Modular Benchmarking Framework Documentation

Shadow Robot Company

Aug 19, 2020

Contents

This is the starting point for the Modular Benchmarking Framework Documentation. **The documentation is still in progress!**

CHAPTER 1

Motivation

Robotic manipulation is a high level and complex task which combines a range of different components. Although one might want to compare the overall system, it becomes hard to dissociate and evaluate each component of a manipulation pipeline, and therefore to obtain a fair comparison between different specific solutions. Hence, we propose the Modular Benchmarking Framework, a ROS-based framework that enables to: 1) easily benchmark different solutions with minimal coding and integration effort and 2) separately compare each component of a pipeline, which is essential to drive progress in this area. Robot arms with a ROS interface and manipulators can be integrated without too much effort, allowing to save effort and time that could be spent on developing algorithms to solve real world problems. This framework also allows easy integration of a wide range of methods for manipulation without caring about which part is robot-specific, allowing researchers that are relatively new to robotics to easily experiment their methods.

CHAPTER 2

Contents

2.1 Installing the framework

Our software is deployed using Docker. Docker is a container framework where each container image is a lightweight, stand-alone, executable package that includes everything needed to run it. It is similar to a virtual machine but with much less overhead. Follow the instructions in *this section* to get the latest Docker container up and running.

2.1.1 Hardware specifications

In order to run our software and the ROS software stack you will need to meet some hardware requirements.

- CPU: Intel i5 or above
- RAM: 4GB or above Hard Drive: Fast HDD or SSD (Laptop HDD can be slow)
- Graphics Card: Nvidia GPU (optional)
- OS: Ubuntu 16.04 or 18.04 (Active development)

The most important one is to have a fast HDD or an SSD.

2.1.2 Installing the framework

We have created a one-liner that is able to install Docker, download the image and create a new container for you. To use it, you first need to have a PC with Ubuntu installed on it (16.04 or 18.04 tested).

Prerequisite

We **strongly** advise to run the one-liner on a machine without any version of docker installed. If you have never installed it, you can skip to the next subsection. If you are already using doccker, please be aware that the resulting container *might* not work. If it is the case, you can run the following lines:

```
sudo apt purge -y docker-engine docker docker.io docker-ce
sudo apt autoremove -y --purge docker-engine docker docker.io docker-ce
```

These instructions are uninstalling docker but should not remove any of the containers already stored on your machine.

You have a nvidia card

If the machine you are going to use to run the framework has a Nvidia card **and the nvidia drivers are on**, then run the following line

```
bash <(curl -Ls bit.ly/run-aurora) docker_deploy product=hand_e nvidia_docker=true␣
↪tag=kinetic-nvidia-release reinstall=true sim_icon=false image=shadowrobot/modular_
↪benchmarking_framework container_name=<container_name>
```

You can change `<container_name>` by the name you want to give to the container that you are going to use.

If you have a nvidia card but are running on the xorg drivers, go to the other subsection!

You don't have a nvidia card

If you don't have a nvidia graphic card or if you do and don't use the nvidia drivers, please run

```
bash <(curl -Ls bit.ly/run-aurora) docker_deploy product=hand_e nvidia_docker=false␣
↪tag=kinetic-release reinstall=true sim_icon=false image=shadowrobot/modular_
↪benchmarking_framework container_name=<container_name>
```

You can change `<container_name>` by the name you want to give to the container that you are going to use.

2.1.3 Future releases

For now, the docker that you have downloaded contains Ubuntu 16.04 and ROS Kinetic. We are currently working on the release of the framework using Ubuntu 18.04 and ROS Melodic. Stay tuned!

2.2 Configuring the framework

The Modular Benchmarking Framework allows a wide range of configuration at different levels: the **config files**, the **task_constructor** and the **launch file**. So all you need to modify is located in the **API** package. *The framework should natively be configured to run the EZGripper in simulation without any modification.*

2.2.1 Config files

All the config files are stored in the **config** folder of the *modular_framework_api* package. They gather all the different components that you might want to change in order to run your robot, without requiring any code. We will provide a detailed description of each of them.

Hardware connection

This YAML file must contain all the *hardware_interface* that ROS should be aware of when using the framework on the physical robot. If you are not familiar with this concept, you can read more about it on this tutorial. The first parameter in the YAML file, **robot_hardware** should contain the list of all the *robot hardware* that are going to be defined in the same file. Each of the specific hardware interface must be derived from `RobotHW`. The framework will then automatically create a combined hardware interface. The YAML file must then specify for each hardware interface the different parameters that must be loaded on the ROS param server in order to run the hardware. The parameters are implementation-dependent. **The framework provides natively a compatible hardware interface for Universal Robot arms**. The source code can be found on github. For our implementation, the following parameters must be filled:

- `robot_id`: Allow to load a robot which name consists of a prefix (usually given in the URDF file)
- `robot_ip_address`: Robot's IP address
- `control_pc_ip_address`: IP address of the machine that is controlling the robot arm
- `speed_scale`: Robot arm speed scale [0.0 - 1.0]
- `payload_mass_kg`: Manipulator weight (initial payload of the robot)
- `payload_center_of_mass_m`: Estimated manipulator's payload centre of inertia
- `topic_to_wait_for`: Name of the topic published by the hardware interface to notify that the controllers can be loaded
- `robot_program_path`: Path to the directory of Universal Robot's program used to communicate with the arm.

