



Share4.0

Pilot Action Report

WP3 – DT 3.2.1.3

Version 1  
Company – Oswald Bratu

Date 30.11.2022

\\fhbgld.at\Home_FB\FB\fbtkremsner\ProfileData\Desktop\IMPROVE_Logos PPs\IMPROVE_Logos PPs\PP3_PROFACTOR\PROFACTOR_logo.jpg

# Table of contents / Inhalt

[Table of contents / Inhalt 2](#_Toc124173555)

[Executive Summary 3](#_Toc124173556)

[1 Introduction 4](#_Toc124173557)

[2 System Design 5](#_Toc124173558)

[3 Elaboration of System Design 6](#_Toc124173559)

[3.1 OPC UA based agents 7](#_Toc124173560)

[3.1.1 Semantic elements of the OPC UA data model 8](#_Toc124173561)

[3.1.2 Implementing skills of an OPC UA based agent using SCXML state charts 9](#_Toc124173562)

[3.1.3 Implementing Configurations for an OPC UA based agent in SCXML 11](#_Toc124173563)

[4 Cloud System for Workflow Transition 12](#_Toc124173564)

[4.1 The web platform ‚https://share40.profactor.at’ 12](#_Toc124173565)

[4.2 Menu ‘Workflow-> AgentTypes‘ 13](#_Toc124173566)

[4.3 Menu ‘Workflow->Workflow Agents Blockly’: 15](#_Toc124173567)

[4.4 Export Code - (using Menu ‘Workflow->Workflows’) 23](#_Toc124173568)

[4.5 Preparing the local Environment for Starting the Program Code 24](#_Toc124173569)

[4.6 Starting the Code 27](#_Toc124173570)

[4.7 Using Jupyter Notebooks 28](#_Toc124173571)

# Executive Summary

Share4.0 aims to improve the collaboration of key players for research and innovation through new forms of cooperation and practicable work processes. In this way, direct and long-term oriented pilot projects with a high degree of effectiveness are activated and implemented. This is done based on two selected fields of work:

* Industrial assistance systems
* Resilient, sustainable production systems.

Overall, the challenge of improving the lack of cooperation (in particular: consistency, strategy, resources, utilization, excellence) will be met by means of a high-quality bundle of service support.

As target groups, SMEs, research facilities and universities, business support organizations (tech parks, business development, etc.) and regional/local authorities will be involved in the project processing and its follow-up activities. The project processing and implementation will be carried out with all project partners in the whole program area. The planned change will be achieved mainly through the following main outcomes:

1. Working base including governance model for Smart Industry Network SK-AT (Organizational Handbook)

2. Two pilot projects of excellence

3. Strategy and action plan 2021-2027

4. Memorandum of Understanding

These main results, defined as new products and services for research and innovation, provide a significant benefit for the target groups through the high degree of implementation in the project and beyond the project duration, thus making a significant contribution to an innovative, smart SK-AT region.

This deliverable document summarizes the main concepts and software implementation to for the SHARE 4.0 Pilot Action use cases in the field of industrial, intelligent assistance systems.

# Introduction

This document shows how the Pilot Action for Industrial, Intelligent assistance systems has been realized end executed. In order to realize the Pilot Action and already developed framework has been used and adapted for the specific use case in the SHARE 4.0 project.

The main goal of the pilot action is to have two collaborative robots which are executing the same workflows in two different locations. In this case one robot is located in Austria and the other is located in Slovakia. In order to realize the target it is necessary that both collaborative robots have identical skills so that they are not hindered by physical limitations. Second, in the mean of data sharing economy a cloud-system has to be created in order to share and access the knowledge of the robots to ensue the “knowledge transfer” including parameters like grasping points for a certain object and the “workflow” data which is process which was created in one location to be used also elsewhere.

Overall, the demonstration scenario looks as following:

* A robotic process is realized (workflow) or parameterized on robot (A)
* The relevant workflow / parametrization data is uploaded and synchronized on a cloud / REST service
* A robot at a different location (SK) can access the data and replicate the behaviour

Ein Bild, das Tisch enthält.

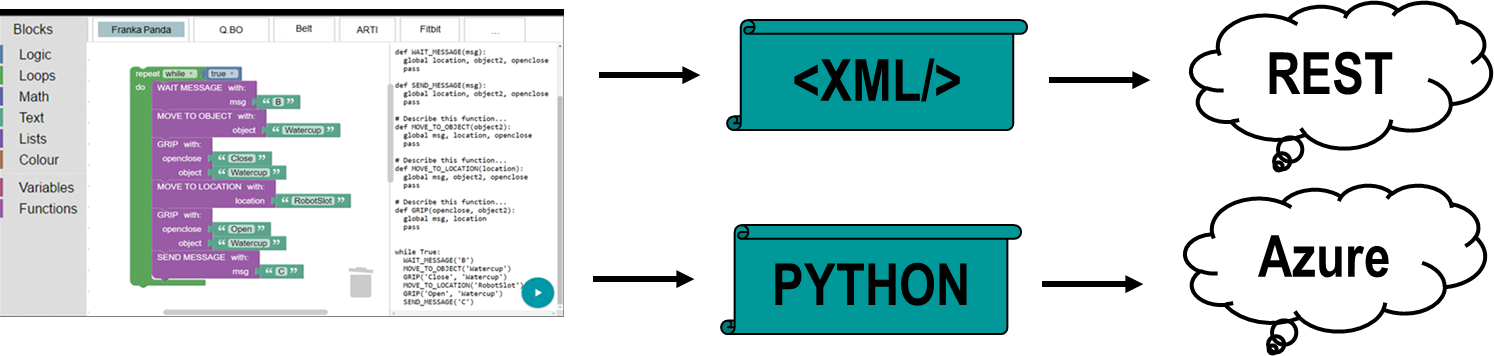
Automatisch generierte Beschreibung

# System Design

This chapter describes the overall realize and design of the pilot action. It will give insights into the development of the cloud system and interaction between the two collaborative robots and the installation and teaching of the operative tasks of the robot which at the end results into the shared workflow.

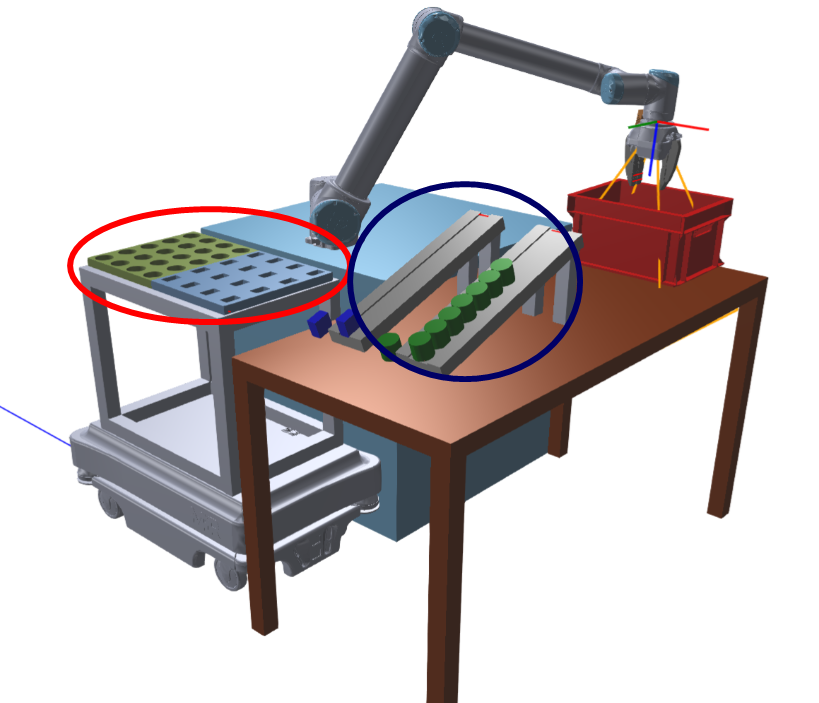
**System setup:**

* “Near” identical hardware setup at both locations:
  + Robot system with parallel gripper / or suction gripper
  + Work-cell with “sorted-parts” storage (gravity chutes) and commissioning tray
* Both systems run XRob including PlugBot & ROBxTASK extensions
* Blockly-Editor for ROBxTASK workflows
* ROBxTASK runtime environment for OPC UA
* REST-Server (e.g. FAST-API, Flask) to host the Blockly-XML-Code or SCXML-Code
* Or Std-Cloud-Service to share Blockly-generated Python code.



**Demonstration sequence**

* In area (A) a workflow is being developed and configured
  + Parts are grasped from presorted storages (fixed locations)
  + Parts are commissioned onto tray storage (different filling patterns)
* The workflow is encoded in a) Blockly, or b) SCXML
* The workflow is exported to a) XML or b) executable code (Python)
* The workflow is cloud-synchronized a) REST, or b) Std-Cloud System
* In area (SK) data access via cloud is triggered
* The workflow is downloaded and interpreted
* The robot system replicates the same workflow,  
  but maybe based on different hardware (e.g. gripper, robot)

NOTE: the configuration of robot-locations, calibration etc. is specific  
to the area (A) or (SK) and not affected.

