



# Online Reconfiguration of Distributed Robot Control Systems for Modular Robot Behavior Implementation

Malte Wirkus<sup>1</sup> · Sascha Arnold<sup>1</sup> · Elmar Berghöfer<sup>1</sup>

Received: 20 December 2019 / Accepted: 7 July 2020  
© The Author(s) 2020

## Abstract

The use of autonomous robots in areas that require executing a broad range of different tasks is currently hampered by the high complexity of the software that adapts the robot controller to different situations the robot would face. Current robot software frameworks facilitate implementing controllers for individual tasks with some variability, however, their possibilities for adapting the controllers at runtime are very limited and don't scale with the requirements of a highly versatile autonomous robot. With the software presented in this paper, the behavior of robots is implemented modularly by composing individual controllers, between which it is possible to switch freely at runtime, since the required transitions are calculated automatically. Thereby the software developer is relieved of the task to manually implement and maintain the transitions between different operational modes of the robot, what largely reduces software complexity for larger amounts of different robot behaviors. The software is realized by a model-based development approach. We will present the metamodels enabling the modeling of the controllers as well as the runtime architecture for the management of the controllers on distributed computation hardware. Furthermore, this paper introduces an algorithm that calculates the transitions between two controllers. A series of technical experiments verifies the choice of the underlying middleware and the performance of online controller reconfiguration. A further experiment demonstrates the applicability of the approach to real robotics applications.

**Keywords** Robot programming · Robot control architectures · Robot autonomy · Model-based development · Model-driven engineering · Robot control

## 1 Introduction

Autonomous robotic systems, that are versatile in their application areas, will at some point face the problem that a change in control policy is needed in order to account for a new situation or task. There can be numerous reasons for a required adaption of the controller. For example; a sensor-equipped robot might need to switch its data processing pipeline to make use of different sensing hardware to cope with changes in the environment. The robot also might be confronted with a task requiring interacting with specific objects, which can only be recognized with a certain sensor-processing algorithm. A further example would be that for a legged system, a change in the ground properties might

require exchanging the gait control subsystem. With an increasing number of possible controller configurations for a robot, also the software complexity of the coordination layer increases when the possible controller transitions have to be implemented manually. For a software developer it can be difficult to grasp all possible permutations of controller reconfiguration for robots that are truly versatile in their capabilities.

A usual approach to handle complexity in computer science is to modularize, i.e. to divide the software into smaller, manageable parts that are independent from each other. The robotics community largely adopted the idea to modularize software with the advent of component-based software development frameworks like the well-known Robot Operating System (ROS) [1], the Robot Construction Kit (Rock) [2], or Yet Another Robot Platform (YARP) [3].

The frameworks aim to achieve two main goals: On one hand, they simplify developing software components by providing tools or automatizing certain parts of the implementation, such as inter-process communication or

---

✉ Malte Wirkus  
malte.wirkus@dfki.de

<sup>1</sup> Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), Robert-Hooke-Str. 1, 28359 Bremen, Germany

data visualization. The frameworks provide a common structure and interfaces for software components, and thereby unburden the software developers with many design decisions, while enforcing a common software structure ensures technical compatibility between different components. The second main goal of the development frameworks is achieved by using a software-package management system that allows simple retrieval and installation of required components: The frameworks provide access to a large selection of working software components that can be used in different applications and robotic systems. The robot controller is eventually created by interconnecting the interfaces of the individual components and adapting component properties to the given hardware or environment.

The component-based software development frameworks contributed largely to the ability of research groups to prototype individual technology demonstrations - we see the robotics community growing and increasingly producing output of robots solving a multitude of individual tasks. For example, the *Video Friday*<sup>1</sup> blog-articles from the *IEEE Spectrum Automation blog*, nowadays often links to more than a dozen new videos weekly of different robots mastering different individual task. The amount of individual technology demonstrations is astonishing, but demonstrations of robots that can autonomously cope with a broad range of different situations are rarely shown. We believe that this evolution of the robotics community is influenced by the current robotics software development frameworks, which are good tools for implementing static robot controllers for individual technology demonstrations, but provide little support for developing flexible control solutions as needed by highly versatile robots. In ROS, for example, the robot application often consists of a static component network, usually described by a XML-file (*launch file*), which specifies the different components that are to be started and also assigns configuration values to the components [4]. Different behaviors are then triggered by altering some of the component properties at runtime or by injecting data into the control network. The single static control network must therefore take into account all the different behaviors of the robot, which is contrary to the idea of modularity. Of course, further launch file can represent additional component networks, but an online transition between these control networks is not possible, so the complete robot controller must be switched off if another network is needed.

As an improvement over the current state, we propose that the individual robot behaviors should be modeled independently of each other. During the development of

the individual controllers, no assumptions should have to be made about which sequences of controllers are allowed and which are not, or what the robot's range of functions will ultimately be. Furthermore, no rigid architecture should prescribe certain approaches such as plan-based or reactive control, which would then have to be used to implement all the different behaviors.

We therefore present a software for modular modeling of robotic controllers, as well as their execution and online switching. To allow calling the individual controllers any time and in any order, the calculation of the necessary operations for transitioning between them is accomplished automatically at runtime. To account for the high computational effort, especially of complex sensor processing routines, the controllers can be executed on distributed hardware.

This is an extension of the known component development frameworks. The pool of available software components provided by the frameworks is used to model the controllers, but a runtime management layer is added for coordinating their execution. The proposed workflow and software simplify the development of the robot software to such an extent that a robot system can cover any number of different tasks without the need for additional development effort to implement transitions between tasks or to consider any dependencies between them. Although ROS provides many more available software components, we base our implementation on Rock for the reasons explained in Section 2.

After giving an overview about the related work and the model-driven software development methodology, we present the metamodels for controllers, individual software components and controller transitions in Section 2. The software architecture that allows to coordinate the distributed control networks at runtime and an algorithm that computes the transition between two controllers is explained afterwards in Section 3. To validate the proposed approach, a series of experiments was conducted, which are presented and discussed in Section 4. The last section concludes the work and discusses possible future work.

## 1.1 Related Work

The design of modern robotic software development frameworks was already envisioned back in the mid-nineties by Steward and Khosla [5]. As others (e.g. [6]) before they proposed to use software components that provide communication ports to let individual software parts exchange data. The crucial difference to previous work is that Steward and Khosla highlighted the possibility for component re-use, when general software components provide well-defined interfaces. In [5] they outlined how

<sup>1</sup>Video Friday in IEEE Spectrum Automation blog: <https://spectrum.ieee.org/tag/videoFriday> (last checked September, 2019)

research groups could share software components with standardized interfaces using the internet, and assemble robot controllers from these components without the need of programming them manually. Nowadays, this idea has become a reality with the modern robot development frameworks mentioned in the previous section. This has significantly changed the general technical situation compared to the mid-1990s. Nevertheless, another central point is missing in today's robotics frameworks, which has already been described in [5], so we will transfer it in our work to the today's situation: In [5], the need for reconfiguring robot controllers to adapt to different situations was anticipated. The authors identified that for this purpose at a higher-level of control, full control over the software component's runtime state and interface-connections is required, e.g. to replace components and redirect the data flow accordingly. While ROS [1] components lack this level of control, components for the Orocos Real-Time Toolkit (RTT) [7, 8] or the Rock framework [2] (which is based on RTT), allow for external coordination. The work presented in this paper is based on the possibility of coordinating components from outside. Therefore we use Rock as a basis for our implementation. In addition, Rock components provide an offline readable component interface specification, the *model* of the component. This means that without running it, the data connection interfaces (and other interfaces) of a component are readable for humans and software. This makes it easier to model controllers, since the interfaces can be presented to the developer as a reference e.g. with the help of graphical or text-based tools. However, these tools are out of scope of this paper.

A model-driven software engineering (MDE), or model-driven development (MDD) process [9] is based on abstract representations (*models*) of certain aspects of a given system or problem domain. As shown in Fig. 1, the models can be result of, or input to numerous different computational or manual processes. These processes generate, interpret or modify models, or transform them to a different representation that focuses a different concern of the system. Compatibility between individual processes is

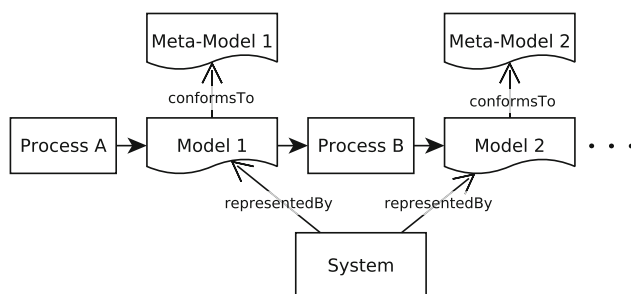


Fig. 1 Example model-driven development process

ensured by the use of *metamodels* that specify all relevant properties of the system and thus define the *language* for creating valid models. By focusing on the representation of the problem domain, MDD highlights a representation of knowledge about a domain rather than the algorithmic solution to a problem.

In the robotics community, e.g. the BRICS - Best practice in robotics [10] and RobMoSys<sup>2</sup> research initiatives promote the rigorous use of model-driven software development to cope with the complexity for system integration and the generation of individual algorithmic solutions.

In this spirit, with [11] and [12], software ecosystems were introduced, that provide a model-based workflow for developing robotic control software. Both works allow to a certain extent the online adaptation of the modeled controllers. In [12] however, this is limited to the variation of configuration values for software components without allowing a complete change of the controller topology. Both systems also have in common that they are designed to solve several different problems, such as real-time control system, high-level coordination and controller design at once. Thereby [12] offers clearer defined metamodels and interfaces between the individual modules, whereas [11] impedes the extensibility or integration of external tools by relying on a Ruby-embedded model representation and having a very tight integration between the individual software parts for model representation, control and modeling. This compromises one of MDD's potential strengths that, through the use of well-defined models representations, independent development tools can work together regardless of their technical realization.

A growing fraction of the robotics research community seems to recognize the advantages of MDD: Nordman et al. identified in a survey on domain specific modeling for robotics [13] an increasing number of publications regarding model-driven development for robotics in recent years. Of the publications included in the survey, the majority are in the field of modeling individual robot capabilities. With our work we provide a representation that could be used for such modeling tools for individual robot capabilities as the resulting common runtime representation.

Another earlier example for reconfigurable controllers is [14]. To better account for post-implementation changes of machine controllers, the authors propose to model the controllers using pre-installed software components that are loaded at system startup and interconnected according to the model. Their approach supports only offline reconfiguration, i.e. the controller needs to be shut down completely in order to load a new configuration. Nevertheless, they argue that modeling the controllers reduces development efforts when compared to traditional

<sup>2</sup>RobMoSys-website: <https://robmosys.eu/>

programming. Adaption of a robot controller at runtime is supported by the work presented in [15] and [16]. The focus in this work is the specification of constraints that define when a controller adaption has to take place rather than the representation of effect of the adaption. The reaction is thus limited to an adjustment of properties of the presently running components. Both examples have in common, that a single language specifies the conditions under which a change in behavior occurs, and also how the controller is reconfigured. In [17] Klotzbücher et. al argue that these two mechanisms: (a) Detection of the need of adaption and selecting the appropriate reaction (*Coordination*) and (b) applying the change on a particular system (*Configuration*), should be treated separately. They provide a language that allows specifying transitions of runtime states of components to enable the reconfiguration of the robot controller. However, no changes in the topology of the component network are possible, so that ultimately all controllers for all different robot behaviors must fit into a single rigid architecture.

We agree with the argumentation in [17] to provide separated models for coordination and configuration. Both processes represent system dynamics at different abstractions. Coordination is concerned with the high-level dynamics of a task. Its modeling primitives are semantic entities that are valid for a certain application domain, such as specific object or world conditions. The conditions are mapped to events and discrete representations of actions that manipulate the world state. These events and actions can be valid for different robots when they are acting in the same environment, even though the different robots might be very different in their technical implementation. Thus, implementation details should be kept at a minimum when describing high-level dynamics of a task. Configuration on the other hand, describes *how* to implement a certain action and thus is inherently technical. Its modeling primitives are

implementation details, e.g. software components and their interconnection or operations that are applied to them.

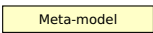
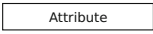
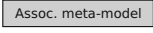

Following this terminology, this paper addresses the *Configuration*: In the next section, we introduce models for expressing dynamically changing control systems. The dynamics include adjusting component properties, runtime states, as well as changes of the overall controller topology. Afterwards, in Section 3, we propose a software architecture and implementation to carry out the modeled controller transitions on distributed hardware.

## 2 Modeling Online-Reconfigurable Distributed Software Systems

In MDD, metamodels define the modeling domain by specifying certain entities that exist in the domain and their relevant properties and relationships. This section introduces the metamodels that we use to model controllers and modify them.

A controller is described by a network consisting of interconnected pre-implemented software components. The components are computation units (named Tasks), that provide data ports for communicating with other Tasks. Task Networks can be manipulated by a discrete set of operations acting on the network's Tasks. A sequence of these operations specifies a procedural transition between two Task Networks. A declarative controller reconfiguration is realized by deriving the transition from the currently present controller to a given target configuration, where both controller configurations are represented as Task Networks. The following sub-section will introduce the meta-models for valid Tasks, Task Networks and Transitions in detail. Afterwards, Section 3 introduces an algorithm for the automatic derivation of a Transition, between two arbitrary Task Networks.

**Table 1** Legend for UML class diagrams that describe the metamodels

Classes		
	Meta-model	Yellow classes denote the metamodel of the system represented by the figure.
	Attribute	Meta-models comprise of different attributes. These attributes are denoted as white classes.
	Assoc. meta-model	An attribute can be associated to a model conforming to a different metamodel. The referred metamodel is visualized with a gray class.
	Assoc./Attribute	When an attribute is associated with an attribute of a different metamodel, this is represented by using the name pattern: "metamodel name/attribute".
Aggregations/Associations		
<i>conformsTo</i>		The attribute is a model that conform the associated metamodel.
<i>refersTo</i>		The attribute contains a reference to another model by storing a unique identifier of the referred model.
<i>instanceOf</i>		The attribute is an instance of the model that conforms to the associated metamodel.
For aggregations and associations where no multiplicity is given, the multiplicity 1 is assigned.		