Obviously, some parameters (even if renamed) must absolutely be loaded, such as the robot and pc ip addresses. In any case, please make sure to specify the `type` field if you are using another robot arm or if you prefer using your own implementation. An example of hardware connection config file for a UR5 only can be found here.

Generative methods parameters

Whatever the method used to generate grasps, joint states, trajectories or poses, they rely on some parameters provided by the user. If you are using the framework for benchmarking, you may not want to modify the same information across the different files but instead centralize them in a single YAML file loaded on the ROS param server and access them through the ROS API. This file can be filled with anything the user needs to load and access in different nodes without hardcoding it and that is susceptible to change. For instance, the one we provide contains information about the name of the origin frame and the values specifying a cropping area. You can also add, for example, the path to a neural network that needs to be evaluated.

Manipulator controller

Since we want to make this framework usable for everyone, we designed it so you can control the manipulator with custom made controllers. This way, if your hardware is not fully integrated to ROS or you control the hardware through some third-party drivers, you can take the most of our framework. It is possible to do so by creating an action-based controller. Whether it is in simulation or with the physical robot, this config file allows you to specify how to control the manipulator. Here are the different parameters:

- `manipulator_group_name`: If you are using Moveit, name of the group given to the manipulator.
- `manipulator_controller_action_server_name`: If you are using your own action-based controller, you can specify its name here.
- `package_manipulator_action_server`: If you are using you own action-based controller, specify the package in which it is located.

- `manipulator_controller_node_name`: If you are using your own action-based controller, specify the name of the node running it.
- `manipulator_controller_action_server_type`: Type of the node in which the action server is launched (can be a .py file if written in python).

If you are using controllers integrated to ROS and exported as plugin, set `package_manipulator_action_server` as `""`, and don;t pay attention to the two following parameters.If you want to find out if your robot is *fully integrated* to ROS or how to create the action-based controller, you can go here. If you are working in **simulation** and/or your manipulator can be controlled using Moveit, you can use our generic Moveit-based action server that we provide as an example of grasp controller. If your controller requires more parameters, you can of course add them to this config files (such as `max_torque` for instance).

Connection to an external manipulator

This config file gathers all parameters required to connect a manipulator that is **not** fully integrated in ROS and/or is not using an ethernet connection. If the manipulator only has an USB interface and you do not want to change the port address everytime in the code, you can store all this information on the ROS param server. The file natively provided gives an example of the information needed to communicate with the EZGripper.

Motion Planner config

This config file allows to configure the motion planner that is going to be used when planning for a specific group during your task. For now, you can choose any motion planner among this list. All of the parameters of this configuration file **must be filled**! .

- The string corresponding to `planner_name` must exactly match the name of the planner defined in your moveit package config. If you are using what we provide, the name should match what is declared in this file.
- `robot_speed_factor` allows to scale the speed of the robot when executing a trajectory (value between 0 and 1)
- `arm_group_name` specifies the group for which all the other parameters are going to be set. The group name must match the group defined in the srdf file! An example of a srdf file with some groups defined can be found here.
- `number_plan_attempt` is the number of times the plan is computed from scratch before the shortest solution is returned.
- `planning_max_time` specifies the maximum amount of time before timing out when planning.

/! Please note that **for now** the planner **must** be part of the **OMPL** planner manager and is going to be used by Moveit. We are working on creating an interface so that home made motion planners can be integrated */!*

If you are interested in planner benchmarking, you might be interested in this.

Known joint states

Most of the time, even when running highly automated methods, we know some *waypoints* of the robot (e.g starting joint state of the robot). For instance when cleaning a surface by picking objects, they are usually all dropped at the same pre-defined location. Instead of hardcoding them in the code, you can gather them in the config file called **named_joint_states.yaml**. They must be defined with the following rule:

```
waypoint1_name:
 joint_name_1: <value_11>
 joint_name_2: <value_21>
```

(continues on next page)

(continued from previous page)

```
 ...
 joint_name_n: <value_n1>

waypoint2_name:
 joint_name_1: <value_12>
 joint_name_2: <value_22>
 ...
 joint_name_n: <value_n2>

...
```

Just make sure that the given names in this YAML file are **unique** (i.e not giving twice the same name to two joint states). You can however define joint states for **different groups** in the same file. The joint states defined here can also be used to automatically create trajectories (please see *here*).

Known trajectories

In some cases, part of the task performed by the robot is known in advance and we are expecting the robot to always repeat a given trajectory. For instance, after grasping an object at an unknown pose, go to pose A through pose B and C. For such cases, you can predefined some trajectories based on the joint states that are defined in **named_joint_states.yaml**. The framework can then retrieveand execute this known trajectory by its name. The different trajectories can be configured to match a given timeline and must be declared following this rule:

```
trajectory1_name:
  - {name: <first_point_name>, interpolate_time: <value_1>, pause_time: <pause_time_1>
↪}
  - {name: <second_point_name>, interpolate_time: <value_2>, pause_time: <pause_time_
↪2>}
  ...
  - {name: <last_point_name>, interpolate_time: <last_value>, pause_time: <last_pause_
↪time>}

trajectory2_name:
  - {name: <point_1_name>, interpolate_time: <value_1>, pause_time: <pause_time_1>}
  - {name: <point_2_name>, interpolate_time: <value_2>, pause_time: <pause_time_2>}
  ...
  - {name: <point_n_name>, interpolate_time: <value_n>, pause_time: <pause_time_n>}

...
```

The parameters `interpolate_time` and `pause_time` are in seconds and allow you to control the *flow* of the trajectory. The first one constrains the trajectory to be at the given waypoint after `interpolate_time` from previous waypoint. The other parameter allows to pause the robot in a specific pause before going to the next waypoint. We believe these two parameters are enough to create a wide range of behaviours with respect to speed and so on. An example of trajectory can be found here.

