Grasping Robot Integration and Prototyping (GRIP) Documentation

Shadow Robot Company

May 13, 2022

CONTENT

This is the starting point for the GRIP Documentation. **The documentation is still in progress!**

CHAPTER

ONE

OVERVIEW

The Grasping Robot Integration & Prototyping (GRIP) framework is a robot-agnostic software that allows for visual programming and fast prototyping of robotic grasping and manipulation tasks. As any robotic use-case, grasping and manipulation require a multitude of components that need to be coordinated. However, integrating a new component in a new or existing pipeline is very challenging due to the variety of tools, format, language that are used in the literature. To tackle this problem, we developped a set of new interfaces which aim at easing the integration of external components while being able to use them when programming the robot. GRIP contains a GUI that we developped in order to guide users through all the steps from robot integration to task execution. This GUI comprises two mains parts: robot integration and task design/execution.

CHAPTER

TWO

DEMO VIDEOS

Videos of use cases implemented with GRIP are available here, here and here.

CHAPTER

THREE

SOURCE CODE

You can access GRIP's source code here.

CHAPTER

FOUR

TABLE OF CONTENTS

4.1 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 detailed below to get the latest Docker container up and running.

4.1.1 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.

4.1.2 Downloading the container

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 running Ubuntu (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 Docker, 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.

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

```
$ bash <(curl -Ls bit.ly/run-aurora) docker_deploy product=hand_e nvidia_docker=true tag=kinetic-v0.0.6
```

You can change *<container_name>* by the name you want to give to the container that you are going to use. For instance, if you want your container to be named *GRIP_test*, then you would need to run the following command:

```
$ bash <(curl -Ls bit.ly/run-aurora) docker_deploy product=hand_e nvidia_docker=true tag=kinetic-v0.0.6
```

Note: If you don't have a Nvidia graphic card or are running on the Xorg drivers, please use **nvidia_docker=false**

Troubleshooting

When running the one-liner, if you encounter an error saying:

/dev/fd/63: line 246: 1728 Segmentation fault (core dumped) pip3 install –user -r ansible/data/ansible/requirements.txt

please run `rm -rf ~/.local` (make sure to make a copy of important files if you have any) and re-run the one-liner.

If you try to install GRIP on a machine with Ubuntu 16.04, please add the `--branch v1.0.16` after `docker_deploy`, as follows:

```
$ bash <(curl -Ls bit.ly/run-aurora) docker_deploy --branch v1.0.16 product=hand_e nvidia_docker=true ta
```

4.1.3 Running the container

We are going to assume in this section that you named your container *GRIP_test* in the one-liner!

At this stage, you should have both a docker image and container created for you. First, let's make sure that's the case. You should see the following

```
$ docker ps -a
CONTAINER ID   IMAGE                                    ...   NAMES
3447e1i08b16   shadowrobot/grip_framework:kinetic-devel    ...   GRIP_test
```

To start the container, you just need to run

```
$ docker start GRIP_test
```

A new Terminator window will pop up, and will allow you to navigate inside the container. **None** of the operations you are going to run in this terminal will affect your native Ubuntu session.
For instance if you install a text editor in the container, you won't be able to run in in your graphic Ubuntu session! So feel free to install and configure your favorite text editor and everything that you need to work efficiently.

4.1.4 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 20.04 and ROS Noetic. This should be released in the next few days, so stay tuned!

4.2 Getting started

This page introduces the different notions and steps required to start working with GRIP.

4.2.1 Getting familiar with the environment

As specified in the *instructions to install GRIP*, the framework is deployed using Docker.

If it's the first time you are hearing about docker and you want to know more about it, we advise you to read this simplified explanation. In a nutshell, you can work in the Terminator window that pops up when you start the container without worrying about corrupting your native Ubuntu session (and that's pretty nice!).

The container that you have installed is running Ubuntu 16.04, i.e. all the software that you can find natively on a fresh install of Ubuntu are there as well. And good news, the full ROS Kinetic stack is already installed for you! So you don't need to install MoveIt! or any other software that is installed with ROS.

Note that instead of using the default **Gazebo 7.x**, GRIP natively runs **Gazebo 9.x**.

How to navigate inside the container

When the Terminator window opens, you will be located at `/home/user/`. From this point, you can use all the classic bash commands to navigate within the file system (e.g. `cd`, `ls`, `mv`, etc.).

