Title:	A Reconfigurable robot workCell for fast set-up of automated assembly processes in SMEs
Acronym:	**ReconCell**
Type of Action:	Innovation Action
Contract Number:	680431
Starting Date:	1-11-2015
Ending Date:	31-10-2018

Deliverable Number:	D6.1
Deliverable Title:	Technical report on software and hardware components in the workcell
Type (Public, Restricted, Confidential):	PU
Authors :	Martin Bem, Robert Bevec, Miha Deniša, Timotej Gašpar, Barry Ridge, Igor Kovač, Minija Tamosiunaite, Tatyana Ivanovska, Andrej Gams, Christian Schlette, Norbert Krüger, T. Rajeeth Savarimuthu, Rune Larsen, and Aleš Ude
Contributing Partners:	JSI, UGOE, SDU, MMI, BOR

Estimated Date of Delivery to the EC:	31-10-2016
Actual Date of Delivery to the EC:	04-11-2016

Contents

1 Executive Summary **3**

2 Minimal Workcell Design **4**
2.1 Hardware 4
2.2 Software 5

3 Hardware Architecture **6**
3.1 Passive linear unit 7
3.2 Reconfigurable jig modules (hexapods) 8
3.3 Trolleys 8
3.4 Plug and Produce Connectors 8
3.5 Robot end-effectors 8
3.6 Quick tool changer 9
3.7 Tool hanger module 9

4 Software Architecture **9**
4.1 Simulink Real-Time Target Interface 9
4.2 Robot State Publisher 13
4.3 Robot Control Interface 15
4.4 State Machine Interface 22
4.5 Simulation and Visual Programming Module 23
4.6 Vision Module Interface 23
4.7 Digital Interface 24

References **25**

Appendices **26**

A Simulink Real-Time Target output data **26**

B Simulink Real-Time Target input data **27**

1 Executive Summary

This deliverable describes the functions, inputs and outputs of different components in the ReconCell hardware and software environment, which was developed to facilitate the set-up of automated assembly solutions [7]. The information provided here is vital for partners who need to interface with the components and as an in-depth description of the robot workcell structure.

In the first part of the deliverable we describe the minimal design of the reconfigurable workcell that can be deployed. An overview of the minimal hardware and software components is provided. ROS (Robot Operating System) [9] is used as a basis for software development.

In Section 3 we provide descriptions of all the components that attach to the minimal workcell design and make it a functional, reconfigurable robotic workcell. The descriptions of several reconfigurable elements are complemented with technical documentation in Deliverable **D1.1**: Technical report on the design of innovative workcell elements.

Section 4 describes the software architecture of the reconfigurable workcell. While the backbone is the same, ROS-based communication network as used in the minimal workcell design, several aspects of the design and several optional software elements are described in this section, including the integration of the VEROSIM software package for the visualization and visual programming.

The appendices provide the technical specifications of the communication packages.

2 Minimal Workcell Design

2.1 Hardware

The minimal workcell design encompasses only the framework that enables the cell to to be upgraded with application specific modules. It consists of:

- Basic frame
- Robot Module
  - Robot
  - SLRT Server for robot control
  - Quick tool changer
  - Force/torque sensor
  - Gripper/end-effector tool
- Computer running *roscore* and basic ROS nodes (Robot State Publisher, Robot Control Interface, State Machine Interface)
- Plug & Produce connectors

The basic frame is made of structural square steel tubing connected with the BoxJoint system. BoxJoint is a modular system for assembling beam frames without welding. It uses a system of plates connected with nuts and bolts. The flexibility of the BoxJoint system enables the frame to be changed depending on the application needs. The detailed frame design is described in **D1.1**: Technical report on the design of innovative workcell elements (section "Reconfigurable Frame").

An essential part of the minimal core workcell design is a robotic arm. It is the core component that executes the desired assembly task. It is fixed on the frame using the aforementioned BoxJoint system, which allows mounting the robot at different positions on the frame depending on the task. A computer running real time OS (SLRT) is used for controlling the robot. A force/torque sensor is mounted at the end-effector. Following the F/T sensor, the tip of the robot is equipped with a pneumatic tool changer. It enables us to quickly exchange end-effectors based on the task requirements. To provide maximum grasping reliability, a different set of gripper fingers can be used for each assembly part. The tool changer increases the flexibility by enabling end-effector changes with no human intervention, thus allowing the robot to manipulate different assembly parts.

The basic frame can be expanded by connecting peripheral modules via the Plug and Produce connectors. The connectors provide mechanical coupling as well as power, communication and pneumatic connections. A more extensive description of the Plug & Produce connectors is given in Section 3.4.

Figure 1 depicts a variant of the minimal workcell with three Plug & Produce connectors on each side (6 altogether) and the robot. Other minimal designs are possible. The same scheme also refers to the software architecture, shown in Figure 2.

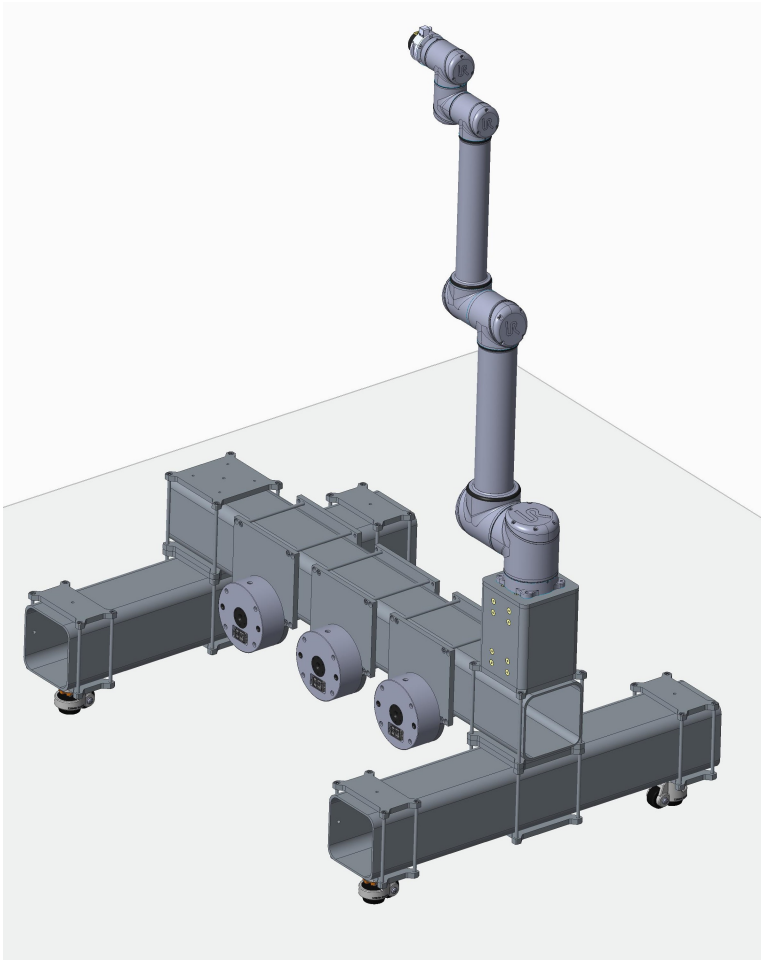

Figure 1: Example rendering of the minimal workcell design.

2.2 Software

ReconCell's core system also includes the necessary software tools to enable programming of the robot's motion. The essential elements of the software architecture are:

- a robot module,
- a computer with the ROS system running *roscore*.

Figure 2 shows a representation of the minimal software architecture.

