AutoCell

Generated by Doxygen 1.8.14

Contents

Chapter 1

Main Page

To generate the Documentation, go in Documentation directory and run `make`.

It will generate html doc (in `output/html/index.html`) and latex doc (pdf output directely in Documentation directory (`docPdf.pdf`).

Chapter 2

Presentation

What is AutoCell

The purpose of this project is to create a Cellular Automate Simulator.

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 5

File Index

5.1 File List

Here is a list of all files with brief descriptions:

Chapter 6

Class Documentation

6.1 Automate Class Reference

`#include <automate.h>`

Public Member Functions

- Automate (QString filename)

  Create an automate with only a cellHandler from file.
- Automate (const QVector$<$ unsigned int $>$ dimensions, CellHandler::generationTypes type=CellHandler::empty, unsigned int stateMax=1, unsigned int density=20)

  Create an automate with only a cellHandler with parameters.
- Automate (QString cellHandlerFilename, QString ruleFilename)

  Create an automate from files.
- virtual $\sim$Automate ()

  Destructor : free the CellHandler and the rules !
- bool saveRules (QString filename) const

  Save automate's rules in the file.
- bool saveCells (QString filename) const

  Save cellHandler.
- bool saveAll (QString cellHandlerFilename, QString rulesFilename) const

  Save both rules and cellHandler in the differents files.
- void addRuleFile (QString filename)
- void addRule (const Rule $*$newRule)

  Add a new rule to the Automate. Careful, the rule will be destroyed with the Automate.
- void setRulePriority (const Rule $*$rule, unsigned int newPlace)

  Modify the place of the rule in the priority list.
- const QList$<$ const Rule $*$ $>$ & getRules () const

  Return all the rules.
- bool run (unsigned int nbSteps=1)

  Apply the rule on the cells grid nbSteps times.
- const CellHandler & getCellHandler () const

  Accessor of m_cellHandler.

Private Member Functions

- bool loadRules (const QJsonArray &json)
  Load the rules of the json given.

Private Attributes

- CellHandler * m_cellHandler = nullptr
  CellHandler to go through.
- QList< const Rule * > m_rules
  Rules to use on the cells.

Friends

- class AutomateHandler

6.1.1 Detailed Description

Definition at line 15 of file automate.h.

6.1.2 Constructor & Destructor Documentation

6.1.2.1 Automate() [1/3]

```
Automate::Automate (
            QString cellHandlerFilename )
```

Create an automate with only a cellHandler from file.

Parameters

cellHandlerFilename	File to load

Definition at line 120 of file automate.cpp.

References m_cellHandler.

6.1.2.2 Automate() [2/3]

```
Automate::Automate (
            const QVector< unsigned int > dimensions,
```

```
CellHandler::generationTypes type = CellHandler::empty,
unsigned int stateMax = 1,
unsigned int density = 20 )
```

Create an automate with only a cellHandler with parameters.

Parameters

dimensions	Dimensions of the CellHandler
type	Generation type, empty by default
stateMax	Generate states between 0 and stateMax
density	Average (%) of non-zeros

Definition at line 133 of file automate.cpp.

References m_cellHandler.

6.1.2.3 Automate() [3/3]

```
Automate::Automate (
            QString cellHandlerFilename,
            QString ruleFilename )
```

Create an automate from files.

Parameters

cellHandlerFilename	File of the cellHandler
ruleFilename	File of the rules

Definition at line 144 of file automate.cpp.

References loadRules(), and m_cellHandler.

6.1.2.4 ∼Automate()

```
Automate::∼Automate ( ) [virtual]
```

Destructor : free the CellHandler and the rules !

Definition at line 179 of file automate.cpp.

References m_cellHandler, and m_rules.

6.1.3 Member Function Documentation

6.1.3.1 addRule()

```
void Automate::addRule (
             const Rule * newRule )
```

Add a new rule to the Automate. Careful, the rule will be destroyed with the Automate.

Definition at line 229 of file automate.cpp.

References m_rules.

Referenced by MainWindow::addAutomatonRules().

6.1.3.2 addRuleFile()

```
void Automate::addRuleFile (
             QString filename )
```

Definition at line 286 of file automate.cpp.

References loadRules().

Referenced by MainWindow::addAutomatonRuleFile().

6.1.3.3 getCellHandler()

```
const CellHandler & Automate::getCellHandler ( ) const
```

Accessor of m_cellHandler.

Definition at line 281 of file automate.cpp.

References m_cellHandler.

Referenced by MainWindow::createTab(), and MainWindow::updateBoard().

6.1.3.4 getRules()

```
const QList< const Rule * > & Automate::getRules ( ) const
```

Return all the rules.

Definition at line 247 of file automate.cpp.

References m_rules.

6.1.3.5 loadRules()

```
bool Automate::loadRules (
             const QJsonArray & json )  [private]
```

Load the rules of the json given.

Returns

Return false if something went wrong

Parameters

json	JsonObject wich contains the rules

Definition at line 7 of file automate.cpp.

References MatrixRule::addNeighbourState(), CellHandler::getDimensions(), m_cellHandler, and m_rules.

Referenced by addRuleFile(), and Automate().

6.1.3.6 run()

```
bool Automate::run (
            unsigned int nbSteps = 1 )
```

Apply the rule on the cells grid nbSteps times.

Parameters

nbSteps	number of iterations of the automate on the cell grid

Definition at line 256 of file automate.cpp.

References CellHandler::begin(), CellHandler::end(), m_cellHandler, m_rules, and CellHandler::nextStates().

6.1.3.7 saveAll()

```
bool Automate::saveAll (
            QString cellHandlerFilename,
            QString rulesFilename ) const
```

Save both rules and cellHandler in the differents files.

Definition at line 222 of file automate.cpp.

References saveCells(), and saveRules().

6.1.3.8 saveCells()

```
bool Automate::saveCells (
            QString filename ) const
```

Save cellHandler.

Definition at line 213 of file automate.cpp.

References m_cellHandler, and CellHandler::save().

Referenced by saveAll().

6.1.3.9 saveRules()

```
bool Automate::saveRules (
            QString filename ) const
```

Save automate's rules in the file.

Returns

False if something went wrong

Definition at line 191 of file automate.cpp.

References m_rules.

Referenced by saveAll().

6.1.3.10 setRulePriority()

```
void Automate::setRulePriority (
            const Rule * rule,
            unsigned int newPlace )
```

Modify the place of the rule in the priority list.

2 rules can't have the same priority rank

Parameters

rule	Rule to move
newPlace	New place of the rule

Definition at line 240 of file automate.cpp.

References m_rules.

6.1.4 Friends And Related Function Documentation

6.1.4.1 AutomateHandler

```
friend class AutomateHandler [friend]
```

Definition at line 20 of file automate.h.

6.1.5 Member Data Documentation

6.1.5.1 m_cellHandler

`CellHandler* Automate::m_cellHandler = nullptr [private]`

CellHandler to go through.

Definition at line 18 of file automate.h.

Referenced by Automate(), getCellHandler(), loadRules(), run(), saveCells(), and ∼Automate().

6.1.5.2 m_rules

`QList<const Rule*> Automate::m_rules [private]`

Rules to use on the cells.

Definition at line 19 of file automate.h.

Referenced by addRule(), getRules(), loadRules(), run(), saveRules(), setRulePriority(), and ∼Automate().

The documentation for this class was generated from the following files:

- automate.h
- automate.cpp

6.2 AutomateHandler Class Reference

Implementation of singleton design pattern.

`#include <automatehandler.h>`

Public Member Functions

- Automate ∗ getAutomate (unsigned int indexAutomate)

  Get an automate from the list according to its index.
- unsigned int getNumberAutomates () const

  Get the number of automates contained in the automate list.
- void addAutomate (Automate ∗automate)

  Add an automate in the automate list.
- void deleteAutomate (Automate ∗automate)

  Delete an automate from the automate list.

Static Public Member Functions

- static AutomateHandler & getAutomateHandler ()

  Get the unique running automate handler instance or create one if there is no instance running.
- static void deleteAutomateHandler ()

  Delete the unique automate handler if it exists.

Private Member Functions

- AutomateHandler ()

  Construct an automate handler.
- AutomateHandler (const AutomateHandler &a)=delete
- AutomateHandler & operator= (const AutomateHandler &a)=delete
- ∼AutomateHandler ()

  Delete all the automates contained in the automate handler.

Private Attributes

- QList< Automate ∗ > m_ActiveAutomates

  list of existing automates

Static Private Attributes

- static AutomateHandler ∗ m_activeAutomateHandler = nullptr

  active automate handler if existing, nullptr else

6.2.1 Detailed Description

Implementation of singleton design pattern.

Definition at line 10 of file automatehandler.h.

6.2.2 Constructor & Destructor Documentation

6.2.2.1 AutomateHandler() [1/2]

```
AutomateHandler::AutomateHandler ( )  [private]
```

Construct an automate handler.

Definition at line 10 of file automatehandler.cpp.

Referenced by getAutomateHandler().

6.2.2.2 AutomateHandler() [2/2]

```
AutomateHandler::AutomateHandler (
            const AutomateHandler & a )  [private], [delete]
```

6.2.2.3 ~AutomateHandler()

```
AutomateHandler::~AutomateHandler ( )  [private]
```

Delete all the automates contained in the automate handler.