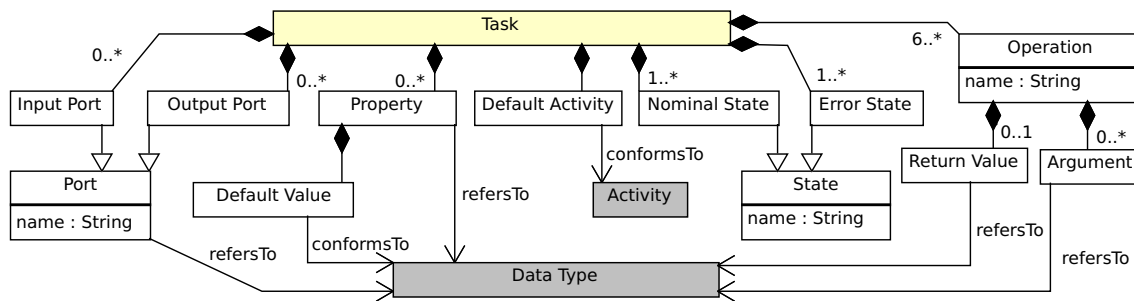


Fig. 2 Meta-model of a Task

We use UML class diagrams to represent the metamodels in this paper. Table 1 summarizes the semantics of the different class- and association-types we use for representing the metamodels.

### 2.1 Component-Based Controller Modeling

Tasks represent the atomic processing units of a Task Network. They are re-usable software components such as hardware drivers, control- or data processing algorithms, etc.. The Task-model captures information about the external interfaces of the software component, serving two purposes:

1. Provide technical interfaces to allow external software to control the software components.
2. Provide the knowledge about the presence of the interfaces without the need to execute the component, i.e., to present the interfaces to the user or software tools.

To represent software components and data types, that can be used on the component’s external interface, the models that are used in the Rock framework<sup>3</sup> are adopted. For completeness, the metamodel for Rock Tasks, Data Types and Deployments are briefly described here. For further details on the Rock framework and the underlying technology for developing real-time software components, Orocos-RTT<sup>4</sup>, the interested reader is invited to read [2] and [7].

As shown in the graphical representation of the metamodel *Tasks* in Fig. 2, Tasks consist of the following parts:

- *Port*: To interact with other software components, a Task can read incoming data from its *Input Ports* and write data to its *Output Ports*. Each port is associated with a *Data Type* (cf. Fig. 3) specifying the only valid type that can be exchanged over the port.

- *Operation*: Tasks can provide Operations, i.e. procedures, which can be called from remote processes and systems. With a name, a return value and arguments, the model an Operation provides similar information as a normal function header. At runtime, calling of an operation leads to the synchronous execution of an associated function with the same header.
- *Property*: Tasks can optionally define a set of typed configuration parameters. Each property is identified by a name that is unique within the component, a data type and a default value.
- *Nominal State/Error State*: Each Tasks execution is controlled with a state machine implementing a common life cycle. Each component contains certain pre-defined states and permissible state transitions that can be used to start or stop data processing, apply configuration values, and execute standardized fault recovery methods. The state transitions between the predefined states are triggered by calling specific operations.  
More details on the life cycle will be given in Section 3.2 and Table 4.
- *Activity*: Runtime characteristics (*Activity*, cf. Figure 5) of a *Task* such as it’s triggering mechanism (e.g. on retrieval of new data or periodic triggering) are relevant for the deployment of a *Task*. Nevertheless it is foreseen, that the component developer already pre-initializes them with a sane default value, such that a working default deployment can be generated.

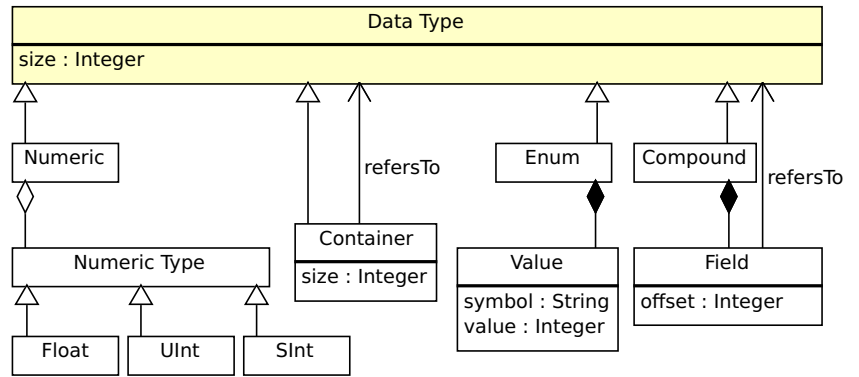
Properties and ports of a Task are associated with a *Data Type*. For modeling the data type, we use the *typelib*<sup>5</sup> metamodel that Orocos-RTT [8] and Rock [2] use. The model of a data type is used for framework features such as data serialization and de-serialization, conversion to CORBA IDL format or translating values from configuration files to the runtime data types and vice versa. For (de-) serialization, *typelib* assumes a rigid mapping of the individual type models to runtime types

<sup>3</sup>Rock-Website: <https://www.rock-robotics.org/>

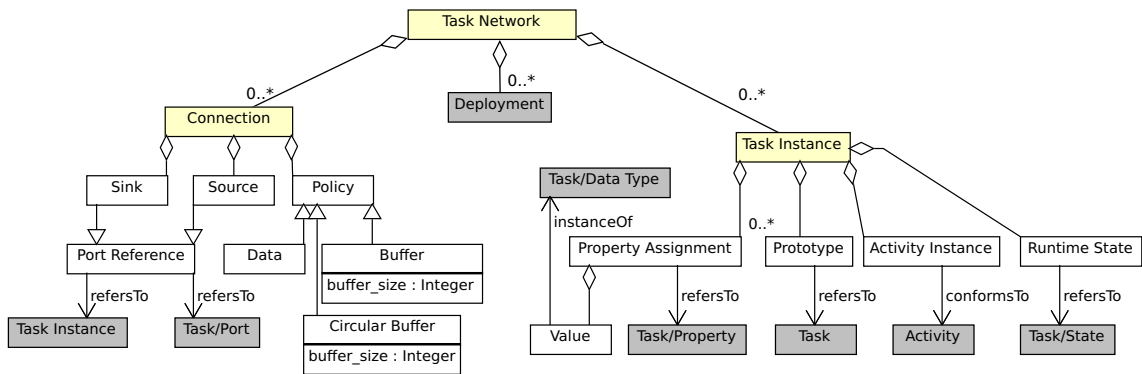
<sup>4</sup>Orocos-RTT Website: <http://www.oroocos.org/rtt/>

<sup>5</sup>Typelib Source Code and Documentation: <https://github.com/orocos-toolchain/typelib>

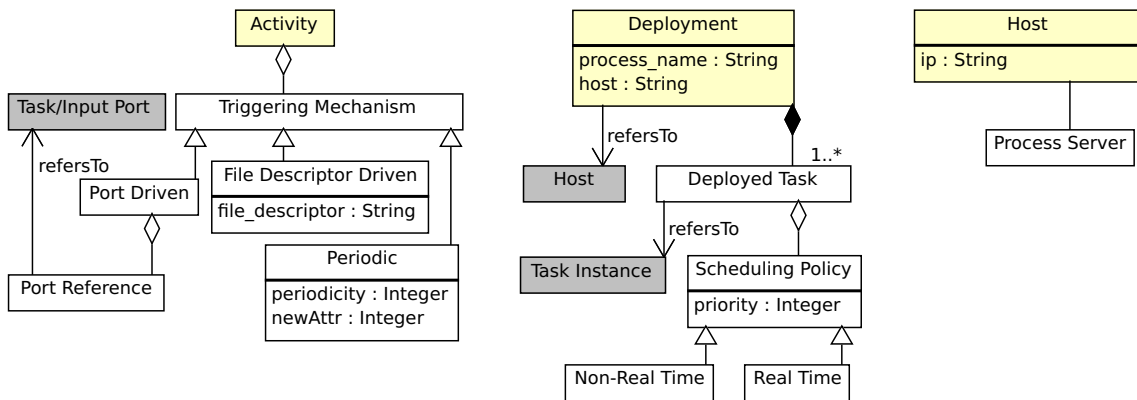
**Fig. 3** Meta-model of a Data Type



**Fig. 4** Meta-model of a Task Network



**Fig. 5** Meta-models for deployment information



in the target language (e.g. C++). The Data Type model describes four different categories of types:

- *Numeric*: This represents the primitive types. Three sub-categories of numeric types are discriminated: floating-point numbers, signed integers and unsigned integers.
- *Container*: Represent array-like structures. A container always refers to a Data Type indicating the type for every element stored inside the container.
- *Enum*: Relates the numeric value of an enumeration type (as in C/C++) with its symbolic representation.
- *Compound*: Represents complex data structures with different members (*Fields*) of different Data Types.

To model a controller for a robot, a *Task Network* is composed. To achieve this, instances of Tasks are created, configured, interconnected and deployed to target hardware. As illustrated in Fig. 4, for each instantiated Task (*Task Instance*) in the Task Network, the properties are explicitly configured with *Property Assignments*. The state in the life cycle of the Task is specified with the *Runtime State* attribute and certain runtime characteristics are assigned with the *Activity Instance* (cf. Figure 5). The *Prototype* specifies which model the instantiated component conforms to. Note that several instances of the same Task can exist, which is used, for example, when the same driver component is instantiated several times for redundant hardware devices. The individual instances are distinguished from each other by a unique name.

The connections between the ports of the individual Task Instances form the topology of the Task Network. In addition to the data source and sink, a *Connection* also describes the connection policy, which can be either unbuffered (*Data*), buffered or via a ring buffer.

The implementations of individual Tasks are compiled into libraries. With the *Deployment*, any number of Task

Instances are combined to form executable programs that link against the respective libraries containing the Task implementations. The Deployments are assigned to a specific execution hardware (*Host*) and an RTT-scheduler (real-time or non-real-time) with a specific scheduling priority. Each Host is associated with a *Process Server*, a program in charge of handling the processes on the particular execution hardware.

When two Task Instances of the same Deployment communicate with each other (intra-process communication), a more performant data transfer method can be used compared to communication between two different processes (inter-process) or between two different computers (remote). The Rock Framework automatically ensures that the most suitable communication method is used.

Implementation details on the Process Server and the overall runtime architecture are given in the Section 3.

### 2.2 Transition Modelling

To dynamically reconfigure the robot controller, the structure of the running Task Network is altered. Since all components conform the *Task* metamodel, a set of actions, that is common for all components, can be defined, to describe a complex restructuring of a component network when the actions are chained sequentially.

Figure 6 illustrates the metamodel for *Transitions*, which can contain any sequences of the following *Actions*:

- *Deploy/Undeploy*: The Deploy action represents the instruction to execute a specific Deployment, which corresponds to starting a new process. The Undeploy action, on the other hand, is the instruction to end the process of the referenced Deployment.
- *Call Operation*: Represents a call to an Operation of a Task Instance. A value must be assigned for each argument specified for the referred Operation.

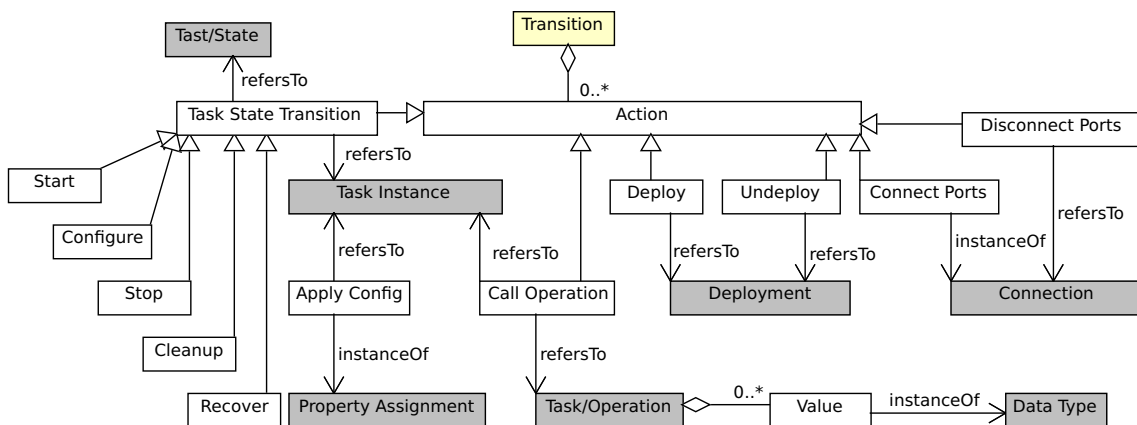


Fig. 6 Meta-model of a Transition

- *Task State Transition*: The different Task State Transitions refer the initiation of a runtime state transition of a Task Instance.
- *Connect/Disconnect Ports*: Refers to the action of establishing or removing a connection between two ports.

For the different metamodels presented in this section, we implemented runtime-representations in C++, Python and/or Ruby and defined text-based serializations in YAML or XML. The serialization formats are used for persistent storage and data exchange between the tools described in the next section. Examples for YAML representations for a Task Network and a Transition are given in Appendix A.

### 3 Implementation

In model-based development, metamodels, like the ones from the previous section, allow to model certain aspects of a specific problem, but not necessarily restrict the processing that is conducted on them. In contrary, as indicated by Fig. 1, there can exist independent tools that focus on different aspects of the modeled system and process the models in various ways. This section now introduces a software that is responsible for the execution and runtime management of the modeled controllers as well as for the automatic calculation of transitions between them. Thereby, the software makes use of all metamodels from the previous section with *Component Network* and *Transition* being the central representations in the implementation. Table 2 and later Table 3 formally describe some operations on the model representations that are used in the implementation of the software *rock-runtime*.

The following sub-section gives an overview about the overall system architecture, followed by a detailed description of the Network Operation Solver (NetOpSolver), an algorithm that generates a sequence of actions that reconfigure a controller network, in the subsequent subsection.

