Behavior Design with State Machines

MSc. Robotics

Simon Bøgh

Associate Professor

Faculty of Engineering, Department of Materials and Production

Spring 2020

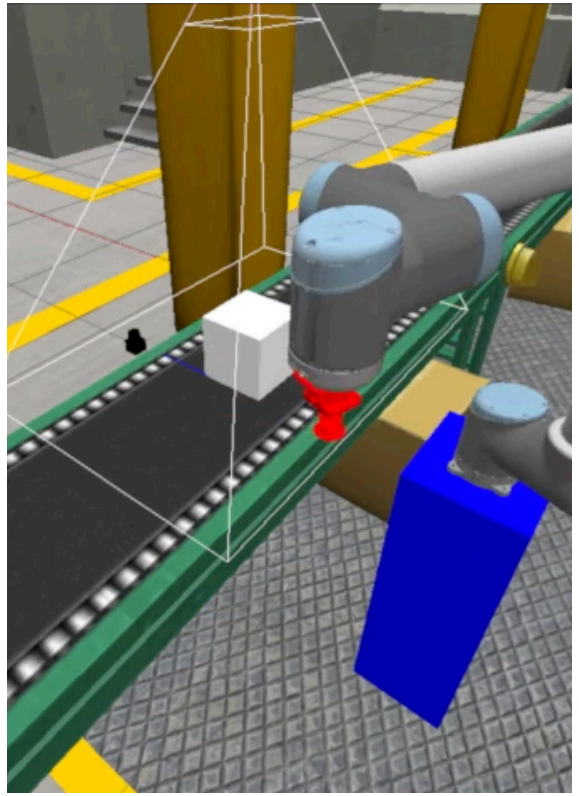
AALBORG UNIVERSITY
DENMARK

Introduction

- Robot needs to **perform a coordinated series of actions** to solve a task

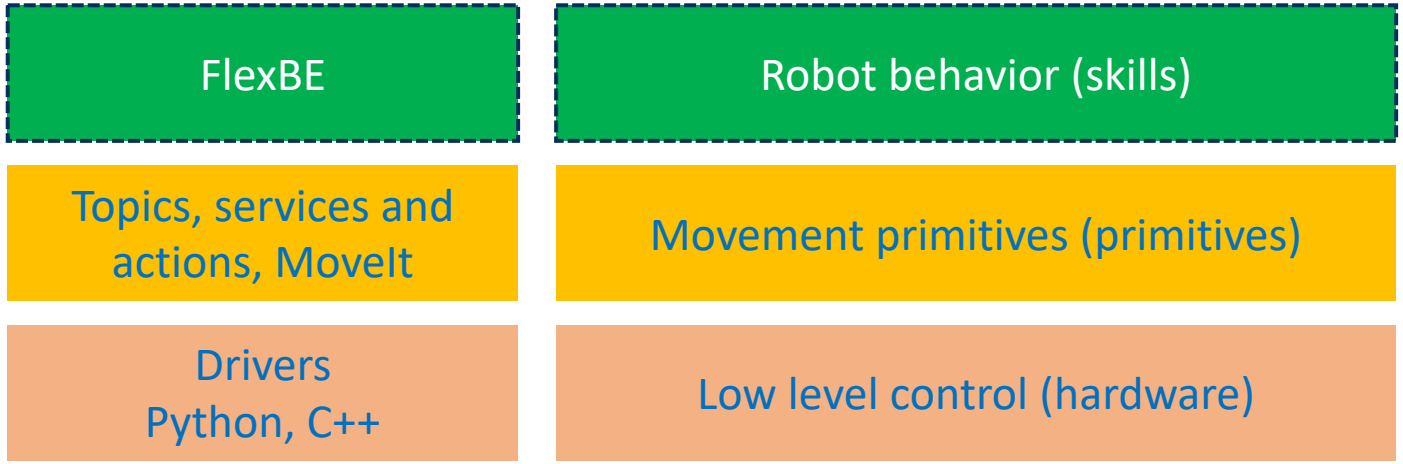
- We have seen how to **program some of the individual actions** needed in such tasks using e.g.
  - Publishers, services and ROS actions
  - MoveIt and move_group node manipulation of robot arm

- Next we will look at behavior design with state machines
  - Basic concepts of robot behavior design with state machines
  - FlexBE: helps you **create complex robot behaviors** without the need for manually coding them



Behavior design for robots

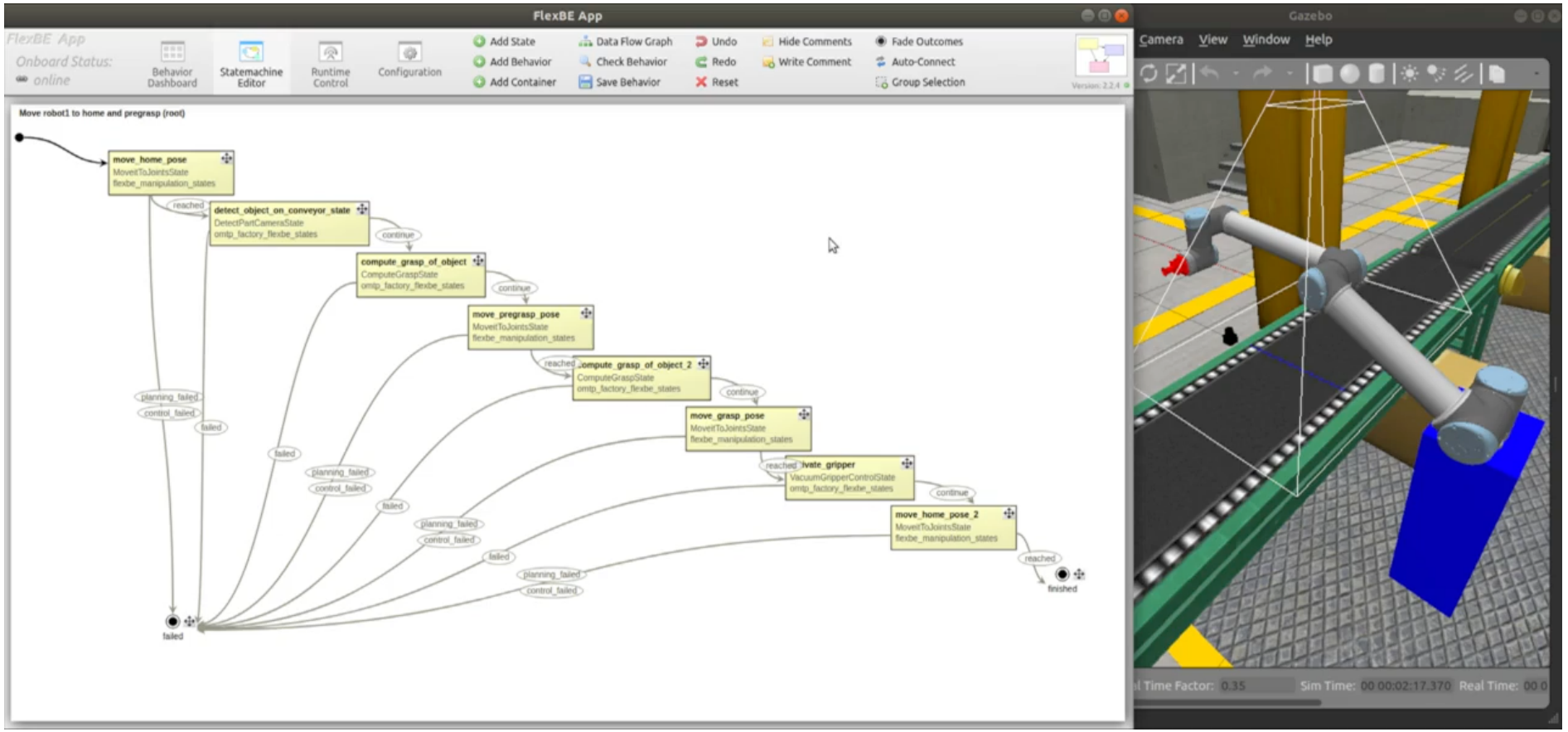
- We will learn how to put the learned capabilities together to solve a task
- We will design and implement a robot behavior in ROS



Outline

- Robot behavior design with state machines
- FlexBE tool to implement them
- Create a FlexBE behavior through graphical interface
- Many standard FlexBE states
- Program a custom FlexBE state e.g. for picking behavior
- Integrate a robot pick and place task in the OMTP factory
- FlexBE website: http://philserver.bplaced.net/fbe/discover.php

FlexBE – behavior engine with state machines



https://youtu.be/zIgJwyTCqbY

Behavior design

Sequence of actions

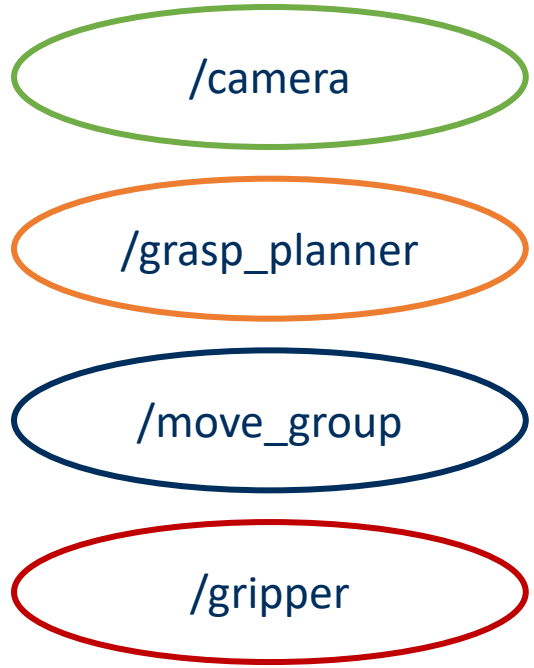
Robot behavior

- Why do we need behavior design?
- A robot needs to carry **a set of actions** to complete a task
  - Detect object
  - Compute grasp
  - Move to pick up
  - Grasp
  - Retreat
- These actions need to be **executed in specific order** when the robot is **in the correct state**
- We need to **coordinate** this set of actions

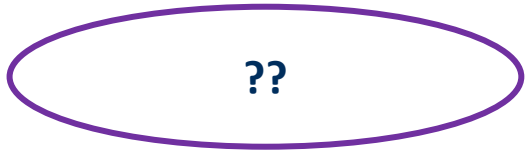
How do we program this kind of behavior?

- Detect object
- Compute grasp
- Move to pick up
- Grasp
- Retreat

Coordinated set of **actions**



- **Coordination?**



What we are learning in this lecture

Behavior design methods

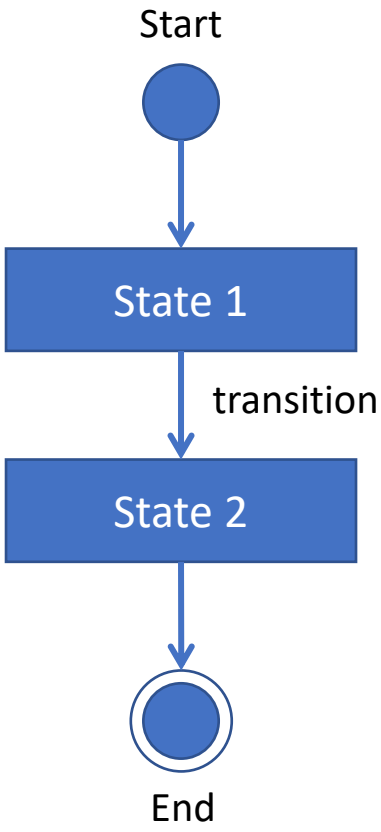
- Different methods exist e.g.
  - Finite state machines
  - Flow charts
  - Behavior trees
  - …
- Which one to use depends on your needs, for example
  - Complexity of the task/behavior?
  - Sequential behavior or will the robot be doing many things in parallel?

Robot state machines

- We will focus on state machines
  - Intuitive way to program robot behavior
- A state machine **defines a behavior as a sequence** of actions that are executed
  - During state execution **only one state is active at a time**
  - One state execution **has to finish before transitioning** to the next state
- Appropriate to design **sequential behavior**
- Each state or action may require
  - **Input data** for its execution (object pose for computing grasp)
  - **Produce output data** (joint values for the robot arm to grasp object)

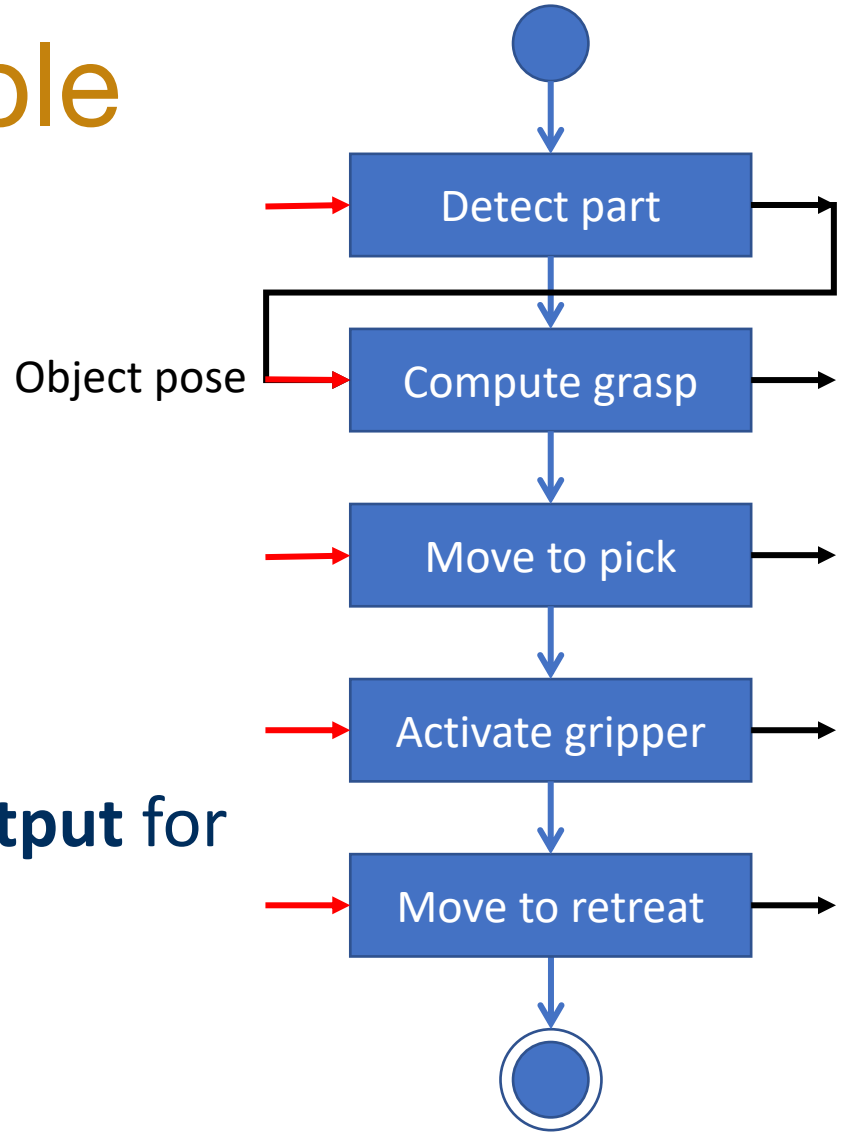
Robot state machines

- State machines represent behavior in a visual form
  - **States** represented as **blobs**
  - **Transitions** represented as **arrows** that connect the state blobs
  - **Entry point: circle** (indicates what is the first state)
  - **End of behavior execution: transition to double circle**



