PROJ-H-402 - Computing Project: Extending ARGoS and TAM with Python

Alberto Parravicini

1 Introduction

ARGoS is a physics-based simulator for swarm robotics. Working with ARGoS requires the user to know either **C++** or **Lua**. The goal of this project was to provide an easy-to-use interface written in **Python**, a simple language often known also by people unfamiliar with programming.
TAM (Task Abstraction Module), a device used in swarm robotic, was also partially reimplemented in Python, and it is fully supported by the ARGoS simulator.

The following report is divided in two main sections, which describe the Python wrapper to ARGoS, and the TAM implementation, respectively.
For each section, it is provided an explanation of the design process, and of the structure of the code. Instruction to install the software are also provided.

2 A Python wrapper for ARGoS

2.1 *Introduction*

ARGoS is a complex physics-based simulator used in swarm robotics to model the behaviour and the actions of small robotic devices, such as `e-puck` or foot-bot. Each of these robots has a multitude of different sensors and actuators, that can be used to understand and interact with the surrounding environment or robots.

Working with real robots can be expensive and time-consuming, and therefore a simulator is an essential tool in the development of most applications and researches.
Even more so, a simulator can be an invaluable resource for students wishing to learn more about swarm robotics.

As a consequence, it is clear the importance of having a simulator which is not only powerful, but also very easy to use.
Swarm robots usually follow very simple instructions and patterns, and there are obvious advantages in being able to implement such tasks in a fast and error-free way.

A simulation in ARGoS is composed by three parts:

- A **configuration** file, written in an **xml**-like format, which specifies the entities (the robots, and other objects) that are part of the simulation. This file also loads the required controllers and loops functions (see below), and other options used by the simulation.
- One or more **controllers**, which describe the behaviour of the robots. Controllers have a set of predefined abstract functions that have to be implemented by the user, in order to specify the actions of the robots at the start, at the end, and at each step of the simulation.
- A **loop function**, if necessary, that will customize the events at the start or end of each simulation step.

ARGoS is implemented in **C++**, and uses **C++** as the main choice for writing simulations. Even though this language is known to most people working in robotics, it is still rather cumbersome to use in practice, due to its inherent complexity, and the need to recompile the code at each small update.
Moreover, it can be argued that the performances offered by **C++** can hardly be beneficial when the tasks to run are relatively simple to execute, as it is often the case for swarm robots.

ARGoS also offers a wrapper written in **Lua**, which allows to write simulations in a faster and simpler way.
Still, **Lua** is still not a very well known language: most developers, especially students, probably won't be able to use this wrapper without spending additional time to learn the language.
This wrapper doesn't have all the features of the original **C++** implementation, but it still allows to control the most important actuators and sensors of the robots.

It looks natural at this point to look for a different solution, which can provide the accessibility of the **Lua** wrapper, without sacrificing any functionality.
This is the idea behind the **Python** wrapper for ARGoS: a wrapper which is developed to be as similar as possible to the **Lua** version, and that offers the same functionalities, if not more.

2.2 *Installation*

This section will describe how to install ARGoS and the **Python** wrapper, and how to run simulations with it. Specific details about the structure of the simulations are left to the later sections.

The wrapper was mainly tested on **Windows 10**, using the **Linux subsystem** (basically, a shell for **Ubuntu 16.04**). In order to install it, it is recommended to follow the guide at http://www.terriblesysadmin.com/?p=76.
The wrapper was also briefly tested on a native **Archlinux** distribution.

2.3 *Installing ARGoS*

Here it is described how to install the basic ARGoS simulator.
This guide is adapted from the one found at https://github.com/ilpincy/argos3.

- Download the ARGoS source code at https://github.com/ilpincy/argos3.
- (Alternatively, clone the repository).
  ```
  git clone https://github.com/ilpincy/argos3.git argos3
  ```
- Install the dependencies.
  ```
  sudo apt-get install libfreeimage-dev libfreeimageplus-dev
  qt5-default freeglut3-dev libxi-dev libxmu-dev liblua5.2-dev
  lua5.2 doxygen graphviz graphviz-dev asciidoc
  ```
- Move into the repository folder, called `argos3`.