### 3.1 Distributed Runtime Architecture

The task of rock-runtime is to manipulate system processes, TaskContexts and data connections in such a way that the running control system is reconfigured in correspondence with Task Networks which can be requested at any time, e.g. by a high-level mission coordination or plan execution software. Figure 7 illustrates the architecture of rock-runtime comprising of the following software parts:

- Network Operation Solver (NetOpSolver): an algorithm to determine the Transition  $\phi_{min}$  consisting of the minimum set of operations that reconfigure the currently running controller  $C_{act}$  to match the desired controller  $C_{set}$  (both represented as Task Networks).
- RTTTaskManager: A software module handling all interactions with RTT components, represented by the Task Instance model. This module's task is to realize a particular action  $a \in \phi_{min}$  on the respective component.
- ProcessServer: Manages the processes on a specific runtime platform. On each computer involved in a Task Network, a ProcessServer runs which takes care of starting and stopping processes on this computer when the RTTTaskManager requests this by calling the corresponding operations.
- TaskNetworkHandler: The central software part, providing the interface to higher-level software. The TaskNetworkHandler coordinates the system-wide runtime state by invoking the other mentioned software parts. Requests of the a new  $C_{set}$  or requests for applying certain Transitions can be issued to the TaskNetworkHandler.

The ProcessServer and TaskNetworkHandler are implemented as individual Rock components (cf. Figure 7). The RTTTaskManager and NetOpSolver are provided as C++ classes in libraries that are linked to the TaskNetworkHandler component.

**Table 2** Definition of operations on Task Network and Transition models

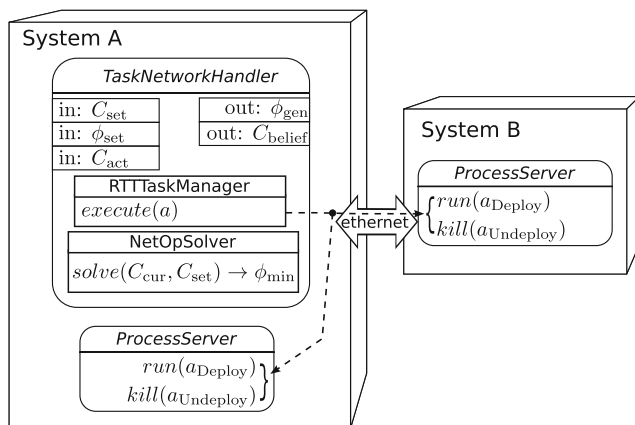
<b>Definition 1</b>	Each model conforming to any of the metamodels can uniquely be identified by an <i>id</i> .
<b>Definition 2</b>	Two Task Instances $x$ and $x^*$ are equal ( $x = x^*$ ), if all their attributes are equal. They are similar ( $c \approx c^*$ ), if at least their id and Prototype are equal. For two Connections $e, e^*$ , two Deployments $d, d^*$ , and Task Networks $C, C^*$ the equality operator $=$ is defined analogically.
<b>Definition 3</b>	The operator $\oplus$ applies a <i>Transition</i> $\phi$ to a <i>Task Network</i> . The operation $C \oplus \phi \mapsto C^*$ predicts the effect of a transition. The result is a <i>Task Network</i> $C^*$ , where $C^* \neq C$ , if $\phi$ is not empty ( $\phi \neq \emptyset$ ) and <i>minimal</i> .
<b>Definition 4</b>	The operator $ \phi $ can be applied to a <i>Transition</i> $\phi$ and returns the number of Actions within the <i>Transition</i> . A <i>Transition</i> $\phi_{min}$ is <i>minimal</i> , if there exists no other transition $\phi$ with $ \phi  <  \phi_{min} $ that has the same effect, when applied to the same <i>Task Network</i> $C$ .



**Table 3** Definition of set-operations on Task Networks

<b>Definition 5</b>	A Task Network can be seen as a set composed by its Task Instances $\mathbf{x}$ , Connections $\mathbf{e}$ and Deployments $\mathbf{d}$ .
<b>Definition 6</b>	The existence $\in$ of a Connection $e$ or a Deployment $d$ in a Task Network $C$ is given, if $C$ contains a Connection $e_A = e$ or Deployment $d_A = d$ . The existence of a Task Instance $x$ in a Task Network $C_A$ ( $x \in C$ ) is given, if $C$ includes a Task Instance $x_A \approx x$ .
<b>Definition 7</b>	The set difference of two Task Networks $C_A$ and $C_B$ (or complement of $C_A$ with respect to $C_B$ ) is defined by $C_A \setminus C_B = \{x   (x \in C_A) \wedge (x \notin C_B)\}$ .
<b>Definition 8</b>	The intersection of two Task Networks $C_A$ and $C_B$ are those components $x_a \in C_A$ and $x_b \in C_B$ , that are similar ( $x_a \approx x_b$ ) and present in both Task Networks.

Each computer system involved in the distributed control system runs a ProcessServer that takes care of executing or terminating the processes specified by the Deploy and Undeploy actions. The Task Manager and the Process Servers on the individual runtime systems communicate with each other using RTT operation calls. All additional Actions (such as Start, Apply Config, Connect Ports, etc.) are implemented in the by RTTTaskManager directly by using the standardized Rock interfaces of the components referred by the actions. For interacting with external software, like a plan execution program, the TaskNetworkHandler provides input ports and RTT-operations, on which TaskNetworkHandler listens for Task Network and Transition requests. When a certain controller configuration is requested, the TaskNetworkHandler calls the NetOpSolver in order to generate the required Transition. The TaskNetworkHandler keeps tracks of all TaskNetwork- and Transition-requests and maintains a Task Network  $C_{belief}$  that represents the current state of the controller, which is also passed to the NetOpSolver ( $C_{cur} = C_{belief}$ ). The TaskNetworkHandler provides an input port that allows resetting  $C_{belief}$  by passing the actual running component network  $C_{act}$ .



**Fig. 7** Architecture of the rock-runtime software

### 3.2 Network Operation Solver

By using Definition 1 – 4, Eq. 1 describes the Network Operation Solver (NetOpSolver) as a function that finds the minimal set of Actions that will establish a desired target configuration  $C_{set}$  when they are executed.

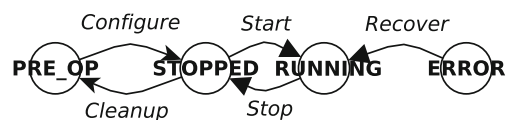
$$solve(C_{cur}, C_{set}) \mapsto \phi,$$

where  $C_{cur} \oplus \phi = C_{set}$ , and  $|\phi|$  is minimized. (1)

We implemented the NetOpSolver as a special purpose algorithm that is shown in detail in Algorithm 1. The algorithm identifies the differences between  $C_{des}$  and  $C_{cur}$ , and for each mismatching item selects a predefined partial Transition, that eliminates the mismatch. The following paragraphs give details of the NetOpSolver, that makes use of set-operation on Task Networks that are defined in Table 3.

Line 2-4 of Algorithm 1 identify topological differences between  $C_{cur}$  and  $C_{set}$ :

- $C_-$  contains the Task Instances, Connections and Deployments of  $C_{cur}$ , that need to be removed, because they are no longer exist in  $C_{des}$ .
- $C_+$  contains the sub-network of  $C_{des}$ , that newly needs to be created, because it was not present in  $C_{cur}$ .
- $C_{\cap}$  contains the sub-network that is present in  $C_{cur}$  and  $C_{des}$ , but where the components are not *equal* but *similar*. Here a state change in the individual components life-cycle or a reassignment of properties is required (cf. line 17-26).



**Fig. 8** Valid state transitions of the common life cycle for Task Instances

**Algorithm 1** Network Operation Solver (NetOpSolver).

---

**Input:** Initial network  $C_{cur}$  and the target component network  $C_{set}$

**Output:** Minimal Transition  $\phi$  where  $C_{cur} \oplus \phi = C_{set}$

*Initialize empty Transitions :*

- 1:  $\phi \leftarrow \emptyset$
- Determine required topological changes by creating complements and the intersection of  $C_{cur}$  and  $C_{set}$*
- 2:  $C_+ \leftarrow C_{set} \setminus C_{cur}$
- 3:  $C_- \leftarrow C_{cur} \setminus C_{set}$
- 4:  $C_{\cap} \leftarrow C_{cur} \cap C_{set}$

*Generate shutdown sequences for components in  $C_-$*

- 5: **for** each TaskInstance  $x \in C_-$  **do**
- 6:    $\phi \ll \text{genStateTransition}(x, \text{stateOf}(x), \text{PRE\_OP})$
- 7: **end for**
- Generate Disconnect and Undeploy actions*
- 8: **for** each Connection  $e \in C_-$  and Deployment  $d \in C_-$  **do**
- 9:    $\phi \ll \text{makeDisconnect}(e)$
- 10:    $\phi \ll \text{makeUndeploy}(d)$
- 11: **end for**

*Generate actions to add new Deployments and Connections specified in  $C_+$*

- 12: **for** each Connection  $e \in C_+$  and Deployment  $d \in C_+$  **do**
- 13:    $\phi \ll \text{makeConnect}(e)$
- 14:    $\phi \ll \text{makeDeploy}(d)$
- 15: **end for**
- Generate startup transitions for all Tasks in  $C_+$*
- 16: **for** each  $x \in C_+$  **do**
- 17:    $\phi \ll \text{genStateTransition}(x, \text{PRE\_OP}, \text{stateOf}(x))$
- 18: **end for**

*Handle potential state or configuration changes in  $C_{\cap}$*

- 19: **for** each TaskInstance  $x_{\cap} \in C_{\cap}$  **do**
- 20:    $x_{cur} \leftarrow \text{find } x \in C_{cur}, \text{ where } x \approx x_{\cap}$
- 21:    $x_{set} \leftarrow \text{find } x \in C_{set}, \text{ where } x \approx x_{\cap}$
- 22:    $\phi \ll \text{genStateTransition}(x_{\cap}, \text{stateOf}(x_{cur}), \text{stateOf}(x_{set}))$
- If property assignment is required, also generate the needed life cycle state transitions*
- 23:   **if**  $\text{propertyAssignmentOf}(x_{cur}) \neq \text{propertyAssignmentOf}(x_{set})$  **then**
- 24:      $\phi \ll \text{genStateTransition}(x_{\cap}, \text{stateOf}(x_{cur}), \text{PRE\_OP})$
- 25:      $\phi \ll \text{makeApplyConfig}(x_{\cap}, \text{propertyAssignmentOf}(x_{set}))$
- 26:      $\phi \ll \text{genStateTransition}(x_{\cap}, \text{PRE\_OP}, \text{stateOf}(x_{cur}))$
- 27:   **end if**
- 28: **end for**

*Sort individual actions in  $\phi$  according to their applicability: (1. Recover, 2. Stop, 3. Disconnect, 4. Cleanup, 5. Undeploy, 6. Deploy, 7. Apply Config, 8. Connect, 9. Start)*

- 29:  $\phi \leftarrow \text{sortByApplicability}(\phi)$
- 30: **return**  $\phi$

---

The functions  $\text{propertyAssignmentOf}(\text{TaskInstance})$  and  $\text{stateOf}(\text{TaskInstance})$  return the corresponding attribute (Property or State) of the TaskInstance given as param-

eter. The functions  $\text{makeApplyConfig}(\text{TaskInstance}, \text{PropertyAssignment})$ ,  $\text{makeConnect}(\text{Connection})$ ,  $\text{makeDeploy}(\text{Deployment})$  etc. create Actions for a Transition, that modify a Task Network such that the entity which is given as argument is either created or removed.

It was already mentioned in Section 2 that every running Task contains a state machine, implementing a uniform life cycle for all software components. Figure 8 illustrates the allowed transitions (italics) between the individual states (bold) within this life-cycle. Based thereon, Table 4 lists the sequences of actions required to transition between any given constellation of initial and target state of a Task. The ERROR state can only be triggered within the component and not by calling an operation. The function  $\text{genStateTransition}(\text{TaskInstance}, \text{State init}, \text{State target})$  generates the correct sequence of TaskStateTransition-actions to setup the state *target* on the given TaskInstance based on this state chart.

Lines 23-27 of Algorithm 1 account for the convention present in Rock, that the assignment of configuration parameters are applied to a component when it is in the PRE.OP-state. Realizing a reconfiguration of a running component thus requires generating a state transition to PRE.OP before applying a Apply Config-action and afterwards a full state transition to the RUNNING-state again.

Line 29 of the Network Transition Solver ensures the correct order of actions and removes duplicate state transitions. In general, actions that regress the lifecycle of a component or reduce the component network (components or connections are removed) are executed before actions that advance runtime states or extend the component network.

## 4 Experimental Results

The overhead caused by the framework for data exchange and for interaction with the software components influences the feasibility of using the software framework for real-time control tasks. To validate the choice of using Rock/Orocos-RTT, and the dynamic controller reconfiguration performance, various performance parameters of the system were measured in an experimental evaluation. A first experiment series acquires data about the middleware's performance in relation to data sample and component network sizes. A second series investigates times for starting or reconfiguration controllers with changing component network sizes.

These theoretical experiments are supplemented by another practical experiment which examines the performance under realistic conditions. In this experiment, different controllers created with the proposed framework are executed on a real robot system and the reconfiguration

**Table 4** Operations required to transition between runtime states

Start \ Target	PRE_OP	STOPPED	RUNNING	ERROR
PRE_OP	–	ApplyConfig, Configure	ApplyConfig, Configure, Start	x
STOPPED	Cleanup	–	Start	x
RUNNING	Stop, Cleanup	Stop	–	x
ERROR	Recover, Stop, Cleanup	Recover, Stop	Recover	–

times between them are determined. Thereby the applicability of the framework to real robotic problems as well as the performance of the adaptation for real controllers is shown.

The experiments are executed on two different computers that are connected with a commercial Gigabit Ethernet switch from D-Link:

- System A: Mini-ITX (mITX) board with Intel Core i7-3610QE CPU @ 2.30GHz, 16GiB memory and a SATA connected SSD from Samsung (550 MB/s seq. read, 520 MB/s seq. write).
- System B: COM Express (COMe) Compact Pin-out Type 6 embedded system with Intel Core i7-7600U CPU @ 2.80GHz, 16GiB memory and a SATA connected SSD from Samsung (550 MB/s seq. read, 520 MB/s seq. write).

On both systems the operation system Ubuntu 18.04 is installed with no significant adaptations compared to its standard configuration.

