REMODEL - Robotic tEchnologies for the Manipulation of cOmplex DeformablE Linear objects

Deliverable 5.1 – Planner architecture

Version 2020-10-06

Project acronym: REMODEL
Project title: Robotic tEchnologies for the Manipulation of cOmplex DeformablE Linear objects
Grant Agreement No.: 870133
ObjectsTopic: DT-FOF-12-2019
Call Identifier: H2020-NMBP-TR-IND-2018-2020
Type of Action: RIA
Project duration: 48 months
Project start date: 01/11/2019
Work Package: WP5 – Cable Manipulation Planning, Execution and Interactive Perception
Lead Beneficiary: UNIBO
Authors: All partners
Dissemination level: Public
Contractual delivery date: 31/10/2020
Actual delivery date: 30/10/2020
Project website address: https://remodel-project.eu

1 Scope

The planner architecture is one of the keys for building the robotic system, because a clear and well-structured definition of the components and their communication interfaces allows building the system on strong foundations.

This enable each module to be implemented independently by the different partners and then easily connected based on the predefined interfaces. To this end, the ROS middleware will be exploited, and the components of the system will be precisely defined together with their interfaces. The process will consist in setting up tests both in simulation and with the real robot that will progressively show more complex planning challenges.

REMODEL must be able to determine the next action to be performed by the robot in a dynamically changing environment toward the execution of the assembly task. While motion planning determines how to perform a particular action, task planning determines which actions to perform.

Though there is an extensive body of work on automated task planning [42][43], the vast majority is not directly applicable here, since the initial condition is unknown due to the unknown initial deformation of the objects to manipulate. To cope with changing initial conditions, we must replan often, and therefore rapidly. This disqualifies planners such as [44][45]. Even more agile planners such as [46], focusing on fast plan generation, are generally disqualified as they lack the expressivity required to model the switchgear domain and to express and optimize the relevant quality measures.

Therefore, we intend to use a planner that makes use of domain-specific guidance, an area where we have extensive experience. Such planners have proven highly effective and expressive in a variety of domains [47][48][49].

Through the REMODEL planner, we will provide support for specifying assembly preferences, in order to generate plans fulfilling the task requirements as well as safety or environmental constraints. New optimization techniques will be developed that are adapted to assembly plans, in order to generate high quality solutions with the performance that is required in the given context. The robot plan can be also adapted to the actual working conditions based on manually programmed skills and preferences, exploiting machine learning techniques to deal with task uncertainties.

[42] D. Nau, M. Ghallab, and P. Traverso. Automated Planning: Theory & Practice. Morgan Kaufmann Publishers Inc., 2004.

[43] M. Ghallab, D. Nau, and P. Traverso. Automated Planning and Acting. Cambridge University Press, 2016.

[44] S. Richter and M. Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. J. of Artificial Intelligence Research, 39(1):127–177, 2010.

[45] I. Cenamor, T. de la Rosa, and F. Fernandez. IBACOP and IBACOP2 planner. In IPC planner abstr., 2014.

[46] V. Vidal. YAHSP3 and YAHSP3-MT in the 8th int. planning competition. In IPC planner abstr., 2014.

[47] T. Au, et al. SHOP2: an HTN planning system. CoRR, abs/1106.4869, 2011.

[48] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. Artificial Intelligence, 116(1–2):123–191, 2000.

[49] J. Kvarnstrom and P. Doherty. TALplanner: A temporal logic based forward chaining planner. Annals of Mathematics and Artificial Intelligence, 30(1):119–169, 2001.

[50] J. Kvarnstrom. Planning for Loosely Coupled Agents Using Partial Order Forward-Chaining, Int. Conf. on Automated Planning and Scheduling (ICAPS), 2011.

2 Planner Description

The REMODEL planner will leverage on a set of low-level actions, implemented as ROS action servers with standardized interface, able to react to changing products and environmental conditions thanks to the input of the sensors (vision, force and tactile sensors) and of the production knowledge database (T3.1). Those low-level actions can be combined in macro actions or directly called by the REMODEL planner. Therefore, the REMODEL planner will establish for each task to be carried out along the manufacturing (e.g. the connection of a cable, the routing of a wire, the placing of a connector) which is the set and the order of low-level actions and macro actions, each of them addressing a subtask, required to accomplish the task itself. This selection will be performed on the base of the information provided by the product database (T3.1) according to the task and component description, by means of a proper association between task and component characteristics and the required actions to be involved in the task execution.