Sensor plugins

If you are using a specific sensor which outputs should be considered when performing motion planning, you can develop your own plugin so that Moveit integrates the sensor's values in the planning process. The documentation avout how to create your plugin can be found here. If you are using point clouds or depth maps, you can use already existing plugins. If you are developing your own plugin for a given sensor, make sure to add it within the `sensors` field according to the format of the provided example. You can also add all the different parameters required to run the different plugins in this file (loaded in the namespace `/move_group/<your_param>`).

Sensors config

Adding sensors to the environment requires to declare them in the tf2 tree. Instead of creating a node for each sensor in which we have to modify the different parameters everytime the scene changes a bit, you can provide the different values in this config file and the different transforms will be published for you. The only thing you need to do is to declare each sensor with the following format:

```
sensor_name_1:
  frame_id: <frame_id_1>
  origin_frame_id: <origin_1>
  frame_x: <x_coordinate_1>
  frame_y: <y_coordinate_1>
  frame_z: <z_coordinate_1>
  frame_roll: <roll_1>
  frame_pitch: <pitch_1>
  frame_yaw: <yaw_1>

sensor_name_2:
  frame_id: <frame_id_2>
  origin_frame_id: <origin_2>
  frame_x: <x_coordinate_2>
  frame_y: <y_coordinate_2>
  frame_z: <z_coordinate_2>
  frame_roll: <roll_2>
  frame_pitch: <pitch_2>
  frame_yaw: <yaw_2>

...
```

Please note that if any of the fields is missing, the frame will not be added and the corresponding transform won't be published. You can also add all the parameters that you want to use in other nodes and you do not want to hard-code since they might change, such as the name of the topics on which data is published. **You can add sensors that are attached to moving links as well.** For example, if your add to this file a new sensor of which origin is `eg_manipulator`, with any coordinates, you will se that when the robot is moving the frame is changing as well. Which allows for instance to integrate wrist mounted cameras or tactile sensors easily. The following YAML file depicts a setup with a dynamic and a static camera:

```
fixed_kinect:
  frame_id: "simulated_kinect2"
  origin_frame_id: "world"
  frame_x: -0.1645
  frame_y: 0.375
  frame_z: 2.72
  frame_roll: 3.14159265359
  frame_pitch: 0
  frame_yaw: -1.57079632679

wrist_mounted_realsense:
  frame_id: "simulated_realsense"
  origin_frame_id: "eg_manipulator"
  frame_x: 0.
  frame_y: -0.075
  frame_z: -0.165
  frame_roll: 0.436332313
  frame_pitch: 0
  frame_yaw: 0
```

2.2.2 Task Constructor

The different files contained in the folder *task_constructor_script* describe through a simple and intuitive *YAML* interface the task the robot should perform. Given some *states* representing atomic actions, we believe it possible to create very complex and non-linear behaviours that can be used to solve real world problems. A flexible way to connect these states to create use cases, while controlling the behaviour when facing an issue, is using state machines. For this purpose we rely on the smach library. We created a layer that simplifies the construction of the state machines, so that users without any knowledge regarding state machines, smach or even ROS can program the robot to perform some specified tasks. We provide a set of states allowing to perform most of the generic tasks related to manipulation, but that can be used for other use cases. The user only needs to link the states together in a YAML file. This interface allows to easily create a wide range of use cases and complex closed-loop behaviours just by changing the transitions between the states. You can nest state machines inside others in order to condense more complex behaviours. We provide two task constructor scripts as examples. One is defining a pick and hold on a physical robot, the other one can be used in simulation for picking objects based on a given Grasp Pose Detection method. **Note that the file's fieldname must match with the argument `state_machine_to_load` the launch file.** A more in-depth tutorial about the task constructor and the provided states can be found here.

2.2.3 Launch file

This launch file contains other important information that are required to start the framework. We are going to review all the different arguments that you can set to make the framework fit what you need.

- `simulation`: If set to `false`, automatically launches the framework to communicate with the physical robot. Default is `true` (and run Gazebo 9).
- `description_package`: Name of the ROS package containing *scenes*, *worlds* and *models* you may want to use to set up your environment. You can find out more about what is a description package here.
- `robot_package`: Name of the ROS package containing information about the robot such as the *urdf file* and the *controller file*. You can find out more about what is a robot package here.
- `robot_urdf_file`: Name of the urdf file containing the description of the **whole** robot. It must contain both the arm(s) and the manipulator(s). The file should be contained in the `robot_package`.
- `world_file`: Name of the Gazebo **.world** file describing the robot's setup for simulation. An explanation about how to create such files can be found here.
- `scene_file`: Name of the file containing all the information about the collisions so Moveit can plan according to the different obstacles. An explanation about how to create such files can be found here.
- `urdf_args`: Optional arguments that you may need to pass along with the urdf file (for instance to set the position of the robot). **If not needed, leave empty!**
- `moveit_config_package`: Name of the moveit config package required to operate the robot with Moveit. A tutorial about how to create one using the assistant is available here. If you don't manage to properly create it you can follow this tutorial.
- `controller_file`: Name of the file containing the different controllers that must be loaded by ROS. The file should be contained in the `robot_package`.
- `simulation_starting_pose`: **Used only in simulation** and defines the values of each joints in Gazebo when starting the robot.
- `manipulator_prefix`: String that is contained in all the manipulator's link. For instance, it can be `rh`.
- `states_directory`: Path of the directory containing the states that you want to use to create your state machine.
- `templates_directory`: Path of the directory containing the templates used to generate the different state machines you might need