Pick state machine example

- Robot actions
  - Detect object
  - Compute grasp
  - Move to pick up
  - Grasp
  - Retreat
- We need to **define values for input and output** for **each state**

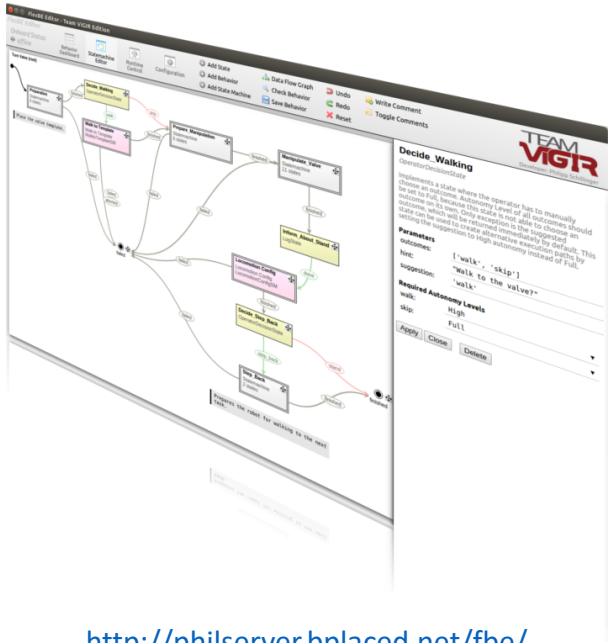


Behavior design with FlexBE

The flexible behavior engine

What is FlexBE?

- Open source high-level behavior engine **coordinating the capabilities of a robot** in order **to solve complex tasks**
- Good operator integration and extensive user interface
  - Besides executing behaviors in full autonomy, the **operator can restrict execution** of certain transitions or **trigger them manually**
- FlexBE App **visual behavior editor**
- Monitor and control behavior execution
- Tools to create your own repository of states and behaviors for your robot application



http://philserver.bplaced.net/fbe/

Our goal with FlexBE

- **Goal**: design robot behaviors
  - **Compose a behavior** with pre-defined states
  - **Configure** states for our state machine
  - Configure each state by **passing run-time data** between states

FlexBE installation

Behavior engine and App

Installation

- Getting started
  - ROS wiki: http://wiki.ros.org/flexbe
  - Official website: http://philserver.bplaced.net/fbe/download.php
  - Github: https://github.com/team-vigir/flexbe_behavior_engine
  - Tutorials: http://wiki.ros.org/flexbe/Tutorials

- Download and installation
  - **Behavior engine** (binary or source) (define and run behaviors)

```
$ sudo apt install ros-$ROS_DISTRO-flexbe-behavior-engine
$ cd omtp_course_ws/src
$ git clone https://github.com/team-vigir/flexbe_behavior_engine.git
```

  - **FlexBE App** (graphical user interface for editing and monitoring behaviors)

```
$ cd omtp_course_ws/src
$ git clone https://github.com/FlexBE/flexbe_app.git
```

Installation

- You can also create an application menu shortcut for the FlexBE App, similar to other code editors like Eclipse or PyCharm

```
$ rosrun flexbe_app shortcut create  # or "remove" to remove it again
```

- There already exist some collections of *states*
  - You can clone the repositories you need and use the states
  - **generic_flexbe_states** (https://github.com/FlexBE/generic_flexbe_states)
    - A useful collection of non-robot-specific states, e.g., for **move_base**, **MoveIt!**, **PCL**, **OpenCV**

FlexBE Workspace Setup (optional)

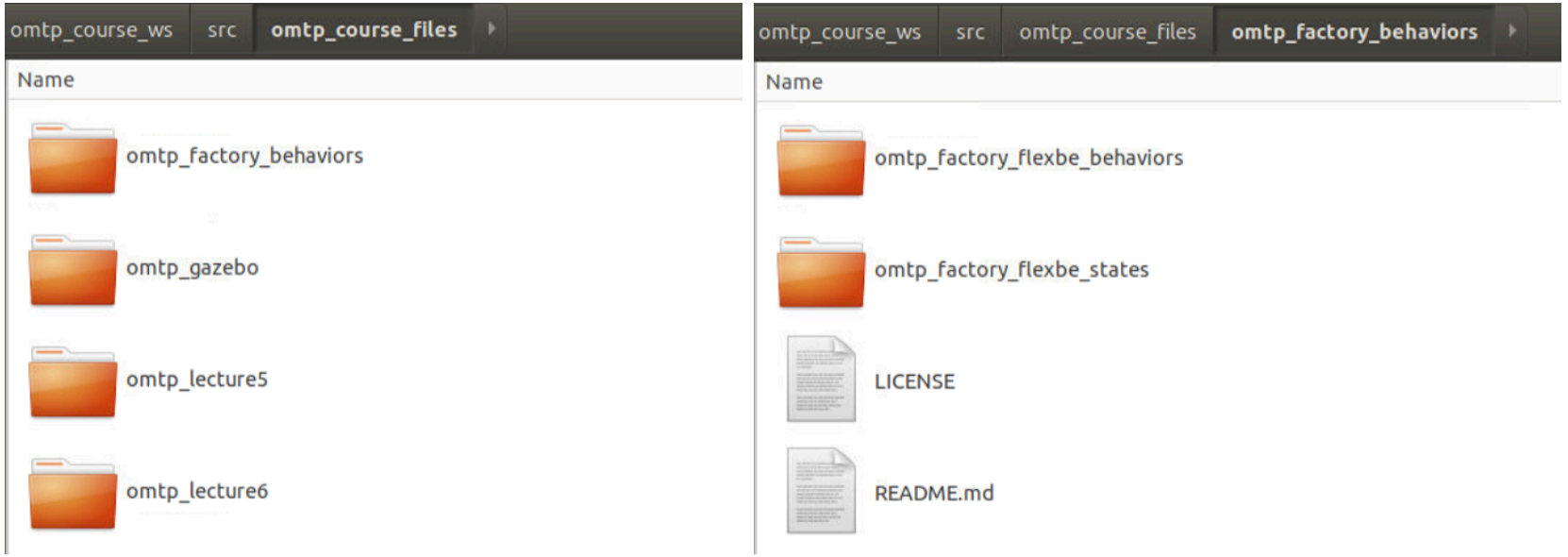
- Before making our first behavior
- Create your own repository for the development of *states* and *behaviors*
- In “catkin_ws/src/omtp_course/”

```
$ mkdir omtp_factory_behaviors
$ cd omtp_factory_behaviors

$ rosrun flexbe_widget create_repo omtp_factory

$ catkin build
$ source ~/omtp_course_ws/devel/setup.bash
```

```
parallels@omtp:~/omtp_course_ws/src/omtp_course$ rosrun flexbe_widget create_repo omtp
Will initialize new behaviors repo omtp_behaviors ...

(1/4) Checking package dependencies...
Done

(2/4) Fetching project structure...
Cloning into 'omtp_behaviors'...
remote: Enumerating objects: 86, done.
remote: Total 86 (delta 0), reused 0 (delta 0), pack-reused 86
Unpacking objects: 100% (86/86), done.
Branch 'feature/flexbe_app' set up to track remote branch 'feature/flexbe_app' from 'origin'.
Switched to a new branch 'feature/flexbe_app'

(3/4) Configuring project template...
Done

(4/4) Initializing new repository...
Initialized empty Git repository in /home/parallels/omtp_course_ws/src/omtp_course/omtp_behaviors
/.git/
[master (root-commit) 319e744] Initial commit
 16 files changed, 432 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 LICENSE
 create mode 100644 README.md
 create mode 100644 omtp_flexbe_behaviors/CMakeLists.txt
 create mode 100644 omtp_flexbe_behaviors/config/example.yaml
 create mode 100644 omtp_flexbe_behaviors/manifest/example_behavior.xml
 create mode 100644 omtp_flexbe_behaviors/package.xml
 create mode 100644 omtp_flexbe_behaviors/setup.py
 create mode 100644 omtp_flexbe_behaviors/src/omtp_flexbe_behaviors/__init__.py
 create mode 100644 omtp_flexbe_behaviors/src/omtp_flexbe_behaviors/example_behavior_sm.py
 create mode 100644 omtp_flexbe_states/CMakeLists.txt
 create mode 100644 omtp_flexbe_states/package.xml
 create mode 100644 omtp_flexbe_states/setup.py
 create mode 100644 omtp_flexbe_states/src/omtp_flexbe_states/__init__.py
 create mode 100644 omtp_flexbe_states/src/omtp_flexbe_states/example_action_state.py
 create mode 100644 omtp_flexbe_states/src/omtp_flexbe_states/example_state.py

Congratulations, your new repository  omtp_behaviors  is ready to be pushed!
Please run the following commands to push it:
  git remote add origin [your_repo_url]
  git push origin master

Your new repository already contains an example behavior and examples for writing own states:
  rosed omtp_flexbe_states example_state.py
  rosed omtp_flexbe_states example_action_state.py
```

FlexBE Workspace Setup

- Folder structure: *behaviors/* and *states/*



Behavior manifest xml-file

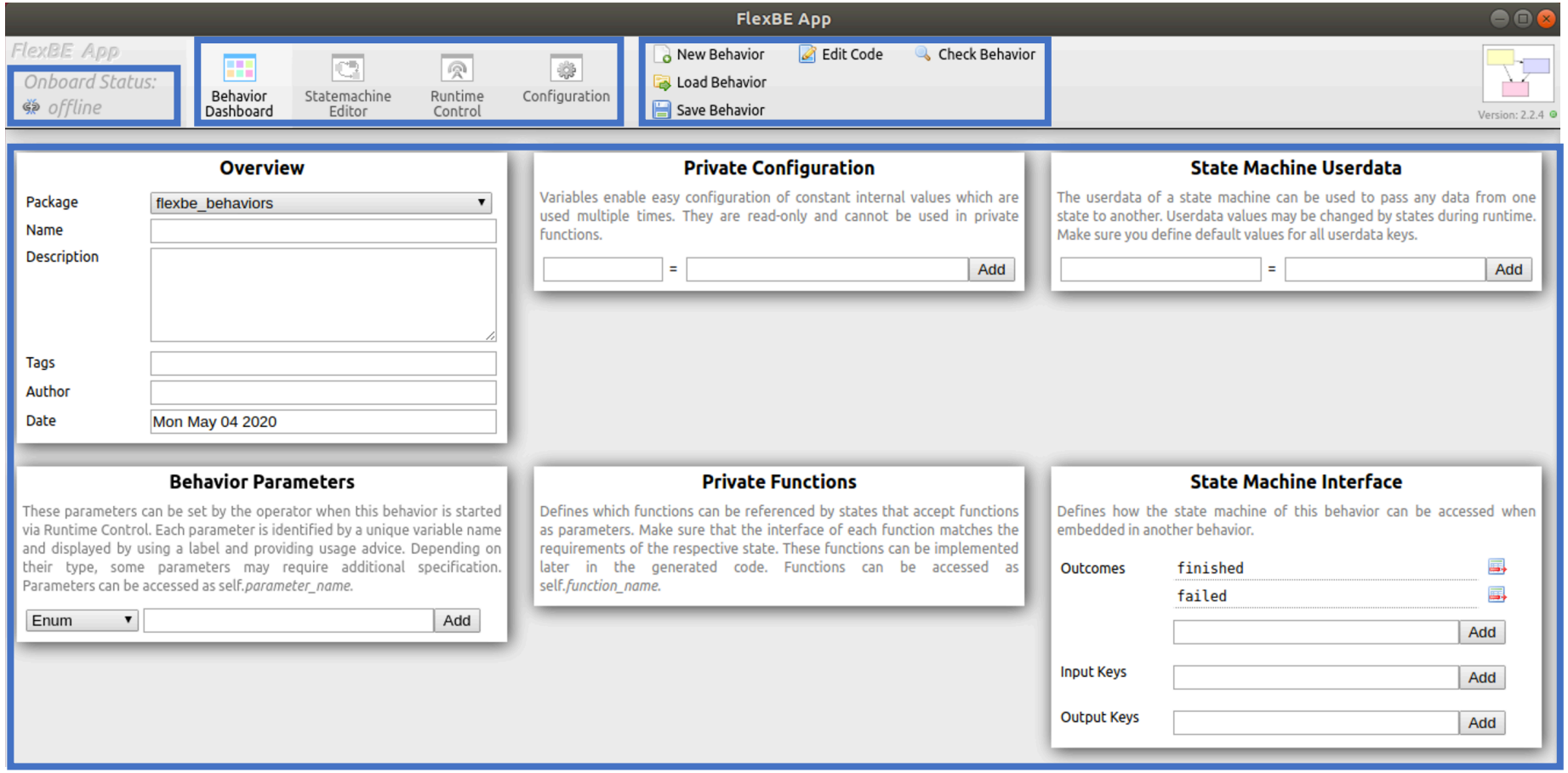
- Your new ROS package ***omtp_factory_flexbe_behaviors***
  - Will be **present in your repository**
  - Will be the **point of reference for FlexBE** in order to store behaviors
- Contains a **folder** called ***manifest***, holding all behavior manifests
- A behavior manifest is an abstract interface declaration each behavior defines, stored as an **XML-file**
- This is how **FlexBE knows about which behaviors are available**, where to find them, and much more

