Universität Augsburg
Fakultät für Angewandte Informatik

Embedding Real-Time Critical Robotics Applications in an Object-Oriented Language

Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
der Fakultät für Angewandte Informatik
der Universität Augsburg

eingereicht von

Michael Vistein

März 2015

Erstgutachter:	Prof. Dr. Wolfgang Reif
Zweitgutachter:	Prof. Dr. Alexander Knapp
Tag der mündlichen Prüfung:	21. Mai 2015

Abstract

Industrial robots are very flexible machines that can perform almost any task – depending on the tools attached and the program they run. Nowadays industrial robots are mostly programmed using proprietary programming languages provided by the robots' manufacturers. These languages are mostly based on very old programming languages and lack support for modern concepts such as object-oriented design; and programs can rarely be reused. To reduce the cost of robot programming, improving reusability is a key instrument. This can be achieved e.g. by using an object-oriented design process. However, standard off-the-shelf programming languages cannot fulfill the hard real-time requirements of robotics applications and robot control.

This thesis introduces a data-flow graph based interface that allows the specification of real-time critical tasks, the *Real-time Primitives Interface (RPI).* Larger robot applications can be split up into independent parts that inherently require real-time safety (such as single motions, or synchronized tool actions). Each such part can be expressed using the RPI and executed with all timing guarantees. The tasks themselves can be specified and joined using an object-oriented interface. To achieve guaranteed transitions from one or more tasks to another set of tasks, *synchronization rules* are introduced. A reference implementation, the *SoftRobot RCC* has been created to execute robot programs specified using RPI. To convert programs specified using the object-oriented Robotics API framework, an *automatic mapping algorithm* from Java-based applications to data-flow-based real-time tasks is presented.

Danksagung

Die vorliegende Dissertation entstand während meiner Arbeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Softwaretechnik an der Universität Augsburg, teilweise im Rahmen des Forschungsprojektes „SoftRobot“. Zunächst möchte ich mich für die Betreuung und das Erstgutachten dieser Arbeit bei Herrn Prof. Dr. Wolfgang Reif ganz herzlich bedanken. Er ermöglichte mir die Forschung an diesem spannenden und hoch-aktuellen Thema und unterstützte mich durch viele anregende Gespräche. Für seine Tätigkeit als Zweitgutachter möchte ich mich bei Herrn Prof. Dr. Alexander Knapp bedanken. Gleichzeitig gilt mein Dank auch meinen Kollegen der Robotik und Mechatronik Gruppe, Dr. Andreas Angerer, Dr. Alwin Hoffmann, Miroslav Macho, Ludwig Nägele, Andreas Schierl und Constantin Wanninger. Viele Ergebnisse dieser Arbeit sind durch die zahlreichen, konstruktiven Diskussionen in dieser Gruppe entstanden. Ebenso gilt mein Dank allen anderen Kollegen am Lehrstuhl für das sehr angenehme Arbeitsklima und die hilfreichen Diskussionen und Anregungen bei unseren Lehrstuhlklausuren und Oberseminaren.

Während der Laufzeit des Forschungsprojektes „SoftRobot“ wurden viele Ideen im Rahmen von regelmäßigen Projekttreffen mit den Projektpartnern MRK Systeme GmbH und KUKA Laboratories GmbH entwickelt. Hierzu geht mein Dank an Markus Bischof, Alexander Checkler, Dr. Peter Heiligensetzer, Sebastian Pieniack und Franz Stegmair. Ebenso gilt mein Dank Herrn Prof. Dr. Frank Ortmeier, der maßgeblich zur Beantragung des Projektes sowie zu den ersten Schritten im Projekt beigetragen hat.

Für das aufmerksame Korrekturlesen möchte ich mich bei Dr. Andreas Angerer, Prof. Dr. Alexander Knapp sowie bei meinem Vater, Kenneth Vistein, bedanken. Abschließend gilt mein aufrichtiger Dank meiner Familie, die mich während der Erstellung dieser Arbeit fortwährend unterstützt hat.

Michael Vistein

Contents

Chapter 1

Introduction

The application of robots in manufacturing processes has been increasing ever since the first industrial robot was put into service in a production plant of General Motors in 1961 [50]. The first patent for an industrial robot was filed by George Devol on a *programmed article transfer* [32] already in 1954 and was granted in 1961. Together with Joseph Engelberger, Devol founded the company *Unimation, Inc.* which then produced the *Unimate* robots. These robots were hydraulically actuated and employed mostly for spot-welding tasks in the automotive industry [93].

In 1973, the IRB-6 industrial robot was introduced, which was controlled by a microcomputer and driven electrically. It not only allowed simple point-to-point motions, but also more complex path motions were possible which allowed the robot to be used for seam welding [50]. Today, industrial robots are utilized in a broad variety of industries in ever growing numbers. The World Robotics Report 2014 [64] lists a number of 178,132 new industrial robots in the year 2013, with a total of over 1.3 million devices in use worldwide. The total number of robots in use is expected to rise to almost 2.0 million in 2017. The automotive industry is still the largest branch utilizing robots, however other branches are investing more and more into automating their production as well.

One key benefit of industrial robots is their high precision and repeatability. Standard KUKA robots for example have a repeatability of ± 0.1 mm, i.e. a position once programmed will always be reached within this threshold during every further run of the program. For example to meet safety requirements of some products such as cars or airplanes, it is imperative that all welding seams are produced exactly as defined during the product design. The use of robots greatly helps achieving these quality requirements in an extremely fast way. Industrial robots also can manipulate objects safely which are way too heavy for human workers. The largest robots currently can lift over 1 t (KUKA KR 1000 1300 titan PA: 1.3 t, FANUC M2000iA/1200: 1.2 t). A third scenario for the utilization of robots are dangerous environments. Some processes such as laser welding cannot be safely operated by humans. Another area that is likely to see large growth for the application of robots is the decommissioning of nuclear power plants.

According to Hägele et al. [50, Section 42.4], the way of programming industrial robots has changed over time. Early robots were programmed with only joint-space motions. Some systems (in particular painting robots) supported manual guiding for teaching all necessary positions. This was possible by back-driving the actuator, which also could be rather lightweight due to low precision requirements. Furthermore, the mechanical structures were created without singularities, thus allowing the arm easily to follow external motions. Later with the requirement of path motions (i.e. motions in operation space such as linear or circular paths) it was necessary to include an inverse kinematics function in the robot controller to calculate the required joint angles or positions for trajectories specified in Cartesian space. These inverse kinematics calculations need to be performed at very high frequencies, thus the mechanical structure of robots was adjusted to allow for fast inverse kinematics calculations. This introduced singularities in the workspace, and back-driving the manipulator close to those singularities no longer was possible. However, with the inverse kinematics function, it was now possible to move the robot in Cartesian space using a joystick (so-called “jogging”); and also input data retrieved from CAD designs could be included in the robot programs. Altogether, robot programs changed from repeating joint position tasks closer to “standard” computer programs [50].

Each robot manufacturer started developing its own programming language, often borrowing concepts from one of the popular general purpose programming languages at the time the development started. ABB robots for example are programmed using *rapid* [102], FANUC robots using *Karel*, Stäubli robots using *VAL3* [121], and KUKA robots with *KRL* [77]. All these languages have in common that they are tailored to the needs of robotics applications. All languages have special commands for motions, and usually it is possible to blend one motion into another by simply marking a motion as blend-able. This also requires a specific execution semantic, as motion commands can no longer simply be executed one after another. In order to allow for motion blending, it is inevitable to know and plan the next motion prior to finishing the current one. Another common feature not available with standard programming languages is the ability of executing programs backwards or jumping to arbitrary motion commands, e.g. to correct positions which are slightly misplaced during the testing phase. The languages of the different manufacturers are incompatible to each other, thus switching manufacturers is very difficult; and new employees often have to learn a new programming language, even if they have prior experience with industrial robots of another manufacturer.

1.1. Motivation

The domain of software engineering has made great progress during the past decades. Standardized processes such as the Unified Process (UP) [68] or modeling technologies such as the Unified Modeling Language (UML, standardized as ISO/IEC 19505) [58, 94] aim at increasing the quality of software. With good documentation and a well chosen design, the software development can be faster, and also errors are likely to be found earlier during the development process, thus reducing the cost. Object-oriented software design is a very common practice today to model software according to real-world "objects". This helps at understanding the software design and also at creating small, independent and exchangeable software parts. These parts ideally later can be changed, e.g. to correct errors without the need to change the whole system. Furthermore, independent components also can be reused in other projects, thus saving development time and cost. In order to further reduce programming errors, many modern programming languages relieve the developer from manual memory management, which has been a constant cause of program bugs (e.g. memory leaks, buffer overflows). "Managed" languages such as Java or C# therefore use an automatic memory management system which employs a garbage collector which detects and frees unused memory. The increase of software development efficiency for industrial applications is said to be at least 20% compared to C or C++, and over 50% compared to traditional PLC[1] programming languages [97]. Managed languages also employ a Virtual Machine (VM) for program execution. This adds an abstraction layer between the operating system (and hardware), and the application. Virtual machines aim at portability of the programs, i.e. an application compiled once can be executed on systems with different hardware specifications, processor types and operating systems.

Upper levels of industrial automation systems (ERP[2], SCADA[3]/MES[4]) are already often implemented today using managed, object-oriented programming languages, particularly popular is the Java language [97]. Advantages of these languages are the broad availability of libraries for communication, access to database systems or graphical user interfaces. Furthermore, standardized approaches to distributed systems already exist and can be used.

Unfortunately the developers of industrial robot programs cannot profit from these advantages so far. The sources of traditional robot programming languages often predate many of the more recent advances. Unlike modern programming languages and environments which are mostly developed either by companies that are specialized in programming languages, or large communities, the robot programming languages have to be maintained by the robot manufacturers themselves. The need for proprietary programming languages has arisen from both the need for special (blendable) motion commands and the need for a real-time safe execution of robot programs (i.e. each program step always takes exactly the same time).

The managed languages are not suitable for direct control of industrial robots for several reasons. The automatic memory management, in particular the garbage collection,

[1]Programmable Logic Controller
[2]Enterprise Resource Planning
[3]Supervisory Control and Data Acquisition
[4]Manufacturing Execution System

introduces indeterminism for execution times which are not acceptable for direct hardware control. Furthermore, the standard virtual machines usually prevent applications from direct hardware access, which can be necessary for communication with hardware devices such as an actuator or a sensor.

Of course it would be possible for the robot manufacturers to create new, object-oriented programming languages to benefit from at least some of the advantages. However, this would once again just be a new, proprietary and hard-to-maintain programming language, and benefits from using a standardized and widely used (thus well known) programming languages cannot be attained.

The main goal of the *SoftRobot* project was to create a new programming framework for industrial robots, which is based on top of an unmodified modern object-oriented programming language with automatic memory management. Java and C# have been chosen as languages for the reference implementation, although the concepts created during the project are independent of the actual language. All real-time requirements for industrial robots, including special concepts such as motion blending or force-based manipulation tasks are supported.

To solve the issues imposed by using an unmodified standard language for applications, the gap between real-time hardware control and non real-time applications must be bridged. The *Real-time Primitive Interface (RPI)* has been created as a generic and extensible interface for the specification of hard real-time safe tasks. The *Robotics API* provides an object-oriented framework for robotics applications. All programs using the Robotics API can create real-time critical tasks that are automatically translated into *primitive nets*, the description language for tasks offered by RPI. As a reference implementation for RPI, the *SoftRobot RCC* has been created which provides real-time safe hardware access for a broad variety of hardware devices.

1.2. Main contributions

The main contributions of this thesis are the **Real-time Primitives Interface** (RPI) including the specification language of **primitive nets**, the **synchronization mechanism** for multiple primitive nets and the **automatic mapping algorithm** from object-oriented task models to executable, real-time safe primitive nets. A **reference implementation** for the execution environment for RPI and for the mapping algorithm has been created. Altogether, these results now allow to **embed real-time critical robotics applications in an object-oriented language**. The design of the object-oriented programming interface and the modeling of robotics applications has not been in the focus of this work and has been done previously by A. Angerer [1].

Real-time Primitives Interface

Prior to this work, robotics applications in the industrial domain were programmed using proprietary programming languages, which were executed real-time safe as a whole, requiring specialized programming languages and execution environments. During this work it has been shown that it is possible to partition typical applications into parts which require real-time safe execution and other non real-time parts where execution on best-effort base is sufficient.

Based on these findings, the *Real-time Primitives Interface (RPI)* has been developed. RPI is based on a data-flow language and allows the specification of arbitrary tasks as so-called *primitive nets* which must be executed real-time safely. Primitive nets consist of *primitives* which are connected using *links*. Primitives provide the most basic calculation functions that are required. RPI is intended to be automatically generated by the robotics application which can run on any operating system using any programming language without specific real-time requirements. Although RPI is based on a data-flow language, it has been designed to meet specific requirements of the robotics domain by including a life-cycle model and building blocks for application-to-net and inter-net communication.

With RPI it is possible to create multi-robot applications, where robots can be both controlled completely independent, but also with hard real-time synchronization. Applications can switch between both modes at any time. Multiple robots can also be used to perform tasks cooperatively.

Besides controlling actuators, it is also possible to include sensors in robotics applications. Sensors can be used to influence the planned trajectory for an actuator, but also the overall program flow can be controlled by sensor events. Since sensor events can be handled real-time safely, it is possible to guarantee time limits for effects of such events.

Synchronization of multiple independent real-time tasks

Primitive nets provide means to specify atomic real-time tasks. Such tasks cannot be modified or extended once they are started. As previously stated, robotics applications can be partitioned into small, real-time critical parts. However, also the transitions between these parts sometimes should be done with timing guarantees, either to increase the production cycle time or to allow tasks to switch while an actuator is still moving. Switching purely with means of the non real-time robotics application is not sufficient in these cases. The synchronization mechanism of the Real-time Primitives Interface allows multiple independent real-time tasks to be synchronized with guaranteed transition times. This mechanism allows to start or stop multiple tasks synchronously and ensures that these transitions are only performed if all timing guarantees can be fulfilled.

It is now possible to blend multiple motions into each other without having to specify the whole chain of motions at once. Compliant motions apply force to a work-piece and thus the actuator must never be out of active control. With the synchronization mechanism it is possible to switch to a new task while ensuring continuous control of all actuators.

Real-time safe reference implementation for RPI

A reference implementation of RPI including the synchronization mechanism has been created, the SoftRobot RCC. It includes a set of primitives which provide basic calculation functionality, access to sensors and actuators as well as communication with the robotics application. Real-time device drivers are available for a broad variety of industrial robots, ranging from standard industrial robots such as a KUKA KR-16, a Stäubli TX-90L to the novel 7-DOF[5] KUKA lightweight robot. Driver support is also provided for periphery devices such as fieldbus couplers (providing digital and analog input and output ports), laser distance sensors or force/torque sensors. Besides industrial robots, also experimental systems such as the KUKA youBot or even flying quadrotors can be controlled with the SoftRobot RCC. The reference implementation has been designed and developed using object-oriented technologies. Its modularity and easy extensibility allows to integrate new hardware devices with very little effort. Drivers supporting hardware can be loaded and unloaded at runtime, without the need to reset the overall application.

Automatic mapping of object-oriented task descriptions to real-time tasks

To facilitate the programming of industrial robots, the Robotics API has been created, which provides an object-oriented programming model. A mapping algorithm automatically transforms all real-time critical tasks of robotics applications written in Java or C# into primitive nets which can be executed real-time safely on the SoftRobot RCC. The mapping algorithm supports multi-robot applications, and reactions to sensor events can be flexibly defined with both real-time safe reactions as well as non real-time interaction with the application. Several basic motion types are supported such as point-to-point motions in joint-space or linear motions in Cartesian space. Multiple motions can be blended into each other, based on the synchronization mechanism of the RPI.

Using the results presented in this work it is now possible to use managed, object-oriented programming languages also for the programming of industrial robots – without losing real-time safety, if required. The flexible, object-oriented interface to real-time tasks offered by the Robotics API allows for an easy integration of further advanced technologies, such as service oriented architectures for cell level control [54]. A current research project applies the hard real-time reactions to sensor events provided by the SoftRobot RCC to enable an industrial robot to work together with a human worker safely (cf. Section 12.2).

1.3. Structure of this work

Chapter 2 introduces important concepts of the industrial robotics domain. A short introduction to traditional robot programming concepts is given using the KUKA Robot Language (KRL) as an example.

[5]degrees of freedom

The real-time requirements of industrial robot applications are evaluated in Chapter 3. A set of typical applications in the industrial domain is analyzed and the current solutions for real-time safe program execution are explained.

The results of the analysis of real-time requirements led to the creation of the SoftRobot architecture, which separates robotics applications into a novel real-time hardware control part (the SoftRobot RCC), and a non real-time, object-oriented programming framework, the Robotics API. The architecture is explained in Chapter 4, and also the requirements that must be fulfilled by the new architecture in order to provide an improvement over current systems is specified.

The Real-time Primitives Interface (RPI) provides the interface between robotics applications and the real-time execution core. The components and the life-cycle of primitive nets, the main task specification in RPI, are introduced in Chapter 5 and further explained using a set of examples. Each primitive net specifies a single task which must be executed real-time safely, and each net must be specified completely before it can be started. Multiple such tasks can be combined using synchronization rules, which are introduced in Chapter 6. This allows the overall program flow to reside within the non real-time application, primitive nets are only created and started when necessary.

The SoftRobot RCC is a real-time execution environment for primitive nets. Chapter 7 presents the software architecture of the SoftRobot RCC and lists a set of basic primitives which are sufficient for the execution of most robotics applications. The communication interface between the RCC and applications is also described. Finally, the debugging interface for developers is presented. Chapter 8 explains the execution mechanism for multiple primitive nets which are combined using synchronization rules. Special attention is paid to the resource efficient thread allocation algorithm. The real-time device drivers that are required for real hardware control are introduced in Chapter 9.

Robot applications are not intended to be written directly as primitive nets, but rather using the object-oriented Robotics API. Chapter 10 introduces the basic concepts of the Robotics API and describes the automatic mapping mechanism. Using this mechanism, real-time tasks modeled in the application using the object-oriented framework can be automatically transformed into primitive nets which then can be executed real-time safely on a RCC.

Several applications are evaluated in Chapter 11 to demonstrate the usefulness of the concepts introduced in this work. Performance measurements are presented, alongside with an example demonstrating the real-time performance of the architecture for synchronizing two robots of different manufacturers. Finally, the effort necessary to extend the SoftRobot RCC to new hardware devices is analyzed based on the integration of a new manipulator.

This thesis is concluded in Chapter 12 with a short summary and an outlook to possible future extensions.

Chapter 2

Basics

In this chapter, some basic concepts of (industrial) robots and robotics applications are introduced. Readers who are already familiar with the industrial robotics domain may skip this chapter. The first section introduces the basic components of robotics hardware. One of the main features of every industrial robot is its ability to move to any position within its working space, thus the following section introduces the most common types of motions and combinations thereof. Finally a short overview of (software) programs and applications for industrial robots is given.

2.1. Robotics hardware

According to the ISO 8373:2012 [67] norm, an industrial robot is defined as an

> "automatically controlled, reprogrammable, multipurpose manipulator, programmable in three or more axes, which can be either fixed in place or mobile for use in industrial automation applications" [67, Section 2.9].

and consists of

- "the manipulator, including actuators"
- "the controller, including teach pendant and any communication interface (hardware and software)." [67, Section 2.9]

The manipulator is a "machine in which the mechanism usually consists of a series of segments, jointed or sliding relative to one another, for the purpose of grasping and/or moving objects (pieces or tools) usually in several degrees of freedom" [67, Section 2.1].

There are several different types of industrial robots available, differing in the type and number of joints and therefore in their ability to manipulate the environment. Prismatic

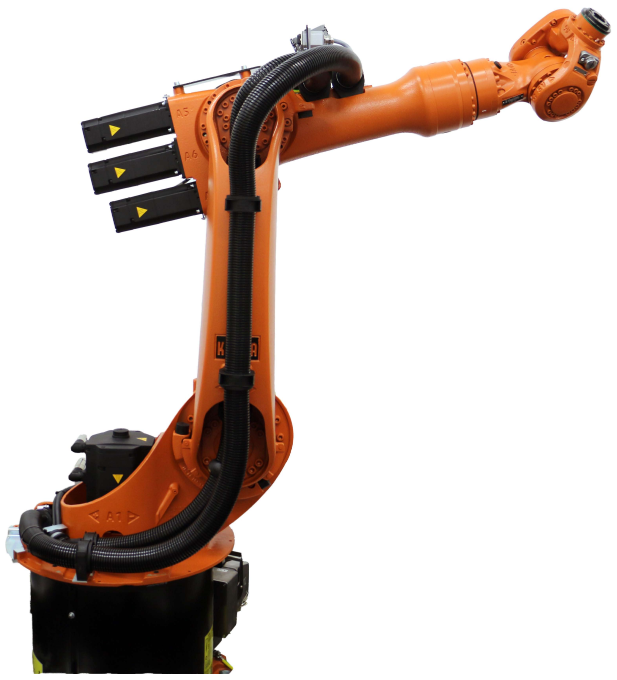


Figure 2.1.: KUKA KR-16, a typical industrial robot with six revolute joints.

(sliding) joints are mainly used in portal systems to cover a large cuboid working space. Typical industrial manipulators (articulated arms) are built with a series of 4 to 6 revolute joints.

An object can be positioned and oriented with six degrees-of-freedom (DOF), three translational and three rotational DOF. A manipulator which should be able to manipulate objects in 3D-space needs to possess at least six DOF (joints) which must properly be distributed along the mechanical structure [108, p. 4]. Manipulators with less DOF are limited in their ability to position or orientate the object, but however can still be sufficient for many tasks. A common task e.g. is palletizing goods which only need the three translational DOF and one rotational DOF (rotating around the vertical axis). Palletizing robots are commonly made with only four rotational joints or three prismatic joints and one rotational joint. Manipulators with more than six DOF are redundant, i.e. there is an unlimited number of poses to reach a given position and orientation. In the context of this work, the terms “robot” or “manipulator” denote an articulated arm with six revolute joints, unless stated differently.

Figure 2.1 shows a typical six DOF articulated arm as it is used for most industrial robotics applications. The manipulator is mounted with its base to the ground or ceiling and provides a mount flange to attach the end-effector (e.g. a gripper, welding torch, etc.) which actually manipulates the working-pieces. The mounting flange is defined in the ISO 9409-1 [66] norm. The flange is attached to the wrist of the robot, which is made up by the last three joints. In many systems the axes of the last three joints intersect in

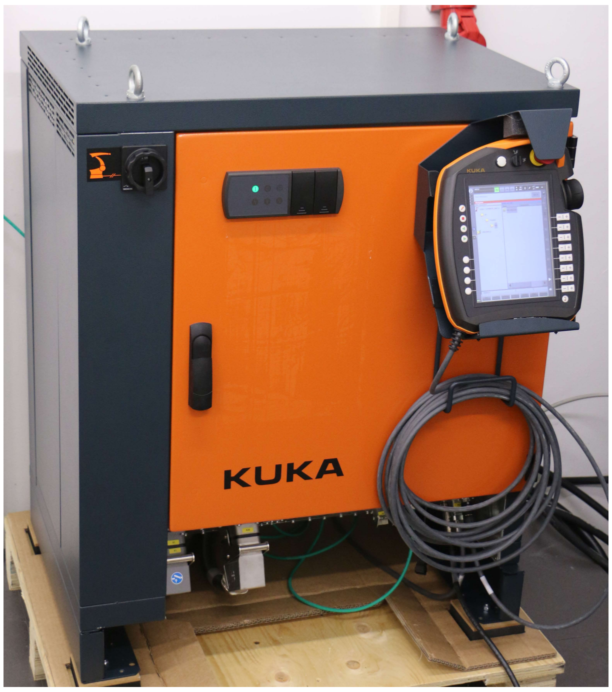


Figure 2.2.: KUKA KRC-4 robot controller with KUKA KCP teach pendant

a single point and form a so-called spherical wrist, which allows highest dexterity [108, p. 10]. The first three joints (“base joints”) are used to position the wrist, while the last three joints determine the orientation of the end-effector.

According to ISO 8373, an industrial robot consists not only of the manipulator itself but also of a robot controller including a teach pendant. Figure 2.2 shows such a typical robot controller, here an example manufactured by KUKA. The robot controller contains a programmable motion controller which is responsible for planning and executing all motions of the manipulator. Often, an industrial computer with special software is used for this purpose. Furthermore, power electronics such as the power supply and variable frequency drive (VFD) modules to interface with the manipulator’s motors are included in the controller case. The teach pendant is used to program the industrial robot “online”, i.e. to move the robot manually and save the positions in a program for later automated execution.

2.2. Robot kinematics

The pose of an end-effector can be completely defined using the position and orientation of the end-effector with respect to a reference coordinate system. For many applications, a “world” coordinate system is defined as a global reference, and the position and orientation of each robot in a working cell then are subsequently defined with respect to the world coordinate system. Each robot has its own robot-base coordinate system, thus the position and orientation of the robot is defined by describing the translation and rotation between the world coordinate system and the robot base coordinate system.

In general, to describe the position of a coordinate system O' with respect to a reference coordinate system O, a vector is used. In Cartesian space (i.e. 3D-space), a vector $v \in \mathbb{R}^3$

$$\vec{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

is used to describe the position of the coordinate system O' as

$$\vec{O'} = \vec{O} + x \cdot \vec{e_x} + y \cdot \vec{e_y} + z \cdot \vec{e_z}$$

with $\vec{e_x}$, $\vec{e_y}$ and $\vec{e_z}$ being the unit vectors of coordinate system O.

The orientation of the coordinate system O' can be described in relation to O by defining the unit vectors $\vec{e'_x}$, $\vec{e'_y}$ and $\vec{e'_z}$ of O' in coordinate system O. For coordinate system transformations, the unit vectors can be combined into a single rotation matrix. Using homogeneous coordinates [108, p. 56], it is possible to describe the whole transformation from O to O' using a single matrix:

$$\begin{pmatrix} \vec{e'_x} & \vec{e'_y} & \vec{e'_z} & \vec{v} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In the robotics domain, rather than using transformation matrices, it is more common to describe the orientation of a coordinate system using Euler angles. Those describe three consecutive rotations around axes of the base coordinate system. Depending on the order of rotations applied, different result are reached, therefore it is necessary to agree upon a common convention. For this work, the following convention is used:

1. Rotation around X-axis of reference coordinate system with angle C
2. Rotation around Y-axis of reference coordinate system with angle B
3. Rotation around Z-axis of reference coordinate system with angle A

The same convention is used by KUKA and often described in literature as Roll-Pitch-Yaw (RPY) angles [108, p. 51]. Intuitively the same rotation can also be reached by first rotating the coordinate system around its Z-axis with angle A, then around its (new) Y-axis with angle B and finally around its (new) X-axis with angle C. Using this convention, it is possible to precisely describe the position of the coordinate system O' with respect to coordinate system O with only three values for position and further three values for orientation. One orientation can be expressed with at least two sets of Euler angles, one for $B \in \left]-\frac{\pi}{2}, \frac{\pi}{2}\right[$ and the second for $B \in \left]\frac{\pi}{2}, \frac{3 \cdot \pi}{2}\right[$. For $B = \frac{\pi}{2}$ and $B = \frac{3 \cdot \pi}{2}$ the rotations A and C have the same rotation axis, thus only the sum of A and C is relevant for the overall rotation. This is called a representation singularity, and an infinite amount of value-pairs for A and C can be given.

The *direct kinematics* function maps joint coordinates into Cartesian coordinates. For an *open-chain* manipulator (i.e. a manipulator that has $n + 1$ links connected with n joints, where each joint provides a single DOF [108, p. 60]), the direct kinematics function is rather easy to calculate. According to the *Denavit-Hartenberg* (DH) convention [108, p. 61] a coordinate system is placed in each joint and finally on the flange. The transformation from a coordinate system CS_n to the following coordinate system CS_{n+1} is defined by four distinct parameters, the so-called DH-parameters d, a, α and ϑ. For revolute joints, the parameter ϑ, and for prismatic joints the parameter d contains the position of the joint. Using the four DH-parameters, it is possible to create a transformation matrix to describe the transformation from one coordinate system to the following one. By chaining all transformations from CS_0 to CS_n the direct kinematics functions can be expressed. For each distinct set of joint values, the direct kinematics function yields a single solution in Cartesian space (although when Euler angles are used, different representations of this single solution are possible).

The *inverse kinematics* function maps a position and orientation in Cartesian coordinates to a set of joint angles. Unlike the direct kinematics function, the inverse kinematic is not easy to determine for an open-chain manipulator. Different approaches exist for solving the inverse kinematics function. A numerical solver (e.g. employed by KDL [112]) can be used to approximate the inverse kinematics function. For the actuators described in this work, a geometrical approach has been used. The inverse kinematics function is created by describing geometrical dependencies of joints and the position. The main disadvantage of the geometrical solution (besides the effort to find the dependencies) is that a solution found can only be applied to other robots with a very similar structure (i.e. with the same orientation of a joint in respect to its neighboring joints). The main advantage however is, that once the overall function has been determined, solutions for a Cartesian position can be calculated very fast. For typical industrial robots with a spherical wrist, the inverse kinematics function yields up to 8 solutions. If at least two joint-axes are identical, infinite solutions for the inverse kinematics exist since only the sum of both joint angles is relevant. This situation is called a “singularity” or “singular pose”.

2.3. Robot motions

Moving the manipulator (and therefore, the end-effector attached to the flange) to different positions is a key functionality of most programs. For a robot to move from one position to another position, a trajectory must be planned. As with every physical object having inertia, a robot cannot simply move with some velocity but rather has to be accelerated and later decelerated before stopping.

In general, two types of robot trajectories are possible:

- Joint space trajectories
- Operation space trajectories

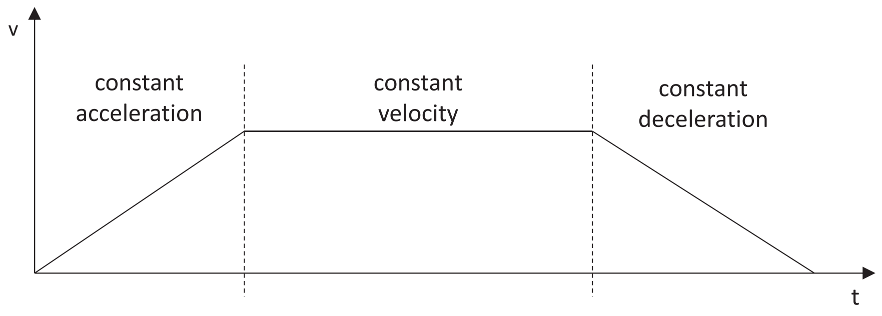


Figure 2.3.: Trapeze velocity profile of single joint point-to-point motion

For joint space trajectories, each joint is moved from the start position to the destination position with a defined velocity or acceleration profile. The movement of each joint may be synchronized to start and stop at the same time as the other joints. Motions in joint space are the fastest motions possible because each joint only moves the minimum required distance. In some robot programming languages (e.g. the KUKA robot language KRL) these motions are called "point-to-point" or "PTP". In general, this type of motion is used when a fast repositioning of the robot is required and the path of the end-effector does not matter. In most robot programming languages, a programmed motion starts at the current position, and only the destination must be explicitly programmed. For joint space motions, it is possible to use joint coordinates (i.e. the destination angle for each joint) as well as Cartesian coordinates. In the latter case, the inverse kinematics function is used to determine the destination joint angles. Because the inverse kinematics functions is not unique, it is necessary to specify additional meta-data to select the proper solution. KUKA e.g. uses two bit-arrays "status" and "turn" for this purpose, which describe i.a. whether a joint is turned in positive or negative direction. If Cartesian coordinates are used, the destination position must not be in a singular pose, otherwise the inverse kinematics is undefined.

For operation space trajectories, the path of the end-effector is planned with a certain velocity and/or acceleration profile, and the motions of the single joints are inferred by applying the inverse kinematics functions for many points along the trajectory. Operation space trajectories are used if the path of the end-effector is relevant to the application, e.g. if a welding torch has to follow the seam. Common trajectories in operation space are linear and circular paths. More recently also paths defined by splines are possible. A spline is defined by a multitude of support points and polynomial functions that interconnect all points. For many applications it is also desired to keep a constant velocity of the end-effector during the motion, e.g. to provide high quality welding seams.

Figure 2.3 shows a velocity-time diagram for a point-to-point motion of a single joint with a trapezoidal velocity profile. The motion can be split into three phases: 1. constant acceleration, 2. constant velocity and 3. constant deceleration. For very short motions, it is possible that phase 2 is missing, if the joint cannot reach full velocity before the deceleration phase has to start. A trapezoidal velocity profile is a simplified variant of velocity profiles used by commercial robot controllers. The abrupt transition from constant acceleration to constant velocity causes unlimited jerk (second derivation of the velocity), which induces considerable stress into the mechanics of the manipulator. To limit the jerk, it is possible to apply a trapezoidal profile to the acceleration and deceleration and thus split the motion into 7 phases (increasing acceleration with constant jerk, constant acceleration, decreasing acceleration, constant velocity, increasing deceleration, constant deceleration and finally decreasing deceleration). This motion profile is commonly referred to as “Double S” or “bell” profile [9, Section 3.4].

Simultaneous point-to-point motions of multiple joints can be synchronized in three different ways:

Asynchronous: All joint motions are started synchronously, however no further synchronization among the different joints is performed. Each joint will use maximum velocity and acceleration to reach its individual destination.

Synchronous: The start and stop times of all joints are synchronized, i.e. some joints will be slowed down to reach their destination within the same time as the slowest joint. This does not slow down the overall motion process, but as not all joints need to apply full velocity, wear and tear of the mechanics is reduced.

Fully-Synchronous: Not only the start and stop of the overall motion is synchronized, but also the times of transition from acceleration to constant velocity and from constant velocity to deceleration. This further reduces wear and tear, because the high jerk when switching from acceleration to constant velocity only occurs twice during a motion, and not twice for each joint. However, because the length of the acceleration phase has to be adjusted, it is possible that this motion profile leads to slightly slower movements.

For hardware motion control, the trajectory is split up into small interpolation steps of usually between 1 ms and 20 ms. For each interpolation step, the position, velocity and acceleration is calculated and fed into a closed-loop controller, which then performs the micro-interpolation (i.e. the interpolation between the calculated steps).

Point-to-point motions in joint space can be directly interpolated and set-points generated for closed-loop control. For motions in operation space, set-points must first be calculated in operation space and then transformed into joint space by applying the inverse kinematics function for each interpolation step. Trajectories in operation space cannot traverse singularities of the manipulator, because the inverse kinematics function does not yield valid results in these positions. Furthermore, small motions in close proximity of a singular pose yield large joint movements, thus even close passing of a singular pose can lead to joint velocities which exceed the maximum allowed velocity.

Besides motions based on pre-calculated trajectories, also motions influenced by sensor readings are possible. Manipulation and assembly tasks are becoming more and more important in robotics [18]. Those tasks often contain uncertainty about the work piece (e.g. the peg-in-hole problem), thus compliant motions are used. Those motions eliminate some uncertainty by maintaining contact between the workpiece, tool and other parts of the environment with the robot adjusting its trajectory to follow the (unknown) form of the work-piece based on (force) sensor values.

2.4. External motion control

Some hardware devices allow “external” motion control, i.e. the desired trajectory is not planned and executed by the hard- and software supplied by the manufacturer, but rather by a system provided by the customer. For electrical drives (and thus also for industrial robots), external control can be performed on several different levels:

- **Power/Torque:** The external system can directly control the power applied to the drive. The current consumed by an electric drive is roughly proportional to the created torque.
- **Velocity:** The external system supplies the desired velocity of the drive.
- **Position:** The desired (angular) position is provided by the external motion controller.

While the first option allows for very low level of access, the latter options require further logic integrated into the device, since the drives internally always need to be controlled by applying power. In the context of this work, position control in particular is of interest. For *cyclic position control*, new position set-points must be provided by the external motion controller in strict time intervals, and the hardware device attempts to reach the given position within one time interval. Therefore the set-points need to be reachable within one interval, and the velocity of the trajectory must be steady. Otherwise, the hardware device will not be able to follow the trajectory due to the inertia of the system. For good results, the acceleration of the trajectory should also be steady.

For larger systems such as industrial robots, a simple closed loop controller for converting position set-points into velocity and further into torque set-points is not sufficient, but attention has to be paid to the dynamics of the system. For instance much more torque is required to move a robot joint upwards against gravity instead of downwards, thus the power applied to the drive depends not only on the requested velocity, but also on the current position, the direction of the motion and also the current payload of the robot. Most industrial robots that allow for external motion control internally compensate for all these effects. Thus it is sufficient for the external motion controller to supply a trajectory without taking dynamic effects into account.

2.5. Motion blending

A commonly used feature in industrial robotics is the so-called “motion blending”. Often some auxiliary points on a robot trajectory are only programmed because the direct connection of two points is impossible due to obstacles in the working space. Stopping

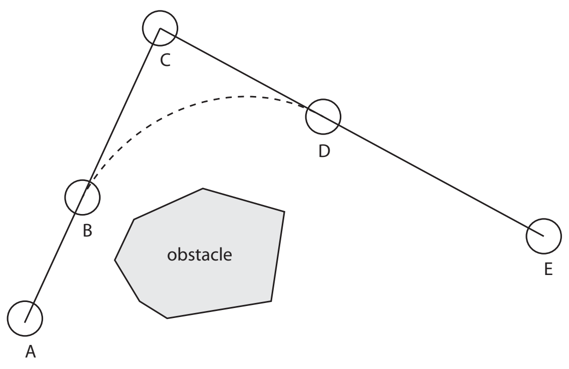


Figure 2.4.: Solid: Trajectory of two linear motions from A to C and C to E; Dashed: Trajectory with motion blending enabled (adapted from [125])

the robot at each of these auxiliary points is neither necessary nor desirable, because decelerating and accelerating wastes lots of time and energy. Also wear and tear of the robot system is reduced by using motion blending instead of a sequence of self-contained motions. With motion blending, the programmed trajectory will be left before the auxiliary point is reached and blended into the trajectory of the following motion, i.e. the programmed auxiliary point is never actually reached. Figure 2.4 shows an exemplary trajectory. Two subsequent linear motions in Cartesian space have been programmed, the first from point A to point C, and the second from point C to point E. Without motion blending, the robot will move along the solid trajectory, stopping shortly at point C (changing direction of movement in a single point without stopping is physically impossible). With motion blending enabled, the robot will leave the trajectory from A to C at some point B and follow the dashed trajectory which enters the trajectory from C to E at some point D. The position of the points B and D may be configurable, the closer these points are to point C the more the robot has to decelerate to be able to blend both motions. Most robot programming languages allow the developer to specify a maximum distance that points B and D may be away from point C.

The example depicted in Fig. 2.4 shows motion blending between two linear motions in Cartesian space. The KUKA robot controller e.g. also supports blending between all possible types of motions, in particular it is possible to blend a joint space point-to-point motion into a Cartesian space linear motion and vice versa. Motion blending can be applied to an unlimited number of subsequent motions, creating a single long, continuous motion of the manipulator, consisting of several independently programmed segments.

2.6. Robot programs and applications

This section introduces common robot programming techniques currently used with examples from the KUKA robot language (KRL). Listing 2.1 shows an exemplary

```
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 BRAKE
20    INTERRUPT ON 3
21
22    TRIGGER WHEN DISTANCE=10 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
```

Listing 2.1: Example KRL program with function `example()`, taken from [90]

KRL program which demonstrates some of the most commonly used features for robot programming. Complete references of the KRL language can be found in [76, 77].

The KRL programming language is an imperative language that is interpreted at run-time. It supports the common features of imperative programming language such as branches (**`IF ... THEN ... ELSE`**) and loops (**`FOR ... ENDFOR`**), but also adds some robotics-specific functions, predominantly motion commands such as **`PTP`** for point-to-point motions or **`LIN`** for linear motions in Cartesian space. Variables are statically typed and must be declared first in any function or procedure. Mixing variable declarations and other commands is not allowed. Types include standard types such as integer or floating point numbers, but also structured types for representation of positions in Cartesian space (`POS`) or joint space (`AXIS`). User-defined structures are also possible. Variables can be defined in different scopes. Variables defined within a procedure (e.g. Listing 2.1 lines 2 – 4) are only accessible within the same procedure. Each KRL program is accompanied by a so-called "DAT file" which contains further variables which are accessible from all functions within a single KRL program. Those variables are used amongst others for storing taught frames. It is possible to add other variables which are required program-wide, and values written to those variables are remanent (i.e. the last value is stored between program runs). Besides variables local to single functions or programs, also system-wide, so-called "global" variables are available. Those variables are usually prefixed with a `$`-sign and are used to access system functionality such as maximum velocities. In line 7 of Listing 2.1 the velocity of each joint in all subsequent joint-space motions is reduced to max. 60%.

Motion commands The KRL language has special commands for robot motions: **`PTP`** for point-to-point motions in joint space, **`LIN`** for linear motions and **`CIRC`** for circular motions in Cartesian space. Releases of the KUKA software system within the last few years also support spline motions (**`SPL`**), even combined with linear **`SLIN`** or circular **`SCIRC`** segments.

Each motion command requires a parameter containing the destination position for the motion; the current position is always used as the starting position. The destination position can either be specified using joint values (type `AXIS`, e.g. line 10) or Cartesian coordinates (type `POS`, e.g. line 14). If Cartesian coordinates are used, the global variables `$BASE` and `$TOOL` are used for specification of the reference coordinate system and the tool coordinate system. Both types of coordinate specification can be used for joint space and Cartesian space motions. If Cartesian coordinates are used for joint space motions, additional attributes "status" and "turn" must be included to specify which inverse kinematics solution is to be used. Motions in operation space ignore status and turn attributes because from any given starting pose only a single solution of the inverse kinematics solution is reachable within each interpolation step.

Motion blending can be activated by using the keyword **`C_PTP`** for point-to-point motions or one of the keywords **`C_DIS`**, **`C_VEL`** or **`C_ORI`** for operation space motions (cf. line 24). Blending allows the robot controller to start with the following motion command before the motion with the blending keyword has finished. Motion blending is not possible if any command between the two motion commands prevents the controller from planning the blended motion in time. Examples for such commands are I/O operations or very time consuming calculations.

Tool commands Industrial robot arms often have tools attached to their flange to manipulate working pieces. Common tools include grippers, welding torches, drills or screwdrivers. Most of these tools have in common that they need some control, e.g. they need to be turned on or off depending on the position of the manipulator, thus it is common to trigger tool actions by the robot program at appropriate positions. The communication of the robot controller with the tool is often realized using a fieldbus, such as ProfiBus [119], ProfiNet [38], DeviceNet [11] or EtherCAT [69]. Every device connected to the bus is able to read and write binary data from and to other devices. The KUKA controller maps this data to global variables which can be accessed in a KRL program. The global array `$IN` represents so-called "digital input" values, i.e. 1-bit information received from the fieldbus, and `$OUT` represents “digital output” values, i.e. 1-bit data written to the fieldbus. Digital I/O is used e.g. for commanding a gripper to open or close, or to turn a welding torch on and off. Besides single bit data, it is also possible to transmit numerical values using multiple bits. In KRL, these values are called “analog” and can be accessed using the `$ANIN` and `$ANOUT` variables. Directly accessing I/O using the global variables between two motion commands prevents blending of those motions, because the robot controller has to wait until the first motion has finished before I/O is performed. More details about commands that prevent motion blending can be found in Section 3.5.1.

Parallel programs Neither the KRL language nor the KRC controller itself support parallel robot programs. Only a single robot program may be active at a single point in time, although the *submit interpreter* always executes a second KRL program in the background. This program may contain monitoring tasks, but cannot actively control the manipulator. Furthermore, no guarantees about execution times or cycle times can be given, as the submit interpreter runs with lower priority than the main KRL program.

Using **`TRIGGER`** or **`INTERRUPT`** it is possible to synchronize (small) sub-functions with the main program execution. Whenever such a sub-function is triggered, the main program is interrupted and the sub-function executed. If the sub-function is executed fast enough (i.e. before the motions already planned from the main program are completed), the robot motion will not be affected. Triggers can be used e.g. to synchronize tool commands with the programmed motion. It is possible to start such an action with a given delay after a motion has started or before it is completed. For motions in operation space, it is also possible to define triggers based on the distance traveled since the start of the motion. In line 22 of Listing 2.1, the digital output number 2 is activated 20 ms after 10 mm of the motion to point `cpos` (line 23) have been completed.

While triggers are commonly used to synchronize tool commands to motion progress, interrupts are more generic and can be defined on any logical condition, e.g. on the value of a digital input. The condition is evaluated cyclically and once it becomes true, the main program is interrupted and the specified action executed. Such an action can be a sub-function, which may also contain motion commands. Before such a command can be used however, the robot must be stopped using the **`BRAKE`** or **`BRAKE_F`** commands. The first command stops using a standard deceleration ramp (i.e. the robot continues to move on the programmed trajectory), while the latter command brings the robot to standstill as fast as possible, but allows it to leave the programmed trajectory. In line 19 of Listing 2.1 an interrupt with priority 3 is defined on the condition of digital input number 2 becoming false. The action is to brake the robot and terminate. In line 20, the interrupt is activated, and finally deactivated again on line 27. The priority is taken into account if multiple interrupts become active at the same time. In this case, only the interrupt with the highest priority will be executed.

Chapter 3

Real-time requirements in robotics applications

Industrial robots are expected to deliver very precise and highly repeatable results. One key building block to achieve those requirements is the use of hard real-time systems. The following sections go into detail about what real-time systems are, and why they are required for robotics applications. Furthermore, the implications of real-time systems on software development are discussed, and solutions currently in use are presented.

3.1. Real-time: Definition

In order to be able to specify the real-time requirements of robotics applications, first a definition of a real-time system must be given. Biyabani et al. [13, p. 152] define hard real-time systems as follows:

> "Hard real-time systems are those systems in which the correctness of the system depends on both the logical result of the computation as well as the time at which such a result is produced. In many hard real-time systems it is crucial for the tasks in the system to meet their specified deadlines, otherwise the tasks are worthless, or worse, cause catastrophic results. This strictness in meeting deadlines makes scheduling an important issue in the correctness and reliability of the system."

Not all real-time systems must fail necessarily catastrophically if a deadline is missed, thus it is possible to distinguish between two levels of real-time requirements. Buttazzo [26, p. 8] defines those two system classes as follows:

> "Depending on the consequences that may occur because of a missed deadline, real-time tasks are usually distinguished in two classes, hard and soft:
>
> - A real-time task is said to be *hard* if missing its deadline may cause catastrophic consequences on the environment under control.
> - A real-time task is said to be *soft* if meeting its deadline is desirable for performance reasons, but missing its deadline does not cause serious damage to the environment and does not jeopardize correct system behavior."

Robotics applications can require both categories of real-time systems. The following section investigates different levels of robotics applications and the real-time requirements of those levels.

For each hard real-time task, the *worst case execution time (WCET)* can be specified, which must be below the required deadline of the system in order to provide useful results. The execution of a task may actually take less time than the WCET (e.g. depending on the user input), but it must be guaranteed not to exceed the WCET under any circumstances.

In the remainder of this work, an algorithm, program, etc. is called *real-time safe*, if it can be executed with a deterministic run-time, i.e. a WCET can be specified.

It should also be noted that a real-time system is not to be confused with a "fast" system. As long as a system is able to achieve its task reliably within the given deadline, it is a real-time system. If another system is performing the same task at twice the speed most of the time, it may still not be a real-time system if it is possible that the deadline is missed eventually.

3.2. Necessity of real-time in robotics applications

Programs for industrial robots often consist of a series of robot motion and tool commands. Those commands form the overall process of the robot application, and abstract from low level control of the robot and its tools. In order to achieve good performance, multiple levels of hardware control can be found. The first level is the motion control level. At this level, every single motion of the robot is planned and executed. On top of this layer is the application layer, which controls the overall course of events of the program, i.e. the coordination of single motions and their combination with tool actions as well as the interaction with external systems (e.g. using a fieldbus). Motion control in the first layer is usually done by software created by the robot manufacturer, while applications are developed by the end-users of the robot system.

3.2.1. Motion control level

Industrial robots are often controlled with nested closed loop controllers [108, Section 8.3]. The hardware interface usually offers the ability to regulate the power consumed by the servomotor driving a single joint, which is roughly proportional to the torque applied by the motor. On top of this power or torque control, velocity and position controllers can be applied as cascading controllers. Velocity controllers try to achieve a given velocity of a joint by controlling the torque applied. Position controllers try to reach a given position by controlling the velocity of a joint. To support velocity and position control, most robot joints are equipped with encoders which can measure the current position of the joint. Using this position information (and the derived velocity), a closed loop controller can be constructed.

If such a low-level robot controller is to be implemented as part of the robot control software (in contrast to being embedded into the robot hardware), a hard real-time system is required for good quality results. The measured values must be retrieved and new set-points calculated regularly. Large jitter in calculating the new set-points would lead to imprecise motions. For a velocity controller, the target velocity will not be reached exactly if the last commanded torque is applied too long or too short. Identically for a position controller, the target position will not be reached exactly if the robot is moving too long or too short with a previously set velocity.

In order for a robot arm to move to a destination point in a certain way (e.g. on the fastest way, on a straight line, etc.), a trajectory must be planned. This trajectory is then split up into small motion interpolation steps, which are usually between 1 ms and 12 ms long. A new set point in joint space is calculated for each interpolation step. These position set-points are then fed into the aforementioned position controller. The new set points must be provided at exact identical intervals, otherwise the velocity of the motion is not steady, leading to a very jerky movement of the joint. This means very high wear and tear of the mechanics and can seriously damage the robot hardware.

Conclusion Controllers for robot hardware (based on torque, velocity or position control) implemented in a software robot controller absolutely require hard real-time. Missing deadlines always leads to either not reaching a given target exactly (which in turn impacts the robot's overall accuracy and reliability) or to jerky, wearing robot motions, potentially even causing damage. The execution of a trajectory must also be performed within a real-time system to guarantee a precise execution of the trajectory. Thus, for every single motion of a robot, hard real-time is required.

3.2.2. Application level

If an application for a robot purely consists of a sequence of motions, real-time is only required for each motion command on its own (cf. motion control level). Using such a simple sequence of motions, the robot comes to standstill after each motion, and some non-deterministic delay at this point will not lead to a failure of the system (however, performance will be degraded since less workpieces can be processed per time). Real-world applications usually are far more complicated, and also require tool actions to be executed at given points in time or at a given position of the trajectory. For example, a welding torch must always be turned on and off precisely at the beginning and the end of the welding seam. Failure in the timing of tool actions can lead to massive damage, e.g. an incorrect welding seam negatively influences the quality of the products.

Many applications use the concept of motion blending (cf. Section 2.5) to speed up transfer motions around obstacles in the workspace. Blending one motion into another inherently requires a hard real-time controller just like any single motion does (two motions blended into each other can be considered a single, continuous motion). Most robot systems even allow motion blending to be used for an arbitrary number of points, achieving a continuous motion among all those points. Motion blending requires knowledge about future motions, i.e. it is not sufficient to process one motion command after each other, but the motion that is blended into must also be planned before the point where the trajectories diverge is reached (point B in Fig. 2.4 on Page 17). Although hard real-time is required to execute a blended motion, soft real-time is sufficient for the overall program execution. If the successive motion is not planned in time for motion blending, the preceding motion can be simply completed as if no motion blending was requested, albeit with reduced performance. Robotics applications should not depend on successful motion blending, in particular for debugging purposes it is common practice to disable motion blending temporarily in order to fine-tune previously taught (auxiliary) positions.

Conclusion Some parts of robot applications also require hard real-time for execution. Tool actions generally need to be synchronized with robot motions, and the synchronization must be guaranteed to achieve high quality products and to avoid damage to both the workpiece and the tools. For the overall program flow, soft real-time is desirable to achieve high performance and low cycle times. However, missing deadlines here generally only raises cycle times but does not directly cause damage.

Overall, only tool actions inevitably require hard real-time. In most other cases, soft real-time is sufficient. In Section 3.4 some currently used applications and technologies are analyzed to further determine the grade of hard real-time required in typical robotics applications.

3.3. Implications of real-time on software

The requirement of real-time has several impacts on the software development for such systems. Both the underlying operating system as well as the application software must support real-time. Special care must be taken during the development of applications to avoid non-deterministic behavior.

3.3.1. Operating system support

General purpose operating systems such as Microsoft Windows or Linux do not provide hard real-time capabilities. The process scheduling facilities of such operating systems aim at providing good performance for the whole system, thus providing computing time to all processes. Any process may be interrupted at any time (schedulers usually are preemptive), thus no guarantees for deadlines are possible.

Special real-time operating systems such as VxWorks, QNX and Linux with Xenomai or RTAI extensions provide explicit support for hard real-time applications. Processes with real-time requirements are scheduled strictly conforming to their priority, even allowing high priority processes to stall the system in general. It is also possible to delay the handling of hardware interrupts in order not to interrupt important real-time tasks.

3.3.2. Development of real-time applications

Programming of real-time systems can be done using different programming languages. Very common is the use of the C and C++ programming languages due to their close proximity to hardware. Often the operating system provides libraries that provide an application programming interface (API) which must be used to achieve real-time capabilities. The Linux extensions Xenomai and RTAI allow the use of all standard Linux functions (so-called "syscalls"), however a program is no longer real-time safe when these functions are used.

An application programmer must take care of a couple of special characteristics of the program when developing for real-time systems:

- Separation of real-time and non real-time code. Often, not all parts and/or threads of an application require hard real-time. A clean separation of both aspects should be done, because switching from hard real-time to non real-time mode breaks any timing guarantees.
- Memory allocation cannot be done real-time safe. If additional memory is required, the operating system must assign a new hardware memory location to the application which breaks real-time. Thus all required memory must be allocated before a section of the program is entered where hard real-time is required. Memory must also not be freed while being in such a section, thus after all real-time tasks have been completed, a clean-up phase is required.

  If the Linux operating system is used, it should be noted that allocating memory using `malloc` or `new` is not sufficient, because memory allocation is only performed on first real memory access. Thus each memory page freshly allocated should actually be written to first, before real-time operations are started.
- Access to hard disks is not possible due to unpredictable delays e.g. in positioning the heads and rotating the platters. Thus logging etc. must be done purely in pre-allocated memory.

- Xenomai and RTAI allow the use of arbitrary Linux syscalls. However, each call effects a switch of the thread from real-time to non real-time mode and back. While the thread is in non real-time mode, no timing guarantees can be given.

Overall, programming applications for real-time systems requires more effort, and errors can easily occur. Those errors need not necessarily be easy to find, because breaking real-time need not immediately have any effect on the application. If the application is tested with little load on the system, deadlines might by achieved even without real-time support. Only when high load occurs are deadlines missed.

Modern object-oriented programming languages such as Java or C# are not suitable for programming real-time systems out of the box. Those languages offer automatic memory management (using a garbage collector) which does not allow manual management of resources, as required for real-time systems. Especially the garbage collector, which can run at any time, stopping all threads of the application, will break real-time. However, for the Java programming language there is a real-time implementation available (AICAS JamaicaVM[1]) which also supports real-time garbage collection [109].

3.4. Real-time in industrial robotics

In order to determine the degree of hard real-time that is required by today's robotics applications, the KUKA Robot Language (KRL) is examined. Furthermore a set of technology extensions available for industrial robots is examined in relation to the real-time requirements of programs written using those extensions. This evaluation is done based on products of the companies KUKA Roboter GmbH and MRK Systeme GmbH, both project partners in the SoftRobot project (cf. Chapter 4). The set of analyzed software packages has been determined in close cooperation with both companies, using their expertise in robotics applications.

3.4.1. KUKA Robot Language (KRL)

KUKA robots come with a proprietary programming language, the KUKA Robot Language (KRL) [76, 77]. This programming language allows the developer to specify robot programs on a rather high level, i.e. issuing motion commands, tool commands, synchronizing motions and tool actions etc.

Motion commands

KRL programs support four different types of motions: Point-to-point, linear, circular and spline motions. For more details on KRL syntax please refer to Section 2.6. Each motion itself requires real-time for motion control (cf. Section 3.2.1). However, if two

[1] `http://www.aicas.com/`

motion commands are simply put one after the other, the robot moves to the first given point, stops and starts moving to the next given point. Between these motions (i.e. while the robot does not move), no hard real-time is required.

Using motion blending, subsequent motions can be blended into each other. Because the robot never stops, continuous motion control is required, thus hard real-time here is also required between two motions. KRL allows blended motions to be included in loops, thus even infinite motions where the robot never stops are possible in theory (but rather seldom required in practice).

Controlling I/O

Tool commands are usually issued by setting values to digital or analog outputs, and reading values from digital or analog inputs. Those inputs and outputs are usually connected to a fieldbus, which connects the robot to other devices such as a gripper control, a welding control or a programmable logic controller (PLC). Outputs and inputs can be used like normal variables and read/written anywhere in a KRL program, although accessing I/O using these variables blocks motion blending. Setting outputs during a motion is also possible using special trigger commands which execute the output operation at a certain point in space or time of the motion.

Result KRL programs provide hard real-time whenever required, e.g. when a tool command, or more general, any I/O communication, needs to be synchronized with a moving manipulator. The developer of robotics applications in KRL is completely relieved from all programming difficulties or issues related to real-time software development. KRL programs are always interpreted real-time safe. Motion blending is performed deterministically, i.e. it either succeeds for every or no run of a certain program.

3.4.2. GripperTech

The *GripperTech* technology package [79] provides support for standard electrical or pneumatic grippers. These grippers are connected to the robot controller with several digital input and output channels, which can be controlled by reading and writing digital input and output signals. Some grippers need a constant value on an output to remain open or closed, whereas some other grippers only require a short pulse to trigger a gripping action. Using digital input signals, the current state of the gripper can be verified, and allows the program to be stopped if the gripping process has failed.

The gripping actions themselves do not necessarily require real-time. Reading/writing digital inputs/outputs is not time critical, as long as the robot does not move while the gripping motion is performed. If a pulse signal is required with a strict timing restriction, a real-time system will be needed to control the signal pulse.

If the gripping action needs to be performed while the robot is in motion, e.g. at a certain distance from a programmed point or after a certain amount of time has elapsed after the motion has started, real-time is required to guarantee a precise execution of the program.

Result As long as the robotic application does not require the gripper to be used while the robot is in motion, the application itself does not need to be run with real-time constraints. If synchronous gripping while moving is desired, real-time is required for synchronization.

3.4.3. SpotTech

The *SpotTech* technology package [79] is designed to facilitate the development of spot-welding applications. The package is installed in combination with *GripperTech*, because welding guns for spot welding share some characteristics with grippers. The SpotTech package provides two new programming interfaces (so-called “inline forms”) which support moving to a welding spot, closing the welding gun and activating the welding systems (inline form `SPOT`) and retracting from the welding point again (inline form `RETRACT`).

The `SPOT` command moves the robot to a given destination point. The welding gun is closed during the motion such that it completes the closing movement at exactly the same time the robot stops at the destination point. Afterwards, the welding controller (often a programmable logic controller, PLC) is instructed to start welding. The completion of the welding process is determined by evaluating the responses from the welding controller.

The `RETRACT` opens the welding gun (either completely or to an intermediate pre-closed position) and starts a movement to an intermediate position. The following `SPOT` command continues the started motion and also takes into account whether the welding gun has been opened completely or just to the pre-closed position. When the gun has not been opened completely, much less time is needed for the gun to close, thus the closing command must be issued later for the following welding spot.

Result All commands included in this technology package require hard real-time, because the opening and closing commands must be issued to the welding gun at the proper time. Issuing the closing command too early would lead to the welding gun being closed before the robot has come to a stop, leading to damage on both the welding gun and the workpiece. Closing the welding gun too late leads to longer cycle times or even bad quality welding spots, if the failure of closing is not properly recognized.

3.4.4. BendTech

The *BendTech* technology package [78] provides a graphical programming interface for applications for bending and folding steel sheets. A typical work flow is to pick up a steel sheet from a pallet with a vacuum gripper and verify whether a single sheet has been picked up (steel sheets tend to stick together). Because the supply pallets are not always exactly placed at the same location, the steel sheet must be centered at the gripper. This is usually done by dropping the sheet into a special appliance, where it falls into a defined position, and picking it up again afterwards. The robot then inserts the steel sheet into the press, stopping when contact buttons detect the proper position of the sheet. Because the steel sheet is only inserted into the press with one side, the robot still has to hold up the sheet. The robot application then triggers the press to start, and the piston moves down and forms the steel sheet. The robot has to move synchronously with the piston to continue supporting the now upwards moving sheet. After the press has finished bending, the steel sheet is extracted and delivered for further processing.

When inserting the steel sheet, the contact push buttons must be monitored and the motion immediately stopped when contact is detected. This must be done with real-time guarantees, otherwise damage might occur if the inserting movement is stopped too late.

While the steel sheet is bent by the press, the robot has to follow the motion of the piston synchronously. This can only be done by processing current position values of the piston and generating new robot position set-points within a real-time system. If the delay between the measured piston position and the generated robot position is too large or fluctuates, the robot will not be able to follow the sheet properly.

Result The first part of such a program until the steel sheet is inserted into the press is mainly a program with gripping actions and thus has the same real-time requirements as identified for gripping or spot-welding applications (cf. Section 3.4.2). The second part however requires real-time synchronization with the piston for the whole bending process in order to follow the sheet properly.

3.4.5. Robot Sensor Interface

The *Robot Sensor Interface (RSI)* technology package [82] allows to influence the position of the robot by external sensors. For some applications, it is necessary to adjust the programmed robot trajectories with data gathered by sensor systems. For example, it is possible that the exact location of a welding seam differs slightly due to manufacturing tolerances in previous production steps.

With the Robot Sensor Interface, it is possible to overlay a motion programmed with KRL with external data. Usually, the external data provides relative correction values which are immediately applied to the running program. It is also possible to provide absolute position data, thus controlling the robot completely externally without a KRL program providing motions. External data can be supplied for example over a network connection.

Results All functions offered by this package require hard real-time. All external sensor information must be immediately processed and merged into the running program.

3.4.6. LaserTech

The *LaserTech* technology package [80] provides a programming interface for laser cutting and welding processes. For precise welding seams or cuts, the laser beam must be turned on and off synchronously with the robot being at the proper position. Furthermore, it can be necessary to adjust the laser power depending on the current speed of the robot.

Result The laser must be controlled synchronously with robot motions, which requires hard real-time.

3.5. Current solutions for real-time requirements in industrial robotics

As could be seen in the previous sections, industrial robotics applications inherently possess real-time aspects. Thus all robot manufacturers provide means of specifying real-time robot programs. Usually real-time safety is ensured by using proprietary domain specific languages which can be interpreted real-time safe and executed under a real-time operating system. The following sections provide an overview of the real-time implementations for some robot systems. The real-time capabilities of several more research projects are discussed in Section 4.3.

3.5.1. KUKA Robot Language

As it can be seen exemplary in Section 3.4, robotics applications developed with the KUKA Robot Language (KRL) can perform different tasks which require real-time programming, including motion blending. The KUKA Robot Controller (KRC) consists of an industrial PC (standard x86 hardware) which is connected to the motor controllers via a fieldbus (EtherCAT is used in case of the KRC-4 controller). The robotics application as well as motion planning and execution is performed on the PC, which therefore runs two operation systems: VxWorks [128] for all real-time critical tasks and Microsoft Windows XPe (for KSS 8.2) [76] or Windows 7 (for KSS 8.3) [77] for displaying the graphical user interface (non real-time). KRL programs are developed using an integrated editor running on the Windows part of the controller. For execution, KRL programs are transmitted to the real-time part of the controller and thus can be interpreted with real-time guarantees. Once a KRL program has been started, it will be executed completely within the VxWorks part of the system; the Windows-based user interface only shows diagnostic information such as highlighting the currently active motion command.

Using an interpreter running on a real-time operation system is already sufficient for the real-time requirements in motion control (cf. Section 3.2.1), as every single motion command can reliably and deterministically produce the set-points required by the hardware controllers. Some real-time requirements on application level however cannot yet be fulfilled, such as motion blending. To perform motion blending, it is necessary to process at least one motion ahead of the motion that is currently performed by the manipulator in order to plan the proper blending trajectory.

The KRL interpreter utilizes the so-called “advance run” to calculate trajectories for motion blending. While the main program counter points to the motion currently executed, the advance program counter already points up to 5 motions ahead. All commands are completely processed during the advance run, i.e. all motions are planned and all other non-motion commands (e.g. calculations) are executed. Changing the value of (local) variables during the advance run is possible because previous motions have already been completely planned and thus are not affected by changing variable values. Furthermore, KRL programs are always single-threaded, thus no other threads can be influenced. Not all KRL commands can be processed during the advance run. All commands directly influencing the environment such as I/O commands must be synchronized with the main program run, otherwise tool actions etc. would be triggered several motion commands too early. Pure writing commands (e.g. setting outputs) can be delayed until the main program counter has reached the command. Reading commands (e.g. retrieval of sensor values) however always need to be processed during the main run. Hence the KRL interpreter automatically stops the advance run whenever a command is found that must be processed during the main run, effectively disabling motion blending at this point.

KRL programs are always single-threaded; multiple threads are not possible. Triggers and interrupts allow tool commands (writing I/O) to be synchronized with motions or other events, and even complete sub-functions can be started. Motion commands are valid within such a sub-function if the robot is previously braked. Executing such a sub-function interrupts the main run of the program. Thus if the sub-function performs extremely time consuming calculations but does not interact with the actuator directly, all motions already planned during the advance run will be executed and the robot stopped afterwards. After a sub-function has been executed, it is possible to continue execution of the program or to abort the function where the interrupt was triggered.