Part of the robot module is a custom Simulink Real-Time Target Server (SLRT), which was developed within the project to ensure that the robot can be reliably controlled with the maximum frequency of the provided robot controller (125 Hz in case of UR10 robots). Advanced trajectory generation methods and feedback control strategies are implemented on this server. High sampling frequencies are for example necessary for high-quality force control. Our real-time controller therefore runs with a higher frequency (1000 Hz) than the provided robot controller. This means that in case of UR-10 robot, the real-time controller can process 8 samples from the force sensor before calculating new motor commands, which are sent to the robot controller. This way we can improve force control.

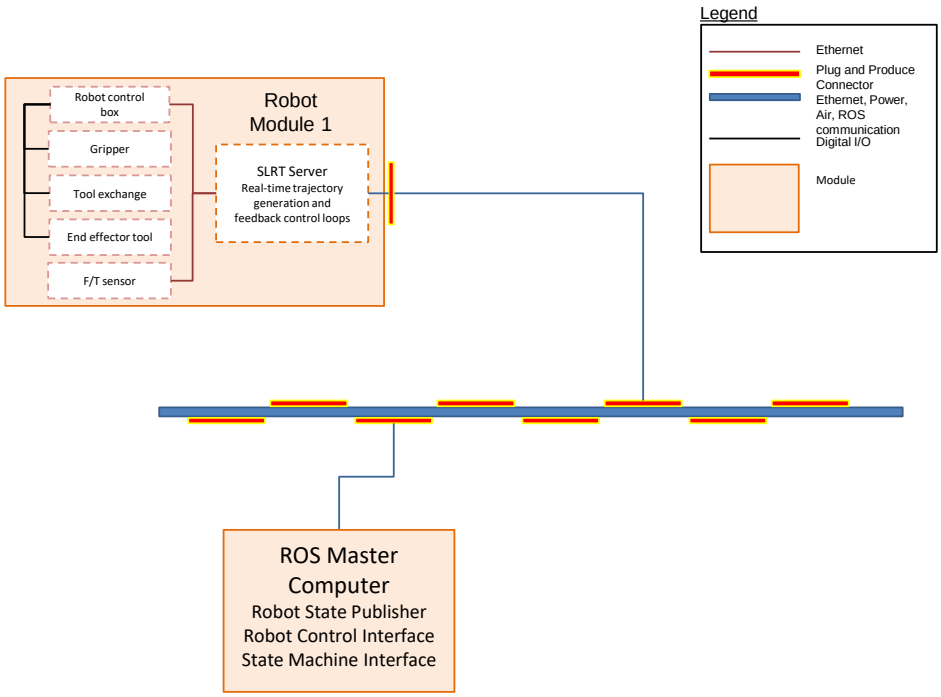

Figure 2: Schematics of the minimal workcell software and hardware architecture.

ROS architecture provides a good framework for connecting different devices on a shared network. It enables programming of the desired robot actions using data from the connected devices. The ROS Master Computer is running *roscore*, which is a collection of nodes and programs that are pre-requisites of a ROS-based system. *roscore* is necessary in order for ROS nodes to communicate. ROS master computer also hosts nodes that handle communication between the robot module and the ROS system (Robot State Publisher and Robot Control Interface). Robot State Publisher handles the communication of the robot state (and other hardware included in the robot module) from the SLRT Server to the ROS-based system. Robot Control Interface implements action servers and services and handles communication to the SLRT Server. The programming of action sequences is done using State Machine Interface (SMACH) [2], which also runs on ROS Master Computer.

3 Hardware Architecture

The hardware is composed of the following components:

- Minimal cell
- Passive linear unit
- Reconfigurable jig module
- Trolleys
- Plug & Produce connectors

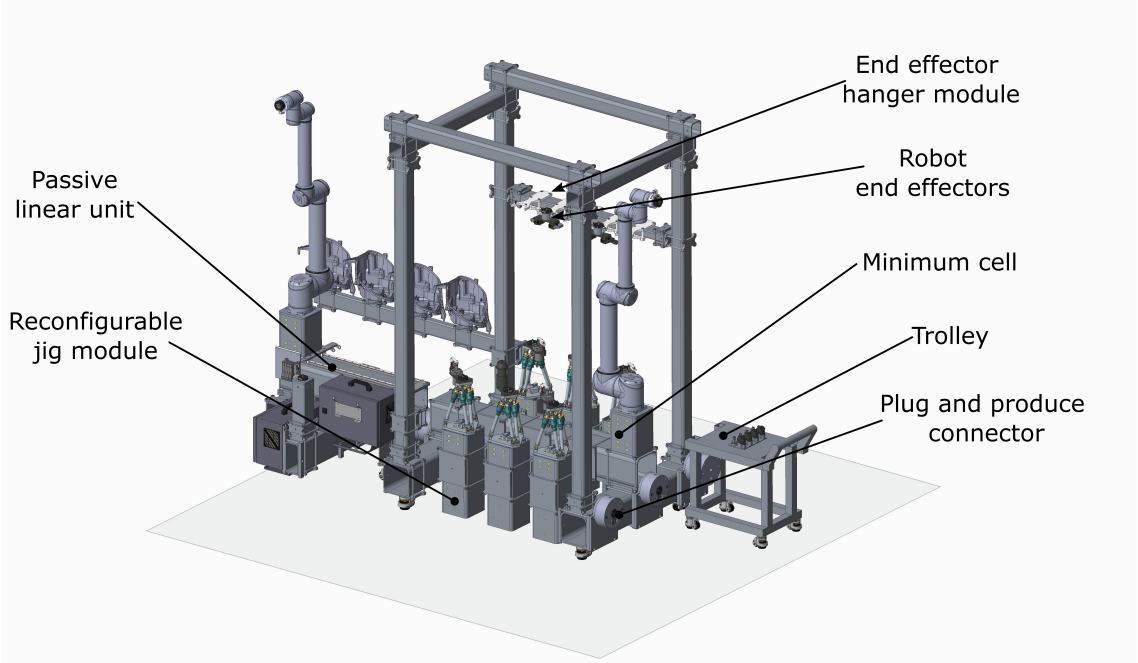

Figure 3: Hardware architecture consisting of the minimal cell and several peripheral elements.

- Robot end-effectors
- Quick tool changer
- Tool hanger Module
- Vision Module
- Simulation Module

Some of the above mentioned components can be seen in the extended workcell shown in Figure 3. The complete software architecture is depicted in Figure 4. While the minimal workcell was described in Section 2, in the following we explain the remaining components.

3.1 Passive linear unit

The purpose of the passive linear unit is to expand the work area of the robot with minimum additional cost. Conventional actuated solutions are extremely expensive and thus inappropriate for SMEs. A way to reduce the cost without significantly reducing the functionality is to omit the actuation and position sensing from the linear unit. The robot itself is used to propel itself along the linear rail. This is achieved by connecting the tip of the robot to the frame and then by using the robot actuators and position control to move the base of the robot. This approach is appropriate for applications where the need to move the robot is not too frequent. A detailed description of the linear rail can be found in Deliverable **D1.1**. The unit uses the digital interface described in Section 4.7 to release its breaks.

3.2 Reconfigurable jig modules (hexapods)

Many robotic workcells use application-specific jigs. This is not a big issue in mass production where the time intervals between production changes are typically long. In SMEs, however, production often occurs in small series of highly customized parts. In this case, product-specific jigs present a significant hindrance for the use of robots. To overcome this problem, it is useful to develop affordable reconfigurable jigs. Aspects of reconfigurable fixture design are reviewed in [6]. In ReconCell we use passive Stewart-Gough platforms for holding workpieces during robotic operation, which are in this deliverable also referred to as hexapods. The hexapods are not actuated and contain no position sensing equipment. To maintain the position, hydraulic brakes are used in each leg of the hexapod. Reconfiguration is done by grabbing the top platform of the hexapod with the robot and after releasing the brakes, moving it to a new position. The detailed description of the reconfigurable jigs can be found in Deliverable **D1.1**. The developed hexapods use the digital interface described in Section 4.7 to release the breaks or trigger fixture clamps.