Run FlexBE App

- If you made a shortcut with the command on the previous slide, you can find FlexBE in the program menu
- CLI commands
  - Run FlexBE App alone:
    ```
    $ rosrun flexbe_app run_app
    ```
  - App + Behavior Engine:
    ```
    $ roslaunch flexbe_app flexbe_full.launch
    ```

FlexBE App user interface

ROS connection

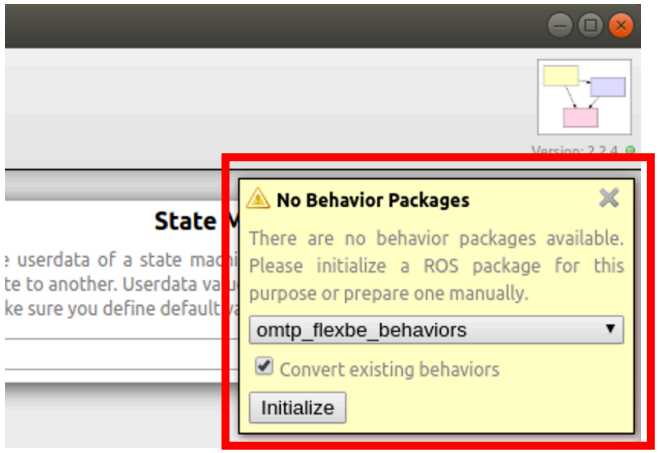


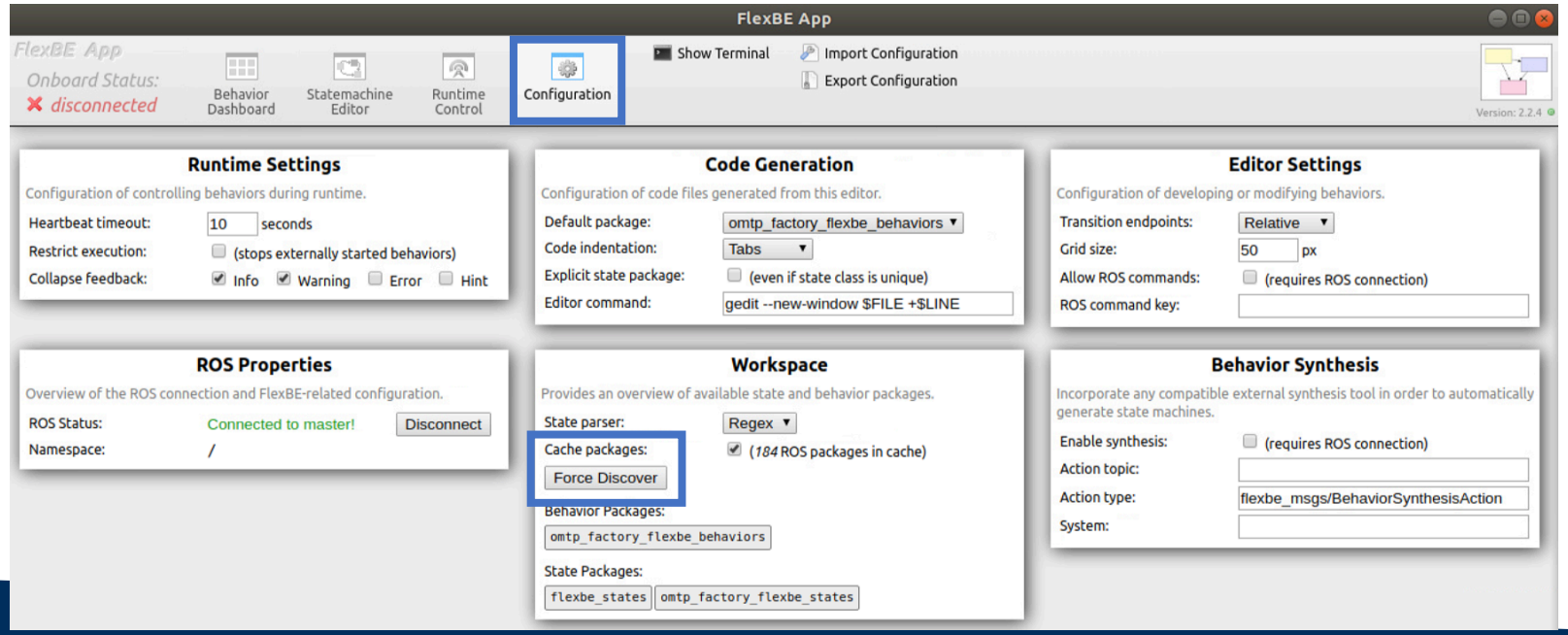
Initialize behavior package

- Initialize the package inside the FlexBE App

```
$ rosrun flexbe_app run_app
```

- Configuration > Workspace > “Force Discover”

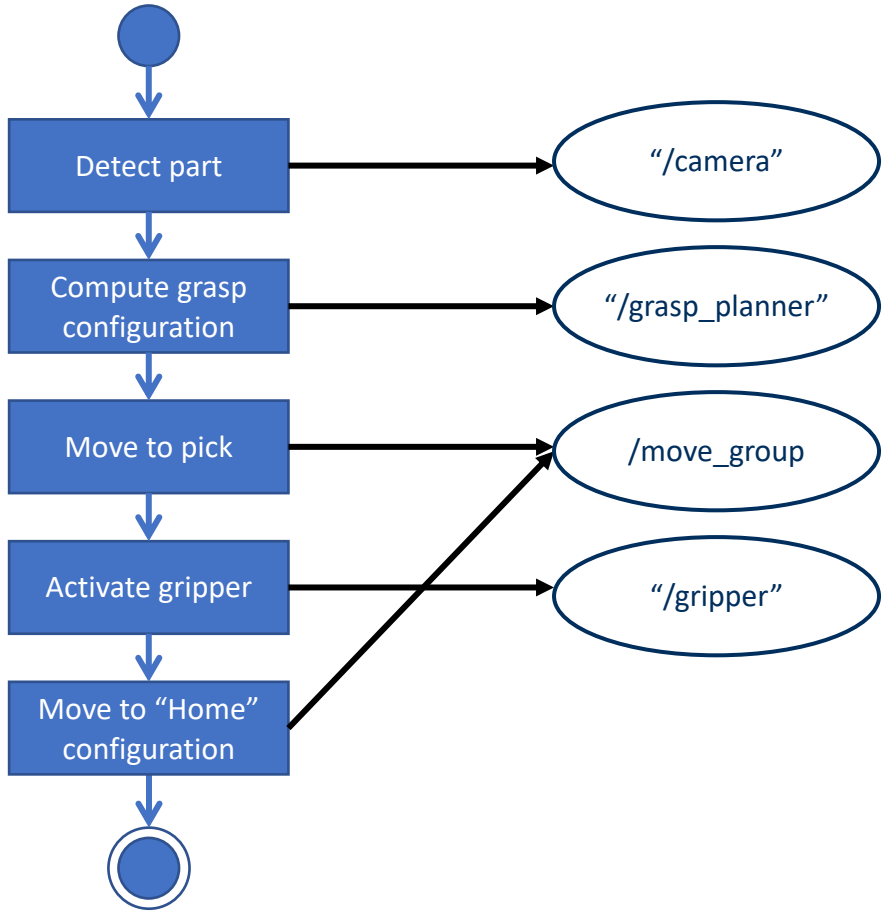




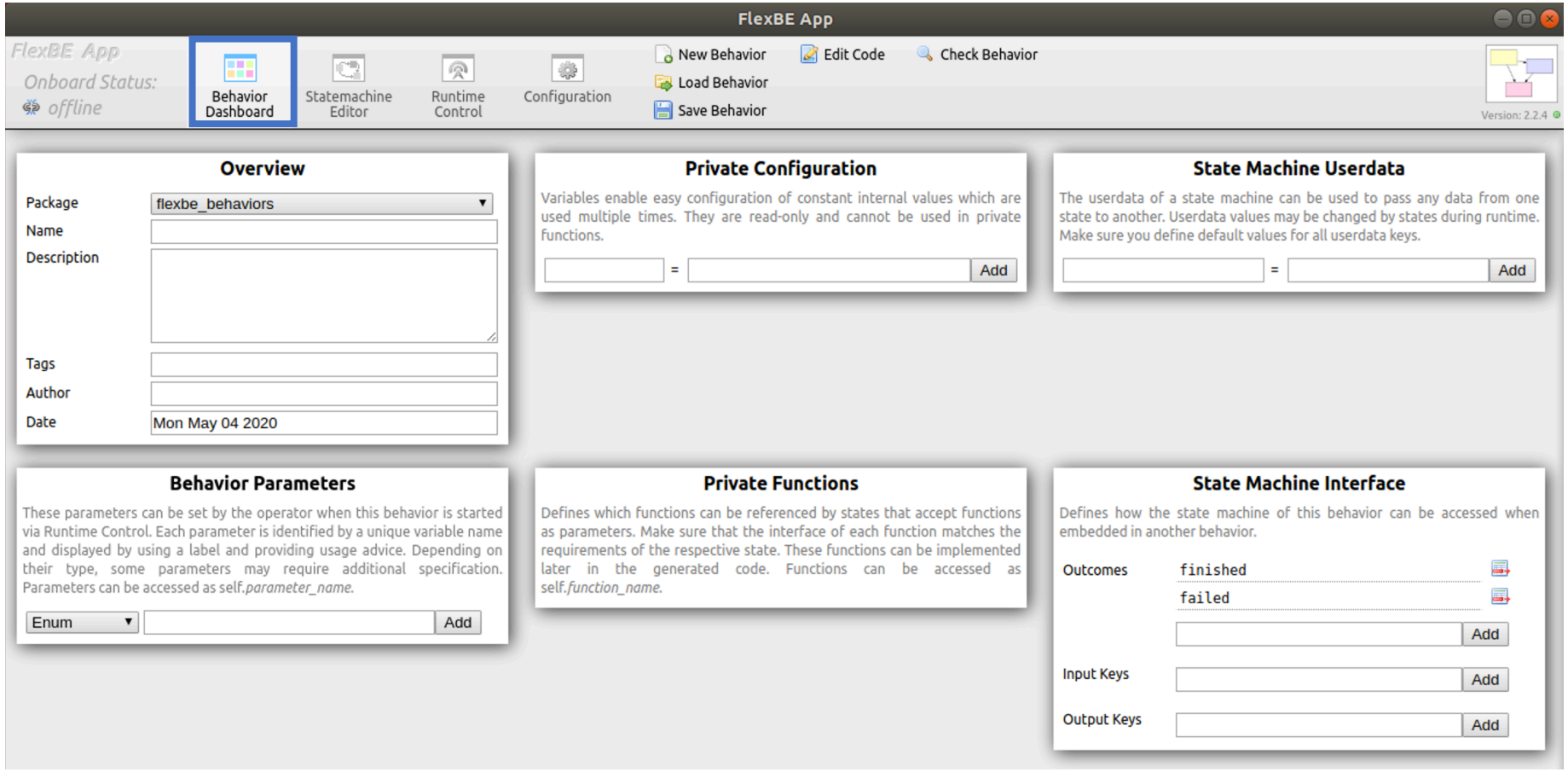
Creating a behavior with FlexBE

Simple behavior example

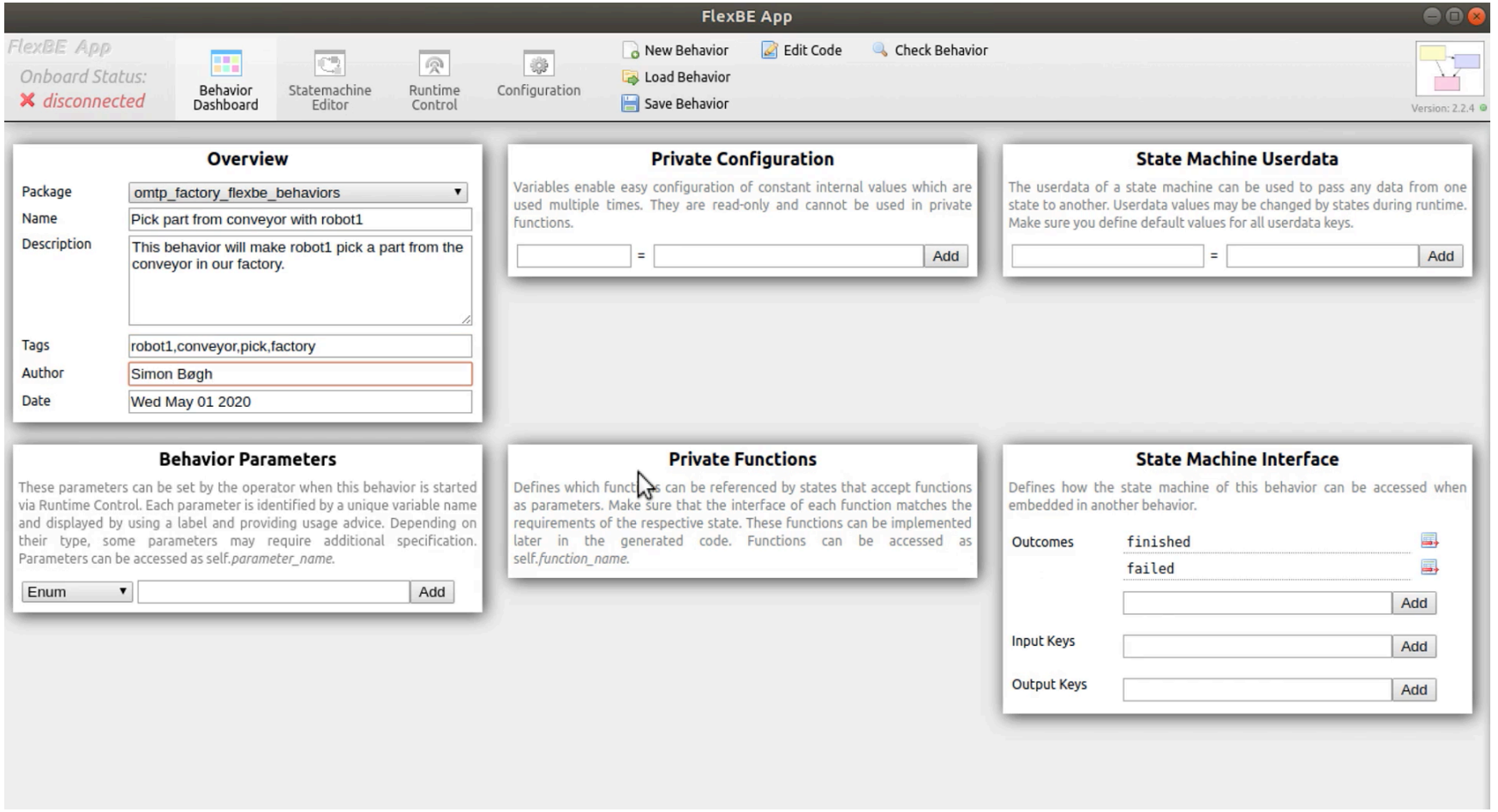
- We will create a simple pick behavior with robot 1
- **Behavior**: Pick a part with robot 1 from the conveyor
  - We will **“R1Home”** joint configuration information defined in the SRDF-file in **omtp_moveit_config**