Besides the currently active KRL program, there is also the so-called “submit interpreter” which always runs in the background. The submit interpreter executes a KRL program and is intended for background tasks such as monitoring user safety devices. It can be used as a replacement for a programmable logic controller in small environments. The submit interpreter shares resources with the main KRL program and is lower prioritized; it must not be used for time critical tasks. Furthermore, no commands that actively control the actuator (such as motion commands) are allowed.

The KRL language does not support programming multiple robots with a single program. It is possible to attach up to six external axes (e.g. a linear unit or a turn-and-tilt table), but no second articulated arm. Multi-robot applications are possible using the RoboTeam [81] option package which links multiple KUKA controllers. However, each robot still needs its own KRL program and synchronization is established by setting appropriate labels in each program.

The real-time safe execution of KRL programs is ensured by transmitting the source code of the program developed using the graphical editor to the VxWorks real-time operating-system for execution. KRL is a domain specific language which has been designed with real-time aspects in mind, thus it is possible to interpret all commands real-time safe. The language restricts the end user from creating indeterministic programs.

3.5.2. Stäubli VAL3

Stäubli industrial robots are equipped with the VAL3 [121] programming language. Similar to KUKA controllers, the Stäubli CS8C controller also uses the VxWorks real-time operating system on standard x86 hardware. However, Stäubli does not utilize two different operating systems for real-time motion control and non real-time user interaction, but rather uses a text-based user interface on their teach pendant (MCP) which is directly controlled from the VxWorks environment.

Motion commands in VAL3 (`movej` for point-to-point, `movel` for linear and `movec` for circular motions) are always non-blocking, i.e. control flow immediately returns to the next line in the program after the motion itself has been queued. The motion queue is processed asynchronously of the robot program and only synchronized upon explicit request using e.g. the `waitEndMove` command. It is also possible to interact with the motion queue from a program in form of stopping and resuming motion execution and by emptying the motion queue. The motion queue concept is comparable to the advance run of KUKA's KRL language and also allows for motion blending, however the VAL3 motion queue can also be manipulated by the program.

It is possible to execute several tasks (programs) simultaneously on a Stäubli robot controller. All tasks are executed with the same processor, thus the tasks are interleaved. It is possible to create asynchronous tasks which are executed as fast as possible, as well as synchronous tasks which are executed repeatedly with a fixed cycle time. Synchronization of tasks is explicitly done by the programmer using traditional methods such as a mutex.

Real-time safety of VAL3 programs is achieved similar to KRL programs by limiting the language to functions which can be deterministically interpreted. By allowing the end user to create multiple threads and synchronizing those threads, it is possible to create programs where motion blending cannot be guaranteed (e.g. if a task is blocked before the following motion command has been enqueued). Single motions however will always be executed atomically and thus deterministically.

3.5.3. OROCOS

The Open Robot Control Software (OROCOS) [21, 23, 24] project provides a C++ framework for the development of real-time robotics applications. It aims at being a modular and flexible framework that supports all sorts of robotics devices and computer platforms (i.e. processor architectures and operating systems). The main components of the Orocos project are:

- *Real-time Toolkit (RTT)* [113]: Component framework for real-time robotics applications. The RTT framework creates an abstraction layer of the operating system (currently Linux RTAI and Xenomai extensions are supported for real-time applications, and standard Linux and Windows for non real-time applications). It provides an object-oriented interface for threads, mutexes, etc. upon which robotics applications can rely. The component framework supports basic communication mechanisms for inter-component communication with real-time guarantees. Components can even be transparently distributed across multiple systems, in this case inter-component communication is performed using CORBA[2] [124] (without real-time).
- *Orocos Component Library (OCL)*: Support tools for RTT, such as for setting up or monitoring RTT based applications.
- *Kinematics and Dynamics Library (KDL)* [112]: KDL provides libraries to support many generic concepts such as vectors, frames, rotation-matrices which are required for kinematics and dynamics calculations in robotics applications. Attention was paid to ensure real-time safety of all algorithms.

Orocos allows developers to create applications with a rather high level of abstraction from hardware and operating system specifics. Applications developed with the RTT can be tested on non real-time operating systems and later executed on real-time Linux systems without changes to the code.

Unlike KRL or VAL3, Orocos is aimed at providing support for generic real-time robotics applications with complete control of all aspects such as motion planning, motion interpolation, dynamics calculations etc. Orocos does not impose a specific architecture on the system, but rather aims at providing a real-time core which is suitable for different architectures [24]. Orocos does not provide any hardware device drivers to interact with industrial robots directly (although some device drivers have been developed for robots in the lab at KU Leuven), but eases the development of such drivers by providing an appropriate framework.

Real-time safety of applications developed using the Orocos framework is achieved by using a programming language (C++) which can generate deterministic code and execute it under a real-time operating system. It is completely up to the developer to ensure that all algorithms are real-time safe, i.e. that memory management is performed at the right times and that no other actions are performed which introduce indeterminism (such as I/O). Unlike programming with KRL or VAL3, using the Orocos framework does not prevent the developer in any way to create indeterministic code, however allows direct interaction with hardware devices for experts.

[2]Common Object Request Broker Architecture

3.6. Conclusion

All analyzed applications and technology packages show, that a certain amount of hard real-time is required for robotics applications. Every motion for itself requires a real-time motion controller, and generally tool actions also need to be synchronized to robot commands with real-time guarantees. Most of the time, these synchronization conditions specify that a certain tool action must happen at a certain position of the robot's path, or after a certain amount of time has passed since starting a motion. However, there are also cases where tool actions need to be synchronized to the whole robot motion (cf. LaserTech), or the robot motion needs to be synchronized to external tools (cf. BendTech, Robot Sensor Interface).

More importantly, it could also be seen that between two distinct robot motions, it is usually acceptable if small delays occur, although these delays can reduce the cycle time. With motion blending, no gap occurs between two motions, thus all motions connected with motion blending must be considered as a single real-time critical motion.

The real-time requirements for most robotics applications can be summed up as follows:

- Each single motion requires real-time safe motion interpolation.
- It must be possible to synchronize tool actions with robot motions with precise timing guarantees.
- Between multiple motions short breaks are usually tolerable.
- Motion blending combines several otherwise independent motions into a single continuous motion. However rarely failing to perform motion blending usually is tolerable.

It can be concluded that the overall program flow of robotics applications generally does not need to be executed real-time safely, but rather a best-effort approach is sufficient. This allows to partition robotics applications into a non real-time general program flow that coordinates short real-time critical tasks. It is necessary however, that the developer can flexibly define which parts of the applications need to be executed real-time safely, e.g. for synchronizing tool actions or reactions to sensor events.

Generally the developers of robotics applications should not need to care about writing real-time safe code. The programming languages supplied by commercial robot manufacturers hide all issues related to real-time from the developer. This is very important to facilitate the development of applications and to reduce the likeliness of erroneous programs.

Chapter 4

The SoftRobot architecture

The joint research project "SoftRobot" was conducted by the Institute for Software & Systems Engineering at Augsburg University (ISSE)[1], KUKA Laboratories GmbH (KUKA)[2] (the research department of the robot manufacturer KUKA located in Augsburg) and the system integrator MRK Systeme GmbH (MRK)[3]. It ran from October 2007 until March 2012. The project was funded by the European Union and the Bavarian government.

The project aimed at creating a new programming paradigm for industrial robotics by introducing modern programming languages into the domain of industrial robotics. Thereby all special requirements of the robotics domain, especially hard real-time requirements, need to be met.

4.1. Goals

Industrial robots today are usually programmed using proprietary programming languages supplied by the robot manufacturer. There is no common language used by all robot manufacturers (unlike e.g. IEC 61131-3 [59] which defines common languages for programmable logic controllers (PLCs)), but each manufacturer provides its own language.

Besides the need for developers to learn a new programming language for each different robot system, this approach also has some other drawbacks. The programming languages must be maintained by the robotics companies, while general purpose languages are

[1]http://www.isse.de/
[2]http://www.kuka-labs.com/
[3]http://www.mrk-systeme.de

usually maintained either by companies specialized in programming languages, or by large communities. This often leads to the robot programming languages not supporting modern features (e.g. object orientation) and having a very limited set of instructions.

During the SoftRobot project, a new programming platform should be developed which allows to program industrial robots using a modern, object-oriented programming language. This provides several advantages: Modern methods of software engineering can be applied, it is possible to create a (object-) model of the application which helps understanding the system, good community support is available for most modern object-oriented programming languages and many developers are already confident using these languages. Furthermore, a programming language with automatic memory management should be used to reduce the likeliness of programming errors further.

At the beginning of the development of the SoftRobot platform, several key requirements for the SoftRobot platform have been identified. Those requirements have also been discussed in [3] and in [125, 126].

1. *Usability:* The current proprietary robot programming languages (e.g. KUKA KRL) offer a high grade of abstraction for the developer. Using only a few keywords, it is possible to write programs for robots and interact with tools attached to the robot. The developers do not need to care about real-time issues at all when using these languages. When using the new SoftRobot architecture, the same user experience should be available, i.e. small programs should also be easy to write. Additional features introduced by SoftRobot which are not possible with the current languages should not complicate small and easy programs.
2. *Multi robot systems:* The SoftRobot platform should support applications consisting of multiple robot systems. Programming those robots should be possible using a single program or multiple programs.
3. *Sensor support:* New applications will require more support for external sensors (e.g. a camera, a force-torque sensor). Those sensors should be easy to integrate into robotics applications.
4. *Extensibility:* It should be possible to extend an existing application based on the SoftRobot platform to use another type of robot, or multiple robots without the need to completely rewrite the application. Furthermore, it should also be possible to extend the SoftRobot platform itself, e.g. by supporting new kinds of robotic devices (e.g. mobile manipulators, flying robots).
5. *Special industrial robotics concepts:* Special requirements of industrial robotics should be catered for. These requirements are for example motion blending (cf. Section 2.5) and force manipulation with compliant motions (cf. Section 2.3).

In order to fulfill the requirements, the new SoftRobot platform must support the execution of real-time safe programs (in particular for requirements 3 and 5, and for the synchronization of multiple systems as of requirement 2). However, in order to fulfill requirement 1, the end-user should not be required to develop the application

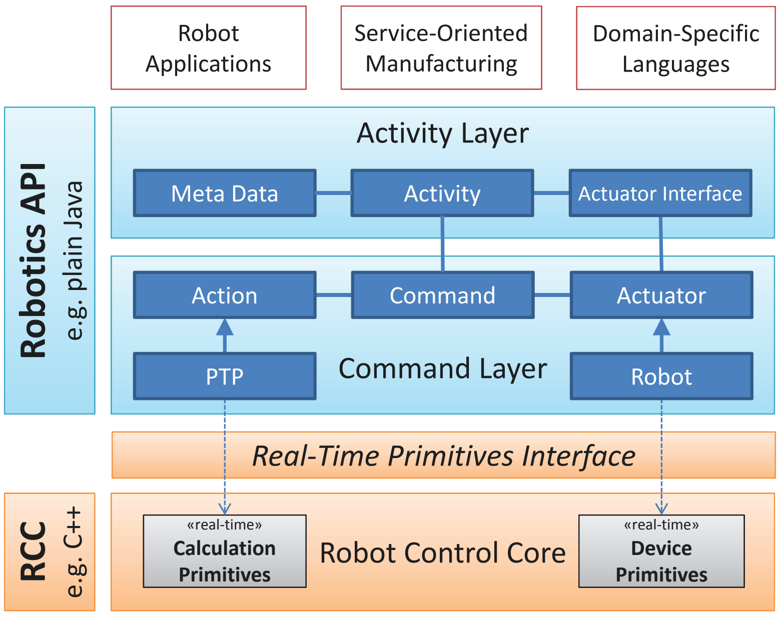


Figure 4.1.: The SoftRobot architecture (adapted from [125])

code itself in a real-time safe manner, with all implications to memory management, avoidance of I/O, etc. (cf. Section 3.3.2). Since programming languages with automatic memory management generally do not provide real-time safe program execution, a new architecture had to be developed.

4.2. Architecture

In order to fulfill all requirements and goals specified in Section 4.1, a three tier architecture has been chosen for the SoftRobot platform. Figure 4.1 depicts the overall architecture. This architecture has first been introduced in [55]. Splitting the system into different layers with some providing hard real-time while others do not was driven by the finding that robotics applications usually only require hard real-time for small tasks, while the overall program flow does not (cf. Section 3.6).

The lowest layer, the *Robot Control Core (RCC)*, communicates directly with the hardware (e.g. robots, sensors) and must be implemented using a programming language and operating system which both support real-time execution, e.g. C++ and Linux/Xenomai. On top of the RCC, the *Robotics API* provides support for application developers to create robotics applications using a large class library for robot control. The Robotics API does not need to support real-time execution by running under a real-time operating system, but must provide concepts to specify real-time tasks for execution on the RCC. The class library can be implemented using any modern, object-oriented programming language, e.g. Java or C#. The Robotics API communicates with the RCC using the

Real-Time Primitives Interface (RPI). The Robotics API can be used directly by robotics applications or by other systems further facilitating robot programming, such as domain specific languages (DSL) for highly specialized applications.

By splitting the architecture up into a real-time safe and a non real-time part, it is now possible to use object-oriented programming languages with automatic memory management and still provide hard real-time support when necessary.

4.2.1. Robot Control Core

The *Robot Control Core (RCC)* is the lowest layer of the SoftRobot architecture and communicates directly with all hardware. It is the only part of the SoftRobot system that must be developed real-time safe and must be executed using a real-time operating system. The RCC requires a real-time driver for every type of hardware (i.e. every type of robot, sensor, etc.) that is to be supported. Many hardware device drivers require new set-points or provide new measurements at a high frequency (usually up to 1 kHz). The RCC is responsible for deterministically providing or consuming these values.

Besides hardware device drivers, the RCC also must provide real-time safe implementations of some basic calculation primitives which can be used by robotics applications to describe algorithms. These primitives can provide very basic functionality such as addition or multiplication of numbers, but also more complex operations e.g. working on vectors or matrices. All aspects of a robotics application which are required to be real-time safe must be described using only the basic calculation primitives provided by the RCC. By using these flexibly combinable and real-time safe primitives for providing new functionality, neither the robotics application itself nor the Robotics API needs to be real-time safe.

Within the SoftRobot project, a reference implementation of the RCC has been developed, the *SoftRobot RCC*. This reference implementation is described in detail in Chapter 7.

4.2.2. Real-time Primitives Interface

The *Real-time Primitives Interface (RPI)* defines the protocol that allows robotics applications to communicate with the RCC, in particular for the specification of real-time tasks. This protocol defines two different aspects that are needed for robotics applications:

- *Real-time primitive nets:* Real-time primitive nets define tasks for the RCC which must be run with hard real-time guarantees. The tasks are described using a data-flow language which is cyclically evaluated. An in-detail description of the language is provided in Chapter 5. The language has been designed for automatic generation by the application. The Robotics API provides a mapping algorithm that generates the required real-time primitive nets from the application code (cf. Section 10.2).

- *Synchronization of primitive nets:* While real-time primitive nets provide hard real-time capabilities, it is sometimes sufficient to provide only soft real-time. Motion blending is one example where a best-effort approach can be acceptable. If blending one motion into another fails, the robot comes to a short stop but the application will still provide useful results, however with reduced performance.

  Primitive nets need to be specified in one piece, i.e. no changes to the primitive net are possible once the execution has started (otherwise the net would be no longer real-time safe). It is possible to specify blended motions within a single primitive net in order to gain guaranteed motion blending, however for large sequences of motions this also leads to huge primitive nets.

  Using the synchronization features of the Real-time primitives interface, it is possible to request the immediate and synchronous start and stop of primitive nets once a specified event occurs. The acceptance of such a synchronization rule can only be done on a best-effort base, however it can be guaranteed that once the synchronization rule has been accepted, starting and stopping of the given primitive nets will be strictly synchronous. The synchronization mechanism for multiple primitive nets is described in detail in Chapter 6.

The Real-time Primitives Interface was first introduced in [127] and later refined in [125, 126].

4.2.3. Robotics API

The Robotics API provides a huge class library which can be used for developing robotics applications using a modern, object-oriented programming language. During the SoftRobot project, the Robotics API has been implemented using Oracle Java [70]. Using IKVM.NET [65], an automatic transformation of all Java classes to .NET classes is possible, thus the implemented Robotics API can also be used with any .NET based language (e.g. C#, VB.NET or F#).

The Robotics API provides support for all kinds of actuators and sensors which are supported by hardware device drivers of the RCC, as well as generic task functionality such as different types of motions for robot arms. Furthermore, the Robotics API provides modeling techniques for the environment of the robotics application, e.g. by providing a world model using frames (a point in space including orientation) and transformations between frames.

The Robotics API can be split up into two parts, the *Command Layer* and the *Activity Layer*. The command layer describes robot tasks on a very abstract level, i.e. a command is formed by an action and an actuator. An action defines *what* should be done (e.g. a point-to-point (PTP) motion), while an actuator represents any kind of device that is capable of executing that action. Action themselves do not provide information *how* their task must be executed, as this may be highly dependent on the type of actuator that is used. At run-time, commands are mapped to primitive nets which are executed by an RCC. The information of how to execute an action is retrieved during this mapping process. More details about this automatic mapping process are described in Section 10.2. The command layer also provides a complex event mechanism that allows to execute further arbitrary commands in real-time on the occurrence of an event (which can be described by using arbitrary sensor information such as the current position of the robot or an external camera system).

Although it is possible to create arbitrarily complex robotics applications using only the command layer, this is not a very convenient way to do so. The activity layer adds convenience functionality to provide an easy-to-use programming interface for common tasks.

The concepts of the Robotics API have been first introduced in [4] and have been refined in [3].

4.2.4. Applications

Robotics applications can be developed on different levels of the Robotics API, and with different target users. Classic applications controlling a single robot and the attached periphery are likely to be developed using the Java programming language on top of the activity layer, thus using prepared robot motion instructions and code structures comparable to existing proprietary robot programming languages such as KRL. Using the same programming interface, it is also possible to develop multi-robot applications, including real-time synchronization of the robots.

If the functionality provided by the activity layer is not sufficient, it is also possible to develop an application based directly on the command layer of the Robotics API. This is e.g. required for system integrators who wish to integrate a new piece of hardware. Programming on that layer offers the full flexibility and real-time guarantees that are available with the SoftRobot platform, but less convenience functionality is provided, thus the command layer is harder to use than the activity layer.

However, not all end users need to interact with the Robotics API or even any classic programming language at all. For special applications it is possible to create a *Domain Specific Language (DSL)* on top of the Robotics API, which focuses only on the special domain it is intended for. Such a DSL can be e.g. a graphical programming language, or an application of programming by demonstration, i.e. recording and replaying motions executed by a human. Building a DSL on top of the Robotics API is a great improvement to proprietary programming languages, mainly because of the availability of a broad variety of tools and already existing solutions e.g. for DSL generation or human-machine-interfaces (HMI), which are usually not available for specialized robot programming languages.

For the control of a complete production environment, a service-oriented architecture (SOA) can be a solution that is easier to develop, maintain, and extend in comparison to classic systems controlled by a programmable logic controller (PLC). Studies showed that the use of the Java programming language can lead to dramatic increases in efficiency compared to the traditional IEC-61131-3 programming languages commonly used for PLCs [97]. Methods for the development of service-oriented systems based on the Robotics API have been created by Hoffmann et al. [56, 57].

4.3. Related work

For a long period of time, robot programming has been done using dedicated robot programming languages (e.g. commercial languages: KUKA KRL [77], Stäubli VAL3 [121]; research projects: WAVE [95] and AML [118]). Research recently has been mainly concentrating on experimental and mobile robotics, as the industrial robotics domain has even been thought of as a solved problem [50, p. 983]. However, advantages of using a general purpose programming language for the robotics domain have been identified earlier. For non object-oriented programming languages, RCCL [52] for the C and PasRo [14] for the Pascal programming language are examples. With object-oriented programming languages becoming more and more popular, there have also been attempts to integrate those languages into the robotics domain. ZERO++ [96], MRROC++ [130], RIPE [87], the Robotics Platform [85] and SIMOO-RT [5] are some examples. A more in-depth analysis of some of these projects is provided in Sections 5.6 and 6.5.

Integrating current off-the-shelf industrial robots in advanced applications such as programming by demonstration using speech recognition (cf. [98]) or creating an object-oriented model for a specific task such as plastic injection molding (cf. [47]) can be a tedious task. The SoftRobot architecture significantly reduces the effort required for such tasks by providing a ready-to-use object-oriented interface which is independent of the actual robot that is used. The integration of additional technologies such as speech recognition is much easier if a widely used programming language is available also for the robot program.

Many robotics systems are created with three interacting tiers. This has been denoted as a *3T architecture* by Bonasso et al. [15]. Applications are separated in this paper into “a dynamically reprogrammable set of reactive skills coordinated by a skill manager” as the bottom layer, “a sequencer that activates and deactivates sets of skills” as the middle layer and “a deliberative planner that reasons in depth about goals, resources and timing constraints” as the top layer. Simmons et al. [110] denotes the three layers from bottom to top as “behavior”, “executive” and “planning” layers. The SoftRobot architecture follows a comparable separation of concerns. The lowest layer, the behavioral layer is provided by the RCC which performs direct hardware control. The RCC can be reconfigured dynamically by providing new primitive nets. The executive layer is provided by the Robotics API which is able to combine the basic primitives offered by the RCC into primitive nets for the desired task. The planning layer is represented by robotics applications using the Robotics API. Robotics applications can decide what tasks to perform, at which time to do so and e.g. which robots to use (if multiple actuators are available).

Component based software engineering has gained popularity in the recent years in the robotics domain, in particular for mobile or service robots. It aims at creating reusable components that can be composed to new applications while requiring only small amounts of new code to be written (cf.[19, 20]). The Orocos framework [21] provides a C++ based component framework which can be used to create real-time safe components. The MiRPA project [42, 44] created a real-time capable middleware that even allows to distribute components across different systems without losing real-time safety. For component based software design of robotics applications, a separation of concerns into five different areas (the “5Cs”) has been proposed as best-practice [22, 99, 123]. The five concerns are: communication, computation, coordination, configuration and composition. The ROS project [101] gained much popularity in the robotics research domain in recent years, particularly by providing good tool support and a large component library which provides reusable components for many common tasks. Initially, the ROS project provided support for service robots, in particular the PR-2 robot from Willow Garage. Real-time aspects did not play an important role at the beginning, although it is possible to integrate Orocos to create single real-time safe components. The ROS-Industrial project tries to extend the ROS framework to support industrial applications.

The SoftRobot project took a different approach by providing an object-oriented programming interface for application developers, and hiding the real-time aspects in lower layers as much as possible. The development of the object-oriented programming interface in the SoftRobot project was driven by the belief that standard industrial robotics applications with a rather linear workflow can be programmed best using “traditional” programming languages with an explicit interface rather than by composing components. Internally, the SoftRobot project also employs components which can be flexibly rearranged for new applications, in particular in the Real-time Primitives Interface with the primitives. Unlike most component based systems however, RPI enforces a strictly synchronous execution of all components within a single primitive net.

Despite providing an explicit programming interface, the Robotics API can also be used to create new programming paradigms which are more suitable for applications in certain fields. For instance, the Factory 2020 example used for the final project presentation (cf. Section 11.2.1) has been programmed using a service-oriented platform and coordinated using state charts with *State Chart XML* (SCXML) [114].

The iTaSC (instantaneous task specification and control) approach [31] aims at creating a novel robot programming paradigm for complex robotics tasks including sensor-based manipulation with geometric uncertainties. Tasks are specified by describing constraints for a “virtual kinematic chain” between frames on objects (such as the robot or the workpiece) and distinct features on these objects. Using automated solvers and optimization algorithms, the trajectories for the robot (or for multiple robots) can be determined. The constraint based programming approach seems promising for certain complex tasks. Section 5.6.4 describes a possible integration scenario of the iTaSC task specification and constraint solving functionality into primitive nets.

Chapter 5

Real-time Primitive Nets

The *Real-time Primitives Interface (RPI)* describes a small but very generic interface for the specification of real-time safe tasks. In the SoftRobot project, RPI is used for the communication of the non real-time Robotics API and the real-time Robot Control Core (RCC). In the context of this work, the term *robotics application* denotes any program that is based on RPI, unless stated explicitly otherwise. Robotics applications can be built using the Robotics API, however RPI is a generic interface that can also be used by any other framework. RPI was first introduced in [127] and later refined in [125, 126]. Using RPI, it is now possible for robotics applications to run without a real-time safe operating system while still being able to create real-time safe tasks. The tasks are specified by a flexible combination of basic building blocks. Seamless, real-time safe transitions between such tasks are possible using the synchronization mechanism explained in Chapter 6.

5.1. Introduction

The Real-time Primitives Interface (RPI) has been inspired by the data-flow language Lustre [51], used in the commercial tool SCADE [104]. Although some concepts are shared with other data-flow languages, RPI is tailored to the needs of the robotics domain. For example the concept of *Fragments* has been introduced which behave much like *nodes* in Lustre, but additionally also serve as performance improvement by activating or deactivating large parts of the data-flow graph as required.

The main concept of RPI is the *primitive net*. A primitive net describes a task which must be executed real-time safe. Before the execution of such a primitive net can start, it must be completely specified, i.e. no further changes are possible (with the exception of

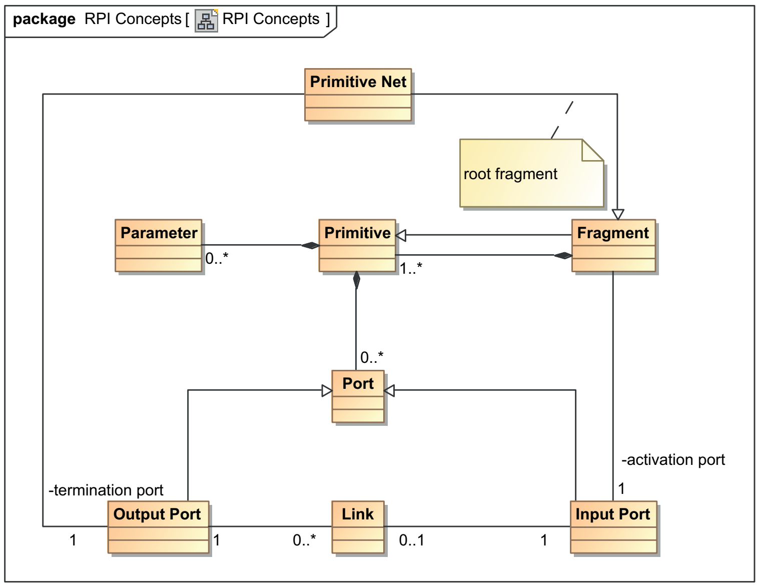


Figure 5.1.: UML concept diagram of the Real-time Primitives Interface

processing of sensor values within the primitive net). The robotics application generates one primitive net for all tasks that need to be synchronized and creates another primitive net if two actions do not need to be strictly synchronized. Because robotics application in general need not to be real-time safe, a delay between the submissions of two subsequent primitive nets is acceptable. Therefore, with the exception of multiple synchronized primitive nets (cf. Chapter 6), all robots and other systems must be in a safe state once the execution of the primitive net has finished. For robots a safe state usually means that the robot is neither moving, nor applying force to the environment.

RPI is designed to be generated by the robotics application. It is not intended to be human readable or writable, although with some experience, it is possible to design small primitive nets by hand. However, real-world primitive nets tend to grow large, partially due to the fine granularity of building blocks provided by the SoftRobot RCC.

5.2. Components

The Real-time Primitives Interface consists of only a few basic components, which are sufficient to build large real-time safe tasks. The basic concepts are displayed in Fig. 5.1. Primitive nets consist of basic building blocks, so-called *primitives* which can be connected by *links* to transfer data from one primitive to another primitive. The configuration of primitives is done by *parameters*. The execution of primitive nets is performed cyclically, i.e. every primitive contained in a primitive net is executed once in each execution cycle.

5.2.1. Primitives

Primitives form the basic building blocks of primitive nets. Primitives can have *input* and *output ports*, and can be configured using *parameters*. Primitives are neither required to have input ports nor output ports, even primitives with no ports at all are valid. In each execution cycle, all values from the input ports are read, and new values for the output ports must be provided. Primitives can perform arbitrary calculations during execution, however all operations must be real-time safe such that every primitive can guarantee a worst case execution time (WCET).

Primitives can perform very basic operations such as logical operators (AND, OR, etc.), mathematical functions (add, subtract, multiply, etc.), but also more complex operations such as calculating trajectories. Hardware devices are also represented as primitives. Sensors are primitives with only output ports, whereas actuators are primitives with only input ports (with the exception of error and diagnosis outputs).

Input and output ports are strictly typed, and only ports with matching types can be connected. Types can be basic types such as integer or double types, but also more complex types are possible. Complex types could be for example Cartesian coordinates, consisting of (double) values for the X, Y and Z direction. Furthermore, it is also possible to use a special *null* value to indicate that no valid value is available.

A primitive can be instantiated multiple times within the same primitive net. Multiple instances of a primitive that control the same hardware device are allowed, however only one such instance may be active during any given execution cycle. Activation of such primitives can be either done using fragments (cf. Section 5.2.3) or a dedicated activation input port. The implementation of primitives must contain code to detect situations where two instances try to control the same hardware within a single execution cycle and emit an appropriate error code.

Parameters can be used to configure primitives. Parameters are typed like ports and behave very similar to input ports, with the exception that the value of a parameter does not change during the execution. Unlike input port values, parameter values are already available when the primitive is initialized. During initialization phase, no real-time constraints must be obeyed. Primitives may require certain configuration information during initialization e.g. to allocate the right amount of memory.

Primitives may have an internal state which is preserved between two execution cycles. This can be used for example to follow a trajectory in order to provide new set-points in each execution cycle. Primitives must not preserve information between two different primitive nets. If information preservation is required, this should be done by means of the robotics application. If several instances of the same primitive are configured identically (i.e. their input ports are connected to the same sources and they have the same set of parameters), those instances must be free of side-effects within a single execution cycle. This implies that all instances provide the same output values within the same execution cycle. This behavior is exploited by the mapping process of the Robotics API to eliminate redundant parts of a primitive net automatically (cf. Section 10.2.7).

5.2.2. Links

Links are used to connect ports of two primitives. Only ports of the same data type can be connected. If a conversion of data types is required, special conversion primitives must be inserted. Each input port can be connected to exactly one output port, however an output port may be connected to several input ports. It is not necessary for all ports to be connected, however primitives may require input ports to be connected. Actuator primitives for example need their set-point input port to be connected, otherwise the primitives cannot perform their tasks.

The output value of a primitive is always transmitted to the input port of the following primitive before the execution of that following primitive is started. This design allows for a fast propagation of values through the primitive net, e.g. values received from sensors can be processed and delivered to actuators within a single execution cycle.

Links may not form *unguarded* cycles in a primitive net. A primitive net contains an unguarded cycle, if it is possible to reach a primitive again by purely navigating links from output to input port without coming across a *Pre* primitive. Such cycles would imply that the input for a primitive instantaneously depends on the result of this very same primitive. Although there might be a unique fixpoint to solve such a situation, this is not necessarily the case (e.g. connecting the result of an addition primitive to one of its inputs could create an unsolvable equation such as $x = x + 2$). For the same reason, Lustre does not allow equations where a variable instantaneously depends on itself [27, Section 4.1]. *Guarded* cycles within a primitive net contain at least one *Pre* primitive. This special primitive reads the value from its input ports, and only delivers the value to its output port in the next execution cycle. The output port provides *null* during the first execution cycle. The *Pre* primitive works similar as the `pre` operator in the Lustre language [51]. Only guarded cycles are valid in primitive nets. To determine the execution order for all primitives, the primitive net is topologically sorted (*Pre* primitives are treated in a special way, cf. Section 5.3.1). Unguarded cycles are detected during this step.

5.2.3. Fragments

Using primitives and links, it is possible to create flat primitive nets. The semantics requires all primitives to be executed once in each execution cycle, however, not always all primitives have a task to do all the time. For example, some parts of a primitive net can be required for error handling. In this case it is sufficient to detect once that no error has occurred, and not every single primitive required for error handling has to detect this situation individually. *Fragments* provide means of a hierarchical structure in primitive nets that also allows to activate and deactivate complete branches globally.

Fragments group primitives and behave like primitives themselves, i.e. they can be used in primitive nets like every other primitive. Fragments also have input and output ports, but no parameters. The input ports of a fragment are connected to input ports of primitives contained in the fragment, and output ports of primitives are connected to output ports of the fragment. Because fragments can be considered as primitives, fragments can also be nested.

Every fragment has an additional Boolean input port for activation besides the input ports required for the contained primitives. If this port receives a *false* value, the contents of the fragment are not executed. This can be used for example to disable the execution of an error handling fragment while no error has occurred. The same effect could be obtained by connecting an active input port to each single primitive, however this would require many more links and also much more computation time during runtime.

Because fragments can completely disable the execution of the contained primitives, no primitive may expect to be executed in every execution cycle. Because some primitives may need the current time, the runtime environment has to provide the current cycle counter and the cycle period to every primitive.

The primitive net itself is also a fragment. This *root fragment* must not have any input ports, but is required to have exactly one output port for signaling the desire of the primitive net to terminate (cf. Section 5.3.3).

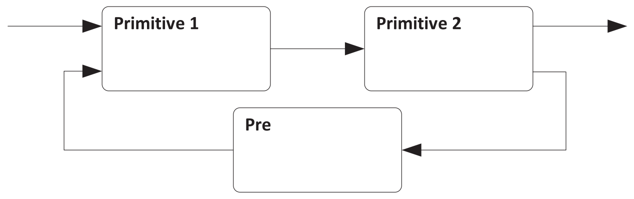
5.3. Runtime

The Real-time Primitives Interface has been designed for the needs of robotics applications. Not only the static structure of the data-flow language and the execution semantics for each execution cycle is defined, but also the overall lifecycle of a primitive net. This section describes the execution of a single primitive net, and Chapter 6 describes the synchronization of multiple primitive nets.

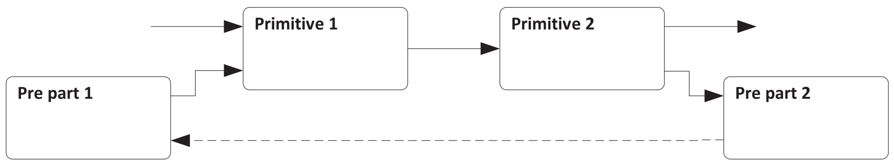
5.3.1. Basics

The execution of primitive nets is performed cyclically. Every primitive instance is executed once in each execution cycle, with the exception of primitives contained in inactive fragments. Before the execution of the primitive net starts, all primitives are topologically sorted according to the links. By sorting the primitives, it can be guaranteed that all primitives connected to input ports of other primitives will already have been executed and thus always current values are present at the input ports. This also enables fast value propagation. New sensor values can be completely processed and delivered to actuators within only a single execution cycle.

Unguarded cycles (i.e. cycles not containing a *Pre* primitive) within a primitive net are not allowed. If cycles are required (e.g. for closed-loop control), *Pre* primitives can be used to delay the propagation of values to the next execution cycle (cf. `pre`-operator in Lustre). Figure 5.2a depicts a primitive net with a guarded cycle. Topological sorting of this net without modification of the *Pre* primitive would be impossible, thus the *Pre* primitive is split up into two parts before the primitive net is topologically sorted. The



(a) Primitive net with a guarded cycle containing a *Pre* primitive



(b) The *Pre* primitive is split in two parts prior to the topological sorting of the primitive net

Figure 5.2.: Splitting of a *Pre* primitive for topological sorting

resulting net is shown in Fig. 5.2b. The input ports of the original *Pre* primitive and their connections are taken over by part two of the split *Pre* primitive, while the output ports and connections are taken by part one. Both parts are connected internally (displayed as a dashed arrow), but this connection is not considered during the sorting process, thus the cycle is resolved. Part one of the primitive will always be sorted for execution prior to part two and thus can deliver the value received by part two and stored in memory during the previous execution cycle. During the first execution cycle, a *null* value is delivered by part one.

5.3.2. Execution phases

Each execution cycle of a primitive net is split up into three phases:

1. Read sensor values
2. Execute each primitive and propagate values throughout the primitive net
3. Write new set-points to all actuators

During the first phase, all primitives that are related to sensors must acquire current sensor values from the underlying hardware devices. After sensor data acquisition has been completed, all primitives (including the sensor primitives again) are executed. The order has been determined earlier by topologically sorting all primitives with respect to the connecting links. During this phase, all calculations are performed, i.e. sensor values processed and new set-points generated. In the final third phase, all primitives related to actuators must distribute the new set-points to their hardware devices.

Three separate phases have been designed to guarantee that all sensor values are read simultaneously (and not depending on the location of the sensor primitive within the primitive net) and actuators also provided simultaneously with new set-points. The

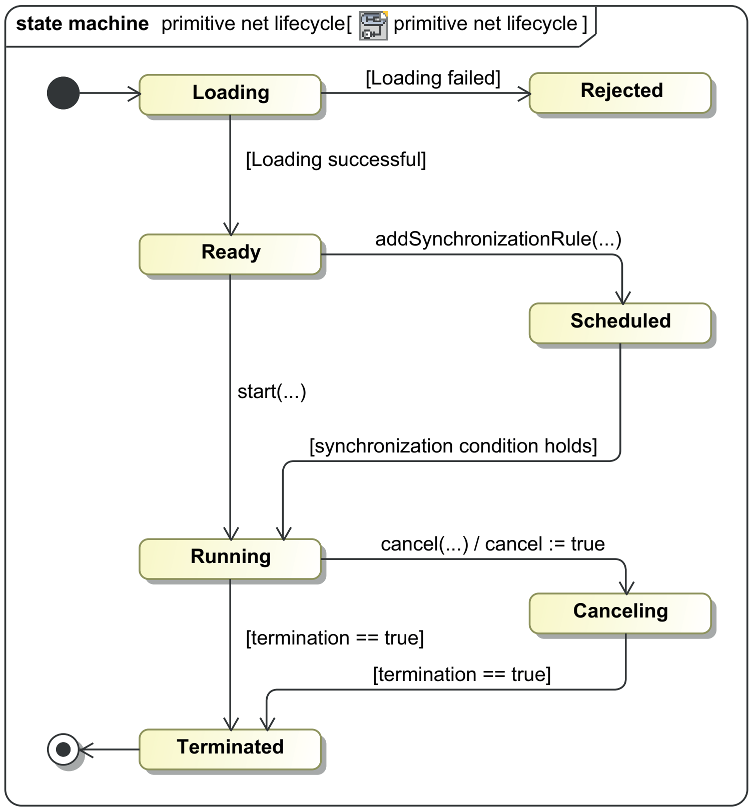

state machine primitive net lifecycle[primitive net lifecycle]
Loading
[Loading failed]
Rejected
[Loading successful]
Ready
addSynchronizationRule(...)
Scheduled
start(...)
[synchronization condition holds]
Running
cancel(...) / cancel := true
Canceling
[termination == true]
[termination == true]
Terminated

Figure 5.3.: Lifecycle of a primitive net

phases one and two may be interrupted at any time since no modification to any hardware device has been made. Phase three must always be executed atomically in order to guarantee a consistent system state, i.e. either all actuators or no actuator have been provided new set-points.

Phase two is usually the most time consuming phase since all complex calculations need to be performed during this phase. For the synchronization mechanism (cf. Chapter 6) it is important that it is possible to interrupt any primitive net before phase three has started to allow for a fast transition to a successive primitive net, even after an execution cycle for the prior primitive net has already been started.

5.3.3. Lifecycle

During its lifetime, a primitive net can have several states, which are also depicted in Fig. 5.3:

- **Loading:** The primitive net has been submitted to the execution environment and is currently being loaded and initialized.

- **Rejected** The primitive net could not be loaded by the execution environment. There are different reasons for primitive nets being rejected:
  - The transmitted primitive net is syntactically invalid, i.e. not formed according to Fig. 5.1.
  - At least one primitive is not available in the given execution environment.
  - At least one primitive could not be initialized, e.g. due to invalid parameters, unconnected but required input ports or lack of memory.
  - The primitive net contains unguarded cycles.
- **Ready:** Loading of the primitive net is finished, the net is ready to be started.
- **Scheduled:** The primitive net is currently scheduled for the immediate execution once the condition of a synchronization rule becomes true. See Chapter 6 for more details on the synchronization of multiple primitive nets.
- **Running:** The primitive net is currently running.
- **Canceling:** The primitive net has been requested to terminate, however the primitive net has not yet terminated.
- **Terminated:** The primitive net's execution has stopped.

The real-time synchronization of multiple robotics devices is implicitly achieved by placing the trajectory generation for all devices in the same primitive net and thus simultaneously generating set-points for all actuators. Because in every execution cycle all primitives are executed, trajectories for multiple devices will be synchronized throughout the whole life-time of the primitive net.

Primitive nets are not designed to be hand-crafted. As introduced in Section 4.2, RPI serves as an interface between the non real-time safe robotics application and the hard real-time execution environment. Section 10.2 explains the so-called "mapping" process which is used by the Robotics API to transfer Java-based programs automatically into primitive nets.

The robotics application must serialize and transmit the generated primitive net to the execution environment. The Robot Control Core immediately starts to load the primitive net, which enters the state *Loading*. During the loading phase, all primitives are instantiated and initialized. Code executed at this time does not need to be real-time safe, i.e. the primitives may perform all necessary initialization steps such as memory allocation. Values of parameters are already available to the primitives, however it is not yet possible to read input ports. If the initialization succeeds, the primitive net enters the state *Ready*, otherwise it is *Rejected*.

A primitive net ready for execution can either be started directly, or scheduled for later execution. Scheduling allows a primitive net to be started immediately after a logical condition becomes true, e.g. after one or more other primitive nets are ready to handover to the next primitive net. Synchronization of primitive nets is described in detail in Chapter 6. Once the primitive net has been started or the synchronization condition becomes true, it enters the *Running* state. During this phase, the primitive net is executed cyclically with fixed intervals. Primitives may only perform real-time safe operations now. At each execution cycle, actuator primitives must be provided with new set-points which can be used for direct hardware control.

Whenever a primitive net has performed its task, it must signal its desire to terminate to the execution environment in order to stop the cyclic execution, and to allow further primitive nets to control the hardware. The root fragment of the primitive net has a discrete termination output port. Once this port is set to *true*, the execution environment will stop the cyclic execution after the current cycle, and the primitive net enters the *Terminated* state.

Besides the ability to detect the termination desire of a running primitive net, the execution environment must also provide external means for interrupting a primitive net. For example, it can be necessary to react to user input by stopping the current task and starting another task. The execution environment must provide a *Cancel* operation. After calling this operation, the primitive net enters the *Canceling* state, and the output port of a dedicated *cancel* primitive becomes *true*. It is up to the primitive net to decide whether to terminate prematurely or continue running as usual. The primitive net should try to reach a safe state as fast as possible (i.e. no robot is moving, no force is applied to the environment, etc.). However if this is not possible, the primitive net will not be forced to terminate. A second operation, the *abort* operation will cause the selected primitive net to be aborted unconditionally, without any chance of cleaning up. This may leave the system in an undefined and potentially dangerous state and is only designed for debugging purposes (e.g. if an ill-designed primitive net refuses to terminate).

5.3.4. Worst Case Execution Time

It is possible to calculate a Worst-Case Execution Time (WCET) for a primitive net. Each primitive has a WCET for its most complicated operation during *Running* state, and the WCET for the whole primitive net is the sum of all primitives' WCET (plus some overhead for transmitting the values from output to input ports). However, this WCET is not always meaningful, because usually not all primitives will actually perform difficult operations at the same time. In particular with fragments, it is common that only parts of a primitive net are actually running.

If the WCET of the primitive net is smaller than the cycle time, it is obvious that all primitives can be successfully executed once in each cycle. If, however, the WCET is larger than the cycle time, this does not imply that the primitive net cannot be successfully executed. As already stated, not all primitives are active at the same time or need the WCET, thus the average execution time is much lower. The determinism provided by the real-time operating system still guarantees that a primitive net that has been executed properly once will also be executed properly later. Robotics applications usually perform (almost) identical tasks repeatedly, which can be tested after the development. If the task is successful during testing, it will also be done properly later on as long as the environment does not change.

5.3.5. Communication

Primitive nets need a bidirectional communication to the robotics application and also to other running primitive nets. Communicating status values to the robotics application is necessary to provide the user with feedback of the currently running task. The structure of a primitive net is immutable once it has reached state *Ready*. In some cases however it is desirable to influence the behavior during runtime, e.g. to reduce the overall program velocity for testing new applications. Therefore, a communication channel from the robotics application to the primitive net is required.

To facilitate these communication requirements, *communication primitives* are used. One set of primitives allows to feed values into the primitive net and thus acts similar to other sensor primitives. The value received over the communication channel is read during phase one of the primitive net execution cycle and does not change during this cycle, even if new values are received asynchronously. Another set of primitives transmits values received during the net execution cycle to the robotics application during phase 3 of the net execution cycle.

Communication using these primitives is not real-time safe. Hence, a primitive net must always be designed so that the reliability and repeatability is not hampered. Changing the overall velocity for example does not influence a program, as long as all parts of the primitive net are changed at the same time. Since the new velocity is provided using a sensor primitive, the new value will propagate throughout the whole primitive net immediately (just like all other values do).

It is possible to connect sensors to the primitive net using the communication primitives, e.g. if a device driver is available for the operating system under which the robotics application is running. However care must be taken that this connection is not real-time safe, and reactions to sensor events cannot be guaranteed. If such guarantees are required, a real-time safe device driver must be implemented in the RCC.

Besides communication between a primitive net and the robotics application, communication primitives can also be used for inter net communication. These communication primitives can be completely real-time safe, i.e. values written in phase three of the first primitive net will be available during phase one of the second primitive net. Values are only updated during phase one, thus the (asynchronous) change of a communication value during phase two or three will only be noticed in the next execution cycle.

5.4. Examples

The following examples demonstrate the key concepts of the Real-time Primitives Interface. Figure 5.4 shows the notation used for all examples. Primitives are denoted by rectangles

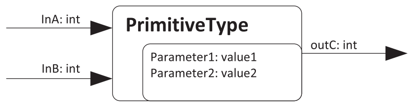


Figure 5.4.: A single primitive with ports and parameters

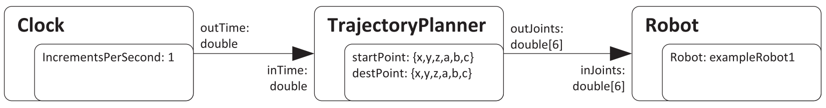


Figure 5.5.: Primitive net showing basic principles of the Real-time Primitives Interface

with rounded borders, showing the type of primitive as text. Links are shown as arrows pointing towards the input port of a primitive. Ports are not explicitly modeled; the name and type of a port is written near the link and the primitive. Input ports are always shown on the left side, and output ports always on the right side of a primitive. The inner rectangle shows parameters for the primitive.

Fragments are depicted using the same structure, however the inner rectangle is omitted. As fragments behave similar to primitives, they can also be modeled like primitives. However, the inner rectangle is omitted as fragments do not have parameters.

The primitives used in the following examples are designed to support the examples and are not necessarily available in the SoftRobot RCC reference implementation. A list of basic primitives available in the SoftRobot RCC can be found in Section 7.3. The Real-time Primitive Interface itself does not mandate any particular primitive; also the granularity of primitives (fine grained primitives with only basic arithmetic operations versus primitives containing a whole trajectory planner) is to be chosen by the implementation. Section 5.5 provides an overview of the granularity chosen for the SoftRobot project.

5.4.1. Basic concepts

Figure 5.5 shows a conceptual, fictive example for a very simple motion, constructed with three primitives. Primitive nets should not hard code the cycle time, because different execution environments may have different cycle times. The *Clock* primitive can be configured for a given number of increments per seconds and always writes the current increment counter (started from activation of the primitive) to its output port. For example, if increments per second is set to 1, the output will be the current run-time of the net in seconds.

The (fictive) *TrajectoryPlanner* primitive takes the current time as input and calculates a set-point in joint space for the given time. The *TrajectoryPlanner* primitive internally has algorithms to calculate the desired motion type (point-to-point, linear, . . .), and in case of path motions also the possibility to calculate the inverse kinematics for the robot, i.e. to transform Cartesian set-points into joint-space set-points.

The resulting array of joint set-points is written to the *Robot* primitive, which uses the set-points for direct hardware control. These set-points may be either directly transferred to the hardware, or used internally e.g. in a closed loop control to calculate velocity set-points in case the hardware does not support position control.

5.4.2. One DOF point-to-point motion

The following example concentrates on a point-to-point motion for a single degree-of-freedom (DOF), e.g. a single robot joint. This type of motion is planned in joint-space and thus it is independent on the type of joint. It can be used for both prismatic as well as revolute joints. The point-to-point motion in this example is divided into three phases. In the first phase, the robot joint is accelerated constantly. Once the desired velocity has been reached, the joint continues moving with constant velocity. At some point, the joint must start decelerating in order to stop at the desired final position. Figure 2.3 on Page 14 shows the velocity/time diagram for such a motion. A major drawback of such a simple motion profile is the immediate switch from full acceleration to no acceleration (during constant velocity), which leads to an infinite jerk and applies a large stress to the mechanics. In real robotics applications, a more complicated motion profile is used, limiting the jerk. However, for this example, demonstrating the basics of the Real-time Primitives Interface, this motion is sufficient.

The example in Section 5.4.1 uses three primitives for motion control, with a high level of logic integrated into each primitive. For higher flexibility, the Robotics API uses a much finer granularity of primitives to generate the desired functionality (cf. Section 5.5). Figure 5.6 demonstrates a more realistic example of a primitive net generated by the Robotics API. Some port names are omitted for better readability. For each phase of the motion, one fragment is used to calculate the trajectory. The *Clock* primitive delivers the current time in seconds since the start of the primitive net to each of the motion fragments and to the *DoubleGreater* primitives, which compare the current time to the times t_A and t_C which denote the end of the acceleration phase and the end of the constant velocity phase, respectively. The *DoubleGreater* primitive emits a *true* signal on its output port if the value on input port *inFirst* is greater than the value of input port *inSecond.* If a comparison with a constant value is desired, one input port can be substituted by a parameter. The output ports are connected with the *inActive* input ports of the three fragments to turn the execution of each fragment on and off at the appropriate times. To enable the *Acceleration* fragment, the result of the first *DoubleGreater* has to be negated.

All motion fragments calculate a set-point for the current time and provide it on their outPosition output port. The details of such a fragment are explained later. The *DoubleConditional* primitives use the Boolean data-flow provided at their *inCond* input to decide whether to forward the data received on the *inA* input (if *inCond* is *false*), or the *inB* input (otherwise). By cascading the *Conditional* primitives, the output of the

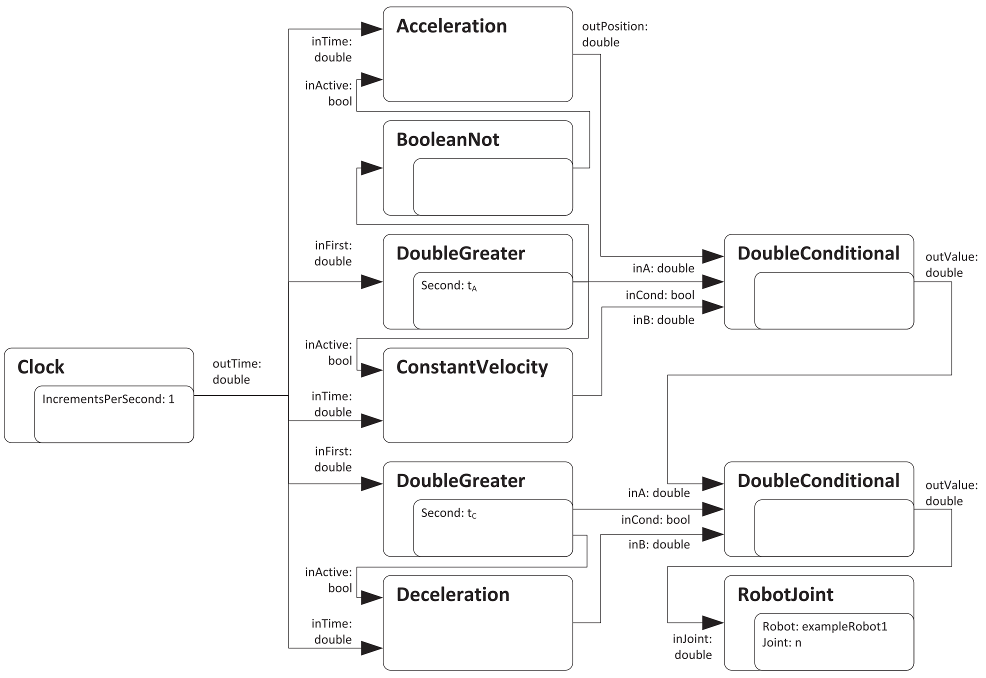


Figure 5.6.: Primitive net showing a real-life point-to-point motion. Three fragments are used.

Acceleration fragment is used until the time t_A has been reached. Between t_A and t_C the output of the *ConstantVelocity* fragment is used. After t_C the output of the *Deceleration* fragment is forwarded to the *RobotJoint* primitive which is responsible for hardware control. The *RobotJoint* primitive takes a value of type double, which represents the joint position in radians.

The *Acceleration* fragment calculates the current position $s(t)$ with the following equation

$$s(t) = \frac{1}{2}at^2 + v_0 t + s_0 \tag{5.1}$$

where t is the current time (w.r.t. to the start of the acceleration phase), a the acceleration, s_0 the start position and v_0 the velocity at start.

Figure 5.7 shows the *Acceleration* fragment in detail. The names of input and output ports are largely omitted for better readability. The fragment has an input port *inTime* and an output port *outPosition*. *DoubleValue* primitives are used to inject constant values (like a, s_0, v_0) into the primitive net. *DoubleAdd* and *DoubleMultiply* primitives are used for adding or multiplying two double values read from their input ports. The time at input port *inTime* will be the current time of the primitive net (t_{cur}). If the *Acceleration* fragment is not the first motion of the primitive net, t_{start} provides the time the acceleration should start so that t used within the *Acceleration* fragment is relative

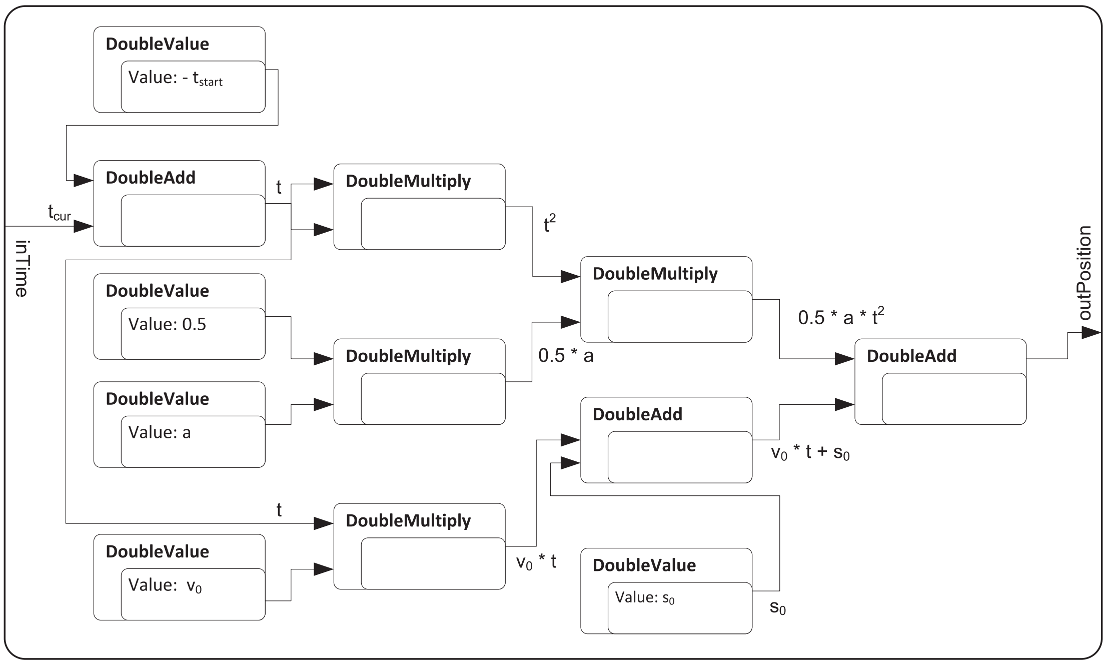


Figure 5.7.: Contents of the Acceleration fragment

to the start of the acceleration. After all calculations have been performed, the result value is written to the *outPosition* output port.

The *Acceleration* fragment already is quite complex, as it consists of 12 rather small primitives. However, primitive nets are not intended to be handcrafted, but rather to be generated by the Robotics API. The combination of primitives can easily be generated by recursively parsing either a mathematical formula, or a Java-code representation of the formula. The *ConstantVelocity* and *Deceleration* fragments are of comparable size.

By embedding the acceleration, constant velocity and deceleration parts into fragments, it is possible to activate and deactivate large parts of the primitive net. At no time more than one of the three fragments will be active.

5.4.3. Point-to-point motion for a six DOF robot

The example in Section 5.4.2 only considers a single joint for the point-to-point motion. This kind of motion planning is already sufficient, for example, for a single linear axis, however most industrial robots consist of six or more joints which need to be synchronized during a motion.

For point-to-point motions several types of synchronization exist (cf. Section 2.3). The Real-time Primitives Interface achieves synchronization of tasks implicitly by its cyclic execution behavior. For an asynchronous point-to-point motion for a six-DOF robot, it is sufficient to insert the fragments and primitives from Fig. 5.6 six times into a single

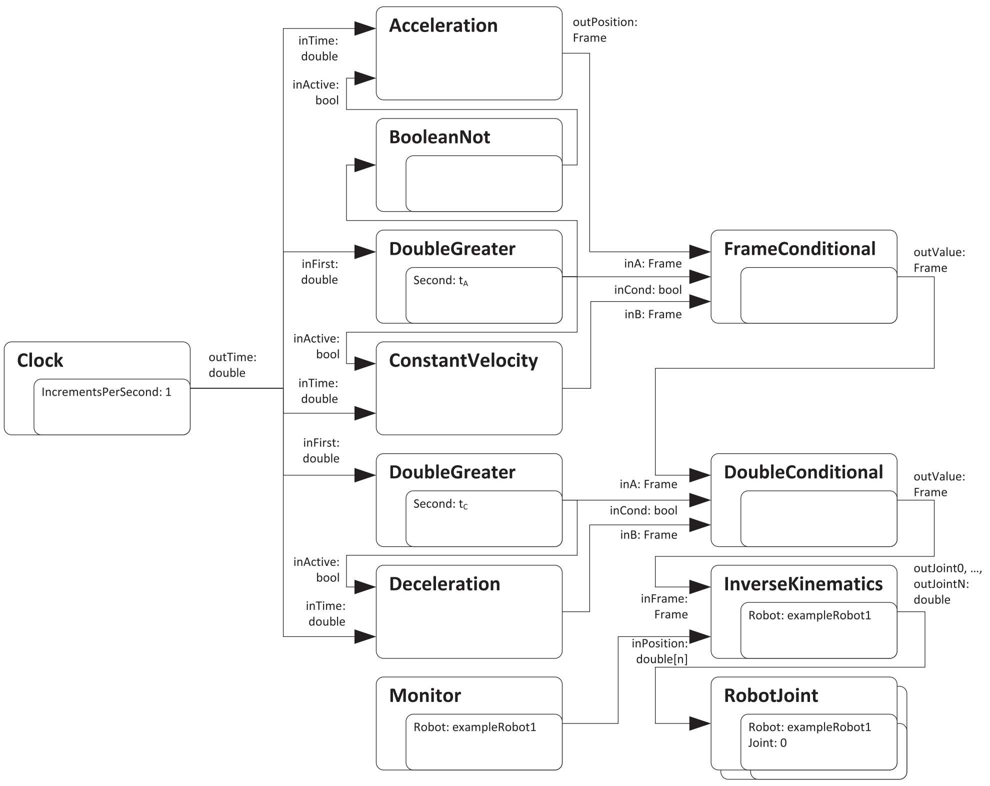


Figure 5.8.: Primitive net showing a linear motion.

primitive net. Because all parts of the primitive net will be run once per execution cycle, all joints will start moving simultaneously. A synchronized or fully synchronized motion can be achieved by providing proper values for a, v_0, t_A and t_C. These values can be calculated completely within the robotics application. This easily allows to generate primitive nets for a synchronized motion with an arbitrary number of joints without the need to modify the real-time execution environment. The use of robot hardware control primitives which only control a single joint (e.g. the *RobotJoint* primitive) also helps for an easy generation of multi-joint primitive nets.

5.4.4. Linear motion

A linear motion is a motion in operation space, i.e. the desired trajectory is specified in Cartesian space in contrast to point-to-point motions, where the trajectory is defined in joint space.

Figure 5.8 shows a primitive net for a linear motion in Cartesian space. At first glance, the primitive net looks similar to the 1-DOF point-to-point motion example in Fig. 5.6. The trajectory of a linear motion also consists of an acceleration phase, a phase with constant velocity and a deceleration phase. The difference is, that for point-to-point motions, all set-points are calculated as joint positions, whereas in this example all set-points are calculated in Cartesian space. Therefore the data-type *Frame* is used. A frame consists of a vector describing the position and another vector describing the rotation in relation to a base coordinate system.

After the three fragments have calculated the trajectory, and the *FrameConditional* primitives have selected the proper values from each fragment, the set-points are transmitted to the *InverseKinematics* primitive. This primitive calculates the joint positions for the given Cartesian position (cf. Section 2.2). Therefore it must know exactly which robot should be used, because the inverse kinematics function is highly depending on the robot hardware. Furthermore, the current joint-position is required, because the inverse kinematics function for a 6-DOF robot yields multiple (usually eight) results for the same Cartesian position. Those results represent different configurations of the robot (e.g. elbow up or down). For a linear motion, it is not possible to switch between two such configurations within a single execution cycle, thus the *InverseKinematics* fragment can choose the configuration closest to the current joint positions. The current positions are read from the *Monitor* primitive, and the resulting joint positions are once again transmitted to the *RobotJoint* primitives (one for each joint).

Linear motions are always calculated for the whole robot, and not for a single joint. A linear motion in Cartesian space can only be accomplished by a synchronized motion of all joints of the robot.

5.4.5. Synchronized robot motions

For some tasks, not only the movement of multiple joints of a single robot needs to be synchronized, but also the motions of multiple robots. The Real-time Primitives Interface has been developed with this use-case in mind. One example could be the handling of a large, heavy workpiece which cannot be lifted by a single robot. Especially with large workpieces, simply employing a larger and stronger robot may be no option, because the workpiece should be supported at multiple points. For such use-cases, multiple synchronized robots can be used. All robots need to move exactly synchronous, such that all their tools keep their position with respect to each other.

Typically, path motions need to be synchronized. The Robotics API solves this task by geometrically coupling all robots. The trajectory for one robot will be explicitly planned (using a primitive net similar to Fig. 5.8), while all other robots will keep their tool at the same relation to the main robot's tool.

Figure 5.9 shows a simplified primitive net for a synchronized linear motion for two robots. The *TrajectoryPlanner* fragment generates the desired trajectory for the main robot (e.g. using the primitive net from the example in Section 5.4.4). The upper *InverseKinematics* primitive converts the Cartesian set-points into joint-space set-points (the technically required *Monitor* primitives have been omitted for a cleaner diagram). For the second robot, the trajectory set-points are converted by the *FrameTransform* primitive which

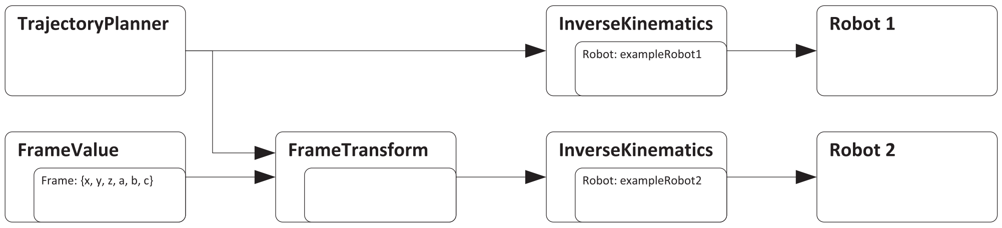


Figure 5.9.: Primitive net showing a synchronized linear motion.

adds the offset for the flange of the second robot to the trajectory. The offset is specified using the *FrameValue* primitive. After the offset has been added, the inverse kinematics function is used as with the first robot. This example assumes that both robots share a common coordinate system and that the *InverseKinematics* primitives know the offset of both robots' bases. In real-life primitive nets generated by the Robotics API, each robot will have its own root coordinate system in which the *InverseKinematics* primitives work. All further transformations necessary for integration in a common world root coordinate system will be calculated by the Robotics API, and appropriate frame transformation primitives will automatically be included.

The synchronization of both robots is implicitly achieved because trajectory planning and execution is combined into a single primitive net and the trajectory planning is geometrically coupled. An alternative solution is to create two independent primitive nets and to start both simultaneously using the synchronization interface (cf. Chapter 6). In this case only the start of both primitive nets is synchronized, but the deterministic execution of the real-time operating system ensures that both nets are executed with the same speed, i.e. both perform the same number of execution cycles in the same time.

