1 April 2015

Jakub Ciecierski

Cellular automaton
Technical documentation

Contents

1 Document metric

Document metric					
Project:	Cellular Automaton	Company:	WUT		
Name:	Technical Documentation				
Topics:	Technical Documentation				
Author:	Jakub Ciecierski				
File:	technical.pdf				
Version no:	0.1	Status:	Under development	Opening date:	2015-03-03
Summary:	Technical documentation for cellular automaton				
Authorized by:	Wadysaw Homenda Lucjan Stapp	Last modification date:			2015-04-01

2 History of changes

History of Changes			
Version	Date	Who	Description
0.1	2015-04-01	Jakub Ciecierski	Definition of the main purpose of the document

3 Technology

The technology of my choice is C# programming language under .NET framework. It is known to be a robust tool which allows for relatively easy creation of complex applications.

4 Algorithms

4.1 Rule

Algorithm for creating **Rules** has been prepared thoroughly to yield a robust and flexible way to implement transition of generation.

4.1.1 Definitions

We define **Rule** as a set of **Transitions** each having **Transition Function** which computes if **Cell** belonging in input **Neighborhood** should transition to **New State**.

Rule is equipped with a **default Transition** which is used for transitions for neighborhoods for which no transition was defined - it can simply be a transition which does not change a state

4.1.2 Algorithms

- **Creating Rule**
    1. Create **Rule** for given **Neighborhood Type**
    2. Create a **Transition** we want to add to Rule set:
        - (a) Define a custom **Transition Function** which takes a neighborhood and returns a boolean value
        - (b) Define **New State** to which a Cell in valid neighborhood should transition.
        - (c) Add **Transition** to **Rule**.
    3. Repeat Point 2. to add more **Transitions**
- **Apply Rule**
  Input: Neighborhood. **Output: New state**
    1. Look for proper Transition which will successfully apply to input Neighborhood.
    2. If no Transition Function was defined for this neighborhood, apply Default Transition

4.1.3 Examples

- Example 1 **Creating Rule**
  1. User selects **Rule** for 4-point neighborhood - **Rule** is created.
  2. User wants to add a transition for specific layout of neighbors (in specific states) - **Transition** is created.
     (a) A 4-point neighborhood NB_U is created based on the user's choice:
        - Local Cell State: '0'
        - Upper and Bottom neighbor State: '1'
        - Left and Right neighbors State: '0'

        Transition Function is defined so that it accepts transition if input neighborhood NB_I is exactly equal to NB_U

     (b) **New State**: '1' is defined from user's choice

     (c) **Transition** is added to **Rule** set

- Example 2 **Creating Rule**
  1. User selects **Rule** for 24-point neighborhood - **Rule** is created.
  2. User wants to add a transition stating that 5 neighbors should be in state '1', 4 neighbors should be in state '0' and Local State should be in state '0' - **Transition** is created.
     (a) A Transition Function is defined as follow:
        ```
        If input neighborhood NB_I has 5 cell in state '1' AND
        has 4 cells in state '0' AND
        local cell is in state '0' THEN
        return TRUE
        ```
     (b) **New State**: 1 is defined from user's choice

     (c) **Transition** is added to **Rule** set

4.2 Generation

In this section we define sequential algorithms which compute next generation of cellular automaton

4.2.1 Definitions

Grid is a two-dimensional matrix containing cells in some state, and a **Rule** which is used to compute new states of each cell. To achieve this, we define an algorithm **Next Generation**.
The grid can operate in **Wrapping** mode which simulates an infinite space - for cells, which neighbors would other wise run out of bound of the matrix, neighbors are taken from the opposite side of the matrix - Like in Asteroids game.