- Create a folder `build_simulator`.
  ```
  mkdir build_simulator
  ```
- Move into the folder `build_simulator`.
  ```
  cd build_simulator
  ```
- Run *cmake*
  ```
  cmake ../src
  ```
- Run *make*
  ```
  make
  ```
- Compile the documentation (required!).
  ```
  make doc
  ```
- Install ARGoS.
  ```
  sudo make install
  ```
- Make sure that ARGoS is installed system-wide.
  This step is taken from https://lonesysadmin.net/2013/02/22/error-while-loading-shared-libraries-cannot-open-shared-object-file/
  Make sure that the dynamic linker checks `/usr/local/lib`:
  ```
  cat /etc/ld.so.conf
  include ld.so.conf.d/*.conf
  /usr/local/lib
  ```
- Update the cache.
  ```
  sudo ldconfig
  ```
- It is possible to check if ARGoS runs correctly by running the examples found at https://github.com/ilpincy/argos3-examples

2.4 *Installing e-puck*

The **Python** wrapper fully support **e-puck**, one of the robots that can be used in ARGoS. This section details how to install the **e-puck** plugin.
More details can be found at
https://github.com/lgarattoni/argos3-epuck/blob/master/doc/Installation.pdf

- Clone the repository the **e-puck** plugin repository.
  ```
  git clone https://github.com/lgarattoni/argos3-epuck.git argos3-epuck
  ```
- Move into the repository folder, `argos3-epuck`.
- Create a `build` directory, move into it.
  ```
  mkdir build
  cd build
  ```
- Install **e-puck**
  ```
  cmake ../src
  make   sudo make install
  ```
- Check that **e-puck** appears among the available entities in ARGoS. `argos3 -q entities`
- Optionally, try some of the examples found in the previous folder, as there are also examples that support **e-puck**.

2.5 *Installing the Python wrapper*

Finally, it is possible to install the **Python** wrapper. Note that it requires to install **Boost** (more details on this are given later).
The wrapper was tested with **Boost 1.61.0**, but this guide should apply also to more recent versions of the library.

This section is based on http://www.boost.org/doc/libs/1_61_0/more/getting_started/unix-variants.html and on https://eb2.co/blog/2012/03/building-boost-python-for-python-3-2/, using **Python 3.5** instead of **3.2**.

- Download **Boost** at http://www.boost.org/users/history/version_1_61_0.html
- Move in the folder where **Boost** was downloaded, and move it to `/usr/local/`.
  ```
  mv boost_1_61_0.tar.bz2 /usr/local
  ```
- Extract **Boost** in the same location.
  ```
  tar -bzip2 -xf boost_1_61_0.tar.bz2
  ```

- Install the *developer* version of Python.
  ```
  sudo apt-get install python3-dev
  ```
- Move to the **Boost** folder.
  ```
  cd /usr/local/boost_1_61_0/
  ```
- Build **Boost**.
  ```
  ./bootstrap.sh -with-python=python3.5
  ./b2
  sudo ./b2 install
  ```
- Clone the **Python** wrapper repository.
  ```
  git clone https://github.com/KenN7/argos-python argos-python
  ```
- Move into the repository folder, create a `build` folder and move into it.
  ```
  mkdir build
  cd build
  ```
- Build the wrapper.
  ```
  cmake ../src
  make
  ```
- From the build folder, it is possible to launch some examples to see if the wrapper is working correctly. For instance:
  ```
  argos3 -c ../examples/aggregation_10.argos
  ```

2.6 *Technical details of the implementation*

This section will describe the overall structure of the wrapper, and the main design choices that were taken in the implementation.