5.4.6. Reaction to events

A primitive net needs to notify the execution environment of its desire to terminate by setting a Boolean output port of the root fragment to *true*. After the execution environment has read *true* on this port, no further execution cycle will be started for the primitive net, the current cycle however will be completed with all phases.

The request for termination is only one example for an event that can occur during the execution of a primitive net. For many tasks it is necessary to react to events such as passing a certain path position or after a given time since the start of the motion has passed. Such events are very common for the control of tools like grippers, welding torches, etc. Reaction to these events must occur within a defined time frame, i.e. they must be handled real-time safe. Events are triggered in primitive nets by providing a Boolean output port (which indicates the occurrence of the event) that is connected to the activation input port of the fragment containing the event-handling code. The event handling concept of the Robotics API (cf. Section 10.1.1) builds upon states being represented by Boolean output ports of primitives. Several sensor values can be aggregated and compared to create a single event using appropriate calculation primitives. The event handling fragment will be executed within the same net execution cycle that the event has been detected, thus real-time safe event handling is guaranteed.

5.5. Required primitives and granularity

It is not possible to specify a fixed set of basic primitives which are necessary to solve all robotics related tasks. Special hardware will always require special primitives to make them work. However, it is still desirable to have a small set of basic primitives which can be combined into a large variety of programs. It should seldom be required to develop new primitives.

Using high level primitives such as the *TrajectoryPlanner* in the example in Section 5.4.1 leads to an easy to understand primitive net, however the primitive is hard to reuse. Even if the *TrajectoryPlanner* module was highly configurable, it would still be necessary to develop a new version whenever a new motion type is required.

One aim of the SoftRobot project was to put as much logic as possible into the high level programming language, so that changes to the (C++) Robot Control Core are as minimal as possible. Using primitives with very complex functionality does not help to achieve this aim. The reference implementation of the Robot Control Core created during the SoftRobot project (cf. Chapter 7) uses very basic primitives such as add, multiply or equality for numerical data types (integer and double), and AND or OR for Boolean data types. Besides those primitive data types, also more complex, robotics specific data types are used. For Cartesian positions, the Frame data type is used, which combines six double values (three for the position, and three for the orientation). The basic primitives and data types of the SoftRobot Robot Control Core are described in detail in Section 7.2.

Using such basic primitives, almost all algorithms can be created by a combination of those primitives. A major drawback of this approach however, is that the primitive nets tend to grow very large (thousand primitives for blended motions are not uncommon). Using the concept of fragments (cf. Section 5.2.3), the run-time performance is acceptable, but the generating, loading and parsing of primitive nets still takes significant time. To counter these problems, some very commonly used functions have been implemented as native primitives. For example, for event handling, it is often necessary to detect a raising edge in a Boolean data-flow to trigger the start of an event handler. The detection of a raising edge can be accomplished with several basic primitives including some *Pre* primitives to preserve the state of the last execution cycle. However, this detection is so commonly used that the replacement by a single primitive saves a considerable amount of primitives in a net. Besides the reduced number of primitives, also the time required to execute the single (complex) primitive is slightly smaller than executing multiple primitives.

5.6. Related work

Several research projects have been using general purpose programming languages for industrial robotics applications. Because many projects propose a multi-tiered architecture (cf. Section 4.3) which separates between hardware control and the overall program logic (which often is split up into a planning and a behavior control part), an interface is required which allows the planned tasks to be executed. In the SoftRobot architecture, the Real-time Primitives Interface (RPI) allows a flexible real-time task specification and separates between the real-time safe robot control core, and non real-time safe robotics applications.

The following sections provide an overview of a selected range of related work for the Real-time Primitives interface. At first in Section 5.6.1 some projects which provide an integration of robotics applications in standard general purpose programming languages are presented. Special attention is paid to the integration of real-time aspects in these projects. Section 5.6.2 compares RPI to other projects and systems based on a data-flow language. Section 5.6.3 analyzes the cyclic execution nature of RPI in comparison to other projects, and Section 5.6.4 provides a structural overview of the real-time components in other projects for motion control. Furthermore, a possible integration scenario for the iTaSC project is drafted. Finally, in Section 5.6.5, a summary is provided.

5.6.1. Integration of robotics applications in general purpose programming languages

Hayward et al. [52] started the RCCL (Robot Control C Library) project to provide a robot programming framework using the C language to run on a UNIX operating system. RCCL based programs are split into two layers: a real-time safe trajectory control layer (typically running with 50 Hz frequency) and the non real-time safe planning layer, in which the user's C program is executed. Using library functions, it is possible to specify robot motions which are issued to a motion queue. This queue is shared between both layers using shared memory technology. The motion queue also supports kinematics and inverse kinematics calculations, and motion goals may depend on sensor values, e.g. the motion goal can be a moving conveyor which is followed by the robot until another motion command or a stop command is submitted to the motion queue. Lloyd et al. [84] extended RCCL to support multiple robots and controllers with multiple processors. Synchronization can be performed either by geometrically linking one robot to the other (i.e. one robot is instructed to follow the motions of the other with a fixed geometrical link, e.g. to share the load between both robots), or by delaying the start of further motions until one or more robots have completed their tasks. The SoftRobot architecture also separates between a real-time safe execution layer (the RCC) and a non real-time safe planning layer (the Robotics API); the main difference to RCCL is the design of the interface between both layers. RCCL provides a set of commands that can be issued to the trajectory control layer using the motion queue. These commands describe rather complex, monolithic tasks such as a motion, i.e. the definition of how a motion must be executed is made in the lowest layer. Custom extensions are possible, but need to be implemented low-level in a real-time safe way and included in the RCCL runtime system. RPI in contrast offers a set of very fine grained primitives which allow the specification of how a motion must be executed (and many more commands) within the planning layer, thus offering a much greater flexibility. Extensions to support new types of motions etc. can be purely done within the planning layer, without the need to modify the real-time safe RCC.

The Multi-Robot Research-Oriented Controller (MRROC++) [130, 131] is a C++ framework for robotics applications, supporting the operation of multiple robots simultaneously. It is an object-oriented extension of the previous MRROC project [129]. It has been implemented using the QNX operating system for real-time execution on x86 hardware. The framework defines three main components: *effectors* (robots, tools, etc.), *receptors* (sensors) and the *control subsystem*. The *Master Process* (MP) controls the overall task by communicating with multiple *Effector Control Processes* (ECP, one for each effector). Each ECP is responsible for the execution of the task submitted by the MP for a single effector. If multiple effectors need to be synchronized, the ECP simply forward data received by the MP, which itself is then responsible for synchronizing all ECP. Hardware control is performed by the *Effector Driver Processes* (EDP) and *Virtual Sensor Processes* (VSP). VSP provide (aggregated and processed) sensor data retrieved from hardware sensors, while EDP accept set-points for positions the effector should reach. EDP and VSP are completely task-independent, while ECP and MP are hardware-independent to ensure best reusability of the components. Robot programs are developed by creating new ECP and MP components directly using C++ templates and classes provided by MRROC++. Communication between the different processes is performed using buffers. The concepts of MRROC++ can partly also be found in RPI. Devices such as actuators and sensors can have specific device drivers (cf. Chapter 9, equivalent to EDP and VSP) which expose their functionality using primitives and are completely task-independent. The task itself is dynamically created as primitive nets, and thus the task itself does not need to be implemented real-time safe (unlike MP and ECP), which allows to use languages such as Java or C# which are more comfortable, but inherently not capable of real-time safe program execution.

The Generic Robotic and Motion Control API presented by Burchard et al. [25] is based on the Robot Independent Programming Language (RIPL) [87]. RIPL is a C++ framework that provides an object hierarchy to represent typical robotics entities, and polymorphism is used to create hardware specific implementations for devices, which can then be used in task specific application code. Real-time safe operation is possible if the relevant C++ code parts are executed on a real-time operating system, the examples in [87] use the VxWorks operating system. Distributed systems are also possible, even with real-time safe communication, if appropriate communication channels are available and manually implemented in the control code. The Generic Robotics and Motion Control API uses the object-oriented interface provided by RIPL and allows remote access to objects using CORBA. Distribution on this layer is no longer real-time safe. Using RIPL, all real-time safe code must be manually implemented in C++, in contrast to the automated generation of primitive nets. RPI allows for multiple robots, sensors etc. purely by creating non real-time application code. The real-time safe aggregation of sensor values and distribution of commands to the robots is done based on the primitive net.

5.6.2. Data-flow programming languages

The Real-time Primitives Interface is partially built on Lustre [27, 51], which is a synchronous data-flow programming language. Lustre uses a declarative set of equations to define the system, i.e. variables (e.g. from inputs or other equations) can be combined using arithmetic (e.g. $+$, $-$), relational ($=$, $\geq$) or conditional (`if then else`) operators. Only a few basic types are directly supported by Lustre (Boolean, integer, real and tuples), while more custom data-types can be defined in the host language to which the Lustre program is compiled (e.g. C). Lustre allows the definition of "nodes", which contain equations to describe their output data in relation to the input data, and which can be used transparently in other Lustre equations (like a function call). The execution of Lustre programs is done cyclically, i.e. the values of all variables are determined in each cycle. Equations, where a variable depends on its own value, are only allowed if the `pre` operator is used so that the value of the last cycle is used. A complete set of equations can be transformed into an acyclic data-flow graph (with exemption of cycles containing the `pre` operator). RPI uses the same execution semantics as Lustre, however does not use the equational syntax, but rather a direct representation of the resulting data-flow graph. In order to provide maximum flexibility, RPI does not have any built-in operators, but rather relies on primitives for all operations. Primitives can be compared to nodes in Lustre, and in fact all built-in operators in Lustre (used in infix notation) could also be expressed by calls of nodes (in prefix notation), e.g. `X = A + B` could also be written as `X = ADD(A, B)` with a node `ADD`.

The notion of time differs slightly between Lustre and RPI. While both languages perform a cyclic execution, the basic unit of one step in RPI always refers to a fixed (but user-definable) timespan, while in Lustre a step is not bound to a physical time. Besides the basic clock, Lustre also allows the definition of slower clocks. Steps of these clocks occur simultaneously to the steps of the basic clock, but not necessarily at every step, and can be defined using the `when` operator, depending on a Boolean data-flow which defines whether a step is active or not. This concept can be compared to fragments in RPI, whose activation can also be controlled using a Boolean data-flow.

The commercially available development environment "SCADE Suite" [104] allows the creation of data-flow based programs which are closely related to Lustre or the very similar Esterel language [8]. SCADE is commonly used to create safety related applications, because it is possible to verify the correctness of the designed program, and because certified compilers (e.g. fulfilling DO-178B [33] for aviation software) exist.

Although the Real-time Primitives Interface builds upon classic data-flow languages, it offers some distinct features for robotics applications. A key concept of RPI is the dynamical generation of new primitive nets, i.e. applications are not specified by hand-crafting one large data-flow graph, but rather by the combination of multiple short-lived primitive nets. This allows the control flow to remain in the robotics application, which therefore can flexibly react to events and adjust the generation of all subsequent tasks. These short-lived tasks and in particular the automatic generation of the primitive nets render the results of verification on the level of a single net rather useless. However, the execution on a real-time operating system still provides determinism, i.e. a task executed successfully once will also be successful each further time.

The overall execution semantics of primitive nets is identical to Lustre, however RPI adds some robotics specifics such as the sensor reading and actuator writing phases for each execution cycle. These phases allow for synchronized access to hardware and furthermore allow primitive nets to be interrupted even after an execution cycle has been started (this is required for fast transitions using the synchronization mechanism, cf. Chapter 6). The life-cycle of primitive nets is another robotics specific extension to data-flow graphs. Primitive nets can be parameterized after they have been loaded (and before they are started), which allows to design primitive nets (and the generating process) independent of the actually used hardware. The life-cycle model also allows to cancel primitive nets, which is necessary as a way to stop the execution. Due to the inertia of moving robots, a controlling primitive net may not simply be terminated, but rather there must be means to request a graceful termination to give the primitive the chance to execute operation-specific clean-up logic in order to avoid any dangerous state of the actuator.

5.6.3. Cyclic execution in real-time environments

The cyclic execution semantics of RPI is very closely related to the execution semantics of the IEC-61131-3 [59] languages for programmable logic controllers (PLC), which are widely used in production plants. The norm defines a set of five manufacturer independent languages, although not all manufacturers support all languages and some add proprietary languages. The five defined languages are

- **Instruction List (IL):** Textual, very closely related to assembler programming.
- **Ladder Diagram (LD):** Graphical programming language, comparable to circuit diagrams of relay logic hardware.
- **Structured Text (ST):** Text based programming, resembles Pascal.
- **Sequential Function Chart (SFC):** Graphical programming language consisting of states and transitions. While a state is active, tasks contained in the state (e.g. a further LD diagram or I/O operations) are executed. Transitions between states are guarded by logical conditions. If a transition is taken, the state the transition exits is deactivated and the state the transition enters is activated.

- **Function block diagram (FBD):** Graphical programming language consisting of function blocks (FB), which read data from input ports and write data to output ports. The input ports of FB can be connected to output ports of other FB, however no cycles can be formed.

The structure of function block diagrams closely resembles other data-flow graphs such as a Lustre program or a primitive net. PLC programs can directly manipulate memory locations, which can either represent hardware items (such as input and output ports connected to the PLC) or internal variables. In FBD programs, memory is usually read on the left side of the diagram, processed with several function blocks and the results finally written to memory on the right side. Using internal variables, cycles in the diagram are implicitly possible, as values written to memory can be read in the next cycle. An explicit syntax such as the *pre* operator does not exist. In contrast, RPI does not have an inherent concept of variables or memory, however it is possible to define primitives which act identically.

The PLC main program is executed cyclically, i.e. all graphs are completely evaluated in every cycle, and results written to memory. The cycle times can be configured. Before the start of a new cycle, the calculated results are propagated to all hardware devices, and current values are read from devices and provided to the PLC program. Besides the main program, it is also possible to define programs which are executed on the occurrence of events such as errors, hardware faults or timers which interrupt all lower priority programs. Real multi-threading is usually not supported. The RPI in contrast allows the execution of any number of primitive nets in parallel. If not enough physical CPUs are available, the operating system is responsible for scheduling and interleaving all running primitive nets, but in no case is a running primitive net interrupted because of the start of a new primitive net. Event-based interaction is also possible using the synchronization mechanism (cf. Chapter 6), including a seamless transition from one task to another (e.g. robots can be taken over by successive tasks even in full motion). RPI does not use the implicit memory-location based addressing of devices, but rather uses primitives with an explicit interface for all hardware access, which eases understanding a program.

5.6.4. Real-time component structure

Stewart et al. [116] present an approach to robot controlling using “port based objects” (PBO), which is based on their “Chimera II” real-time operating system [115]. All those objects are independent processes and interact with each other by communicating through input and output ports, which are connected between different PBOs. Examples for such objects are (inverse) kinematics, trajectory generation or interpolation objects, although other granularities are also possible. Each PBO is dynamically reconfigurable at run-time, thus it is possible to adapt the system to changing requirements or even to support dynamically reconfigurable robot systems. RPI primitives also have input and output ports for communication, but differ in some aspects. The execution of primitive nets is strictly synchronous, which is necessary to support the synchronization of multiple

Software pattern	**corresponding SoftRobot component or concept**
Scanner	Control logic included in hardware drivers (cf. Chapter 9), access through sensor and actuator primitives
Actuator	
Controller	
Generator	Flexible and extensible definition using the RPI data-flow graph and basic primitives
Observer	
Heartbeat	Strict periodic execution of primitive nets
Command interpreter	RCC parses and creates commands specified using primitive nets
Execution engine	Net execution and communication parts of the RCC
Reporter	

Table 5.1.: Comparison of software patterns described by Bruyninckx et al. [24] and the SoftRobot architecture

actuators. Therefore only one process is used for each primitive net. Distribution to multiple processors can be achieved by synchronizing primitive nets with others, using the synchronization mechanism (cf. Chapter 6). Dynamic reconfigurability is not necessary in RPI due to the rather short-lived primitive nets and the capability to create new primitive nets with an adapted configuration. This allows the application to react more flexibly because not only some configuration items, but the whole primitive net structure can be changed if necessary. Real-time safe, seamless transitions (i.e. continuous control) are still possible using the synchronization mechanism.

Bruyninckx et al. [24] describe a software pattern for a generic motion control core, and a real-time implementation for the Orocos platform [21]. According to the proposed pattern, several distinct components are required to create such a generic system. *Scanner* components are responsible for retrieving signals from sensors, *actuator* components can command set-points to hardware devices, and *generators* create trajectories. The set-points created by the generators and sensor data are further processed using *observer* components. The results can be further processed by *controller* components which can apply control laws to deliver hardware-specific set-points. A *heartbeat* creates events that trigger the execution of the other components, based on external events or time. Finally, the activation and configuration of all components is handled by an *execution engine*, which is supplied with commands (programs) from a non-real time *command interpreter*. A *reporter* component is responsible for communicating data to the outside, e.g. for displaying status data to the user. All components described in the pattern can also be found in the Real-time Primitives Interface. Table 5.1 provides an overview how the proposed components map to parts of RPI. It should be noted that RPI does not use only monolithic components as the proposed Orocos implementation does; but in particular the generator and observer parts (e.g. the trajectory generator) are created as required by combining multiple primitives. This increases the overall flexibility of the system and allows for easy adaption to new tasks.

To allow for programming applications by the specification of constraints for the robot in either joint or Cartesian space, the integration of the iTaSC [31] approach seems very interesting. The computation for an iTaSC application according to [123] consists of several layers of “functional entities”, with the outer layer providing access to sensors or actuators, the middle layer describes the environment of the robot and other objects, while the innermost layer provides the actual task. Each layer consists of several functional entities (sub-layers are again functional entities) which are connected to each other. The structure of this approach could also be provided using a primitive net. Sensors and actuators are already available as primitives. For all other functional entities, either specific primitives have to be created (e.g. for the constraint solver), or existing primitives used to form the desired function. The “virtual kinematic chain” could be expressed using the existing geometric primitives (frames, frame transformations, etc.) and the world model of the Robotics API plus the automatic mapping of geometric relations to primitive net fragments. The iTaSC project offers a domain specific language [122] which could be employed to parameterize the (rather complex) primitives. Generally, only a single task should be expressed with a single primitive net, switching between tasks can be done by means of the robotics application or using synchronization rules (cf. Chapter 6) if real-time switching is required. Functional entities can communicate bidirectionally in the iTaSC system, while primitives can only have a unidirectional connection to other primitives in a primitive net (otherwise illegal unguarded cycles occur). For many tasks however it seems sufficient to transmit data in one direction without delay (in the direction from sensors to calculation to actuators), and to feedback data with a delay of a single execution cycle (by forming guarded cycles in the primitive net).

5.6.5. Summary

Many single aspects of the Real-time Primitive Interface have already been covered in other (research) projects, however none has (to the author’s knowledge) achieved the same level of flexibility for the specification of robotics tasks. In particular the specific life-cycle of primitive nets and the highly dynamic creation process allows to integrate real-time tasks into non real-time applications while still allowing the application to control the overall program flow.

The fine granularity of primitives provides a great flexibility which allows the specification of new control algorithms etc. most of the time without the need to modify the real-time execution environment. Even, if modifications are necessary (e.g. for the integration of a new hardware driver), the existing fine-grained primitives can be largely reused. Furthermore, not can only a set of predefined motion commands be run, but arbitrarily complex tasks can be created using multiple robot systems and a real-time event mechanism to react to sensor data. The data-flow graph concept is a well-known technology for real-time systems and has been adjusted to meet the requirements of RPI, such as the robotics specific life-cycle and an easy automated generation of primitive nets. Cyclic, synchronous execution is also well known from data-flow based systems, however it is also already predominant in industry automation systems based on programmable logic controllers. It guarantees for a fast propagation of sensor values and a synchronized control of actuators. The synchronization mechanism explained in the next chapter allows for real-time safe transitions between multiple real-time tasks with continuous control of all hardware devices, but without the need of all tasks being created at the same time or from within a real-time safe system.

Chapter 6

Synchronization of multiple real-time primitive nets

In the previous chapter, real-time primitive nets have been introduced. Such a primitive net encapsulates a task which must be executed by a robotics controller with strict timing guarantees. Primitive nets can be seen as atomic tasks, once they have been loaded and started on a Robot Control Core (RCC), their structure cannot be changed anymore, and their behavior is only modifiable using a limited, non real-time communication protocol which can inject values into primitives of a running net (cf. Section 5.3.5).

One of the key requirements for the design of every primitive net is that, once the primitive net voluntarily terminates (i.e. it sets the termination output port to *true*), no actuator may be left in an unsafe state, i.e. moving, applying force etc. With this guarantee, it is possible to split large robotics applications into smaller pieces of real-time critical tasks which can be executed sequentially, and controlled by an application using a non real-time capable programming language (cf. Section 3.6). Although almost all applications can be designed using this mechanism, the performance might not be optimal if the system has to come to a complete stop after each single task. Motion blending (cf. Section 2.5) tries to avoid exactly such unnecessary stops. It is possible to implement motion blending by simply including both (or all subsequent) motions into a single primitive net. However this approach is severely limited as a primitive net has to be constructed completely before it can be executed, thus always only a fixed number of motions can be included. Furthermore, the resulting primitive net grows huge.

Manipulation and assembly tasks are getting more and more important for robotics applications [18]. To challenge the uncertainties present in these tasks, often compliant motions are used, because the contact of two workpieces constrains the motions and thus reduces the uncertainty. Compliant motions however require the actuators to apply a defined force or torque upon a workpiece. Whenever a primitive net terminates itself, it must not leave any actuator in any potentially unsafe condition without control, i.e. no actuator must apply force. Thus all compliant motions would have to be encapsulated into a single primitive net, and reaction to external events can only happen if they are already known during creation of the primitive net.

To accommodate the requirement of being able to seamlessly switch from one primitive net to another one, the concept of *synchronization rules* has been introduced into the real-time primitives interface. This allows a primitive net to terminate in an unsafe condition, if it can be guaranteed that one or more successive primitive nets will take over controlling all devices.

Depending on the problem to solve, different levels of coordination and synchronization complexity is required. For the motion blending use-case with a single robot, it is sufficient to prematurely terminate a running primitive net (the first motion) and subsequently start a new primitive net with real-time guarantees (which performs the motion blending and the second motion).

More complex synchronization can be required for example for compliant motions, especially if multiple actuators are involved. If two robots are working together (controlled by a single primitive net), they can enter a *maintaining* phase after finishing their work in which they may still apply a controlled force. If both robots are needed for separate use afterward, it is necessary that the maintaining primitive net is terminated, and *two* successive primitive nets are started.

A previous version of the synchronization mechanism described in this section was first published in [125, 126].

6.1. Synchronization rules

Synchronization rules are used to exactly specify the conditions for the synchronization of multiple primitive nets. The synchronization happens implicitly by stopping and starting the appropriate primitive nets at the same time. This time is expressed using the *synchronization condition*, a propositional logical formula which depends on a combination of Boolean variables which each primitive net can provide. A primitive net expresses its ability to be taken over by an appropriate successor using these variables. Primitive nets can provide any number of such variables to support different scenarios. If the synchronization condition evaluates to *true*, the synchronization rule is activated. Such a synchronization rule σ can be expressed as a 4-tuple

$$\sigma = (C, \omega, \psi, \alpha) \tag{6.1}$$

with

$\wedge$	**f**	**i**	**t**
f	f	f	f
i	f	i	i
t	f	i	t

(a) and

$\vee$	**f**	**i**	**t**
f	f	i	t
i	i	i	t
t	t	t	t

(b) or

a	$\neg$ **a**
f	t
i	i
t	f

(c) not

Table 6.1.: Truth table for three-value logic with the operators *and* ($\wedge$), *or* ($\vee$) and *not* ($\neg$).

- C: *synchronization condition* which activates the synchronization rule.
- ω: set of primitive nets which must be unconditionally *stopped*.
- ψ: set of primitive nets which must be *canceled*.
- α: set of primitive nets which must be *started* at exactly the same time for synchronization.

The sets ω, ψ and α have to be disjoint. The synchronization condition C uses a three-value logic, i.e. the values *true*, *false* and *indeterminate* are possible. Special primitives are used in the primitive net to provide the named Boolean variables. The *indeterminate* value is required for several reasons. Synchronization conditions can contain variables from primitive nets which have not yet been started (i.e. nothing can be said about the state of these nets) or have already terminated (cf. Section 6.2 for more details about this case). Furthermore, if the primitive providing the named variable is contained in an inactive fragment (cf. Section 5.2.3), no current value is available. All these situations are expressed using the indeterminate value. These variables can be combined in the synchronization condition using logical *and* ($\wedge$), *or* ($\vee$) and *not* ($\neg$) operators. Table 6.1 lists truth tables for these operators applied to three-value logic. Variables from multiple primitive nets can be used in a single condition. Although in many cases the primitive nets whose variables are used are included in ω or ψ, this is not a necessity. For example, a primitive net which monitors for errors may activate a synchronization rule which aborts or cancels primitive nets on the occurrence of a certain error, but remains running to continue monitoring other errors.

In each execution cycle of any given primitive net, all synchronization conditions that contain a variable from this primitive net must be evaluated. If such a condition becomes *true*, the synchronization rule is activated. An activated synchronization rule can be executed if certain preconditions are met:

- All primitive nets contained in α are ready for execution, i.e. they are in state *Ready* or *Scheduled* according to Fig. 5.3 (Page 49).
- All necessary resources (i.e. hardware devices) for starting the nets in α are available (cf. Section 6.2).

If an activated synchronization rule cannot be executed due to unmet preconditions, this is considered an error and the synchronization rule is permanently discarded. At this time, no primitive net has been influenced in any way. All primitive nets in ω will continue running, and nets in ψ have not been canceled. Because no primitive net may rely on being taken over, active hardware control is never lost and thus this specific error cannot lead to a potentially dangerous situation with uncontrolled devices.

When a synchronization rule is executed, all primitive nets in ω are unconditionally terminated and those contained in α are started simultaneously and thus synchronization is achieved. When primitive nets are terminated, it must be ensured that they are not currently in phase three (cf. 5.3.2) where actuators have been partly provided new set points. Prematurely interrupting a running execution cycle is possible during phases one and two since the primitive net has not yet caused any change to the system. Primitive nets contained in ψ are signaled with a cancel event, but no guarantees can be given about when these primitive nets terminate (they may even not terminate at all if they do not support canceling). It is guaranteed that no primitive net contained in ω will be running once a synchronization rule is executed. It should be noted that this guarantee can also be fulfilled if a certain primitive net has never been started or has already terminated.

Synchronization rules where no primitive net in either ω or ψ is running are valid, however at least one primitive net contained in the synchronization condition C must be running in order to trigger a synchronization rule. If all primitive nets providing variables for C terminate, it is sufficient to evaluate the condition one last time. If the condition is not *true* this time, it can never become *true*. After termination, synchronization variables can only change their value to *indeterminate*, which can never cause the condition to become *true* (cf. Table 6.1). Synchronization rules with only terminated primitive nets providing the synchronization condition can be discarded after the final evaluation of C.

6.2. Resources

Most robotics devices can only be actively controlled by a single source, e.g. there cannot be two applications controlling a single joint of an articulated robot simultaneously. To ensure that two primitive nets never try to control the same device simultaneously, the concept of *resources* has been introduced in the Real-time Primitives Interface. Primitives controlling hardware devices need to specify which resources they need. The RCC has to ensure that at no time two primitive nets are executed which access the same resource. Within a single primitive net, multiple primitives can use the same resource, however the primitive implementation must detect and prevent multiple primitives controlling the same device or – if applicable – merge the different control data.

Resources are locked on a per-net base instead of a per-application base. Although locking resources based on applications can be easier, it hinders the development of larger multi-robot systems which are not controlled by a single application. Modern large-scale applications can be created using a service-oriented architecture (SOA) [56, 57] which profits from being able to control robotics devices from a multitude of different applications and services. Having to lock and unlock resources manually would load an additional burden onto the developer.

Whenever a primitive net is started, all resources required by primitives within the net are locked. If any resource cannot be locked, the start of the net is aborted. Once a primitive net terminates, all resources are released. When synchronization rules are used, resources from primitive nets in ω (terminated nets) can be reused by primitive nets contained in α (started nets), if the primitive net's in ω are still running at the time the synchronization rule is executed. Resources from canceled nets (ψ) are highly unlikely to be available because canceling usually takes a couple of execution cycles.

If a primitive net contained in ω has already terminated at the time the synchronization rule is executed (synchronization rules can still become active as long as at least one primitive net contained in the synchronization condition C is still running), it cannot be guaranteed that all previously held resources are still available. The moment a primitive net terminates, all resources are freed unless they are immediately reused by a started primitive net during the execution of a synchronization rule, thus these resources can be used by any primitive net. The reuse of resources is also used to determine the state of the Boolean variables of a primitive net which are used in synchronization rules after it has terminated. As long as no resource that has been in use is acquired by any other primitive net, the last values of the variables are kept alive. As long as no resource is used again, it can be safely assumed that the state of the system is still identical to the state at the time the primitive net terminated. Therefore, the same conditions (e.g. the positions of actuators) can be assumed as if the primitive net is still running. The moment the first resource is acquired by another net, all Boolean variables are set to *indeterminate*. Thus no synchronization rule assuming a certain system state by reading those variables will be executed.

Resources are not kept locked on the termination of a primitive net for any potential successor, thus it can happen that an activated synchronization rule cannot actually be executed due to missing resources. This is mainly caused by erroneous applications or multiple applications running (accidentally) concurrently. Resources cannot be locked "pro-actively" because synchronization rules may be specified even after the affected primitive nets already have terminated.

6.3. Example: motion blending

To demonstrate the concept of synchronization rules, motion blending is demonstrated on the level of primitive nets in this section. The same setup as in Section 2.5 is used, Fig. 6.1 repeats the general setup.

The direct path from point A to E is blocked due to an obstacle. To avoid the obstacle, the robot has to be programmed using an auxiliary point C. If two separate motions are

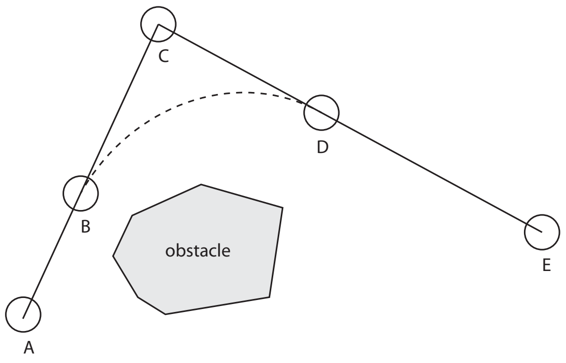


Figure 6.1.: Motion blending (adapted from [125])

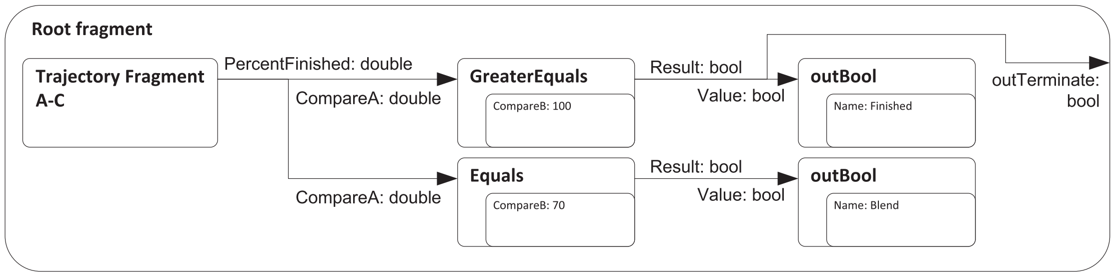


Figure 6.2.: Primitive net for linear motion from point A to C with named Boolean variables for completion of 70% and 100% of the trajectory

used (which result in two independent primitive nets on the RCC), the robot would drive from point A to C and then stop, before it continues to point E. This stop is neither necessary nor desirable. It is acceptable, if the robot leaves the programmed trajectory at some point B and re-enters the trajectory again on point D. As previously described, it is still desirable to split both motion parts into different primitive nets and use a synchronization rule for real-time switching of the motion primitive nets.

A total of three primitive nets is required for this task. The first primitive net P_A contains the motion from point A to point C, while the second primitive net P_B is capable of taking over the moving robot at point B. If for some reasons P_B is not ready for execution when the robot passes point B (e.g. due to the non real-time behavior of the controlling application), P_A will continue to point C as originally specified and stop the robot. A third primitive net P_C will then continue the motion from point C to E (without motion blending)

Figure 6.2 demonstrates the overall architecture of primitive net P_A. A fragment is responsible for planning and executing the whole motion from A to C (an example for such a fragment can be seen in Fig. 5.8 on Page 57). It has an output port which delivers the current percentage of the motion completed (for Cartesian motions, this could also be the distance traveled). This port is connected to two comparison primitives. One compares the grade of completion for equality with 70% and, if the trajectory has reached exactly 70%[1], sets the Boolean variable named “Blend” to *true* using a special primitive connected to the result port. This port is only set to *true* during one execution cycle of the primitive net. If the net has not been taken over, this variable will be set to *false* again.

The percentage finished output port is also compared to 100% using another primitive. Once the whole motion has been completed, the Boolean variable named “Finished” is set to *true*, as well as the *outTerminate* output port of the root fragment. The RCC will then terminate the execution of P_A.

The synchronization rules for P_B and P_C can be specified and transmitted to the RCC any time after P_A has been loaded. For motion blending, the synchronization rule looks as follows:

$$(P_A.\text{Blend}, \{P_A\}, \emptyset, \{P_B\}) \tag{6.2}$$

If the Boolean variable “Blend” is set to *true* in primitive net P_A, this net is terminated and primitive net P_B is started. P_B can rely on the actuator being at point B (within a very small ε that can be ignored due to the inertia of the system) and moving with a known speed, thus it can immediately start issuing a trajectory to blend over to point D and finally move to point E without ever stopping the robot. If the synchronization rule (6.2) is specified after the robot has already passed point B, this rule will never become active. In this case, another synchronization rule is required for continuing the motion after the robot has stopped at point C:

$$(P_A.\text{Finished}, \{P_A\}, \emptyset, \{P_C\}) \tag{6.3}$$

This synchronization rule will start primitive net P_C after P_A has signaled that the motion has finished. This synchronization rule will work well both with being specified while P_A is still running, as well as after P_A has already terminated. In the first case, the synchronization rule will actively terminate P_A, while in the latter case the primitive net will terminate itself using the *outTerminate* output port of the root fragment. Even if some time has passed between P_A terminating itself and the specification of synchronization rule (6.3), the primitive net P_C can still rely on the robot being motionless at point C. If some other primitive net had moved the robot since P_A terminated, the resources protecting the robot would have been transferred to this net and thus the Boolean variable “Finished” would have been set to *indeterminate*.

Motion blending with the synchronization mechanism is performed on a best-effort base. If the robotics application loads the successive primitive net fast enough and specifies the appropriate synchronization rule, motion blending will be performed. If, for some reason, the application does not finish these tasks before point B is traversed, motion blending

[1]Technically this equality check needs to allow a small range ε around 70% such that this event is not missed due to the discrete, cyclic execution of the primitive net

will be ignored. Because the application is not real-time safe, this may occasionally happen. Under no circumstances however will the actuator be out of active control.

6.4. Example: maintaining force

A second example for the use of synchronization rules is the peg-in-hole problem. For this problem, the workpiece, a cylindrical peg, has to be inserted into a round hole. Since the position of the hole is not known precisely enough, the robot cannot simply move to the right position and insert the peg. A common solution is to use compliant motions, i.e. to move the workpiece repeatedly until it comes into contact with the environment. To avoid damage, the contact force must be controlled and thus limited to acceptable values.

Once contact has been established, the measured forces have to be evaluated to decide on the next step to take. It would be possible to encode the whole peg-in-hole problem into a single primitive net, however this has a major drawback. Every time the workpiece comes into contact with the environment, several reactions are possible depending on the measured forces and torques. In a single primitive net, reaction to all cases must already be included, and reactions that follow the chosen reaction etc. until the peg has been inserted. This leads to a giant primitive net, and the main application meanwhile does not control the program flow.

Using synchronization rules, it is possible to create a primitive net which only performs two tasks: Move the workpiece into the desired position and, after contact, maintain a specified contact force. It should be noted that this primitive net must not terminate once contact has been established, since this would leave the actuator out of active control while it applies force to the environment. As soon as contact has been made, a new primitive net containing the reaction can be created by the application and transmitted to the RCC. The new primitive net must be able to take over the robot at the current position while applying force. Using the synchronization rule it is guaranteed that the transition from the first to the second primitive net will be without any interruption. If there is an estimation (e.g. using heuristics) which reaction will be required, the second primitive net can also be created and loaded on the RCC before contact is made. This saves the time required to start the reaction if the estimation was right. If it turns out that in fact another reaction is required, the previously specified synchronization rule simply is not activated, and a third primitive net can be loaded to handle the new situation.

6.5. Related work

The current commercially available robot controllers allow switching from one motion command to the next either by interpreting the code ahead of time (e.g. KUKA KRL, cf. Section 3.5.1) or filling a motion queue (e.g. Stäubli VAL3, cf. Section 3.5.2). Both approaches provide deterministic execution for command switching based on the fact that the overall program is run real-time safe, however also limits the power of the programming languages (since all commands must be executable real-time safely). Synchronization rules for primitive nets are also executed deterministically, however since the control application is not real-time safe, it is possible that the synchronization rule itself is not available deterministically. Thus switching from one motion command to the next is done on a best-effort base, however still with the guarantee that no actuator will be left out of active control. Synchronization rules offer the advantage of achieving almost the same determinism without the limitations of the proprietary, real-time safe programming languages.

Software for embedded systems is often decomposed into several periodic tasks and conditions to switch between the tasks. The Giotto language introduced by Henzinger et al. [53] calls a set of periodic *tasks* that are active *modes* and defines *mode switches* for changing the set of active tasks. A mode has input and output ports for communication with other modes. A mode switch is triggered if the so-called *exit-condition*, a logical condition defined on mode ports, becomes *true*. Giotto guarantees that mode switches will occur real-time safe. Mechatronic UML [6] defines a process for model-driven design of mechatronic applications. One key concept is the *Real-Time Statechart* which is an extension of UPPAAL timed automata [7]. States represent situations in which a system can be, and transitions allow to switch between states. Before transitions can take place, *guards* (described as logical conditions) must become true. This common concept of guards has been adopted for synchronization rules for primitive nets. Giotto and Mechatronic UML both allow to switch back and forth between modes or states respectively. Although synchronization rules build upon guards, another approach to task specification is used. Both Giotto and Mechatronic UML are intended for an "offline" system design, i.e. the application is designed by the full specification of all states, transitions etc. and later executed. The Real-Time Primitives Interface in contrast has been designed for real-time tasks to be generated ad-hoc, and the overall program flow control must remain with the main application all the time. Therefore it is not necessary to allow switching back and forth using synchronization rules, but rather new rules and nets can be added as required. This allows the application to react flexibly and – to a certain degree – even to unforeseen events.

Finkemeyer et al. [41] introduce the *adaptive selection matrix* for robotics applications which allows to switch instantaneously between different open and closed loop controllers within *manipulation primitives* [43]. The adaptive selection matrix allows to independently switch controllers for each degree-of-freedom (DOF) of the system (usually six Cartesian DOF) depending on current sensor readings to form a *hybrid move*. This allows, for example, to use one distance-based controller for a robot to approach a work-piece and immediately switch to a compliant motion upon contact with the work-piece. Switching control mode must occur immediately (i.e. within one control cycle) in order not to exceed the maximum force upon contact. Switching to different control modes upon sensor events can be achieved with primitive nets using synchronization rules. Section 6.4 explains an example for a hybrid move. If different controllers are required for the different degrees-of-freedom, one primitive net for each DOF can be used. The results calculated in each independent primitive net can be transmitted to a further primitive using inter-net communication primitives (cf. Section 5.3.5). This net can further process the data and control the actuator on joint level. Synchronization rules provide the advantage that arbitrary primitive nets can be used and not only a predefined set of possible control laws. This also allows to re-use existing implementations and also to integrate hybrid moves seamlessly into other tasks which are necessary in a manipulation scenario such as standard transfer motions.

The ORCCAD architecture introduced by Borrelly et al. [16] partitions robotics applications into two separate concepts: *robot tasks* and *robot procedures*. A robot task contains a control law for an actuator which must be evaluated continuously, e.g. the control of a robot motion. Robot tasks can be parametrized, but their goals cannot be changed during run-time. Before a robot task can be started, specified *pre-conditions* need to be met, and the task is stopped once the specified *post-conditions* are met[2]. During the execution of a robot task, *events* can be received or emitted. The overall program logic is specified using robot procedures which provide a sequence of robot tasks to execute, and information how to handle events raised in the robot tasks. Events can be handled e.g. by stopping one robot task and starting a new one; thus event-based transitions between robot tasks are possible. Robot procedures are programmed using the MAESTRO language [29]. ORCCAD provides complete tooling for the development of applications and can automatically generate Esterel code for robot tasks and robot procedures which can be used to formally verify properties of the system (e.g. only one robot task controls an actuator at every single point in time). The whole program is compiled into C/C++ code and executed on a real-time operating system. The synchronization rules introduced in this chapter also allow to create event-based robotics applications by switching between different sets of primitive nets for real-time control. Synchronization rules however can be created on demand and in particular the generation of synchronization rules does not need to be done on a real-time operating system. Using a high-level programming language such as Java offers much more possibilities in comparison to a rather limited domain-specific language such as MAESTRO. Several aspects which can be (manually) verified in ORCCAD are automatically checked by the runtime environment in the SoftRobot architecture, e.g. the resource concept (cf. Section 6.2) automatically prevents multiple primitive nets from accessing the same actuator due to a misspecified synchronization rule. Although these checks are done at run-time, it is still guaranteed that all actuators remain under control, because such errors are detected before any modification is made to the system. The deterministic execution of primitive nets guarantees high repeatability, therefore testing applications whether they perform the desired task properly yields a high degree of reliability even without verification.

The Task Control Architecture (TCA) introduced by Simmons [111] is a message based framework for robotics applications that decomposes large applications into multiple

[2]The definition of *post-conditions* by Borrelly et al. does not match the typical definition used in computer science but rather describes the condition which triggers the termination of the robot task.

smaller tasks which are arranged in a task tree. A task can be further decomposed by adding children to a task node in the tree. Children on the same level of the tree represent tasks that may be run either in parallel or sequential, if additional constraints are specified. Even if two tasks are marked as sequential, the second task will already be prepared while the first one is executed, however the planning can also be delayed using a further constraint. Special monitor tasks can be used to detect failed tasks. If such a task failure is detected, control can be switched to another task to react to the new situation, however planning of this task is delayed. Like with RPI, resources are used to prevent multiple tasks from being executed in parallel which use the same hardware device. The Task Description Language (TDL) [110] builds upon these mechanisms and provides a C++ extension for the specification of the tasks themselves, and also for the constraints leading to the proper sequence of tasks. Primitive nets and synchronization rules do not use any hierarchical structure to describe the sequence of tasks. In fact synchronization rules are only intended to enable the real-time transition from one set of tasks to another. The logic to decide upon such transitions remains within the high-level robotics application and can be expressed by means of the programming language. Constructs of standard programming languages allow for much more flexibility in the sequencing of tasks than a strictly hierarchical tree notation does. A specific fault handling is neither required on the level of primitive nets nor on the level of synchronization rules. Failure handling can be performed within the robotics application (cf. [1, Section 6.6]) and is automatically mapped to the appropriate primitives, links and synchronization rules.

6.6. Conclusion

The Primitive nets introduced in Chapter 5 provide a flexible and extensible system to execute real-time safe tasks and are based on well-known data-flow language concepts. Using fine-grained primitives, arbitrary tasks ranging from implementing control laws to trajectory planning or synchronizing tool actions can be specified. Extensions such as fragments allow for performance optimization while the specific life-cycle caters for the specifics of robotics applications.

Encoding large tasks within a single data-flow based program however also has some drawbacks. Since the program must be completely specified prior to starting it in order to fulfill all real-time requirements, the main application cannot remain in control of the program flow. Long sequences of tasks (e.g. several motions blended into each other) do not always need hard real-time, i.e. a short break between two tasks is tolerable if it can be ensured that no dangerous situations arise (i.e. actuators out of active control while moving or applying force to the environment). Synchronization rules allow multiple independent tasks (i.e. primitive nets) to execute sequentially including handing over actuators in (potentially) dangerous situations. Using synchronization rules, control will only be handed over if it can be guaranteed that the successors are ready for taking over, otherwise a fallback strategy continuing control will be applied. This allows the overall program control to remain within the non real-time application. Usually all successive primitive nets will be created and loaded in time, allowing for a smooth and continuous program execution. If, however, for any reason the application is not able to provide the successive tasks fast enough, the system will either be brought to a safe condition, or control is maintained until the application catches up and supplies appropriate successive commands.

Chapter 7

The Robot Control Core: an execution environment for real-time primitive nets

The SoftRobot RCC is the reference implementation of the Robot Control Core (RCC) layer and was developed during the SoftRobot project. It implements the Real-time Primitives Interface (RPI) and thus is able to execute programs which have been written using the Robotics API. The SoftRobot RCC contains a set of basic primitives which are sufficient for a large number of robotics applications, in particular in the industrial robotics domain. The SoftRobot RCC also provides device drivers for a variety of robotics hardware which has been available at the University of Augsburg. The SoftRobot RCC is designed to be flexible and easy to extend, thus it is possible to add further primitives if required, as well as creating hardware device drivers for new hardware support.

The SoftRobot RCC is intended to run using a real-time operating system to provide reliable and constant execution times. Real-time capabilities are achieved on the Linux operating system, using the hard real-time extensions RTAI [10, 34] or Xenomai [48]. Furthermore, it is also possible to run the SoftRobot RCC with a standard operating system for development or simulation purposes, when no hard real-time for hardware control is required. A (non real-time) port to the popular Microsoft Windows operating system is also available.

To conform with hard real-time requirements, the system has been developed using the C++ programming language [117]. This language is commonly used for real-time applications and provides direct access to the computer hardware, in particular to the memory. Native C++ does not provide any automatism for memory management, thus the developer has to take care of all memory allocations, usages and de-allocations, which enables the developer to guarantee that certain parts of a program will never perform any non real-time conform operation (such as memory management).

Besides the necessity to use a rather low-level hardware centric programming language for the real-time critical parts, the SoftRobot project is aimed at placing as much logic as possible into the higher levels of the architecture (cf. Section 4.1). Using this strategy, the benefits from modern object-oriented programming languages can be used optimally, and errors related to memory management can be reduced as much as possible without losing real-time capabilities.

7.1. Software architecture

The SoftRobot RCC consists of several independent components, which are linked by the central *Registry* component. Components of the SoftRobot RCC are for example:

- Primitive net execution environment
- (Hardware specific) primitives
- Communication infrastructure
- Hardware device drivers

Figure 7.1 shows an overview of the SoftRobot RCC architecture. Some basic functionality is implemented directly in the RCC Core package, while most device specific functionality is provided by run-time loadable extension libraries.

Robotics applications communicate with the SoftRobot RCC using the HTTP protocol [40]. Therefore, the RCC contains a built-in web server component. The web server provides standard web pages which can be viewed using a common web browser (for more details see Section 7.6.2) and a special communication protocol for robotics applications, DirectIO (see also Section 7.6.3), which uses the WebSocket technology [39].

The web server communicates with the central *Registry* component which manages all available primitives, devices and the currently active primitive nets. The Registry serves as a facade (cf. facade pattern [45]) for the components communicating with robotics applications. Using the Registry, it is possible to load new primitive nets, start those nets and monitor their execution. It is also possible to load extension libraries and to start new device drivers. During initial start-up, the Registry is responsible for loading all necessary extension libraries and the initial creation of all necessary devices which have been specified using configuration files.

The RCC Core implementation does not contain any device specific implementation, and also only generic primitives are contained. Run-time loadable extension libraries are used for all device specific extensions. Using extension libraries, it is only necessary to load the devices and primitives which are necessary for a given application. Furthermore, new

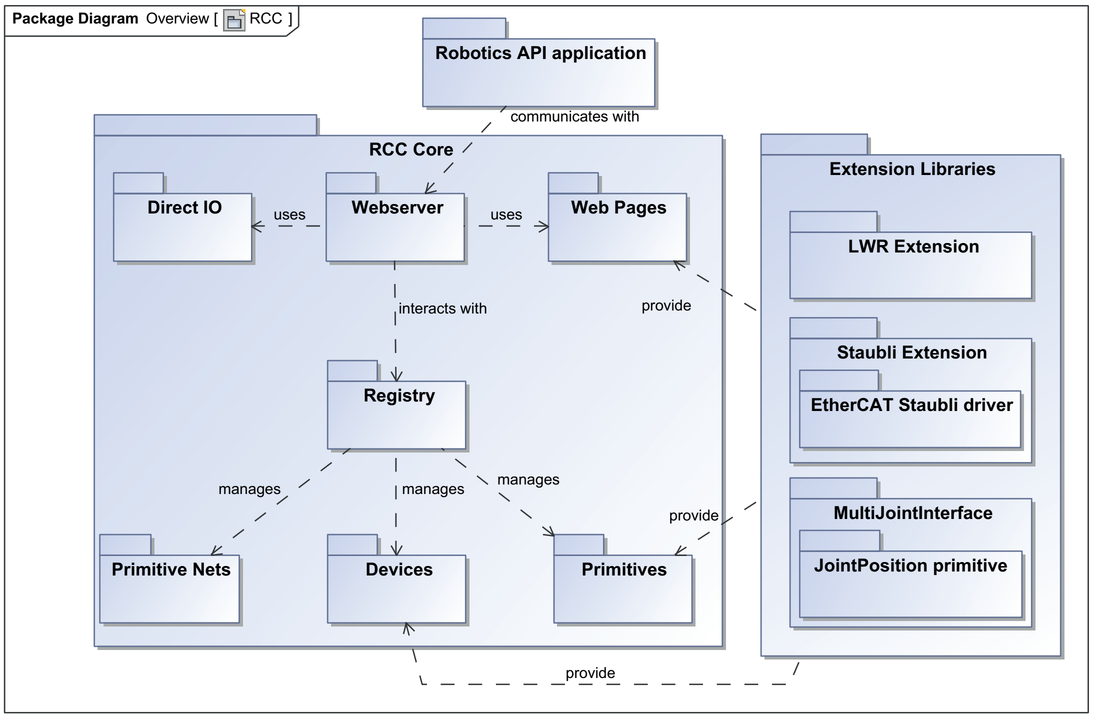


Figure 7.1.: Architecture overview

hardware can be integrated more easily and no changes have to be made to the core system.

Extension libraries can provide implementations for primitives, devices and also for custom web pages which are delivered using the built-in web server. When providing new devices, the extension library often also contains hardware specific drivers for communication with the hardware device. However, it is also possible that several devices are controlled by the same driver implementation, e.g. all hardware devices using the same fieldbus can use the same driver for accessing this fieldbus.

7.2. Primitives implementation

Figure 7.2 shows the relevant classes for the implementation of primitives. A primitive is derived from the abstract class *Primitive*. The template class *Parameter* is used for static parameters. RPI ports are created using the template classes *InputPort* and *OutputPort*. The template parameter must be bound to a valid data type, which can either be a primitive data type (such as `int`, `double`) or a custom, complex data type, while the latter option has some restrictions to comply with the real-time requirements of the system (e.g. arrays, cf. Section 7.3.3).

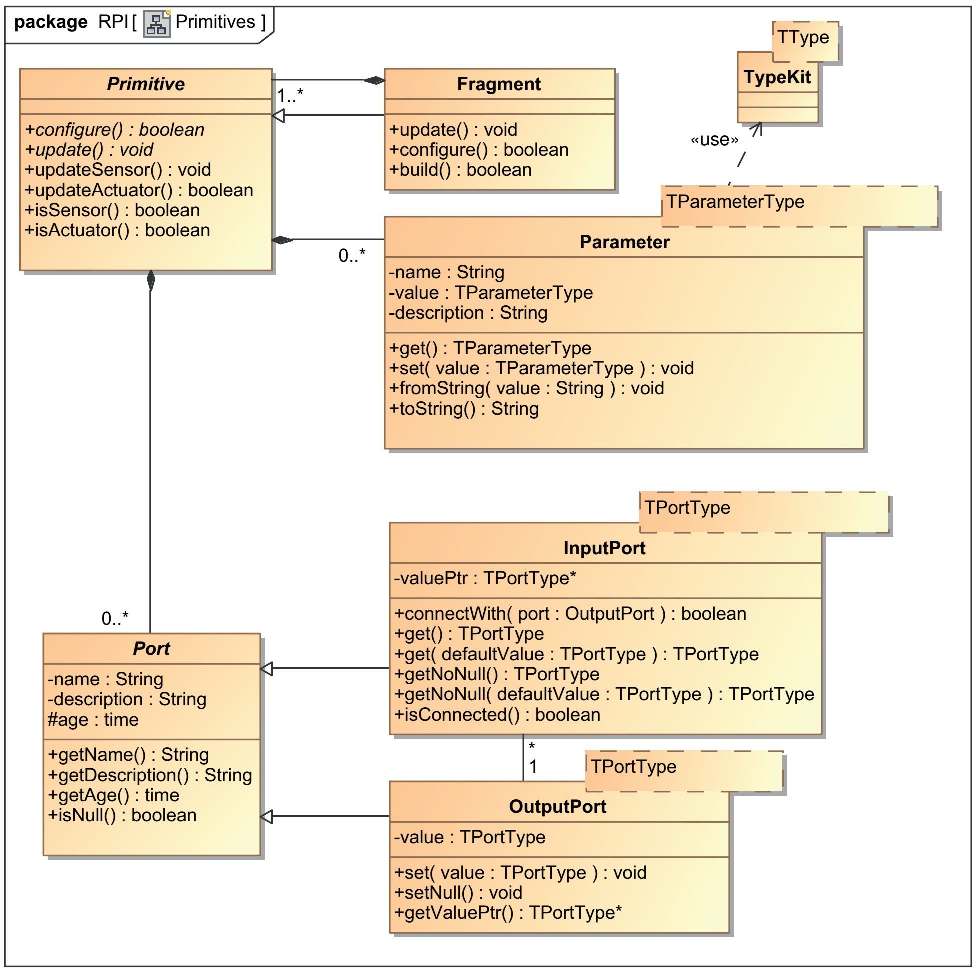


Figure 7.2.: UML class diagram of primitives

Class Primitive

All primitives must provide a constructor to create an instance of the primitive. Within this constructor, all parameters and ports must be registered using their names to provide a name-based access to ports and parameters. This is required since the instantiation of a primitive net is done based on a textual representation transmitted to the Robot Control Core over a network connection (cf. Section 7.6). During the run of the constructor, neither parameters nor ports may be accessed to retrieve data.

The abstract method `configure()` must be implemented by all primitives. This method will be called during the loading phase of a primitive net. All parameters will already be available so that the primitive can use the parameter values specified by the robotics application (e.g. the name of the robot which should be controlled). Data from input ports is not yet available, and output ports may not be written to. This method is called before the real-time execution of the primitive net starts, i.e. all tasks which could break real-time (in particular memory allocation) must be performed in this method or in the primitive's constructor. The primitive may return *false* in this method if any initialization process fails. In this case, the initialization of the whole primitive net is aborted, and the net enters the state *Rejected*.

The method `update()` is called cyclically during the real-time execution of a primitive net. The implementation must be done with real-time aspects in mind, i.e. no operations which may break hard real-time may be performed. This includes any kind of memory allocation, disk I/O or calls to standard operations system functions ("syscalls"). Parameters and ports can be fully accessed in the `update()` method. Data read from input ports is guaranteed to be current, because the `update()` method for all other primitive instances connected to input ports will already have been completed. Every primitive should write values to all output ports in the `update()` method. The SoftRobot RCC implementation automatically sets an output port to *null* if no value has been written within the `update()` method.

Primitives responsible for sensors and actuators are treated in a special way, since sensors are always read in phase 1 of the primitive net execution cycle, and actuators must be provided with new set-points in phase 3 (cf. Section 5.3.2). A primitive must overwrite the `isSensor()` method to return *true* to mark it as being responsible for a sensor. Its `updateSensor()` method will then be called immediately after the start of an execution cycle, so that all sensor primitives will be called approximately at the same time.

Primitives controlling actuators likewise must overwrite the `isActuator()` method to return *true*. Their `updateActuator()` method will be called after all primitives' `update()` methods have been called. Actuator control should be performed in this method. Besides providing all actuators with new set-point values approximately at the same time, this also provides better support for error handling. If a problem occurs during the execution of the primitive net which requires the net to be aborted, either none or all actuators will have been provided with new values.

Primitives should not access ports in the `updateSensor()` or `updateActuator()` method. For sensors, no valid data will be available at input ports because `updateSensor()` is called before any `update()` method is called. Analogously, data written to output ports in actuators will never be read by the primitive net. Sensor/Actuator primitives may however also use the `update()` method to access ports, which will be called just with like any other primitive.

Class Parameter

The class *Parameter* provides access to static parameter values for primitives, which do not change during the life of a primitive net. The class is provided as template and must be bound to a specific data type. This can be e.g. a primitive data type such as `int` or `double`, but also a complex custom data type (e.g. `KDL::Frame` for representation of a coordinate system in 3D space). Each parameter in a primitive has a unique name. The value of the parameter can be set and retrieved directly using the `set(...)` and `get()` methods. Parameters also can carry a human readable description text which is used in automatically generated web pages to provide some documentation of the interface of a primitive.

The value of a parameter will be specified together with the overall structure of the primitive net in a textual representation. The class Parameter uses a helper class *TypeKit* to convert from a string to a concrete data type and vice versa. An appropriate type kit must be registered for every data type which is used in a parameter. TypeKits for `int`, `bool` and `double` are available with the SoftRobot RCC core library.

Class Port

The class *Port* with its two subclasses *InputPort* and *OutputPort* is responsible for the real-time communication between two primitives. Just like parameters, ports are also named and can carry a descriptive text. Furthermore, ports carry an *age* attribute to store the count of the execution cycle when the port has been written to the last time. Using this attribute, it is possible to detect whether the data provided is current or outdated. Ports are also typed and must be bound to a concrete data type.

A new value should be set to an *OutputPort* in each execution cycle of the primitive net in the `update()` method of the primitive. The method `set(value)` updates the internal value of the output port and also updates the age of the port to represent the current execution cycle. Because primitive data types in C++ do not support an explicit *null* value, there is also the method `setNull()` which forces an output port to become *null*.

Input ports can be read using several methods, which differ in their handling of *null* values. All get methods will return the current value of an input port, if it is not *null*. Because primitive data types in C++ cannot represent a *null* value, all methods will always return a valid value. The primitive implementation must explicitly call the method `isNull()` to check whether an input port has valid data or not. If an input port is *null*, the `get(...)` methods will either return the default value of the data type or the supplied default value. The `getNoNull(...)` method will return the last value that has been set to the port, or the default value of the data type if the connected output port has never been set. An *InputPort* does not need to be connected to any output port. The use of the `get(...)` methods on unconnected input ports behaves as if the post would carry a *null* value. The `getNoNull(...)` function will always return the default value in this case.

During the loading phase of a primitive net, an input port is connected to an output port using its `connectWith(...)` method. Output ports are not connected, only input ports are connected to output ports. It is possible to check whether an input port has been connected with its `isConnected()` method. This can be used by a primitive in its `configure()` method to fail configuration if a required input port is not connected. The output port creates an instance of the specified data type. Therefore, the given data type must provide a publicly accessible, parameter-less constructor. If a complex data type requires memory allocation, instances must perform allocation in their constructor, or the primitive must perform all necessary initialization tasks during its `configure()` method. For performance reasons, data is only written to the output port, while all input ports only hold a pointer to the data stored in the output port, so that no data is unnecessarily copied. However, input ports may never change the data they receive from an output port, as there may be other input ports connected which still require the original value. The use of pointers for linking input and output ports is also the reason why there is no dedicated class required for the RPI concept of links.

When a new simple value for an output port is set, the value is copied into the memory that has been previously allocated for the output port and to which the connected input ports point. Complex data-types must provide an appropriate copy-constructor and assignment operator to prevent deep copies of the content when assigning the results of the `get(...)` methods of an input port to a local variable. For more details please refer to Section 7.3.3 which describes the required mechanisms exemplary for arrays.

Class Fragment

Fragments are a special variant of primitives which can themselves contain primitive nets. Hence, the class *Fragment* is derived from the class *Primitive*. The `update()` and `configure()` methods of fragments are implemented to recursively call the corresponding methods for all primitives contained in the fragment. The `update()` methods of the contained primitives are called according to the topological order of the primitives. Failures in the configuration of primitives result in failure of the configuration of the fragment. The construction of a fragment is done by calling the method `build()`. Every fragment carries a named input port *inActive* for activation. Each time the `update()` method of the fragment is called, the value of this port is evaluated. Only if it is *true*, the `update()` methods of the contained primitives are called. Besides this named input port, a fragment can have additional input and output ports which are connected to primitives inside the fragment. More details about creating primitive nets and fragments can be found in Section 7.4.

7.3. Basic primitives provided by the SoftRobot RCC

The SoftRobot RCC provides a set of basic primitives which are required for running applications. Only basic primitives are embedded into the SoftRobot RCC; many primitives are provided by additional extension libraries, which can be added and removed at run-time. Some of these extensions are hardware specific, while others serve general purpose functions. More details about the extension loading mechanism can be found in Section 9.5.

The following sections describe the most important basic primitives, beginning with core functionality to control the execution of primitive nets and continuing with basic functionality to create data-flow graphs. The set of primitives presented here has proven to be sufficient for most basic algorithms, however device specific primitives are additionally required for hardware control. Section 9.2 introduces some further basic primitives for hardware interaction.

7.3.1. Core functions

Only one primitive is required for controlling the execution of primitive nets.

Cancel

Input ports	none
Output ports	outCancel: Boolean
Parameters	none

The *Cancel* primitive emits a *false* value on its single output port by default. If the execution environment requests the primitive net to cancel (cf. Section 5.3.3), a *true* value is emitted. Multiple instances of this primitive may be used within a single primitive net, which will always have an identical value on their output port.

7.3.2. Primitives for general data-flow

The following primitives are used to create the general data-flow graph, i.e. to inject constant values, check for *null* or store and access a history of previous values. All primitives are available for different data types. The general primitives are available for Boolean, Integer and Double values. To facilitate the development, C++ templates have been used. In the following descriptions, the letter “T” has been used to specify the generic data type. For example, the *Core::TValue* primitive is available at run-time as *Core::BooleanValue*, *Core::IntValue* and *Core::DoubleValue*.

Core::TValue

Input ports	none
Output ports	outValue: T
Parameters	value: T

The *TValue* primitives are intended for injecting constant values into a primitive net. Using this primitive, it is possible to supply a constant value to any other primitive with matching port type.

Core::TIsNull

Input ports	inValue: T
Output ports	outValue: Boolean
Parameters	none

The *TIsNull* primitives check whether the input port *inValue* received a valid value, or is *null*. The input can be *null* either because the connected primitive explicitly specified a *null* value, or because the primitive did not provide any data at all (e.g. because the primitive is contained in a deactivated fragment). If a *null* value is detected, the output port *outValue* is *t*rue, otherwise it is *false*.

Core::TSetNull

Input ports	inValue: T inNull: Boolean
Output ports	outValue: T
Parameters	none

The *TSetNull* primitives allow to inject a *null* value into a data-flow. If the input port *inNull* is *false*, the data from input port *inValue* is directly forwarded to *outValue*. *Null* values are also forwarded. If *inNull* is *true*, a *null* value is set on *outValue* independent of *inValue*.

Core::TAtTime

Input ports	inValue: T inAge: double
Output ports	outValue: T
Parameters	age: double maxAge: double

The *TAtTime* primitive provides a history of a data-flow in the primitive net. Each execution cycle, the current value present at *inValue* input port is saved. Using the *inAge* input port or the *age* parameter, it is possible to select a time in history for which the saved value should be written to the *outValue* output port. The *age* parameter is only used when the input port *inAge* is not connected.

The *maxAge* property allows to specify how long values should be archived. Because the primitive is used under real-time constraints and memory can only be allocated at startup, it is not possible to change the length of the history once the primitive has been instantiated.

The age and maximum age is specified in seconds. The primitive automatically calculates the right number of execution steps for the given times. The time of an execution cycle is considered from the ideal time of its start of execution (included) to the start of the next execution cycle (excluded). Times are considered ideal, i.e. jitter from the real-time operating system is not taken into account.

If a *null* value is read in *inValue*, it will be replicated to *outValue* when the appropriate time is requested. If age specifies a time before the start of the primitive net, also a *null* value is written to *outValue*.

Core::TSnapshot

Input ports	inValue: T inSnapshot: Boolean
Output ports	outValue: T
Parameters	value: T

The *TSnapshot* primitives allows to freeze the value in a data-flow at a certain time. When the input port *inSnapshot* is true, the value read from *inValue* is saved and written to *outValue*. The value is preserved as long as *inSnapshot* is not set to true again (i.e. a raising edge is detected). The value from property *value* is used as an initial value, if no snapshot has been taken so far. *Null* values are preserved, i.e. if there is a *null* value at the time of the snapshot, the *null* value will also be written to the output port.