3.3 Trolleys

A fast method for inputing parts and peripheral equipment is needed. In ReconCell we are going to use Plug & Produce connectable trolleys. The trolleys are still to be developed, but they will be equipped with special trays that will maintain the position of the parts. The peripheral equipment will be mounted directly to the trolley. This means that the position and orientation of equipment and/or assembly parts will be known as soon as the trolley is connected to the workcell. The proposed Plug & Produce connectors will also provide pneumatic and electric connections to power the peripheral devices mounted to the trolleys. The detailed description of the trolleys can be found in Deliverable **D1.1**.

3.4 Plug and Produce Connectors

To facilitate the reconfiguration process, we propose to use Plug & Produce connectors. One end of the connector is to be mounted to the frame of the workcell, while the other will attach to different peripheral modules. The connector provides mechanical coupling as well as all other electrical and pneumatic connections to deliver full functionality of the peripheral module. The connector locks by pushing both sides together. To disconnect it we need pneumatics, controlled by the digital interface described in Section 4.7. The detailed description of the Plug & Produce connector can be found in Deliverable **D1.1**.

3.5 Robot end-effectors

In order to manipulate objects and their parts, the robot needs different end-effectors. To ensure reliable grasping it is common to develop task-specific end-effectors. In ReconCell we are going to use parallel, angular and centric pneumatic grippers. Some of the task-specific fingers have already been developed. All end-effectors are designed so that they can be stored in the tool hanger module.

3.6 Quick tool changer

Due to various task-specific grippers and end-effectors, the need for quick end-effector changes arises. To comply with this demand, a Destaco QC-30 quick tool changer [4] has been integrated into the workcell. It is composed of two parts, one permanently attached to the tip of the robot, the other to different end-effectors. The two parts can be connected together using pneumatics, controlled by the robot's digital interface described in Section 4.7.

3.7 Tool hanger module

End-effectors currently not in use have to be stored within reach of the robot. The storing positions must be known with sufficient accuracy so that a robot can later retrieve them. A tool hanger module was developed for this purpose. It uses two centering pins to position the end-effectors. All ReconCell end-effectors are compatible with the developed tool hanger module. The detailed description of the tool hanger modules can be found in Deliverable **D1.1**.

4 Software Architecture

This section describes various software components that are either part of the minimal workcell software architecture (described in Section 2.2) or are introduced for specific applications in different production settings. These software components are integrated into the overall software architecture shown in Figure 4. In this section we provide a detailed description of software components included in "Robot module", which belongs to the core software of the workcell and implements real-time robot control strategies and communication with the robot and peripheral hardware attached to the robot. The non-real-time part of the system is implemented using ROS (Robot Operating System) [9] and consists of several ROS nodes that run on ROS Master Computer. Other ROS-based modules are added based on task requirements, e. g. "Vision module" and "Digital interface unit". These modules provide additional functionality to the robot workcell. All software modules are connected to Ethernet and communicate via ROS [9].

4.1 Simulink Real-Time Target Interface

Universal robot UR10 [0] was selected as the default robot for the ReconCell system. Out of the box UR10 allows the user to program movements using predefined control strategies. However, the ReconCell software architecture provides an abstraction layer to support switching between different robots. The aim is to provide a number of trajectory and feedback control strategies independently of the selected robot and enable the programming of new strategies via a suitable control interface. For this reason we developed a real-time server that runs at 1000 Hz and communicates with the selected robot at the highest frequency allowed by the robot's control box. In case of UR10, the highest frequency is 125 Hz. The developed real-time server also enables the implementation of advanced feedback control algorithms, e. g. force control, without being limited to a specific robot.

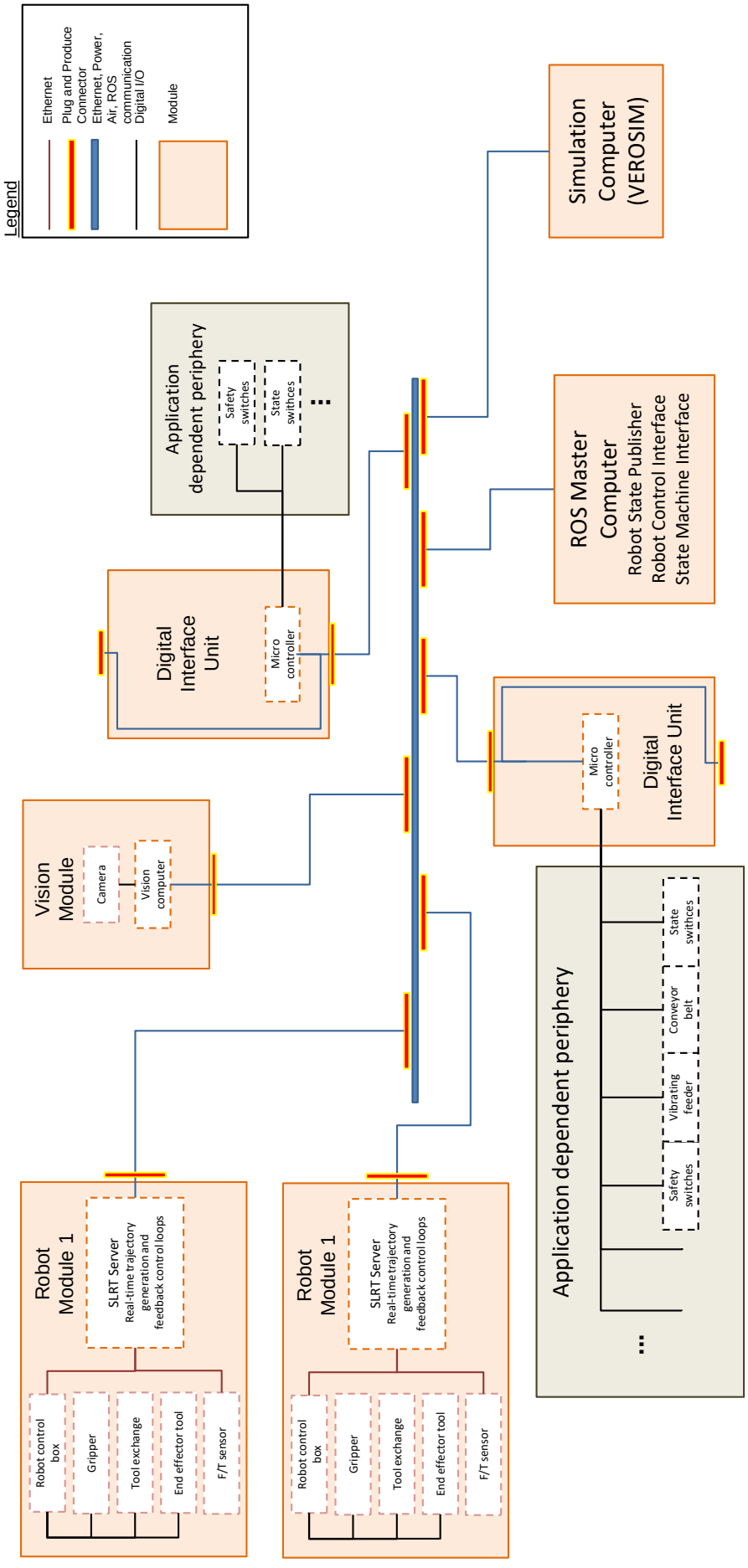

Figure 4: System overview of a reconfigurable robotic workcell with various software and hardware modules.

The developed server (denoted as SLRT Server in Figs. 2 and 4) accepts higher-level commands, e. g. action sequences, from the Robot Control Interface (see Section 4.3) and applies the implemented control strategies and commands to execute the required robot motion. To implement the server we used MATLAB Simulink Real-Time Target (SLRT) system as it provides a comprehensive user interface to program complex control algorithms that run in real time. The SLRT Server also broadcasts information about the state of the robot and hardware modules connected directly to the robot control box via Ethernet using custom UDP packets. This information is received by Robot State Publisher (see Section 4.2) running on ROS Master Computer and robot simulation system (see Section 4.5) running on a separate Simulation Computer.

