Instantaneous switching between real-time commands for continuous execution of complex 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. 2012. "Instantaneous switching between real-time commands for continuous execution of complex robotic tasks." In *2012 IEEE International Conference on Mechatronics and Automation, 5-8 Aug. 2012, Chengdu, China*, edited by Shuguo Wang, Tatsuo Arai, Xinkai Chen, Jie Zhao, and Yili Fu, 1329–34. Piscataway, NJ: IEEE.
https://doi.org/10.1109/icma.2012.6284329.

DOI: 10.1109/ICMA.2012.6284329

Instantaneous Switching between Real-Time Commands for Continuous Execution of Complex Robotic Tasks

Michael Vistein, Andreas Angerer, Alwin Hoffmann, Andreas Schierl and Wolfgang Reif
Institute for Software & Systems Engineering, University of Augsburg
Universitätsstraße 6a, 86135 Augsburg, Germany
{*vistein,angerer,hoffmann,schierl,reif*}@*informatik.uni-augsburg.de*

Abstract— **An application program for one or even several industrial robots usually consists of a number of disjoint commands, with each command controlling the robot to perform a certain task like motions or tool interactions. Sometimes it is desirable to be able to switch from one such command to another command with time guarantees for the switching progress, e.g. for blending one motion into another. In this paper we propose an approach to achieve this with two separate commands, where the second command can be created while the first is already being executed.**

Index Terms— **robot programming, motion blending, real-time, command scheduling**

I. 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 [1] that separates the layers *Planning*, *Execution* and *Behavioral Control* of robot task structures.

Within the research project *SoftRobot*, a software architecture [2] has been created that enables programming complex real-time critical industrial robot applications in Java (see Fig. 1). 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. Planning and Execution are handled by the Java-based *Robotics API* [3] and the concrete applications running on top of it.

In this work, we focus on the interface to the RCC, the *Realtime Primitives Interface (RPI)*, which was introduced

This work presents results of research project *SoftRobot* which was funded by the European Union and the Bavarian government and completed in 2012. The project was carried out together with KUKA Laboratories GmbH and MRK-Systeme GmbH and was kindly supported by VDI/VDE-IT GmbH.

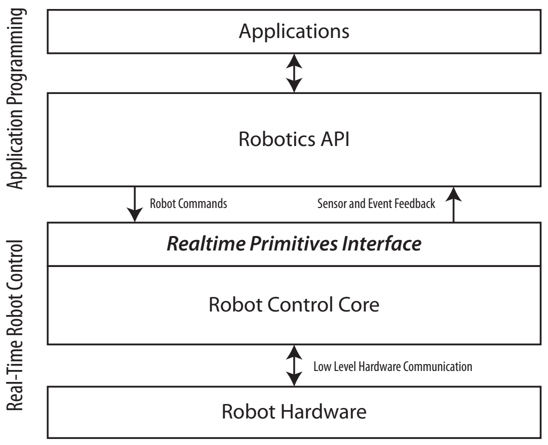

Fig. 1. SoftRobot architecture

in [4]. This interface accepts commands specified in a dataflow language. Such real-time commands 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.

In certain cases, it is necessary to instantaneously switch between commands in a real-time critical way. A classical 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 [5] contains formally verified mechanisms that guarantee minimal latency for e.g. the transitions between sequential robot behaviors. TCA [6] provides a mechanism for synchronization of planning and execution steps in a message based control system. In MiRPA [7], transitions between skill primitives preserve the system stability as well, while MRROC++ [8] supports a seamless takeover of a preceding motion instruction. Other popular robot control frameworks like Orocos [9] require the application developer to implement mechanisms for real-time critical transition between robot behaviors if required, while some frameworks like ROS [10] [11] do not provide real-time support at all.

This paper proposes a generic mechanism for real-time critical switching from a running command to dynamically loaded subsequent commands. Sect. II gives an overview of the RPI basics. Sect. III describes use cases that require the aforementioned switching extension. Sect. IV and V provide details about two new concepts that were included in RPI to tackle these requirements. Sect. VI illustrates these concepts using a detailed example. Finally, the paper is concluded in Sect. VII.

II. Realtime Primitives Interface

The Realtime Primitives Interface provides a generic interface to (real-time) robot controllers. It allows a very flexible specification of complex (sensor-based) *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.