- `state_machine_to_load`: Specify the name of the state machine to run (should end with *.py*). Leave empty if you want to generate a new one.
- `task_constructor_script`: Name of the yaml script to use for generating the state machine that is going to be used. **Not used** if `state_machine_to_load` is not empty.
- `generated_state_machine_name`: Name of the file containing the generated state machine (should end with *.py*). **Not used** if `state_machine_to_load` is not empty.

Modifying the parameters in the configuration file and in the launch file allows setting up the robot (and its environment) so they can be used by the framework. Having new states and new *task constructor scripts* allow creating different use cases in a flexible way without requiring an extensive knowledge about state machines. With all these options, we believe that you can configure the framework to fit your needs. If not, you can have a closer look to the core and try to modify it.

2.3 Integrating a robot to the framework

As stated in the previous section, the framework natively supports Universal Robot arms and is provided with all the packages required to operate (both in simulation and on physical setup) an EZgripper attached to an UR5 arm. It is nevertheless likely that you are working with a different hardware. We are hence going to detail how to integrate different kind of hardware.

2.3.1 Common steps

The first step is to create an urdf (or xacro) file in which both the arm(s) and the manipulator(s) are defined such as here or here. The second step is to create a *moveit_config* package for the robot. You can either use the Moveit! Setup Assistant or follow this step by step tutorial.

2.3.2 Using your robot in simulation

Now that you have your moveit config package ready, the only remaining step is to make it compatible with Gazebo. In order to use ros controllers with your robot, you need to add some elements to your urdf file. Gazebo uses a specific element that links actuators to joints, called `<transmission>`. Each of these elements must contain **at least**:

- `<joint name= >`: which corresponds to the name of a defined joint in your urdf file that you want Gazebo to be able to actuate.
- `<type>transmission_interface/SimpleTransmission</type>`: which specifies the type of transmission. More information here.
- `<hardwareInterface>` that should be present in both `<joint>` and `<actuator>` tags. It states *gazebo_ros_control* plugin what hardware interface to load (position, velocity or effort interfaces).

The last step is to add the *gazebo_ros_control* plugin to the urdf file. The plugin should look like this:

```
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <!-- Optional - Default being the value of robot name in the urdf/sdf file-->
    <robotNamespace>my_name_space</robotNamespace>
    <!-- Optional - The period of the controller update (in seconds), default is␣
↪Gazebo's period -->
    <controlPeriod>my_value</controlPeriod>
    <!-- Optional - The location of the robot_description (URDF) on the parameter␣
↪server, default being '/robot_description' -->
```

(continues on next page)

(continued from previous page)

```
    <robotParam>my_param_value</robotParam>
    <!-- Optional - The pluginlib name of a custom robot sim interface to be used␣
↪(see below for more details), default being 'gazebo_ros_control/DefaultRobotHWSim' -
↪->
    <robotSimType>my_value</robotSimType>
  </plugin>
</gazebo>
```

If you want to use your own controller, you can follow this tutorial and change the proper parameters. Now that your robot can be (in simulation at least) actuated using ROS controllers, you need to specify them both in the `controllers.yaml` file of your moveit config package and in the *controller file*. Now you should be able to control, in simulation, your robot with Moveit and see that the motions planned and executed in Rviz match with one happens in the Gazebo window launched when the framework is being run in simulation mode.

2.3.3 How to know if my manipulator is *fully integrated to ROS*?

If you are using a manipulator that can be actuated using ROS controllers, then it means that your manipulator is fully integrated to ROS. It also means that your robot has a hardware interface. If you have a doubt about that, try to find out if you have any file containing the definition of classed derived from `hardware_interface::RobotHW`. If you do, you can directly go to *this subsection*, otherwise please read the following subsection.

2.3.4 Using a physical manipulator that does not have a ROS hardware interface

If you are using a manipulator that cannot be operated using ROS controllers, no worries you can still integrate your hardware and make it work with the framework. It is highly likely that a ROS node launched in one of your launch files is performing the control and send the information directly to the robot's drivers. If you have developed it you know where it is, otherwise you just need to find it and isolate it. In order to integrate it to the framework, you need to transform it to an action server. You will then be able to control it as you wish by sending action requests that you can pre-empt and get feedback if anything goes wrong. You can find an example here. Once you have the action server that is able to control the manipulator, change the different options in `manipulator_controller_parameters.yaml` so your node can be launched (**do not forget to** `chmod +x` **if it's a python file**). Now you can create a second *controller file* in which you will only have the ROS controller defined for the robot arm (like this). In `start_framework.launch`, change the name of the controller file to be used to point to the one you just created. In order to avoid having bothersome warning and error messages (everything should still work though), do not forget to **comment out** the reference to the controller for your manipulator in `controllers.yaml` of your moveit config package. And here you are, you should now be able to control the manipulator via the framework. In order to test it, you can create a minimalist action client and after launching the framework, run the client in order to send any command to the manipulator. It should now move.