To move directly to the catkin workspace, you can run

```
$ cd projects/shadow_robot/base
```

or

```
$ roscd && cd ..
```

A catkin workspace is a folder where you modify, build and install ROS packages. It must contain three folders:

- **src**: must contain the different ROS packages to build and run
- **devel**: development space
- **build**: build space

No need to worry, everything has already been set up for you and is ready to be used without further knowledge. However, if you want to know more about catkin and workspaces, you can find more here and here.

What is already there

If you are having a look at what is inside the **src** folder of the catkin workspace, don't be scared if you only see the folder `grip`, you will be able to install a lot of other ROS packages along with it. In fact, we made this container as *plain* as possible to avoid any confusion and potential conflict.

This gives you the freedom to install whatever you want inside your container. For instance, if you want to test a work that relies on specific simualtors such as MuJoCo, you can install them without any additional overhead.

We haven't included any additional and fancy text editor or IDEs, so everyone can install its favorite one (e.g. Visual Studio Code, JetBrains IDEs, atom, etc.).

4.2.2 What is GRIP

GRIP is a ROS-based robot-agnostic software that allows for visual programming and fast prototyping of robotic grasping and manipulation tasks. The main purpose of this tool is to facilitate the integration of software and hardware components into complex robotic systems.

Although the Robot Operating System (ROS) is not the only middleware available to operate robots, it remains one of the most used, especially in academia. For this reason, instead of re-implementing numerous features already provided by this operating system, we decided to make the most of it.

If you've never used ROS and are reluctant to thoroughly learn this stack, GRIP is made for you. In fact, we tried to reduce as much as possible the ROS knowledge required to use our framework, to make sure GRIP remains accessible and easy to use. In addition, we provide a documentation with several examples for all the steps from robot integration to task design and execution.

In order to simplify all the steps necessary to run a robot task, we have implemented an intuitive and reactive Graphical User Interface (GUI) for both the integration stage and task design. If you want to have a look at the code, all related files are located in the grip_api package. The code corresponding to all the other components of this framework, can be found in the grip_core package. Let's now have a quick overview of the different steps required to run a task with GRIP.

Note: Disclaimer: GRIP's focus is not to be highly performant, but rather to provide users a tool to easily reproduce their own setups and integrate components to screen them, before potentially integrating them to their existing optimised pipeline!

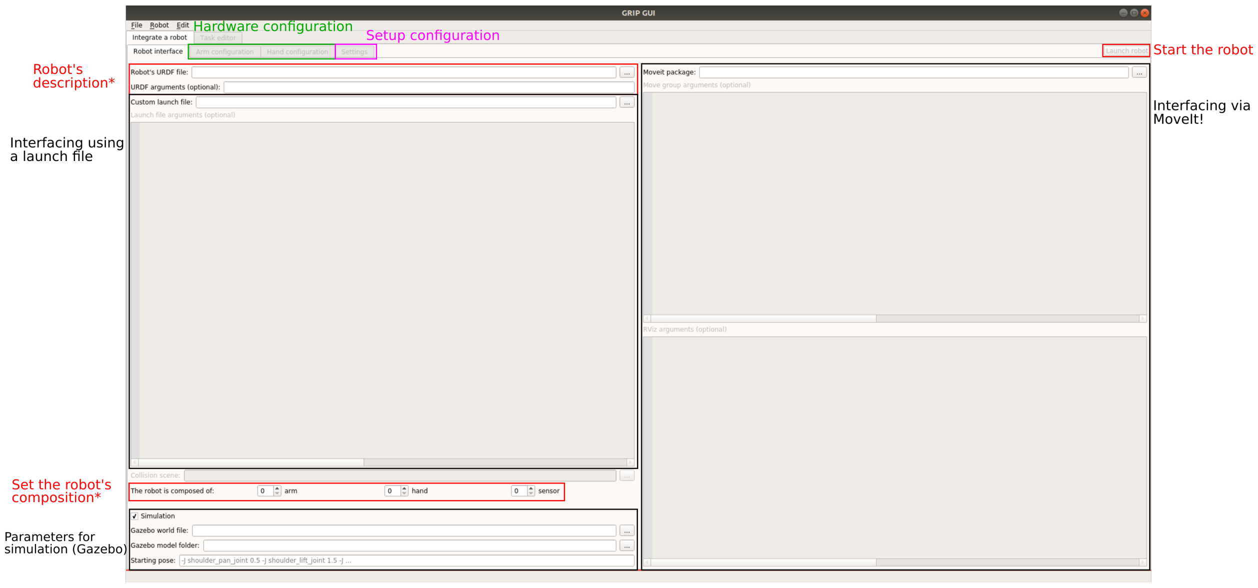
Integration stage

Interfacing a robot is the first step required to program it. For this reason, GRIP features several non-exclusive integration modalities allowing you to re-use already existing components, and thus saving you time. If you start from scratch, no worries, the other pages of the documentation should greatly help you!