A. Commands

A *command* controls a single task that should be performed by a robotic system. Such tasks can be an industrial robot moving from one point to another, a robot moving along a line while a welding torch is turned on and off at appropriate times or even two robots cooperating to transport a heavy workpiece together. All these actions have in common that they are inherently real-time critical. For example, the interpolation of a simple transfer motion of a single robot is real-time critical as it should result in a smooth and precise movement. The cooperation of multiple robots requires a precise synchronization of all robots, otherwise these robots or the environment might be severely damaged. To achieve this synchronization, real-time critical tasks must be encapsulated into one command which can be issued to the robot controller and be executed there respecting hard real-time constraints. While a command is running, hardware can be controlled e.g. by issuing new set-points to a robot at precise intervals (like e.g. every 1 ms).

After a command has finished, the system must be in a safe and stable state. Due to the separation of device control and application logic, there can be no guarantee that another command is already available to resume controlling the robotic system. Hence, it is important to be in a safe and stable state, because an unknown period of time without active hardware control can occur after a command has finished. Therefore, no robot should be in motion or be applying force to a workpiece after a command has finished.

B. Command Components

RPI is designed as a dataflow language similar to the Lustre [12] language, used in the commercial SCADE Suite, and tailored to industrial robotics. The key building blocks are *primitives* interconnected with *links*, which together form *commands* which can be executed by a Robot Control Core. The execution is performed cyclically, i.e. the whole command is evaluated repeatedly at exact intervals and therefore hardware can be controlled precisely.

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[1].

Multiple instances of the same primitive can be used in a single command. 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 command, 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 commands to use the same resources accidentally, and the RCC guarantees that all requested resources will be available once a command has been accepted for execution. Parameters can be used to configure primitive instances. Actuator primitives for example require a parameter indicating which device actually to control. Parameters can only be set upon specification of a command and are immutable during execution. Primitives may have an internal state to preserve information like the current position of a trajectory over execution cycles.

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.

3) Commands: Multiple primitives connected with links form commands. These commands usually must be acyclic, cycles may only exist if they contain at least one instance of the special *Pre* primitive which delays the dataflow for one command execution cycle.

C. Execution

The execution of a command consists of several steps. Its life cycle, i.e. all possible states, is shown in Fig. 2. At first, all primitives are instantiated and links are created (cf. state *Loading*). At this time, primitives may perform any necessary

[1]Additionally, actuator primitives can have special error outputs which are treated separately.

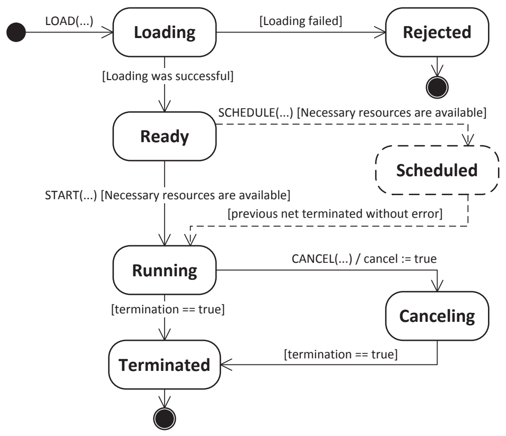
LOAD(...)
Loading
[Loading failed]
Rejected
[Loading was successful]
Ready
SCHEDULE(...) [Necessary resources are available]
Scheduled
START(...) [Necessary resources are available]
[previous net terminated without error]
Running
CANCEL(...) / cancel := true
Canceling
[termination == true]
Terminated
[termination == true]

Fig. 2. Command lifecycle (dashed items are extensions for command scheduling, c.f. Sect. IV)

initialization, including memory allocation, which must not be done later while being executing in a real-time context. At this stage, no resources are allocated yet for the command. All primitives are topologically sorted, thereby ignoring links connected to the special *Pre* primitives. If no error occurs, the command is *Ready* for execution, otherwise it will be *Rejected*. After the command has been successfully loaded, the start of the command can be requested. If all required resources are available, the command enters the state *Running*, blocking the required resources from further usage by other commands. If not all resources are available, the command cannot be started and remains ready for later starting.

While a command is running, it is executed repeatedly at a constant frequency (1 kHz by default). In each execution cycle, every primitive is executed exactly once, i.e. the primitive can read its input ports, performs calculations and can finally write data on its output ports. Primitives must guarantee a worst case execution time (WCET) during this phase. Hence, all tasks with unknown WCET (e.g. memory allocation) must already be performed during the loading phase. Because the primitives are executed in the determined topological order, it is guaranteed that every primitive will have current values available at all input ports, and new data values propagate through the whole command within a single execution cycle.

