Visual Parametric Maze Generator DSL *

Corentin Moiny[1]

304, 5e Avenue Mailloux, La Pocatière. Quebec. Canada

University of Montreal[a]

[a]*2900 Edouard Montpetit Blvd, Montreal, Quebec. Canada*

Abstract

This project is a model-driven approach to facilitate the generation of parametric generation of maze. Generation features are: (1) a rectangle of a given size (2) user drawing inside of the maze (3) giving personality to the make generation opening a range of complexity based on a probabilistic approach. In this project, a domain-specific language will be presented. This DSL is justified by the capacity of the Generator system to take account provided parameters. This able any user with no-specific background to generate maze with a chosen behaviour.

Keywords: MDE, Maze, Generator, Parametric, Python, Epsilon, DSL, Java, Visual

2010 MSC: 00-01, 99-00

1. Introduction

A. In the context of our Model-driven Engineering project assignment, I was charged to design a visual DSL to generate parametric mazes using a external Python program that I have also implemented. The goal of this project is to empower parameters understanding with the DSL and than produce probabilistic mazes. With this approach, anyone could generate maze with minimal or

*Full source code is available on GitHub.

Email address: `corentin.moiny@umontreal.ca` (University of Montreal)

[1]2020

no engineering knowledge. Parametric maze generation is not a new concept, our approach was highly inspired by Design-Centric Maze Generation by Paul Hyunjin Kim and al[1]. From this paper I reused the maze cells concept where each one of them represent a 3x3 tiles on the maze. I also reused the same types of rates (and added one more), as in the paper, with a probabilist approach.

B. The section *Solution* starts with information on the pipeline of the project to understand the scope. Followed section is more specific on the MDE approach used, it present: (1) the DSL with a model instance, (2) the Meta-model to give a more abstract view of the MDE solution to enable better comprehension of the DSL, (3) the M2T transformation to be handled by the Generator. The next part is the software engineering approach where the maze generator is describe by presenting: (1) a class diagram, (2) the cell generation process, (3) cell representations and a (4) explained maze output. This will lead to the evaluation of this project and the related works, followed by a conclusion.

2. Solution

In the following sections, I give details on the solution choices used and the purposes behind theses.

2.1. Overview

The project is split into two very distinct part: (1) MDE and (2) Generation. To give a good synopsis of the project, I provided Figure 1 to grasp how it was build. We can observe the purple part to be MDE related and the yellow part to be Software Engineering related. The pipeline of the project is structured as follow: (1) Build the Meta-model. (2) Generate *Emfactic* sources and designed the DSL semantic. (3) Create a dynamic instance of the root object. (4) Define transformation. (5) Produce a JSON valid output from this M2T transformation. (5) Fetch the data with the Generator program. (6) Last but not least, generate the maze output.

