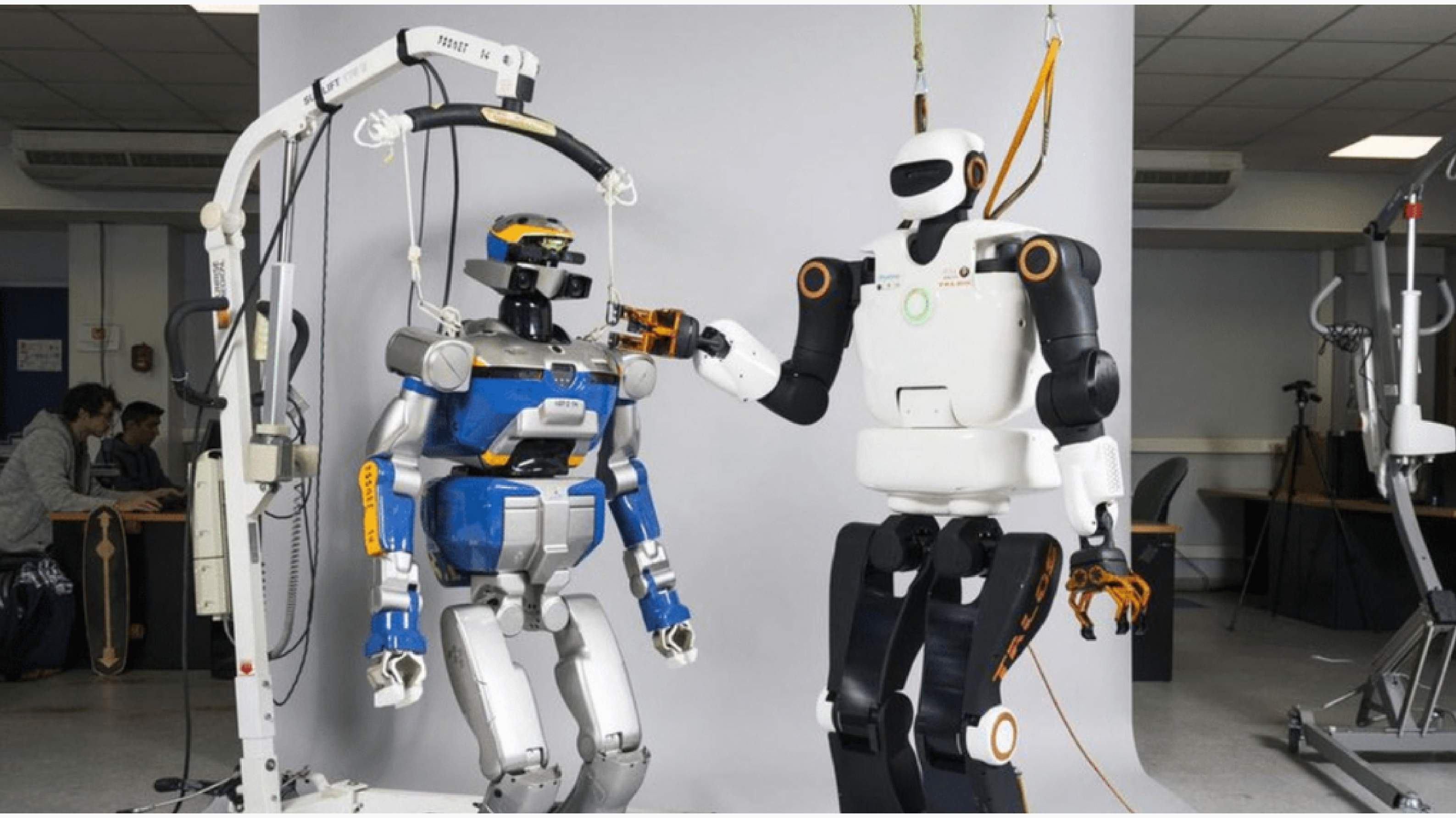
JUSTINE FRICOU
3RD YEAR
PGE 2024

INTERNSHIP REPORT

CONNECTION TO A ROBOT AND VISUALIZATION OF ITS INTERNAL LOGIC

APRIL - AUGUST 2022



LAAS-CNRS
7 AV. DU COLONEL ROCHE
31400 TOULOUSE, FRANCE

EPITECH TOULOUSE
40 BD DE LA MARQUETTE
31000 TOULOUSE, FRANCE

TUTORS: FLORENT LAMIRAUX
& MAXIMILIEN NAVEAU

TABLE OF CONTENTS

ACKNOWLEDGEMENT

I would like to thank the tutors who accompanied me throughout this internship, Mr. Florent Lamiraux and Mr. Maximilien Naveau, for the time they gave to my training and their availability, but also for the trust they showed me and their encouraging words regarding my work.

SECTION 1

TO A NEW EMPLOYEE

WELCOME TO THE GEPETTO TEAM

This guide will provide you with the information needed to understand the organization of the team and the purpose of its works.
You will find a presentation of the LAAS-CNRS, a view of how the Gepetto team operates, and the overall architecture of the projects you will be taking over to help you get started.
Definitions for all of the technical vocabulary specific to the projects (words and expressions followed by an asterisk) can be found in a glossary at the end of the document.

Please feel free to contact any of your supervisors if you have any questions regarding the projects or the day-to-day operations in Gepetto.

TABLE OF CONTENTS

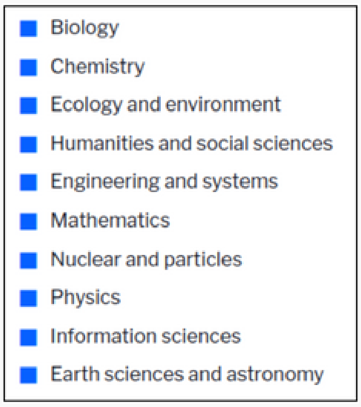
1. PRESENTATION OF THE GEPETTO TEAM

1.1. A brief background on the team

1.1.1. The CNRS

The Gepetto team is part of the CNRS (French National Center for Scientific Research), a French public institution. It is a research organization, internationally recognized for its work, and under the supervision of the French Ministry of Higher Education and Research.

