Towards re-orchestration of real-time component systems in robotics

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

Angaben zur Veröffentlichung / Publication details:

Vistein, Michael, Alwin Hoffmann, Andreas Angerer, Andreas Schierl, and Wolfgang Reif. 2017. "Towards re-orchestration of real-time component systems in robotics." In *2017 First IEEE International Conference on Robotic Computing (IRC), 10-12 April 2017, Taichung, Taiwan*, edited by Chun-Ming Chang, Jianquan Liu, Bruce MacDonald, and Adriana Tapus, 60–68. Piscataway, NJ: IEEE. https://doi.org/10.1109/irc.2017.61.

DOI: 10.1109/IRC.2017.61

Towards Re-orchestration of Real-time Component Systems in Robotics

Michael Vistein, Alwin Hoffmann, Andreas Angerer, Andreas Schierl, and Wolfgang Reif
Institute for Software and Systems Engineering
University of Augsburg, Augsburg, Germany

***Abstract*—Over the last decade, there has been a trend towards component-based systems in robotics software engineering. Although the aspect of real-time is important to control manipulators with high velocities or forces, it has often been neglected. In this paper, we present a runtime environment for real-time component systems in robotics. It allows for a generic specification of computation tasks and employs a flexible, rule-based mechanism for composing and coordinating multiple components. Moreover, the runtime environment is capable of seamlessly re-orchestrating the composition of real-time components at run-time to adapt to new tasks – even when the robotics system is moving with high speed. For illustration, two examples are given.**

I. Introduction

Building robotics software today often means to develop software for single components that represent some algorithms or tasks. The overall system itself is comprised of a set of single components which should ideally be reusable. This trend towards component software engineering in robotics is obvious since about a decade now (e.g., see [1], [2], [3]) and manifests in the widespread and of course component-based Robot Operating System (ROS) [4] which is currently the de-facto standard in the research community. However, real-time is important in robotics to safely control the devices – especially with high velocities or force control. In ROS, for example, all real-time issues must be handled inside a single component [5]. To handle real-time requirements in robotics, the OROCOS framework [6] or aRDx [7] can be used which allows for implementing and executing real-time components. When using OPRoS [8] or OpenRTM [9] it highly depends on the communication layer, whether real-time issues must be handled inside a single component.

As stated above, real-time requirements should actually be essential for safe and high-performance robotics applications. However, a real-time component system should not be static, but be able to change the structure at runtime to adapt to new tasks – even when controlling devices with high-speed. When this re-orchestration is possible, automation tasks can easily be rearranged or adapted in a service-oriented manner – as suggested by Industry 4.0 initiatives. Hence, the main contribution of this paper is a flexible, rule-based mechanism to compose and coordinate real-time component systems. Furthermore, we introduce a runtime environment which is able to use this mechanism in order to re-orchestrate the structure of its real-time component system at run-time. Overall, this paper presents concepts, techniques and implementation notes how to realize such a rule-based re-orchestration.

The above mentioned runtime environment is based on a software architecture [10] which is geared towards realizing complex, sensor-guided manipulation tasks of single robots or small teams of robots. The main idea is that robotics software is developed against the modular *Robotics API* [11] and robot operations are executed with hard real-time guarantees on the underlying *Robot Control Core (RCC)* [12]. Thus, the RCC takes care of all real-time critical parts of the robotics systems and is responsible for controlling hardware devices. However, to fulfill the demanding requirements imposed by the software architecture mentioned above, the RCC has been designed to be a flexible runtime environment for composing and coordinating real-time components.

To promote a separation of concerns for robotic component systems, the idea of *5 Concerns* (5Cs) was introduced by Prassler et al. [13] and refined later (cf. [14] and [15]). The idea was inspired by Radestock and Eisenbach [16] where different concerns (4Cs) have been identified to facilitate the designing of large maintainable distributed systems. According to this, there are five different concerns in robotics to cope with in order to develop maintainable software:

- *Computation* defines the continuous behavior of an individual component. Hence, it represents the algorithmic or functional part of the system and its implementation provides an added value to the system [14].
- *Communication* is responsible for transporting required data to computational components.
- *Configuration* allows to influence the behavior and performance both of computation and communication.
- *Composition* describes how single components are connected to achieve an overall system behavior. However, the coupling between single components should be minimized.
- *Coordination* describes how connected components work together to achieve an overall system behavior. According to [14], coordination provides the *discrete behavior* of the system.

Each of the five concerns (5Cs) is used to describe the runtime environment and its concepts and ideas for building

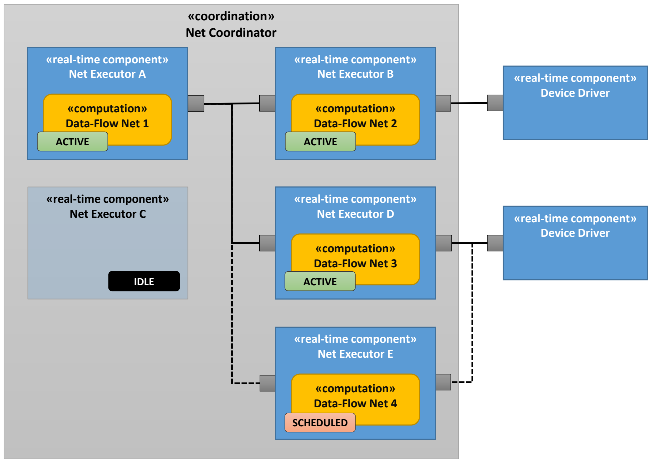


Figure 1. *Device drivers* are responsible for hardware control, whereas *Net Executor* components can execute generic computation tasks. The composition and coordination of computation tasks is realized by the *Net Coordinator* which can stop an active computation task (e.g., Data-Flow Net 3) and start a scheduled task instead (e.g., Data-Flow Net 4). A drawn line between two ports indicates active communication of two real-time components. Dotted lines between two components indicate future connections which will be established as soon as the scheduled component will be activated.