Core::TPre

Input ports	inValue: T
Output ports	outValue: T
Parameters	none

The *TPre* primitives delay the data-flow for exactly one cycle. Every cycle in a primitive net must contain at least one *TPre* primitive, otherwise the primitive net cannot be executed (cf. Section 5.2.2). During the first execution cycle, the output port *outValue* provides a *null* value, in all subsequent execution cycles the value provided to *inValue* during the last cycle is available, including potential *null* values.

Core::TConditional

Input ports	inTrue: T inFalse: T inCondition: Boolean
Output ports	outValue: T
Parameters	true: T false: T

The *TConditional* primitive selects the value of *inTrue* or *inFalse* and writes it to *outValue* depending on the value of *inCondition*. *Null* values are preserved. The parameters *true* and *false* are used if the *inTrue* or *inFalse* input ports are unconnected. This serves as a shortcut if a static value is required, and thus saves the use of a *TValue* primitive.

7.3.3. Array type data-flows

The Real-time Primitives Interface does not only support primitive data types such as Boolean or integer, but also more complex data types. RPI provides a set of primitives to handle arrays of primitive data types, which are commonly used e.g. for transmitting values for all joints of an articulated arm together.

Array data-flows can be produced and consumed by custom primitives (e.g. device specific primitives), or by using the following basic primitives. Many primitives handling arrays have a property which specifies the size of the array. Because no memory may be allocated during real-time operation of the system, the size of all arrays must be known at the time the primitive net is loaded.

The complex data-type for arrays is internally based on the *shared_array* type provided by the Boost library [30]. The default semantics in C++ is to create a copy every time an object is assigned to a new variable, used as a function parameter, etc. The *shared_array* type is internally implemented to share the real data memory between all copies that are created due to the copy semantic of C++. Assigning the results of the `get(...)` methods to a local variable when reading an input port thus automatically creates a copy of the array object, the contents of the array however are shared with the output port (and all other connected input ports). Therefore the contents of the array must not be modified after having been read from an input port.

The Array data type is specified as Array<T> in the style of C++ templates (which are in fact used in the implementation of the SoftRobot RCC) in the following descriptions.

Core::TArray

Input ports	none
Output ports	outArray: Array<T>
Parameters	size: integer

The *TArray* primitive creates a new, empty array data-flow with the size given as parameter. The created array will have all values initialized with the default value of the data type T (e.g. *false* for Boolean, 0 for integer, etc.). The *TArray* primitive can be used, if a primitive requires an array data type for input, but no other primitive generates an appropriate array, i.e. the array must be created manually.

Core::TArrayGet

Input ports	inArray: Array<T>
Output ports	outValue: T
Parameters	index: integer

The *TArrayGet* primitive extracts a single value from an array. The array must be provided at input port *inArray*, and the desired index using parameter *index*. The output port *outValue* contains the item of the array. If the index is out of bounds, a *null* value is written to outValue.

The *TArrayGet* primitive does not need to know the size of the array, because it can access the already allocated memory of the originating output port.

Core::TArraySet

Input ports	inArray: Array<T> inValue: T
Output ports	outArray: Array<T>
Parameters	index: integer size: integer

The *TArraySet* primitive allows to set one item in an array to a new value. The array to change is read from input port *inArray* and the modified array is written to output port *outArray*. Using properties, the index and size of the array are specified. The new value is read from input port *inValue*.

The size of the array must be known at the initialization of the primitive net, because in order to set a value in an array, the array itself must be copied. Input ports only provide a pointer to the memory of the output port. Because the array provided by the previous primitive may be required unaltered at another primitive (output ports may be connected to any number of input ports), the modification of an array may only be performed on a copy.

Core::TArraySlice

Input ports	inArray: Array<T>
Output ports	outArray: Array<T>
Parameters	from: integer size: integer

The *TArraySlice* primitive extracts a part of an array. A new array with the size given as parameter is created. It is filled with the data from the input array (*inArray*), starting with index *from*. The input array must have a size of at least *from* + *size*.

7.3.4. Comparing values in a primitive net

Besides the aforementioned basic primitives, there is also a set of primitives which is only defined for data types which are comparable. The SoftRobot RCC contains implementations for the following primitives for the data types integer and double.

Core::TEquals

Input ports	inFirst: T inSecond: T
Output ports	outValue: Boolean
Parameters	first: T second: T epsilon: T

The *TEquals* primitive allows to check on values for equality. The values can be specified either using the input ports *inFirst* and *inSecond*, or alternatively using the parameters *first* and *second*. The parameters are only used, if the input ports are not connected. Because analog values cannot be represented exactly using a digital system, it is possible that two values differ slightly although they are considered equal. Using the parameter *epsilon*, a limit for deviation of two values which should still be considered equal can be given.

Special values such as “not a number” (NaN) are handled according to the IEEE 754 [63] standard. NaN never equals any value including itself. *Null* values are handled analog to NaN, i.e. *null* values are also considered unequal to any value.

Core::TGreater

Input ports	inFirst: T inSecond: T
Output ports	outValue: Boolean
Parameters	first: T second: T

The *TGreater* primitive works almost identically to the *TEquals* primitive with the exception, that no *epsilon* parameter is required. The result value *true* is returned, if the first value is greater than the second. There is no *TSmaller* primitive as it is sufficient to swap the operands for this purpose.

7.3.5. Arithmetic operations in primitive nets

For data types which support basic arithmetic operations, some primitives are provided for addition, multiplication and division. The following primitives are implemented for integer and double data types in the SoftRobot RCC.

Core::TAdd

Input ports	inFirst: T inSecond: T
Output ports	outValue: T
Parameters	first: T second: T

The summands of the *TAdd* primitive can be specified either by using the input ports *inFirst* and *inSecond*, or using the parameters *first* and *second*. The output port *outValue* contains the sum of both summands.

Specifying NaN (not a number) or *null* as one or both summands will result in a NaN value. There is no subtraction primitive, because subtraction can also be achieved by adding the negative value (which can be achieved by multiplication with -1, if necessary).

Core::TMultiply

Input ports	inFirst: T inSecond: T
Output ports	outValue: T
Parameters	first: T second: T

The *TMultiply* primitive has the same signature as *TAdd* and returns the product of both specified values. NaN values are handled according to the IEEE 754 standard, i.e. specifying a NaN value results in a NaN value.

Core::TDivide

Input ports	inFirst: T inSecond: T
Output ports	outValue: T
Parameters	first: T second: T

The *TDivide* primitive also has the same signature as *TAdd*. Values are handled according to the IEEE standard, e.g. a division by zero results in a NaN value.

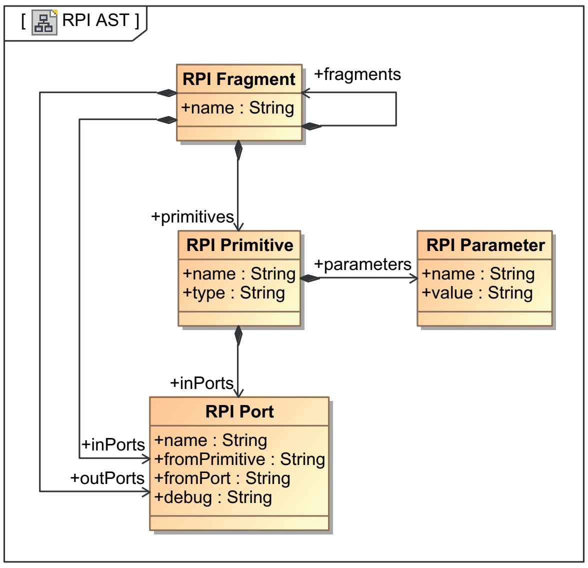


Figure 7.3.: UML class diagram representing the abstract syntax tree of a primitive net

7.4. Primitive net creation

Before a primitive net can be executed, it must be transmitted from the robotics application to the SoftRobot RCC, where it should be executed. For receiving the primitive net, the integrated web server is used. Therefore, the primitive net must be encoded as text. There are two ways of encoding available, a human readable XML format and a more compact and therefore faster custom language. Both interfaces are described in detail in Section 7.6. After the primitive net has been received, the text is parsed into a special data structure which represents the abstract syntax tree (AST) of the primitive net specification. Beginning from this AST, in the next step all primitives are instantiated and their ports connected. If the configuration of all primitives succeeds, the primitive net is now ready for execution. The following sections describe the abstract syntax tree and the instantiation of new primitives in more detail.

7.4.1. Abstract syntax tree of primitive nets

Figure 7.3 shows a UML class diagram of the abstract syntax tree which is generated during parsing of the primitive net specification. All attributes are of type String, because no conversion of types is done during the parsing process.

Primitives are represented by class *RPI Primitive*. The attribute *type* specifies, which type of primitive should be created (e.g. *DoubleValue*, *BooleanAnd*, . . .). The attribute *name* is a unique identifier of this concrete primitive instance and is later used for the creation of links between primitives. Parameters of primitives are parsed into instances of the class *RPI Parameter*, which contains two attributes, one for the name of the parameter and one for the value. The type of a parameter does not need to be specified in the AST, because it is internally known by the primitive implementation.

Communication ports are represented by objects of the class *RPI Port*. Ports have an attribute *name* as unique identifier of the port and two attributes *fromPrimitive* and *fromPort* to describe the link. The attribute *debug* allows to enable debugging features for the given port (cf. Section 7.7). For primitives, all available input and output ports are known based on the type of the primitive. Therefore, it is sufficient to specify only one direction of a link between two ports. It has been chosen to specify input ports for primitives in the AST, because unlike output ports, input ports can only be connected to a single port. Fragments, however, do not have any predefined interface, thus it is necessary to specify both input and output ports of a fragment.

For output ports on fragments, the attributes *fromPrimitive* and *fromPort* refer to primitives which are contained in the considered fragment. Creating this “virtual” link allows data from output ports of primitives contained in a fragment to be available outside the fragment. For input ports, the attributes *fromPrimitive* and *fromPort* refer to other primitives or fragments contained on the same level (i.e. an input port cannot be connected directly to an output port of a primitive that is contained in a sub-fragment – an output port of that sub-fragment must be used instead). Furthermore, they can also refer to input ports which have been defined in the fragment they are directly contained in. This is necessary to provide data from the outside of a fragment to the inside. To connect to the containing fragment, the name of this fragment must be used as attribute *fromPrimitive*.

Finally, the class *RPI Fragment* represents fragments, which have an attribute *name* as unique identifier. The whole primitive net specification consists of one single fragment, the root fragment, which contains all further primitives and sub-fragments.

7.4.2. Instantiation of primitives

Figure 7.4 shows a UML class diagram to demonstrate the instantiation of primitives with the primitive *DoubleValue* as an example. Primitives are instantiated by using the factory pattern [45]. Usually, primitives are implemented within extension modules, which are not necessarily built together with the main executable. It is desirable to use the C++ operators `new` and `delete` only for classes which have been generated by the same compiler, in case the memory layout of private elements in classes differs between two different compilers, thus factories provided by the extension libraries are used. For more details please refer to Section 9.5.

The class *Fragment* is responsible for instantiating all primitives and connecting the appropriate ports. The class *Net* initially creates the root fragment, stores the abstract syntax tree (AST) of the primitive net (cf. Section 7.6) in the root fragment and calls its

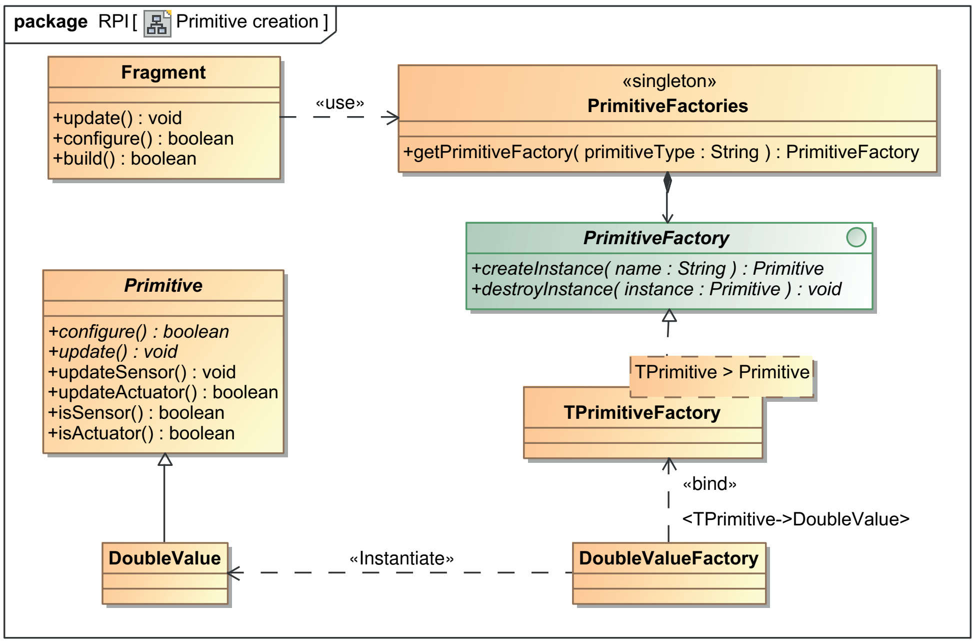


Figure 7.4.: Creation of primitive instances with example primitive "DoubleValue"

`build()` method. The first step of the `build()` method is to create instances of *Fragment* for all sub-fragments that are required and recursively calling `build()` on these fragments. After all sub-fragments have been created, instances of all primitives directly contained in the current fragment are created.

To create a primitive instance, the method `getPrimitiveFactory(type)` is called on the singleton class *PrimitiveFactories*. During the loading phase of an extension library, an instance of a factory for every primitive in the library is created and registered with *PrimitiveFactories*. If no appropriate factory is found, the net is rejected due to an unavailable primitive. Otherwise, a reference to the desired factory is returned. All factories must implement the *PrimitiveFactory* interface, which has the method `createInstance(name)` to create a new instance of a primitive, and `destroyInstance(primitive)` to remove a given instance. Using these factories, the fragment can create instances of all primitives.

In the next step, all ports must be connected according to the specification provided by the robotics application. The fragment iterates over all sub-fragments and primitives in the AST and tries to create links according to the *inPorts* lists of the AST. To create a link, the concrete instances of primitives are looked up using their name as a unique identifier. It is checked by both the source and the destination primitive that the specified ports exist and that their port types match. If any error occurs, the primitive net is rejected. If a connection from a primitive's input port is made to the fragment itself, the specified port name and input port of the primitive is recorded in the fragment, but no connection is yet established.

After all input ports have been connected, output ports for the fragment are generated. Similar to connections from an input port to the surrounding fragment, output ports of fragments are only stored in a list attached to the fragment, but not yet directly connected. The real connections will be established once the surrounding layer has its ports connected. If ports of a fragment need to be connected (either due to connecting its input ports or another primitive wants to connect to an output port), the port names are looked up in the list and the connection is created directly between the participating primitives. In the implementation, input and output ports of fragments are purely virtual and replaced by direct connections of primitives on different layers, thus achieving maximum performance. Having an explicit notation of ports on fragments however ensures encapsulation and reusability of fragments.

After all ports inside the fragment have been connected, the primitives (and sub-fragments) can now be sorted topologically according to the links among them. *Pre* primitives and links connected to them are left out from the sorting, because they are allowed to form cycles and always delay the data transmission to the next execution cycle. If there is a cycle without a *Pre* primitive, the sorting fails and the primitive net is rejected with a list of all primitives that were part of the cycle. An ordered list of all primitives is saved.

As the next step, all primitives need their parameters to be set to the specified values. The conversion of the value provided as a string to the concrete data type of the parameter is performed by the *TypeKit* which must be provided for every parameter. If a parameter is specified which does not exist on the given primitive, or the specified value cannot be successfully converted to the required data type, the creation of the primitive net fails.

As the last step, the `configure()` method is called for every primitive in the primitive net. The primitives must now perform all non real-time initializations, and check whether all parameters are set correctly and all necessary input ports are connected. If no error occurred, the primitive net is now ready for execution and its state is changed to *Ready*. At this time, all memory resources already need to be allocated to the net. However, hardware resources will not yet be allocated as there may be other primitive nets which still are using these resources.

7.5. Primitive net execution

Figure 7.5 shows a UML class diagram of the execution environment for primitive nets. After the robotics application has specified a primitive net to load, an instance of class *Net* is created by the Registry and the root fragment is built (cf. Section 7.4.2). A primitive net can be associated with a *Session*, which serves as a grouping container for different primitive nets which have been created by the same robotics application and can be used for debugging purposes (e.g. to remove all remains of a crashed robotics application).

Each primitive net has a state which is one of the states introduced in Section 5.3.3, Fig. 5.3. For technical reasons, one more state has been added: *Unloading*. This state is

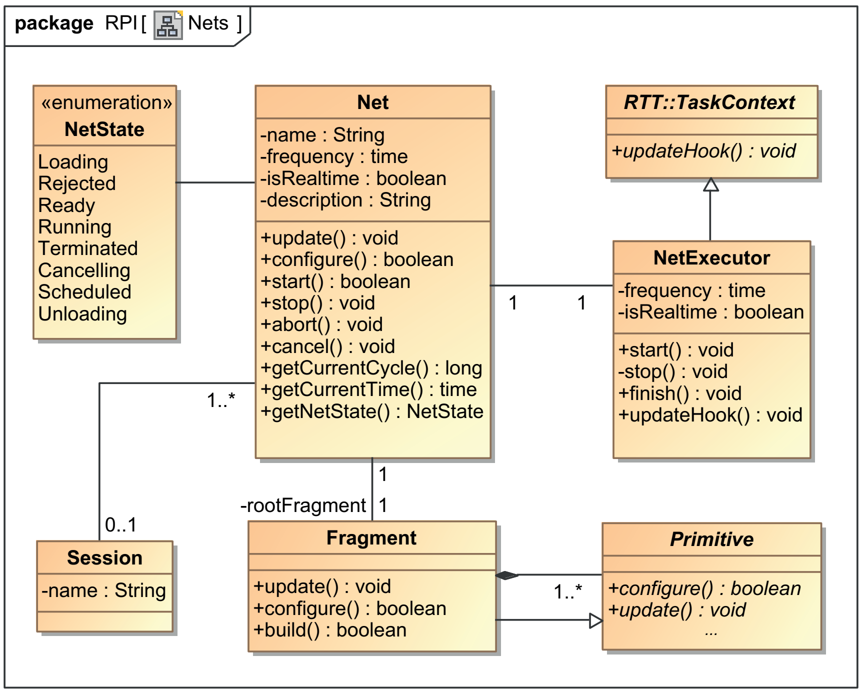


Figure 7.5.: UML class diagram for the primitive net execution environment

entered once the Registry has been asked to completely unload a primitive net. Primitive nets with state *Unloading* are hidden from applications, however they might still need some time to clean up resources before they can be completely removed from the execution environment.

The class *Net* does not provide means for real-time execution itself, but utilizes the class *NetExecutor*. This class is derived from the *TaskContext* class which is provided by the Orocos Real-Time Toolkit (RTT) [21] which abstracts from all threading and real-time related concerns of the underlying operation system. Instances of the class *NetExecutor* can run more than one primitive net during their lifetime. If a primitive net is initially started, a new NetExecutor is also created. If another primitive net is scheduled for immediate execution after another primitive net, the existing NetExecutor can be reused. This allows for a smooth transition from one primitive net to another without losing a single execution cycle or having different cycle times.

The *NetExecutor* class uses the *TaskContext* to create a new thread on the real-time operation system which is executed cyclically at the given frequency. The priority of the thread is determined by the attribute *isRealtime*. Primitive nets which do not require hard real-time are started with a lower priority. The TaskContext calls its `updateHook()` method in each execution cycle, which is overridden in the *NetExecutor* class. The *NetExecutor* subsequently calls the `update()` method in the currently associated *Net*.

The execution of a primitive net happens in four stages:

1. All real-time sensor values are retrieved by calling the `updateSensor()` method of all sensor primitives.
2. The main calculation of the primitive net is triggered by calling the `update()` method of the root fragment. Each fragment contains a topologically ordered list of the primitives (and fragments) it contains and calls their `update()` method in the appropriate order. This allows every fragment to receive current values at its input ports and to populate its output ports with newly calculated values.
3. All newly calculated set-points for actuators are transmitted to the drivers by calling the `updateActuator()` method of all actuator primitives.
4. The value of the *outTerminate* output port of the root fragment is checked. If it is true, the execution of the primitive net is terminated. The associated *NetExecutor* is also stopped and the operating system thread removed, if the *NetExecutor* is not required for further primitive nets (for more details please refer to Chapter 8).

If a primitive net is canceled, the *Cancel* primitive will emit the value *true* on its output port. It is up to the design of the primitive net to terminate gracefully by eventually setting the *outTerminate* ouput port of the root fragment to *true*. If no *Cancel* primitive is included in the primitive net, nothing will happen. Aborting a primitive net prevents the *NetExecutor* from starting a new execution cycle; a currently running cycle will still be completed.

Lustre programs are ultimately compiled into a host language (such as C), which is then again compiled into machine code. For the SoftRobot RCC reference implementation we have opted not to use a compiler based approach, but rather to interpret primitive nets. Although interpreting has some performance drawbacks, it also offers certain advantages:

- Compiling a primitive net takes some time, which adds up to the delay between the time when the robot application has issued a primitive net, and when it is ready for execution. In particular with the synchronization mechanism (cf. Chapter 6) this delay could prevent successful blending from one primitive net into the other.
- The main components, the primitives, themselves are already compiled C++ code. Thus the interpreter is only responsible for calling a sequence of methods to execute all primitives in the right order. Data transmission is performed by shared access to the same memory for input and output ports, thus no overhead is created here.

7.6. Communication interface

Robotics applications need to communicate with the Robot Control core to load and start primitive nets, to issue synchronization rules for multiple synchronized primitive nets and to receive status information about running primitive nets. It must also be possible for running primitive nets to transfer data bidirectionally between the application and the

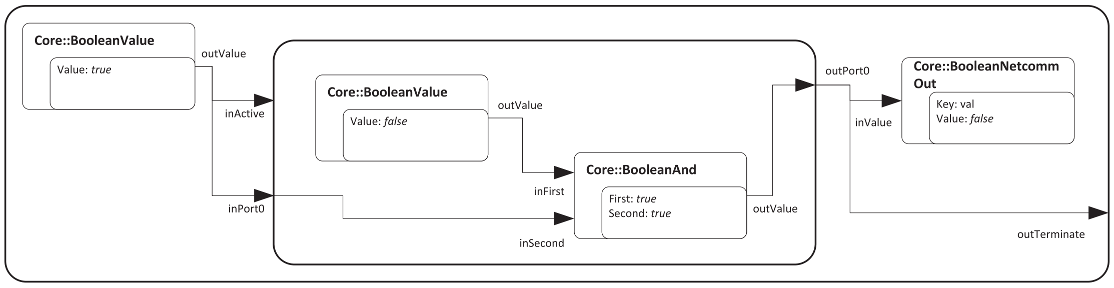


Figure 7.6.: Example primitive net used for demonstrating the transmission of primitive nets

primitive net. Such data includes status information (e.g. a certain point on a trajectory has been reached) which can be displayed by the application to the user, but also some forms of external control of the primitive net such as a global override value which enables the user to slow down the overall execution velocity for development purposes. Because neither the application nor the communication channel is real-time safe, primitive nets are designed always to reach a safe state even if the communication channel is interrupted or delayed.

Special primitives, the *NetComm* primitives, are available to transmit data between a primitive net and the robotics application. For the communication protocol, two different variants are available with the SoftRobot RCC. The first variant uses plain HTTP with XML files (and XSLT files for better human readability) for all communication aspects. One drawback of using plain HTTP however, is that all information must be requested from the RCC by polling. Furthermore, encoding everything as an XML document can cause considerable overhead. To mitigate these issues, a special protocol called “DirectIO” was developed, which can either be used over a plain TCP socket or alternatively tunneled over HTTP using the WebSocket technology [39].

Figure 7.6 shows an exemplary primitive net which will be used for demonstrating the transmission of primitive nets from the Robotics API to the RCC using both communication protocols. The root fragment contains one *BooleanValue* primitive having the value *true*, one *BooleanNetcommOut* primitive (with the parameter *key* set to “val” and default value *false*) and one sub-fragment. This sub-fragment again contains two primitives, another *BooleanValue* (with value *false)* and a *BooleanAnd* primitive. Furthermore, the fragment has a named input port *inPort0* and a named output port *outPort0*. Both the activation input port of the sub-fragment and the named input port *inPort0* are connected to the *BooleanValue* primitive outside the fragment. Inside the sub-fragment, the *BooleanAnd* primitive is connected to the second *BooleanValue* primitive and to a named input port *inPort0*. Its result is written to a named output port *outPort0*. The communication primitive receives its data from the port *outPort0* from the sub-fragment, as well does the named output port *outTerminate* of the root fragment. The value of this named port controls the termination of the primitive net.

Executing this primitive net will result in an infinitely running primitive net. The *BooleanAnd* primitive will receive one *true* and one *false* signal and thus emit *false*. This value is transmitted to the robotics application via the *BooleanNetcommOut* primitive, but also written to the *outTerminate* output port. As long as this value does not receive a *true* value, the primitive net continues running.

7.6.1. Communication primitives

For communication of a robotics application with a currently running primitive net, two special classes of communication primitives have been introduced.

TNetcommIn

Input ports	none
Output ports	outValue: T outLastUpdated: nsecs
Parameters	Key: string Value: T

TNetcommIn primitives are used for inserting data into a running primitive net. Using one of the communication channels between the robotics application and the Robot Control Core, it is possible to update the value of the primitive, which can be retrieved within the primitive net using the output port *outValue*. The output port *outLastUpdated* carries a time-stamp of the last update from the robotics application. To address a *TNetcommIn* primitive, the parameter *Key* is used as a unique identifier. Using the parameter *Value*, it is possible to set a default value which is available on *outValue* from the start of the primitive net, even if no update has been performed by the robotics application yet. If no default value is set, the default value of the data type is used.

TNetcommIn primitives are available for all common data types. Primitive data types include Boolean, integer or double. Complex data types such as arrays or custom defined data structures are also possible. Because transmission is performed using a string representation, bidirectional conversion of the data type and a string must be possible. *TypeKits* are used for that purpose just like with parameters.

TNetcommOut

Input ports	inValue: T
Output ports	none
Parameters	Key: string Value: T

The *TNetcommOut* primitive works in the opposite direction as *TNetcommIn* primitives to retrieve values from a running primitive net. Using parameter *Key* as a unique identifier, it is possible to retrieve the last value that has been received on input port *inValue* in the robotics application. If no new value is available (e.g. the *TNetcommOut* primitive was in an inactive fragment), the last recorded value is still available together with a time-stamp to recognize old values. If no value has ever been set, the value from parameter *Value* is used.

TInterNetcommIn/Out

Input ports	none
Output ports	outValue: T
Parameters	Key: string Value: T RCC: string Net: string

Input ports	inValue: T
Output ports	none
Parameters	Key: string Value: T RCC: string Net: string

The *TInterNetcommIn* and *TInterNetcommOut* primitives are intended for communication across multiple primitive nets. A *TInterNetcommOut* primitive writes data to a *TNetcommIn* primitive in another primitive net, and *TInterNetcommIn* reads the value from a *TNetcommOut* primitive respectively. Using the parameter *Net*, the remote primitive net can be addressed, and *Key* is used to find the appropriate primitive. The parameter *RCC* is currently unused and is intended for a future extension to support real-time communication across multiple RCCs.

The values of all input communication primitives are only updated during phase 1, and output communication primitive write their values during phase 3 of the primitive net execution. This guarantees that the values read within a primitive net remain consistent during the execution cycle. The communication between multiple primitive nets running on a single RCC is real-time safe, i.e. values written in one primitive net (during phase 3) will be available on the next start of an execution cycle of the other primitive net.

7.6.2. Plain HTTP communication

The SoftRobot RCC provides a HTTP/1.1 [40] compatible web server, which is based on mongoose.[1] The web server mainly provides XML data instead of HTML web pages, because the main user of the web server is the robotics application, not the human user. XML is intended to be machine readable, whereas HTML is more intended for human readable web pages. For convenience, XSLT files [28, 73] are provided which enable an XSLT capable web browser to render plain XML files as human readable HTML web pages. Data transmission from the application to the RCC is done using HTTP-POST requests. All connections originate from the robotics application and data transmissions only occur upon request. Polling is required to receive updated values.

[1] Available from: `https://code.google.com/p/mongoose/`, the MIT licensed version has been used in the SoftRobot RCC.

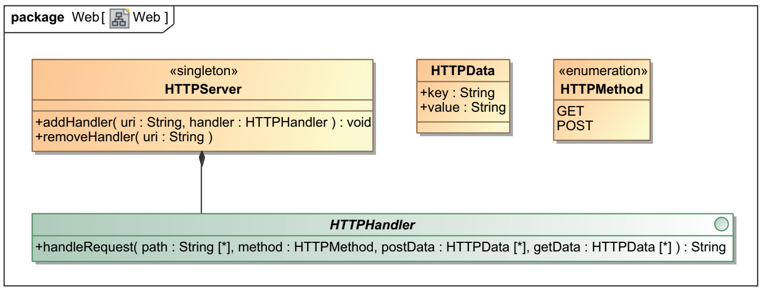

package Web[Web]
«singleton»
HTTPServer
+addHandler(uri : String, handler : HTTPHandler) : void
+removeHandler(uri : String)
HTTPData
+key : String
+value : String
«enumeration»
HTTPMethod
GET
POST
HTTPHandler
+handleRequest(path : String [*], method : HTTPMethod, postData : HTTPData [*], getData : HTTPData [*]) : String

Figure 7.7.: Architecture of web server

Architecture of web server

Figure 7.7 shows a UML class diagram with the main components of the integrated web server. The singleton class *HTTPServer* coordinates all HTTP requests and dispatches them to the right handler. For some basic functionality, web handlers are already delivered with the SoftRobot RCC. Examples for such handlers are to provide lists of all available primitives, or of the currently running primitive net. Additional web pages can be provided by extension libraries by registering new web handlers during their loading phase.

To create a web handler, the interface *HTTPHandler* must be implemented, and an instance of the created class must be registered with *HTTPServer*. To register a web handler, the desired URI (uniform resource identifier) must be provided to the method `addHandler(...)`. The URI can be a static string (e.g. `/devices/sampleDevice/`), but can also contain wildcards (e.g. `/devices/sampleDevice/*`). The first URI will only match to requests to the exact address. Wildcards can be replaced by any string, including the empty string, but at most to the next path delimiter "/". Multiple wildcards are possible, even in the middle of the URI (e.g. `/devices/robot/*/joint/*` would match to `/devices/robot/robot1/joint/0`).

When a HTTP request arrives at the web server, a look up for the right handler is performed and the handler's `handleRequest(...)` method is called. The parameter *path* of this method call provides a list of the requested path parts (separated using the path delimiter "/"). The handler can use these path parts to retrieve values which have been used for the wildcard-parts of the URI. The parameter *method* informs the handler whether a HTTP GET or HTTP POST request has been made. Many handlers provide information on GET requests, while actually performing actions only on POST requests. Each HTTP request can contain additional data appended to the URI, separated by a question mark (e.g. `uri?key1=value1&key2=value2`). The key/value-pairs of the request is provided using the parameter *getData*. For POST requests, additional data can be transmitted in the request independent of the URI. These key/value-pairs are available in the parameter *postData*. For GET requests, this parameter is empty. The `handleRequest(...)` method must return a string, which is directly transmitted to

```
<rpinet>
   <fragment id="frag">
      <primitive id="bv_false" type="Core::BooleanValue">
         <parameter name="Value" value="false"/>
      </primitive>
      <primitive id="b_and" type="Core::BooleanAnd">
         <parameter name="First" value="true"/>
         <parameter name="Second" value="true"/>
         <port name="inFirst" fromprimitive="bv_false" fromport="outValue"/>
         <port name="inSecond" fromprimitive="frag" fromport="inPort0"/>
      </primitive>
      <inport name="inActive" fromprimitive="bv_true" fromport="outValue"/>
      <inport name="inPort0" fromprimitive="bv_true" fromport="outValue"/>
      <outport name="outPort0" fromprimitive="b_and" fromport="outValue"/>
   </fragment>
   <primitive id="bv_true" type="Core::BooleanValue">
      <parameter name="Value" value="true"/>
   </primitive>
   <primitive id="ncout" type="Core::BooleanNetcommOut">
      <parameter name="Key" value="val"/>
      <parameter name="Value" value="false"/>
      <port name="inActive" fromprimitive="bv_true" fromport="outValue"/>
      <port name="inValue" fromprimitive="frag" fromport="outPort0"/>
   </primitive>
   <outport name="outTerminate" fromprimitive="frag" fromport="outPort0"/>
</rpinet>
```

Listing 7.1: XML representation of a primitive net including a fragment

the HTTP client. The format of this response is completely up to the actual handler. Usually, XML data is returned, however some extension libraries may also opt to create HTML pages directly (e.g. for debugging purposes).

Primitive net creation

For the creation of new primitive nets, a special web handler exists at the URI `/nets/`. This handler expects an XML formatted description of a primitive net as POST parameter, which is then parsed and forwarded to the Registry. The full XML schema definition (XSD) for primitive nets can be found in Appendix A.1.

Listing 7.1 shows the XML representation of the primitive net introduced in Fig. 7.6. The root fragment is implicitly created using the root tag of the XML document. Inside this tag, further primitives and fragments can be nested. All primitives and fragments need to have a unique identifier, specified with attribute `id`. This identifier is used to connect ports from one primitive to another, or to named ports on fragments (e.g. the

```
1 <?xml version="1.0" encoding="UTF-8"?><?xml-stylesheet type="text/xsl" href="/xsl/net.
      xsl" ?>
2 <net name="rpinet0" status="RUNNING" desc="">
3    <data key="outval">false</data>
4 </net>
```

Listing 7.2: Status data for primitive net

ports *inPort0* and *outPort0*). The SoftRobot RCC uses the RapidXml [71] project as DOM parser [86] to create the abstract syntax tree of the primitive net (cf. Section 7.4.1).

Live communication

When a primitive net is loaded, the RCC assigns a unique name to the primitive net. Using the URI `/nets/`*`netname`*`/`, it is possible to retrieve status data from the net. The XML telegram listed in Listing 7.2 is returned for instance after the primitive net drafted in Listing 7.1 has been started.

Line 1 contains the XML header and a reference to a XSLT file for rendering the output on a web browser. The root tag *net* in line 2 contains all relevant data for the primitive net, such as the name, the current status and optionally a human-readable description. Using *data* tags (e.g. in line 3), the values of all communication primitives are transmitted. For *NetcommOut* primitives, the given key has prefix “out”, for *NetcommIn* primitives, the key is prefixed with “in”. In order to provide new values for *NetcommIn* primitives, a POST request to the URI must be done with all desired key/value pairs in the POST data.

Live data will only be provided by the RCC upon request. Once the RCC receives a request for net status, all relevant data is gathered. Because the real-time safe execution of the primitive net must not be disturbed, no global locking is performed to retrieve the communication data. Therefore, it might happen that the values presented on the status page may actually come from different execution cycles. There is also no guarantee that the robotics applications will notice all values that have ever been written to a communication primitive. If the value is overwritten too soon, and the robotics application did not request a net status meanwhile, the value will be lost. Therefore, robotics applications must not rely on being able to receive short peaks on data values from primitive nets. If knowledge about such short peaks is required, the detection must be done within the primitive net.

7.6.3. DirectIO

Communication using plain HTTP as described in the previous section has some major drawbacks. The representation of primitive nets using XML is rather easy to understand, however it causes a lot of overhead for the XML tags, XML attribute names, etc.

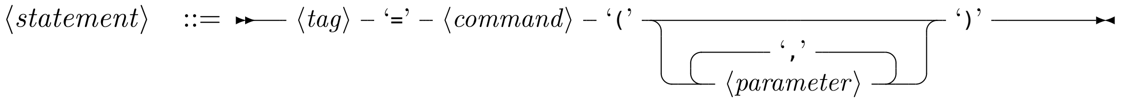
Furthermore, polling for live communication data is not very efficient, especially if it is unknown whether new data is available or not. To conquer these problems, a new protocol called “DirectIO” has been introduced, which is tailored to the communication needs of robotics applications and the RCC. DirectIO supports the loading of new primitive nets using a special domain specific language (DSL), as well as transmitting live communication data. For live data communication, the RCC notifies the application of new data. Besides these advantages, the DirectIO protocol also has some disadvantages. The DirectIO communication protocol cannot be extended as easily as adding a new handler for new extensions. Furthermore, debugging is much harder. With plain HTTP, a standard web browser can be used to interact with the RCC.

DirectIO is designed to be used over a TCP socket. It can be implemented using its own TCP connection, or by tunneling over HTTP using WebSockets. Tunneling over WebSockets has several advantages: Only one port must be known (and configured) for connecting to the RCC, and web-applications using JavaScript can easily access WebSockets as well. The SoftRobot RCC implements WebSockets. A connection to the DirectIO interface can be made by connecting to the URI `/DirectIO/`. In the following sections, the robotics application is referred to as client, because applications initiate the connection to the RCC.

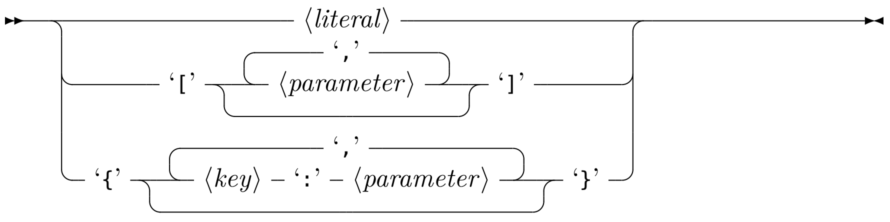
Protocol definition

The DirectIO protocol uses two forms of communication. At first, all communication happens synchronously, i.e. the client sends a command to the RCC, which is immediately responded to. However, there are some commands which enable asynchronous data transfer, i.e. the RCC may transmit data packages at any time. All statements contain a tag which can be chosen freely by the client. All responses from the RCC to the topic addressed by the initial command (including asynchronous transmissions) will carry the same tag. This allows the client to distinguish among different asynchronous responses.

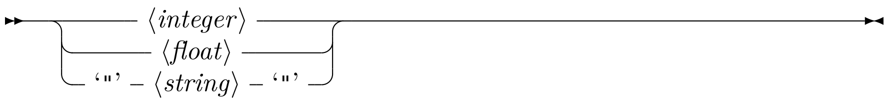
A DirectIO statement consists of the tag, the command to execute and optionally parameters for the command. A parameter can either be a literal (a string, an integer or a floating-point value), a list of parameters (including the empty list) or a list of key-value pairs with further parameters as values. A slightly simplified syntax-diagram (not all non-terminals are included) of the DirectIO language follows. The full grammar in Extended-Backus-Naur form can be found in Appendix A.2.



⟨*parameter*⟩ ::=



⟨*literal*⟩ ::=



After the communication link has been established, a handshake must be performed to ensure that the RCC and the client use the same protocol version:

```
< id1=ver("2.0")
> id1=ok("handshake␣ok")
```

In this example, < denotes a command sent by the client, and > a command sent by the RCC. In line 1, the client uses tag `id1` with command `ver` to announce that it is compatible with version 2.0 of the DirectIO protocol. In line 2, the RCC responds using the same tag and an `ok` command. Following this conversation, the full DirectIO protocol may be used.

DirectIO commands

The following commands are available using DirectIO and can be sent from the application to the RCC:

nse Creates a new session with given name. Returns the ID of the newly created session.

ase Aborts a given session. All primitive nets within the session are aborted and the session is destroyed.

nene Creates a new net, takes a (optional) session ID and the primitive net encoded in DirectIO format as parameters. Additional, optional parameters specify the cycle time of the primitive net and whether it is a real-time critical net. If those parameters are omitted, the default cycle time is used and the primitive net is considered as real-time critical.

nest, neca, neab, neun Starts, cancels, aborts or unloads a primitive net. The primitive net ID must be given as the parameter.

nesc Specifies a synchronization rule for sets of primitives (for more details see Sections 6.1 and 8.1).

gne Requests the current state of a primitive net. Arguments are the primitive net name and a refresh timeout. After submitting the *gne* command, the RCC will send the current state of the primitive net along with the current values of all communication primitives. If the state of the primitive net changes, the client will be notified immediately. If a value of a communication primitive changes, the client will be notified, unless there has been a notification already within the refresh time. If a value changes more often than the given refresh time, value changes will be lost. The status of the net will be sent from the RCC as an argument of command *ns*; the values of the communication primitives as an argument of command *nc*.

snc Sends new communication values to the RCC. Multiple communication primitives for multiple primitive nets can be transmitted within a single command. Key/value lists are employed to address the proper communication primitives.

gse Requests updates for all primitive nets within a session. After issuing this command, the status of all primitive nets is transmitted to the robotics application. After this initial transmission, only changes in primitive nets' states will be transmitted using command *st*.

gde Requests information for available devices on the RCC. An initial list of all devices will be transmitted, and further device state changes will be transmitted asynchronously. For more details on devices and device interfaces please refer to Chapter 9.

DirectIO responses

Responses from the RCC to the client use the same syntax as DirectIO commands sent by the client. The RCC always uses the same tag for replying to a command. All commands are immediately acknowledged with either an `ok` message or an `err` message. Both messages can contain further parameters which either indicate additional information about the successful command (e.g. the name of the newly created session) or an error message if a command did not succeed.

Besides the `ok` and `err` messages, the following asynchronous messages are also transmitted once activated:

Primitive net updates Updates for primitive nets are requested using the `gne` command. The RCC asynchronously transmits updates to the primitive net's state using `ns` messages and updates to the value of communication primitives using the `nc` message. Every status change of the primitive net will be reported to the client, however changes to values of communication primitives will only be reported once within the given refresh time.

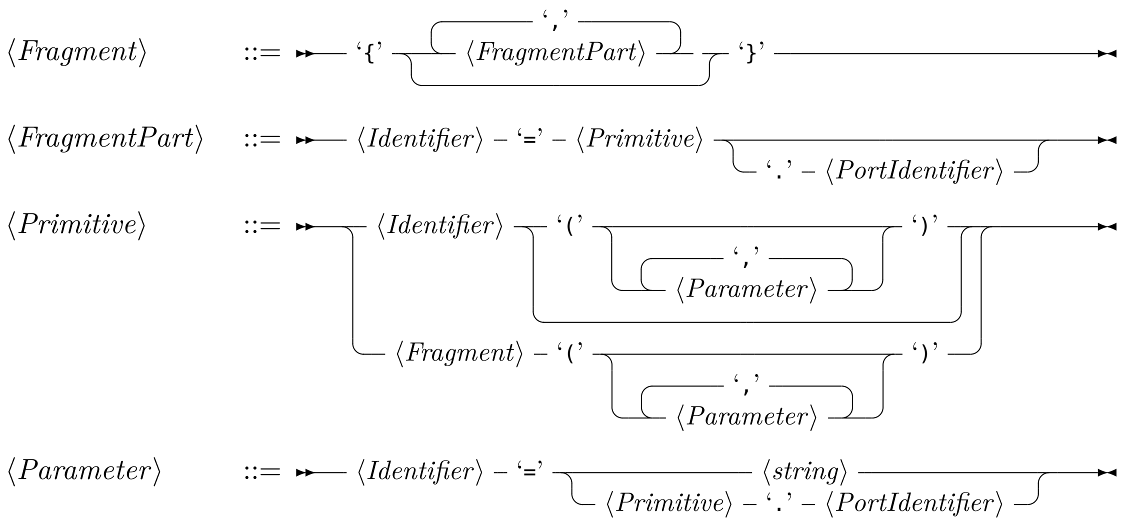
After the `gne` command has been received, the current net status and a complete list of all communication primitives are initially transmitted before the command is acknowledged.

Device information Updates for available devices are requested using the `gde` command. If a new device is available, a `da` message is transmitted to the client. This message contains the type, all implemented device interfaces and an optional map of configuration parameters for each device interface (cf. Section 9.2). If multiple devices are available at the same time, all devices are transmitted in a single `da` message to the client. If a device is removed from the RCC, a `dr` message is sent to the client. If a device changes its state (e.g. from safe-operational to operational), a `ds` message is sent to the client.

After the `gde` command has been received, one `da` and one `ds` message containing all currently available devices and their states are initially transmitted to the client. Examples for these messages are shown in Section 9.3.

Primitive net creation

For the transmission of primitive nets, a domain specific language (DSL) has been developed which can be transmitted using DirectIO (as string literal). The language for primitive net creation has not been embedded into the main DirectIO protocol in order to allow the net creation language also to be used with the plain HTTP protocol. A slightly simplified syntax-diagram (omitting non-terminals for character definitions) follows; the complete grammar can be found in Appendix A.3.


⟨Fragment⟩ ::= '{' ⟨FragmentPart⟩ ',' '}'
⟨FragmentPart⟩ ::= ⟨Identifier⟩ – '=' – ⟨Primitive⟩ '.' – ⟨PortIdentifier⟩
⟨Primitive⟩ ::= ⟨Identifier⟩ '(' ⟨Parameter⟩ ',' ')'
⟨Fragment⟩ – '(' ⟨Parameter⟩ ',' ')'
⟨Parameter⟩ ::= ⟨Identifier⟩ – '=' ⟨string⟩ ⟨Primitive⟩ – '.' – ⟨PortIdentifier⟩

Fragments, including the root fragment, are enclosed by curly braces. Therefore, all DirectIO net descriptions start with { and end with }. Within a fragment, Primitives can be specified as follows:

```
bv_true=Core::BooleanValue(Value='true')
```

This creates a *Core::BooleanValue* primitive and assigns the unique identifier *bv_true* to it. Between parentheses following the primitive type, all parameters can be specified in a `key='value'` syntax. For port connections, the syntax `key=`*`fromprimitive.fromport`* can be used, where *`fromprimitive`* is the unique identifier of the source primitive and *`fromport`* the output port name.

Fragments are created almost identically:

```
frag={...}(...)
```

Between the curly braces, all primitives and sub-fragments can be created. Furthermore, named output ports can be created by writing `portname=`*`primitive.outport`*. Between the round braces, input ports of the fragment can be connected just like with primitives. Primitives may refer to named input ports of the fragment by using the keyword `parent` as the source primitive name.

As an extension, it is also possible to use anonymous primitives. Primitives with only a single output port are often also connected to only a single successive primitive. In this case, there is no need for assigning an identifier to this primitive.

```
b_and=Core::BooleanAnd(inFirst=Core::BooleanValue(Value='false').outValue,...)
```

creates a primitive of type *Core::BooleanAnd* with the identifier *b_and*. The input port *inFirst* is connected to an anonymous primitive of type *Core::BooleanValue*. Because this primitive is used nowhere else, assigning an identifier and later referring to this identifier can be saved. In particular the *TValue* primitives are commonly used this way. By using anonymous primitives, some data overhead which otherwise would need to be transferred is saved.

The whole primitive net depicted in Fig. 7.6 is shown in Listing 7.3 using the DirectIO net representation. Please note that all indenting has only be done for presentation, the whole primitive net is transmitted without any spaces or line breaks on the wire. In line 1, the root fragment starts. In line 2, the sub-fragment named *frag* is started. Line 3 adds the primitive *b_and*, whose ports and parameters are set in lines 4 and 5. An anonymous primitive has been used for input port *inFirst*, and a named input port of the parent fragment for *inSecond*. Line 6 defines the named output port *outPort0* for the sub-fragment and connects it to primitive *b_and*. In line 7 the input ports of the sub-fragment are connected to the appropriate output port. Lines 8 to 13 create the other two primitives of the root fragment, and line 14 connects the *outTerminate* output port of the root fragment. Line 15 finally closes the root fragment.

The order of primitives and fragments within the DirectIO net description is not relevant. It is possible to connect to output ports of primitives which are defined later in the description. The SoftRobot RCC uses a parser which has been created by the Coco/R [89] parser generator. The results of this parser are used to create a data structure in the

```
{
 frag={
  b_and=Core::BooleanAnd(
    inFirstCore::BooleanValue(Value='false').outValue,
    inSecond=parent.inPort0,First='true',Second='true'),
   outPort0=b_and.outValue
 }(inActive=bv_true.outValue,inPort0=bv_true.outValue),
 bv_true=Core::BooleanValue(Value='true'),
 ncout=Core::BooleanNetcommOut(
  inActive=bv_true.outValue,
   inValue=frag.outPort0,
   Key='val',
   Value='false'),
 outTerminate=frag.outPort0
}
```

Listing 7.3: DirectIO representation of primitive net

format of the AST (cf. Section 7.4.1). Names for anonymous primitives are generated randomly as needed.

Example

Listing 7.4 shows an exemplary communication trace between a robotics application and the SoftRobot RCC. Commands sent from the robotics application are marked with < while responses from the RCC are marked with >.

Lines 1 and 2 are the protocol handshake. In line 3, a new primitive net is created. This primitive net contains two Boolean communication primitives (one for each communication direction) which are directly connected. The termination output is also connected to the inbound communication primitive (which has a default value of *false*). The primitive net is not assigned to a session (second parameter 0) and has the description “Demo-Net”. Line 4 acknowledges the creation of the primitive net and returns the identifier *rpinet0* for the new primitive net. In line 5, the robotics application registers a listener to all events belonging to *rpinet0*. This is responded with the net state *READY* in line 6 and furthermore acknowledged in line 7. The communication values are not transmitted because the primitive net is not yet running. In line 8, the application requests the primitive net to be started. This command is acknowledged in line 10. Line 9 uses the identifier of the `gne` command to notify the application about a status change of the primitive net, and in line 11 the value of the communication primitive *k2* is sent. Interleaving of direct responses to commands and asynchronous events with different tags can happen. The application must use the identifier to assign responses properly. In line 12, the value of the communication primitive with key *k1* is updated. The command is acknowledged in line 13. Lines 14 and 15 are results of this change in value. Due to the direct connection of both communication primitives, the value of the outwards

```
1  < a=ver("2.0")
2  > a=ok("handshake ok")
3  < b=nene("{ncin=Core::BooleanNetcommIn(Key='k1',Value='false'),ncout=Core::
       BooleanNetcommOut(inValue=ncin.outValue,Key='k2',Value='false'),outTerminate=ncin.
       outValue}",0,"Demo-Net")
4  > b=ok("rpinet0")
5  < c=gne("rpinet0",0.5)
6  > c=ns("READY")
7  > c=ok()#
8  < d=nest("rpinet0")
9  > c=ns("RUNNING")
10 > d=ok()
11 > c=nc({outk2:"false"})
12 < e=snc({rpinet0:{ink1:"true"}})
13 > e=ok()
14 > c=nc({outk2:"true"})
15 > c=ns("TERMINATED")
16 < f=neun("rpinet0")
17 > c=nc({outk2:"true"})
18 > c=ns("TERMINATED")
19 > f=ok()
```

Listing 7.4: Example communication trace between a robotics application and the SoftRobot RCC

communication primitive changes (line 14), and furthermore the primitive net itself terminates (line 15). In line 16 the final unloading of the primitive net is requested, which is acknowledged in line 19. Lines 17 and 18 are a final status update of the primitive net. After the primitive net has been unloaded, it cannot be accessed any more, and no further status messages will be transmitted from the RCC.

Comparison with plain HTTP protocol

The DirectIO protocol offers a shorter syntax, reducing the amount of data transfer necessary between the robotics application and the RCC. Furthermore, by pushing new net communication values upon availability relieves the application from constantly polling. The following table compares the size of a primitive net for a point-to-point motion transmitted using the XML format on one side and the DirectIO format on the other side:

net	primitives	size XML	size DirectIO	ratio
(a)	454	125,656 Bytes	62,376 Bytes	49.6%
(b)	344	79,000 Bytes	29,179 Bytes	36.9%

Both primitive nets have been automatically generated by the Robotics API. Primitive net (b) has been furthermore automatically optimized (cf. Section 10.2.7). This procedure substituted redundant primitives with a single instance and, more importantly, renames all primitives with short random names. With shorter names, the overhead of the XML syntax preponderates.

7.7. Debugging

Primitive nets are intended for automatic generation, and robotics application developers should not directly come into contact with primitive nets. The Robotics API (cf. Chapter 10) provides a Java-based framework that performs such an automated generation of primitive nets from a Java-based robotics application. During the development of the Robotics API (or any other framework that generates primitive nets) or of RCC extensions e.g. for new hardware support, it is possible that debugging must be performed on the level of primitive nets, i.e. to diagnose wrong links between primitives. Errors in the execution of primitive nets can occur on different stages:

1. Syntactic error in primitive net specification
2. Invalid primitive net specification
3. Configuration error for a primitive
4. Semantic error in primitive net

Syntactic errors in the primitive net specification are very rare due to the automated generation process. The Robotics API e.g. generates the XML or DirectIO primitive net specification from Java proxy objects, thus unless this algorithm is changed, syntactic errors are not to be expected. The SoftRobot RCC rejects syntactically invalid primitive nets with an error message generated by the parser.

A primitive net specification can be syntactically correct but still be invalid. This happens if specified primitives are not available, unknown input or output ports are used or if port types of two linked primitives do not match. A common reason for primitives not being available is a missing (hardware) extension which would provide the appropriate primitive. Cycles in primitive nets without a *Pre* primitive are also a reason for an invalid primitive net specification.

Even if all primitives are available and properly linked, the configuration of a primitive can still fail. For example, if the specified robot for a *JointPosition* primitive is not available, this primitive's configuration will fail. Some primitives also fail configuration if a required input port is not connected.

Errors 1 to 3 can be detected automatically by the SoftRobot RCC, and primitive nets containing one of those errors cannot be started (and enter state *Rejected*). An error message containing the reason for rejecting the primitive net is available for debugging purposes. Primitive nets can also contain semantic errors, i.e. they can be loaded perfectly well and even be started, but do not perform the task they are designed for properly. These errors cannot be detected automatically, and finding the root cause of such errors can be very time consuming. Therefore, the SoftRobot RCC provides some debugging support to help find semantic errors in primitive nets.

7.7.1. Debugging live values of primitives

One important piece of information for diagnosing errors in primitive nets is the current value of links connecting primitives. Due to low cycle times (usually under 10 ms), it is almost impossible to read those values "live". Using the communication primitives (cf. Section 7.6.1), it is possible to transmit values from a running primitive net to an external system (usually this is the robotics application which started the primitive net, but a debugging environment could also access this data) near-live, however the communication primitives cannot guarantee that all values are transmitted (especially if values fluctuate very fast).

If it is desirable to have access to all values that have been sent over a link, the SoftRobot RCC offers special debugging primitives which write all values they receive to a ring buffer. This ring buffer can be retrieved from the RCC at any time, even after the primitive net has terminated (until it is unloaded). Using this log, offline debugging is possible. To facilitate the use of the debugging primitives, it is possible to specify input ports of primitives for which a debugging log should be created. When the primitive net is loaded, debugging primitives are automatically inserted and connected to the same output port as the monitored input port. Debugging is specified using input ports, because output ports are neither specified using the XML protocol nor using DirectIO. For each input port, the size of the ring buffer can be given as the time for which the log should be kept. Log entries older than this time are automatically overwritten with newer data. Because the primitive net is executed with real-time guarantees even while debugging is enabled, it is not possible to re-size the ring buffer during run-time.

The debug primitives are available for all data-types that have an appropriate *TypeKit*, i.e. also complex data types can be logged. If debugging is requested for a data-type without a *TypeKit*, a warning message will be logged and the net will be created without the debugging primitive. The main functionality of the primitive net is not impacted, only debugging is restricted.

The SoftRobot RCC has a configuration parameter which controls how debugging primitives are created. Debugging can be disabled completely, debugging primitives can be inserted as requested by the application (default), or all input ports can be logged with a given ring buffer size. Creating debugging logs does impact performance of the application. For each input port which has debugging enabled, an entry in the ring buffer must be written in each primitive net cycle. If debugging is enabled for a very large number of input ports in a very large primitive net, this can cause considerable overhead. Furthermore, memory consumption can also be high if a large number of ports has to be logged for a long period of time. All memory is allocated before the primitive net is started and only freed once the primitive net has been completely unloaded.

Debugging primitives are handled just like any other primitive, i.e. if an input port inside a fragment is debugged, no values will be logged while the fragment is inactive. The debugging primitive stores the current cycle count together with the current port value so that phases of inactivity can be clearly seen.

The generated log can be retrieved using an HTTP request. A debugging web handler is registered for each primitive net and returns a single file containing one row of debug values (tabulator separated for all debug primitives in the net) per primitive net cycle. Every row also contains the primitive net cycle counter, and the time of the primitive net (seconds since the net's start). For complex data types, *TypeKits* are used identically to communication primitives for log file generation.

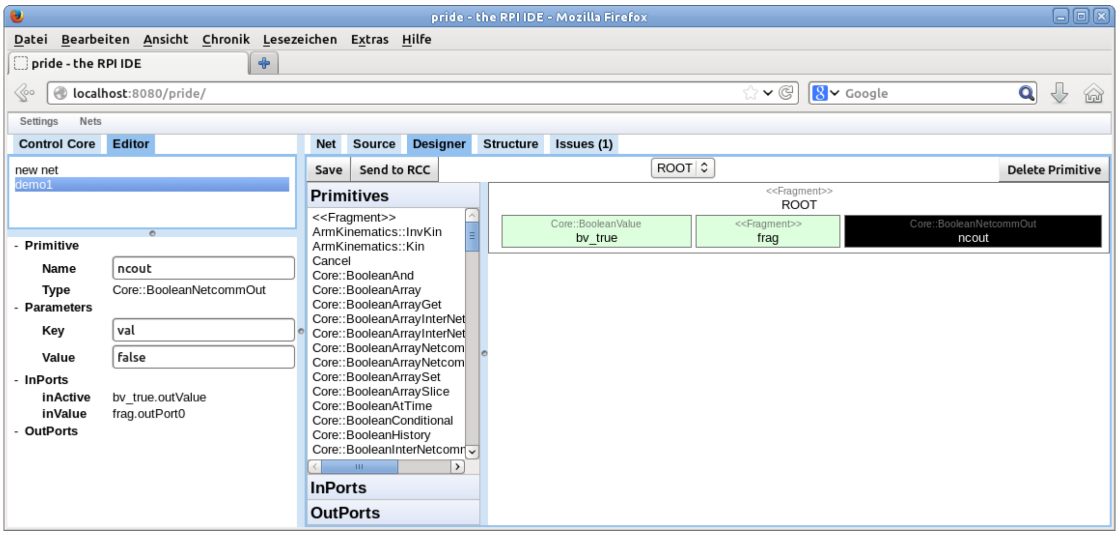
7.7.2. prIDE - a web-based debugging environment

For more debugging support, prIDE ("RPI-IDE", an integrated development environment for primitive nets) is a web-based tool that allows graphical design of primitive nets, as well as accessing the debugging log of a primitive net. prIDE has been developed in Java using the Google Web Toolkit (GWT) [49]. The GWT compiler produces a set of HTML pages and JavaScript code which can be executed by most modern browsers (in particular Google Chrome and Mozilla Firefox). The SoftRobot RCC can serve all required files using the built-in web server.

Designing primitive nets

Developers can hand-craft primitive nets using prIDE. Figure 7.8 shows a screenshot of the primitive net editor. The primitive net currently being edited is the same primitive net as shown in Fig. 7.6. It is possible to add primitives from a list, to configure all parameters and to connect input ports to output ports of other primitives. Primitives are automatically ordered from left to right, according to the data-flow among them (*Pre* primitives are recognized). If a primitive is selected in the editor, all parameters are displayed and primitives which are connected with the selected primitive are highlighted. In Fig. 7.8, the primitive *ncout* of type *Core::BooleanNetcommOut* has been selected, and in this example its input ports are connected to both other primitives. Fragments are displayed like normal primitives and can be entered by selecting the fragment from a dropdown box.

After the design of the primitive net is finished, it can be transferred to the RCC using the XML net representation (cf. Section 7.6.2), and subsequently executed. It is also possible to retrieve the XML representation of the primitive net from prIDE, and the XML for existing primitive nets can be parsed by prIDE and displayed in the editor. prIDE is also connected to the running RCC and can retrieve all primitive nets that are currently available. The RCC provides the XML representation of every primitive net (even if it has been created using the DirectIO protocol), thus prIDE can display the structure of every primitive net on the RCC. Primitive nets retrieved from the RCC can



- Primitive

Name	ncout
Type	Core::BooleanNetcommOut

- Parameters

Key	val
Value	false

- InPorts

inActive	bv_true.outValue
inValue	frag.outPort0

- OutPorts

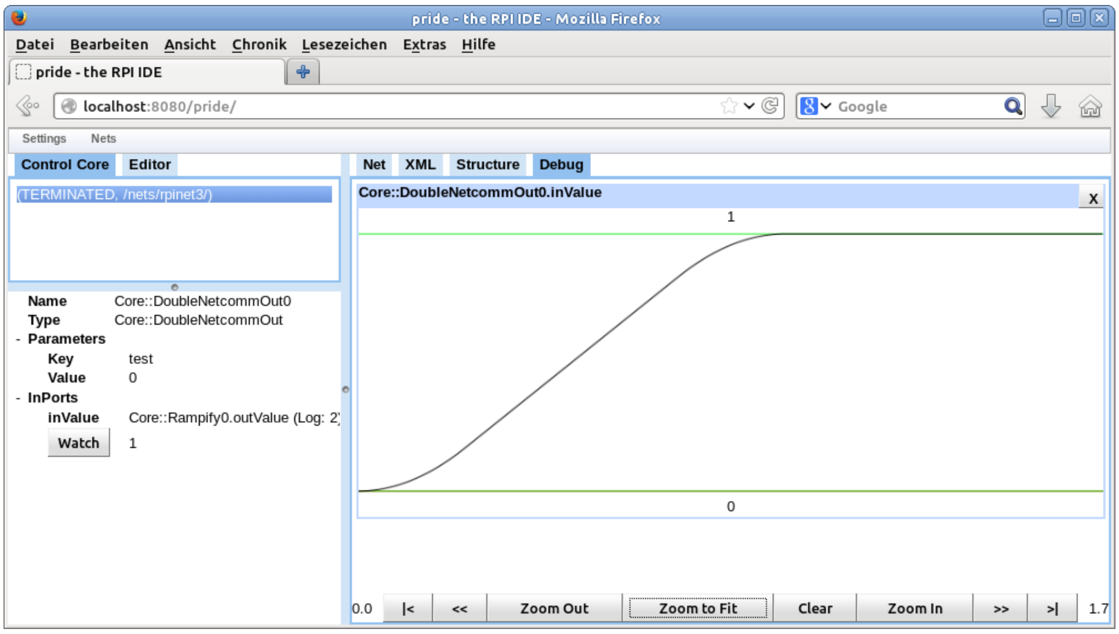
Figure 7.8.: prIDE primitive net designer

also be modified using the editor and a copy transmitted to the RCC again as a new primitive net.

Debugging primitive nets

Besides the capability to design new primitive nets, prIDE is also able to help with debugging existing primitive nets. As previously mentioned, existing nets can be parsed and their structure displayed. prIDE offers a view for examining the structure of a primitive net which is very similar to the editor, but lacks all functionality for adding or modifying primitives and thus has more space available for displaying large primitive nets. Many primitive nets contain large structures of primitives which form Boolean expressions or mathematical formulas. prIDE can automatically hide those calculation primitives from view and display a more readable infix notation for input ports of primitives which are connected to those structures. Primitives providing constant values ($TValue$) are also hidden. For example, if a primitive A is connected to a $BooleanAnd$ primitive, which on its part is connected to a $BooleanValue$ primitive with value *true* and some other B, prIDE will display “B.outPort && *true*” for A’s input port.

To hide basic primitives, prIDE contains a list of all known calculation primitives. During the graphical rendering, all primitives contained in the list are simply suppressed. If the user selects a non-basic primitive which has a hidden primitive connected to its input port, a textual infix representation for the connected primitive is requested. For each primitive a pretty printer is available, e.g. the $BooleanAnd$ primitive pretty prints itself as “$inValueA$ && $inValueB$”. For all primitives connected to input ports of the hidden primitive, the same algorithm is applied recursively. If a non basic primitive is found, no pretty printer is available and a textual representation in the form “*source-primitive.output-port*” is


pride - the RPI IDE - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
pride - the RPI IDE
localhost:8080/pride/
Google
Settings Nets
Control Core Editor
Net XML Structure Debug
(TERMINATED, /nets/rpinet3/)
Core::DoubleNetcommOut0.inValue
1
0
Watch
0.0 |< << Zoom Out Zoom to Fit Clear Zoom In >> >| 1.7

Name	Core::DoubleNetcommOut0
Type	Core::DoubleNetcommOut
- Parameters	
Key	test
Value	0
- InPorts	
inValue	Core::Rampify0.outValue (Log: 2)
Watch	1

Figure 7.9.: prIDE plotting a debug value from a trajectory generator. A single joint is moved from position 0 to position 1. X-axis: time, Y-axis: position

rendered. Since *Pre* primitives are not considered basic primitives and no unguarded cycles may exist in a primitive net, this algorithm is bound to terminate.

prIDE is able to display values of input ports of primitives for which debugging has been enabled. Values are continuously read from the RCC and stored in the browser's memory which enables the user to step through all cycles of a primitive net. Unlike the real-time implementation of the RCC, allocating more memory is no issue within a web application (as long as the browser has access to enough memory). Besides raw values, prIDE can also plot a time/value graph for integer and floating point links. Figure 7.9 shows an exemplary time/position graph for a primitive net which generates a trajectory with constant acceleration, constant velocity and constant deceleration phases (cf. Section 5.4.1).