Before going into more details in the *tutorials*, we provide here a quick overview of the GUI specifically dedicated to help you interfacing hardware and/or software. In order to start GRIP, you first need to run

```
$ roslaunch grip_api start_framework.launch
```

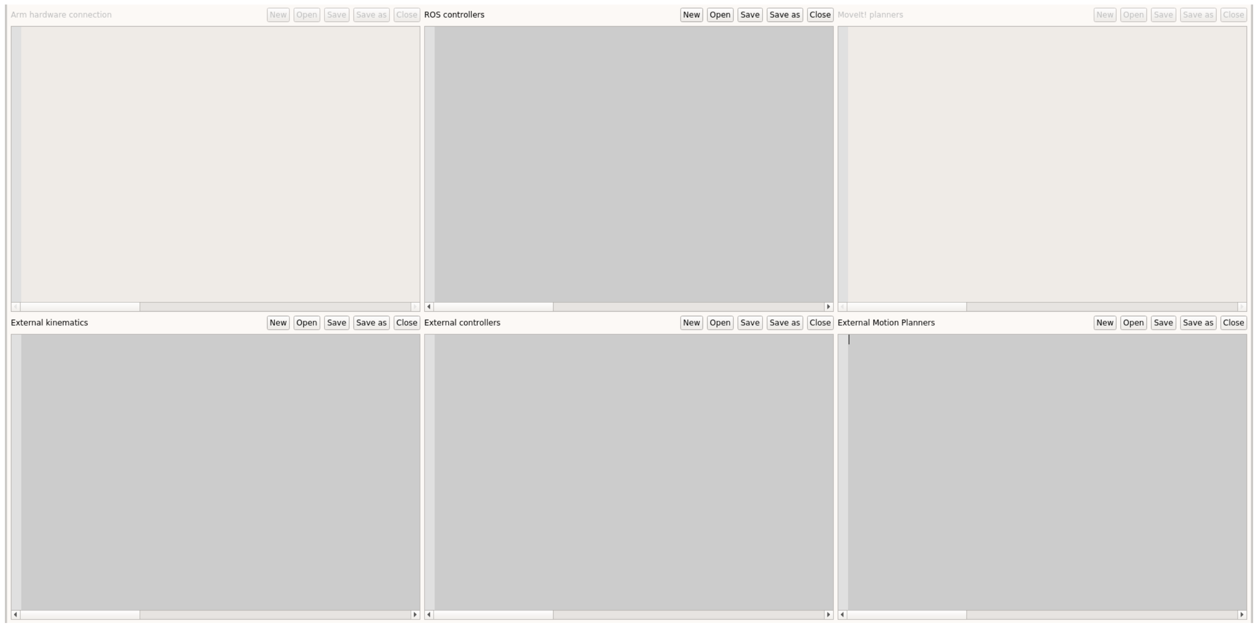
You should have the following GUI appearing (without the annotations obviously).



Please note that the launch button is here **disabled** and will become **enabled** only when a robot has been successfully integrated! This should help you save quite some time trying to start your robot while it's going to fail because something is missing…

The black areas correspond to optional robot's interfacing modalities, that are going to be detailed in the tutorials.