# Elaboration of System Design

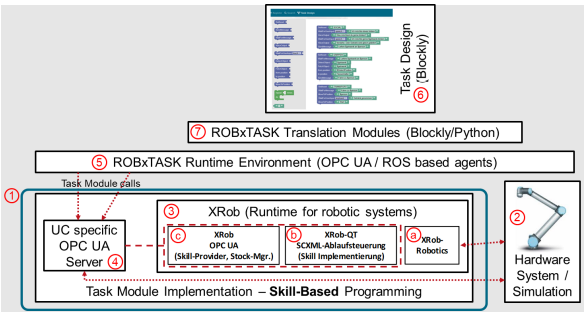
Following the the system design overview, this section describes the implementation approach towards SHARE 4.0 runtime-environment modules – including their dependencies – to enable workflows outside a ROS environment.

The proposed and realized approach includes the XRob framework [6], a framework to enable setup and configuration of robotic processes and a skill-based programming interface based on OPC UA [7].

Following figure depicts the resulting architecture, where the runtime-environment module (task module implementation – see (1)) directly interacts with the robotic environment (2) using OPC UA communication.

This runtime module includes two main elements. Firstly, the XRob runtime (3) provides software modules to trigger functionalities of robot systems (via the robot controller (a)), to configure automation of robot system and integrated peripherals (b), and to expose certain XRob-functionalities as configurable/callable methods of a linked OPC UA Server (c). Secondly, an OPC UA Server (4), tightly linked to the XRob programming interfaces, acts as a main interface towards the runtime environment and the simulation environment. This OPC UA Server exposes all functionality implemented by lower-level software and hardware (real-world) towards the runtime environment (5) conforming with the defined Task-Modules in SHARE 4.0.

The execution of workflow tasks (individual process steps) modelled using the Blockly-based task designer (6) and translated to executable code using translation modules (7), can then be triggered by the runtime environment using the provided OPC UA interface



Following this architecture, the upcoming subsections focus on the implementation of OPC UA based agents (including their Task-Modules / Skills) and the realized interfaces to interact with sensors, actors and robots. It is worth mentioning that the identical interfaces can be used to both access real system setup agents.

## OPC UA based agents

The XRob framework provides an application programming interface API (C++ based) that realizes the skill-based programming interface described in the previous works [7] and [8]. These collection of software modules enables the configuration and setup of an OPC UA Server, including a data model that consists of the semantic main elements as follows: Skills (Task-Modules), Configurations, Variables and Messages. All afore mentioned elements will be described briefly in upcoming subsections.The generation of the OPC UA Server data model is performed automatically, based on e.g. implemented Skill logics. As of current, two convenience modules for implementing Skill logics are available in XRob (also see Figure 2): a) LibXRobOPCUA – enables automatic Skill logic generation based on XRob process recipes [6] and provides a stock-management library to administrate the state of parts stocks (e.g. relevant for commissioning tasks), and b) LibXRobQt – provides a means of Skill logic implementation based on SCXML [9] finite state machines. Additionally, XRob provides a C++ API to enable Skill implementation by requiring a minimum level of OPC UA domain knowledge [8].

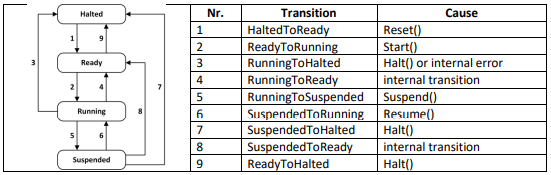
Using the mentioned software libraries, Skill (Task-Modules) were developed to enable control of actors, sensors and robots based on OPC UA data modelling and communication. The upcoming sections will introduce the key-concepts and give examples of the used methods.

### Semantic elements of the OPC UA data model

The convenience modules introduced earlier, build on semantic elements within OPC UA data models that are introduced in the following.

**Skills (Task-Modules)**

A skill represents an abstraction of a capability or function of a hardware or software component. Its definition origins from the concept of skill-based programming. The abstraction ensures that the mapped function can be used as generically as possible in different systems. Structurally, a skill forms a function call that can have both transfer parameters and return parameters. Skill execution internally follows a finite state machine (FSM) whose structure is related to the standards of OPC UA Programs.



It has to be pointed out, that Skills can also be part of a more complex Skill definition. This means one Skill can invoke existing Skills as a sub-skill (hierarchically) to foster reuse of certain generic / repeatedly used functionalities.

**Configurations**

A configuration enables the storage of typically static data that are necessary for the correct functioning of a Skill.

**Variables**

Variables consist of one or more parameters that can be continuously written during the execution of a skill and can be read by clients.

**Messages**

In certain situations, or process steps, user interaction may be required. To enable this, messages are introduced. These structures allow messages to be fetched and answered based on different user roles. In addition to the reply option, further parameters can also be transferred when replying. For the implementation of OPC UA based agents in ROBxTASK, mainly Skills, Configurations and Variables will be used.

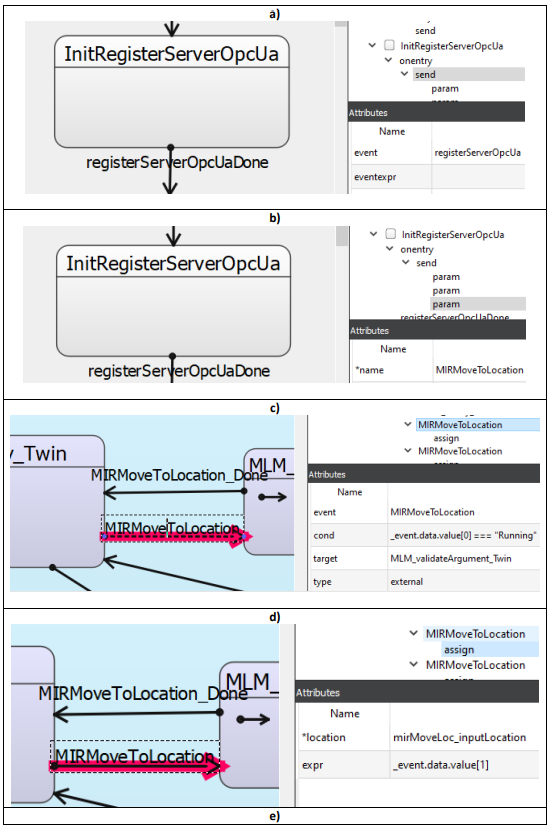
### Implementing skills of an OPC UA based agent using SCXML state charts

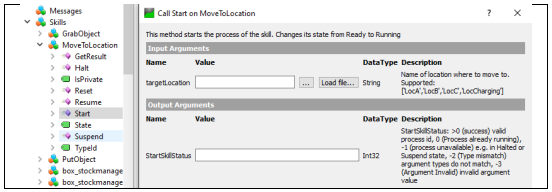
With the existing ServiceScxml as a part of LibXRobQt, XRob already provides a module for executing state charts according to the SCXML standard [12]. The module uses the QT framework to link ScxmlStateMachines with QtEvents and implements the skill-based programming interface according to [7].

Using predefined QtEvents together with specified configuration strings (defined format), OPC UA server structures for skills, configurations, variables and messages can be generated from the SCXML state machine at execution time. The configuration strings have different formats for the individual elements of the PlugBot interface. The following core elements are generally valid:

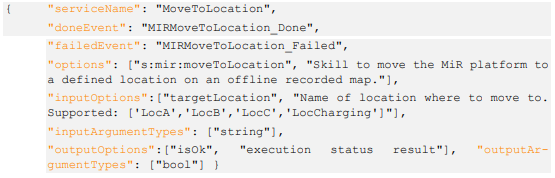
* Name, Description and Type-ID to uniquely identify the semantic element (see 3.1.1.1).
* Event name to describe an event that is triggered in the SCXML state chart once an interaction with these elements takes place via an OPC UA client. For example, when a skill is called.
* Further events which can report the successful or faulty processing of the “reaction” to a trigger event (e.g. skill start).
* Description of the parameters / arguments including data type and verbal description.

The next series illustration shows an example for registering the “MoveToLocation” skill on the OPC UA server. This skill is part of the MIR-Platform OPC UA based agent. A) Skills are registered by sending the event “registerServerOpcUa”. B) The exact skill specification is sent as a “param” node with the event “registerServerOpcUa”. The name of the param node determines the event name that is triggered when the skill is called via OPC UA. C) In the SCXML state chart, a state transition can be triggered by this event. D) The input parameters of the skill are included in the event and can be assigned to an SCXML variable upon the state transition is triggered. E) After the SCXML initialisation, the skill is visible on the created OPC UA server.