The main building block of the wrapper is the **Boost.Python** library, an interface between **C++** and **Python**.
Boost.Python allows to write **C++** functions that can be called from any **Python** script, as if they were natively implemented in **Python**.
This allows a Python script to interact with the original ARGoS simulator, which is written in C++.

Boost.Python allows to export C++ classes, each with its own attributes and functions, usable in a transparent way by Python.
The wrapper is compiled as a shared library which can be loaded at the start of a simulation. When this happens, a Python interpreter is launched by the wrapper.
Then, the interpreter will load the classes exported by the wrapper, which can be used in the simulation.

The wrapper also parses the `.argos` configuration file, so that it knows which sensors and controllers will be used in the simulation, and is able to instantiate the required classes.

The implementations of sensors and actuators are divided into 2 main categories: the ones that are generic, usable by any type of robot (such as the **Differential Steering Actuator, or the LED lights**), and the ones that are specific to a single class of robots (for instance, the **Footbot** has a gripper that can be used).

The wrapper is structured so that generic sensor and actuators are contained in a single file, while robot-specific devices are contained in other files, one for each type of robot.
More specifically, we have a file for the generic sensors and actuators, 1 for the devices of **Footbot** and 1 for **e-puck**.

Moreover, there exists some generic classes that are used to wrap specific data-types used by ARGoS, such as the *colors* or the *angles*.

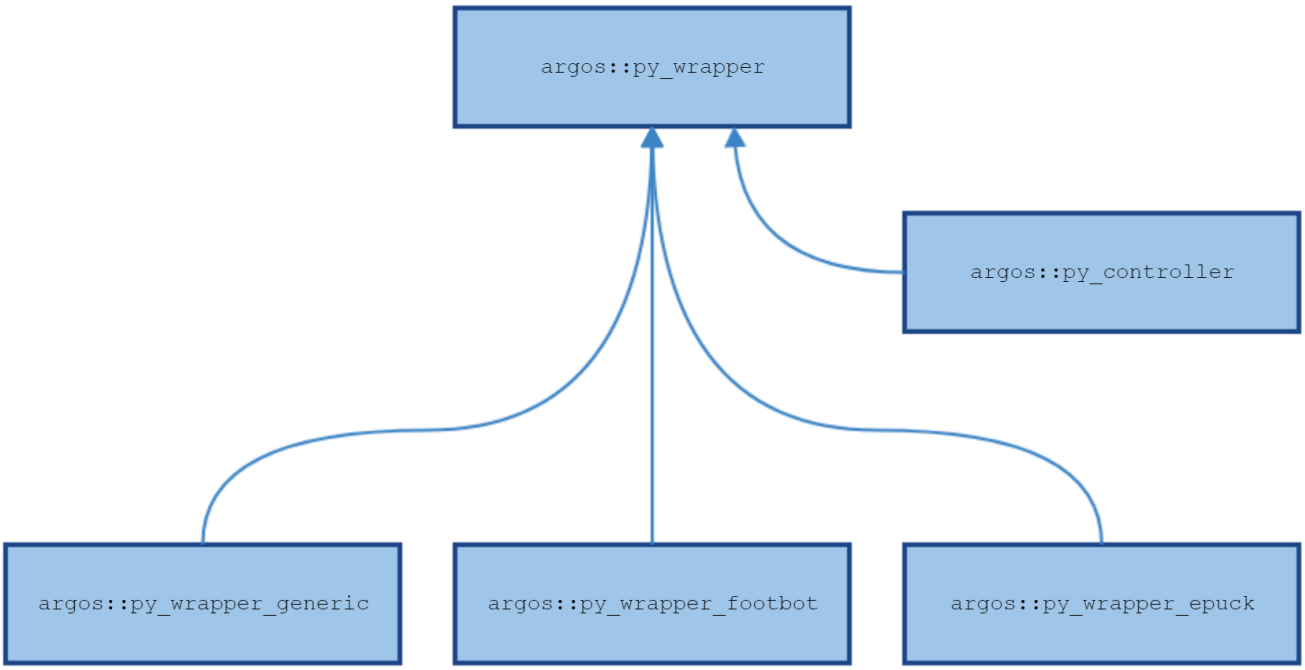

Figure 1: *Scheme of the structure of the wrapper.*

2.7 *Description of the wrapper components*

This section provides a brief description of the content of each file of the wrapper.