4.1.1 TCP/UDP Converter

In this section we describe how the communication between UR10 robot and SLRT Server has been implemented. The UR10 robot controller provides the programming functionality via URScript, a scripting language developed by Universal Robots. A program written in URScript must run on the robot controller and nowhere else. We used URScript to write a program that can accept robot control data over the network and control the robot accordingly. Unfortunately, URScript language provides only TCP-based communication functions, which are not compatible with the SLRT system, because the latter only allows UDP communication. In order to make it possible for the two systems to communicate, we developed a program for the UR10 control box that receives data via UDP, sends the received data via TCP and vice versa. Thus the program accepts UDP data coming from the Simulink Real-Time Target (SLRT) Server and passes the received data to the robot controller via TCP communication protocol. It also receives data coming from the robot controller via TCP and re-sends the data to the SLRT Server via UDP. The schematics is shown in Figure 5.

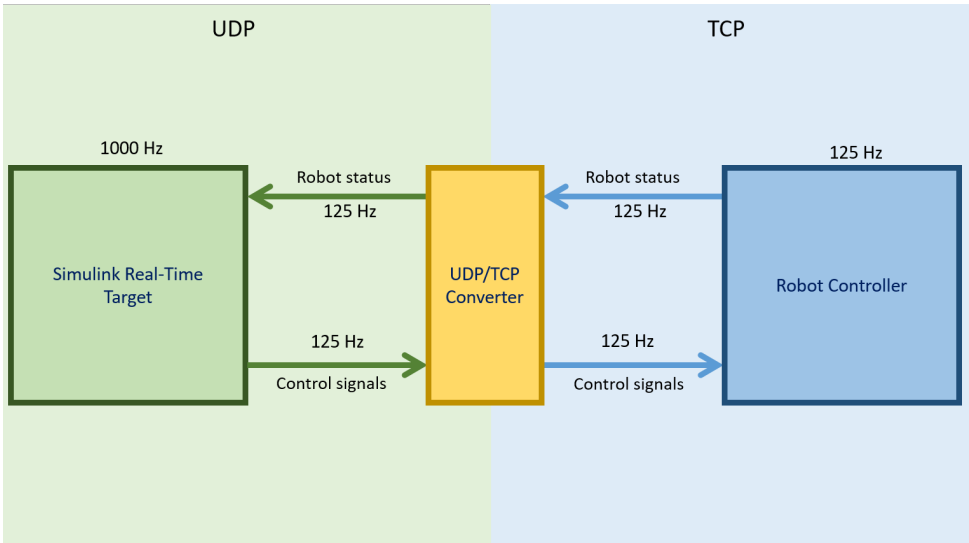

Figure 5: Block diagram showing the functionality of the UDP/TCP Converter. The converter runs on UR10 control box.

4.1.2 Robot control modes

The Simulink Real-Time Target server controls the robot by sending the desired joint positions to the robot via UDP. In order to achieve the desired robot motion, the server has to calculate the new desired joints at every sample time. Various feedback control and trajectory generation strategies were implemented on the server to meet the most common robot motion needs in the context of automated assembly. Besides trajectory generation and feedback control strategies, a mode that sets the robot into gravity compensation mode was also implemented. This functionality was used to implement the recording of complex robot movements via kinesthetic guiding. The recorded data are used to learn dynamic movement primitives (DMPs).

To start a motion according to the selected strategy, two packets should be sent to the SLRT Server in the following order:

1. A packet with the data necessary to calculate the new trajectory or to directly control robot joints.

2. A packet with the mode for the desired motion strategy.

These two packets are sent via two different ports.

Once the server receives the mode and the data necessary to execute the desired motion, it will start sending new reference joint positions to the robot. The currently available strategies and modes are listed in Table 1.

Table 1: Mode selection receiver

Mode selection	
Port	10000
Size	1
Data type	uint8
Motion strategy	**Mode byte value**
Trapezoidal velocity profile joint space trajectory	5
Minimum jerk positional trajectory & minimum jerk SLERP trajectory for orientation	13
Dynamic Movement Primitive in Cartesian Space	12
Dynamic Movement Primitive in joint space	11
Direct joint control	4
Admittance force control with DMP adaptation	16
Gravity compensation	101

- **Trapezoidal velocity profile joint space trajectory** is a control strategy that generates a trajectory between two joint configurations so that the joints reach the desired percentage of the maximum joint velocity at each joint throughout the motion [10]. The maximum velocity and acceleration of each joint is described in the documentation of UR10 robot.

- **Minimum jerk positional trajectory and minimum jerk SLERP for orientation** is a trajectory generation strategy used for Cartesian space motion. The positional part of the trajectory is calculated using minimum jerk principle [8], while the orientation part is calculated using **S**pherical **L**inear Int**erp**olation (SLERP) [3].

- **Dynamic Movement Primitive in joint or Cartesian space** is a trajectory generation strategy that can be used both for joint space control and Cartesian space control with slight modifications. Dynamic Movement Primitive (DMP) [5] is a combination of dynamical systems (second order differential equations) and provides convergence towards the desired goal configuration, robustness to perturbation, and indirect dependence to time. To send the desired DMP to the SLRT Server, it is only necessary to send the DMP parameters. The server then decodes the trajectory and controls the execution of the given trajectory. The Cartesian space DMP implementation follows the instructions provided in [11]. The current implementation allows using up to 184 radial basis functions per degree of freedom to approximate the desired movement. This limit corresponds to the maximum allowed package size (1472 bytes) of a UDP receiver provided by the Simulink Real-Time library. While the current limit was high enough in all our experiments, we plan to update the SLRT Server in the future to remove this limit.

- **Direct joint control** is a mode that simply forwards the received joint positions to the robot controller. It is up to the client to send a smooth trajectory to the SLRT Server, one trajectory sample at a time.

- **Admittance force control with Cartesian DMP adaptation** is the mode in which the robot follows the desired trajectory encoded as a Cartesian space DMP while minimizing the error between the reference force and the measured forces using the Force/Torque sensor attached to the robot end-effector. In this mode, the movement of the robot depends both on the sensed forces and the desired positions [1].

4.1.3 Output data

Besides sending the data to the robot controller, the SLRT Server also transmits robot module data to the network. These data are used by higher-level programs to perform safety checks and evaluate the executed trajectories and forces and torques. The data are broadcast at different ports to ease the programming of receivers at the other end. The data broadcast to the network are described in Appendix A.

4.2 Robot State Publisher

The robot state publisher is a ROS node, written in Python, that publishes the current robot states to topics using standard ROS messages. It runs on ROS Master Computer (see Fig. 4). To do this, it first reads the current robot states from the Simulink Real-time Target Server (SLRT Server) over UDP ports, structures the data using various ROS message types, and republishes them to ROS topics, which can be read by other ROS nodes. The topics broadly fit into the following categories:

- Robot general states

Robot general states:	**Minimum jerk positional trajectory and minimum jerk SLERP for orientation flags:**
joint_states	cart_lin_task_received_pulse
tcp_pose	cart_lin_task_finished
digital_outputs (GPIOs)	**Cartesian space DMP flags and data:**
digital_inputs (GPIOs)	cart_dmp_received_pulse
Force and torque sensor measurements:	cart_dmp_id
forces_sensor_space	cart_dmp_execute
forces_tool_compensated	cart_dmp_reset
Robot general flags:	cart_dmp_phase
received_control_mode	cart_dmp_finished
mode_sent_to_robot	**Joint space DMP flags and data:**
robot_idle	joint_dmp_received_pulse
running	joint_dmp_id
finish_successful	joint_dmp_execute
error	joint_dmp_reset
command_received_pulse	joint_dmp_phase
Trapezoidal velocity profile joint space trajectory flags:	joint_dmp_finished
joint_trap_vel_received_pulse	**Direct joint control flags:**
joint_trap_vel_finished	joint_direct_received_pulse

Table 2: Robot state publisher topics.