Its mission is to advance science by conducting research that is in the interest of the economical, social and cultural advancement of France. One of its key values is sharing knowledge, both to the scientific community and to the general public.
It is divided into 10 national institutes, each dedicated to a specific field of research, as listed in Fig.1.



- Biology
- Chemistry
- Ecology and environment
- Humanities and social sciences
- Engineering and systems
- Mathematics
- Nuclear and particles
- Physics
- Information sciences
- Earth sciences and astronomy

Figure 1: Research fields of the CNRS (source: cnrs.fr)

1.1.2. The Gepetto team in LAAS-CNRS

The LAAS (Laboratory for Analysis and Architecture of Systems) is part of the CNRS. Located in Toulouse, France, it conducts research meant for applications to various fields, such as space, health, environment or energy.

It consists of 6 departments:

- Trustworthy Computing Systems & Networks
- Microwaves and Optics: from Electromagnetism to Systems
- Robotics
- Micro Nano Bio Technologies
- Decision and Optimization
- Energy Management

The Gepetto team belongs to the Robotics department, and focuses its research on movements of anthropomorphic systems, covering several themes such as automatic motion generation, analysis and simulation of human movement, or artificial muscles.

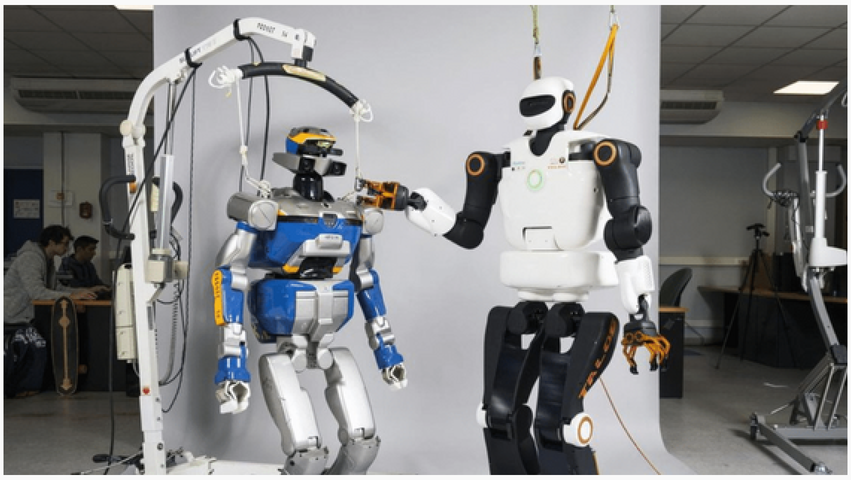


Figure 2: Humanoid robots of Gepetto (source: ladepeche.fr)

The projects presented in this document are destined to help with automatic movement generation of robots. A more detailed description of their purpose can be found in the 'Context and architecture of the projects' section.

1.1.3. Rob4Fam

These projects are conducted as part of the Rob4Fam (Robots For the Future of Aircraft Manufacturing) partnership.

It is a joint laboratory between the LAAS and Airbus whose objective is to develop robots able to analyze their environment and adapt to it. Indeed, a practical use of such robots would be aircraft manufacturing, which could be automated with mobile and autonomous robots able to work alongside human workers.

1.2. Organization of the team

The Gepetto team is composed of researchers tutoring post-docs, PhD students and interns.
The research team is supported by research engineers who maintain the software and the robots.

During the training there are weekly appointments with the supervisors to discuss the progress of the work:

- What has been done?
- What will be done next?
- What issues are being encountered?

This is an important meeting that allows the supervisors to keep track of work, and to make sure that what is being developed meets the practical needs of the team.
Indeed, as a developer (as opposed to a user), one only has a global vision of the day-to-day uses of the products. It is therefore crucial to try and understand the usage patterns, and to regularly check with the future users that the features are implemented in a way that makes their work easier and quicker.

As part of the Rob4Fam laboratory, there is an additional meeting every Monday afternoon. These will allow the researchers in charge of the partnership to keep up with the work of every group of the collaboration, and forward the information to their colleagues at Airbus in order to demonstrate the progress of the LAAS-side team and perpetuate the partnership.
During this meeting, every member will describe what they have done during the previous week, and what they plan on doing for this new week. They will also talk about any issue they are encountering, so that other members can provide help and advice after the meeting.
The meetings are conducted in English as international students and researchers are part of the team. Before each of them, each member is asked to write down everything they will mention in the Rob4Fam chat channel.

Occasionally, researchers from other organizations or universities from around the globe will be visiting and presenting their work. Members of the team are encouraged to attend these conferences. For researchers and research students, these events are an occasion to learn about new advancements in the field and discuss with the experts who conducted the works. For non-specialists, they are a great opportunity to learn more about robotics as a whole.

2. CONTEXT AND ARCHITECTURE OF THE PROJECTS

2.1. The Stack of Tasks

To control the robots, the Gepetto team has developed the ‘Stack of Tasks’ (SoT).
A robot can be described in two parts: its processor and its body. The processor can be seen as a brain. The body has motors to make it move, and sensors which generate data (for example, a camera can help detect nearby objects). It sends this data to the processor, which analyzes it and makes decisions on how the body should move (i.e controls its motors). This makes a loop between the processor and the body: the sensors send data to the processor, which in turn calculates what commands to send to the motors. The motors will execute the commands, and the sensors will then send feedback to the processor, which will determine if more movements have to be done (e.g to maintain the robot’s balance or keep the movement going), and so on.