Definition at line 18 of file automatehandler.cpp.

References m_ActiveAutomates.

6.2.3 Member Function Documentation

6.2.3.1 addAutomate()

```
void AutomateHandler::addAutomate (
            Automate * automate )
```

Add an automate in the automate list.

Parameters

automate	to be added to the automate list

Definition at line 78 of file automatehandler.cpp.

References m_ActiveAutomates.

Referenced by MainWindow::openFile(), and MainWindow::receiveCellHandler().

6.2.3.2 deleteAutomate()

```
void AutomateHandler::deleteAutomate (
            Automate * automate )
```

Delete an automate from the automate list.

Parameters

automate	automate to delete

Definition at line 89 of file automatehandler.cpp.

References m_ActiveAutomates.

Referenced by MainWindow::closeTab().

6.2.3.3 deleteAutomateHandler()

```
void AutomateHandler::deleteAutomateHandler ( )  [static]
```

Delete the unique automate handler if it exists.

Definition at line 39 of file automatehandler.cpp.

References m_activeAutomateHandler.

6.2.3.4 getAutomate()

```
Automate * AutomateHandler::getAutomate (
            unsigned int indexAutomate )
```

Get an automate from the list according to its index.

Parameters

indexAutomate	Index of a specific automate in the automate list

Returns

Pointer on the requested automated if the parameter index fits with the list size

Definition at line 55 of file automatehandler.cpp.

References m_ActiveAutomates.

Referenced by MainWindow::addAutomatonRuleFile(), MainWindow::addAutomatonRules(), MainWindow::createTab(), MainWindow::nextState(), MainWindow::reset(), MainWindow::runAutomaton(), MainWindow::saveToFile(), and MainWindow::updateBoard().

6.2.3.5 getAutomateHandler()

```
AutomateHandler & AutomateHandler::getAutomateHandler ( )  [static]
```

Get the unique running automate handler instance or create one if there is no instance running.

Returns

the unique running automate handler instance

Definition at line 29 of file automatehandler.cpp.

References AutomateHandler(), and m_activeAutomateHandler.

Referenced by MainWindow::addAutomatonRuleFile(), MainWindow::addAutomatonRules(), MainWindow::closeTab(), MainWindow::createTab(), MainWindow::handlePlayPause(), MainWindow::nextState(), MainWindow::openFile(), MainWindow::receiveCellHandler(), MainWindow::reset(), MainWindow::runAutomaton(), MainWindow::saveToFile(), and MainWindow::updateBoard().

6.2.3.6 getNumberAutomates()

```
unsigned int AutomateHandler::getNumberAutomates ( ) const
```

Get the number of automates contained in the automate list.

Returns

number of automates in the automate list

Definition at line 67 of file automatehandler.cpp.

References m_ActiveAutomates.

6.2.3.7 operator=()

```
AutomateHandler& AutomateHandler::operator= (
            const AutomateHandler & a )  [private], [delete]
```

6.2.4 Member Data Documentation

6.2.4.1 m_activeAutomateHandler

`AutomateHandler * AutomateHandler::m_activeAutomateHandler = nullptr [static], [private]`

active automate handler if existing, nullptr else

Initialization of the static value.

Definition at line 14 of file automatehandler.h.

Referenced by deleteAutomateHandler(), and getAutomateHandler().

6.2.4.2 m_ActiveAutomates

`QList<Automate*> AutomateHandler::m_ActiveAutomates [private]`

list of existing automates

Definition at line 13 of file automatehandler.h.

Referenced by addAutomate(), deleteAutomate(), getAutomate(), getNumberAutomates(), and ~AutomateHandler().

The documentation for this class was generated from the following files:

- automatehandler.h
- automatehandler.cpp

6.3 Cell Class Reference

Contains the state, the next state and the neighbours.

`#include <cell.h>`

Public Member Functions

- Cell (unsigned int state=0)

    Constructs a cell with the state given. State 0 is dead state.
- void setState (unsigned int state)

    Set temporary state.
- void validState ()

    Validate temporary state.
- void forceState (unsigned int state)

    Force the state change.
- unsigned int getState () const

    Access current cell state.
- bool back ()

    Set the previous state.
- void reset ()

    Reset the cell to the 1st state.
- bool addNeighbour (const Cell *neighbour, const QVector< short > relativePosition)

    Add a new neighbour to the Cell.
- QMap< QVector< short >, const Cell * > getNeighbours () const

    Access neighbours list.
- const Cell * getNeighbour (QVector< short > relativePosition) const

    Get the neighbour asked. If not existent, return nullptr.
- unsigned int countNeighbours (unsigned int filterState) const

    Return the number of neighbour which have the given state.
- unsigned int countNeighbours () const

    Return the number of neighbour which are not dead (=0)

Static Public Member Functions

- static QVector< short > getRelativePosition (const QVector< unsigned int > cellPosition, const QVector< unsigned int > neighbourPosition)

  Get the relative position, as neighbourPosition minus cellPosition.

Private Attributes

- QStack< unsigned int > m_states

  Current state.
- unsigned int m_nextState

  Temporary state, before validation.
- QMap< QVector< short >, const Cell * > m_neighbours

  Cell's neighbours. Key is the relative position of the neighbour.

6.3.1 Detailed Description

Contains the state, the next state and the neighbours.

Definition at line 11 of file cell.h.

6.3.2 Constructor & Destructor Documentation

6.3.2.1 Cell()

```
Cell::Cell (
            unsigned int state = 0 )
```

Constructs a cell with the state given. State 0 is dead state.

Parameters

state	Cell state, dead state by default

Definition at line 7 of file cell.cpp.

References m_states.

6.3.3 Member Function Documentation

6.3.3.1 addNeighbour()

```
bool Cell::addNeighbour (
            const Cell * neighbour,
            const QVector< short > relativePosition )
```

Add a new neighbour to the Cell.

Parameters

relativePosition	Relative position of the new neighbour
neighbour	New neighbour

Returns

False if the neighbour already exists

Definition at line 83 of file cell.cpp.

References m_neighbours.

6.3.3.2 back()

```
bool Cell::back ( )
```

Set the previous state.

Returns

Return false if we are already at the first state

Definition at line 58 of file cell.cpp.

References m_nextState, and m_states.

6.3.3.3 countNeighbours() [1/2]

```
unsigned int Cell::countNeighbours (
            unsigned int filterState ) const
```

Return the number of neighbour which have the given state.

Definition at line 110 of file cell.cpp.

References m_neighbours.

Referenced by NeighbourRule::matchCell().

6.3.3.4 countNeighbours() [2/2]

```
unsigned int Cell::countNeighbours ( ) const
```

Return the number of neighbour which are not dead (=0)

Definition at line 123 of file cell.cpp.

References m_neighbours.

6.3.3.5 forceState()

```
void Cell::forceState (
            unsigned int state )
```

Force the state change.

Is equivalent to setState followed by validState

Parameters

state	New state

Definition at line 41 of file cell.cpp.

References m_nextState, and m_states.

6.3.3.6 getNeighbour()

```
const Cell * Cell::getNeighbour (
            QVector< short > relativePosition ) const
```

Get the neighbour asked. If not existent, return nullptr.

Definition at line 103 of file cell.cpp.

References m_neighbours.

Referenced by MatrixRule::matchCell().

6.3.3.7 getNeighbours()

```
QMap< QVector< short >, const Cell * > Cell::getNeighbours ( ) const
```

Access neighbours list.

The map key is the relative position of the neighbour (like -1,0 for the cell just above)

Definition at line 96 of file cell.cpp.

References m_neighbours.

6.3.3.8 getRelativePosition()

```
QVector< short > Cell::getRelativePosition (
            const QVector< unsigned int > cellPosition,
            const QVector< unsigned int > neighbourPosition )  [static]
```

Get the relative position, as neighbourPosition minus cellPosition.

Exceptions

QString	Different size of position vectors

Parameters

cellPosition	Cell Position
neighbourPosition	Neighbour absolute position

Definition at line 140 of file cell.cpp.

Referenced by CellHandler::foundNeighbours().

6.3.3.9 getState()

```
unsigned int Cell::getState ( ) const
```

Access current cell state.

Definition at line 49 of file cell.cpp.

References m_states.

Referenced by MatrixRule::matchCell(), and NeighbourRule::matchCell().

6.3.3.10 reset()

```
void Cell::reset ( )
```

Reset the cell to the 1st state.

Definition at line 69 of file cell.cpp.

References m_nextState, and m_states.

6.3.3.11 setState()

```
void Cell::setState (
            unsigned int state )
```

Set temporary state.

To change current cell state, use setState(unsigned int state) then validState(). (

Parameters

state	New state

Definition at line 20 of file cell.cpp.

References m_nextState.

6.3.3.12 validState()

```
void Cell::validState ( )
```

Validate temporary state.

To change current cell state, use setState(unsigned int state) then validState().

Definition at line 30 of file cell.cpp.

References m_nextState, and m_states.

6.3.4 Member Data Documentation

6.3.4.1 m_neighbours

```
QMap<QVector<short>, const Cell*> Cell::m_neighbours [private]
```

Cell's neighbours. Key is the relative position of the neighbour.

Definition at line 37 of file cell.h.

Referenced by addNeighbour(), countNeighbours(), getNeighbour(), and getNeighbours().

6.3.4.2 m_nextState

```
unsigned int Cell::m_nextState [private]
```

Temporary state, before validation.

