Flexible and continuous execution of real-time critical robotic tasks

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

Angaben zur Veröffentlichung / Publication details:

Vistein, Michael, Andreas Angerer, Alwin Hoffmann, Andreas Schierl, and Wolfgang Reif. 2014. "Flexible and continuous execution of real-time critical robotic tasks." *International Journal of Mechatronics and Automation* 4 (1): 27–38. https://doi.org/10.1504/ijma.2014.059773.

DOI: 10.1504/IJMA.2014.059773

Flexible and Continuous Execution of Real-Time Critical Robotic Tasks

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

Institute for Software & Systems Engineering
Augsburg University
Universitätsstraße 6a, 86135 Augsburg, Germany
E-Mail: vistein@informatik.uni-augsburg.de

Abstract Today, industrial robots are usually programmed using specialized programming languages, different for every robot manufacturer. These languages provide good usability, because they are tailored to the functionality traditionally offered by robots. However, these languages are reaching their limits with the growing integration of sensors or multiple robot systems. Therefore, we propose an architecture based on the separation of application control and the execution of real-time robotic tasks. This article describes a flexible and extensible interface for the specification and continuous execution of robotic tasks.

Keywords: robot programming; real-time; object-oriented programming; task scheduling; industrial robots; dataflow language; robotic tasks; flexibility

Biographical notes: Michael Vistein received his B. Sc. degree in computer science from the University of Augsburg in 2007 and his M. Sc. degree in software engineering from the University of Augsburg, the Technische Universität München and the Ludwig-Maximilians-Universität München in 2008.

Andreas Angerer received his B. Sc. degree in computer science from the University of Augsburg in 2007 and his M. Sc. degree in software engineering from the University of Augsburg, the Technische Universität München and the Ludwig-Maximilians-Universität München in 2008.

Alwin Hoffmann received his B. Sc. and M. Sc. degrees in information management from the Technische Universität München and the University of Augsburg, in 2005 and 2007, respectively, and his Diploma in computer science from the University of Augsburg, in 2008.

Andreas Schierl received his B. Sc. degree in computer science from the University of Augsburg in 2007, and the M. Sc. degree in software engineering from the University of Augsburg, the Technische Universität München and the Ludwig-Maximilians-Universität in 2009.

Michael Vistein, Andreas Angerer, Alwin Hoffmann and Andreas Schierl are member of the Institute for Software and Systems Engineering, University of Augsburg. They were involved in several research projects in the field of robotics and have published about 20 refereed journals, conference and workshop papers.

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

1 Introduction

Industrial robots can be used in very different work scenarios, from welding or gluing to quality assurance. In order to achieve such a broad variety, a flexible way of programming is required. However, most industrial robots today are programmed using proprietary languages provided by robot manufacturers. These languages show their limitations when the application scenario reaches a certain complexity. To overcome this, researchers have developed a variety of approaches and frameworks. A quite popular general approach employs a three-tiered system structure (Bonasso et al., 1995) that separates the layers *Planning*, *Execution* and *Behavioral Control* of robot task structures.

Within the research project *SoftRobot*, a software architecture has been created that enables programming complex real-time critical industrial robot applications in Java. Using this popular and wide-spread language gives robot developers access to a great ecosystem of libraries and development tools as well as a large community that actively brings forward the language and its extensions. To support the inevitable hard real-time requirements of robot control (e.g. for achieving high precision and exact synchronization between robots and its tools), the *SoftRobot* architecture relies on a so-called *Robot Control Core (RCC)* for Behavioral Control. Execution and Planning are handled by the Java-based *Robotics API* and applications running on top of it.

In this work, we focus on the interface to the RCC, the *Realtime Primitives Interface (RPI)*. This interface accepts so called *primitive nets*, which are tasks specified in a dataflow language. Such real-time tasks are equivalent to robot behaviors or a combination of such behaviors. The object-oriented Robotics API includes a transformation layer for mapping high-level robot tasks to dynamically generated dataflow-based commands in RPI. This transformation is transparent to the application developer.

The advantages of using a general-purpose programming language have been identified earlier. The RCCL project (Hayward and Paul, 1986) for the C language and the PasRo project (Blume and Jakob, 1986) for Pascal are very early examples. Since object-oriented languages are more and more popular, robot-specific libraries have also been implemented using these languages. Examples are ZERO++ (Pelich and Wahl, 1997), MRROC++ (Zieliński, 1997), RIPE (Miller and Lennox, 1990), the Robotics Platform (Loffler et al., 2004) and SIMOO-RT (Becker and Pereira, 2002). However, the level of abstraction of these frameworks is not as high as when using traditional robot programming languages, so a profound knowledge of real-time programming is required.

At the beginning of the SoftRobot project, many common use-cases for industrial robots have been examined. It could be noticed that in almost all use-cases not the whole application requires hard real-time, but only certain parts of the application. For example, every single motion needs real-time for a smooth trajectory, or tools need to be synchronized to the motion path. Between two motions, usually a short break is acceptable (although not desirable for lower cycle-times). Therefore we decided that the overall control flow of a robotics application can be performed using a standard programming language, and only the real-time critical tasks must be specified by other means.