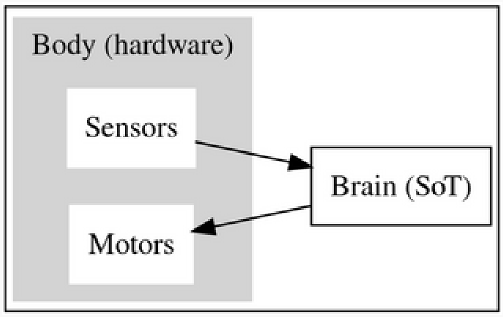


Figure 3: Diagram of the processor-body system in a robot

Here, the processor is the SoT. It consists of a list (‘a stack’) of steps to go through (‘tasks’) to determine a command to send to the motors, based on the sensors’ data.
This set of steps (called entities) can be represented as a graph (see Fig.4) which will be executed to compute the command.

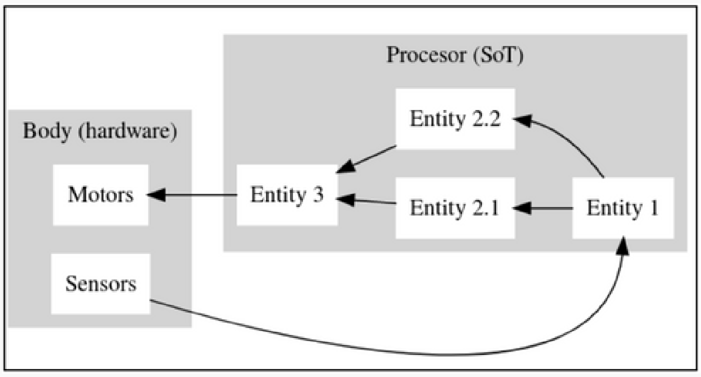


Figure 4: Diagram of the SoT working in pair with the robot’s hardware

Every entity of the calculation has inputs and outputs, which can be anything, but are typically numbers, vectors, or matrices.
In Fig.5, entity 1 is an addition. Its three inputs are the outputs of three sensors. An arrow linking an output to an input is called a signal. For each of them, their value is given above the arrow. For example, sensor 1 gives the value ‘2’ as output, which is brought to entity 1’s first input via a signal. Entity 1 adds the values of its three inputs (2, 3 and 5), and gives 10 as output.
This output is linked to entity 2’s first input. The second input value is not given by a sensor, but by a fixed value, called a ‘variable’. This value is fixed by the researchers working on the robot. Entity 2 performs a euclidean division with its two inputs (10 / 3), and gives the quotient (3) and the remainder (1) as two separate outputs. Each of them are then given to one of the motors as input.

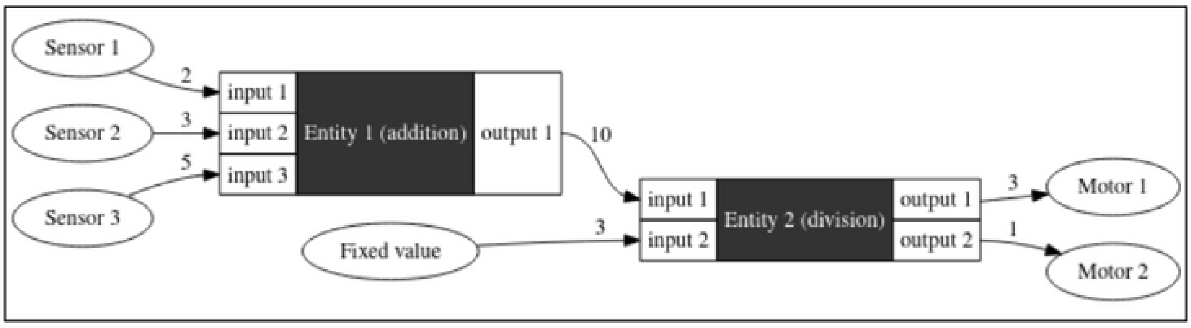
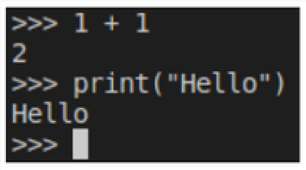


Figure 5: Example of SoT graph showing inputs and outputs

The dynamic_graph package* implements these entities, the logic linking them and the calculation of the graph.
The robot’s processor is actually C++ code embedding a python interpreter* containing the dynamic graph.

Fig.6 shows the use of a python interpreter. The user typed-in the “1 + 1” command, and the interpreter executed it before displaying the result (“2”).
The user then typed-in the “print(“Hello”)” command, which means “display ‘Hello’”. On the following line, the interpreter has indeed displayed the line.



```
>>> 1 + 1
2
>>> print("Hello")
Hello
>>> 
```

Figure 6: Example of interactions with a python interpreter.

Researchers in Gepetto need to be able to easily visualize the graph and interact with it, to rapidly understand what is happening inside the robot: which values were given as output for each entity, which entities were executed or not, etc. This allows them to understand what went right and what needs to be fixed.

This is the purpose of the two projects described in this document: the sot-ipython-connection package, which aims at facilitating interactions with the dynamic graph, and the sot-gui package, which displays it in an ergonomic way.
These two packages are open source*, as are most of Gepetto's works, in accordance with the CNRS commitment to the sharing of knowledge.

2.2. The sot-ipython-connection package

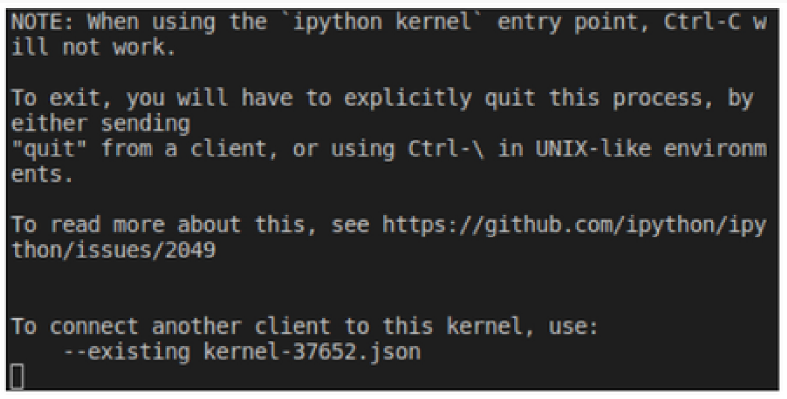
2.2.1. Purpose of the project

A robot has its own embedded computer onboard, which means that the only way to interact with it is remotely. Hence the researchers need access to the python interpreter running on the robot.