Definition at line 35 of file cell.h.

Referenced by back(), forceState(), reset(), setState(), and validState().

6.3.4.3 m_states

`QStack<unsigned int> Cell::m_states [private]`

Current state.

Definition at line 34 of file cell.h.

Referenced by back(), Cell(), forceState(), getState(), reset(), and validState().

The documentation for this class was generated from the following files:

- cell.h
- cell.cpp

6.4 CellHandler Class Reference

Cell container and cell generator.

`#include <cellhandler.h>`

Classes

- class iteratorT

  Implementation of iterator design pattern with a template to generate iterator and const_iterator at the same time.

Public Types

- enum generationTypes { empty, random, symetric }

  Type of random generation.
- typedef iteratorT< const CellHandler, const Cell > const_iterator
- typedef iteratorT< CellHandler, Cell > iterator

Public Member Functions

- CellHandler (const QString filename)

  Construct all the cells from the json file given.
- CellHandler (const QJsonObject &json)

  Construct all the cells from the json object given.
- CellHandler (const QVector< unsigned int > dimensions, generationTypes type=empty, unsigned int state↩
Max=1, unsigned int density=20)

  Construct a CellHandler of the given dimension.
- virtual ∼CellHandler ()

  Destroys all cells in the CellHandler.
- Cell ∗ getCell (const QVector< unsigned int > position) const

  Access the cell to the given position.
- unsigned int getMaxState () const

  Return the max state of the CellHandler.

- QVector< unsigned int > getDimensions () const
  Accessor of m_dimensions.
- void nextStates () const
  Valid the state of all cells.
- bool previousStates () const
  Get all the cells to their previous states.
- void reset () const
  Reset all the cells to the 1st state.
- bool save (QString filename) const
  Save the CellHandler current configuration in the file given.
- void generate (generationTypes type, unsigned int stateMax=1, unsigned short density=50)
  Replace Cell values by random values (symetric or not)
- void print (std::ostream &stream) const
  Print in the given stream the CellHandler.
- const_iterator begin () const
  Give the iterator which corresponds to the current CellHandler.
- iterator begin ()
  Give the iterator which corresponds to the current CellHandler.
- bool end () const
  End condition of the iterator.

Private Member Functions

- bool load (const QJsonObject &json)
  Load the config file in the CellHandler.
- void foundNeighbours ()
  Set the neighbours of each cells.
- void positionIncrement (QVector< unsigned int > &pos, unsigned int value=1) const
  Increment the QVector given by the value choosen.
- QVector< QVector< unsigned int > > * getListNeighboursPositionsRecursive (const QVector< unsigned int > position, unsigned int dimension, QVector< unsigned int > lastAdd) const
  Recursive function which browse the position possibilities tree.
- QVector< QVector< unsigned int > > & getListNeighboursPositions (const QVector< unsigned int > position) const
  Prepare the call of the recursive version of itself.

Private Attributes

- QVector< unsigned int > m_dimensions
  Vector of x dimensions.
- QMap< QVector< unsigned int >, Cell *> m_cells
  Map of cells, with a x dimensions vector as key.
- unsigned int m_maxState

6.4.1 Detailed Description

Cell container and cell generator.

Generate cells from a json file.

Definition at line 20 of file cellhandler.h.

6.4.2 Member Typedef Documentation

6.4.2.1 const_iterator

```
typedef iteratorT<const CellHandler, const Cell> CellHandler::const_iterator
```

Definition at line 94 of file cellhandler.h.

6.4.2.2 iterator

```
typedef iteratorT<CellHandler, Cell> CellHandler::iterator
```

Definition at line 95 of file cellhandler.h.

6.4.3 Member Enumeration Documentation

6.4.3.1 generationTypes

```
enum CellHandler::generationTypes
```

Type of random generation.

Enumerator

empty	Only empty cells.
random	Random cells.
symetric	Random cells but with vertical symetry (on the 1st dimension component)

Definition at line 99 of file cellhandler.h.

6.4.4 Constructor & Destructor Documentation

6.4.4.1 CellHandler() [1/3]

```
CellHandler::CellHandler (
            const QString filename )
```

Construct all the cells from the json file given.

The size of "cells" array must be the product of all dimensions (60 in the following example). Typical Json file:

```
{
"dimensions":"3x4x5",
"cells":[0,1,4,4,2,5,3,4,2,4,
         4,2,5,0,0,0,0,0,0,0,
         2,4,1,1,1,1,1,2,1,1,
         0,0,0,0,0,0,2,2,2,2,
         3,4,5,1,2,0,9,0,0,0,
         1,2,0,0,0,0,1,2,3,2]
}
```

Parameters

filename	Json file which contains the description of all the cells

Exceptions

QString	Unreadable file
QString	Empty file
QString	Not valid file

Definition at line 25 of file cellhandler.cpp.

References foundNeighbours(), and load().

6.4.4.2 CellHandler() [2/3]

```
CellHandler::CellHandler (
              const QJsonObject & json )
```

Construct all the cells from the json object given.

The size of "cells" array must be the product of all dimensions (60 in the following example). Typical Json object:

```
{
"dimensions":"3x4x5",
"cells":[0,1,4,4,2,5,3,4,2,4,
         4,2,5,0,0,0,0,0,0,0,
         2,4,1,1,1,1,1,2,1,1,
         0,0,0,0,0,0,2,2,2,2,
         3,4,5,1,2,0,9,0,0,0,
         1,2,0,0,0,0,1,2,3,2]
}
```

Parameters

json	Json object which contains the description of all the cells

Exceptions

QString	Not valid file

Definition at line 76 of file cellhandler.cpp.

References foundNeighbours(), and load().

6.4.4.3 CellHandler() [3/3]

```
CellHandler::CellHandler (
            const QVector< unsigned int > dimensions,
            generationTypes type = empty,
            unsigned int stateMax = 1,
            unsigned int density = 20 )
```

Construct a CellHandler of the given dimension.

If generationTypes is given, the CellHandler won't be empty.

Parameters

dimensions	Dimensions of the CellHandler
type	Generation type, empty by default
stateMax	Generate states between 0 and stateMax
density	Average (%) of non-zeros

Definition at line 98 of file cellhandler.cpp.

References empty, foundNeighbours(), generate(), m_cells, m_dimensions, and positionIncrement().

6.4.4.4 ∼CellHandler()

```
CellHandler::∼CellHandler ( ) [virtual]
```

Destroys all cells in the CellHandler.

Definition at line 131 of file cellhandler.cpp.

References m_cells.

6.4.5 Member Function Documentation

6.4.5.1 begin() [1/2]

```
CellHandler::const_iterator CellHandler::begin ( ) const
```

Give the iterator which corresponds to the current CellHandler.

Definition at line 327 of file cellhandler.cpp.

Referenced by print(), Automate::run(), save(), and MainWindow::updateBoard().

6.4.5.2 begin() [2/2]

```
CellHandler::iterator CellHandler::begin ( )
```

Give the iterator which corresponds to the current CellHandler.

Definition at line 320 of file cellhandler.cpp.

6.4.5.3 end()

```
bool CellHandler::end ( ) const
```

End condition of the iterator.

See iterator::operator!=(bool finished) for further information.

Definition at line 336 of file cellhandler.cpp.

Referenced by print(), Automate::run(), save(), and MainWindow::updateBoard().

6.4.5.4 foundNeighbours()

```
void CellHandler::foundNeighbours ( ) [private]
```

Set the neighbours of each cells.

Careful, this is in O(n∗3^d), with n the number of cells and d the number of dimensions

Definition at line 434 of file cellhandler.cpp.

References getListNeighboursPositions(), Cell::getRelativePosition(), m_cells, m_dimensions, and positionIncrement().

Referenced by CellHandler().

6.4.5.5 generate()

```
void CellHandler::generate (
            CellHandler::generationTypes type,
            unsigned int stateMax = 1,
            unsigned short density = 50 )
```

Replace Cell values by random values (symetric or not)

Parameters

type	Type of random generation
stateMax	Generate states between 0 and stateMax
density	Average (%) of non-zeros

Definition at line 241 of file cellhandler.cpp.

References m_cells, m_dimensions, positionIncrement(), random, and symetric.

Referenced by CellHandler().

6.4.5.6 getCell()

```
Cell * CellHandler::getCell (
            const QVector< unsigned int > position ) const
```

Access the cell to the given position.

Definition at line 141 of file cellhandler.cpp.

References m_cells.

6.4.5.7 getDimensions()

```
QVector< unsigned int > CellHandler::getDimensions ( ) const
```

Accessor of m_dimensions.

Definition at line 155 of file cellhandler.cpp.

References m_dimensions.

Referenced by MainWindow::createTab(), Automate::loadRules(), and MainWindow::updateBoard().

6.4.5.8 getListNeighboursPositions()

```
QVector< QVector< unsigned int > > & CellHandler::getListNeighboursPositions (
            const QVector< unsigned int > position ) const [private]
```

Prepare the call of the recursive version of itself.

Parameters

position	Position of the central cell (x1,x2,x3,..,xn)

Returns

List of positions

Definition at line 493 of file cellhandler.cpp.

References getListNeighboursPositionsRecursive().

Referenced by foundNeighbours().

