotics API extensions. This section will present the *Robot Arm extension*, which introduces the definition of a robot (arm) as well as its structure and the operations it supports at minimal. The Robot extension is, however, not self contained. It needs to be extended further to allow for controlling *concrete* robots (e.g., a KUKA LWR). Thus, most classes provided by the Robot Arm extension are again abstract and need to be concretized by further extensions.

The structure of the Robot Arm extension is presented in Fig. 10.1. This extension vertically consists of two parts: The *robotics.robot* package, which contains the definition of what a robot is and what its capabilities are, and the *robotics.runtime.softrobotrcc.robot* package, which provides support for controlling robots using the SoftRobotRCC runtime environment. The package *robotics.robot* introduces concepts for describing robot arms (such as joints, links, base and tool frames) and their basic operations (like LIN, PTP and further motion types). The package *robotics.runtime.softrobot.robot* mainly includes new rules for transforming robot-related operations to RPI primitive net fragments. The

RPI-specific parts of the package *robotics.runtime.softrobot.robot* could be extracted to a more generic package *robotics.runtime.rpi.robot*, which would be reusable by other runtime adapters that are compatible with RPI.

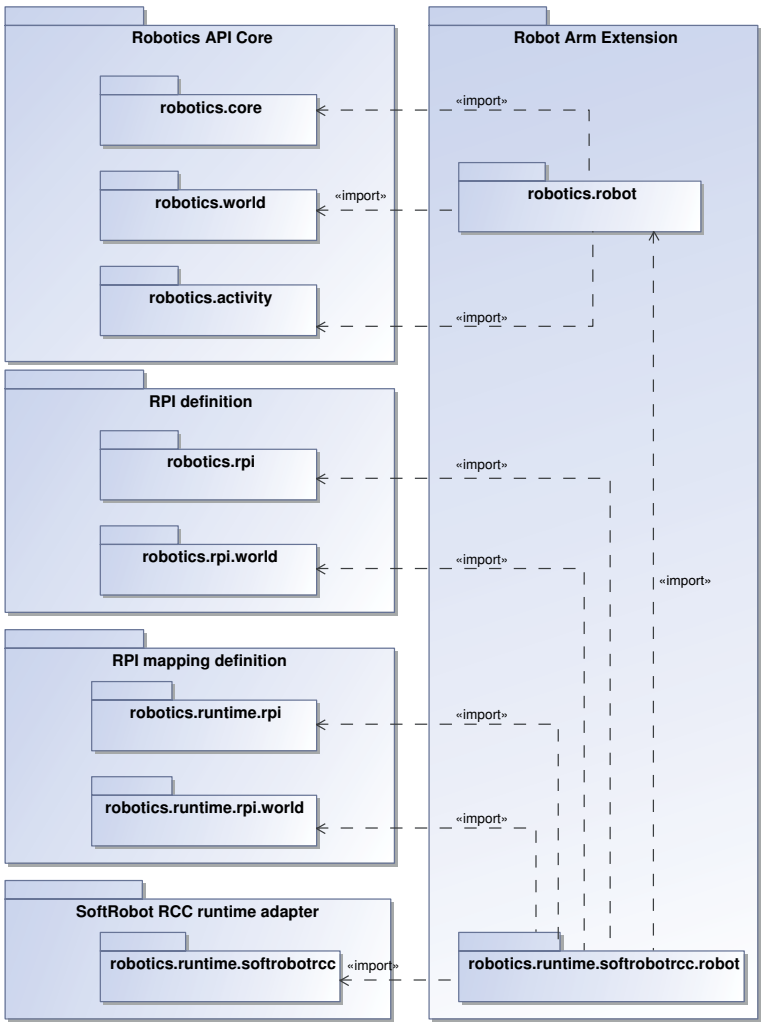

Figure 10.1: Structure of the Robotics API Robot Arm extension.

As illustrated in Fig. 10.1, *robotics.robot* is independent of a concrete runtime environment, as it has neither dependencies to any of the RPI-related packages, nor to *robotics.runtime.softrobotrcc*. These dependencies are required only by the more specific package *robotics.runtime.softrobotrcc.robot*. *robotics.robot* imports *robotics.activity*. This is necessary, as the package contributes Activities for operating robots.

Sect. 10.1 introduces the basic model of robot arms as Robotics API Actuators. The model of motions, concepts for motion planning and an approach for continuous execution of motions based on Activities is presented in Sect. 10.2. Parameters for controlling

motion execution are subject of Sect. 10.3. Sect. 10.4 presents the design of Actuator-Interfaces that provide application developers convenient means to operate all kinds of robot arms. The model of the KUKA LWR illustrates the application of all those concepts to a concrete, sophisticated hardware device (Sect. 10.5). Sect. 10.6 demonstrates how dynamic teams of real-time synchronized robots can be simply composed from the concepts presented before. The last section sums up the results.

10.1 A General Model of Serial Robot Arms

The Robotics API Robot Arm extension was designed for controlling robot arms that are commonly used in industrial environments. It supports modeling arms with an arbitrary number of joints that are ordered in a serial structure. The Robot Arm extension consist of a generic part (independent of a concrete RoboticsRuntime) and a part that is specific for controlling robot arms with the SoftRobotRCC. The Robot Arm extension is abstract in the sense that it does not contain models of concrete robot types. Instead, the main design goal of this extension is to provide generic functionality that is reusable for a lot of different robot arm models. Thus, the effort required for integrating support for new robot arms should be kept minimal. This section gives an overview about the design of the Robot Arm extension and illustrates how the concerns are separated to achieve the aforementioned goal.

The Robot Arm extension defines a set of interfaces that model robot arms as part of the Robotics API's device model (see Sect. 6.1). The diagram in Fig. 10.2 depicts those interfaces, the relationships between them and the relationships to the generic device model. The interface *RobotArm* represents robot arms in the sense explained above, consisting of an arbitrary number of joints, which are modeled by the interface *Joint*. Implementations of RobotArm as well as Joint may rely on a RoboticsRuntime-specific driver (*RobotArmDriver* and *JointDriver*) and delegate part of their functionality to those drivers. This is however not mandatory; implementations of RobotArm and Joint may also work without such drivers (e.g., consider I/O based robot arms or joints).

Each implementation of Joint has to implement the following methods that provide properties of the joint:

- `getFixedFrame()` provides a Frame that is the geometric representation of the fixed part of the joint w.r.t. the previous joint in the robot structure,
- `getMovingFrame()` provides a Frame that is the geometric representation of the moving part of the joint w.r.t. its fixed Frame,
- `getHomePosition()` returns the (rotatory or translatory) value that is defined as home position of the joint.

Implementations have to ensure that fixed Frame and moving Frame are connected by one or more Relations that model the dynamic behavior of the joint. For example, the

model of the LWR's joints locates both Frames on the joint's rotation axis at the point where the respective links are mounted (cf. Fig. 7.4). The Joint implementation used to model the LWR employs a DynamicConnection that derives the transformation between fixed and moving Frame from the joint's commanded position Sensor. This Sensor is one of four DoubleSensors each Joint implementation has to provide:

- `getCommandedPositionSensor()` provides a Sensor that measures the position the joint is currently commanded to be, i.e. its goal position,
- `getMeasuredPositionSensor()` provides a Sensor that measures the joint's actual position,
- `getCommandedVelocitySensor()` provides a Sensor that measures the velocity the joint is currently commanded to have, i.e. its goal velocity,
- `getMeasuredVelocitySensor()` provides a Sensor that measures the joint's actual velocity.

The same four methods also constitute the JointDriver interface. Thus, if a Joint implementation employs a JointDriver, it can delegate the task of creating those Sensors to the driver. In this case, only those four methods have to be implemented specifically for each concrete RoboticsRuntime. All other methods can be implemented independently of the RoboticsRuntime used.

Finally, Joint defines two methods that can be used to query kinematic properties:

- `getTransSensor(p : DoubleSensor)` returns a Sensor that measures the transformation of the moving Frame relative to the fixed Frame given a DoubleSensor that measures the joint's position,
- `getVelSensor(p : DoubleSensor)` returns a Sensor that measures the Cartesian velocity of the moving Frame relative to the fixed Frame given a DoubleSensor that measures the joint's velocity.

These methods can be used for two purposes: When given a DoubleSensor that measures the joint's goal position or velocity (or its current position or velocity), the current Cartesian position or velocity of the joint's moving Frame relative to its fixed Frame can be measured. This approach is used e.g. by the LWR's Joint implementation to create an appropriate Relation between fixed and moving Frame. The second possible use is calculating the forward kinematics function for hypothetic joint positions, which is important e.g. when planning motions. In this case, DoubleSensors to use as input for the calculation can be constructed from constant values. The Robotics API core provides mechanisms for this (see Sect. 6.2).

The RobotArm interface is designed as an aggregation of Joints, which can be retrieved by the method `getJoints()`. Additionally, each RobotArm has to provide a *base Frame*

(`getBase()`) and a *flange Frame* (`getFlange()`). The base Frame should be connected to the Joint structure in a way that it can be used as a reference Frame, e.g. for defining motion goals. The flange Frame should be located at the point of the robot arm where end-effectors can be mounted. The position of base and flange Frames in the LWR's implementation can be seen in Fig. 7.4.

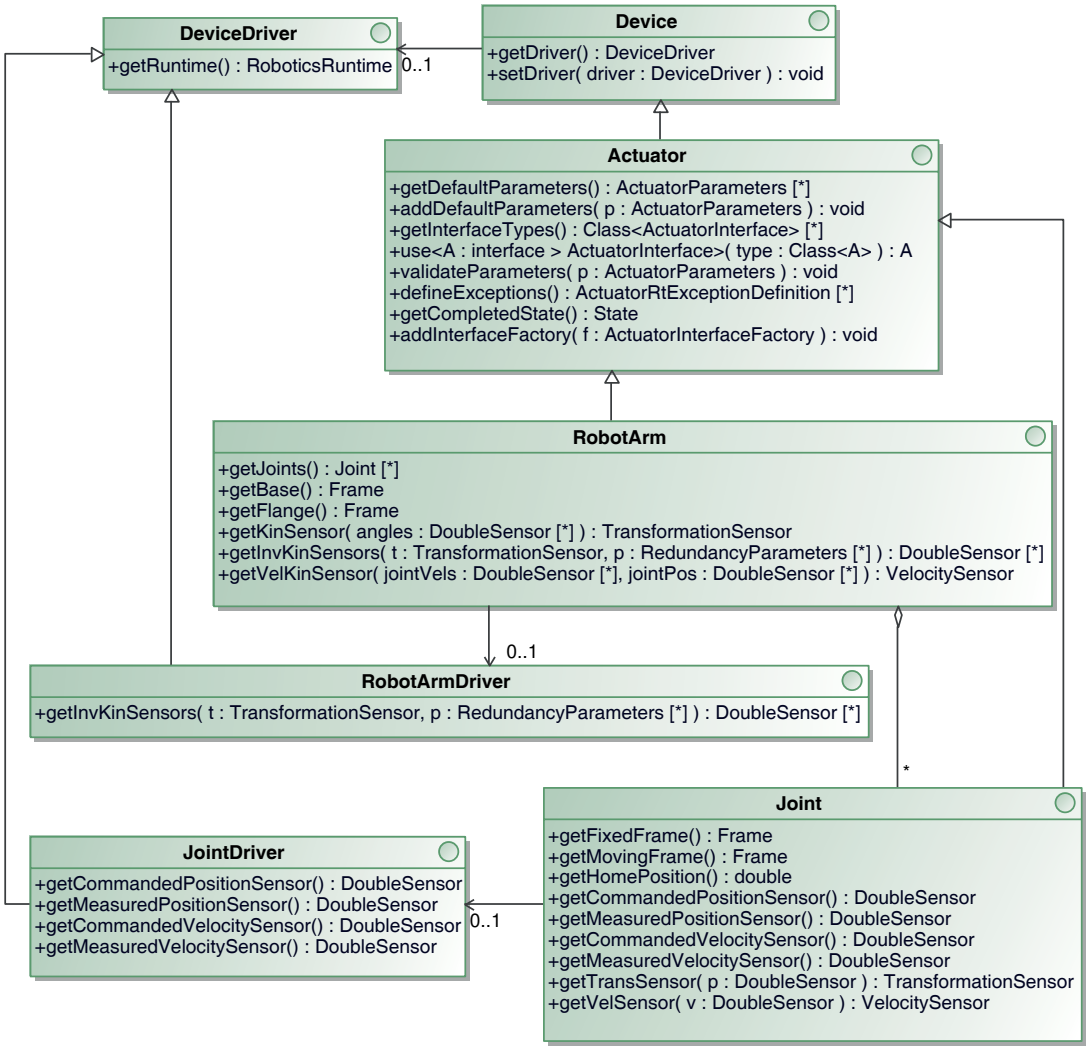

Figure 10.2: Interfaces forming the model of a robot arm.

RobotArms furthermore have to implement a set of methods to provide real-time information about kinematic properties:

- `getKinSensor(angles)` provides a Sensor that measures the transformation of the flange Frame relative to the base Frame, given a set of DoubleSensors *angles*

that measure the positions of all joints. Thus, the provided Sensor effectively calculates the forward position kinematics function.

- `getInvKinSensors(t, p)` returns a set of Sensors that provide feasible positions for each Joint, given a TransformationSensor t that specifies the intended position of the flange Frame relative to the base Frame. Additionally, *RedundancyParameters* p can be supplied that specify strategies to resolve kinematic redundancies.

- `getVelKinSensor(jointPos, jointVels)` provides a Sensor that measures the velocity of the flange Frame relative to the base Frame, given a set of DoubleSensors *jointPos* that measure the position of all joints, and a set of DoubleSensors *jointVels* that measure the velocity of all joints.

RedundancyParameters are a subclass of ActuatorParameters, i.e. the RedundancyParameters interface is a specialization of the ActuatorParameters interface. RedundancyParameters will be explained in Sect. 10.3.

Similar to Joint, RobotArm may delegate some functionality to an optional *RobotArmDriver*. Implementations of this driver however only have to implement a single method, which is `getInvKinSensors(...)`. All other methods defined in RobotArm can be implemented in a generic way by combining functionality provided by its Joints appropriately. This will be sketched in the following.

The Robot Arm extension provides abstract implementations of the Joint and RobotArm interfaces, named *AbstractJoint* and *AbstractRobotArm* (see Fig. 10.3). In contrast to the interfaces they implement, AbstractJoint and AbstractRobotArm require a driver to work properly. Both abstract classes implement nearly all methods required by the interfaces. However, the method's implementations merely delegate to the respective driver's methods whenever possible. AbstractJoint and AbstractRobotArm declare their drivers as mandatory configuration properties. They perform validation of ActuatorParameters that affect them by overriding `validateParameters(...)` and declare additional exceptions via overriding `defineExceptions()`. AbstractRobotArm furthermore defines an array of Joints as mandatory configuration property (not shown in Fig. 10.3).

Two methods defined by Joint can not be implemented by AbstractJoint in a generic way. Those are `getTransSensor(...)` and `getVelSensor(...)`, which are marked abstract in AbstractJoint in Fig. 10.3. As defined above, the methods are responsible for providing Sensors that measure transformation and velocity of the Joint's moving Frame relative to its fixed Frame. The implementation of those Sensors depends mainly on the type of joint used: moving revolute joints induces rotation between moving and fixed Frame, moving prismatic joints leads to changes in the translation between the Frames. The Robot Arm extension provides classes for modeling revolute and prismatic joints (not shown in Fig. 10.3) which implement the two methods appropriately.

To form a complete robot arm, the moving and fixed Frames of successive Joints have to be connected appropriately. In other words, the geometric properties of the links connecting each pair of physical joints have to be modeled in order to operate the robot

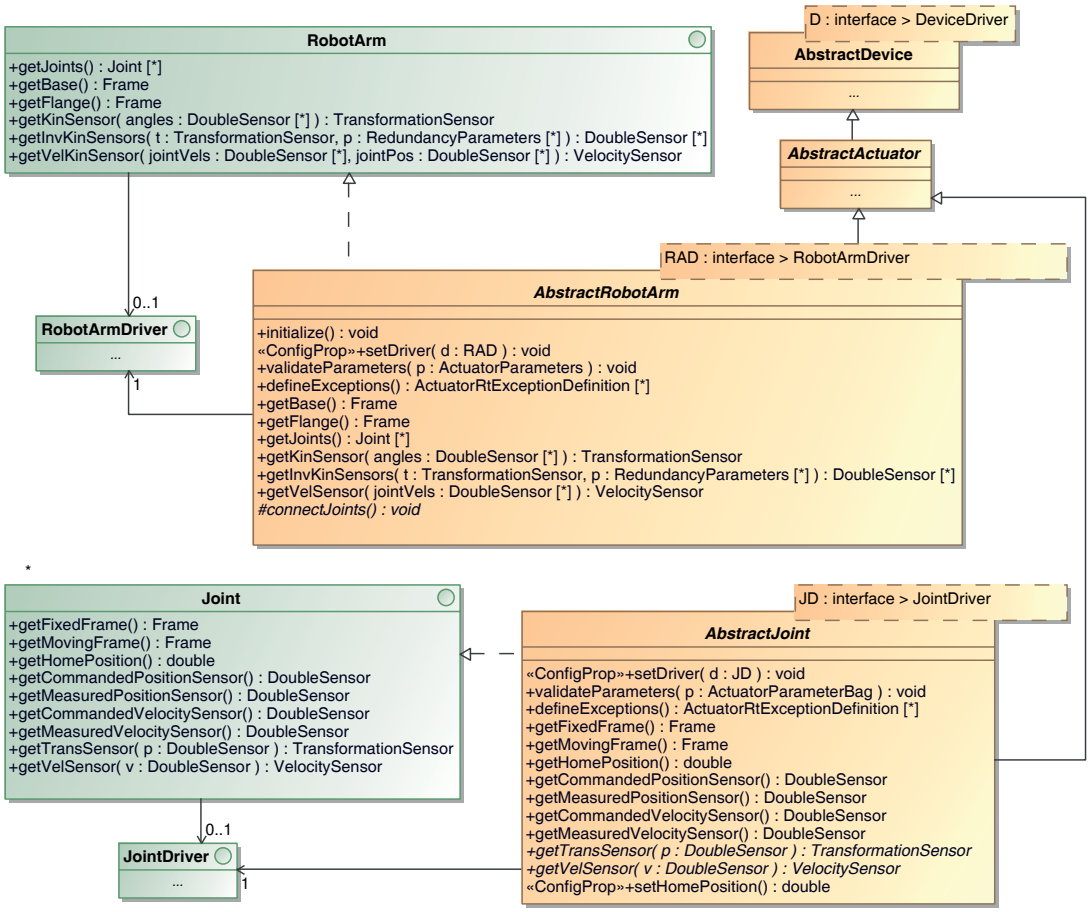

Figure 10.3: Base classes for driver-based robot arms.

arm. Each concrete RobotArm implementation is responsible for performing this task. AbstractRobotArm employs the Template Method Pattern (cf. [86], pp. 325) by defining the abstract method `connectJoints()`, which is called by `initialize()`, i.e. in the initialization phase of the RoboticsObject. The contract for implementations of the abstract method is defined as follows. It is guaranteed that base and flange Frame as well as all Joints have been set and initialized already. In turn, the method has to connect each pair of successive Joints by establishing an appropriate Relation between moving Frame of one Joint and fixed Frame of the successive Joint. Additionally, the base Frame has to be connected to the fixed Frame of the first Joint and the moving Frame of the last Joint has to be connected to the flange Frame. In a previous section, an example of the resulting object structure in case of an LWR has been shown (cf. Fig. 7.5).

AbstractRobotArm implements `getKinSensor(...)` and `getVelSensor(...)` in a generic (runtime-independent) way. This is easily possible by relying on the methods

provided by each Joint and the RobotArm's geometric model (cf. Sect. 7.4). The generic implementation of `getKinSensor(angles : DoubleSensor[])` determines, for each Joint, the Sensor measuring the transformation between fixed and moving Frame for the given joint angle using the Joint's method `getTransSensor(...)`. To determine the Sensor measuring the (static) transformation between a Joint's moving Frame and the successive Joint's fixed Frame, the moving Frame's method `getRelationSensorTo(...)` is called with the fixed Frame as argument. In the same way, the Sensors measuring the displacements of the RobotArm's base Frame and its flange Frame are determined, respectively. By multiplying all TransformationSensors in the correct order (using the method `TransformationSensor#multiply(...)`), the desired TransformationSensor for the complete RobotArm structure is created. In a similar manner, a generic implementation of `getVelSensor(...)` can be realized.

The aforementioned implementations create complex structures of composed geometric Sensors. This is another example of how the flexible and compositional structure of the Robotics API's Sensor model can be employed to provide generic functionality. However, when such complex Sensor structures are used in Commands, they might lead to high computational effort during real-time Command execution. This problem can be alleviated by overriding the abovementioned methods in subclasses of AbstractRobotArm and providing specialized Sensors with a simpler structure. In this case, the drivers for each type of Joint and RobotArm should be enhanced to provide such Sensors as well. RoboticsRuntime implementations can then provide native, efficient support for calculating the Sensor values.

The general model of robot arms in the Robotics API Robot Arm extension is geared mainly towards position control of those devices, as this is the most commonly used control mode of today's industrial robots. The RobotArm interface does e.g. not require implementations to provide inverse velocity kinematics or other more advanced kinematic functions. Nevertheless, it is possible to offer Actions and Activities for velocity-based motion of robot arms and map those to position control by integrating over the goal velocities. Furthermore, this kind of model is able to support the Lightweight Robot's compliance features, as they have an orthogonal character. For other kinds of control modes, e.g. pure/hybrid force control or torque control, the model can easily be extended.

10.2 Modeling and Planning Motions

The central capability of any industrial robot arm is its ability to move the end-effector in a defined manner. The movement of the robot arm structure itself is usually less important, as long as it doesn't collide with itself or other objects. In most cases, the crucial aspect is the motion of the end-effector that is mounted to a robot, like a gripper, a welding torch or any other tool that is appropriate to perform the robot systems' task. This section will introduce the model of motions in the Robotics API's Robot Arm extension as well as the way certain motions are planned before their execution.

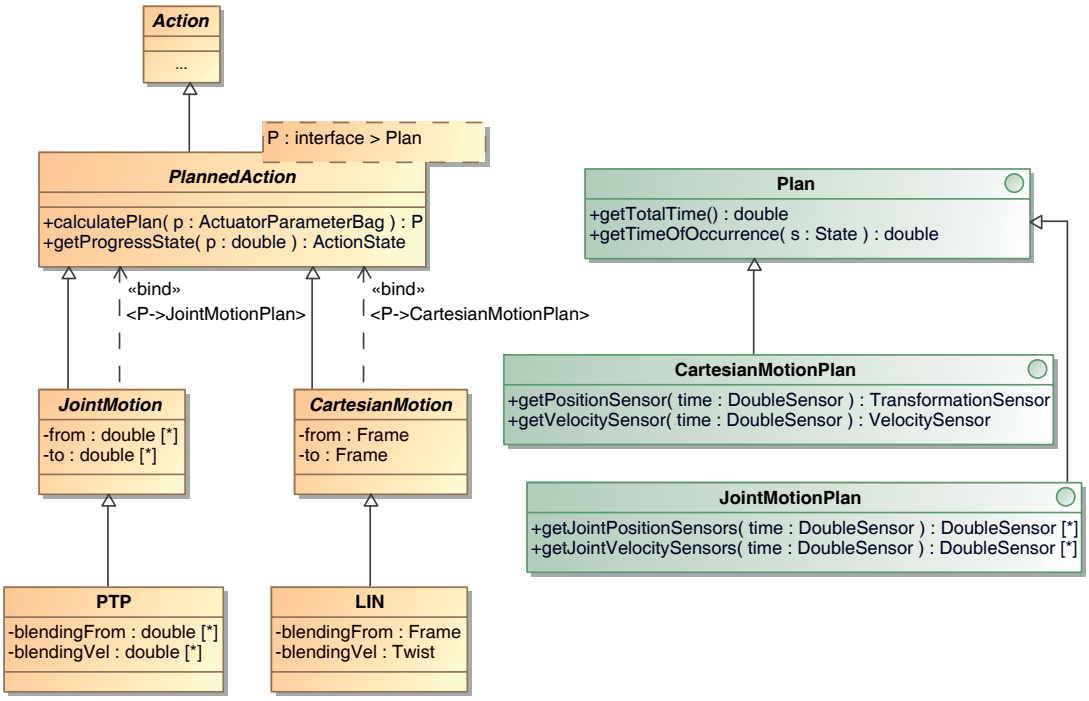

Figure 10.4: Actions for basic motions defined by the Robot Arm extension.

Motions are realized as special Actions by the Robot Arm extension. The Robotics API core makes a difference between *Action* and *PlannedAction*, which is relevant in this context:

- An *Action* does not have to provide any information about its execution modalities, in particular not about its duration. Thus, Actions may also run indefinitely.
- In contrast, *PlannedAction* is a special kind of Action which has a finite execution time and has to provide States that resemble the execution progress. It has to be able to calculate a *Plan* of its execution.

The separation between PlannedAction and Plan makes sense, as Plans can be reused and combined in other Actions. For example, CartesianBezierPlan can be used for planning bezier spline movements, but also for planning blending between subsequent motions of a certain type.

The Robot Arm extension employs PlannedActions and Plans to realize planned motion Actions for RobotArms (cf. Fig. 10.4). JointMotion models joint space motions, whereas CartesianMotion models Cartesian space motions. Both provide particular types of plans: JointMotionPlan and CartesianMotionPlan. The algorithms for motion planning

are not in the scope of this work. Instead, this section will focus on the way plans are evaluated when the execution of motion Actions is prepared.

Fig. 10.4 also contains two concrete motion Actions: PTP, modeling a standard joint space point-to-point motion, and LIN, modeling a linear motion in Cartesian space. Both Actions inherit information about the desired motion's start and goal. Both are capable of moving a robot arm from two initial configurations: 1) After a halt, when the robot is not moving relative to the specified start configuration, and 2) when blending from a previously executed motion, which implies that the robot arm is in motion. To handle the second case, both Actions contain additional parameters that represent the robot arm's position, velocity and acceleration (in joint and Cartesian space, respectively) at the moment blending is started. Note that a concrete instance of PTP or LIN has a fixed set of parameters and is thus only able to provide a Plan for one of the abovementioned initial configurations.

