*i*Atal Manual

Agathe Mollé Grégoire Jadi Hugo Mougard Joseph Lark
Loïc Jankowiak Noémi Salaün Rémi Bois

December 17, 2012

Contents

Preface

This manual documents the use of the application *i*Atal. As a reference manual, it will describe the different features of the application. It includes a tutorial in order to help the beginners. You will also find an help to compile, install and run the program. All the specific terms will be explained in a glossary.

Introduction

The application described here is named *i*Atal. It's a rover behaviour simulation platform. A rover is a robot which executes some tasks in a self-sufficient way, and in an unknown environment. It owns sensors that help it to feel and understand its close environment and actuators that allow it to move and to interact with the map.

The *i*Atal program can deal with any map in TMX[2] format. Those maps can be created with Tiled[1] or manually. You can also create your own exploration strategy, your rover's sensors and actuators, thanks to Python. Those features will be detailed further.

1 How to run *i*Atal

1.1 Requirements

First and foremost, *i*Atal is intended to run on an Unix-like system. It uses extensively linux-centric libraries such as gtkmm and the glib.

You can try to install it and run it on Windows since the GTK+ libraries have been ported back there, but no support is provided in this manual or anywhere else in the documentation to do so.

Be aware of *i*Atal dependencies. In the listing below, the dev suffixes are only needed if you compile the application. To run it, the standard packages are fine enough.

The following listing gives the package names for a Debian setup (so Ubuntu is good to go too). They should be similar enough in the distribution you use.

- the GTK+ library for C++ version 3.0 or higher with the headers for development. (`gtkmm-3.0-dev`)
- the Boost library (`libboost-all-dev`)
- CppUnit to run the unit tests
- Doxygen to generate the documentation

1.2 Installing *i*Atal

The install process is GNU compliant:

```
$ sh autogen.sh
$ ./configure
$ make
```

The `configure` script can in particular be called with an option to use python 3.2 instead of python 2.7 if you prefer the latter:

```
$ ./configure --with-python-3.2
```

1.3 Running *i*Atal

There are two different ways to run the program :

- Whithout any option :

  ```
  $ ./ui &
  ```

  The program will then start empty and you will have to set the map and the rover strategy later. (See chapter...)

- Directly with a personnalised map and/or exploration strategy :

  ```
  $ ./ui [-h|--help] [{-m|--map} map_path] [{-s|--strategy} strat_path]
  ```

  Description of these options :

  - `-h` or `--help` : Produce help message
  - `-m map_path` or `--map map_path` : Set the path to the map to use. It must be a file in TMX format (.tmx) (See chapter...)
  - `-s strat_path` or `--strategy strat_path` : Set the path to the strategy to use. It must be a file in Python (.py). It requires a map to be set too. (See chapter...)

2 Importing a map

2.1 Creating a TMX map

To create your own map, it is strongly recommended to use the software Tiled[1]. Tiled generates XML files wich are on the TMX format, supported by *i*Atal. However, you should know that *i*Atal map loader is limited to maps with 3 layers. Those layers are :

- ground
- obstacles
- objects

*i*Atal only understands TMX maps with separated tilesets. The tilesets are described in TSX files (.tsx). It is required that both the tileset and the layers have the same name (the three names above).

*i*Atal ignores all the layers or tilesets that are not named accordingly to the convention above. It also ignores objetc groups and recognizes 1 property per tile only. That means that above 1 property per tile, *i*Atal has undefined behaviour: it's not clear which one it will chose.

2.2 Loading your map

Once your map is created, you just have to save it, and to load it in *i*ATAL, through the GUI or thanks to the -M option of the CLI.

3 Defining a robot and a strategy with Python

This section will describe how to define a robot and a strategy to apply to the selected map. The user will have to implement some functions in order to describe how the robot will behave and some of its features. These functions have to be located in a file which will be given to the application. The table 1 presents the mandatory functions to use in the strategy file.

Name	Purpose
robot_init	Sets the start position, the direction faced by the robot and the sprites used to represent it
strat	Describes an iteration of the strategy used. Sensors and actuators are used here. The robot may move or perform some actions in each iteration.
isEnded	Function returning True when the strategy is finished (the robot completed its job) or False if more iterations are to come.

Table 1: The mandatory functions

3.1 Defining a robot with robot_init

The robot_init function sets some informations such as the direction faced by the robot at start, the position of the robot at start, and the sprites used to draw the robot. These 3 actions use the global map_ variable. These three actions should be performed (or else the program may be malfunctionning), but some other actions could be performed such as the description and initialization of the sensors and actuators of your robot. You could also decide to give some extra-features to your robot such as a battery-state, this is the right place to define it, semantically (defining your actuators, sensors or extra-features somewhere else in the file is still possible).

We will now see the basic actions that must be performed and some example of what you can do.

3.1.1 Where the robot starts

This action is *mandatory*. You have to specify where your robot starts on the map. You have to use the following function :

```
map_.setPosition(xPosition, yPosition)
```

The xPosition (resp. yPosition) corresponds to the abscissa (resp. ordinate) where the robot will be placed at start. These coordinates refer to the TMX map loaded. The (0,0) box is located on the top-left of the TMX map.