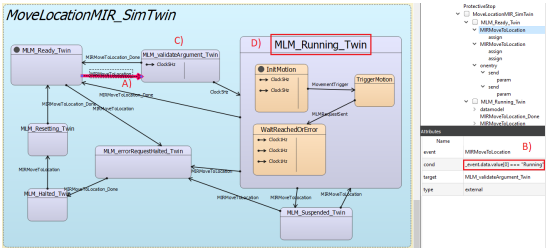




Along with the registration step for Skills, a configuration parameter needs to be passed that is specified in JSON format:



It should be noted here that the configuration string also contains the description of the skill arguments, their types and the verbal description.The ServiceSCXML can also be configured in such a way that if a skill is called via OPC UA, the skill status is also given with the event. In this way, it is then possible to implement the states "Running", "Ready", "Halted", "Suspended" with the help of SCXML logic. An example of this is shown in the following illustration for the skill "MoveToLocation" (example for the MIR Platform in simulation).

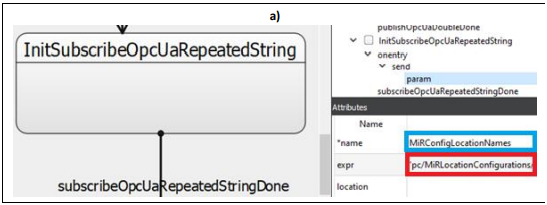


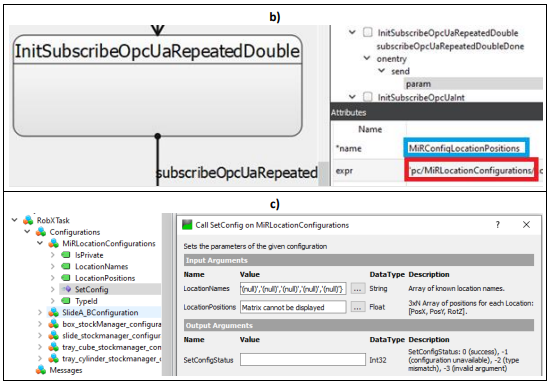
In the "MoveToLocation" transition within the transition condition (see B), the state (here: "Running") that is to be assumed is queried. The subsequent SCXML states (see C and D) perform the validation of skill input arguments and the actual skill logic (here MLM\_Running\_Twin).

### Implementing Configurations for an OPC UA based agent in SCXML

Similarly, to the implementation of skills, Configurations can be described in SCXML and thus generated to be used by OPC UA based agents. Configurations are registered using QTEvents of the format “subscribeOpcUa”. The keyword “Repeated” denotes an array containing values of the specified “SimpleDatatype” (e.g. integer, float, double, and string – types). Following Figure depicts a sequence to setup a Configuration consisting of two parameters of different type as follows.

* MiRConfigLocationNames: subscribeOpcUaRepeatedString ➔ array of strings
* MiRConfigLocationPositions: subscribeOpcUaRepeatedDouble ➔ array of double values





# Cloud System for Workflow Transition

## The web platform ‚https://share40.profactor.at’

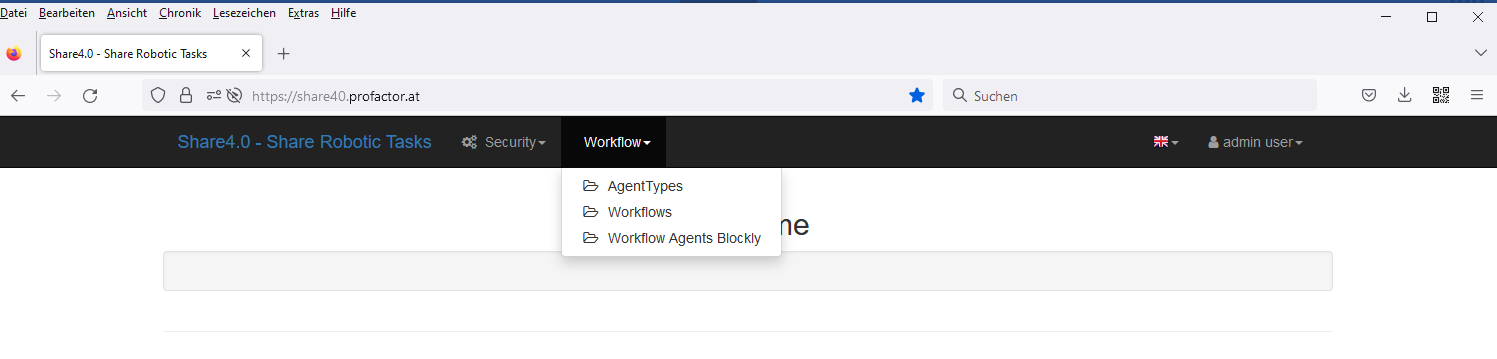
The prototype of the web platform is currently available for project users at url: ‚https://share40.procator.at‘. A ‘not admin – user’ is available in this share40 web environment. Thus, anyone can log in with

user: share40

password: share40$project

HINT: this service will end in December 2022

After login the relevant menu entries can be found below of the main menu ‘Workflow’:



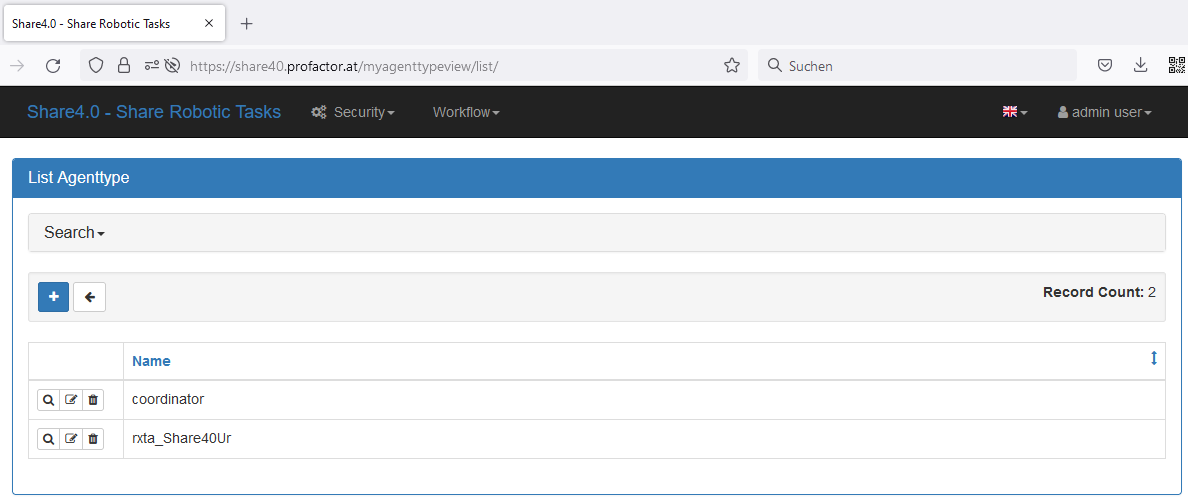
*HINT: On the remote pc we used the FireFox browser and added some bookmark-menu-points and saved the login password -data. So you can immediately use the browser to build robotic programs using blockly.*

Some words about ‘Workflows’:

The system can also be used to develop robotic workflows using multiple robotic devices. The idea is, that a ‘agent type’ represents a special robotic device (type) and enables to call the skills of this agent type by an individual agent which can be built in the platform by using a Google Blockly based editor. When using workflows, one device can send a message to another device and thus a multistep process ca be built. For a workflow a ‘coordinator’ role may also be useful to start the workflow.   
But mostly we don’t need workflows or a coordinator. So, for the project it’s ok, if a workflow contains only one agent – which then can call the appropriate skills of the corresponding agent type.

## Menu ‘Workflow-> AgentTypes‘

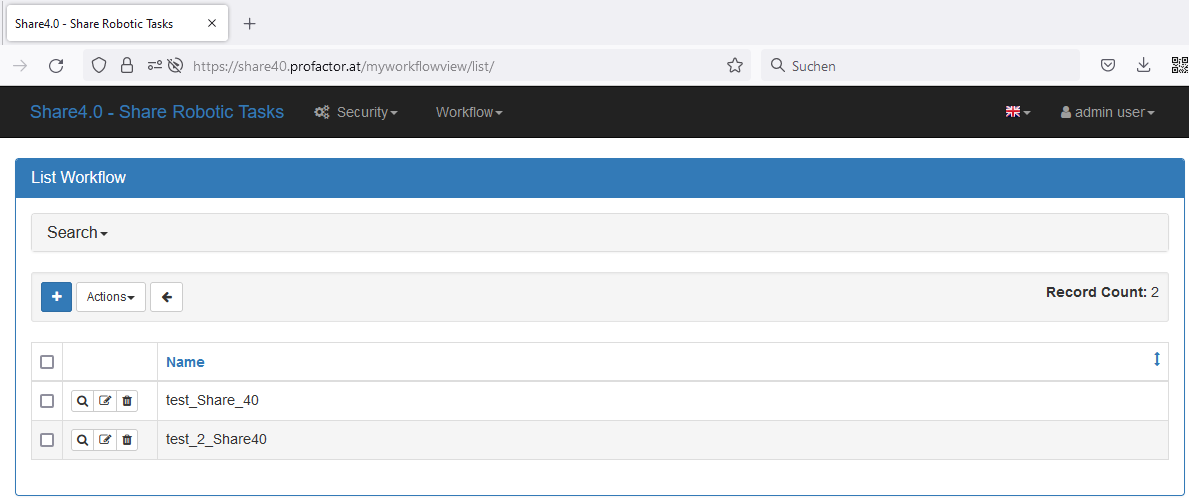
Here you can find only 2 agent types. *coordinator* and *rxta\_Share40Ur*.