The Robot Arm extension defines special ActuatorInterfaces for RobotArms. Some of those use PTP and LIN Actions to construct Activities for point-to-point and linear motions. The different types of ActuatorInterfaces are presented in Sect. 10.4. The rest of this section is dedicated to illustrating how Activity scheduling, the meta-data stored in Activities and the information provided by Plans can be combined to create motion Activities that allow for blending in between them.

Assume that a RobotArm ρ is currently executing a motion Activity a. This Activity provides meta data M_a about its execution and has started a Command C_a on the RoboticsRuntime that is controlling the RobotArm. In this situation, a subsequent motion Activity s is being scheduled by the ActivityScheduler. This Activity may now create a Command C_s that is able to take control over ρ. Additionally, it has to provide appropriate meta data M_s, i.e. ActivityProperties describing ρ's behavior during the execution of C_s. This situation and the interactions between s and a are depicted in Fig. 10.5.

s is allowed to inspect M_a for planning its own execution. In an ideal case, a can provide meaningful data (i.e., position, velocity and acceleration) about ρ's state in two situations:

1. At a previously defined point in time (relative to the start time of C_a) at which motion blending is allowed (*blending case*).
2. After C_a has terminated or has started maintaining, i.e. the main operation of a is finished, but ρ might still be moving to follow a's motion goal. (*non-blending case*)

Note that in the first case, motion blending is allowed only at exactly this point in time.

The meta data provided by a for the first situation is denoted by M_a^b and for the second situation by M_a^r. Both data is essential, as the scheduling of Activities is not guaranteed to be performed within a defined timeframe (see Sect. 8.2). This implies that s has to

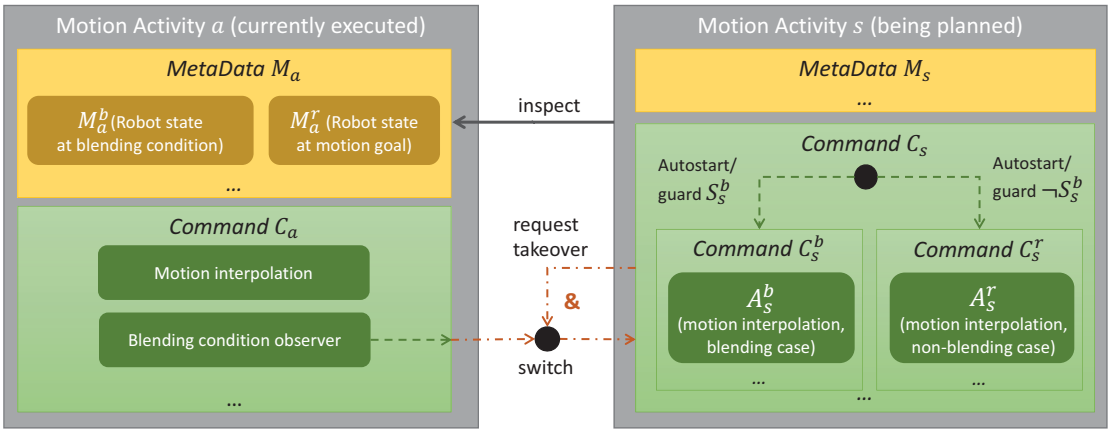

Figure 10.5: Schematic illustration of Activity-based motion blending. Solid arrows indicate interaction inside the Activity layer, purely dotted arrows indicate interaction inside the Command layer, and dot/line arrows indicate interaction between Commands in a RoboticsRuntime.

consider both situations when creating its Command C_s, as C_s might be started too late, i.e. at a time where C_a does not longer allow to be taken over. Thus, C_s is created in the following way:

- C_s is defined to be a TransactionCommand.
- A RuntimeCommand C_s^b is created that contains a motion Action A_s^b which is able to take over ρ in the blending case. The position and velocity of ρ in this situation are taken from M_a^b, allowing A_s^b to create an appropriate Plan.
- In the same manner, a RuntimeCommand C_s^r is created. This Command's Action A_s^r is configured to start moving ρ after C_a has taken it to its motion goal. Thus, the Action's *from* property as well as its *blendingFrom* property are defined to be the motion goal specified by M_a^r. Its *blendingVel* property is defined to be zero.
- C_s^b is added to C_s as child command which is automatically started together with its parent. The automatic start is, however, guarded by a State S_s^b. This State is active if and only if ρ is in the situation described by M_a^b. To define the State, the Sensors provided by ρ measuring its position and velocity can be employed.
- C_s^r is added to C_s as auto-started child command as well. In this case, its starting is guarded by the negation of State S_s^b. This way, C_s^b and C_s^r are effectively mutually exclusive.

s signals the ActivityScheduler that its Command C_s is able to take control of ρ. The ActivityScheduler will thus schedule C_s as successor of C_a. The RCC is then expected to

terminate C_a and instantaneously switch to execution of C_s, if C_a indicates that it can be taken over. The implementation of C_a can do this by activating its *takeoverAllowed* State. Remember that C_a may only do this if ρ is in the state described by M_a^b. Further, note that C_s has to integrate the same logic to be blendable by subsequent motions.

Finally, s itself has to provide meta data M_s for planning of subsequent motions. Assume that for s a blending condition has been defined as well, which implies that M_s is expected to provide information about the same two situations (blending and non-blending case) like M_a. The main challenge for the construction of C_s is to ensure that when the defined blending condition is met, ρ will be in exactly the state described by M_s, regardless of which Action (A_s^b or A_s^r) is actually executed. Usually the execution time of A_s^b will be smaller compared to that of A_s^r[1]. It is the responsibility of the *Plans* of A_s^b and A_s^r to interpret blending conditions, which are represented by special States, according to the Actions' parameterization. Furthermore, s has to construct C_s such that it allows to be taken over regardless which execution path was chosen.

As demonstrated in this section, the design of motion Actions in the RobotArm extension and their integration into the Activity concept allows for planning (continuous) motion execution completely inside the Robotics API. If Plans provide accurate information and Commands for motions are carefully designed, safe continuous motion execution can then be mapped to pure Command switching in the Robot Control Core.

10.3 Parameters for Motions

Application developers need means to parameterize the execution of motion (and other) operations in order to adapt them to various scenarios. In this section, motion specific as well as some general parameters are presented. All such parameters are modeled by implementations of the *ActuatorParameters* interface and can thus be used with all operations in a generic way. Though, not all parameters might be applicable to all kinds of operations.

General parameters

An important general parameter that should be obeyed by all operations of all Actuators (if feasible) is the *OverrideParameter*. It is intended to be a general modifier for execution speed of operations. The OverrideParameter can be configured to act *relative* or *absolute*. In the former case, it interacts with the global RoboticsRuntime override (see Sect. 6.7), and the resulting speed override factor is determined by multiplying the factor specified by the OverrideParameter with the global override factor.

BlendingParameter is another general parameter, in the sense that it applies to all path motions. It specifies the relative motion progress at which blending to a subsequent motion is allowed.

[1] Reducing cycle times is one of the motivations why motion blending is even offered.

Joint parameters

To configure joint-specific dynamic parameters for a robot arm, *JointParameters* may be used. An instance of JointParameters specifies, for a single joint, the minimum and maximum allowed positions as well as the minimum and maximum values allowed for absolute velocity and acceleration. For parameterizing all joints of a RobotArm, a set of JointParameters can be aggregated in an instance of *MultiJointParameters*.

Cartesian parameters

Similar to dynamic parameters for joints, such may also be specified for Cartesian operations. *CartesianParameters* allow for specifying maximum linear and angular velocities and accelerations with respect to the Cartesian frame which is being moved by the operation. To specify this frame, *MotionCenterParameter* should be used. The Frame encapsulated in a MotionCenterParameter is employed in all path movements to describe the Frame that is actually moving in the requested way, e.g. the robot's flange or the tip of an end-effector.

Kinematic parameters

Position and orientation in Cartesian space do not form a unique specification of a robot configuration, as the calculation of the inverse kinematics function will in most cases yield multiple results. Intuitively, the same pose in space can be achieved by different joint configurations (e.g., most spherical wrists can simply be 'turned around', resulting in an identical pose, but different joint configurations). To resolve such ambiguities in a generic way and independent of a concrete robot model, *HintJointParameter* allows to specify a 'hint' value for each joint. Motion operations are expected to choose the kinematic solution for which the resulting joint configuration is closest to the hint values. This proximity should be calculated for each joint separately.

Summary

The parameters introduced in this section can be specified for each single operation. However, RobotArm implementations should provide reasonable default parameters of each kind to relieve application developers from having to supply all parameters every time. Note that, as introduced in Sect. 6.5, the set of default parameters can be specified per Actuator and per ActuatorInterface to change behavior in a wider scope of an application.

10.4 Reusable ActuatorInterfaces for Robot Arms

By construction, robot arms are very flexible and are able to perform arbitrarily complex motions. This section gives examples of motion patterns that have proven to be useful for many scenarios. The Robot Arm extension defines a set of ActuatorInterfaces which provide Activities for those patterns. All those ActuatorInterfaces are modeled

as interfaces. They are designed to be usable with various kinds of robot arms. Some robot arms may not be able to execute all kinds of Activities, e.g. when they do not have enough degrees of freedom as of their mechanical construction. Some of those interfaces can however be reused for other kinds of devices. Examples will be given in this section. The Robot Arm extension also provides implementations of all those interfaces which construct Activities that implement the respective operations for all kinds of RobotArms.

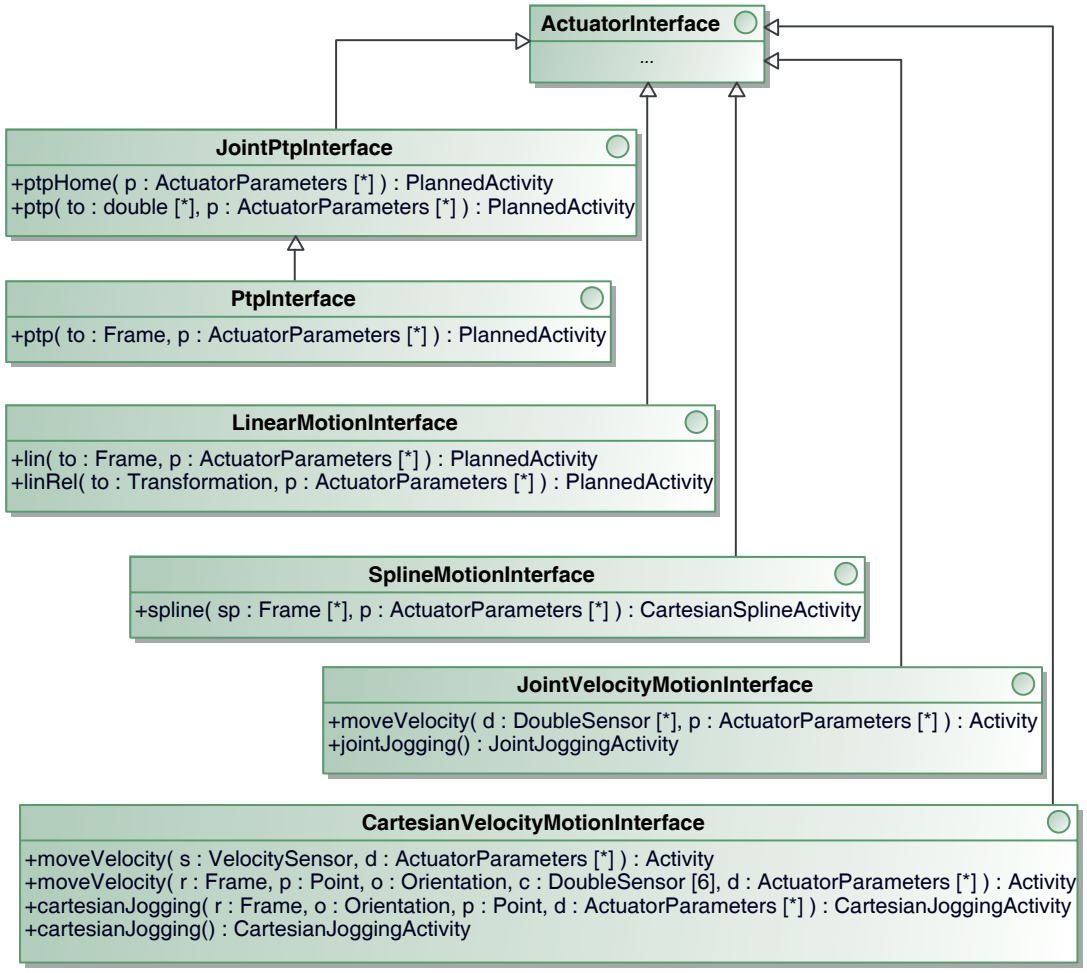

Figure 10.6: ActuatorInterfaces provided by the Robot Arm extension for various kinds of motions.

Fig. 10.6 gives an overview of those ActuatorInterfaces and the operations they provide. In the following, LinearMotionInterface and CartesianVelocityMotionInterface as well as the Activities they use will be presented in detail, as these ActuatorInterfaces are frequently used in the application examples presented in this thesis. The operations

provided by other ActuatorInterfaces listed above will be summarized in the second half of this section.

LinearMotionInterface and LinearMotionActivity

Linear Cartesian motions are used very frequently in robotic applications, like the PortaitBot and Assembly Cell applications presented in this work. LinearMotionInterface provides the methods `lin(...)` and `linRel(...)` for performing linear motions to absolute and relative targets, respectively. An absolute target is specified by a Frame, whereas a relative target is specified using a Transformation that is interpreted relative to the Actuator's position when the Activity is started. Both methods return a *PlannedActivity*, which is a special type of Activity that can provide States describing its execution progress. PlannedActivity is designed as an interface. Linear motions are implemented by a class called *LinearMotionActivity*, which inherits basic functionality from AbstractActivity and additionally implements the PlannedActivity interface. LinearMotionActivity assumes that it controls a single RobotArm and implements the abstract method `prepare(...)` such that the Activity's Command lets the RobotArm's motion center execute a linear Cartesian motion. It is able to take over a previous Activity that controls the same RobotArm and to blend from the motion performed by this Activity into the new linear motion. Conversely, it provides ActivityProperties that describe the RobotArm's state at a predefined blending condition, which enables a successive motion to take over the LinearMotionActivity.

Another important property of LinearMotionActivity is its ability to move a RobotArm's motion center Frame to targets that are moving relative to the RobotArm's base Frame. This is to a large part delegated to the runtime-specific implementation of the LIN Actions used. This implementation has to consider the relative movement of its 'to' Frame during motion interpolation. However, this is only feasible if the 'from' Frame is moving uniformly, i.e. if there exists a static way between 'from' and 'to' Frame. Otherwise, the Plan calculated by the LIN motion may become inaccurate. LinearMotionActivity on the one hand ensures that this precondition is met when creating a Command. This is achieved by checking for a static way between start and target Frames of the motion and aborting execution otherwise. On the other hand, LinearMotionActivity performs active maintaining of the RobotArm's target position, which involves controlling the RobotArm so that it's motion center Frame follows the movements of the target frame relative to the robot base even if the actual linear motion is yet completed. This allows subsequent Activities to deal with such dynamic targets in the same way like LinearMotionActivity. For following movements of the target Frame, a HoldCartesianPosition Action is used. This simple Action merely employs a RelationSensor (cf. Sect. 7.4) to measure the current position of the target Frame relative to the RobotArm's base Frame and uses the resulting values as interpolation values for the RobotArm's motion center Frame. This ensures that the motion center Frame follows the target Frame as closely as possible. It can, however, also lead to errors if the target Frame is moving faster than the RobotArm is able to follow. This can be detected by the RobotArm and usually leads to the Activity execution being aborted with an exception.

In the following, the implementation of LinearMotionActivity will be outlined based on the structure of the resulting Command. This structure is representative also for the Activities that implement point-to-point and spline movements. Fig. 10.7 shows the structure of the Command generated by an instance of LinearMotionActivity that is able to take over a preceding Activity. Note that the figure does not show all relevant instances and also not all relationships between instances to preserve clarity. The important elements not contained in the figure will be explained later in this section.

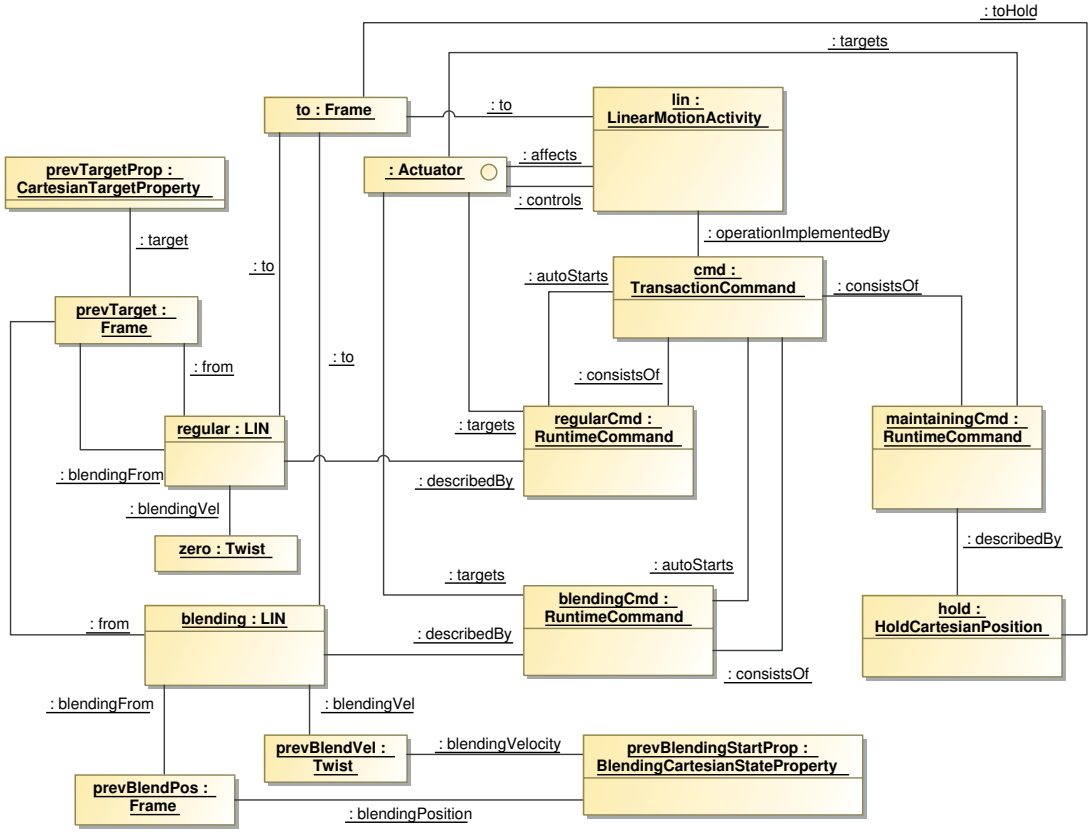

Figure 10.7: Instance diagram illustrating the structure of the Command created by LinearMotionActivity.

The figure contains the LinearMotionActivity instance *lin* (top right part), the Command *cmd* it has created (below *lin*) and the three RuntimeCommands *regularCmd*, *blendingCmd* and *maintainingCmd* that are children of *cmd*. The former two RuntimeCommands realize the two alternative execution paths (blending case and non-blending case) that are necessary to cope with all possible situations when blending over the previous motion, as described in Sect. 10.2. The RuntimeCommand *regularCmd* is responsible for performing a linear motion without blending over a previous motion (non-blending case). Its associated LIN Action *regular* is therefore parameterized with

zero blending velocity by the Twist instance *zero*. Its 'from' Frame is associated with the target Frame *prevTarget* of the previous motion, as is its 'blendingFrom' Frame. In the depicted case, the previous motion has provided information about its target Frame by the CartesianTargetProperty *prevTargetProp* (top left in the figure). The 'to' Frame of Action *regular* is associated with the Frame *to* supplied as target to the LinearMotionActivity (i.e. the Frame that was supplied to the LinearMotionInterface's method `lin(...)`). The second RuntimeCommand, `blendingCmd`, is responsible for a linear motion that is blending from the preceding motion at a predefined condition (blending case). Thus, the 'blendingFrom' and 'blendingVel' properties of the LIN Action *blending* associated with *blendingCmd* are taken from the BlendingCartesianStateProperty *prevBlendingStartProp* supplied by the preceding Activity. The 'from' property of *blending* is associated with *prevTarget* as well, and the Actions 'to' Frame is associated with *to* like in the non-blending case.

The two RuntimeCommands *regularCmd* and *blendingCmd* are added as conditional auto-started Commands (cf. Sect. 6.4) to the TransactionCommand *cmd*. Their starting is guarded by a BooleanSensor *regularCondition* (not shown in the figure). This BooleanSensor is constructed from a RelationSensor (cf. Sect. 7.4) that measures the distance between the RobotArm's motion center Frame and *prevTarget*. By extracting translation and rotation component sensors from the RelationSensor, performing adequate comparisons operations on the resulting Sensors and re-combining those Sensors with boolean Sensor operations, the BooleanSensor *regularCondition* is constructed. This Sensor is then used as condition to auto-start *regularCmd*, and the complementary (negated) Sensor is used as auto-start condition for *blendingCmd*.

As mentioned above, LinearMotionActivity performs active maintaining to keep the RobotArm at the target Frame, even if this Frame is moving. This is achieved by *cmd*'s third child Command, which is the RuntimeCommand *maintainingCmd*. This Command uses the HoldCartesianPosition Action *hold* to let the RobotArm hold its motion center Frame at the position specified by Frame *to*. The *maintainingCmd* is started as soon as *regularCmd* or *blendingCmd* have finished, so that there is no delay in following the target. The top-level Command *cmd* allows to be taken over as soon as *maintainingCmd* is active so that successive Activities can take over control of the RobotArm. Furthermore, *maintainingCmd*'s Active State is used to take the LinearMotionActivity to status Maintaining (cf. Sect. 8.3), which allows successive Activities to be scheduled (cf. Sect. 8.2).

The LinearMotionActivity *lin* provides ActivityProperties (not shown in the figure) to enable pre-planning by successive Activities. It constructs a CartesianTargetProperty that specifies the Activity's Frame *to* as target. Furthermore, it constructs a BlendingCartesianStateProperty that provides information about the Actuator's position and velocity at the predefined blending condition. For this to work, the set of ActuatorParameters supplied to the Activity has to contain a BlendingParameter that specifies the condition for blending, i.e. the motion progress at which blending is allowed. Note that blending is allowed only when the motion has reached this progress *exactly*. With this blending condition, all necessary information about blending position and blending

velocity can be extracted from one of the LIN Actions' Plan. As both LINs are expected to behave the same (at least after blending over from a preceding motion has finished), the choice which LIN's Plan to use is arbitrary. With this information, the BlendingCartesianStateProperty can be constructed. The complete TransactionCommand *cmd* is allowed to be taken over if either one of the LIN Actions allows takeover, or as soon as *maintainingCommand* is active. In any case, the RobotArm will be in one of the states described by the two ActivityProperties. Cancel signals are forwarded by *cmd* to all of its child Commands, which gives application developers the possibility to stop control of the RobotArm by LinearMotionActivity in any execution state.

Fig. 10.7 assumes that the preceding Activity provided a CartesianTargetProperty as well as a BlendingCartesianStateProperty. This might not be the case for some Activities. In particular, if no BlendingParameter was specified for a preceding path motion Activity, the latter property will not be provided. In this case, the LinearMotionActivity that is being planned may simply skip the construction of *blendingCmd* and indicate that it is not able to take over control of the RobotArm. The ActivityScheduler will then wait for the preceding Activity to terminate or reach Maintaining status before scheduling the new Activity. The preceding Activity is expected to keep the RobotArm's motion center at the target specified in its CartesianTargetProperty. If the preceding Activity does not provide a CartesianTargetProperty, the situation is more difficult, as neither of the LIN Actions can be pre-planned (both require the previous target Frame provided by this property). In this case, the LinearMotionActivity may throw a particular kind of exception as soon as this situation is detected by its implementation of `prepare(...)`. In this case, the ActivityScheduler will await complete termination of the preceding Activity and retry scheduling of the erroneous Activity (cf. Sect. 8.2). LinearMotionActivity detects the case that the preceding Activity is no longer running (nor maintaining) and that pre-planning is thus not necessary. In this case, LinearMotionActivity simply retrieves the current position of the RobotArm and uses it as start position. This is feasible, as the RobotArm will not move if no Activity is controlling it anymore.

CartesianVelocityMotionInterface and CartesianJoggingActivity

Velocity-based motions proved to be helpful when a robot's motion should be guarded or guided by sensors, such as in the Tangible Teleoperation and Assembly Cell applications. Different means for velocity-based motions are provided by *CartesianVelocityMotionInterface*, which is depicted in Fig. 10.8 together with some concepts that are used to realize appropriate Activities. In the following, the implementation of two methods of CartesianVelocityMotionInterface will be described, as they are of particular importance for the realization of the example applications.

