Aleksander Lipka

Alek.lipka@gmail.com

Description

An application that will simulate a cellular automaton with a   
predefined size of neighborhood relative to specified cell

Cellular Automaton

Warsaw University of Technology: MiNI Faculty

****

Document metric

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Project | Cellular automaton | |  | Company: | MiNI PW |  |
|  |  |  |  |  |  |  |
| Name: | Specification | |  |  |  |  |
|  |  | |  |  |  |  |
| Topics: | Full specification:  Requirements + Bussiness Analysis | | | | |  |
|  |  | | |  |  |  |
| Author: | Aleksander Lipka | | |  |  |  |
|  |  | |  |  |  |  |
| File: | Cellular Automaton - Specification.pdf | | | | |  |
|  |  |  |  |  |  |  |
| Version no: | *v 1.0* | Status: | Final | Final date: | 2016-06-02 |  |
|  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Summary: | Full specification of all requirements and technical aspects | | | | |  |
|  |  |  |  | Last  modification  date: |  |  |
| Authorized by |  |  |  | 2016-06-02 |  |
|  |  |  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| Version | **Date** | **Who** | **Description** |
| 0.3 | 2016-03-24 | Aleksander Lipka | Creation of requirements specification |
| 0.6 | 2016-04-21 | Aleksander Lipka | Creation of technical requirements specification |
| 0.9 | 2016-06-01 | Aleksander Lipka | Joining of two documents |
| 1.0 | 2016-06-01 | Aleksander Lipka | Final Check |

**Requirements Specification**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| Version | **Date** | **Who** | **Description** |
| 0.1 | 2016-03-10 | Aleksander Lipka | Definition of the main purpose of  the document + specification |
| 0.2 | 2016-03-11 | Aleksander Lipka | Added missing points to specification – schedule, dictionary, evaluation and risk |
| 1.0 | 2016-03-24 | Aleksander Lipka | Final Check |

0. Schedule

* 2016-02-25 Thursday Lab 1 Start, theme’s offer, preliminary choice
* 2016-03-03 Thursday Lab 2
* 2016-03-10 Thursday Lab 3 requirement specification technical
* 2016-03-17 Thursday Lab 4 requirement specification - second term for LS group Thursday 24
* 2016-04-07 Thursday Lab 6 technical project
* 2016-04-14 Thursday Lab 7 technical project - second term
* 2016-04-21 Thursday Lab 8
* 2016-04-28 Thursday Lab 9 code of modules
* 2016-05-05 Thursday Lab 10 version 0.98
* 2016-05-12 Thursday Lab 11 version 0.99
* 2016-05-19 Thursday Lab 12
* 2016-06-02 Thursday Lab 13 version 1.0 testing
* 2016-06-09 Thursday Lab 14 test report
* 2016-06-16 Thursday Lab 15 acceptation, final grades

1. Overview

The project aims to develop the application that will simulate a cellular automaton with a predefined size of neighborhood relative to specified cell. The application shall visualize the step-by-step simulation according to initial configuration.

2. Dictionary

- **Cellular Automation** – a cellular automation is a finite collection of cells located in a grid, each in one of a finite number of states. In this project there will be only three states: dead, alive, inactive,

- **Rule** – a statement that makes cells either alive or dead, rules determines the states of cells and their neighbors,

- **Neighborhood** – set of nearby cells to the current one,

- **Step** – next or previous function call.