Behind in the platform, some code was registered for any concrete agent type to describe the skills that are available for this agent type when building a concrete software agent using the blockly editor. Here new agent types could be registered. For the project we only need ‘rxta\_Share40Ur’ which stands for the abilities of a Universal Robot arm (UR5, UR10, …). So, you don’t need any functionality within this form/menu. Please don’t change things here.

**Menu ‘Workflow->Workflows’ - Overview**

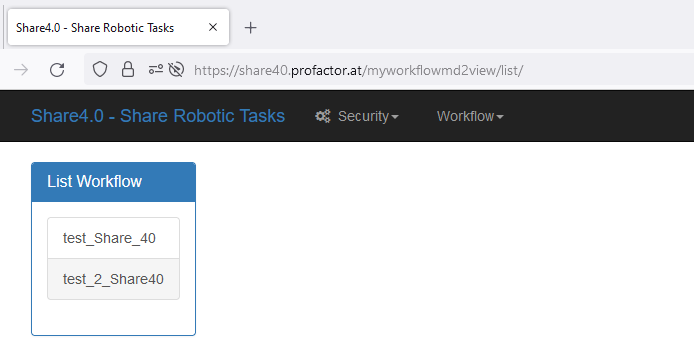
Two initial workflows were added for testing purposes.



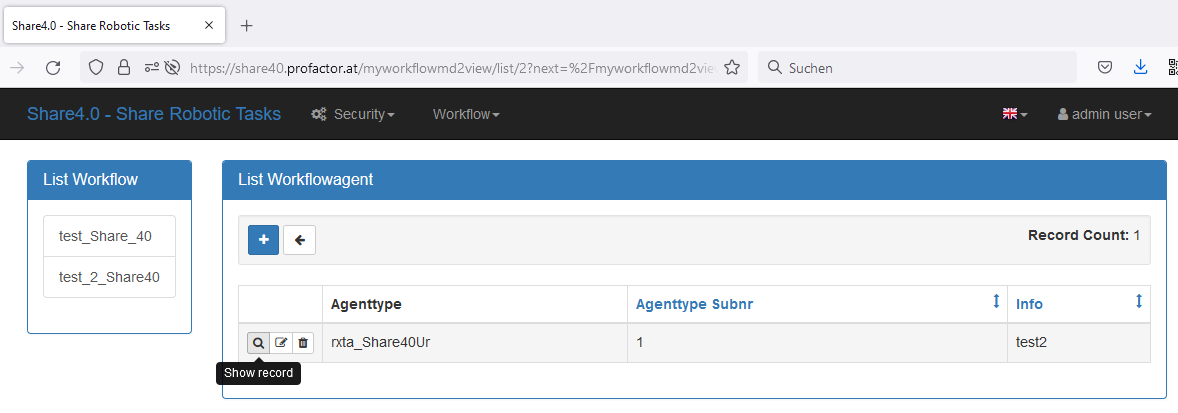
Feel free to add additional ones later (and add one agent (01) per workflow). Later you need this form to ‘download the agent-python code’ for your blockly - so we discuss some more features of this form later.

## Menu ‘Workflow->Workflow Agents Blockly’:

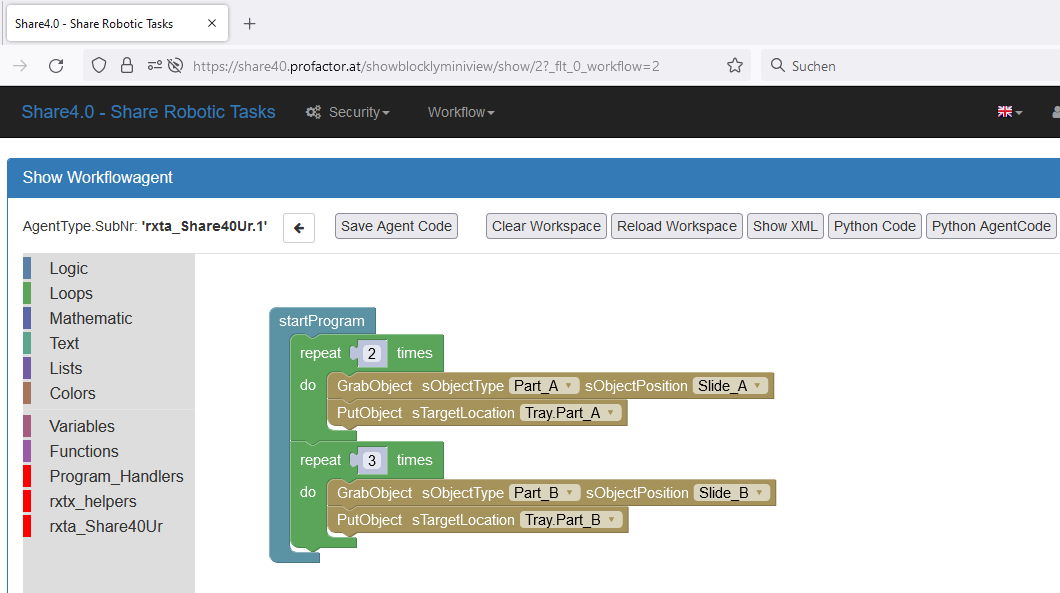
Here you could add multiple agents to a concrete workflow. Usually we only need one agent which already was added in the samples. Thus, you can immediately edit the code for an UR agent.   
(1) Choose one ‘Workflow’ on the left side. E.g. ‘test\_2\_Share40’



(2) Now you can see that the list of agents of this workflow only contains one ‘rxta\_Share40Ur’ agent. (remember we don’t need to build a real workflow spanning above multiple robots – so only one agent is needed)  
Click on ‘magnifier’ symbol on the left (Show Record) to open the Blockly Editor.



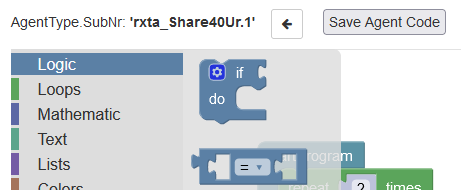
Now you can use the Blockly Editor to write your robotic program.



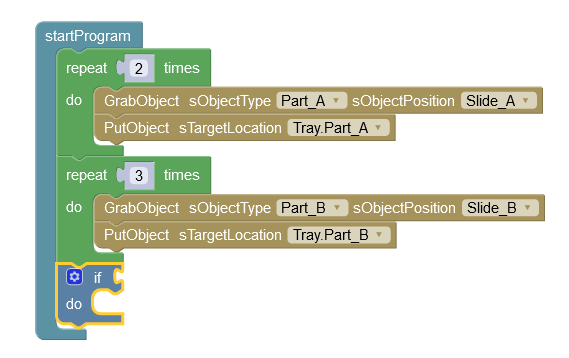
Relevant Menu-Items on Top:

* **Save Agent Code**: Saves the current Blockly Code into the database.
* **Clear Workspace**: Clears all Blockly Code on the dashboard.
* **Reload Workspace**: Reloads the current dashboard by using the last saved version of the database.
* **Show XML**: Shows the internal Google Blockly representation for the current Blockly Code.
* **Python Code**: Shows what python code would be generated only for the visable Blockly Code.
* **Python AgentCode**: Shows what code will be generated finally for this concrete agent.  
  *HINT: maybe it’s sometimes useful to copy this code to your external programming environment, instead of always exporting the total zip-code if there are only small program changes.*

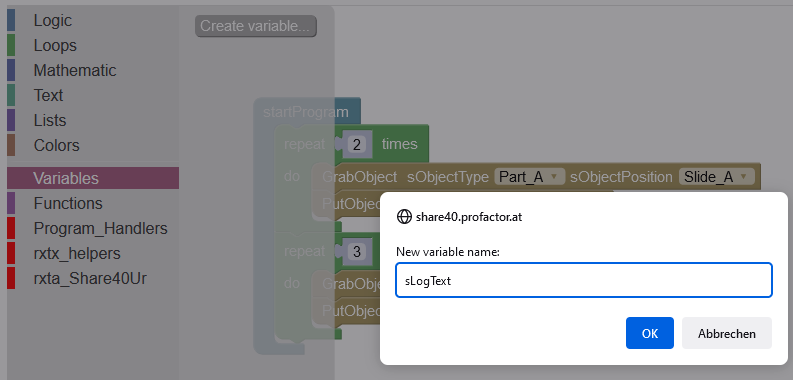
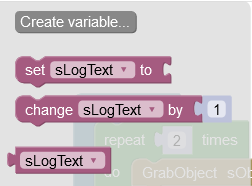
On the left side of the google blockly editor, you can see some entries that hold useful graphical constructs to build your program using the editor. You therefore have to click on an entry in the container list and then an overlay appears where you can choose an individual entry.