In certain cases however, it is necessary or desirable to switch instantaneously between two task in a real-time critical way. A typical example is moving a robot to a certain goal while concurrently planning subsequent movements. Once a movement command has finished (in a possibly *unstable* state, e.g. with the robot moving at high speed), a newly planned motion command should immediately take control of the robot retaining stable control. Such a real-time critical transition between robot behaviors is also tackled in several existing robot control systems: ORCCAD by Borrelly et al. (1998) contains formally verified mechanisms that guarantee minimal latency for e.g. the transitions between sequential robot behaviors. TCA (Simmons, 1992) provides a mechanism for synchronization of planning and execution steps in a message based control system. In MiRPA (Finkemeyer et al., 2003), transitions between skill primitives preserve the system stability as well, while MRROC++ (Zieliński and Winiarski, 2010) supports a seamless takeover of a preceding motion instruction. Other popular robot control frameworks like Orocos (Smits et al., 2009) require the application developer to implement mechanisms for real-time critical transition between robot behaviors if required, while some frameworks like ROS (Quigley et al., 2009) do not provide real-time support at all.

This article is a revised and extended version of a paper entitled "Instantaneous Switching between Real-Time Commands for Continuous Execution of Complex Robotic Tasks" presented at 2012 IEEE International Conference on Mechatronics and Automation, August 5-8 2012, Chengdu, China (c.f. Vistein et al., 2012). The article proposes a generic interface for specifying robotic tasks with a mechanism for real-time critical switching from one running command to dynamically loaded subsequent commands. Section 2 discusses some requirements which have been essential for the design of the SoftRobot architecture, which is explained in Section 3. The static structure of the interface between high-level programming and low-level robot control, the Realtime Primitives Interface, is described in Section 4 and is followed by a description of the run-time behavior of commands specified using this language in Section 5. Examples illustrating the previously described interface are shown in Section 6. The requirements of the architecture which have been identified at the beginning of the article are recapitulated in Section 7 and their satisfaction by the proposed design is evaluated. In Section 8 a reference implementation of the SoftRobot architecture is explained. The article is finally concluded with Section 9.

2 Requirements

The main goal of the SoftRobot project was to introduce modern methods of software engineering into the domain of industrial robotics. In this domain, the programming languages and tools are traditionally provided by the robot manufacturers, and each manufacturer employs its own set of languages and tools. These languages are tailored to the needs of industrial robotics: They support features unique to robotics, such as special motion commands, I/O commands or actions triggered by path positions. They support real-time operations while still being abstract enough not to bother the developer with details. However, these languages usually are based on rather old concepts. Because each manufacturer has to advance the development of their language in-house, the languages could not evolve with the same speed as other general-purpose languages, and therefore modern methods of software-engineering such as object-oriented design or service-oriented architectures are not possible or very hard to implement.

Just substituting the special languages by an existing general-purpose language is not possible. Most general-purpose languages lack support for real-time programming, which is essential in the robotics domain. A common requirement is that operations of robot tools must be triggered at exact points in space or time, e.g. a welding torch must be turned on once the robot has reached a certain point. Even a single motion is real-time critical, because motions are usually interpolated at a high frequency, and each interpolation step must be accurately processed to guarantee a smooth motion.

Besides the reqirement for real-time programming, the design of the architecture in the SoftRobot project and the Java-based Robotics API has been influenced by the following requirements:

1. **Usability.** Traditional programming languages for robots provide a very abstract interface to the developer. Therefore, developers should have a intuitive programming interface and especially should not have to care about real-time issues.
2. **Multi robot systems.** It should be possible to control several robots using a single program.
3. **Sensor support.** It should be possible to integrate sensors e.g. for sensor-controlled motions.
4. **Extensibility.** The system should not be limited to a predefined set of robots, sensors, algorithms, etc., but rather be flexible and extensible.
5. **Special industrial robotics concepts.** Besides standard motions, also special features for the industrial robotics domain should be supported. Two examples are force manipulation and motion blending.

The two special industrial robotics concepts of force manipulation and motion blending are explained more in detail in the following sections.

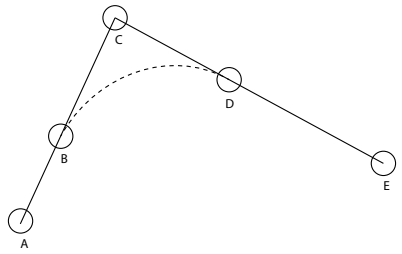

Figure 1 Motion blending

2.1 Force Manipulation

In robotics, manipulation or assembly tasks (Brock et al., 2008) are getting more and more important. However, these tasks are very challenging due to the necessity to handle uncertainty and to perform these tasks reliably despite of low tolerances of assembled products (cf. the peg-in-hole insertion problem). Hence, compliant motions are used for assembly because such motions are constrained by the contact between some part held by the robot and other parts in the environment, which reduces uncertainty.

The goal of compliant motions is to establish a contact force between the robot (or its held part) and the environment. The decision of how to proceed, i.e. which is the next motion command to issue, is usually made depending on the measured force and torque (c.f. Finkemeyer et al., 2003). Because of the robot's contact to the environment, the system should never be out of active control and, due to safety reasons, should be able to react to sudden events.

2.2 Motion Blending

To achieve high throughput, the idle time of an industrial robot should be as low as possible. Therefore, commercial programming languages for industrial robots like the KUKA Robot Language (KRL) or RAPID from ABB support *motion blending*, which allows the robot to start the next motion before the last one has finished. Using this feature, the time required for deceleration and acceleration can be avoided at the price that the intermediate point is not exactly reached. This is perfectly acceptable e.g. for auxiliary points which help the robot to avoid obstacles. Fig. 1 shows example trajectories for two linear motions from A to C and from C to E. If motion blending is not desired, the solid black line will be used as trajectory and the robot will temporarily come to halt at C. If motion blending is used (dashed line), the robot will leave the linear path at some point B and rejoin the linear path on some other point D without the need to halt.