- Force/torque sensor measurements
- Robot general flags
- Direct joint control flags
- Trapezoidal velocity profile joint space trajectory flags
- Minimum jerk positional trajectory and minimum jerk SLERP for orientation flags
- Joint and Cartesian space DMP flags and data

Table 2 shows the topics that are published in more detail. The Robot Control Interface, which implements action servers described in Section 4.3, makes heavy use of the data published to these topics in order to check the current robot joint position states, the current sensor states (e. g. forces and torques), whether or not DMPs are correctly loaded for execution, etc.

It is worth nothing here that the topics described in Table 2 are specified as *relative* (as opposed to *global* or *private*) ROS names that may be associated with any given *namespace*. Thus, if only one UR10 robot exists in the workcell, the topics might be associated with the

`/ur10` namespace, and thereby be referred to as, for example, `/ur10/joint_states`. However, if more than one robot exists in the workcell, ROS namespacing functionality can be exploited by giving the robots separate namespaces, e.g. `/ur10_1` and `/ur10_2`. Their respective state topics can thereby be referenced separately as for example `/ur10_1/joint_states`, `/ur10_2/joint_states`, etc., in a seamless fashion.

4.3 Robot Control Interface

The main function of the ROS-based Robot Control Interface is to define robot control modes and provide the necessary data to the SLRT server to execute the desired robot movements. It is implemented as an ensemble of action servers that run on ROS Master Computer (see Fig. 4). Each control mode defines an action offered by the server. The benefit of using ROS provided action servers to trigger robot motion is the ability to cancel the request during execution and to get periodic feedback about how the request is progressing. If an action is preempted, the robot does **not** enter an emergency state and does **not** require any restart procedure. The client receives appropriate result messages in order to handle the preemption in its scheme and continue with another action if desired.

Action servers are created using *actionlib* package, which is a part of the *ros_base* meta-package and represents one of the core functionalities of ROS. This assures good support and compatibility for future upgrades of ROS. The package provides client interface in order to send requests to the SLRT server and read the feedback and result messages from the SLRT server and possibly other ROS nodes if required by future needs of the ReconCell system.

4.3.1 Robot Module Actions and Services

The Robot Module currently offers the following motions

- Joint space trapezoidal velocity profile trajectory
- Cartesian space input - joint space trapezoidal velocity profile trajectory
- Cartesian space minimum jerk linear and minimum jerk SLERP trajectory
- Joint space Dynamic Movement Primitive
- Cartesian space Dynamic Movement Primitive
- Cartesian space admittance force control with DMP adaptation

and control services

- Direct Joint Control
- Robot Controller Digital Outputs

4.3.2 Joint space trapezoidal velocity profile trajectory

This action provides an interface for controlling the robot using a trapezoidal velocity profile in joint space. It accepts a joint configuration and a relative speed value as a goal, forwards this to the Simulink Real-Time Target (SLRT) server and triggers the motion. The robot moves from the current joint configuration to the desired joint configuration taking into account the desired relative speed: $0 < speed\_percent \leq 1$, where the value of 1 corresponds to the maximum velocity of each robot joint. Each joint accelerates to the desired velocity with a constant acceleration value and then decelerates at the same rate to stop softly. During execution the SLRT server publishes the actual joint configuration of the robot as feedback. The final joint configuration is returned as the result after successful completion, preemption or failure. The action server node name and message description is provided in Table 3. The implementation on SLRT server is described in Section 4.1.

Table 3: Joint space trapezoidal velocity profile trajectory action name and message description.

joint_trap_vel_action		
JointTrapVel.action	*Variable*	*Description*
Goal	float32[] joints float32 speed_percent	6 joints $\in [0, 1]$
Result	float32[] joints	6 joints
Feedback	float32[] joints	6 joints

4.3.3 Cartesian space input - joint space trapezoidal velocity profile trajectory

This action provides an interface for controlling the robot using a trapezoidal velocity profile in joint space with Cartesian input. It accepts the desired task space pose (position and orientation) and relative velocity value as a goal, sends these data to the SLRT server and triggers the motion. The server calculates the desired joint configuration using inverse kinematics and moves the robot in the same manner as described in Section 4.3.2. During execution the server publishes the actual pose of the robot as feedback. The final pose is returned as the result after successful completion, preemption or failure. The action server node name and message description is found in Table 4.

4.3.4 Cartesian space minimum jerk positional and minimum jerk SLERP orientational trajectory

This action provides an interface for controlling the robot using minimum jerk positional trajectories and minimum jerk SLERP orientational trajectories. It accepts the desired Cartesian position and orientation of the robot's end-effector, where the desired orientation is specified as a unit quaternion, and the desired travel time towards the desired pose. This information is sent to the SLRT Server, which triggers and controls the real-time robot motion. The robot moves from the current position to the desired position in a straight line and along the shortest path on the orientation manifold, which is calculated by SLERP [3]. If the desired travel time causes joint velocities that would exceed the maximum robot speed, the movement is rejected before

Table 4: Cartesian input, joint space trapezoidal velocity profile trajectory action name and message description.

cart_trap_vel_action		
CartTrapVel.action	*Variable*	*Description*
Goal	float32[] position float32[] orientation float32 speed_percent	x,y,z quaternion $\in [0, 1]$
Result	float32[] position float32[] orientation	x,y,z quaternion
Feedback	float32[] position float32[] orientation	x,y,z quaternion

execution. During execution the server publishes the current pose of the robot as feedback. The final robot pose is returned as result after successful completion, preemption or failure. The action server node name and message description is found in Table 5.

Table 5: Cartesian space minimum jerk positional and minimum jerk SLERP orientational trajectory action name and message description.

cart_lin_task_action		
CartLinTask.action	*Variable*	*Description*
Goal	float32[] position float32[] orientation float32 desired_travel_time	x,y,z quaternion in seconds
Result	float32[] position float32[] orientation	x,y,z quaternion
Feedback	float32[] position float32[] orientation	x,y,z quaternion

4.3.5 Joint space dynamic movement primitive

This action provides an interface for controlling the robot using dynamic movement primitives (DMPs) in joint space [5]. The server accepts the desired DMP parameters as input and sends these data to the SLRT server, which triggers and controls the real-time robot motion. During execution the server publishes the actual joint configuration of the robot as feedback. The final joint configuration is returned as result after successful completion, preemption or failure. The action server node name and message description is found in Table 6.

Table 6: Joint space dynamic movement primitive action name and message description. See [5] for the meaning of parameters.

joint_dmp_action		
JointDMP.action	*Variable*	*Description*
Goal	float32 N	number of DMP kernel functions (N)
	float32 a_z	DMP parameter α_z
	float32 a_x	DMP parameter α_x
	float32 tau	DMP parameter τ
	float32 id	DMP ID
	float32[] goal	DMP goal g
	float32[] y0	DMP starting position
	float32[] dy0	DMP starting velocity
	float32[] c	DMP parameter c - kernel centers
	float32[] sigma	DMP parameter σ
	float32[] w1	DMP weights for joint 1 motion, N weights
	float32[] w2	DMP weights for jointjoint 2 motion, N weights
	float32[] w3	DMP weights for jointjoint 3 motion, N weights
	float32[] w4	DMP weights for jointjoint 4 motion, N weights
	float32[] w5	DMP weights for jointjoint 5 motion, N weights
	float32[] w6	DMP weights for jointjoint 6 motion, N weights
	float32[] tasktrans	task transformation matrix (default $\mathbf{I}$)
Result	float32[] joints	6 joints
Feedback	float32[] joints	6 joints

4.3.6 Cartesian space dynamic movement primitive

This action provides an interface for controlling the robot using dynamic movement primitives (DMPs) in Cartesian space [11]. The server accepts the desired DMP parameters and sends these data to the SLRT server, which triggers and controls the real-time robot motion. During execution the server publishes the actual pose of the robot as feedback. The final pose is returned as result after successful completion, preemption or failure. The action server node name and message description is found in Table 7.

