Université Libre de Bruxelles

Projet d'année PROJ-H-402

Automode Visualization Tool

Authors :
Yordan **Azzolin** *000392842*

Professor :
Mauro **Birattari**
Reader :
Antoine **Ligot**

1 Introduction

This report will cover the implementation of the visual tool for automode FSM (finite state machine) mode. This tool also creates visuals for the BTrees (Behaviour Trees) but this report will not talk about that as it is the goal of another report. This was part of a students project as year's project. The language used is javascript as it is a very commonly used language for the design or website. We also have a json database to record the models and the parameters as archetype for all the elements used. We chose json for its affinity with the javascript and the user-readability and modifiability. The GUI is made from scratch as it is also an academic project and one of the points is to learn. We decided that this project was a good way to learn about GUI and how to make them using SVG. This programs purpose is to generate the string used as description of the FSM in Automode from a graph made with the tool ; and from a string, generate a graphical view of the FSM as a graph. This report will start with explaining what every files purpose is, starting from the first layer up to the last one containing the specifics of each mode this tool can be used in. Then, this report will explain how the specifics of the SFM work. I will explain what a node and an edge are in this case, and how the import export work.

2 Generalities about the file system

This section will talk about the different folders making the tool and what is expected to be found in each file in this file system.

2.1 root layer

In this layer we have the files that are not directly needed to run the visualizer. It contains the user manual, called INSTRUCTION.md. This explains how to use each part of the program. Another file on that layer is the README.md that explains how to use the two other files there, the run.py and the config.ini. The run.py uses python3 to create a lightweight server using the module flask. This server is used to execute the command line to lauch a simulation using argos with the epuck and automode module. The other file needed for the server is the .ini file where the path for argos and automode is required to be written by hand. The last item in it is the "src" directory that will be the second layer.

2.2 src layer

This layer is the common layer. It contains everything that make the tool run.

— *array_utils.js* This contains utility function to work with arrays in a more efficient way. It contains the function "add" applied to arrays to verify if an item is in an array and push said item if it is not in the array, for example.

— *cmdlinestring.js* This contains anything related to exporting importing string to and from either a file or the comand line.

— *elementmodels_default.js* This contains the default value for the nodes and the edges. Those values are used if the values in the json files are not avaiable.

— *grapheditor.js* This contains the GraphEditorElement "super-class", the GraphEditor-Tool "class" and the GraphEditor "class". The GraphEditorElement is the canvas class from which the node and edge classes will inherit. In this file all the common function for both are declared. The next class is the canvas for every tool like "create node" or "create edge" It defines the common functions in all of them. The last class defined here is the

GraphEditor class. This is the controller class. It is the entry point for the data and exit point for the distribution of it in the rest of the program.

— *grapheditorelements.js* In this file, the function inheriting form the GraphEditorElement class are defined. The two classes that inherit are the GraphEditorNode and the GraphEditorEdge. They are the classes that define the Node and the Edges. Once created those objects will contain the data and the controller of this data. They also will draw the node and edge they are linked to on the screen using SVG. The node and edge behaviour is very different between the FSM and the BTree, but the functions to access them and create them are the same.

— *grapheditortools.js* As the same say, it contains the tools used to interact with the graph being created. I will be linked to the graph editor to send it the commands of the user : "select", "create", "drag"...

— *graph_utils.js* The utility functions for the graph are inside this file. There is a function to create a graphical element, a function to add 2 vectors and a function to substract them.

— *init.js* This contains the function to initialize the work environment. It contains 2 functions for the 2 modes and functions to load the json files corresponding to that mode.

— *jquery − 3.3.1.js* This is a library found online (https ://jquery.com) to load the json files into the models and parameters canvas. They dont load or save the data the user inserts though, just the look and content of nodes and edges.

— *sidepanel.js* This takes care of the area where the user can choose which parameters to give to which graph element. It consists of scrolling menus that contain the avaiable options.

— *grapheditor.html* This is the web page that draws the graphs and lets the user interact with them.

— *style.css* Obligatory CSS file to help organise and stylize the different parts of the visual.

— *fsmfolder/btreefolder* those are the folders containing the mode specific files.

2.3 fsm layer

This and its brother folder "btree" contains the following files :

— *edgemodels.json* The graphical part of the edge only, no information about the data.