Chapter 8

Synchronized execution of multiple real-time primitive nets

The SoftRobot RCC supports seamless switching from one set of primitive nets to another set of different primitive nets using synchronization rules as introduced in Chapter 6. This mechanism allows the robotics application to maintain control of the overall program flow while transitions between small (real-time) tasks can still be performed real-time safely.

8.1. Specification of synchronization rules

According to Section 6.1, a synchronization rule σ is defined as a 4-tuple $(C, \omega, \psi, \alpha)$ with the synchronization condition (C) and the sets of primitive nets to terminate (ω), to cancel (ψ) and to start (α). The synchronization condition is a propositional logic expression using Boolean variables which are provided by each primitive net to describe the internal state of the net. Using the SoftRobot RCC, these variables are provided by re-using the Boolean net communication primitive *BooleanNetcommOut* (cf. Section 7.6.1). This primitive is perfectly suited as it has a Boolean input port and supports adding a name to the value.

Before a synchronization rule can be specified, all involved primitive nets must have been loaded on the RCC. The synchronization rule can then be specified by the robotics application by posting the synchronization rule to the appropriate address using the plain HTTP communication channel, or by using the `nesc` command (cf. Section 7.6.3) over DirectIO. The `nesc` command has four parameters identical to the 4-tuple σ. The synchronization condition is specified using a string, denoting the variables in the form *netname.variablename*. A combination of multiple variables is possible using the *and* (&), *or* (|) and *not* (!) operators as well as parentheses where necessary. The SoftRobot RCC uses a simple recursive descending parser generated by Coco/R [89] to parse the synchronization condition and to create an abstract syntax tree.

8.2. Evaluation and execution of synchronization rules

The Boolean variables for the synchronization condition are provided in primitive nets by using the Boolean communication primitives. Changing the value of such a variable can be considered identically to propagating new set-points to hardware devices, thus the communication primitives only update their internal data during the third phase of the primitive net execution (cf. Sections 5.3.2 and 7.5) when all actuators are updated. This ensures that the the Boolean variables can be read at any time and always represent the same state as the actuators, even if the next execution cycle of the primitive net has already been started.

Synchronization rules can only be triggered by the synchronization condition C becoming *true*. Therefore it is sufficient to attach a synchronization rule to all primitive nets that contribute at least one Boolean variable for C. Once a synchronization rule has been received, the synchronization condition is parsed and memory references to the Boolean variables are stored to allow for a fast evaluation of the condition. Every primitive net evaluates all relevant synchronization conditions after it has finished phase three (updating actuators). As long as the evaluation takes place, no other primitive net participating in this certain synchronization condition may enter phase three to guarantee a consistent state of the system and the synchronization condition.

If the synchronization condition evaluates to *true*, the RCC attempts to acquire all resources necessary to start the primitive net in α. Resources in use by primitive nets contained in ω are transferred to the newly started primitive nets, all other resources are locked if they are available. If any resource cannot be successfully acquired, the synchronization rule is discarded without any modifications to the system (i.e. no primitive net is terminated or started). If all resources are available, all running primitive nets in ω are terminated. If a primitive net has already entered phase one or two, the execution of these phases will be interrupted. This is possible, because modifications to the system may only be performed during phase 3, thus aborting a primitive net during phase 1 or 2 causes no undesired effect. All primitive nets contained in ψ are canceled (i.e. the *Cancel* primitive will have a *true* output port in the next execution cycle) and the primitive nets contained in α are started synchronously.

The *outTerminate* output port of a primitive net is checked after all synchronization conditions that are affected by this primitive net have been evaluated, thus conditions becoming *true* during the last execution cycle of a primitive net will be attended to. After the primitive net has terminated, the values of the Boolean variables are kept in memory to allow further evaluations of synchronization conditions triggered by other (still running) primitive nets. If the last active primitive net involved in a certain synchronization condition terminates and the condition did not become active, the corresponding synchronization rule can be discarded. After a primitive net has been terminated, the values of the Boolean variables can only change to *indeterminate* if any resource is re-used which cannot activate a synchronization rule (the change of a variable to *indeterminate* can at most change the value of a synchronization condition from *false* to *indeterminate*, but never to *true*, cf. Table 6.1).

8.3. Thread management for multiple synchronized primitive nets

Each primitive net is executed by a dedicated (real-time) thread on the (real-time) operating system, using the Orocos Real-Time Toolkit (RTT) [21] as an abstraction layer from operation system specifics. Threads are encapsulated in the *NetExecutor* class. Creating and starting new threads cannot be done real-time safe, thus for synchronization rules it is important that an appropriate thread for each primitive net to be started (set α) is available and ready once the synchronization rule is executed. An appropriate thread is defined by several requirements:

1. The thread must already be running before the synchronization rule is triggered.
2. The thread must not be allocated to another primitive net which may be executed simultaneously.
3. The thread must be able to execute a primitive net with the desired execution cycle time (the SoftRobot RCC allows to specify the required cycle time during the creation of a primitive net).
4. The thread must have the desired execution priority for the primitive net (the SoftRobot RCC allows two different priorities: real-time and non-real-time, the latter can be used e.g. for monitoring tasks).

In order to fulfill requirement number one, it can be necessary to start new threads during the creation of the primitive net which are idle until the net is eventually started by a synchronization rule.

The following additional requirements should be fulfilled whenever possible:

5. The thread executing a primitive net which controls a certain actuator should also be reused if a successive primitive net controls the same actuator. Different threads with the same frequency can be running with a slight phase-shift. Reusing the same thread reduces the impact created by this phase-shift which could lead at most to the delay of one cycle time while switching from one thread to another one.
6. The total number of threads should be as low as possible.

In particular to fulfill the latter two requirements, a planning of execution threads is required (i.e. it is not possible simply to allocate a new thread for each primitive net that is started in a synchronization rule). During the creation of a new primitive net, one certain thread is allocated for later execution, independent of which synchronization rule actually starts the primitive net. To determine possible reuse of threads, hardware resources controlled by the primitive nets are considered. Each primitive net provides a set of resources it requires (each primitive can specify the list of resources it controls). Because hardware resources are inherently mutually exclusive, two primitive nets sharing at least one resource can never be executed simultaneously and thus the same thread can safely be used for execution of both primitive nets (if all other parameters such as execution cycle and execution priority match).

To assign threads to a new primitive net, a table of threads, assigned primitive nets and commonly controlled resources is required. The set of commonly controlled resources is the intersection of resources required by all primitive nets assigned to a certain thread. This set describes the resources which allow the primitive nets to share the same thread (i.e. prohibiting the primitive nets from running simultaneously) and therefore this set cannot be empty.

Threads are assigned to primitive nets during the creation of the primitive net. Although each primitive net is initially assigned to a thread, the assigned thread may change any time prior to the start of the net, however at any time a suitable thread must be assigned. Algorithm 8.1 lists pseudo-code for the allocation algorithm used in the SoftRobot RCC. If the new primitive net does not require any resource, a new thread is created and not added to the thread table. Without any resources, it is impossible to decide whether two nets will be executed simultaneously or not.

If the new primitive net requires at least one resource, the thread table is searched for a thread where the intersection of the primitive net's required resources and the commonly required resources listed in the table is not empty, and all parameters (such as cycle time and priority) match. It is possible that more than one thread listed in the thread table matches these conditions, e.g. if the newly created primitive net controls multiple actuators which have been controlled independently by several primitive nets previously. In this case, any thread can be selected (Algorithm 8.1 always selects the first thread found). If no suitable thread is found, a new one is created and added to the thread table. The new primitive net can be added to the assigned primitive nets, and the commonly required resources are updated in the thread table to contain only those resource contained in the intersection of all required resource of all assigned primitive nets, including the new one. Using this method, the set of commonly required resources can shrink during the creation of a new primitive net, but not become empty.

Using this algorithm, only suitable threads are selected, i.e. all primitive nets assigned to a certain thread are never executed simultaneously due to resource conflicts. By using the resources as a criteria for thread selection it is also ensured that the same thread will be reused for different primitive nets controlling the same hardware device when possible. This is generally not possible if multiple devices are first controlled independently by multiple primitive nets and later by a single primitive net or vice versa, however even in this case at least one thread will be reused.

Algorithm 8.1 Allocation of a suitable thread for a new primitive net

```
netresources ← getNetResources(net)
if netresources = $\emptyset$ then
    thread ← allocateNewThread(getThreadParameters(net))
    addAssignedNet(thread, net)
    return thread
else
    foundThread ← null
    for all thread : getThreadTable() do
        if (getResources(thread) $\cap$ netresources $\neq \emptyset$) then
            if parameterMatch(thread, net) then
                foundThread ← thread
                break
            end if
        end if
    end for
    if foundThread = null then
        foundThread ← allocateNewThread(getThreadParameters(net))
        setResources(foundThread, netresources)
    end if
    commonResources ← getResources(foundThread) $\cap$ netresources
    setResources(foundThread, commonResources)
    addAssignedNet(foundThread, net)
    return foundThread
end if
```

Threads must also be managed if a primitive net terminates, or is removed without ever having been started. Algorithm 8.2 shows pseudo-code for these cases. At first, the primitive net must be removed from the list of assigned nets for the thread it was previously assigned to. If the list of assigned net now becomes empty, the thread can be stopped and removed from the thread table. Otherwise, the commonly required resources must be updated by creating the intersection of required resources for all remaining nets. Since the set of commonly required resources might increase due to this step, it is possible that the sets of commonly required resources for two (or more) different threads now have a non-empty intersection. This implies that those threads are now mutually exclusive, and at most one can be actively executing a primitive net. All other threads must be idle. It is now possible to move all assigned nets from the idle threads to either the non-idle thread, or to an arbitrary thread if all threads are idle. This reduces the amount of idling threads without potentially causing conflicts.

The described algorithm for allocating a thread to a primitive net intentionally does not use any information about synchronization rules, not even in the special case of primitive nets without resources. The threads are rather already assigned during the creation of a

Algorithm 8.2 Termination of a primitive net

```
thread $\leftarrow$ getThreadForNet(net)
removeAssignedNet(thread, net)
**if** getAssignedNets(thread) $= \emptyset$ **then**
    **delete** thread
**else**
    newresources $\leftarrow$ *infiniteSet*
    **for all** net : getAssignedNets(thread) **do**
        newresources $\leftarrow$ newresources $\cap$ getNetResources(net)
    **end for**
**end if**
setResources(thread, newresources)
**for all** otherthread : getThreadTable() **do**
    **if** getResources(otherthread) $\cap$ newresources $\neq \emptyset$ **then**
        reallocateNets(otherthread, thread)
        **delete** thread
        **break**
    **end if**
**end for**
```

primitive net, when no synchronization information is yet available. There are several reasons for this decision:

- It cannot be guaranteed that a thread from a primitive net O contained in ω in a synchronization rule is available for executing a primitive net A from α because:
  - The primitive net O may already have terminated, and another (independent) synchronization rule could have taken over the thread.
  - The primitive net O has never been started, thus there is no thread to take over.
- Synchronization rules should be independent. To circumvent the issues of the last item, it could be possible to check that there is no synchronization rule that takes the required thread away. However, this consideration is only true as long as no further synchronization rule is added, i.e. for each additional synchronization rule it would be necessary to check whether it influences any existing rule. If conflicts arise, either the new rule would have to be rejected, or old rules re-planned. Re-planning can be difficult, because all existing rules could be activated at any time during the (potentially time consuming) re-planning-process.

Example

To demonstrate Algorithms 8.1 and 8.2, the following example uses five primitive nets with different sets of required resources.

Net	**Required resources**
net_a	$\{\text{robot}_x\}$
net_b	$\{\text{robot}_y\}$
net_c	$\{\text{robot}_x, \text{robot}_y\}$
net_d	$\{\text{robot}_x\}$
net_e	$\{\text{robot}_y\}$

The primitive nets net_a and net_b each control a single robot independently. After both primitive nets have finished their task, they should be taken over by a single successor net_c which has to control both robots simultaneously. Finally control should by distributed to two independent primitive nets net_d and net_e again after net_c has finished its work.

After net_a has been created, the thread table contains the following information

Thread	**Assigned nets**	**Common resources**
T_A	$\{\text{net}_a\}$	$\{\text{robot}_x\}$

Since no entry has been in the table before, no appropriate thread could be located and a new one has been inserted, carrying all required resources from net_a as common resources. The creation of primitive net net_b leads to the insertion of a second row

Thread	**Assigned nets**	**Common resources**
T_A	$\{\text{net}_a\}$	$\{\text{robot}_x\}$
T_B	$\{\text{net}_b\}$	$\{\text{robot}_y\}$

Thread T_A could not be reused, since the intersection of required commands of T_A and net_b is empty. Therefore a new thread T_B has been started an inserted into the table. Net_c finally can use T_A or T_B, since both threads have one robot in common with the net. In this example, T_A has been chosen.

Thread	**Assigned nets**	**Common resources**
T_A	$\{\text{net}_a, \text{net}_c\}$	$\{\text{robot}_x\}$
T_B	$\{\text{net}_b\}$	$\{\text{robot}_y\}$

Once net_a terminates, it is removed from the list of assigned nets, and the common resources are recalculated based on the intersection of all remaining assigned nets

Thread	**Assigned nets**	**Common resources**
T_A	$\{\text{net}_c\}$	$\{\text{robot}_x, \text{robot}_y\}$
T_B	$\{\text{net}_b\}$	$\{\text{robot}_y\}$

It now can be seen that two threads both share the common resource robot_y, which implies that both threads are mutually exclusive. In this case only one thread can be currently running (in this example T_B). All other threads can be terminated after relocating their tasks to the single currently active thread.

Thread	**Assigned nets**	**Common resources**
T_B	$\{\text{net}_b, \text{net}_c\}$	$\{\text{robot}_y\}$

After net_b has terminated, only one primitive net is currently active.

Thread	**Assigned nets**	**Common resources**
T_B	$\{\text{net}_c\}$	$\{\text{robot}_x, \text{robot}_y\}$

The net$_d$ can be created and assigned to T_B, for net$_e$ however a new thread has to be started.

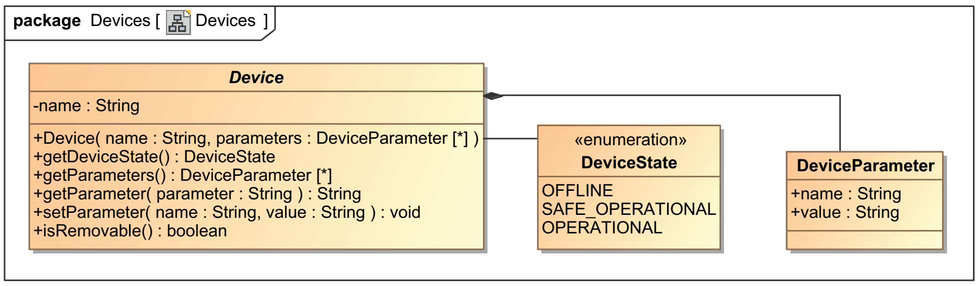
Thread	**Assigned nets**	**Common resources**
T_B	$\{\text{net}_c, \text{net}_d\}$	$\{\text{robot}_x\}$
T_C	$\{\text{net}_e\}$	$\{\text{robot}_y\}$

This example could be performed with the minimum number of threads possible, and threads have been reused where possible. The resource robot$_y$ was controlled by T_B both within net$_b$ and net$_c$. robot$_x$ had to switch from T_A to T_B during the transition from net$_a$ to net$_c$, however at least one robot necessarily had to switch since two threads had to be merged into a single one.

Chapter 9

Devices and drivers

The SoftRobot RCC reference implementation is not only capable of a real-time safe execution of primitive nets, but can also communicate reliably with external hardware devices such as robots, sensors or other periphery devices. The SoftRobot RCC needs a device specific driver for each piece of hardware it should control. In the context of this work, the term *device* describes a certain piece of hardware, e.g. one specific robot arm. The term *driver* describes the piece of software that is required to control hardware. For every device, a driver is required. Multiple instances of the same type of hardware require multiple devices but may be controlled by the same driver, and a driver may also support different types of devices which are closely related (e.g. robot arms with different payloads but the same hardware interface). For each device a class derived from base class *Device* (cf. UML class diagram in Fig. 9.1) is required. Every device must therefore provide some functionality common to all devices.



package Devices [Devices]
Device
-name : String
+Device(name : String, parameters : DeviceParameter [*])
+getDeviceState() : DeviceState
+getParameters() : DeviceParameter [*]
+getParameter(parameter : String) : String
+setParameter(name : String, value : String) : void
+isRemovable() : boolean
«enumeration»
DeviceState
OFFLINE
SAFE_OPERATIONAL
OPERATIONAL
DeviceParameter
+name : String
+value : String

Figure 9.1.: UML class diagram for devices

Every device has a *DeviceState* which can be one of the following:

Offline The device driver has been loaded, and a device instance is available at the RCC, but no communication with the hardware device is possible. This can occur for example if a device is connected over a network link and the network connection has been interrupted.

Safe operational The device is connected to the RCC and it is possible to retrieve data from the device (e.g. the current position of a robot). It is not possible to control the device. This state happens for example, if a robot has been successfully connected, but an emergency stop button has been pressed.

Operational The device is online and the RCC has full control over the device. This should be the most common state for any hardware device.

Devices can have a set of parameters which are stored as instances of class *DeviceParameter*. Such parameters can be used for a variety of different configuration tasks. Common configuration parameters specify the communication channel to the hardware, e.g. the network address, an EtherCAT slave ID, a CAN id, etc. Many articulated arm robots also have configuration parameters for their Denavit-Hartenberg parameters [108, p. 61] for use in their kinematics functions. All parameters can be set upon creation of a device, which is usually done either by using a configuration file on start-up of the RCC, or during the start-up of a robotics application. Some parameters (e.g. communication parameters) cannot be changed once a device has been created.

The `isRemovable()` method is used to determine whether a device instance may be removed from the RCC. Some devices may not support proper unloading and may thus block any attempt of removing themselves. If a driver in an extension library blocks unloading, this will also block any attempt to unload or replace this loadable extension once such a device has been created.

Devices can provide specific web pages which are served by the RCC's integrated web server (cf. Section 7.6.2). Device specific web pages usually contain hardware status information which may be of interest to the user such as device serial numbers, motor temperatures, etc. Drivers can provide hardware specific primitives that are required for the operation of the devices supported by the driver.

9.1. Device usage

The usage of devices is partially identical to the use of primitives. New instances of devices are also created using the factory pattern. Factories for all supported devices must be registered with the central Registry once an extension library is loaded. These factories provide support for creating and destroying instances of the device.

Extension modules can be loaded and unloaded at run-time. When such an extension is unloaded, it must be ensured that no currently active primitive net is using any primitive or device provided by this extension module. To prevent devices from being removed

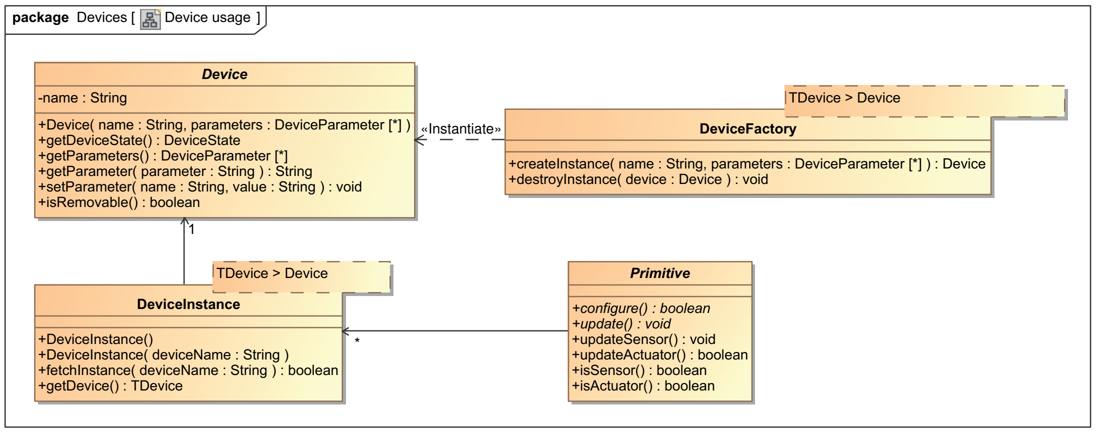

package Devices [Device usage]
Device
-name : String
+Device(name : String, parameters : DeviceParameter [*])
+getDeviceState() : DeviceState
+getParameters() : DeviceParameter [*]
+getParameter(parameter : String) : String
+setParameter(name : String, value : String) : void
+isRemovable() : boolean
«Instantiate»
TDevice > Device
DeviceFactory
+createInstance(name : String, parameters : DeviceParameter [*]) : Device
+destroyInstance(device : Device) : void
1
TDevice > Device
DeviceInstance
+DeviceInstance()
+DeviceInstance(deviceName : String)
+fetchInstance(deviceName : String) : boolean
+getDevice() : TDevice
*
Primitive
+configure() : boolean
+update() : void
+updateSensor() : void
+updateActuator() : boolean
+isSensor() : boolean
+isActuator() : boolean

Figure 9.2.: Usage of devices in RCC

while they are in use, every usage of a device must be registered. To facilitate this task, the template class *DeviceInstance* is provided. To access a device, an instance of the DeviceInstance class must be bound to the appropriate type and the desired name of the device must be specified either using the constructor or the `fetchInstance()` method. This automatically looks up whether a device with the given name exists, whether it has the right type and also locks this device from being removed. Using the `getDevice()` method the device can be accessed. It is guaranteed that a device will not disappear as long as at least one *DeviceInstance* of the device exists. In its destructor, *DeviceInstance* unlocks the device such that the developer does not need to care about releasing used devices.

9.2. Device interfaces

Many devices need drivers which are tailored exactly for supporting this device. For example, it is generally not possible to control robots of two different manufacturers with the same driver; sometimes even different robot series from the same manufacturer need different drivers. However, all those robots still share a lot of common features. Device interfaces have been introduced to support different hardware devices supporting a common feature set. Using these device interfaces, no changes in the robotics application are required to change a driver implementation in the RCC, as long as the same device interfaces are still supported.

But not only robotics applications can profit from device interfaces. Often hardware drivers need to rely on other hardware devices to perform their task. Many systems are connected to the robot controller using a fieldbus, e.g. EtherCAT or CAN. There are generic device interfaces for EtherCAT or CAN, which are independent of the used communication hardware, thus it is possible to use the same driver with different hardware

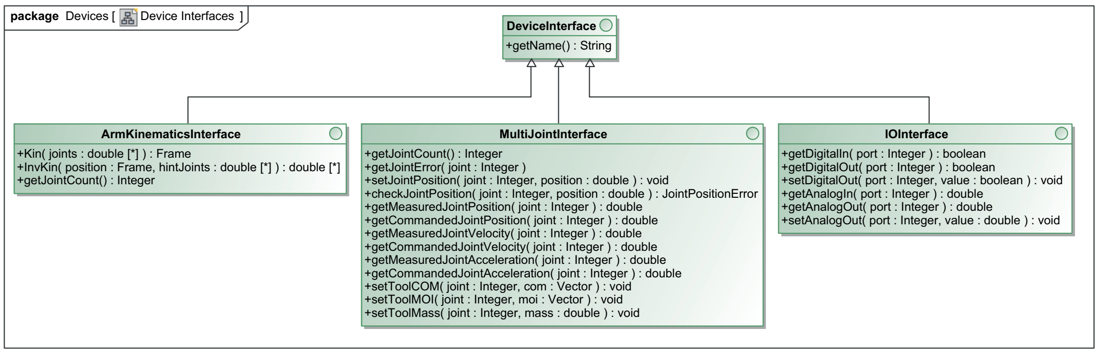

package Devices [Device Interfaces]
DeviceInterface
+getName() : String
ArmKinematicsInterface
+Kin(joints : double [*]) : Frame
+InvKin(position : Frame, hintJoints : double [*]) : double [*]
+getJointCount() : Integer
MultiJointInterface
+getJointCount() : Integer
+getJointError(joint : Integer)
+setJointPosition(joint : Integer, position : double) : void
+checkJointPosition(joint : Integer, position : double) : JointPositionError
+getMeasuredJointPosition(joint : Integer) : double
+getCommandedJointPosition(joint : Integer) : double
+getMeasuredJointVelocity(joint : Integer) : double
+getCommandedJointVelocity(joint : Integer) : double
+getMeasuredJointAcceleration(joint : Integer) : double
+getCommandedJointAcceleration(joint : Integer) : double
+setToolCOM(joint : Integer, com : Vector) : void
+setToolMOI(joint : Integer, moi : Vector) : void
+setToolMass(joint : Integer, mass : double) : void
IOInterface
+getDigitalIn(port : Integer) : boolean
+getDigitalOut(port : Integer) : boolean
+setDigitalOut(port : Integer, value : boolean) : void
+getAnalogIn(port : Integer) : double
+getAnalogOut(port : Integer) : double
+setAnalogOut(port : Integer, value : double) : void

Figure 9.3.: Some commonly used device interfaces

interfaces. Some commonly used device interfaces are shown as a UML class diagram in Fig. 9.3.

Robotics applications can access hardware devices independently of the underlying driver by using primitives that are provided alongside with device interfaces. These primitives work transparently with all hardware devices that implement the given device interface. Device interface furthermore can also provide web pages which are attached to every device implementing the interface. Such web pages display information about the underlying hardware devices in a standardized form, e.g. all robot arms can display their current joint angles with the *MultiJointInterface*.

ArmKinematicsInterface

The *ArmKinematicsInterface* provides methods for calculating the direct and inverse kinematics function for a robot. The direct kinematics function takes a list of all joint values for a robot (this can be both rotational as well as translational joints, depending on the robot hardware), and returns a *Frame*. A frame describes both a position in space as well as the orientation. Therefore, the values X, Y and Z are used for the position, and Euler angles A, B and C are used for orientation. The reference coordinate system is always the base of the robot.

The inverse kinematics function takes a position given as *Frame* (relative to the robot's base), and calculates joint values to reach the given position. Because the inverse kinematics function usually is not unique (for a 6-DOF robot it yields up to 8 solutions), a list of hint joints must also be given to the method. Hint joints are joint values known to be near the desired solution, such that the inverse kinematics function can chose the closest solution. When points are programmed into the program, hint joints can be saved from the current position of the robot. Even if the measured point is later moved manually, the hint joint usually will still be close enough to prevent "wrong" solutions from being chosen.

Input ports	inJoints: Array<double>
Output ports	outFrame: Frame
Parameters	Robot: String

Table 9.1.: The *Kin* primitive

Input ports	inFrame: Frame inHintJoints: Array<double>
Output ports	outJoints: Array<double>
Parameters	Robot: String

Table 9.2.: The *InvKin* primitive

The *ArmKinematicsInterface* provides a web page which can be used by the robotics application to query the kinematics function of a robot. This avoids the need for duplicating the kinematics functions for each robot once in the RCC and once again in the application. For applications that need real-time access to the kinematic functions, two primitives are provided.

The *Kin* and *InvKin* primitives (cf. Tables 9.1 and 9.2) provide access to the kinematics and inverse kinematics function for the robot specified with parameter *Robot*. The implementation of the respective functions is done in the device driver implementation of the concrete robot. Cartesian positions are represented as data type *Frame*, and joint values as arrays of type double. The primitives expect arrays of appropriate size for the given robot, i.e. a robot with 6 joints needs 6 values for *inJoints* and *inHintJoints*. The primitives will always return a single solution for the inverse kinematics function, because for a generic robot it is not known how may solutions are available, even an infinite amount of solutions may be possible (e.g. for the KUKA LWR with 7 joints). If no solution can be found (e.g. because the specified frame is outside the working area of the robot), not-a-number (NaN) values will be returned.

MultiJointInterface

The *MultiJointInterface* is a generic interface to multi-joint kinematics (e.g. articulated arms, portal systems, . . .). Using the `setJointPosition(...)` method it allows to define new set-points for the actuator. Such a set-point must be reachable by the actuator within one execution cycle. The planning of the desired trajectory is up to the user and not performed by the device. The driver may however perform some interpolation if necessary for smooth hardware control. Using the `checkJointPosition(...)` method it is possible to check whether any given joint position is valid. Invalid positions are usually out of the range the joint can reach.

Using the *MultiJointInterface*, it is possible to query the current joint position, velocity and acceleration of an actuator. Two different values are available: measured and com-

Input ports	none
Output ports	outCmdPos: double outMsrPos: double outCmdVel: double outMsrVel: double outCmdAcc: double outMsrAcc: double outError: int
Parameters	Robot: String

Table 9.3.: The *JointMonitor* primitive

Input ports	inPosition: double
Output ports	outErrorConcurrentAccess: Boolean outErrorJointFailed: Boolean outErrorIllegalPosition: Boolean outError: int
Parameters	Robot: String Axis: int

Table 9.4.: The *JointPosition* primitive

manded. Measured values are based on values retrieved from sensors built in the actuator. Commanded values reflect the values sent to the device using the `setJointPosition(...)` method. Some devices may not have sensors for all values. For example, if a device only has a sensor for the current position but not for velocity and acceleration, the device driver must derive these values internally.

Besides joint position, there are also tool related methods in the *MultiJointInterface*. Many hardware controllers need to know the mass of any attached tool in order to control the hardware correctly. Therefore, using `setToolMass(...)` it is possible to define the mass currently attached to the actuator. Using `setToolCOM(...)` the center of mass can be defined, and using `setToolMOI(...)` the moment of inertia. All those methods also take a joint as argument, because some robot systems support tools not only being attached to their end-effector, but also to other parts of the kinematic chain (e.g. many KUKA robots support an additional payload mounted on joint 3 for welding hardware). Some other hardware might not need additional load data and thus simply ignore calls to these methods.

There are three primitives available which use the *MultiJointInterface*. All primitives have parameters for configuring the robot and joint. Control is always done on a per joint level, i.e. each joint needs its own set of primitives for reading the current position or for commanding a new position. The *JointMonitor* primitive (cf. Table 9.3) provides information about a joint with the current position, velocity and acceleration (both

Input ports	inMass: double inCOM: Vector inMOI: Vector
Output ports	outCompleted: Boolean outError: int
Parameters	Robot: String Axis: int

Table 9.5.: The *ToolParameters* primitive

measured and commanded) and an error number in case an error has occurred with the joint (e.g. the joint motor is unpowered). The *JointPosition* primitive (cf. Table 9.4) accepts new set-points for the configured joint. Only one primitive may actively control any specific joint of a robot. Multiple primitives with the same configuration may exist within the same primitive net, however only one of those primitives may be supplied with new set-points during any given primitive net execution cycle. The primitive has several output ports for signaling errors. Boolean typed output ports are available for the most common errors (concurrent access: more than one primitive has tried to actively control the joint; joint failed: the joint has failed for some reason, e.g. it was not powered; illegal position: the commanded position is outside of the valid range of joint positions) and an integer typed output port for other errors, which may be specific to the concrete robot driver. The *ToolParameters* primitive finally allows to configure the payload attached to a robot joint using its input ports *inMass*, *inCOM* (center of mass) and *inMOI* (moment of inertia). The new payload is configured with the attached robot as soon as new values are made available to the primitive. Because switching the tool configuration takes time for some robot systems, the primitive provides an output port for signaling the completion of the switching process. The primitive also has an error output port which may be device specific.

IOInterface

The *IOInterface* provides generic support for digital and analog inputs and outputs. Regarding the *IOInterface*, digital inputs and outputs can represent a Boolean value and analog inputs and outputs a floating point value. The mapping of these values to a concrete hardware interface must be performed by the driver. Digital ports can be provided for example by bus terminals which can switch a voltage supply on and off, but could also be integrated directly into a gripping system for controlling opening and closing requests.

The *IOInterface* provides methods for reading inputs as well as reading and writing outputs. Each device may support multiple ports, thus a port number must be specified. The numbering of ports is up to the driver, usually each type of port starts counting from zero.

Input ports	inValue: Boolean
Output ports	none
Parameters	Device: String Port: int

Table 9.6.: The *IO::OutBool* primitive

```
1 < a=gde()
2 > a=da({LbrLeft:{interfaces:{armkinematics:{},io:{},multijoint:{}},type:"kuka_lwr_fri
    "},LbrRight:{interfaces:{armkinematics:{},io:{},multijoint:{}},type:"kuka_lwr_fri
    "},rcc:{interfaces:{},type:"rcc"}})
3 > a=ds({LbrLeft:"off",LbrRight:"off",rcc:"op"})
4 > a=ok()
```

Listing 9.1: Example DirectIO communication for device status

The *IOInterface* provides six primitives to control digital and analog I/O. Table 9.6 exemplarily shows the *IO::OutBool* primitive which allows the primitive net to write to a digital output. It has one input port which carries the value the output should be set to and requires configuration for the device which controls the I/O hardware, and a port number (in case the device supports more than one input or output). Primitives are available for digital and analog inputs and outputs. Inputs can be read, outputs can be read and written using the appropriate primitives, thus three primitives are required for analog and three for digital I/O.

9.3. Communication with applications

Robotics applications often need information about devices available at a certain Robot Control Core. The SoftRobot RCC provides two methods for robotics applications to receive this information. The first method provides a web handler that lists all devices, their state and the device interfaces they implement under the URI `/devices/`. Each device and device interface may provide additional web handlers to provide more detailed, device specific information for the application. The *MultiJointInterface* for example provides the current measured and commanded joint positions, and the *ArmKinematicsInterface* offers both direct and inverse kinematics calculations on a web page.

It is also possible to receive device information using the DirectIO protocol. Because this protocol cannot be extended as easy as the web server, information on this protocol is restricted to generic device status information such as new devices, changes in device states and removed devices.

Listing 9.1 shows an example for requesting device status using DirectIO. The handshake has been omitted. In line 1, the robotics application requests device information using the `gde` command. In line 2, the RCC responds with the device added (`da`) message, containing a map from device name to another map containing two entries *type* and *interfaces*. The type of the device is transmitted as string, the device interfaces the device implements are again a map from the device interface to a map of additional parameters. In Listing 9.1 two devices of type `kuka_lwr_fri` are available as “LbrLeft” and “LbrRight”. Both implement the device interfaces *armkinematics*, *io* and *multijoint* which all do not provide further details (further details would be a device interface specific map from string to string). A third device “rcc” of type `rcc` does not implement any device interfaces. This is a virtual device for configuring the RCC itself.

In line 3, the device status (`ds`) message reports both *LbrLeft* and *LbrRight* as offline (i.e. the robots are not connected to the RCC) and the virtual RCC device as operational. Line 4 finally acknowledges the `gde` command. Further updates of devices will be transmitted asynchronously to the robotics application. For instance if a new device is added, a further `da` message will be sent. If one of the robots connects to the RCC, another `ds` message with the new state (safe-operational or operational) will be transmitted.

9.4. Hardware device drivers

Device drivers for a set of different robotics hardware have been developed and integrated into the SoftRobot RCC. Appendix B provides an overview of hardware devices currently supported by the SoftRobot RCC. The following sections provide in-depth descriptions of the implementations for some of the devices.

In order to control different manipulators using different communication technologies, some form of time compensation for different cycle times is necessary. The solutions of the SoftRobot RCC for this problem are explained with the Stäubli TX90L robot as an example in Section 9.4.3. Some robots offer functionalities which are well beyond the capabilities of traditional robots, e.g. integrated force and torque sensors or impedance controllers. Section 9.4.4 introduces the KUKA lightweight robot and the integration of this device with all specific functionality into the SoftRobot RCC.

9.4.1. The EtherCAT fieldbus

EtherCAT is a fieldbus which was developed by Beckhoff and uses standard Ethernet frames as defined in IEEE 802.3 Ethernet protocol [62] with 100 MBit/s bitrate for communication. As physical media, standard copper Ethernet cables can be used. If the distance between two nodes is more than 100 m, optical fiber cables can also be used. The EtherCAT fieldbus is designed as a master/slave system, i.e. there is one master and multiple slave devices connected to the bus. The master device is responsible for managing the overall bus operations, e.g. configuring slaves or initiating process data transmissions. The master can be equipped with standard issue network interface hardware, while slaves use specialized EtherCAT interface hardware (FPGA[1] or ASIC[2]) [100], which allows for a very fast and real-time safe communication.

An EtherCAT fieldbus is logically organized as a ring. The master transmits an Ethernet frame which is received and forwarded by every slave and eventually is returned to the master. Physically, a bus topology is very commonly used, although other topologies such as a star are also possible. EtherCAT slaves usually have at least two connection ports. If both ports are connected, frames received on one port will immediately be forwarded to the other port. If the second port is not connected, the frame will be reflected to the first port. Thus a frame will pass all slaves on the bus before it is sent back by the last slave in the line and again passes all slaves until the master is reached. Because EtherCAT uses fully bidirectional Ethernet connections (i.e. two different wire pairs are used for transmitting and receiving data), no collisions between frames can occur.

EtherCAT slaves read and write data to Ethernet frames while they pass the device. Frames need not be received and stored completely before the data can be processed, because all necessary modifications of the frame are performed "on-the-fly" by the dedicated EtherCAT slave chips. This allows for very short latencies in slaves and thus for very short cycle times for the whole bus system. Typical cycle times can be as low as 50 µs [103].

EtherCAT slaves can have four different states: *Init*, Pre-Operational (*Pre-Op*), Safe-Operational (*Safe-Op*) and Operational (*Op*) [35]. After being powered on, all slaves are in state *Init*. After the fieldbus has been initialized by the master, all slaves enter state *Pre-Op*. In this state, communication between the master and slaves is possible using the mailbox protocol, e.g. for configuring the slaves. The mailbox protocol allows the master to send messages to slaves and to receive their answers, however no cyclic process data exchange is possible. One important configuration item for slaves can be the configuration of the process image, i.e. which data (and in which layout) needs to be transmitted cyclically between the master and the slave. Not all slaves support this configuration; some slaves have a fixed, predefined process image (e.g. the simple digital input and output bus terminals from Beckhoff always map their inputs and outputs to consecutive bits in the process image). After all slaves have been configured, they enter the state *Safe-Op*. In this state, cyclic exchange of process data between the master and the slaves starts. The master may read current sensor values from slaves, however slaves must remain in a safe state, i.e. generally they may not perform any action such as switching outputs or moving an actuator. As last step, the master requests a state change of all slaves to state *Op*. In this state, uninterrupted cyclic exchange of process data takes place, and slaves are fully operational and controllable.

A device interface is used for the EtherCAT master implementation in the SoftRobot RCC. The device interface provides all necessary operations to interact with an EtherCAT bus, but abstracts from a concrete implementation. This allows for exchanging the underlying implementation at any time without the need for modifying device drivers

[1]Field Programmable Gate Array
[2]Application-specific integrated circuit

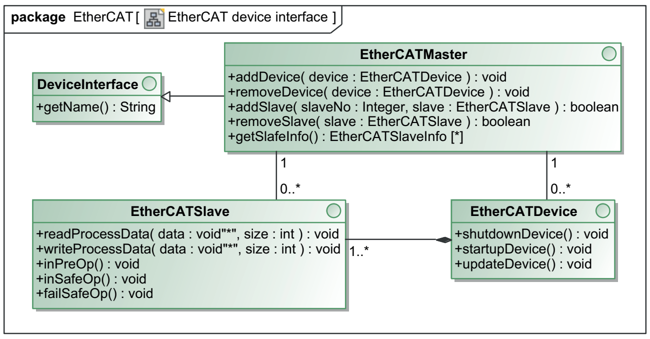

package EtherCAT [EtherCAT device interface]
DeviceInterface
+getName() : String
EtherCATMaster
+addDevice(device : EtherCATDevice) : void
+removeDevice(device : EtherCATDevice) : void
+addSlave(slaveNo : Integer, slave : EtherCATSlave) : boolean
+removeSlave(slave : EtherCATSlave) : boolean
+getSlafeInfo() : EtherCATSlaveInfo [*]
1
0..*
1
0..*
EtherCATSlave
+readProcessData(data : void"*", size : int) : void
+writeProcessData(data : void"*", size : int) : void
+inPreOp() : void
+inSafeOp() : void
+failSafeOp() : void
1..*
EtherCATDevice
+shutdownDevice() : void
+startupDevice() : void
+updateDevice() : void

Figure 9.4.: UML class diagram of the EtherCAT device interface

which depend on the EtherCAT bus. The EtherCAT master is provided by a slightly modified version of the Simple Open EtherCAT master (SOEM) [74] project, which has been made compatible with the real-time network stack provided by the RTnet project [75]. A real-time network stack is necessary to avoid breaking real-time guarantees by accessing non real-time functionality of the operating system for network access. SOEM can use any network interface card (NIC) supported by the operating system for interfacing with the EtherCAT bus, however the use of RTnet limits the number of real-time capable NICs.

Figure 9.4 shows a UML class diagram of the device interface for the EtherCAT bus. The interface *EtherCATMaster* provides all functionality for device drivers which use the EtherCAT fieldbus. The SoftRobot RCC EtherCAT device interface uses two concepts for representing hardware devices connected to the fieldbus: *EtherCAT devices* and *EtherCAT slaves*. An EtherCAT device consists of one or more EtherCAT slaves. On the hardware level, there are only slaves connected to the bus, however some hardware devices may be built from multiple slaves (e.g. the KUKA youBot robot arm has one EtherCAT slave controller for each joint). Although technically every slave can be controlled completely independent of all other slaves, synchronization of the slaves logically belonging to the same device is desirable. Therefore the interfaces *EtherCATSlave* and *EtherCATDevice* are designed according to the observer pattern [45], such that the driver for a single slave as well as the driver for the whole device can be notified of important conditions of the EtherCAT bus.

Figure 9.5 shows a UML sequence diagram for the life-cycle of an exemplary EtherCAT device which uses a single slave on the EtherCAT bus. The life-cycle of an EtherCAT device consists of three phases. In the first phase, the device is initialized and registers itself with the EtherCAT master (step 1). If a device has a fixed set of controlled slaves, instances of the slave controllers can also be created in this phase (step 2).

Whenever the EtherCAT bus is ready for startup (i.e. after starting the RCC or after a modification has been made to the physical layout of the EtherCAT bus, e.g. by plugging in a new device), the `startupDevice()` method is called for all devices (step 3).

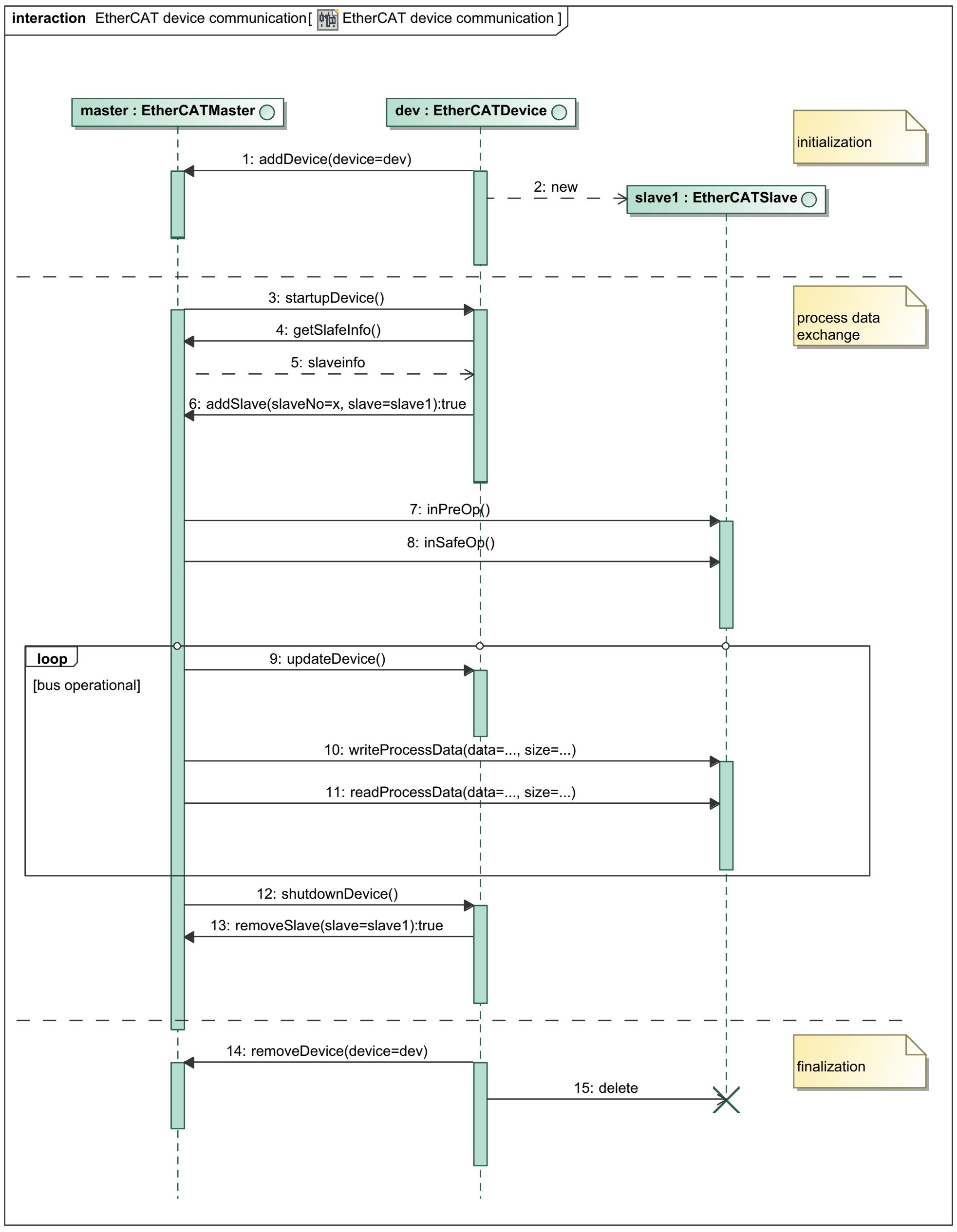


Figure 9.5.: UML interaction diagram with the life-cycle of an EtherCAT device using a single slave

Within this method, the EtherCAT devices may request a list of all slaves attached on the bus (steps 4 and 5). This list contains the slave numbers, manufacturer and device identifiers and possibly also a plain text name of a device. Using this information, the device can register its slaves with the master (step 6). This way, the master can be completely independent from any device implementation. Devices also can have configuration information to identify the appropriate slave if multiple devices of the same type are connected to the bus. After all devices have registered their slaves, the EtherCAT master requests the *PreOp* state for all hardware slaves and notifies the respective slave drivers when the *PreOp* state has been reached by calling the *inPreOp()* method (step 7). During this method, slaves can perform any initialization which is necessary for their hardware during *PreOp*. After all slaves have finished, the master requests state *SafeOp* and again notifies all slaves (step 8).

After one process data cycle has been completed while being in the *SafeOp* state, all EtherCAT slaves are requested to enter state *Op*. During this state, process data is cyclically exchanged between the master and all slaves. At the beginning of each process data cycle, all devices are notified of an upcoming exchange cycle (step 9). This allows devices to synchronize data among all their slaves, e.g. a consistent set of set-points can be aggregated for all joints of a robot. After all devices have been notified, all slave instances are requested to write their current process image (step 10). To write process data, an untyped pointer to a memory location (`void*`) together with the size of this memory is provided to each slave. The layout of the process data is up to each individual slave and must match the expectations of the used hardware components. After all slaves have written their process data to the memory, the master transmits the data of all slaves physically to the bus. After all hardware slaves have processed the process data, it is returned to the master, which again calls all slave instances to extract their process data (step 11). Again, an untyped memory pointer is used, such that every slave implementation can implement the appropriate memory layout. Although it may seem a little counter intuitive that the drivers must write their process image before they can read values, however this process is mandated by the design of the EtherCAT bus which requires all process data frames to originate from the master and be returned afterwards.

Sometimes it is necessary to shut down the EtherCAT bus. This happens for instance, if a new device driver is registered or the physical structure of the bus is altered (one or more devices are added or removed). All hardware slaves are brought into state *PreOp*, and each device is notified by calling the `shutdownDevice()` method (step 12). Devices should remove all their slaves from the master (step 13) during this method. After the bus is ready again, the system continues again from step 3. If an EtherCAT device should be completely removed, this can be done once the bus has been shut down (steps 14 and 15).

The EtherCAT bus provides a clock synchronization mechanism using distributed clocks. Although the process data travels through all slaves with minimal latencies, no two slaves will receive the process data at exactly the same time. For many applications however it is desirable to synchronize certain events, e.g. all inputs of a machine should be evaluated at the same time to generate a consistent view of the current state. To satisfy these requirements, slaves can implement a distributed clock feature to synchronize their clocks. The first slave in the bus which supports the distributed clocks feature is usually designated as reference clock, and all further slaves adjust their local clocks to the reference clock. To compensate for latencies between slaves, signal propagation times are measured during start up of the bus and incorporated into the adjustments of the local clocks. Using these mechanisms, all clocks can be synchronized with differences ≤ 1 µs [35].

Two interrupts can be defined (SYNC0 and SYNC1) which occur synchronously for all slaves at a defined time in relation to the process data cycle. Such an interrupt is used e.g. to synchronize the collection of all inputs prior to the process data telegram being processed by each slave. Those interrupts are triggered by the local clock of the respective slave which is synchronized to all other clocks using the distributed clock mechanism. The transmission of process data however is triggered by the master, thus the local clock of the master must also be synchronized to the distributed clock.

The SoftRobot RCC reference implementation automatically configures the first slave being capable of the distributed clock feature as reference clock and also synchronizes its process data cycle time to the reference clock of the bus. This synchronization is necessary because the Real-Time Clock (RTC) of the computer running the RCC and the reference clock in the first EtherCAT slave may run at slightly different frequencies, thus the generated SYNC0 and SYNC1 events and the process data transmissions would slightly drift apart. Synchronization is possible because each process data telegram also contains the current reference time of the EtherCAT bus (measured in ns since start of the distributed clock). Some care must be taken because some slaves use a 32-bit distributed clock which will already wrap after ≈ 4.3 s. The SoftRobot RCC detects such wraps and internally uses a 64-bit clock which supports a continuous operation of the fieldbus for ≈ 585 years.

9.4.2. Beckhoff I/O bus terminals

The company Beckhoff provides bus terminals for a broad variety of digital and analog I/O. Very common digital variants switch or detect +24 V power, while analog variants often work with voltages in a range of 0 V up to +10 V or with currents from 0 mA to 20 mA. The I/O bus terminals are a very basic example of devices using the EtherCAT fieldbus.

The Beckhoff I/O device driver implements both the *EtherCATDevice* as well as the *EtherCATSlave* interface. In the SoftRobot RCC reference implementation, one device has exactly one slave, each bus terminal is controlled by its own instance of the device driver. Although multiple bus terminals are connected to a single bus coupler, each bus terminal has its own embedded slave controller and therefore shows up as an independent slave on the bus. All specific bus terminal drivers (e.g. EL1008 for an 8 port digit input with 24 V or EL2008 for an 8 port digital output with 24 V) inherit from the abstract

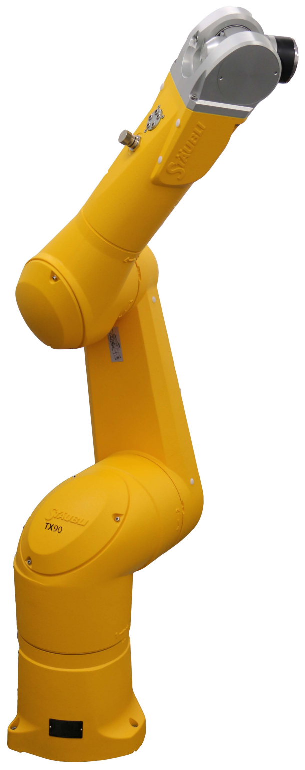


Figure 9.6.: Stäubli TX90L robot

class *AbstractBeckhoffIODevice* which provides implementations which are common for all Beckhoff bus terminals. The specific device drivers are responsible for mapping the *IOInterface* to the appropriate process image. For digital I/O the process image usually consists of one bit per input or output, e.g. the EL1008 bus terminal provides one byte of input data (8 bit, one for each of its 8 electrical output ports) and no output data. Analog I/O often use 12 or 16 bit wide integer representations which are mapped to their input/output range, thus no floating point values are required.

9.4.3. Stäubli TX90L

The Stäubli TX90L robot (cf. Fig. 9.6) is a 6-DOF industrial robot from the Swiss manufacturer Stäubli with a nominal payload of 7 kg. It is controlled by a Stäubli CS8C controller, which supports the VAL3 programming language as well as the uniVAL drive interface. Using the uniVAL interface, it is possible to completely integrate the robot into a machine tool, including motion control of the robot. Traditionally, robots are often only partly integrated into machines, because every robot needs its own programs, and motion control of each robot is only performed by its own controller. UniVAL allows to specify set-points for each joint of the robot at a high frequency using a real-time fieldbus, thus motion control can be performed outside the robot controller.

The uniVAL interface

The uniVAL interface [120] is based on the CANopen DS 402 drives profile [60, 61] which allows several different operation modes such as cyclic position, cyclic velocity or interpolated position. The Stäubli TX90 robot uses the cyclic position mode defined in DS 402, which requires the external motion controller to provide new position set-points for each joint every 2 ms or 4 ms (configurable). The uniVAL drive software internally calculates all necessary values for directly controlling the servo motors, including the calculations necessary for dynamics compensation (cf. Section 2.4). If the external motion controller does not provide new set-points in time, the robot is stopped. The SoftRobot RCC is connected to the uniVAL controller using the EtherCAT fieldbus.

The DS 402 profile was originally defined for the CANopen protocol [17], which is used on top of the CAN fieldbus [37]. Using CoE (CANopen over EtherCAT), it is possible to use EtherCAT as fieldbus technology with the CANopen protocol on top. CANopen uses two special communication objects: PDOs (process data objects) for cyclic data exchange, and SDOs (service data objects) for configuration. The object directory (OD) collects all communication objects of a device. Each communication object has a unique 16-bit long identifier and optionally an 8-bit long sub-index. The indexes for commonly used communication objects are assigned by the CANopen profiles (like the DS 402 profile for uniVAL), while other device specific communication objects have an index in the manufacturer specific range. Manufactures of CANopen devices usually provide an EDS (Electronic Data Sheet) file which describes the OD of the given device. The cyclic PDOs can contain a multitude of different data which is available in the OD (although not all objects from the OD may be mapped to PDOs). The layout of PDOs can be configured by defining which communication objects are to be mapped in which order into the process image. This configuration is performed by sending SDOs to dedicated communication objects during the *PreOp* phase. The uniVAL controller provides a single slave on the EtherCAT bus which has several communication objects for every joint. Usually all six joints are mapped into PDOs simultaneously. Besides support for controlling all joints, there are also communication objects available for digital and analog I/O. The CS8C controller supports a small number of digital I/O to be directly connected to the robot controller, and furthermore hardware status information is available as inputs (e.g. the state of the emergency stop buttons or the temperature of the controller). The SoftRobot RCC provides a driver implementation for the uniVAL protocol, including partial support of the DS 402 profile (partial, because uniVAL does not use all features of DS 402).

Timing of process data and real-time primitive net cycles

The Stäubli robot controller expects new set-points to be provided cyclically. Primitive nets calculate those set-points also cyclically, however some kind of synchronization between both is necessary. The primitive net must produce new set-points at least with the same frequency as the robot controller expects new set-points during the process data cycle, otherwise sometimes no new set-points would be available (which causes the robot controller to issue an emergency stop immediately). But it is not sufficient just providing any new set-points in each process data cycle, the joint position values and in particular the resulting joint velocities which are provided must also be steady (cf. Section 2.4). In case the primitive net is executed slightly faster than the EtherCAT process data cycle, at some point in time one value of the primitive net will be lost, resulting in a position jump for the robot. Thus it would be desirable if the primitive net execution would be synchronous to the process data cycle. One key feature of the SoftRobot architecture is the simultaneous support of multiple robot systems. Those robots are not necessarily all connected to the same EtherCAT bus (maybe they are even connected using a completely different technology) and thus might all have slightly different cycle frequencies. Although it might be possible to synchronize the clock of the RCC with one of the other clocks, it might not always be possible to synchronize all other clocks as well. Therefore, a mechanism to compensate for timing differences of the underlying robot controller and the RCC itself is required.

In each process data cycle, the position set-point for each joint is calculated using the following equation

$$j = \underbrace{j_l}_{\text{old position}} + \underbrace{\frac{j_c - j_l}{t_c - t_l}}_{\text{velocity}} \cdot \underbrace{(t - t_l)}_{\text{estimated time}} \tag{9.1}$$

In each process data cycle, the set-point j_c which has been provided by the primitive net most recently at time t_c is known, as well as the respective values from the last process data cycle j_l and t_l. Using those values, the desired velocity of the primitive net can be calculated. Using the current time t and t_l it is possible to interpolate the joint position j for the current time. In order to get good interpolation results, the primitive net must produce new set points with at least the same frequency as the process data is calculated. If the value of $|t - t_c|$ grows large (i.e. more than 5 to 10 times the cycle time), it must be assumed that no primitive net is currently controlling the robot. In case the robot is still moving, the driver must decelerate all joints and bring the robot to standstill.

The uniVAL system requires new set-points to be calculated for the next SYNC0 event (cf. Section 9.4.1). The new set-point will be written to the fieldbus during the process data cycle, but the robot controller will only interpret the new set-point once the SYNC0 event occurs. Thus the time t in Eq. (9.1) should not be the time of the process data cycle, but the time of the next SYNC0 event. The EtherCAT driver provides a method to request the estimated time of the next SYNC0 event. This time is calculated from the time value of the distributed clock from the last process data cycle and the configuration of the SYNC0 event.

Although the process data cycle is synchronized to the distributed clock of the EtherCAT bus, some jitter will occur which also affects the time t of the next SYNC0 event. This jitter also influences the estimated time in Eq. (9.1) and leads to jerky position values. Because no sudden changes in the process data cycle times are expected, the value t can be smoothed over time using the sliding average of the last process data cycle times.

$$\hat{t}_x = \hat{t}_{x-1} + \frac{1}{n}\sum_{i=0}^{n-1}\left(t_{x-i} - t_{(x-1)-i}\right) = \hat{t}_{x-1} + \frac{t_x - t_{x-n}}{n} \tag{9.2}$$

The time $\hat{t}_x$ is the estimated time of the current process data cycle x, which is calculated from the estimated time of the last process data cycle $\hat{t}_{x-1}$ and the sliding average of the last n cycle times (absolute measured times $t_x, t_{x-1}, \ldots$). For the first n cycles, the original cycle times must be used until a smoothed value is available. As soon as the communication with the robot has been established and the robot has entered state *Op*, set-points need to be provided continuously, even if no primitive net is currently running. In this case, constant values are provided. Since the sliding average of the process data cycle times can be calculated beginning with the first process data cycle, the first n process cycles usually will pass with degraded performance before the first primitive net is started. Tests have shown that smoothing the time over 50 process cycles yields good results for the Stäubli TX90L robot. It should be noted that the sliding average is only calculated for the cycle time, but not for the desired joint position. Thus new set-points provided by primitive nets will not be delayed.

Driver architecture

The Stäubli TX90L driver implements the *MultiJointInterface*, the *ArmKinematicsInterface* and also the *IOInterface* (cf. Section 9.2). It uses the *EtherCATMaster* device interface as well as the appropriate observer interfaces to interact with the EtherCAT bus.

The UML class diagram shown in Fig. 9.7 illustrates the architecture of the Stäubli TX90L device driver. For the sake of clarity, many methods of the interfaces and classes have been omitted. The abstract class *StaubliDevice* provides all methods that are required for Stäubli robots, in particular the `setJointPosition(...)` method which is used by the *JointPosition* primitive in a primitive net to set the next set-point. Two implementations of the Stäubli device are available: *StaubliSimulation* and *StaubliEtherCAT*. The first class provides a simulated version of the Stäubli device, while the second one controls the real hardware using the EtherCAT bus. Because both implementations inherit from the common *StaubliDevice* class, both implementations are exchangeable. Thus a robotics application can be tested using a simulation and run on the real robot without any change in the application, only the configuration of the RCC needs to be altered. The simulated Stäubli device has no dependencies to any EtherCAT related functionality and thus can also be used on systems where EtherCAT is not available.

The *MultiJointSimulation* class provides generic support for simulating devices using the *MultiJointInterface*. The concrete implementation of a robot simulation only needs to simulate special features (e.g. sensors) of the robot, while the general simulation (i.e. the motion of the joints) will be provided by *MultiJointSimulation*. Not only the simulation of a multi-joint system is a common task, but also the cyclic calculation of position set-points. Thus the *StaubliEtherCAT* device inherits from the *CyclicPositionDevice*

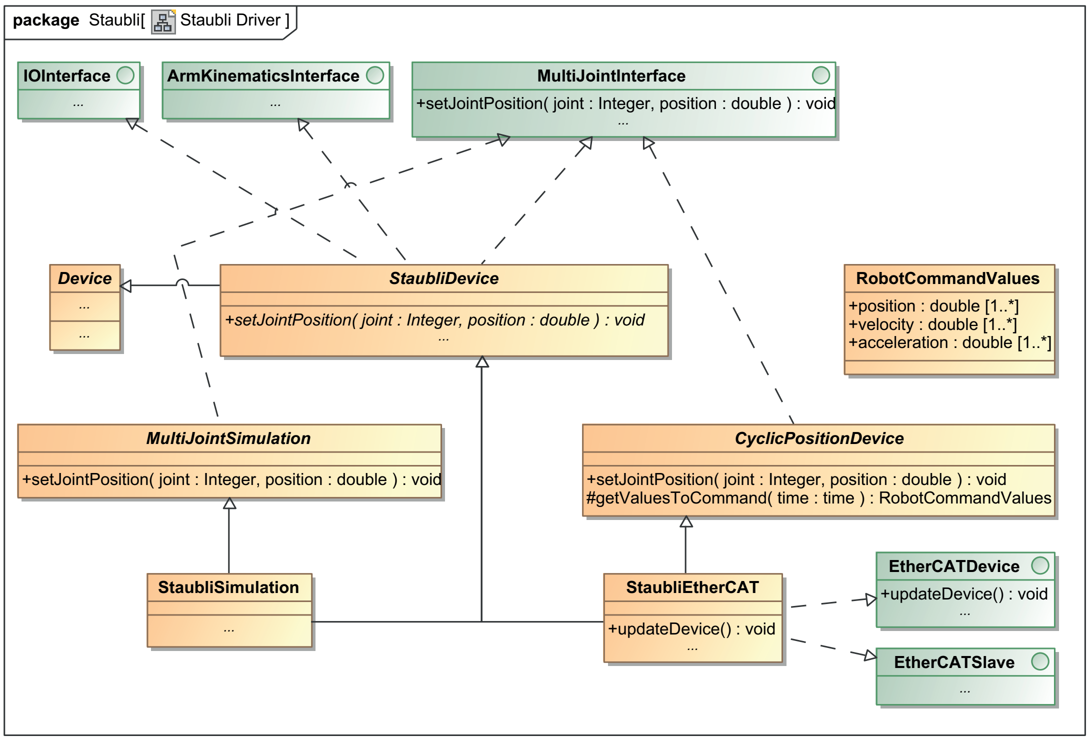

package Staubli[Staubli Driver]
IOInterface
ArmKinematicsInterface
MultiJointInterface
+setJointPosition(joint : Integer, position : double) : void
Device
StaubliDevice
+setJointPosition(joint : Integer, position : double) : void
RobotCommandValues
+position : double [1..*]
+velocity : double [1..*]
+acceleration : double [1..*]
MultiJointSimulation
+setJointPosition(joint : Integer, position : double) : void
CyclicPositionDevice
+setJointPosition(joint : Integer, position : double) : void
#getValuesToCommand(time : time) : RobotCommandValues
StaubliSimulation
StaubliEtherCAT
+updateDevice() : void
EtherCATDevice
+updateDevice() : void
EtherCATSlave

Figure 9.7.: UML class diagram for the Stäubli TX90 device driver

class which provides an implementation of the set-point estimation algorithm described in the previous section. The class implements the `setJointPosition(...)` method and thus receives new set-points from a running primitive net. The *StaubliEtherCAT* class implements the observer interfaces *EtherCATDevice* and *EtherCATSlave* for being notified in every process data cycle. In each cycle, the method `getValuesToCommand(...)` is called with the expected time of the next SYNC0 event to calculate the values which must be written in the process image. Although *CyclicPositionDevice* provides values for position, velocity and acceleration for every joint, the uniVAL interface only requires the position.

9.4.4. KUKA lightweight robot

The KUKA lightweight robot (LWR) 4/4+ [12] (cf. Fig. 9.8) was the first robot which was supported by the SoftRobot RCC and was used as the reference robot system throughout the SoftRobot project. The LWR offers some unique features compared to other industrial robots:

- With a weight of only 15 kg it is able to lift up to 15 kg, although under normal conditions the payload is limited to 7 kg.