3 Solution: The SoftRobot Architecture

To fulfill all requirements posted in Section 2, a two tier architecture as depicted in Figure 2 has been introduced by Hoffmann et al. (2009). By separating the *Robot*

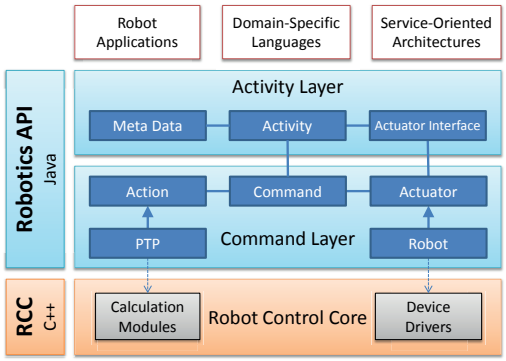

Figure 2 SoftRobot architecture

Control Core (RCC), written in C++ and the *Robotics API*, written in Java, it is possible to use a general-purpose programming language (Java directly on top of the Robotics API, but also e.g. C# using IKVM.NET) also for programming industrial robots. This is possible, because applications do not inherently require real-time for the application logic, but only for parts of the program like motions or tools actions. One of the big benefits of modern object-oriented programming languages is the integrated memory management (e.g. garbage collection, array bounds checks, ...), which has always been a major cause for bugs in applications.

3.1 Robot Control Core

The Robot Control Core takes care of all real-time critical parts of the robotics systems and provides a flexible, generic interface, the *Realtime Primtives Interface* (RPI), to the upper layer, the Robotics API. RPI is a dataflow based language, consisting of basic calculation modules (called primitives), which can be combined to form primitive nets. The Robotics API automatically creates such primitive nets (c.f. Schierl et al., 2012b) which then are transferred to the Robot Control Core for execution. Once the Robot Control core has fully received such a primitive net, it deterministically executes all contained primitives, hence guaranteeing real-time characteristics.

Besides the basic calculation blocks, the Robot Control Core also needs to implement device specific drivers to communicate with sensors and actuators. Those drivers also need to be implemented in C++ for real-time requirements. Actuator drivers expect to receive set-points (e.g. position, velocity, acceleration) at a high frequency (usually 1 kHz) from a running primitive net. Sensor drivers provide (raw or processed) sensor data also at a high frequency for further processing within a primitive net.

3.2 Robotics API

The Robotics API consists of two layers, the Command Layer, which is directly communicating with the Robot Control Core, and an additional Activity Layer which provides more convenient access to the most common functions of the Robotics API. The Command Layer has a very abstract model of robotic tasks, which consists of Actions which can be performed by Actuators. Assigning an Action to an Actuator yields a Command, which can be executed. Actions can be for example a Point-to-Point (PTP) motion, and an Actuator can be a specific type of robot. Actuators need a corresponding driver and primitives in the Robot Control Core, whereas Actions usually map to a set of calculation primitives in the RCC which together form the requested action. Using an event mechanism, multiple commands can be combined e.g. to perform tool operations based on sensor data.

The Command Layer provides a very generic, flexible and extensible interface for writing arbitrarily complex applications. It is mainly designed for system integrators who want to integrate new products into the system. The Activity Layer wraps commonly used functions into more simple Java methods and is therefore targeted at robot-application developers. More details about the Robotics API can be found in Angerer et al. (2010).

4 Realtime Primitives Interface

The Realtime Primitives Interface provides a generic interface to (real-time) robot controllers. It allows a very flexible specification of complex commands for multiple robotic devices from a high-level programming interface, such as the Robotics API. These commands can be executed respecting hard real-time constraints. Hence, RPI facilitates the separation of real-time device control and high-level application logic.

RPI is designed as a dataflow language similar to the Lustre (Halbwachs et al., 1991) language, used in the commercial SCADE Suite, and is tailored to industrial robotics. The key building blocks are *primitives* interconnected with *links*. Sets of primitives and links can be grouped as *fragments*, which themselves can be used like primitives. The resulting network of primitives, fragments and links (called *primitive net*) can be executed by a Robot Control Core. The execution is performed cyclically, i.e. the whole primitive net is evaluated repeatedly at exact intervals. In each cycle it is possible to generate new set-points, therefore hardware can be controlled precisely.

4.1 Primitives

The basic calculation blocks in RPI are provided by primitives. These primitives have *input* and *output ports* and can be configured using *parameters*. In each execution cycle, values from the input ports are read, calculations performed and result values written to output ports. Primitives are not required to have both input and output ports, even primitives with no ports at all are syntactically valid. Primitives can encapsulate very different functionality. There are primitives providing very basic support such as logical operators (AND, OR) or mathematical functions (addition, subtraction) as well

as more complex calculation modules which can generate trajectories or calculate (inverse) kinematics. Hardware components are also represented as primitives. Sensors (force/torque sensors, light barriers, etc.) are represented as primitives with output ports only, whereas actuators (robots, conveyor belts, etc.) are represented as primitives with input ports only, with the exception of error outputs.

Input and output ports are typed. The reference implementation in the SoftRobot project supports both basic types such as integer and double, but also complex composed types. Such complex types can for example represent positions in space using X, Y and Z coordinates, or fixed-size arrays of a given type. The used types must be specified completely prior to starting a primitive net (especially the size of arrays), because no memory allocation may happen during execution to guarantee real-time semantics. Each type supports a special *null* value to signal that no value is available.