For all experiments we composed different Task Networks from the three Tasks shown in Fig. 9. The *MessageProducer* produces data samples  $\chi_{out}$  of the type *DataSample* (cf. Listing 1).

The size of the data sample depends on the length of the string stored in the *payload* variable. To create the data sample, *MessageProducer* generates a random string with a configurable length for the payload variable and stores a timestamp in microsecond resolution after constructing the *DataSample* object and immediately before writing it to the output port.

Component networks used in the experiments create a processing chain by connecting the output port of *MessageProducer* to a varying number of *MessageRelay* components that simply read samples from their input port and forward them to the output port. The last component in

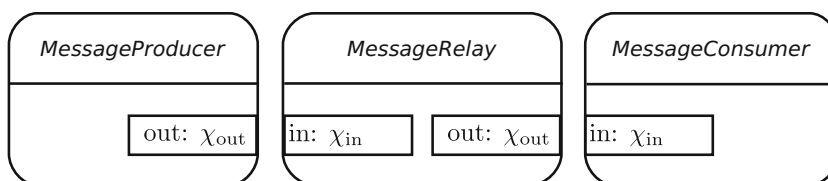
a processing chain is the *MessageConsumer* that calculates the duration between the creation of the data sample in the *MessageProducer* and its retrieval in the *MessageConsumer*, by comparing the timestamp stored in  $\chi_{in}$  to the current time. The transport duration for all received sample is written to a file.

The experiments also investigate the effect that different deployment configurations induce. We distinguish the following three cases:

1. Intra-process communication: Each two directly connected Tasks are running in the same process.
2. Inter-process communication: Each two directly connected Tasks are running in different processes, but the processes run on the same system.
3. Remote communication: Each two directly interconnected Tasks are running on different systems, thus, network communication is required to exchange data.

In the following, we will use a simplified notation to describe the topology of Task Networks. We refer to *MessageProducer*, *MessageRelay* and *MessageConsumer* components with the symbols *P*, *R* and *C*. The computer system (A, B) and the process, where the Task is deployed to, is indicated by a prefixed superscript. An arrow denotes a data connection between two components (since there is no component involved with multiple input or output ports this notation is for the present case unambiguous). For example,  $^{A_1}P$  refers to a Message Producer that is deployed to process 1 on system A. The processing chain  $^{A_1}P \rightarrow ^{B_1}C$  shows a case of remote communication where a data sample is sent from a *MessageProducer* running on system A to a *MessageConsumer* running on system B. In all experiments, the *MessageProducer* are triggered periodically and the other two components are triggered upon receiving a new data sample.

**Fig. 9** Software components used in evaluation experiments



```

class DataSample{
public:
    std::string payload;
    long long nSample;
    base::Time timestamp;
    size_t size(){
        return payload.size()+sizeof(nSample)
            +sizeof(timestamp);
    }
};

```

Listing 1 C++ type definitions for data samples  $\chi$

### 4.1 Data Transfer Times

In the first experiment (E1.1), we investigate how the size of a data sample influences its transmissions times in different deployment cases. We measure the time for the two-way transmission of a data samples  $\chi$  between components. The experiment is repeated with changing sizes of  $\chi$  and different deployment cases. The following Eqs. 2–4 show the used component networks.

$${}^{A_1}P \rightarrow {}^{A_1}R \rightarrow {}^{A_1}C \tag{2}$$

$${}^{A_1}P \rightarrow {}^{A_2}R \rightarrow {}^{A_3}C \tag{3}$$

$${}^{A_1}P \rightarrow {}^{B_2}R \rightarrow {}^{A_1}C \tag{4}$$

The payload size of  $\chi$  varies between 10 B and 100 MB in multiple steps. Small sizes (< 1 kB) are included, as they are usually used for real-time control task such as joint control. Larger sizes that often find use in sensor processing (< 10 MB) are included as well, and with payload size of 100 MB we also test a data sample size that is so large, that it is of no practical use in most robotics control applications.

Figure 10 shows a plot with the mean data transfer times collected over a period of 15 seconds for the different versions of the experimental setup. As expected, the result of the experiment shows that the performance of intra-process communication is better than that of inter-process communication, and that remote communication causes the highest overhead. The question the experiment aims to answer is whether the communication overhead is sufficiently low to use the framework for robot control.

The data shows that for sample sizes up to multiple kB, the communication overhead is not the limiting factor to achieve control frequencies with 1 kHz in the intra-process case.

For the inter-process case the communication overhead is also still less than 1 ms, but for a 1 kHz controller there is not much time left for the actual calculation, which would be needed in a real control application. In the remote communication case, the overhead alone prevents a control frequency of 1 kHz. However, control frequencies of 500

Hz or 100 Hz can be achieved even with sample sizes larger than 100 kB respectively 1 MB in the intra-process case.

This however is a theoretical statement. In practice, it is likely that larger data sizes will require also more time for processing the data. For example, it usually takes longer to process a FullHD image than multiplying a joint position vector. Thus for larger sample sizes it is more likely that the actual processing is actually the limiting factor to achieve high control rates.

The second experiment (E1.2) investigates the impact of an increasing number of data processing components on the theoretically achievable control rate. We measure the time for sending a data sample of a fixed size (100 B) through component networks that vary in the number of MessageRelay components. The network topologies used for the experiment are given in Eqs. 5–7.

$${}^{A_1}P \rightarrow {}^{A_1}R_1 \rightarrow \dots {}^{A_1}R_n \rightarrow {}^{A_1}C \tag{5}$$

$${}^{A_1}P \rightarrow {}^{A_2}R_1 \rightarrow \dots {}^{A_{n+1}}R_n \rightarrow {}^{A_1}C \tag{6}$$

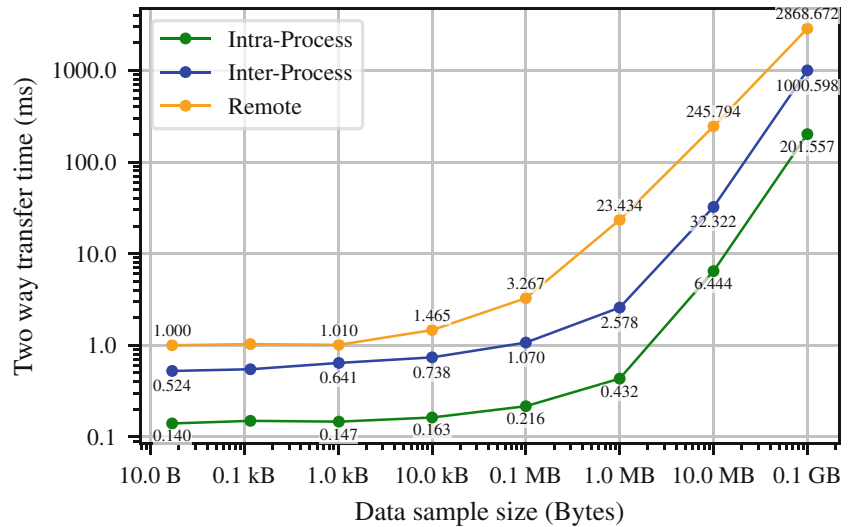
$${}^{A_1}P \rightarrow {}^{B_1}R_1 \rightarrow {}^{A_2}R_2 \rightarrow \dots {}^{B_k}R_n \rightarrow {}^{A_1}C \tag{7}$$

Similar to the first experiment, again the duration between the creation of  $\chi$  in  $P$  and its arrival in  $C$  is measured. The data collection is repeated for different number of Relay components ( $n = 5, 10, 25, 50, 100, 150, 200, 250$ ) and different deployment cases. With  $n = 5$  and  $n = 10$ , typical processing chain sizes are considered. Component networks composed from 25 or 50 individual components still appear as realistic sizes, but a chained interconnection of such a number of components is a rare case in robotics. In the real experiment, which will be presented in more detail in the next subsection, we show, for example, a controller that implements an autonomous exploration behavior on a rover utilizing a LiDAR sensor for self-localization and mapping (SLAM) of the environment (cf. Section 4.3.1 and Appendix B.1). The longest processing chain in that controller contains 13 components consisting of the sensor and actuator drivers (3), sensor processing of the point cloud data (2), slam and pose filtering (2), setpoint generation and trajectory planning (2), trajectory and motion control (2) sample dispatching and device I/O (2).

Even though the larger values of  $n$  seem not of practical use for robotics applications, we include these values to the experiment to investigate the potential for controllers with more fine-granular component decomposition or significantly more complex systems than those that are currently common.

As in E1.1, the data of E1.2 (shown in Fig. 11) reflects the expectation that the intra-process deployment case performs better compared to the other deployment cases. For each

**Fig. 10** Data transfer times of data samples with increasing sizes



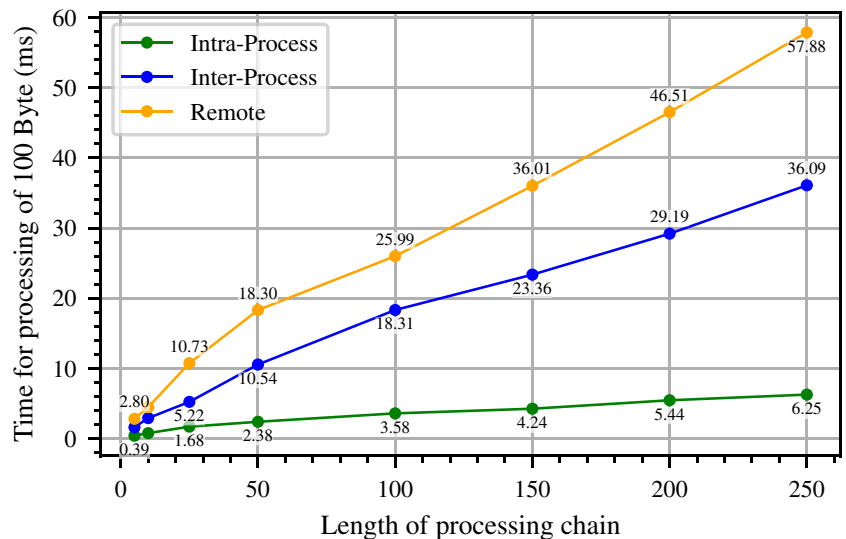
deployment case, the transfer time is increasing nearly-linearly with the number of components, but with different slopes of approx.  $0.02$ ,  $0.14$  and  $0.22 \frac{\text{ms}}{\text{transport}}$ .

The experiment shows that if processing chains are kept short ( $< 10$ ), a 1 kHz control frequency could be achieved in the intra-process case. For a control frequency of 100 Hz, the communication overhead play no significant role for any practically relevant processing chain in the intra-process case. In a real robotics application a deployment case like the inter-process or especially the remote case, where each single data connection crosses process or system boundaries, are quite unconventional. Nevertheless, the results show that the 100 Hz target can be achieved for longer processing chains, if the processing times inside the components remain moderate.

### 4.2 Task Network Operation Times

The second series of experiments examines the controller startup and reconfiguration times. The experiment E2.1 investigates how the number of interconnected components influences start-up/shutdown times of a Task Network. We again use networks from Eqs. 5 – 7 that were also used for E2.1. For the individual component networks, that vary in network sizes and deployment case, we measure the duration to setup the running component network, and to completely shut it down again. For a component network with  $n$  MessageRelay components,  $1 + n$  data connections have to be established or removed respectively. For starting a component network,  $(n + 2) * 3$  Task State Transitions and two property assignments have to be applied. To shut

**Fig. 11** Data transfer times with increasing length of the processing pipeline



it down again,  $(n + 2) * 2$  Task State Transitions are needed. The amount of Deploy/Undeploy actions varies with the deployment case: While only one deployment must be started (or terminated) for the intra-process case,  $n + 1$  Deploy/Undeploy actions are required for the other two cases.

For each component network, the measurement of startup/shutdown times was repeated 5 times. Figure 12 shows two plots with the mean execution times. Both plots show the same data but differ in the visual section of the horizontal axis, such that the plot on the left only shows the data for smaller networks. The data shows that shutting down component networks is generally faster than starting them, in part due to the less required Task state transitions. In addition, the performance varies dramatically for different deployment cases. While a controller consisting of 25 components grouped into a single deployment can be started in milliseconds, it takes more than 1s if all components are deployed locally into individual processes, or more than 1.5 s in the remote case.

These observations already indicate that the Deploy action is rather costly, and that it makes a noticeable difference, if a deployment is started locally or remotely. An additional experiment (E2.2) confirms this observation. E2.2 examines the execution time for individual action types by repeatedly executing them while taking the time for their completion. For example, to measure the average time of a *Start* action, it is ensured in advance that a Task Instance is currently running and is in the state *STOPPED*. A Transition is then requested that contains only a single *Start*-action for this Task Instance, and the time until the Transition was completed successfully is measured. Similar procedures are followed for the other action types. Figure 13 summarizes the resulting average execution time for each action type.

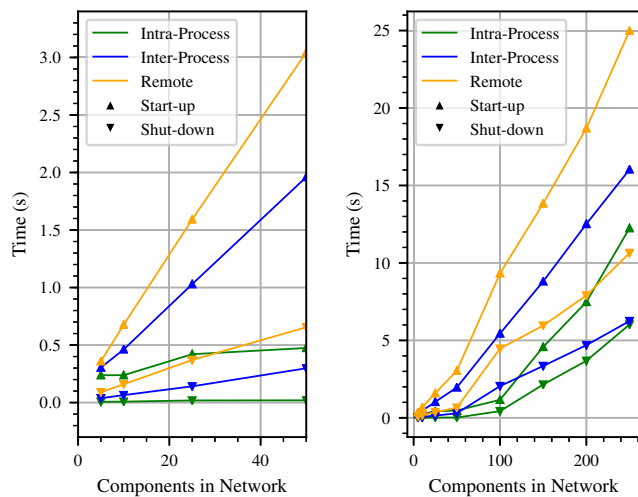


Fig. 12 Time measurements for starting and shutting down component network with different sizes and deployment constellations

This test identifies which operations are more costly than others and how the local and remote execution of the actions differ. The State Transition Actions except *Configure* all need around  $0.5ms$  with an overhead of  $1.3ms$  when network communication is involved. The action *Configure* is much slower, because in addition to the operation call, the handle of the Task Context is refreshed by the CNDHandler afterwards. This is necessary because additional ports can be created during the configuration of the component, which can only be accessed after the Task Context handle has been renewed. This is a rather costly operation and also necessary for the “Deploy” action, which is likewise relatively slow. However, terminating Deployments with the Undeploy action takes even longer than starting them.