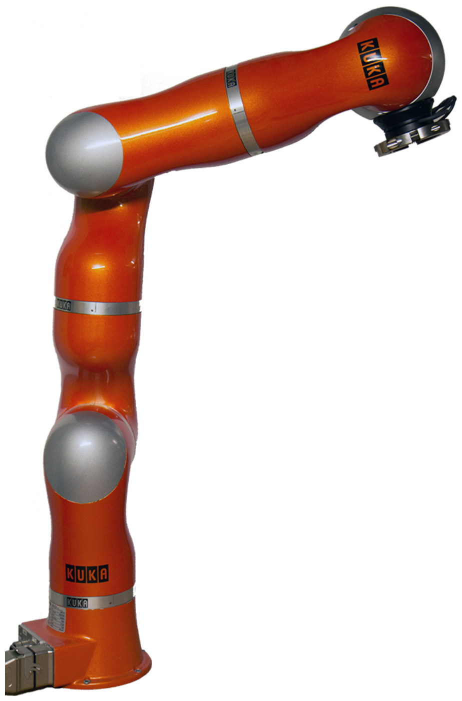


Figure 9.8.: KUKA lightweight robot LWR4

- The LWR has 7 joints, the additional rotatory joint has been inserted between the second and third joint of a conventional 6-DOF industrial robot. The additional joint allows the robot to perform motions in the null-space, i.e. the robot elbow can move while the flange stays stationary. Due to this additional degree of freedom, the inverse kinematics function of the LWR yields an infinite number of solutions for any given Cartesian position.
- Each joint is equipped with a torque sensor. This allows precise measurements of torques applied to each joint instead of estimating those values e.g. from motor currents. Using the torque sensors, the LWR provides built-in functionality for impedance control.

The LWR is controlled using a special version of the standard KUKA controller, the KR C2lr controller. The controller runs a special version of the KUKA Software System (KSS) 5.6.5 which has been extended to support the additional features of the lightweight robot. The LWR can be programmed either by using the KUKA KRL programming language, or by using the Fast Research Interface (FRI) [106] which allows a fast, cyclic position control for each joint using a UDP network link.

The architecture of the LWR device driver included in the SoftRobot RCC is very similar to the device driver of the Stäubli TX90L robot, although no support for the

EtherCAT fieldbus is required. Like the Stäubli driver, the LWR driver also implements the *MultiJointInterface*, the *ArmKinematicsInterface* as well as the *IOInterface* and thus can be used in application identically to the Stäubli robot. Besides implementing the interface of standard robots, the LWR device driver also provides some new primitives which expose the unique features (e.g. the force-torque sensors and the different control modes) of the LWR to robotics applications.

Control modes of the LWR

The KUKA lightweight robot controller supports four different control modes, which can affect the position of the robot. At every moment in time, the robot has a "commanded position" and a "measured position". The commanded position is set either by running a KRL program containing motion commands or by supplying new set-points over the FRI connection, while the measured position is retrieved from the encoders embedded in each joint.

- **Position control:** In this control mode, the robot behaves like any other industrial robot and tries to reach the commanded position exactly. The robot tries to compensate for external forces and applies an emergency stop if any force or torque exceeds its capabilities. This control mode should not be used when the robot is expected to make contact with the environment. The commanded and measured positions are usually very close to each other.
- **Joint impedance control:** In this control mode, each joint reacts independently like a spring to external forces. The robot still tries to reach the commanded position, however this position is not necessarily reached if there are any external forces. For each virtual spring, the stiffness and damping can be configured.
- **Cartesian impedance control:** This control mode is very similar to joint impedance control with the difference being that the virtual spring is defined in Cartesian space for the tool center point and not individually for each joint.
- **Gravity compensation:** In this control mode, the robot compensates for the earth's gravity, but does not apply any further force. This allows the user to move the robot (and any attached payload) virtually without any force by directly touching the robot.

To achieve good results for these control modes, the robot needs to have a precise model of itself, and also needs to know some data about the attached payload, including the mass, the center of mass and the moment of inertia of the payload. For Cartesian impedance control, the tool center point is also important because the virtual spring is calculated for this point.

Fast Research Interface

The *Fast Research Interface (FRI)* [106] allows external motion control of the lightweight robot using UDP packets over a network link. The configuration of the FRI connection is done using a KRL program running on the KUKA robot controller. In this program, the IP address and port of the external motion controller is specified, and the FRI connection is controlled. There are three modes the FRI system can have: *OFF*, *MONITOR* and *COMMAND*. Without any configuration, FRI is in state *OFF*. After the connection has been opened in the KRL program, FRI enters state *MONITOR*. In this state, the KUKA controller transmits UDP packets containing the current status of the robot at a configurable interval. This interval can be as short as 1 ms. The external motion controller must respond to each packet in a timely manner, however it is not possible to control the LWR yet. If the KUKA controller receives enough packets with good quality (i.e. neither missing responses nor receiving responses too late), it is possible to enter *COMMAND* mode. In this mode, full control of the robot is possible.

The data packet transmitted from the robot to the external motion controller contains, amongst others, the following data:

- The measured and commanded positions of each joint
- The measured and commanded Cartesian position of the tool center point (TCP)
- The measured absolute torque of each joint
- The estimated external torque of each joint
- The estimated external force and torque applied to the TCP
- Values read from KRL variables
- A sequence number of the packet

The external motion controller must reply to each packet with a packet containing, amongst others, the following data:

- The commanded position of each joint
- The commanded Cartesian position of the TCP
- The stiffness and damping for each joint
- The stiffness and damping of the TCP for Cartesian impedance mode
- Additional forces or torques which should be applied
- Values which must be written to KRL variables
- The sequence number of the packet for which the response is sent

The data telegrams always contain the same set of data, although not all parameters are relevant in every control mode. For example, in position control mode the commanded joint stiffness or damping is ignored. During *MONITOR* mode, the commanded joint and Cartesian positions must mirror the measured positions. As soon as *COMMAND* mode has been reached, those values must no longer be mirrored (slight jitter in measured positions causes unstable behavior) but rather be filled with the desired values. The data telegrams contain a set of values which are read from or written to variables in the KRL program. This can be used for communication between the external motion controller and the KRL program.

Unfortunately it is not possible to change the control mode of the robot using FRI, only the control parameters (stiffness, damping) can be set using FRI. In order to change the control mode itself, or to set new load data (e.g. if a workpiece has been gripped, the mass of the workpiece must be set in order to achieve good impedance control), KRL commands must be used. This can only be done while FRI is not in *COMMAND* mode. The LWR driver uses some of the KRL variables to send all necessary new control parameters to the KRL program, which then switches from *COMMAND* mode to *MONITOR* mode, updates all parameters in the KUKA controller and tries to switch back to *COMMAND* mode. This process takes some time (up to 10 s) and occasionally fails, requiring manual intervention at the robot controller.

The built-in gravity compensation mode can also not be selected while using FRI. Although it is possible to activate gravity compensation using a KRL program (and thus it would be possible using the FRI communication link described previously), its use in automated programs is not possible due to two reasons: First, the KUKA controller does not allow the gravity compensation mode to be used at all when in AUTO operation mode (i.e. the robot program is running without any interaction and without the need of holding a dead man's switch), which is the default operation mode when using FRI. Second, even if one would accept using such a switch and use KUKA's T1 or T2 operation mode, all KRL programs are interrupted as long as gravity compensation is active, thus deactivating gravity compensation would only be possible by manual interaction of the user with the KUKA control panel (KCP). To mitigate these problems, Robotics API based applications use joint impedance mode with very low stiffness and moderate damping values instead of "real" gravity compensation. If the stiffness is set low enough, the robot does not try to reach its commanded position, but will still react to any external forces just like with the gravity compensation control mode.

When using joint impedance to emulate the "real" gravity compensation mode, the controller has to ensure that the commanded position of the robot eventually reaches the real position. With very low stiffness parameters, it is possible to move the robot away from the currently commanded position by applying force to the robot (as it is desired for hand-guiding the robot), without the robot trying to move back to the commanded position. As soon as the stiffness parameters are set to higher values (or even the position controller is activated), the robot would try to move back. Since this is not a planned motion, but rather implied by the closed-loop controller, this motion can be extremely fast. If the commanded and measured positions only differ for a very small distance (smaller than approx. 5 cm) the robot simply jumps to the original destination, for larger distances an emergency stop is triggered by the KRC's safety controller. To prevent such issues, the SoftRobot RCC continuously tries to maintain a small difference of commanded

Input ports	none
Output ports	outMeasured: double outEstimated: double
Parameters	Robot: String Axis: int

Table 9.7.: The *TorqueMonitor* primitive: measured and estimated torques at an joint of the LWR

and measured positions during gravity compensation. However, it is not possible simply to mirror the measured to the commanded position. The KRC's safety controller also monitors the commanded position for excessive speed or acceleration. When simply mirroring the measured position, it is very easy to exceed those defined maximum values, although the robot itself is perfectly able to follow the motion. Therefore, the SoftRobot RCC uses an online trajectory generator (OTG) to let the commanded position follow the measured position while limiting the velocity and acceleration of the commanded position. Before gravity compensation mode is stopped, the RCC waits for the commanded position to be very close to the measured position.

The process data cycle of FRI can be configured from 1 ms up to 100 ms. The KRL program is interpreted using the standard KUKA interpolation cycle time of 12 ms. When using data cycle times smaller than 12 ms, massive jitter with a periodic time of 12 ms can be experienced. To compensate for this jitter, the LWR driver also uses the *CyclicPositionDevice* class to calculate appropriate set-points. Unlike EtherCAT, FRI does not provide an estimation of the time the new set-point will be expected, thus the LWR device driver can only use the point of time the telegram is generated as time t in Eq. (9.1) (Page 143).

Support of advanced LWR features

The LWR device driver implements the *MultiJointInterface* to move the arm, the *ArmKinematicsInteface* for the (standard) kinematics and inverse kinematics functions and the *IOInterface* for interaction with peripheral devices which are directly connected to the KUKA KR C2lr controller via fieldbus (usually DeviceNet). Using those interfaces, robotics applications can already interact with the LWR like with any other standard industrial robot, however they cannot yet utilize the special features of the LWR. To support all features of the LWR, some additional primitives are provided by the LWR driver.

The *TorqueMonitor* primitive (cf. Table 9.7) reports the torque measured at a single joint, as well as the estimated external torque which is applied to the joint. The estimated external torque is adjusted for the effects of gravity on the robot and the configured payload. The *ForceMonitor* primitive (cf. Table 9.8) reports the estimated force and torque that is applied to the tool center point of the LWR. The force is represented as a

Input ports	none
Output ports	outForce: Vector outTorque: Vector
Parameters	Robot: String

Table 9.8.: The *ForceMonitor* primitive: estimated forces/torques at the TCP of the LWR

vector in the tool coordinate system, while the torque is represented as three torques which would be applied around the coordinate axes of the tool coordinate system. Those values are directly provided by the KUKA controller over the FRI protocol. Standard joint values (position, velocity and acceleration) can be retrieved using the *JointMonitor* primitive from the *MultiJointInterface* device interface (cf. Section 9.2). The payload currently attached to the robot may be configured using the *ToolParameters* primitive which is also provided by the *MulitJointInterface*.

To switch the LWR's control mode, the *ControlStrategy* primitive is provided, which allows the primitive net to select a new control mode. Like the *ToolParameters* primitive, this primitive also provides an output port for signaling the completion of the switching process because this may take up to 10 s. The robotics application should ensure that the measured and commanded positions are identical within a short range before attempting to switch the control mode. If the difference of measured and commanded position is very large, the controller will not perform the control mode switch at all. But even if the difference is small enough for the controller to switch, very fast robot movements could result. For example, if the robot is currently in an impedance mode with a rather soft spring, it will move as fast as possible to its commanded position when switched to position control mode.

The control mode parameters can be set using the *JointImpParameters* and *CartImpParameters* primitives. Setting new controller parameters takes effect immediately, thus those primitives do not have a completion output port. New controller parameters can be set any time, even if the commanded and measured positions are not identical. It should be noted however, that applying stronger stiffness values can also lead to very fast robot movements.

The *Kin* and *InvKin* primitives provided by the *ArmKinematicsInterface* can also be used with the LWR. The inverse kinematics function will solve the additional redundancy introduced by the seventh joint by selecting a solution close to the provided hint value for the joint. The LWR driver additionally offers its own kinematics function primitives *LWRKin* and *LWRInvKin* which have additional support for an α value, which specifies the angle the elbow is out of the vertical plane traditional industrial robots can move in. This angle can be achieved using the additional 7th joint and thus is specific to the LWR.

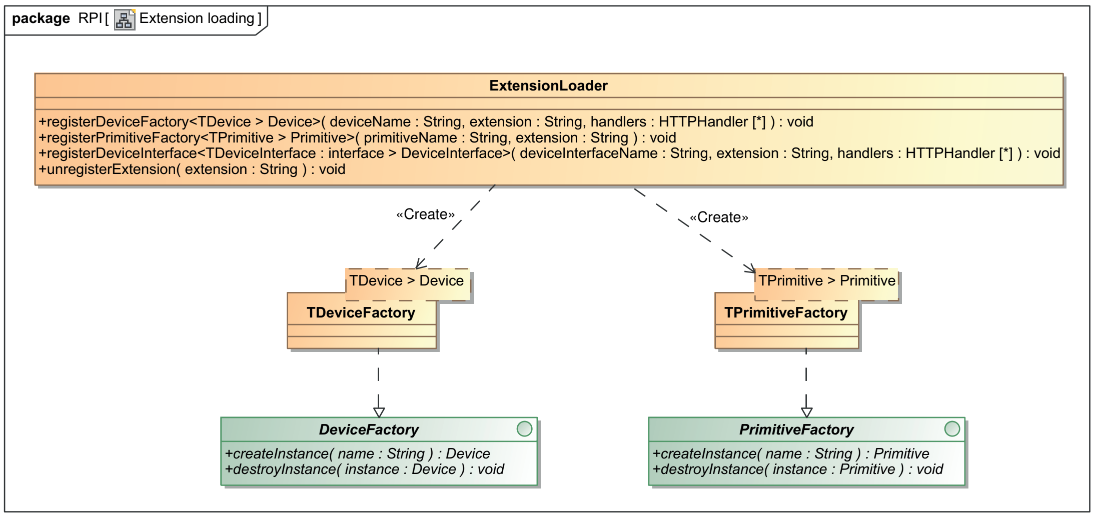

package RPI [Extension loading]
ExtensionLoader
+registerDeviceFactory<TDevice > Device>(deviceName : String, extension : String, handlers : HTTPHandler [*]) : void
+registerPrimitiveFactory<TPrimitive > Primitive>(primitiveName : String, extension : String) : void
+registerDeviceInterface<TDeviceInterface : interface > DeviceInterface>(deviceInterfaceName : String, extension : String, handlers : HTTPHandler [*]) : void
+unregisterExtension(extension : String) : void
«Create»
«Create»
TDevice > Device
TDeviceFactory
TPrimitive > Primitive
TPrimitiveFactory
DeviceFactory
+createInstance(name : String) : Device
+destroyInstance(instance : Device) : void
PrimitiveFactory
+createInstance(name : String) : Primitive
+destroyInstance(instance : Primitive) : void

Figure 9.9.: UML diagram of ExtensionLoader class

9.5. Loadable extension modules

The SoftRobot RCC allows loading and unloading extension modules during runtime. On the one hand, this allows loading only those device drivers that are really necessary, and on the other hand also allows replacing driver implementations without the need of restarting the whole system. Loading and unloading modules is performed by using the appropriate operating system functions (e.g. `dlopen`, `dlsym` and `dlclose` on Linux). Extension modules are compiled as shared objects on Linux (extension `.so`) or dynamically loadable libraries (extension `.dll`) on Windows.

As the first step for loading an extension, the extension file is opened. If the specified file is not a valid loadable module, this step will fail. This step also fails, if dynamically linking the module is not possible due to missing dependencies. In a next step, three named methods are looked up: `version`, `load` and `unload`. Those methods must be defined once in each loadable extension module within an **`extern`** `"C"` declaration in the C++ source code in order to prevent name mangling by the C++ compiler. Extension loading is aborted if any of the three methods is not found. At first, the parameter-less method `version` is called. This method returns a 64-bit integer value which describes the version of the RCC it was built for. If this version does not match the current RCC version, loading is aborted. Loading an extension with non-matching version could be potentially dangerous if any interface has changed meanwhile, because memory addresses are generated at compile time and not verified at run time.

If the versions of the RCC and the loadable extension module match, the `load` method of the extension module is called and a reference to `unload` is saved. The `load` method takes an object of type *ExtensionLoader* as an argument, which provides several methods for the extension to register its devices, primitives and device interfaces. Figure 9.9 displays this class and its methods with signatures. For all devices and primitives an extension provides, factories must be created which later instantiate the devices and primitives. The register methods of *ExtensionLoader* automatically generate those factories. Unlike "normal" methods, the code for template methods in C++ will always be generated by the same compiler which also compiles the calling program code. Therefore it is possible to provide these convenience methods centrally for all extension modules. The generated factories encapsulate the `new` and `delete` operators, thus they are created by the same compiler as the primitives or devices. The register methods require the class of the device or primitive as a template parameter, the name of the device or primitive and the name of the extension (for centrally unloading all devices and primitives of an extension). For devices, a list of *HTTPHandler* objects can be specified to provide new web server handlers for the device (see also Section 7.6.2). If necessary, further initialization steps can be performed by the extension module during its load method.

When an extension should be unloaded, it must be ensured that no instance of a primitive or device exists any more. The Registry holds a reference counter for all primitive and device instances and will only allow unloading an extension if the counters for all primitives and devices are zero. In this case, the `unload` method is called with an object of type *ExtensionLoader*. This object can be used to call the `unregisterExtension` method which destroys all factories which have been created during the loading of the extension.

Listing 9.2 shows the loading and unloading code for the extension module for a simulated KUKA lightweight robot. Line 1 forces the C++ compiler into plain C mode which deactivates name mangling (otherwise the `dlsym` method would not be able to find the methods). Line 3 defines the name of the extension which must be used to register and unregister all devices and primitives. In line 5, the version method starts, which returns the version id of the extension. The constant `rcc_bin_version_id` is read from the header files of the main RCC source code, thus the version compiled into the extension module matches the version of the RCC of which the header files have been used. `rcc_bin_version_t` is a type definition for the version number (because primitive types such as `int` may have different sizes on different architectures). `RTT_EXPORT` is a macro definition by Orocos which is required on the Windows platform to mark methods which should be exported in a DLL file. Line 10 defines the `load` method. In lines 12 to 14, *HTTPHandler* instances are created. The given URIs are relative to the device's URI, i.e. the handlers will ultimately be registered as `/devices/`*`devicename`*`/` and `/devices/`*`devicename`*`/kinematics`. In line 16, the class *LwrSimulation* is registered as device. Lines 19 to 22 implement the `unload` method which unregisters all devices (and primitives, if registered).

```
extern "C"
{
   const std::string ext_kuka_lwr_sim = "kuka_lwr_sim";

   rcc_bin_version_t RTT_EXPORT version()
   {
      return rcc_bin_version_id;
   }

   void RTT_EXPORT load(RPI::ExtensionLoader loader)
   {
      HTTPHandlerList handlers;
      handlers.push_back(HTTPHandlerInfo("/", new LBRFRIControllerHandler()));
      handlers.push_back(HTTPHandlerInfo("/kinematics",new LBRFRIKinematicsHandler()));

      loader.registerDeviceFactory<LwrSimulation>(dev_lwr_joint_sim, ext_kuka_lwr_sim,
          handlers);
   }

   void RTT_EXPORT unload(ExtensionLoader loader)
   {
      loader.unregisterExtension(ext_kuka_lwr_sim);
   }
}
```

Listing 9.2: Example code for loading and unloading the extension module for a simulated KUKA lightweight robot

Chapter 10

Embedding RPI in the Robotics API

The Robot Control Core provides the Real-time Primitives Interface, which allows the specification of arbitrary real-time safe tasks using a data-flow graph based language. However, this interface is not suitable for programming robots directly by end-users, on the contrary, it has been designed for automatic generation by a framework such as the Robotics API. The Robotics API provides a generic application programming interface (API) for robotics applications, as well as the so-called “mapping” algorithm which automatically generates the required primitive nets for the application.

The Robotics API has been developed using the Java programming language, however the concepts of the Robotics API can also be implemented using any other object-oriented programming language. An automatic conversion for the .NET platform (in particular for the C# programming language) can be done using IKVM.NET [65].

10.1. Robotics API architecture

The Robotics API consists of two independent layers, the *Command layer* and the *Activity layer*. The command layer directly communicates with the *Robot Control Core* (RCC) and offers a generic interface for programming arbitrary robotic devices, consisting of a number of actuators and sensors. A real-time safe event mechanism is provided for reactions to sensor data.

While the command layer offers a very flexible interface for any kind of robotics devices, it lacks an easy-to-use syntax for typical applications for industrial robotics, such as it is usually provided by the currently used proprietary robot programming languages. The activity layer provides such an interface on top of the command layer.

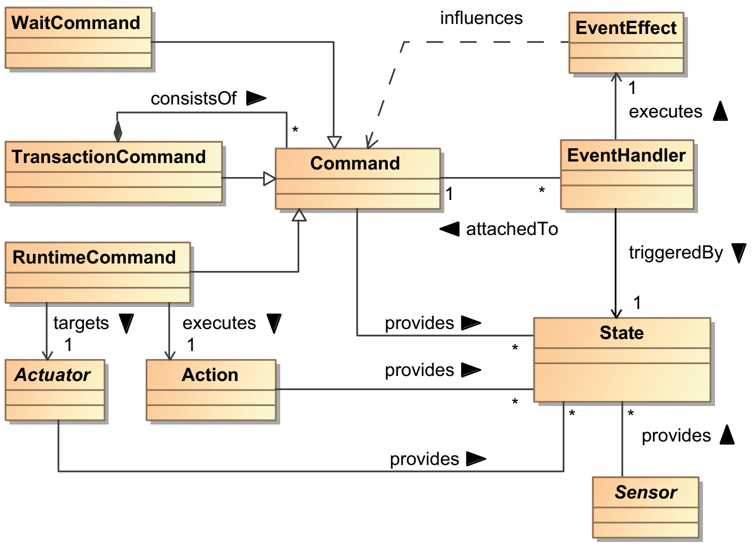


Figure 10.1.: UML class diagram of the core classes of the command layer in the Robotics API (diagram taken from [3])

The command layer has no dependencies to the activity layer, thus it is possible to use the Robotics API with only the command layer. The activity layer relies heavily on the command layer and thus cannot be used alone.

The architecture of the Robotics API as well as the concepts of the command and activity layers have been developed by A. Angerer [1]. The concepts are repeated in this work in order to introduce the automated transformation process from commands specified in the Robotics API to primitive nets, the so-called mapping process. The main concepts of the Robotics API have been published in [2–4].

10.1.1. Command layer

The Robotics API command layer provides an object-oriented model for industrial robotics applications. The most important basic concept is an *Actuator* that performs an *Action*. *Commands* consist of at least one actuator and one action and can be influenced by a *Sensor*, which can trigger *Events*. The event-handling within the command layer is done real-time safe on the RCC, thus the triggering and timing of events is deterministic and highly reliable.

Figure 10.1 shows the core classes of the Robotics API command layer. The following sections describe the concepts more in detail.

Actuator

An *Actuator* is any kind of controllable device. Examples are complete articulated robots, but also single robot joints, as well as peripheral devices such as grippers, welding equipment, etc. Concrete instances of the class *Actuator* describe a certain type of hardware with all relevant properties (e.g. the number of joints, the mass, allowed maximum velocities and accelerations), but do not contain any implementation for hardware control. These classes serve as proxy objects for the devices implemented in the RCC (cf. Chapter 9). Instances of these classes can be used in robotics applications and ultimately will be mapped to primitives (or fragments) during program execution. This mapping process is done completely automatically (cf. Section 10.2).

When used in robotics applications, actuators can be configured by applying a set of *ActuatorParameters*. All actuators define a set of default parameters, such as the maximum allowed speed of any joint. However the application developer may decide to attach another parameter specifying a lower speed limit for a specific motion in the program. Parameters specified explicitly in the application always override the default values of an actuator.

Instances of the class *Actuator* should be stateless, i.e. they must not store any state of the hardware they represent. Such information must always be retrieved directly from the RCC. Multiple applications are allowed to interact simultaneously with a single RCC, thus the state of a hardware device might have changed without the application having noticed, which would render the application's state outdated.

Action

An *Action* specifies any single action that can be performed by an actuator. Examples for actions are robot motions, but also tool actions such as opening or closing a gripper. Like the class *Actuator*, the class *Action* also is an abstract description of the action to perform without any implementation details. During the mapping process, an action will be converted into a primitive net providing all necessary input for an actuator to perform the specified action. The mapping of an action specifying a point-to-point motion for an articulated arm e.g. generates a primitive net which provides interpolated set-points to the device of the according actuator.

Separation of Action and Actuator

The separation of *Action* and *Actuator* is done on purpose, because there is no finite set of actions a certain actuator can perform. Using two separate concepts, adding new actions to existing actuators is possible, and furthermore the same actions can also be used for different types of actuators (e.g. a generic point-to-point motion action may not only apply to articulated joint robots, but also to robots using linear joints or even mobile and flying systems).

Command

A *Command* represents any kind of operation that can be executed atomically in a real-time context. Atomically also means that no modification of the command is possible from the application once it has been started (although the effects of the command may be influenced on a non real-time base, e.g. by changing the overall override speed). In the SoftRobot platform, a *Command* will be ultimately transformed to a primitive net for execution on the RCC. The *Command* concept is a modified variant of the "command pattern" as specified in [45]. Several different variants of *Command* are available.

One type of Command is the *RuntimeCommand* which combines an *Action* and an *Actuator* into an executable form. The Action specifies *what* the Command must do, while the *Actuator* specifies *who* should perform the operation. Like action and actuator, the *RuntimeCommand* also provides no information about *how* it must be executed. This is once again up to the runtime environment and done while mapping the *RuntimeCommand* to a primitive net.

A *WaitCommand* simply waits for a given period of time without doing anything. This type of command is usually used in composed structures of commands, e.g. if some action must be delayed in relation to another action such as a tool action that must be performed a certain time after a motion has started. *WaitCommands* will also be executed by the RCC, thus the specified time will be precisely met with precision of one primitive net execution cycle.

A *TransactionCommand* combines multiple *Commands* into one real-time context. It is configurable as to whether a *Command* should be started together with its surrounding *TransactionCommand*, or individually using the event mechanism for commands. Because *TransactionCommands* themselves are also *Commands*, they can be nested into other *TransactionCommands*.

Sensor

A *Sensor* is any kind of real-time capable data source. Such data sources can be classic hardware sensors such as joint encoders embedded into a robot joint, or a force and torque sensor attached to a robot's flange. Sensors are not limited to hardware devices, but can be any kind of data source that can be reliably calculated. Any combination of sensor values, obtained e.g. by adding or multiplying values of other sensors is again a sensor. Besides sensors provided by hardware or calculated values, it is also possible to create "virtual" sensors whose values are supplied by the robotics application at run-time. In contrast to most hardware sensors, no real-time guarantees can be given for this data.

All sensors can be mapped to primitives or combinations of primitives for execution on the RCC. Physical sensors are represented by primitives which are provided by the device drivers for the hardware sensors. Not only specific sensor hardware pieces provide sensor data, but also most actuators have integrated sensors. Each joint in a robot can provide at least its current position, often the current velocity and acceleration can also be measured. Some robots with special abilities such as the KUKA lightweight robot for example can also provide measurements of external torques applied to the joints.

“Virtual” sensors are mapped to *TNetcommIn* primitives to inject a data source into a running primitive net. New values can be provided by the application using the non real-time communication channel between the Robotics API and the RCC (cf. Section 7.6.1). Furthermore, virtual sensors providing a constant value are possible (using *TValue* primitives).

Sensors can be data sources for all kinds of data types. For real-time execution, the data type must also be available on the RCC. Common data types for sensors are integer or double for numerical values, or Boolean for sensors with only two possible states. Furthermore, sensors may also provide complex spatial data types such as vectors or frames (position including orientation). An example for such a sensor is a position detection system.

It is possible to combine multiple sensors with basic arithmetic operations such as add or multiply. The result yields another sensor, which itself can be part of further calculations. The basic arithmetic operations are mapped to calculation primitives (e.g. *TAdd*, *TMultiply*, cf. Section 7.3.5) on the RCC. The use of multiple primitives that provide only basic arithmetic functionality allows the developer to specify almost arbitrarily complex calculations which can be automatically transformed into primitive nets and later be executed in a real-time context. It is also possible to compare different sensor values (equality, greater than, . . .), yielding yet a new sensor with data type Boolean.

Sensors can not only be used in a real-time context, but a robotics application can also request the current value of any sensor by calling the `Sensor#getCurrentValue()` method anywhere in the application. The Robotics API checks whether the current value of this sensor can be retrieved “cheaply”, i.e. all required values are already known. If this is not the case, a primitive net with the required sensor structure is created and evaluated on the RCC. If an application requires continuous non real-time information about sensor values, a *SensorListener* can be attached to the sensor. This listener will be called each time a change of the sensor value is detected. It can be configured how changes of the sensor value should be handled in case the last called listener is still running. Value changes can either be queued or dropped. *SensorListeners* are not executed in a real-time context, thus there is no guarantee that all value changes will be reported (especially short value pulses are likely to being missed).

State

A *State* describes a discrete state of a robotics system which can be either *active* or *inactive*. States are not identical to the traditional state in a state machine, but rather captures one discrete feature of a complex robotics system. Different types of a *State* originate from different sources:

SensorState is provided by a Boolean sensor and is active as long as the sensor reports the value *true*. States can also be derived from non-Boolean sensors by first creating a Boolean sensor using a condition, e.g. the `isGreater` function can be applied to two Double sensors (or one Double sensor and a constant value) and creates a new Boolean sensor which provides a SensorState.

ActuatorState is provided by an actuator and can monitor certain actuator-specific properties, e.g. whether an actuator is powered or not.

ActionState is provided by an action, e.g. an action provides states whether it is completely finished or whether a certain percentage of an action has been completed.

CommandState is provided by commands, e.g. whether a command is finished, started, or currently executing.

Besides those states provided by parts of the robotics application or the hardware, states can also be derived from other states:

AndState combines two states and is active if both other states are active.

OrState also combines two other states and is active if at least one of the states is active.

NotState is active, if another state is not active and vice versa.

LongState is active, if another state has been active without interruption for a given amount of time.

Event

Different commands can be combined into one *TransactionCommand* that is executed real-time safe. It is possible to start all commands synchronously, however, often it is required to start one command and trigger the execution of other commands based on certain conditions. Such an event can be triggered by entering or leaving a *State*. A state is entered when it was *inactive* and becomes *active*, and is left when it becomes *inactive* after being *active*. Events can be limited to the first occurrence of a *State* being entered or left.

An event is handled by an *EventHandler* which is attached to a *Command*. The *EventHandler* monitors the appropriate state and triggers an *EventEffect* if necessary (i.e. the state has been entered or left). *EventEffects* can influence other commands. The following *EventEffects* exist:

- *CommandStarter* starts another command in the same *TransactionCommand*
- *CommandStopper* unconditionally stops another command
- *CommandCanceller* requests another command to terminate gracefully
- *WorkflowEffect* is an EventEffect which does not directly influence another command in the real-time context, but is propagated to the robotics application. This event effect does not provide any real-time guarantees.

- *CommandSynchronizer* is a “virtual” event effect which assigns a token to the event for usage in a synchronization condition. See Section 10.2.5 for more details.

With the exception of the *WorkflowEffect*, all event effects will be processed real-time safe within the RCC and thus are guaranteed to occur at the first primitive net execution cycle the condition becomes true. Unlike non real-time sensor listeners, events are guaranteed not to miss any sensor value (which is transformed to a state) if the value is available for at least one execution cycle.

The *CommandStopper* will immediately stop another command (i.e. the parts of the primitive net representing the command will not be executed again in the next execution cycle). If the affected command was actively controlling an actuator and no other command is started simultaneously that takes over control, the actuator will no longer be actively controlled. Depending on the hardware, this can lead to an emergency stop of the actuator and possibly even damage the hardware. If a *TransactionCommand* is stopped, all commands contained in the transaction are also stopped.

The *CommandCanceler* will only issue a cancel request to the command (implemented using a Boolean data-flow) which allows the command to stop gracefully, e.g. a robot currently moving can be stopped first. A command may also decide to ignore a cancel request completely if there is no safe way of canceling at the moment. A *RuntimeCommand* will forward the cancel request to its *Action*, while a *TransactionCommand* ignores canceling by default. However, the transaction command will enter its *Cancel* state, thus an *EventHandler* can be attached to that state which then cancels specific inner commands.

Examples

To demonstrate the expressiveness of the Robotics API, some real Java code examples are provided which demonstrate the main concepts introduced in the previous sections. The code examples abstract from configuration aspects, i.e. Java objects for robot devices already exist. The Robotics API provides support for specifying configuration files to configure what devices etc. are available, and provides easy means to retrieve the appropriate objects within the robotics application. This step is, however, unnecessary for demonstrating the main concepts of the command layer.

Single point-to-point motion The following example demonstrates a point-to-point motion in joint space for a single robot.

```java
RoboticsRuntime runtime = ...; // Retrieved from configuration
RobotArm robot = ...;  // Retrieved from configuration
Action ptp = new PTP(new double[] { 0, 0, 0, 0, 0, 0 },
   new double[] { 1, 1, 1, 1, 1, 1 });

Command cmd = runtime.createRuntimeCommand(robot, ptp);
cmd.execute();
```

In line 1, an object of type *Runtime* is retrieved from configuration management. A *Runtime* abstracts from the concept of a real-time system executing a command (e.g. the SoftRobot RCC) and encapsulates all necessary components for utilizing this environment. This contains the complete mapping algorithm (cf. Section 10.2), but also all necessary communication links (cf. Section 7.6). In line 2, an instance of the device which is to be controlled is retrieved. The device is stored in a variable of type *RobotArm*, which is a special interface for articulated arms. In line 3, a new *Action* is created. The constructor of the class *PTP* takes two arguments which specify the angles of all actuator joints at the beginning and at the end of the motion. In this example, the action specifies a point-to-point motion for any 6-DOF device from a position where all joints are at position 0 to a position where all joints are at position 1. The Robotics API always uses base SI units, thus the position 1 means either 1 m for translational joints or 1 rad for rotational joints. The same point-to-point action can be used for both types of joints, because the concept of a point-to-point motion is independent of the type of joint used.

In line 7, the newly created *Action* is combined with the *Actuator* to create a new *Command*. The combination is performed by the *Runtime*, which also ensures that the specified action can be executed by the specified actuator (e.g. if the point-to-point motion action is created for a 4-DOF robot, it cannot be combined with a 6-DOF robot). The command is finally executed in line 7. The default behavior for execution of a command is blocking, i.e. the control flow of the application will stop until the actuator has stopped moving.

Neither the *Action* nor the *Device* may store any information about the current condition. As a result, it is not sufficient to specify just the destination of the motion, but also the start position must be specified. If the robot's joints are not exactly at the position specified as the start position, execution of the command will fail and an exception will be thrown. In many cases, the current position of the robot is not known to the programmer, but the actual position is to be used. This can be done by using a sensor providing the current joint position.

```
Action ptp = new PTP(robot.getJointAngles(),
   new double[] { 1, 1, 1, 1, 1, 1 });
```

It should be noted that the `getJointAngles()` method is evaluated at the time the *Action* is created. Changes to the robot's position between the creation and the execution of the *Action* will not be taken into account. This must be kept in mind particularly when actions are embedded in transactional commands, i.e. when other actions are planned for prior execution.

Reaction to sensor events The integration of sensor data and the reaction to changes of this data are key features of the Robotics API. The following example demonstrates both sensor integration, as well as the event handling mechanism.

```
RoboticsRuntime runtime = ...;
RobotArm robot = ...;
```

```
3  Action ptp = new PTP(robot.getJointAngles(), new double[] { 1, 1, 1, 1, 1, 1 });
4
5  RuntimeCommand cmd = runtime.createRuntimeCommand(robot, ptp);
6
7  State state = robot.getJoint(0).getMeasuredPositionSensor().isGreater(0.5);
8  EventEffect effect = new CommandCanceller(cmd);
9  cmd.addStateEnteredHandler(state, effect);
10
11 cmd.execute();
```

Lines 1 to 5 are identical to the previous example and create a command to move a robot from its current position to a position where all joints have the position 1 rad. In line 7, the sensor which reports the measured position of the first joint is accessed, and a state created which is active if the joint position is greater than 0.5 rad. In line 8 an event effect is defined, which cancels the running command. In line 9, the state and the effect are combined using the "state entered" paradigm, and finally the command is executed in line 11.

When starting the application while the first joint has a position smaller than 0.5 rad, the robot will start a point-to-point motion to the given joint coordinates. Once the first joint reaches a position greater than 0.5 rad, the command is canceled (i.e. the robot starts to brake). The event is handled within the real-time context of the RCC, thus the robot motion will be reliably and immediately canceled once the state is entered. The command has to provide an appropriate handler for the cancel event, which in the case of a point-to-point motion will immediately start with the deceleration phase and thus stop the robot.

Combining several commands Multiple commands can be combined using a *TransactionCommand*. The following example executes two point-to-point motions strictly sequentially.

```
1  RoboticsRuntime runtime = ...;
2  RobotArm robot = ...;
3  double[] dest1 = new double[] { 1, 1, 1, 1, 1, 1 };
4  Action ptp1 = new PTP(robot.getJointAngles(), dest1);
5  double[] dest2 = new double[] { 1.5, -1.5, 1, 1.2, 1, 1.5 };
6  Action ptp2 = new PTP(dest1, dest2);
7
8  RuntimeCommand cmd1 = runtime.createRuntimeCommand(robot, ptp1);
9  RuntimeCommand cmd2 = runtime.createRuntimeCommand(robot, ptp2);
10
11 TransactionCommand tc = runtime.createTransactionCommand(cmd1, cmd2);
12
13 tc.addStartCommand(cmd1);
14 tc.addStateFirstEnteredHandler(cmd1.getCompletedState(),
15       new CommandStarter(cmd2));
16
```

```
17 tc.execute();
```

In lines 3 to 9, two subsequent commands are created which each perform a point-to-point motion. The second command uses the destination position of the first command as the start position. In line 11, a new *TransactionCommand* is created. The first command is added as start command in line 13, i.e. this command will be started immediately when the transaction command is started. The second command is started using a *CommandStarter* event effect when the first command has been finished (i.e. it enters the “command completed” state for the first time). The transaction command is then executed in line 17.

Using a *TransactionCommand* embeds all commands into the same real-time context, thus in the example above, the second motion will be executed right after the first has finished, and the operation as a whole is deterministic. If both commands were executed individually, the Java application would be responsible for starting the second motion after the first has finished, which does not provide a deterministic timing.

Motion blending (cf. Section 2.5) can be achieved by placing all motions into a single *TransactionCommand*. A specific state must be defined on the blending condition (e.g. 20 cm prior to the completion of the motion, cf. point B in Fig. 2.4). When this state is first entered, a *CommandStopper* has to terminate the first motion, and a *CommandStarter* has to start the second motion. The command for the second motion must contain a specific *Action* that is capable of taking over the moving robot, automatically planning the blending trajectory (point B to D) and finally completing the originally specified second trajectory (point D to E).

Motion blending achieved by placing all motions into a single *TransactionCommand* is guaranteed to succeed since all motions are executed within a single real-time context, however this is not the recommended mechanism. Rather it is advisable to use two independent commands which are joined by a synchronization rule, thus the robotics application maintains control of the program flow. In this scenario, motion blending is performed on a best-effort base, i.e. if the robotics application does not supply the second motion command in time (prior to reaching the blending condition), the original motion will be completed and motion blending will simply be ignored. This does impact the performance of the overall system (since the time savings achieved through motion blending are lost); the safety of the system however is never in danger. The actuator is always controlled by a running primitive net as long as it is moving.

10.1.2. Activity layer

The command layer introduced in Section 10.1.1 provides a very powerful and flexible interface for any robotics application without being restricted to any specific application domain (e.g. it is not only possible to control articulated arms, but also mobile robots or even flying robots). This flexibility however also incurs a rather complex syntax for specifying robot commands, in particular if compared to traditional robot programming languages such as KRL.

A simple point-to-point motion to a point given in joint-coordinates looks as follows when programmed using KUKA's KRL language:

```
PTP {AXIS: A1=10, A2=20, A3=-40, A4=50, A5=-90, A6=20}
```

Using the Robotics API's command layer, the same motion[1] can be programmed using

```
runtime.createRuntimeCommand(robot, new PTP(robot.getJointAngles(), new double[] { 10,
    20, -40, 50, -90, 20 } )).execute();
```

It is not very surprising that a domain specific language (DSL) such as KRL offers a more convenient programming interface for all purposes for what it was intended. However, the extensibility of such languages is usually very limited, thus new requirements such as sensor integration or cooperating robots are difficult to implement.

To support developers for "standard" robotics application, a change or supplement to the Robotics API's command layer is required. It would be possible to redesign the command layer for better robotics application support, however the Robotics API has proven to be a stable platform, and it was not intended to sacrifice the flexibility of the API just to simplify the development of some applications.

It is possible to extend the command layer using a domain specific language which aims at developers for standard industrial robotics applications. Some research has been done by Mühe et al. [90] to support the execution of legacy KRL applications using the Robotics API. However, using a DSL as main programming paradigm contradicts the aims of the SoftRobot project of supporting the robotics domain using a general purpose programming layer. Using a DSL, it is once again difficult to benefit from the large ecosystems of modern programming languages and environments.

It was decided to extend the Robotics API command layer with another Java-based layer, the *Activity Layer*. This solution has the advantage of using the Java programming language while still being able to provide an abstraction of the command layer to facilitate the development of many robotics applications. The activity layer is based on the command layer, however the command layer is not coupled in any way to the activity layer. Thus it is still possible to use the command layer without the activity layer. The following examples are provided to illustrate how end-users interact with the Robotics API, but do not go into detail about the implementation of the activity layer. More information about the activity layer can be found in [3] and [1, Ch. 8].

The activity layer introduces two new concepts into the Robotics API, the *Activity* and the *ActuatorInterface*. An *Activity* encapsulates a real-time operation which may control multiple actuators. The real-time behavior is implemented using a *Command*. Besides the real-time operation, the activity also contains meta-data which provides information about the states of all affected actuators during and after the execution of the activity, e.g. the final position of all robots.

[1]the conversion between radians and degrees has been omitted for clarity

Activity objects provide two methods for execution, `execute()` and `beginExecute()`. When using the former method, the activity is executed completely and control is returned to the application once the activity has finished. The latter method returns control to the application once the activity has been started on the RCC. This allows the application to perform further actions such as starting the next activity while the current one is still being executed asynchronously.

For the creation of *Activity* objects and their execution, it would seem intuitive to add methods for appropriate activities to devices. For example, a point-to-point motion activity could be created and executed using the following line of code:

```
robot.ptp(new double[] { 10, 20, -40, 50, -90, 20 }).execute();
```

However, this syntax would break the separation of action and actuator as introduced in Section 10.1.1, since an actuator object would need implementation for all possible actions.

The activity layer adds the concept of *ActuatorInterface* to provide a collection of functionalities which are supported by an actuator. For example all robots offer the *PtpInterface* for point-to-point motions in joint space, and the *LinInterface* for linear motions in Cartesian space. Further *ActuatorInterfaces* can be added to actuators at run-time, i.e. without modifying the code of the actuator itself.

The developer can query all currently supported *ActuatorInterfaces* from an actuator and request the use of such an interface using the actuator's `use` method. All *ActuatorInterfaces* provide methods to create activities, which then can be executed. For example, the point-to-point motion from the previous example can now be written as

```
robot.use(PtpInterface.class).ptp(new double[] { 10, 20, -40, 50, -90, 20 }).execute();
```

Although this syntax is not as compact as it is in DSLs such as KUKA's KRL, it combines a straight forward syntax with the flexibility and extensibility of separated action and actuator concepts.

A more compact syntax is not possible due to limitations of the Java programming language. Using extension methods in C# [36] or mixins in Scala [88], the requirement for the `use` method could be removed and an almost straight-forward syntax established.

Using the meta-data contained in activities and the asynchronous execution using `beginExecute()`, motion blending can be elegantly specified:

```
1  PtpInterface ptpi = robot.use(PtpInterface.class);
2
3  ptpi.addDefaultParameters(new BlendingParameter(0.8));
4  ptpi.ptp(new double[] { 10, 20, -40, 50, -90, 20 }).beginExecute();
5  ptpi.ptp(new double[] { 40, 10, 50, 30, -40, 50 }).beginExecute();
```

In line 1, an instance of the *ActuatorInterface* for point-to-point motions is stored in a local variable. In line 3, the blending condition is set to 80% completion of the motion (for joint space motions the blending condition can only be specified as progress, not as distance). This setting is applied to all motions using this instance of the *PtpInterface*. In lines 4 and 5 the two point-to-point motions are specified just like any other motion. The meta-data available in the activity layer is used to retrieve the starting position for each motion which is required for the underlying *Commands*. With the asynchronous execution, line 5 can already be evaluated while the motion in line 4 has not yet completed. In line 5 a new command will be created and attached to the previous motion command using a synchronization rule.

10.2. Mapping Robotics API commands to primitive nets

A Robotics API *Command* describes *what* a robotics system should do, however it does not contain information about *how* it should be done. To execute such a command on a Robot Control Core, it must be converted into one or more primitive nets. This process is the so-called *mapping* process and was first introduced in [105]. During this process, the information about *how* to execute a command is introduced.

During mapping, all parts of a command are recursively converted into primitive net fragments. For each part of a command (e.g. Actions, Actuators, Sensors, Events, but also nested Commands within a TransactionCommand) a *mapper* implementation creates a primitive net fragment for this part. Those fragments are later connected into a large primitive net which represents the whole Command. Each mapper implementation contains all information about how this specific part of the task can be encoded in a primitive net fragment.

The Robotics API provides proxy classes for all primitives that are available on the RCC. Using these proxies, it is possible to create a Java-representation of a primitive net or a primitive net fragment during the mapping process. After all parts of a Robotics API command have been mapped and connected, the resulting structure of proxy objects can be transformed straightforwardly into either the XML (cf. Section 7.6.2) or the DirectIO (cf. Section 7.6.3) representation of primitive nets which can be transmitted to the RCC.

The mapping process happens in several steps. Although the mapper implementations for each part of a command create a primitive net fragment containing all relevant primitives for this part, they cannot already create all necessary links between primitives, because often connections of primitives within different command parts are necessary. To solve this problem, mapping implementations can provide and require virtual *data-flow ports* both for input and output purposes. Data-flow output ports from one fragment can be connected to data-flow input ports of another fragment during a step in the mapping process where both parts already have been mapped (i.e. in a higher hierarchy level). A pair of input and output data-flow ports can be seen as some connection possibility to transmit a certain type of data. A connection of two Boolean data-flow ports for example can be directly mapped to a connection of a Boolean output port of one primitive or fragment with an Boolean input port of another primitive or fragment. However, data-flow ports do not need to map to a single link in a primitive net. It is possible to create complex data-flow types which consist of multiple links, possibly of different types. For example there is a *JointPositionDataflow* type, which represents the type of set-points for a number of joints of an actuator. For a 6-DOF articulated arm, this data-flow will be mapped to six single links of type Double. Each type of data-flow is represented by a distinct Java type in the Robotics API to allow the mapping algorithms to distinguish between different types of data-flows, even if they are ultimately mapped to the same type of links in a primitive net (e.g. joint set-points for a 6-DOF articulated robot arm and a Cartesian position consisting of three values for position and three values for orientation). Data-flow types are directly implemented by using appropriate Java classes. Data-flow ports can be connected to input and output ports of the primitive proxies just like other input or output ports. Using data-flow ports instead of directly employing primitive input and output ports has several advantages. It allows the creation of complex data-flow types without the need to implement those types also within the real-time execution environment. Furthermore, by using special data-types, the built-in type checks of the Java compiler can be used to detect bad connections. Additionally, the Robotics API has special *link builder* algorithms that provide automatic transformations for different data-flow types, e.g. for transformations of frames between different base coordinate systems.

10.2.1. Basic Robotics API concepts

All basic concepts of the Robotics API command layer (Action, Actuator, etc., cf. Section 10.1.1) are mapped into fragments. These fragments have well defined data-flow input and output ports so that they can later be connected to each other as required.

Actions provide set-points for the actuator as an open-loop control. The mapper for a specific action has to provide a combination of primitives that generate the desired set-points. Examples of primitive nets which generate set-points for a point-to-point motion can be found in Sections 5.4.2 and 5.4.3. Action mappers are always provided with two Boolean data-flow input ports which represent the active and cancel states of their context (usually the command they belong to). Using the active state, the action is signaled when to start set-point generation, and the cancel state signals for a graceful termination of set-point generation (e.g. braking an actuator, but not simply terminating). Each action fragment provides at least two data-flow output ports. The main output of the action mapper is a data-flow output port to provide the generated set-points (e.g. a *JointPositionDataflow* for a point-to-point action). Furthermore, a Boolean data-flow output port is used to provide the *Completed* state which is required for every Action. Besides these two data-flow output ports, a further Boolean data-flow output port is provided for each state the action supports. These additional states can be dynamically created by the user of the action, e.g. a point-to-point action can be requested to provide states for motion progress.

The primitive net fragment resulting from mapping an *Actuator* must be capable of processing the set-points generated by an action. Therefore, the actuator fragment

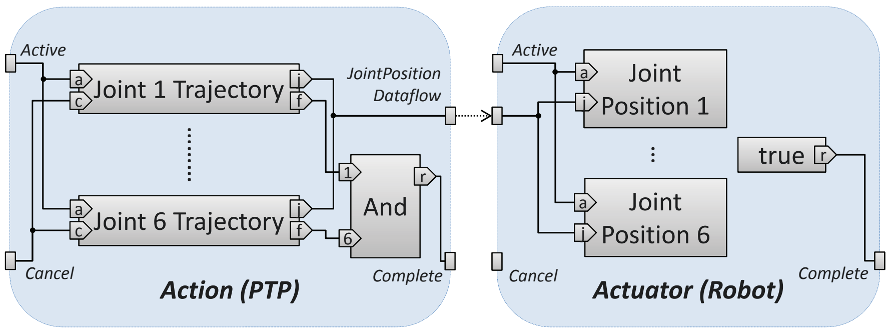


Figure 10.2.: Result of mapping a point-to-point action and a 6-DOF articulated arm (adapted from [105])

is provided with the results of the action mapping using an input data-flow port of the appropriate type. Usually a primitive from a driver or a device interface (e.g. the *JointPosition* primitive for an articulated arm) is used to process the set-points and transmit them to the hardware device. If necessary, it is possible to create a closed loop controller within the actuator implementation by not only relying on the set-point provided by the action, but also by reading sensor values of the actuator such as current encoder values. Just like actions, actuators also are provided with input data-flow ports for the active and cancel states, and are expected to provide an output data-flow port for their completed state. Many actuators furthermore provide output data-flow ports for any state (including error states for hardware failures) they can have.

Figure 10.2 depicts the primitive net fragments created by mapping a point-to-point (PTP) action and a 6-DOF articulated arm actuator. The two generated primitive net fragments are displayed as rounded rectangles (shown as shaded areas), data-flow ports as small rectangles on the border of the fragment. Inside the fragments, primitives are also displayed as rectangles, with little arrows for input and output ports. The PTP action is mapped to a collection of 6 trajectory generating primitives which are all connected to the active (port *a*) and cancel (port *c*) data-flow input ports. They all provide set-points for each joint on their output port *j* and furthermore provide an output port *f* which is *true* once the trajectory has been finished. Those output ports are connected using the *And* primitive to provide the *Completed* state once all trajectories are finished (for asynchronous motions it is possible that not all trajectories finish at the same time). All set-points are collected into a single data-flow output port of type *JointPositionDataflow*.

The actuator is mapped into 6 *JointPosition* primitives, which are all connected to the single data-flow input port. Besides the set-points, the primitives are also supplied with the active state to detect when active control of the robot is required. It is important that an actuator fragment only actively controls the hardware when its active input is *true*, because larger (transactional) commands can have multiple instances of the same actuator fragment. The articulated arm driver used in this example is able to reach a given set-point within one control cycle (if the trajectory of set-points is physically feasible at all), thus it always returns *true* on the completed state. The overall motion is only finished once both the action and the actuator report *true* on their *Completed* output data-flow ports. In this example, the actuator ignores the cancel input (as it finishes its task in a single execution cycle anyways).

The connection of the action's data-flow output port carrying the set-points and the actuator's data-flow input port (dotted arrow in Fig. 10.2) is neither created by mapping the action nor the actuator, but by mapping the surrounding command (cf. next section).

Sensors are also mapped to primitive net fragments and provide an output data-flow port which represents the current value of the sensor. It is possible to create derived sensors which rely on values generated by their underlying sensor and some calculation primitives, e.g. to add, subtract or multiply a sensor value with a constant or another sensor value. If any states are defined on a sensor or a derived sensor (e.g. sensor value greater than a constant), these states are ultimately mapped to Boolean data-flow output ports by inserting appropriate comparison primitives.

10.2.2. Basic executable commands

A *RuntimeCommand* is the smallest executable command of the Robotics API that controls an actuator. It consists of one *action* and one *actuator*. All fragments resulting from mapping commands share a common interface such that they can be combined transparently and nested into other (e.g. transactional) commands. Every command fragment has two Boolean data-flow input ports *Start* and *Cancel* which are used to start and respectively cancel the execution of the command and provides two Boolean data-flow output ports *Active* and *Completed* which signal whether the command is currently running or has completed its task.

The *Action* and *Actuator* a command consists of are mapped to fragments as described in Section 10.2.1, an example can be found in Fig. 10.2. During the mapping phase of the command, the data-flow output port of the action providing set-points is connected with the appropriate data-flow input port of the actuator. If the data-types of both ports do not match, an automatic type-conversion is attempted (cf. Section 10.2.3).

To control the activation of a command using the command's data-flow input ports, a special *Activation* fragment is used which has three input ports: *start*, *stop* and *cancel*, and provides three output ports: *active*, *cancel* and *started*. The three input ports are used to determine when to start execution (setting output *active* and *started* to *true*) and when to stop execution (setting output port *active* to *false*). The *started* output ports remains *true* once a command has been started and allows to determine whether a command has ever been run.

Figure 10.3 displays the results of mapping a *RuntimeCommand*. The parts *Action* and *Actuator* are mapped as previously described, the content of their fragments is omitted for clarity's sake. The Start and Cancel ports of the *Activation* fragment are connected to the data-flow input ports of the command fragment. The Active data-flow input ports

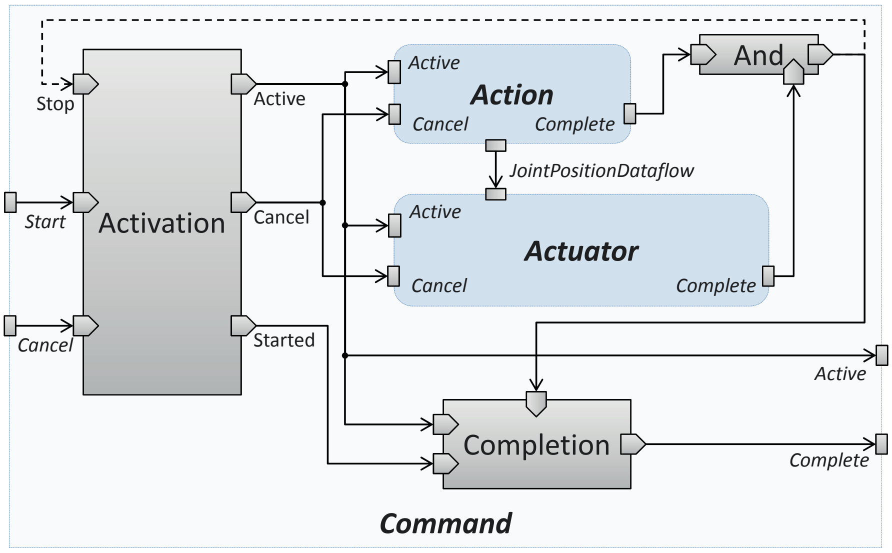


Figure 10.3.: Result of mapping a *RuntimeCommand* (adapted from [105])

of the action and actuator fragments are connected to the active output of the *Activation* fragment. The cancel port is handled analogously.

The termination of the command is also partly handled by the activation fragment. Both *Complete* output data-flow ports of the action and actuator fragments are combined (using an *And* primitive) and connected to the Stop input port of the *Activation* fragment. This connection is displayed as a dashed line in Fig. 10.3, because it needs to be delayed for one cycle, like every event effect. Both action and actuator fragments signal completion during the last active execution cycle, thus any effect of this event must only happen in the next execution cycle. A *Pre* primitive is used for connecting these ports to delay the transmission to the next cycle. Without this delay, the resulting primitive net would also contain an unguarded cycle which is not allowed.

To stop the execution of a command, the activation fragment switches off the active output port, thus disabling the execution of the action and actuator fragments. The *Completion* fragment signals *true* to the *Complete* data-flow output port if it received *false* from the *active* port and *true* from the *started* port, i.e. the command has ever been started and is no longer running. An additional shortcut from the combined *Complete* ports of action and actuator is also connected to the *Completion* fragment without a delay. This guarantees that the *Complete* output port of the runtime command changes to *true* in the same execution cycle the action and actuator have completed. The *Active* data-flow output port is directly connected with the *active* port of the activation fragment.

Besides action and actuator, commands can also contain events which must be mapped into the primitive net. In the context of a single runtime command, not all event effects as introduced in Section 10.1.1 are possible. The command stopper and canceler event effects can only apply to the same runtime command, and starting other commands is not possible.

All events are represented by a *State* which provides a Boolean data-flow output port on a fragment. This fragment can be e.g. the actuator, but also a fragment created by the mapping of a sensor. The command stopper and command cancel event effects are implemented by connecting the Boolean data-flow output port with the stop or cancel input ports of the *Activation* fragment, using a delayed connection (using a *BooleanPre* primitive). If multiple events stop or cancel the command, all connections are joined by a logical *or* to allow each single event to cause the desired effect.

Workflow effects are also mapped into the primitive net. The result of this mapping process contains a *BooleanNetcommOut* primitive (cf. Section 7.6.1) to signal the event to the application. While this event is recognized within the real-time system (i.e. states being active for only a single execution cycle reliably cause this event effect to be triggered), the processing of the event in the Java application is no longer guaranteed to be deterministic.

10.2.3. Automatic type conversion

In Fig. 10.2, the set-points generated by the action could directly be consumed by the actuator. For articulated arms and motions in joint-space, usually no conversion is required. Motions in Cartesian-space however cannot simply be fed into an actuator fragment representing such an articulated arm; an inverse kinematics transformation has to be performed.

The required data-type conversion is neither created by mapping the action nor the actuator. Including type-conversions already at this stage would limit the applicability of an action or actuator to a matching correspondent, and also lead to multiple implementations of very similar concepts (e.g. one actuator for joint-space and one for Cartesian-space). Both actions and actuators can provide *converters* that can also be mapped to primitive net fragments and that can perform the required type conversions. During the mapping process of the surrounding command, an appropriate converter will be selected and mapped between the data-flow ports when required.

To determine which converter is appropriate, the (Java-)types of the data-flow ports are considered. During the mapping process of action and actuator, instances of the data-flow ports are created which can carry additional meta-data (e.g. the number of joints in a *JointPositionDataflow* or the base coordinate system for a *CartesianPositionDataflow*). The converter uses this meta-data to create an appropriate conversion fragment.

Converters are commonly required when motions are defined in Cartesian space (e.g. linear or circular trajectories). The action fragment generates set-points in Cartesian space (i.e. X, Y and Z coordinate for translation and Euler angles A, B and C for orientation) using the *CartesianPositionDataflow* type, while the actuator expects set-points in joint space (using *JointPositionDataflow* type). The actuator provides a converter module

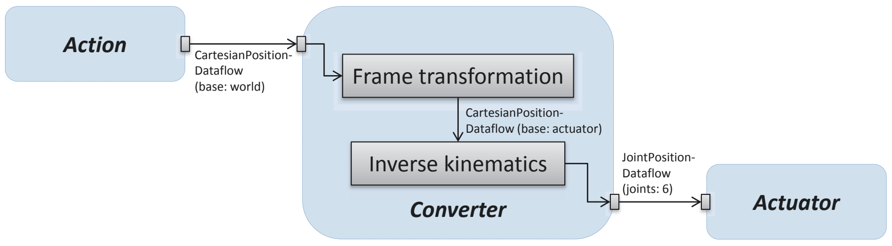

Action
CartesianPosition-Dataflow (base: world)
Frame transformation
CartesianPosition-Dataflow (base: actuator)
Inverse kinematics
JointPosition-Dataflow (joints: 6)
Converter
Actuator

Figure 10.4.: Converter fragment for conversion from Cartesian space to joint space

which is able to create a conversion fragment. Figure 10.4 outlines a primitive net combining the aforementioned action and actuator. The converter fragment contains an inverse kinematics primitive to convert from Cartesian space to joint space. The inverse kinematics primitive expects Cartesian positions in the actuator's base coordinate system. If the set-points generated by the action are specified using another coordinate system (e.g. the global world coordinate system), an additional frame transformation fragment is included which is able to perform the required coordinate system transformation. An in-detail description of frame transformations in the Robotics API can be found in [1, Chapter 7].

10.2.4. Multiple commands in a single real-time context

Multiple *Commands* can be combined into a single *TransactionCommand*. The developer can define the behavior of every command using event effects, e.g. to start a command directly after another one has finished. It is also possible to synchronously start multiple commands, which then will be executed in parallel. Such a transactional command is mapped into a single primitive net and thus is executed within the real-time context of the RCC. Switching between different commands within a transactional command is strictly deterministic. Because a *TransactionCommand* is a *Command* just as a *RuntimeCommand* is (cf. Fig. 10.1), the primitive net fragment created during mapping a *TransactionCommand* must have the same interface (i.e. data-flow input and output ports) as a *RuntimeCommand*. This allows to transparently nest transactional commands into other transactional commands.

To illustrate the mapping process for a transactional command, a simple electrical two finger gripper is used as an example. The gripper opens as long as it receives a high signal on its open input (e.g. from a fieldbus) and closes as long as a high input is set on the close input. If the gripper detects it being in a final position (either completely opened or closed), it activates a high signal on an output. A command to open the gripper has to perform the following steps:

1. Set high signal to input open to start the opening process.
2. Wait until the gripper signals reaching final position.
3. Set low signal to input open to stop the gripper motor.

```java
 1 // Retrieve from configuration ...
 2 RoboticsRuntime runtime = ...;
 3 DigitalOutput outOpen = ...;
 4 DigitalInput inFinished = ...;
 5
 6 // Create commands for all three steps
 7 Command setHigh = runtime.createRuntimeCommand(outOpen, new SetDigitalValue(true));
 8 Command wait = runtime.createWaitCommand();
 9 Command setLow = runtime.createRuntimeCommand(outOpen, new SetDigitalValue(false));
10
11 // Create transaction command containing all single commands
12 TransactionCommand tc=runtime.createTransactionCommand(setHigh, wait, setLow);
13 tc.addStartCommand(setHigh);
14
15 // Create sequential order of commands
16 tc.addStateFirstEnteredHandler(setHigh.getCompletedState(), new CommandStarter(wait));
17 tc.addStateFirstEnteredHandler(wait.getCompletedState(), new CommandStarter(setLow));
18
19 // Cancel waiting once final position has been reached
20 tc.addStateFirstEnteredHandler(inFinished.getSensor().isTrue().or(tc.getCancelState()),
       new CommandCanceller(wait));
21
22
23 // Execute the command
24 tc.execute();
```

Listing 10.1: Java code for opening an electrical two finger gripper within a single *TransactionCommand*

Each step can be programmed using a *RuntimeCommand* as described in Section 10.2.2, and the process flow of all three steps can be achieved by using events and combining the commands in a transaction. Listing 10.1 shows an exemplary application written in Java to perform the steps within a single *TransactionCommand*. At first, three *RuntimeCommands* are created, one for each step. All three commands are added to one *TransactionCommand* and linked to each other using appropriate event effects. The *WaitCommand* created in line 8 is a special command which controls no actuator but rather waits for a given period of time or – in this example – until it is canceled. The first time the *true* state of the digital input *inFinished* is entered (i.e. the gripper signaled reaching final position on its output), the wait command is canceled and thus the final step is performed.

The mapping process for a *TransactionCommand* is in many ways very similar to mapping a *RuntimeCommand*. At first, all children elements (in this case: other commands) are mapped. Later, connections among these commands are created as event effects and the overall activation connections are created. The same fragments for activation and completion handling are used.

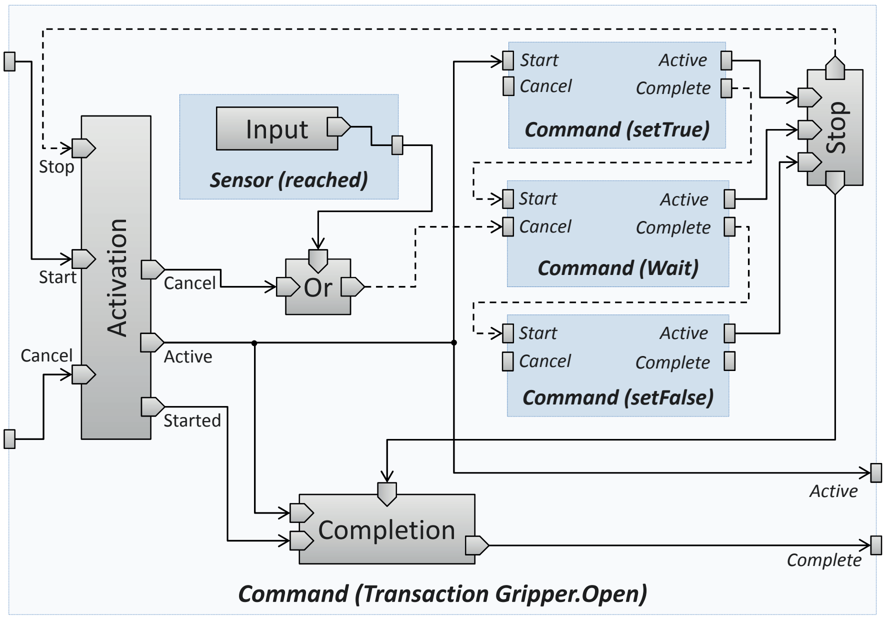


Figure 10.5.: Result of mapping a *TransactionCommand* (adapted from [105])

Figure 10.5 displays the results of mapping the *TransactionCommand* created in Listing 10.1. The three children *Commands* (each with the four typical data-flow ports) can be seen as slightly darker rectangles on the right side. The transactional command's data-flow ports (Start, Cancel, Active and Complete) are connected to the *Activation* and *Completion* fragments identically as in a *RuntimeCommand*. The end of execution of a transactional command is detected by evaluating whether at least one child is still active. If no child is active, the surrounding command also terminates.