Fig. 3 shows a exemplary command, consisting of four primitives and three links. This example is designed for the illustration of RPI and therefore is simplified in comparison to the commands automatically generated by the Robotics API (cf. Fig. 1). 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

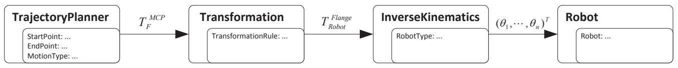

Fig. 3. Command example: a robot follows a Cartesian trajectory.

actuators are provided to the respective hardware drivers. Due to the topological sorting, new set-points can be generated from the first execution cycle on.

If a command is in the *Running* state, it can be requested to gracefully terminate execution by calling the *cancel* operation. This will be signaled to the command using a special *Cancel* primitive and the command may terminate at its own decision. If a command decides to terminate, it must signal this by setting a designated Boolean-typed dataflow to *true*. If this value is detected, the RCC will terminate the running command after the current cycle has completed and actuators have been provided with new set-points. After the command has terminated, all resources are freed. Another designated integer-typed dataflow can be used to indicate an error number, which can later be retrieved, even after the termination of the command.

III. Requirements Revisited

The main contribution of RPI is that it allows to describe flexible, encapsulated real-time critical actions and therefore separates real-time control from high-level programming. It has already been successfully applied to various scenarios like synchronized control of multiple robots. However, it became clear that the requirement that a command can only terminate in a safe and stable actuator state (i.e. the robot is not moving) is too strong in some cases and must be revised. The following use cases illustrate safety (cf. Sect. III-A) as well as performance reasons (cf. Sect. III-B) that make it necessary to extend the execution mechanism of real-time commands in RPI (cf. Sect. III-C).

A. Assembly Tasks

In robotics, manipulation or assembly tasks [13] are getting more and more important. However, these tasks are very challenging due to the necessity to handle uncertainty and 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 (cf. [7]). 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 unforeseen events.

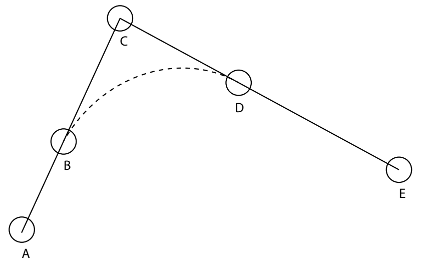

Fig. 4. Motion blending

B. 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. 4 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.

Blending between two motion commands in RPI by simply issuing two subsequent commands is not possible, because there might be a time gap between the execution of both commands. Moreover, a command is not allowed to terminate while moving the robot at a considerable velocity which would be necessary for blending.

C. Solution Approach

The straightforward solution for both use cases was to combine all motion parts into a single command. However, this approach suffers from two major drawbacks: (1) This solution will massively increase the size of commands and, thus, will not scale with the complexity of the robotic task. Because every primitive in a command must be executed once in each execution cycle, it can be very hard to achieve small cycle times such as 1ms, which are necessary for smooth actuator control. (2) The control flow of the application needs to be coded into the real-time command to some extent. Hence, standard features for control flow in the (high-level) programming language (such as keywords like *for*, *if*, etc.) cannot be used, and furthermore (infinite) loops are impossible, because the real-time command needs to be generated completely prior to execution.

As this solution showed to be infeasible in general, an universal approach for meeting these revised requirements was developed. The main idea is to allow instantaneous switching from a currently running command to real-time commands which are issued later to the robot controller. Instantaneous means that the consecutive command will be running in the next execution cycle after the previous command has been terminated. Hence, a real-time command is now allowed to terminate in an unstable actuator state if and only if a consecutive command is present and will take over control of at least every affected actuator. The robot system must in general be in a safe and stable state if there is no running command at all. To allow instantaneous command switching, the execution mechanism of RPI had to be extended by *command scheduling* and *fragments* which are explained in detail in the next two sections.

IV. Command Scheduling

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

If a command C_s has been successfully scheduled for execution after another command C_p, it enters a new *Scheduled* state (cf. Fig. 2). The command C_p will be notified using a special *Takeover* primitive which emits a *true* value on a Boolean-typed dataflow. The command C_p can then decide to terminate prematurely and to allow being taken over. If the command C_p terminates without error, the command C_s will be executed in the next cycle, i.e. there is no gap and the interval between the last execution cycle of C_p and the first execution cycle of C_s is equal to the interval between two execution cycles within one command.