- **GUI** – Graphical User Interface, visible and interactive part of the program

3. Requirements

Functional:

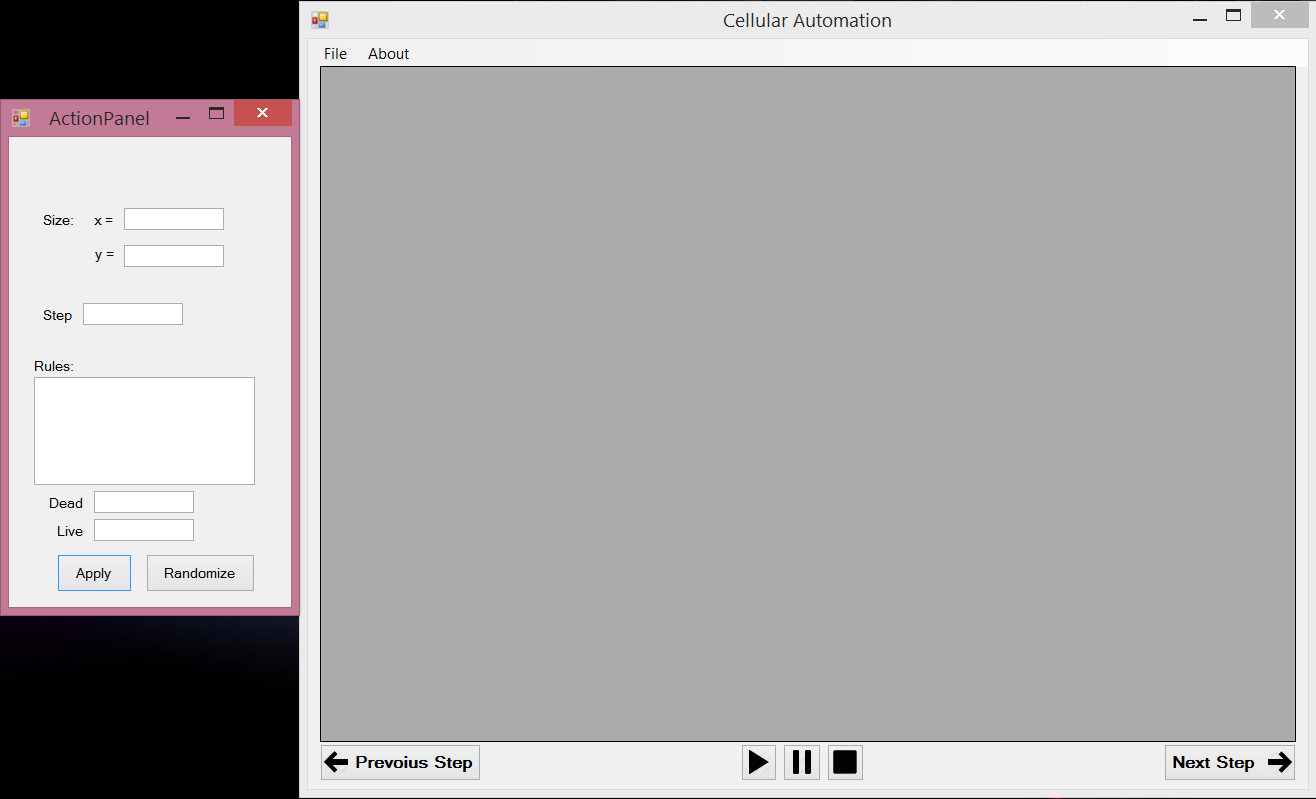
* Application shall be divided into the grid and a navigation/action panel
* Grid can be in any finite number of dimensions
* Cells in grid must have a finite number of states
* When the state of the cell is changed, the cell changes its color
* One cell will have 24 points neighborhood
* User can apply new rules from user action panel
* Application shall distinguish contrary rules and solve the problems
* Application shall start with the predefined set of rules and a starting state
* Application should run without any problem in our faculty laboratories
* Application should be presented in the executable file format

Non-functional:

* The error handler will be applied to rules maker so that the program won’t brake under the wrong syntax
* When the user will want to exit the program, the save prompt will appear first
* The program should run smoothly without any visible delays
* Application’s GUI should be done in English
* New/Load/Save options should be included in the menu bar

GUI will be composed out of two main windows – one with the grid and one with the panel with all action and navigational controls. Below is the list of all controls added to each window with a brief explanation:

4. Front-End Interface (GUI)



Picture 1 Preview(may change)