- `py_controller`: main entry point of the wrapper. It will initialize the Python interpreter, and parse the `.argos` configuration file in order to initialize and load the correct sensors and actuators. Functions to be executed by the Python script at the start, end, or at each step of the simulation are also wrapped here. Their implementation is in Python, but the actual execution is handled by the wrapper, in this file.

- `py_wrapper`: this file holds the implementation of the functions that instantiate the sensors and actuators. More importantly, it also contains the wrappers to some utility data-structures, and it exports all the classes that can be used by Python.
  Depending on the situation, it is possible to export functions of each class, or attributes, that can be accesses by Python either as read-only, or read-write. Note that the naming convention of the exported function and properties tries to replicate as close as possible the one used by the **Lua** wrapper, so that the moving from one wrapper to the other is as seamless as possible.

- `py_actusensor_wrapper_generic`: this file contains the generic actuators and sensor, the ones that are usable by any robot inside ARGoS.
  It contains implementations for:

  - Wheels actuator
  - Omnidirectional Camera sensor
  - Range and Bearing device
  - LED lights

  The classes relative to these actuators and sensor contains functions that call the original **C++** implementations of each device.

- `py_actusensor_wrapper_footbot`: file that contains the actuators and sensors specific to the **footbot**.
  The actuators and sensors available are:

  - Gripper actuator
  - Proximity sensor
  - Ground sensor
  - Light sensor
  - Distance scanner

  - Turret device

- `py_actusensor_wrapper_epuck`: file that contains the actuators and sensors specific to the **e-puck**.
  The actuators and sensors available are:
  - E-puck Wheels actuator
  - Proximity sensor
  - Ground sensor
  - Range and Bearing device
  - LED lights actuator

2.8 *Usage of the wrapper*

In order to write a simulation with the wrapper, it is required to provide to ARGoS a configuration file written in **xml**-like format, and one or more controllers used by the robots.
The configuration file is written just like a regular configuration file used by ARGoS, with the only difference that it is required to specify the path to the wrapper library.

Below, an example of the Python wrapper section to be added to the configuration file.

```
<python_controller id="python" library="path/to/libpy_controller_interface.so">
    <!-- Normal actuator/sensor configuration follows -->
    <actuators>
        <differential_steering implementation="default"/>
        <range_and_bearing implementation="default"/>
    </actuators>
    <sensors>
        <footbot_proximity implementation="default" show_rays="true"/>
        <footbot_base_ground implementation="rot_z_only"/>
        <range_and_bearing implementation="medium" medium="rab"/>
    </sensors>

    <!-- The controller scripts are passed here: -->
    <params script="path/to/examples/aggregation_1_python.py"/>
</python_controller>
```

Then, it is required to write a controller script for the robots (note that when using the wrapper it is still possible to use C++ controller, together with the Python ones). In Python, a controller must implement the following functions:

- `init`: function that is called at the start of the simulation. Usually it will set the local variables of the robot, such as its current state.
- `controlstep`: this is the main part of the simulation, the function that is called at each time-step. This function defines the main behaviour and actions of the robot.
- `reset`: function called when the simulation is restarted. Usually, it is used to reset the local variables.
- `destroy`: function that is called at the end of the simulation. It can be used to print information about the simulations, for example.

3 PyTAM: a TAM implementation in Python

The second part of the report is an explanation of the Python implementation of the **TAM** device, and how it is connected to ARGoS through a C++ interface.