Algorithm **Get Neighborhood** is used to retrieve neighborhood - defined over a Rule - of each cell

4.2.2 Algorithms

- **Get Neighborhood**
  Input: Cell of grid. **Output:** Neighborhood of input Cell

  1. Retrieve **neighborhood type** from **Rule**
  2. Calculate neighborhood of cell based on the retrieved type

- **Next generation**
  Output: Cells in updated states
  Data: StatesTmp - List of new states of each cell

  1. For each cell in Grid:
     (a) **Get Neighborhood** of cell
     (b) **Apply Rule** for that neighborhood
     (c) Add new state to StatesTmp
  2. For each cell in Grid:
     (a) Updated cell's states based on **StatesTmp**

4.3 Rule Editor

4.3.1 Definition

Rule Editor defines the way in which the rules can be created. Algorithms in this logical module are responsible for creating rules that make sense and don't over lap - two Transition yielding different values for the same neighborhood. We define several possibilities of rule creation depending on neighborhood types:

- For 4 and 8 point neighborhoods: Layout of neighborhood matters - positions of cells and their states (See 3.1.3 Example 1)
- For 4, 8 and 24 point neighborhoods: Number of neighbors with specific state (See 3.1.3 Example 2)
- For 24 point neighborhoods: We look for number of neighbors with specific state in each column

Thanks to the flexible Rule creation methods, the application becomes much more scalable since it allows for adding new neighborhoods and Transitions for fun and creative Generation production.

5 Data structures

5.1 Grid

Is used to represent an automata, in reality **Grid** is simply a two dimensional matrix. **Nested Lists** are used to represent such structure.

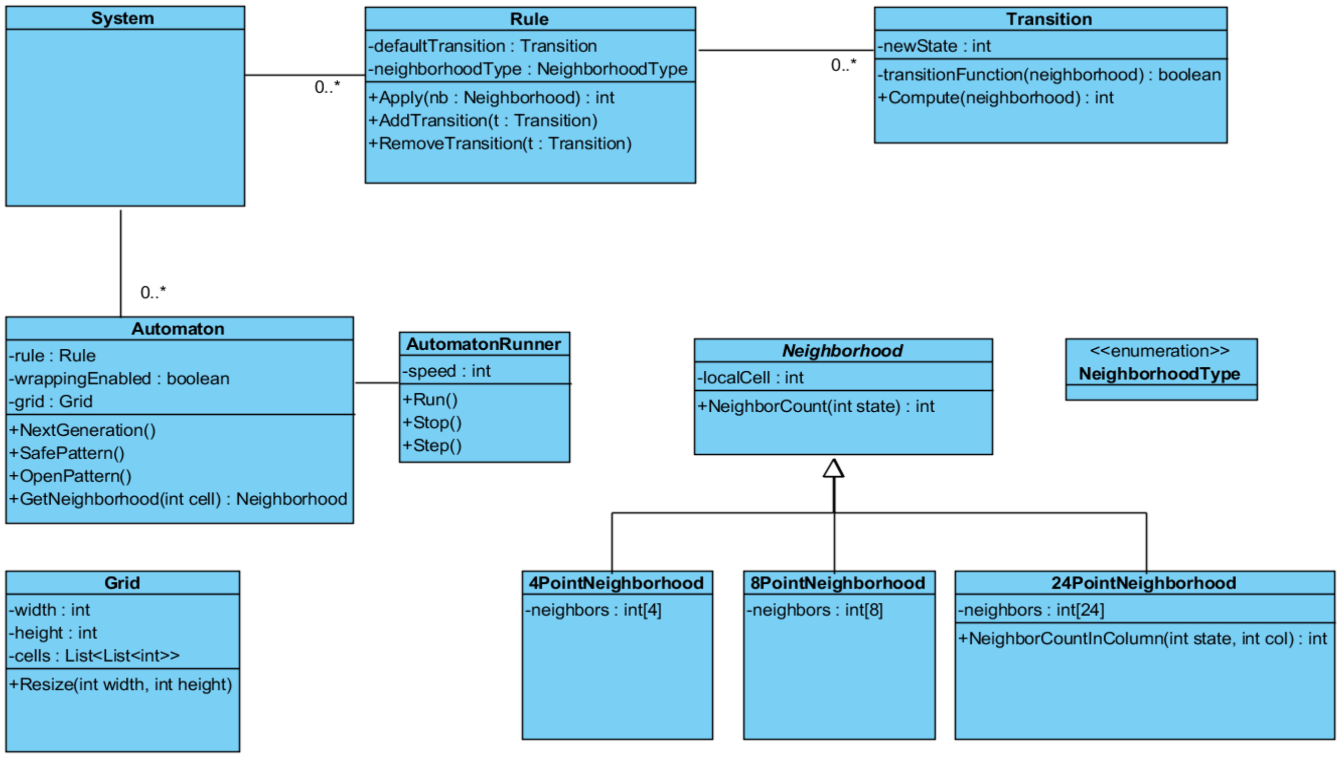
For a grid with width N and height M, We have M Lists each containing List of N cell elements.