— *edgeparams.json* The content of the edge. It is the data that the user and the program needs. It will contain all the transitions in case of the FSM. (And nothing at all in the case of btree since the edges only show which node is the child of which)

— *expoter.js* Export in this case is to translate the graph into a string. This file contains everything to do that. It will gather the elements of the graph and turn them into a string that works in AutoMode. It fulfills the same purpose whether it is on FSM mode or btree. I uses the grammar explained below to create a string and writes it in the area purposed to this effect. The grammar used for the btrees is different and will not be explained in this report as it was not point of the project.

— *importer.js* this file is the opposite of the exporter, it reads the string and using the same grammar, generates the graph that represents this string. Again, the idea is the same for the btree but the method and the grammar is different and will not be explained in this rapport.

— *nodemodels.json* The graphical part of the node only, no information about the data.

— *nodeparams.json* The content of the node. It is the data that the user and the program needs. It will contain all the behaviours in case of the FSM. (And the different types of nodes and their behaviour in the case of btree since the nodes are all the data of this type of graph)

3 Specifics about the Finite State Machines part

The individual part of the project starts here. This part shows the implementation of the FSM mode of the tool and what is related to it. The core about FSM are the states :

3.1 nodes

The nodes of the graph represent the states of the finite state machine. Each node can have its own behaviour, which you can choose in the right panel. Their behaviour is described in the node parameter file in the fsm directory. If another behaviour is needed, one can just add that new behaviour to this file, and it will be loaded with the other behaviours when calling the init function. The json file is structured as follow :

— First comes the node type, in this case the only one is State, at the id "0".

— Then, comes the behaviour types, in this case the possible ones are : "Exploration", "Stop", "Phototaxis", "Anti-phototaxis", "Attraction" and "Repulsion". The point of each state is not discussed here as it has no bearing in the rest of the reasoning. Every behaviour is independent and non inter-compatible, as a repulsion node cannot also be an attraction node.

— The last layer is the detail layer. It had the detailed parameters of said behaviour type.

example : a node can be in a state (node type) of attraction (behaviour) with a attraction(att) of 5. It will have as variables found in the json, a nodename 'State', a nodeid of '0', a categoryid of "s" and inside a last variable that is a list of the parameters of each transition. The behaviour "attraction" has a parameters inside params called "Attraction" that has an id of "att" with minimal and maximal values.

3.2 edges

The edges in this FSM represent the transitions. Each transition like the nodes will have their parameters which you can choose in the same panel. Their behaviour is described in the edge parameter file that is in the same place as the node one. This json file is structured the same way as the node file :

— First comes the edge type, in this case Transition with also the id "0".