2.3.5 Using a physical manipulator that does have a ROS hardware interface

If you already have a ROS hardware interface for your manipulator, you might need to slightly change it. The only constraints are that the given hardware interface must have a **parameter-less** constructor and must have an `init` method with the following signature: `init(ros::NodeHandle &n, ros::NodeHandle &robot_hw_nh)`. You can find an example of hardware interface here. You can now use the standard ROS controllers on the manipulator. If you opt for this solution, do not forget to specify it in the *controller file* and to possibly change the nature of the controller in `controllers.yaml` of your moveit config package. You should now be able to control the physical robot with Moveit. The action server node provided in the framework can be used (but you can use your own if you wish). If you want to use a more convoluted controller for specific use cases, you can still create it. The class must be derived from `controller_interface::ControllerBase` and contain an action server. You can then create a plugin so the controller can be recognized by ROS and use it freely. An example is given here. Once again, define it

in the *controller file* and comment out any reference to the manipulator in `controllers.yaml` of your moveit config package. Once the framework launched, you should be able to control the manipulator through a very minimalist action client.

2.4 Using the task constructor

One of the goal of the Modular Benchmarking Framework is to be able to separate each component used for solving a problem related to robotic manipulation. This way, we can easily re-use any component that seems powerful in a given context to another and study how good or bad it can be transferred or combined with other components. In order to do so, we need to be able to easily create a wide range of tasks the robot should execute independently of what hardware it is or which method is being used. Then, we can thoroughly compare the different metrics obtained during the experiments and conclude that, for instance, grasp synthesis method A is better than method B given a specific motion planner, controller and so on for a specific task.

2.4.1 Principle

We believe that most of the tasks related to manipulation can be represented as a (complex) combination of *atomic* actions, such as *planning*, *predicting a pose*, *generating a trajectory* and so on. Considering such *simple* actions and linking them together has several benefits. The first one is to be able to create more complex and task-oriented behaviours through the links between the actions. The other one is that we can link each component of a method to one or several actions and therefore allow to analyse the *real* performance of each component individually and test different combinations. Considering each action to be a *state*, then a task can be defined as a state machine.

2.4.2 Designing a state machine

We propose an intuitive way to create state machines, which does not require any knowledge about ROS or how to implement state machines using smach. The only thing you need to do is to describe in a YAML file which states (or state machines) you want to use and how to link them, that's all. The YAML file should be stored inside the folder `task_constructor_scripts` of the *modular_framework_api* package.

Format of the YAML files

Each task constructor script (YAML file) must be structured the following way:

```
name: <state_machine_name>
source: <template_filename>
node_name: <name_of_node_running_state_machine>
outcomes: <list_of_state_machine_outcomes>
states:
  - <state1>:
      source: <state_filename>
      <option_1_state1>: <value_option_1_state1>
      <option_2_state1>: <value_option_2_state1>
      ...
      <option_n_state1>: <value_option_n_state1>
      transitions: {<state_outcome1>: <state2>, <state_outcome2>: <a_state_or_state_
↪machine_outcome>}
  - <state2>:
      source: <state_filename>
      <option_1_state2>: <value_option_1_state2>
      <option_2_state2>: <value_option_2_state2>
```

(continues on next page)

(continued from previous page)

```
      ...
      <option_n_state2>: <value_option_n_state2>
      transitions: {<state_outcome3>: <state_machine_1>, <state_outcome4>: <a_state_
↪or_state_machine_outcome>}
  - <state_machine_1>:
      source: <template_filename>
      <option_1_state_machine_1>: <value_option_1_state_machine_1>
      <option_2_state_machine_1>: <value_option_2_state_machine_1>
      ...
      <option_n_state_machine_1>: <value_option_n_state_machine_1>
      transitions: {<state_outcome3>: <another_state>,  <state_outcome3>: <a_state_or_
↪state_machine_outcome>}
      states:
        - <state11>:
            source: <state_filename>
            <option_1_state11>: <value_option_1_state11>
            <option_2_state11>: <value_option_2_state11>
            ...
            <option_n_state11>: <value_option_n_state11>
        ...
        - <state_n1>:
            source: <state_filename>
            <option_1_state_n1>: <value_option_1_state_n1>
            <option_2_state_n1>: <value_option_2_state_n1>
            ...
            <option_n_state_n1>: <value_option_n_state_n1>
  ...
  - <staten>:
      source: <state_filename>
      <option_1_state1>: <value_option_1_state1>
      <option_2_state1>: <value_option_2_state1>
      ...
      <option_n_state1>: <value_option_n_state1>
      transitions: {<state_outcome_n>: <a_state_or_state_machine_outcome>, <state_
↪outcome_n+1>: <a_state_or_state_machine_outcome>}
```

Let's go over the above template and explain the different parts.

Script header

The header **must** be composed of:

- `name`: Name you want to give to the file containing the generated state machine
- `source`: Name of the *template file* that is going to be used to generate the state machine. It **must** be contained in the `templates_directory` arguments passed in the launch file.
- `node_name`: Name you want to give to the node that is going to run the generated state machine
- `outcomes`: List of different outcomes of the root state machine. For instance `[success, failure]`.
- `states`: Defines all the states or state machines that you are going to need to create your task.