CartesianVelocityMotionInterface's method `moveVelocity(s, d)` is the most basic way to create an Activity for velocity-based motion. It creates a *MoveVelocityActivity*, which relies on a *MoveCartesianVelocityAction*. The supplied VelocitySensor `s` is passed to MoveVelocityActivity, which uses it to parameterize the MoveCartesianVelocityAction. Note that the signature of the `moveVelocity(...)` method does not specify which

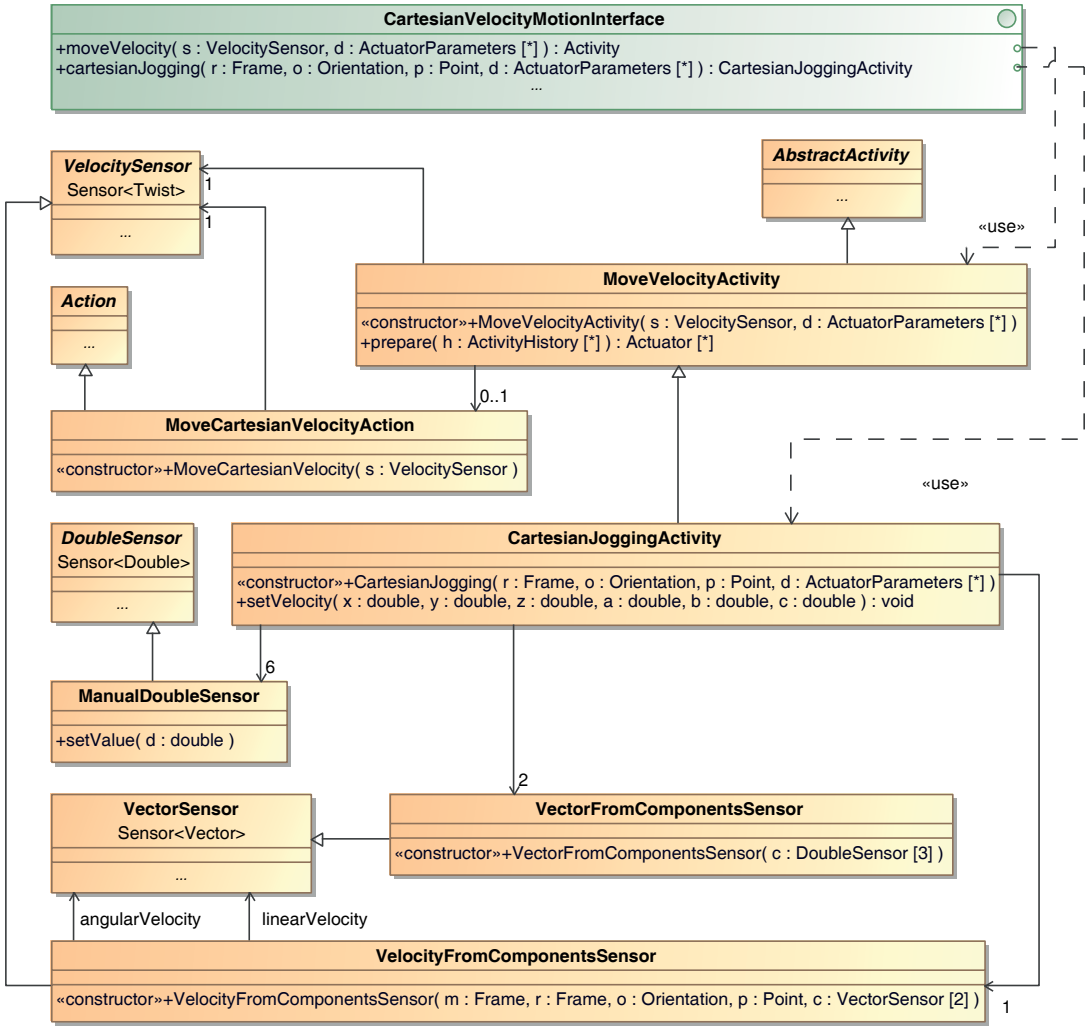

Figure 10.8: Velocity-based Activities provided by the RobotArm extension.

exact kind of Activity is returned. MoveVelocityActivity does not provide further useful methods and is thus hidden from application developers.

MoveCartesianVelocityAction has to be implemented by the RoboticsRuntime used and is expected to move the Actuator's motion center Frame with the velocity that is measured by the supplied VelocitySensor. The SoftRobotRuntime maps this Action to an online trajectory generation algorithm that is able to generate a smooth velocity profile. The SoftRobotRCC integrates a real-time capable online trajectory generation algorithm developed by Kroeger et al. [23, 134] and provides it as an RPI primitive. This allows for real-time critical motions based on VelocitySensors.

CartesianVelocityMotionInterface provides several overloaded versions of the `moveVelocity(...)` method to provide application developers a variety of operations for different use cases. For example, there exists a variant of the method that accepts six DoubleSensors which specify each component of the target velocity. The implementation of this variant employs *VectorFromComponentsSensor* and *VelocityFromComponentsSensor* to create a VelocitySensor. In this case, the mathematical parameters for interpreting the velocity correctly have to be specified explicitly, similar to the CartesianJoggingActivity that is explained below. A further variant accepts six double values (and mathematical velocity parameters) to allow developers to specify constant target velocities. It employs *ConstantDoubleSensor* to create DoubleSensors with the specified constant values and then combines those like described above. Further variants exist that implicitely choose common values for the mathematical velocity parameters.

A further pair of methods provided by CartesianVelocityMotionInterface (see Fig. 10.8) are `cartesianJogging(r, o, p, d)` and a variant which requires only ActuatorParameters d and chooses a common combination of values for `r`, `o` and `p`. The semantics of those parameters was introduced in Sect. 7.1 and will be briefly recapitulated. Note that the same semantics also applies to overloads of `moveVelocity(...)` that require this set of parameters.

- `m` is the *moving Frame*, i.e. the Frame whose velocity is controlled. This Frame is not specified as method parameter. Instead, the motion center Frame, which is either specified explicitly as part of the ActuatorParameters d or taken from the default parameters of the ActuatorInterface, is used as moving Frame.
- `r` is the *reference Frame*, i.e. the Frame relative to which `m` is moving.
- `o` is the *Orientation* which serves as basis for interpreting velocity values.
- `p` is the *pivot Point*, i.e. the center of rotation. When a non-zero angular velocity is specified, the moving Frame will rotate around this point.

In the abovementioned method variant `cartesianJogging(d)`, the RobotArm's base Frame is used as reference Frame and this Frame's Orientation is used as jogging Orientation. The configured motion center Frame's Point is used as pivot Point. This represents a commonly used parameterization for Cartesian jogging.

All variants of `cartesianJogging(...)` create and return a *CartesianJoggingActivity*. This Activity is a more specific MoveVelocityActivity that is designed for applications that need to 'manually' control a RobotArm's velocity, like e.g. the Tangible Teleoperation application. By inheritance, it reuses the implementation of `prepare(...)` of MoveVelocityActivity. Additionally, it provides a method `setVelocity(...)` that can be used in applications to update the RobotArm's target velocity at any time. To realize this, CartesianJoggingActivity constructs a VelocityFromComponentsSensor from two VectorFromComponentsSensors. The parameters for interpreting the velocity values (m, r, o, p in the constructor of VelocityFromComponentsSensor) are taken from the CartesianJoggingActivity's own parameter set. The VectorFromComponentSensors are constructed from a triple of ManualDoubleSensors each. As described in Sect. 6.2, ManualDoubleSensor is a DoubleSensor whose values can be set and updated by applications. The implementation of `setVelocity(...)` just updates the values of all ManualDoubleSensors, which are automatically propagated to the Command that is executed by the RCC. CartesianJoggingActivity furthermore tracks its own ActivityStatus to give feedback to applications. For example, when `setVelocity(...)` is called before the Activity has been started or after it has terminated (regularly or due to some error), an exception is thrown which can be handled appropriately by applications.

All presented Activities for velocity-based motions allow to be taken over at any time. However, they do not provide any ActivityProperties describing the state of the RobotArm due to their sensor-based character. They simply cannot predict the precise state of the Actuator at any time instant. Thus, Activities willing to take over those velocity-based Activities have to be able to cope with any state the RobotArm is in. This applies to MoveVelocityActivity itself, as the online trajectory generator used is able to generate robot trajectories instantaneously, no matter what the initial state is. Thus, this Activity's `prepare(...)` implementation always indicates that it can take over previous Activities. However, application developers have to be aware that the resulting motion will not have a predefined Cartesian path, which makes it unsuitable for some scenarios.

Further ActuatorInterfaces

This section will give a brief overview about further ActuatorInterface provided by the RobotArm extension. It will not go into detail about their implementation, which is similar to those ActuatorInterfaces presented in the last two sections.

JointPtpInterface provides basic point-to-point motions specified in joint space, i.e. by assigning each joint a target rotation angle. *PtpInterface* extends JointPtpInterface and adds a method that allows to specify a motion goal in Cartesian space by supplying a Frame. Note that the motion is still performed in joint space. The method's implementation uses the robot's inverse kinematics function to calculate a possible joint configuration for the given Cartesian position and then relies on the methods provided by JointPtpInterface. The two ActuatorInterfaces were separated to reduce coupling between them: JointPtpInterface could be used for any Actuator consisting of one or multiple joints, whereas PtpInterface is only usable for RobotArms, as it relies on methods that calculate

the inverse kinematics function. *SplineMotionInterface* provides Activities for Cartesian spline motions. The spline paths are modeled by a sequence of control points, specified as Frames. Spline motions are used in applications to perform motions along complex shaped workpieces. *JointVelocityMotionInterface* provides methods for velocity-based motion of robot arms in joint space. Similar to Cartesian velocity based motions, either motions controlled by arbitrary sensors or manual jogging motions are possible.

All presented ActuatorInterface are designed to be usable with any RobotArm. Robot arms with less than six joints or different mechanical structures might not be able to execute all operations exactly like defined. In particular, motions with a fixed Cartesian path (like linear and spline motions) might pose problems. In this case, it might be useful to allow such arms to violate some path constraints while fulfilling others. For example, experiments with the KUKA youBot arm (which consists of merely five joints) have shown that linear motions are useful in many scenarios, even though the end-effector orientation cannot be maintained as specified due to the kinematic constraints.

Many of the presented ActuatorInterfaces can furthermore be reused for Actuators other than RobotArms. For example, JointPtpInterface and JointVelocityMotionInterface are usable for all Actuators that incorporate joints, like e.g. linear axes. In this case, even the RobotArm-specific implementations of those interfaces might be usable. The ActuatorInterfaces for Cartesian motions can e.g. also be used for mobile robot platforms, which either have to project the specified paths on their configuration space or reject paths that do not completely lie inside this space. It might be possible to reuse the RobotArm-specific implementations in this case as well. Furthermore, some of those interfaces could be used to control aggregates of multiple Actuators. For example, neither the KUKA youBot's arm nor its omnidirectional robot base are able to execute path motions without sacrificing some of the linear or angular constraints. However, the combination of both Actuators provides even more degrees of freedom than required for Cartesian motions. Appropriate implementations of ActuatorInterfaces that control the *aggregate* of both Actuators can divide the tasks into sub-tasks for arm and base and synchronize them appropriately.

10.5 The LWR as a Special Robot Arm

The KUKA Lightweight Robot can be seamlessly integrated in the model of robot arms defined by the Robot Arm extension. To fully support the KUKA LWR, the LWR extension provides two new packages, *robotics.robot.kuka.lwr* and *robotics.runtime.softrobotrcc.robot.kuka.lwr*. Besides defining the physical structure of the LWR, the former package also introduces new types of operations supported by the LWR. The latter package, *robotics.runtime.softrobotrcc.robot.kuka.lwr*, provides SoftRobotRCC-specific support for real-time execution of the aforementioned operations. The first part of this section will introduce the way the LWR is modeled as a Robotics API Actuator, while the second part goes into details about specific operations that are supported by the LWR.

The LWR Actuator

By construction, the LWR is a robot arm consisting of seven revolute joints. Thus, it can be modeled with the interfaces provided by the Robot Arm extension in a straightforward way. Furthermore, functionality provided by AbstractRobotArm and AbstractJoint can be reused, as the assumptions made by these abstract implementations (see Sect. 10.1) also apply to the LWR. However, each joint is equipped with an additional sensor that measures the external torque applied to the joint. Furthermore, the robot's internal controllers can estimate the force and torque applied to the end-effector from the torques measured at each joint. These two properties are reflected in the interfaces and classes provided by the Robotics API LWR extension as illustrated in Fig. 10.9.

The LWR extension provides the classes *LWRJoint* and *LWR* as concrete implementations of the interfaces RobotArm and Joint defined by the Robot Arm extension. LWRJoint builds on RevoluteJoint and AbstractJoint, which are provided by the Robot Arm extension as well. RevoluteJoint provides complete implementations of `getTransSensor(...)` and `getVelSensor(...)` (see Sect. 10.1) by constructing appropriate Sensors from single components, where the dynamic components are taken from Sensors provided by JointDriver.

The responsibility of LWRJoint is merely to check ActuatorParameters for their validity in its implementation of `validateParameters(...)`, and to provide a DoubleSensor to application developers which measures the current external torque applied to the joint. The respective Sensor is retrieved from LWRJoint's driver, which consequently is of type *LWRJointDriver*. This driver extends the generic JointDriver by the capability to provide such a Sensor.

LWR extends AbstractRobotArm and is responsible for ActuatorParameters validation as well. In addition, it defines further LWR-specific exceptions. This in particular includes exceptions that relate to the LWR's internal controller modes and switching between them, which is explained later. LWR implements the method `connectJoints()` to set up the LWR's structure appropriately. Furthermore, it redefines the method `getJoints(...)` to give developers access to the correct type of joints. Finally, a set of methods (e.g. `getForceXSensor()`) is provided to access the LWR's sensors that measure the estimated Cartesian forces and torques at the end-effector. The construction of these Sensors is, however, delegated to *LWRDriver*, which extends the generic RobotArmDriver by appropriate methods to retrieve these LWR-specific Sensors.

Modeling the Lightweight Robot as a Robotics API Actuator is straightforward and requires minor effort. Most of the presented concepts are runtime-independent and are thus contained in the package *robotics.robot.kuka.lwr*. The exceptions are *SoftRobotLWRJointDriver* and *SoftRobotLWRDriver*, which are implementations of LWRJointDriver and LWRDriver for the SoftRobotRuntime and thus reside in *robotics.runtime.softrobot.robot.kuka.lwr*. A comparison of the complexity of the runtime-independent and runtime-specific parts of the LWR implementation is presented in Chap. 11.

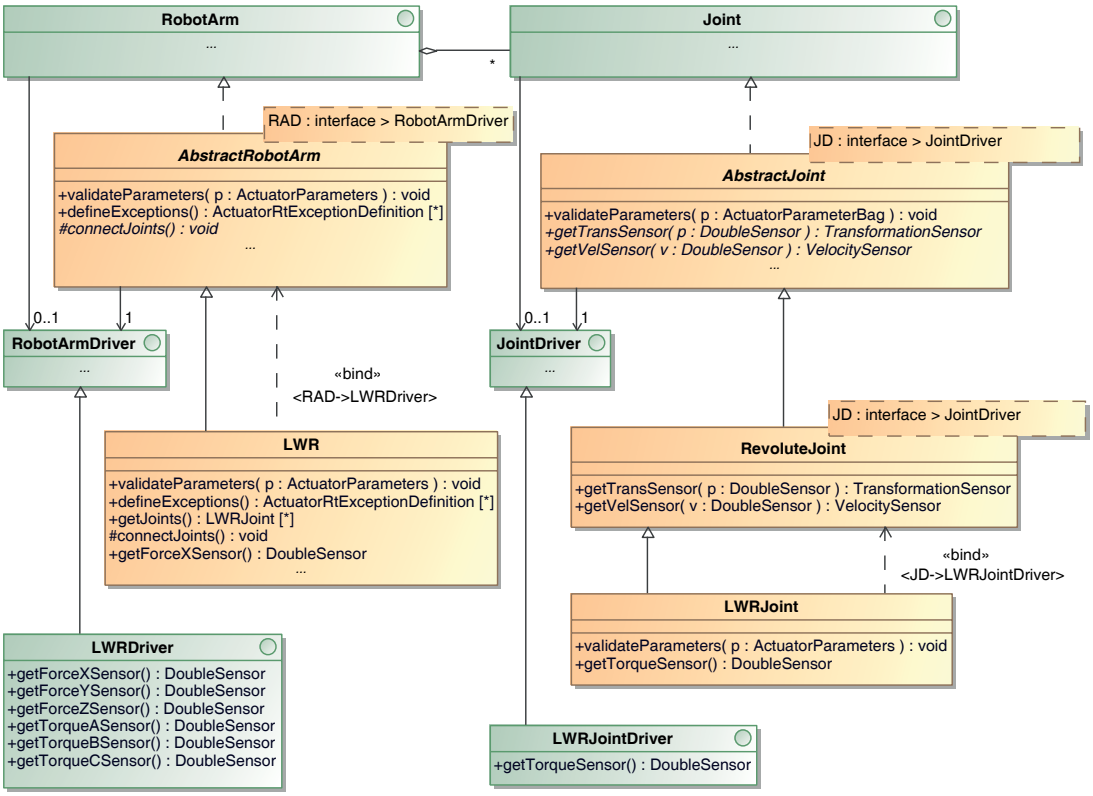

Figure 10.9: The LWR as a concrete robot arm.

LWR-specific ActuatorInterfaces

The LWR's hardware and controller design allows for some functionality that is not provided by most other robot arms. The ability to measure torques and forces has been incorporated into the LWR's Actuator model as described above. The provided Sensors can be used in arbitrary Activities, for example to guard their execution, or trigger other Activities. Other features of the LWR are provided in form of Activities by new kinds of ActuatorInterfaces.

LwrControllerInterface provides Activities for switching among the LWR's internal controller modes and setting controller-specific parameters. The LWR extension supports all three modes provided by the LWR, which are *PositionMode*, *JointImpedanceMode* and *CartesianImpedanceMode*. In position mode, the LWR sacrifies any compliance in favor of best possible position accuracy. No specific parameters can be set. The parameters of the impedance controllers consist of specifications of stiffness and damping, which the LWR extension combines in the concept *ImpedanceSpringSettings*. When JointImpedanceMode is activated, ImpedanceSpringSettings have to be specified for each joint. In CartesianImpedanceMode, a *compliance frame* has to be specified by means of a Transformation relative to the robot flange. In addition, ImpedanceSpringSettings have

to be specified which define the compliance settings for each translatory and rotatory component w.r.t. the specified compliance frame. Finally, the ImpedanceSpringSettings for each joint have to be specified also in this mode.

Besides switching controller modes, LwrControllerInterface can also create Activities for solely changing the compliance frame. As the end-effector forces and torques are always measured w.r.t. the compliance frame, it can be desirable to change this frame to fit various application scenarios. Unfortunately, this has a side-effect when CartesianImpedanceMode is used as controller mode, as the compliance frame is used. This is, however, a limitation of the LWR's interface provided by KUKA.

ToolInterface is an ActuatorInterface which provides Activities to switch the tool parameters used internally by the robot's controllers. The definition of a *RobotTool* by the Robotics API contains the tool's mass, the center of mass relative to the robot flange, and the moment of inertia relative to the center of mass.

Finally, the LWR provides *GravCompInterface*, which can create Activities for operating the LWR in gravity compensation mode. When such an Activity is running, the robot merely compensates its own weight (and that of an attached tool) and can be moved by hand freely. This mode is convenient e.g. for teaching configurations by hand.

Some of the above operations, like controller mode or tool settings, intuitively have the characteristics of ActuatorParameters. Modeling those operations as parameters would allow application developers to specify them orthogonally to the operations that form the actual workflow in a robotics application. However, experiments with the Lightweight Robot and other industrial robots as well (e.g. by Stäubli Robotics) have shown that operations like switching tool configurations can in general not be performed deterministically. The same applies to controller switching with the LWR[2]. Thus, if mechanisms for automatic switching of tools and controllers were e.g. incorporated into motion Activities, those Activities might lose their deterministic nature. This, in turn, would hinder composability of Activities, which is considered a major drawback. An example of the powerfulness of deterministically combinable Activities is given in the next section.

Modeling the above operations via Activities furthermore has the advantage that important preconditions can be checked. For example, switching the controller mode of an LWR when it is applying force to the environment is in general considered dangerous. On the other hand, modifying only the active controller's parameters (e.g. reducing stiffness) may be allowed. By employing ActivityProperties to pass information about the active controller in between Activities, it is possible to check whether switching controllers or adapting parameters is allowed. This decision can also consider further information about the Actuator's state, e.g. the force currently applied to the environment.

Some of the ActuatorInterfaces introduced in this section, e.g. ToolInterface and GravCompInterface, could also be used for other robots with the same abilities. However, other robots might also handle tool switching by ActuatorParameters, if this is possible

[2] which is not even supported directly by the FRI interface and required some implementation tricks

in a deterministic way for the concrete robot. Thus, this decision is left to the respective extensions.

10.6 A Robot-of-Robots

Controlling multiple robot arms in a single Robotics API application is easily possible and to some extent a natural advantage of its object-oriented design. However, for some scenarios, like the cooperative workpiece carrier transport in the assembly cell application, a tight and precise synchronisation of multiple arms is necessary. To ease the realization of such tasks, the MultiArmRobot extension has been developed. A MultiArmRobot is defined here as an aggregation of multiple robot arms whose motion is synchronized with respect to a common motion center Frame. The basic idea in the design of the MultiArmRobot extension is as follows: If multiple robot arms are performing the same kind of Cartesian motion, with the same motion center Frame and same cartesian parameters (such as velocity and acceleration), it is possible to reduce the geometric synchronization problem to a time synchronization problem, provided that motions are planned by the same deterministic algorithm. As the Robotics API holds this assumption and provides built-in mechanisms for exactly time-synchronized operations, an elegant and easy MultiArmRobot implementation is possible and is presented in this section.

The MultiArmRobot extension provides a class *MultiArmRobot*, which is designed to be an Actuator without driver. The integration of MultiArmRobot in the Robotics API's device model is illustrated in Fig. 10.10. MultiArmRobot aggregates the RobotArm Actuators that it consists of and allows for configuring and accessing them by the methods `setRobotArms(...)` and `getRobotArms()`.

MultiArmRobot implements the RobotArm interface, so that it can be used like any other RobotArm in applications. Its implementation of `getJoints` returns the Joints of all inner RobotArms. Its implementations of `getBase` and `getFlange` return the base and flange Frames of the first of its inner RobotArms. The implementation of the methods for calculating the forward position and velocity kinematics functions delegate the task to the first of the inner RobotArms, as all other RobotArms are expected to move synchronously with the first one. The method for calculating the inverse kinematics function delegates this to all of the inner RobotArms and merges their results to the returned array, which will then contain one set of DoubleSensors per inner RobotArm.

MultiArmRobot has to implement some additional methods, including redefining methods from base classes:

- The method `getState()`, for which AbstractDevice provides a generic implementation, is redefined so that the MultiArmRobot's state is provided as the 'weakest' State of all RobotArms it controls.
- `validateParameters()` delegates validation of ActuatorParameters to all RobotArms, which ensures that given ActuatorParameters are valid for each arm.

- Being a purely derived Actuator, MultiArmRobot does not need to define additional exceptions. Thus, the implementation of `defineExceptions()` returns only an empty array.

Note that the exception handling mechanism of the Robotics API's Command model (cf. Sect. 6.4) ensures that an error of any Actuator that is controlled by the same Command will lead to the Command being aborted (unless explicitly specified otherwise). MultiArmRobot can rely on this mechanism for safe operation, as it is guaranteed that all RobotArms will stop when any error occurs.

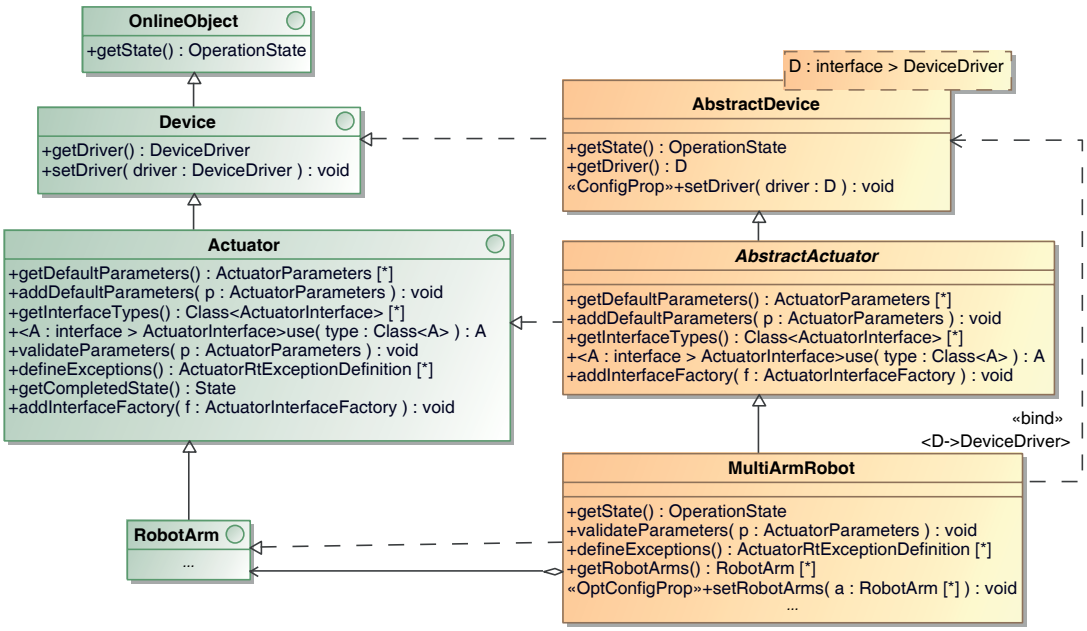

Figure 10.10: Model of a MultiArmRobot device.

MultiArmRobot is equipped with a single ActuatorInterface, which is an implementation of LinearMotionInterface. Being able to move linearly was enough for the intended use of MultiArmRobot in the assembly cell scenario. However, extending MultiArmRobot with further Cartesian motion capabilities is trivial. Fig. 10.11 depicts the classes *MultiArmActivity* and *MultiArmLinearMotionInterface* and their contexts. MultiArmActivity is a simple extension of ParallelActivity, which is part of the Robotics API's Activity model (see Sect. 8.6). It enhances the generic ParallelActivity by the PlannedActivity interface, adding the ability to provide a State indicating the progress of the parallely executed Activities. The uniform Cartesian motions of robot arms in this context are expected to also have a uniform progress. Thus, the implementation of `getProgressState(...)` merely creates an AndState aggregating the progress States of all parallel motion Activities (which are all PlannedActivities, as required by LinearMotionInterface).