6 Modules

- Logic
  - System - contains all application component, such as rules and automatons saved in memory
  - Automaton - the entire logic behind running an automaton, a grid of cells, rules, neighborhoods.
  - Rule Editor - defining how rules should be defined
- Graphical User Interface (GUI)
  - Main application Window
  - Grid - The automaton's grid is being drawn here
  - Editors - Used to change settings for all application components
  - Browser - Used to search for saved grids, rules.

7 Modelling

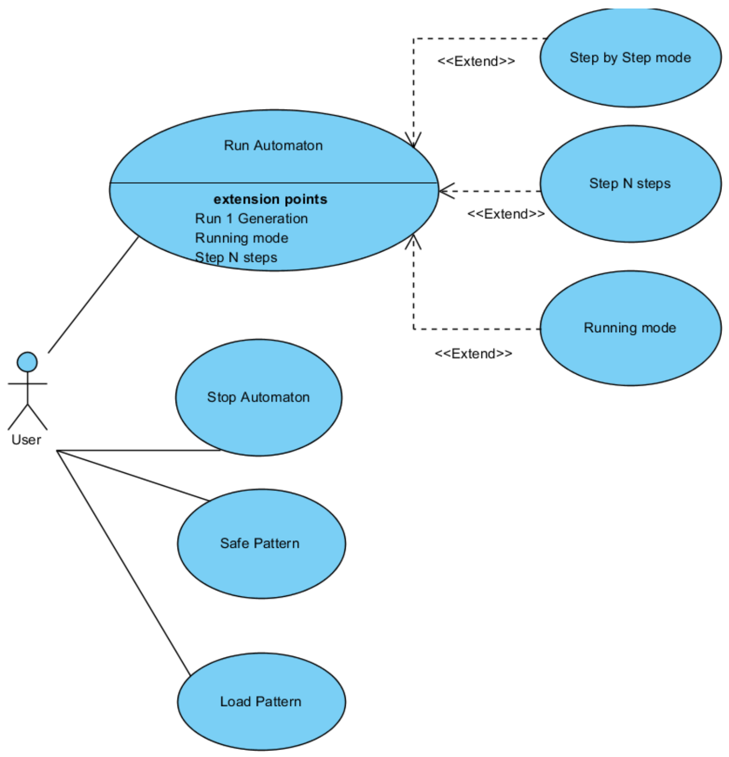
7.1 Class diagrams


System
0..*
Rule
-defaultTransition : Transition
-neighborhoodType : NeighborhoodType
+Apply(nb : Neighborhood) : int
+AddTransition(t : Transition)
+RemoveTransition(t : Transition)
0..*
Transition
-newState : int
-transitionFunction(neighborhood) : boolean
+Compute(neighborhood) : int
0..*
Automaton
-rule : Rule
-wrappingEnabled : boolean
-grid : Grid
+NextGeneration()
+SafePattern()
+OpenPattern()
+GetNeighborhood(int cell) : Neighborhood
AutomatonRunner
-speed : int
+Run()
+Stop()
+Step()
Neighborhood
-localCell : int
+NeighborCount(int state) : int
<<enumeration>>
NeighborhoodType
Grid
-width : int
-height : int
-cells : List<List<int>>
+Resize(int width, int height)
4PointNeighborhood
-neighbors : int[4]
8PointNeighborhood
-neighbors : int[8]
24PointNeighborhood
-neighbors : int[24]
+NeighborCountInColumn(int state, int col) : int