The sot-ipython-connection package is what makes this connection possible. It is based on ipython[1], a python interpreter with added features making interacting with the dynamic graph easier. A few examples of theses features are:

- access to the history of past commands sent to the interpreter ;
- completion (i.e the interpreter suggests the end of a word or a command as the user is typing it) ;
- additional syntax (i.e some actions can be performed with fewest commands than with a conventional interpreter).

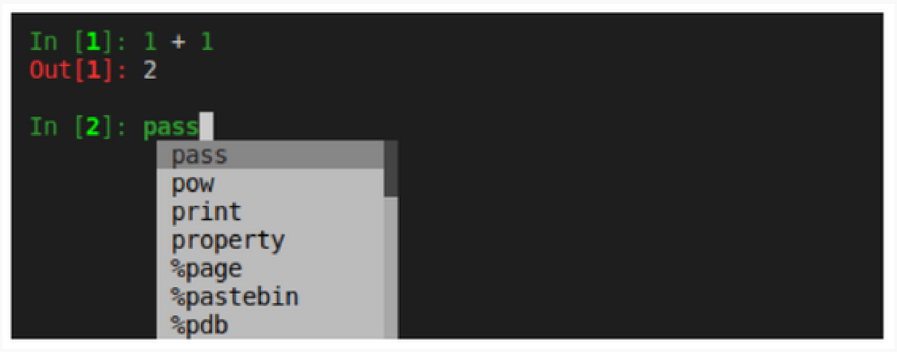
sot-ipython-connection implements the interpreter as an ipython kernel*, with which we can communicate thanks to a client*.



```
NOTE: When using the `ipython kernel` entry point, Ctrl-C w
ill not work.

To exit, you will have to explicitly quit this process, by
either sending
"quit" from a client, or using Ctrl-\ in UNIX-like environm
ents.

To read more about this, see https://github.com/ipython/ipy
thon/issues/2049


To connect another client to this kernel, use:
    --existing kernel-37652.json
```

Figure 7: Screenshot of an ipython kernel running. It can only be interacted with via a client.

Fig.8 shows the use of an ipython client. The "1+1" command has been typed-in by the user. The client sent it to the kernel, which executed it and sent the result ("2") to the client, which then displayed it.
This screenshot also demonstrates the 'completion' feature of ipython: all available commands starting with a 'p' are suggested, and the chosen one (here: 'pass') is automatically typed-in as the command.



```
In [1]: 1 + 1
Out[1]: 2

In [2]: pass
        pass
        pow
        print
        property
        %page
        %pastebin
        %pdb
```

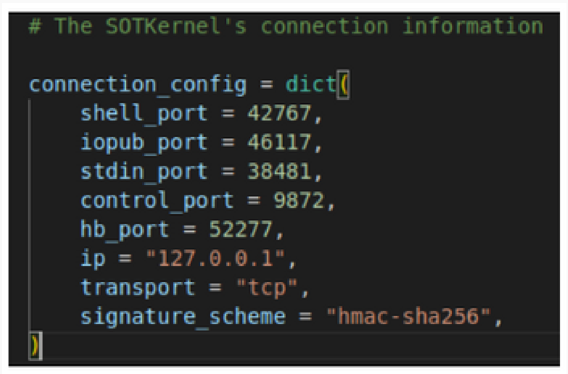
Figure 8: Screenshot of an ipython client

2.2.2. Architecture of the project

As explained above, sot-ipython-connection relies on ipython. It implements custom variants of its tools, with added features specific to the use of the SoT.
These variants are the sot_interpreter and the sot_script_executer. Like the ipython kernel and client, they are both developed in the python language using respectively the SOTKernel and SOTClient classes*.

SOTKernel

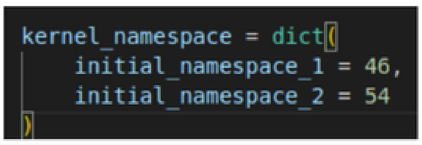
The sot_interpreter is a simple program that launches a SOTKernel, which is a variant of the ipython's kernel. This variant allows the user to decide which ports* will be used for the connection with the client. The user can define these ports in the connection_config.py file.
To know more about communication between a kernel and clients with ipython, visit [2].



```
# The SOTKernel's connection information

connection_config = dict(
    shell_port = 42767,
    iopub_port = 46117,
    stdin_port = 38481,
    control_port = 9872,
    hb_port = 52277,
    ip = "127.0.0.1",
    transport = "tcp",
    signature_scheme = "hmac-sha256",
)
```

Figure 9: Content of the connection_config.py file

Another additional feature is a preset namespace* for the kernel. When working on a robot, new kernels are launched very frequently. Often, the kernels need to contain the same namespace. Instead of creating these variables after each launch, the user can define a namespace in kernel_namespace_config.py, which will be automatically added to the kernel at launch.



```
kernel_namespace = dict(
    initial_namespace_1 = 46,
    initial_namespace_2 = 54
)
```

Figure 10: Content of the kernel_namespace_config.py file

The last feature implemented with SOTKernel is the possibility of running the kernel in a non-blocking call. When a program is executed in a regular way, each of the lines of its source code are executed one after the other. A line cannot be executed before the previous one is complete. Executing a line of code this way is called a “blocking call”. Thus, when the kernel is launched in a program, the rest of the program will not be executed until the kernel has been stopped. The kernel being supposed to run for a long time, this can be an issue in some cases. SOTKernel offers the possibility to launch the kernel in a non-blocking way, to let it run in the background while continuing to execute the rest of the program.