Behavior Dashboard

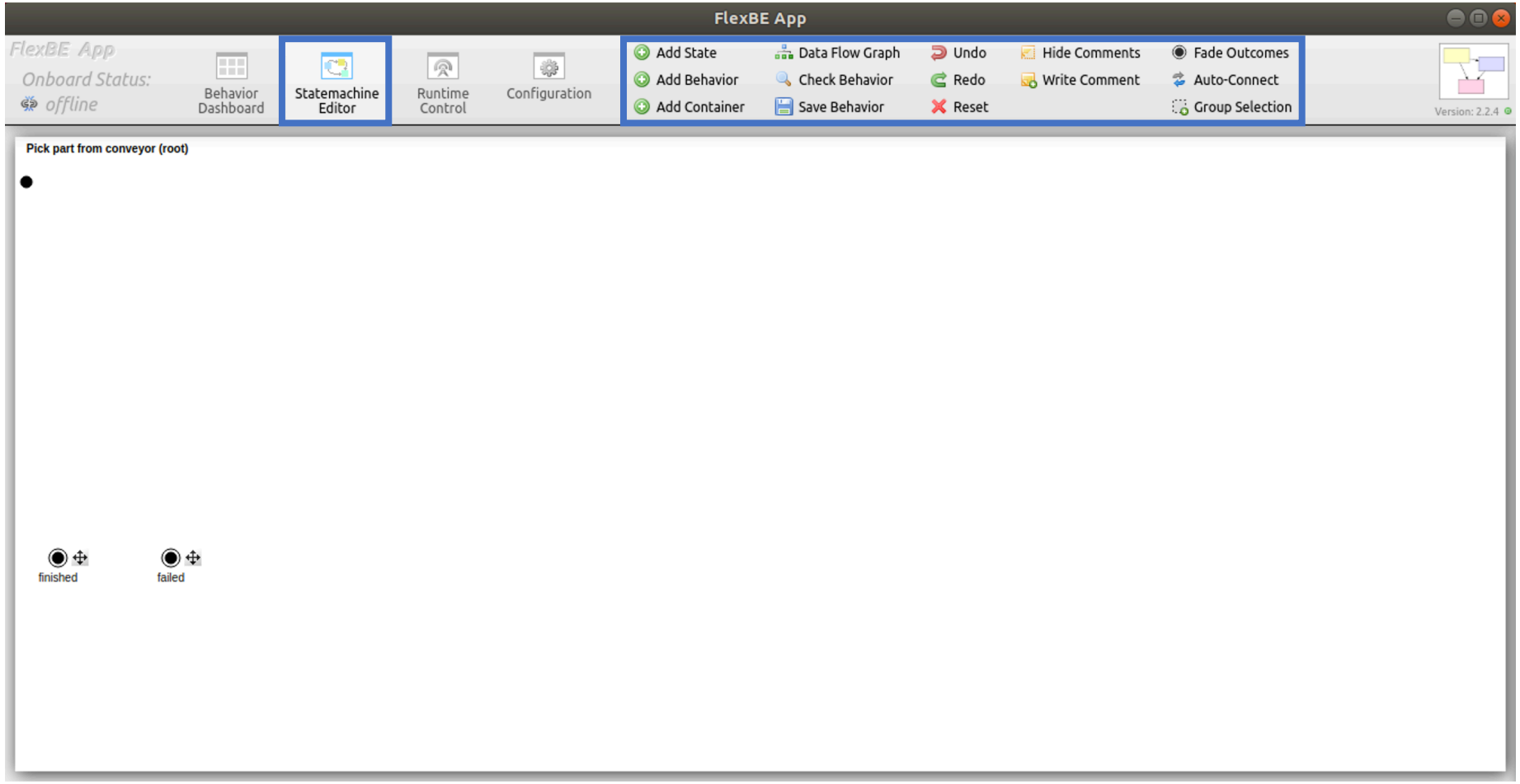


Creating new behavior



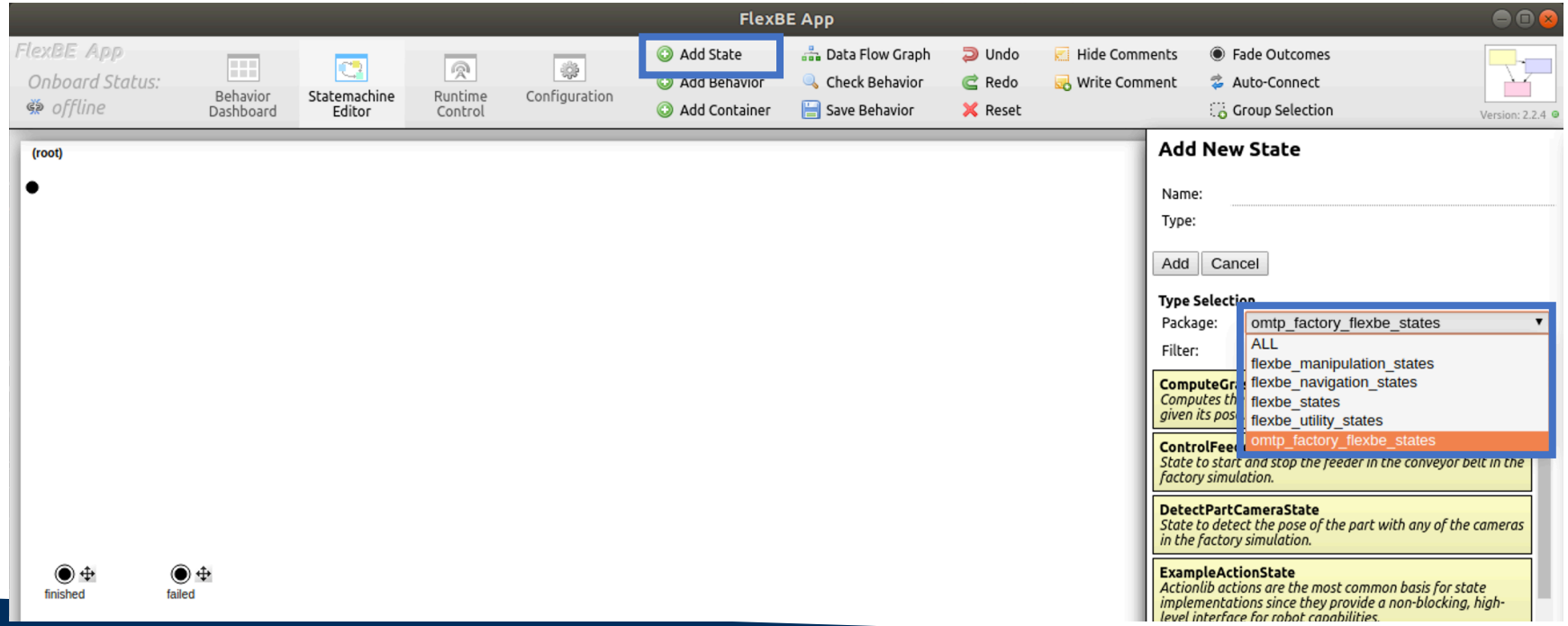
Statemachine Editor

Statemachine Editor



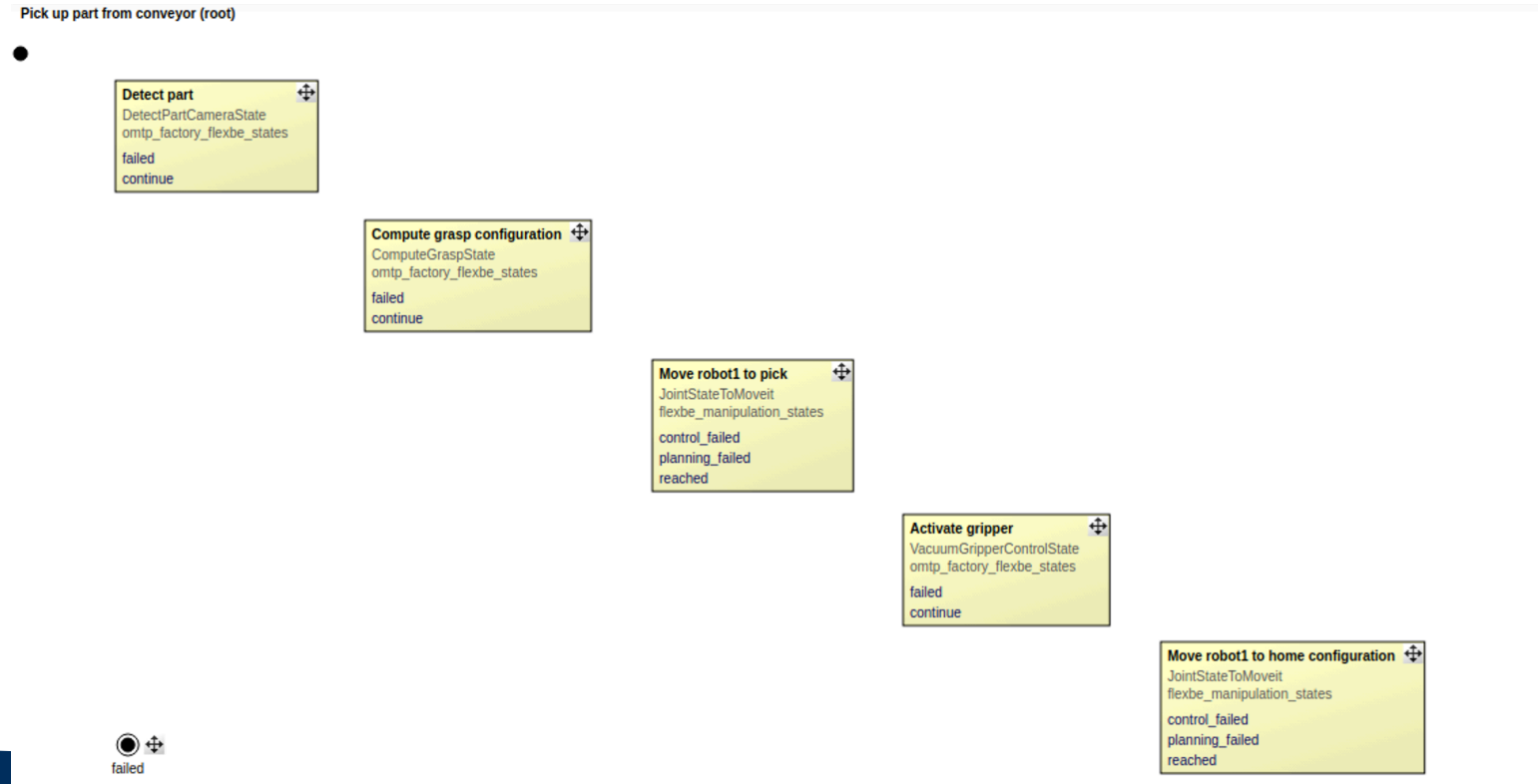
Adding a state

- Generic states (MoveIt, navigation etc.)
- Specific OMTP factory FlexBE states



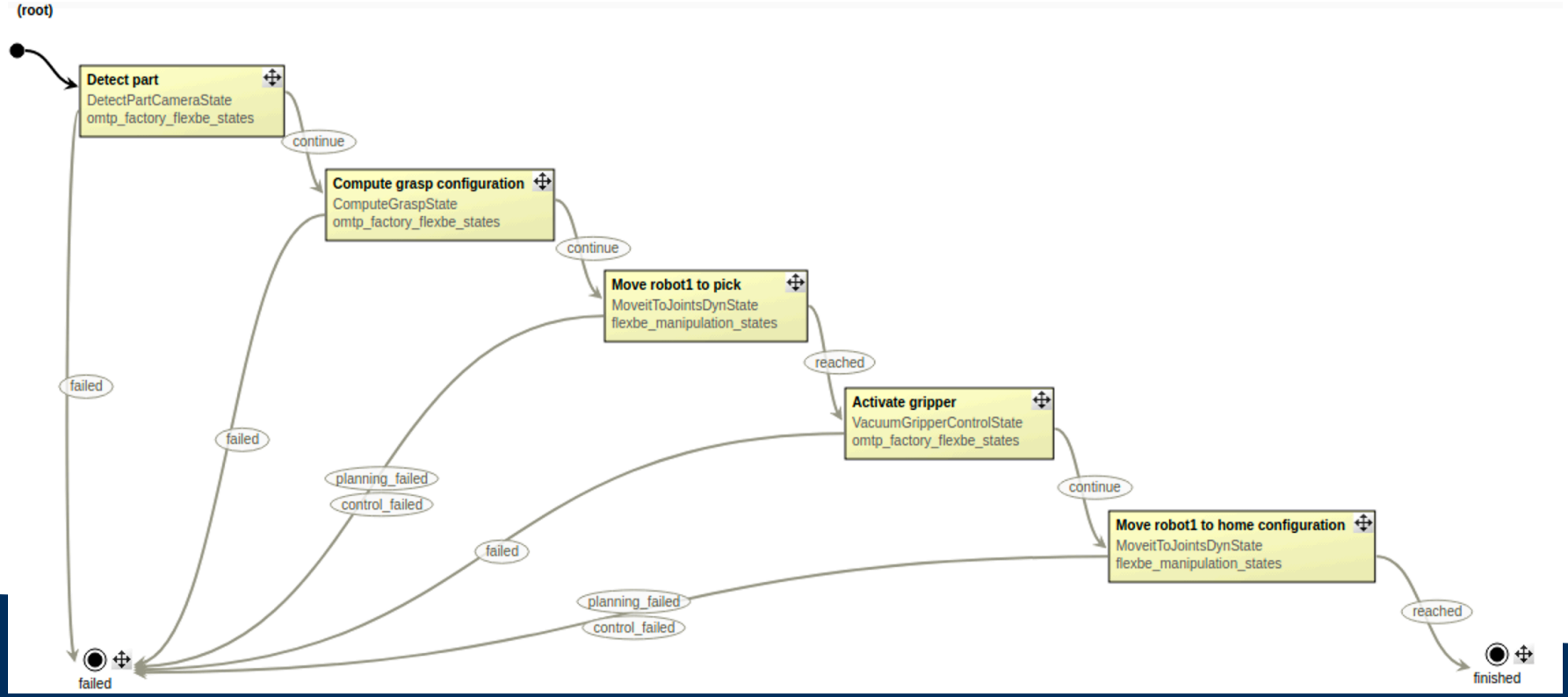
Sequence of states

Place the states in the execution sequence needed for the behavior



Adding state transitions

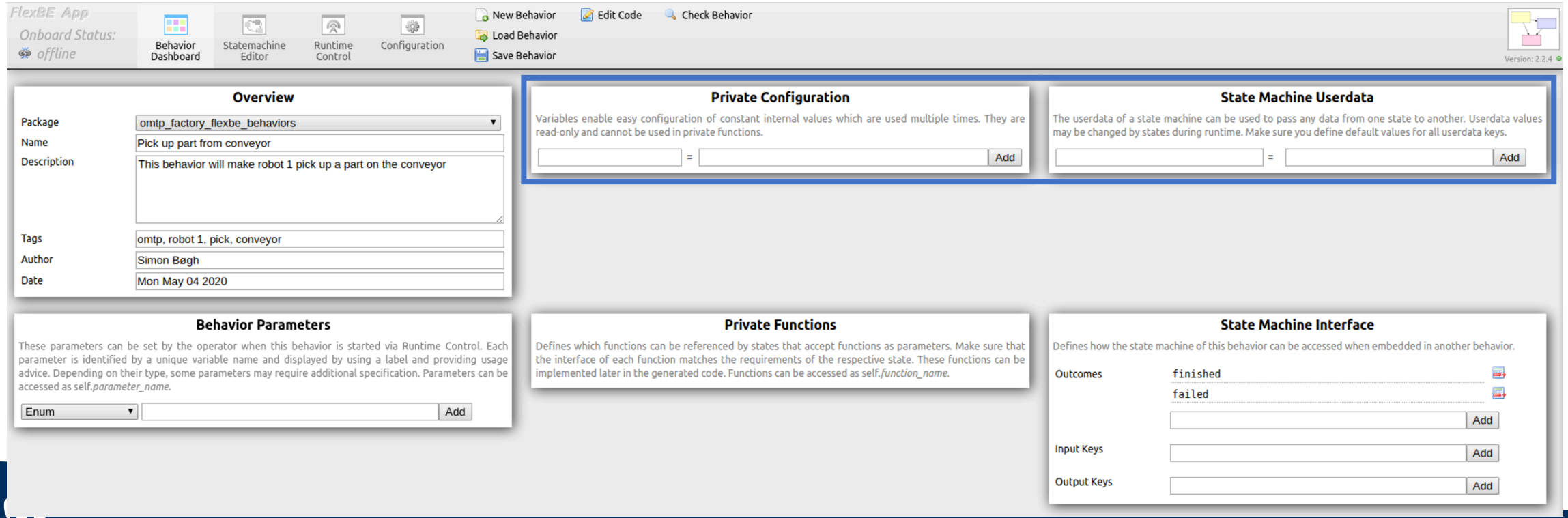
- Click to **connect the states** with state ***transitions***
- Some transitions go to **“failed”** others to the **next state** or **“finished”**



Configure Behavior

Behavior configuration

- Define **variables** for **constant data (read-only)** and **data that can change (userdata)**

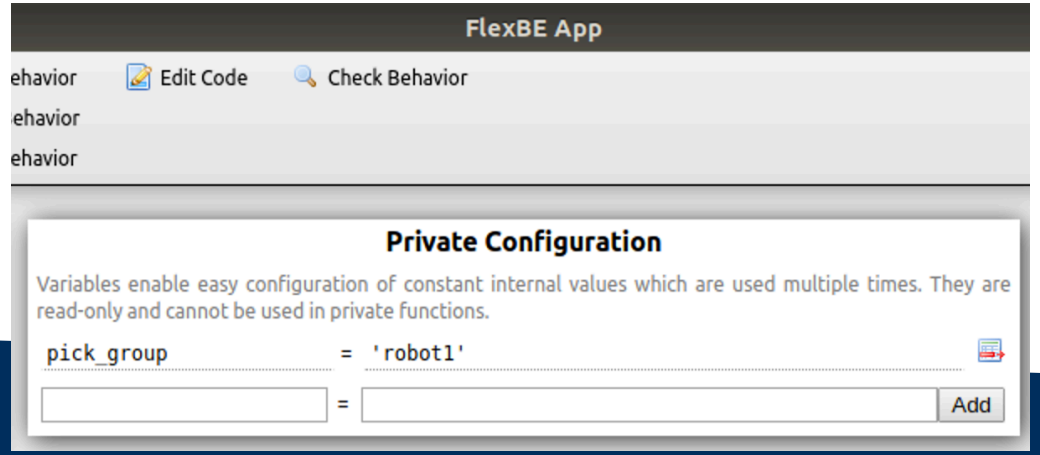


Constant (read-only)

- **Constant (read-only)** values used in the behavior
  - For example **”robot1”** in **omtp.srdf** generated by MoveIt Setup Assistant
  - Similar to a constant in programming languages

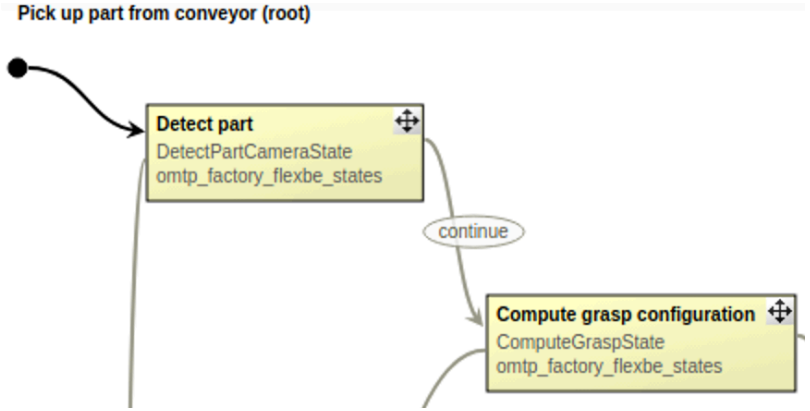
```
<group name="robot1">
    <chain base_link="robot1_base_link" tip_link="vacuum_gripper1_suction_cup" />
</group>
```

- In FlexBE, variables are defined in the **Private Configuration** section of the **FlexBE App**



Userdata

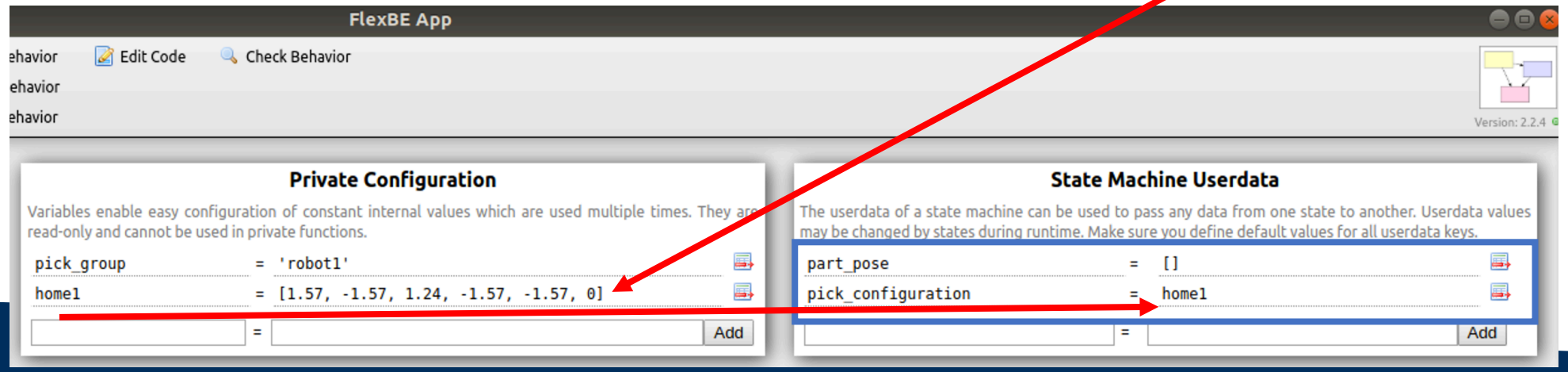
- Data can flow between states and **change at runtime**
- **Userdata** variables in FlexBE
  - **“Detect part” state** obtains pose of the part to pick by using the **logical camera**
  - The pose is used by the **“Compute grasp configuration” state** to obtain the joint configuration to pick the part
  - Two runtime variables are needed:
    - **Pose** of the part to pick
    - **Joint configuration** of where to move