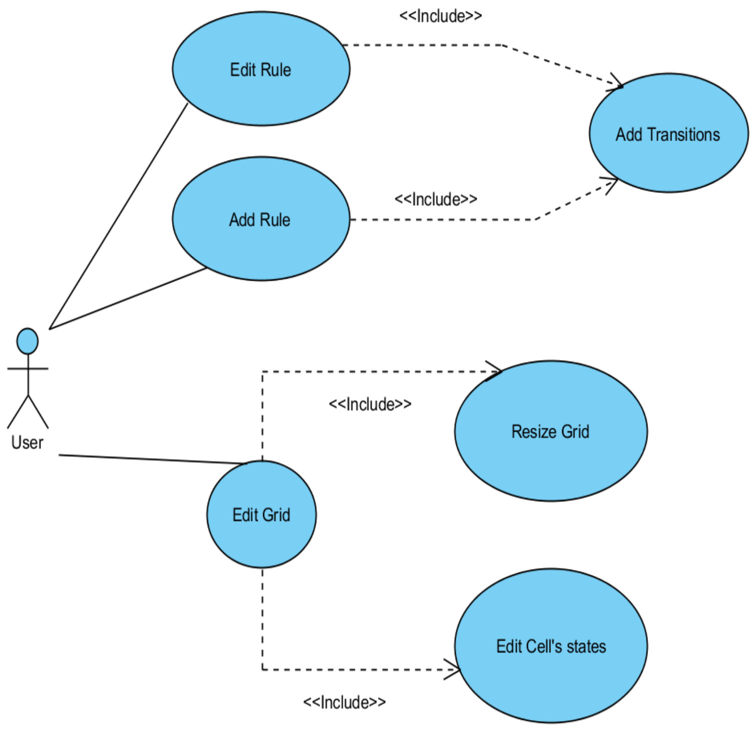
- **System** - contains all existing automata and rules saved in memory.
- **Rule** - a set of Transitions, used to compute new generation.
- **Transition** - contains a new state and a pointer to a some function defining this transition.
- **Automaton** - class encapsulating cellular automaton.
- **Grid** - data structure representing the cellular automaton in 2D
- **Neighborhood** - Defines different neighborhoods.
- **AutomatonRunner** - responsible to run Automaton.