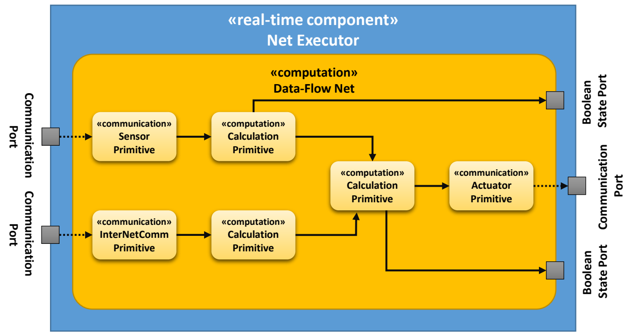


Figure 2. Computation tasks are specified in a data-flow language and composed of real-time primitives. Different means for communicating or signaling internal state are available. Each data-flow net is executed periodically by a *Net Executor* with real-time guarantees. Hence, it is possible to specify and execute arbitrary computation ranging from sensor processing to actuator control. A drawn line indicates communication between real-time primitives inside a data-flow net. Dotted lines indicate external communication established through ports of the *Net Executor*.

and re-orchestrating a real-time component system. It is worth mentioning that the RCC has a generic data-flow language to specify computation tasks and a declarative mechanism to compose and coordinate such computation tasks with synchronization rules. As a consequence, it allows for a re-orchestration of real-time components at run-time – even when a robotics system is moving with high speed.

The main concepts are shown in Fig. 1 incorporating each of the five concerns. Computation tasks, for example, can be specified in a data-flow language and are executed with reusable real-time executor components. The details can be found in Sect. II. For communication, there are different possibilities available depending on the purpose. For example, data-flow nets or their wrapping components respectively can communicate with each other in a generic and yet real-time capable way over dedicated ports. All communication options as well as configuration possibilities are described in Sect. III.

The composition is given by the set of currently active or scheduled net executors with data-flow nets and corresponding device drivers. A scheduled net executor is already loaded, but not yet executed. However, depending on a set of rules it can be started immediately and, e.g., can control a moving device by starting to communicate with its device driver. Accordingly, active net executors can be stopped. Because the set of active or scheduled data-flow nets can evolve slowly over time or even change rapidly, the composition of the overall system can change to adapt to new tasks. The coordination between components can be specified in a declarative way by synchronization rules. Such a synchronization rule describes under which circumstances running components or – to be precise – their data-flow nets should be stopped and scheduled ones should be started. Details are given in Sect. IV.

Moreover in Sect. V, the paper will present some notes about implementing such a runtime environment in order to seamlessly re-orchestrate real-time components at run-time. A short evaluation and two examples using industrial manipulator arms are introduced in Sect. VI – each example highlighting a different aspect of the presented approach. Finally, the paper will conclude with Sect. VII.

II. Computation

Computation defines the continuous behavior of an individual component and can thus be specified using a data-flow language (see Sect. II-A) and executed periodically (see Sect. II-B). Real-time components for controlling devices – thus constantly communicating with them – are treated separately (see Sect. II-C).

A. Specification of Computational Tasks

To be able to specify flexible computation modules, the *Realtime Primitives Interface* (RPI) was introduced in [17]. It consists of a data-flow language with a formal semantics, both inspired by Lustre [18]. The main concept of RPI are *data-flow nets* which describe a piece of real-time computation (see Fig. 2). Before the execution of a data-flow net can start, it must be completely specified, i.e., no further structural changes are possible. Hence, a data-flow net consists of basic building blocks – called primitives – which can be connected by links to transfer data from one primitive to another. The execution is performed cyclically, i.e., every contained primitive is executed once in each execution cycle.

Primitives can have both input and output ports and configuration is possible using parameters. 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 very basic operations such as logical operators ($\wedge$, $\vee$, etc.), mathematical functions (add, subtract, multiply, etc.), but also more complex operations such as calculating trajectories. Although arbitrary computation can be performed, all operations must be real-time safe such that every primitive can guarantee a worst case execution time. Furthermore, primitives may have an internal state which is preserved between two execution cycles. This can be used for example to interpolate trajectory in order to provide new set-points in each execution cycle.

Devices are also represented as primitives: Sensors are modeled as primitives with only output ports, whereas actuators are primitives with only input ports (cf. Fig. 2). Input and output ports are strictly typed, and only ports with matching types can be connected. Basic types such as Boolean, integer or double, but also more complex types (e.g., Cartesian coordinates) are possible. It is also possible to use a special null value to indicate that no valid data is available.