Userdata

- Default values are required
  - **part_pose** can be an empty array []
  - **pick_configuration** we should set a safe value. If the default value is used, the robot would move to that position
    - The value is a constant, so let's define it

```
<group_state name="R1Home" group="robot1">
    <joint name="robot1_elbow_joint" value="1.57" />
    <joint name="robot1_shoulder_lift_joint" value="-1.57" />
    <joint name="robot1_shoulder_pan_joint" value="1.24" />
    <joint name="robot1_wrist_1_joint" value="-1.57" />
    <joint name="robot1_wrist_2_joint" value="-1.57" />
    <joint name="robot1_wrist_3_joint" value="0" />
</group_state>
```



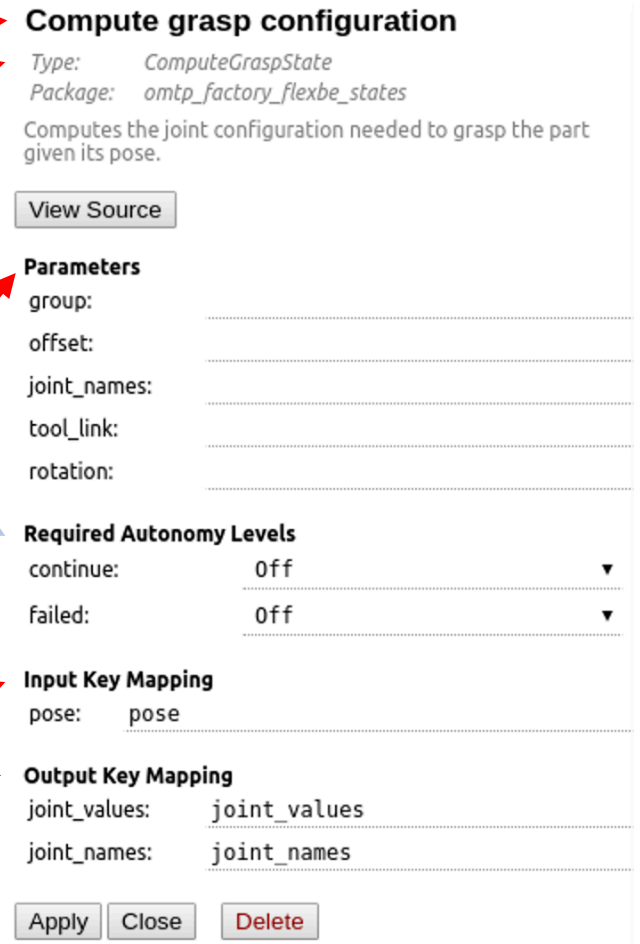
State properties

- FlexBE states
  - A **state encapsulates an atomic action** in the behavior (the smallest action)
  - ROS nodes provide **capabilities** through **topics, services and ROS actions**
  - We have to **compose states through these capabilities**

- A FlexBE **state**
  - Represents a **concrete use of** one or more of those **capabilities** to perform a **specific action**
  - **For example:**
    - The **move_group** has the **capability** to plan and execute motions for a robot manipulator
    - One **state** can be the action to **command the move_group** to execute a motion to reach a specific position

Properties of a FlexBE state

- Name
- State implementation: class *ComputeGraspState*
- Required Autonomy Levels (not covered)
  - Allow an operator to activate states during behavior execution
- Interface to the state implementation
  - **Parameters**: properties that are constant during execution
  - **Input and Output Keys**: define the runtime data needed for input, and produced as output
  - **Outcomes**: e.g. “continue”, “failed” seen on the transitions arrows

