Document metric

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Project | Cellular automaton | |  | Company: | MiNI PW |  |
|  |  |  |  |  |  |  |
| Name: | Technical Specification | |  |  |  |  |
|  |  | |  |  |  |  |
| Topics: | Technical Analisis | |  |  |  |  |
|  |  | | |  |  |  |
| Author: | Aleksander Lipka | | |  |  |  |
|  |  | |  |  |  |  |
| File: | Technical Requirements -Cellular automaton.pdf | |  |  |  |  |
|  |  |  |  |  |  |  |
| Version no: | *v 1.2* | Status: | Technical | Final date: | 2016-06-16 |  |
| specification |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Summary: | Specifiy the requirements demanded by the course | | | | |  |
|  |  |  |  | Last |  |  |
| Authorized by |  |  |  | modification | 2016-04-21 |  |
|  |  |  |  | date: |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| 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

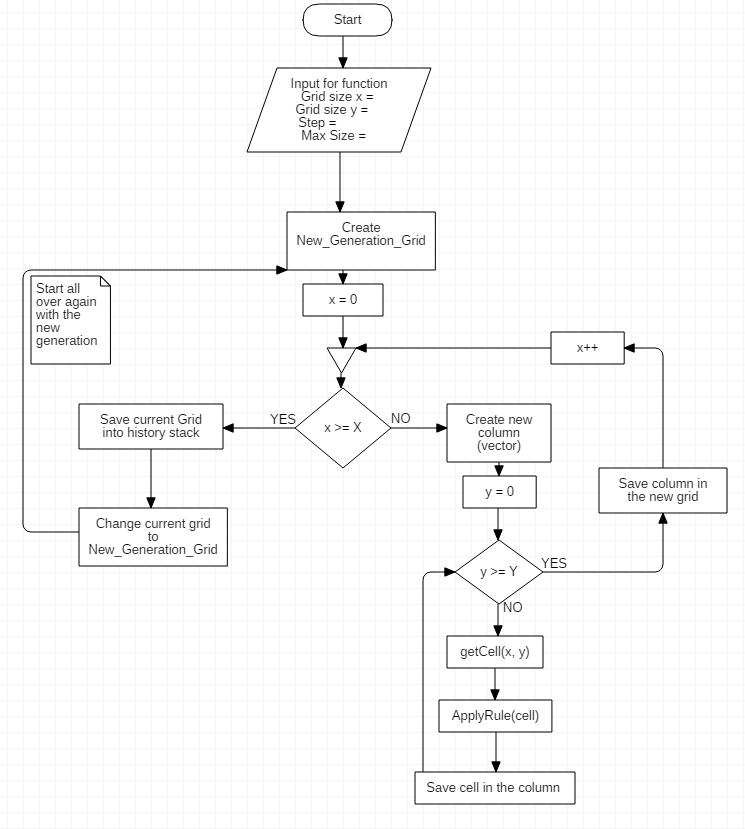
As this is a course project with strict work segmented deadlines, this model fits best the development process.