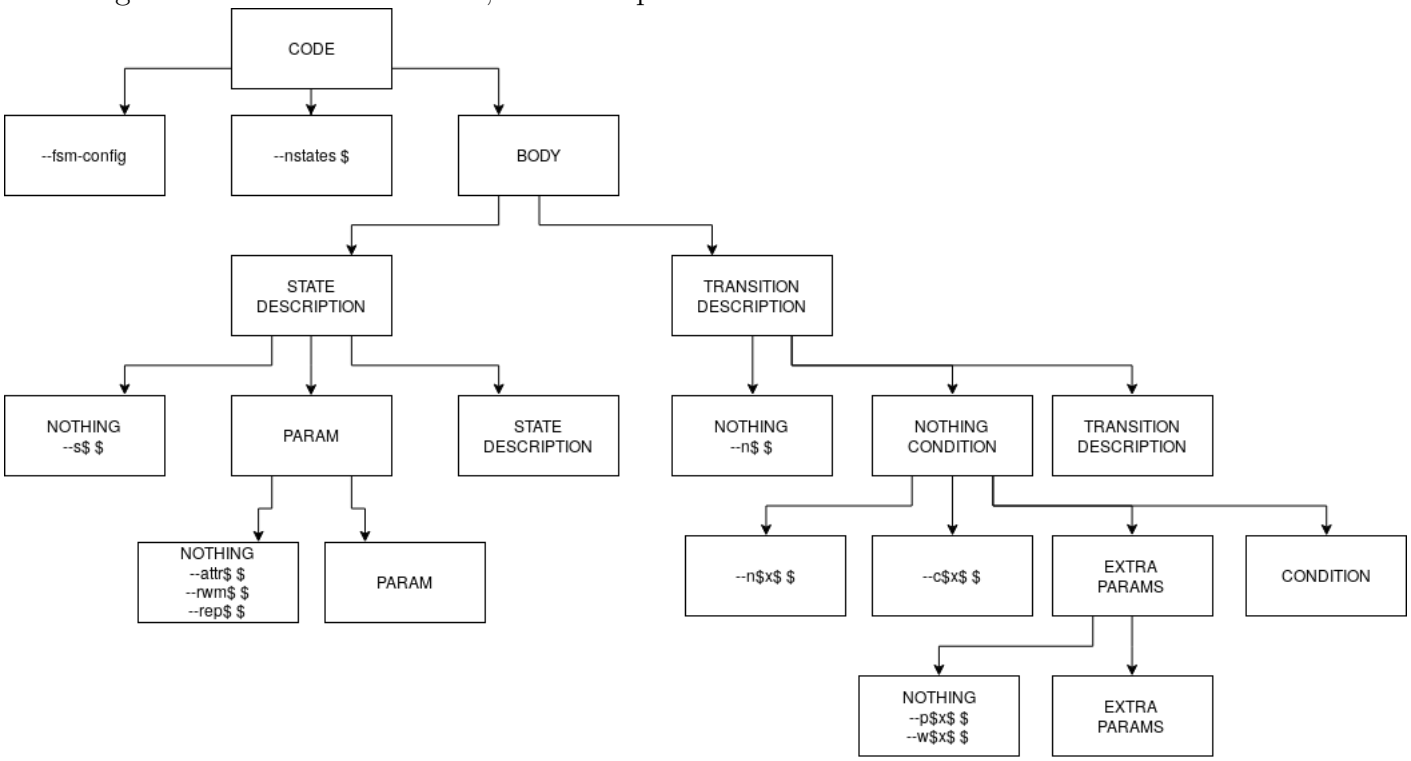
— Then comes the conditions, in this case : "Black/White/Grey floor" "(inverted) neighbour count" "fixed probability" "light". Those are exclusive, as there is no mixed condition.

— Those have inner parameters to define every state, like the most often present : probability.

example : an edge could be to switch state if the ground is black with a probability of 0.5. The condition would be "black ground" and the parameter is the probability.

3.3 import/export grammar used

The grammar used is the follow, seen as represented on this action tree :



This is the same for the import and export, meaning one must follow this grammar to be able to draw a graph from the tool. The way of reading this tree is from top to bottom then from left to right. Meaning that the first element to be present has to be "–fsm-config", then "–nstates $" with the dollar sign signifying on this graph a number. A valid string would be "–fsm-config –nstates 1 –s0 –rwm0 0".

3.4 export

The process to export is as follows :
First, generate the config part ; then gather all the nodes and generate the number of states.
For each state, write down the state number and it's type, then depending on the type, generate the parameter if one is needed.
The next step will be to gather the edges, write where they are coming from and to where they are going as well as the type of transition it is.
For each transition, write the parameters corresponding to that transaction. Along the wayn there will be checks that will trow errors if the grammar is not respected but from the graph creation there can be no mistake done as the user is limited to what the tool has to offer. Indeed if it works on the graph it will be written on the string. Though, some "illegal" moves might still be done as the tool offers a lot of liberty to be more easily modded to account with the eventual future updates of automode. One of them is for example drawing 2 edges from one node to another. This will write the 2 edges as 2 separate edges and will not bring an error on the syntax because it is a logic error.

3.5 import

The process to import is the reversed of the export process, the grammar given forces an order for every part of the string, which allows to use a left-to-right procedure :
First, the importer will verify if there is a config and nstate part. If there is not, it will make

an alert explaining where the problem is.
For the number of states it will read which behavior of state it is and create the according nodes.
Once it is done, the code will take every transitions it finds one after the other. Once it has a transition, it will verify the source and the destination, then crate an edge with the parameters given. If some parameters are missing it will make an alert explaining that a parameter is missing.
That done, the graph is complete with every node and edge. The exporter rewrites the line as it sees the graph to avoid unexpected behaviours. This importer will read the syntax corresponding to the grammar used above but it might yield unexpected results if the grammar is used incorrectly, or yield errors to strings that might work in automode but do not respect the grammar stated above. As the grammar stated above is compatible with the automode grammar, it is not a real issue.