Table 7: Cartesian space dynamic movement primitive action name and message description. See [11] for the meaning of parameters.

cart_dmp_action		
CartDMP.action	*Variable*	*Description*
Goal	float32 N	number of DMP kernel functions (N)
	float32 a_z	DMP parameter α_z
	float32 a_x	DMP parameter α_x
	float32 tau	DMP parameter τ
	float32 id	DMP ID
	float32[] goal	DMP goal g
	float32[] y0	DMP starting position
	float32[] dy0	DMP starting velocity
	float32[] c	DMP parameter c - kernel centers
	float32[] sigma	DMP parameter σ
	float32[] w1	DMP weights for motion in x, N weights
	float32[] w2	DMP weights for motion in y, N weights
	float32[] w3	DMP weights for motion in z, N weights
	float32[] w4	DMP weights for quaternion rotation (ω_i), N weights
	float32[] w5	DMP weights for quaternion rotation (ω_j), N weights
	float32[] w6	DMP weights for quaternion rotation (ω_k), N weights
	float32[] tasktrans	task transformation matrix (default $\mathbf{I}$)
Result	float32[] position	x,y,z
	float32[] orientation	quaternion
Feedback	float32[] position	x,y,z
	float32[] orientation	quaternion

4.3.7 Cartesian space admittance force control with DMP adaptation

This action provides an interface for controlling and adapting a trajectory encoded as a Cartesian space DMP by minimizing the error between a reference force and the measured forces using the Force/Torque sensor attached to the robot end-effector [1]. The server accepts the desired DMP, offset and force reference parameters and sends these data to the SLRT Server, which triggers and controls the real-time robot motion. The offsets can also be set to zero,

in which case admittance force control is executed on the defined DMP. During execution the server publishes the measured torques / forces and the robot pose as feedback. In case of successful completion the server returns the offsets corresponding to the executed movement and the current pose. The offsets can then be used in the next iteration to improve the reference tracking and task execution time. In case of failure or preemption just the robot pose is returned. The action server node name and message description is found in Table 8.

This mode has been tested but not yet fully integrated into ROS system. Therefore it is not mentioned among the published robot states in Table 2.

Table 8: Cartesian space admittance force control with DMP adaptation action name and message description. See [11] for the meaning of parameters.

cart_admit_dmp_action		
CartAdmitDMP.action	*Variable*	*Description*
Goal	float32 N	number of DMP and offset kernel functions (N)
	float32 a_z	DMP parameter α_z
	float32 a_x	DMP parameter α_x
	float32 tau	DMP parameter τ
	float32 id	DMP ID
	float32[] goal	DMP goal g
	float32[] y0	DMP starting position
	float32[] dy0	DMP starting velocity
	float32[] c	DMP parameter c - kernel centers
	float32[] sigma	DMP parameter σ
	float32[] w1	DMP weights for motion in x, N weights
	float32[] w2	DMP weights for motion in y, N weights
	float32[] w3	DMP weights for motion in z, N weights
	float32[] w4	DMP weights for quaternion rotation (ω_i), N weights
	float32[] w5	DMP weights for quaternion rotation (ω_j), N weights
	float32[] w6	DMP weights for quaternion rotation (ω_k), N weights
	float32[] tasktrans	task transformation matrix (default $\mathbf{I}$)
	float32[] c_o	offset kernel centers
	float32[] sigma_o	offset parameter σ_o
	float32[] w1o	position offset weights in x, N weights
	float32[] w2o	position offset weights in y, N weights
	float32[] w3o	position offset weights in z, N weights
	float32[] w4o	quaternion offset weights in i, N weights
	float32[] w5o	quaternion offset weights in j, N weights
	float32[] w6o	quaternion offset weights in k, N weights
	float32 N_f	number of force / torque reference kernel functions (N_f)
	float32[] c_f	kernel centers
	float32[] sigma_f	parameter σ_f
	float32[] w1f	force reference weights in x, N_f weights

	float32[] w2f	force reference weights in y, N_f weights
	float32[] w3f	force reference weights in z, N_f weights
	float32[] w4f	torque reference weights around x, N_f weights
	float32[] w5f	torque reference weights around y, N_f weights
	float32[] w6f	torque reference weights around z, N_f weights
Result	float32 N	number of DMP and offset kernel functions (N)
	float32[] c	offset kernel centers
	float32[] sigma	offset parameter σ_o
	float32[] w1o	position offset weights in x, N weights
	float32[] w2o	position offset weights in y, N weights
	float32[] w3o	position offset weights in z, N weights
	float32[] w4o	quaternion offset weights in i, N weights
	float32[] w5o	quaternion offset weights in j, N weights
	float32[] w6o	quaternion offset weights in k, N weights
	float32[] position	x,y,z
	float32[] orientation	quaternion
Feedback	Wrench forces	measured forces / torques
	float32[] position	x,y,z
	float32[] orientation	quaternion

4.3.8 Direct Joint Control

The Robot Module also offers *direct joint control*. This mode should generally **NOT** be used, since all common control principles are covered by other control modes. Only a user that designs his own controller requires this. The user takes over all responsibility for safety of the robot, the workcell and anyone or anything in proximity to the robot. This control mode is useful to enable direct control from our robot simulation system VEROSIM.

This mode is controlled by a ROS service, which accepts a boolean to turn this mode on or off. The service responds with a boolean that indicates the success of changing the mode and a string message for information, e. g. explanation of trigger failure. The robot must not be executing any other action for this mode to be successfully turned on. While it is on, all other action queries are rejected. The user must turn direct joint control off in order to again use other control modes. The service message description is found in Table 9.

After direct joint control mode has been turned on, the robot joint configurations must be sent to the SLRT Server directly. For security the SLRT Server is on a separate network, therefore any client that wants to control the robot directly must connect to that network. The SLRT Server accepts joints on a specific port as described in Appendix B and passes them to the robot controller.

4.3.9 Robot Controller Digital Outputs

The control box provided by UR10 robot offers 10 general purpose digital outputs. These outputs can be sent to arbitrary hardware components, e. g. grippers. The outputs are controlled over a ROS *set_ output* service, which is part of the Robot Control Interface and accepts an

Table 9: Direct joint control service name and message description.

joint_direct_switch		
std_srvs/SetBool.srv	*Variable*	*Description*
Request	bool data	true to turn on/ false to turn off
Reply	bool success string message	true if desired state achieved/ false otherwise description of failure

output number and a boolean value. These two values are sent to the SLRT Server which passes the boolean value to the selected hardware component via UR10 control box. The service responds with a boolean that indicates the success of setting the output and an informational message. The service message description is found in Table 10.

Table 10: Robot controller digital outputs service name and message description.

set_output		
Output.srv	*Variable*	*Description*
Request	uint8 outputNr bool value	$\{0, 1, ..., 9\}$ true to turn on / false to turn off
Reply	bool success string message	true if desired state achieved / false otherwise description of failure or success

In order to make it easier to control hardware that appears in the ReconCell system often, we designed additional services with more descriptive names. These services manage the same outputs as the *set_output* service, but do not require the user to remember which output number the hardware is connected to. The names and description of these services are in Table 11.