Using a state

Each state you want to use should be defined as follow:

```
- <state_name>:
    source: <state_filename>
    <state_option_1>: <state_option_1_value>
    <state_option_2>: <state_option_2_value>
    ...
    <state_option_n>: <state_option_n_value>
    outcomes: <list_of_outcomes>
    transitions: {<outcome_name>: <state_or_outcome>, <outcome_name>: <state_or_
↪outcome>}
```

The can give any name to your states, we just advise you that they are meaningful in order to get a hang of what the task is about just by looking at the name of the differents states.

- `source` **must** be the name of the python file in which the state is implemented. The file **must** be located in the `states_directory` argument of the launch file.
- As long as the states are properly formatted (see *here*) you can also define the different options directly in the YAML file.
- `outcomes` define the different outcomes of the states. It must be a list, such as `[success, failure]`.
- `transitions` **must** be specified as they are defining how to link the different states (or state machines) together. For each potential outcome of the state, specifies what should be done next. If you want the state called `DummyState` to follow your state if the latter outputs `success` then this field should be `{success: DummyState}`.

Nesting a state machine

Some components of a behaviour can themselve be designed as whole state machines. We can imagine a small state machine that plan and move the robot arm to its initial pose. You can directly create such nested state machines inside the task constructor script using the following template:

```
states:
  - ...
  - ...
  - <state_machine_name>:
      source: <template_filename>
      <state_machine_option_1>: <state_machine_option_1_value>
      <state_machine_option_2>: <state_machine_option_2_value>
      ...
      <state_machine_option_n>: <state_machine_option_n_value>
      transitions: {<outcome_name>: <state_or_outcome>, <outcome_name>: <state_or_
↪outcome>}
      states:
        - <state_machine_state_1_name>:
            source: <state_filename>
            <state_1_option_1>: <state_1_option_1_value>
            <state_1_option_2>: <state_1_option_2_value>
            ...
            <state_1_option_n>: <state_1_option_n_value>
        ...
        - <state_machine_state_m_name>:
            source: <state_filename>
            <state_m_option_1>: <state_m_option_1_value>
            <state_m_option_2>: <state_m_option_2_value>
            ...
            <state_m_option_n>: <state_m_option_m_value>
```

You can of course create several nested state machines in the same root state machine or created recursively nested state machines. The only restriction is that the source file of the nested state machines **must** be *template files* (you can find some here), and you **must** define the transitions in order to know what to do when the nested state machine is done. Please note that we already provide template files allowing to create a state machine **fully compatible** with the framework, as well as a concurrent state machine. In our implementation, one of the option for the state machine is to specify the information it should have access to (userdata). When using nested state machine, we often want to kind of *inherit* the same userdata as the parent. You can do it with `userdata: self.userdata`.

Miscellaneous

Although we tried our best to simplify the creation of state machines by preventing users to dive into different tutorials and face the numerous boilerplate that would come with it, defining a complex state machine might be a bit painful. That is why when *creating new states* (or using the one we provide) you can define some default values for the options that are unlikely to be changed and not specify them in the task constructor script. When creating state machines (especially nested ones), it might be a bit painful to copy/paste some parts such as the input/output keys that might be the same for several states. To simplify this it is possible, in a state or a state machine, to create a `params` field which **must** be a dictionary in which you can store values and reuse them in the script. An example of `params` could be `params: {outcomes: [success, fail]}`. If this field has been added to a state machine then all the children states can use `outcomes: params.outcomes`. Please note that the same principle can be applied within a given state in order to shorten the definition of a state and make (once you get used to it) the task constructor script more compact and readable.

Examples

In this subsection we are going to give some examples of state machines based on the provided states and the framework templates. The point is not to have meaningful examples in term of task but rather to demonstrate how to use the task constructor.

Making a task constructor script more compact

Let's start with the following task constructor script:

```
name: just_planning
source: framework_state_machine
node_name: just_planning_sm
outcomes: [sm_successful, sm_failed]
states:
  - PlanMotion:
      source: state_plan
      target_state_type: joint_state
      target_state_name: stability_pose
      plan_name: current_to_stability
      starting_state_type: ""
      starting_state_name: ""
      outcomes: [success_to_plan, fail_to_plan]
      input_keys: []
      output_keys: []
      io_keys: [arm_commander]
      transitions: {success: sm_successful, fail: sm_failed}
```

If you are launching the framework using this config file then you should have a state machine that plans from the current robot pose to the joint state named `stability_pose`. The state named `PlanMotion` showed above

contains all the possible parameters that can be changed in our implementation. Taking the most of the default values defined in the state, we could simply shorten it to

```
name: just_planning
source: framework_state_machine
node_name: just_planning_sm
outcomes: [sm_successful, sm_failed]
states:
  - PlanMotion:
      source: state_plan
      target_state_type: joint_state
      target_state_name: stability_pose
      plan_name: current_to_stability
      outcomes: [success_to_plan, fail_to_plan]
      transitions: {success_to_plan: sm_successful, fail_to_plan: sm_failed}
```

Now, using the `params` trick you could have exactly the same behaviour created from

```
name: just_planning
source: framework_state_machine
node_name: just_planning_sm
outcomes: [sm_successful, sm_failed]
states:
  - PlanMotion:
      source: state_plan
      params: {target_state_type: joint_state, target_state_name: stability_pose,␣
↪plan_name: current_to_stability, outcomes: [success_to_plan, fail_to_plan]}
      transitions: {success_to_plan: sm_successful, fail_to_plan: sm_failed}
```

Here, we are using `params` within a state, and since **its keys are matching the names of the state's options** then we don't need to specify them afterwards. We will provide another example of how to use the `params` trick.