1. **Main panel:**

- matrix (with default size) 🡪 main grid, where everything will happen

- matrix fill the whole window (except for navigational bar and menu strip)

- menu strip (File -> New) 🡪Resets everything to its default state

(File -> Load visualization) 🡪Loading some saved visualization

(File -> Save visualization) 🡪Saving current visualization

- play-pause-stop buttons 🡪 action buttons for visualization

- next/previous button 🡪 One step back or forward in the visualization

2. **Settings Panel:**

- textboxes for:

* Size of matrix 🡪 height and width of the matrix
* Step 🡪the step needed to apply the rule
* Rules 🡪 place to enter custom rules (rules have to be in chosen lang. syntax)
* Live 🡪 no. of alive cells in the rule
* Dead 🡪 no. of dead cells in the rule

- apply button 🡪 Button to apply all the changed rules and settings

- randomize button 🡪 Enters random number and data into rules

5. Supplementary specification

Non-functional features are the features that does not have anything to do with the main program algorithm. They are useful for user and sometimes necessary for the program to work correctly.

Additional features in my program:

- **Stop button** 🡪 permanently stops current visualization (resets the board)

- **Pause button** 🡪 pauses current visualization

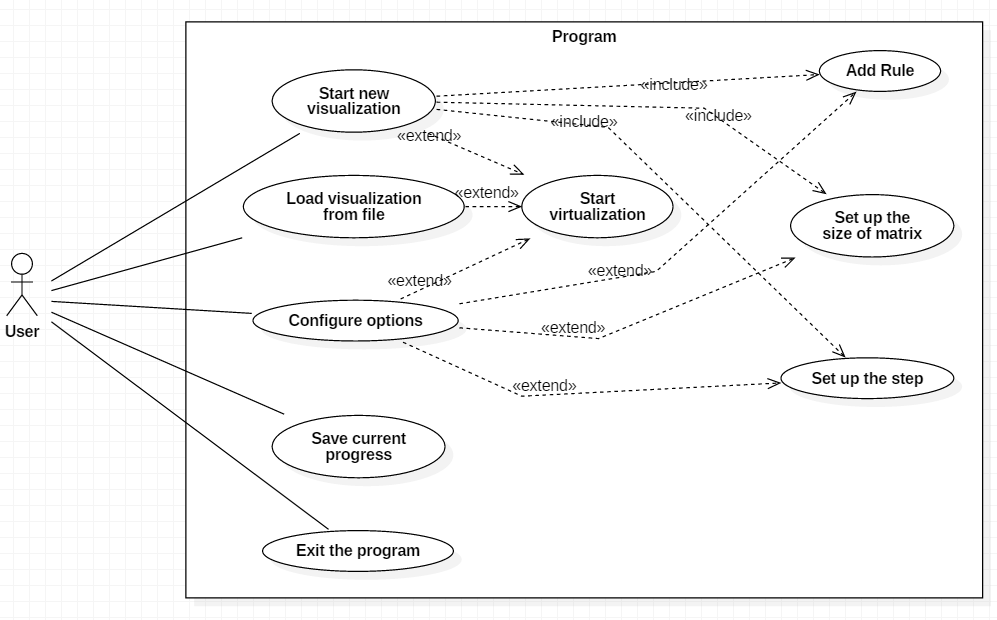
- **Play button** 🡪 start or resumes visualization

- **About menu button** 🡪 the about section will have the information about the current version of the program as well as the manual for the application itself

- **“Make a step” button** 🡪 (while in pause/stop mode) makes one step of the grid

- **“Reverse a step” button** 🡪 (while in pause/stop mode) go back with the previous step

- **Randomize button** 🡪 fills rule making section with randomize data and rules



6. UseCase Diagram

Above Use Case diagram shows us the way the user can communicate with the program. We assume that the program is already launched. The only things user can do right after the start are: create, load or edit visualization and also save current progress or exit the application. Depending on what user chose the options he can then start the visualization or customize the settings.