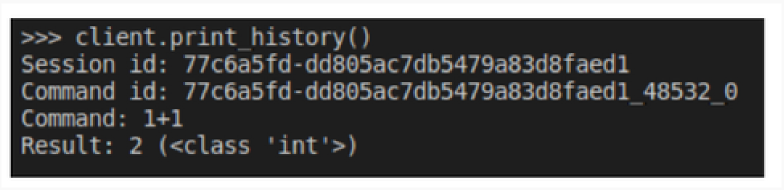
SOTClient

sot_script_executer is a program that uses a SOTClient (a variant of the ipython’s client) to connect to a SOTKernel and execute python scripts (i.e programs) on it. These scripts can be local (stored on the computer executing the client), or on the computer running the kernel. Similar to the namespace configuration at launch for the SOTKernel, the user will often need to execute the same numerous lines of code on the kernel after launching it. To make this task simpler and quicker, sot_script_executer allows them to store these lines into one single script, and execute this script on the kernel by entering a single command.

An additional feature of SOTClient is the possibility to reconnect to a new kernel. Researchers will very regularly need to stop the kernel and launch a new one, with new parameters. The SOTClient allows them to change kernels without needing to launch a new client every time they do.

To finish, SOTClient offers the possibility of displaying a history of the past commands, with more information than a regular ipython kernel. It stores each command it sends, alongside the response of the kernel, thanks to the SOTCommandInfo class. This class therefore stores:

- the content of the command ;
- its result or the error sent by the kernel if the command was incorrect ;
- the identifier of the client which sent the command (to know which client sent the command, as several of them can be connected to a same kernel) ;
- the identifier of the message, to differentiate similar commands.



```
>>> client.print_history()
Session id: 77c6a5fd-dd805ac7db5479a83d8faed1
Command id: 77c6a5fd-dd805ac7db5479a83d8faed1_48532_0
Command: 1+1
Result: 2 (<class 'int'>)
```

Figure 11: Example of a past command in the command history.

Tests

The correct execution of the sot-ipython-connection package features is ensured thanks to the pytest testing-tool.
A BaseTestClass is used as a base for each test case: before each of them, a SOTKernel is launched, and a SOTClient is connected to it. The client is then made to send specific commands, and the results are checked.

2.2.3. Future of the project

Future improvements for this project would be to enhance the 'history' feature of SOTClient. For now, a client history only contains commands sent by this client. This means that the history of a client is lost if the latter is stopped or in case of a crash.
The objective is to store, in each client, every command processed by the kernel, i.e every command sent by every client currently connected to this specific kernel.

In order to do this, the 'IOPub' port defined in connection_config.py can be used. On this port, the kernel broadcasts every command it processes, and its response, which means any client can have access to information on any other client's commands through this port.

Once this is done, functions will have to be created to allow the user to display the history: they should be able to choose whether they want it to be a specific client history (by providing its identifier), or that of all clients.
Adding a timestamp to each command could also be useful to help the user to differentiate similar commands.

Creating a new sot_history.py script running a SOTClient, dedicated to storing and displaying the history of all clients could be useful for users by providing them with a log of all of the kernel's actions. This way, they could have this one client constantly displaying the history, and another regular client dedicated to interacting with the kernel.
Making this a graphical interface in a regular window (as opposed to a terminal-based program) could improve the ergonomy thanks to a clearer display and by facilitating the filtering or sorting of the commands by client, timestamp, content, etc.

Functional tests will have to be written to cover these new functionalities. They can be added to the test_client_history.py file, which already covers the existing 'history' features.

The current architecture of the project has been designed taking into account these future additions, thus this task should not require a lot of changes in the existing source code.
To know more about the ipython client SOTClient is based on, visit [3].

Another important future task for this package is to be officially included in the dynamic_graph_python package. This package is, in essence, the python binding* of the C++ dynamic_graph package presented in Section 2.1. This way, this package will officially be used by the team.

2.3. The sot-gui package

2.3.1. Purpose of the project

A dynamic graph used for controlling a robot contains numerous entities. To efficiently work on improving the results of their experiments, researchers need to quickly and easily understand what is happening in the dynamic graph at all times. The sot-gui package allows them to visualize the graph: they can know which parts of it have been executed, which inputs were given to which entity, what its output was, etc.

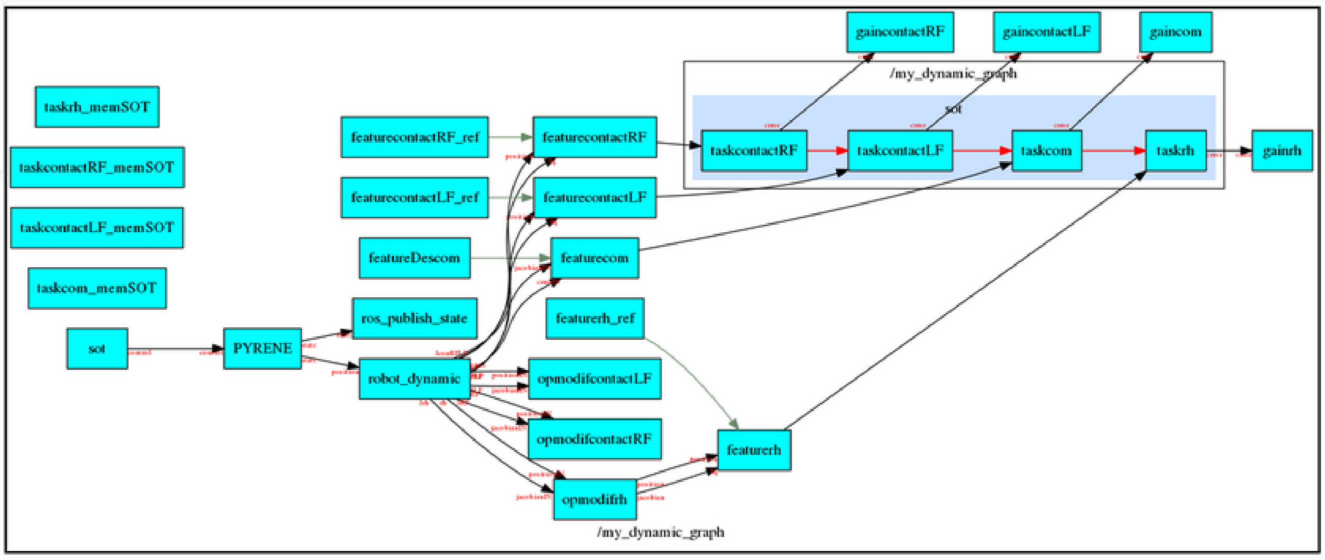


Figure 12: Example of a dynamic graph (source: gepettoweb.laas.fr)