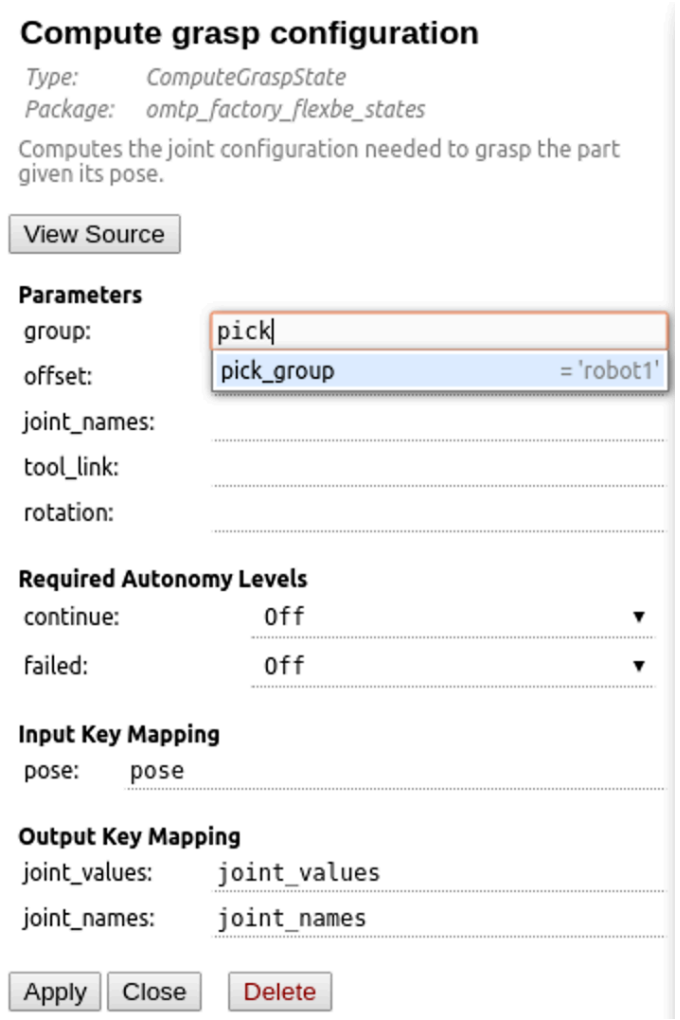


State parameters

- Configure the **constant** properties of the state for the behavior
- Use **quotation marks** to give a **literal** value: ‘robot1’

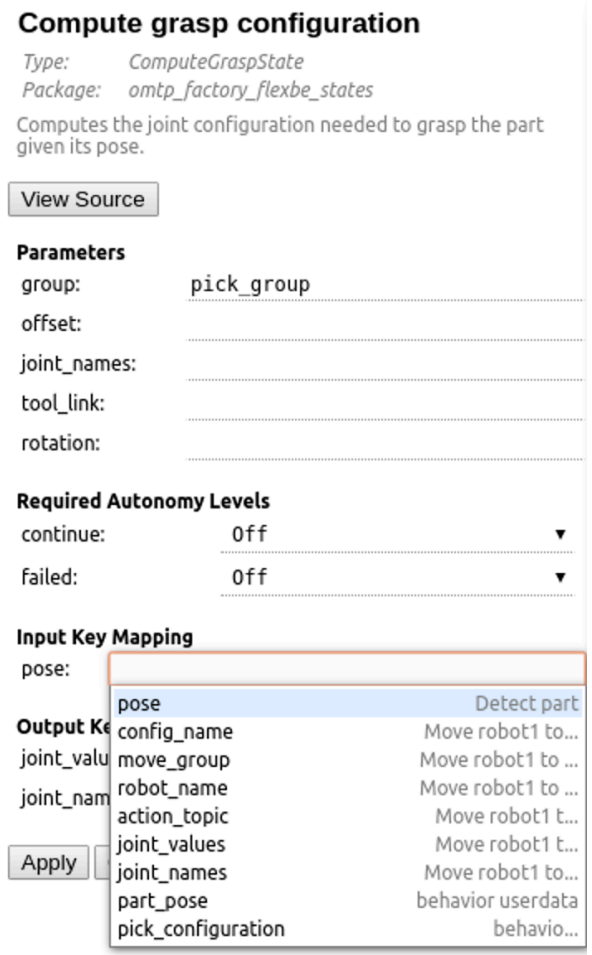
  or

- Use a behavior constant variable: *pick_group*
- **Takes a constant, NOT userdata**



Input and Output Key Mappings

- **Input** keys: data that the state **requires**
- **Output** keys: data that the state **produces**
- **Takes userdata** (runtime data) **NOT constants**



Input and Output Key Mappings

detect_object_on_conveyor_state

Type: *DetectPartCameraState*
Package: *omtp_factory_flexbe_states*

State to detect the pose of the part with any of the cameras in the factory simulation.

View Source

Parameters

ref_frame:	'robot1_base_link'
camera_topic:	'/omtp/logical_camera'
camera_frame:	'logical_camera_frame'

Required Autonomy Levels

continue:	Off
failed:	Off

Output Key Mapping

pose:	object_pose

Apply Close Delete

compute_grasp_of_object

Type: *ComputeGraspState*
Package: *omtp_factory_flexbe_states*

Computes the joint configuration needed to grasp the part given its pose.

View Source

Parameters

group:	'robot1'
offset:	0.25
joint_names:	['robot1_elbow_joint','robot1_sho
tool_link:	'vacuum_gripper1_suction_cup'
rotation:	3.1415

Required Autonomy Levels

continue:	Off
failed:	Off

Input Key Mapping

pose:	object_pose

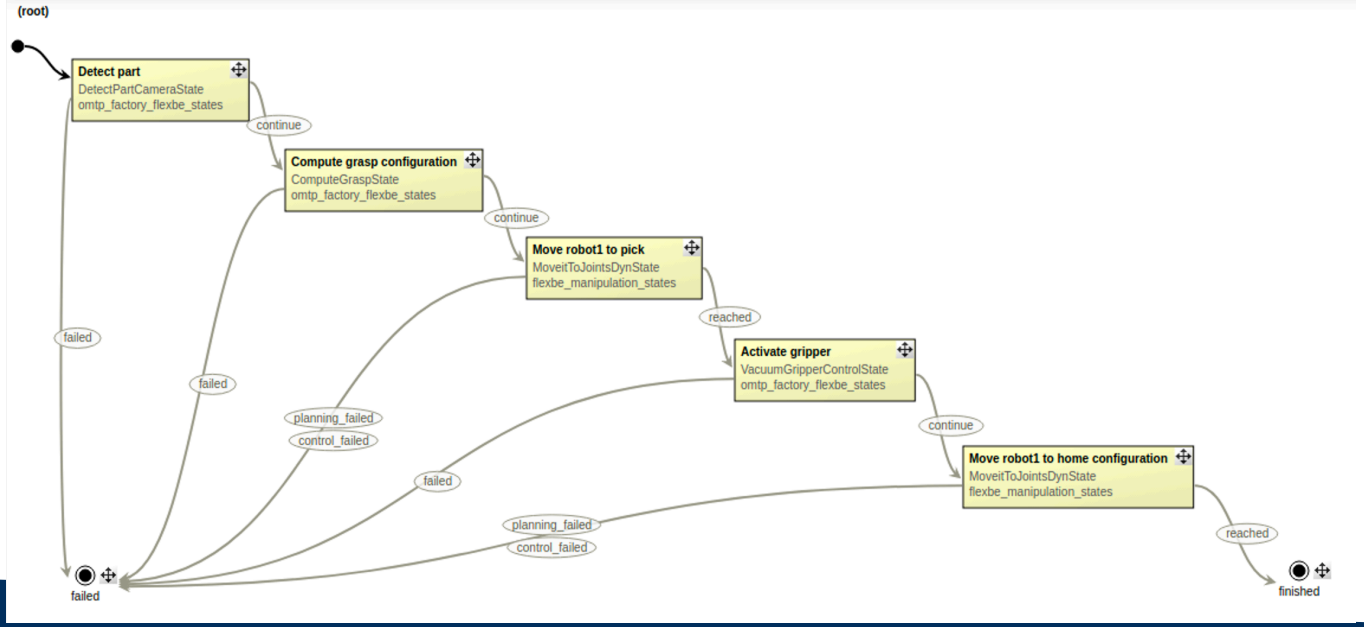
Output Key Mapping

joint_values:	grasp_joint_values
joint_names:	grasp_joint_names

Apply Close Delete

State Outcomes

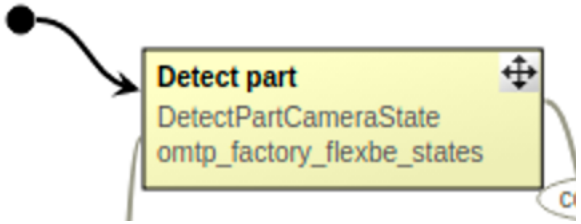
- Define possible **results of execution**: **“continue”**, **“failed”** …
  - They define the possible transitions from the state
  - Shapes the possible execution flow in the state machine



Configure the Remaining States

Detect part state

- ref_frame: reference frame we want the pose
- pose: variable where the detected part pose should be sent



Detect part

Type: *DetectPartCameraState*
Package: *omtp_factory_flexbe_states*

State to detect the pose of the part with any of the cameras in the factory simulation.

View Source

Parameters

ref_frame:	'robot1_base_link'
camera_topic:	'/omtp/logical_camera'
camera_frame:	'logical_camera_frame'

Required Autonomy Levels

continue:	Off
failed:	Off

Output Key Mapping

pose:	part_pose

Apply Close Delete

Computer grasp configuration state

- Already configured earlier
- Remember to make **userdata variable** for **joint_names**

Compute grasp configuration

Type: *ComputeGraspState*
Package: *omtp_factory_flexbe_states*

Computes the joint configuration needed to grasp the part given its pose.

View Source

Parameters

group:	pick_group
offset:	0.25
joint_names:	['robot1_elbow_joint','robot1_sho
tool_link:	'vacuum_gripper1_suction_cup'
rotation:	3.1415

Required Autonomy Levels

continue:	Off
failed:	Off

Input Key Mapping

pose:	object_pose

Output Key Mapping

joint_values:	pick_configuration
joint_names:	joint_names

Apply Close Delete

Private Configuration

Variables enable easy configuration of constant internal values which are used multiple times. They are read-only and cannot be used in private functions.

pick_group	=	'robot1'
home1	=	[1.57, -1.57, 1.24, -1.57, -1.57, 0
robot1_joint_na	=	['robot1_elbow_joint','robot1_shoul
robot1_tool_lin	=	'vacuum_gripper1_suction_cup'

= Add

State Machine Userdata

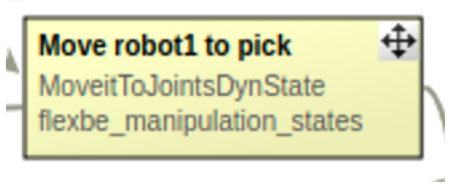
The userdata of a state machine can be used to pass any data from one state to another. Userdata values may be changed by states during runtime. Make sure you define default values for all userdata keys.

part_pose	=	[]
pick_configuration	=	home1
joint_names	=	[]

= Add

Move robot1 to pick state

- **move_group**: robot1 planning group. Saved as pick_group variable
- **action_topic**: topics where MoveIt is listening to action calls
- **joint_values**: pick_configuration variable defined earlier



Move robot1 to pick

Type: *MoveitToJointsDynState*
Package: *flexbe_manipulation_states*

Uses MoveIt to plan and move the specified joints to the target configuration.

View Source

Parameters

move_group:	pick_group
action_topic:	'/move_group'

Required Autonomy Levels

reached:	Off
planning_failed:	Off
control_failed:	Off

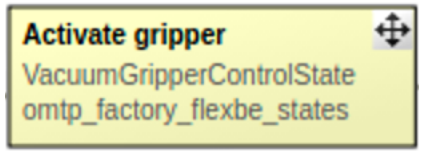
Input Key Mapping

joint_values:	pick_configuration
joint_names:	joint_names

Apply Close Delete

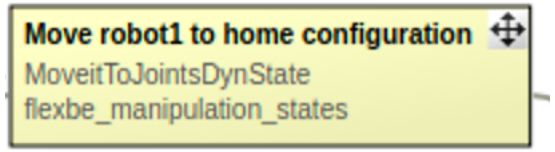
Activate gripper state

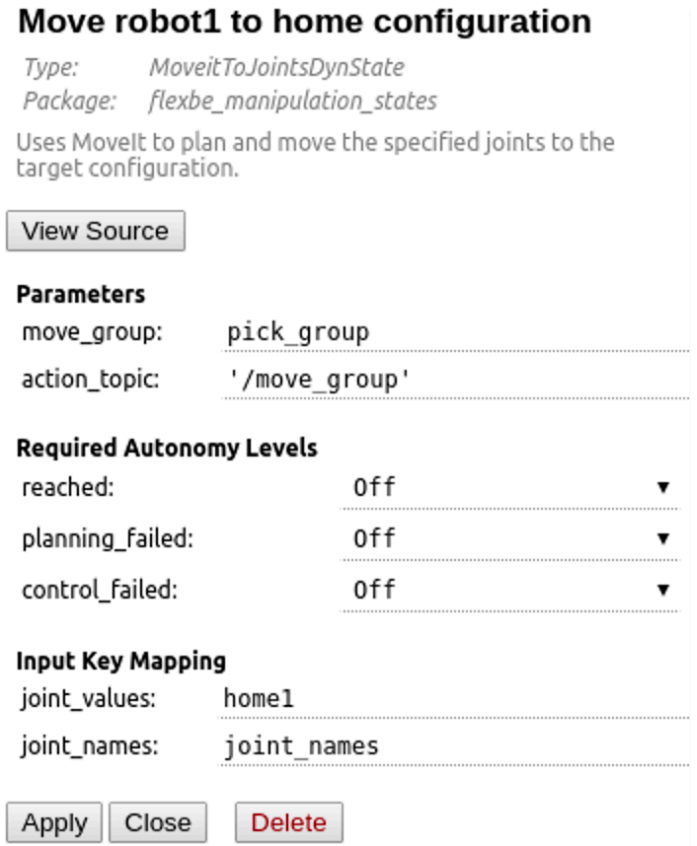
- **enable**: boolean to activate the gripper, we set to ‘true’
- **service_name**: topic name for the gripper control