6.4.5.9 getListNeighboursPositionsRecursive()

```
QVector< QVector< unsigned int > > * CellHandler::getListNeighboursPositionsRecursive (
              const QVector< unsigned int > position,
              unsigned int dimension,
              QVector< unsigned int > lastAdd ) const  [private]
```

Recursive function which browse the position possibilities tree.

Careful, the complexity is in O(3^dimension)
Piece of the tree:

```
            x_d -1
          /
x_(d-1)-1/_ x_d
         \
          \
            x_d +1

            x_d -1
          /
x_(d-1)  /_ x_d
         \
          \
            x_d +1

            x_d -1
          /
x_(d-1)+1/ x_d
         \
          \
            x_d +1
```

The path in the tree to reach the leaf give the position

Parameters

position	Position of the cell
dimension	Current working dimension (number of the digit). Dimension = 2 <=> working on x2 coordinates on (x1, x2, x3, ..., xn) vector
lastAdd	Last position added. Like the father node of the new tree

Returns

List of position

Definition at line 534 of file cellhandler.cpp.

References m_dimensions.

Referenced by getListNeighboursPositions().

6.4.5.10 getMaxState()

```
unsigned int CellHandler::getMaxState ( ) const
```

Return the max state of the CellHandler.

Definition at line 148 of file cellhandler.cpp.

References m_maxState.

6.4.5.11 load()

```
bool CellHandler::load (
               const QJsonObject & json )  [private]
```

Load the config file in the CellHandler.

Exemple of a way to print cell states :

```
QVector<unsigned int> position;
for (unsigned short i = 0; i < m_dimensions.size(); i++)
{
     position.push_back(0);
}
for (unsigned int j = 0; j < m_cells.size(); j++)
{
     std::cout << m_cells.value(position)->getState() << " ";
     position.replace(0, position.at(0)+1);
     for (unsigned short i = 0; i < m_dimensions.size(); i++)
     {
         if (position.at(i) >= m_dimensions.at(i))
         {
             position.replace(i, 0);
             std::cout << std::endl;
             if (i + 1 != m_dimensions.size())
                 position.replace(i+1, position.at(i+1)+1);
         }

     }
}
```

Parameters

json	Json Object which contains the grid configuration

Returns

False if the Json Object is not correct

Definition at line 371 of file cellhandler.cpp.

References m_cells, m_dimensions, m_maxState, and positionIncrement().

Referenced by CellHandler().

6.4.5.12 nextStates()

```
void CellHandler::nextStates ( ) const
```

Valid the state of all cells.

Definition at line 162 of file cellhandler.cpp.

References m_cells.

Referenced by Automate::run().

6.4.5.13 positionIncrement()

```
void CellHandler::positionIncrement (
            QVector< unsigned int > & pos,
            unsigned int value = 1 ) const  [private]
```

Increment the QVector given by the value choosen.

Careful, when the position reach the maximum, it goes to zero without leaving the function

Parameters

pos	Position to increment
value	Value to add, 1 by default

Definition at line 464 of file cellhandler.cpp.

References m_dimensions.

Referenced by CellHandler(), foundNeighbours(), generate(), and load().

6.4.5.14 previousStates()

```
bool CellHandler::previousStates ( ) const
```

Get all the cells to their previous states.

Definition at line 172 of file cellhandler.cpp.

References m_cells.

6.4.5.15 print()

```
void CellHandler::print (
            std::ostream & stream ) const
```

Print in the given stream the CellHandler.

Parameters

stream	Stream to print into

Definition at line 306 of file cellhandler.cpp.

References begin(), and end().

6.4.5.16 reset()

```
void CellHandler::reset ( ) const
```

Reset all the cells to the 1st state.

Definition at line 184 of file cellhandler.cpp.

References m_cells.

6.4.5.17 save()

```
bool CellHandler::save (
            QString filename ) const
```

Save the CellHandler current configuration in the file given.

Parameters

filename	Path to the file

Returns

False if there was a problem

Exceptions

QString	Impossible to open the file

Definition at line 199 of file cellhandler.cpp.

References begin(), end(), m_dimensions, and m_maxState.

Referenced by Automate::saveCells().

6.4.6 Member Data Documentation

6.4.6.1 m_cells

```
QMap<QVector<unsigned int>, Cell* > CellHandler::m_cells [private]
```

Map of cells, with a x dimensions vector as key.

Definition at line 135 of file cellhandler.h.

Referenced by CellHandler(), foundNeighbours(), generate(), getCell(), load(), nextStates(), previousStates(), reset(), and ∼CellHandler().

6.4.6.2 m_dimensions

```
QVector<unsigned int> CellHandler::m_dimensions [private]
```

Vector of x dimensions.

Definition at line 134 of file cellhandler.h.

Referenced by CellHandler(), foundNeighbours(), generate(), getDimensions(), getListNeighboursPositionsRecursive(), load(), positionIncrement(), and save().

6.4.6.3 m_maxState

`unsigned int CellHandler::m_maxState [private]`

Definition at line 136 of file cellhandler.h.

Referenced by getMaxState(), load(), and save().

The documentation for this class was generated from the following files:

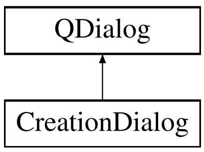
- cellhandler.h
- cellhandler.cpp

6.5 CreationDialog Class Reference

Automaton creation dialog box.

`#include <creationdialog.h>`

Inheritance diagram for CreationDialog:



Public Slots

- void processSettings ()

Signals

- void settingsFilled (const QVector< unsigned int > dimensions, CellHandler::generationTypes type=Cell↩ Handler::generationTypes::empty, unsigned int stateMax=1, unsigned int density=20)

Public Member Functions

- CreationDialog (QWidget ∗parent=0)

Private Member Functions

- QGroupBox ∗ createGenButtons ()

  Creates radio buttons to select cell generation type.

Private Attributes

- QLineEdit ∗ m_dimensionsEdit
- QSpinBox ∗ m_densityBox
- QSpinBox ∗ m_stateMaxBox
- QPushButton ∗ m_doneBt
- QGroupBox ∗ m_groupBox
- QRadioButton ∗ m_empGen
- QRadioButton ∗ m_randGen
- QRadioButton ∗ m_symGen

6.5.1 Detailed Description

Automaton creation dialog box.

Allow the user to input settings to create an automaton

Definition at line 13 of file creationdialog.h.

6.5.2 Constructor & Destructor Documentation

6.5.2.1 CreationDialog()

```
CreationDialog::CreationDialog (
            QWidget * parent = 0 )
```

Definition at line 5 of file creationdialog.cpp.

References createGenButtons(), m_densityBox, m_dimensionsEdit, m_doneBt, m_stateMaxBox, and processSettings().

6.5.3 Member Function Documentation

6.5.3.1 createGenButtons()

```
CreationDialog::createGenButtons ( )  [private]
```

Creates radio buttons to select cell generation type.

Validates user settings and sends them to MainWindow.

Definition at line 51 of file creationdialog.cpp.

References m_empGen, m_groupBox, m_randGen, and m_symGen.

Referenced by CreationDialog().

6.5.3.2 processSettings

```
void CreationDialog::processSettings ( ) [slot]
```

Definition at line 72 of file creationdialog.cpp.

References m_densityBox, m_dimensionsEdit, m_randGen, m_stateMaxBox, m_symGen, and settingsFilled().

Referenced by CreationDialog().

6.5.3.3 settingsFilled

```
void CreationDialog::settingsFilled (
            const QVector< unsigned int > dimensions,
            CellHandler::generationTypes type = CellHandler::generationTypes::empty,
            unsigned int stateMax = 1,
            unsigned int density = 20 ) [signal]
```

Referenced by processSettings().

6.5.4 Member Data Documentation

6.5.4.1 m_densityBox

```
QSpinBox* CreationDialog::m_densityBox [private]
```

Definition at line 30 of file creationdialog.h.

Referenced by CreationDialog(), and processSettings().

6.5.4.2 m_dimensionsEdit

```
QLineEdit* CreationDialog::m_dimensionsEdit [private]
```

Definition at line 29 of file creationdialog.h.

Referenced by CreationDialog(), and processSettings().

6.5.4.3 m_doneBt

`QPushButton* CreationDialog::m_doneBt [private]`

Definition at line 32 of file creationdialog.h.

Referenced by CreationDialog().

6.5.4.4 m_empGen

`QRadioButton* CreationDialog::m_empGen [private]`

Definition at line 35 of file creationdialog.h.

Referenced by createGenButtons().

6.5.4.5 m_groupBox

`QGroupBox* CreationDialog::m_groupBox [private]`

Definition at line 34 of file creationdialog.h.

Referenced by createGenButtons().

6.5.4.6 m_randGen

`QRadioButton* CreationDialog::m_randGen [private]`

Definition at line 36 of file creationdialog.h.

Referenced by createGenButtons(), and processSettings().

6.5.4.7 m_stateMaxBox

`QSpinBox* CreationDialog::m_stateMaxBox [private]`

Definition at line 31 of file creationdialog.h.

Referenced by CreationDialog(), and processSettings().

6.5.4.8 m_symGen

`QRadioButton* CreationDialog::m_symGen [private]`

Definition at line 37 of file creationdialog.h.

Referenced by createGenButtons(), and processSettings().

The documentation for this class was generated from the following files:

- creationdialog.h
- creationdialog.cpp

6.6 CellHandler::iteratorT< CellHandler_T, Cell_T > Class Template Reference

Implementation of iterator design pattern with a template to generate iterator and const_iterator at the same time.