TAM [1] is a device used to abstract complex tasks in swarm robotics. A **TAM** interacts with one or more **e-puck** robots, and their collective behaviour can be programmed to model a more complex task.
The motivation behind **TAM** is that often in swarm robotics it is necessary to model complex interactions between robots and the environment, such as moving objects, or reaching locations in a certain order, and so on.
However, it is not practical to actually deal with physical entities, when not strictly necessary for the accuracy of the simulation.
Instead, it is possible to consider a device such as **TAM**, which is able to abstract this type of interactions and simplify the creation of a simulation.

TAM is able to check when a robot is present inside it, and it is able to communicate simple codes to a nearby robot by making use of an infra-red communication device.

TAM is implemented in **Java**, and connected to ARGoS through a C++ interface.
The real **TAM** device communicates with the robots by using an **XBee** device, while in ARGoS the **TAM** communication is simulated to be faithful to the original behaviour.

Initially, the plan of the project was to create a Python interface between the Java and the C++ layer, so that it would have been possible to program the simulated **TAM** using Python (similarly to what was done with the wrapper).
However, it was noticed how adding an extra layer of complexity would make things more complex and hard to manage. Instead, rebuilding from scratch the implementation of **TAM** would be a better approach in the long term, in spite of the initial additional work required.

Here it is provided a Python implementation of the general structure of **TAM**, plus a Python implementation of a simulated version of the **TAM** that can be used from ARGoS.
Moreover, it is possible to program the **TAM** controller in Python, similarly to what was presented for the previous Python wrapper. Additionally, it is presented a C++ interface between this **TAM** implementation and ARGoS itself, which is required to simulate **TAM** inside ARGoS.

The structure of the implementation, presented in the next sections, is modelled on the basis of the original **TAM** implementation, and the existing code-base was reused wherever possible.

Note that this implementation doesn't support yet the use of a real **TAM**: it is provided the general structure required to implement the code on real device, along with the documentation, the synchronization mechanism (modelled after the one used in Java), and the necessary data structures.
It would have taken a much larger amount of time to implement all the code on the real device, making use of **XBee** and more.
Still, having the time it would be easy to extend the current implementation to support the real device.

3.1 *Installing PyTAM*

Here it is presented a brief guide on how to install **PyTAM**. It is assumed that both **ARGoS** and the **e-puck** plugin have been correctly installed.
It is not required to have the **Python** wrapper installed in order to use **PyTAM**, but it is required to have **Boost.Python**.

- Make sure that **ARGoS**, the **e-puck** plugin, and **Boost.Python** have been properly installed.
- Clone the repository at https://github.com/KenN7/argos3-pytam
  ```
  git clone https://github.com/KenN7/argos3-pytam argos3-pytam
  ```
- Move into the repository folder, then into the `argos3` folder, create a `build` folder and move into it.
  ```
  cd argos3
  mkdir build
  cd build
  ```
- Build **PyTAM**.
  ```
  cmake ../src
  make
  ```
- From the build folder, it is possible to launch some examples to see if PyTAM is working correctly. For instance:
  ```
  argos3 -c ../src/testing/test_tams.argos
  ```

PyTAM is divided into 2 main blocks: one is the Python implementation itself, while the other is the C++ interface that connects it to ARGoS.

The Python implementation is based on the interfaces of TAM, of the experiment, and of the controller.

- `TAMInterface` contains the abstract functionalities that must be implemented by TAM: it is possible to modify the LED colors, check if a robot is present, and so on. More importantly, it is possible to set the controller of the TAM, which will regulate its behaviour.
- `ExperimentInterface` is the core of the TAM simulation: TAMs are assumed to be connected, as it might be required to check if they all contain a robot and so on. The ExperimentInterface class provides functionalities that are used to program the global behaviour of the TAMs in the simulation.
- `ControllerInterface` is the interface to program the single controllers used by each TAM, their behaviour in the simulation. Different TAMs can have different controllers, of course.
  The structure of a TAM controller is the same of the ones for the other robots, seen in the previous section. It is offered an initialization function, a reset function, and a control step function. It is also possible to access the pseudo-random number generator of each TAM (in case one wants to use different PRNG on different TAMs).