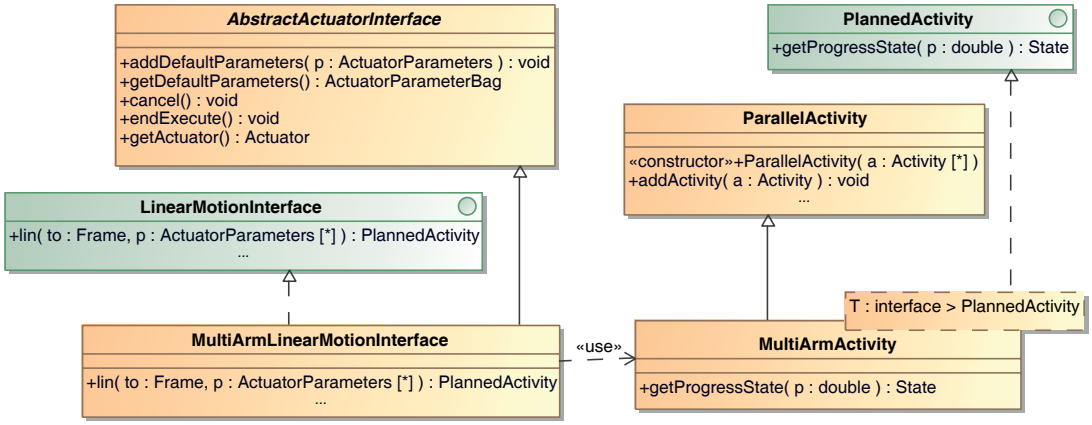

Figure 10.11: ActuatorInterface for linear motion of a MultiArmRobot.

MultiArmLinearMotionInterface creates a motion Activity for each inner RobotArm by employing the LinearMotionInterface implementation provided by this arm. The following steps are performed by MultiArmLinearMotionInterface's implementation of `lin(...)` when called with target Frame f and ActuatorParameters P:

- A common MotionCenterParameter μ is calculated by first inspecting P. If no MotionCenterParameter is supplied here, the MultiArmRobot's default MotionCenterParameter is retrieved. If this also fails, an exception is thrown. An alternative would have been to choose the default MotionCenterParameter of an arbitrary RobotArm (e.g. the first of the configured ones). This could, however, lead to motions that are hard to foresee for the application developer.
- The same procedure as with MotionCenterParameters is performed to find CartesianParameters ς to use for all RobotArms.
- Then, a linear motion Activity λ_i is created for each RobotArm i in the following way:
  - The RobotArm's LinearMotionInterface implementation Λ_i is retrieved.
  - The default DeviceParameters P_i of this LinearMotionInterface are retrieved by calling its method `getDefaultParameters()`.
  - μ and ς are added to P_i, overriding any MotionCenterParameter and CartesianParameters that were present in P_i.
  - λ_i is created by calling Λ_i's method `lin(...)` with arguments f and P_i.
- Finally, a MultiArmActivity containing all λ_i is created and returned.

The design of the MultiArmRobot extension highlights two aspects: On the one hand, modeling dynamic teams of robots is possible by defining new kinds of Actuators with

little effort. For application developers, a MultiArmRobot can be used like any other Actuator. Activities created by a MultiArmRobot's ActuatorInterfaces will be scheduled automatically such that they don't interfere with other Activities controlling the inner RobotArms. This is ensured by the Activity scheduling algorithm, which respects the controlled and affected Actuators of each Activity. On the other hand, it shows the power of the Activity composition mechanisms: The Activities created by MultiArmLinearMotionInterface are 'full-fledged' linear motion Activities in the sense that they provide States indicating their progress. Furthermore, they automatically support motion blending between successive multi-arm motions. This is possible as the composed Activities provided by the Activity Layer combine meta data of all affected Actuators appropriately (Sect. 8.6) and the cross-Activity handshake (as illustrated in Sect. 10.2) works for an arbitrary number and type of Actuators. Fig. 10.12 shows some pictures taken from the Factory 2020 application, showing a series of blended linear motions performed by a MultiArmRobot consisting of two KUKA LWRs. A video can be found on YouTube[3].

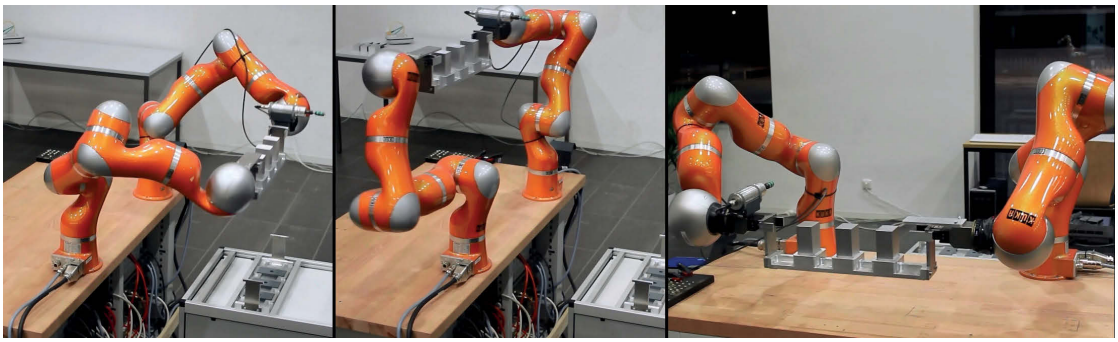

Figure 10.12: A MultiArmRobot performing a series of blended motions in the Factory 2020 application.

The biggest advantages of the presented design are its simplicity and its independence of a RoboticsRuntime implementation. The MultiArmRobot extension in the presented form consists of only around 250 lines of code, and it may be used with any RoboticsRuntime, as it purely relies on runtime-independent Robotics API concepts The main disadvantage is limited scalability. For a MultiArmRobot with n RobotArms, the same Cartesian motion profile is planned and executed n times. A more efficient implementation could plan the motion only once and supply the interpolation values to all RobotArms. This would not add much complexity, but require adding a MultiArmRobotDriver and appropriate implementations of this driver in each RoboticsRuntime. A second, completely different alternative could employ a master-slave pattern for realizing a MultiArmRobot. It would have to assign one RobotArm a master role, plan motions only for this RobotArm and instruct all other RobotArms to follow the master robot w.r.t. a common motion center Frame. In contrast to the variant discussed above, no special driver for

[3] `http://www.youtube.com/watch?v=gf3673XkHCw`

MultiArmRobots would be necessary. However, all slave RobotArms would have to provide an ActuatorInterface capable of creating Activities to follow moving Frames. The quality of those Activities then determines the overall synchronisation quality.

10.7 Summary

The general model of robot arms provided by the RobotArm extension is able to capture diverse kinds of industrial robot arms. At the same time, the design maximizes reuse of common aspects by appropriate abstractions. Various kinds of operations can be implemented in the framework provided by the Activity concept. This allows to realize even complex, context sensitive operations like blending across motions in a generic way. At the same time, composability of Activities is preserved, which is exemplified by the straightforward MultiArmRobot design.

Chapter 11

Evaluation

Summary: The contributions of this work are evaluated in this chapter. This is done based on the application examples introduced in the beginning, by investigations of reusability and performance, and by evaluating the degree of fulfillment of the initial requirements to the SoftRobot architecture. The software design of the Tangible Teleoperation application based on the Robotics API has been published in [38], while results regarding reusability and performance have been published in [31].

The Sects. 11.1 to 11.3 demonstrate how the challenging PortraitBot, Tangible Teleoperation and Assembly Cell applications have been realized with the concepts introduced in this work. To evaluate reusability and extensibility of the Robotics API's design, the distribution of the Lines Of Code throughout the reference implementation of the SoftRobot architecture is investigated in Sect. 11.4 with very positive results. The performance of the reference implementation in practical scenarios is studied in Sect. 11.5. Finally, Sect. 11.6 assesses how the Robotics API contributes to fulfilling the requirements to the SoftRobot architecture that were stated in Chap. 4.

11.1 Realization of the PortraitBot Application

The 'PortraitBot' was introduced in Sect. 3.1 and consists of a setup of two robot arms. One of them, the 'drawing robot', is equipped with a pen and can draw on a canvas that is held by the second robot arm, the 'canvas robot'. A camera mounted at the canvas recognizes human faces in its field of view and the PortraitBot can draw sketches of those faces. When drawing, the drawing robot compensates movement of the canvas, as the camera is rotated in order to track movements of the portrayed human.

The PortraitBot application was realized as a standard Java application. It consists of a simple user interface based on the Java Swing libray, application logic for image processing and robot control, and some utility functions for saving and loading drawable sketches. This section will give an overview about the image processing logic and the Robotics API concepts that are used for synchronized robot motions. The user interface, which is mainly responsible for tuning image processing parameters, store and load sketches, display preview images and trigger the start of the drawing process, will not be further explained, nor will the utility functions.

The image processing logic necessary to recognize and track faces in video streams and to extract sketches from streams are based on the OpenCV[1] and JavaCV[2] libraries. JavaCV is employed to access the C++ based OpenCV library from Java via the Java Native Interface (JNI)[3]. In 2013, OpenCV was extended with built-in support for JNI-based access from Java. However, the PortraitBot application was developed earlier and thus relied on the alternative JavaCV project.

The class *PortraitBotApp* represents the actual application. It used the class *VideoInputFrameGrabber* provided by the JavaCV library for capturing the video stream of the attached camera. PortraitBotApp contains a cyclic loop that first polls the camera image captured by a VideoInputFrameGrabber instance. Then, it calls the following internal methods to extract a sketch from the captured image:

- `trackFace(...)` uses an OpenCV Cascade Classification algorithm (see [135]) to determine a rectangular region in the camera image that is likely to contain a human face,
- `filterContours(...)` applies a series of OpenCV functions for creating a gray-scale image of the detected region, as well as smoothing and filtering this image in order to amplify the contours,
- `extractSketch(...)` uses an OpenCV function to find the contours in the pre-processed image region and create a sketch structure from them (see below).

The extracted sketch is stored by the PortraitBotApp. For representing a two-dimensional sketch, a set of classes was introduced:

- *Point2D* represents a point in 2D-space,
- *Path2D* is an aggregation of Point2D and thus represents a path in 2D space that consists of a series of straight lines, defined by a pair of successive points,

[1] `http://opencv.org/`
[2] `http://code.google.com/p/javacv/`
[3] `http://docs.oracle.com/javase/6/docs/technotes/guides/jni/`

- *Sketch2D* is an aggregation of Path2D. It represents all Path2D instances of the final, drawable sketch. Sketch2D provides utility methods to prepare the sketch for efficient drawing, e.g. by eliminating very short line segments and adjacent parallel lines, and by sorting all Path2D instances so that they can be drawn with minimal intermediate movements.

All the abovementioned classes also implement the Java interface *Serializable* and can thus be used to serialize sketches for storing them to disk and loading them again. The method `extractSketch(...)` employs the information returned by OpenCV line detection algorithms to construct an appropriate Sketch2D instance.

All methods to control the two LWRs movements are encapsulated in the Facade (cf. [86], pp. 185) class *RobotInterface*. Its most important methods are presented in the following.

The method `initialize()` is called by PortraiBotApp when the application is starting. The method moves both robot arms to a pre-defined start position. Once they have arrived, the drawing arm is geometrically synchronized with the canvas arm using a HoldCartesianPositionActivity. The Frame to hold is constructed by taking a snapshot of the drawing arm's motion center Frame relative to the canvas arm's motion center Frame. The situation is depicted in Fig. 11.1: C is located in the center of the canvas and is the canvas robot's motion center Frame. S is located at the tip of the drawing pen and coincides with the drawing arm's motion center Frame (which is not shown in the figure) at the current time instant. However, as S was snapshot relative to C, there exists a StaticConnection r between C and S, causing S to follow C's movement. S is used as Frame to hold for the drawing arm's HoldCartesianPositionActivity. As explained in Sect. 10.4, the Activity is purely maintaining the position of the robot's motion center Frame at the given Frame and can be taken over by path motions.

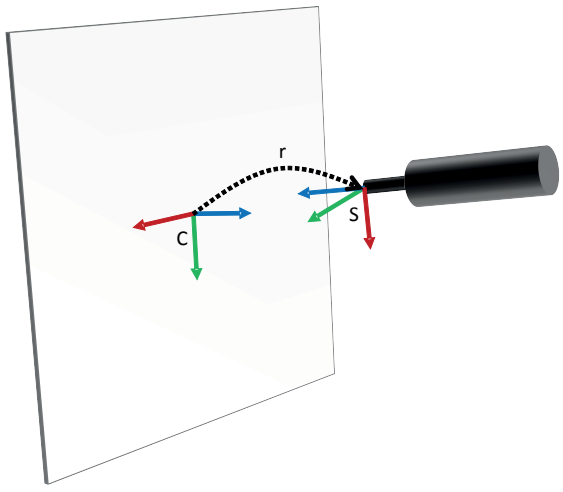

Figure 11.1: Frames during initial synchronisation of robot arms in the PortraitBot application.

Path motions are then used by the RobotInterface's method `drawSketch(...)` to draw all paths in the supplied Sketch2D. In particular, each line segment of each path is drawn using a LinearMotionActivity. The information encoded in the Point2D instances of each Path2D instance is used to construct Robotics API Frames that are located on the canvas surface (i.e. that are connected to C with static Relations only). The first linear motion takes the drawing arm's pen tip from the initial Frame S (see above) to the Frame corresponding to the first Path2D's start point. Then, all lines in the path are followed by successive linear motions. The LinearMotionActivities used consider motions of the goal Frame during interpolation of the path. Additionally, they actively maintain the robot arm's position at the target Frame, as described in Sect. 10.4. Each LinearMotionActivity is able to take over control of the robot arm from the previous Activity which is maintaining the robot arm's relative position. To overcome gaps between consecutive paths in the sketch, some utility Frames are constructed at a certain distance to the canvas. Approaching those Frames with linear motions takes the pen from the end of one path to the start of the next one.

Independently of the sketch drawing process, the canvas robot is controlled so that it points the camera directly towards the face recognized in the video stream. RoboticsInterface provides the method `updateFaceTracking(hOffset : double, vOffset : double)` for this purpose. The parameters specify the current horizontal and vertical offset of the recognized face from the center of the camera image, relative to the full image size. For example, if the face is in the center of the camera image, the offsets are both zero, and if it is in the top left corner, the offsets are both -1. Based on those offsets, a simple Java-based control algorithm adjusts the camera's rotation in order to keep the face in the center of the camera image. The resulting motion of the canvas robot arm is automatically calculated by the CartesianJoggingActivity that is employed (see Sect. 10.4). Fig. 11.2 shows the Frames that are relevant for Cartesian jogging. A Frame M is defined to describe the location where the camera is mounted on the canvas relative to the canvas center Frame C, using the Relation r1. Another Frame called L describes the location of the camera lens. It is related to M with the Relation r2. By using M as intermediate Frame, manual adjustments to the camera orientation by rotating around the mount point can be easily reflected in the application by merely adjusting r1's first rotation component. The Frame L located at the center of the camera lens is used as jogged Frame as well as orientation Frame for Cartesian jogging. Additionally, L's origin is used as Pivot Point. The canvas robot's base is used as reference Frame. With this parameterization, `updateFaceTracking(...)` can calculate and command rotatory velocities that directly rotate the camera lens Frame L.

The PortraitBot application integrates velocity control of a robot arm, synchronized relative path motions of a second robot arm and complex image processing with effects on robot control in a single standard Java application that is able to run on a single computer system[4]. Velocity-based control is easily possible due to the powerful CartesianJoggingActivity provided by the Robotics API. Synchronizing path motions to moving targets

[4]The PortraitBot application was run on a MS Windows system during demonstrations, but Linux versions of all libraries exist, thus deployment to a linux-based real-time robot control system is possible.

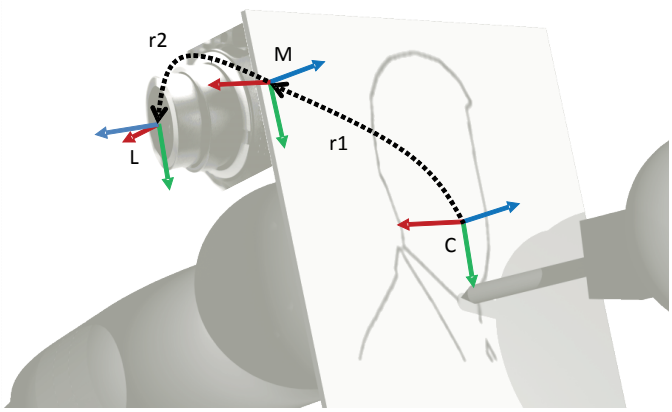

Figure 11.2: Frames used for face tracking in the PortraitBot application.

is possible with minimal effort as well, as the Activities used employ the Robotics API's world and sensor models to consider such dynamic behavior in their implementations. In fact, the parts of the programs that move both robot arms are completely independent of each other once an explicit synchronization step has been performed. The Robotics API is technically realized as a standard Java library, which enables seamless integration of many existing libraries like the powerful OpenCV library. The SoftRobotRuntime implementation employs network protocols to communicate with the SoftRobotRCC, thus Robotics API applications can run on any operating system with Java support and a network connection to a SoftRobotRCC instance. System integration is not necessary at all.

A video of the PortraitBot application is available on youTube[5]. The video shows a demonstration at the final project meeting of SoftRobot. In this demonstration, the camera was not mounted at the canvas. Instead, visitors could take a seat and have a picture taken of them. The extracted sketch was then stored in a list of drawing tasks and the PortraitBot continuously processed those tasks. Visitors could return later and fetch their finished portraits. To demonstrate the synchronization between the robot arms, the canvas robot followed a predefined motion pattern. The structure of the PortraitBot application was not changed, however. The code for velocity-based face tracking was just disabled and replaced by a series of sequential, blended linear movements for the canvas robot.

11.2 Realization of the Tangible Teleoperation Application

As illustrated in Sect. 3.2, the Tangible Teleoperation enables remote control of a system with two robot arms. The operator can use an interface that offers interaction via

[5] http://youtu.be/UTmoL87UVJQ

multi-touch and tangible elements. Feedback is provided by a camera mounted to one of the robot arms, a 3D visualization of the robots' poses, and a visualization of the contact forces measured by the robots. The operator can choose between *direct mode* and *observer mode*. In the former mode, only the single *observer robot* with its attached camera is controlled. In the latter mode, the second robot arm, called the *manipulator robot*, is controlled. In this mode the observer robot lets the camera face towards the manipulator robot and is synchronized to its movements.

The application and UI logic was implemented using C# and the Microsoft .NET framework [39]. The multi-touch tangible user interface was implemented based on the Microsoft PixelSense SDK[6], 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[7]. In this way, the complete functionality of the Robotics API could be used without restrictions by referencing the converted Java library as well as a set of libraries provided by the IKVM.NET project in the .NET application for the PixelSense platform.

In the Tangible Teleoperation application, the main implementation effort was put in the user interface parts. This section will give a brief overview of the tangible and multi-touch user interface elements that have been developed and the libraries that were used to alleviate the realization. It will then go into details about the Robotics API concepts that were used to control the robot arms appropriately and to access data for providing feedback to the operator.

The general layout of the Tangible Teleoperation UI is displayed in Fig. 11.3. A large portion of the UI is reserved for displaying the video stream delivered by the camera, which is positioned centered at the top of the screen. To the left and right of the camera picture, controls for opening and closing of the grippers and moving the robot arms to some predefined positions are located. In the bottom center of the screen, a 3D visualization of the robots' poses is presented. For controlling the robot arms during teleoperation, a combination of physical objects and virtual user interface elements is used. The motion of the active robot arm is controlled using the 3D mouse SpaceNavigator by 3Dconnexion. This 6-DOF mouse can be used to intuitively control Cartesian movement of the robot end-effector. The SpaceNavigator is at the same time employed as a tangible user interface element: When it is placed somewhere on the free space of the teleoperation screen, a tag sticked to its bottom is recognized by the application. The user interface control shown in Fig. 11.4 is displayed and at the same time, movement of the robot arms is enabled. From then on, the SpaceNavigator controls the robot movement. The displayed control provides additional selectable options, in particular switching from direct mode to observer mode and vice versa. In observer mode, the camera robot can be rotated around the manipulator robot's end-effector in two dimensions (up/down and left/right). Additionally, the user can zoom into and out of the perspective by moving the camera robot closer to or further away from the

[6] http://msdn.com/windows/surface
[7] http://www.ikvm.net/

manipulator robot. These three degrees of freedom are controlled by a second tangible interface element, which is a simple hemispherical object with another tag attached to it. Upon contact to the PixelSense screen, the UI control shown in Fig. 11.5 is displayed. In contrast to the SpaceNavigator's UI control, this control does not follow movements of the tangible object. Instead, moving the tangible element outside the center of the visualization rotates the camera robot, while rotating the tangible element when it is inside the center zone controls the zoom level by moving the camera robot closer to or away from the manipulator robot's tool.

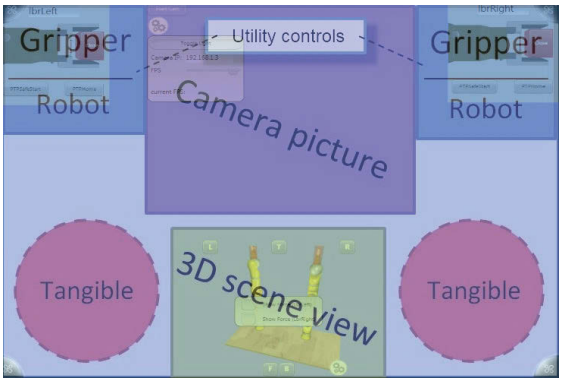

Figure 11.3: Overview of the operator user interface in the Tangible Teleoperation application. Image source: [38]

A set of .NET libraries was used to create the UI components. As the PixelSense user interface is based on the Windows Presentation Foundation (WPF), various existing

Figure 11.4: Visualization of the manipulator robot controller. Image source: [38]

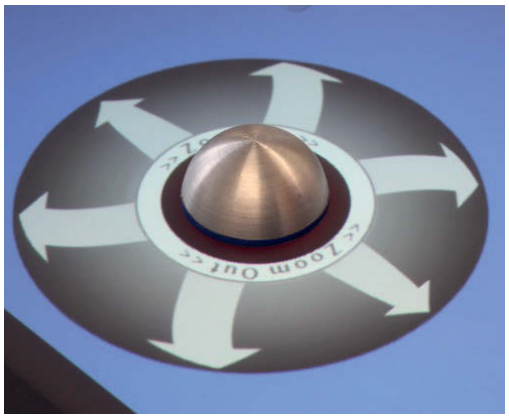

Figure 11.5: Visualization of the observer robot controller. Image source: [38]

controls can be used for the PixelSense as well and can be extended for multi-touch and tangible interaction. To display the video stream delivered by the wireless camera, classes from the AForge.NET Framework[8] were used and embedded in a user interface component for the PixelSense UI. To create the 3D visualization component, WPF controls from the 3D Tools[9] project were employed. Those controls were embedded in a PixelSense component and extended by multi-touch functionality to make them freely rotatable and zoomable with one and two fingers, respectively. To keep the visualization synchronized to the actual poses of the robot arms, the Robotics API's SensorListener concept is used to get current position values of all joints of the two robots. SensorListeners are also used to retrieve and visualize the contact forces measured by the LWR's integrated torque sensors. Finally, PixelSense UI components for the utility controls were created purely based on the PixelSense SDK.

The application and robot control logic necessary for the Tangible Teleoperation application is very simple. Besides some point-to-point movements for approaching initial and safe positions for recovery on errors, all other robot control functionality is velocity-based and realized with CartesianJoggingActivities. The application logic itself consists of a state machine that decides which robot and gripper operations are allowed in which situation. For example, when one of the tangible elements is placed on the PixelSense screen, manual robot control should be enabled. However, this is only allowed once the robots have finished some initial point-to-point motions. Similar restrictions apply when switching from direct mode to the observer mode, as the observer robot is then moved to its initial observing position before further control is possible. The state machine will not be presented in detail here. The following explanation will rather focus on the way CartesianJoggingActivities are employed.

The Frames that are relevant in both operation modes are displayed in Fig. 11.6. In direct mode, a single CartesianJoggingActivity is employed to control the camera robot directly. The camera robot's gripper Frame is used as jogged Frame, its Orientation is used as jogging orientation and its origin is used as Pivot Point for the CartesianJoggingActivity. Deflections of the SpaceNavigator 6D mouse are mapped to translation and rotation velocities of the CartesianJoggingActivity in a straightforward way. As the camera Frame is only minimally displaced relative to the camera robot's gripper Frame, the operator experiences movements from a first-person perspective.

In observer mode, two CartesianJoggingActivities are used, one per robot arm. For the manipulator robot's Cartesian jogging, its gripper Frame and its base Frame are used as jogged Frame and reference Frame, respectively. The gripper frame is also used as Pivot Point. As jogging orientation Frame, the observer robot's camera Frame is used. This has the effect that the mapping of 6D mouse deflections to velocities always follows the camera perspective. For the observer robot, a second CartesianJoggingActivity is used. Here, the camera Frame is used as jogged Frame. By using the manipulator robot's gripper Frame as reference Frame for this jogging Activity, the camera robot

[8] http://code.google.com/p/aforge/
[9] http://3dtools.codeplex.com/

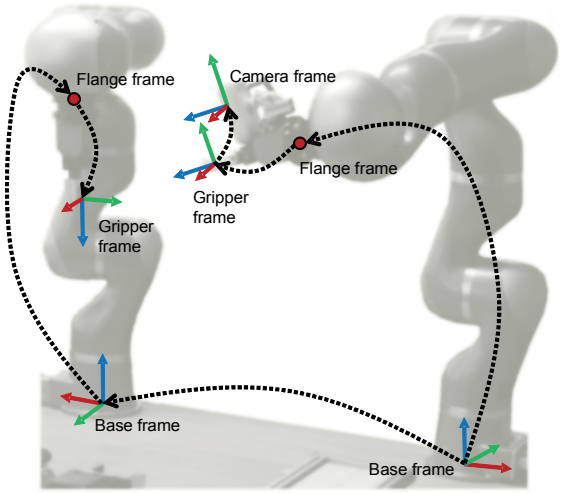

Figure 11.6: Important Frames in the Tangible Teleoperation application.

automatically follows all movements of the manipulator robot's gripper. To allow proper adjustments of the observer perspective, the manipulator robot's gripper Frame is also used as Pivot Point for the camera robot's jogging Activity. Thus, the camera robot will always rotate around the manipulator robot's gripper. Finally, the camera Frame is used as jogging orientation Frame to relate velocities to the camera perspective.