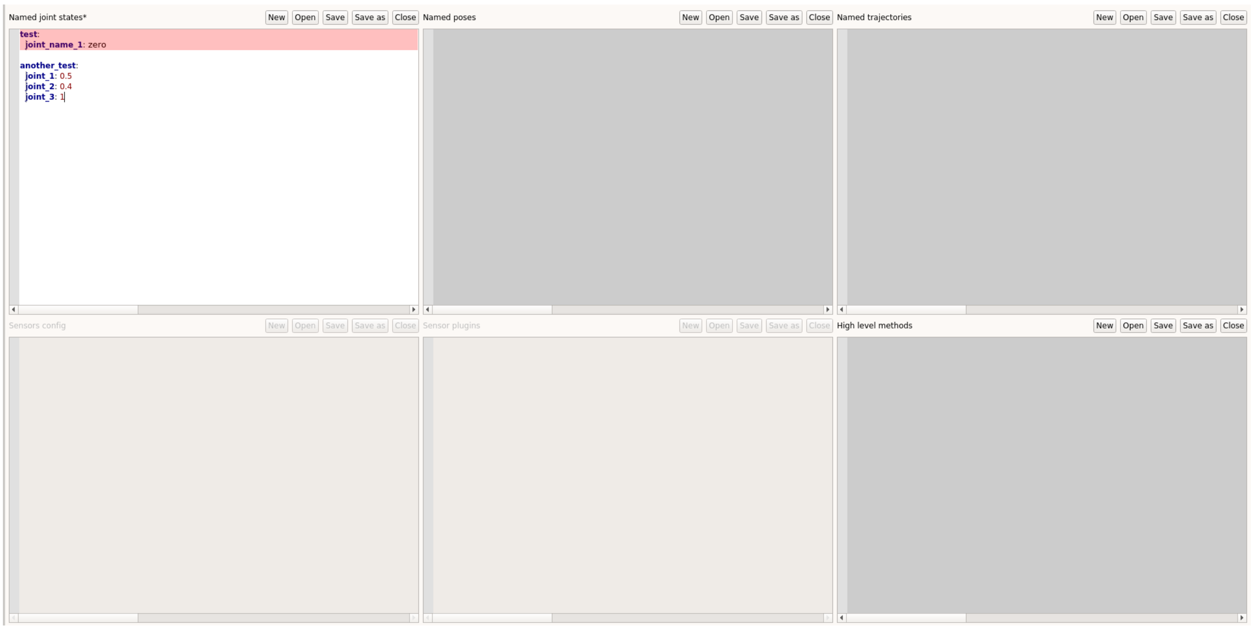
The two areas on the left (i.e. **Robot's description**, **Robot's composition**) are required regardless of the modalities you want to use to interface your robot. Depending on the composition of your robot, the appropriate hardware configuration tabs (area in green) will become available. Here is what such tabs look like



In order to configure a robot arm (or hand/gripper), you need to specify several components, e.g. which controller, kinematics library and planner to use to operate it. In the above image, you can see that two areas are grayed out (i.e. disabled), and that's fine, it means that the current configuration does not allow the user to set these fields. In other words, we have implemented a reactive mechanism that enables/disables the different fields according to the current inputs to help you configure the robot. For instance, the `MoveIt! planners` editor will be available if and only if a MoveIt! configuration package has been previously provided.

For each editor, you can either open existing **YAML** configuration files or create your own via the push buttons you can see. Once done, if you want to integrate a controller you have implemented yourself, you can just press the + symbol that appears in the margin. A sequence of dialogues pop up and if you follow them, your component will be successfully integrated to GRIP. More details about this process can be found in the tutorials.

Each editor has a live syntax check that will signal you whether some information you provided don't follow the expected format. For instance, as illustrated in the following figure, the editor `Named joint states` of the `Settings` tab shows an error. In fact, the value corresponding to the joint *joint_name_1* should be an integer or float, not a string. Such mechanism, along with autocompletion should help you to efficiently interface and configure your robot.