Fig. 5 shows two possible execution traces for scheduling one command C_s for execution after another one (C_p). In case (a), the command C_s enters state scheduled (S) before the currently running command C_p is in an interval in which it allows being taken over (marked with TA). In the first execution cycle during this interval, the command C_p detects that a scheduled command is waiting and thus is terminating prematurely, and the command C_s is immediately started. In case (b), the command C_s was scheduled too late, i.e. it reached scheduled state (S) after the TA interval of the running command. The command C_p continues execution until its regular end and terminates, which immediately triggers the execution of command C_s. This command however has to take care of a different starting situation.

Commands are allowed to silently ignore takeover requests, but may never require to be taken over. If there is no successor, a command 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, commands aiming at taking

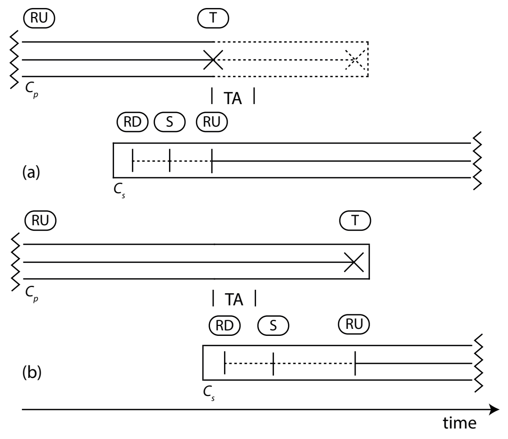

Fig. 5. Scheduling of two commands. Command states: (RD) ready, (RU) running, (S) scheduled, (T) terminated. TA specifies the interval in which the first command allows being taken over.

over another command 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 commands, because scheduled commands will most likely require resources which are currently in use. Scheduled commands are allowed to use any resource that is in use by the command 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 command to prevent other commands from using these resources while the scheduled command is waiting.

To support motion blending, the first running command, which cannot know details about the following motion, must be able to terminate prematurely on a given blending condition, and the following command 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 command which is capable of taking over the motion. A condition to decide when to allow being taken over can be e.g. if a certain percentage of the motion distance has passed. Using this scheduling mechanism, motion blending can be performed only on a best-effort base. 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 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 command, or more complex motion types like splines [14] should be considered.

V. Fragments

Usually, not all parts of a command are active at the same time. In motion blending for example, the robot can either be at the blending point (cf. point B in Fig. 4) or the previous motion proceeded the robot to its original destination (point C). Both situations however cannot happen at the same time. To handle such cases within a command, currently Boolean dataflows are used which tell every primitive using an input port whether or not they are currently active. Primitives usually 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 itself, which can cause considerable overhead. With the aforementioned realization of motion blending, this overhead caused execution time problems of RPI commands in some cases and led to the introduction of the *Fragments* concept.

Fragments allow a hierarchical design of primitive nets. Command parts which cannot be active at the same time can be separated into different fragments. Fragments can be used just like any other primitive, but contain nets of primitives themselves. If a fragment is disabled using the aforementioned activation dataflow, only the fragment itself will check its activation state, but none of the contained primitives which can be even further fragments. For hierarchical primitive nets, this reduces the necessary overhead.

Fragments have – just like other primitives – input and output ports but no parameters. The input ports of a fragment behave like output ports inside the fragment, thus primitives inside 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. The whole command itself is considered as the *root fragment*. The root fragment has two distinguished outputs: Those two outputs carry the special dataflows for detection of termination or error codes (cf. Sec. II-C).

Using fragments, primitives still need to be topologically sorted before execution. But it is sufficient to sort the content of each fragment (including the root fragment) on its own, as there is no interference between two different fragments. Cycles are allowed if at least one *Pre* primitive is used on the path, but cycles cannot span across multiple fragments. The *Pre* primitive must always be used at the same hierarchy level as the cycle.

VI. Example

The following example demonstrates command scheduling and fragments for motion blending. Two commands are necessary for making a robot follow the Cartesian trajectory from Fig. 4. The first command specifies a linear motion from point A to C which can be taken over at point B. The second command is more complex as it specifies two possible initial states: (1) If the last command could be taken over, move from B to D on an curved path. (2) If the last command 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 commands that solve this task. Lst. 1 shows the corresponding Robotics API program in Java for executing these two motions in a loop (for details see [15]).