The Tangible Teleoperation application is a further example of how robotics applications can profit from modern software ecosystems. Direct access to many libraries and tools, like those provided by the IKVM.NET, AForge.NET and 3D Tools projects, greatly simplify application development. In particular, the tools and libraries of IKVM.NET are an example of the interoperability of modern programming languages, which is usually not the case with proprietary robot programming languages. Like in the PortraitBot application, the built-in network communication between the Robotics API and the SoftRobotRCC allows for deploying applications on any system that has network access, like e.g. the PixelSense device.

Besides removing the need for system integration, the Robotics API provides several powerful concepts that are employed by the Tangible Teleoperation application. SensorListeners are used to provide various feedback to the operator. The sensor values are delivered with very short latency. During test runs with various operators, control and feedback values were transmitted without noticeable delay. When it comes to velocity-based robot control, the application demonstrates the powerfulness of the Cartesian jogging concepts provided by the Robotics API. All control and synchronization of the two robot arms could be realized just by appropriate parameterization of CartesianJoggingActivities. Furthermore, the Robotics API's support for the Lightweight Robot's different control modes was used to set a defined compliance. Thus, contact forces are easily controllable by operators. Further details about the Tangible Teleoperation

application can be found in [38] and a video can be found on youTube[10].

11.3 Realization of the Assembly Cell Application

The Assembly Cell application controls two Lightweight Robots equipped with grippers and a screwdriver such that they transport workpiece carriers and assemble workpieces from parts. The application does not provide any user interface. Instead it is part of a hierarchical service-oriented architecture that assigns tasks to certain services and orchestrates those services appropriately to achieve the desired workflow. This work will not go into detail about the distribution of functionality to single services, nor the orchestration of those services. However, all robot control tasks are implemented inside various services based on the Robotics API. This section will describe how the challenges of cooperative workpiece transport, compliant assembly of workpieces and compliant fetching, inserting and tightening of screws are realized with the Robotics API.

The Assembly Cell's worfklow starts with the two LWR's cooperative transport of the workpiece carriers onto the workbench, and it ends with the transport back onto the mobile platform that delivered the carriers. To initially grip the carriers, the two LWR's apply a force-guarded searching strategy to detect the exact carrier location, which can vary due to inaccurate movement of the mobile platform and missing tight fixations of the carriers on the platform.

During transport of the carriers, tight synchronization of the two LWR's movements was necessary to achieve a safe cooperative motion without damage to the carriers or the robots. This was achieved using the MultiArmRobot presented in Sect. 10.6. The two LWRs were combined to one MultiArmRobot device once they had both gripped a workpiece carrier. From this point on, they can perform exactly synchronized linear motions w.r.t. a common motion center Frame. Fig. 11.7 depicts this motion center Frame C as well as the start Frame S, goal Frame G and some intermediate Frames I_1-I_4 that were used to model the transport motion. Between each pair of intermediate Frames, a linear motion was performed, and all motions were connected with motion blending to achieve one smooth, continuous motion from start to goal.

A challenge during the development of the two-arm transport motion was to choose the intermediate Frames and, in particular, the joint configurations of both LWR arms at those Frames appropriately. During the motion sequence, a large portion of the LWRs workspace is used. To stay within the limits of each joint during the sequence, the joint configuration at each intermediate point had to be chosen carefully and the redundancy of the robot arms had to be employed to be able to perform the complete motion. For the Assembly Cell application, feasible joint configurations were determined by experiments that involved guiding the robot arms by hand in gravity compensation mode. When a set of feasible joint configurations was found for each robot arm, it was stored for each of the two arms and assigned to the Frame that represented the current

[10] `http://youtu.be/lbRjwjArg2c`

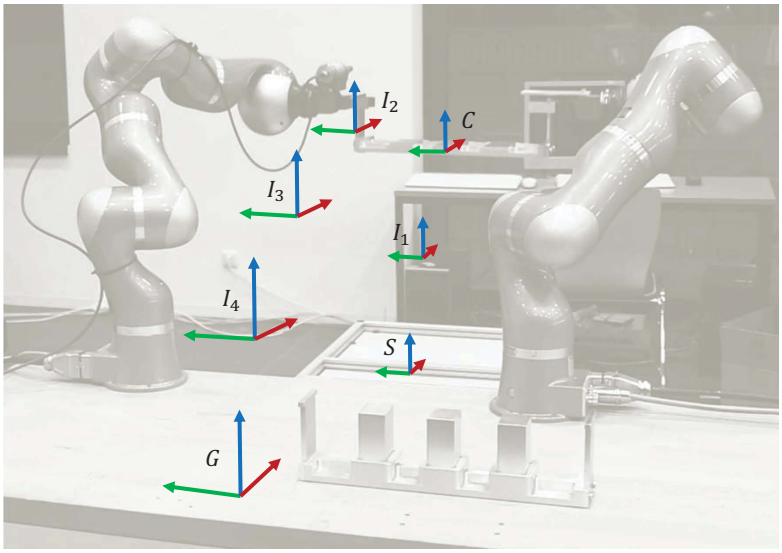

Figure 11.7: Some intermediate Frames that are part of the workpiece carriers' path when transported onto the workbench.

position of the workpiece carrier. All of those Frames together formed the motion path. The LinearMotionActivities controlling the MultiArmRobot's movement along the path were parameterized with HintJointParameters that contained the predetermined joint configurations for each robot arm.

When both workpiece carriers have been placed onto the workbench, each of the workpieces is assembled from its two parts, a screw is fetched and the two workpiece parts are screwed together. The final workpiece is then put back in one of the carriers. When all workpieces have been assembled, the carriers are transported back onto the mobile platform. The process of fetching workpiece parts will not be described in detail, as it was realized merely with predefined linear and point-to-point motions. In the following, an overview will be given of the assembling process of the two workpieces. The process of fetching screws will then be explained in detail, as it is the most complex part of the workflow. Finally, an overview of the screw inserting and tightening process will be given.

To assemble the workpiece parts, the workpiece cover (top part) is positioned directly in front of the workpiece corpus (bottom part) as depicted in Fig. 11.8. Both LWRs are then switched to their Cartesian impedance controller and a velocity-based motion is used to move the cover straight towards the corpus. When contact is established, the cover is pressed onto the corpus with increasing force. The motion is canceled once the contact force has exceeded a predefined threshold. Experiments have shown that this combination of accurate prepositioning and force-based assembly yields safe and reproducible results.

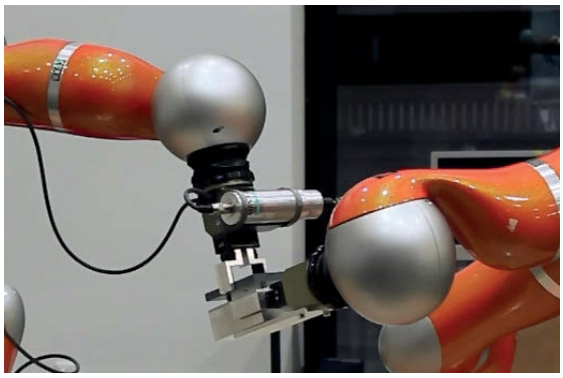

Figure 11.8: Pre-positioning the workpiece parts for compliant assembly.

After assembly, the workpiece parts are mounted together. For this purpose, a screw is fetched from a screwplate that serves as a magazine and is mounted at the edge of the workbench. The positions of all screws in the magazine are known, and the application also keeps track of the screws that have already been removed. The most crucial part of fetching screws is inserting the screwdriver tip in the screw's head. The tip is magnetic and is thus able to carry the screw once it has been fetched. However, the tip is not completely fixed and can bend sidewards to some extent. Thus, predefined movements are not sufficient for fetching screws safely. Fig. 11.9 shows three intermediate situations of the force-based screw fetching process.

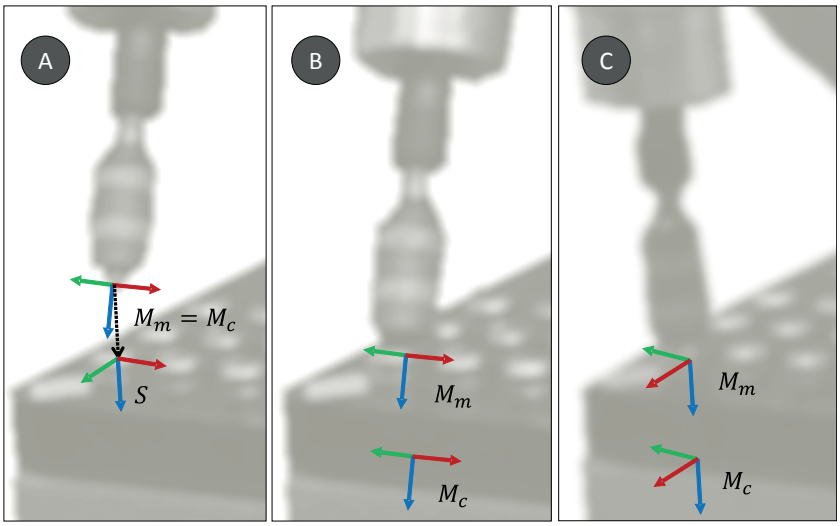

Figure 11.9: Frames involved in the three steps for fetching a screw.

In the situation depicted in A (left part of the figure), the screwdriver has been positioned straight above the screw's head, but tilted sidewards. The known frame S is located exactly inside the screw's head. The Frames M_m and M_c represent the measured and

commanded motion center Frame of the screwdriver robot, respectively. In situation A, both coincide. The screwdriver tip is then moved towards the screw head and builds up a defined force on it. A velocity-based motion Activity with a force-based cancel condition is used to achieve this. In particular, one of the CartesianVelocityMotionInterface's `moveVelocity(...)` methods is used with the following parameterization:

- M_c is used as moved Frame,
- the robot arm's base Frame is used as reference Frame,
- M_c's origin is used as pivot point and
- S' Orientation is used as orientation for `moveVelocity(...)`.

The variant of `moveVelocity(...)` that is used expects each component of the Cartesian velocity to be specified as double value. For the desired motion, a low z velocity is adequate (indicated by the dotted arrow in the left part of Fig. 11.9), whereas all other velocity components are set to zero. The cancel condition for the velocity-based motion is created by considering the y and z components of the Cartesian force measured by the LWR relative to the current tool Frame (which is defined to be M_c). By using some calculation functions provided by different Robotics API Sensors, a State is constructed that becomes active when the measured absolute force in the y-z-plane exceeds a threshold. This State is used as cancel condition. To ensure that the contact force is built up slowly and in a controlled way, the LWR is configured to use its Cartesian impedance controller.

The velocity-based motion has positioned the screwdriver tip on the head of the screw, exerting a defined force on it. This situation (B) is shown in the center part of Fig. 11.9. In this case, the commanded (M_c) and measured (M_m) motion center Frames do not coincide: M_m is located at the actual physical position of the screwdriver tip, which is measured by the LWR's joint encoders. M_c is located at the 'ideal' position the screwdriver tip was commanded to be by the previous velocity-based motion. In this situation, a linear motion is used to turn the screwdriver tip upwards and rotate it around the tip's axis in one motion, while at the same time maintaining the current force on the screw. To achieve this, the linear motion's goal is created based on the current position of M_c in situation B, using a combination of the Frame's `plus(...)` and `snapshot(...)` methods. The right part of Fig. 11.9 (C) shows the situation after this linear motion has been executed and M_c has arrived at the abovementioned goal. Experiments during the development of the Assembly Cell application have shown that this combination of a rotating motion of the screwdriver tip and a force applied onto the screw head create a situation that allows for reliably loosening the screw.

Two steps remain to be done in situation C: First, the screwdriver is started in reverse direction, which loosens the screw. The force on the screw is still maintained until the screwdriver is stopped. Then, the LwrMotionInterface's method `releaseForce()` is used to fully release all force. This method internally executes a linear motion that

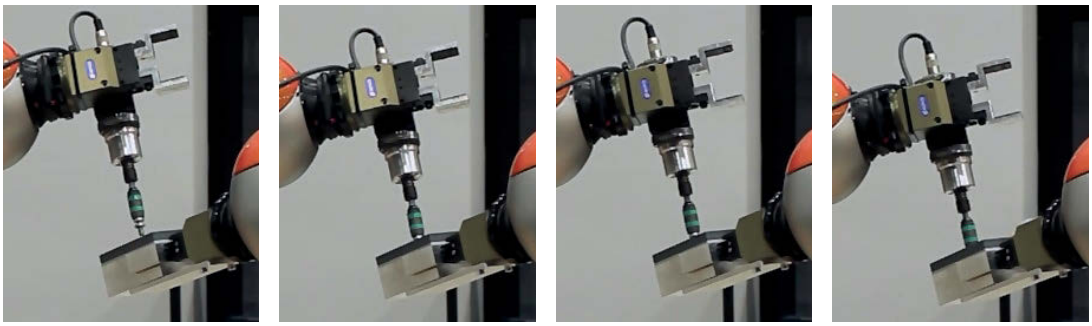

Figure 11.10: The four steps for inserting and tightening screws.

moves the LWR's commanded motion center Frame towards its measured motion center Frame, which effectively releases the force induced by the LWR's compliance.

Inserting and tightening screws is performed in a similar manner like loosening screws and will be outlined in the following. The process is supported by the design of the workpiece cover, which provides some mechanical guidance for inserting the screw. Fig. 11.10 shows the four steps that are performed. First, the screwdriver and its attached screw are prepositioned next to the screw hole, but slightly tilted compared to the hole's axis. From this position, the screw's tip is moved velocity-based straight towards the screw hole until a contact force is detected. The contact force is kept low at this point to prevent the screw from snapping from the screwdriver's tip. The screwdriver is then turned upwards so that the screw is aligned with the screw hole's axis. In the fourth and last step, a velocity-based motion is used to build up a force on the screw. When this force has reached a certain level, a screwdriver Activity is started to tighten the screw. The screwdriver Activity is added as triggered Activity to the velocity-based motion Activity. Thus, the force on the screw is increased during tightening, which is partially compensated by the screw moving downwards the thread inside the hole. The movement velocity and compliance settings of the LWR are chosen appropriately to keep the force within acceptable bounds. By using the screwdriver Activity's CompletedState as cancel condition for the motion, the force is no longer increased once the screw has been tightened. Finally, the applied force is released and the screwdriver is moved away from the workpiece. During the inserting process, both LWRs use their impedance controllers to provide controlled active compliance.

The Assembly Cell application demonstrates many of the powerful capabilities of the Robotics API in one application: the presented MultiArmRobot is employed for synchronizing the two LWRs. Velocity-based motions, guarded by force sensors, are used for establishing environmental contact with the robot arms. Triggered Activities are employed for controlling the screwdriver during motions. Again, a video of this application is available on youTube[11].

[11] http://youtu.be/gf3673XkHCw

11.4 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 compliant Robot Control Core should be kept small. It is inevitable to integrate a real-time capable device driver for many kinds of devices 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 give a rough idea of the complexity of the architecture and the size of reusable parts, Fig. 11.11 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 basic parts of the SoftRobot architecture (*SoftRobot basics*), including the Robotics API *core*, *activity* and *world* components, the generic parts of the algorithm for transforming the basic Robotics API models to RPI primitive nets, and the SoftRobotRCC including a real-time capable RPI interpreter. This does not comprise support for any concrete device.
- The *Robot Arm extension*, introducing the definition of a robot arm consisting of joints into the architecture.
- The KUKA *LWR*, Staubli *TX90* and Universal Robots *UR5 extensions*, introducing vertical support for three different kinds of robot arms, all based on the Robot Arm extension.
- The *Robot Base 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 Robot Base extension.
- The *youBot Base* extension, introducing support for the KUKA youBot's mobile platform, based on the Robot Base extension.

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

- The *Activity packages* of the Robotics API, providing definitions and implementations of Activities,

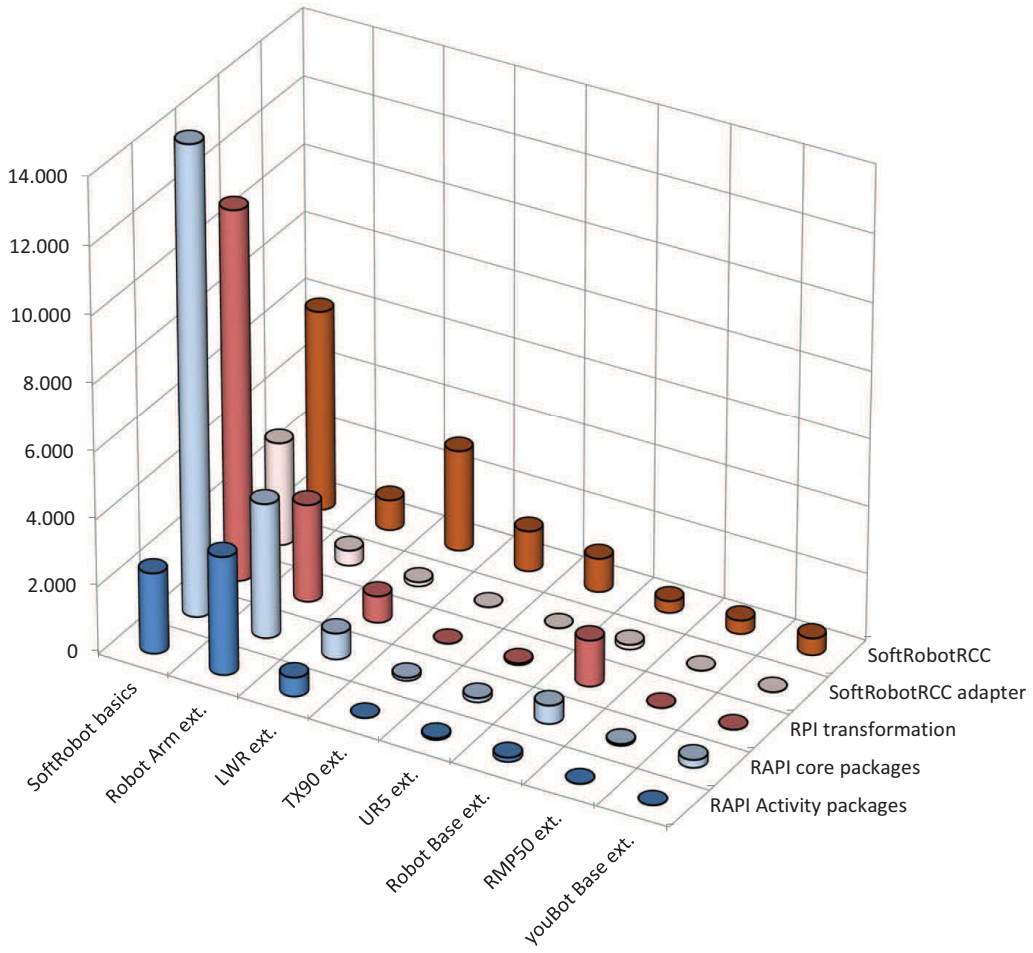

Figure 11.11: Distribution of Lines Of Code in the SoftRobot architecture reference implementation. Adapted from [31].

- the *core packages* of the Robotics API, providing definitions and implementations of Commands, Actions, Sensors and States,
- the packages related to the *RPI transformation* algorithm, which includes the basic algorithm as well as transformation rules for new Devices and Actions,
- the *SoftRobotRCC* adapter, including the generic communication protocols as well as optimizations introduced by certain Device and Sensor implementations, and
- the *SoftRobotRCC* itself, including the basic RPI interpeter and primitives as well as additional primitives and drivers for concrete devices.

When comparing the LOC (vertical axis) of the four Java-based layers and the SoftRobotRCC layer, it can be observed that the biggest part of the generic core component of the architecture is implemented inside the Robotics API. When introducing the generic Robot Arm 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 a UDP protocol, uniVAL for the Staubli based on CANopen and Ethercat). The picture is similar for the generic Robot Base extension and the concrete RMP50 and youBot Base extension components. The Activity support for robot bases is still rather preliminary and thus the numbers in the Robot Base Activity packages are expected to increase, while the RPI transformation should be quite complete already.

As a second result, note 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, however, already provided by the *SoftRobot basics* component. This generic code is responsible for the transformation of all basic models of the Robotics API. This code is reused by all extensions to the Robotics API that rely on those models. The *Robot Arm extension* contains a considerable amount of RPI transformation code as well. This is due to the introduction of various Actuators and Actions to model and operate robot arms. However, the elaborate generic model of robot arms and the according generic RPI transformation rules considerably reduce the effort for further extensions that support concrete robot types.

The Staubli *TX90 extension* only requires minimal Java code in the Robotics API. This code merely 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 Robot Arm extension. In particular, no lines of code are required for RPI transformation and the adapter to the SoftRobotRCC. Almost the same applies to the UR extension. Some code had to be added to support the robot's "Freedrive" mode, which allows it to be moved by hand. As this mode should be usable in applications, the UR extension contributes appropriate Action and Activity implementations. The KUKA Lightweight Robot with its integrated torque sensors and different controller modes required additional code on all Java layers.

Though the LOC measure is not a precise way to measure reusability, the clear results indicate a very high degree of reusability of the generic part of the SoftRobot architecture and in particular the Robotics API. 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 application examples[12], one type of robot could have been replaced by another type by literally just replacing the physical hardware.

11.5 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 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 is canceled when the robot moved too close to a fixed obstacle, causing the robot to brake immediately;
- a linear motion of two synchronized arms to a pre-defined cartesian goal, using the MultiArmRobot implementation.

All the abovementioned 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, which was excluded from the measurements.

In the sensor guarded point-to-point motion, the robot's encoders and the forward kinematics funtion 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 canceled. 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 [13] installed. The SoftRobot RCC was hosted as a real-time process on this PC and communicated with the robots via a dedicated network adapter. The Robotics API benchmark application was executed on the same PC, but with lower priority.

[12] Excluding the force-based manipulation operations in the Assembly Cell application, admittedly.
[13] `http://www.xenomai.org/`

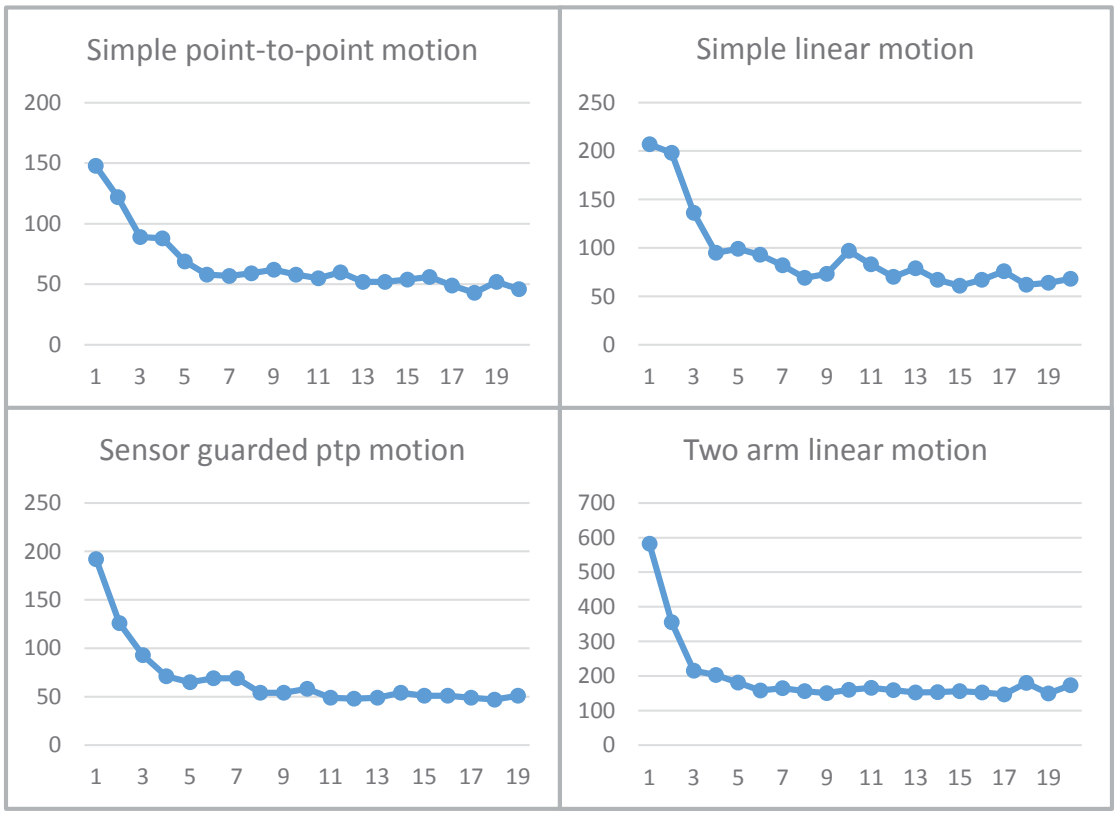

Figure 11.12: Time needed to start different types of Activities. Each type of Activity was executed multiple times in a loop (horizontal axis), and the duration until the RCC started the generated Command was measured (vertical axis, values in milliseconds). Adapted from [31].

Fig. 11.12 shows the benchmark results. In all tests, the performance increased significantly after the third to fifth motion had been executed. The hypothesis is that this phenomenon is caused by the Just-In-Time compiler in the Java Virtual Machine. This behavior also occurred consistently when a mixed series of point-to-point and linear motions was executed and those motions were parameterized with various goals. It is expected that this behavior can be observed in practical applications as well. Thus, it was included in the presented results.