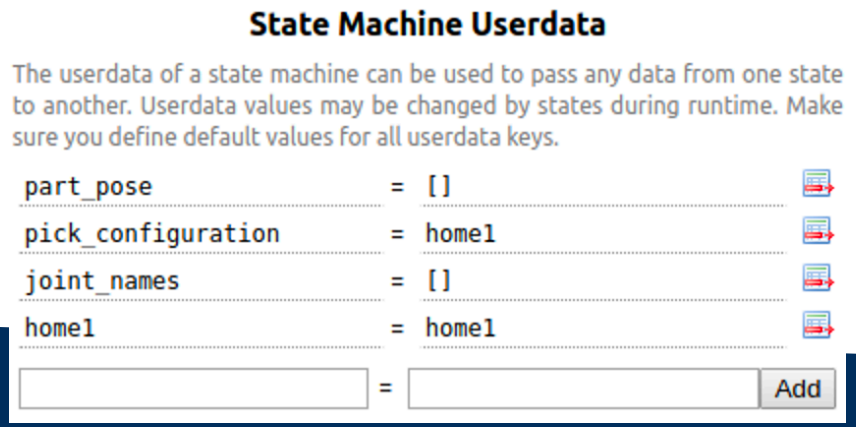


Move robot1 to home configuration state

- Same as the other MoveIt state configured
- **joint_values**:
  - Makes sense to use the constant home1
  - **BUT! State keys only support Userdata variable** (data that can change) to support data that may change during run-time

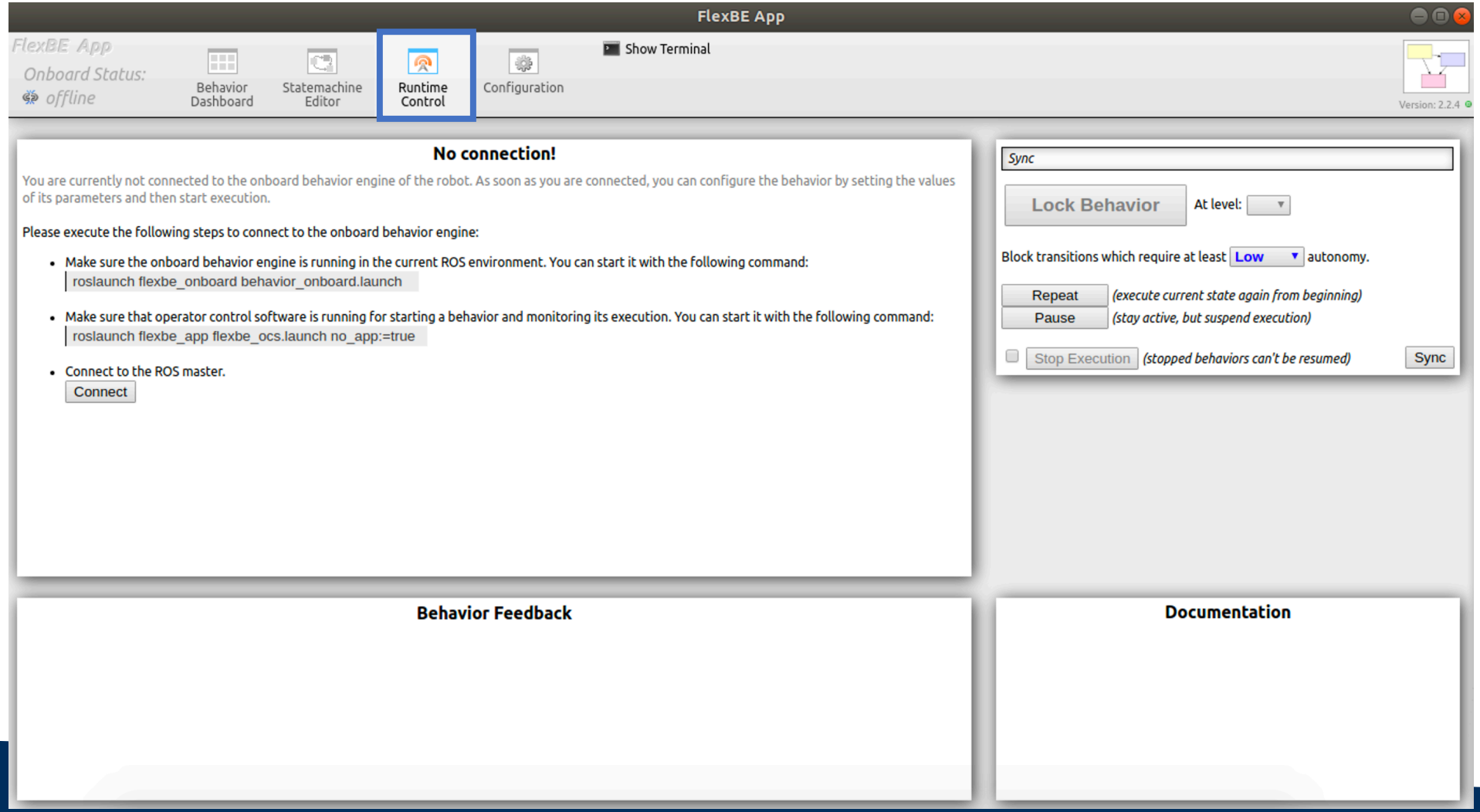






Executing a Behavior with FlexBE

Runtime Control (offline)



Launch FlexBE for behavior execution

- The **FlexBE App** only allows us to **make and edit behaviors**
- We **need a connection to the engine** and our **OMTP environment** to execute the behavior

- Launch **OMTP Factory**

```
$ roslaunch omtp_gazebo omtp_pick_demo.launch
```

- Launch the whole **FlexBE behavior engine** *(close the FlexBE App first)*

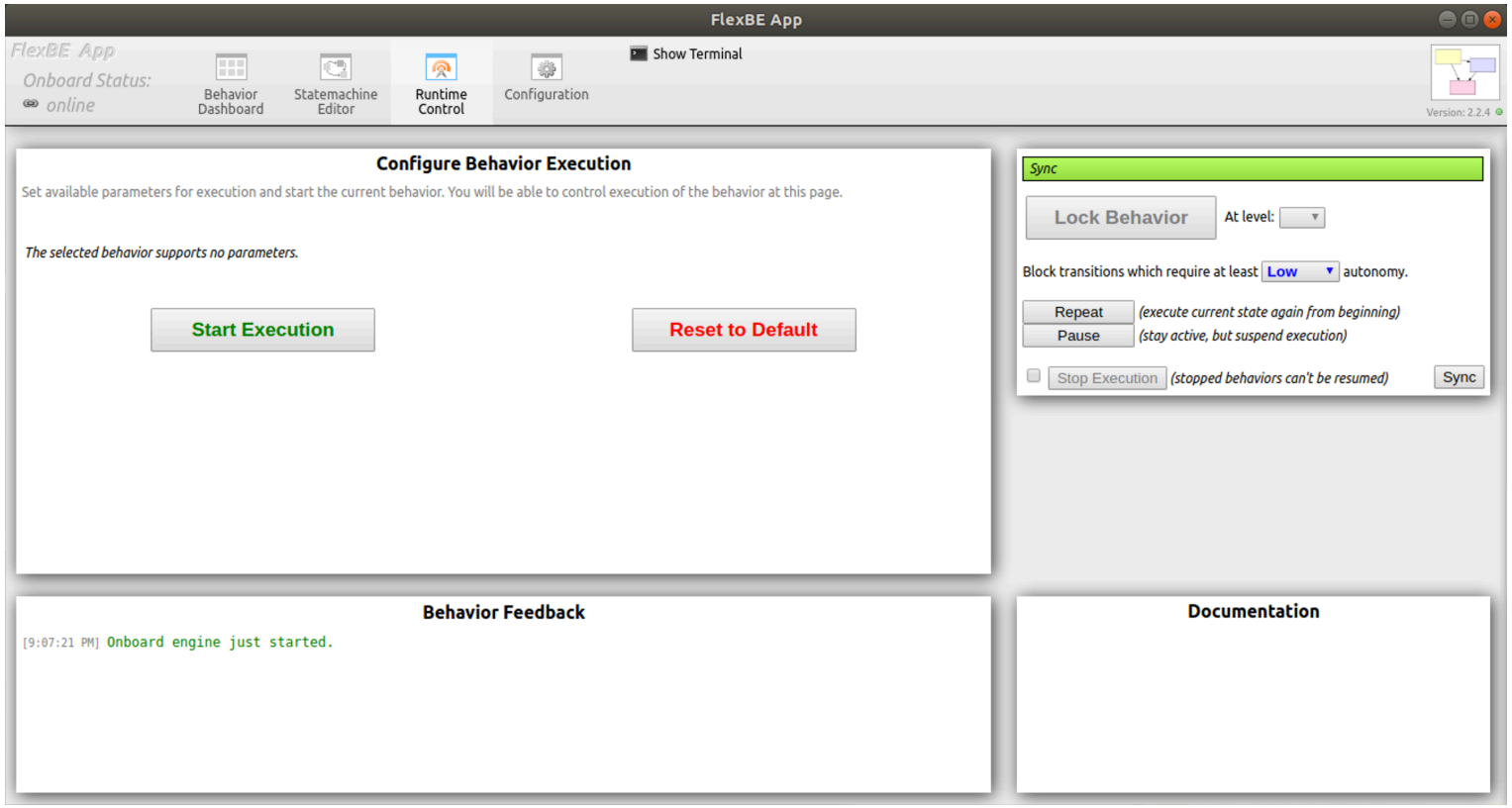
```
$ roslaunch flexbe_app flexbe_full.launch
```

- **Spawn object** on conveyor

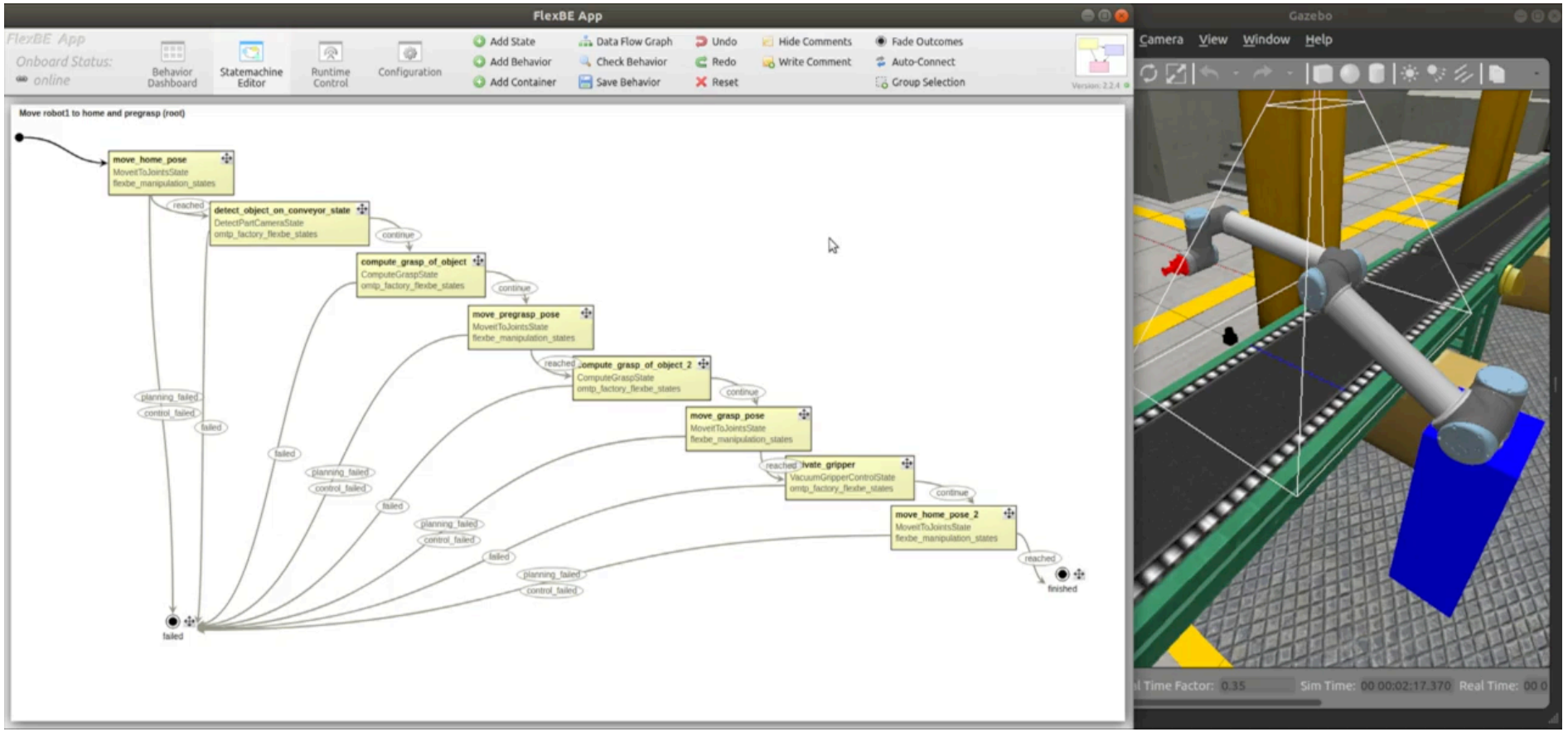
```
$ rosservice call /start_spawn
```

Runtime Control (online)

- Load your behavior
- Press “Start Execution”



Behavior execution



https://youtu.be/zIgJwyTCqbY

Create State Implementation

Controlling the conveyor

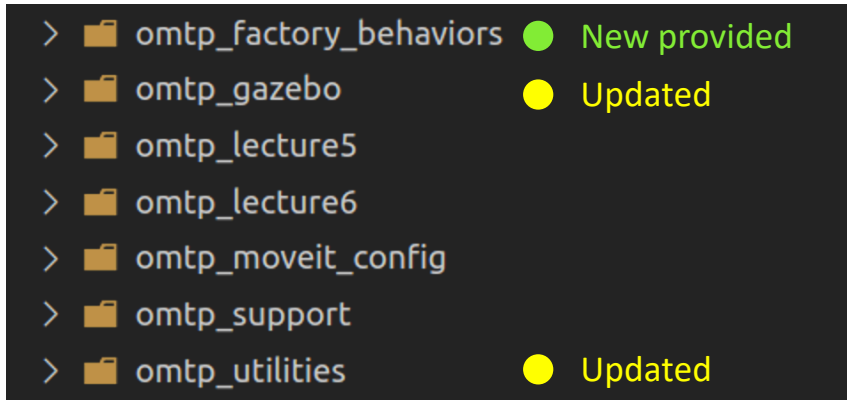
Program a FlexBE state

Example: set_conveyor_power_state.py (primary functions)