Public Member Functions

- iteratorT (CellHandler_T *handler)

  Construct an initial iterator to browse the CellHandler.
- iteratorT & operator++ ()

  Increment the current position and handle dimension changes.
- Cell_T * operator-> () const

  Get the current cell.
- Cell_T * operator* () const

  Get the current cell.
- bool operator!= (bool finished) const
- unsigned int changedDimension () const

Private Attributes

- CellHandler_T * m_handler

  CellHandler to go through.
- QVector< unsigned int > m_position

  Current position of the iterator.
- bool m_finished = false

  If we reach the last position.
- QVector< unsigned int > m_zero

  Nul vector of the good dimension (depend of m_handler)
- unsigned int m_changedDimension

  Save the number of dimension change.

Friends

- class CellHandler

6.6.1 Detailed Description

template<typename CellHandler_T, typename Cell_T>
class CellHandler::iteratorT< CellHandler_T, Cell_T >

Implementation of iterator design pattern with a template to generate iterator and const_iterator at the same time.

Example of use:

```
CellHandler handler("file.atc");
for (CellHandler::const_iterator it = handler.begin(); it != handler.end(); ++it
      )
{
     for (unsigned int i = 0; i < it.changedDimension(); i++)
         std::cout << std::endl;
     std::cout << it->getState() << " ";
}
```

This code will print each cell states and go to a new line when there is a change of dimension. So if there are 3 dimensions, there will be a empty line between 2D groups.

Definition at line 41 of file cellhandler.h.

6.6.2 Constructor & Destructor Documentation

6.6.2.1 iteratorT()

```
template<typename CellHandler_T , typename Cell_T >
CellHandler::iteratorT< CellHandler_T, Cell_T >::iteratorT (
              CellHandler_T * handler )
```

Construct an initial iterator to browse the CellHandler.

Parameters

handler	CellHandler to browse

Definition at line 574 of file cellhandler.cpp.

References CellHandler::iteratorT< CellHandler_T, Cell_T >::m_position, and CellHandler::iteratorT< CellHandler_T, Cell_T >::m_z

6.6.3 Member Function Documentation

6.6.3.1 changedDimension()

```
template<typename CellHandler_T , typename Cell_T >
unsigned int CellHandler::iteratorT< CellHandler_T, Cell_T >::changedDimension ( ) const
[inline]
```

Definition at line 80 of file cellhandler.h.

References CellHandler::iteratorT< CellHandler_T, Cell_T >::m_changedDimension.

6.6.3.2 operator"!=()

```
template<typename CellHandler_T , typename Cell_T >
bool CellHandler::iteratorT< CellHandler_T, Cell_T >::operator!= (
              bool finished ) const  [inline]
```

Definition at line 79 of file cellhandler.h.

References CellHandler::iteratorT< CellHandler_T, Cell_T >::m_finished.

6.6.3.3 operator*()

```
template<typename CellHandler_T , typename Cell_T >
Cell_T* CellHandler::iteratorT< CellHandler_T, Cell_T >::operator* ( ) const  [inline]
```

Get the current cell.

Definition at line 75 of file cellhandler.h.

References CellHandler::iteratorT< CellHandler_T, Cell_T >::m_handler, and CellHandler::iteratorT< CellHandler_T, Cell_T >::m_p

6.6.3.4 operator++()

```
template<typename CellHandler_T , typename Cell_T >
iteratorT& CellHandler::iteratorT< CellHandler_T, Cell_T >::operator++ ( )  [inline]
```

Increment the current position and handle dimension changes.

Definition at line 47 of file cellhandler.h.

References CellHandler::iteratorT< CellHandler_T, Cell_T >::m_changedDimension, CellHandler::iteratorT< CellHandler_T, Cell_T
CellHandler::iteratorT< CellHandler_T, Cell_T >::m_handler, CellHandler::iteratorT< CellHandler_T, Cell_T >::m_position,
and CellHandler::iteratorT< CellHandler_T, Cell_T >::m_zero.

6.6.3.5 operator->()

```
template<typename CellHandler_T , typename Cell_T >
Cell_T* CellHandler::iteratorT< CellHandler_T, Cell_T >::operator-> ( ) const [inline]
```

Get the current cell.

Definition at line 71 of file cellhandler.h.

References CellHandler::iteratorT< CellHandler_T, Cell_T >::m_handler, and CellHandler::iteratorT< CellHandler_T, Cell_T >::m_p

6.6.4 Friends And Related Function Documentation

6.6.4.1 CellHandler

```
template<typename CellHandler_T , typename Cell_T >
friend class CellHandler [friend]
```

Definition at line 43 of file cellhandler.h.

6.6.5 Member Data Documentation

6.6.5.1 m_changedDimension

```
template<typename CellHandler_T , typename Cell_T >
unsigned int CellHandler::iteratorT< CellHandler_T, Cell_T >::m_changedDimension [private]
```

Save the number of dimension change.

Definition at line 91 of file cellhandler.h.

Referenced by CellHandler::iteratorT< CellHandler_T, Cell_T >::changedDimension(), and CellHandler::iteratorT< CellHandler_T, C

6.6.5.2 m_finished

```
template<typename CellHandler_T , typename Cell_T >
bool CellHandler::iteratorT< CellHandler_T, Cell_T >::m_finished = false [private]
```

If we reach the last position.

Definition at line 89 of file cellhandler.h.

Referenced by CellHandler::iteratorT< CellHandler_T, Cell_T >::operator!=(), and CellHandler::iteratorT< CellHandler_T, Cell_T >::

6.6.5.3 m_handler

```
template<typename CellHandler_T , typename Cell_T >
CellHandler_T* CellHandler::iteratorT< CellHandler_T, Cell_T >::m_handler [private]
```

CellHandler to go through.

Definition at line 87 of file cellhandler.h.

Referenced by CellHandler::iteratorT< CellHandler_T, Cell_T >::operator*(), CellHandler::iteratorT< CellHandler_T, Cell_T >::opera
and CellHandler::iteratorT< CellHandler_T, Cell_T >::operator->().

6.6.5.4 m_position

```
template<typename CellHandler_T , typename Cell_T >
QVector<unsigned int> CellHandler::iteratorT< CellHandler_T, Cell_T >::m_position [private]
```

Current position of the iterator.

Definition at line 88 of file cellhandler.h.

Referenced by CellHandler::iteratorT< CellHandler_T, Cell_T >::iteratorT(), CellHandler::iteratorT< CellHandler_T, Cell_T >::operat
CellHandler::iteratorT< CellHandler_T, Cell_T >::operator++(), and CellHandler::iteratorT< CellHandler_T, Cell_T >::operator->().

6.6.5.5 m_zero

```
template<typename CellHandler_T , typename Cell_T >
QVector<unsigned int> CellHandler::iteratorT< CellHandler_T, Cell_T >::m_zero [private]
```

Nul vector of the good dimension (depend of m_handler)

Definition at line 90 of file cellhandler.h.

Referenced by CellHandler::iteratorT< CellHandler_T, Cell_T >::iteratorT(), and CellHandler::iteratorT< CellHandler_T, Cell_T >::op

The documentation for this class was generated from the following files:

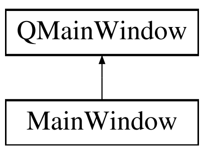
- cellhandler.h
- cellhandler.cpp

6.7 MainWindow Class Reference

Simulation window.

```
#include <mainwindow.h>
```

Inheritance diagram for MainWindow:



Public Slots

- void openFile ()

  Opens a file browser for the user to select automaton files and creates an automaton.
- void saveToFile ()

  Allows user to select a location and saves automaton's state and settings.
- void openCreationWindow ()

  Opens the automaton creation window.
- void receiveCellHandler (const QVector< unsigned int > dimensions, CellHandler::generationTypes type=CellHandler::generationTypes::empty, unsigned int stateMax=1, unsigned int density=20)

  Creates a new cellHandler with the provided arguments and updates the board with the created cellHandler.
- void addAutomatonRules (QList< const Rule *> rules)
- void addAutomatonRuleFile (QString path)
- void forward ()

  Skips the number of steps chosen by the user and sets the automaton on the last one.
- void closeTab (int n)

  Closes the tab at index n. Before closing, prompts the user to save the automaton.
- void runAutomaton ()
- void handlePlayPause ()
- void reset ()

Public Member Functions

- MainWindow (QWidget *parent=nullptr)

Private Member Functions

- void createIcons ()

  Creates Icons for the MainWindow.
- void createActions ()

  Creates and connects QActions and associated buttons for the MainWindow.
- void createToolBar ()

  Creates the toolBar for the MainWindow.
- void createBoard ()
- QWidget * createTab ()

  Creates a new Tab with an empty board.
- void createTabs ()

  Creates a QTabWidget for the main window and displays it.
- void addEmptyRow (unsigned int n)

  Add an empty row at the end of the board.
- void updateBoard (int index)

  Updates cells on the board on the tab at the given index with the cellHandler's cells states.
- void nextState (unsigned int n)

  Shows the nth next state of the automaton on the board.
- QTableWidget * getBoard (int n)

Private Attributes

- QTabWidget * m_tabs
- QIcon m_fastBackwardIcon

  Icons.
- QIcon m_fastForwardIcon
- QIcon m_playIcon
- QIcon m_pauseIcon
- QIcon m_newIcon
- QIcon m_saveIcon
- QIcon m_openIcon
- QIcon m_resetIcon
- QAction * m_playPause

  Actions.
- QAction * m_nextState
- QAction * m_previousState
- QAction * m_fastForward
- QAction * m_fastBackward
- QAction * m_openAutomaton
- QAction * m_saveAutomaton
- QAction * m_newAutomaton
- QAction * m_resetAutomaton
- QToolButton * m_playPauseBt

  Buttons.
- QToolButton * m_nextStateBt
- QToolButton * m_previousStateBt
- QToolButton * m_fastForwardBt
- QToolButton * m_fastBackwardBt
- QToolButton * m_openAutomatonBt
- QToolButton * m_saveAutomatonBt
- QToolButton * m_newAutomatonBt
- QToolButton * m_resetBt
- QSpinBox * m_timeStep
- QTimer * m_timer

  Simulation time step duration input.
- Automate * m_newAutomate
- bool running
- QToolBar * m_toolBar
- unsigned int m_boardHSize = 25

  Toolbar containing the buttons.
- unsigned int m_boardVSize = 25
- unsigned int m_cellSize = 30

6.7.1 Detailed Description

Simulation window.