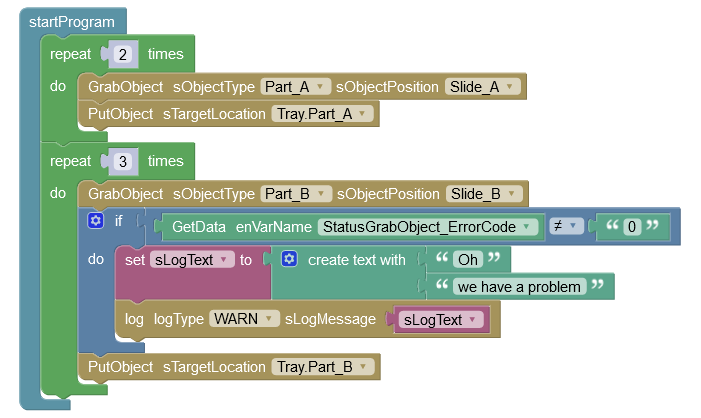
To modify the program, e.g. left klick on the ‘if’ entry and hold the mouse button and then ‘drag it’ on the dashboard and release it on an appropriate position.

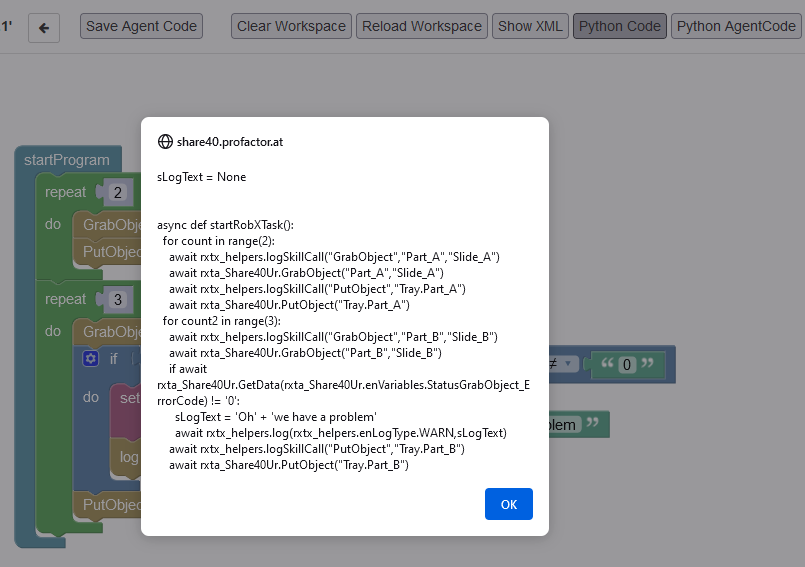


The first 6 entries on the left **(‘Logic’, ‘Loops’, ‘Mathematic’, ‘Text’, ‘Lists’, ‘Colors’**) contain Google Blockly standard functionality to build programs.   
**‘Variables’** allow you to define individual program variables, which may be useful for generating more complex code. After you add a new variable you get 3 additional blocks for programming. A set block, a change block and a value block to use the content of the variable.

You need Variables e.g. if you want to use some Blockly functions which return a value (e.g. a block which returns the value of a sensor). These blocks then have a nose on the left side. You cannot insert these blocks with noses on the left directly inside a normal block’s list. You must first insert some construct which can handle a return value. E.g. An ‘If’ with a condition cell. Or as said a ‘set variable’ block to save the return value to an internal variable, which then can be used later in programming.

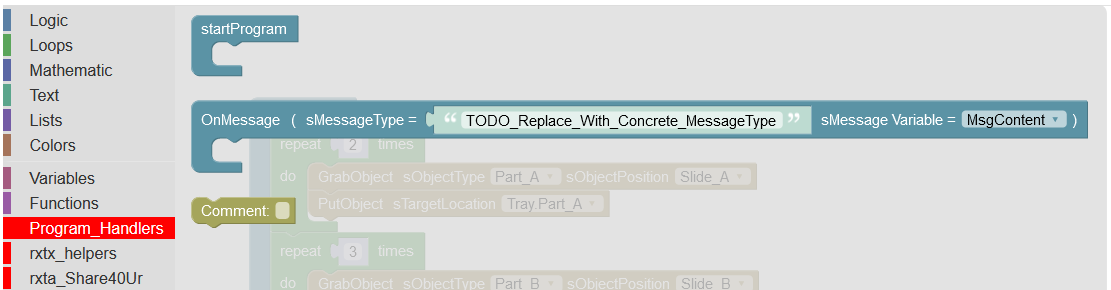
  
*HINT: In real program code any variable you declare is ‘global’ within the whole program code!*



**‘Functions’** is a more advanced feature which allows you to build internal functions in your code. You won’t need this mostly.

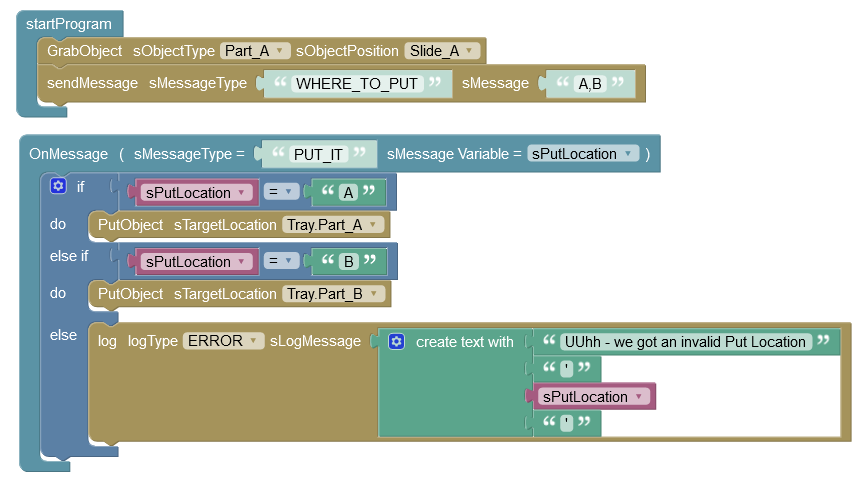
The next 3 menu items are more share40 specific and need more attention:

**‘Program\_Handlers’** contain the main entry points for your code. You must at least use one of ‘startProgram’ or ‘OnMessage’. Usually startProgram has to be used!



The first thing you should do when programming on an empty dashboard, is to drag such a ‘startProgram’ block inside. Then put all your individual program code in the blocks list. When you start the agent later the ‘startProgram’ handler is called immediately. Thus, the behaviour of this construct is similar to a usual ‘main()’ function.

Only if you need real workflows spanning above multiple agents the ‘OnMessage’ handlers makes sense. This then can be used if such an agent shall not do anything until he e.g. gets a message of another agent in the workflow.   
But maybe also the usage of both program handlers makes sense. E.g. if one agent starts something – then sends a message to another agent and finally waits for a message to receive back, before executing another skill. Such a sample is shown in the following sample code.



But usually your individual Code should contain only a ‘startProgram’ Block from ‘Program\_Handlers’.

**‘rxtx\_helpers’** contains some helpful runtime environment blocks. E.g.

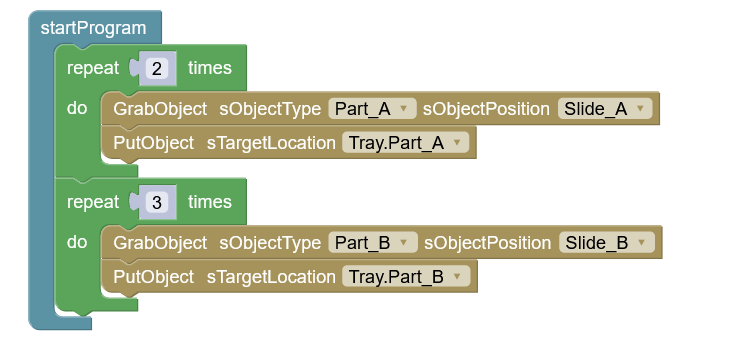
* **getEnvironmentVariable** – to get the content of an environment variable.
* **log** - to log some text to the logfile (and the distributed logging environment).
* **sendMessage** - to send a message of a give type and content to some receiver agents, that are currently running and waiting (have an OnMessage block) of the same message type.
* **Sleep** - to pause for some seconds.
* **Stop** - to kill the agents process.   
  *HINT: if an agent code only contains a startProgram it will be finished after the last command of the block list was executed. But as soon as there is (additional) at least one OnMessage block in your agent code – the agent will never end. You should then use the ‘stop()’ block to define the end of your agent’s lifetime.*
* **toNumber** - to convert a string to a numeric value (usually numeric blocks are blue).
* **toString** - to convert a numeric value to a string (usually string blocks are green).