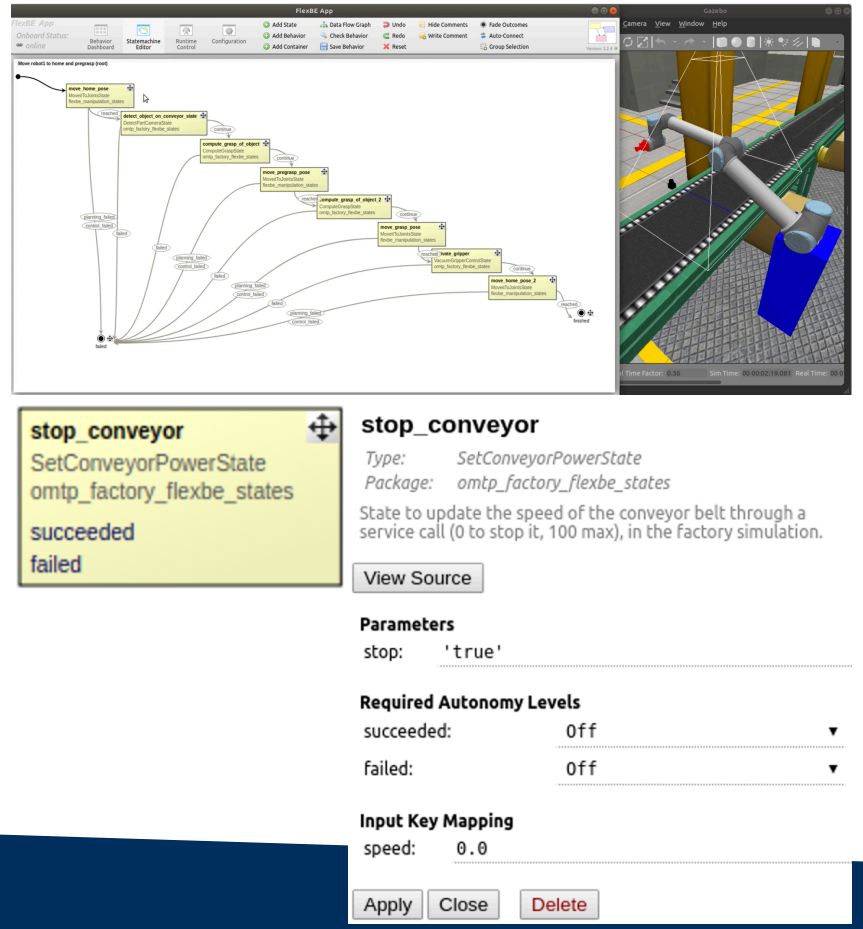
```
class SetConveyorPowerState(EventState):
      '''
      State to update the speed of the conveyor belt through a service call (0 to stop it, 100 max), in the factory simulation.

      -- stop        bool    If 'true' the state instance stops the conveyor belt, ignoring the speed inputkey

      >= speed       float   Value to set the speed of the conveyor belt.

      <= succeeded           Speed of the conveyor belt has been succesfully set.
      <= failed              There was a problem setting the speed.
      '''


      def __init__(self, stop):
          # Declare outcomes, input_keys, and output_keys by calling the super constructor with the corresponding arguments.
          super(SetConveyorPowerState, self).__init__(outcomes = ['succeeded', 'failed'], input_keys = ['speed'])


      def execute(self, userdata):
          # (periodic) check conditions to trigger outcomes
      def on_enter(self, userdata):
          # start state actions, init var
      def on_exit(self, userdata):
          # Stop running processes
      def on_start(self):
          # (once) initialize resources
      def on_stop(self):
          # Clean up
```

Assignments

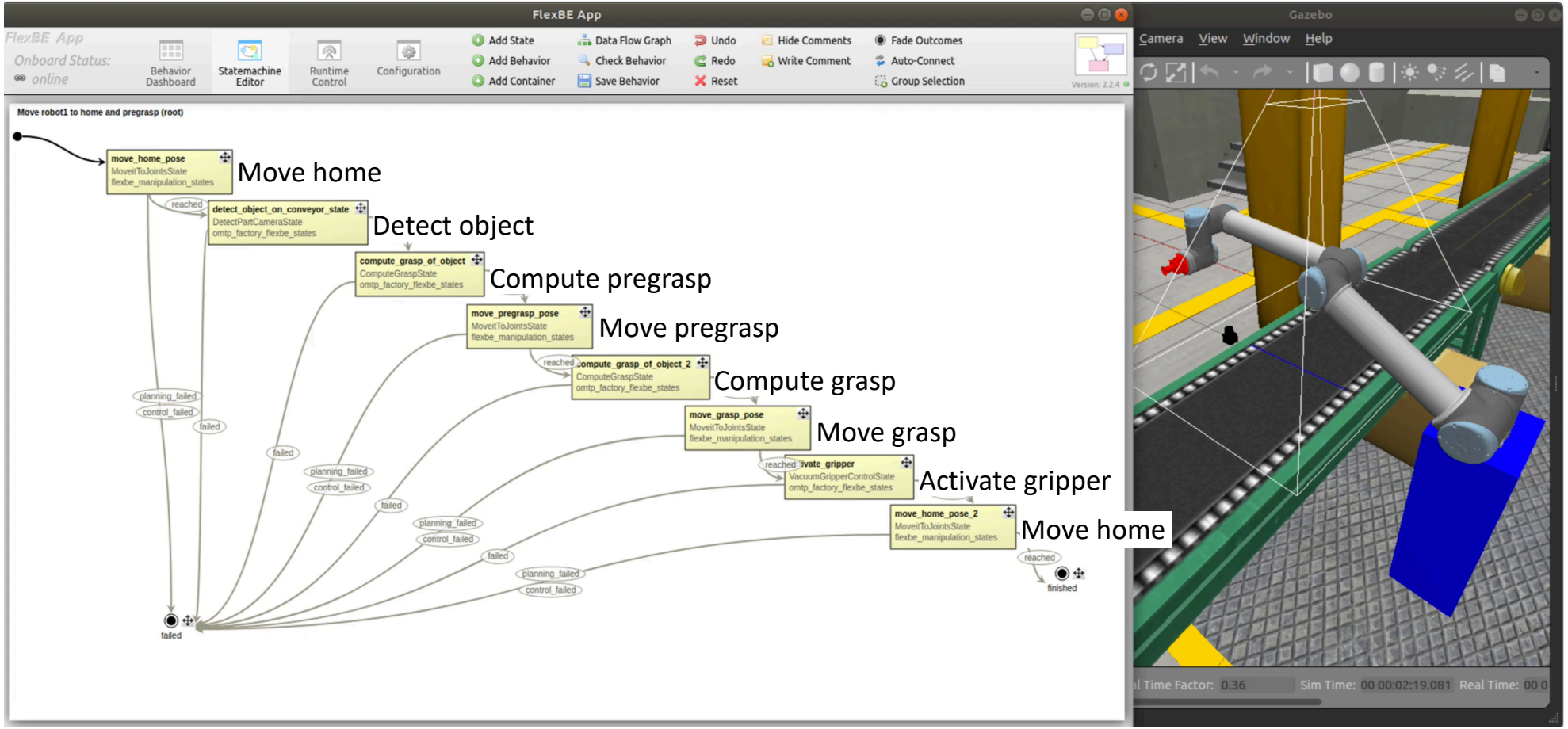
Assignments



1. **Design a robot behavior in FlexBE including the conveyor state**
   - The goal for this assignment is two-part
     a) Build the pick and place pipeline similar to the one on the next slide
     b) Use the conveyor state to bring an object to the area where the camera can detect it and where robot1 can pick it up
   - Open-ended: design the pipeline as you see fit



Example of pick and place behavior

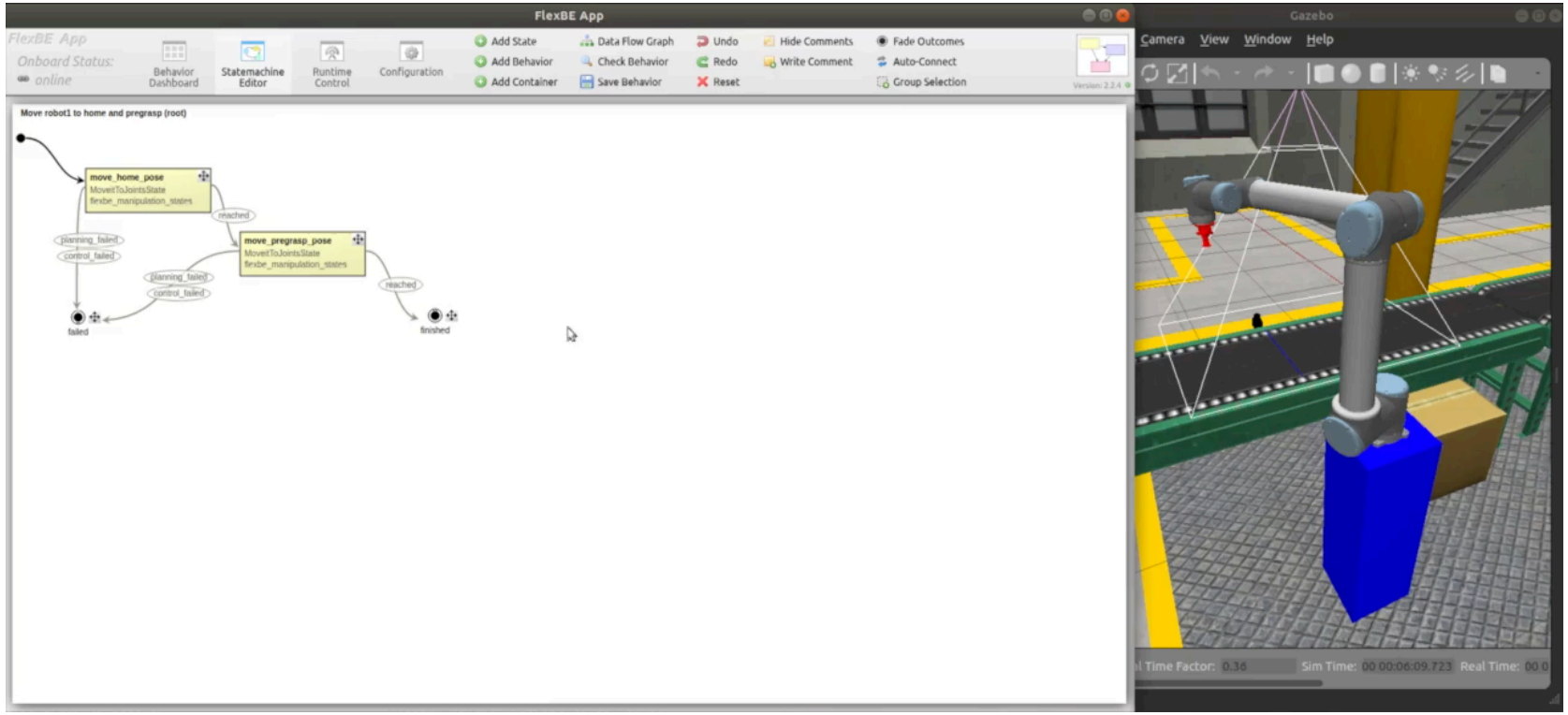


Useful commands

1. **Make your own folder with behaviors and states (this folder is also provided)**
   - `$mkdir omtp_factory_behaviors`
   - `$ cd omtp_factory_behaviors`
   - `$ rosrun flexbe_widget create_repo omtp_factory`
   - `$ catkin build`
   - `$ source ~/omtp_course_ws/devel/setup.bash`
   - **NOTE:** this will create a git-repo as well. Delete the hidden file .git, so you do not have a repo inside a repo
   - **Download additional generic states: generic_flexbe_states** (https://github.com/FlexBE/generic_flexbe_states)
     - A useful collection of non-robot-specific states, e.g., for **move_base**, **MoveIt!**, **PCL**, **OpenCV**
2. **Make behavior (offline)**
   - Launch FlexBe App: `$ rosrun flexbe_app run_app`
3. **Execute behavior (online)**
   - Launch OMTP Factory: `$ roslaunch omtp_gazebo omtp_pick_demo.launch`
   - Launch the whole FlexBE behavior engine (close the FlexBE App first): `$ roslaunch flexbe_app flexbe_full.launch`
   - Spawn object on conveyor: `$ rosservice call /start_spawn`
   - Load and execute the behavior in the FlexBE App

Simple move behavior

- Start out with a simple behavior, move robot1 to two poses



Submit assignment to git repo

- Submit assignment to git repo and post on MS Teams
- Update your README.md
- Add a small video/gif showing FlexBE executing the behavior in Gazebo
  - Note: compress video with handbrake and save as mp4

Extras

Notes for reference

$ rosservice type /compute_ik | rossrv show

moveit_msgs/MoveItErrorCodes error_code

- int32 SUCCESS=1
- int32 FAILURE=99999
- int32 PLANNING_FAILED=-1
- int32 INVALID_MOTION_PLAN=-2
- int32 MOTION_PLAN_INVALIDATED_BY_ENVIRONMENT_CHANGE=-3
- int32 CONTROL_FAILED=-4
- int32 UNABLE_TO_AQUIRE_SENSOR_DATA=-5
- int32 TIMED_OUT=-6
- int32 PREEMPTED=-7
- int32 START_STATE_IN_COLLISION=-10
- int32 START_STATE_VIOLATES_PATH_CONSTRAINTS=-11
- int32 GOAL_IN_COLLISION=-12
- int32 GOAL_VIOLATES_PATH_CONSTRAINTS=-13
- int32 GOAL_CONSTRAINTS_VIOLATED=-14
- int32 INVALID_GROUP_NAME=-15
- int32 INVALID_GOAL_CONSTRAINTS=-16
- int32 INVALID_ROBOT_STATE=-17
- int32 INVALID_LINK_NAME=-18
- int32 INVALID_OBJECT_NAME=-19
- int32 FRAME_TRANSFORM_FAILURE=-21
- int32 COLLISION_CHECKING_UNAVAILABLE=-22
- int32 ROBOT_STATE_STALE=-23
- int32 SENSOR_INFO_STALE=-24
- int32 NO_IK_SOLUTION=-31
- int32 val