Multiple instances of the same primitive can be used in a single primitive net. However, all primitives must ensure that exclusive resources are not used concurrently. For example a primitive representing a single actuator might appear several times in different parts of a primitive net, but only one primitive instance may actively control the hardware in a single execution cycle. The RCC reference implementation supports resource allocation to prevent multiple primitive nets from using the same resources accidentally, and the RCC guarantees that all requested resources will be available once a primitive net has been accepted for execution.

Parameters can be used to configure primitive instances. Actuator primitives for example require a parameter indicating which device instance to control. Parameters can only be set upon specification of a primitive net and are immutable during execution.

Some primitives have an internal state to preserve information (like the current position of a trajectory) during the execution of the primitive net. However, most primitives do not preserve any information for the next primitive net. Transport of information from one primitive net to the next should be done by means of the high-level programming language for maximal flexibility.

4.2 Links

Multiple primitives are interconnected using links. Input and output port are typed, and only ports with matching type can be connected. An input port of a primitive can be connected to at most one output port, but several input ports may be connected to a single output port. Both input and output ports may be unconnected, however primitives can require input ports to be connected.

Links may not form cycles, except when using a special *Pre* primitive. For more details see Section 5.1.

4.3 Fragments

Multiple primitives with connecting links can be grouped together into a *fragment*, which then behaves like any other primitive. Therefore, fragments also have input and output ports, but no parameters. The input ports of a fragment behave like output ports inside the fragment, thus primitives contained in the fragment can connect their input ports to input ports of the fragment. Output ports of the fragment behave like input ports to the inside and can be connected to output ports of primitives contained in the fragment. Because fragments have the same interface as primitives to the outside, they can also be nested into other fragments.

Usually, not all primitives within a primitive net are active at the same time. For example, whenever several motions must be executed by the robot sequentially and with real-time guarantees, all three motions must be represented within the same primitive net. But in this primitive net, it is never possible that more than one motion is active at the same time. To handle such cases within a primitive net, Boolean dataflows are used which tell every primitive using an input port whether or not they are currently active. Primitives need to check this input at the beginning of execution and immediately terminate if they are not supposed to be active. However, every single primitive has to check whether it is active or not, which can cause considerable overhead.

Fragments allow a hierarchical design of primitive nets. Net parts which cannot be active at the same time can be separated into different fragments. If a fragment is disabled using the aforementioned activation dataflow, only the fragment will check its activation state, but none of the contained primitives. For hierarchical primitive nets, this reduces the necessary overhead.

The whole primitive net itself is considered as the *root fragment*. The root fragment has two distinguished outputs: Those two outputs carry special dataflows for detection of termination and errors (cf. Sec. 5.2).

5 Execution

The Realtime Primitives Interface is tailored to the need of robotics applications. This influences the execution semantics, the lifecycle of a single primitive net and also the process of switching from one primitive net to the next.

5.1 Execution Semantics

Primitive nets are executed cyclically, i.e. every primitive instance is executed once in each execution cycle. Each primitive must read its input ports, perform all necessary calculations and write the results to the output ports. All primitives are topologically sorted using the links between the primitives, hence new values will propagate throughout the whole primitive net within one execution cycle.

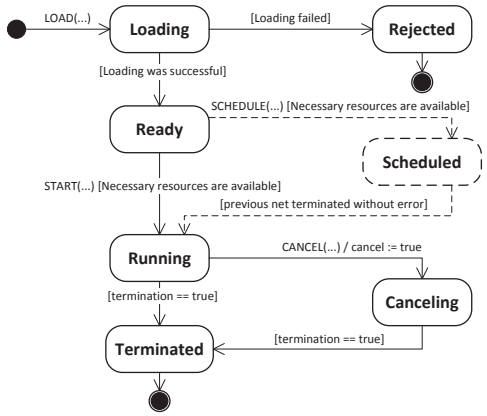

Figure 3 Primitive net lifecycle (dashed items are extensions for net scheduling, c.f. Sect. 5.3)

In order to allow a primitive net to being topologically sorted, links must not form cycles. However, there are use-cases where cycles in the dataflow are inevitable, e.g. for closed-loop control. To accommodate for these cases, a special *Pre* primitive exists, which deliberately delays the dataflow to the next execution cycle. Links connected to Pre primitives are ignored when the primitive net is topologically sorted, hence cycles with links are allowed if a Pre primitive is inserted at an appropriate position. Pre primitives can also be used inside fragments.

A Robot Control Core can execute multiple primitive nets simultaneously, but those nets must not access the same resources. It is for example possible to control two robots independently with two different primitive nets, but it is not allowed to control a single robot from multiple nets at the same time.

5.2 Lifecycle

Primitive nets have a life cycle with several possible states of execution (c.f. Fig. 3):

- **Loading** The primitive net has been successfully transmitted to the RCC, which is currently loading and preparing all necessary primitives.
- **Rejected** If the RCC cannot successfully load all necessary modules, the primitive net must be rejected. This can occur for several reasons:
  - The specified primitive is not available at this RCC.
  - A specified parameter of a primitive is invalid, e.g. the specified robot is not available.
  - A required input port of a primitive is not connected.
- **Ready** The execution environment has successfully instantiated all necessary primitives, and thus, the primitive net is now ready for execution.
- **Scheduled** The primitive has been scheduled for immediate execution after another primitive net (c.f. Sect. 5.3).
- **Running** The RCC is currently executing the primitive net and controlling the devices. Unique resources are assigned to this net and cannot be used by any other net.
- **Canceling** The execution environment initiated the canceling of the primitive net, and the net has the possibility to gracefully stop its execution.
- **Terminated** The execution of the primitive net has finished. All devices are inactive and unique resources have been made available again.