Users need to be able to interact with the displayed graph. Examples of useful interactions are:

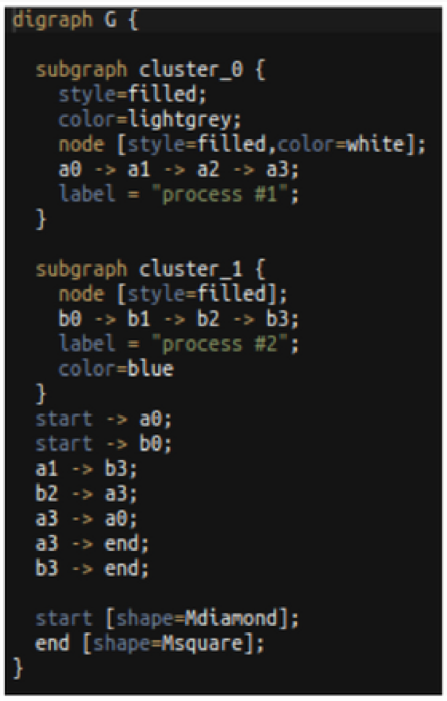
- grouping entities into one single node to make the graph more readable (clusterization) ;
- displaying more or less information on entities and signals depending on the level of zoom ;
- clicking on an element to open a new side tab displaying information on this element ;
- filtering entities based on their name, type, etc. In order to either hide, highlight or only display them.

Interacting with a displayed graph is not to be confused with interacting with the dynamic graph in the kernel: sot-gui allows the user to interact with the displayed elements to make the graph more legible. It does not modify the content of the dynamic graph controlling the robot. This allows for a safe usage of the gui by beginners.

2.3.2. Architecture of the project

Determination of the graph's layout

'dot' is a program allowing its user to visualize a graph in several forms: pdf, png, svg, etc. When provided with 'DOT code' describing a graph, it computes a layout of the graph and displays it, as can be seen in Fig.13.



```
digraph G {

  subgraph cluster_0 {
    style=filled;
    color=lightgrey;
    node [style=filled,color=white];
    a0 -> a1 -> a2 -> a3;
    label = "process #1";
  }

  subgraph cluster_1 {
    node [style=filled];
    b0 -> b1 -> b2 -> b3;
    label = "process #2";
    color=blue
  }
  start -> a0;
  start -> b0;
  a1 -> b3;
  b2 -> a3;
  a3 -> a0;
  a3 -> end;
  b3 -> end;

  start [shape=Mdiamond];
  end [shape=Msquare];
}
```

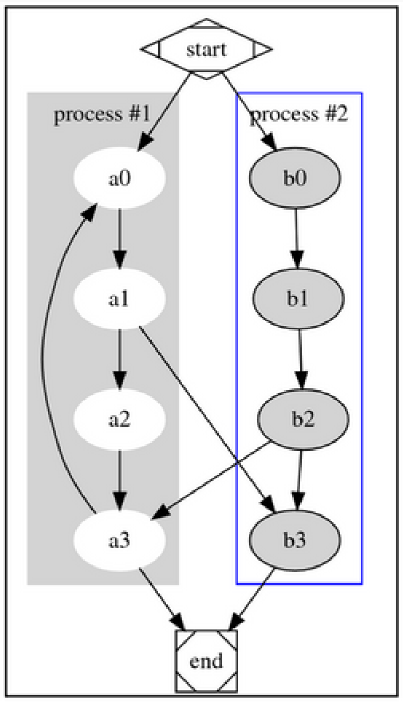


Figure 13: Example of DOT code and the associated graph generated by dot

Documentation on the syntax of the DOT language is accessible by visiting [4].
Nodes and edges (i.e arrows) on the graph can be given attributes to set their color, shape, label, etc.

Graph display

Simply relying on dot to visualize the dynamic graph would not be sufficient: outputs such as pdf and png do not allow users to interact with the displayed elements.
Therefore, sot-gui relies on dot to compute the graph's layout, and handles the display of each element itself. In addition to visual outputs such as png of pdf, dot can generate json code* describing the graph's layout:

- position of the elements ;
- coordinates of the polygons vertices ;
- color of the outlines ;
- background colors ;
- sizes and fonts of the labels ;
- etc.

The python library* used for the display is PySide [5]. It allows to display graphical elements in a window and interact with them through the mouse or keyboard. It is the python binding of the famous C++ graphical library Qt which is open-source and widespread in the Graphical User Interface community.