The REMODEL planner will be based on the FlexBE capabilities, in order to facilitate the definition of new behaviors (i.e. combination of low-level and/or macro actions) and to exploit its behavior engine to run and monitor the task execution. FlexBE behaviors will be defined as nested state machines, in which each state will address the execution of a single task operation.

A new REMODEL behavior generator exploiting XML notation for the automatic definition of FlexBE behaviors will allow for an easy and flexible implementation of new behaviors and macro actions during task execution.

The REMODEL planner will feature four levels of abstraction to properly address and generalize to all the REMODEL use cases:

- A Use Case Supervisory level for the dynamic definition and implementation of each manufacturing application according to the information provided by the production knowledge database (T3.1)
- A Task Supervisory level for the execution of each task behavior (e.g. full deployment of a single cable in a gearbox)
- A Behavior Control Level for the execution of macro operations (e.g. the connection of a cable, the routing of a wire, the placing of a connector)
- An Action Control Level for the execution of each single operation (e.g. detect the cable, pick the cable, place the cable) through ROS actions servers.

System failures in the execution will be specifically handled on the corresponding level according to the severity of the problem.

The Use Case Supervisor (Fig. 1) will generate and implement the ordered sequence of tasks provided by the production knowledge database (T3.1). The Use Case Supervisor will be also connected to the User Interface defined in T3.2 to enable the user to monitor and interact with the execution of the robot tasks. By exploiting the REMODEL task behavior generator, it will define and implement the FlexBE state machine for each specific task through dynamic composition of macro operations and actions. A complete description of the Behavior generator will be presented in Chapter 3.

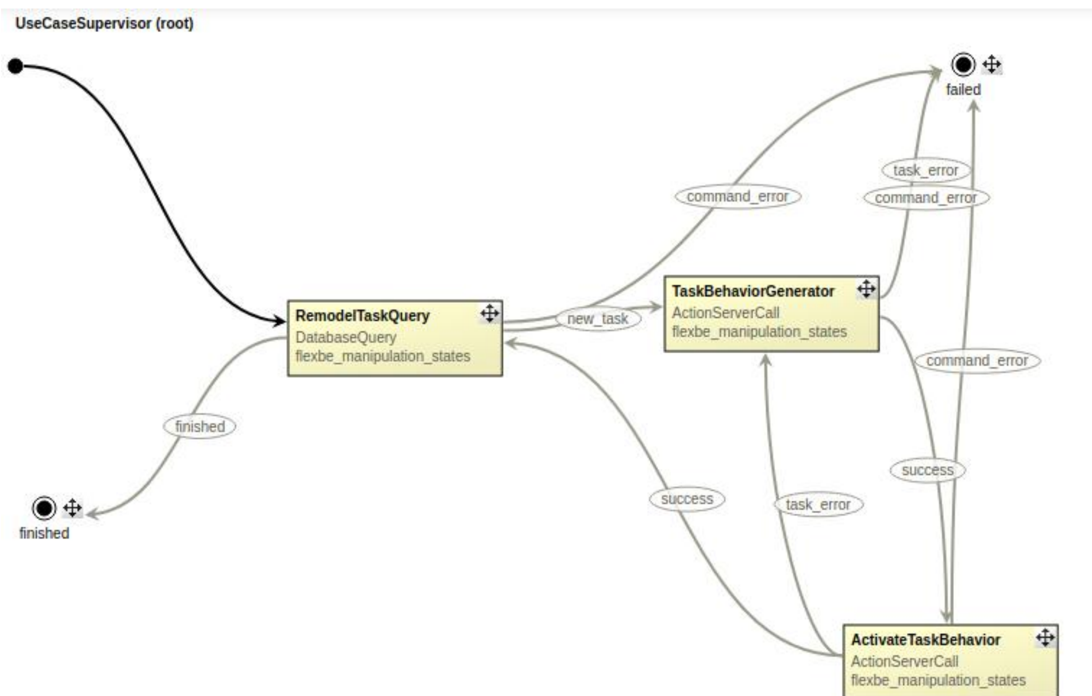


Figure 1. The Use Case Supervisor.

At the beginning of the process or every time a task is completed successfully, the supervisor will automatically load the information for the next task from the production knowledge database (T3.1) and implement its new state machine if not available or simply launch it if already available. In case of failure in the execution of a task, the supervisor will exploit the information provided by the system (sensor data, failure information) to rebuild the state machine or change the parameters to address the arisen problems, eventually by including some action to repeat the required measurements. The generation of task behaviors and failure policies will be implemented according to the user specifications. The vector of task parameters will be passed along the state machine and implemented in each step of the work at the action server level.