When a Java application is executed, the Robotics API automatically generates primitive nets and subsequently loads them into the RCC using the operation `LOAD(net)`. First, the Robot Control Core inspects and validates the primitive net (checks for syntactical validity, matching types of linked ports, etc.) and then starts to initialize all necessary primitives. Hence, the net is in state *Loading*. The operation is asynchronous, i.e. the primitive net does not necessarily have to have finished loading when the operation `LOAD` returns. The return value is a unique identifier for the net. After the execution environment has finished loading and preparing all necessary modules, the primitive net state changes to *Ready*. If loading the net fails, it enters state *Rejected*.

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

Besides, there are two more operations for controlling a running primitive net. The operation `CANCEL(net_id)` tries to gracefully stop the specified net, whereas `ABORT(net_id)` immediately stops the execution of the primitive net. The cancel operation is implemented by injecting a Boolean dataflow into the primitive net, which can decide which action to take and eventually terminate.

A primitive net must signal its termination by setting a Boolean dataflow to *true* which is connected to a designated *Termination* output port on the root fragment. If a primitive net is canceled, it must also signal termination after finishing. If a primitive net is aborted, no signaling is used, but the primitive net is immediately terminated without notice.

Primitives must guarantee a worst case execution time (WCET) for all operations performed during the *Running* state of primitive net in order to satisfy real-time requirements of the robotic system. Therefore, all necessary initialization operations such as memory allocation must be performed during the *Loading* phase.

5.3 Primitive Net Scheduling

For some applications like e.g. motion blending, it is desirable to be able to switch from one primitive net (the first motion) to another primitive net (the second motion) without the robot needing to stop. Embedding both motions into a single primitive net is possible, but creates huge primitive nets and requires hard-to-understand code instead of simple successive motion commands in the high-level programming language. Furthermore, it is not possible to specify an infinite chain of motions, because a primitive net can only be started once it has been completely specified.

Real-time guarantees can only be given within one primitive net. Because the high-level programming language cannot guarantee any timing limits, it cannot be assumed that a successive primitive net will be completely loaded on the RCC before the currently running primitive net is finished. Therefore, a device may never be in an unstable state (e.g. moving, applying force) after a primitive net has finished, if there is no successor completely loaded and waiting for execution.

With the primitive net scheduling feature, it is possible to request a second primitive net (N_s) to be executed right after a specified first primitive net (N_p) has terminated. The RCC guarantees that no temporal gap will occur between the execution of both commands, if the second primitive net was completely loaded and initialized before the first primitive net terminates. By encapsulating distinct motion tasks in separate primitive net and sequentially submitting (and scheduling) those nets to the RCC, it is now possible to use standard language features to handle motion blending on the robotics application level. Even infinite chains of motions with motion blending are possible.

If a primitive net N_s has been successfully scheduled for execution after another primitive net N_p, it enters the dedicated *Scheduled* state (cf. Fig. 3). The primitive net N_p will be notified using an injected Boolean *Takeover* dataflow. The primitive net N_p can then decide to terminate prematurely and to allow being taken over. If the primitive net N_p terminates without error, the primitive net N_s will be executed in the next cycle, i.e. there is no gap and the interval between the last execution cycle of N_p and the first execution cycle of N_s is equal to the interval between two execution cycles within one primitive net.

Fig. 4 shows two possible execution traces for scheduling one primitive net N_s for execution after another one (N_p). In case (a), the primitive net N_s enters state scheduled (S) before the currently running primitive net N_p is in an interval in which it allows being taken over (marked with TA). In the first execution cycle during this interval, the primitive net N_p detects that a scheduled primitive net is waiting and thus is terminating prematurely, and the primitive net N_s is immediately started. In case (b), the primitive net N_s was scheduled too late, i.e. it reached scheduled state (S) after the TA interval of the running primitive net. The primitive net N_p continues execution until its regular end and terminates, which immediately triggers the execution of primitive net N_s. This net however has to take care of a different starting situation.

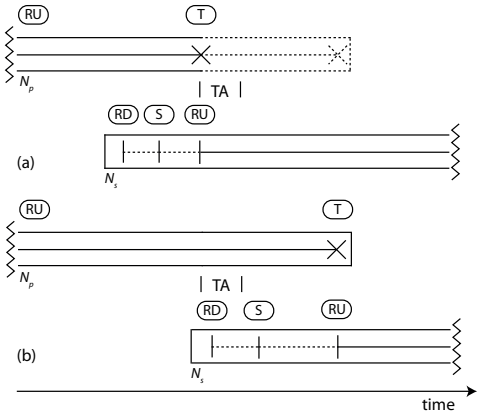

Figure 4 Scheduling of two primitive nets. Net states: (RD) ready, (RU) running, (S) scheduled, (T) terminated. TA specifies the interval in which the first primitive net allows being taken over.

Primitive nets are allowed to silently ignore takeover requests, but may never require to be taken over. If there is no successor, a primitive net must always be able to reach a safe and stable state, i.e. bringing all actuators to halt and remove any potential dangerous situation like applying force at at workpiece before terminating. On the other side, primitive nets aiming at taking over another primitive net must deal with the case that the predecessor has terminated regularly without giving the opportunity to take over prematurely. Special care must be taken for resources occupied by primitive nets, because scheduled primitive nets will most likely require resources which are currently in use. Scheduled primitive nets are allowed to use any resource that is in use by the primitive net on which they are scheduled, plus any resource which is not in use at the time of scheduling. Such unused resources must be immediately allocated to the scheduled primitive net to prevent other primitive nets from using these resources while the scheduled primitive net is waiting.