**“rxta\_Share40Ur”** – contains the skills and some additional functions for the UR agent type.

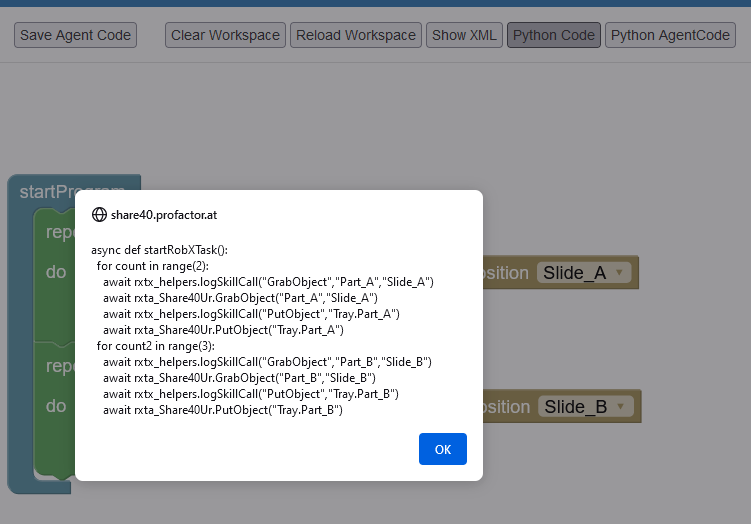
* **GetData** – can be used to get the content of an OPC UA variable of the UR – Needed to get the content of an OPC UA variable that holds details on the execution of a skill.   
  <Skill>\_State … contains the current state of the skill. (e.g. running = 1,…)  
  <Skill>\_Error\_Code … if the skill execution was faulty this contains an additional error code   
  <Skill>\_StatusMessage … if the skill execution was faulty you may find details here
* **GrabObject** – executes the grab skill for the UR by using the specified arguments given for ObjectType (Part\_A or Part\_B) and ObjectPosition.(Slide\_A or Slide\_B or Slide.Part\_A or Slide.Part\_B)
* **PutObject** - – executes the put skill for the UR by using the specified arguments given for TargetLocation.(Tray.Part\_A or Tray.Part\_B)
* **SetData** – would allow to set OPC UA variables to a given value – this is unsupported.
* **getResult()** – returns the result of the last Skill execution (GrabObject, PutObject) as a ‘boolean string’ = ‘True’ if it was ok, ‘False’ otherwise.
* **getResultBool()** – returns the result of the last Skill execution (GrabObject, PutObject) as a Boolean value – can be used immediately e.g. in an if statement.   
  *HINT: We decided to have skills not return values directly in blockly – because then this would need ‘left nose’ blocks and force the programmer to at least put some if block or set variable around it. This way if someone is interested in return values of skills he then can use getResultBool() after the skill call. This allows Blockly code usually to remain simpler.*
* **OnVariableChange** – could be used to implement a handler which would be called if some OPC UA variable’s value was changed. You usually don’t need this.

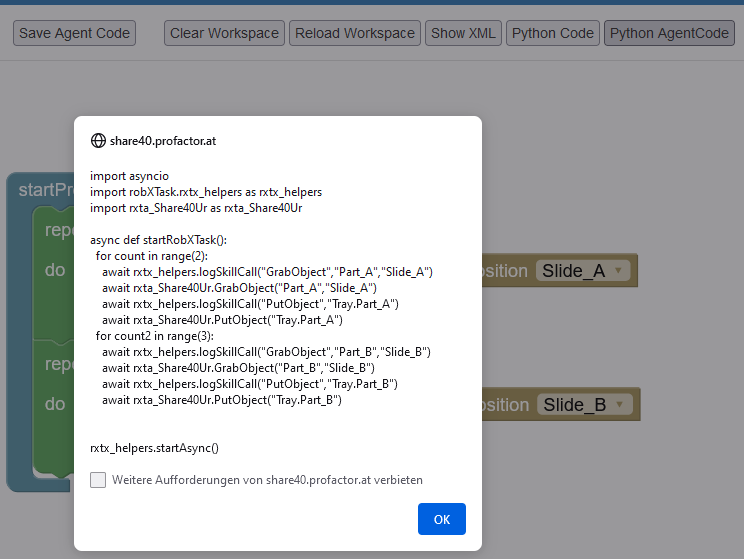
So in short:

You usually drag a ‘startProgram’ handler inside your dashboard. And then put some of the skills from rxta\_Share40Ur’ inside. Maybe embedded in e.g. a loop.



Don’t forget to press ‘Save Agent Code’ after changes!   
To get a Feeling how the final python code will look like press ‘Python Code’.

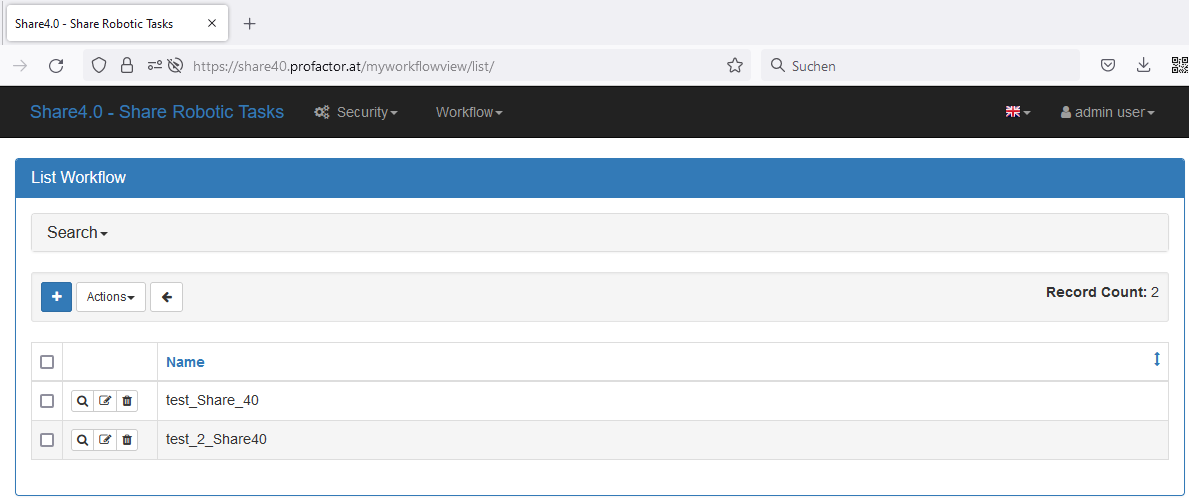
  
‘Python AgentCode’ shows how this python code will be embedded in the runtime environment.



Now we want to run this code. Please ensure again, that you pressed ‘Save Agent Code’ .  
Then go to Workflow->Workflows – find your workflow in the list and export the code as described below.

## Export Code - (using Menu ‘Workflow->Workflows’)

To export the workflow code you generated using Google Blockly, please change again to the workflow overview.

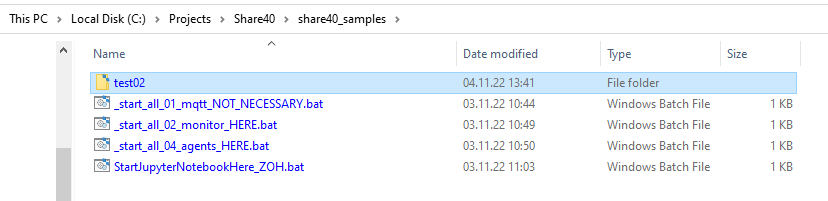


|  |  |
| --- | --- |
| To export the program code for a special workflow, please click on the ‘magnifier’ symbol (Show record) to the left of the relevant entry in the workflow list. |  |
| In the form that appears now, please click on the ‘Export Workflow’ button and finally press the OK button that is displayed later, to start the download of your program code. |  |

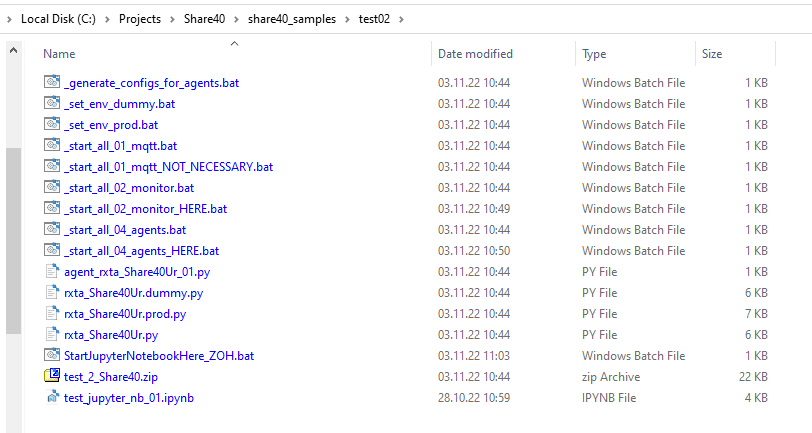
## Preparing the local Environment for Starting the Program Code

Now extract all files from the zip file into the same folder. (e.g. C:\Projects\Share40\share40\_samples\test02)

Then copy from c:\projects\share40\sample\_code all files additionally to the subfolder.