A further experiment (E2.3) is concerned with the time needed to reconfigure controllers. A processing chain with  $n$  relay components  $R$  (cf. Eqs. 8 and 10) is reconfigured online such that half of its MessageRelay components are replaced with different instances  $R'$  of the MessageRelay component (cf. Eqs. 9 and 11).

$$A_0 P \rightarrow^{B_1} R_1 \rightarrow \dots \rightarrow^{B_n} R_n \rightarrow A_0 C \quad (8)$$

$$A_0 P \rightarrow^{B_1} R_1 \rightarrow \dots \rightarrow^{B_n} R_{\frac{n}{2}} \rightarrow^{B_{n+1}} R'_1 \rightarrow \dots \rightarrow^{B_{n+\frac{n}{2}}} R'_{\frac{n}{2}} \rightarrow A_0 C \quad (9)$$

$$A_0 P \rightarrow^{A_0} R_1 \rightarrow \dots \rightarrow^{A_0} R_n \rightarrow A_0 C \quad (10)$$

$$A_0 P \rightarrow^{A_0} R_1 \rightarrow \dots \rightarrow^{A_0} R_{\frac{n}{2}} \rightarrow^{A_0} R'_1 \rightarrow \dots \rightarrow^{A_0} R'_{\frac{n}{2}} \rightarrow A_0 C \quad (11)$$

The experiment compares two different cases:

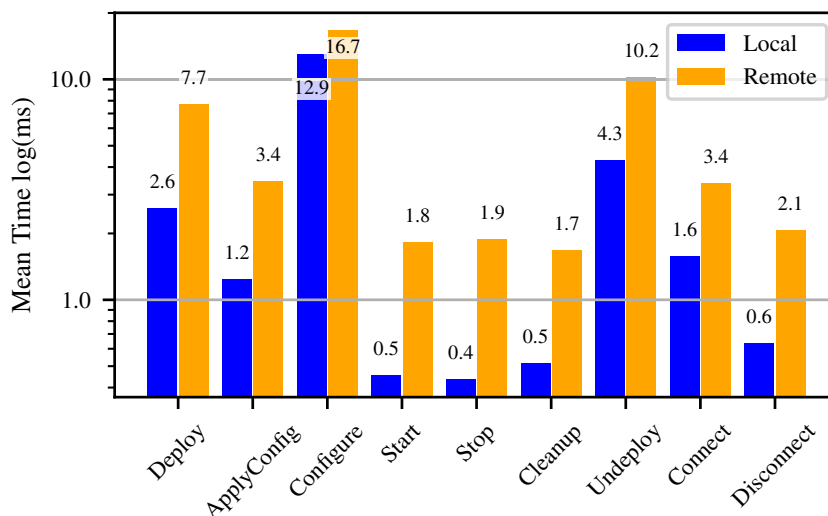
1. The transition of Eqs. 8 to 9 is the *worst case*. Since each component is running in a separate Deployment, and all deployments that are subject to the reconfiguration are on a remote system, many costly remote Deploy and Undeploy actions are required to perform the transition.
2. The transition of Eqs. 10 to 11 is the *best-case* scenario. Here all components are running on the same computer within the same process.

While the worst-case scenario requires  $n/2$  Deploy and Undeploy actions of remote processes in the transition, no Deploy or Undeploy actions are needed in the best case scenario, since all involved components are part of a single large Deployment, which is already started with Eq. 10 and may not be terminated as long there are still components from the deployment running.

Both cases are compared to the case where a static controller reconfiguration (restart) is performed, i.e. between Eqs. 8 and 9 a transition to an empty network is carried out.

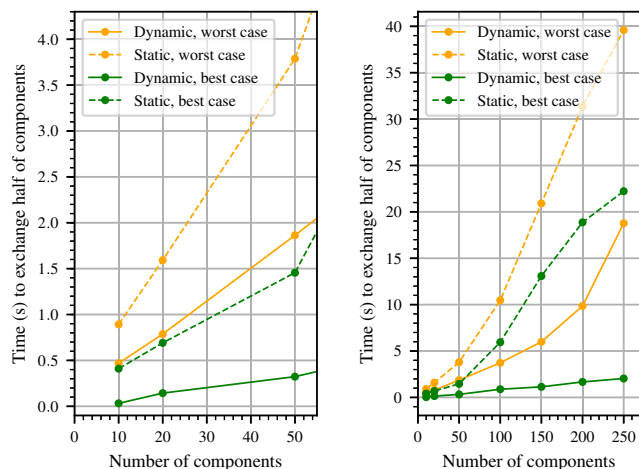
The experiment was executed multiple times for each alternative while measuring the time for completing the reconfiguration. Figure 14 shows the mean execution times

**Fig. 13** Time to perform an individual operation of a component



for each reconfiguration alternative. The worst-case online-reconfiguration can be carried out in less than two second for usual component network sizes with less than 50 components. In the best-case scenario, this value is reduced to 320ms.

In particular, the Deploy/Undeploy and Configure actions have execution times that make it impossible to achieve network reconfiguration in a single control cycle for the control frequencies common in robotics (i.e. 10-1000 Hz). While the data suggests that a complete reconfiguration at a low control frequency for simpler controllers could theoretically be performed in one cycle, it should be noted that there is no guarantee that no real-time violations will occur. In some cases, however, these violations may cause problems with system stability.



**Fig. 14** Time to reconfigure a component network by exchanging half of its components compared to stopping and starting the new component network

### 4.3 Case Study with a Real Robot

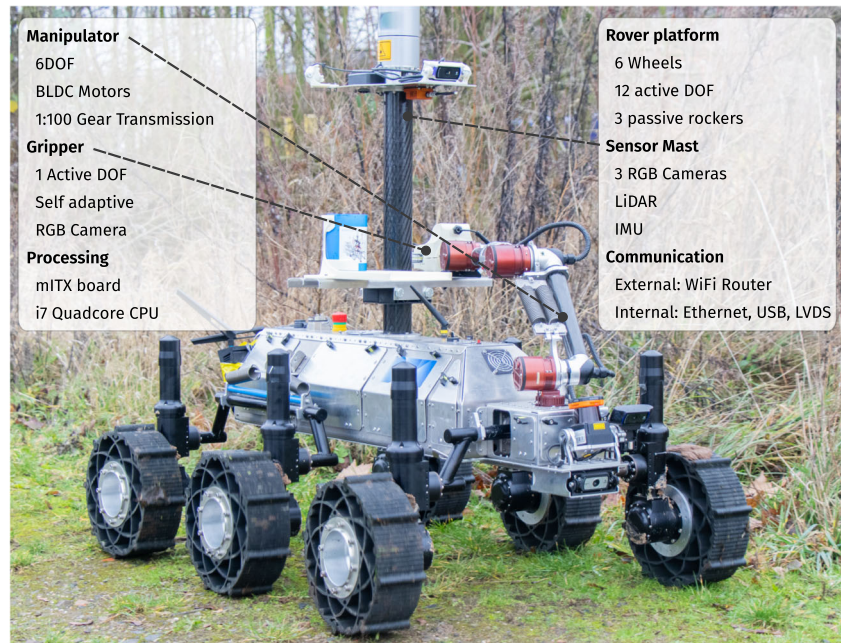
The validation tests from Sections 4.1 and 4.2 examine the theoretically expected runtime performance of the proposed system through abstract experiments. The components in the experiments intentionally do not do significant computational operations, so that only the framework’s overhead is measured. A further experiment investigates how the runtime performance scales to real robot application scenarios. For this purpose we implemented the following four different controllers for a real robot and measured the reconfiguration times between all different controllers:

- Search & Explore (S&E): An area exploration behavior with simultaneous object detection.
- Goal Navigation (GNav): Navigation to a goal which is detected by camera.
- Manipulation (Manip): A visual servoing controller to guide the arm towards an object.
- Manipulator Tele-Operation (MTele): Control of the manipulator with a 3D mouse.

As target platform we chose the system *Artemis* [18] which is depicted in Fig. 15. Artemis is a six-wheeled rover, with the wheels mounted on passive rockers to compensate for ground irregularities. It is equipped with a sensor mast that holds a LiDAR, an IMU and three Full HD USB cameras. A 6 DOF manipulator with a under-actuated and sensorized gripper is attached to the front of the robot, where also a tilting laser scanner and two further cameras USB cameras are mounted.

All onboard computation is realized on *System A*, that was introduced earlier: an Intel i7-3610QE CPU @ 2.30Ghz with 16Gb RAM, hosted on a mini-ITX board from Kontron. The operating system Ubuntu 18.04 and the software subject to testing are installed on a Samsung

Fig. 15 The robot Artemis



SSD. The robot is controlled through an SSH connection established via WiFi using an ASUS router that creates a 2.4 GHz and 5 GHz WiFi network specifically for the robot.

The upcoming paragraphs first introduce the individual controllers followed by the statistical evaluation of this experiment. We use a graphic representation to explain the controllers (see Fig. 16). The Task Instances are shown as boxes with rounded corners, with the unique ID of the Task Instance written in bolt letters in the upper part of the component. Below, the prototype of the component is shown (e.g. *example::Task*). To allow a more compact representation of complex Task Networks, several Task Instances can be grouped in a single box. In this case, the number of Task Instances within the group is written instead of the prototype (e.g. *5 Components*). Input ports of a component are always displayed on the left side, output ports always on the right side and data connections are displayed as an arrow between an output and an input port. In most cases, we do not show ports of a component that are not connected. Additional information of the Task Network

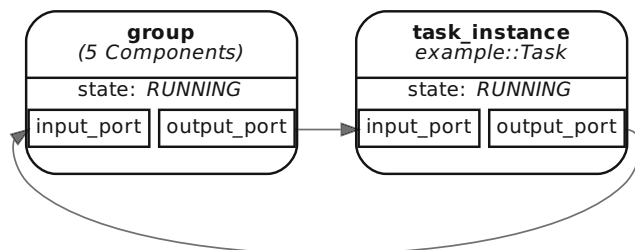


Fig. 16 Example of a graphical notation for a Task Network

model, such as task or connection properties or deployment information, is not included in the visual representation.

This paper is not concerned with evaluating the performance of the implemented controllers, but with the framework that manages these controllers. However, for illustration we have executed them all in an informal experiment and show in Fig. 17 some impressions from the real-life execution of the four different controllers on the robot Artemis.

#### 4.3.1 Search & Explore

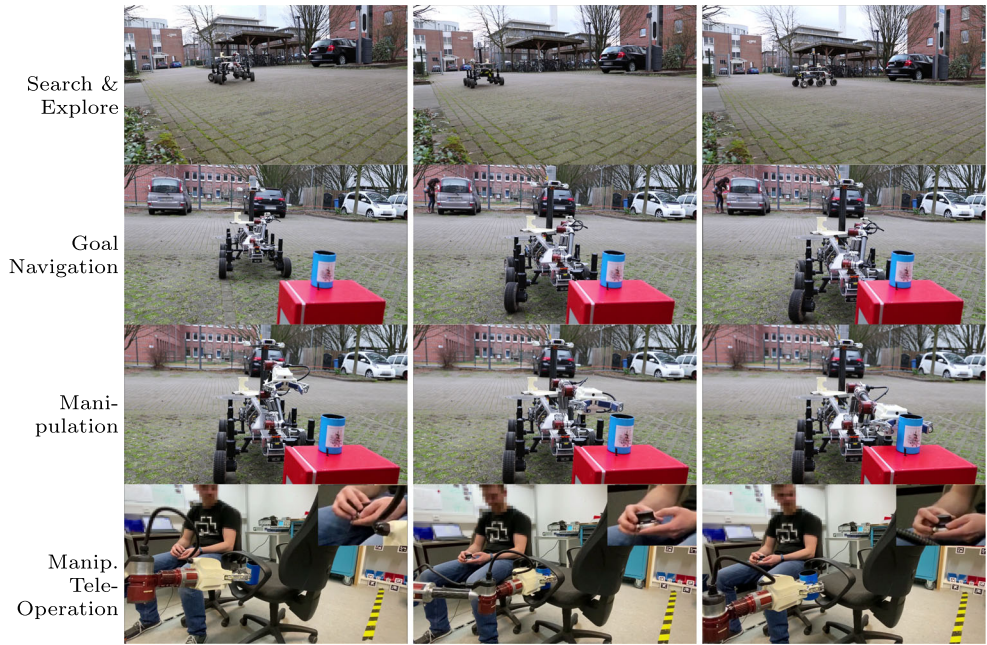
This controller (see Fig. 18) controls the robot in such a way that it explores and maps an unknown area and at the same time searches for a specific object. To achieve this, the component *area\_explorer* generates target coordinates at the borders of the currently known map as input for a navigation planner (*ugv\_nav\_planner*). The map is provided by a graph-based SLAM solution<sup>6</sup> based on the *g<sup>2</sup>o* library [19] using the LiDAR and IMU on the sensor mast. By continuously generating target coordinates at the outer borders of the known area, the map is extended to previously unknown regions.

During the exploration, the images from the frontal camera mounted on the sensor mast are processed by a component that executes the ArUco algorithm [20, 21] on the camera images to detect the red box with the cup on top, which is visible in Fig. 17. (*aruco\_detector*). For this purpose, the box has been specially prepared with an ArUco marker so that it can be detected.