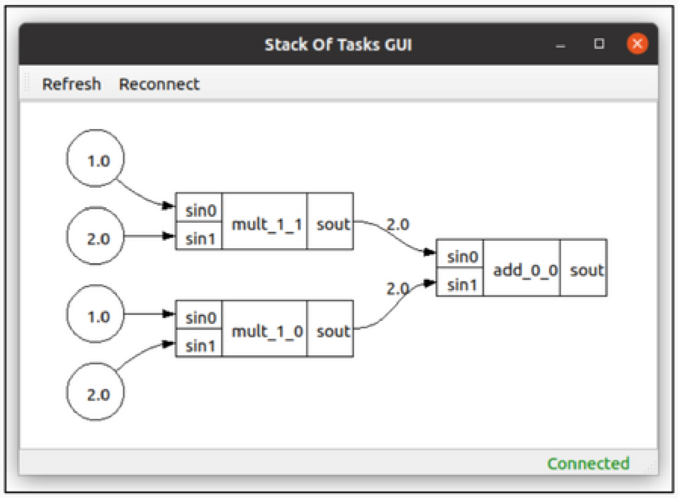


Figure 14: Example of a simple graph displayed in a window with PySide

To get data from the dynamic graph located in a SOTKernel, sot-gui uses a SOTClient contained in the DGCommunication class. This is the class handling all communications with the dynamic graph: creating the SOTClient, (re)connecting it to a kernel, detecting when the kernel has stopped, etc.
All the graph information on entities and signals are stored in a Graph object*.
As illustrated in Fig.15, It first calls a DGCommunication object to fetch the data, then uses this information to generate DOT code representing the graph, thanks to the DotDataGenerator class.
This code is given to dot, producing a json output, which is used by the JsonToQtGenerator to generate graphical elements, which are stored in the Graph object too.
All these elements are then displayed in the window thanks to the MainWindow class.

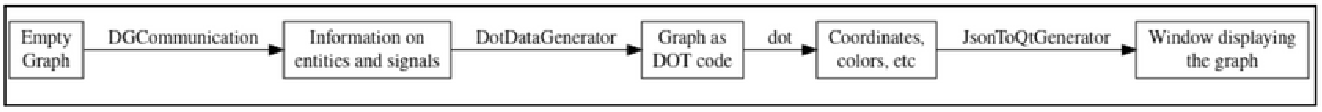


Figure 15: Overview of the steps to display the graph in a window

Modifications of the display

There are three levels of refreshing the elements displayed on the window, all described in the following paragraphs along with some of their applications.

When working on a robot, users need to launch new kernels very frequently. To prevent them from having to open a new window every time, there is a ‘reconnect to kernel’ feature. This simply uses DGCommunication to reconnect the SOTClient to the latest SOTKernel. To display the graph after a reconnection, one must refresh the graph.
The ‘refresh the graph’ feature can also be useful when the kernel has not changed, but its content has.
-> First level: the Graph object is emptied and the entirety of the process illustrated in Fig.15 must be repeated.

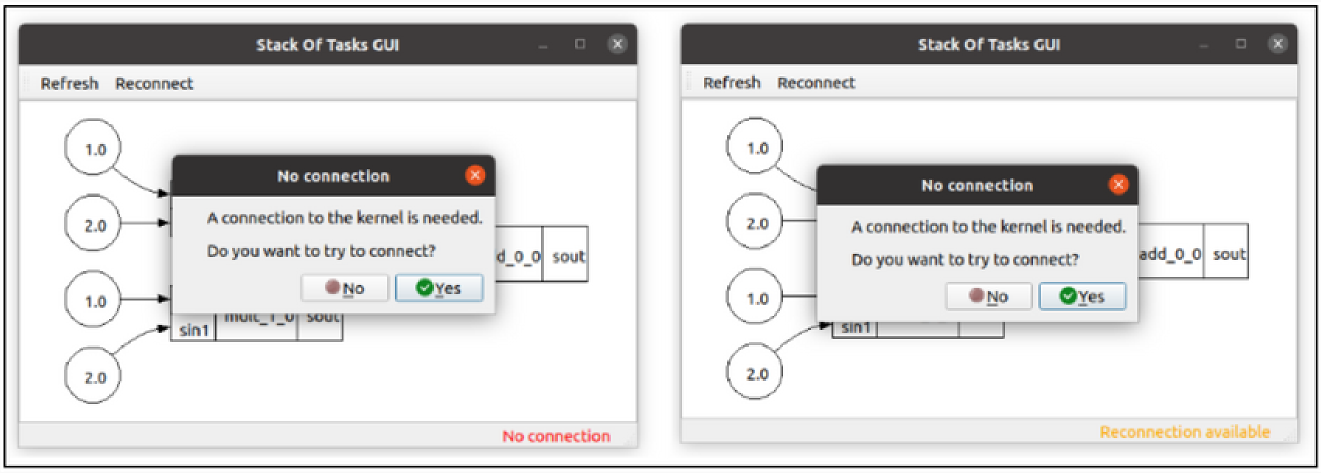


Figure 16: sot-gui alerting on the missing connection when trying to refresh the graph. On the left, the alert just popped up and no kernel is running. On the right, the user has launched a new kernel and a reconnection can now be attempted.

Another existing feature is the clusterization of entities. When a graph contains numerous entities, it can be useful to group some of them into one single node to make the graph more legible.
-> Second level: in this case, data from the dynamic graph does not have to be refreshed: DotDataGenerator will generate new DOT code, taking into account the need to clusterize the given entities. A new layout will be generated and new elements will be displayed.
The same process will be used for the following feature: adapting the amount of information displayed on the graph according to the zoom level. No data has to be fetched, as it will be the same information displayed, but with more or less details (e.g only displaying the name of the entities vs. displaying their name and their inputs/outputs names).

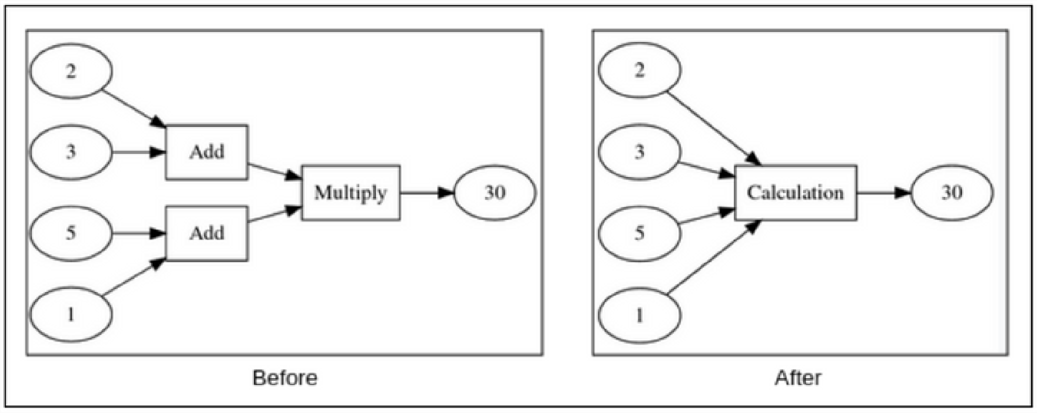


Figure 17: Example of clusterization of nodes

To finish, hovering the mouse above an element (node or edge), or clicking on it, creates a new tab with information on the element (e.g its identifier, the values of their inputs and outputs, etc).
-> Third level: in this case, there is no need to fetch new data from the dynamic graph, nor to update the layout with dot: entities and edges and their graphical elements will not change. New graphical elements will be added to the window, independently of the graph.