There is also a simple **LEDColor** class that handles the LEDs of the TAM by using bit-masks.

TAM.py and **Coordinator.py** are the classes that implement the functionalities of the real TAM device. As mentioned before, it is provided the general structure of the functions they have to implement, along with their documentation. However some functions, the ones that would have to be tested on the real device, are currently not implemented.

TAM_argos implements the TAM interface in a way that can be used by the ARGoS simulator. The C++ interface will communicate with this class, during the simulations.

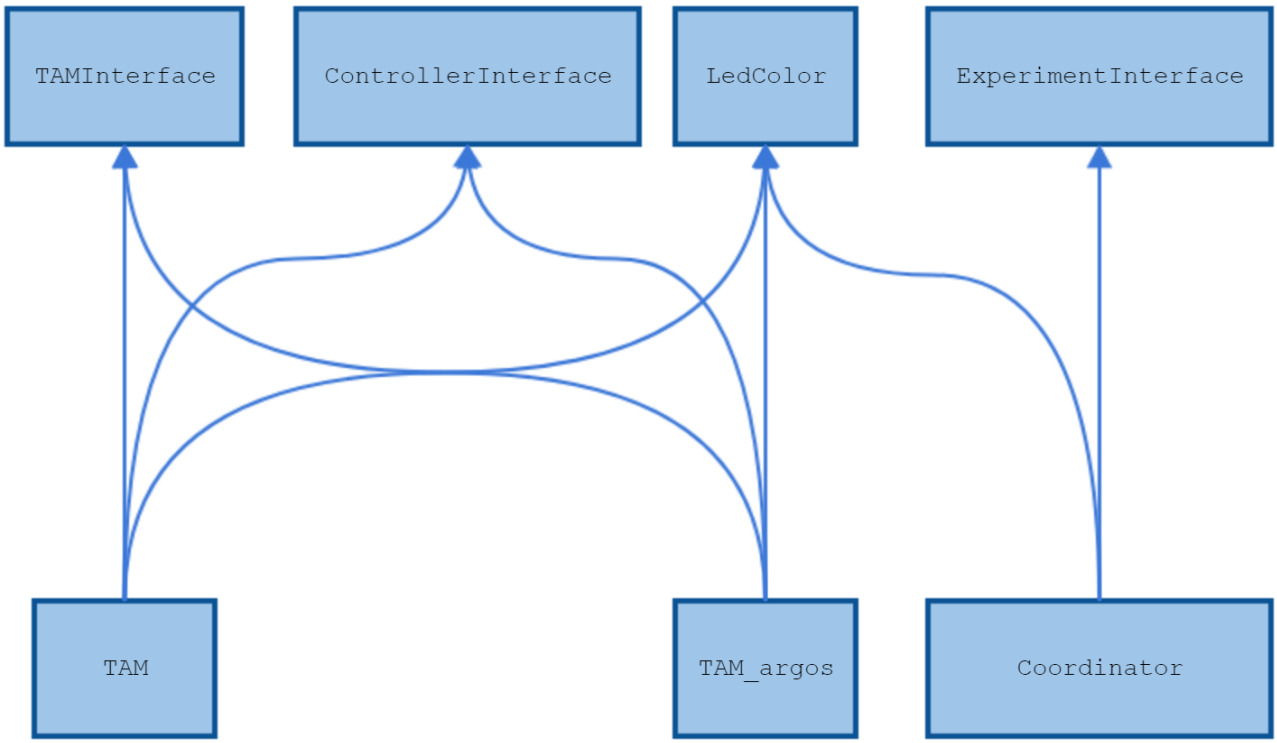

Figure 2: *Scheme of the structure of the Python component of PyTAM.*

The C++ interface reflects the original implementation, and was adapted to support the new Python implementation of **TAM**.