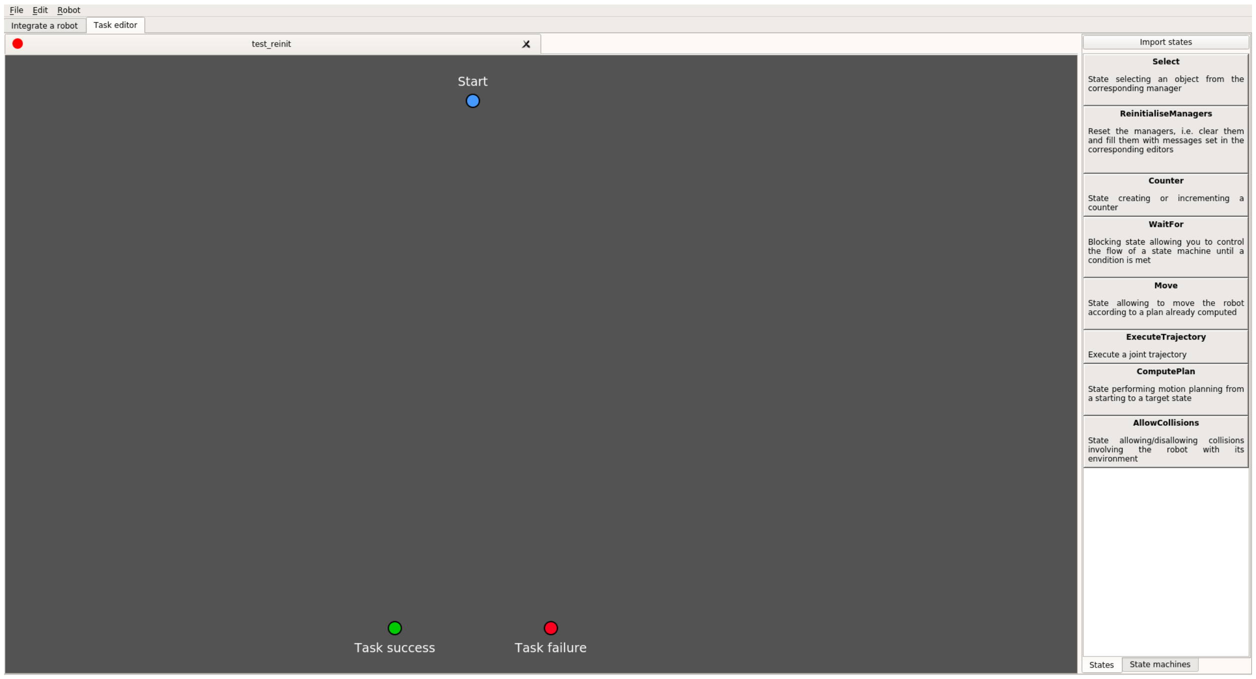
```
test:
  joint_name_1: zero

another_test:
  joint_1: 0.5
  joint_2: 0.4
  joint_3: 1
```

Once you have configured everything you need, you can start designing your task!

Task design and execution

Now that you have interfaced a robot and all the methods you might want to run for a specific task, GRIPS allows you to efficiently and graphically design a wide range of tasks with the components you need!

If the robot configuration is valid, you should see the `Task editor` enabled at the top left of your screen. If you click on it, you should see an interface similar to this one (some building blocks on the right might change)



Select

State selecting an object from the corresponding manager

ReinitialiseManagers

Reset the managers, i.e. clear them and fill them with messages set in the corresponding editors

Counter

State creating or incrementing a counter

WaitFor

Blocking state allowing you to control the flow of a state machine until a condition is met