Tests

In the same way as sot-ipython-connection, the sot-gui package has functional tests developed with pytest.
In this case, instead of testing each feature independently, several types of graphs are tested (e.g normal graph, entities with no inputs, etc).

A BaseTestClass is used. For each test, a SOTKernel is launched and a dynamic graph is created on it thanks to sot_script_executer. Then, a Graph object is created, which will handle every step from fetching data thanks to DGCommunication, to creating graphical elements thanks to JsonToQtGenerator. The list of generated graphical elements is then checked.

2.3.3. Future of the project

Future improvements for this project would be to enhance the ergonomy of the graphical interface, and to test the new functionalities.

Features regarding interactions with the displayed graph are as follows:

- filtering of the entities (displaying only entities with a specific name, type, etc): this would require a level-2 refresh of the elements on the window, as the layout would have to be recomputed ;
- indicating which parts of the dynamic graph have been recomputed or not, using colors (green: up to date ; orange: not up to date ; red: never computed). This would require a level-3 refresh, as graphical elements would not be replaced: only their colors would be modified.

Another useful feature could be interacting with the dynamic graph content, i.e modifying the kernel content by sending commands via the client. This could prevent the need of switching between two clients (one for display and one for control) for some tasks. This should be discussed with supervisors before any implementation.

Once the necessary features are completed, sot-gui should be included in another package: sot-gepetto-viewer. This is another graphical interface for other needs when working with the Stack of Tasks. sot-gui is currently developed separately, as sot-gepetto-viewer uses PythonQt instead of PySide for display. Research should be done to determine if these two libraries can be made compatible.

GLOSSARY

Class
Structure representing a particular kind of object, defining a set of methods and properties common to objects of this type.

Client
A program running locally, and connected to the kernel, through which a user can interact with the kernel.

dynamic-graph
The package that implements the SoT's entities, the logic linking them and the calculation of the graph.

Entity (dynamic-graph)
Step of the dynamic graph's calculation. Essentially, what one calls a node in a graph.

Function
Reusable parts of code.

Interpreter
Program which executes instructions in a programming language.

Json
Widely used textual data format.

Kernel
Program running remotely with which one can communicate through a client.

Library
Collection of functions dedicated to a specific use.

Method
Function bound to an object, allowing to use or modify it.

Namespace
Set of variables contained in a program, or part of a program.

Non-blocking call
Line of code running in the background while the rest of the program continues being executed.

Object
Instance of a class.

Open source
Software whose source code is freely available and can be modified and redistributed.

Package
Pieces of code reusable in several programs.

Port
Virtual point where network connections start and end, allowing one program to connect with another.

Python binding of C++ code
Python code through which one can use existing C++ code in a python program.

Signal (dynamic-graph)
Link between a node’s output and a node’s input.

Stack of Tasks (SoT)
List of steps to go through to determine what commands to send to the motors, based on the sensors’ data. This allows to control a robot.

Variable
Value, stored with a name, and retrievable thanks to this name.

BIBLIOGRAPHY

[1] Link to the ipython website:
https://ipython.org/

[2] Documentation on communication between a kernel and clients with ipython:
https://jupyter-client.readthedocs.io/en/latest/messaging.html#messaging-in-jupyter

[3] ipython client documentation:
https://www.adamsmith.haus/python/docs/jupyter_client.BlockingKernelClient

[4] Dot syntax documentation:
https://graphviz.org/doc/info/lang.html

[5] Pyside documentation:
https://wiki.qt.io/Qt_for_Python

SECTION 2

TO MY TUTOR

Dear Mr. Lamiraux,

I am writing to express my interest in joining the Gepetto team for a new project.

The internship I did in your team during my 3rd year at Epitech has shown me how well-suited a career as a research engineer would be for me. It confirmed my desire to contribute to research through my work as a developer. Working in Gepetto has shown me it is possible for my career to have such a meaning, by advancing science and knowledge while being surrounded by kind and talented people.

During my stay at LAAS, Mr. Naveau mentioned the need to create a new interface to help with connecting new robots to the SoT. This project would be directly linked to one of the packages I worked on during my internship: sot-ipython-connection.
Hence, I already have an overall understanding of how a robot interfaces with the SoT, which would make working on this project easier for me, especially for contributing to the design of the interface's architecture.

Moreover, I know the main part of this project would be implementing this new architecture. I have worked with C++ before and am used to object oriented programming (notably thanks to my last internship in Gepetto working with Python), as well as low-level programming thanks to my several years experience with the C language.
I am also already familiar with the team's habits and good practices in terms of packaging, testing, and documentation, which I endeavor to respect in my projects.

Indeed, producing clean and legible code is high on my list of priorities, as I know one can save a considerable amount of time with a quick and proper understanding of a project.
During my internship, I have picked on a new habit: adding typing and minimal documentation as I code. I know this new project would require a lot of documentation on the new architecture, and I am eager to discover how to produce complete documentation in a real setting.

I enjoy working in this team with its habits and customs, and I get along well with the other members, which can be just as important as the technical aspects of the mission.

I believe I demonstrated how hardworking and diligent I am during my internship in Gepetto. I hope you will choose to work with me again for this project.
Would you be available next week to discuss this further?

Thank you for your time.

Sincerely,
Justine Fricou