To support motion blending, the first running primitive net, which cannot know details about the following motion, must be able to terminate prematurely on a given blending condition, and the following primitive net must be able to take over at this particular point. The robot might be still moving with considerable velocity. The Robotics API or any high-level planning and execution layers must use information about the preceding motion to craft a primitive net which is capable of taking over the motion. A condition to decide when to allow being taken over can for example be if a certain percentage of the motion distance has passed. Using this scheduling mechanism, motion blending can be performed only on a best-effort basis. Depending on the load of the high-level system and the speed of e.g. planning algorithms, blending motions may or may not be successful. However, it is

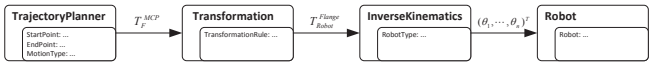

Figure 5 primitive net example: a robot follows a Cartesian trajectory.

guaranteed that no unsafe situation can occur in which devices lack necessary control. If it is not acceptable that at a certain position the blending of two motions occasionally fails, those motions must be forced into one large primitive net, or more complex motion types like splines (Horsch and Jüttler, 1998) should be considered.

6 Examples

The concepts of the realtime primitives interface and the execution of primitive nets are demonstrated with two short examples. The first example demonstrates one primitive net with its internal structure. The second example covers the real-time switching between two primitive nets for motion blending. Additionally, a complete application example is given.

6.1 Single Primitive Net

Fig. 5 shows an exemplary primitive net, consisting of four primitives and three links. This example is designed for the illustration of RPI and therefore is simplified in comparison to the primitive nets automatically generated by the Robotics API. The *TrajectoryPlanner* primitive generates a trajectory in Cartesian space (for some motion center point (MCP) in a given frame F) during its construction phase. In each execution cycle, a new point of the trajectory is delivered to the *Transformation* primitive which transforms this point to Cartesian coordinates of the robot flange in the robot base coordinate system. These coordinates are then converted to an array of joint values by the *InverseKinematics* primitive and finally used as input values of the Robot primitive. After all primitives have been executed, the new set-points for all actuators are provided to the respective hardware drivers. Due to the topological sorting, new set-points can be generated starting from the first execution cycle.

6.2 Primitive Net Scheduling

The following example demonstrates how primitive net scheduling and fragments can be employed for motion blending. Two primitive nets are necessary for making a robot follow the Cartesian trajectory from Fig. 1. The first primitive net specifies a linear motion from point A to C which can be taken over at point B. The second primitive net is more complex as it specifies two possible initial states: (1) If the last primitive net could be taken over, move from B to D on an curved path. (2) If the last primitive net could not be taken over, move linearly from C to D. Finally, if point D has been reached, continue moving linearly to point E. Fig. 6 and 7 show the real-time primitive nets that solve this task. Lst. 1

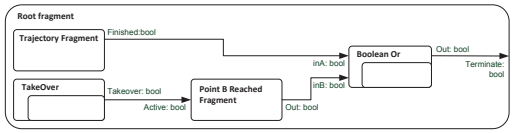

Figure 6 Primitive net for a motion from point A to C

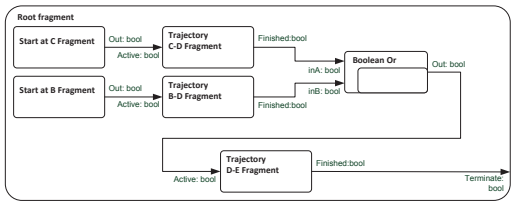

Figure 7 Primitive net for a motion from point C to E with blending

```
for(int i = 1; i < 10; i++) {
  robot.lin(pointC).beginExecute();
  robot.lin(pointE).beginExecute();
}
```

Listing 1 Java code for example motion program

shows the corresponding Robotics API program in Java for executing these two motions in a loop (for details see Schierl et al., 2012a).

The first primitive net (cf. Fig. 6) contains a *Trajectory Fragment* which internally calculates the trajectory and, in every execution cycle, delivers a new set-point to the robot. The *TakeOver* primitive returns true on its output if another primitive net is scheduled. The *Point B Reached Fragment* is only activated and thus evaluated in case of a scheduled successor primitive net. If either this fragment decides that point B has been reached or the main trajectory fragment has finished the motion, the primitive net will signal termination to the root fragment's Terminate port (by combining both Boolean outputs using an OR primitive).

The second primitive net (cf. Fig. 7) employs two special fragments *Start at C Fragment* and *Start at B Fragment* which can decide whether the motion was started at point B or C respectively. Depending on the outcome of these fragments, either the trajectory fragment for the motion from C to D or from B to D is activated. Both fragments can be quite large, but the performance impact is reduced because the content of one fragment remains constantly inactive. After either fragment has finished, the final fragment for the motion from D to E is activated.