Each task behavior (see Fig. 2) is composed as a series of actions (yellow blocks) and low-level behaviors describing macro-actions as placing a connector or routing a cable (purple blocks). The task supervisor monitors the execution of each task and send any failure information to the use case supervisor in case a new planning is needed.

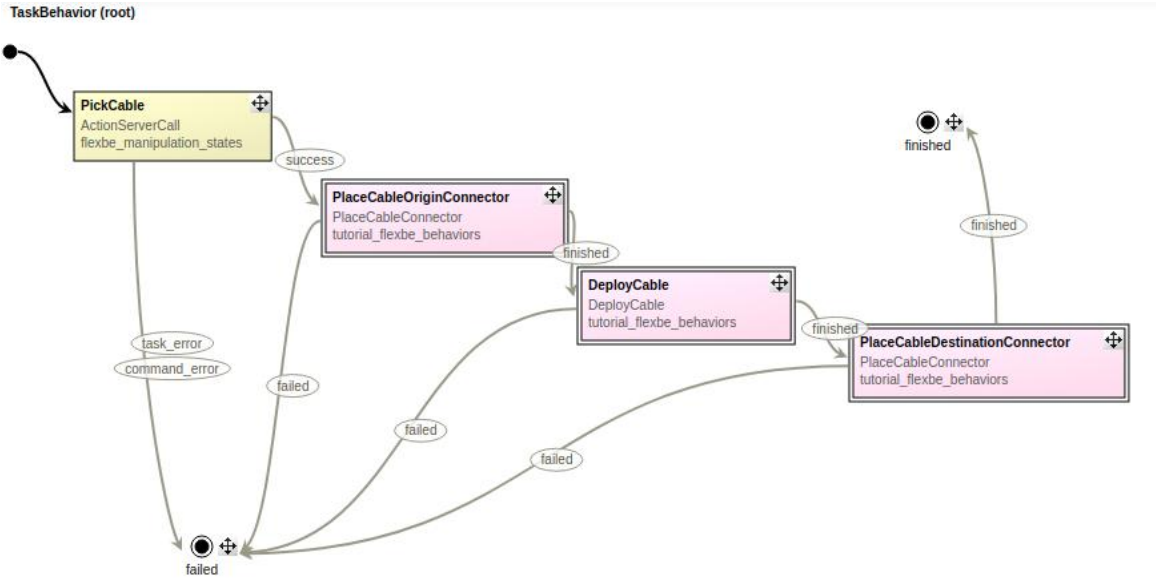


Figure 2. The Task Supervisor.

Each low-level behavior is defined as a state machine implementing several actions and failure policies. Behaviors that have been defined previously can be nested inside a new behavior according to the task needs. Any new behavior will be generated through the REMODEL behavior generator during the planning phase of each task.