Links are used to connect identically typed ports of two primitives. Each input port can be connected to exactly one output port, however an output port may be connected to several input ports. The output value of a primitive is always transmitted to the input port of the following primitive before the execution of the latter is started. This allows for a fast propagation of values through the data-flow 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, i.e., it is not possible to reach a primitive again by purely navigating links from output to input ports without coming across a special *Pre* primitive. Such cycles would imply that the input for a primitive instantaneously depends on the result of this very same primitive. Pre primitives delay the propagation of values to the next cycle and thus can break up cycles when necessary.

Each data-flow net can have typed communication ports (provided by special primitives) which can be used to transmit data from and to other nets, devices or external systems (cf. Sect. III-A). Furthermore, there are also Boolean state ports which allow a running computational task to report its current state to the *Net Coordinator* which is used for coordination of multiple tasks (cf. Sect. IV-B).

B. Execution of Computational Tasks

Each computational task, specified as a data-flow net, is executed in a dedicated real-time component, the *Net Executor*. Hence, a *Net Executor* is responsible for executing exactly one computational task at a time. Execution is performed periodically, i.e., all primitives contained in the data-flow net are executed once in each cycle. Typical execution frequencies are 0.5 kHz to 1 kHz. As multiple data-flow nets can run in parallel and are executed independently using their own real-time executor, different execution frequencies and priorities are possible. For example, a monitoring task reporting measurements to an human-machine-interface (HMI) may have lower frequency and priority than a closed-loop controller.

According to the input-process-output model, each execution cycle is split into three separate phases which are executed sequentially:

1) Reading sensor values (over communication ports)
2) Performing computation
3) Writing actuator values (over communication ports)

While phases 1 and 3 are mainly for communication (cf. Sect. III-A), the computation is completely performed in phase 2. During the first phase, all primitives connected to sensors are requested to update their current sensor values, which usually includes communication with the respective device driver (cf. Sect. II-C). This allows to retrieve consistent sensor values, i.e., all values are provided for the same point in time. This is necessary as the execution order of primitives is generated by topologically sorting the data-flow net, which could put sensor primitives near the end of an execution cycle. As actuators are only provided with new set-points in phase 3, no changes to the system are made in both previous phases. This allows to interrupt the execution of a data-flow net during any time before the third phase has been started which is important for switching between different behaviors in real-time as Sect. IV-B will show. Phase 3 mainly consists of providing new data to actuator drivers over communication port. Potentially time consuming communication with hardware devices must be done within the actuator drivers and not within a computation task. Therefore, the execution of phase 3 quick compared to phase 2.

The real-time component of a data-flow net can have several states during its life-cycle (cf. Fig. 3). Once a data-flow net has been successfully loaded, it is in state *Ready*. However, it also can be *Rejected*, if it is syntactically invalid or at least one primitive is not available. Being *Ready*, a data-flow net can either be directly started entering the state *Running*, or *scheduled* for later execution. Scheduled data-flow nets are started using *synchronization rules* and allow real-time transitions between multiple data-flow nets. More details are explained in Sect. IV-B. Once a data-flow net has finished its computational task, cyclical execution is stopped and the state changes to *Terminated*. The same applies if it was canceled, e.g., by a supervisory application or component.

C. Device Drivers

Hardware devices can be controlled from computational tasks by using specific actuator primitives. Since computational tasks can run using different frequencies, and also

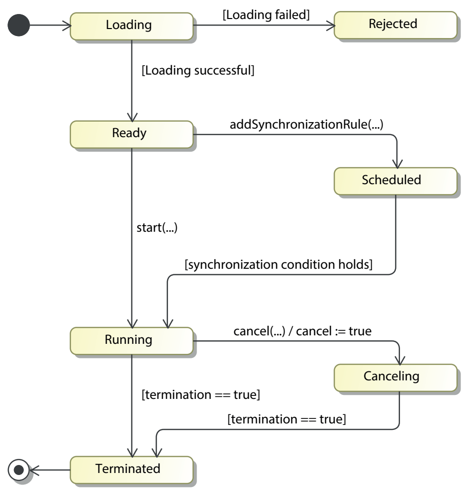


Figure 3. Life-cycle of a data-flow net running inside a *Net Executor*

different parts of hardware may require set-points at different levels, each device is controlled by its own real-time driver component. Usually, device drivers are also running cyclically with a device-specific frequency and issue set-points to the underlying hardware at given points in time, e.g., using a fieldbus technology such as EtherCAT or Ethernet. To ensure that computational tasks can have exclusive access to devices (e.g., actuators), special locking or reservation mechanisms must be taken into consideration (cf. Sect. V).

Similar to computational components, device components also have different states depending on the connection to the underlying hardware. If no connection exists, the component is *Offline* where it is neither possible to retrieve current sensor values (e.g., joint angles) nor to issue new set-points. Some device drivers offer a state called *Safe-Operational* in which it is possible to read all sensor values, but no control is available. Finally, all device drivers must support an *Operational* state in which actual hardware control is possible.

To improve the reusability of device drivers, specific *device interfaces* have been created which abstract from the concrete underlying hardware. For example, there is a generic device interface for robot joints which can be used in data-flow nets for joint-level control. The same actuator primitive can be used for any type of joint which supports this specific interface, such as robot joints, linear units or turn-and-tilt tables. Using configuration mechanisms (cf. Sect. III-B), it is possible to adjust computational tasks to hardware specifics without the need to change the overall structure of a data-flow net.

III. Communication & Configuration

This section will give an overview of the different communication means that are available (cf. Sect. III-A). Moreover, different possibilities for configuration are shown in Sect. III-B.