The command created for the first step is marked as "start command" in line 13, thus its active data-flow input port is directly connected to the active output port of the activation fragment. As soon as the transactional command is started, the command for the first step will be started simultaneously.

In lines 16 and 17, the commands for the second and the third step are linked with their predecessor's completed state using the command starter event effect. This event effect can be mapped by connecting the predecessor's Complete data-flow output port with the successor's Start data-flow input port. Like all event effects, this connection must be delayed for one execution cycle (i.e. the successor shall only be started in the cycle after the predecessor has finished, and not during the same execution cycle). Therefore it is necessary that the *Complete* output port of the single command is not already delayed.

In line 20, a command canceler event effect is created to stop the execution of the wait command for step two in two different situations. The first situation is encountered, when the gripper reports reaching its final position. To detect this situation, the *isTrue* state of a digital input port connected to the grippers "position reached" output port is used. The digital input is a sensor and thus recursively mapped, the resulting fragment providing a Boolean data-flow output port for the requested state. This state is combined with a logical *or* with the transactional command's cancel state. If the transactional command itself is canceled (e.g. by the user trying to abort a program), it causes the wait command to be canceled as well. The command canceler event effect then causes the combined state to be connected to the wait command's Cancel data-flow input port (using a delayed connection like for all event effects).

Unlike in a *RuntimeCommand*, where the cancel event of the command is automatically distributed to action and actuator, the cancel event of a *TransactionCommand* is only distributed to children if explicitly programmed. Only the developer can decide which children commands can be safely canceled, and which must continue to run. In the example of the two finger gripper, canceling the overall transaction command must not lead to step three not being executed. If the command for step two is canceled before the gripper has reached its final position, only the following execution of step three will actually stop the gripper. Simply aborting the whole transactional command would lead to the gripper motor not stopping at all.

10.2.5. Synchronizing multiple independent commands

Using a *TransactionCommand*, it is possible to create sequences of commands which are executed one after another within the same real-time context, thus being able to guarantee the timely execution of each command. However to achieve this, all commands must be combined into a single Java *Command*, which does not allow to use mechanisms of the Java programming language for control flow (such as *if* branches). For many applications, hard real-time between different commands is not required, e.g. for motion blending (cf. Section 2.5). The synchronization mechanism introduced in Chapter 6 allows executing multiple distinct commands sequentially with a best-effort approach. Motion blending and other transitions that require real-time transitions from one command to another will be executed if the execution environment can guarantee the required real-time transition.

A synchronization rule was defined in Section 6.1 as 4-tuple (cf. Eq. (6.1))

$$\sigma = (C, \omega, \psi, \alpha)$$

with ω, ψ and α being primitive nets to stop, cancel and start when condition C becomes true. Within this section, ω, ψ and α can also be considered as commands, since every top-level command (i.e. any command not contained within another command) can be mapped to a single primitive net. Generally, commands stopped, canceled or started by a synchronization rule are not different to any other "ordinary" command. Canceled commands need to handle their cancel state appropriate to terminate, otherwise the synchronization rule will have no effect on them. Commands started by a synchronization rule must be able to cope with the state of the system, which may contain actuators in an unsafe state (e.g. moving or applying force). If the synchronization condition C is properly defined, commands will only be started in exactly the situation they expect.

Commands can create *CommandSynchronizer* event effects which can be part of the synchronization condition C. *CommandSynchronizer* can be used like any other event effect. During the mapping process, all *CommandSynchronizers* are mapped to primitives which make the attached event available to the synchronization mechanism. In the reference implementation SoftRobot RCC, communication primitives (cf. Section 7.6.1) are used for this purpose.

Example: motion blending

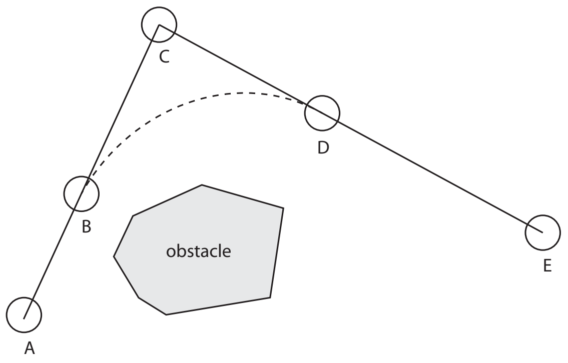


Figure 10.6.: Motion blending (adapted form [125])

The motion blending concept is used to demonstrate the mapping process of multiple synchronized commands. The desired action is a linear motion from point A to point C and a subsequent linear motion to point E, while the actuator should not stop at point C but rather continue moving on a curved connecting line (cf. Fig. 10.6). Three *Commands* are required for this task:

1. Linear motion from point A to point C.
2. A combination of the curved connecting line (point B to D) and the remainder of the second motion (D to E).
3. Linear motion from point C to point E.

For this example, all three commands are mapped into individual primitive nets, and not embedded into a *TransactionCommand*. Usually commands 1 and 2 are sufficient for executing the desired task. Due to non-real-time behavior of the Robotics API application however it is possible that command 2 is not ready for execution when command 1 has reached point B. Because the robot may not be left in an uncontrolled state, the execution of command 1 must be continued until the motion stops in point C. Command 3 is only required to restart in a safe way after motion blending has failed. After command 1 has completed in point C the system is in a safe state; there are no timing requirements for

```
1  RoboticsRuntime runtime = ...;
2  RobotArm robot = ...;
3
4  double[] dest1 = new double[] { 1, 1, 1, 1, 1, 1 };
5  PTP ptp1 = new PTP(robot.getJointAngles(), dest1);
6  RuntimeCommand cmd1 = runtime.createRuntimeCommand(robot, ptp1);
7
8  CommandSynchronizer syncToken = new CommandSynchronizer();
9  cmd1.addStateFirstEnteredHandler(ptp1.getMotionTimeProgress(0.8), syncToken);
10
11 runtime.synchronize(null, new Command[] {}, new Command[] {}, new Command[] { cmd1 });
12
13 double[] dest2 = new double[] { 1.5, -1.5, 1, 1.2, 1, 1.5 };
14 PTPFromMotion ptp2 = new PTPFromMotion(dest1, dest2, cmd1.getPlan(ptp1).
       getJointPositionsAt(0.8), cmd1.getPlan(ptp1).getJointVelocitiesAt(0.8));
15 RuntimeCommand cmd2 = runtime.createRuntimeCommand(robot, ptp2);
16
17 CommandHandle cmd2handle = runtime.synchronize(syncToken, new Command[] { cmd1 }, new
       Command[] {}, new Command[] { cmd2 });
18
19 cmd2handle.waitComplete();
```

Listing 10.2: Java code for two commands started using synchronization rules

the creation of command 3, it is even sufficient to create command 3 only after failure of motion blending has been detected.

Listing 10.2 shows the Java code for two blended point-to-point motions. The first motion is created in lines 4 to 6. In lines 7 and 8 a *CommandSynchronizer* event effect is created and attached to the event "motion has progressed 80%". The command is finally started in line 11 by creating a synchronization rule, which has *null* as condition (i.e. it is started immediately), does not stop or cancel any commands and starts *cmd1*. The `execute()` method is not used since this method would block until the motion has completed.

The second point-to-point motion is planned in lines 13 to 15. In contrast to the first motion, not only the start and stop positions are required, but also the position of the point where blending should be started is required. Furthermore, since the robot is moving at this point, the current velocities of all joints are required. The first motion command can be queried for these values. The second motion finally is added to a synchronization rule in line 17 which is scheduled for execution once the *CommandSynchronizer* in the first command is triggered. When the second *Command* is started, the first one must be terminated. Line 19 waits until the second motion has completed since line 17 does not block. The example does not contain code to handle the case when the second motion is not ready in time. In this case the first motion will be completed and the second never started, since the event defined in line 9 did not become active when the second motion was ready.

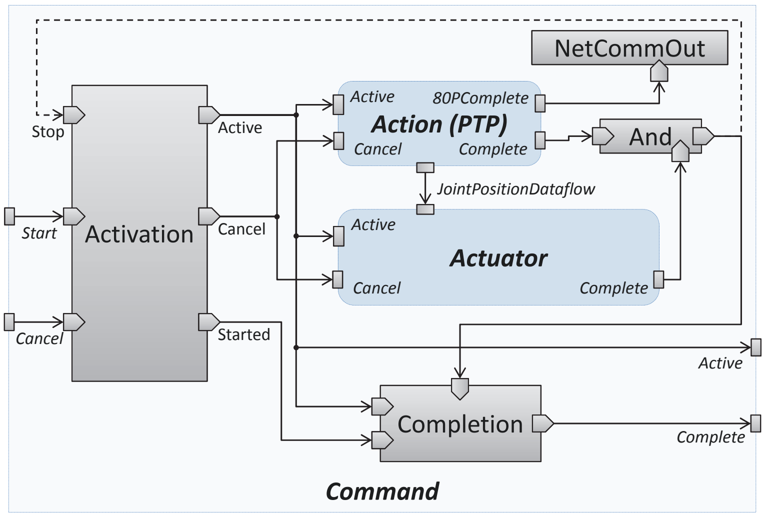


Figure 10.7.: *RuntimeCommand* for first motion

Figure 10.7 shows the resulting primitive net after mapping the first point-to-point motion command. Compared to a standard command, the action provides an additional Boolean data-flow output port *80PComplete* which signals reaching 80% of the motion. This data-flow output port is connected to a *NetCommOut* primitive which provides the Boolean variable required for the synchronization condition. The command generated for the second motion does not differ from any other command. The contained *Action* must be able to start with a moving robot.

The activity layer of the Robotics API relieves the developer from many of the tedious tasks required in Listing 10.2 to ensure that the second motion can take over the first (e.g. the position and velocity at the start of the second motion). The activity layer saves meta-data for each previous *Activity* which is used to infer all required conditions at the start of any successive motion.

10.2.6. Complete example for a generated primitive net

The mapping algorithm has been explained in the last sections on a rather abstract level in order to give an overview of how real-time tasks of robotics applications can be automatically translated into primitive nets. For real-world applications, the shown examples are missing some important features, in particular error handling has been omitted for simplicity's sake. Error handling in the Robotics API is done using states, all actions and actuators provide states for errors that can occur. With the appropriate event handlers, the application can react to errors both in a real-time safe way, if the error handling is also mapped into the primitive net; or by aborting the execution and notifying the (non real-time) application.

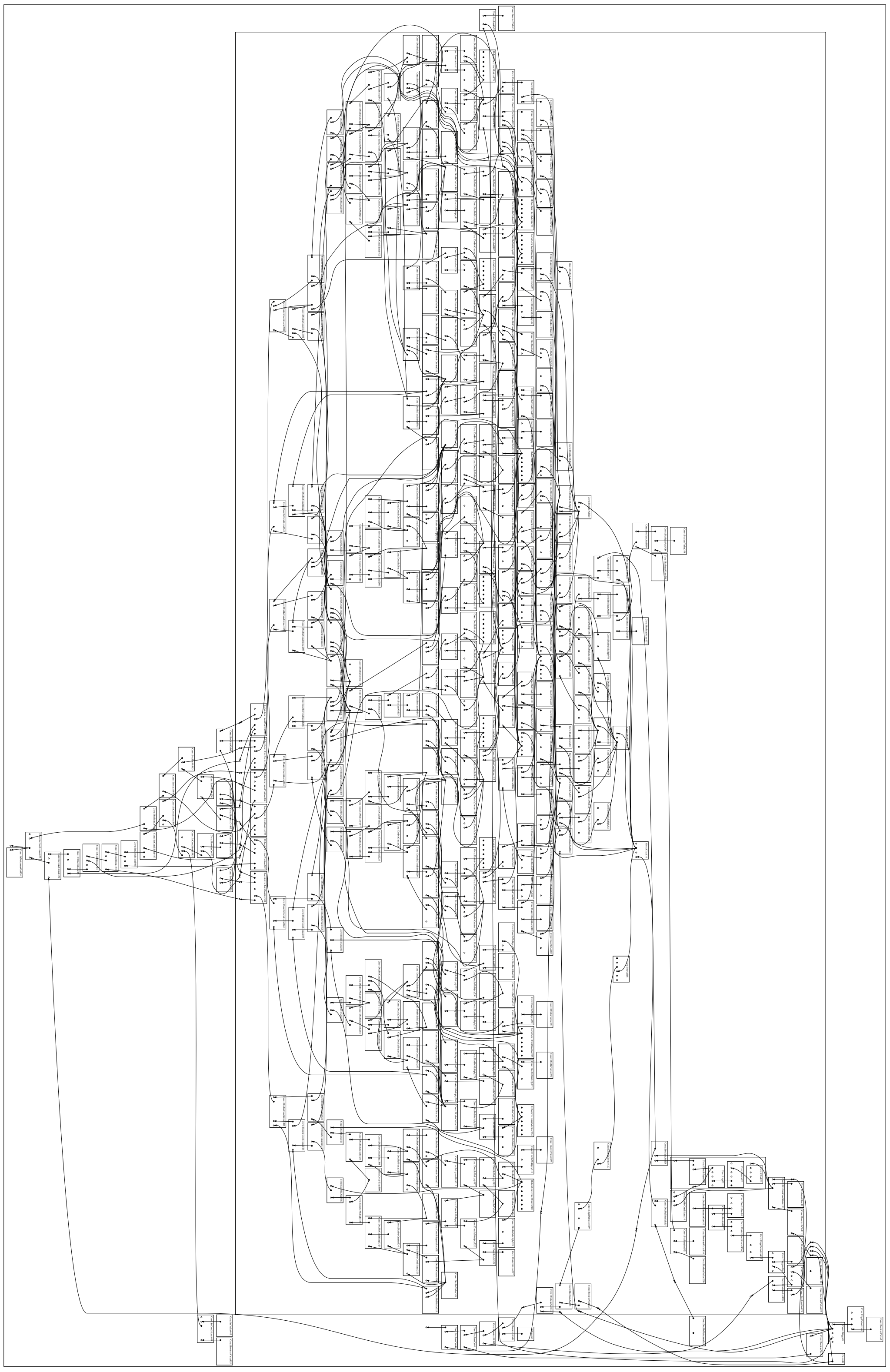


Figure 10.8.: Primitive net of a point-to-point motion for a 6-DOF robot

The default settings include error handling for some of the most common errors that occur with robots: loss of power supply to the drives (e.g. due to an external emergency stop) or the attempt to move joints outside their allowed range. The inclusion of all these events and handlers leads to a rather large primitive net. Figure 10.8 depicts a point-to-point motion as it was generated by the Robotics API for a 6-DOF Stäubli TX90L industrial robot. The graphical representation has been created using the *dot* program [46] from the *graphviz* package.

10.2.7. Automatically optimizing generated primitive nets

The mapping algorithm creates primitive nets by recursively creating the structures required for the different concepts in the command layer. Thus every command, every event etc. is mapped completely independent of all other items. This allows for a very flexible and extensible mapping algorithm, but results in potentially huge primitive nets. Since every action, command or event handler shares some primitives (e.g. primitives for injecting constant values, error handling fragments, . . .), parts of the generated primitive net are redundant.

An automated optimization step removes these redundancies after the mapping has completed. Every RPI fragment is analyzed to find multiple primitives which can be represented by a single instance. Two primitives are considered identical if all the following conditions are met:

1. Both primitive instances have the same type
2. The two input ports with the same name of both primitive are connected to the same source output port of the same instance of source primitive
3. The parameters of both primitive instances are identical

Two primitive instances fulfilling all conditions can be considered equal and replaced by a single instance since primitives must not have – within a single execution cycle – any side effects. After primitives have been replaced by a common representative, all primitives connected to the output ports must be revisited again using the optimizing algorithm since condition 2 could now be fulfilled. The optimization algorithm terminates if a fix-point is reached, i.e. no further identical primitives are found.

The time required to transfer a primitive net to the RCC is reduced by the automated optimization system by replacing all primitive names in the generated net with very short automatically generated identifiers. This reduces the size of a primitive net encoded in either the XML or the DirectIO format.

10.2.8. An Eclipse-based graphical editor for primitive nets and mapping code

As seen in Section 10.2.1, basic *Actuator* and *Action* objects are mapped to fragments, providing the relevant functionality based on a combination of primitives. The resulting primitive net fragments must be specified with means of Java proxy-objects, describing all

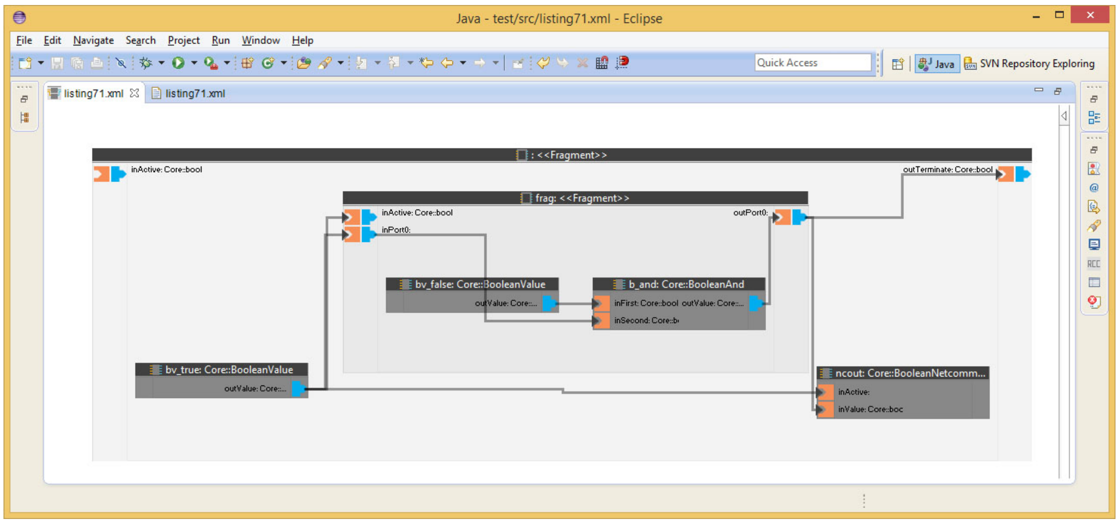


Figure 10.9.: Eclipse plugin displaying a primitive net

necessary primitives and connections. The textual representation of these primitive nets in Java is rather hard to read, and mistakes in writing can easily happen (e.g. accidentally connecting the wrong ports or even primitives).

To support the development of mapping code, a plugin for the Eclipse IDE has been developed by Kaup [72] during his master's thesis. This plugin supports the graphical design of primitive nets, and can automatically generate Java source code containing the appropriate proxy objects, including all connections. Basic syntax checking is integrated, i.e. the user is prevented from connecting incompatible ports, or from creating cycles without *Pre* primitives. The available primitives are extracted from the Robotics API runtime environment using reflection. Thus the editor automatically adjusts to changes in the Robotics API, such as new hardware devices. The plugin also supports extracting primitive nets from Java source files as long as it is possible to statically analyze the code (i.e. loops etc. are not interpreted). This feature is mainly aimed at providing a bidirectional way of editing primitive nets and Java mapping code.

Each primitive net that is created using the editor can also be used as a fragment and inserted into other primitive nets, which allows for an easy reuse. During the creation of the Java source code representing such a primitive net, one class is generated for each fragment, which also implements the interface of a primitive. Thus it is possible to include a fragment into another fragment in both the graphical editor as well as in the Java source code. In the graphical editor it is possible to hide the contents of a fragment for better overview of the primitive net; the content can be accessed by clicking on the fragment. Figure 10.9 displays the Eclipse plugin with a graphical representation of the primitive net which was introduced in Listing 7.1 (Page 105).

Chapter 11

Evaluation

The concepts of the Real-time Primitives Interface and the synchronization mechanisms for multiple primitive nets are evaluated in this chapter. To perform this evaluation, the Robot Control Core reference implementation *SoftRobot RCC* is used.

The evaluation is performed in different aspects. At first, the real-time performance and scalability of the system is investigated. Afterward, some applications which demonstrate the real-time capabilities of the system are analyzed. The extensibility of the SoftRobot RCC implementation is evaluated by the integration of a new type of robotics hardware. Finally, the goals identified in Section 4.1 are reviewed.

11.1. Real-time performance

In order to evaluate the real-time performance of the SoftRobot RCC reference implementation, the execution times for primitive nets for a set of standard motions have been measured. For all measurements, the hard- and software listed in Table 11.1 has been used. The computer system has been fine-tuned for real-time operations, i.e. all unnecessary components such as sound or USB interfaces have been deactivated. This is necessary primarily to avoid having to share interrupt lines with multiple devices. Besides these custom settings in the BIOS setup, the system is completely standard "off-the-shelf" hardware.

Three different motion commands have been evaluated: A point-to-point motion in joint space (PTP), a linear motion in Cartesian space (LIN), both executed by a single robot, and a linear motion (SYNC-LIN) performed simultaneously by two robots have been analyzed. All motions have been created using the command layer of the Robotics API (cf. Section 10.1.1). The PTP and LIN motions both consist of a single runtime command

System	Fujitsu Esprimo P710 E90+
Processor	Intel Core i5-3470 (4 cores)
Memory	8 GiB
Operating system	Xubuntu 12.04.5 LTS
Kernel	Linux 3.2.21 x64 with Xenomai real-time extensions
Real-time software	Xenomai 2.6.2.1

Table 11.1.: Hardware used for performance measurements

	PTP	LIN	SYNC-LIN
№ primitives	457	670	1350
Avg. cycle time	1.9993 ms	1.9992 ms	1.9992 ms
Standard deviation	$9.1564 \cdot 10^{-6}$ ms	$1.733 \cdot 10^{-5}$ ms	$1.8886 \cdot 10^{-5}$ ms
Min. cycle time	1.9860 ms	1.9804 ms	1.9809 ms
Max. cycle time	2.0146 ms	2.0143 ms	2.0167 ms
Min. execution time	0.0568 ms	0.0964 ms	0.0945 ms
Standard deviation	0.0057 ms	0.0039 ms	0.0042 ms
Max. execution time	0.0966 ms	0.1562 ms	0.3566 ms
Standard deviation	0.0045 ms	0.0119 ms	0.0400 ms

Table 11.2.: Measured performance results for three different motions

with one action and one actuator. The SYNC-LIN motion consists of two runtime commands (one for each robot) which are started synchronously by embedding them into a single transaction command. All motions have been repeated 100 times. Simulated robot devices have been used which behave identically to real hardware devices.

Table 11.2 shows the results for all three motions. Three different aspects have been evaluated: the total number of primitives that are required for the motion, the cycle time for the primitive net and the minimum and maximum real execution times within a single execution cycle. All times have been measured using the internal real-time clock of the computer system. The current time is recorded every time an execution cycle starts; the difference of this time in two subsequent execution cycles is the current cycle time. This time is averaged over the whole lifetime of the primitive net. The average cycle time in Table 11.2 is the mean of the average cycle times from each of the 100 primitive nets. The standard deviation is calculated from these 100 values. The minimum and maximum cycle times are the absolute minimum and maximum times for all cycles in all primitive nets.

The execution times are the times an execution cycle of the primitive net is actively running, i.e. the time difference from starting an execution cycle until all primitives have been executed once. For each primitive net, the shortest and longest execution cycle has been reported. Table 11.2 lists the averages and standard deviations of these values for 100 primitive nets.

The point-to-point motion has the lowest number of primitives of all three motions. The linear motion is slightly more complex due to two reasons: While the point-to-point motion can be calculated completely in joint-space, the linear motion has to be calculated in operation space and later converted into joint-space using the inverse kinematics function which requires several primitives. Furthermore, the point-to-point motion used in this evaluation used a trapeze velocity profile with three phases (cf. Section 2.3) while the linear motion used a double S profile with seven phases. The synchronized linear motion internally consists of two linear motions. The resulting primitive net is slightly larger than twice the size due to the synchronization of both motions.

The average, minimum and maximum cycle times for all three motions do not differ significantly. All primitive nets have been configured to run with a cycle time of 2 ms. Keeping precise cycle times is the sole duty of the underlying real-time operating system and depends largely on its ability to schedule all real-time threads appropriately. The task of the primitive net does not influence the cycle times as long as it is possible to execute the primitive net within a single cycle. If the task is too complex, or if too many primitive nets are running concurrently, the real-time operating system can no longer achieve the required cycle times. The SoftRobot RCC detects such situations and terminates primitive nets which overrun the allocated cycle time. For all three evaluated motions, the scheduler managed very good cycle times almost all the time with the worst deviation being in the range of only 17 µs.

The minimum execution times are between 57 µs and 97 µs for all motions. The minimum execution times are of no particular interest since the real-time operating system will always be able to schedule the thread for the next cycle appropriately. The maximum execution times are much more of interest to determine whether the primitive net can be executed real-time safely. The standard deviation of the maximum execution time is significantly higher than the standard deviation of the average cycle times. This is mainly due to the real-time operating system interrupting the primitive net execution thread for other high priority threads such as hardware device drivers. Nevertheless the execution times are usually within 10% of the average maximum time.

Figure 11.1 shows the execution times exemplary for the point-to-point motion. Each execution phase (cf. Section 5.3.2) is displayed separately. It can be seen that after a comparably slow first execution cycle, most subsequent execution cycles take approximately the same time, however there are some outliers whose execution times differ at most 30 µs. Those outliers do not correspond to any specific event within the primitive net, but are rather caused by indeterminism of the x86 computer system. Experiments have shown that actively using the graphics adapter causes many hardware interrupts which can create jitter up to 150 µs, therefore the small outliers in Fig. 11.1 are to be expected. They do not impact the overall real-time performance of the system, since the cycle times of the primitive nets are not affected. All measurements for this section have been performed using a remote connection to the RCC, thus avoiding load on the graphics adapter.

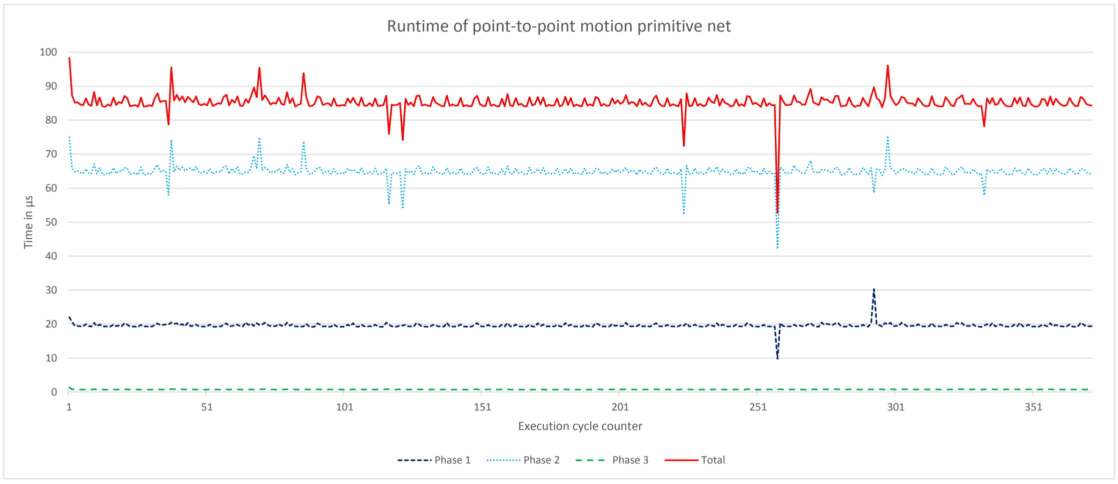


Figure 11.1.: Execution times for a point-to-point motion primitive net, separated into the three execution phases

Point-to-point motions have the shortest maximum execution times. For linear motions, the additional inverse kinematics functions take a considerable amount of time in each execution cycle. Therefore, the synchronized linear motions are expected to take at least twice the execution time of a single linear motion plus some overhead for synchronization. With only 0.357 ms execution time for two synchronized robots and a 2 ms cycle time, it is possible to scale the system to support many more synchronized robots. However the maximum execution time must not reach the cycle time. In particular the overall processor load should not exceed 70%, otherwise – depending on the scheduler used – the system might no longer be able to schedule all tasks correctly [83]. For each additional robot, the RCC also needs some calculation time for the device drivers which are always executed in parallel to the primitive nets. It is also possible to reduce the cycle time to lower values. This however only provides an advantage if the connected hardware devices support control with high frequencies such as the KUKA lightweight robot does. The other robots (Stäubli TX90L and KUKA KR-16) used for the evaluation only support control cycles with 4 ms, thus further increasing the primitive net cycle time does not provide any benefit.

Besides the hard real-time performance of the SoftRobot RCC, also the time required by the Robotics API to generate primitive nets is important for the overall performance of robotics application. An analysis of these times has been performed by Angerer et al. [3, Section 7.3]. By average it takes 50 ms to 200 ms until a motion specified using an *Activity* is started. The first three to five motions can take significantly longer to start (up to 600 ms) which is most likely due to the just-in-time compiler of the Java virtual machine. A long delay in starting activities can have an adverse impact on production cycle times if motion blending is desired and the successive primitive is not loaded in time. For applications with many motions with a duration much shorter than 1 s, motion

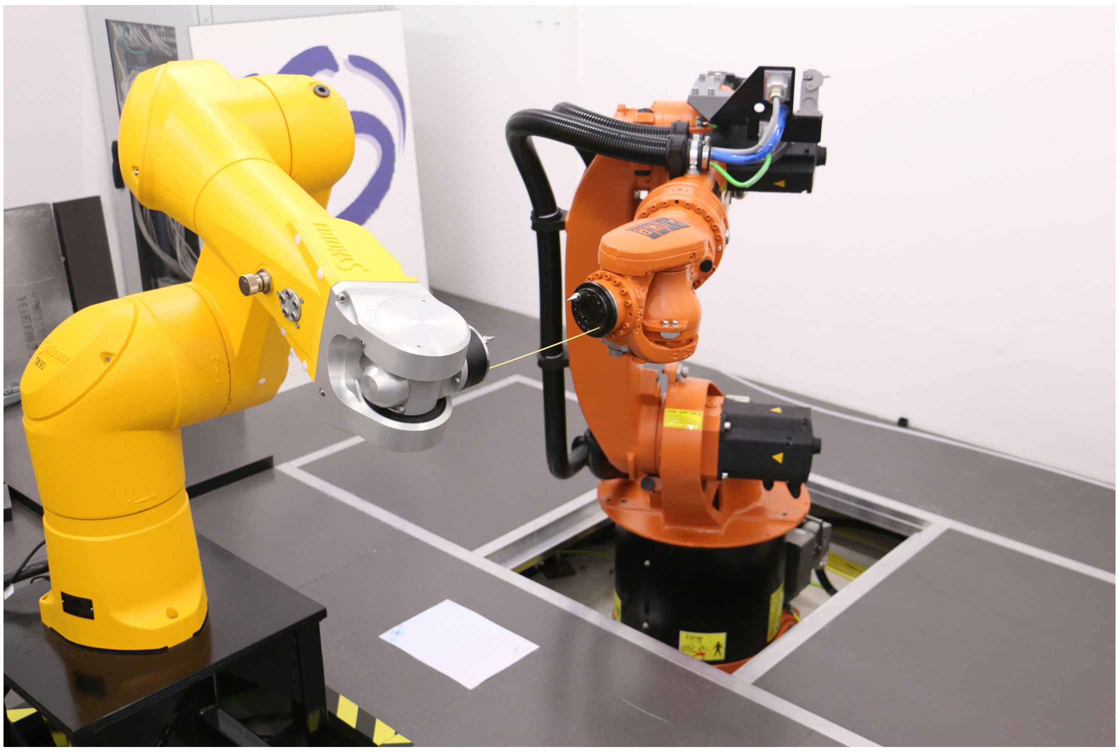


Figure 11.2.: Stäubli TX90L and KUKA KR-16 robots holding a spaghetti between the flanges for synchronous motions

blending might not be working reliably. For longer motions however the measured times show that the Robotics API will be able to provide the successive primitive in time. However, motion blending is still performed on a best-effort base as there cannot be guarantees due to the Java application generally not being real-time safe.

11.2. Applications

To demonstrate the capabilities of the SoftRobot architecture, several small applications have been developed. This section explains three applications with special attention to real-time control. One application demonstrates the real-time synchronization of multiple robots, the second applications shows (real-time) reactions to external events and finally an example for the design of a closed loop controller using primitive nets is presented.

11.2.1. Synchronized robots

In order to evaluate the real-time synchronization of multiple actuators (cf. requirement 2 in Section 4.1), an application moving two robot synchronously on a linear trajectory has been created. To show that both robot move exactly with the same velocity and acceleration, a fragile spaghetti has been mounted between the flanges of the robots which bends or breaks if too much force is applied due to unsynchronized motions. The overall setup can be seen in Fig. 11.2. For this evaluation, robots from two different manufacturers have been used. The first robot is a Stäubli TX90L, controlled over the EtherCAT fieldbus, and the second robot is a KUKA KR-16, controlled using the Remote Sensor Interface (RSI). A video of the experiment is available under `http://video.isse.de/spaghetti/`.

Before two robots can perform a synchronized motion in Cartesian space, a common base coordinate system must be defined. Since the exact mounting positions of both robots were not known, it was decided to measure a common base coordinate system for both robots. Figure 11.2 shows the base coordinate system as a sheet of paper attached to the floor between both robots. The metal spikes attached to both robots' flanges have been used to measure the coordinate system using the "3-point method". For this method, at first the origin of the coordinate system must be "touched" using the spike, and subsequently a point on the positive X-axis and a second point on the positive XY-plane must also be touched in order to define the orientation of the base coordinate system.

The experiments consisted of two linear motions, first from left to right (seen when standing directly in front of the Stäubli robot, i.e. the Stäubli robot was moving forwards and the KUKA robot backwards) and then back again. For robot cooperation, only motions in operation space are viable, thus a linear motion has been chosen. In order to achieve a smooth trajectory, a "Double S" profile with limited jerk has been implemented according to [9, Section 3.4]. The speed was limited to 2 m/s, the acceleration to 10 m/s^2 and the jerk to 100 m/s^3.

The first experiments however showed that both robots did not move perfectly synchronously – although the generated trajectories were perfectly in sync. Each time a linear motion was performed from left to right, the spaghetti was bent, while in the opposite direction the spaghetti was torn. Apparently the Stäubli robot was starting the motion slightly earlier than the KUKA robot. In order to get reliable data about the motions, an external motion tracking system was used to capture the precise positions of the robots with a high frequency. Four Vicon MX-T40s cameras with each 4 megapixel resolution (cf. Fig. 11.4a) were positioned around the robots, and infra-red reflecting markers were attached to the flange of each robot (cf. Fig. 11.4b). The Vicon cameras are equipped with a 12.5 mm lens which provides a field-of-view with $67° \times 52°$, yielding an angular resolution of approximately 0.014°. The flanges of the robots were in approx. 3 m distance of the camera lens, resulting in a positioning resolution of approx. 0.7 mm for each camera. By using four cameras and interpreting grayscale images, the resolution is further increased. Unfortunately no exact information about the overall precision is available from Vicon. The Vicon system captured the positions of the markers with 250 Hz. Figure 11.3 shows a diagram of the measured deviance of the distance of both flanges. It can be seen that for the first motion the flanges are getting approximately 1 cm closer, while the distance increases about the same amount during the second motion. Between both motions, the original distance is restored. The captured velocity profiles

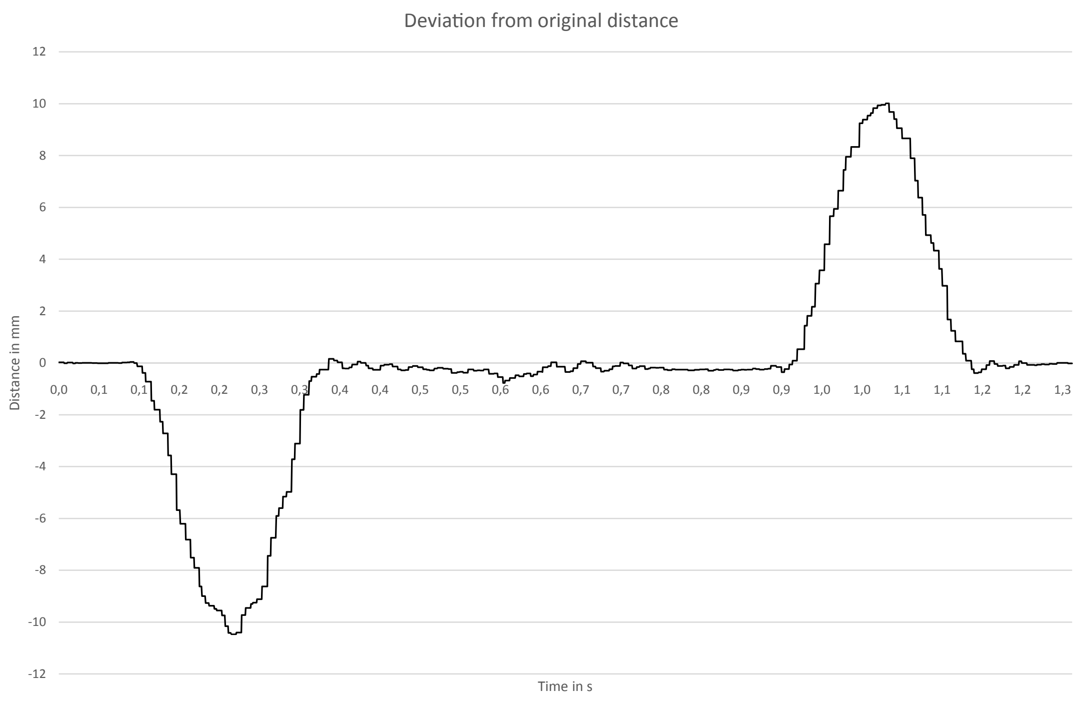


Figure 11.3.: Distance of the robots' flanges without additional synchronization

showed that both robots followed the same profile, i.e. no robot accelerated faster or slower. The KUKA robot however started to accelerate 4-8 ms later.

This behavior can be explained with two main characteristics:

- The KUKA controller performs some internal smoothing of the trajectory which takes a few milliseconds.
- The Stäubli robot expects new set-points always for the upcoming SYNC0 event. Due to the distributed clocks feature of the EtherCAT bus, the time of this event is known in advance. For the KUKA robot however, no such information is available.

In order to improve the performance of the system, the time the set-points are expected for by the KUKA controller has been set several milliseconds to the future (cf. variable t in Eq. (9.1) on Page 143). During several experiments a value of 6 ms has shown to provide best results. New measurements show that the deviation of the distance between the flanges is now reduced to $\pm$ 1 mm. The resulting diagram can be seen in Fig. 11.5. Using the available measuring technology, results much smaller than 1 mm cannot be expected.

To further increase the synchronization performance, the robots need to be synchronized on a lower level. Both systems currently are controlled with a cycle time of 4 ms, however these cycles are not synchronized to each other. Since the KUKA controller does not

(a) Vicon MX-T40s camera

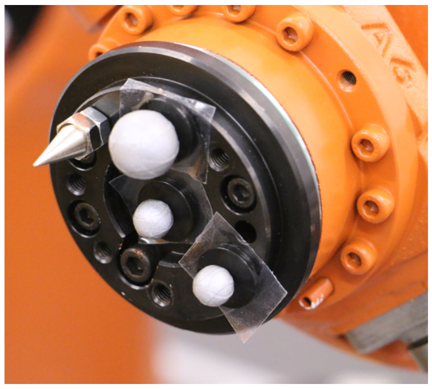

(b) Infrared reflecting markers attached to the flange of the robot

Figure 11.4.: Motion capturing system used to track both robots' flanges

provide information about when set-points are required, this time can only be estimated within the time frame of one cycle. With a top speed of 2 m/s, 4 ms time difference can lead up to 8 mm of position difference. The distributed clock feature of the EtherCAT fieldbus would greatly help for synchronization, since all devices connected to the bus will expect new set-points at the same time (with only a jitter of ≤ 1 µs, cf. Section 9.4.1).

The application has been programmed using the *MultiArmRobot* concept of the Robotics API (cf. [1, Section 10.6]). The *MultiArmRobot* provides a virtual actuator which can be used in robotics applications like any other robot, albeit only motions in operation space are possible. The aggregated robots keep the relations of their flanges with respect to each other during all motions executed with the virtual actuator. Internally, the motion is planned and distributed to all participating robots. Using a *TransactionCommand*, the start of all robot motions is synchronized real-time safely. The *MultiArmRobot* takes care to plan motions with appropriate velocities and accelerations so that all robots are capable of executing the motion. Thus programming applications with multiple robots do not differ from single-robot applications; the necessary synchronization is achieved by automatically creating a suitable primitive net combing all actuators.

The "Factory 2020" demonstration was the final demonstrator for the SoftRobot project and included many of the challenges for modern robotics applications. In particular, cooperative manipulation using two robots was required. The task was to pick up heavy work-pieces from an automated guided vehicle to a platform for further processing. Two KUKA lightweight robots were used to simultaneously lift the work-pieces to split up the weight between two robots. After all work-pieces have been transferred, two parts had to be assembled using an electric screw-driver. After both robots cooperated in assembling the work-piece, one of the robots held the piece while the other inserted the screw, using the built-in force-torque sensors to locate the screw. Using this demonstrator, the capabilities of the SoftRobot platform for real-time synchronized motions as well as

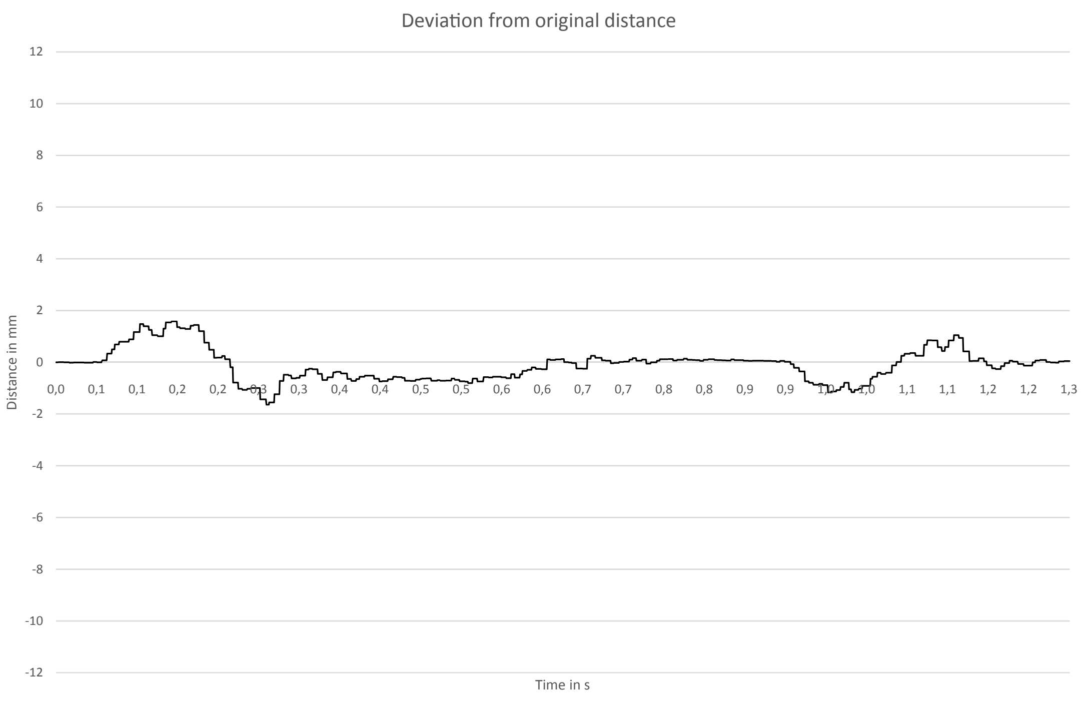


Figure 11.5.: Distance of the robots' flanges with additional 7 ms difference between both trajectories

sensor-based, cooperative assembly tasks could be shown. A video[1] of the demonstrator is available, and a more in-depth analysis of the tasks can be found in [3, Section 7.1.3] and [1, Section 11.3].

11.2.2. Real-time motion blending with I/O

The KUKA robot language (KRL) uses the concept of advance execution to support motion blending, i.e. to allow one motion to be blended into the next without being completed (cf. Section 2.5). Advance execution means that the program code interpretation (advance program counter) advances the currently executed motion (main program counter) several steps (in KRL: at most 5 motion steps). During the interpretation of a motion command by the advance program counter, the trajectory is already planned. Because of this, modifying any variable a motion later relies on (but prior to the motion's real execution) does not affect the motion at all. Thus any non-motion related program statements (logic, arithmetic) are also interpreted completely during the advance run.

If a KRL program requires information from the environment (e.g. the value of a digital input), this information can not be retrieved during the advance run, because the

[1] `http://video.isse.de/factory`

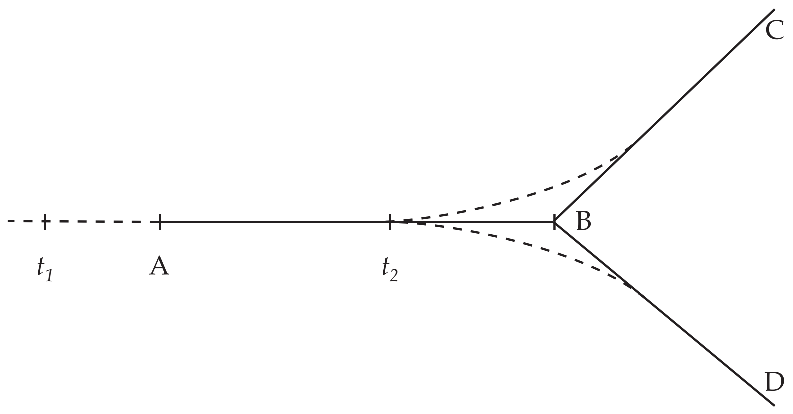


Figure 11.6.: Decision points t_1 (advance execution) and t_2 (last possible point of decision) for a blended motion from point A over B to either C or D

information might have changed until the main run reaches the same position. Therefore the KRL interpreter by default stops advance execution upon such a command and waits until the main run has caught up. This allows the current value for environment information to be used, but prevents motion blending to work. It is possible to allow the use of environment information without stopping the advance execution; this however leads to the usage of possibly outdated environment data.

An example for a motion depending on the evaluation of the environment is drafted in Fig. 11.6. Depending on the outcome of the evaluation, either a motion from point A over B to C or from A over B to D is desired. Both motion parts shall be blended into each other, thus two possible trajectories for the blended motion exist (dashed, curved lines around point B). Using KRL advance execution, it is either possible to disable motion blending and take the decision after point B has been reached and the robot has stopped. Or otherwise if advance execution is forced, the decision will be taken at some point t_1 before the motion has even reached point A. Later changes in the environment cannot be taken into account for the decision. It is not possible to define when or where exactly t_1 occurs. Using the real-time primitives interface together with the synchronization rules, it is possible to delay the time of decision to point t_2 which is the last point where/when a decision can be made, i.e. when the trajectories start to diverge.

It is possible to pack the whole motion from A to either C or D into a single primitive net, which internally checks the environment and takes the proper decision. Using this method, it can be guaranteed that the decision will be made at point t_2, and that motion blending will be performed reliably. For the same reasons as explained in Chapter 6 however, it is desirable to split the motion up into separate primitive nets which can be combined using the synchronization mechanism. Using this approach, three primitive nets are necessary: the first for the motion from A to B, including the environment monitoring and one primitive net each for the possible motions from t_2 to C and from t_2 to D.

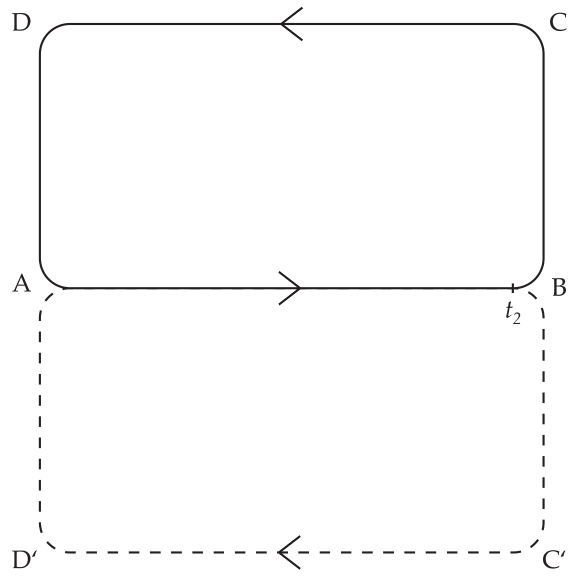


Figure 11.7.: Real-time I/O demonstration: solid rectangle ABCD is performed if push-button is not pressed at point t_2, rectangle ABC'D' otherwise

Do demonstrate the real-time decision taking capabilities of the Robotics API and the SoftRobot RCC, an application has been created that moves a KUKA lightweight robot continuously on four linear motions arranged in a rectangle (points ABCD in Fig. 11.7). Each motion is blended into the next motion, thus the robot never stops. Right before point B of the rectangle, the state of a push-button connected via fieldbus is evaluated. If the button is not pushed, the motion continues as usual. If the button is pushed at the moment the robot reaches point t_2, a different rectangle with points ABC'D' is performed. To notify the user of an imminent decision, three traffic-light like lamps are used that count down to point t_2.

Two results could be observed with this experiment:

1. During long, continuous execution, motion blending was done almost on every point, even though motion blending is only performed on a best-effort base. However, there have been rare cases when the robot stopped at one point of the rectangle, in particular when the computer executing the Java program was under high load.
2. The decision which rectangle to perform could be delayed until the last possible point in time. Unlike in KRL programs, it was perfectly sufficient to start pressing the button moments before point t_2 was reached, in particular long after the motion from A to B had started.

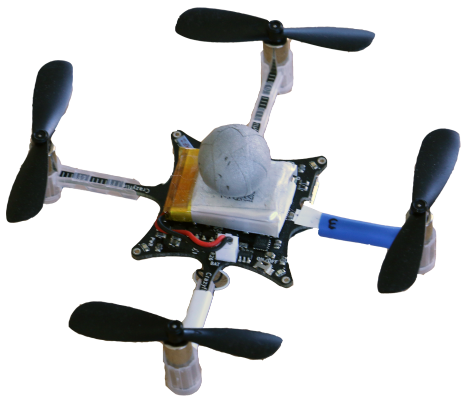


Figure 11.8.: Crazyflie miniature quadcopter with infrared reflecting marker for position tracking

11.2.3. Implementing closed-loop controllers using RPI: Controlling a quadcopter

The crazyflie is a miniature quadcopter, a helicopter equipped with four propellers driven by electric motors (cf. Fig. 11.8). Quadcopters fly by controlling the thrust of each propeller individually. By increasing and decreasing the thrust of appropriate propellers, the quadcopter can rotate around each axis and thus move in all directions. A quadcopter can stay motionless in the air when it is completely horizontal and the torques of all four motors sum to zero. In order to stay stable in the air, a quadcopter requires accelerometers to measure its position in relation to the gravity force of the earth. The crazyflie has an in-built controller which allows the user to specify the desired roll and pitch angles, a yaw position and the overall thrust (to raise or sink). Using the accelerometer, the thrust required to achieve the given parameters for each propeller is calculated internally.

Although the internal controller enables the quadcopter to hover stably in the air, the quadcopter will not rest at the same position for a long time. Since a flying object cannot directly measure its velocity with respect to the earth, the quadcopter will eventually drift due do the air draft. In order to control the absolute position of a quadcopter, an external positioning system is necessary. This can be achieved both by externally monitoring the quadcopter as well as the quadcopter monitoring known fixed points in the environment. For a student project at the Institute for Software and Systems Engineering, four Vicon MX-T40s cameras (cf. Fig. 11.4a) and an infrared reflecting marker on the crazyflie have been used. The Vicon system reports the current position of the marker in Cartesian space with high frequency, however no information about the orientation is available. For measuring the orientation, at least three IR markers are required. These are already too heavy for the small quadcopter. To mitigate this issue, the built-in compass has been used.

Figure 11.9 shows an overview of the cascaded closed-loop controller used in this project. The desired position x_{des} and velocity v_{des} of the crazyflie is provided as input. Using the

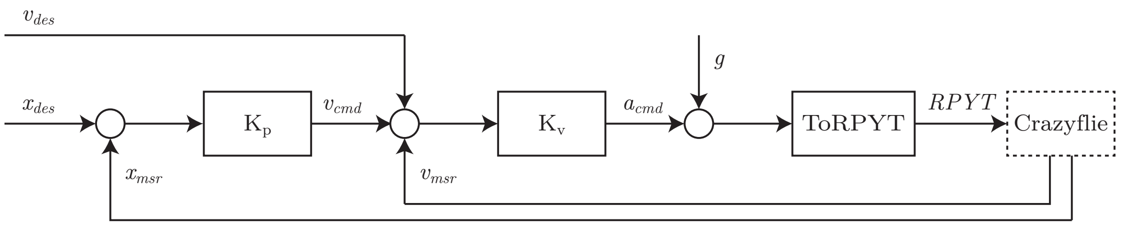


Figure 11.9.: Cascaded closed-loop controller for crazyflie quadcopter

measured position x_{msr} the first P-controller K_p creates the velocity to command v_{cmd}. Together with the desired velocity and the measured real velocity v_{msr} the second layer P-controller K_v creates the acceleration to command a_{cmd}. The resulting acceleration must be extended by the constant gravity force g before it can be converted into roll, pitch, yaw and thrust values for the crazyflie. Depending on the coordinate system used, an adjustment for the current yaw angle is necessary before the single components of the acceleration vector can be applied to roll, pitch and thrust of the crazyflie. The *ToRPYT* block in Fig. 11.9 performs this task together with the calculation of the required angles from the requested accelerations (using the trigonometric functions *asin* and *atan*).

The application has been developed in Java. The closed loop controller is mapped to a primitive net which is specified using the proxy objects provided by the Robotics API. A generic primitive for PID control has been implemented which is used for the position and velocity controller from Fig. 11.9. In this demonstration it was sufficient to use only the proportional part of the controller, the integrative and derivative parts were deactivated. The resulting primitive net does not show the closed loop of the controller, but the measured values are rather read from a sensor primitive connected to the tracking system.

Figure 11.10 shows the primitive net used to control the crazyflie, rendered using the *dot* tool from the *graphviz* package. The upper right part contains primitives to retrieve the current position measurements from the Vicon tracking system and the compass reading from the crazyflie itself. Using several timing primitives, the current velocity is calculated from the position value of the last primitive net execution cycle. The *Core::DoubleNetcommIn* primitives serve as communication interface to the application for supplying the target position in X, Y and Z coordinates. Together with the current position, these values are fed into three primitives of type *Ctrl::PID* which contain the PID controller logic (although only the P component was used in this project). The resulting velocities are combined with the additional target velocities and the current measured velocities and fed into the second set of *Ctrl::PID* primitives for the velocity controller. These primitives use the same PID controller as the position controller, albeit with other P parameters. The resulting acceleration vector is still in the world coordinate system, the crazyflie however might be rotated around the world's Z axis (yaw). The current rotation of the crazyflie is retrieved by comparing the current compass reading of the crazyflie with the known orientation of the world. The acceleration is converted

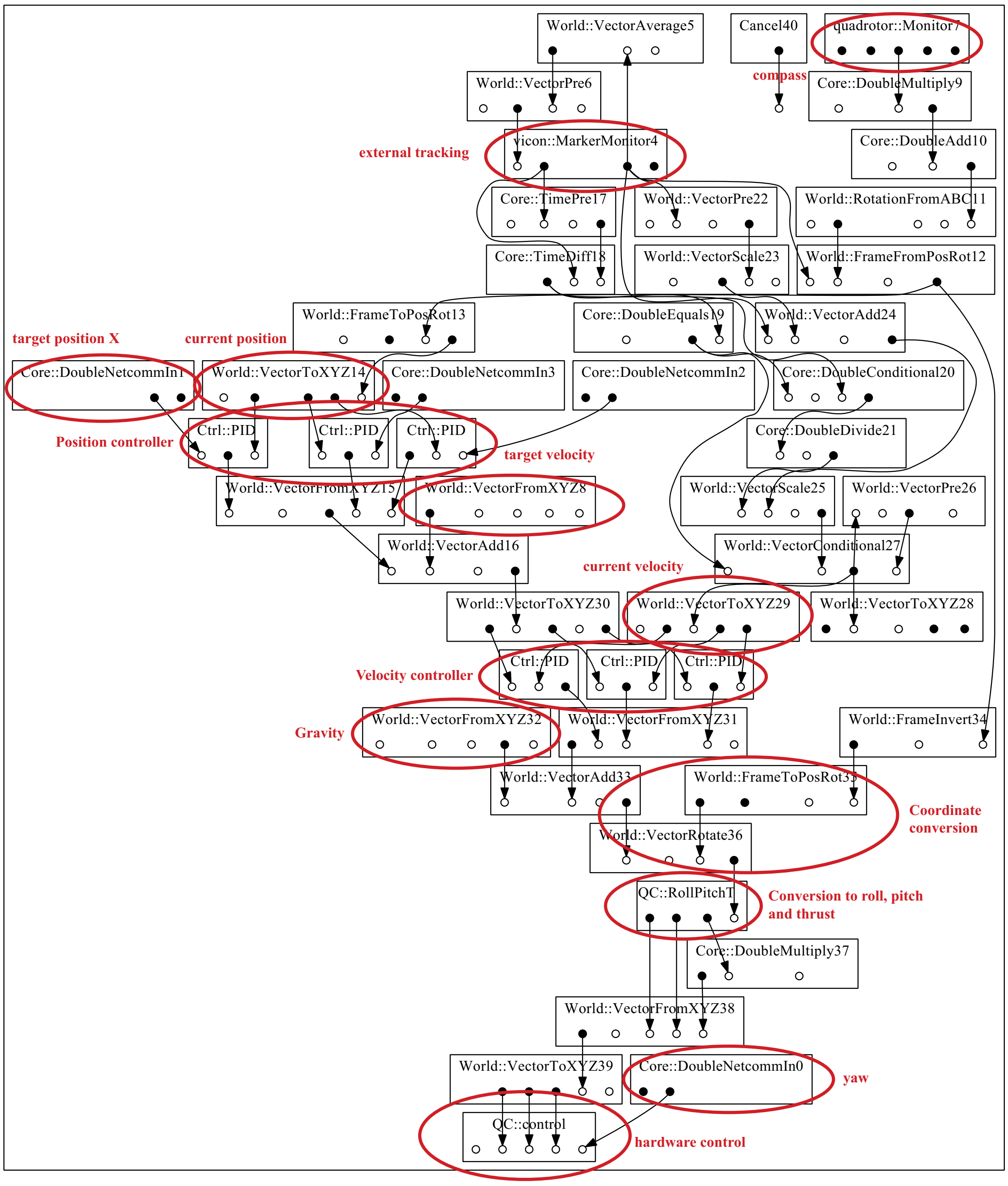


Figure 11.10.: Primitive net for a closed-loop controller for a quadcopter

to the local coordinate system of the crazyflie before it is fed into the *QC:RollPitchT* primitive that calculates the roll, pitch and thrust values from the acceleration vector using *asin* and *atan* functions. The resulting values are fed into the controller primitive of the crazyflie. The yaw angle is directly read from the application, since the crazyflie internally controls the yaw angle using the compass, therefore no controller is required.

The student project clearly showed that primitive nets can be used to implement closed-loop controllers. The control frequency depends on the primitive net execution cycle which defaults to 2 ms on the SoftRobot RCC; however this time is configurable. For the crazyflie, 2 ms have been fast enough for good quality results. For this example, specific stateful PID-controller primitives have been used. It would also be possible to create a fragment that performs the same calculations using multiplication, division and addition primitives, and several *Pre* would be required for the derivative and integral parts of the controller to “store” values from the previous execution cycle. Embedding the controller (consisting of only 7 lines of C++ code) into a primitive therefore simplifies the primitive net slightly.

11.3. Hardware extensibility: KUKA robots using RSI

A key requirement for a modern software system is its extensibility. The Robotics API Development Platform including the SoftRobot RCC has been developed with this requirement in mind. The Real-time Primitives Interface provides a flexible interface for the specification of real-time tasks, and the SoftRobot RCC is easily extendable with new hardware devices. This extensibility is evaluated exemplary for the development and testing of a driver for a new robot manipulator.

The KUKA lightweight robot (LWR) has been supported previously by the SoftRobot RCC (cf. Section 9.4.4) using the Fast Research Interface (FRI) which allows providing set-points in both joint space as well as Cartesian space with a cycle time up to 1 ms. Using FRI, the KUKA controller is no longer performing motion planning or executing the robotics application itself. Other “standard” KUKA robots do not offer the FRI option and are generally only used by executing KRL programs directly on the KUKA controller. To control standard KUKA robots with the SoftRobot RCC, an interface similar to FRI is required. KUKA offers the Robot Sensor Interface (RSI) [82] option which is intended to overlay a motion programmed in KRL with external correction data, e.g. using a camera-based system to correct the position of a pre-programmed welding seam. However, it is also possible to use RSI without a pre-programmed KRL-based motion, but rather specifying the whole motion through set-points, much like using FRI. RSI supports six joints for the robot arm itself as well as up to six additional so-called external joints which can be used e.g. for a linear unit or a turn-and-tilt table.

Starting with the KRC-4 controller from KUKA, RSI is capable of receiving new set-points with a cycle time of 4 ms. Optionally it is also possible to use a cycle time of 12 ms with the KRC internally interpolating the motion. Communication between the external motion controller (in this study the RCC) and the KUKA controller is done using UDP/IP on a dedicated network segment. In each control cycle (i.e. usually every 4 ms) the KUKA controller sends a UDP packet containing XML encoded data to the external controller and expects a reply within the cycle time with a new set-point. If a configurable amount of replies is lost or received late, the KUKA controller deactivates the RSI connection and brakes the manipulator with an emergency stop. Hard real-time is required on the external motion controller in order to reliably be able to reply within 4 ms.

The KUKA controller needs some configuration data for communication with an external system. The configuration consists of two parts, the “RSI diagram” which describes communication connections among different “RSI objects”, and a configuration specifying the format of the XML telegrams exchanged between the KUKA controller and the external motion controller. The RSI option is not only capable of using a network connection for correction data, but can also use other data available such as provided by a fieldbus. So-called “RSI objects” are available to read from and write to input and outputs. Furthermore, an object for Ethernet communication and objects which apply correction values to the current motion in joint or Cartesian space are available. Using the RSI diagram, those objects can be connected to each other, much like primitives can be connected in a primitive net. The Ethernet object offers a variable number of input and output ports which can be connected to any other object to transmit data from the KRC to the external controller (e.g. values from fieldbus inputs) and to receive data from the external controller for further processing (e.g. new set-points or new values for fieldbus outputs). The second configuration file defines the mapping of the input and output ports of the Ethernet object to XML attributes in the telegrams, and further communication properties such as IP address and port.

Besides both static configuration files, a KRL program is also required, even if the motion is completely remote controlled. The KRL program is responsible for loading the configuration files and initiating the RSI connection to the external controller (the KRC is always the UDP client, while the external motion controller provides a UDP server socket). The KRL program can activate external control by using a dedicated KRL command which blocks until the external controller signals its desire to terminate control by setting the Boolean input port of the RSI stop object to *true*. After terminating external control, the KRL program continues with normal program execution. Using RSI it is possible to provide set-points for the actuator (and up to six optional external axes) and to set fieldbus outputs in real-time. However, it is not possible to update certain internal configuration data of the KRC such as tool load data (i.e. mass, center of mass and moment of inertia of the attached tool) which is required by the KRC for precise hardware control. This information can only be updated by means of the executed KRL program, hence the external motion controller needs to interact with the KRL program. This interaction can only be performed while external control is turned off, in particular while the manipulator is not in motion.

```
<Rob Type="KUKA">
   <RIst X="445.0" Y="0.0" Z="810.0" A="-180.0" B="0.0" C="180.0" />
   <RSol X="445.0" Y="0.0" Z="810.0" A="-180.0" B="0.0" C="-180.0" />
   <AIPos A1="0.0" A2="-90.0" A3="90.0" A4="0.0" A5="90.0" A6="0.0" />
   <ASPos A1="0.0" A2="-90.0" A3="90.0" A4="0.0" A5="90.0" A6="0.0" />
   <EIPos E1="0.0" E2="0.0" E3="0.0" E4="0.0" E5="0.0" E6="0.0" />
   <ESPos E1="0.0" E2="0.0" E3="0.0" E4="0.0" E5="0.0" E6="0.0" />
   <MACur A1="0.0" A2="0.0" A3="0.0" A4="0.0" A5="0.0" A6="0.0" />
   <MECur E1="0.0" E2="0.0" E3="0.0" E4="0.0" E5="0.0" E6="0.0" />
   <IPOC>4485717</IPOC>
</Rob>
```

Listing 11.1: Example RSI XML packet transmitted by the KUKA KRC-4 controller

```
<Sen Type="ImFree">
   <EStr>SoftRobotRCC running</EStr>
   <AKorr A1="12.5" A2="33.2" A3="12.33" A4="0.0" A5="5.4" A6="0.0" />
   <EKorr E1="0.0" E2="0.0" E3="0.0" E4="0.0" E5="0.0" E6="0.0"/>
   <Stop>0</Stop>
   <Tool X="0.0" Y="10.0" Z="40.0" M="5" />
   <IPOC>4485717</IPOC>
</Sen>
```

Listing 11.2: Example RSI XML response generated by the SoftRobot RCC

RSI integration in the SoftRobot RCC

To integrate KUKA robots using RSI, a SoftRobot RCC driver for the RSI protocol had to be developed. The KUKA controller regularly transmits UDP packets containing XML telegrams with the current joint positions to the external motion controller. Each packet must be responded to by sending new set-points, also encoded in XML.