Displays the automaton's current state as a board and contains user interaction components.

Definition at line 18 of file mainwindow.h.

6.7.2 Constructor & Destructor Documentation

6.7.2.1 MainWindow()

```
MainWindow::MainWindow (
            QWidget * parent = nullptr )  [explicit]
```

Definition at line 4 of file mainwindow.cpp.

References createActions(), createIcons(), createToolBar(), m_tabs, and running.

6.7.3 Member Function Documentation

6.7.3.1 addAutomatonRuleFile

```
void MainWindow::addAutomatonRuleFile (
            QString path )  [slot]
```

Definition at line 373 of file mainwindow.cpp.

References Automate::addRuleFile(), AutomateHandler::getAutomate(), and AutomateHandler::getAutomateHandler().

Referenced by openFile(), and receiveCellHandler().

6.7.3.2 addAutomatonRules

```
void MainWindow::addAutomatonRules (
            QList< const Rule *> rules )  [slot]
```

Definition at line 366 of file mainwindow.cpp.

References Automate::addRule(), AutomateHandler::getAutomate(), and AutomateHandler::getAutomateHandler().

Referenced by openFile(), and receiveCellHandler().

6.7.3.3 addEmptyRow()

```
void MainWindow::addEmptyRow (
            unsigned int n )  [private]
```

Add an empty row at the end of the board.

Used only in case of 1 dimension automaton

Parameters

n	Index of the board

Definition at line 341 of file mainwindow.cpp.

References getBoard(), and m_cellSize.

Referenced by updateBoard().

6.7.3.4 closeTab

```
void MainWindow::closeTab (
            int n )  [slot]
```

Closes the tab at index n. Before closing, prompts the user to save the automaton.

Definition at line 359 of file mainwindow.cpp.

References AutomateHandler::deleteAutomate(), AutomateHandler::getAutomateHandler(), m_tabs, and saveToFile().

Referenced by createTabs().

6.7.3.5 createActions()

```
void MainWindow::createActions ( )  [private]
```

Creates and connects QActions and associated buttons for the MainWindow.

Definition at line 53 of file mainwindow.cpp.

References forward(), handlePlayPause(), m_fastForward, m_fastForwardBt, m_fastForwardIcon, m_newAutomaton, m_newAutomatonBt, m_newIcon, m_openAutomaton, m_openAutomatonBt, m_openIcon, m_playIcon, m_playPause, m_playPauseBt, m_resetAutomaton, m_resetBt, m_resetIcon, m_saveAutomaton, m_saveAutomatonBt, m_saveIcon, openCreationWindow(), openFile(), reset(), and saveToFile().

Referenced by MainWindow().

6.7.3.6 createBoard()

```
void MainWindow::createBoard ( )  [private]
```

6.7.3.7 createIcons()

```
void MainWindow::createIcons ( ) [private]
```

Creates Icons for the MainWindow.

Definition at line 23 of file mainwindow.cpp.

References m_fastBackwardIcon, m_fastForwardIcon, m_newIcon, m_openIcon, m_pauseIcon, m_playIcon, m_resetIcon, and m_saveIcon.

Referenced by MainWindow().

6.7.3.8 createTab()

```
QWidget * MainWindow::createTab ( ) [private]
```

Creates a new Tab with an empty board.

Definition at line 135 of file mainwindow.cpp.

References AutomateHandler::getAutomate(), AutomateHandler::getAutomateHandler(), Automate::getCellHandler(), CellHandler::getDimensions(), and m_cellSize.

Referenced by openFile(), and receiveCellHandler().

6.7.3.9 createTabs()

```
void MainWindow::createTabs ( ) [private]
```

Creates a QTabWidget for the main window and displays it.

Definition at line 327 of file mainwindow.cpp.

References closeTab(), and m_tabs.

Referenced by openFile(), and receiveCellHandler().

6.7.3.10 createToolBar()

```
void MainWindow::createToolBar ( ) [private]
```

Creates the toolBar for the MainWindow.

Definition at line 96 of file mainwindow.cpp.

References m_fastForwardBt, m_newAutomatonBt, m_openAutomatonBt, m_playPauseBt, m_resetBt, m_saveAutomatonBt, m_timeStep, and m_toolBar.

Referenced by MainWindow().

6.7.3.11 forward

```
void MainWindow::forward ( ) [slot]
```

Skips the number of steps chosen by the user and sets the automaton on the last one.

Definition at line 314 of file mainwindow.cpp.

References nextState().

Referenced by createActions().

6.7.3.12 getBoard()

```
QTableWidget * MainWindow::getBoard (
             int n ) [private]
```

Definition at line 319 of file mainwindow.cpp.

References m_tabs.

Referenced by addEmptyRow(), reset(), and updateBoard().

6.7.3.13 handlePlayPause

```
void MainWindow::handlePlayPause ( ) [slot]
```

Definition at line 377 of file mainwindow.cpp.

References AutomateHandler::getAutomateHandler(), m_pauseIcon, m_playIcon, m_playPauseBt, m_timer, m_timeStep, runAutomaton(), and running.

Referenced by createActions().

6.7.3.14 nextState()

```
void MainWindow::nextState (
             unsigned int n ) [private]
```

Shows the nth next state of the automaton on the board.

Definition at line 252 of file mainwindow.cpp.

References AutomateHandler::getAutomate(), AutomateHandler::getAutomateHandler(), m_tabs, and updateBoard().

Referenced by forward().

6.7.3.15 openCreationWindow

```
void MainWindow::openCreationWindow ( )  [slot]
```

Opens the automaton creation window.

Definition at line 217 of file mainwindow.cpp.

References receiveCellHandler().

Referenced by createActions().

6.7.3.16 openFile

```
void MainWindow::openFile ( )  [slot]
```

Opens a file browser for the user to select automaton files and creates an automaton.

Definition at line 179 of file mainwindow.cpp.

References AutomateHandler::addAutomate(), addAutomatonRuleFile(), addAutomatonRules(), createTab(), createTabs(), AutomateHandler::getAutomateHandler(), m_tabs, and updateBoard().

Referenced by createActions().

6.7.3.17 receiveCellHandler

```
void MainWindow::receiveCellHandler (
            const QVector< unsigned int > dimensions,
            CellHandler::generationTypes type = CellHandler::generationTypes::empty,
            unsigned int stateMax = 1,
            unsigned int density = 20 )  [slot]
```

Creates a new cellHandler with the provided arguments and updates the board with the created cellHandler.

Definition at line 230 of file mainwindow.cpp.

References AutomateHandler::addAutomate(), addAutomatonRuleFile(), addAutomatonRules(), createTab(), createTabs(), AutomateHandler::getAutomateHandler(), m_tabs, and updateBoard().

Referenced by openCreationWindow().

6.7.3.18 reset

```
void MainWindow::reset ( ) [slot]
```

Definition at line 409 of file mainwindow.cpp.

References AutomateHandler::getAutomate(), AutomateHandler::getAutomateHandler(), getBoard(), m_cellSize, m_tabs, and updateBoard().

Referenced by createActions().

6.7.3.19 runAutomaton

```
void MainWindow::runAutomaton ( ) [slot]
```

Definition at line 400 of file mainwindow.cpp.

References AutomateHandler::getAutomate(), AutomateHandler::getAutomateHandler(), m_tabs, running, and updateBoard().

Referenced by handlePlayPause().

6.7.3.20 saveToFile

```
void MainWindow::saveToFile ( ) [slot]
```

Allows user to select a location and saves automaton's state and settings.

Definition at line 199 of file mainwindow.cpp.

References AutomateHandler::getAutomate(), AutomateHandler::getAutomateHandler(), and m_tabs.

Referenced by closeTab(), and createActions().

6.7.3.21 updateBoard()

```
void MainWindow::updateBoard (
            int index ) [private]
```

Updates cells on the board on the tab at the given index with the cellHandler's cells states.

Definition at line 269 of file mainwindow.cpp.

References addEmptyRow(), CellHandler::begin(), CellHandler::end(), AutomateHandler::getAutomate(), AutomateHandler::getAutomateHandler(), getBoard(), Automate::getCellHandler(), and CellHandler::getDimensions().