- `dynamics2d_tam_model` and `qtopengl_tam` are used to handle the 3D model of the TAM in ARGoS, and are left unchanged.
- `tam_entity` is also left unchanged, and it is the class that updated the local information of a TAM in the simulator.
- `tams_entity` is the main component of the C++ interface, and it is the class that interacts with the Python implementation of TAM.
  Here, the Python modules of the TAM interface to ARGoS are loaded, and the required TAMs instantiated.

  Boost.Python doesn't only allow to call C++ functions from Python, but also the opposite, i.e. to call Python functions inside C++.
  This is exactly what happens in this C++ class.
  Some parts of this class have been left unchanged, such as the check for the presence of a robot inside the TAM.

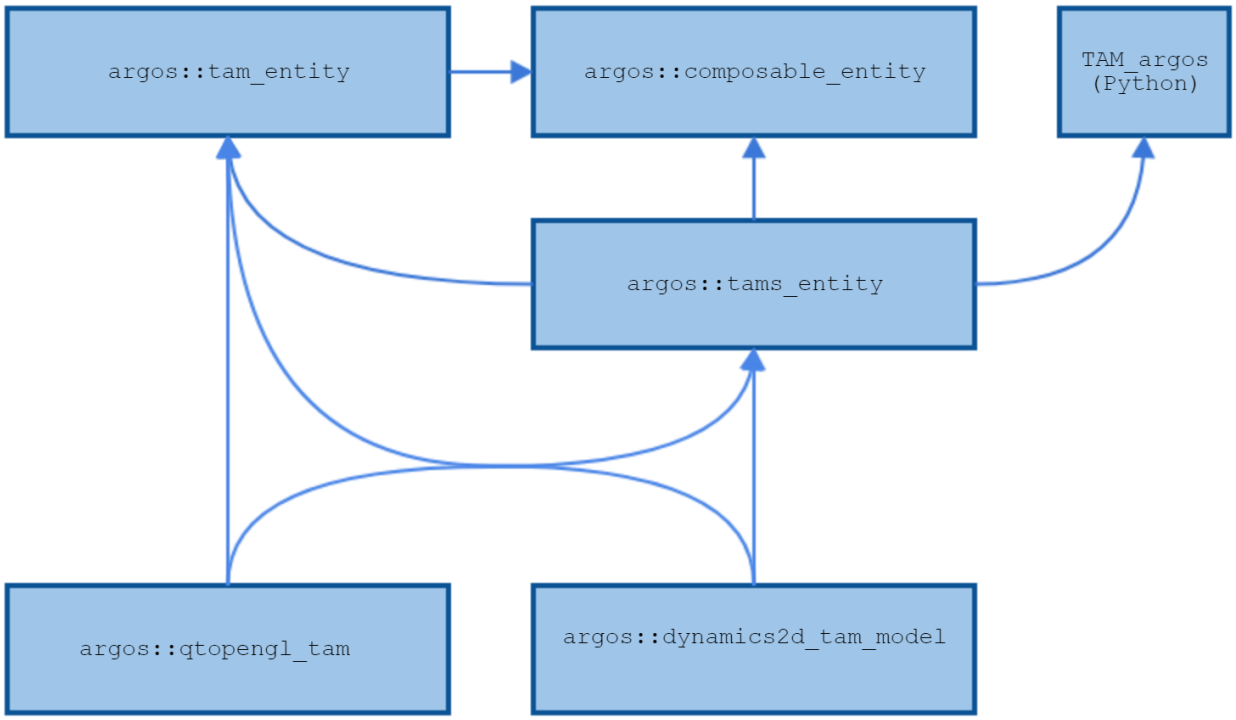

Figure 3: *Scheme of the structure of the C++ component of PyTAM.*

References

[1] Arne Brutschy, Lorenzo Garattoni, Manuele Brambilla, Gianpiero Francesca, Giovanni Pini, Marco Dorigo, and Mauro Birattari. The tam: abstracting complex tasks in swarm robotics research. *Swarm Intelligence*, 9(1):1–22, 2015.