7. Solution evaluation

The project will take in total approximately 120 hours of work. 20 hours for specification, another 20 hours for technical specification. Project code implementation will take about 80 hours including the testing part. No delays are taken into the account.

8. Risk Analysis

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Risk | Probablity (1-5) | Severity (1-5) | Score (PxS) | Action to Prevent / Manage Risk |
| Human |  |  |  |  |
| Illness | 3 | 2 | 6 | Systematical workflow |
| Organisational |  |  |  |  |
| Failure to meet schedule deadline | 3 | 3 | 9 | Systematical workflow, good time management |
| Technical |  |  |  |  |
| Software output is unusable | 1 | 4 | 4 | Develop a wider expertise to the problem |
| Fatal crash | 2 | 4 | 8 | Testing |
| Legal |  |  |  |  |
| Intelectual property right issues | 1 | 2 | 2 | Consult used resources |

**Technical Specification**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| Version | **Date** | **Who** | **Description** |
| 0.3 | 2016-04-06 | Aleksander Lipka | Algorithms, Applied Technology, Data Structures used |
| 0.6 | 2016-04-07 | Aleksander Lipka | Added Class Diagram |
| 0.9 | 2016-04-07 | Aleksander Lipka | Added Production model, State Diagrams, Module Description; Final check |
| 1.0 | 2016-04-14 | Aleksander Lipka | Corrected mistakes with algorithm and modules |
| 1.1 | 2016-04-20 | Aleksander Lipka | Algorithms have been described and corrected |
| 1.2 | 2016-04-21 | Aleksander Lipka | Algorithms description modification |

1. Production model

For the purpose of this project waterfall production model was chosen. The waterfall model is a sequential design process, used in software development processes, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of conception, initiation, analysis, design, construction, testing, production/implementation and maintenance.

**Advantages of waterfall model:**

- This model is simple and easy to understand and use.

- It is easy to manage – each phase has specific deliverables and a review process.

- In this model phases are processed and completed one at a time. Phases do not overlap.

- Waterfall model works well for smaller projects where requirements are very well understood

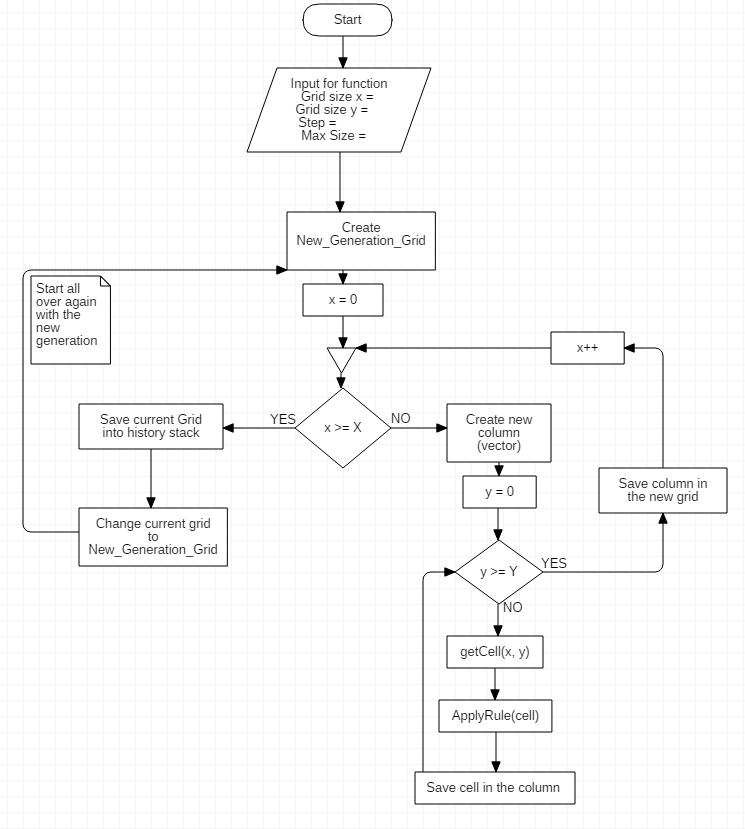
As this is a course project with strict work segmented deadlines, this model fits best the development process.