Move

State allowing to move the robot according to a plan already computed

ExecuteTrajectory

Execute a joint trajectory

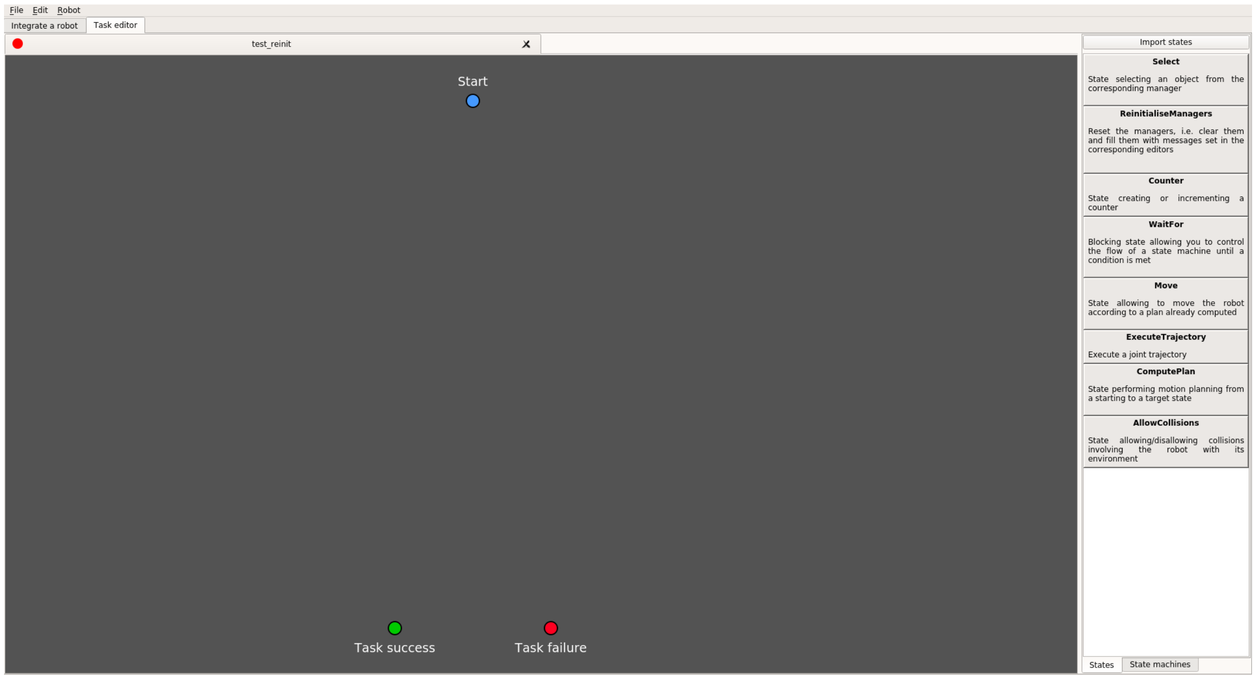
ComputePlan

State performing motion planning from a starting to a target state

AllowCollisions

State allowing/disallowing collisions involving the robot with its environment

On the right hand side of the screen you should be able to see 2 tabs (`States` and `State Machines`), which both contain the different elements that you can use to articulate and design the task you want your robot to execute. Each building block should have a name and a small description that helps you understand what it does. Please note that **the building blocks that are avaialble depend on your robot configuration**. In order to use them, you just need to drag-and-drop them inside the grey area. Once dropped, all the states will be represented as a box-like object with common elements.



Select

State selecting an object from the corresponding manager

ReinitialiseManagers

Reset the managers, i.e. clear them and fill them with messages set in the corresponding editors

Counter

State creating or incrementing a counter

WaitFor

Blocking state allowing you to control the flow of a state machine until a condition is met

Move

State allowing to move the robot according to a plan already computed

ExecuteTrajectory

Execute a joint trajectory

ComputePlan

State performing motion planning from a starting to a target state

AllowCollisions

State allowing/disallowing collisions involving the robot with its environment

As you can see, all the boxes that appear when you drop a state or a state machine have some sockets (colored circles). They allow you to link the different boxes (i.e. states) so you can easily handle the flow of of your task. So depending on the outcome of a state, you can choose what to do by linking the socket that represents this outcome to another state (more specifically its input socket). If don't know the principle behing state machines, you can have a look at that.

Note: By default, the green socket represents the "success" outcome (i.e. the code ran without any error). The red socket represents the "error" outcome, meaning that something went wrong when executing the code.

For all the states (except ReinitialiseManager), you will see some editable slots that will allow you to configure the state to some extent. Note that some slots are directly linked to the robot configuration. For instance, using the drop-down menu of `input_type` in the state **Select** and set it to `joint state`, you will be able to pick an element (for the slot `input`) among all the joint states that you have registered during the integration stage!

You can double click on the name of a state (or right click on it and select `Rename`) to rename it, as it can help to have a better grasp of what the state is doing at first glance, especially in large and complex behaviors. The editor area is interactive, meaning that you can move all the different objects as much as you want. Similarly, you can navigate inside the editor; you can zoom in and zoom out with the wheel of your mouse and move around by pressing on the wheel.

In order to execute a task, the sockets of **all** the states need to be connected to another socket. In order to link the `Start` socket (big blue one, initially at the top of the screen) to another blue socket (of a state), please double click on the big socket and drag it to the target socket. For all the other sockets you just need to click once and drag to the target socket. **Once the flow of the state machine is correct**, you will see the icon on the top left corner of the main editor (first one that was open) turn green. This means that the state machine has a correct flow. **If the robot is running**, you will be able to execute the task by right-clicking in the maain editor and click on `Execute`.

If everything is clear so far, you can move to the *tutorials*.

4.3 Tutorials

This page contains a list of tutorials demonstrating how to use GRIP. The purpose of these tutorials are to demonstrate (1) the different ways to integrate hardware and software, (2) how to design and execute tasks with GRIP.

Each tutorial contains a link to the associated resource that might be needed depending on the user's level of knowledge. The list of resources can be found *here*.

4.3.1 Using the embedded simulation mode

4.3.2 Integration stage

Interfacing a robot via MoveIt!