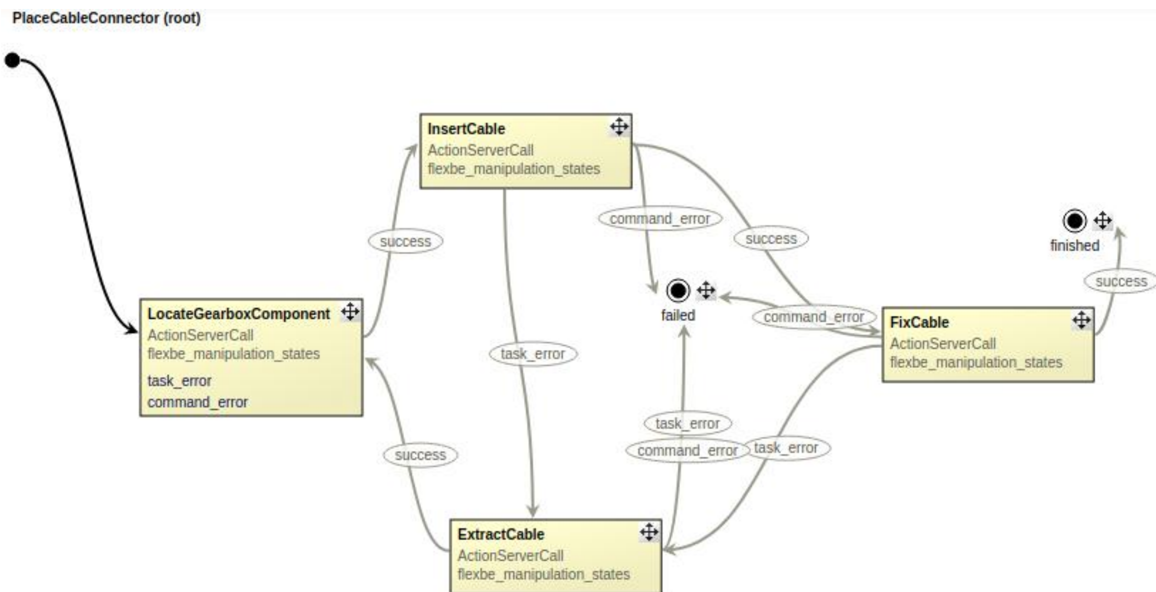


Figure 3. The state machine for the place connector operation.

In Figure 3 an example of a state machine for the placement of a cable connector is proposed. For a different task, the sequence and type of action servers to call will change accordingly. The system is characterized by three actions and a failure recovery state. By traversing the state machine, the system will localize the requested component and connector through the product data and a vision system (LocateGearboxComponent state). and will perform the insertion of the cable (InsertCable state) and the fixing of the connector (FixCable state). In case of failure for environmental causes (wrong detection, unsteady deploy-

ment, etc.) a recovery action will be attempted (ExtractCable state) and the localization will be performed again. The number of attempts for each state and their recovery policies will be defined according to the user request. In case of repeated failures or in presence of external factors affecting the task, the system will mark the operation as failed and send all data available to the supervisory level for real time modifications of the state machine.

3 ActionLib States

FlexBE states are the high-level building blocks from which behaviors are constructed and are supposed to interface with the capabilities of a REMODEL robotic system. ROS actionlib provides an interface where robot capabilities can be provided by ROS nodes implementing an action server and states can then access them by acting as an action client. The actionlib interface is designed for long-term (longer than one update cycle) actions as it is non-blocking and optionally provides feedback while being executed. This makes it a perfect interface to be used along with FlexBE. Therefore, it is highly recommended to base the development of REMODEL abilities on action interfaces.

3.1 Action Definition

As an example of action interface, TECNALIA provided the action definition for the PlacePin ability.

```
# Define the goal
string id # Identifier for traceability

string robot_1_name
string robot_2_name

float64 path_space
float64 connector_space
bool centered   # Set true to center all poses
bool rotate   # Set true to rotate and put the long axis in Y

bool show_in_TF

float64 timeout # In seconds
---
# Define the result
string id

bool success
string message
float64 elapsed_time # In seconds

int16 error_code
string error_message
---
# Define a feedback message
string id
string message
float64 elapsed_time # In seconds
float64 completed_percentage
```

The general structure it is suggested to adopt for the implementation of REMODEL abilities can be summarized as in the following:

- In the goal an ID to the action is provided, as well as a timeout. Here also the action specific parameters are provided;
- The result provides the ID, success (boolean), an optional message and the elapsed time, as well as error management variables;
- The feedback provides the ID, an optional message, the elapsed time and the estimation of the completed action percentage.

3.2 Flexbe Action Client

In the following, a potential implementation of the FlexBe ActionState calling the PlacePin Action Sever previously described in reported as an example of how to implement ActionStates for all the REMODEL abilities. To implement the Flexbe ActionState t for the REMODEL abilities, make sure to import the action client proxy and the required message types of the action interface:

```
from flexbe_core.proxy import ProxyActionClient
# example import of required action
from remodel.msg import PlacePinAction, PlacePinGoal
```

3.2.1 Declaration

It is recommended to create the ActionState client in the constructor of the state as this will check the availability of the action server before starting the behavior and thus, reduce the risk for runtime failure.

In order to declare the required ActionState client, add the following code to the constructor:

```
self._topic = 'remodel_abilities/placepin'
self._client = ProxyActionClient({self._topic: PlacePinAction})
```

3.2.2 Sending a Goal

Typically, a state sends its goal once when it becomes active in order to trigger a certain action. Thus, create and send the action goal in the *on_enter* callback of your state. You can access *userdata* input keys here as required.

```
goal = PlacePinGoal()
goal.id = userdata.placepin_id

self._error = False
try:
    self._client.send_goal(self._topic, goal)
except Exception as e:
     Logger.logwarn('Failed to send the PlacePin command:\n%s' % str(e))
     self._error = True
```