Referenced by nextState(), openFile(), receiveCellHandler(), reset(), and runAutomaton().

6.7.4 Member Data Documentation

6.7.4.1 m_boardHSize

```
unsigned int MainWindow::m_boardHSize = 25  [private]
```

Toolbar containing the buttons.

Board size settings

Definition at line 66 of file mainwindow.h.

6.7.4.2 m_boardVSize

```
unsigned int MainWindow::m_boardVSize = 25  [private]
```

Definition at line 67 of file mainwindow.h.

6.7.4.3 m_cellSize

```
unsigned int MainWindow::m_cellSize = 30  [private]
```

Definition at line 68 of file mainwindow.h.

Referenced by addEmptyRow(), createTab(), and reset().

6.7.4.4 m_fastBackward

```
QAction* MainWindow::m_fastBackward  [private]
```

Definition at line 40 of file mainwindow.h.

6.7.4.5 m_fastBackwardBt

```
QToolButton* MainWindow::m_fastBackwardBt  [private]
```

Definition at line 51 of file mainwindow.h.

6.7.4.6 m_fastBackwardIcon

```
QIcon MainWindow::m_fastBackwardIcon [private]
```

Icons.

Definition at line 26 of file mainwindow.h.

Referenced by createIcons().

6.7.4.7 m_fastForward

```
QAction* MainWindow::m_fastForward [private]
```

Definition at line 39 of file mainwindow.h.

Referenced by createActions().

6.7.4.8 m_fastForwardBt

```
QToolButton* MainWindow::m_fastForwardBt [private]
```

Definition at line 50 of file mainwindow.h.

Referenced by createActions(), and createToolBar().

6.7.4.9 m_fastForwardIcon

```
QIcon MainWindow::m_fastForwardIcon [private]
```

Definition at line 27 of file mainwindow.h.

Referenced by createActions(), and createIcons().

6.7.4.10 m_newAutomate

```
Automate* MainWindow::m_newAutomate [private]
```

Definition at line 61 of file mainwindow.h.

6.7.4.11 m_newAutomaton

```
QAction* MainWindow::m_newAutomaton [private]
```

Definition at line 43 of file mainwindow.h.

Referenced by createActions().

6.7.4.12 m_newAutomatonBt

```
QToolButton* MainWindow::m_newAutomatonBt [private]
```

Definition at line 54 of file mainwindow.h.

Referenced by createActions(), and createToolBar().

6.7.4.13 m_newIcon

```
QIcon MainWindow::m_newIcon [private]
```

Definition at line 30 of file mainwindow.h.

Referenced by createActions(), and createIcons().

6.7.4.14 m_nextState

```
QAction* MainWindow::m_nextState [private]
```

Definition at line 37 of file mainwindow.h.

6.7.4.15 m_nextStateBt

```
QToolButton* MainWindow::m_nextStateBt [private]
```

Definition at line 48 of file mainwindow.h.

6.7.4.16 m_openAutomaton

```
QAction* MainWindow::m_openAutomaton [private]
```

Definition at line 41 of file mainwindow.h.

Referenced by createActions().

6.7.4.17 m_openAutomatonBt

```
QToolButton* MainWindow::m_openAutomatonBt [private]
```

Definition at line 52 of file mainwindow.h.

Referenced by createActions(), and createToolBar().

6.7.4.18 m_openIcon

```
QIcon MainWindow::m_openIcon [private]
```

Definition at line 32 of file mainwindow.h.

Referenced by createActions(), and createIcons().

6.7.4.19 m_pauseIcon

```
QIcon MainWindow::m_pauseIcon [private]
```

Definition at line 29 of file mainwindow.h.

Referenced by createIcons(), and handlePlayPause().

6.7.4.20 m_playIcon

```
QIcon MainWindow::m_playIcon [private]
```

Definition at line 28 of file mainwindow.h.

Referenced by createActions(), createIcons(), and handlePlayPause().

6.7.4.21 m_playPause

`QAction* MainWindow::m_playPause [private]`

Actions.

Definition at line 36 of file mainwindow.h.

Referenced by createActions().

6.7.4.22 m_playPauseBt

`QToolButton* MainWindow::m_playPauseBt [private]`

Buttons.

Definition at line 47 of file mainwindow.h.

Referenced by createActions(), createToolBar(), and handlePlayPause().

6.7.4.23 m_previousState

`QAction* MainWindow::m_previousState [private]`

Definition at line 38 of file mainwindow.h.

6.7.4.24 m_previousStateBt

`QToolButton* MainWindow::m_previousStateBt [private]`

Definition at line 49 of file mainwindow.h.

6.7.4.25 m_resetAutomaton

`QAction* MainWindow::m_resetAutomaton [private]`

Definition at line 44 of file mainwindow.h.

Referenced by createActions().

6.7.4.26 m_resetBt

`QToolButton* MainWindow::m_resetBt [private]`

Definition at line 55 of file mainwindow.h.

Referenced by createActions(), and createToolBar().

6.7.4.27 m_resetIcon

`QIcon MainWindow::m_resetIcon [private]`

Definition at line 33 of file mainwindow.h.

Referenced by createActions(), and createIcons().

6.7.4.28 m_saveAutomaton

`QAction* MainWindow::m_saveAutomaton [private]`

Definition at line 42 of file mainwindow.h.

Referenced by createActions().

6.7.4.29 m_saveAutomatonBt

`QToolButton* MainWindow::m_saveAutomatonBt [private]`

Definition at line 53 of file mainwindow.h.

Referenced by createActions(), and createToolBar().

6.7.4.30 m_saveIcon

`QIcon MainWindow::m_saveIcon [private]`

Definition at line 31 of file mainwindow.h.

Referenced by createActions(), and createIcons().

6.7.4.31 m_tabs

```
QTabWidget* MainWindow::m_tabs [private]
```

Definition at line 22 of file mainwindow.h.

Referenced by closeTab(), createTabs(), getBoard(), MainWindow(), nextState(), openFile(), receiveCellHandler(), reset(), runAutomaton(), and saveToFile().

6.7.4.32 m_timer

```
QTimer* MainWindow::m_timer [private]
```

Simulation time step duration input.

Definition at line 59 of file mainwindow.h.

Referenced by handlePlayPause().

6.7.4.33 m_timeStep

```
QSpinBox* MainWindow::m_timeStep [private]
```

Definition at line 58 of file mainwindow.h.

Referenced by createToolBar(), and handlePlayPause().

6.7.4.34 m_toolBar

```
QToolBar* MainWindow::m_toolBar [private]
```

Definition at line 63 of file mainwindow.h.

Referenced by createToolBar().

6.7.4.35 running

```
bool MainWindow::running [private]
```

Definition at line 62 of file mainwindow.h.

Referenced by handlePlayPause(), MainWindow(), and runAutomaton().

The documentation for this class was generated from the following files:

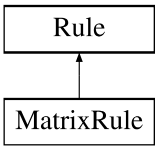
- mainwindow.h
- mainwindow.cpp

6.8 MatrixRule Class Reference

Manage specific rules, about specific values of specific neighbour.

```
#include <matrixrule.h>
```

Inheritance diagram for MatrixRule:



Public Member Functions

- MatrixRule (unsigned int finalState, QVector< unsigned int > currentStates=QVector< unsigned int >())

  Constructor.
- virtual bool matchCell (const Cell *cell) const

  Tells if the cell match the rule.
- void addNeighbourState (QVector< short > relativePosition, unsigned int matchState)

  Add a possible state to a relative position.
- void addNeighbourState (QVector< short > relativePosition, QVector< unsigned int > matchStates)

  Add multiples possible states to existents states.
- QJsonObject toJson () const

  Return a QJsonObject to save the rule.

Private Attributes

- QMap< QVector< short >, QVector< unsigned int > > m_matrix

  Key correspond to relative position and the value to matchable states.

Additional Inherited Members

6.8.1 Detailed Description

Manage specific rules, about specific values of specific neighbour.

Definition at line 13 of file matrixrule.h.

6.8.2 Constructor & Destructor Documentation

6.8.2.1 MatrixRule()

```
MatrixRule::MatrixRule (
            unsigned int finalState,
            QVector< unsigned int > currentStates = QVector<unsigned int>() )
```

Constructor.

Parameters

finalState	Final state if the rule match the cell
currentStates	Possibles states of the cell. Nothing means all states

Definition at line 21 of file matrixrule.cpp.

6.8.3 Member Function Documentation

6.8.3.1 addNeighbourState() [1/2]

```
void MatrixRule::addNeighbourState (
            QVector< short > relativePosition,
            unsigned int matchState )
```

Add a possible state to a relative position.

Definition at line 60 of file matrixrule.cpp.

References m_matrix.

Referenced by getRuleFromNumber(), and Automate::loadRules().

6.8.3.2 addNeighbourState() [2/2]

```
void MatrixRule::addNeighbourState (
            QVector< short > relativePosition,
            QVector< unsigned int > matchStates )
```

Add multiples possible states to existents states.

Definition at line 67 of file matrixrule.cpp.

References m_matrix.

6.8.3.3 matchCell()

```
bool MatrixRule::matchCell (
            const Cell * cell ) const  [virtual]
```

Tells if the cell match the rule.

Parameters

cell	Cell to test

Returns

True if the cell match the rule

Implements Rule.