Listing 11.1 shows an exemplary packet as received from the KUKA KRC-4 controller. The first two lines describe the Cartesian position of the robot with regard to the tool and base which is currently selected on the KUKA controller. `RIst` describes the measured position, while `RSol` describes the commanded position, which does not necessarily have to be reached yet. The following lines describe the currently measured position of the robot's joints (`AIPos`) and the external joints (`EIPos`) as well as the commanded positions (`ASPos` and `ESPos`). The tags `MACur` and `MECur` list the electrical currents applied to the corresponding motors. Finally the tag `IPOC` contains a strict monotonic increasing counter value which must be included in the response to allow the KUKA controller to detect missed packets. All units are in mm or degrees. Additional data such as the values of inputs can be configured to be included in the XML telegram.

The external motion controller needs to reply immediately to each packet received from the KRC. Listing 11.2 shows an exemplary reply. It contains a status message which

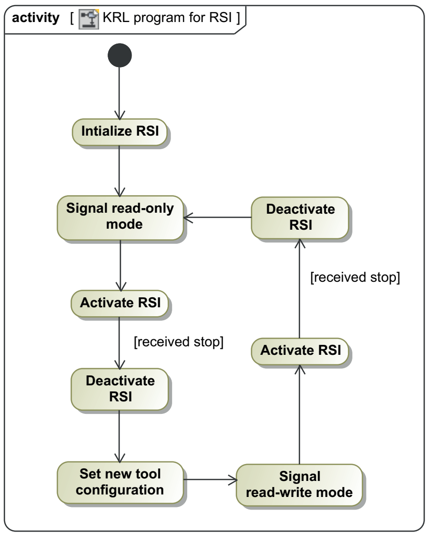


Figure 11.11.: UML activity diagram of KRL program which controls the RSI connection to the external motion controller.

is displayed on the KUKA teach pendant, set-points for the manipulator (`AKorr`) and the external joints (`EKorr`), the stop request bit, tool configuration and the counter value copied from the received packet which is currently being answered. When the stop bit is set to “1”, the KRL program terminates remote control and continues with normal program execution. It should be noted that all set-points provided to the KUKA controller have to be relative to the position where the remote connection has been activated, while the measured positions transmitted to the external motion controller are absolute values.

Because the RSI protocol expects set-points relative to the position of the robot when remote connection is started and does not allow changes to tool load configuration while remote control is enabled, the SoftRobot RCC uses a two step process for initializing the RSI connection. During the first step, a read-only connection to the KUKA controller is established which is used to read the current position of the robot and to transmit the desired tool load configuration into temporary variables on the KUKA controller. The KRL program subsequently activates the new tool configuration and allows the SoftRobot RCC to enter step two which performs the actual active external control of the system. Figure 11.11 shows the program flow of the KRL program which is used to interact between the KUKA controller and the SoftRobot RCC. The main loop first activates read-only mode by transmitting an appropriate flag to the SoftRobot RCC, which then internally stores the current position of each joint and replies with all joint corrections set to zero, the desired tool configuration and an active stop request. As soon as the KRL program recognizes the stop request, remote control is deactivated, the new tool configuration is loaded and the read-only mode disabled. Afterward remote control

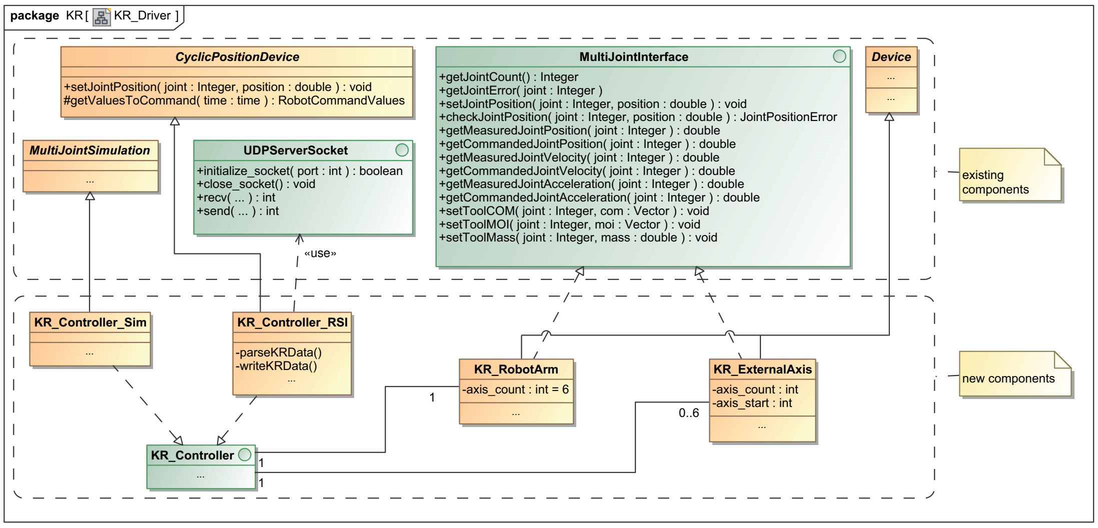

package KR [KR_Driver]
CyclicPositionDevice
+setJointPosition(joint : Integer, position : double) : void
#getValuesToCommand(time : time) : RobotCommandValues
MultiJointInterface
+getJointCount() : Integer
+getJointError(joint : Integer)
+setJointPosition(joint : Integer, position : double) : void
+checkJointPosition(joint : Integer, position : double) : JointPositionError
+getMeasuredJointPosition(joint : Integer) : double
+getCommandedJointPosition(joint : Integer) : double
+getMeasuredJointVelocity(joint : Integer) : double
+getCommandedJointVelocity(joint : Integer) : double
+getMeasuredJointAcceleration(joint : Integer) : double
+getCommandedJointAcceleration(joint : Integer) : double
+setToolCOM(joint : Integer, com : Vector) : void
+setToolMOI(joint : Integer, moi : Vector) : void
+setToolMass(joint : Integer, mass : double) : void
Device
...
...
MultiJointSimulation
...
UDPServerSocket
+initialize_socket(port : int) : boolean
+close_socket() : void
+recv(...) : int
+send(...) : int
«use»
existing components
KR_Controller_Sim
...
KR_Controller_RSI
-parseKRData()
-writeKRData()
...
KR_RobotArm
-axis_count : int = 6
...
KR_ExternalAxis
-axis_count : int
-axis_start : int
...
new components
1
0..6
KR_Controller
...
1
1

Figure 11.12.: UML class diagram for KR driver

is enabled once again. The SoftRobot RCC now transmits new set-points relatively to the previously stored initial position and does not signal a stop request. Only once the tool configuration needs to be changed, another stop is requested which makes the KRL program stop remote control and restart the main loop.

Figure 11.12 shows a UML class diagram of the RSI driver implementation in the SoftRobot RCC. Classes in the upper part of the diagram have already existed, only the four classes and one interface in the lower part of the diagram had to be created. The first class *KR_Controller_RSI* contains the implementation of the RSI protocol as described above. It uses an existing implementation of a UDP server socket for communication with the KUKA controller, and inherits from the *CyclicPositionDevice* class (cf. Section 9.4.3 and Fig. 9.7) for interpolation of the set-points provided by the primitive net. The RSI driver creates its own real-time thread for reliable communication with the KUKA controller. A second class *KR_Controller_Sim* also implements the generic *KR_Controller* and provides simulation support. The RSI driver and the simulation driver are exchangeable and thus allow for easy testing of robotics applications for KUKA KR robots without having to use a real robot.

The class *KR_RobotArm* provides the device for the 6-DOF robot arm and the class *KR_ExternalAxis* the device for external axes. Both classes implement the *MultiJointInterface* (cf. Section 9.2) and thus can be used with the same primitives (and therefore also the same applications) as all other robots. The robot arm device always controls six joints and has a direct connection to the RSI driver. Because a standard KUKA controller always controls a single robot, each *KR_RobotArm* is associated with exactly one *KR_Controller* and vice versa. Up to six additional external axes can be controlled using a single KUKA controller. Therefore instances of the *KR_ExternalAxis* class have

Robot	Robot controller	Software versions	
KUKA KR6 R900 sixx	KR C4 compact	KSS 8.2.3,	RSI 3.1
KUKA KR16-3	KR C4	KSS 8.2.22,	RSI 3.1
KUKA KR210 R3100 ultra	KR C4	KSS 8.3,	RSI 3.2

Table 11.3.: Hardware and software for which the RSI implementation has been tested.

an association with a *KR_Controller*, but a controller may have up to six external axes connected. Each external axis object must be configured with the number of joints (e.g. a linear axis has only one joint, while a turn-and-tilt table uses two joints) and the number of the first joint used (to allow connecting e.g. a linear axis as external joint 1 and the turn-and-tilt table as joints 2 and 3).

Many of the functionality required to implement the *MultiJointInterface* can be delegated to the RSI driver which inherits the handling of set-points from the *CyclicPositionDevice*. Some features are specific to KUKA controller or the RSI protocol (e.g. the mechanism for changing tool load data) and thus are newly implemented in the RSI driver.

Evaluation of the RSI integration

A driver for the RSI protocol has been integrated in the SoftRobot RCC and supports a broad variety of KUKA industrial robots. As previously described, only very few parts had to be developed from scratch and thus the integration could be completed within 4 working days by one developer (with some prior knowledge of KUKA robots, the RSI protocol and the KRL programming language). Besides the development of the RSI driver and the two devices in the RCC, also a Java representation of the KUKA robots had to be created, which took less than half a working day. The Robotics API already had support for robots using the *MultiJointInterface*, thus it was sufficient to create a new class which precisely describes the new manipulator (with its Denavit-Hartenberg parameters, mass, etc.) and depends on the existing device driver.

The RSI integration has been tested with robots of different sizes, ranging from systems for small payloads (up to 6 kg) up to large payloads (210 kg). The tested hardware is listed in Table 11.3. All robots have been successfully tested with point-to-point, linear and blended motions with velocities up to the maximum joint velocity as allowed according to the technical information of the robot systems.

Overall, the effort required for the integration of a new type of robotics controller has been very low. Many generic functionality could be re-used and only few hardware specifics had to be developed from scratch. Table 11.4 shows some statistical data of the RSI driver implementation. Although lines of code may not be the perfect metric for reusability, it provides an indication of the effort required for the development. More than 60% of the overall code required for the implementation of the KR driver could be reused from generic implementations. On top of the components shown, the RPI

New components	**Lines of code**
RSI driver (including both devices)	671
Simulation of KR robots	48
Real-time (inverse) kinematics	213
KRL code	88
Total	1.020

Reused components	**Lines of code**
MultiJointInterface, CyclicPositionDevice	860
Generic (inverse) kinematics components	380
UDP communication infrastructure	310
Total	1.550

Table 11.4.: Statistical data of RSI driver implementation

execution environment consists of approximately 15.000 lines of code which is completely independent of any hardware specific driver and is used for every robotics application.

11.4. Realization of requirements

In Section 4.1 a set of requirements has been identified which must be achieved with the SoftRobot architecture. In this section, the requirements are reviewed again and their realization is analyzed.

1. *Usability:* The real-time aspects of robotics applications are completely hidden by the separation of the real-time safe RCC, and non real-time robotics applications. The Real-time Primitives Interface provides means to specify tasks which need to be real-time safe (e.g. synchronized tool commands), and the Robotics API event mechanism will automatically be translated into real-time safe primitive nets.

   The activity layer has been introduced to create an easy-to-use programming interface. Using this layer, it is possible to create standard applications (containing an industrial robot and several tools) with a syntax that is very similar to the specialized domain specific languages currently in use for robot programming. Despite this easy-to-use syntax, it is still possible to use the advanced and real-time safe event mechanism of the Robotics API.

2. *Multi-robot systems:* The SoftRobot RCC supports multiple actuators simultaneously, only limited by the available resources (memory, computing performance). The simultaneous control of four industrial robots with a single RCC has been successfully tested in the lab. The RCC allows access from several applications simultaneously, and ensures that no device is accessed contradictorily using a resource concept. A single application can also control multiple actuators, and the synchronous execution semantics of primitive nets automatically synchronizes all

actuators controlled by a single instance. Thus it is both possible to use multiple programs for multiple robots, as well as a single program for multiple robots.

Systems consisting of two robots have also been evaluated in Section 11.2.1. Motion synchronization is possible to the degree that is supported by the underlying hardware controllers, i.e. some form of time synchronization among all controllers is required.

3. *Sensor support:* The sensor and event concepts of the Robotics API allow for an easy integration of sensors into robotics applications. The evaluation of sensor values can be programmed using Java and is automatically transformed into primitive nets for a real-time safe execution. This allows to use sensors to influence robot trajectories while they are executed. Triggering events can happen as late as possible, i.e. it is not necessary to evaluate sensor values prior to starting a motion as it is required with several current robot systems (cf. Section 11.2.2).
4. *Extensibility:* Both the SoftRobot RCC as well as the Robotics API can be flexibly extended with new features. Such features can be new hardware devices, but also new algorithms for path planning, reactions to sensor events, etc. The SoftRobot RCC provides a flexible system of generic hardware interfaces which allows to integrate new hardware devices and minimizes the need to modify existing robotics applications. An example of the integration of a new robot system has been evaluated in Section 11.3.
5. *Special industrial robotics concepts:* Motion blending, as well as force and torque based manipulation tasks are supported by the Robotics API and the SoftRobot RCC. Motion blending can be achieved by creating multiple successive primitive nets and joining them by supplying appropriate synchronization rules. This allows the main program flow to remain in the robotics application rather than having to embed the whole chain of motions into a single real-time task. Creating primitive nets one after another can only be performed on a best-effort base without real-time capabilities of the application, however the system architecture ensures that at no time the system is left out of active control. If a successive primitive net cannot be loaded in time, the previous primitive net will either complete the motion to standstill, or, in the case of force-based manipulation tasks, keeps running and controlling the robot until eventually the successor is started. Motion blending using multiple successive and independent primitive nets has been evaluated in Section 11.2.2.

All requirements that have been identified at the beginning of the SoftRobot project could be fulfilled. The SoftRobot RCC provides the real-time execution environment, and the Robotics API provides a layered programming interface. Using either the command or the activity layer, complex, sensor-based multi-robot applications can be created. The whole architecture is neither limited to a specific set of hardware devices nor to a predefined set of tasks; it can be flexibly extended with new hardware, control algorithms etc.

Chapter 12

Conclusion

In this work, a novel architecture for industrial robot programming has been introduced, in particular the base layers that enable a real-time safe execution of robot programs have been explained. In this chapter, first the results achieved in this work are summarized and then an overview of current and future projects building upon the results is given. Finally an outlook to future extensions and applications of the Real-time Primitives Interface and the SoftRobot RCC with respect to distribution for mobile or large scale systems is provided.

12.1. Summary

Industrial robots nowadays are usually programmed with proprietary, manufacturer-dependent programming languages. In Chapter 1, advantages of modern software engineering methods, in particular the application of object-oriented design, for the robotics domain have been introduced. Basic concepts of industrial robotics have been explained in Chapter 2.

One major issue preventing an easy application of modern general-purpose programming languages to the industrial robotics domain is the inherent requirement for real-time safe execution of robot programs, which cannot be guaranteed by most programming languages, mainly due to automated memory management and a high level of hardware abstraction. These real-time requirements have been analyzed in detail in Chapter 3 and it was found that most robotics applications in fact do not require hard real-time during the whole duration of their execution. Every single motion must be planned and executed with exact timing guarantees, otherwise fast, precise and smooth motions cannot be performed. Interaction with peripheral tools such as welding torches, grippers, etc. also needs to be synchronized with the main actuator. Between two subsequent motions however, soft real-time is sufficient, i.e. having the robot stop for some tenths of a second more or less will not break the system, but still reduce the overall performance of the system and thus should be avoided when possible. A major finding was that it is possible, in general, to partition robotics applications into small parts which inherently require hard real-time, but which then can be combined to form the overall program flow with only soft real-time requirements.

Based on the results of the previous chapter, the architecture developed during the SoftRobot project is presented in Chapter 4. A three tier architecture has been created which provides real-time safe execution of tasks with the Robot Control Core as the base layer. The middle tier is represented by the Robotics API, which provides an object-oriented programming model for robotics applications, and automatically creates real-time tasks for execution by the Robot Control Core. The top tier consists of individual robotics applications which can use the Robotics API directly as a programming interface; furthermore it is possible to create application specific programming interfaces by creating domain specific languages or even using a service oriented architecture for cell level control.

The Real-time Primitives Interface (RPI) introduced in Chapter 5 allows the specification of real-time tasks for execution on the Robot Control Core. The basic concept is the primitive net, which is a data-flow graph consisting of multiple so-called primitives, which are interconnected with links. Primitives provide basic calculation functions which can be combined to large algorithms using the appropriate links. Primitive nets are executed cyclically and every primitive is executed once per execution cycle, unless explicitly deactivated. The synchronous execution of all primitives allows for an implicit synchronization of sensors and multiple actuators. All tasks that are embedded in a primitive net are executed with precise timing guarantees on a real-time system.

While all tasks embedded in a single primitive net are executed real-time safely, it is still desirable to switch from one primitive net to another with timing guaranteed. The switching condition can be specified using the synchronization rules that are explained in Chapter 6. These rules consist of a logical condition when to switch from one set of primitive nets to another set. Switching occurs instantaneously (i.e. with hard real-time) and only if all new primitive nets are ready for starting. Otherwise the old set of primitive nets will continue running and thus keep the system in a safe state. Synchronization rules can be used for example to achieve motion blending while keeping the program flow in the main robotics application, and also force/torque based manipulation tasks containing several steps can be implemented safely across multiple primitive nets.

The SoftRobot RCC, a reference implementation for the Robot Control Core concept has been created, and is explained in Chapters 7 to 9. The reference implementation has been written in C++ and is executed on the Linux operating system with Xenomai real-time extensions to ensure real-time safety. Chapter 7 explained the execution environment for single primitive nets, including a set of basic primitives and the communication interface to the robotics application. Chapter 8 concentrated on the synchronization mechanism for multiple primitive nets which allows to switch from one set of primitives synchronously to a new set of primitives. Special attention was paid to minimizing the number of required threads while still guaranteeing the availability of an appropriate execution thread when necessary. To control real robots, manufacturer and possibly even type specific device drivers are required. The SoftRobot RCC provides a modular and extensible system for such device drivers which was explained exemplary for several robots and some other generic periphery devices in Chapter 9. In particular, this system provides generic device interfaces so that robotics applications can be used independently of the robot manufacturer or type, as long as the hardware provides the same functionality (i.e. same number of joints). It is also possible to replace the hardware drivers by simulation drivers to test applications realistically.

The Robotics API is the primary programming interface for robotics applications in the SoftRobot project. In Chapter 10 a general overview of the command layer is provided which builds the foundation of the Robotics API. The command layer allows to create an object-oriented model of real-time tasks which can then be transformed into primitive nets using an automated mapping algorithm. Using this algorithm, tasks with multiple actuators and sensor-triggered, event-based interactions can be automatically transformed into the required data-flow graphs of primitive nets.

The results of the previous chapters are finally evaluated in Chapter 11. For this purpose, several demonstration applications have been developed. An overview of the scalability and real-time performance of the reference implementation has been provided, as well as descriptions of applications that demonstrate the synchronization of multiple robots and real-time reactions to external events. The effort for the integration of a new robot was evaluated, and finally the requirements identified in Section 4.1 have been reviewed.

Using the results of this thesis, it is now possible to embed real-time critical (robotics) tasks into programs written using managed, object-oriented programming languages. The Real-time Primitives Interface provides the base language for the real-time task specification. Complex robotics tasks including sensor-based events and cooperating robots can be created using a set of basic calculation primitives. Most of the time it is only necessary to alter the real-time execution environment to include support for new hardware devices. Robotics applications can be created purely within the managed, object-oriented programming language. The synchronization mechanism of the RPI allows switching between different real-time tasks with guaranteed maximum transition times, even allowing actuators to be in full motion or to apply force while tasks are being switched. The Robotics API command model allows to specify real-time critical tasks in an object-oriented way. The mapping algorithm presented in this thesis can automatically transform these tasks into primitive nets for real-time safe execution. Ultimately it is now possible to create robotics applications using a standard, off-the-shelf object-oriented programming language and thus to benefit from all advantages such a language provides (advanced software-engineering methods, good community support, many skilled developers, etc.). The SoftRobot RCC allows to execute primitive nets (and thus the robot programs created using the Robotics API) real-time safely using the Linux operating system. Hardware drivers for several industrial robots, sensors and further periphery devices from different manufacturers are included and allow the Robotics API to be used on real hardware.

12.2. Current applications and outlook

Overall, the Real-time Primitives Interface has proven to be a dependable base for real-time task specification during the past few years. Together with the reference implementations for the Robot Control Core as well as the Robotics API, it has been in use for several past as well as for ongoing research projects.

The Institute for Software and Systems Engineering has developed an offline programming platform in cooperation with the German Aerospce Center (DLR) for the production of carbon-fiber-reinforced plastic (CFRP) products [92]. The technique of offline programming is employed in order to reduce the overall development time for robotics applications, but in particular to reduce the time the real robot hardware is blocked from revenue production for development tasks. The offline programming platform provides a 3D-view of the robot cell and allows the developer to preview all tasks. In order to provide a realistic simulation, the Robotics API and the SoftRobot RCC have been embedded with simulation drivers. The SoftRobot RCC is responsible to execute all motions with the same profile as they would be executed with real robots. For the offline programming platform, the RCC is executed under the Windows operating system without any real-time extensions. Nevertheless, the Robotics API generates the same primitive nets as for a real system, and the RCC executes all tasks identically – although set-points are not produced with a reliable frequency. For simulation purposes, this is not a problem, the worst-case is that the simulated robot moves jerky. However, the simulated trajectories are identical to the trajectories of a real robot connected to a (real-time safe) RCC. Until now the resulting program is transformed into KRL code and then executed by the standard KUKA robot controllers. However plans exist to use the SoftRobot RCC together with the RSI connection (cf. Section 11.3) also for real hardware control. The modular design of the SoftRobot RCC allows to provide the same interface for real and simulated robots, which allows to switch from offline to online programming with only changing a configuration file.

The research project *SafeAssistance* intends to enable safe interaction of human workers and industrial robots without separating the workspace. Capacitive sensors mounted to the robot are used to detect humans (and other items in the workspace) by measuring the disturbance of an electric field. The absolute value measured by the sensors cannot be directly used to decide whether a worker (or any other obstacle) is in the robot's way. The workspace usually contains many items, in particular electrical machine tools, which also disturb the electric field. Therefore it is necessary to create a model of the working space with known items, i.e. expected sensor values must be learned for many different positions of the robot. When a robot program is executed, the current sensor readings must be compared to the recorded sensor readings to recognize any changes in the world which indicate an obstacle. Since it is impossible to record all possible positions of the robot in the working space, during the execution of a program the current position of the robot must be matched to one of the recorded positions. An algorithm to search for an approximate nearest neighbor in the FLANN library [91] has been used for this purpose. Both for creating the initial world model as well as for later comparing the world model with the current situation, it is necessary to synchronize the times when the capacitive sensors and the joint positions of the robot are read. Using the SoftRobot RCC greatly facilitates this task, since both the capacitive sensors and the robot position sensors can be included into a single primitive net which automatically synchronizes the sensor reading times. A real-time safe implementation for the approximate nearest neighbor search algorithm has been implemented in the form of a primitive. This allows to define reactions of the robot based on the comparison of the world model and the current situation inside the primitive net which is controlling the robot. These reactions can be completely defined using the event mechanism of the Robotics API and the Real-time Primitives Interface, thus they will be handled deterministically and reliably, which is a key requirement for a safety-related system.

The current mapping algorithm creates one primitive net for each task which must be executed real-time safely. Multiple actuators which need to perform tasks in cooperation are embedded into a single primitive net to benefit from the synchronization of primitives inside the net. Current research is undertaken to use the synchronization mechanism (cf. Chapter 6) to split large transactional commands into multiple primitive nets. Switching between commands can then be delegated to the RCC by supplying appropriate synchronization rules. If all primitive nets that belong to a certain transaction are loaded on the RCC before the first set of commands is started, the same real-time safety for switching can be achieved as by embedding everything into a single primitive net. This new approach yields several advantages: first, the complexity of each primitive net is dramatically decreased since every net only contains a single command. Today's primitive nets for transactional commands contain large structures to determine which command has to be active at every point in time. These structures are no longer required if the decision is encoded into logical conditions which are evaluated by the execution environment. And second, splitting multi-actuator commands into single independent primitive nets is a first step to a distributed system. In particular with mobile systems it is an interesting perspective to have every robot run its own program (in the form of primitive nets) with inter-robot synchronization still being possible (provided an appropriate communication channel is available).

Distributing large applications while still being real-time safe is a key requirement for further increasing the scalability of the SoftRobot architecture. Although several robots currently can be controlled by a single RCC, scalability is naturally limited by the available computing power. A current project at the Institute for Software and Systems Engineering is to control the novel "multifunctional cell" (MFC) which has been built in Augsburg for the German Aerospace Center (DLR)[1]. This system is intended for the automated production of carbon-fiber-reinforced plastics (CFRP) and consists of two

[1]`http://www.dlr.de/bt/desktopdefault.aspx/tabid-8372/14306_read-36171/`

KUKA KR-270 robots mounted on a common linear unit and three portal robots with each a spherical hand. If even more robots are required, two additional KR-210 robots, again mounted on a common linear unit, can be integrated into the cell. Altogether, the facility has 46 joints to control for only the robots, not including possible further actuated tools or periphery devices. Since CFRP parts can be very large, multiple robots must cooperate during the manufacturing process. To increase the flexibility of the process, it is desirable to assign robots to different groups as required. At the moment, all robots can only be programmed using the standard controllers which require one KRL program per robot. Synchronizing such a large number of systems with many independent programs is a very tedious task. The vision of the research project is to extend both the programming model of the Robotics API, as well as the real-time execution core of the SoftRobot RCC to support an easy programming of this large cell using a single Java program, or, if desired, multiple (synchronized) programs. To achieve this aim, it will be necessary to extend the current programming interface of the Robotics API to provide a convenient way to develop such large scale applications. The real-time part of the system, in particular the SoftRobot RCC will have to be distributed across multiple computers. A vision is to create a distributed version of the RPI which allows to use synchronization rules for primitive nets which are running on different systems, while still providing the same level of real-time performance as it is possible for a single system at the moment.

Appendix A

Language definitions

The SoftRobot RCC uses several languages for communicating with external systems such as a robotics application using the Robotics API. This appendix lists the grammar for each communication protocol.

A.1. XML specification of primitive nets

XSD schema definition for describing primitive nets. XML documents according to this schema are transmitted from a Robotics API application to the RCC using the HTTP protocol (cf. Section 7.6.2).

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="http://schema.isse.de/SoftRobot/RCC/RPINet.xsd"
   elementFormDefault="qualified" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns="http://schema.isse.de/SoftRobot/RCC/RPINet.xsd">
    <xsd:element name="rpinet" type="rpinet"></xsd:element>
    <xsd:complexType name="rpinet">
      <xsd:complexContent>
         <xsd:restriction base="fragment">
            <xsd:attribute name="id" type="xsd:string"
               use="optional">
            </xsd:attribute>
         </xsd:restriction>
      </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="primitive">
      <xsd:sequence>
         <xsd:element name="parameter" type="parameter" minOccurs="0"
```

```
            maxOccurs="unbounded">
         </xsd:element>
         <xsd:element name="port" type="port" minOccurs="0"
            maxOccurs="unbounded">
         </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="type" type="xsd:string" use="required"></xsd:attribute>
      <xsd:attribute name="id" type="xsd:string" use="required"></xsd:attribute>
    </xsd:complexType>
    <xsd:complexType name="parameter">
      <xsd:attribute name="name" type="xsd:string" use="required"></xsd:attribute>
      <xsd:attribute name="value" type="xsd:string" use="required"></xsd:attribute>
    </xsd:complexType>
    <xsd:complexType name="port">
      <xsd:sequence></xsd:sequence>
      <xsd:attribute name="name" type="xsd:string" use="required"></xsd:attribute>
      <xsd:attribute name="fromprimitive" type="xsd:string"
         use="required">
      </xsd:attribute>
      <xsd:attribute name="fromport" type="xsd:string"
         use="required">
      </xsd:attribute>
        <xsd:attribute name="debug" type="xsd:int"></xsd:attribute>
    </xsd:complexType>
    <xsd:complexType name="fragment">
      <xsd:sequence>
         <xsd:element name="fragment" type="fragment" minOccurs="0"
            maxOccurs="unbounded">
         </xsd:element>
         <xsd:element name="primitive" type="primitive" minOccurs="0"
            maxOccurs="unbounded">
         </xsd:element>
         <xsd:element name="outPort" type="port" minOccurs="0"
            maxOccurs="unbounded">
         </xsd:element>
         <xsd:element name="inPort" type="port" minOccurs="0" maxOccurs="unbounded"></
             xsd:element>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:string" use="required"></xsd:attribute>
    </xsd:complexType>
</xsd:schema>
```

A.2. Grammar for DirectIO protocol

The DirectIO protocol is used for communication between a robotics application and the RCC. It is used for creating new primitive nets and controlling their execution, as well as for other communication. The full grammar in extended Backus-Naur form (EBNF) follows:

⟨*statement*⟩ ::= ⟨*identifier*⟩ '`=`' ⟨*identifier*⟩ '`(`' [⟨*parameter*⟩ { '`,`' ⟨*parameter*⟩ }] '`)`' ;

⟨*parameter*⟩ ::= ⟨*literal*⟩
| '`[`' [⟨*parameter*⟩ { '`,`' ⟨*parameter*⟩ }] '`]`'
| '`{`' ⟨*keyvalue*⟩ { '`,`' ⟨*keyvalue*⟩ } '`}`';

⟨*keyvalue*⟩ ::= ⟨*identifier*⟩ '`:`' ⟨*parameter*⟩;

⟨*literal*⟩ ::= ⟨*integer*⟩
| ⟨*float*⟩
| ⟨*string*⟩;

⟨*nondigit*⟩ ::= '`A`' ... '`Z`' | '`a`' ... '`z`' | '`_`';

⟨*digit*⟩ ::= '`0`' ... '`9`';

⟨*character*⟩ ::= ⟨*nondigit*⟩ | ⟨*digit*⟩;

⟨*regularStringChar*⟩ ::= ⟨*ANY*⟩ - '`"`' - '`\`';

⟨*string*⟩ ::= '`"`' { ⟨*regularStringChar*⟩ | '`\\`' | '`\"`' } '`"`';

⟨*identifier*⟩ ::= ⟨*nondigit*⟩ { ⟨*character*⟩ };

⟨*integer*⟩ ::= ⟨*digit*⟩ { digit };

⟨*float*⟩ ::= { ⟨*digit*⟩ } '`.`' ⟨*digit*⟩ { ⟨*digit*⟩ };

A.3. Grammar for primitive nets over DirectIO

The DIO net protocol is used to transmit primitive nets over the DirectIO protocol.

⟨*Fragment*⟩ ::= '`{`' [⟨*FragmentPart*⟩ { '`,`' ⟨*FragmentPart*⟩ }] '`}`';

⟨*FragmentPart*⟩ ::= ⟨*identifier*⟩ '`=`' ⟨*Primitive*⟩ ['`.`' ⟨*Identifier*⟩];

⟨*Primitive*⟩ ::= ⟨*identifier*⟩ ['`(`' [⟨*Parameter*⟩ { '`,`' ⟨*Parameter*⟩ }] '`)`']
| ⟨*Fragment*⟩ '`(`' [⟨*Parameter*⟩ { '`,`' ⟨*Parameter*⟩ }] '`)`';

⟨*Parameter*⟩ ::= ⟨*identifier*⟩ '`=`' ⟨*string*⟩
| ⟨*Primitive*⟩ '`.`' ⟨*identifier*⟩;

⟨*nondigit*⟩ ::= '`A`' ... '`Z`' | '`a`' ... '`z`' | '`_`' | '`:`';

⟨*digit*⟩ ::= '`0`' ... '`9`';

⟨*character*⟩ ::= ⟨*nondigit*⟩ | ⟨*digit*⟩;

⟨*regularStringChar*⟩ ::= ⟨*ANY*⟩ - '`'`' - '`\`';

⟨*string*⟩ ::= ‘'’ { ⟨*regularStringChar*⟩ | ‘\\’ | ‘\'’ } ‘'’;

⟨*identifier*⟩ ::= ⟨*nondigit*⟩ { ⟨*character*⟩ };

Appendix B

Hardware devices supported by the SoftRobot RCC

Table B.1 on Page 216 lists hardware that has been successfully connected to the SoftRobot RCC. Manufacturer and type are listed for all devices, and the software version of the controller if applicable. Several interfaces are used. Some controllers are connected using a standard network connection and UDP/IP packets. TCP is not used because real-time connections are using dedicated network segments, thus packet loss or reordering is not expected. Furthermore, retransmission of lost packets would not provide any benefit since time limits would most likely be broken already. On top of UDP, two proprietary protocols defined by the hardware manufacturer are used.

Some other devices use the EtherCAT fieldbus technology (cf. Section 9.4.1). On top of EtherCAT, either a proprietary protocol or the standard DS 402 protocol is used with CANopen over EtherCAT (CoE). Some devices use direct connection with CAN (or CANopen). The SoftRobot RCC contains a driver which enables CANopen using standard CAN hardware, which is accessed using the "socket-can" interface. Finally, a gripper is connected using electrical I/O, i.e. the gripper controller is connected to bus-terminals which then are controlled by the SoftRobot RCC using EtherCAT.

The table also lists the common control cycle times for most devices. For some devices, no cycle time is applicable since the protocol does not offer cyclic communication (e.g. the gripper can be opened or closed at any time using power).

Manufacturer	**Type**	**Version**	**Interface**	**Protocol**	**Cycle time**
KUKA	LWR4	KSS 5.6	UDP	Fast Research Interface (FRI)	1-12 ms
	KR-6 sixx agilus	KSS 8.2		Robot Sensor Interface (RSI)	4 ms
	KR-16				
	KR-210	KSS 8.3			
	youBot	Firmware v2.0	EtherCAT	proprietary	
Stäubli	TX90-L	uniVAL s7.7		CoE with DS 402	
Beckhoff	ELxxxx (bus terminals)			CoE	
Schunk	LWA4		CANopen	DS 402	
Festo	CMMS-ST				
Schunk	MEG		elect. I/O		n/a
	WSG		CAN	proprietary	
	SDH				

Table B.1.: Hardware supported by the SoftRobot RCC as of February 23, 2015

List of Figures

Bibliography

[1] A. Angerer. "Object-oriented Software for Industrial Robots." PhD thesis. University of Augsburg, Mar. 2014. URN: urn:nbn:de:bvb:384-opus4-30641 (cit. on pp. 4, 79, 156, 165, 173, 190, 191).

[2] A. Angerer, M. Bischof, A. Chekler, A. Hoffmann, W. Reif, A. Schierl, C. Tarragona, and M. Vistein. "Objektorientierte Programmierung von Industrierobotern." In: *Proceedings Internationales Forum Mechatronik 2010 (IFM 2010), Winterthur, Switzerland.* Nov. 2010 (cit. on p. 156).

[3] A. Angerer, A. Hoffmann, A. Schierl, M. Vistein, and W. Reif. "Robotics API: Object-Oriented Software Development for Industrial Robots." In: *Journal of Software Engineering for Robotics* 4.1 (2013), pp. 1–22. URL: http://joser.unibg.it/index.php?journal=joser&page=article&op=view&path[]=53 (cit. on pp. 36, 40, 156, 165, 186, 191).

[4] A. Angerer, A. Hoffmann, A. Schierl, M. Vistein, and W. Reif. "The Robotics API: An Object-Oriented Framework for Modeling Industrial Robotics Applications." In: *Proceedings 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems, Taipeh, Taiwan.* IEEE, Oct. 2010, pp. 4036–4041 (cit. on pp. 40, 156).

[5] L. B. Becker and C. E. Pereira. "SIMOO-RT – An Object Oriented Framework for the Development of Real-Time Industrial Automation Systems." In: *IEEE Transactions on Robotics and Automation* 18.4 (Aug. 2002), pp. 421–430 (cit. on p. 41).

[6] S. Becker, S. Dziwok, C. Gerking, C. Heinzemann, S. Thiele, W. Schäfer, M. Meyer, U. Pohlmann, C. Priesterjahn, and M. Tichy. *The MechatronicUML Design Method - Process and Language for Platform-Independent Modeling.* tr-ri-14-337. Technical Report. Version 0.4. University of Paderborn, Mar. 7, 2014. URL: http://www.mechatronicuml.org/files/MechatronicUML_PIM-Spec_v0.4.pdf (visited on 02/02/2015) (cit. on p. 77).

[7] J. Bengtsson and W. Yi. “Timed Automata: Semantics, Algorithms and Tools.” English. In: *Lectures on Concurrency and Petri Nets.* Ed. by J. Desel, W. Reisig, and G. Rozenberg. Vol. 3098. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 87–124. DOI: 10.1007/978-3-540-27755-2_3 (cit. on p. 77).

[8] G. Berry and G. Gonthier. “The ESTEREL synchronous programming language: design, semantics, implementation.” In: *Science of Computer Programming* 19.2 (1992), pp. 87–152 (cit. on p. 63).

[9] L. Biagiotti and C. Melchiorri. *Trajectory Planning for Automatic Machines and Robots.* Berlin, Heidelberg: Springer-Verlag, 2008 (cit. on pp. 15, 188).

[10] E. Bianchi and L. Dozio. “Some Experiences in fast hard realtime control in user space with RTAI-LXRT.” In: *Real time Linux workshop.* 2000 (cit. on p. 81).

[11] S. Biegacki and D. VanGompel. “The application of DeviceNet in process control.” In: *ISA Transactions* 35.2 (1996), pp. 169–176. DOI: 10.1016/0019-0578(96)00022-5. URL: http://www.sciencedirect.com/science/article/pii/0019057896000225 (cit. on p. 19).

[12] 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 (cit. on p. 145).

[13] S. Biyabani, J. A. Stankovic, and K. Ramamritham. “The Integration of Deadline and Criticalness in Hard Real-Time Scheduling.” In: *Real-Time Systems Symposium, 1988., Proceedings.* Dec. 1988, pp. 152–160. DOI: 10.1109/REAL.1988.51111 (cit. on p. 21).

[14] C. Blume and W. Jakob. *PasRo: PASCAL for Robots.* New York, NY, USA: Springer-Verlag New York, Inc., 1985 (cit. on p. 41).

[15] R. P. Bonasso, D. Kortenkamp, D. P. Miller, and M. Slack. “Experiences with an Architecture for Intelligent, Reactive Agents.” In: *J. of Experimental and Theoretical Artificial Intelligence* 9 (1995), pp. 237–256 (cit. on p. 41).

[16] J.-J. Borrelly, E. Coste-Manière, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon, and N. Turro. “The ORCCAD Architecture.” In: *International Journal of Robotics Research* 17.4 (Apr. 1998), pp. 338–359 (cit. on p. 78).

[17] H. Boterenbrood. “CANopen high-level protocol for CAN-bus.” In: *NIKEF, Amsterdam* 20 (2000) (cit. on p. 142).

[18] O. Brock, J. Kuffner, and J. Xiao. “Motion for Manipulation Tasks.” In: *Springer Handbook of Robotics.* Ed. by B. Siciliano and O. Khatib. Berlin, Heidelberg: Springer, 2008. Chap. 26, pp. 615–645 (cit. on pp. 15, 69).

[19] D. Brugali and P. Scandurra. “Component-Based Robotic Engineering (Part I).” In: *IEEE Robot. & Autom. Mag.* 16.4 (2009), pp. 84–96. DOI: 10.1109/MRA.2009.934837 (cit. on p. 42).

[20] D. Brugali and A. Shakhimardanov. "Component-Based Robotic Engineering (Part II)." In: *IEEE Robot. & Autom. Mag.* 20.1 (Mar. 2010). DOI: `10.1109/MRA.2010.935798` (cit. on p. 42).

[21] H. Bruyninckx. "Open robot control software: the OROCOS project." In: *Proceedings 2001 IEEE International Conference on Robotics & Automation.* Seoul, Korea, May 2001, pp. 2523–2528 (cit. on pp. 32, 42, 66, 99, 121).

[22] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali. "The BRICS Component Model: A Model-based Development Paradigm for Complex Robotics Software Systems." In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing.* SAC '13. Coimbra, Portugal: ACM, 2013, pp. 1758–1764. DOI: `10.1145/2480362.2480693` (cit. on p. 42).

[23] H. Bruyninckx and P. Soetens. "Generic real-time infrastructure for signal acquisition, generation and processing." In: *Proceedings 4th Real-time Linux Workshop.* Boston, USA, Dec. 2002 (cit. on p. 32).

[24] H. Bruyninckx, P. Soetens, and B. Koninckx. "The Real-Time Motion Control Core of the Orocos Project." In: *Proceedings 2003 IEEE International Conference on Robotics & Automation.* Seoul, Korea, May 2003, pp. 2766–2771 (cit. on pp. 32, 33, 66).

[25] R. Burchard and J. T. Feddema. "Generic robotic and motion control API based on GISC-Kit technology and CORBA communications." In: *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on.* Vol. 1. Apr. 1996, 712–717 vol.1. DOI: `10.1109/ROBOT.1996.503858` (cit. on p. 62).

[26] G. C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series).* Santa Clara, CA, USA: Springer-Verlag TELOS, 2004 (cit. on p. 21).

[27] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. "LUSTRE: A declarative language for real-time programming." In: *Proceedings 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* Munich, West Germany, 1987, pp. 178–188. DOI: `10.1145/41625.41641` (cit. on pp. 46, 63).

[28] J. Clark et al. "Xsl transformations (xslt)." In: *World Wide Web Consortium (W3C). URL http://www. w3. org/TR/xslt* (1999) (cit. on p. 103).

[29] 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. Sept. 1997, 836–841 vol.2. DOI: `10.1109/IROS.1997.655107` (cit. on p. 78).

[30] B. Dawes, D. Abrahams, and R. Rivera. *Boost C++ Libraries.* URL: `http://www.boost.org` (visited on 02/04/2015) (cit. on p. 91).

[31] J. De Schutter, T. De Laet, J. Rutgeerts, W. Decré, R. Smits, E. Aertbeliën, K. Claes, and H. Bruyninckx. "Constraint-Based Task Specification and Estimation for Sensor-Based Robot Systems in the Presence of Geometric Uncertainty." In: *The International Journal of Robotics Research* 26.5 (2007), pp. 433–455 (cit. on pp. 42, 67).

[32] G. Devol. "Programmed article transfer." US patent 2988237 A. June 13, 1961 (cit. on p. 1).

[33] *DO-178B. Software Considerations in Airborne Systems and Equipment Certification.* RTCA, Dec. 1, 1992 (cit. on p. 63).

[34] L. Dozio and P. Mantegazza. "Linux Real Time Application Interface (RTAI) in low cost high performance motion control." In: *Motion Control* 2003 (2003) (cit. on p. 81).

[35] *EtherCAT Slave Implemntation Guide.* ETG.2200. Version V2.0.0. EtherCAT Technology Group. Jan. 2, 2012. URL: `http://www.ethercat.org/pdf/english/ETG2200_V2i0i0_SlaveImplementationGuide.pdf` (visited on 02/02/2015) (cit. on pp. 136, 140).

[36] *Extension Methods (C# Programming Guide).* Microsoft MSDN documentation. URL: `http://msdn.microsoft.com/en-us/library/bb383977.aspx` (visited on 06/19/2014) (cit. on p. 166).

[37] M. Farsi, K. Ratcliff, and M. Barbosa. "An overview of controller area network." In: *Computing Control Engineering Journal* 10.3 (June 1999), pp. 113–120. DOI: `10.1049/cce:19990304` (cit. on p. 142).

[38] J. Feld. "PROFINET - scalable factory communication for all applications." In: *Factory Communication Systems, 2004. Proceedings. 2004 IEEE International Workshop on.* Sept. 2004, pp. 33–38. DOI: `10.1109/WFCS.2004.1377673` (cit. on p. 19).

[39] I. Fette and A. Melnikov. *The WebSocket Protocol.* RFC 6455 (Proposed Standard). Internet Engineering Task Force, Dec. 2011. URL: `http://www.ietf.org/rfc/rfc6455.txt` (cit. on pp. 82, 101).

[40] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616 (Draft Standard). Updated by RFCs 2817, 5785, 6266, 6585. Internet Engineering Task Force, June 1999. URL: `http://www.ietf.org/rfc/rfc2616.txt` (cit. on pp. 82, 103).

[41] B. Finkemeyer, T. Kröger, and F. M. 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. DOI: `10.1109/ROBOT.2010.5509419` (cit. on p. 77).

[42] 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 International Conference on Intelligent Robots and Systems.* San Diego, USA, Oct. 2007, pp. 76–90 (cit. on p. 42).

[43] B. Finkemeyer, T. Kröger, and F. M. Wahl. “Executing Assembly Tasks Specified by Manipulation Primitive Nets.” In: *Advanced Robotics* 19.5 (2005), pp. 591–611 (cit. on p. 77).

[44] B. Finkemeyer, T. Kröger, and F. M. Wahl. “Placing of Objects in Unknown Environments.” In: *Proceedings 9th IEEE International Conference on Methods and Models in Automation and Robotics.* Miedzyzdroje, Poland, Aug. 2003, pp. 975–980 (cit. on p. 42).

[45] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software.* Addison Wesley, 1994 (cit. on pp. 82, 96, 137, 158).

[46] E. R. Gansner, E. Koutsofios, S. C. North, and K.-p. Vo. “A Technique for Drawing Directed Graphs.” In: *IEEE Transactions on Software Engineering* 19.3 (1993), pp. 214–230 (cit. on p. 181).

[47] 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.* Vol. 362. Lect. Notes in Control & Information Sciences. Springer, 2007, pp. 91–97 (cit. on p. 41).

[48] P. Gerum. “Xenomai-Implementing a RTOS emulation framework on GNU/Linux.” In: *White Paper, Xenomai* (2004) (cit. on p. 81).

[49] *Google Web Toolkit.* URL: `http://www.gwtproject.org` (visited on 12/30/2014) (cit. on p. 116).

[50] M. Hägele, K. Nilsson, and J. N. Pires. “Industrial Robotics.” In: *Springer Handbook of Robotics.* Ed. by B. Siciliano and O. Khatib. Berlin, Heidelberg: Springer, 2008. Chap. 42, pp. 963–986 (cit. on pp. 1, 2, 41).

[51] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. “The synchronous dataflow programming language LUSTRE.” In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1305–1320 (cit. on pp. 43, 46, 63).

[52] V. Hayward and R. P. Paul. “Robot Manipulator Control under Unix RCCL: A Robot Control C Library.” In: *International Journal of Robotics Research* 5.4 (1986), pp. 94–111 (cit. on pp. 41, 61).

[53] T. Henzinger, B. Horowitz, and C. Kirsch. “Giotto: a time-triggered language for embedded programming.” In: *Proceedings of the IEEE* 91.1 (Jan. 2003), pp. 84–99. DOI: `10.1109/JPROC.2002.805825` (cit. on p. 77).

[54] A. Hoffmann. “Serviceorientierte Automatisierung von Roboterzellen.” PhD thesis. University of Augsburg, 2015 (cit. on p. 6).

[55] 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: *Proceedings 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems, St. Louis, Missouri, USA*. IEEE, Oct. 2009, pp. 2108–2113 (cit. on p. 37).

[56] A. Hoffmann, A. Angerer, A. Schierl, M. Vistein, and W. Reif. “Service-oriented Robotics Manufacturing by reasoning about the Scene Graph of a Robotics Cell.” In: *ISR/Robotik 2014; 41st International Symposium on Robotics; Proceedings of.* June 2014, pp. 1–8 (cit. on pp. 41, 73).

[57] A. Hoffmann, A. Angerer, A. Schierl, M. Vistein, and W. Reif. “Service-orientierte Modellierung einer Robotermontagezelle.” In: *Proceedings Internationales Forum Mechatronik 2011 (IFM 2011), Cham, Germany*. Sept. 2011 (cit. on pp. 41, 73).

[58] IEC 19505-1:2012. *Information technology – Object Management Group Unified Modeling Language (OMG UML) – Part 1: Infrastructure*. ISO, Geneva, Switzerland, 2012 (cit. on p. 3).

[59] IEC 61131-3. *Programmable controllers - Part 3: Programming languages*. ISO, Geneva, Switzerland, 2013 (cit. on pp. 35, 64).

[60] 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 (cit. on p. 142).

[61] IEC 61800-7-301. *Adjustable speed electrical power drive systems - Part 7-301: Generic interface and use of profiles for power drive systems - Mapping of profile type 1 to network technologies*. ISO, Geneva, Switzerland, 2007 (cit. on p. 142).

[62] IEEE. “IEEE Standard for Ethernet - Section 1.” In: *IEEE Std 802.3-2012 (Revision to IEEE Std 802.3-2008)* (2012). DOI: 10.1109/IEEESTD.2012.6419735 (cit. on p. 135).

[63] IEEE. “IEEE Standard for Floating-Point Arithmetic.” In: *IEEE Std 754-2008* (2008), pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935 (cit. on p. 93).

[64] IFR statistical department. *World Robotics - Industrial Robots 2014*. Executive Summary. International Federation of Robotics, 2014. URL: http://www.worldrobotics.org/index.php?id=downloads (visited on 01/15/2015) (cit. on p. 1).

[65] *IKVM.NET Home Page*. URL: http://www.ikvm.net (visited on 01/02/2015) (cit. on pp. 39, 155).

[66] ISO. *Manipulating industrial robots – Mechanical interfaces – Part 1: Plates*. Standard 9409-1:2004. International Organization for Standardization, 2004. URL: http://www.iso.org (cit. on p. 10).

[67] ISO. *Robots and robotic devices – Vocabulary*. Standard 8373:2012. International Organization for Standardization, 2012. URL: http://www.iso.org/ (cit. on p. 9).

[68] I. Jacobson, J. Rumbaugh, and G. Boochy. *The Unified Software Development Process*. Addison Wesley Pub Co Inc, Feb. 26, 1999 (cit. on p. 2).

[69] D. Jansen and H. Buttner. “Real-time Ethernet: the EtherCAT solution.” English. In: *Computing and Control Engineering* 15 (1 Feb. 2004), 16–21(5). URL: `http://digital-library.theiet.org/content/journals/10.1049/cce_20040104` (cit. on p. 19).

[70] *Java*. Oracle. URL: `http://java.oracle.com` (visited on 01/05/2015) (cit. on p. 39).

[71] M. Kalicinski. *RapidXml*. URL: `http://rapidxml.sourceforge.net/` (visited on 12/30/2014) (cit. on p. 106).

[72] B. Kaup. “Grafische Bearbeitung einer datenflussbasierten Sprache für die Roboterprogrammierung.” Master’s thesis. University of Augsburg, Aug. 2014 (cit. on p. 182).

[73] M. Kay. *XSLT 2.0 programmer’s reference*. Wiley Pub. Chichester, 2004 (cit. on p. 103).

[74] A. Ketels. *Simple Open EtherCAT Master*. URL: `http://sourceforge.net/projects/soem.berlios/` (visited on 12/30/2014) (cit. on p. 137).

[75] J. Kiszka, B. Wagner, Y. Zhang, and J. Broenink. “RTnet - A flexible Hard Real-Time Networking Framework.” In: *ETFA 2005: 10th IEEE Conference on Emerging Technologies and Factory Automation*. IEEE, 2005, pp. 449–456. URL: `http://doc.utwente.nl/53551/` (cit. on p. 137).

[76] *KUKA System Software 8.2. Operating and Programming Instructions for System Integrators*. Version KSS 8.2 SI V6 en. KUKA Roboter GmbH. Feb. 19, 2014 (cit. on pp. 18, 26, 30).

[77] *KUKA System Software 8.3. Operating and Programming Instructions for System Integrators*. Version KSS 8.3 SI V3 en. KUKA Roboter GmbH. May 28, 2014 (cit. on pp. 2, 18, 26, 30, 41).

[78] *KUKA.BendTech 3.0. For KUKA System Software 5.4 and 5.5*. Version KST BendTech 3.0 V1 en. KUKA Roboter GmbH. Feb. 24, 2009 (cit. on p. 28).

[79] *KUKA.Gripper&SpotTech 3.1. For KUKA System Software 8.2 and 8.3*. Version KST GripperSpotTech 3.1 V1 en. KUKA Roboter GmbH. Nov. 29, 2012 (cit. on pp. 27, 28).

[80] *KUKA.LaserTech 3.0. For KUKA System Software 8.2*. Version KST LaserTech 3.0 V2 en. KUKA Roboter GmbH. May 31, 2012 (cit. on p. 30).

[81] *KUKA.RoboTeam 2.0. For KUKA System Software 8.3*. Version KST RoboTeam 2.0 V3. KUKA Roboter GmbH. May 9, 2014 (cit. on p. 31).

[82] *KUKA.RobotSensorInterface 3.1. For KUKA System Software 8.2*. Version KST RSI 3.1 V1 en. KUKA Roboter GmbH. Dec. 23, 2010 (cit. on pp. 29, 197).

[83] C. L. Liu and J. W. Layland. “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.” In: *Journal of the ACM* 20.1 (Jan. 1973), pp. 46–61. DOI: `10.1145/321738.321743` (cit. on p. 186).

[84] J. Lloyd, M. Parker, and R. McClain. “Extending the RCCL programming environment to multiple robots and processors.” In: *Robotics and Automation, 1988. Proceedings., 1988 IEEE International Conference on.* Apr. 1988, 465–469 vol.1. DOI: 10.1109/ROBOT.1988.12095 (cit. on p. 61).

[85] M. S. Loffler, V. Chitrakaran, and D. M. Dawson. “Design and Implementation of the Robotic Platform.” In: *Journal of Intelligent and Robotic System* 39 (2004), pp. 105–129 (cit. on p. 41).

[86] J. Marini. *Document Object Model.* 1st ed. New York, NY, USA: McGraw-Hill, Inc., 2002 (cit. on p. 106).

[87] D. J. Miller and R. C. Lennox. “An Object-Oriented Environment for Robot System Architectures.” In: *Proceedings 1990 IEEE International Conference on Robotics & Automation.* Cincinnati, Ohio, USA, May 1990, pp. 352–361 (cit. on pp. 41, 62).

[88] *Mixin Class Composition.* Scala Documnentation. URL: http://docs.scala-lang.org/tutorials/tour/mixin-class-composition.html (visited on 06/19/2014) (cit. on p. 166).

[89] H. Mössenböck. “A generator for production quality compilers.” In: *Compiler Compilers.* Ed. by D. Hammer. Vol. 477. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1991, pp. 42–55. DOI: 10.1007/3-540-53669-8_73 (cit. on pp. 111, 120).

[90] H. Mühe, A. Angerer, A. Hoffmann, and W. Reif. “On reverse-engineering the KUKA Robot Language.” In: *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) (cit. on pp. 18, 165).

[91] M. Muja and D. G. Lowe. “Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration.” In: *International Conference on Computer Vision Theory and Application VISSAPP’09).* INSTICC Press, 2009, pp. 331–340 (cit. on p. 209).

[92] L. Nägele, M. Macho, A. Angerer, A. Hoffmann, M. Vistein, M. Schönheits, and W. Reif. “A backward-oriented approach for offline programming of complex manufacturing tasks.” In: *2015 The 6th International Conference on Automation, Robotics and Applications (ICARA 2015).* Queenstown, New Zealand, Feb. 2015 (cit. on p. 208).

[93] S. Y. Nof, ed. *Handbook of Industrial Robotics.* 2nd. New York: John Wiley & Son, 1999 (cit. on p. 1).

[94] *OMG Unified Modeling Language. Infrastructure.* Version 2.4.1. Object Management Group, Aug. 5, 2011. URL: http://www.omg.org/spec/UML/2.4.1/Infrastructure (cit. on p. 3).

[95] R. Paul. “WAVE: A model based language for manipulator control.” In: *Industrial Robot: An International Journal* 4.1 (1977), pp. 10–17. DOI: 10.1108/eb004473. eprint: http://dx.doi.org/10.1108/eb004473. URL: http://dx.doi.org/10.1108/eb004473 (cit. on p. 41).

[96] C. Pelich and F. M. Wahl. “ZERO++: An OOP Environment for Multiprocessor Robot Control.” In: *International Journal of Robotics and Automation* 12.2 (1997), pp. 49–57 (cit. on p. 41).

[97] J. Peschke and A. Lüder. “Java Technology and Industrial Applications.” In: *The Industrial Information Technology Handbook*. Ed. by R. Zurawski. Industrial Electronics Series. Boca Raton, FL: CRC Press, 2005. Chap. 63 (cit. on pp. 3, 41).

[98] J. N. Pires, G. Veiga, and R. Araújo. “Programming by demonstration in the coworker scenario for SMEs.” In: *Industrial Robot* 36.1 (2009), pp. 73–83 (cit. on p. 41).

[99] E. Prassler, H. Bruyninckx, K. Nilsson, and A. Shakhimardanov. *The Use of Reuse for Designing and Manufacturing Robots*. White Paper. Robot Standards and reference architectures (RoSta) consortium, June 2009. URL: http://www.robot-standards.eu/Documents_RoSta_wiki/whitepaper_reuse.pdf (cit. on p. 42).

[100] G. Prytz. “A performance analysis of EtherCAT and PROFINET IRT.” In: *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on*. 2008, pp. 408–415. DOI: 10.1109/ETFA.2008.4638425 (cit. on p. 136).

[101] 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 (cit. on p. 42).

[102] *RAPID Reference Manual*. ABB Group. 2004 (cit. on p. 2).

[103] M. Rostan, J. E. Stubbs, and D. Dzilno. “EtherCAT enabled advanced control architecture.” In: *Advanced Semiconductor Manufacturing Conference (ASMC), 2010 IEEE/SEMI*. 2010, pp. 39–44. DOI: 10.1109/ASMC.2010.5551414 (cit. on p. 136).

[104] *SCADE Suite*. Esterel Technologies. URL: http://www.esterel-technologies.com/products/scade-suite/ (visited on 12/17/2014) (cit. on pp. 43, 63).

[105] 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: *Proceedings 2012 International Conference on Information in Control, Automation & Robotics, Rome, Italy*. 2012 (cit. on pp. 167, 169, 171, 175).

[106] 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 International Conference on Robotics & Automation*. Anchorage, Alaska, USA, May 2010 (cit. on pp. 146, 148).

[107] B. Siciliano and O. Khatib, eds. *Springer Handbook of Robotics*. Berlin, Heidelberg: Springer, 2008.

[108] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo. *Robotics: Modelling, Planning and Control*. 1st. Springer Publishing Company, Incorporated, 2008 (cit. on pp. 10–13, 23, 128).

[109] F. Siebert. "Realtime Garbage Collection in the JamaicaVM 3.0." In: *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems*. JTRES '07. Vienna, Austria: ACM, 2007, pp. 94–103. DOI: `10.1145/1288940.1288954` (cit. on p. 26).

[110] R. Simmons and D. Apfelbaum. "A Task Description Language for Robot Control." In: *Proceedings 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems, Victoria, Canada*. Oct. 1998 (cit. on pp. 41, 79).

[111] R. Simmons. "Concurrent planning and execution for autonomous robots." In: *IEEE Control Systems* 12.1 (Feb. 1992), pp. 46–50. DOI: `10.1109/37.120453` (cit. on p. 78).

[112] R. Smits. *Orocos Kinematics and Dynamics*. URL: `http://www.orocos.org/kdl` (visited on 01/27/2015) (cit. on pp. 13, 33).

[113] P. Soetens. "A software framework for real-time and distributed robot and machine control." PhD thesis. Katholieke Universiteit Leuven, Faculteit Ingenieursweten-schappen, Departement Werktuigkunde, 2006 (cit. on p. 33).

[114] *State Chart XML (SCXML): State Machine Notation for Control Abstraction*. Working Draft 6. W3C, Dec. 2012. URL: `http://www.w3.org/TR/scxml/` (cit. on p. 42).

[115] D. B. Stewart, D. Schmitz, and P. K. Khosla. "The Chimera II real-time operating system for advanced sensor-based control applications." In: *Systems, Man and Cybernetics, IEEE Transactions on* 22.6 (Nov. 1992), pp. 1282–1295. DOI: `10.1109/21.199456` (cit. on p. 65).

[116] D. B. Stewart, R. A. Volpe, and P. K. Khosla. "Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects." In: *IEEE Transactions on Software Engineering* 23.12 (Dec. 1997), pp. 759–776 (cit. on p. 65).

[117] B. Stroustrup. *The C++ programming language*. 4th ed. Addison Wesley, July 17, 2013 (cit. on p. 81).

[118] R. H. Taylor, P. D. Summers, and J. M. Meyer. "AML: A Manufacturing Language." In: *The International Journal of Robotics Research* 1.3 (1982), pp. 19–41. DOI: `10.1177/027836498200100302` (cit. on p. 41).

[119] E. Tovar and F. Vasques. "Real-time fieldbus communications using Profibus networks." In: *Industrial Electronics, IEEE Transactions on* 46.6 (Dec. 1999), pp. 1241–1251. DOI: `10.1109/41.808018` (cit. on p. 19).

[120] *uniVAL drive - innovative robot control*. Stäubli. URL: `http://www.unival-drive.com/` (visited on 01/02/2015) (cit. on p. 142).

[121] *VAL3 Reference Manual.* Version 7. Stäubli. Aug. 23, 2010 (cit. on pp. 2, 32, 41).

[122] D. Vanthienen, M. Klotzbücher, J. De Schutter, T. De Laet, and H. Bruyninckx. "Rapid application development of constrained-based task modelling and execution using domain specific languages." In: *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on.* Nov. 2013, pp. 1860–1866. DOI: 10.1109/IROS.2013.6696602 (cit. on p. 67).

[123] D. Vanthienen, M. Klotzbücher, and H. Bruyninckx. "The 5C-based architectural Composition Pattern: lessons learned from re-developing the iTaSC framework for constraint-based robot programming." In: *Journal of Software Engineering for Robotics* 5.1 (2014), pp. 17–35 (cit. on pp. 42, 67).

[124] S. Vinoski. "CORBA: integrating diverse applications within distributed heterogeneous environments." In: *Communications Magazine, IEEE* 35.2 (Feb. 1997), pp. 46–55. DOI: 10.1109/35.565655 (cit. on p. 33).

[125] M. Vistein, A. Angerer, A. Hoffmann, A. Schierl, and W. Reif. "Flexible and Continuous Execution of Real-Time Critical Robotic Tasks." In: *International Journal of Mechatronics and Automation* 4.1 (Jan. 1, 2014), pp. 27–38. DOI: 10.1504/IJMA.2014.059773 (cit. on pp. 17, 36, 37, 39, 43, 70, 74, 177).

[126] M. Vistein, A. Angerer, A. Hoffmann, A. Schierl, and W. Reif. "Instantaneous switching between real-time commands for continuous execution of complex robotic tasks." In: *Proceedings 2012 International Conference on Mechatronics and Automation, Chengdu, China.* Aug. 2012, pp. 1329–1334. DOI: 10.1109/ICMA.2012.6284329 (cit. on pp. 36, 39, 43, 70).

[127] M. Vistein, A. Angerer, A. Hoffmann, A. Schierl, and W. Reif. "Interfacing Industrial Robots using Realtime Primitives." In: *Proceedings 2010 IEEE International Conference on Automation and Logistics, Hong Kong, China.* IEEE, Aug. 2010, pp. 468–473 (cit. on pp. 39, 43).

[128] *Wind River VxWorks.* Wind River. URL: http://www.windriver.com/products/vxworks/ (visited on 12/31/2014) (cit. on p. 30).

[129] C. Zieliński. "Control of a multi-robot system." In: *2nd Int. Symp. Methods and Models in Automation and Robotics MMAR'95, Międzyzdroje, Poland.* 1995, pp. 603–608 (cit. on p. 62).

[130] C. Zieliński. "Object-oriented robot programming." In: *Robotica* 15.1 (1997), pp. 41–48 (cit. on pp. 41, 62).

[131] C. Zieliński and T. Winiarski. "Motion Generation in the MRROC++ Robot Programming Framework." In: *International Journal of Robotics Research* 29.4 (2010), pp. 386–413. DOI: 10.1177/0278364909348761. eprint: http://ijr.sagepub.com/content/29/4/386.full.pdf+html (cit. on p. 62).