Table 11: Additional robot module services with descriptive names for specific hardware and their message description.

set_tool_exchange, set_gripper, set_linear_axis		
std_srvs/SetBool.srv	*Variable*	*Description*
Request	bool data	true / false
Reply	bool success string message	true if desired state achieved/ false otherwise description

4.4 State Machine Interface

The high-level programming of robot tasks is supported by a state machine framework called SMACH [2]. It is implemented as a separate ROS node running on ROS Master Computer.

The SMACH framework provides a set of Python libraries to easily create new states, connections between them, transfer variables between states and visualize the programmed state machine. A SMACH state machine is able to subscribe or publish to various topics in the ROS system and consequently also trigger robot movements and other action through Robot Control Interface. A simple state machine for moving the robot upon key-press is shown in Figure 6. This state machine is only given as an example to illustrate SMACH framework and is not part of the ReconCell system.

4.5 Simulation and Visual Programming Module

VEROSIM simulation system is an additional module in the ReconCell software architecture that enhances the workcell with advanced robot simulation capabilities. It provides a visual programming environment to enable user-friendly programming of robot tasks, enables workcell modeling and adaptation, dynamic simulation, collision detection, motion planning, sensor simulation, etc. The details on VEROSIM are explained in detail in Deliverable **D4.1**.

The integration of VEROSIM into ReconCell software architecture is shown in Fig. 7. At this stage communication is one directional and intended for visualization of current robot joint positions. VEROSIM connects to the system in two different ways. Firstly, VEROSIM can read robot data via Simulink Real-Time Target server using UDP protocol. The second option for communication uses the implemented Robot State Publisher node and works by subscribing to the appropriate ROS topic to acquire information about the state of the robot (see Table 2).

4.6 Vision Module Interface

Since the vision module heavily depends on the selected cameras, which are task specific, it necessarily contains a specialized software component to capture data from the selected cameras. The output of the Vision Module to the ROS framework is the result of the image processing task, e. g. object pose and/or other measurements. Camera images are sent over ROS only for GUI visualization purposes and even then, bandwidth should be used conservatively.

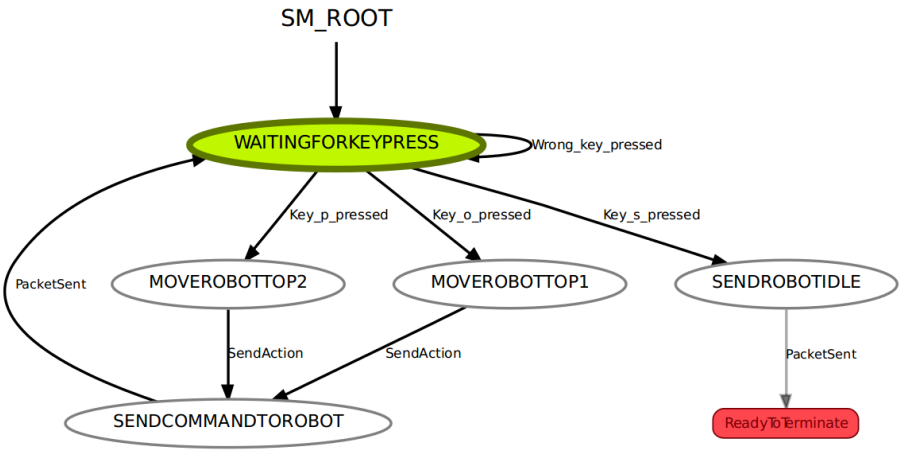

Figure 6: A simple state machine created with SMACH.

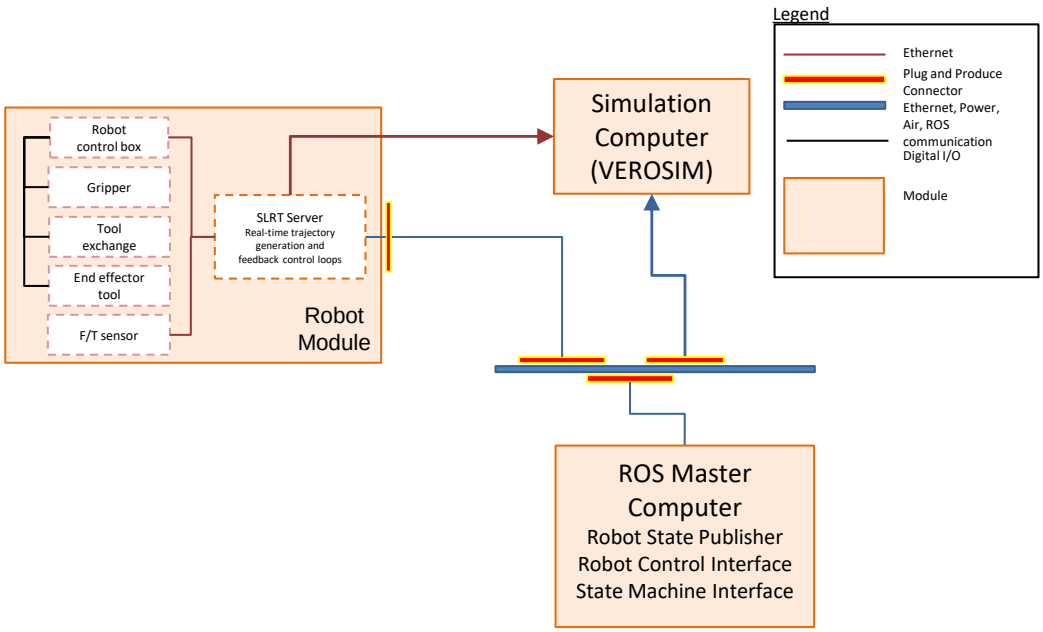

Figure 7: Current VEROSIM integration. Compared to Fig. 4 we have a direct connection between SLRT Server and Simulation Computer. This grants the simulation environment the ability to bypass ROS communication and access real-time data.

The Vision Module ROS interface will be similar to the one implemented in Robot Control Interface node. It will offer actions and services for different tasks where synchronous communication is needed. The Vision Module will execute code written using OpenCV libraries and/or other software packages and return the result using standard ROS message types.

4.7 Digital Interface

The digital interface module serves as a bridge between the ROS network and the application dependent periphery. Common industry components often connect to programmable logic controllers (PLCs) for control, but in ReconCell we offer the digital interface for these standard components over ROS, which runs on a micro controller, e. g. Raspberry Pi. The interface follows the standard introduced in Section 4.3.9 for robot controller digital outputs. Components like *passive linear unit*, *reconfigurable jig module* and other periphery use this digital interface to release breaks, trigger grippers, etc.

References

[1] F. J. Abu-Dakka, B. Nemec, J. A. Jørgensen, T. R. Savarimuthu, N. Krüger, and A. Ude. "Adaptation of manipulation skills in physical contact with the environment to reference force profiles". In: *Autonomous Robots* 39.2 (2015), pp. 199–217.

[2] J. Bohren. *SMACH (State MACHine), a task-level architecture for rapidly creating complex robot behavior*. `http://wiki.ros.org/smach`. Accessed: 2016-10-31.

[3] E. B. Dam, M. Koch, and M. Lillholm. *Quaternions, Interpolation and Animation*. Tech. rep. DIKU-TR-98/5. Denmark: University of Copenhagen, 1998.

[4] *Destaco Automatic & Manual Tool Changers: QC-30*. `http://www.destaco.com/tool-changers-effectors/QC-30`. Accessed: 2016-10-31.

[5] A. J. Ijspeert, J. Nakanishi, H. Hoffmann, P. Pastor, and S. Schaal. "Dynamical Movement Primitives: Learning Attractor Models for Motor Behaviors". In: *Neural Computation* 25.2 (2013), pp. 328–373.

[6] M. Jonsson and G. Ossbahr. "Aspects of reconfigurable and flexible fixtures". In: *Production Engineering Research and Development* 4.4 (2010), pp. 333–339.