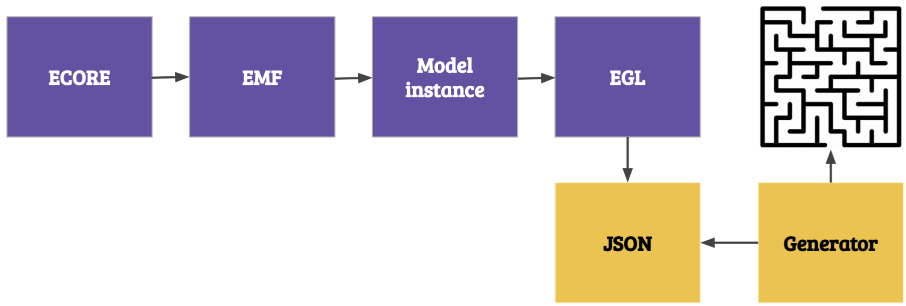

Figure 1: Pipeline of the project

2.2. DSL

The domain specific language represent the parameters used to generate the maze. Presented as a visual syntax in Figure 2, it contains four types of generator. From left to right, generators are represented as blue rectangles: (1) *RGen* is the first step of the maze generation, it gives the initial borders of the maze using a row count (*RC*) and a column count (*CC*) represented as red squares. (2) *FPGen* inject maze cells in this initial shape to force a pattern, it allows users to create drawing in the maze. Cells are represented as orangish square (*Marked 15 with a CP*) where a point is defined inside of it. In Figure 2, we only force a single cell. (3) *SPGen* is the generation of a solution path with specific parameters, allowing to gives different behaviours from the general maze body. Used rates are represented as green circles. (4) *MBGen* is the last step, the maze body generation. Using rates as green circles also. The main reason for choosing the visual, rather than textual, approach for the DSL is for the representation of forced pattern cells in the maze where used is able to create more complex drawing. Based on Eugenia documentation[2], there is a way to integrate custom images into a DSL, after many hours of debugging, I was not able to do it, concluding this is probably a tool issue. My original idea was to integrate maze cells as in Figure 6.

Types of rate. The DSL uses 4 types of rates: (1) StraightRate marked as *SR* that represent weight of straight path. (2) TurnRate marked as *TR* that

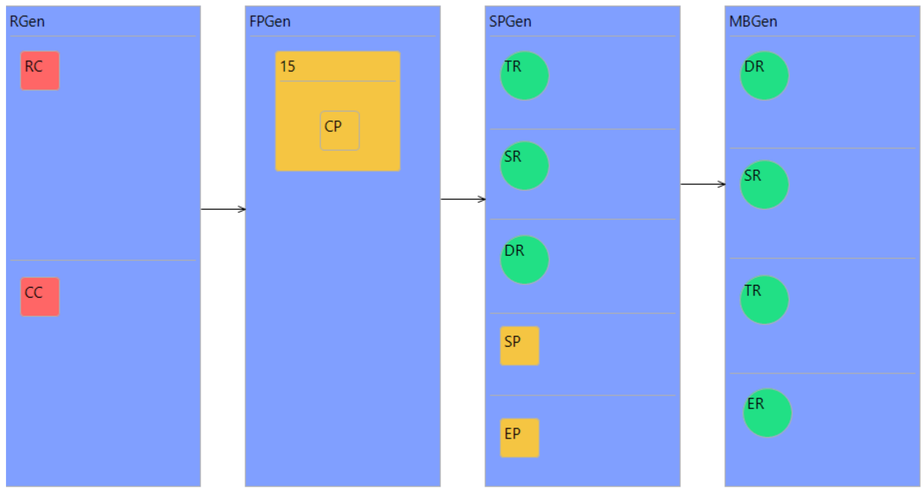

Figure 2: A model instance from the DSL

represent uni-directional turning path. (3) DecisionRate marked as *DR* that represent bi-directional turns and crossroads, the higher the rate is more the player will have to make decision to find the correct path, making the maze more complex to solve. (4) EndRate marked as *ER* that represent the famous dead-end, also known as *cul-de-sac*. The user apply weight on each rates, this will determine the behaviours of generation. Weight are used in a probabilist approach. More details on this process will be given in the Generator section.

2.2.1. Meta-model

As mentioned earlier, our DSL is a representation of different types of parameters to generate a maze. Meta-model is refereed in Figure 3. This generation is a sequential order of four steps: rectangle, forced patterns, solution path, maze body. The root Class of the meta-model is *MazeDiagram* with mandatory attributes for the four generators steps. Three abstract Class are used here to encapsulated shared concepts between generators: (1) *Count* with a integer attribute *value* used by *RectangleGenerator* to represent row and columns counts.