Using a concurrent state machine

Let's say that we want the robot to go to a first known pose and then to another one. You have a lot of different ways to do it, such as creating a trajectory in the framework and execute it, for the sake of this tutorial, we are going to design it as a state machine.

```
name: mock_trajectory
source: framework_state_machine
node_name: mock_trajectory_sm
outcomes: [sm_successful, sm_failed]
states:
  - ConcurrentPlan:
      source: concurrent_state_machine
      params: {state_outcomes: [success, fail], state_io_keys: [arm_commander],␣
↪target_type: joint_state}
      name: simple_concurrent_planning
      outcomes: [sucess_plans, fail_plans]
      userdata: self.userdata
      default_outcome: fail
      outcome_map: {success_plans: {PlanInitPose: success, PlanStabPose: success}}
      transitions: {success_plans: MoveInit, fail_plans: sm_failed}
      states:
        - PlanStabPose:
            source: state_plan
```

(continues on next page)

(continued from previous page)

```
            target_state_type: params.target_type
            target_state_name: initial_pose
            plan_name: current_to_init
            io_keys: params.state_io_keys
        - PlanStabPose:
            source: state_plan
            target_state_type: params.target_type
            target_state_name: stability_pose
            plan_name: init_to_stability
            starting_state_type: params.target_type
            starting_state_name: initial_pose
            outcomes: params.state_outcomes
            io_keys: params.state_io_keys
  - MoveInit:
      source: state_move
      params: {outcomes: [success, fail], io_keys: [arm_commander]}
      plan_name: current_to_init
      transitions: {success: MoveStab, fail: sm_failed}
  - MoveStab:
      source: state_move
      params: {outcomes: [success, fail], io_keys: [arm_commander]}
      plan_name: init_to_stability
      transitions: {success: sm_successful, fail: sm_failed}
```

When using this script with the framework, you should see your robot going successively to `initial_pose` and then to `stability_pose`. Since we know beforehand the starting and ending pose of the robot, it is possible to plan for both motions at the same time, saving execution time. For this purpose we have to use a concurrent state machine. As you can see, the different states executed within the latter requires as an input key `robot_commander`, so we are *inheriting* the userdata from the root state machine. You can also see that `params` here is being used in order to declare in a nested state machine the different parameters that can be used in **all** of its children states.

How to use `params`

The two different examples that we have seen so far show some examples about how to use `params`. However, there are some constraints about how to use this trick. For instance, you **cannot** *add* `params` from a parent and the state itself. As a matter of fact, the `params` declared within a state would **overwrite** the parent's `params`, so keep this in mind. This is also true for the nested state machines. If you have defined a `params` in the header of the root state machine, everything will be overwritten for all the states contained in the named nested state machine. You can see that as the equivalent of *scope* variables.

Real-world state machines

We provide two task constructor scripts that should natively work with the framework. One is made especially for picking an object in simulation and the other is about pick and hold on a physical robot not fully integrated to ROS. Both use-case rely at some point on a grasp-pose detection method that will determine where and how the manipulator should be in order to grasp the object. As explained in the introduction of this tutorial, the state machine itself is almost method-independent so that you can just change the method for benchmarking for instance.

2.4.3 Implementing its own states

As aforementioned, we already provide a set of six states that we think would allow to design a good range of behaviours related to manipulation. However if you are not interested in benchmarking, and you want to always use

the same generative method, you might want to have a specific state that automatically runs the method without the intervention of the user.

Creating the python file

The file containing your state **must** be located inside a python package (with `__init__.py` files in python 2.x). The filename **must** contain only lowercase letters and underscores. The name of the class defined inside the file **must** be the camel case version of the filename. For instance if you want to create a state that generates the next pose of the robot for camera-based servoeing, you file might be named `servoeing_command.py` and the state should be named `ServoeingCommand`. If it's not clear, please have a look at how are named the classes defined in this folder.

Skeleton of a state

In order to make the state fully compatible with our task constructor, the state should be derived from the following template

```
#!/usr/bin/env python

# You must import these two packages
import rospy
import smach
# You can import more packages as well if you need them

class StateName(smach.State):

    # You can of course add more parameters that you are going to be able to set in␣
↪the task constructor script. Make sure to make the name in this signature and the␣
↪one used in the yaml script match.
    def __init__(self, outcomes=["success", "fail"], input_keys=[], output_keys=[],␣
↪io_keys=["<optional_userdata_field>", "<optional_userdata_field>"]):
        # This line must be here since it makes the class a state that can be used by␣
↪smach
        smach.State.__init__(self, outcomes=outcomes, io_keys=io_keys, input_
↪keys=input_keys, output_keys=output_keys)
        # You can initialize whatever you need
        # ...
        # This line must be kept as well. We advise you to order the list of outcomes␣
↪such as the last item is the "negative" outcome
        self.outcomes = outcomes

    # The function execute MUST be here since it gathers all the steps that will be␣
↪run when executing the state.
    # The signature should be kept as it is. You can access any userdata defined in␣
↪the io keys by using userdata.<key>
    # The execute must return at least one of the different outcomes that you have␣
↪defined in the __init__
    def execute(self, userdata):
        # You can implement what you want here
        # You can also call other functions or methods of the class that you may want␣
↪to implement
        # Here is an example
        if self.foo():
            # "Negative" outcomes
            return self.outcomes[-1]
        # Do other stuff
```

(continues on next page)

(continued from previous page)

```
        # ...
        return self.outcomes[0]

    def foo(self):
      # Do stuff
      return True

    # You can create more functions
```

As you can see, creating a new state is quite easy and modular. As a matter of fact, you can add any parameters you might want to change from the task constructor and you can even use methods from external packages. You can also use functions implemented in C++ through, for instance, services or actions that you can call in the state.