<sup>6</sup><https://github.com/dfki-ric/slam3d>



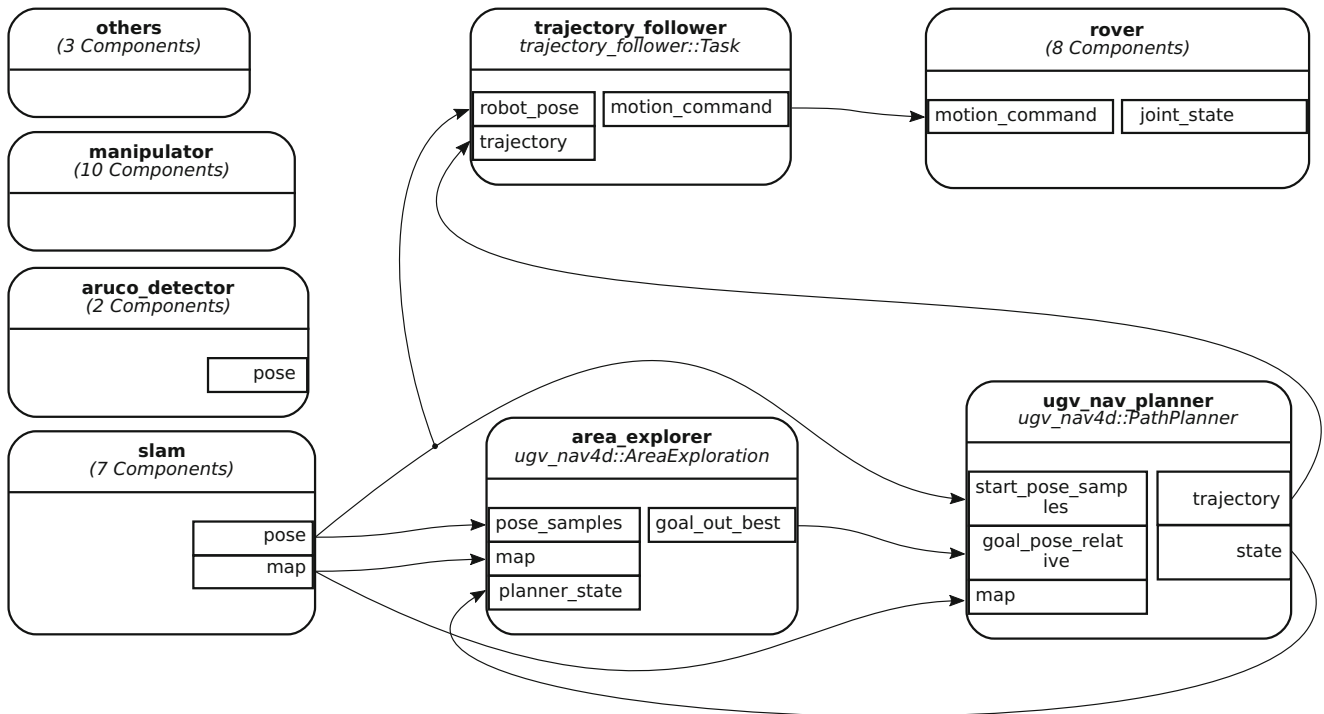
**Fig. 17** Pictures taken while executing the different controllers



The Search & Explore controller consists in total of 33 Task Instances that are distributed over 8 Deployments. There are 49 port connections between the components. The full controller is shown in Appendix B.1.

### 4.3.2 Goal Navigation

The Goal Navigation controller is depicted in Fig. 19. When executed, a data processing pipeline



**Fig. 18** Simplified visual representation of the Search & Explore controller

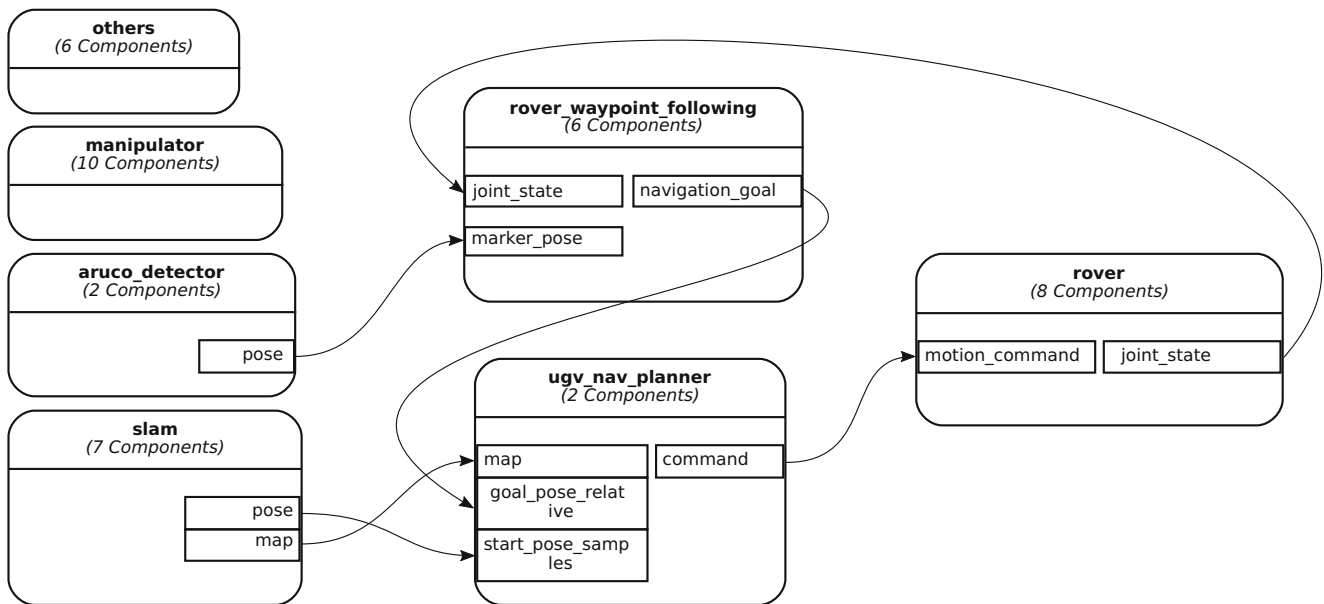


Fig. 19 Simplified visual representation of the Goal Navigation controller

(*rover\_waypoint\_following*) computes goal coordinates in front of the object detected by the *aruco\_detector*. The data processing pipeline adds an offset to the detected object pose and converts it from the camera coordinate system to the *body* frame of the robot, which is located at the bottom of the sensor mast, because the navigation planner expects its input values in that frame. A trajectory controller ensures that the generated trajectory is followed and the *motion\_controller* converts euclidean motion commands into drive velocities and steering angles for the wheels.

To ensure that the planner is not steadily flooded with new goal pose request every time the object is repeatedly detected on a new camera image, a special component filters

repeated samples and allows only to pass a single sample of the object coordinates.

The Goal Navigation controller consists in total of 41 Task Instances that are distributed over 17 Deployments. There are 58 port connections between the components. The full controller is shown in Appendix B.2.

### 4.3.3 Manipulation

The manipulation controller is shown in Fig. 20. It implements an eye-in-hand 3D visual servoing controller for the manipulator. A camera in the gripper is used for ArUco recognition, and a filter component updates the recognized

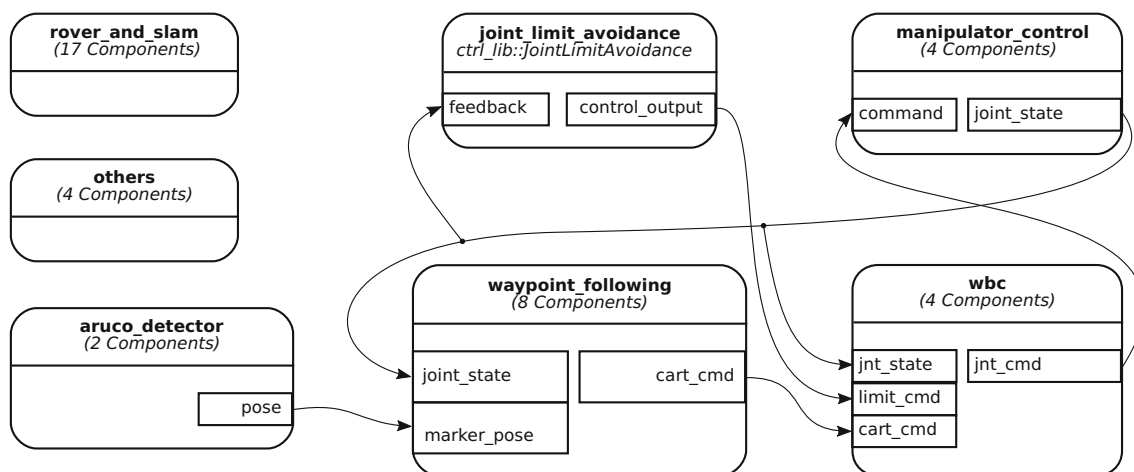
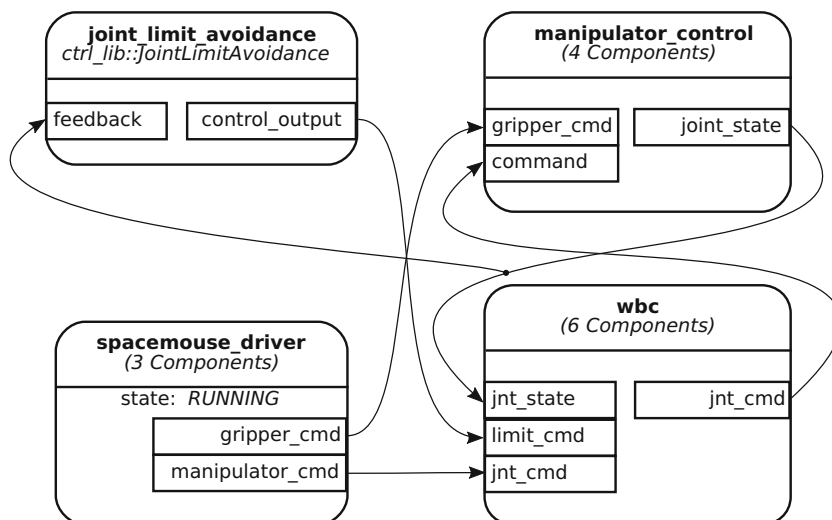


Fig. 20 Simplified visual representation of the Manipulation controller

**Fig. 21** Simplified visual representation of the Manipulator Tele-Operation controller



pose with the movements of the manipulator since the last recognition result, based on the forward kinematics of the manipulator, which is computed with the KDL<sup>7</sup> library at high frequency. In this manner, a sequence of waypoints expressed in the reference system of the detected object is traced and are transformed into the reference system of the robot’s manipulator, so that they can be used as setpoints for the Cartesian controller controlling the manipulator (*waypoint\_following*). Besides the Cartesian controller, a joint limit avoidance controller prevents getting stuck in joint limits. The output of both simultaneously executed controllers is merged in a constraint-based programming approach similar to [22] (*wbc*).

The Manipulation in total consists of 40 Task Instances that are distributed over 16 Deployments and contains 60 connections. The full controller is shown in Appendix B.3.

**4.3.4 Manipulator Tele-Operation**

The Manipulator Tele-Operation controller is shown in Fig. 21. Here an operator uses a 3D mouse (*SpaceMouse* from 3Dconnexion) to create reference twists for the end effector of the manipulator. The resulting reference motion is again merged within the *wbc* component, with the results of the joint limit avoidance controller.

The Manipulator Tele-Operation controller in total consists of 14 TaskInstances that are distributed over 6 Deployments and contains 25 connections. The full controller is shown in Appendix B.4.

**4.3.5 Results**

Each possible transition between two different controllers was executed 10 times while measuring the time from

requesting the controller until the reconfiguration was carried out completely. Table 5 summarizes the mean reconfiguration times and Table 7 shows the numbers of actions required to realize each controller reconfiguration. Finally Table 6 shows for each transition the three most time consuming actions.

The mean transition times between the controllers range from 0.11s to 4.16s with the transition GNav to S&E being by the quickest and all transitions to MTele being by far the slowest. When comparing the actions required for the transition from GNav to S&E (see Table 7) with the expected execution times under ideal conditions from Fig. 13, one can see that the actual transition takes about 1.6 times longer, with most of the time spent configuring the *area\_explorer* component (cf. Table 6). Transitioning to the MTele controller is drastically slowed down by stopping the driver components for the robot’s USB cameras, which stops frame grabbing and closes the devices and its device file descriptor.

The additional time required for a real operation compared to an ideal operation from Fig. 13 results mainly from the calculations that have to be performed within the component implementation. However, it is also apparent from Table 6 that for some different applications of the same actions, there is a large discrepancy in their execution time. A particularly drastic case is the time for the ApplyConfig

**Table 5** Controller reconfiguration times

From \ To	S&E	GNav	Manip	MTele
S & E	-	0.52s	0.55s	4.08s
GNav	0.11s	-	0.59s	4.06s
Manip	0.18s	0.62s	-	4.16s
MTele	2.01s	2.29s	2.23s	-

<sup>7</sup><https://www.orocos.org/kdl>

**Table 6** Most time-consuming actions for each transition

From→To	Action Type	TaskInstance(s)	Time
S&E→ GNav	ApplyConfig	gnav_FK	0.333s
	Configure	gnav_FK	0.024s
	Configure	gnav_SP	0.010s
S&E→ Manip	ApplyConfig	manip_FK	0.324s
	Configure	manip_FK	0.020s
	Configure	wbc	0.018s
S&E→ MTele	Stop	usbcam_tower_front	1.294s
	Stop	usbcam_front	1.290s
	Stop	usbcam_gripper	1.278s
GNav→ S&E	Configure	area_explorer	0.032s
	ApplyConfig	area_explorer	0.005s
	Start	area_explorer	0.004s
GNav→ Manip	ApplyConfig	manip_FK	0.317s
	Configure	wbc	0.025s
	Configure	manip_FK	0.017s
GNav→ MTele	Stop	usbcam_gripper	1.242s
	Stop	usbcam_front	1.238s
	Stop	usbcam_tower_front	1.221s
Manip→ S&E	Configure	area_explorer	0.028s
	Configure	wbc	0.024s
	Stop	wbc_solver	0.006s
Manip→ GNav	ApplyConfig	gnav_FK	0.325s
	Configure	gnav_FK	0.020s
	Configure	wbc	0.015s
Manip→ MTele	Stop	usbcam_gripper	1.276s
	Stop	usbcam_tower_front	1.876s
	Stop	usbcam_front	1.186s
MTele→ S&E	ApplyConfig	area_explorer	0.708s
	Start	usbcam_gripper	0.276s
	Start	usbcam_tower_front	0.252s
MTele→ GNav	ApplyConfig	aruco_detector	0.958s
	Start	usbcam_gripper	0.276s
	Start	usbcam_tower_front	0.255s
MTele→ Manip	ApplyConfig	aruco_detector	0.953s
	Start	usbcam_tower_front	0.189s
	Start	usbcam_gripper	0.184s

action on the *area\_explorer* component in the MTele→S&E transition. In this transition, 708ms are required for the actions, instead of 28ms during the Manip→S&E transition. All transitions to or from MTele require a large number of actions, because the Task Network is much smaller than the others, which are more similar to each other. In addition, during the execution of the numerous actions, many components will continue to run, thereby consuming resources. Using the *htop* tool, we have seen a significant increase in CPU load during the reconfiguration process from/to MTele and suspect this to be the cause of the discrepancy.