6.3 Application example

The use of primitive nets and scheduling was demonstrated in a factory application which was especially developed for the final presentation of the *SoftRobot* project. This application includes a mobile platform which carries raw workpiece parts in supply boxes to a workbench with two KUKA lightweight robots. Those robots cooperatively lift the heavy supply boxes from

the platform onto the workbench, before assembling the workpieces from their parts. To perform these complex tasks, a great amount of real-time cooperation and coordination is required. For example, lifting the supply boxes from the platform to the workbench is performed with several linear motions which must be perfectly synchronized between both robots in order neither to damage the robots nor the supply boxes. To achieve a smooth and fast movement, each linear motion must also be blended into the next one. The assembly of the workpieces is also done cooperatively. The workpieces consist of two parts which must be assembled and then screwed together. Each part is fetched by one robot from the supply boxes. While one robot holds the base part, the top part is inserted by the second robot. Using the built-in force-torque sensors of the lightweight robots, it is possible to detect when the top piece is completely inserted. While the first robot continues holding the assembled workpiece, the second robot uses an attached electric screw-driver to fetch a screw from a magazine and screwing both parts together. The impedance controller of the lightweight robot is employed to apply the force required for screwing. A video of the application can be found at `http://video.isse.de/factory`.

7 Requirement Revisited

After having introduced all features of the Realtime Primitive Interface, it is now time to recapitulate the requirements identified in Section 2.

1. **Usability.** Using the Activity Interface of the Robotics API, programs can be written such as demonstrated in Listing 1. The syntax is very close to the traditional programming languages, hiding all implementation details and real-time issues from the application developer.

2. **Multi robot systems.** By adding multiple actuator primitives into the same primitive net, multiple devices can be controlled synchronously. If no synchronous execution is required, it is also possible to execute multiple primitive nets, each controlling its own device independently. There are special classes provided in the Robotics API which can dynamically combine multiple robots to form a cooperating machine.

3. **Sensor support.** Sensors are integrated as primitives which provide the current measurement value at a very high frequency. Using this value, all kinds of calculations can be performed e.g. to adjust the planned robot motion to external sensor values.

4. **Extensibility.** New sensors and actuators can be integrated by implementing a corresponding driver in the Robot Control Core. By providing the appropriate classes within the Robotics API, access to these devices can be provided to application developers. If the logical interface of a device does not differ from an existing device, applications can use the new device without the need of any change in code.

5. **Special industrial robotics concepts.** Using the command scheduling feature, it is possible to implement features like motion blending or compliant motions in a safe way, without the need of encapsulating the whole program into a single, huge command.

8 Implementation

Within the SoftRobot project, a complete reference implementation of the architecture described in Section 3 has been created. The Robot Control Core has been developed using the Orocos framework created by Bruyninckx (2001) and is therefore called *OrocosRCC*. The OrocosRCC runs under Linux with real-time extensions (RTAI or Xenomai) and also features a Windows version for testing purposes (without any real-time guarantees).

The OrocosRCC currently provides drivers for KUKA lightweight robots (LWR) using the Fast Research Interface (FRI, c.f. Schreiber et al. (2010)). Using this interface, stable robot control is possible using the OrocosRCC at cycle times of 2ms. At least two robots can be synchronously controlled at this speed by one single controller. Besides KUKA lightweight robots, the OrocosRCC also provides support for controlling a KUKA youBot consisting of a 5-DOF manipulator mounted on a 3-DOF mobile platform. Manipulator and platform are controlled using a proprietary protocol over the EtherCAT connection provided by the devices. A recent addition was the possibility to control a Staubli TX90 robot with uniVAL, using the CiA 402 device profile (IEC 61800-7-201, 2007; IEC 61800-7-301, 2007) over EtherCAT. Furthermore, it is possible to control a Segway RMP-50 omnidirectional platform which is connected using CAN over USB.

The driver module controlling a single robot is running an a high-priority real-time thread. By using the scheduling mechanisms provided by the real-time operating system, we can guarantee deterministic cycle times for hardware control as low as 500 µs. Some hardware devices require to synchronize the device driver with the device's clock (e.g. the RCC device driver must be synchronized to SYNC0 events on an EtherCAT bus). In this case, the device driver has to implement some form of PI controller to adjust to the given work cycle.

Primitive nets are executed within their own thread using a fixed cycle time. All hardware devices which need to be synchronized must be controlled from within the same primitive net. By adhering to that rule, it can be guaranteed that all device drivers are provided with new set-points at the same time. Because different devices may run at different cycle times, it can be necessary for

Device	Cycle Time	Protocol
KUKA LWR	2 ms - 12 ms	FRI over UDP
Staubli TX90	2 ms - 4 ms	CiA 402 over EC
KUKA youBot	4 ms	Prop. over EC
Segway RMP-50	50 ms	CAN over USB

Table 1 Supported hardware with tested cycle times for hardware control. Primitive nets have been executed with 2 ms cycle time in all experiments.

the device drivers to calculate smoothed set-points if the primitive net cycle and the hardware control cycle are not perfectly synchronous. Usually primitive nets are not synchronized to hardware drivers, because several different pieces of hardware can run at different clock cycles, and it is not always possible to synchronize all clocks.

Table 1 shows an overview of hardware experimentally tested with our Robot Control Core. All primitive nets have been executed with a cycle time of 2 ms. Hardware control cycle times vary for different types of hardware and are usually lower than the primitive net cycle time. Otherwise there would not be a new set-point for each hardware control cycle, and the hardware driver would need to interpolate the given set-points.

The communication between the Robotics API and the OrocosRCC is performed using TCP/IP sockets. For debugging and development purposes, the OrocosRCC includes an HTTP server, which serves XML pages with status information and receives new primitive nets encoded in XML. For production use, a special TCP/IP based protocol is used to avoid the overhead of HTTP and XML. Using this protocol, the Robotics API can transfer new primitive nets and can control the lifecycle of each primitive net.

There are special primitives which support non real-time data exchange between a running primitive net and the controlling Java application. Using this mechanism, it is e.g. possible to react to events detected by a sensor. However, this communication link cannot guarantee any time limits and therefore cannot be used to react to events where a guaranteed response time is needed. Those events must be completely handled within the primitive net.