A. Communication

Communication is an important aspect in several areas of the runtime environment as shown in Fig. 4. To facilitate communication, these aspects are always handled in a generic way, i.e., the components and, thus, data-flow nets can rely on provided mechanisms for their communication requirements.

1) Intra-net communication: Communication within a single data-flow net is performed using links as described in Sect. II-A. Data exchange is performed during the execution and is strictly synchronous. Data provided in an execution cycle is available for the consumer within the same cycle.

2) Inter-net communication: Communication between different data-flow nets is performed using special inter-net comm primitives (cf. Fig. 4). Similarly, communication with device drivers (i.e., sensor and actuators) is realized. Nets providing data can write new values in each execution cycle to a sink primitive, which stores this value during the third phase of the execution cycle. Receiving data-flow nets use source primitives which read new values during the first phase. Depending on the frequencies of both data-flow nets, different latencies occur. The worst-case latency however is bounded, since all data-flow nets are executed with a guaranteed real-time cycle time.

3) Communication with further components or applications: Both intra-net as well as inter-net communication is performed with bounded latencies. Further components or applications however may not be running with real-time guarantees (e.g., ROS nodes). Although no timing guarantees can be given, communication with them is nevertheless required. For example, status information during execution of a data-flow net can be provided to an application which allows, e.g., to display useful information to the user such as a progress indicator. Furthermore, an application can influence a running data-flow net, e.g., by setting a new global velocity override for testing. However, data-flow nets must always be designed to function safely, even if no external data is arriving. Communication between a data-flow net and further components or applications is performed using the similar primitives as for inter-net communication (cf. Fig. 4).

4) Communication with devices: As the communication with sensors and actuators is highly device-specific, there are no built-in mechanisms. It is rather the responsibility of a real-time device driver to establish a stable and efficient proprietary connection with the hardware as shown in Fig. 4. However, to facilitate the implementation, there are several low-level driver (e.g., for Ethernet or CAN) and protocol stacks (e.g., for EtherCAT or CANopen) which can be reused [19]. If multiple devices access a common communication medium, the low-level driver is responsible for synchronizing access properly.

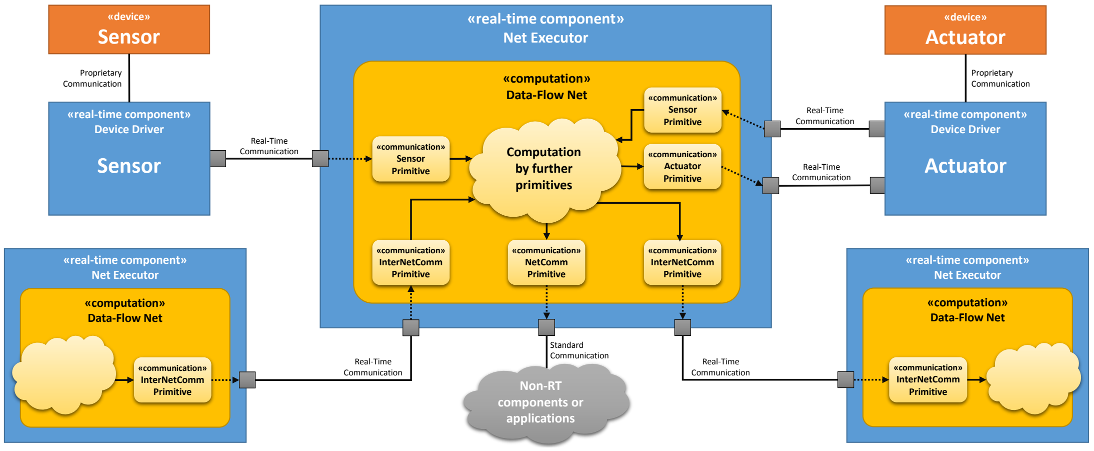

«device»
Sensor
Proprietary Communication
«real-time component» Device Driver
Sensor
Real-Time Communication
«real-time component» Net Executor
«computation» Data-Flow Net
«communication» Sensor Primitive
Computation by further primitives
«communication» Sensor Primitive
«communication» Actuator Primitive
«communication» InterNetComm Primitive
«communication» NetComm Primitive
«communication» InterNetComm Primitive
Real-Time Communication
Real-Time Communication
«device»
Actuator
Proprietary Communication
«real-time component» Device Driver
Actuator
«real-time component» Net Executor
«computation» Data-Flow Net
«communication» InterNetComm Primitive
Real-Time Communication
Standard Communication
Non-RT components or applications
Real-Time Communication
«real-time component» Net Executor
«computation» Data-Flow Net
«communication» InterNetComm Primitive

Figure 4. The runtime environment, i.e., the *Robotics Control Core*, handles different ways of communication inside, between and beyond data-flow nets with different timing guarantees. For this purpose, a *Net Executor* component offers different ports to device drivers, other *Net Executors*, or further components. Special or proprietary communication with devices (e.g., using EtherCAT or CAN) has to be handled separately by device drivers.

B. *Configuration*

Device drivers are required to support configuration in order to specify a unique device identifier or the parameters of its proprietary communication. Although drivers usually are device-specific, several device models can be controlled using the same mechanism, e.g., different sized robot arms by the same manufacturer. To allow for such differences, device drivers also have parameters which describe the hardware in details, such as Denavit-Hartenberg parameters for an industrial robot.