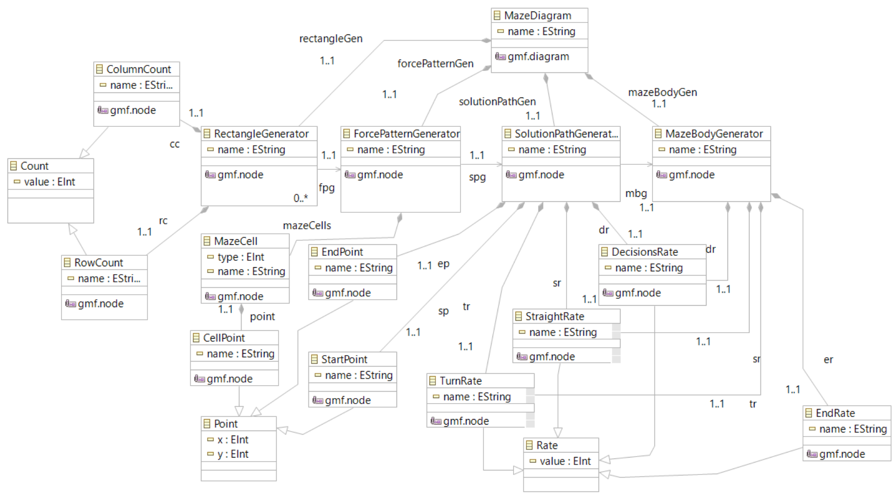

Figure 3: Meta-model of the DSL

(2) *Point* used to locate forced maze cells on the grid for *ForcePatternGenerator* and also to locate starting and ending point for *SolutionPathGenerator* (3) *Rate* to gives weight to probabilistic algorithms that defines the solution path and the maze body (*SolutionPathGenerator, MazeBodyGenerator*). Note the EndRate is only used during the maze body generation as it doesn't makes sense to create solution path with no possible exit. The use of multiplicity *1..1* almost everywhere made sure every parameters are contained in the graph, so parameters will be missing during the generation. Multiplicity for *mazeCells* is *0..** since forcing cells into the maze is an optional features.

2.2.2. Model to Text Transformation

Using a valid model instance, we can produce a JSON file that is readable by the Generator. Once this file is produced using an EGL defined transformation the JSON is put into a folder where the Generator fetch data. No manual operations on the outputted file is needed.

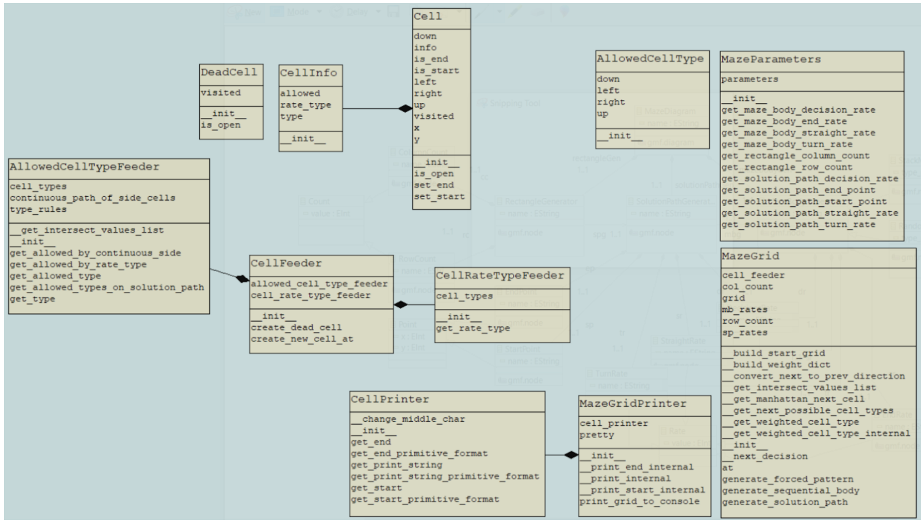

Figure 4: Class Diagram of the Python program

2.3. Generator

Build as an external object oriented program in Python. As presented in the Figure 4, the software hold four main parts: (1) *Cell feeding services* used to provide cell instances during the generation process that are valid considering neighbours and rate type choice. *AllowCellTypeFeeder* provides possible cells concerning the neighbours cells, *CellRateTypeFeeder* gives set of cell that are of a given rate type (2) *Cell* that is internal representation of Maze cells in the program. *Cell* class contains information that are not subject to changes during the generation and *CellInfo* contains changeable information. (3) *Printing services* charged to gives readable output for the user. (4) *Generation algorithms or the MazeGrid Class* for every phase of the generation as represented in the DSL.