Integrating a new state to the task constructor

If you have properly followed the two previous parts, the only remaining step is to import the state in the task constructor script. Fill the `source` field with the name of the file (without the extension). For instance if you want to add the ServoeingCommand state, I would have `servoeing_command`. Then you can add all the options that you have defined in your signature with the **exact same spelling**. You can also natively use the `params` trick. **Don't forget to specify the transitions!** And here you are, you can now create state machines relying on your own states.

2.4.4 Creating its own state machine templates

Our task constructor relies on *template* files that define the backbone of a state machine. We provide template files for creating a basic state machine, and a concurrent state machine. We also provide a template for a state machine compatible with the framework. The major difference is that we define and initialize the userdata (set of variables that can be modified within states and that can be communicated) allowing to take the most of the different functionalities that the framework offers. If you need to modify the initialization, you can either create modify our file, but we **strongly** advise you to just create another one. It can also be useful to create a template if you often use a specific state machine that you don't change much. Here is the guide to create your own template file.

Understanding the Jinja2 part

Our task constructor is making the most of Jinja2, a powerful templating tool. We are going to describe the most important parts of the template file.

Importing packages

The top part of your file should include the following lines

```
#!/usr/bin/env python

# Automatically import the proper states with respect to the state machine defined in␣
↪the task constructor script
{% for state_to_import in state_machine.states_source %}
from {{ state_to_import[0] }}.{{ state_to_import[1] }} import {{ state_to_import[2] }}
{% endfor %}
import smach
import rospy
# You can also import more packages that you may need
```

As you can see, in addition to the classical python import statement, we can find some statements between curly brackets. Such lines are commands telling Jinja2 that these parts should be modified with respect to some objects' content. Here, these few lines automatically import the proper states automatically (given that they are following the rules stated before).

Signature of the class

The class should have the following signature

```
class {{ state_machine.type }}(StateMachineType)
```

That way the name given to the class will be the same as the one you specified in the task constructor script. Please change `StateMachineType` by the kind of state machine you want to use. It can be for instance `smach.StateMachine` or `smach.Concurrence`.

Initialization of the class

The `__init__` function should follow this template

```
def __init__(self, outcomes={% if "outcomes" in state_machine.parameters%}{{ state_
↪machine.parameters.outcomes }}{% else %}["success", "fail"]{% endif %}):
    smach.StateMachine.__init__(self, outcomes=outcomes)
    with self:
    {% for state_name, state in state_machine.components.items() %}
        smach.StateMachine.add("{{ state_name }}", {{ state.type }}({% for param_name,
↪ param_value in state.parameters.items() %}{% if param_name != "name" %}{{ param_
↪name }}={% if param_value is string and "self" not in param_value %}"{{ param_value␣
↪}}"{% else %}{{ param_value }}{% endif %}{% if not loop.last %}, {% endif %}{%␣
↪endif %}{% endfor %}), transitions={{ state.transitions}})
    {% endfor %}
    # You can call other functions here, such as the one responsible for userdata␣
↪initialization
```

This part automatically creates the whole

2.5 The framework's core

The framework's core package gathers all the scripts, nodes, utils and messages available and linking all the parts of the framework. If you want to change any functionality, this is where you should start investigating.

2.5.1 Framework's managers

Regardless of the problem we want to solve, the ROS messages that are being used are most of the time similar and are derived from standard messages. When creating complex behaviours, it is common to have a lot of different topics created to communicate such messages generated in several nodes. In order to make these communications easier, we implemented six managers that allow dealing with the different messages for you. You can learn more about them, and how to create you own here.

2.5.2 Framework's states

In order to easily create some complex tasks, we implemented six states that allow to build the most common tasks in robot manipulation. Of course, we could not think about all the different use-cases and needs. That is why you can find here the details of our states as well as a tutorial to create your own.

2.5.3 Framework's simulation mode

Working in simulation is more and more popular to speed up data collection of the robot's interaction with its environment. In order to do so, we propose a simple command line based interface in order to dynamically interact with the simulated world and place or remove objects at the user's will. You can read more about it here.

2.5.4 Framework's messages

We provide some messages that try to unify the different representations of grasping. We believe that some of their content can also be used for different tasks. In order to interact easily with widely used ROS messages, we also provide some methods in C++ and python. You can find out more about our messages or the different tools here

2.6 Contributing

Since there is always space for improvement, we are happy to receive your feedback. If when you are using the framework you spot some bugs, feel free to raise an issue on the github repository. If you do like this framework but feel like something is missing, you are free to fork it or create pull requests. It can be new functionalities, new states or change of the manager, any improvement is welcome. If you find some parts of the documentation a bit confusing or blurry, feel free to raise an issue or create a pull request on this repository.