For robustness, it is recommended to embed the action call in a try/catch block in case there are any problems during runtime. The variable *self._error* can be used in the execute function to return a failure outcome if any problems occurred:

```
# Check if the client failed to send the goal.
```

```
if self._error:
        return 'command_error'
```

3.2.3 Checking for Result

Finally, in the execution loop, it is possible to check if the action has already finished and evaluate its result. It is possible here to store relevant parts of its result in the *userdata*.

```
if self._client.has_result(self._topic):
      result = self._client.get_result(self._topic)
      elapsed_time = result.elapsed_time

      userdata.elapsed_time = elapsed_time
        if elapsed_time > self._max_time:
            return 'PlacePin takes longer than normal'
        else:
            return 'PlacePin finished normally'
```

It is also possible to access the result status of the action call if there is no notation of success provided by the result message itself:

```
if self._client.has_result(self._topic):
        status = self._client.get_state(self._topic)
        if status == GoalStatus.SUCCEEDED:
      return 'success'
 elif status in [GoalStatus.PREEMPTED, GoalStatus.REJECTED, GoalStatus.RECALLED, GoalStatus.ABORTED]:
       Logger.logwarn('Action failed: %s' % str(status))       return 'failed'
```

Note that you need to import the GoalStatus message provided by actionlib for this:

```
from actionlib_msgs.msg import GoalStatus
```

A complete exmaple of ActionState implementation can be found on ExampleActionState on the FlexBe github repository.

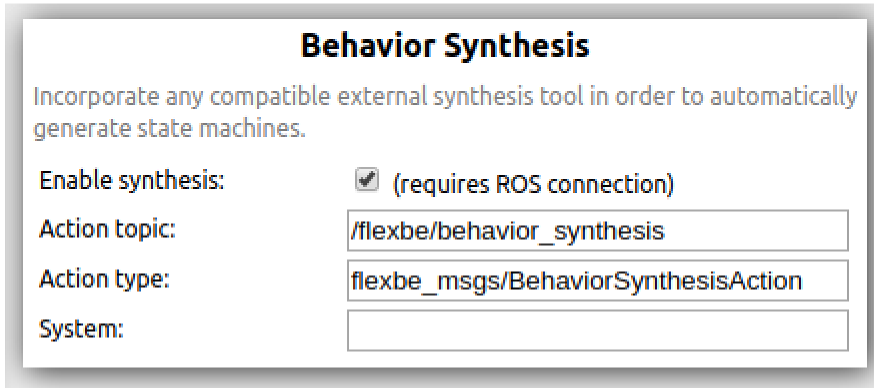
4 Autonomous Behavior Synthesis and Execution

While the FlexBE's editor can be used to manually create behaviors such as the one described in the previous chapter, an interesting feature for a frequently varying task is the possibility to autonomously generate and execute new behavior dynamically.

To automatize the behavior synthesis, a Synthesis Action Server has been implemented. The Synthesis Action Server can be download from the REMODEL gitlab repository https://dei-gitlab.dei.unibo.it/palli_group/flexbesynthetizer, please refer to the repository README.md file for the documentation about the installation and activation of the server. In chapter 4, the XML file format adopted by this Synthesis Action Server to describe the state machine to be synthesized will be described.

4.1 Using the Graphic Synthesis Interface

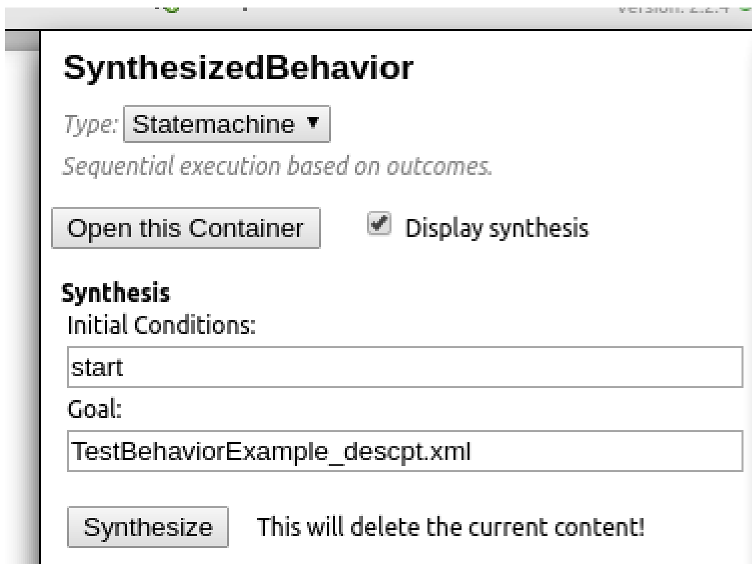
In order to set up the Graphic Synthesis Interface, go to the Configuration view in the FlexBE App where you will find a specific panel for synthesis:



Obviously, check the checkbox in order to “Enable synthesis” option in the editor. This requires ROS connection because the Synthesis Action Server needs to be contacted. You can create a launch file for this purpose which includes both the launch file of the synthesis server and the *flexbe_widget/launch/behavior_ocs.launch* launch file.