Furthermore, configuration is required to adjust the computation components to their specific purpose. By changing the parameters of a data-flow net, computational tasks can be adjusted to a specific device which includes, e.g., different acceleration or jerk limits. Moreover, by adapting the primitives or changing its structure, a data-flow net can be rearranged to reflect a (slighly) different computation. If necessary, even the frequency and priority of the *Net Executor* can be adjusted. Beyond this, the composition of data-flow nets and, thus, real-time components is highly dynamic. As Sect. IV will show, it can easily be changed if required. Hence, a flexible configuration of running real-time components is possible, too.

IV. Composition & Coordination

The composition of real-time components and their interaction which needs to be coordinated defines the overall behavior of the robotics system. The proposed approach allows for a flexible composition of changing computation tasks (cf. Sect. IV-A) and their declarative coordination using synchronization rules (cf. Sect. IV-B). The re-orchestration of real-time components is based on and triggered by a set of rules. These rules, however, are specified and activated by some sort of high-level application (cf. [11]) or by further work-flow supervisory components (cf. [20]).

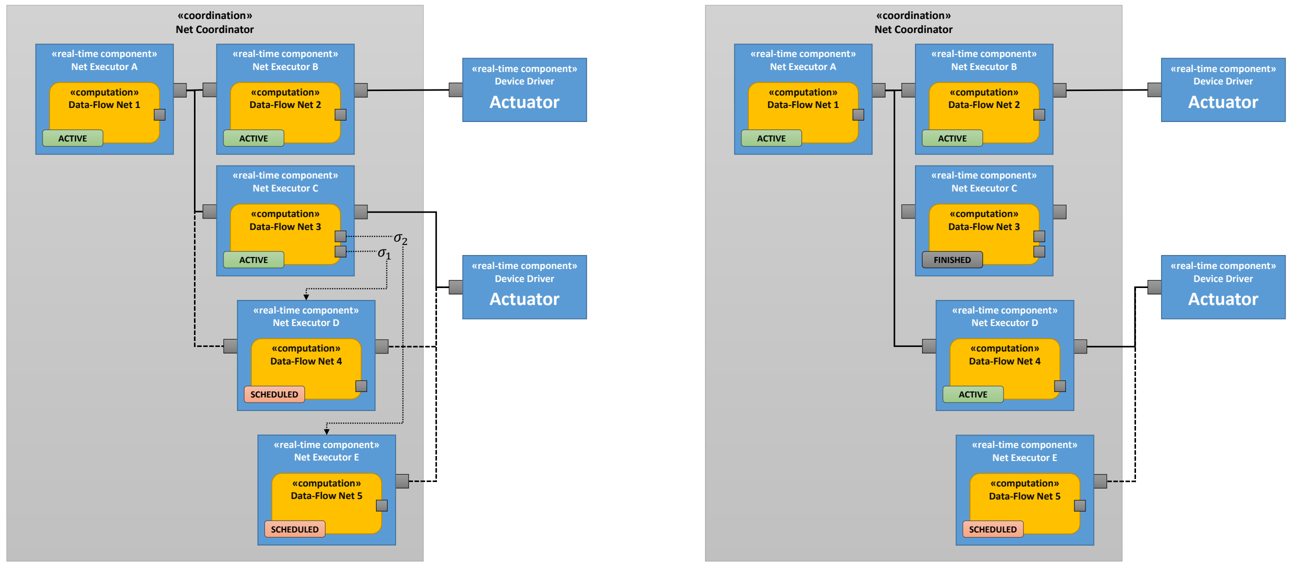
A. *Composition*

The composition is primarily given by operational device drivers and by the data-flow nets currently executed in *Net Executors*. The composition which is defined by device drivers is usually stable, i. e., it will only slightly change over time. However, the computational tasks are somehow volatile as they can be started, scheduled or stopped quite regularly. Hence, the composition is subject to change which allows to implement an evolving real-time component system. From our point of view, this is important because the tasks and/or the environment conditions for a robot are permanently changing (e.g., in service robotics or in Industry 4.0 scenarios). The underlying component system or its composition must be able to adapt in order to react properly.

Using a data-flow language for defining computation tasks and the possibility to execute them with timing guarantees is an important aspect for composition. However, composition is also defined by the connections established between components for communication. To facilitate this, special primitives (e.g., for inter-net data exchange or device access) are introduced which allow for real-time communication with bounded latencies (cf. Sect. III-A). Hence, the set of data-flow nets and device drivers in combination with communication channels in between defines the composition at a very instant of time. This composition can change immediately by defining new data-flow nets and implicitly their connections. However, the overall behavior of the system is only influenced when the interaction between components is changed by activating or stopping the appropriate computation tasks.

B. *Coordination*

A computational task should be designed for one certain basic action, e.g., one motion in joint or Cartesian space.



(a) Composition with two scheduled computation tasks and their according synchronization rules σ_1 and σ_2.

(b) Composition after synchronisation rule σ_1 was successfully triggered and, thus, synchronisation rule σ_2 was discarded.

Figure 5. By specifying computation tasks (with data-flow nets) and scheduling them on the runtime environment, the structure of real-time components can be determined. The *Net Coordinator* is able to change the current composition of real-time components immediately by activating and deactivating components as well as updating communication structures. The coordination is specified using synchronization rules between data-flow nets only.

