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

(TODO) - This is a test abstract again and again.

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 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. (TODO) - Details of the sections presented

⋆Full source code is available on GitHub.

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

[1] 2020

2. Solution

In this section, I give details on the solution choices used and the purposes behind theses.

2.1. DSL

The domain specific language represent the parameters used to generate the maze. Presented as a visual syntax in Figure 1, 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 1, 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.

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 represent uni-directional turning path. (3) DecisionRate marked as *DR* that represent bi-directional turns and crossroads. (4) EndRate marked as *ER* that represent the famous dead-end, also known as *cul-de-sac*. Theses rate values will determine the behaviours of generation and will be used in a probabilist approach. A random number will be generated and is associated to a weighted value, it returns a list of cell types that represent this rate type. The list is than intersected with a possible neighbours list and a random cell is chosen as a type of cell.

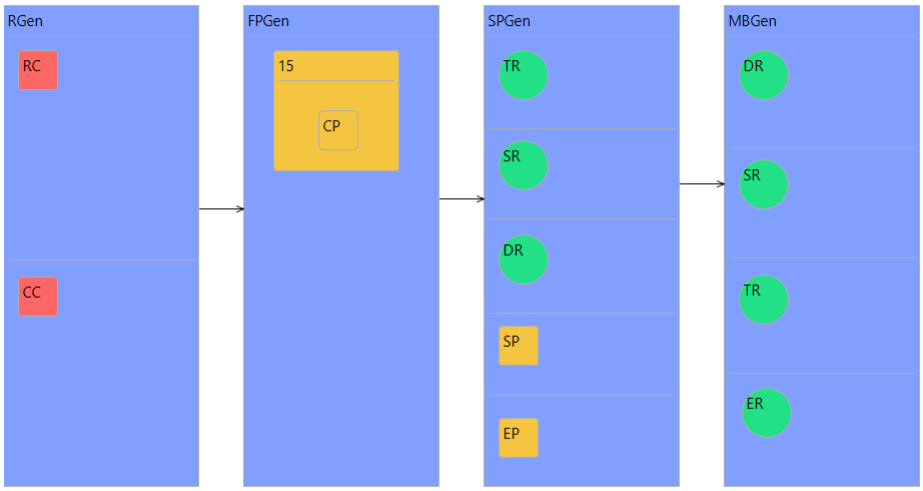

Figure 1: A model instance from the DSL

2.1.1. Meta-model

As mentioned earlier, our DSL is a representation of different types of parameters to generate a maze. 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. (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

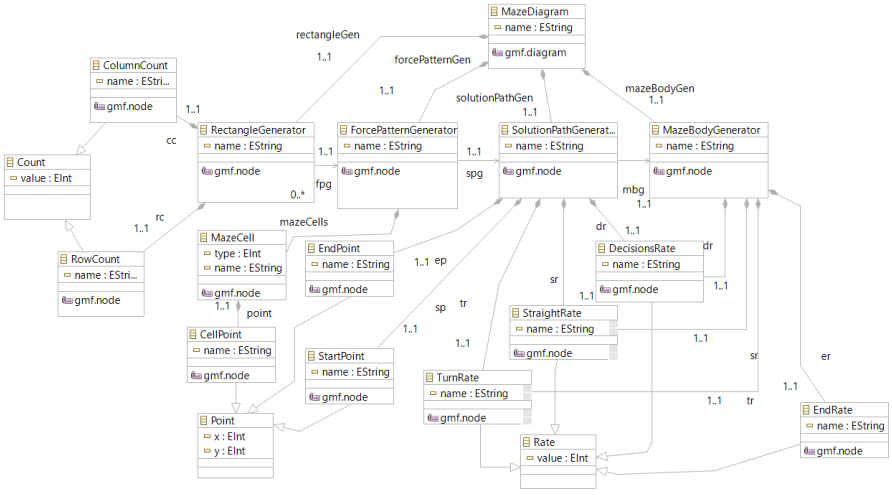

Figure 2: Meta-model of the DSL

is an optional features -¿ Present root object, link with generators (4), shared rates, abstract classes, end rate...

2.2. Generator

Maze cells. The generator uses a total of 16 maze cells types, as in Figure 3. 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.

3. Evaluation

4. Related Work

5. Conclusion

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

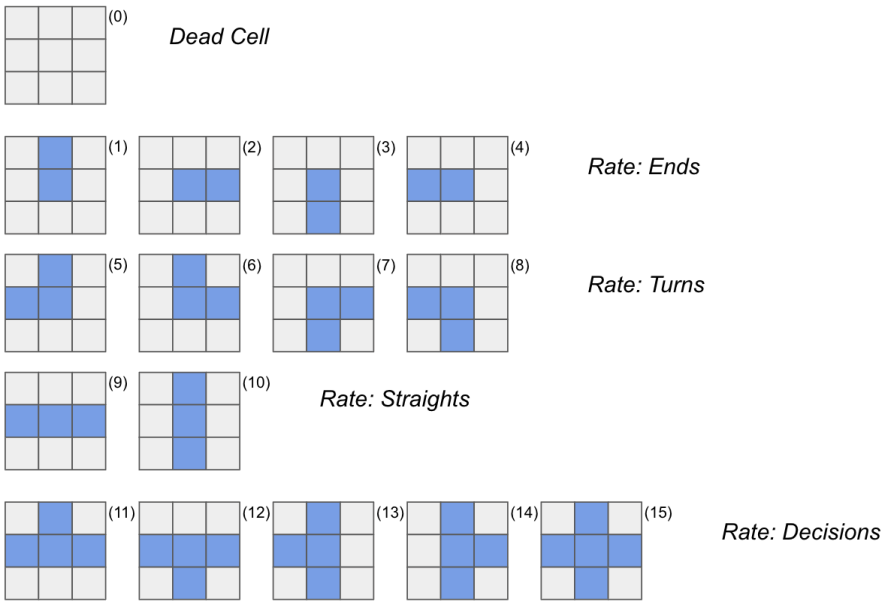

Figure 3: Maze cells with associated rate type

Digital Games, FDG '19, Association for Computing Machinery, pp. 1–9. doi:10.1145/3337722.3341854.

URL https://doi.org/10.1145/3337722.3341854