Use these ‘batch-files’ instead of the extracted batch files (the extracted batch files are more ‘general’ and currently must be adapted for the concrete python environments to be used).



On the remote PC miniconda and a python environment called ‘share40’ was installed. A mosquito mqtt-service was installed and is running – it is used for messaging between agents and also supports distributed logging for agents. A module ‘robxtask\_helpers’ is installed in the python environment to support configuration and startup of agents, and some helper tools for e.g. monitoring workflows were also installed.

We will now describe in short each of these files:

* **\_generate\_configs\_for\_agents.bat** - can be used to generate configuration files for agents and helper modules to change some default setting. Is not needed here – ignore it.
* **\_set\_env\_dummy.bat** – copies all xxx.dummy.py files to xxx.py. This replaces the agent-type stubs with a dummy implementation where any skill call is ‘simulated’ only by waiting for 2 seconds and then continuing the program. This program can be used if you want to test a ‘workflow’ without having the concrete hardware available.
* **\_set\_env\_prod.bat** – copies all xxx.prod.py files to xxx.py. This replaces the agent-type stubs with a ‘opc ua’ calling ‘productive’ implementation where any skill call is executed on the opc ua server.   
  *HINT: The default files xxx.py in the folder are the productive ones!*
* **\_start\_all\_01\_mqtt.bat** - would start the MQTT-Server in docker environment. Not needed here, because the MQTT-Server is installed as a mosquito - service on the remote pc.   
  *You can delete this file.*
* **\_start\_all\_01\_mqtt\_NOT\_NECESSARY.bat** - Not needed here – only to remind you that the MQTT-Server is installed as a mosquito - service on the remote pc.   
  *You can delete this file.*
* **\_start\_all\_02\_monitor.bat** - would start the monitor application to see the distributed logging of the agents. But this is a default version – so it won’t start on this pc. Use the individual start\_all\_02\_monitor\_HERE.bat instead.   
  *You can delete this file.*
* **\_start\_all\_02\_monitor\_HERE.bat** – please execute this file at first to have the monitor application available BEFORE starting the agents.   
  Then change to the monitor-webserver (in firefox) – to ‘reverse path’ and choose an auto-refesh of ‘5 seconds’.
* **\_start\_all\_04\_agents.bat** – starts all available agents of this workflow at once. But this is a default version – so it won’t start on this pc. Use the individual start\_all\_04\_agents\_HERE.bat instead.   
  *You can delete this file.*
* **\_start\_all\_04\_agents\_HERE.bat** – starts all available agents of this workflow at once. Uses python from the concrete virtual environment share40 on this pc.
* **agent\_rxta\_Share40Ur\_01.py** – this python code contains the code for the blockly program you developed in the editor. It is started by ‘\_start\_all\_04\_agents\_HERE.bat’. It imports the module for the agent type stub ‘rxta\_Share40Ur.py’. (which can in concrete be either a ‘dummy’ or a ‘prod’ version by calling \_set\_env\_dummy.bat or \_set\_env\_prod.bat before)
* **rxta\_Share40Ur.dummy.py** – contains a dummy implementation of the agent type, where any skill call is ‘simulated’ only by waiting for 2 seconds and then continuing the program.
* **rxta\_Share40Ur.prod.py** – contains the productive implementation of the agent type, where any skill call is executed on the opc ua server (default: opc.tcp://localhost:4860). This may of course be either a ‘dummy opc ua’ server, or an opc ua server which communicates to a digital twin environment. Or a ‘real opc ua’ server which is connected to a concrete UR. This also enables testing without real hardware available.
* **rxta\_Share40Ur.py** – is the current agent type file = the concrete version used by agent\_rxta\_Share40Ur\_01.py. Its content is either the same as rxta\_Share40Ur.dummy.py or rxta\_Share40Ur.prod.py, depending on the last \_set\_env\_dummy.bat or \_set\_env\_prod.bat call. Per default this file has the same content as rxta\_Share40Ur.prod.py.   
  –> the productive version of the agent type is the default version.

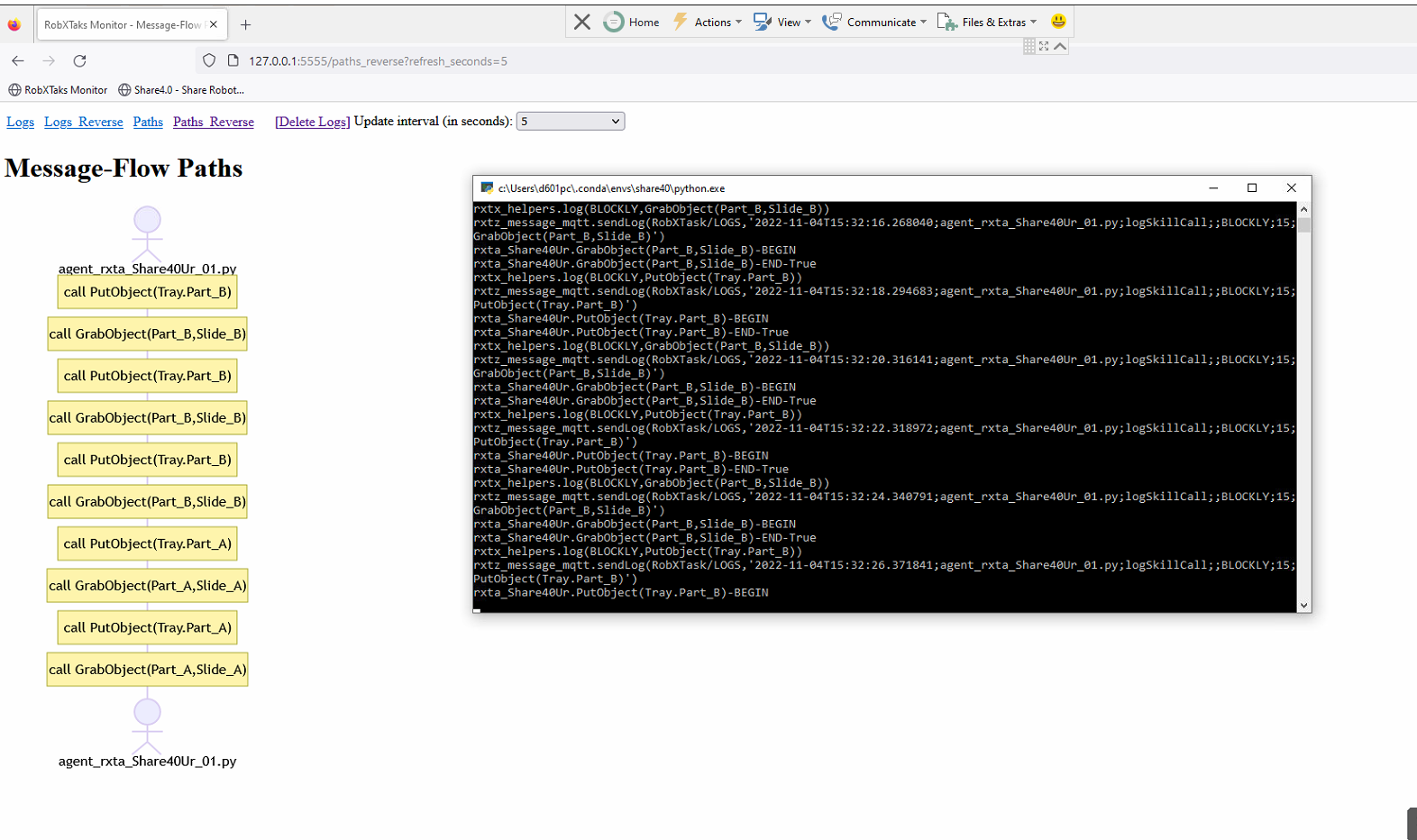
## Starting the Code

First start

***\_start\_all\_02\_monitor\_HERE.bat***

This starts a web-monitor-application (link is also available as a bookmark in Firefox)  
Goto ‘Path\_reverse’ view and activate an update-intercal of 5 seconds.  
The robxtask library uses mqtt which was installed as a mosqitto broker on the pc to send logs – thus the monitor application can listen for that logs. (for multiple Runs call ‘Delete Logs’ to get a fresh window again)