For real-world applications, it is necessary to switch from such basic actions to other basic actions to coordinate the overall functionality. From a component-based point of view, it is necessary to re-orchestrate the components in order to obtain a system. If all computational tasks are completely self-contained, re-orchestration can happen *offline*, i.e., there are no timing requirements for changing from one task to another. In the robotics domain however, there are many cases where hard timing constraints exist for switching between two actions. If a computational task terminates and no other task immediately takes up control, all actuators must be in a safe and stable state, which usually means they must not be moving and must not apply force to the environment. In some applications, it is not desirable to always bring the actuators to standstill for switching basic actions. For example, multiple independent motions are often blended into each other to save time. In this case, the following motion task must take over the previous motion while the robot moves with considerable velocity.

The runtime environment implements *synchronization rules* to allow switching from one set of computation tasks to other sets of computation tasks with guaranteed latencies. A synchronization rule is defined as 4-tuple

$$\sigma = (C,\ \omega,\ \phi,\ \alpha)$$

where C denotes the *synchronization condition*, a propositional logic formula with Boolean variables defined in data-flow nets (using Boolean state ports). If C evaluates to *true*, all data-flow nets contained in ω are immediately terminated and all nets in α are started. Furthermore, the nets contained in ϕ are requested to terminate gracefully (i.e., to cancel). A locking mechanism is used to ensure that once a synchronization condition evaluated to *true*, no data-flow net contained in the condition C may enter execution phase 3 (cf. Sect. II-B). Since variables contained in the condition may only be changed in that phase, a consistent snapshot of the system is ensured.

A separate real-time component, the *Net Coordinator* (cf. Fig. 5), is used to monitor the current value of all synchronization conditions available in the system. If a condition becomes true, it is also responsible for taking all actions such as terminating running tasks and starting new ones. Once a new synchronization rule is created, all Boolean variables used in the synchronization condition are looked up (without real-time requirements) and pointers to the variables are stored in a data structure within the *Net Coordinator*. This allows for a fast evaluation of the current value of the condition each time a variable could have changed. Since these Boolean variables are only changed during phase 3 of the life-cycle, it is sufficient to evaluate a synchronization condition every time one of the nets contained in the condition finishes with phase 3.

Switching between two computation tasks always happens between two execution cycles. If a synchronization rule is triggered, a running computation task which needs to be affected can be in several states:

1) Not currently performing any work, i.e. having finished all work (including phase 3) for the current execution cycle. In this case, the computation task will

not be executed again. If applicable, a computation task from α will be executed in the next execution cycle, offering a seamless transition.

2) Currently in phase 3. The computation task will be allowed to complete phase three, afterwards the computation task is dealt with as in case 1). It should be noted that the affected computation task cannot be part of the synchronization condition C that triggered the re-orchestration. Due to the locking mechanism, either the computation task will have been prevented from entering phase 3, or the evaluation of the condition C would have been delayed until phase 3 has been completed.
3) Currently execution phase 1 or phase 2. In this case, the execution of the computation task can be interrupted immediately. Since phase 3 has not yet been started, work done so far in this execution cycle cannot yet have had any impact on the system. If a computation task from α has to take over, this must be done within the same execution cycle and not be delayed to the next one.

To allow for a hard real-time transition in case 3), the combined WCET of both computation tasks must be smaller that the applicable cycle time (plus some overhead time for switching execution). In cases 1) and 2), no such restriction exists.

Special care must be taken for actuators which are controlled from multiple computational tasks. Most actuators only allow being controlled by exactly one task. Switching from one set of tasks to a new set of tasks therefore must only occur if all devices required by the new tasks are either free or controlled by the nets contained in ω. If this cannot be guaranteed (e.g., because one device is still controlled by another task), re-orchestration is not performed. In this case, all data-flow nets contained in ω will continue running and are responsible for maintaining a safe situation (e.g., gracefully braking a robot or keeping a defined contact force).

Fig. 5 shows two subsequent states of the runtime environment during re-orchestration of computational tasks using synchronization rules. In Fig. 5a, the *base state* is shown where Data-Flow Net 1, 2 and 3 are currently running and controlling two actuators. Data-flow Net 4 and 5 are scheduled and can be triggered by either synchronization rule σ_1 or σ_2. Both data-flow nets require the same actuator and thus are mutually exclusive. For some Boolean state ports x and y in Data-Flow Net 3, the synchronization rules are as follow:

$$\begin{aligned}\sigma_1 &= (\text{net3}.x, \{\text{net3}\}, \emptyset, \{\text{net4}\})\\ \sigma_2 &= (\text{net3}.y, \{\text{net3}\}, \emptyset, \{\text{net5}\})\end{aligned}$$

In Fig. 5b, synchronization rule σ_1 has been triggered. As a result, Data-Flow Net 3 has been terminated while Data-Flow Net 4 has been started. Synchronization rule σ_2 has been discarded, since its condition did not hold during the last evaluation and Data-Flow Net 3 has been terminated preventing further changes to its Boolean state port y. Data-Flow Net 5 is still scheduled as further synchronization rules can be created using it.

V. Implementation Notes

The *Robot Control Core* has been implemented using C++ and is running on the Linux operating system with Xenomai real-time extensions. Each *Net Executor* is implemented as a separate real-time thread and executes one computational task at a time. Creating multiple independent threads allows multiple computational tasks to employ modern multi-core systems and thus to increase the overall system performance. When a new computational task is created, a new thread is only started if there is not already a thread which can be reused for execution. In [19], an algorithm is introduced which handles the allocation of threads and tries to minimize the number of idle threads running by maximizing reuse.