Furthermore, set the action topic as shown in the picture in order to be compatible to the REMODEL Synthesis Action Server. System is an optional configuration field where you can provide an identifier for the used robot system to the Synthesis Action Server, given this is required or supported by the specific server.

After configuration is done, you can go to the Statemachine Editor view and add a new container. You will recognize a new checkbox in the properties of this container which enables you to use the synthesis interface.

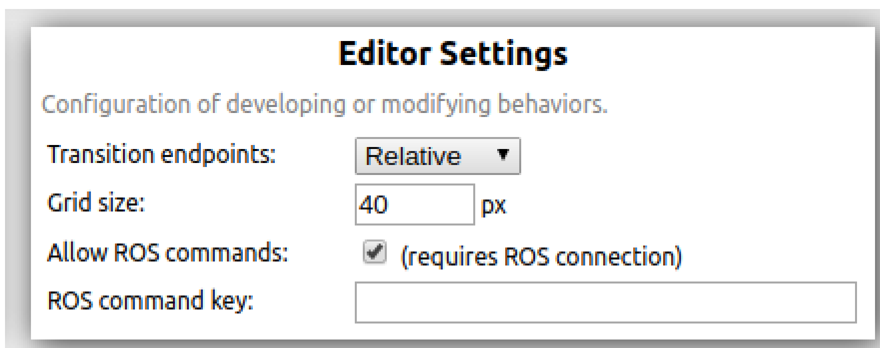


Initial Conditions expects a statement describing the state which is present when entering the container and Goal specifies the path of the XML file describing the state machine to be generated according to the specification provided in chapter 4. Please refer to the documentation of the used synthesis tool for further details on the expected input.

As soon as you click synthesize, an action goal containing the specifications will be sent to the synthesis server and eventually, a result comes back. This result will replace the current content inside the container for which you request synthesis but does not touch any part outside. Thus, you can synthesize several parts of a behavior independently. After synthesis, you can open the container, check the result, and potentially make any adjustments or additions manually. Again, please refer to the documentation of the synthesis server in order to check if it expects you to perform any manual additions.

4.2 Using the Command-Line Synthesis Interface

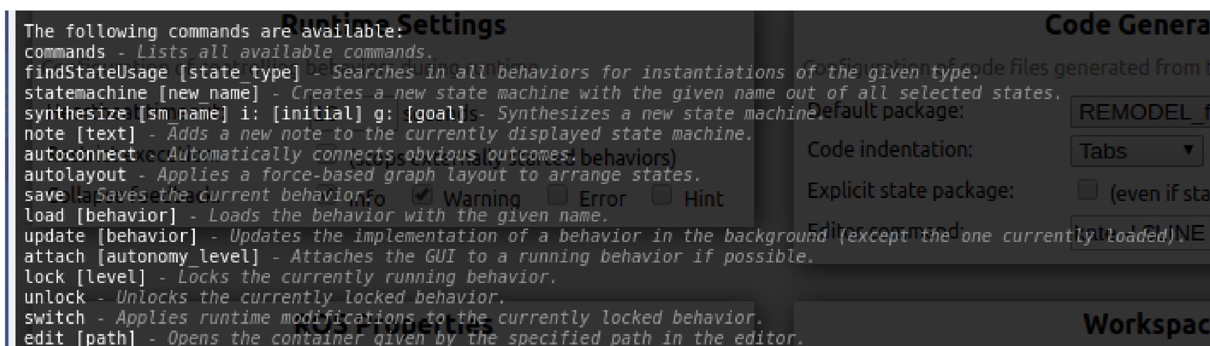
After configuring the Graphic Synthesis Interface in order to enable the connection with the Synthesis Action Server as specified in the previous section, it is possible to access the FlexBe synthesis features by a command line interface provided through the topic */flexbe/uicommand by messages* of the type *flexbe_msgs/UICommand.* To activate the command line interface, go to the Configuration view in the FlexBE App where you will find a specific panel for settings. Here, you need to check the "Allow ROS commands" checkbox as shown in the figure below and eventually set a ROS command key for security reasons.