In summary, the experiment shows that the system can switch between completely different behaviors such as manipulation or autonomous exploration in about half a se-

cond, which for an outside observer is perceived as an instantaneous response to the request. But of course, costly operations that, e.g. involve interaction with slow hardware devices, or sub-optimal implementations within the components can considerably delay the reconfiguration process.

## 5 Conclusions and Future Work

This paper introduced a software system to model robot controllers from software components, execute them on distributed hardware and switch between them dynamically. The proposed software facilitates the modular development of various behaviors by providing a declarative description of the corresponding robot controllers, thereby relieving the developer of the task of implementing procedures for switching between the behaviors. The proposed method allows to implement different behaviors of the robot independently and to call them at any time regardless of the current state of the robot, thus enabling the development of robots with a wide range of different capabilities.

We presented the metamodels of the model-driven development approach, the runtime architecture for distributed controller execution and an algorithm for generating transitions between arbitrary controller networks. The performance of the system was analyzed with a series of experiments. In an additional experiment with a real robot, for which several controllers were implemented with the proposed method, the feasibility for real robot control problems was shown.

The experiments show that the overhead caused by the component development and communication framework Rock/RTT is in an area that does not significantly impede complex real-time controllers composed of numerous components on distributed hardware. Reconfiguring the component network, on the other hand, requires several control cycles, and there are no framework-side means to ensure system stability during this process. This problem was also identified in [5], and the authors suggest implementing a locally stable mode in key software components.

In the locally stable mode, the components ignore all data input from other components and instead execute internally coded feedback loops that maintain the integrity of the system [5].

If a component is running and is not the subject of a reconfiguration action, its execution is not affected by reconfiguration activities on other components in the component network. Because of this property, which results from RTT scheduling, we suggest that instead of activating a locally stable mode in selected components, a basic stability control subsystem that ensures system integrity should be kept running all the time. This is achieved by integrating the subsystem into each of the component networks used. In our practical example with Artemis,

**Table 7** Number of actions required for transitioning between controllers

From \ To	S&E	GNav	Manip	MTele
S & E	–	0 Undeploy 4 Disconnect 9 Deploy 9 Apply config 13 Connect 20 State changes <b>55 Total</b>	0 Undeploy 4 Disconnect 8 Deploy 12 Apply config 15 Connect 34 State changes <b>73 Total</b>	5 Undeploy 32 Disconnect 3 Deploy 5 Apply config 8 Connect 58 State changes <b>111 Total</b>
GNav	9 Undeploy 13 Disconnect 0 Deploy 1 Apply config 4 Connect 20 State changes <b>47 Total</b>	–	9 Undeploy 13 Disconnect 8 Deploy 12 Apply config 15 Connect 50 State changes <b>107 Total</b>	14 Undeploy 41 Disconnect 3 Deploy 5 Apply config 8 Connect 74 State changes <b>145 Total</b>
Manip	8 Undeploy 15 Disconnect 0 Deploy 5 Apply config 4 Connect 34 State changes <b>66 Total</b>	8 Undeploy 15 Disconnect 9 Deploy 13 Apply config 13 Connect 50 State changes <b>108 Total</b>	–	13 Undeploy 43 Disconnect 3 Deploy 6 Apply config 8 Connect 76 State changes <b>149 Total</b>
MTele	3 Undeploy 8 Disconnect 5 Deploy 24 Apply config 32 Connect 58 State changes <b>130 Total</b>	3 Undeploy 8 Disconnect 14 Deploy 32 Apply config 41 Connect 74 State changes <b>172 Total</b>	3 Undeploy 8 Disconnect 13 Deploy 32 Apply config 43 Connect 76 State changes <b>175 Total</b>	–

The sum of all individual actions for each transition is shown in bold face

system stability is ensured by an online trajectory generation component (*trajectory\_generation*, cf. Appendixs B.1–B.4) that is connected upstream of the joint controllers. This component permanently calculates dynamically executable motion commands at control rate, even if the setpoints are only sporadically generated by the higher-level controller [23]. The permanent presence of the online trajectory generator ensures that the robot joint movement remains jerk-free, even if the real-time conditions are violated during setpoint generation, e.g. because the setpoint generator is replaced due to a reconfiguration of the component network. In addition, the system will come to a standstill if the setpoints are not renewed for a longer, configurable time span, by generating zero velocities when the joint controllers are driven speed-controlled. By integrating additional safety-relevant components into the stability subsystem, such as a collision avoidance component, even longer reconfiguration time spans could be bridged.

While the experiments showed satisfactory performance, a further reduction of the processing costs caused by the

framework overhead would be desirable. An easy way to achieve a performance gain for reconfiguring controllers on multi-CPU systems is to parallelize the application of a Transition.

Currently rock-runtime only supports x86 architectures as execution platform. In principle all execution platforms could be targeted, that are supported by the underlying communication framework, as long as the *ProcessServer* can be compiled and executed or re-implemented for that platform. There are first results in porting the Rock system to ARM architectures as well (cf. [24]), but for rock-runtime this was not yet done so far.

An interesting addition is the off- and online validation of the modeled controllers. For example, a semantic analysis to identify invalid or suboptimal patterns in the data flow or deployment setup could be used to detect errors in the controller modeling process earlier. The framework already measures the execution time for all reconfiguration actions. This data could be analyzed by additional tools to give component developers hints on how

to optimize their components further, or to provide offline estimates of expected reconfiguration times. Furthermore, the framework could be extended by the acquisition of execution speeds in normative operation of the components, e.g. to determine mean and worst case execution times for individual software components, on the basis of which an offline estimation of the processing time for entire processing chains within a controller is possible.

With a permanent comparison of the current component network with the last requested one, an online integrity check can be performed. For this purpose, the underlying framework must provide information about all running components and their data connections (runtime introspection). The TaskNetworkHandler module could then be extended accordingly to detect differences between the expected and the actual system state and as a reaction display the error or perform extended safety and error handling operations. With the latest developments in Rock, a runtime introspection of the running component network can already be performed, so that the requirements for the framework for online integrity checking are already fulfilled.

Furthermore, it is possible to improve accessibility and usability by providing additional tools for composing controller networks such as GUIs or domain-specific scripting languages.

**Acknowledgements** The authors would like to thank all developers of the Rock and Orocos RTT framework and especially Sylvain Joyeux and Janosch Machowinski for their fundamental work on Rock. Furthermore, we thank the members of the student project THORO for their support in the validation tests on the robot Artemis and Prof. Hendrik Wöhrle for his valuable input in writing the paper.

This work on this paper was performed within the project D-Rock and Q-Rock, funded by the Federal Ministry of Education and Research (BMBF) under grant number 01-IW-15001 and 01-IW-18003.

**Funding** Open Access funding provided by Projekt DEAL.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

## Appendix A: Examples for Serialized Transition and Task Network Models

---

```
transition:
- type: APPLY_CONFIG
  task_model_type: providers::Waypoints
  task_id: waypoint_provider
  config_names:
  - unstow
- type: TASK_STATE_ACTION
  task_id: waypoint_provider
  task_action: CONFIGURE
- type: TASK_STATE_ACTION
  task_id: waypoint_provider
  task_action: START
```

---

**Listing 2** Example Transition in YAML format

---

```
tasks:
  manipulator_driver:
    type: "ndlcom_aila_joints::
      AILAJointTask"
    config_names: ["default",
      "artemis"]
    state: RUNNING
  manipulator_serial:
    type: "serial_ndlcom::Task"
    config_names: ["default",
      "manipulator"]
    state: RUNNING

connections:
  manipulator_driver_to_serial:
    from:
      task_id: manipulator_driver
      port_name: ndlcom_message_out
    to:
      task_id: manipulator_serial
      port_name: ndlcom_message_in
    transport: CORBA
    type: BUFFER
    size: 50
  manipulator_serial_to_driver:
    from:
      task_id: manipulator_serial
      port_name: ndlcom_message_out
    to:
      task_id: manipulator_driver
      port_name: ndlcom_message_in
    transport: CORBA
    type: BUFFER
    size: 50

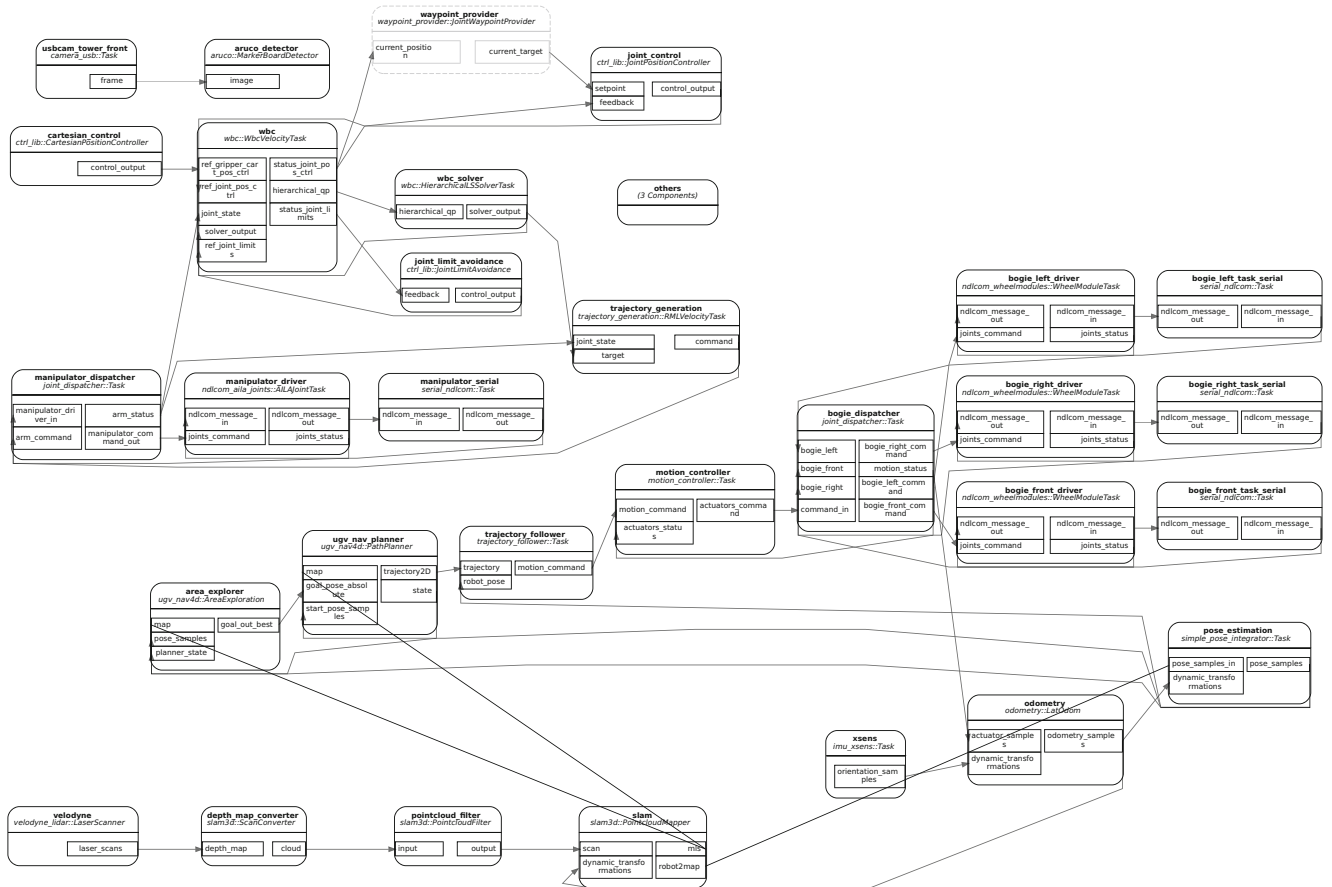
deployments:
  manipulator:
    deployer: orogen
    process_name: sb_manipulator
    hostID: artemis-control
    taskList:
      manipulator_serial:
        manipulator_serial
      manipulator_driver:
        manipulator_driver
```