Overall, basic motions take roughly 50-200ms 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 MultiArmRobot implementation that was employed to execute synchronous linear motions for two RobotArms takes roughly double the time to schedule a motion compared to the single arm case.

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 MultiArmRobot may be limiting performance in this respect. A more detailed evaluation indicated that only about 20-30% of the scheduling time was consumed by planning the motion operation. The rest of the time was needed to transform the Command to an RPI primitive net specification, serialize this graph, transmit it to the RCC via network and build the appropriate RPI primitive net instance on the RCC. To improve performance, the implementation of the MultiArmRobot could 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.

11.6 Realization of Requirements

In the first part of this work, a set of functional and non-functional requirements were presented. Those requirements target the SoftRobot architecture as a whole. However, almost all of them directly affect the Robotics API as well. This section will resume how each requirement is met by the Robotics API.

Realization of the functional requirements

In the following, each functional requirement will be revisited and the Robotics API's realization of this requirement will be discussed.

FR-1: **Drive robot arms along pre-defined paths in joint space.** This functionality is realized directly by the *PtpInterface* defined in the Robot Arm Extension, which provides Activities for such kind of motions.

FR-2: **Driving robot arms along pre-defined paths in Cartesian space.** Analogously to the previous requirement, Activities for such motions are provided by ActuatorInterfaces of the Robot Arm Extension. The relevant ActuatorInterfaces are *LinearMotionInterface* and *SplineMotionInterface*.

The ActuatorInterfaces mentioned above resemble specifications of the basic motion types supported by many industrial robot controllers today. All Extensions that provide support for concrete robot arms are expected to support those motions. As Actuators in the Robotics API are modeled as a dynamic composition of ActuatorInterfaces, all Actuators can be extended by additional operations anytime.

FR-3: **Specify an arbitrary number of interdependent points in Cartesian space.** The Robotics API's world model defines means for representing arbitrary points

as Frames and specifying various kinds of relationships between them. Explicit repositioning of a Frame by altering values of a certain relationship will preserve the relationships of all other connected Frames. Furthermore, the integrated model of geometric sensors even allows for evaluating the dynamic behavior of Frames during hard real-time execution of operations.

FR-4: **Motions should have parameterizable profiles.** ActuatorParameters are a generic mechanism to define all kinds of parameters that affect the execution of operations. The RobotArm Extension defines a set of ActuatorParameters for specifying joint-specific and Cartesian parameters for motion operations, like maximum acceleration and velocity. Compliance to this parameters has to be ensured by implementations of Activities and Actions.

FR-5: **Blending across multiple motions.** If blending is specified by the application developer for a certain motion Activity, it provides information about the dynamic state of the robot at a pre-defined blending condition to a subsequent Activity. This Activity may decide to blend the motion. The elaborate Activity scheduling algorithm then employs a real-time compliant mechanism provided by the SoftRobotRCC for a seamless transition between both operations.

The above requirements put a particular focus on pre-planned motions, as those are the predominant kinds of robot operations supported by today's controllers. However, the Robotics API's world model and Activity model are *general models*, and the realization of the three requirements stated above can be viewed as merely particular instantiations of those models. The Activity model is able to describe compatibility of arbitrary successive operations, which might depend on various state properties of Actuators (e.g., force applied to the environment, currently selected control mode).

FR-6: **Operate multiple devices with one program.** To a great extent, this requirement could be achieved by applying natural characteristics of object orientation, i.e. creating classes that represent certain devices and thus enabling developers to create multiple instances of it. However, to operate those devices independently of each other, the model of operations considers the correct scope (the device instance that should be controlled) in all of its parts.

FR-7: **Synchronize arbitrary device actions based on timing conditions.** The Robotics API's Command model provides the required synchronization mechanisms based on the State concept and gives tight timing guarantees. Clock-based timing delays can be achieved by combining EventHandlers and WaitCommands.

FR-8: **Trigger arbitrary device actions based on the progress of motions.** The time progress of motions can be discretized by particular States, which are constructed by motion Actions (and Activities). Based on those States, arbitrary reactions can be achieved with the Command model.

FR-9: **Geometrically synchronized actions.** This requirement refers to the ability to move devices relative to other, dynamic entities. The most important mechanism to realize this is the Robotics API's Sensor model, in particular the geometric Sensors provided by the world model. Implementations of Actions can rely on this to realize geometrically synchronized operation.

FR-10: **Querying the measurement of sensors and monitoring them.** Robotics API Sensors, which are used to access measurements of physical sensors, support querying of single measurement values. Furthermore, they act as publisher, delivering regular up-to-date measurements to registered subscribers.

FR-11: **Post-process sensor measurements.** Operations for post-processing sensor measurements are provided for various Sensors of basic types, e.g. boolean-type Sensors or double-type Sensors. Sensors measuring data of complex types may provide additional postprocessing methods.

A particular strength of the Robotics API's Sensor model is its real-time compliance. Thus, Sensors may not only be used to query data in the application itself, but may also be used as parameters for real-time critical operations. This applies to basic Sensors as well as derived Sensors that do post-processing of raw measurements.

FR-12: **Trigger arbitrary device actions based on sensor measurements.** All Sensors can define States whose activeness is determined by the respective sensor's measurements. All Sensors of basic type provide appropriate States (e.g. double-type Sensors provide a State that is active when the measurements exceed a certain threshold).

FR-13: **Guard robot motions using sensors.** States can be used to guard execution of Commands and Activities. Sensor-based States are a special case of general States.

FR-14: **Sensor-guided motions.** The real-time compliant nature of Sensors allows runtime-specific implementations of Actions to access up-to-date values of all Sensors and use those values to control their execution.

In sum, the concepts provided by the Robotics API are generic and flexible enough to satisfy all functional requirements that were initially specified. Furthermore, the same concept (e.g. State) could be employed to satisfy different requirements (e.g. FR-7, FR-8 and FR-12). Thus, the API could be kept relatively slim and provides developers with flexible and combinable concepts.

Meeting non-functional requirements

In this part, the degree to which the Robotics API meets the postulated non-functional requirements will be discussed.

NFR-1: **Usability.** Usability has not been investigated formally by e.g. some sort of user study. Evidence of a good usability of the Robotics API for developing robotics applications could be collected in projects with students of various educational levels. In a so-called 'hackathon', a group of undergraduate students of computer science was given access to several KUKA youBots, each equipped with one arm and a mobile base. During one week, they were allowed to perform experiments and develop applications on their own. None of them had experience in robotics. Even though, they were all able to successfully develop applications after a short introduction to the robots and the Robotics API (about 2 hours). The developed applications e.g. enable a youBot to write words on a whiteboard with coordinated movement of robot arm and base, or pick up distant objects from the floor. Interestingly, the students all preferred the Robotics API over a low-level C++ based interface provided by Locomotec, the company who distributes the youBot.

NFR-2: **Performance.** The Activity scheduling performance of the SoftRobot reference implementation has been analysed in Sect. 11.5. From this analysis, it can be concluded that for typical industrial robot applications of today, i.e. applications in which multiple robots perform pre-programmed sequences of operations independently of each other, the performance is good enough to achieve similar cycle time like with standard industrial robot controllers and programming languages. Exceptions might be applications with many motions of very short duration (less than 200ms). In this case, blending between motions might fail in many cases due to insufficient scheduling performance, which could reduce cycle times.

Timing and geometric accuracy depends mainly on two factors: The most crucial factor is the ability of the Robot Control Core to hold all execution timing bounds. This directly affects all timing guarantees of the Robotics API's Command model. The SoftRobotRCC is able to satisfy this in theory as well as in all practical experiments. More detailed results on this will be presented in Michael Vistein's dissertation. On the other hand, the geometric accuracy of pre-planned robot motions naturally depends on the quality of the plans. Though the plans contained in the reference implementation were merely intended to serve as a proof of concept, experiments with a highly precise tracking system have shown that the path accuracy of simple Cartesian-space motions is comparable to that of the same motion executed by KRL on the KRC. Those experiments did not investigate complex motions nor edge cases, in which the motion planning algorithms of the KRC are expected to perform significantly better. However, the results show that the SoftRobot approach in general and the Robotics API design in particular is able to meet the performance of today's industrial robot controllers.

NFR-3: **Robustness.** To achieve a high level of robustness of a software system, various methods like intensive testing or formal analyses (e.g. Failure Modes and Effects analysis) can be applied. Though basic automated testing was performed for the SoftRobot reference implementation, the focus was not on achieving industrial-grade reliability. However, the robustness of a concrete robot system also depends

on the robustness of applications created by developers. To increase fail-safe behavior of applications, the Robotics API provides developers with means of real-time critical handling of errors. The real-time exception model of the Robotics API on the one hand mimicks the exception semantics of modern object-oriented programming languages, and is on the other hand integrated in the host language's exception mechanism. This allows for bringing the system back in a stable state with defined timing bounds and afterwards applying arbitrarily complex recovery strategies.

NFR-4: **Maintainability.** To foster maintainability, the Robotics API separates the main concerns in industrial robotics into five separate models: Device model, Sensor model, Command model, Activity model and World model. Additionally, interfaces, abstract and concrete implementations of certain concepts are separated into different packages with defined dependencies. Both approaches aim at providing a clear structure with low coupling between models and packages and high cohesion inside the distinct packages. Throughout this work, it has been shown that the provided interfaces and classes have a high potential for reusability, which is backed by the results given in Sect. 11.4. Additionally, a generic interface for extensions allows system integrators and robot manufacturers to easily integrate new devices, sensors and operations. A mechanism for configuring the devices that are used in an application decouples the application from many aspects of the concrete workcell setup (e.g., the exact type of robot used, the exact locations of the workpieces) and thus greatly increases reusability of applications.

NFR-5: **Testability.** Certain aspects of robotic applications can be tested offline with the SoftRobot architecture. For this purpose, the SoftRobotRCC allows to replace drivers for hardware devices by drivers that simulate the hardware devices. This is completely transparent for the Robotics API, thus a large part of the application workflow is testable without having to employ the real hardware devices. Additionally, Robotics API applications can profit from modern tools for automated testing, for example unit testing frameworks like JUnit for the Java platform. Convenient step-by-step testing of robot applications can be realized by powerful debuggers that are available for many modern programming languages and environments. Considering direct selection of motion records, experiments were performed with the Eclipse debugger to be able to step directly to arbitrary source code lines. Results are promising, with the obvious restraints considering e.g. undefined state of variables in code that was not executed. However, this essential problem is to the author's knowledge also not solved in industrial robot controllers.

Chapter 12

The Software Ecosystem Complementing the Robotics API

Summary: Modern programming languages promise not only to improve the development of robotics applications, but also to support robot manufacturers in providing better robotics-specific development tools to their customers. This chapter presents three examples, which have been published in [136], [137] and [27].

The first section presents an extension to the popular Eclipse IDE that integrates robotics-specific features in this platform. Section 12.2 introduces a graphical language for specifying Robotics API Commands that is based on Eclipse as well. Finally, Sect. 12.3 presents an approach for interpreting the KUKA Robot Language based on the Robotics API.

12.1 IDE Integration - the Robotics API Eclipse Plugin

Using a modern, general purpose programming language yields among others the possibility to profit from the powerful concepts of today's Integrated Development Environments (IDEs). The Eclipse IDE is often used for developing Java applications. Developers profit from powerful concepts like code completion, various refactoring concepts and integrated debuggers. Such IDEs are often also extensible to allow for integrating specific tools for various domains. In contrast, today's robot development environments shipped with commercial robot controllers usually provide less support e.g. for refactoring and debugging. In turn, they support specific concepts for teaching robot motions by manual robot movement, managing frames and robot tools and testing robot programs e.g. by stepwise execution. In the course of this work, the Eclipse IDE was extended by

a set of plugins. The goal was to provide tools for developing robot applications similar to those offered by todays robot controllers, but in a modern IDE.

The Eclipse IDE provides a user interface that consists of modular, rearrangeable components. Fig. 12.1 illustrates the composition of the Eclipse Workbench from *Views* and *Editors*. Views are single window parts that can be shown and hidden, resized and moved. A certain pre-defined layout of distinct view types is called *Perspective*. A certain type of Editor can be associated to certain types of files. For example, the Java Editor is associated with files that have the extension '.java'.

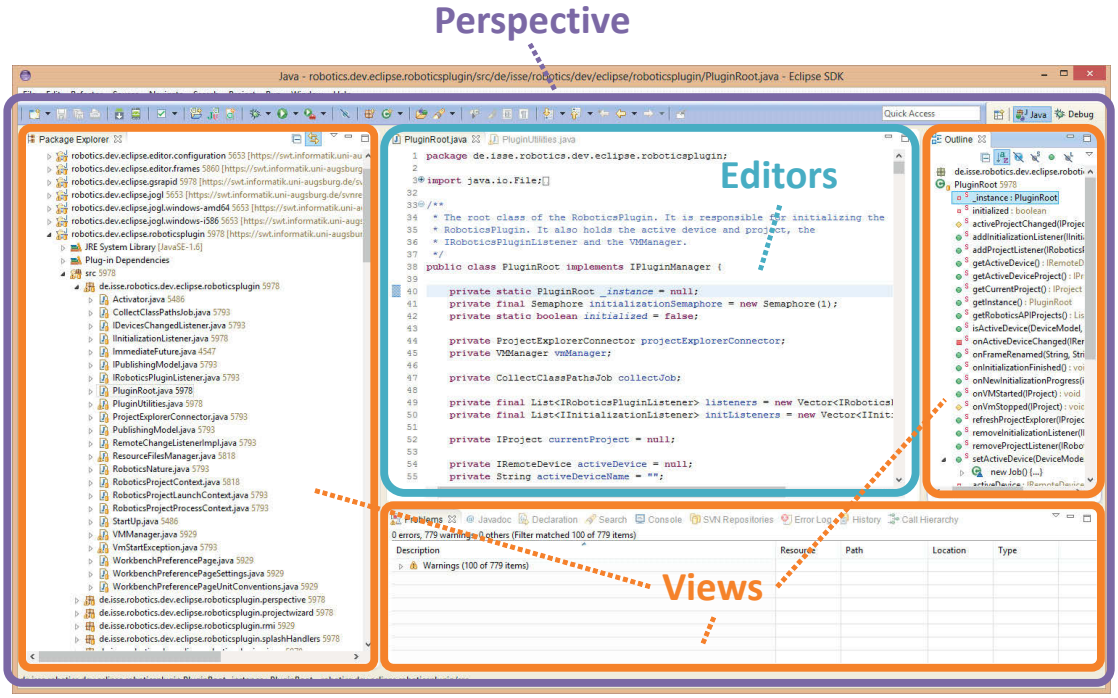

Figure 12.1: The Eclipse Workbench supports different Perspectives, which consist e.g. of Views and Editors. Adapted from [136].

The Robotics API Eclipse plugin extends the Eclipse IDE by the following robotics-specific features:

- A new type of project called 'Robotics API application',
- Views that show Frames and Devices configured in a robotics application,
- Editors to configure additional Devices and Frames,
- Views that allow developers to manually control robot arms and grippers,
- and a 'Robotics' perspective that arranges those robotics-specific views in a convenient way.

Some of the the Views contributed by the Robotics API plugin will be presented in detail. The left part of Fig. 12.2 shows the additional information provided by the Eclipse Project Explorer for Robotics API application projects. The Devices that are available in such projects are displayed in a special node called 'Devices'. A view called 'Cartesian Monitor' (middle part of Fig. 12.2) allows for displaying the Cartesian transformation between arbitrary connected Frames. The current position of all joints of a robot arm is displayed by the 'Joint View' (right part). A view called 'Jogging View' (bottom part) allows for manually moving robot arms using Cartesian jogging or joint-specific jogging.

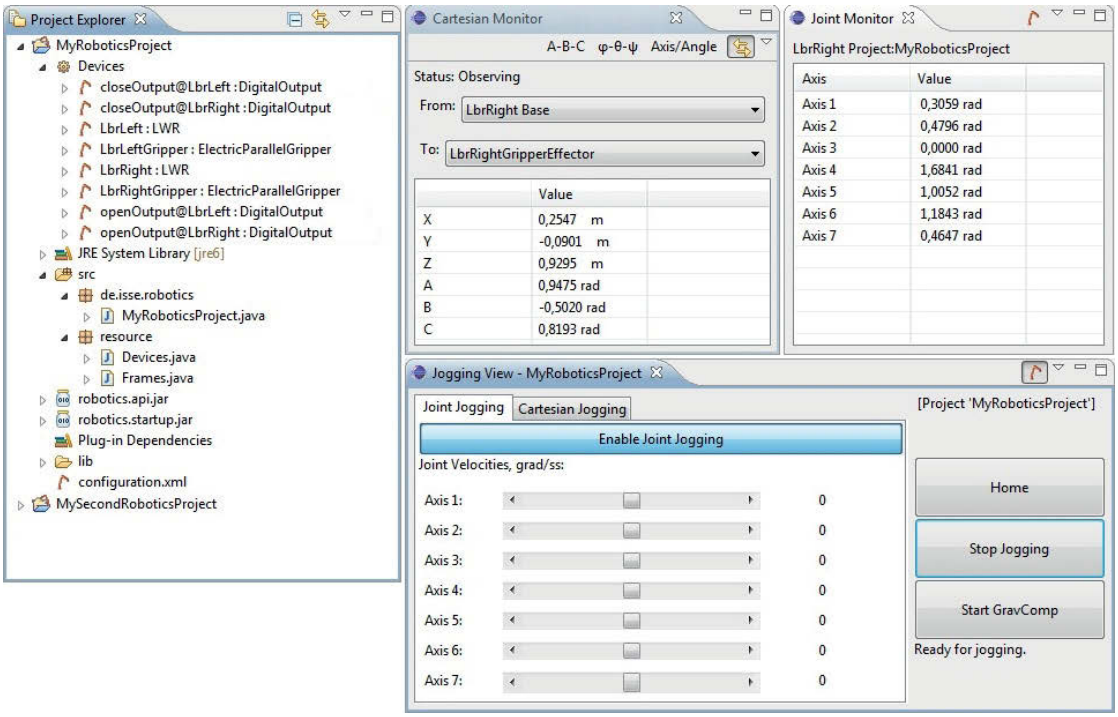

Figure 12.2: The Robotics API Plugin adds robotics-specific views to Eclipse. Adapted from [136].

Besides the abovementioned views, the Robotics API plugin also provides two kinds of Editors that are associated with the file 'configuration.xml' that is part of each Robotics API application project. The 'Devices' Editor allows for configuring new kinds of Robotics API Devices or modifying configurations. The 'Frames' Editor provides a structured view of the Frames that are configured in a Robotics API application. Relations between Frames can be altered, at least if they are non-dynamic. All configured Devices and Frames are persisted in the file 'configuration.xml', which is loaded by the Robotics API upon start of the developed application.

12.2 Graphical Modeling of Robotics API Commands

The Robotics API's command model is a key to flexible specification of real-time critical operations. Commands can be flexibly composed, and the event handling mechanism based on the State concept allows for reactive control of all kinds of Actuators. However, drawbacks considering usability have been identified (see Chap. 8). The introduction of the Activity concept creates a more intuitive interface for application developers, as a lot of common concepts for robotics applications are directly provided by all Activities. The command model still plays an important role, though. In particular, system integrators that extend the Robotics API by introducing new Actuators and Actions should also provide appropriate Activities. In some cases, this is possible by reusing existing Activities and combining them to new ones (e.g., for I/O based tools as presented in Chap. 9). In all other cases, however, new Activities have to be designed and their real-time critical execution logic has to be implemented based on Robotics API Commands. In such cases, developers can employ a graphical language for defining Commands. This section will give an overview about this language, the graphical editor that has been developed and the integrated Java code generator. Details can be found in [137].

A graphical formalism for Robotics API Commands

Fig. 12.3 shows an example of the graphical specification of a Robotics API Command. Graphical Robotics API Commands are specified inside a so called *Diagram*. A Diagram is considered the top-level of a graphical Command specification. To make a diagram valid, the following rules need to be followed:

- At least one Command (cf. Fig. 12.3, item marked 1) has to be present within the created Diagram.
- There has to be at least one entry-point for the command (cf. Fig. 12.3, 2)

In order to start a Command from one of the entry points, so called "StarterConnections" (cf. Fig. 12.3, 3) have to point to the Commands that shall be started first. Further Commands can be added to Diagrams as needed. TransactionCommands inside Diagrams are graphically modeled as special nestable items. They can themselves contain nestable items, in this case further Commands (TransactionCommands or RuntimeCommands). RuntimeCommands are also modeled as nestable items. In contrast to TransactionCommands, they can only contain exactly one Action and one Actuator as inner items. An example is shown in Fig. 12.3. The element labeled (1) is a RuntimeCommand with an Actuator (4) and an Action (5) nested inside.

The prerequisite for defining execution-time scheduling dependencies between Robotics API Commands are States. States can be provided by Commands, Actuators, Actions or Sensors. To reflect this in the graphical language, States can be nested inside the graphical operators that resemble those concepts.

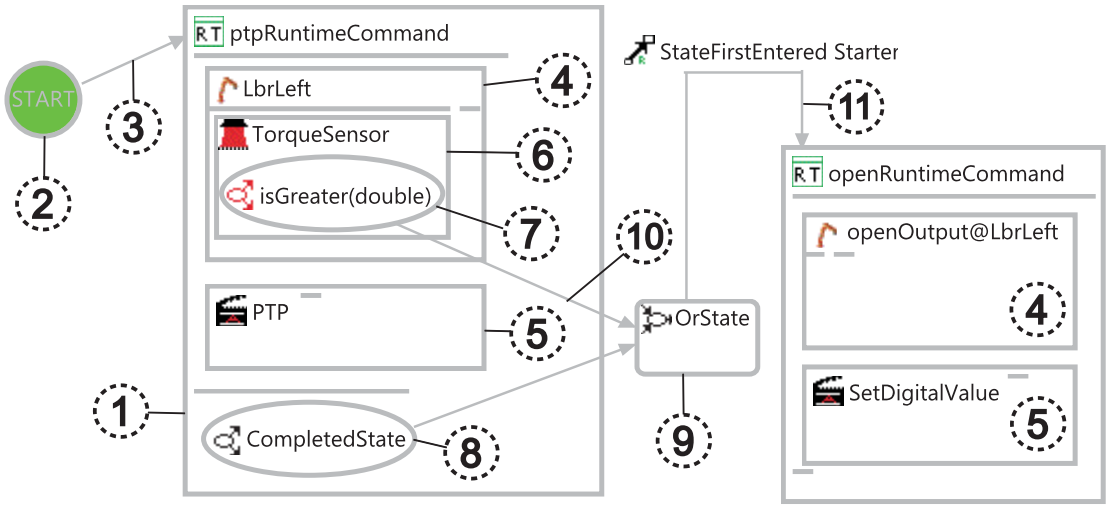

Figure 12.3: A Diagram defined in the graphical language. Adapted from [137].

Sensors themselves can be embedded in a Diagram in two ways: They can either be nested in an Actuator (cf. Fig. 12.3, 6), or, depending on the type of Sensor, be defined as a toplevel sensor in the Diagram. As States can be defined based on Sensor data (e.g. the State that a force sensor measures a value greater than 5N), graphical State items can be nested inside Sensor items (cf. Fig. 12.3, 7). States, however, are not limited to Sensors and can also be added to the graphical representations of Actions, Actuators and Commands. Fig. 12.3 shows a CommandState (8) which belongs to the RuntimeCommand and represents the state that this command has been completed.

In addition to the regular States, LogicalStates have a special graphical representation (cf. Fig. 12.3, 9). They are States that are derived from one or more other State(s). This deriviation is symbolized by the StateConnections in Fig. 12.3, 10. Like the other States, LogicalStates are connected to the commands they shall have an effect on by EventEffect connections (cf. Fig. 12.3, 11).

The EventHandler mechanism is the core concept for scheduling Commands in the graphical language. An EventHandler can be specified graphically by inserting an EventEffect connection originating from a State and targeting a Command. Further details concerning the scheduling are specified as properties of this connection and visualized as labels of the connection. These details include constraints specifying on which kind of event to react (State(First)Entered/State(First)Left) as well as the effect of the event. Possible effects are e.g. starting, stopping and canceling the targeted Command.

Based on the operator semantics described above, the schedule in Fig. 12.3 could be expressed as: "If the TorqueReachedSensor state has appeared for the first time at the Actuator LbrLeft or the RuntimeCommand *ptpRT* is in a completed state for the first time, start the RuntimeCommand *open*".

Graphical editing

To enable users to graphically create Robotics API Commands, an editor for the Eclipse IDE was developed. It consists of three important components (cf. Fig. 12.4):

1. The Tools palette (1), from which basic operators are selected
2. A working Canvas (2), where Diagrams are created
3. A Properties View (3), where specific attributes and parameters can be set for the currently selected graphical entity.

The common working flow is dragging and dropping an element from the Tools palette onto the Canvas (to the right hierarchy level inside the Diagram) and then setting its properties in the Properties View.