2. Algorithms

In my project I will be using three main algorithms: one for adding rules to the ruleList, one for setting state to the current cell and one that will create and traverse through the grid to enumerate the cells and create next generation grid.

First algorithm, responsible for making new grid and then traversing through it (in flowchart):

The algorithm will go through the grid and call ApplyRule() function on every cell. The results will be stored in a new grid and then this new grid will become next generation grid. Current grid will be moved to the history stack that will hold up to three generations back. User will be able to go up to 3 generations back using PreviousStep button (when in Pause mode).



2. The rule adder algorithm:

When the rule is added to the set, it goes through the phase of:

- checking whether there are the same rules already in the ruleList,

- if there are any \*contradictory rules (rules with the same input but different output),

- if the rule is contradictory with itself.

There are two types of rule: defined from a grid(exact) and made by equations. Hence, we have two types of checking: grid and equation check.

- **for exact rules:**

1. Check if there is the same rule in the rule List (if current rule has bigger priority than the one in the set -> \*switch those rules). The rules are the same if they have samely defined grid.

2. Check if there is contradiction with any other rule already in the ruleList

- **for equation rules:**

1. Check if there is the same rule in the rule List (if current rule has bigger priority than the one in the set -> switch those rules). The rules are the same if they have samely defined equations.

2. (In case of \*double equation rule) Check if the rule condtradicts with itself.

3. Check if there is contradiction with any other rule already in the ruleList

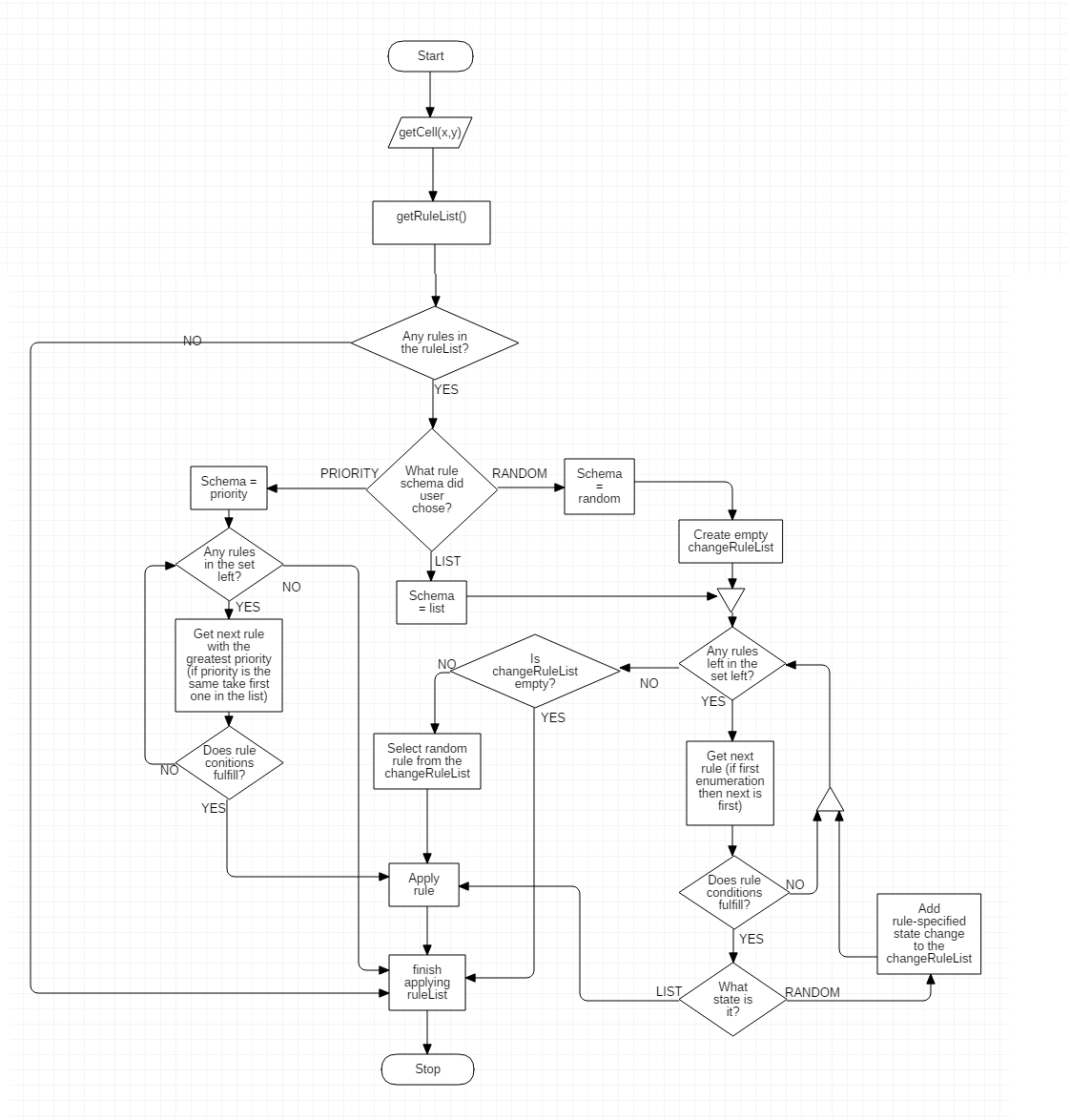
3. Apply rules to cell algorithm:

The algorithm gets a cell from the first algorithm and decide about its final state. Firstly it checks whether there are any rules in the ruleList and what rule applying \*schema the user selected. Then it enumerates the rules according to given schema and finds the final state for the current cell. In case of:

- priority schema – enumerates the ruleList from the biggest to the lowest priority and applies the first rule that holds its conditions.

- random schema - finds out how many rules would apply to the cell in the current generation of grid. It saves possible state changes in the changeRuleList. Then it randomly selects a state from the changeRuleList.

- list schema - works similarily to priority schema except that it enumerates according to when the rule was added. It traverse the ruleList basing on rule id.



3. Data Structures used

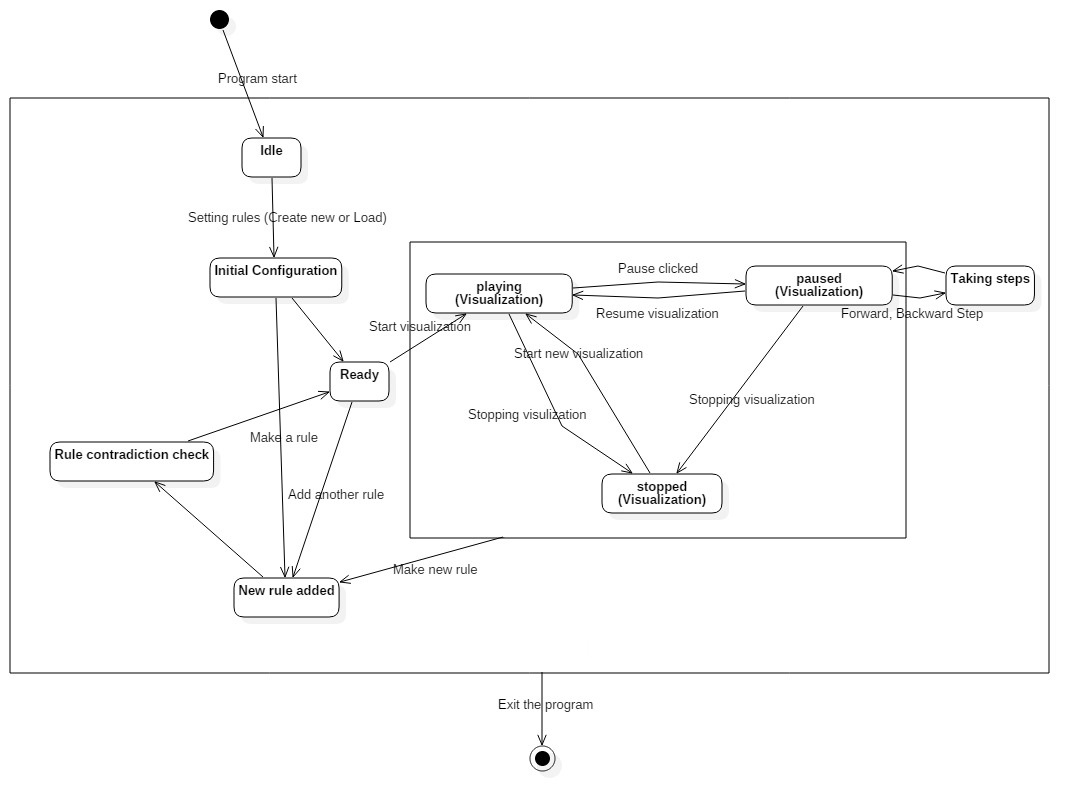
* **Lists** – all my custom data structures will be stored in lists
* **Custom data structures**:

The following class structures will be defined:

* Rule(int id, int nCells, State neighborState, State finalState)
* Cell (int state, int x, int y, color Color)
* Grid(int width, int height, List<Cell> cellList)
* **Enum** – for storing the possible states of the cells

4. State Diagram

Below is the state diagram presenting the possible states of our program:



The application starts in the idle state and then you can either create new rules for the program or load existing one. Even if you load, you can still add additional rules. From the ready state you can start the visualization or add a new rule. From Play-Pause-Stop state loop you can add a new rule and start the visualization all over again.

The technologies I will use in the project are:

5. Applied Technology



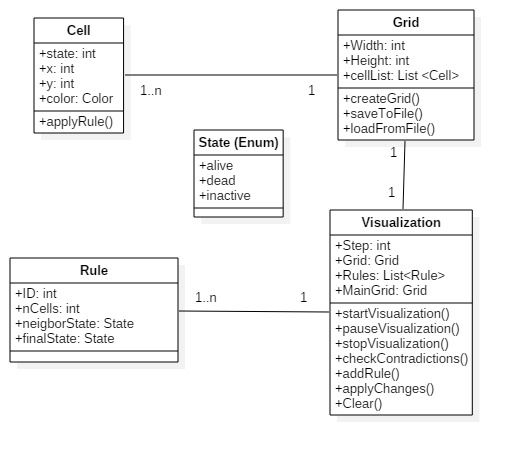
* - .NET C#



* -Windows Forms

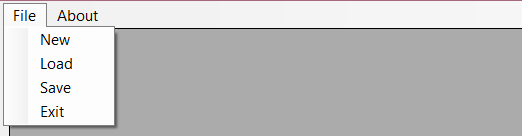
6. Class diagram

Description of the cellular automaton class diagram:

* Rule – represents a single rule
* Cell – represents a single cell in Grid
* Grid – represents a whole system of all cells
* Visualization – logical class of the program, it is the main class of the project
* State – a simple enum type with predefined possible states of the cell