The *Net Executor* which will be used for execution is assigned during the creation of a data-flow net. To determine whether a new executor is required or an existing one can be reused, a table containing all currently existing *Net Executors* is maintained. This table contains a list of hardware resources which are exclusively assigned to a certain executor. During the creation of a data-flow net, the table is searched for an executor where the intersection of resources assigned to the executor and required by the data-flow net is not empty. If such an executor is found, it can be safely assumed that this executor is available for execution, since at least one mutually exclusive resource is shared with all other nets assigned to the executor. To assign the data-flow net, the newly created intersection of resources must be stored in the table. If no viable executor is found, a new *Net Executor* is created and the list of required resources for the new net is stored in the table. If a data-flow net terminates, the resource list for its executor is recalculated by creating the intersection of required resources for all remaining nets previously assigned to the executor. If no further data-flow net is assigned, the *Net Executor* is terminated. Data-flow nets which do not require any resources (e.g., monitoring tasks) cannot be handled using the described algorithm, since nothing can be said about the concurrency of those tasks. Hence, a *New Executor* is always started.

By assigning threads based on the required resources, switching from one task to another task controlling the same hardware most of the time reuses the same thread, thus guaranteeing instantaneous switching without any delay for the next execution cycle. If two independently controlled hardware devices, e.g., are joined into a single common task, one thread will not be reused. If both execution threads were running with a slight delay, at most one cycle time delay may occur during the switching process. This delay is usually handled by the underlying device drivers (e.g., by interpolating).

VI. Examples

A prototypical implementation of the component based robotics framework, including the synchronization mechanism, has already been implemented and first test are very promising. Unfortunately, there are more tests and experiments needed to provide a high quality evaluation, including precise timing measurements for switching actions. All experiments have been done using a cycle time of 2 ms. Typical tasks such as point-to-point or linear motions (including a analytical inverse kinematics) can be executed on standard PC hardware in less than 0.2 ms. Therefore, even case 3) for switching of tasks (cf. Sect. IV-B) can be achieved.

The following examples show some of the capabilities of the component-based robotics framework. All real-time switching actions in these examples have been done using a preliminary variant of the synchronization concept as it was introduced in [12]. This variant provides less flexibility (one computational task can take over work from exactly one other task), but also requires higher computational effort for each single task.

A. Example: Synchronized Robot Motions

To demonstrate the ability of re-orchestration while robots are fast moving, we have created the so-called *spaghetti challenge*. In this application, two industrial robots (a KUKA KR-16 and a Stäubli TX90-L) in a common workspace are geometrically linked by creating a common base coordinate system. Both robots are positioned in a way that a raw spaghetti can be fixed between both flanges. The application then creates a series of linear motions for one of the robots, while the other is instructed to maintain a fixed position relative to the other robot's flange. Each linear motion consists of one computational task which calculates the trajectories of both robots. For a continuous application, all linear motions are prematurely terminated by the *Net Coordinator* using synchronization rules and replaced by the following computational task which takes over the (fast moving) robots, calculates a blending path and continues with the next motion.[1]

Because two robots of different manufacturers with very different control protocols have been used and no documentation about exact latencies within the robot controllers is available, the synchronization can only achieve a certain quality. However, the quality achieved after some experiments for fine-tuning of the system is sufficient to carry the spaghetti without either losing or crushing it. Using an optical tracking system consisting of four Vicon MX-40s cameras and IR-reflecting markers on the robot, a maximum deviation for the distance of both flanges in the range of ± 1 mm has been measured (with velocities up to $2\,\mathrm{m\,s^{-1}}$).

[1]The video attachment is available at: http://video.isse.de/spaghetti/

B. Example: Sensor Processing

We have been using the proposed re-orchestration of real-time components for applications in human-robot-collaboration using capacitive sensors [21]. These sensors can be easily mounted on a robot arm and provide interesting features for safely detecting humans in a robot's workspace. However, the main issue with capacitive sensors is, that not only humans, but generally any conductive material influences the sensor. Hence, we are using an environment model containing information about static objects in the workspace which allows a reliable detection of humans. To store and query a previously recorded environment model, we are using the *Fast Library for Approximate Nearest Neighbors* (FLANN, cf. [22]). As the environment model needs to provide persistent data, FLANN was integrated into the RCC as *Device Driver*. At runtime, a data-flow net is specified which performs a nearest-neighbor search once every cycle to find the data-set closest to a given joint position. The corresponding data-set of sensor values is provided using inter-net communication (cf. Sect. III-A).

An additional data-flow net is specified for the distance estimation of every capacitive sensor, i.e., the current and the expected sensor value (from the environment model) are compared and based on this difference the distance to an obstacle is estimated [21]. Hence, this real-time computation component needs the data provided by the above mentioned environment model component. Finally, another data-flow net computes in every cycle an appropriate reaction of the robot, e.g., a global velocity override based on the lowest estimated distance. All three data-flow nets run as real-time computation components permanently and provide data using inter-net communication. Changing motion commands, which are also specified as data-flow nets and executed using rules (cf. [11]), can use these data sources, e.g., to decelerate in case of an obstacle. An advanced reaction strategy takes the current direction of every sensor into account when calculating the global velocity override. The computation whether a sensor is moving towards or away from a potential obstacle is specified by another data-flow net and executed as a real-time component, too.