---

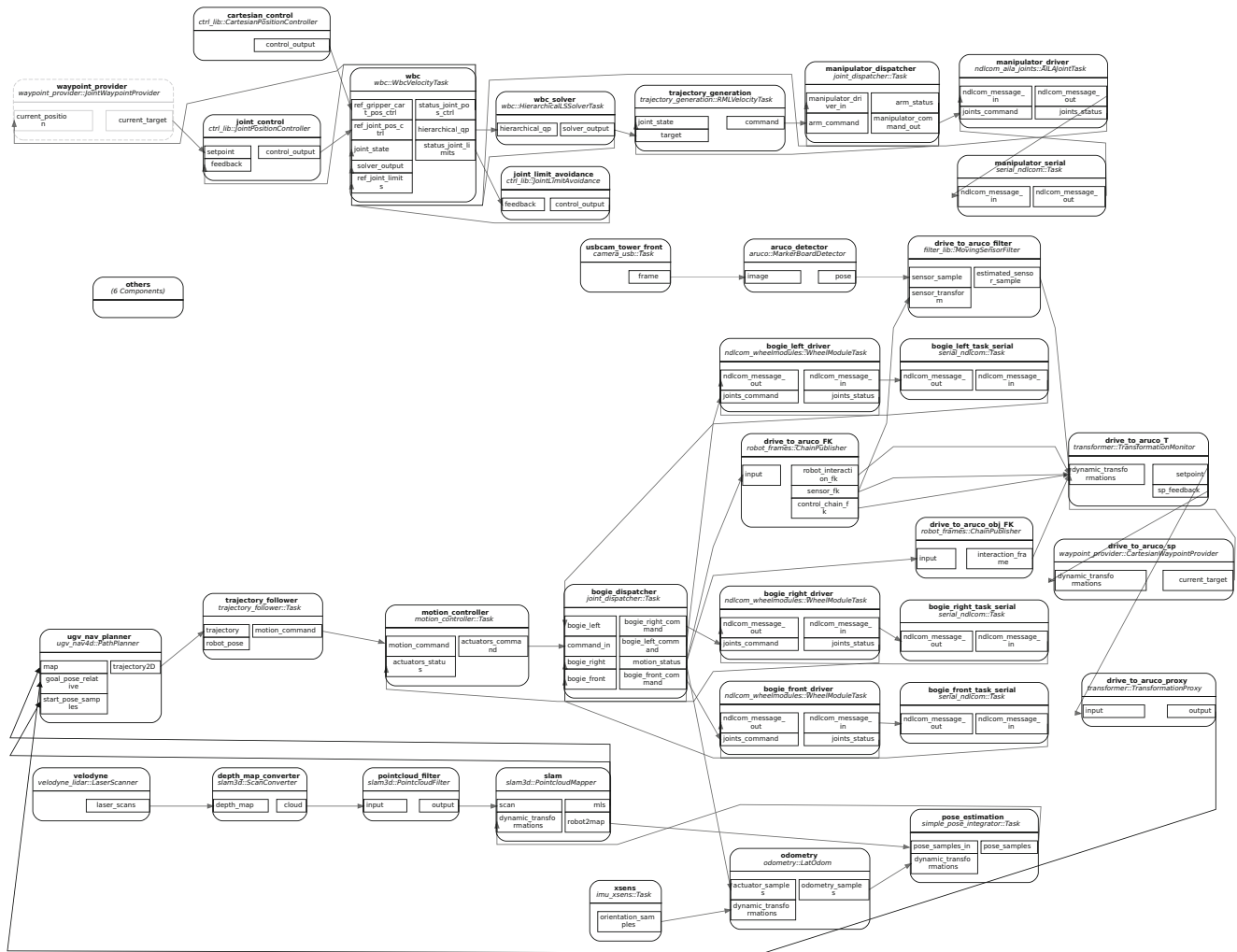
**Listing 3** Example Task Network in YAML format

# Appendix B Example Applications

## B.1 Search & Explore Controller

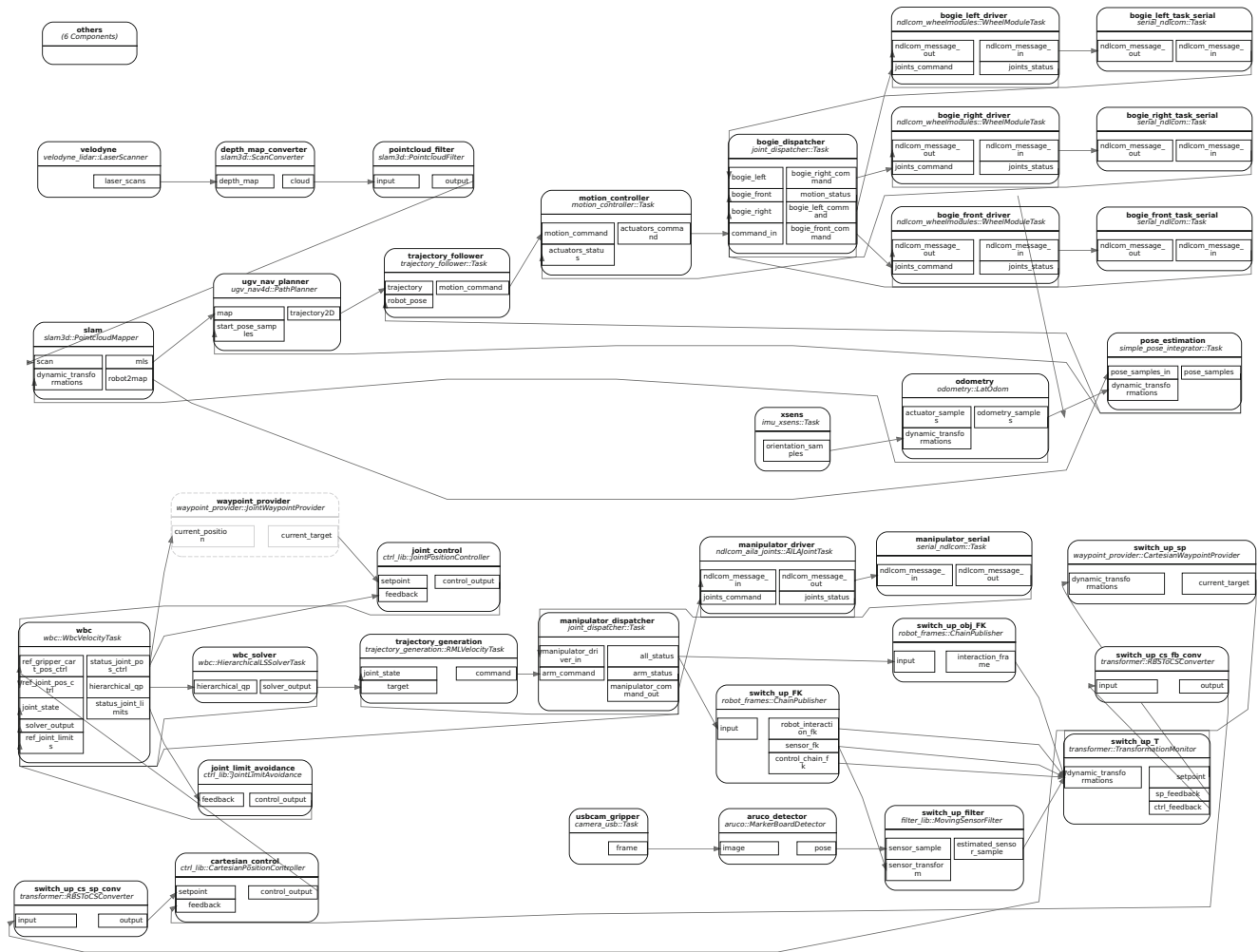


## B.2 Goal Navigation Controller

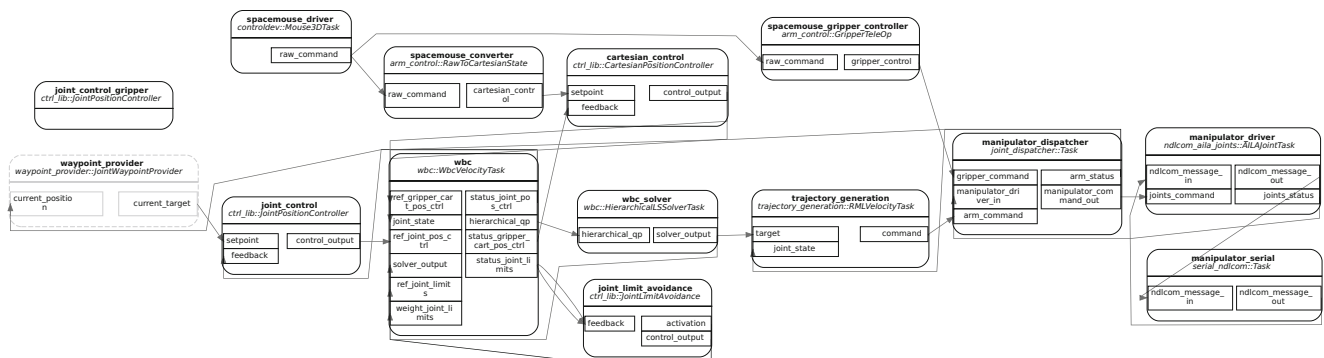




### B.3 Manipulation



### B.4 Manipulator Tele-Operation Controller



## References

1. Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., Ng, A.: ROS: an open-source Robot Operating System. In: ICRA workshop on open source software, vol. 3, pp. 1–5 (2009)
2. Joyeux, S., Schwendner, J., Roehr, T.M.: Modular software for an autonomous space rover. In: Proceedings of the 12th International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS, 2014), (Montreal, Québec, Canada), pp. 1–8 (2014)
3. Metta, G., Fitzpatrick, P., Natale, L.: YARP – yet another robot platform, Version 2.3.20. *Int. J. Adv. Robot. Syst.* **3**(1), 43–48 (2013)
4. Santos, A., Cunha, A., Macedo, N., Arrais, R., dos Santos, F.N.: Mining the usage patterns of ROS primitives. In: in 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), (Vancouver, BC), pp. 3855–3860. IEEE, Sept (2017)
5. Stewart, D.B., Khosla, P.: The chimera methodology: Designing dynamically reconfigurable and reusable real-time using port-based objects software. *Int. J. Softw. Eng. Knowl. Eng.* **6**(2), 249–277 (1996)
6. Lyons, D.M., Arbib, M.A.: A formal model of computation for sensory-based robotics. *IEEE Trans. Robot. Autom.* **5**, 280–293 (June 1989)
7. Soetens, P., Bruyninckx, H.: Realtime hybrid task-based control for robots and machine tools. In: Proceedings of the 2005 IEEE International conference on robotics and automation, (Barcelona, Spain), pp. 259–264 (2005)
8. Soetens, P.: A Software framework for real-time and distributed robot and machine control. PhD thesis, Issue: May ISBN: 9056826875 (2006)
9. Bézivin, J.: On the unification power of models. *Softw. Syst. Modeling* **4**(2), 171–188 (2005). ISBN: 1619-1366, 1619-1374
10. Bischoff, R., Guhl, T., Prassler, E., Nowak, W., Kraetzschmar, G., Soetens, P., Haegele, M., Pott, A., Breedveld, P., Broenink, J., Brugali, D., Tomatis, N.: BRICS - Best Practice in Robotics. In: ISR 2010 (41st International symposium on robotics) and ROBOTIK 2010 (6th german conference on robotics), (Munich, Germany), VDE, pp. 968–975 (2010)
11. Joyeux, S., Albiez, J.: Robot Development: from Components to Systems. In: 6Th national conference on control architectures of robots, (Grenoble, France), INRIA Grenoble Rhône-Alpes May, pp. 1–15 (2011)
12. Schlegel, C., Lotz, A., Lutz, M., Stampfer, D., Inglés-Romero, J.F., Vicente-Chicote, C.: Model-driven software systems engineering in robotics: covering the complete life-cycle of a robot. *Inform. Technol.* **57**(2), 85–98 (2015). ISBN: 1611-2776
13. Nordmann, A., Hochgeschwender, N., Wigand, D., Wrede, S.: A survey on domain-specific modeling and languages in robotics. *J. Softw. Eng. Robot.* **7**, 75–99 (2016)
14. Wang, S., Shin, K.: Reconfigurable software for open architecture controllers (2001)
15. Inglés-Romero, J.F., Lotz, A., Chicote, C.V., Schlegel, C.: Dealing with run-Time Variability in Service robotics: Towards a DSL for non-Functional Properties arXiv:1303.4296 [cs], pp. 1–8, Mar (2013)
16. Fleurey, F., solberg, A.: A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In: Schürr, A., Selic, B. (eds.) *Model Driven Engineering Languages and Systems*, vol. 5795, pp. 606–621. Springer, Berlin (2009)
17. Klotzbücher, M., Biggs, G., Bruyninckx, H.: Pure Coordination using the coordinator-Configurator Pattern, ArXiv, vol. abs/1303.0066, pp. 11–4, Feb arXiv:1303.0066 (2013)
18. Schwendner, J., Roehr, T.M., Haase, S., Wirkus, M., Manz, M., Arnold, S., machowinski, J.: The artemis rover as an example for model based engineering in space robotics (2014)
19. Kummerle, R., Grisetti, G., Strasdat, H., Konolige, K., Burgard, W.: G<sup>2</sup>O: A General Framework for Graph Optimization. In: 2011 IEEE International Conference on Robotics and Automation, (Shanghai, China), IEEE, May, pp. 3607–3613 (2011)
20. Garrido-Jurado, S., Muñoz-Salinas, R., Madrid-Cuevas, F., Medina-Carnicer, R.: Generation of fiducial marker dictionaries using mixed integer linear programming. *Pattern Recogn.* **51**, 481–491 (2016)
21. Romero-Ramirez, F.J., Muñoz-Salinas, R., Medina-Carnicer, R.: Speeded up detection of squared fiducial markers. *Image Vision Comput.* **76**, 38–47 (2018). ISBN: 0262-8856
22. Schutter, J.D., Laet, T.D., De Schutter, J., De Laet, T., Rutgeerts, J., Decré, W., Smits, R., Aertbeliën, E., Claes, K., Bruyninckx, H.: Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty. *Int. J. Robot. Res.* **26**(5), 433 (2007). Publisher: SAGE Publications
23. Kroger, T.: Opening the door to new sensor-based robot applications - the reflexxes motion libraries. In: 2011 IEEE International Conference on Robotics and Automation, (Shanghai, China), pp. 1–4, IEEE May (2011)
24. Roehr, T.M., Willenbrock, P.: Binary packaging for the robot construction kit (2018)

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

**Malte Wirkus** received his Diploma in Computer Science at the University of Bremen in 2010. He joined the Robotics Innovation Center (RIC) of the German Research Center for Artificial Intelligence (DFKI GmbH) in 2010. In different research and industry projects, he gained experiences in the fields of robotic mobile manipulation, multi-agent architectures, human-robot collaboration and space robotics. With his current scientific research interest in control architectures and frameworks for robot application development, he works as researcher and project leader at DFKI-RIC.

**Sascha Arnold** received the Dipl.-Inf. degree in computer science from the University of Bremen, Bremen, Germany, in 2014. His thesis was about robust 3-D environment modeling using pose graph optimization on LiDAR data. He is currently with German Research Center for Artificial Intelligence (DFKI), Kaiserslautern, Germany, as a Researcher. His research interests include localization and mapping in challenging domains, digital signal processing, and sensor fusion, and their application in space and underwater robotics.

**Elmar Berghöfer** received his Diplom (masters degree) in computer science from the University of Bielefeld, Germany, in 2011. His thesis concerns object recognition based on multi-modal sensor information. He joined the German Research Center for Artificial Intelligence DFKI, in Germany in 2011. He gained here experience in research, project management and acquisition. His research interests include, the field of machine learning, in particular object classification, sensor fusion, and on-line learning and currently focuses on the field of data stream mining.