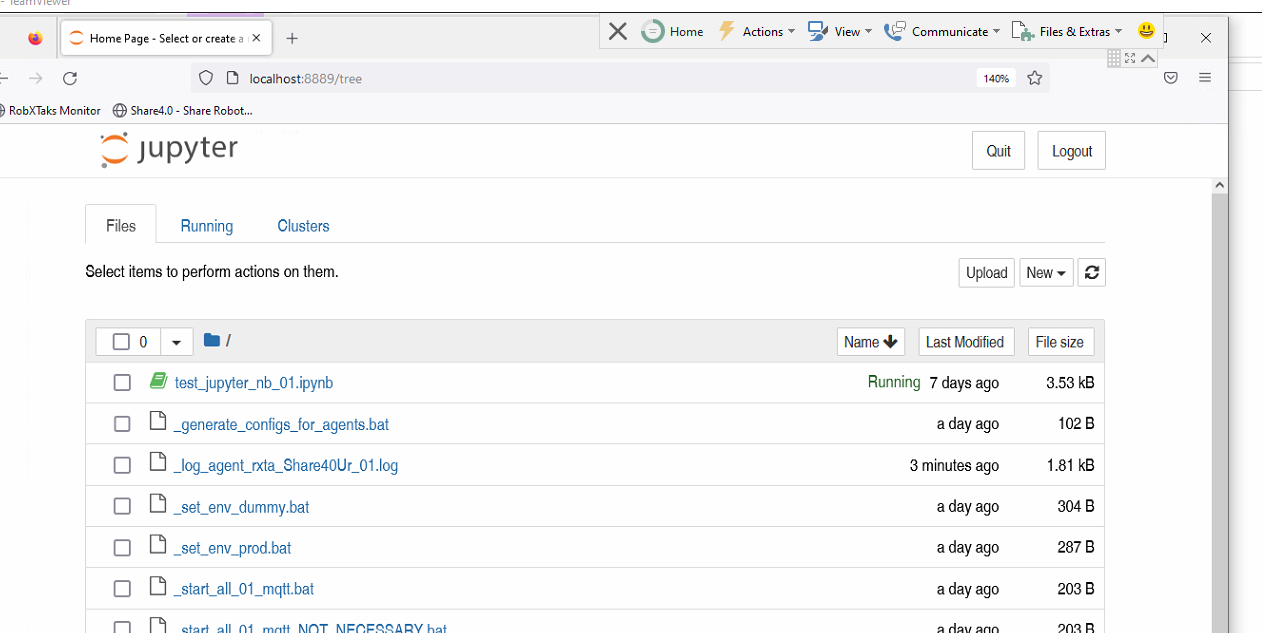
For testing without robotic hardware (dummies) start   
***\_set\_env\_dummy.bat***  
Or – to call skills using the opc ua server call   
***\_set\_env\_prod.bat***  
To start the agent (agent\_rxta\_Share40Ur\_01.py) click:   
***\_start\_all\_04\_agents\_HERE.bat***Now you should see something in the monitor webview ~ the same as in the terminal window…



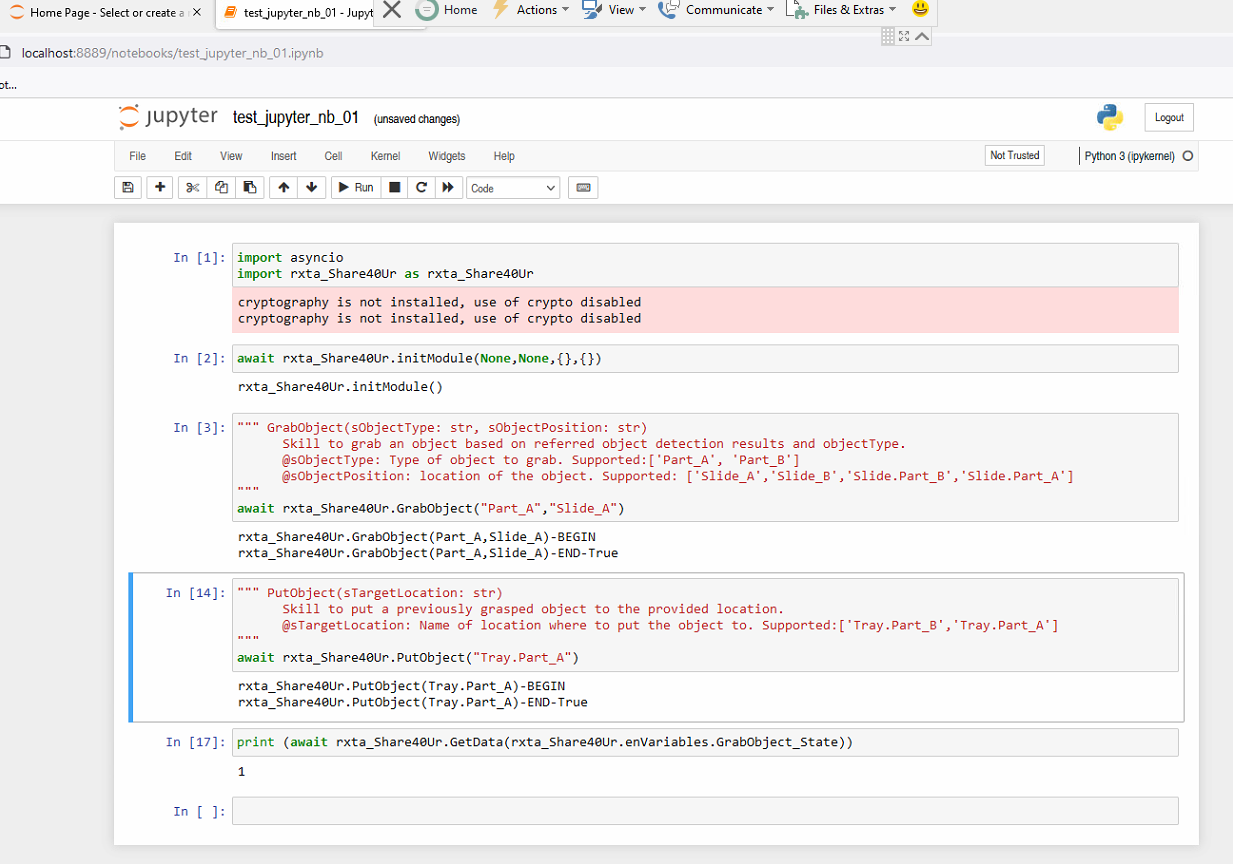
The terminal window should close after the last code statement.   
For multiple runs call ‘Delete Logs’ to get a fresh window again before you start it again.

## Using Jupyter Notebooks

The folder ‘test02’ currently also contains a sample how to call the skills by using Jupyter notebooks.  
Execute ‘StartJupyterNotebookHere\_ZOH.bat’ Now a new webbrowser window should open:



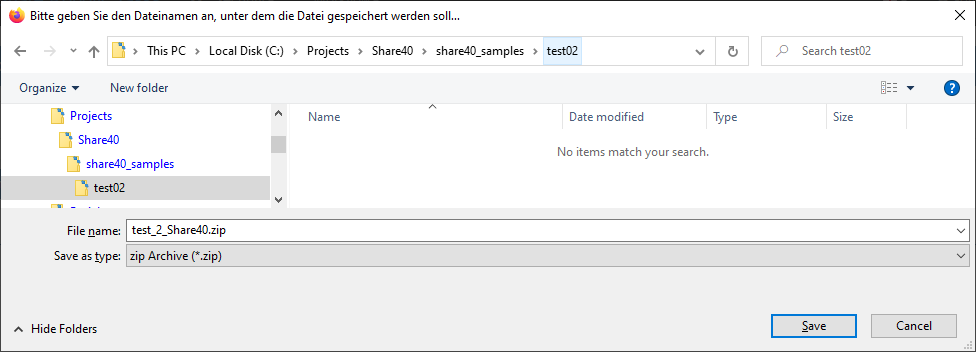
Now open the ‘test\_jupyter\_nb\_01.ipynb ‘ sample.



You can use the ‘run’ button to execute individual cells. Then adapt code in the cell – run again. A very comfortable environment for fast ‘testings’ 😊

But jupyter was not planned to be an execution environment – thus calling the skills you must first initialize the agent type module which may be ‘complicated’. Logging is not supported. Please also keep in mind, that you have to close the notebeook and open it again if you switch the ‘environment’. Refreshing the notebook in the browser seemed not to work.   
In short – if jupyter support should get more attraction in future, here some more work is needed.

Maybe you also decide to install MS-Visual Studio Code to change the python code directly, or edit the jupyter notebook(s) more comfortable. Have fun…



You then get a zip file (containing some batch files and python files) - please download & extract it to some (new) subfolder on the pc of c:\projects\share40\sample\_code.

*HINT: After download you must press cancel – for shame this dialog doesn’t close itself…*