The first command (cf. Fig. 6) contains a *Trajectory Fragment* which internally calculates the trajectory and, in every

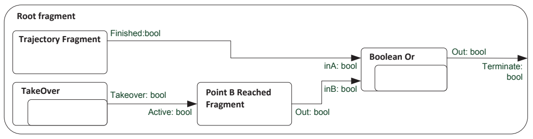

Fig. 6. Command for a motion from point A to C

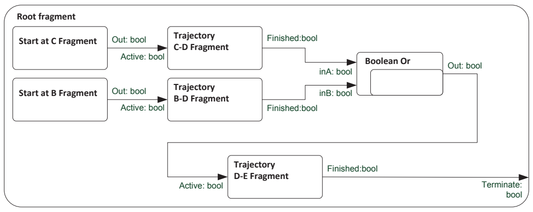

Fig. 7. Command for a motion from point C to E with blending

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

The second command (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.

VII. Conclusion

The introduction of *command scheduling* and *fragments* enables the Robotics API to perform highly dynamic and complex tasks with high performance. This 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, which is far too much to be performed in single commands. 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. To achieve a smooth and fast movement, each linear motion must also be blended into the next one. A video of the application can be found at [16].

Being able to schedule one command for execution directly after another one is sufficient for motion blending, but introduces complex internal command structures covering multiple initial states. A current plan for further extension is the possibility to schedule several commands on one command and decide which command to execute by means of previous the command's result code. With this approach, each successing command can be rather small and straightforward. The decision which command to start next is delegated to the RCC's scheduling algorithm and must only be evaluated once. After the decision for one successor has been made, all obsolete successors can be cleaned up immediately, thus freeing resources.

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

Listing 1. Java code for example motion program

References

[1] R. P. Bonasso, D. Kortenkamp, D. P. Miller, and M. Slack, "Experiences with an architecture for intelligent, reactive agents," *J. of Experimental and Theoretical Artificial Intelligence*, vol. 9, pp. 237–256, 1995.
[2] A. Hoffmann, A. Angerer, F. Ortmeier, M. Vistein, and W. Reif, "Hiding real-time: A new approach for the software development of industrial robots," in *Proc. 2009 IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems, St. Louis, MO, USA*, 2009.
[3] 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 Intelligent Robots and Systems, Taipeh, Taiwan*, 2010.
[4] M. Vistein, A. Angerer, A. Hoffmann, A. Schierl, and W. Reif, "Interfacing industrial robots using realtime primitives," in *Proc. IEEE Intl. Conf. on Automation and Logistics, Hong Kong*, 2010.
[5] J.-J. Borrelly, E. Coste-Manière, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon, and N. Turro, "The ORCCAD architecture," *Intl. J. of Robotics Research*, vol. 17, no. 4, pp. 338–359, Apr. 1998.
[6] R. Simmons, "Concurrent planning and execution for autonomous robots," *IEEE Control Systems*, vol. 12, no. 1, pp. 46–50, Feb. 1992.
[7] 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*, Miedzyzdroje, Poland, Aug. 2003, pp. 975–980.
[8] C. Zieliński and T. Winiarski, "Motion generation in the MRROC++ robot programming framework," *Intl. J. of Robotics Research*, vol. 29, no. 4, pp. 386–413, 2010.
[9] 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.
[10] ROS actionlib. [Online]. Available: http://ros.org/wiki/actionlib
[11] ROS SMACH. [Online]. Available: http://ros.org/wiki/smach
[12] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, Sept. 1991.
[13] O. Brock, J. Kuffner, and J. Xiao, "Motion for manipulation tasks," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Berlin, Heidelberg: Springer, 2008, ch. 26, pp. 615–645.
[14] T. Horsch and B. Jüttler, "Cartesian spline interpolation for industrial robots," *Computer-Aided Design*, vol. 30, no. 3, pp. 217–224, 1998.
[15] A. Schierl, A. Angerer, A. Hoffmann, M. Vistein, and W. Reif, "Using Java for real-time critical industrial robot programming," in *Workshop on Software Development and Integration in Robotics. IEEE Intl. Conf. on Robotics and Automation*, St. Paul, USA, 2012.
[16] Factory 2020. Institute for Software and Systems Engineering. [Online]. Available: http://video.isse.de/factory