9 Conclusion

Hardware control and continuous execution of real-time critical robotic tasks can be performed using a Robot Control Core and the concepts described in this article. Using the SoftRobot architecture with the Robotics API on top of the Robot Control Core, the development of robotics application completely integrated in modern software engineering methods becomes possible. Synchronized real-time control of multiple robots and integration with tools and sensors is thus made available to Java programs. Based on the Java language, a broad variety of tools, libraries, etc. is readily available which aids the development of applications.

There are plans of extending the current primitive net scheduling mechanism to support a finer grade of scheduling rules. Currently all primitive nets which want to take over a running net must be capable of handling multiple different initial situations depending whether scheduling was successful or not. It is planned to extend the scheduling mechanism to select a successing primitive net depending on the result value provided by the previous net. Using this mechanisms, the distinction of cases is delegated to the RCC, allowing for smaller primitive nets and faster freeing of unused resources.

References

A. Angerer, A. Hoffmann, A. Schierl, M. Vistein, and W. Reif. The Robotics API: An object-oriented framework for modeling industrial robotics applications. In *Proc. 2010 IEEE/RSJ Intl. Conf. on Intell. Robots and Systems, Taipeh, Taiwan*, pages 4036–4041. IEEE, Oct. 2010.

L. B. Becker and C. E. Pereira. SIMOO-RT – An object oriented framework for the development of real-time industrial automation systems. *IEEE Transactions on Robotics and Automation*, 18(4):421–430, Aug. 2002.

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

R. P. Bonasso, D. Kortenkamp, D. P. Miller, and M. Slack. Experiences with an architecture for intelligent, reactive agents. *J. of Experimental and Theoretical Artificial Intelligence*, 9:237–256, 1995.

J.-J. Borrelly, E. Coste-Manière, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon, and N. Turro. The ORCCAD architecture. *Intl. Journal of Robotics Research*, 17(4):338–359, Apr. 1998.

O. Brock, J. Kuffner, and J. Xiao. Motion for manipulation tasks. In B. Siciliano and O. Khatib, editors, *Springer Handbook of Robotics*, chapter 26, pages 615–645. Springer, Berlin, Heidelberg, 2008.

H. Bruyninckx. Open robot control software: the OROCOS project. In *Proc. 2001 IEEE Intl. Conf. on Robot. & Autom., Seoul, Korea*, pages 2523–2528. IEEE, May 2001.

B. Finkemeyer, T. Kröger, and F. M. Wahl. Placing of objects in unknown environments. In *Proc. 9th IEEE Intl. Conf. on Methods and Models in Automation and Robotics*, pages 975–980, Miedzyzdroje, Poland, Aug. 2003.

N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sept. 1991.

V. Hayward and R. P. Paul. Robot manipulator control under unix RCCL: A robot control C library. *International Journal of Robotics Research*, 5(4):94–111, 1986.

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

T. Horsch and B. Jüttler. Cartesian spline interpolation for industrial robots. *Computer-Aided Design*, 30(3):217–224, 1998. ISSN 0010-4485. doi: 10.1016/S0010-4485(97)00061-4.

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.

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.

M. S. Loffler, V. Chitrakaran, and D. M. Dawson. Design and implementation of the Robotic Platform. *Journal of Intelligent and Robotic System*, 39:105–129, 2004.

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

C. Pelich and F. M. Wahl. ZERO++: An OOP environment for multiprocessor robot control. *International Journal of Robotics and Automation*, 12(2):49–57, 1997.

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 *Workshop on Open Source Software, IEEE Intl. Conf. on Robot. & Autom., Kobe, Japan*, 2009.

A. Schierl, A. Angerer, A. Hoffmann, M. Vistein, and W. Reif. Using Java for real-time critical industrial robot programming. In *Workshop on Softw. Developm. & Integr. in Robotics. IEEE Intl. Conf. on Robot. & Autom., St. Paul, USA*, 2012a.

A. Schierl, A. Angerer, A. Hoffmann, M. Vistein, and W. Reif. From robot commands to real-time robot control - transforming high-level robot commands into real-time dataflow graphs. In *Proc. 2012 Intl. Conf. on Inform. in Control, Autom. & Robot., Rome, Italy*, 2012b.

G. Schreiber, A. Stemmer, and R. Bischoff. The Fast Research Interface for the KUKA Lightweight Robot. In *Workshop on Innovative Robot Control Architectures for Demanding (Research) Applications. IEEE Intl. Conf. on Robot. & Autom.*, Anchorage, Alaska, USA, May 2010.

R. Simmons. Concurrent planning and execution for autonomous robots. *IEEE Control Systems*, 12(1):46–50, Feb. 1992. ISSN 1066-033X. doi: 10.1109/37.120453.

R. Smits, T. D. Laet, K. Claes, P. Soetens, J. D. Schutter, and H. Bruyninckx. Orocos: A software framework for complex sensor-driven robot tasks. *IEEE Robotics & Automation Magazine*, 2009.

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 *Proc. 2012 Intl. Conf. on Mechatronics and Automation, Chengdu, China*, pages 1329–1334, Aug. 2012. doi: 10.1109/ICMA.2012.6284329.

C. Zieliński. Object-oriented robot programming. *Robotica*, 15(1):41–48, 1997.

C. Zieliński and T. Winiarski. Motion generation in the MRROC++ robot programming framework. *Intl. J. of Robotics Research*, 29(4):386–413, 2010. doi: 10.1177/0278364909348761.