The list of available commands can be retrived by the following command:

```
rostopic pub /flexbe/uicommand flexbe_msgs/UICommand  "command: 'help', key: ' ' "
```

The FlexBe app will show the following list of commands



To load a specific behavior or a template one, such as *my_first_behavior*, the *load* command must be used as reported in the following:

```
rostopic pub /flexbe/uicommand flexbe_msgs/UICommand "command: 'load my_first_behavior', key: ' ' "
```

To synthesize a specific state machine, the *synthesize* command is needed

```
rostopic pub /flexbe/uicommand flexbe_msgs/UICommand "command: 'synthesize SynthesizedBehavior i: start g: TestBehaviorExample_descpt.xml', key: ' ' "
```

where the *SynthesizedBehavior* is the name of the synthesized state machine, *start* is the initial state denoted by the *i:* delimiter and *TestBehaviorExample_descpt.xml* is the path of the XML file describing the state machine to be generated according to the specification provided in chapter 4 and denoted by the *g:* delimiter.

Finally, the generated behavior can be saved by the *save* command:

```
rostopic pub /flexbe/uicommand flexbe_msgs/UICommand "command: 'save', key: ' ' "
```

5 Running Behavior Autonomously

Note that running a behavior in autonomous mode means that all operator interaction features (like state transition confirmation and Autonomy Level) are disabled and the only available command is to force the running behavior to stop. This can be done by sending a message of type *std_msgs/Empty* to the topic */flexbe/commands/preempt*.

In case of a fully autonomous robot with FlexBE as the top-level control instance, the onboard behavior engine is need first:

```
roslaunch flexbe_onboard behavior_onboard.launch
```

5.1 Command Line / Launch File

On any computer connected to the ROS master, run the following command to start execution of the behavior named "Example Behavior":

```
rosrun flexbe_widget be_launcher -b 'Example Behavior'
```

This will command the behavior engine to execute the specified behavior. In the case you run this from a different computer than the onboard computer and have local changes, these changes will be applied.

In order to include this in a launch file, add the following to the respective launch file:

```
<arg name="behavior_name" default="Example Behavior" /><node
name="behavior_launcher" pkg="flexbe_widget" type="be_launcher" output="screen" args="-
b '$(arg behavior_name)'" />
```

5.2 Behavior Action

Alternatively, you can command behavior execution via action call. This way might be best if you want to embed FlexBE behaviors in a different top-level control instance. First run the FlexBE action server:

```
rosrun flexbe_widget be_action_server
```

This action server will listen on the action topic */flexbe/execute_behavior* of type *flexbe_msgs/BehaviorExecution*. For example, you can test executing a behavior from the command line by running the following minimal example:

```
rostopic pub /flexbe/execute_behavior/goal flexbe_msgs/BehaviorExecutionActionGoal
'{goal: {behavior_name: "Example Behavior"}}'
```

5.3 Attaching the User Interface

Although executing in autonomous mode, it might be desired sometimes to attach to a running behavior in order to monitor it, send commands, or make runtime modifications. Considering that the "Example Behavior" is already running in the background, launch the user interface as usual:

```
roslaunch flexbe_widget behavior_ocs.launch
```

When the GUI comes up, it should already notify you that there is a behavior running. Make sure to load the running behavior, "Example Behavior" in this case, first. Otherwise, attaching will complain and give you the name of the required behavior to be loaded. Afterwards, switch to the Runtime Control view of the GUI and click the "Attach" button now being displayed in the main panel (instead of the option to start behavior execution). This will attach the GUI to monitor the current execution and will switch the execution mode from autonomous to supervised, i.e., the default mode when starting behaviors from the GUI.

6 REMODEL State Chart XML

This chapter will describe the XML syntax used in the project for the definition of the behaviors for each task. Each XML document will provide a complete description of the state machine executing a given task. Each task will be characterized by a sequence of actions, implemented as states or low-level behaviors. A set of variables for each task will allow an exchange of information between the states and the specification of each state interaction. In this section a detailed description of each element that can define the XML will be presented. The XML notation proposed here is based on the SCXML notation, see https://www.w3.org/TR/scxml.

6.1 <statemachine>

The top wrapper element, which identifies the file as behavior descriptor for the REMODEL project.