[7] N. Krüger, A. Ude, H. G. Petersen, B. Nemec, L.-P. Ellekilde, T. R. Savarimuthu, J. A. Rytz, K. Fischer, A. G. Buch, D. Kraft, W. Mustafa, E. E. Aksoy, J. Papon, A. Kramberger, and F. Wörgötter. "Technologies for the Fast Set-Up of Automated Assembly Processes". In: *Künstliche Intelligenz* 28.4 (2014), pp. 305–313.

[8] A. Piazzi and A. Visioli. "Global minimum-jerk trajectory planning of robot manipulators". en. In: *IEEE Transactions on Industrial Electronics* 47 (2000), pp. 140–149.

[9] M. Quigley, B. Gerkey, and W. D. Smart. *Programming Robots with ROS: A Practical Introduction to the Robot Operating System*. Sebastopol, CA: O'Rilley Media, 2015.

[10] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo. *Robotics, Modelling, Planning and Control*. Springer, 2009.

[11] A. Ude, B. Nemec, T. Petrič, and J. Morimoto. "Orientation in Cartesian space dynamic movement primitives". In: *IEEE International Conference on Robotics and Automation (ICRA)*. Hong Kong, 2014, pp. 2997–3004.

Appendices

A Simulink Real-Time Target output data

Robot general states		
Port number		15000
Total size in bytes		360
Detailed packet content description	**Data type**	**Vector size**
Actual robot joints	double	6
Desired robot joints	double	6
Actual robot joint velocities	double	6
Desired robot joint velocities	double	6
Actual robot TCP position	double	3
Actual robot orientation quaternion	double	4
Measured forces in sensor space	double	6
Measured forces after tool compensation	double	6
Desired digital outputs	double	1
Digital inputs state	double	1
Robot general flags		
Port number		15001
Total size in bytes		7
Detailed packet content description	**Data type**	**Vector size**
The received control mode	uint8	1
The mode sent to the robot	uint8	1
Robot idle	uint8	1
Error	uint8	1
Running	uint8	1
Command received pulse	uint8	1
Finish succesful	uint8	1
Joint direct control flags		
Port number		15002
Total size in bytes		1
Pulse when a package was received	uint8	1
Joint control with trapezoidal velocity flags		
Port number		15003
Total size in bytes		2

Trapezoidal velocity trajectory received pulse	uint8	1
Trapezoidal velocity trajectory finished	uint8	1
Cartesian minimum jerk trajectory		
Port number		15004
Total size in bytes		2
Cartesian trajectory received pulse	uint8	1
Cartesian trajectory finished	uint8	1
Cartesian DMP flags		
Port number		15005
Total size in bytes		48
DMP trajectory received pulse	double	1
Loaded DMP's ID	double	1
DMP execute	double	1
DMP reset	double	1
DMP phase	double	1
DMP finished	double	1
Joint space DMP flags		
Port number		15006
Total size in bytes		48
Detailed packet content description	**Data type**	**Vector size**
DMP trajectory received pulse	double	1
Loaded DMP's ID	double	1
DMP execute	double	1
DMP reset	double	1
DMP phase	double	1
DMP finished	double	1

Table 12: Simulink Real-Time Target output data description.

B Simulink Real-Time Target input data

Mode selection		
Port number		10000
Total size in bytes		1
Detailed packet content description	**Data type**	**Vector size**
Desired control mode	uint8	1

Joint direct control		
Port number		20000
Total size in bytes		48
Detailed packet content description	**Data type**	**Vector size**
Desired joint positions	double	6
Cartesian space DMP parameters		
Port number		25000
Total size in bytes		192
Detailed packet content description	**Data type**	**Vector size**
N	double	1
a_z	double	1
a_x	double	1
τ	double	1
g	double	7
y_0	double	7
dy_0	double	6
Cartesian DMP parameters - c		
Port number		25001
Total size in bytes		1472
Detailed packet content description	**Data type**	**Vector size**
c	double	184
Cartesian DMP parameters - σ		
Port number		25002
Total size in bytes		1472
Detailed packet content description	**Data type**	**Vector size**
σ	double	184
Cartesian DMP parameters - weights (1 - 6)		
Port number		25003 - 25008
Total size in bytes		1472 each
Detailed packet content description	**Data type**	**Vector size**
w_1 ... w_6	double	184
Cartesian DMP execution flags		
Port number		25009

Total size in bytes		16
Detailed packet content description	**Data type**	**Vector size**
Execute	uint8	1
Reset	uint8	1
Cartesian DMP phase stopping flag		
Port number		25010
Total size in bytes		1
Detailed packet content description	**Data type**	**Vector size**
Phase stopping	uint8	1
Joint space DMP parameters		
Port number		25100
Total size in bytes		176
Detailed packet content description	**Data type**	**Vector size**
N	double	1
a_z	double	1
a_x	double	1
τ	double	1
g	double	6
y_0	double	6
dy_0	double	6
Joint space DMP parameters - c		
Port number		25101
Total size in bytes		1472
Detailed packet content description	**Data type**	**Vector size**
c	double	184
Joint space DMP parameters - σ		
Port number		25102
Total size in bytes		1472
Detailed packet content description	**Data type**	**Vector size**
σ	double	184
Joint space DMP parameters - weights (1 - DOF)		
Port number		25103 - 25108
Total size in bytes		1472 each
Detailed packet content description	**Data type**	**Vector size**

w_1 ... w_{DOF}	double	184
Joint space DMP execution flags		
Port number		25109
Total size in bytes		16
Detailed packet content description	**Data type**	**Vector size**
Execute	uint8	1
Reset	uint8	1
Joint space DMP phase stopping flag		
Port number		25110
Total size in bytes		8
Detailed packet content description	**Data type**	**Vector size**
Phase stopping	uint8	1
Adaptation offset - c		
Port number		27000
Total size in bytes		800
Detailed packet content description	**Data type**	**Vector size**
c	double	100
Adaptation offset - σ		
Port number		27001
Total size in bytes		1472
Detailed packet content description	**Data type**	**Vector size**
σ	double	184
Adaptation offset - weights (1 - 7)		
Port number		27002 - 27008
Total size in bytes		1472 each
Detailed packet content description	**Data type**	**Vector size**
w_1 ... w_7	double	184
Reference Force and Torque profile parameters - c		
Port number		28000
Total size in bytes		1472
Detailed packet content description	**Data type**	**Vector size**
c	double	184

Reference Force and Torque profile parameters - σ		
Port number		28001
Total size in bytes		1472
Detailed packet content description	**Data type**	**Vector size**
σ	double	184
Reference Force and Torque profile parameters - weights (1 - 6)		
Port number		28002 - 28007
Total size in bytes		1472 each
Detailed packet content description	**Data type**	**Vector size**
w_1 ... w_6	double	184
Joint trapezoidal speed profile parameters		
Port number		26000
Total size in bytes		56
Detailed packet content description	**Data type**	**Vector size**
Desired joint positions	double	6
Desired speed percent (1 being 100%)	double	1
Joint trapezoidal speed profile parameters		
Port number		27000
Total size in bytes		64
Detailed packet content description	**Data type**	**Vector size**
Desired cartesian position	double	3
Desired cartesian orientation in quaternions	double	4
Desired travel time	double	1
Desired digital outputs		
Port number		11000
Total size in bytes		8
Detailed packet content description	**Data type**	**Vector size**
Desired GPIO value	double	1
End effector tool data		
Port number		12000
Total size in bytes		8
Detailed packet content description	**Data type**	**Vector size**
Tool translation offset	double	3

Tool rotation in quaternions	double	4
Tool mass center point	double	3
Tool weight (in kg)	double	1
Sensor force offset	double	3
Sensor torque offset	double	3

Table 13: Simulink Real-Time target input data description.