Interfacing a robot via a launch file

Integrating external software

Integrating a sensor

Defining pre-recorded poses

Defining pre-recorded joint states

Defining pre-recorded trajectories

4.3.3 Task design and execution

Running an automated pick and place task

Running an autonomous pick and place task

In progress, the rest will be uploaded shortly!

4.4 States

This page contains all you need to know about states in GRIP, i.e. how to create new states and how to use the provided ones.

4.4.1 Constant states

The source code of the following states can be found here. You should always be able to see them, regardless of the configuration of your robot.

Counter

ReinitialiseManagers

Select

WaitFor

4.4.2 Commander states

The source code of the following states can be found here. These states will become available if and only if your robot uses MoveIt to operate a part of your robot.

Note:

For these states to appear at least **one** MoveIt motion planner must be defined in the robot integration tab.

If **more than one** planner is defined, a configuration slot named `group_name` will appear for all these states. It allows you to choose which part of the robot you wish to configure.

AllowCollisions

ComputePlan

ExecuteTrajectory

Move

4.4.3 Generated states

When you are interfacing an external component or a sensor, GRIP will automatically generate the code of the corresponding states.

ExternalComponent

Sensor

4.4.4 How to create a new state

You can create new states and integrate them to GRIP. Although most of the time it is advised to create and integrate an external component, which will generate a state for you, you might want to create your own set of states. We are going to review the required process below.

Warning: All the interactive functionalities such as dropdown menu and automatic refresh of possible values for slots are **not** supported for external states.

For GRIP to directly account for your states, you can decide to add the source code of your state in `/home/user/projects/shadow_robot/base/src/grip/grip_core/src/grip_core/states`.

Warning: Make sure to add your states here and **not** in the `commander` folder! Otherwise GRIP may not run properly.

The content of any state you want to create should follow this template:

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

import smach
# Import other packages if required

# The name of the state MUST be the CamelCase version of the filename! For instance for␣
↪this state the filename should
# be name_of_state
```

(continues on next page)

(continued from previous page)

```
class NameOfState(smach.State):

    """
      Small description of what the state does
    """
    # Change the argument_to_set by the name of the variable you want to be able to␣
↪configure in the task editor
    # You can also set default values, such as argument_to_set=10
    # The remaining arguments from outcomes onward MUST NOT BE REMOVED
    # You can add more outcomes if you want in he signature
    def __init__(self, argument_to_set1, argument_to_set2, outcomes=["success", "finished
↪"], input_keys=[], output_keys=[], io_keys=[]):
        """
          @param outcomes: Possible outcomes of the state. Default "success" and "failure"
          @param input_keys: List enumerating all the inputs that a state needs to run
          @param output_keys: List enumerating all the outputs that a state provides
          @param io_keys: List enumerating all objects to be used as input and output data
        """
        # Initialise the state, THIS LINE MUST NOT BE REMOVED
        smach.State.__init__(self, outcomes=outcomes, io_keys=io_keys, input_keys=input_
↪keys, output_keys=output_keys)
        # Do whatever you want here
        # ...
        # Outcomes of the state. THIS LINE MUST NOT BE REMOVED
        self.outcomes = outcomes

    # Do NOT change the name or the signature of this function
    def execute(self, userdata):
        """
          @param userdata: Input and output data that can be communicated to other states

          @return: - outcomes[-1] depending on your implementation
                   - outcomes[0] otherwise
        """
        # The code here is executed everytime that the state is run
        # So write what you need
        # ...
        # But don't forget to at least return one outcome!
        # Can be self.outcomes[-1] or self.outcomes[2] if you have more than two␣
↪outcomes. If you do have more than 2,
        # make sure to have them defined in outcomes in the class signature!!!
        return self.outcomes[0]
```

Important:

Regardless of where you are storing the file, make sure to name the file following the underscore naming rule! For instance if your state is named `ComputeJointState`, the name of the file must be `compute_joint_state.py`.

You can find a concrete example of how to create a new state, here.

4.5 Resources

This page contains a list of resources that might help you better understand some notions or procedure required to run GRIP. Depending on your needs, you might not want to read all of them.

If you just want to read what is useful for your use-case, we advise you to follow the relevant *tutorials*, in which the corresponding resource will be linked to.

4.5.1 Integration stage

Creating a MoveIt! configuration package

Creating description files

What is a ROS controller file

What is a ROS hardware interface

Creating custom msg files

Creating a ROS service server

Creating a ROS action server

Making ROS action/service server compatible with GRIP

4.5.2 States

In progress, the rest will be uploaded shortly!