VII. Conclusion

In this paper, we presented an approach and a runtime environment for re-orchestrating real-time component systems in robotics. It was described along each of the *5 Concerns* (5Cs) for robotic component systems [13]. The approach allows for a generic specification of computation tasks and incorporates a flexible, rule-based mechanism for composing and coordinating multiple components. As a consequence, it is possible to re-orchestrate the composition of running real-time components at run-time to adapt to new tasks – even when a robotics system is moving with high speed.

Distributing large applications while still being real-time safe is a key requirement for further increasing the scalability of the approach. Although several robots can be controlled by a single RCC, scalability is naturally limited by the available computing power. Hence, the real-time component system – in particular the RCC – will have to be distributed across multiple computers. The goal is to create a distributed *Net Coordinator* which allows to use synchronization rules for data-flow nets which are running on different systems, while still providing a similar level of real-time guarantees as it is possible for a single system at the moment.

References

[1] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Orebäck, "Towards component-based robotics," in *Proc. 2005 IEEE/RSJ Intl. Conf. on Intell. Robots and Systems, Edmonton*, 2005, pp. 163–168.

[2] D. Brugali, A. Brooks, A. Cowley, C. Cotè, A. C. Domínguez-Brito, D. Letourneau, F. Michaud, and C. Schlegel, "Trends in component-based robotics," in *Software Engineering for Experimental Robotics*, D. Brugali, Ed. Springer, 2007, pp. 135–142.

[3] D. Brugali and A. Shakhimardanov, "Component-based robotic engineering (Part II): System and models," *IEEE Robot. & Autom. Mag.*, vol. 20, no. 1, pp. 100–112, 2010.

[4] 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.

[5] A. Schierl, A. Hoffmann, A. Angerer, M. Vistein, and W. Reif, "Towards realtime robot reactions: Patterns for modular device driver interfaces," in *Workshop on Softw. Developm. & Integr. in Robotics. IEEE Intl. Conf. on Robot. & Autom., Karlsruhe, Germany*, 2013.

[6] H. Bruyninckx, "Open robot control software: the OROCOS project," in *Proc. 2001 IEEE Intl. Conf. on Robot. & Autom., Seoul, Korea*, 2001.

[7] T. Hammer and B. Bauml, "The highly performant and realtime deterministic communication layer of the aRDx software framework," in *Proc. 16th Intl. Conf. on Adv. Robotics, Montevideo, Uruguay*, 2013.

[8] C. Jang, S.-I. Lee, S.-W. Jung, B. Song, R. Kim, S. Kim, and C.-H. Lee, "OPRoS: A new component-based robot software platform," *ETRI Journal*, vol. 32, no. 5, pp. 646–656, 2010.

[9] N. Ando, T. Suehiro, and T. Kotoku, "A software platform for component based RT-system development: OpenRTM-Aist," in *Proc. Simulation, Modeling, and Programming for Autonomous Robots*, ser. LNCS. Springer, 2008, vol. 5325, pp. 87–98.

[10] 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, MO, USA*, 2009, pp. 2108–2113.

[11] A. Angerer, A. Hoffmann, A. Schierl, M. Vistein, and W. Reif, "Robotics API: Object-oriented software development for industrial robots," *J. of Software Engineering for Robotics*, vol. 4, no. 1, pp. 1–22, 2013.

[12] M. Vistein, A. Angerer, A. Hoffmann, A. Schierl, and W. Reif, "Flexible and continuous execution of real-time critical robotic tasks," *Intl. J. Mechatronics & Autom.*, vol. 4, no. 1, 2014.

[13] E. Prassler, H. Bruyninckx, K. Nilsson, and A. Shakhimardanov, "The use of reuse for designing and manufacturing robots," RoSta Consortium, White Paper, 2009.

[14] 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 *Proc. 28th Ann. ACM Sym. on Applied Comp.*, 2013, pp. 1758–1764.

[15] D. Vanthienen, M. Klotzbuecher, and H. Bruyninckx, "The 5C-based architectural Composition Pattern: lessons learned from re-developing the iTaSC framework for constraint-based robot programming," *J. of Software Engineering for Robotics*, vol. 5, no. 1, pp. 17–35, 2014.

[16] M. Radestock and S. Eisenbach, "Coordination in evolving systems," in *Trends in Distributed Systems CORBA and Beyond*, ser. LNCS. Springer, 1996, vol. 1161, pp. 162–176.

[17] M. Vistein, A. Angerer, A. Hoffmann, A. Schierl, and W. Reif, "Interfacing industrial robots using realtime primitives," in *Proc. IEEE Intl. Conf. on Autom. and Logistics, Hong Kong*, 2010.

[18] 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, Sep. 1991.

[19] M. Vistein, "Embedding real-time critical robotics applications in an object-oriented language," Ph.D. dissertation, Univ. of Augsburg, 2015.

[20] 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 *Proc. 41st Intl. Symp. on Robotics, Munich*, 2014.

[21] A. Hoffmann, A. Poeppel, A. Schierl, and W. Reif, "Environment-aware proximity detection with capacitive sensors for human-robot-interaction," in *Proc. 2016 IEEE/RSJ Intl. Conf. on Intell. Robots and Systems, Daejeon, Korea*, 2016, pp. 145–150.

[22] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *Intl. Conf. Computer Vision Theory and Application*. INSTICC Press, 2009, pp. 331–340.