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

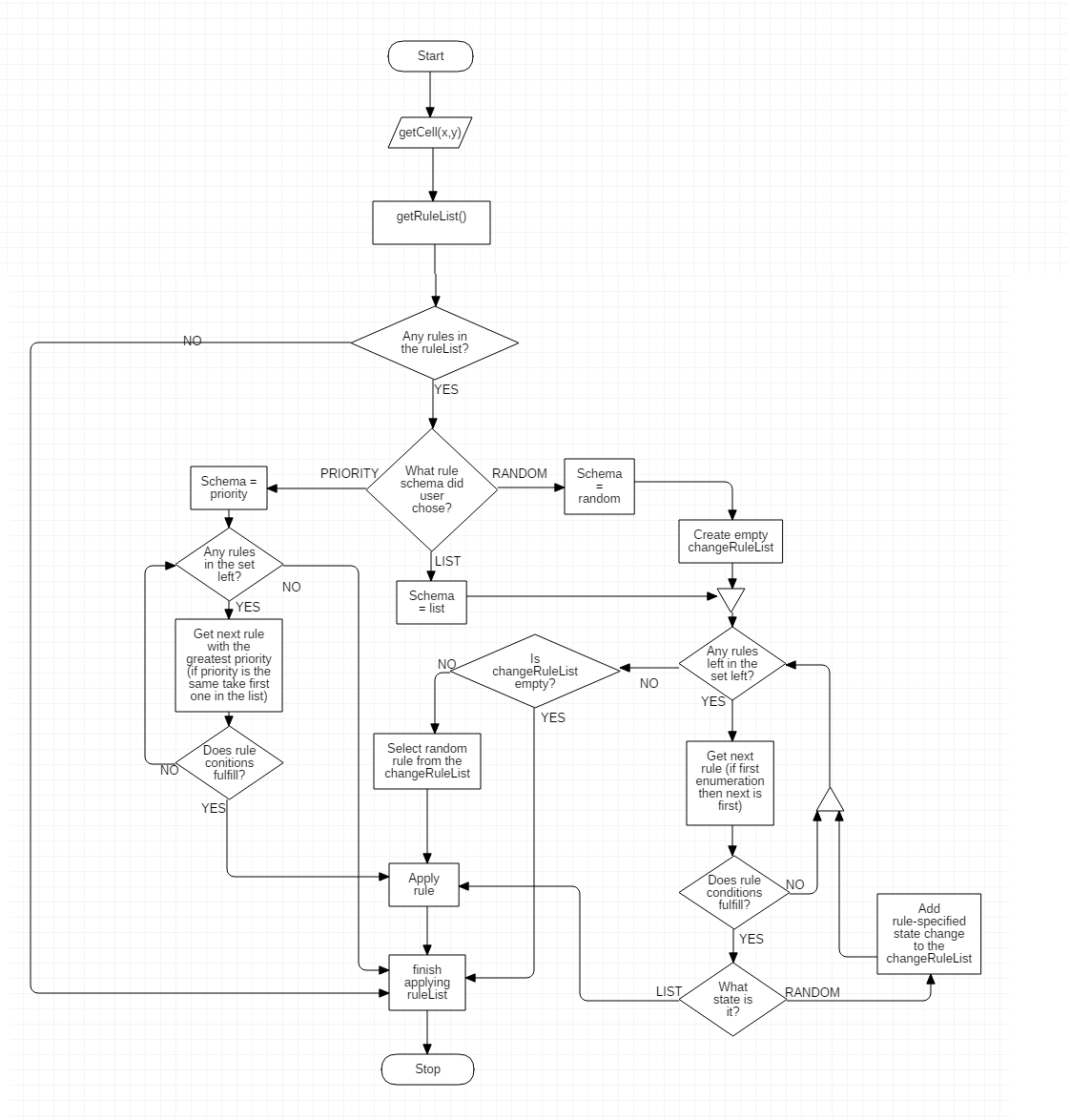
3. Apply rules to cell algorithm:

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

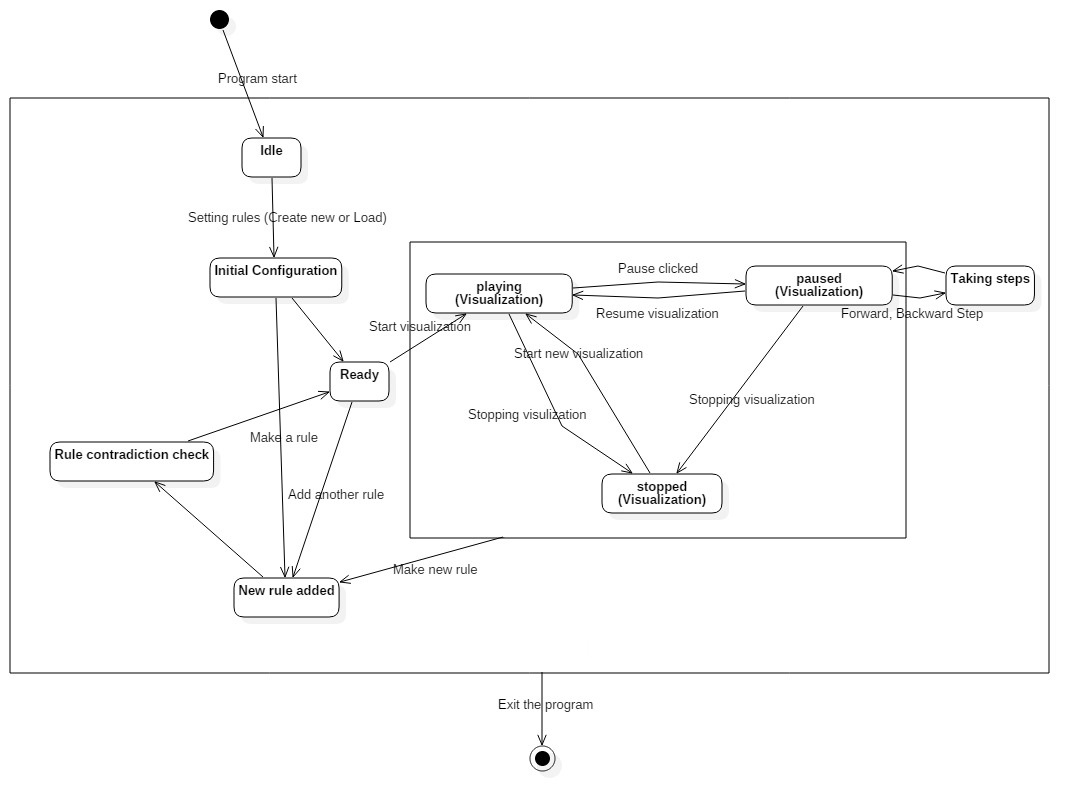
* **Lists** – all my custom data structures will be stored in lists
* **Custom data structures**:

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



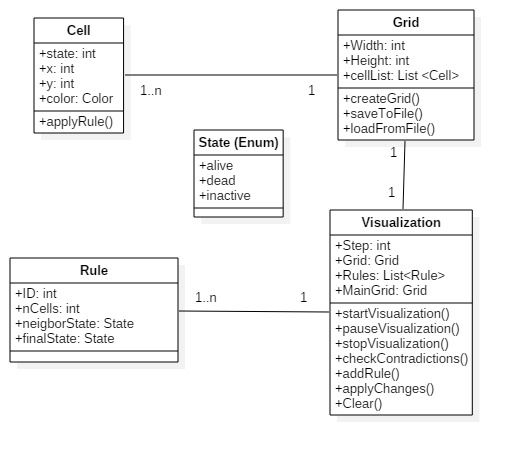
* - .NET C#



* -Windows Forms

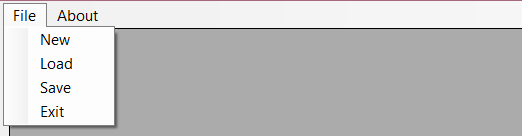
6. Class diagram

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

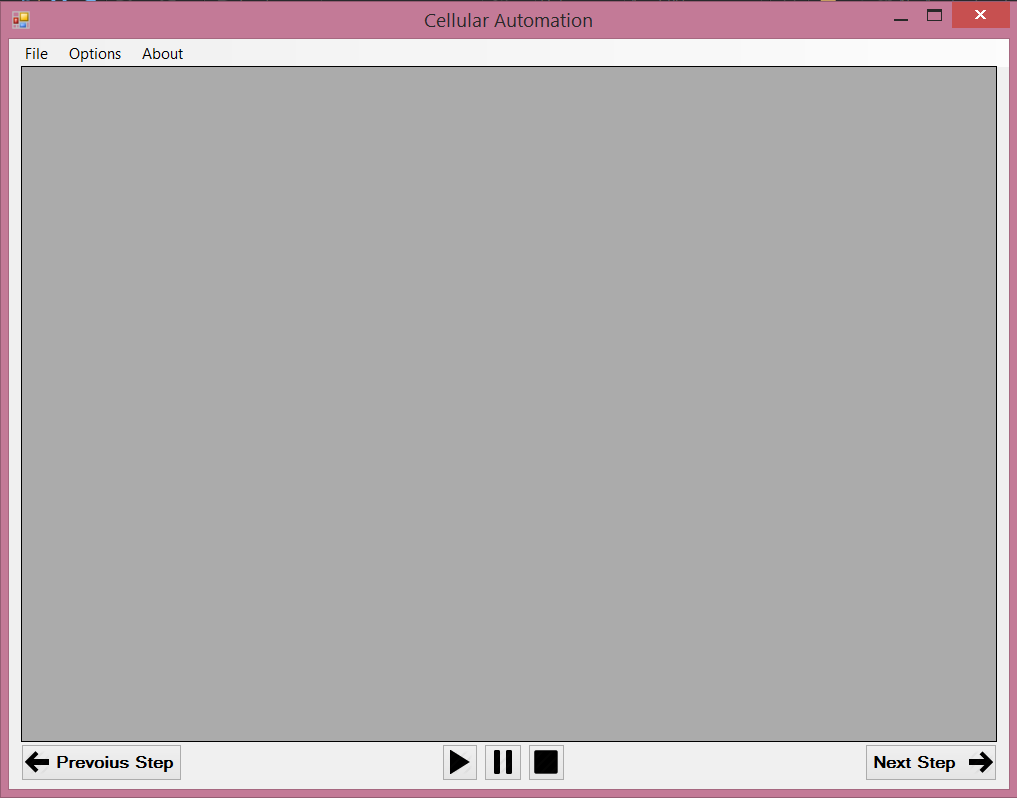
7. Description of modules

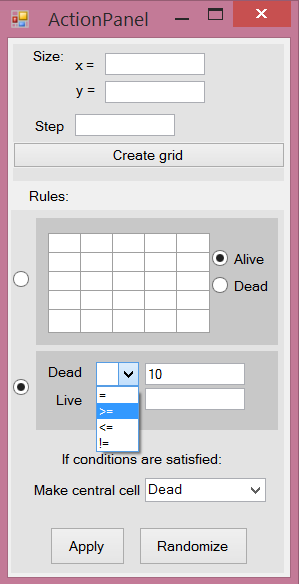
**RuleContradictionChecker module** – the module will check every newly added rule and check if it contradicts with any other rule that is already in the ruleList.

**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.