6.1.1 Attribute Details

Name	Required	Type	Default Value	Description
name	True	string	none	The name of this task behavior
initial	True	string	none	The name of the first state to be executed

6.1.2 Children

- <datamodel>: The model containing all the parameters of the state machine. See 6.2
- <state>: A state of the machine. See 6.3
- <behavior>: A low level behavior exploited in the machine. See 4.4
- <connection>: A remapping of variables between the output and input of two states. See 4.5

A valid REMODEL state chart must have at least one <state> or <behavior> and a <datamodel>.

6.2 <datamodel>

The model of data used in the state machine, comprising of input/output variables, possible outcomes and machine parameters.

6.2.1 Children

- <outcomes>: The possible outcomes for the state machine. See 6.6
- <input >: The input variables of the state machine. See 6.7
- <output >: The output variables of the state machine. See 6.8

6.3 <state>

The description of a state of the machine.

6.3.1 Attribute details

Name	Required	Type	Default Value	Description
id	True	string	none	The name of this state istance
type	True	string	none	The type of istance of the state
topic	False	String	none	The topic for the state action server

6.3.2 Children

- <input>: An input variable of the state. Can be defined multiple times. See 6.7
- <transition>: A transition to a different state according to a specified event. Can be defined multiple times. See 6.8
- <output>: An output variable of the state. Can be defined multiple times. See. 6.9
- <param>: A parameter variable defined in the state. Can be defined multiple times. See. 6.10

6.4 <behavior>

The description of a low-level behavior of the machine.

6.4.1 Attribute details

Name	Requi-red	Type	Default Va-lue	Description
id	True	string	none	The name of this low-level behavior istance
type	True	string	none	The type of istance of the behavior

6.4.2 Children

- <input>: An input variable of the state. Can be defined multiple times. See 6.7
- <transition>: A transition to a different state according to a specified event. Can be defined multiple times. See 6.9
- <output>: An output variable of the state. Can be defined multiple times. See. 6.10

6.5 <connection>

The description of a remapping between a state input variable and the output variable of a different state. The remapping allows the state to use as input the result of a previously executed state given under a different name. Multiple connections can be specified by defining multiple <connection> instances.

6.5.1 Attribute details

Name	Required	Type	Default Value	Description
output_state_id	True	string	none	The name of the state providing the data
output	True	string	none	The name of the variable storing the data
input_state_id	True	string	none	The name of the state requiring the data
input	True	string	none	The name of the variable read from the state

6.6 <outcome>

The possible outcomes of the state machine. At least one outcome is required for the machine to be feasible. Multiple outcomes can be specified by defining multiple <outcome> instances.

6.6.1 Attribute details

Name	Required	Type	Default Value	Description
name	True	string	none	The name of outcome of the state machine

6.7 <input >

The input variables of the state machine, of a state or a behavior. Multiple input variables can be specified by defining multiple <input> instances.

6.7.1 Attribute details

Name	Required	Type	Default Value	Description
name	True	string	none	The name of input variable of the state machine
type	False	string	none	The type of variable stored (int, string, etc.)

6.8 <output >

The output variables of the state machine, of a state or a behavior. Multiple output variables can be specified by defining multiple <output> instances.

6.8.1 Attribute details

Name	Required	Type	Default Value	Description
name	True	string	none	The name of output variable of the state machi-ne
type	False	string	none	The type of variable stored (int, string, etc.)

6.9 <transition>

The transition from the current state to a different state according to a specified event. Multiple istances of a transition other states according to different events can be specified by defining multiple <transition> istances.

6.9.1 Attribute details

Name	Required	Type	Default Value	Description
event	True	string	none	The name of outcome causing the transition
target	True	string	none	The name of the new state after the transition

6.10 <param>

A parameter for the state or behavior. Multiple parameters can be defined by defining multiple <param> istances.

6.10.1 Attribute details

Name	Required	Type	Default Value	Description
name	True	string	none	The name of parameter for the state
value	True	variable	none	The value of the parameter for the state

7 Conclusions

This deliverable describes how the planning of also complex dynamically chaning tasks will be managed and implemented in REMODEL. A guideline for the development and the action server interface as well as for the FlexBe Action States is here provided together with the commands required to control the generation and activation of the behaviors from external applications, such as the production supervisor. An implementation of the behavior synthesis server is also provided to generate dynamic behaviors for XML task description. The documentation of the XML format is also reported in this deliverable.