2.3.1. Determining next generated cell

There are five distinct parts to determine what will be the next generated cell in the maze. This process is roughly exposed in Figure 5 and used for the solution path and the maze body generation steps, the steps are as follow: (1) Generate a list of possible cell types for the currently generated cell considering

$$\text{single\_random}(\ \text{set}(\textit{Allowed Cells}) \cap \text{set}(\textit{Rate Cells})\ )$$

Figure 5: Formula for the next cell generation

left, top, right, bottom neighbours (2) Use associated rate weights in the input JSON file to probabilistically determine what will be the next rate. (3) With this rate, generate a list of valid cell types (4) Intersect both list (5) Choose a random value within this list and assign the type to the currently generated cell.

2.3.2. Maze cells

The generator uses a total of 16 maze cells types, as in Figure 6. Each cells have a list allowed neighbours, this list is computed in class *AllowedCellTypeFeeder*. Generation algorithms will used this features to intersect will other desired cells types to make sure all cells can connect together at the end of the generation as mentioned earlier. Each cells have associated rate to it, the class *CellRateTypeFeeder* is meant to build those lists.

2.3.3. Produced output

The output is produced on the user console where the Python program got executed. Figure 7 illustrate this output. Tiles are to be considered as follow: (1) *Brown* are the walls, (2) *Green* are the paths, (4) Blue is the player spawn point and (4) *Red* is the target point the player wants to reach.

3. Evaluation

The interesting part of this project is the MDE part since this is the real gain. Maze generation algorithm is nothing new in the world on computer science, this parameter-driven approach allow the justification of a DSL. This tool, with more work, could be used as an academic tool to learn about maze and the theory behind it, for example, we could try to find optimal parameters to output the most complex maze, or the easiest. We could also find how many

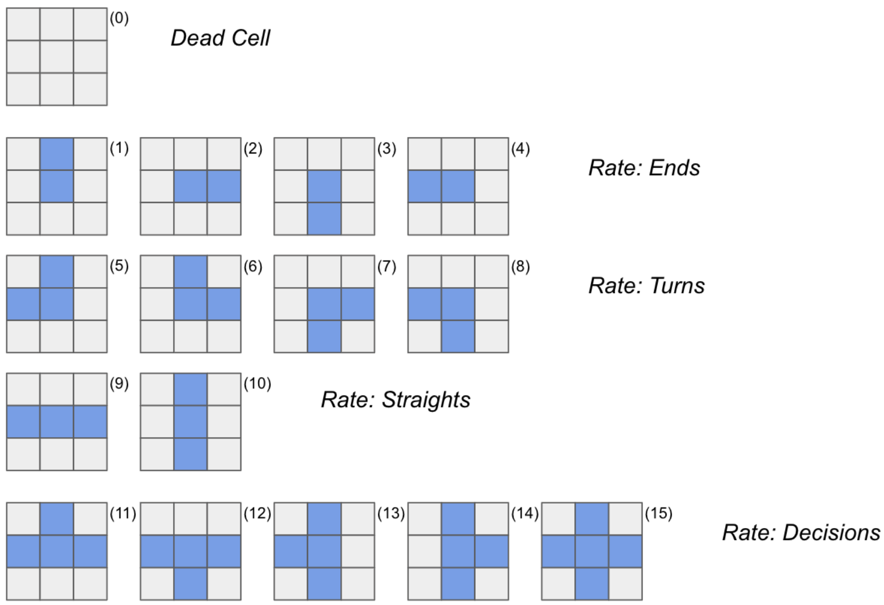

Figure 6: Maze cells with associated rate type

decision a player as to do for finding the exit before considering the maze is not doable by a human being.