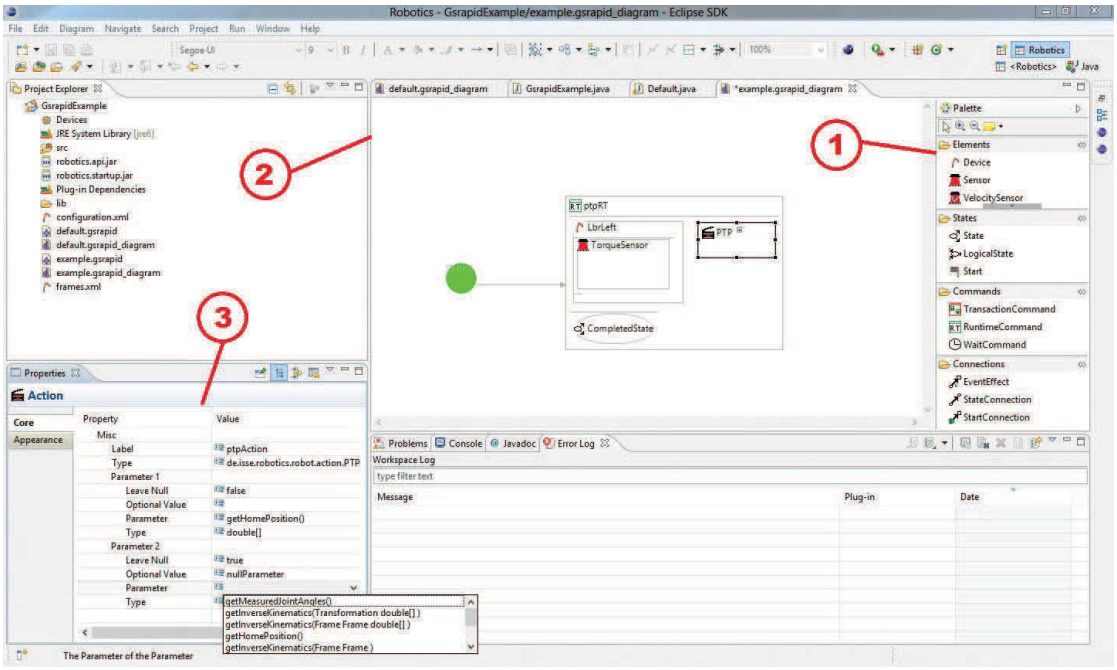

Figure 12.4: Plugin user interface for creating Robotics API Commands in the graphical language. Image source: [137].

A set of technologies and frameworks were used to realize the graphical editor. The *Eclipse Modeling Framework* (EMF, [138]) allows to generate a complete Eclipse plugin based on abstract model definitions. Thus, a lot of code required for managing model instances at program runtime can be generated and the generated code is guaranteed to work correctly, according to the model definitions. EMF also provides support for generating a properties view, which is frequently used when defining Robotics API Command Diagrams. In order to create a complete graphical modeling environment,

though, EMF is not sufficient. This is where the Eclipse Graphical Modeling Project (GMP, [139]) comes into place. It enhances the functionality of EMF by using and adapting components of Eclipse's Graphical Editing Framework (GEF, [140]). By defining three additional models, i.e. the graphical model, the tooling model and the mapping model, a stub for a completely functional graphical modeling plugin can be generated.

Code generation

Once Command Diagrams have been graphically specified, they can be translated to Java code to be usable inside a RoboticsAPI project. For this purpose, a code generator can be triggered by the user. Its main task is to interpret the graphical Command and the values specified for all properties. The challenge here is to correctly process every operator and all inter-operator dependencies (defined by nesting or operator connections). For this purpose, the instance of the EMF model corresponding to a Command Diagram is parsed and a Java Abstract Syntax Tree is generated with the help of Eclipse's Java AST/DOM framework (see [141]). Details on the code generation process can be found in [137].

12.3 KRL@RAPI - Interpreting the KUKA Robot Language with the Robotics API

A major reason for choosing the Java platform for the Robotics API's reference implementation was the large ecosystem of available tools. To demonstrate the potential of such a modern ecosystem compared to the proprietary ecosystem of traditional industrial robot controllers, a proof-of-concept support for the KUKA Robot Language was realized on top of the Robotics API. The chances and limitations of this approach are presented in this chapter.

To be able to parse and interpret KRL code, a language grammar first had to be derived. The procedure to determine a KRL grammar is described in [27]. Based on the grammar, a lexer and a parser for KRL code were generated with the tool ANTLR (ANother Tool for Language Recognition) [142]. Based on an abstract syntax tree of the parsed KRL code, an interpreter was created. This interpreter processes the workflow of the KRL program and respects all variable and method scoping rules specific to the language.

To execute motion instructions, the respective KRL function calls can be mapped to Robotics API Activities. At this point, the semantics of KRL and the Robotics API had to be carefully inspected. In contrast to KRL, the Robotics API and its underlying host language Java do not provide an advance run feature like KRL. However, the asynchronous execution semantics of Robotics API Activities can be employed to emulate an advance run of one[1] motion instruction. This has not been explained in the original work published in [27], as the Activity and Command scheduling mechanisms had not been fully developed at the time of publication. In contrast to KRL, the Robotics API and its Activity model do not enforce an artificial 'advance run stop' like KRL. However,

[1] The Activity scheduling algorithm allows for at most one Activity to be scheduled for execution

the developed KRL interpreter could reflect the original interpreter's behavior in this case.

To correctly handle KRL interrupts, the interpreter manages a stack of execution threads as described in [27]. Triggers are handled similar to interrupts by a separate interpreter thread as well. Thus, no real-time timing guarantees can be given for the execution of triggers and interrupts by the interpreter. It might be possible to map simple KRL triggers to the Robotics API Command event handling mechanism to achieve hard real-time execution. However, as KRL triggers may also call subroutines, this mapping is not possible in all cases. A possibility to execute KRL submit programs was not integrated in the Robotics API based interpreter.

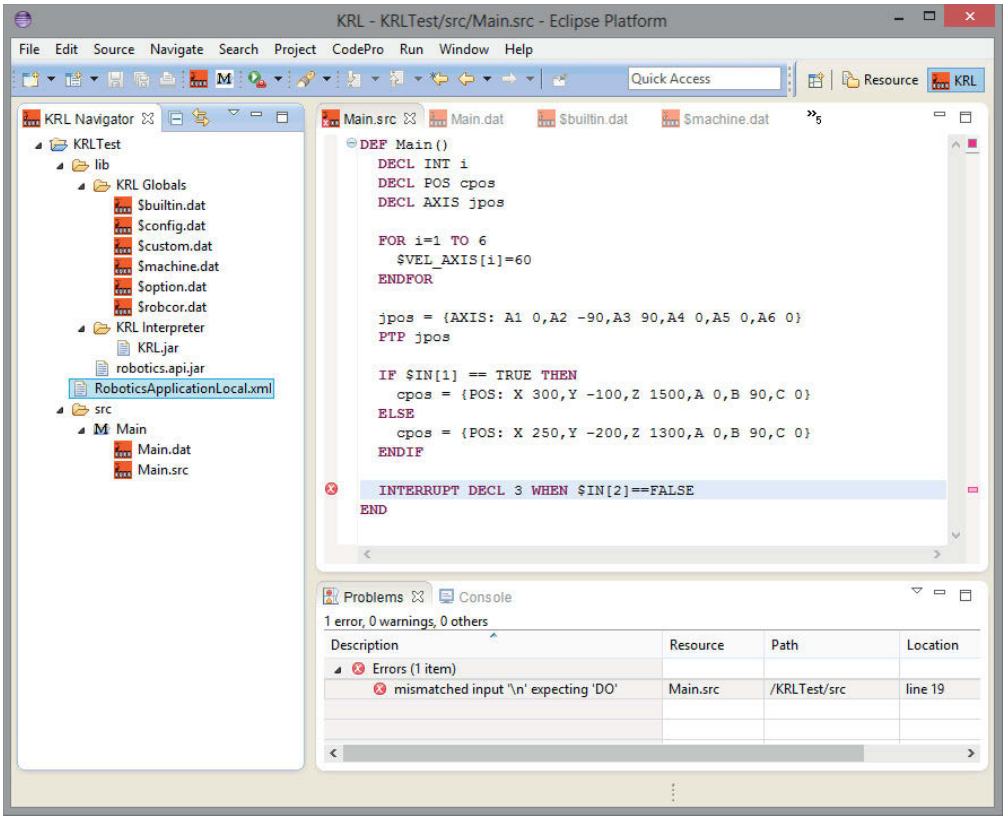

Figure 12.5: Support for the KUKA Robot Language in the Eclipse IDE based on the Robotics API.

The developed approach demonstrates that it is indeed possible to support a domain-specific language like KRL on top of the Robotics API with moderate effort (the work is mainly the result of Henrik Mühe's master thesis). It also shows that reproducing the exact execution semantics of KRL is hard to achieve. In particular, KRL's support for user code in triggers and interrupt handlers is problematic. However, the real-time guarantees that the original KRL interpreter provides for this user code are not clear.

In the course of the work on the KRL interpreter, support for KRL projects was developed in the Eclipse IDE. This includes project templates that reflect the KRL environment provided on the KRC, including all global data files. It also includes code highlighting and syntax checking. Fig. 12.5 shows an example of how KRL is embedded in Eclipse.

Chapter 13

Conclusion

To conclude the thesis, this chapter first summarizes the results that have been achieved and the lessons learned throughout the work on this thesis. The reference implementation of the Robotics API as well as the SoftRobotRCC are used as basis for further research projects at the Institute for Software & Systems Engineering. One of those projects is presented in the second part of this chapter. Finally, an outlook is given that outlines how a new robot programming paradigm called *object-centric programming* can be realized by combining the Robotics API's diverse models on a high level.

13.1 Summary and Lessons Learned

This thesis solved the research challenge of designing a language-independent Application Programming Interface for industrial robotics. This is an important contribution to lifting industrial robotics into the world of modern software engineering, which is achieved by the SoftRobot architecture as a whole. Today's industrial robot controllers mostly enforce developers to use a proprietary robot programming language to develop applications, which causes problems regarding maintainability and reusability and furthermore often leads to high system integration efforts. The languages also seem unable to efficiently support future challenges like multi-robot cooperation and sensor-based motion control. This situation has been examined in Chapter 2. Subsequently, Chapter 3 introduced three application examples that take up many of the aforementioned challenges and could thus only be realized with very high effort on traditional robot controllers.

The SoftRobot architecture is a radically new approach that fosters a separation of concerns between real-time critical robot control and high-level workflow of robotics applications. An in-depth analysis of the industrial robotics domain led to the insight that such a separation is feasible in this domain. A second result of the analysis is a set of requirements to a new software architecture, which was presented in Chapter 4. The same chapter also introduced the general architectural layers of the SoftRobot approach and their responsibilities.

From this point on, the thesis focused on the main contribution, which is the design of the *Robotics API*. This object-oriented framework is the key to efficient software development in industrial robotics. The Chapters 5 to 8 presented the structure of the framework and the design of its *core*, *world model* and *activity model* components. A large focus was put on separating concerns among those components and among the diverse models they introduce. An important lesson learned during this thesis is that separating the *conceptual components* into *programming language components* helps to make dependencies explicit and, in turn, limit those dependencies. This greatly contributes to modularity and reusability and forces software developers to constantly reason about whether the implementation thoroughly follows the software design.

Chapter 6 introduced the Robotics API's *device model*, *sensor and state model* and its *command model*. These three models form the basis for defining real-time critical, sensor-based operations of robotic devices in a very flexible way. This flexibility is well demonstrated by the thorough mechanism for real-time handling of exceptions in Robotics API Commands. This mechanism is realized solely by concepts of the Command model itself and is thus transparent to any real-time runtime environment. For application developers, however, this mechanism provides great support for creating robust applications.

The real-time guarantees w.r.t. Command execution are delegated to a Robot Control Core, which is adapted through the *RoboticsRuntime* interface that was also presented in Chapter 6. For the SoftRobotRCC, the implementation of this adapter transforms Robotics API Commands to RPI primitive nets. The work by Michael Vistein and Andreas Schierl shows that it is possible to transform instances of the extensive Command model to structurally simple RPI specifications. The simple execution semantics of RPI primitive nets in turn makes it easy to reason about real-time guarantees.

The Robotics API's *world model* (Chapter 7) allows to describe important locations in robotics applications, like e.g. where a robot arm is mounted, the current location of the robot's end-effector and the position and orientation of workpieces in the environment. Relative geometric relationships are modeled in an undirected graph. Mechanisms for querying the graph to find a way between arbitrary connected locations and their geometric relationships are provided. A very important contribution of the world model are geometric Sensors, which are modeled using the *sensor model*. This allows to integrate reactions in Commands that are based on geometric measurements. The world model provides geometric Sensors that can measure relative displacements as well as velocities.

The Robotics API's command model is employed in the *activity model* to provide developers with operation patterns recurring in many industrial robotics applications. The activity model was introduced in Chapter 8. The central concept, *Activity*, integrates means for specifying execution guards and further Activities that may be triggered during execution based on an event mechanism. Various ways of composing Activities are provided, which allows to express further operation patterns. The execution semantics of Activities is of particular importance, as it involves reasoning about meta data that describes the state of the devices controlled by an Activity. This allows Activities to

pre-plan their execution based on this state information. On the one hand, compatibility between subsequent operations can be checked on this basis. On the other hand, instantaneous transitions between operations can be planned. By carefully designing the Activity composition patterns mentioned above, this pre-planning capability is preserved even in complex composed operations.

The powerfulness and flexibility of the Robotics API is exemplified with concrete devices and their operations in Chapters 9 and 10. It is demonstrated that the device model can capture completely different kinds of robotic devices and that geometric properties of those devices can be modeled appropriately by concepts of the world model. In particular, the chapters demonstrate how Activities can model operations of robot tools and robot arms in a uniform way. Advanced robotic concepts like continuous execution of pre-planned motions (motion blending) can be realized based on the execution model of Activities. The most fascinating effect is that robot teams of arbitrary size can be tightly synchronized by merely composing the single devices and their operations. This level of composability is a result of the careful software design and is a unique feature of the Robotics API.

Chapter 11 demonstrates the adequacy of the Robotics API to realize the challenging application examples presented in Chapter 3. The chapter further evaluates performance and reusability aspects, which yields very positive results. In particular, the results of the reusability analysis confirm what has been exemplified in Chapters 9 and 10: The Robotics API's models allow for uniform modeling of diverse robotic devices, while at the same time maximizing reusability of generic implementations. The chapter concludes by elaborating on the contributions of the Robotics API to fulfilling all requirements to the SoftRobot architecture as a whole.

Modern programming languages not only provide powerful programming concepts, but often also community-driven tools and libraries. This is a significant difference to proprietary robot programming languages, for which usually only the robot manufacturer or a small community of robot users and system integrators provides some tools. The chances for more efficient software development for industrial robots have been illustrated in the realization of the application examples in Chapter 11. Chapter 12 adds further examples by presenting a set of development tools that were created during the work on this thesis. The chapter shows that it is even possible to interpret the KUKA Robot Language on top of the Robotics API, albeit with slight differences in execution semantics.

In sum, the thesis shows that it is worth to apply modern software engineering approaches to industrial robotics, even if this involves considerable effort and software architectures that are radically different from today's commercial solutions. The gains for application development can be enormous. This insight also started to spread throughout the robotics research community in recent years. One consequence is the establishment of a Journal of Software Engineering for Robotics (JOSER) that published its first issue in 2010. The central design elements that account for the unique character of the Robotics API have been accepted as a contribution to software engineering for robotics [31].

13.2 The Robotics API as Basis for an Offline Programming Platform

The reference implementation of the Robotics API and the SoftRobotRCC have reached a level of maturity that allows for employing them in further research projects. One of those projects, which is ongoing already, constructs a novel offline programming platform for automatic manufacturing of carbon fiber fabrics. Offline programming platforms in general aim at programming automation systems independent of the physical hardware, usually by means of simulation. Offline programming is often done to save time by creating the software for an automation system before this system has been physically built. Furthermore, in ongoing research on automated manufacturing of Carbon Fibre Reinforced Polymers (CFRP), the developed robot end-effectors for handling of raw carbon fibre plies are often too large in dimension to apply conventional teaching methods. Thus, offline programming methods are explored as an alternative.

The approach taken in the aforementioned research project is to simulate hardware components by simple simulation drivers in the SoftRobotRCC. This simulation approach is completely transparent to the Robotics API, which can use existing DeviceDrivers to access the simulated SoftRobotRCC devices. A custom 3D visualization environment for robot cells has been developed and is integrated in the new offline programming platform. The goal of this platform is to automatically calculate feasible solutions for many steps of the CFRP manufacturing process from existing specifications of final parts.

The SoftRobot architecture in general and the Robotics API in particular have shown to be an excellent basis for the offline programming platform. Based on the existing models in the Robotics API, various configurations of a CFRP manufacturing cell could be simulated and visualized. The cell contained a robot arm mounted on a linear unit and various complex CFRP handling tools. Some Robotics API extensions were refactored for better reusability, further refining the granularity of the device model. For example, an extension for generic multi-joint aggregates was extracted from the RobotArm extension. This new extension was used as basis for modeling the linear unit, which was possible with little effort.

The SoftRobot architecture is planned to be used in further research projects. One of its strong aspects, which is the ability to easily coordinate real-time cooperation in teams of robots, should be employed to control a large robot cell for experiments in CFRP manufacturing. This cell consists of a total of five robots that are mounted on various linear units, are equipped with various grippers and is intended to be used to perform collaborative manufacturing of large-scale CFRP fabrics. The issue of efficient application development for such a large team of heterogeneous robots is unsolved.

13.3 Outlook: Object-Centric Programming with the Robotics API

To give an outlook of what is possible by combining the device model, activity model and world model of the Robotics API, this section will conclude the thesis by outlining how the paradigm of *Object-centric Programming* can be realized with the Robotics API by simply combining existing concepts and a minimal extension to the World model. Object-centric programming is based on the idea that robots in industrial applications are merely means to an end for manipulating workpieces, and that the actual operations should be described in terms of the workpieces themselves [143]. In the Robotics API, operations are provided by ActuatorInterfaces. Now, lets assume that ActuatorInterfaces could be provided by other entities than just Actuators, e.g. by Frames and all other classes that model entities of the physical world. Let's focus on the Frame class. Extending it with the methods `addInterfaceFactory(...)` and `use(...)` is possible with minimal effort. The design of an appropriate ActuatorInterfaceFactory is a bit more sophisticated: Lets assume we want Frames to be able to perform linear motions in case they are directly or indirectly connected to a robot arm, for example because the workpiece they are attached to has been gripped by this robot arm's gripper. A Frame's ActuatorInterfaceFactory can retrieve all available RobotArms via the RoboticsRegistry and access their flanges. Then, it can perform a search in the Frame graph to find out whether the respective Frame is connected to one or more robot flanges. If this is not the case, the Frame cannot move and the ActuatorInterfaceFactory cannot create an ActuatorInterface for linear motion. However, if a RobotArm is found, this RobotArm's LinearMotionInterface can be used. By encapsulating the Frame in a MotionCenterParameter and adding it to the LinearMotionInterface's default parameters, this Frame will actually be the one that is moved. The approach even works when the Frame is connected to the flanges of multiple RobotArms: In this case, a MultiArmRobot device can be created on-the-fly to form a team of all RobotArms, and this team's LinearMotionInterface can be used just like in case of a single connected RobotArm.

In the design of the Robotics API, the focus was on separating concerns between the Device, Command/Activity and World model. Having separate models with minimal relationships between them preserves the freedom of how to combine them appropriately in various use cases. The above example demonstrates the powerfulness that lies in the combination of the models on a high level. There are unlimited possibilities for creating all kinds of "intelligent objects" that carry information about what operations they should participate in and how. Combining this with technologies like RFID to encode intelligence directly in the real-world objects could yield a completely new way of thinking about software design.

List of Abbreviations

API	Application Programming Interface
DOF	Degree Of Freedom
FRI	Fast Research Interface (for the Lightweight Robot)
IDE	Integrated Development Environment
IDE	Integrated Development Environment
ISSE	Institute for Software & Systems Engineering
JNI	Java Native Interface
JSPI	Java Service Provider Interface
KRC	KUKA Robot Controller
KRL	KUKA Robot Language
LWR	Lightweight Robot
RCC	Robot Control Core
RPI	Realtime Primitives Interface
SDK	Software Development Kit
UI	User Interface
UML	Unified Modeling Language
XML	Extensible Markup Language

List of Figures

Bibliography

[1] 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. 1, 4, 39, 45

[2] G. C. Devol, "Programmed article transfer," US Patent US 2 988 237 A, 1961. [Online]. Available: http://www.google.com/patents/US2988237 1

[3] G. P. Taylor, "An automatic block-setting crane – meccano model controlled by a robot unit," *Meccano Magazine*, vol. 23, no. 3, p. 172, Mar 1938. [Online]. Available: http://www.meccanoindex.co.uk/MMpage.php?MID=14912&id=1393021568 1, 2

[4] K. Zuse, *Der Computer - mein Lebenswerk (2. Aufl.)*. Springer, 1990. 2

[5] K. Čapek, *R.U.R. (Rossum's universal robots)*. Dover Publications, Mineola, N.Y., 2001. 2

[6] C. Shute. (2007, Sep) The Robot Gargantua. The Meccano Society Of Scotland. Pp. 10-17. [Online]. Available: http://www.meccanoscotland.org.uk/downloads/mag074_September_2007.pdf 2

[7] IFR Statistical Department. (2013) World Robotics 2013 - Executive Summary. International Federation of Robotics. Accessed Jan. 2014. [Online]. Available: http://www.worldrobotics.org/index.php?id=downloads 3, 4

[8] J. N. Pires, "New challenges for industrial robotic cell programming," *Industrial Robot*, vol. 36, no. 1, 2009. 4

[9] M. Hägele, T. Skordas, S. Sagert, R. Bischoff, T. Brogårdh, and M. Dresselhaus, "Industrial Robot Automation," European Robotics Network, White Paper, Jul. 2005. 4, 38

[10] A. Jain and C. Kemp, "EL-E: An Assistive Mobile Manipulator that Autonomously Fetches Objects from Flat Surfaces," *Autonomous Robots*, vol. 28, pp. 45–64, 2010. 4

[11] W. Meeussen, M. Wise, S. Glaser, S. Chitta, C. McGann, P. Mihelich, E. Marder-Eppstein, M. Muja, V. Eruhimov, T. Foote, J. Hsu, R. B. Rusu, B. Marthi, G. Bradski, K. Konolige, B. Gerkey, and E. Berger, "Autonomous door opening and plugging in with a personal robot," in *Int. Conference on Robotics and Automation*, 2010. 4

[12] M. Beetz, U. Klank, I. Kresse, A. Maldonado, L. Mösenlechner, D. Pangercic, T. Rühr, and M. Tenorth, "Robotic roommates making pancakes," in *11th IEEE-RAS International Conference on Humanoid Robots*, Bled, Slovenia, October, 26–28 2011. 4

[13] Coordination Action for Robotics in Europe (CARE). (2009, Jul) Robotic Visions to 2020 and Beyond - The Strategic Research Agenda for Robotics in Europe. European Robotics Technology Platform (EUROP). [Online]. Available: http://www.robotics-platform.eu/cms/index.php?idcat=26 4

[14] ISO/IEC/IEEE, "Systems and software engineering – Vocabulary," International Organization for Standardization, Standard 24765:2010, 2010. [Online]. Available: http://www.iso.org/ 5

[15] ISO/IEC, "Software Engineering – Guide to the Software Engineering Body of Knowledge (SWEBOK)," International Organization for Standardization, Standard 19759:2005, 2005. [Online]. Available: http://www.iso.org/ 5

[16] B. Siciliano and O. Khatib, "Introduction," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Berlin, Heidelberg: Springer, 2008, ch. 0, pp. 1–4. 5

[17] ABB Asea Brown Boveri Ltd. ABB Historical Milestones - About ABB Robotics. Accessed Jan. 2014. [Online]. Available: http://www.abb.com/product/ap/seitp327/583a073bb0bb1922c12570c1004d3e6b.aspx 5

[18] KUKA Roboter GmbH. KUKA Industrial Robots - 1973 The first KUKA robot. Accessed Jan. 2014. [Online]. Available: http://www.kuka-robotics.com/germany/en/company/group/milestones/1973.htm 5

[19] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics: Modelling, Planning and Control*, 1st ed. Springer Publishing Company, Incorporated, 2008. 11, 12, 13, 14, 15

[20] K. Waldron and J. Schmiedeler, "Kinematics," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Berlin, Heidelberg: Springer, 2008, ch. 1, pp. 9–33. 11, 12, 13

[21] ISO, "Robots and robotic devices – Vocabulary," International Organization for Standardization, Standard 8373:2012, 2012. [Online]. Available: http://www.iso.org/ 12

[22] ——, "Manipulating industrial robots – Mechanical interfaces – Part 1: Plates," International Organization for Standardization, Standard 9409-1:2004, 2004. [Online]. Available: http://www.iso.org 12

[23] T. Kröger and F. Wahl, "Online Trajectory Generation: Basic Concepts for Instantaneous Reactions to Unforeseen Events," *IEEE Transactions on Robotics*, vol. 26, no. 1, pp. 94–111, Feb 2010. 15, 182

[24] L. Villani and J. D. Schutter, "Force control," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Berlin, Heidelberg: Springer, 2008, ch. 7, pp. 161–185. 15

[25] T. Kröger and B. Finkemeyer, "Robot Motion Control During Abrupt Switchings Between Manipulation Primitives," in *Workshop on Mobile Manipulation at the IEEE International Conference on Robotics and Automation*, Shanghai, China, May 2011. 15

[26] *KUKA System Software 8.2 - Operating and Programming Instructions for System Integrators*, KUKA Roboter GmbH, 2012, Version: KSS 8.2 SI V4 en. 15

[27] H. Mühe, A. Angerer, A. Hoffmann, and W. Reif, "On reverse-engineering the KUKA Robot Language," *1st International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob '10), 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2010)*, 2010. 16, 217, 223, 224

[28] *KUKA.CR Motion Cooperation 2.1*, KUKA Robot Group, Aug 2007, Version: 2.1 08.07.00. 20

[29] *KUKA.RobotSensorInterface (RSI) 2.1*, KUKA Robot Group, May 2007, Version: 2.1. 20

[30] *KUKA.Ethernet RSI XML 1.1*, KUKA Robot Group, Nov 2007, Version: V1 de. 21

[31] A. Angerer, A. Hoffmann, A. Schierl, M. Vistein, and W. Reif, "Robotics API: Object-Oriented Software Development for Industrial Robots," *J. of Software Engineering for Robotics*, vol. 4, no. 1, pp. 1–22, 2013. [Online]. Available: http://joser.unibg.it/index.php?journal=joser&page=article&op=view&path=53 21, 55, 119, 163, 193, 208, 211, 229

[32] G. Hirzinger, J. Butterfass, M. Fischer, M. Grebenstein, M. Hahnle, H. Liu, I. Schaefer, and N. Sporer, "A mechatronics approach to the design of light-weight arms and multifingered hands," in *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, vol. 1, 2000, pp. 46–54 vol.1. 25