7.2 Use cases

7.2.1 Automaton use cases

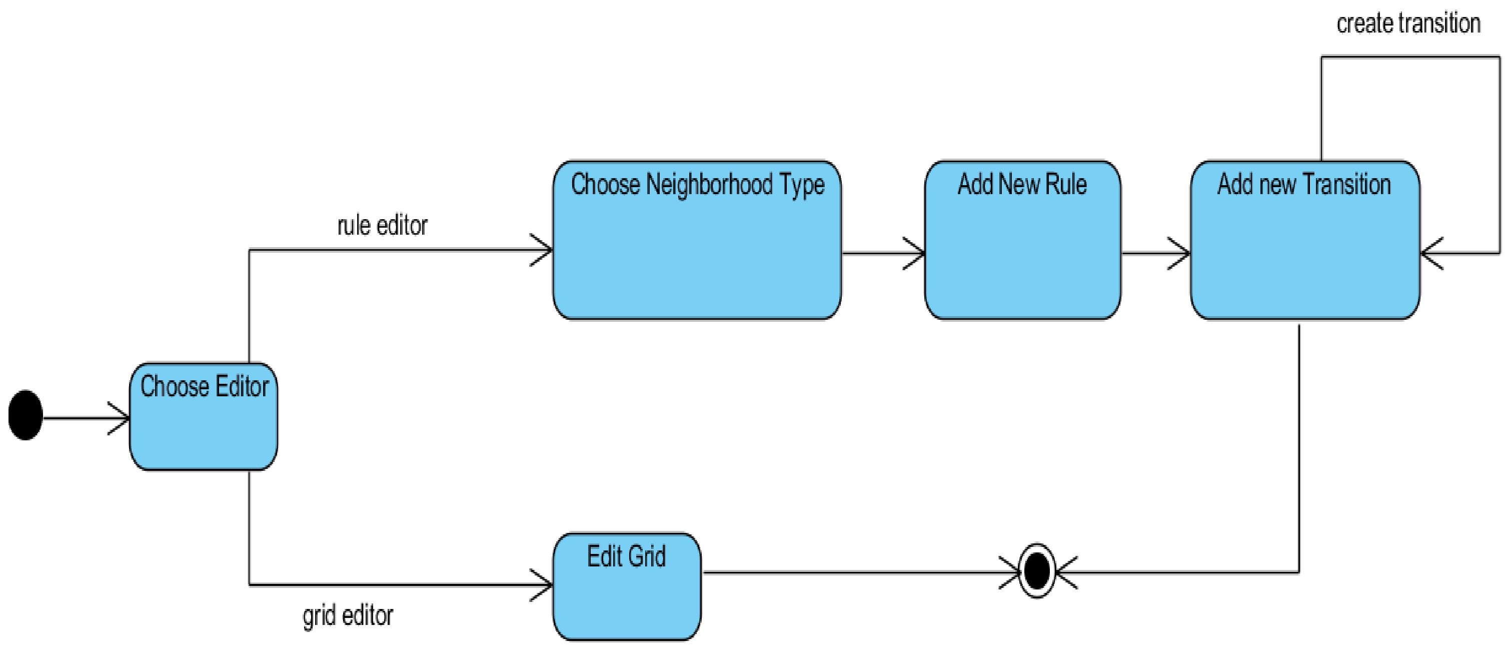


7.2.2 Editor use cases

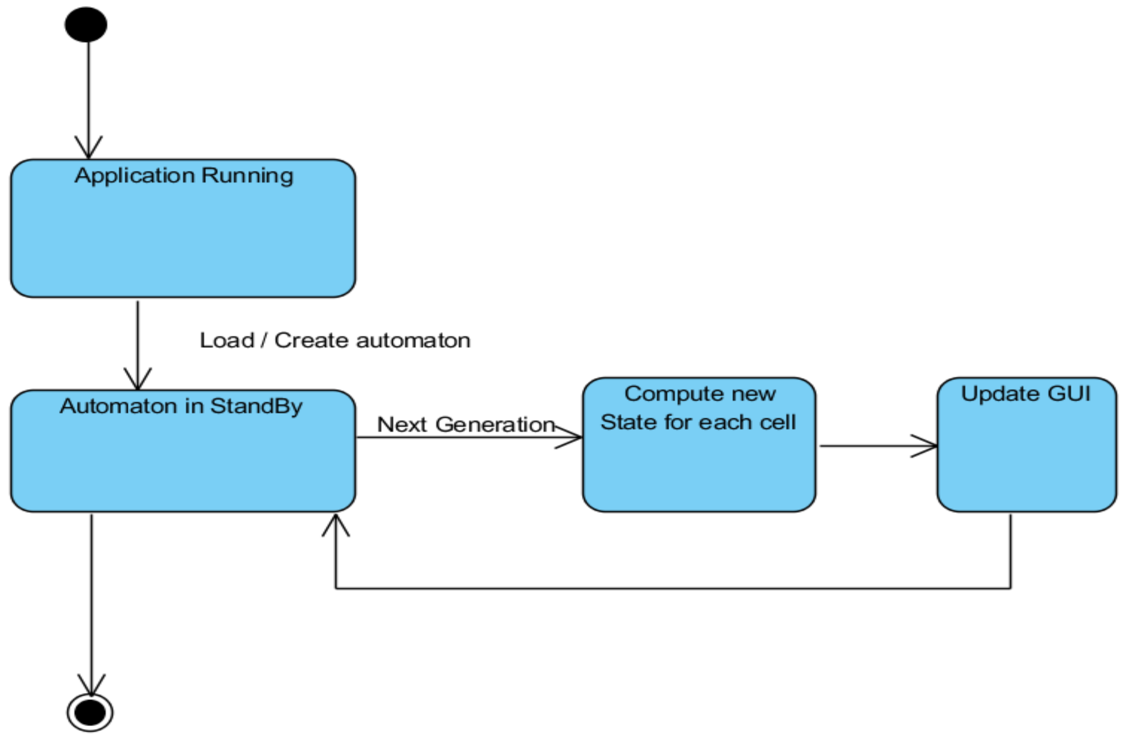


7.3 State diagrams

7.3.1 Editor



7.3.2 Automaton


Application Running
Load / Create automaton
Automaton in StandBy
Next Generation
Compute new State for each cell
Update GUI