4. Related Work

As mentioned earlier, maze generation is not a new things. Plenty on online tools are available, sometimes with less parameters[3]. Also, plenty of techniques exist, for example Kruskals' algorithm "can be used to splits the graph nodes into separate components and repeatedly unifies them using graph links[4]." Since the MDE community is very small today, I asked myself the question if a DSL to help for the generation of mazes exist already. It seems is doesn't in the case of a Maze and exist for a similar concept, a Pacman DSL[5]. Their approach is a Executable DSL where models reacts to their environment allowing simulation of the Pacman game.

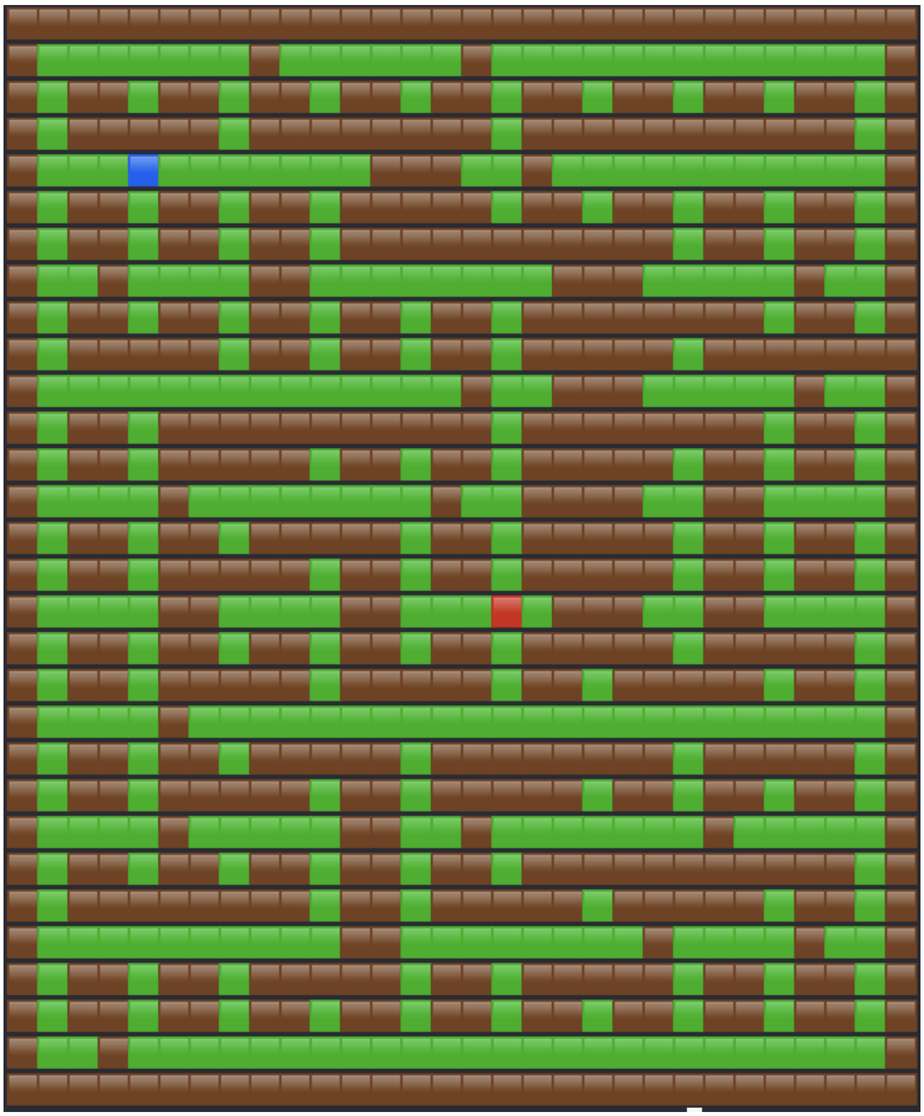

Figure 7: Maze output example

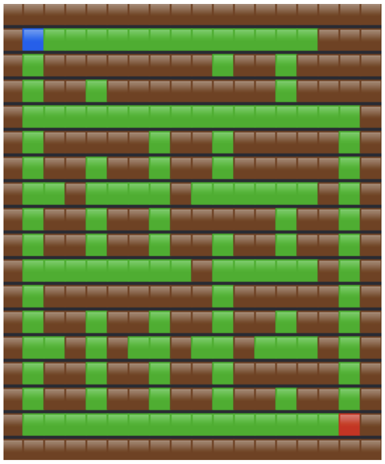

Figure 8: Small maze with very different solution path rates and maze body rates

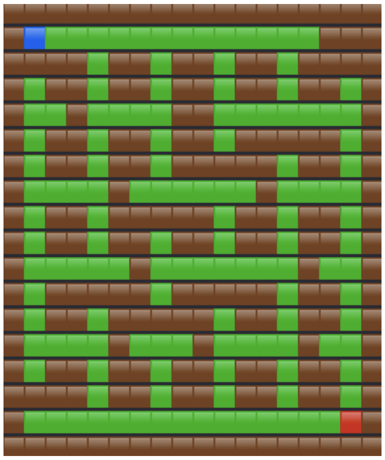

Figure 9: Small maze with very high level of decision

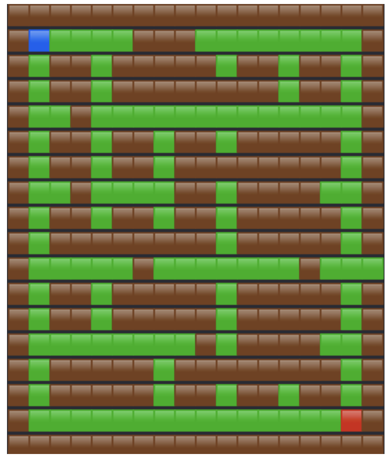

Figure 10: Small maze with cross drawn in it

5. Conclusion

The goal of giving a more user-friendly orientation to generate maze is a success within my work, except for the maze cell design representation in the DSL. The Generator program is also capable to give distinct behaviour to the solution path and the maze body, giving personalities to mazes. As shown in 8, we can clearly identify the solution as composed essentially with straight cells and the maze body with decision cells. In terms of implementation, it lacks in the generation of a single