[33] R. Bischoff, J. Kurth, G. Schreiber, R. Koeppe, A. Albu-Schäffer, A. Beyer, O. Eiberger, S. Haddadin, A. Stemmer, G. Grunwald, and G. Hirzinger, "The KUKA-DLR lightweight robot arm - a new reference platform for robotics research and manufacturing," in *Proc. IFR Int. Symposium on Robotics (ISR 2010)*, 2010. 25

[34] KUKA Laboratories GmbH. KUKA Laboratories GmbH - Lightweight Robotics. Accessed Feb. 2014. [Online]. Available: http://www.kuka-labs.com/en/service_robotics/lightweight_robotics/ 25

[35] C. Ott, C. Ott, A. Kugi, and G. Hirzinger, "On the passivity-based impedance control of flexible joint robots," *Robotics, IEEE Transactions on*, vol. 24, no. 2, pp. 416–429, April 2008. 25

[36] G. Schreiber, A. Stemmer, and R. Bischoff, "The Fast Research Interface for the KUKA Lightweight Robot," in *Workshop on Innovative Robot Control Architectures for Demanding (Research) Applications. IEEE Intl. Conf. on Robot. & Autom.*, Anchorage, Alaska, USA, May 2010. 25

[37] IEC 61800-7-201, *Adjustable speed electrical power drive systems - Part 7-201: Generic interface and use of profiles for power drive systems - Profile type 1 specification.* ISO, Geneva, Switzerland, 2007. 25

[38] 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 Inform. in Control, Autom. & Robot., Rome, Italy*, 2012. 26, 193, 199, 202

[39] .NET Framework. Microsoft Corporation. Accessed Feb. 2014. [Online]. Available: http://msdn.microsoft.com/netframework/ 26, 198

[40] B. Siciliano and O. Khatib, Eds., *Springer Handbook of Robotics.* Berlin, Heidelberg: Springer, 2008. 27, 97, 98, 99, 102, 117

[41] P. Skrzypczyliski, "Supervision and teleoperation system for an autonomous mobile robot," in *Proc. 1997 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, vol. 2, Sep. 1997, pp. 1177 –1181 vol.2. 27

[42] L. Zalud, "ARGOS - system for heterogeneous mobile robot teleoperation," in *Proc. 2006 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, Oct. 2006, pp. 211 –216. 27

[43] M. Micire, M. Desai, J. L. Drury, E. McCann, A. Norton, K. M. Tsui, and H. A. Yanco, "Design and validation of two-handed multi-touch tabletop controllers for robot teleoperation," in *Proc. 15th Int. Conf. on Intelligent User Interfaces*, ser. IUI '11. New York, NY, USA: ACM, 2011, pp. 145–154. [Online]. Available: http://doi.acm.org/10.1145/1943403.1943427 27

[44] T. Seifried, M. Haller, S. D. Scott, F. Perteneder, C. Rendl, D. Sakamoto, and M. Inami, "CRISTAL: a collaborative home media and device controller based on a multi-touch display," in *Proc. ACM Int. Conf. on Interactive Tabletops and Surfaces*, ser. ITS '09. New York, NY, USA: ACM, 2009, pp. 33–40. 27

[45] *Electric Parallel Gripper MEG 50 EC and Controller MEG C 50*, SCHUNK GmbH & Co. KG, Jul 2011. [Online]. Available: http://www.schunk.com/schunk_files/attachments/OM_AU_MEG50-EC__EN.pdf 28, 29

[46] 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. 34

[47] 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. 34

[48] A. Hoffmann, A. Angerer, F. Ortmeier, M. Vistein, and W. Reif, "Hiding real-time: A new approach for the software development of industrial robots," in *Proc. 2009 IEEE/RSJ Intl. Conf. on Intell. Robots and Systems, St. Louis, Missouri, USA*. IEEE, Oct. 2009, pp. 2108–2113. 35, 42

[49] A. Hoffmann, A. Angerer, A. Schierl, M. Vistein, and W. Reif, "Towards object-oriented software development for industrial robots," in *Proc. 7th Intl. Conf. on Inform. in Control, Autom. & Robot. (ICINCO 2010), Funchal, Madeira, Portugal*, vol. 2. INSTICC Press, Jun. 2010, pp. 437–440. 35

[50] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*, 4th ed. USA: Addison-Wesley Educational Publishers Inc, 2009. 36

[51] Hub Technologies. KUKA Robotics. Accessed Jan. 2014. [Online]. Available: http://www.kuka-robotics.com/germany/en/products/software/hub_technologies 38

[52] C. Bredin, "ABB MultiMove functionality heralds a new era in robot applications," *ABB Review*, vol. 1, pp. 26–29, 2005. 38

[53] R. B. Grady and D. L. Caswell, *Software Metrics: Establishing a Company-wide Program*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1987. 40

[54] S. Y. Nof, Ed., *Handbook of Industrial Robotics*, 2nd ed. New York: John Wiley & Son, 1999. 41

[55] IEEE, "IEEE Std 610. IEEE Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries," The Institute of Electrical and Electronics Engineers, Tech. Rep., 1991. 41

[56] ISO/IEC, "Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models," International Organization for Standardization, Standard 25010:2011, 2011. [Online]. Available: http://www.iso.org/ 41

[57] 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 Advanced Robotics, D. Brugali, Ed. Springer - Verlag, 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 41

[58] H. Bruyninckx, "Open robot control software: the OROCOS project," in *Proc. 2001 IEEE Intl. Conf. on Robot. & Autom.*, Seoul, Korea, May 2001, pp. 2523–2528. 41, 43

[59] M. Vistein, A. Angerer, A. Hoffmann, A. Schierl, and W. Reif, "Interfacing industrial robots using realtime primitives," in *Proc. 2010 IEEE Intl. Conf. on Autom. and Logistics, Hong Kong, China*. IEEE, Aug. 2010, pp. 468–473. 42

[60] ——, "Flexible and continuous execution of real-time critical robotic tasks," *International Journal of Mechatronics and Automation*, vol. 4, no. 1, pp. 27–38, 1 2014. 42, 43, 86

[61] ——, "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. 44

[62] VxWorks RTOS. Wind River. Accessed Jan. 2014. [Online]. Available: http://www.windriver.com/products/vxworks/ 45

[63] W. Mahnke and S.-H. Leitner, "OPC Unified Architecture," *ABB Technik*, pp. 56–61, Mar 2009. [Online]. Available: http://www05.abb.com/global/scot/scot271.nsf/veritydisplay/43fd4ef093cb19c0c125762f006cbd96/$file/56-61%203M903_GER72dpi.pdf 45

[64] Intel. (2010) Build a better robot, and automation gets easy. Intel Corporation. Accessed Jan. 2014. [Online]. Available: http://www.intel.cn/content/dam/doc/case-study/industrial-core-kuka-study.pdf 45

[65] R. P. Bonasso, D. Kortenkamp, D. P. Miller, and M. Slack, "Experiences with an architecture for intelligent, reactive agents," *J. of Experimental and Theoretical Artificial Intelligence*, vol. 9, pp. 237–256, 1995. 46

[66] S. Yu, M. Slack, and D. Miller, "A streamlined software environment for situated skills," in *Meeting Paper Archive*. American Institute of Aeronautics and Astronautics, Mar. 1994, pp. –. [Online]. Available: http://dx.doi.org/10.2514/6.1994-1205 46

[67] R. J. Firby, "Adaptive execution in complex dynamic worlds," Ph.D. dissertation, Yale University, New Haven, CT, USA, 1989, AAI9010653. 46

[68] D. Simon, E. C. Castaneda, and P. Freedman, "Design and analysis of synchronization for real-time closed-loop control in robotics," *IEEE Transactions on Control Systems Technology*, vol. 6, no. 4, pp. 445–461, Jul. 1998. 46, 92

[69] R. Simmons, T. Smith, M. B. Dias, D. Goldberg, D. Hershberger, A. Stentz, and R. Zlot, "A layered architecture for coordination of mobile robots," in *Multi-Robot Systems: From Swarms to Intelligent Automata, Proceedings from the 2002 NRL Workshop on Multi-Robot Systems*. Kluwer Academic Publishers, May 2002. 46

[70] D. Brugali and P. Scandurra, "Component-based robotic engineering (Part I)," *IEEE Robot. & Autom. Mag.*, vol. 16, no. 4, pp. 84–96, 2009. 46

[71] D. Brugali and A. Shakhimardanov, "Component-based robotic engineering (Part II)," *IEEE Robot. & Autom. Mag.*, vol. 20, no. 1, Mar. 2010. 46

[72] A. Makarenko, A. Brooks, and T. Kaupp, "Orca: Components for Robotics," in *Workshop on Robotic Standardization. IEEE/RSJ Intl. Conf. on Intell. Robots and Systems*, Beijing, China, Oct. 2006. 46

[73] N. Ando, T. Suehiro, K. Kitagaki, and T. Kotoku, "RT(Robot Technology)-Component and its Standardization - Towards Component Based Networked Robot Systems Development," in *Proc. International Joint Conference SICE-ICASE*. Bexco, Busan, Korea: AIST, Oct. 2006, pp. 2633–2638. 46

[74] 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 Intell. Robots and Systems*, San Diego, USA, Oct. 2007, pp. 76–90. 46

[75] 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. 46

[76] M. Radestock and S. Eisenbach, "Coordination in evolving systems," in *Trends in Distributed Systems CORBA and Beyond*, ser. Lecture Notes in Computer Science, O. Spaniol, C. Linnhoff-Popien, and B. Meyer, Eds. Springer Berlin Heidelberg, 1996, vol. 1161, pp. 162–176. [Online]. Available: http://dx.doi.org/10.1007/3-540-61842-2_34 46

[77] E. Prassler, H. Bruyninckx, K. Nilsson, and A. Shakhimardanov, "The use of reuse for designing and manufacturing robots," Robot Standards and reference architectures (RoSta) consortium, White Paper, June 2009. [Online]. Available: http://www.robot-standards.eu/Documents_RoSta_wiki/whitepaper_reuse.pdf 46

[78] M. Klotzbucher and H. Bruyninckx, "Coordinating robotic tasks and systems with rFSM statecharts," *Journal of Software Engineering for Robotics*, vol. 3, no 1, pp. 28–56, 2012. 46, 47

[79] J. Bohren and S. Cousins, "The SMACH high-level executive," *IEEE Robotics & Automation Magazine*, vol. 17, no. 4, pp. 18–20, Dec 2010. 46, 47

[80] J. Bohren. smach - ROS Wiki. Accessed Jan. 2014. [Online]. Available: http://ros.org/wiki/smach 46

[81] "State Chart XML (SCXML): State Machine Notation for Control Abstraction," W3C, Dec. 2012. [Online]. Available: http://www.w3.org/TR/scxml/ 47

[82] Commons SCXML. The Apache Software Foundation. Accessed Jan. 2014. [Online]. Available: http://commons.apache.org/proper/commons-scxml/ 47

[83] M. Jüngel, E. Kindler, and M. Weber, "Towards a Generic Interchange Format for Petri Nets - Position Paper," in *Proceedings of 21st ICATPN Meeting on XML/SGML based Interchange Formats for Petri Nets*, 2000, pp. 1–5. 47

[84] A. Arcoverde, Jr., G. Alves, Jr., and R. Lima, "Petri nets tools integration through eclipse," in *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*, ser. eclipse '05. New York, NY, USA: ACM, 2005, pp. 90–94. [Online]. Available: http://doi.acm.org/10.1145/1117696.1117715 47

[85] D. Xu, "A tool for automated test code generation from high-level petri nets," in *Proceedings of the 32Nd International Conference on Applications and Theory of Petri Nets*, ser. PETRI NETS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 308–317. [Online]. Available: http://dl.acm.org/citation.cfm?id=2022192.2022212 47

[86] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented software.* Addison Wesley, 1994. 50, 59, 65, 67, 74, 81, 91, 122, 124, 136, 169, 195

[87] *UML 2.4.1*, Object Management Group Std., Aug. 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1/ 50

[88] A. Angerer, A. Hoffmann, A. Schierl, M. Vistein, and W. Reif, "The Robotics API: An object-oriented framework for modeling industrial robotics applications," in *Proc. 2010 IEEE/RSJ Intl. Conf. on Intell. Robots and Systems, Taipeh, Taiwan.* IEEE, Oct. 2010, pp. 4036–4041. 55

[89] A. Angerer, M. Bischof, A. Chekler, A. Hoffmann, W. Reif, A. Schierl, C. Tarragona, and M. Vistein, "Objektorientierte Programmierung von Industrierobotern," in *Proc. Internationales Forum Mechatronik 2010 (IFM 2010), Winterthur, Switzerland*, Nov. 2010. 55

[90] G. Langholz, A. Kandel, and J. Mott, *Foundations of Digital Logic Design*. World Scientific, 1998. [Online]. Available: http://books.google.de/books?id=4sX9fTGRo7QC 64

[91] 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 Inform. in Control, Autom. & Robot., Rome, Italy*, 2012. 85, 86

[92] P. Kriens and B. Hargrave, "Listeners considered harmful: The whiteboard pattern," OSGi Alliance, Whitepaper, 2004. 87

[93] R. Seacord, "Replaceable components and the service provider interface," in *COTS-Based Software Systems*, ser. Lecture Notes in Computer Science, J. Dean and A. Gravel, Eds. Springer Berlin Heidelberg, 2002, vol. 2255, pp. 222–233. [Online]. Available: http://dx.doi.org/10.1007/3-540-45588-4_21 90

[94] Oracle. ServiceLoader (Java Platform SE 7). Oracle Corporation. Accessed Mar. 2014. [Online]. Available: http://docs.oracle.com/javase/7/docs/api/index.html?java/util/package-summary.html 90

[95] 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. 92

[96] E. Coste-Maniere and N. Turro, "The maestro language and its environment: specification, validation and control of robotic missions," in *Intelligent Robots and Systems, 1997. IROS '97., Proceedings of the 1997 IEEE/RSJ International Conference on*, vol. 2, Sep 1997, pp. 836–841 vol.2. 92

[97] G. Berry and G. Gonthier, "The ESTEREL synchronous programming language: design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992. 92

[98] D. J. Miller and R. C. Lennox, "An object-oriented environment for robot system architectures," *IEEE Control Syst. Mag.*, vol. 11, no. 2, pp. 14–23, 1991. 93

[99] 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. 93

[100] M. S. Loffler, D. M. Dawson, E. Zergeroglu, and N. P. Costescu, "Object-oriented techniques in robot manipulator control software development," in *Proc. 2001 American Control Conference*, vol. 6, Arlington, USA, Jun. 2001, pp. 4520–4525. 94

[101] G. T. McKee, J. A. Fryer, and P. S. Schenker, "Object-oriented concepts for modular robotics systems," in *Proc. 39th Intl. Conf. and Exh. on Technology of Object-Oriented Languages and Systems*, Santa Barbara, USA, Jul. 2001, pp. 229–238. 94

[102] I. A. D. Nesnas, A. Wright, M. Bajracharya, R. Simmons, T. Estlin, and W. S. Kim, "CLARAty: An architecture for reusable robotic software," in *Proc. SPIE Aerosense Conference*, Orlando, USA, Apr. 2003. 94

[103] I. A. Nesnas, R. Simmons, D. Gaines, C. Kunz, A. Diaz-Calderon, T. Estlin, R. Madison, J. Guineau, M. McHenry, I.-H. Shu, and D. Apfelbaum, "CLARAty: Challenges and steps toward reusable robotic software," *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, pp. 23–30, 2006. 94

[104] B. P. Gerkey, R. T. Vaughan, K. Stoy, A. Howard, G. S. Sukhatme, and M. J. Mataric, "Most Valuable Player: A robot device server for distributed control," in *Proc. 2001 IEEE/RSJ Intl. Conf. on Intell. Robots and Systems*, Maui, Hawaii, USA, Oct. 2001, pp. 1226–1231. 94

[105] R. T. Vaughan, B. Gerkey, and A. Howard, "On device abstractions for portable, resuable robot code," in *Proc. 2003 IEEE/RSJ Intl. Conf. on Intell. Robots and Systems*, Las Vegas, USA, Oct. 2003, pp. 2121–2427. 94

[106] 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. 94

[107] H. Bruyninckx, P. Soetens, and B. Koninckx, "The real-time motion control core of the Orocos project," in *Proc. 2003 IEEE Intl. Conf. on Robot. & Autom.*, Seoul, Korea, May 2003, pp. 2766–2771. 95

[108] R. Smits, T. D. Laet, K. Claes, P. Soetens, J. D. Schutter, and H. Bruyninckx, "Orocos: A software framework for complex sensor-driven robot tasks," *IEEE Robotics & Automation Magazine*, 2009. 95

[109] R. Smits, "Robot Skills: Design of a constraint-based methodology and software support," Ph.D. dissertation, Katholieke Universiteit Leuven - Faculty of Engineering, Kasteelpark Arenberg 1, B-3001 Leuven (Belgium), May 2010. 95, 117

[110] M. Klotzbücher, P. Soetens, and H. Bruyninckx, "OROCOS RTT-Lua: an Execution Environment for building Real-time Robotic Domain Specific Languages," in *Proc. Intl. Conf. on SIMULATION, MODELING and PROGRAMMING for AUTONOMOUS ROBOTS (SIMPAR 2010) Workshops*, Nov 2010, pp. 284–289. 95

[111] A. Jain, "Unified formulation of dynamics for serial rigid multibody systems," *Journal of Guidance, Control, and Dynamics*, vol. 14, no. 3, pp. 531–542, May 1991. [Online]. Available: http://dx.doi.org/10.2514/3.20672 100, 117

[112] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001. 111

[113] G. Guennebaud, B. Jacob *et al.* (2010) Eigen v3. Accessed Jan. 2014. [Online]. Available: http://eigen.tuxfamily.org 117

[114] R. Smits. Orocos Kinematics and Dynamics. Accessed Jan. 2014. [Online]. Available: http://www.orocos.org/kdl 117

[115] T. D. Laet, S. Bellens, H. Bruyninckx, and J. D. Schutter, "Geometric relations between rigid bodies (Part 2) – From semantics to software," *IEEE Robot. & Autom. Mag.*, pp. 91–102, Jun 2013. 117

[116] T. Foote, E. Marder-Eppstein, and W. Meeussen. tf2 - ROS Wiki. Accessed Jul. 2013. [Online]. Available: http://wiki.ros.org/tf2 117

[117] W. Burgard and M. Hebert, "World modeling," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Berlin, Heidelberg: Springer, 2008, ch. 36, pp. 853–869. 117

[118] D. Shuey, D. Bailey, and T. Morrissey, "PHIGS: A Standard, Dynamic, Interactive Graphics Interface," *IEEE Comput. Graph. Appl.*, vol. 6, no. 8, pp. 50–57, Aug. 1986. [Online]. Available: http://dx.doi.org/10.1109/MCG.1986.276770 117

[119] M. Naef, E. Lamboray, O. Staadt, and M. Gross, "The Blue-c Distributed Scene Graph," in *Proceedings of the Workshop on Virtual Environments 2003*, ser. EGVE '03. New York, NY, USA: ACM, 2003, pp. 125–133. [Online]. Available: http://doi.acm.org/10.1145/769953.769968 117

[120] S. Blumenthal, H. Bruyninckx, W. Nowak, and E. Prassler, "A scene graph based shared 3D world model for robotic applications," in *Proc. 2013 IEEE Intl. Conf. on Robot. & Autom., Karlsruhe, Germany*, 2013, pp. 453–460. 117

[121] 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 121

[122] I. Lütkebohle, R. Philippsen, V. Pradeep, E. Marder-Eppstein, and S. Wachsmuth, "Generic middleware support for coordinating robot software components: The task-state-pattern," *Journal of Software Engineering in Robotics*, vol. 1, no. 2, pp. 20–39, 2011. [Online]. Available: http://joser.unibg.it/index.php?journal=joser&page=article&op=download&path%5B%5D=41&path%5B%5D=5 143

[123] B. Finkemeyer, "Robotersteuerungsarchitektur auf der Basis von Aktionsprimitiven," Doktorarbeit (in German), Technische Universität Braunschweig, Aachen, 2004. 143

[124] B. Finkemeyer, T. Kröger, and F. M. Wahl, "Executing Assembly Tasks Specified by Manipulation Primitive Nets," *Advanced Robotics*, vol. 19, no. 5, pp. 591–611, 2005. 143, 144

[125] B. Finkemeyer, T. Kröger, and F. Wahl, "The adaptive selection matrix - a key component for sensor-based control of robotic manipulators," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, May 2010, pp. 3855–3862. 143, 144

[126] H. Bruyninckx and J. De Schutter, "Specification of force-controlled actions in the "task frame formalism" – a synthesis," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 581–589, Aug. 1996. 144

[127] KUKA System Technology, *KUKA.LaserTech 2.0*, KUKA Robot Group, Dec 2007, Version: V2 de. 162

[128] ——, *KUKA.ServoGun TC 2.0*, KUKA Robot Group, Jun 2007, Version: V2.1. 162

[129] ——, *KUKA.GlueTech 3.2*, KUKA Robot Group, Nov 2007, Version: 3.2 V2 de. 162

[130] SCHUNK GmbH & Co. KG, *Electrical 2-Finger Parallel Gripper WSG 50 - Assembly and Operating Manual*, 02nd ed., Dec 2013, accessed Feb. 2014. [Online]. Available: http://www.schunk.com/schunk_files/attachments/OM_AU_WSG50__EN.pdf 162

[131] ——. (2013, Dec) Servo-electric 3-Finger Gripping Hand SDH. Accessed Feb. 2014. [Online]. Available: http://www.schunk.com/schunk_files/attachments/SDH_DE_EN.pdf 162

[132] K. Etschberger, *Controller Area Network*, 3rd ed. Controller Area Network, Apr 2001. 162

[133] A. Hoffmann, A. Angerer, A. Schierl, M. Vistein, and W. Reif, "Managing Extensibility and Maintainability of Industrial Robotics Software," in *Proc. 16th Intl. Conf. on Adv. Robotics, Montevideo, Uruguay.* IEEE, Nov. 2013. 163

[134] T. Kröger, "Opening the door to new sensor-based robot applications - The Reflexxes Motion Libraries," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, May 2011, pp. 1–4. 182

[135] R. Lienhart and J. Maydt, "An extended set of haar-like features for rapid object detection," in *Proc. IEEE International Conference on Image Processing (ICIP)*, vol. 1, Sep.uses 2002, pp. 900–903. 194

[136] A. Angerer, A. Schierl, C. Böck, A. Hoffmann, M. Vistein, and W. Reif, "Eclipse als Werkzeug zur objektorientierten Roboterprogrammierung," in *Proc. Internationales Forum Mechatronik 2012 (IFM 2012), Mayrhofen, Austria*, November 2012. 217, 218, 219

[137] A. Angerer, R. Smirra, A. Hoffmann, A. Schierl, M. Vistein, and W. Reif, "A graphical language for real-time critical robot commands," in *Proceedings of the Third International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2012)*, 2012. [Online]. Available: http://arxiv.org/abs/1302.5082 217, 220, 221, 222, 223

[138] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, 2nd ed. Boston, MA: Addison-Wesley, 2009. [Online]. Available: http://my.safaribooksonline.com/9780321331885 222

[139] Graphical Modeling Project (GMP). The Eclipse Foundation. Accessed Jan. 2014. [Online]. Available: http://www.eclipse.org/modeling/gmp/ 223

[140] GEF (Graphical Editing Framework). The Eclipse Foundation. Accessed Jan. 2014. [Online]. Available: http://www.eclipse.org/gef/ 223

[141] T. Kuhn and O. Thomann. (2006, Nov) Abstract syntax tree. Accessed Feb. 2014. [Online]. Available: http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html 223

[142] T. Parr and R. Quong, "ANTLR: A predicated-LL(k) parser generator," *Software-Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995. 223

[143] A. Angerer, A. Hoffmann, F. Ortmeier, M. Vistein, and W. Reif, "Object-centric programming: A new modeling paradigm for robotic applications," in *Proc. 2009 IEEE Intl. Conf. on Autom. and Logistics*, Shenyang, China, Aug. 2009. 231

Curriculum Vitae

Name:	Andreas Angerer
Geburtsdatum:	12. Juli 1984
Geburtsort:	Augsburg
Staatsangehörigkeit:	deutsch

Schulische Ausbildung

09/1994-07/2003	Leonhard-Wagner-Gymnasium Schwabmünchen Allgemeine Hochschulreife
10/2003-09/2006	Universität Augsburg (Vordiplom) Studium der Angewandten Informatik
10/2006-09-2007	Universität Augsburg (B. Sc.) Studium der Informatik und Informationswirtschaft
10/2006-09/2008	Universität Augsburg, TU München, LMU München (M. Sc.) Studium des Elite-Studiengangs Software Engineering
10/2008-heute	Universität Augsburg Promotionsvorhaben an der Fakultät für Angewandte Informatik

Beruflicher Werdegang

08/2002-09/2002	Industrie- und Handelskammer Schwaben, Augsburg Aushilfstätigkeiten in der EDV-Abteilung
07/2003-08/2003	Industrie- und Handelskammer Schwaben, Augsburg Praktikum in der EDV-Abteilung
08/2003-09/2003	Industrie- und Handelskammer Schwaben, Augsburg Aushilfstätigkeiten in der EDV-Abteilung
03/2004-04/2004	Industrie- und Handelskammer Schwaben, Augsburg Werkstudent in der EDV-Abteilung
02/2005-06/2006	Fujitsu Siemens Computers, Augsburg Praktikum und Werkstudententätigkeit
08/2007-03/2008	KUKA Roboter GmbH, Augsburg Praktikum und Werkstudententätigkeit
04/2008-09/2008	Universität Augsburg Lehrstuhl für Softwaretechnik Wissenschaftliche Hilfskraft
10/2008-heute	Universität Augsburg Lehrstuhl für Softwaretechnik Wissenschaftlicher Mitarbeiter