You also have to specify in which direction the robot watches. Only maps with 4 directions are currently supported. These directions are defined by these pairs :

Pair	Meaning
(1,0)	East
(0,-1)	North
(-1,0)	West
(0,1)	South

Table 2: The avaible pairs to describe a direction

You have to use the following function which is *mandatory* :

```
map_.setPosition(x,y)
```

Where (x,y) is one of the pairs presented in the table 2.

3.1.2 What the robot looks like

In order to represent the robot on the map, you have to give an image uri which will reprensent it. Since your robot can face 4 different directions, it is asked to give 4 images, one for each direction the robot can face. However, you can put 4 times the same picture.

The order for the pictures is north, south, east, west.

You have to use the following function wich is *mandatory* :

```
map_.setRobotImage(northImg, southImg, eastImg, westImg)
```

Where northImg (resp. southImg, eastImg, westImg) corresponds to the image to use when the robot faces north (resp. south, east, west).

3.1.3 Sensors

You have to describe the robot's sensors. A sensor is a feature that gives the possibility to your robot to have information on its environment. Even if you can instantiate and use a sensor from anywhere, it is highly recommended to declare it in the robot_init() function (or in a subfunction called by robot_init()).

A sensor can get an information about a tile at a fixed range, and on a precise layer. You can find the list of the avaible layers in section 2.1 on page 3.

For example, it can have access to the tile right in front of the robot (range 1), and sense the Object layer (enums.Level.Object).

To declare a sensor, you may use the following syntax :

```
global mySensor
mySensor = sensor(map_,enums.Level.Ground,2)
```

Where enums.Level.Object is part of the avaible layers, 2 is the range of the sensor (it could be 1, 3, or any positive integer). In this example, the sensor gets the tile on the Ground layer at a distance of 2 tiles ahead of the robot.

Of course, it is recommended to use better names for your sensors. Here, we could have called it groundSensor2.

3.1.4 Actuators

You have to describe the robot's actuators. An actuator is a feature that gives the possibility to your robot to modify its environment. Even if you can instantiate and use an actuator from anywhere, it is highly recommended to declare it in the robot_init() function (or in a subfunction called by robot_inti()).

An actuator can modify a tile at a fixed range, on a precise layer and in a definite way. You can find the list of the avaible layers in section 2.1 on page 3. An actuator replaces a tile by another tile. The new tile must be defined in the tileset of the map. Its name is used as the description of the tile.

To declare an actuator, you may use the following syntax :

```
global myActuator
myActuator = actuator(map_,enums.Level.Object,1, ''toto'')
```

Where enums.Level.Object is part of the avaible layers, 1 is the range of the actuator (it could be 2, 3, or any positive integer) and toto is the new tile that will replace the current one. In this example, when an myActuator is activated, it replaces the tile in front of the robot on the Object layer by a tile named "toto" which must me described in the tileset used in the map.

3.1.5 The compass

The compass is another type of sensor wich is provided by the application. Even if you could emulate a compass with python by remembering in which direction you started, and updating your current direction each time you turn, the compass sensor allows you to get your current direction at any time.

You can define a new compass by using this syntax :

```
global compassSensor
compassSensor = compass(map_)
```

3.1.6 Extending your robot abilities

Of course, you can add many features to your robot. You could decide to extend the sensors and actuators with a battery state, and updating it each time you use one of these.

You could also add a memory to your robot, so it can remember the paths it took.

You are free and encouraged to extend your robot abilities so it reflects as much as possible the abilities and constraints of a real robot.

3.2 Defining a strategy with strat

As mentioned above, the editable strat function is the procedure that the robot will follow for one iteration of the overall strategy. That is to say, defining the sensors, actuators of other available actions the robot will use and how, from the tile it is currently on.

Some variables can be memorized in order to change the behavior of the robot for a given iteration of the strategy, but every step of the strategy must be defined by the strat function.

3.2.1 Using the sensors and the actuators

The sensors and actuators are called in the strategy by adding .activate() to the method, for instance

```
groundSensor.activate()
```

will return the information given by the ground sensor.

This is and has to be the only way for the robot to get any information about the map.

3.2.2 Some useful functions

Other functions than the sensors and actuators are available, but they can't be defined as such because they interact with the robot instead of the map. These are the basic movement functions to turn left (turnLeft), turn right (turnRight) and move one tile ahead (walk).

From these basic functions it is possible to build new useful methods, for instance moving two tiles ahead could be defined by two iterations of the basic walk function.

Of course, and as it is said before, you can implement many more extended methods this way.

3.3 Specifying the end with isEnded

Unless you want to create an infinite loop, your strategy has to have an end. You can set the end of the strategy by using the ended variable to "True":

```
global ended
ended = True
```

This strategy ending flag is tested when the iteration is over, so that the program will stop the strategy procedure when it is set to "True".

You may want to use this variable with a condition, so that it will only trigger the end of the strategy on a given iteration.

References

[1] Tiled Map Editor. http://www.mapeditor.org/. [Online; accessed on December 17, 2012].

[2] TMX map format. https://github.com/bjorn/tiled/wiki/TMX-Map-Format. [Online; accessed on December 17, 2012].