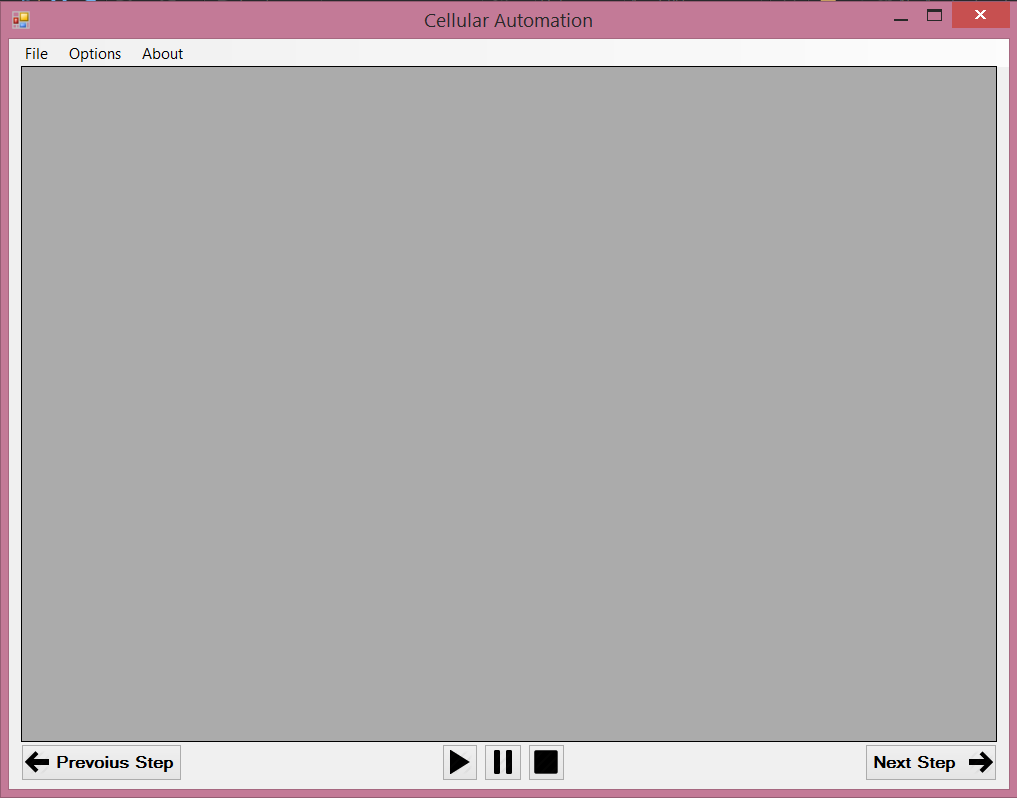
7. Description of modules

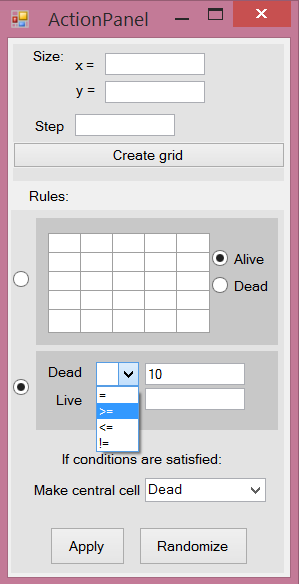
**RuleContradictionChecker module** – the module will check every newly added rule and check if it contradicts with any other rule that is already in the ruleList.

**Menu module** – options module – save, load, exit and implementing existing example rules.



**Grid module** – the main grid module, all visualization will take place in this place. This is the part of GUI that is animated and all the action takes place in here.





**RuleAdder module** – module in which the rules will be described and added to the ruleList. The module will consist of the 5x5 matrix in which you will specify which cells are taken into the count after the step taken, also all other attributes like number of steps to take and number of alive and dead cells.

7. Dictionary

- **Contradiction (rules)** – rules that can never be applied together because of impossible input conditions, like: Alive > 23 and Dead > 23 at the same time.

Possible contradictions:

- Alive+Dead>24,

- = and !=, ex. Alive = 14, Dead != 10

- >= and <= (or <= and >=), ex. Alive >= 14, Dead <= 9.

Contradictions apply both to equation rules and to grid rules.

- **Switch** – the action that deletes the old rule from ruleList and adds the new rule instead.

- **Double equation rules** – equation rule that has specified both Alive and Dead fields at the same time (look in the module addRule). Both conditions has to be satisfied in order for the rule to be applied.

- **Schema** – the main mode of the visualization. Can be either: priority, list or random.