solution, especially in small mazes when we apply a high decision rate we end up having too much possible ways to reach the finish point, as shown in Figure 8. This problem could have been solved with the implementation of a maze body generation algorithm based on a *stack structure* where every cells in it represent an open decision on the solution path from where generation would be triggered. The force pattern feature is a success as it able user to create drawing in the maze, as shown in Figure 10. Some problematic encountered was the creation executing *EVL* constants. With the current implementation a user could, for example, define a player spawn point outside of the actual maze borders, creating a exception to be thrown in the Generator program. This makes the DSL lack in terms of robustness.

References

[1] P. H. Kim, J. Grove, S. Wurster, R. Crawfis, Design-centric maze generation, in: Proceedings of the 14th International Conference on the Foundations of Digital Games, FDG '19, Association for Computing Machinery, pp. 1–9. doi:10.1145/3337722.3341854.
URL https://doi.org/10.1145/3337722.3341854

[2] Eugenia: Nodes with images instead of shapes - epsilon.
URL https://www.eclipse.org/epsilon/doc/articles/eugenia-nodes-with-images/

[3] Maze generator.
URL http://mazegenerator.net/

[4] T. Bouda, Day 84: Maze generation.
URL https://medium.com/100-days-of-algorithms/day-84-maze-generation-634aaca67e34

[5] D. Leroy, E. Bousse, M. Wimmer, B. Combemale, W. Schwinger, Create and play your pac-man game with the GEMOC studio (tool demonstration) 7.