Definition at line 30 of file matrixrule.cpp.

References Cell::getNeighbour(), Cell::getState(), Rule::m_currentCellPossibleValues, and m_matrix.

6.8.3.4 toJson()

```
QJsonObject MatrixRule::toJson ( ) const  [virtual]
```

Return a QJsonObject to save the rule.

Implements Rule.

Definition at line 75 of file matrixrule.cpp.

References m_matrix, and Rule::toJson().

6.8.4 Member Data Documentation

6.8.4.1 m_matrix

```
QMap<QVector<short>, QVector<unsigned int> > MatrixRule::m_matrix  [private]
```

Key correspond to relative position and the value to matchable states.

Definition at line 28 of file matrixrule.h.

Referenced by addNeighbourState(), matchCell(), and toJson().

The documentation for this class was generated from the following files:

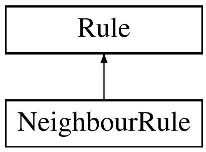
- matrixrule.h
- matrixrule.cpp

6.9 NeighbourRule Class Reference

Contains the rule condition and the output state if that condition is satisfied The rule modifies a cell depending on the number of its neighbours belonging to a range.

```
#include <neighbourrule.h>
```

Inheritance diagram for NeighbourRule:



Public Member Functions

- NeighbourRule (unsigned int outputState, QVector< unsigned int > currentCellValues, QPair< unsigned int, unsigned int > intervalNbrNeighbour, QSet< unsigned int > neighbourValues=QSet< unsigned int >())

  Constructs a neighbour rule with the parameters.
- ~NeighbourRule ()
- bool matchCell (const Cell *c) const

  Checks if the input cell satisfies the rule condition.
- virtual QJsonObject toJson () const

  Return a QJsonObject to save the rule.

Private Member Functions

- bool inInterval (unsigned int matchingNeighbours) const

  According to the requirements : a and b values are chosen by the user. No matter its current state, if the cell has between a and b neighbours living, it lives, else it dies/or stays dead. So the "current cell possible values" vector contains all the possible cell values (0 and 1) and the 2 pair contains (a, b) with an output state set at 1. 2 other rules, respectively with an interval of (0,a-1) and (b+1, 8) and an output state of 0 are created.

Private Attributes

- QPair< unsigned int, unsigned int > m_neighbourInterval

  Stores the rule condition regarding the number of neighbours.
- QSet< unsigned int > m_neighbourPossibleValues

  Stores the possible values of the neighbours to fit with the rule.

Additional Inherited Members

6.9.1 Detailed Description

Contains the rule condition and the output state if that condition is satisfied The rule modifies a cell depending on the number of its neighbours belonging to a range.

Definition at line 13 of file neighbourrule.h.

6.9.2 Constructor & Destructor Documentation

6.9.2.1 NeighbourRule()

```
NeighbourRule::NeighbourRule (
            unsigned int outputState,
            QVector< unsigned int > currentCellValues,
            QPair< unsigned int, unsigned int > intervalNbrNeighbour,
            QSet< unsigned int > neighbourValues = QSet<unsigned int>() )
```

Constructs a neighbour rule with the parameters.

Definition at line 95 of file neighbourrule.cpp.

References m_neighbourInterval.

6.9.2.2 ∼NeighbourRule()

```
NeighbourRule::∼NeighbourRule ( )
```

Definition at line 104 of file neighbourrule.cpp.

6.9.3 Member Function Documentation

6.9.3.1 inInterval()

```
bool NeighbourRule::inInterval (
            unsigned int matchingNeighbours ) const [private]
```

According to the requirements : a and b values are chosen by the user. No matter its current state, if the cell has between a and b neighbours living, it lives, else it dies/or stays dead. So the "current cell possible values" vector contains all the possible cell values (0 and 1) and the 2 pair contains (a, b) with an output state set at 1. 2 other rules, respectively with an interval of (0,a-1) and (b+1, 8) and an output state of 0 are created.

The game of life by John Horton Conway according to wikipedia:

"At each step, the cell evolution is determined by the state of its 8 neighbours as following: A dead cell which has exactly 3 living neighbours starts to live. An alive cell which has 2 or 3 living neighbours stays alive, else it dies."

1 : cell is alive 0 : cell is dead

```
Rule 1: dead cell (state 0) starts living (state 1) if it has exactly 3 living neighbours (in state 1)

unsigned int rule1OutputState = 1; // output state is alive state

QVector<unsigned int> rule1CurrentCellValues;
rule1CurrentCellValues.insert(0); //current cell is dead

QPair<unsigned int, unsigned int> rule1intervalNbrNeighbours;
rule1IntervalNbrNeighbours.first = 3;
rule1IntervalNbrNeighbours.second = 3;

QSet<unsigned int> rule1NeighbourPossibleValues;
rule1NeighbourPossibleValues<<1; //living neighbours

NeighbourRule rule1 = NeighbourRule(rule1OutputState, rule1CurrentCellValues,
      rule1IntervalNbrNeighbours, rule1NeighbourPossibleValues);


Rule 2: alive cell (state 1) dies (goes to state 0) if it has 0 to 1 living neighbours (in state 1)

unsigned int rule2OutputState = 0; // output state is dead state

QVector<unsigned int> rule2CurrentCellValues;
rule2CurrentCellValues.insert(1); //current cell is alive

QPair<unsigned int, unsigned int> rule2intervalNbrNeighbours;
rule2IntervalNbrNeighbours.first = 0;
rule2IntervalNbrNeighbours.second = 1;

QSet<unsigned int> rule2NeighbourPossibleValues;
rule2NeighbourPossibleValues<<1; //living neighbours

NeighbourRule rule2 = NeighbourRule(rule2OutputState, rule2CurrentCellValues,
      rule2IntervalNbrNeighbours, rule2NeighbourPossibleValues);


Rule 3: alive cell (state 1) dies (goes to state 0) if it has 4 to 8 living neighbours (in state 1)

unsigned int rule3OutputState = 0; // output state is dead state

QVector<unsigned int> rule3CurrentCellValues;
rule2CurrentCellValues.insert(1); //current cell is alive

QPair<unsigned int, unsigned int> rule3intervalNbrNeighbours;
rule3IntervalNbrNeighbours.first = 4;
rule3IntervalNbrNeighbours.second = 8;

QSet<unsigned int> rule3NeighbourPossibleValues;
rule3NeighbourPossibleValues<<1; //living neighbours

NeighbourRule rule3 = NeighbourRule(rule3OutputState, rule3CurrentCellValues,
      rule3IntervalNbrNeighbours, rule3NeighbourPossibleValues);
```

Checks if the number of neighbours matching the state condition belongs to the condition interval

Parameters

matchingNeighbours	Number of neighbours matching the rule condition regarding their values

Returns

True if the number of neighbours matches with the interval condition

Definition at line 84 of file neighbourrule.cpp.

References m_neighbourInterval.

Referenced by matchCell().

6.9.3.2 matchCell()

```
bool NeighbourRule::matchCell (
            const Cell * c ) const  [virtual]
```

Checks if the input cell satisfies the rule condition.

Parameters

c	current cell

Returns

True if the cell matches the rule condition

Implements Rule.

Definition at line 115 of file neighbourrule.cpp.

References Cell::countNeighbours(), Cell::getState(), inInterval(), Rule::m_currentCellPossibleValues, and m_neighbourPossibleValues.

6.9.3.3 toJson()

```
QJsonObject NeighbourRule::toJson ( ) const  [virtual]
```

Return a QJsonObject to save the rule.

Implements Rule.

Definition at line 146 of file neighbourrule.cpp.

References m_neighbourInterval, m_neighbourPossibleValues, and Rule::toJson().

6.9.4 Member Data Documentation

6.9.4.1 m_neighbourInterval

```
QPair<unsigned int , unsigned int> NeighbourRule::m_neighbourInterval  [private]
```

Stores the rule condition regarding the number of neighbours.

Definition at line 16 of file neighbourrule.h.

Referenced by inInterval(), NeighbourRule(), and toJson().

6.9.4.2 m_neighbourPossibleValues

```
QSet<unsigned int> NeighbourRule::m_neighbourPossibleValues [private]
```

Stores the possible values of the neighbours to fit with the rule.

Definition at line 18 of file neighbourrule.h.

Referenced by matchCell(), and toJson().

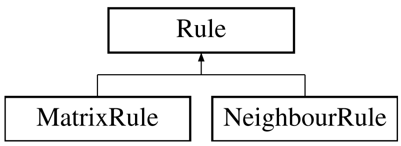
The documentation for this class was generated from the following files:

- neighbourrule.h
- neighbourrule.cpp

6.10 Rule Class Reference

```
#include <rule.h>
```

Inheritance diagram for Rule:



Public Member Functions

- Rule (QVector< unsigned int > currentCellValues, unsigned int outputState)
- virtual QJsonObject toJson () const =0
- virtual bool matchCell (const Cell ∗c) const =0

  Verify if the cell match the rule.
- unsigned int getCellOutputState () const

  Get the rule output state that will be the next state if the cell matches the rule condition.

Protected Attributes

- QVector< unsigned int > m_currentCellPossibleValues

  Stores the possible values of the current cell as part of the rule condition.
- unsigned int m_cellOutputState

  Stores the output state of the cell if the condition is matched.

6.10.1 Detailed Description

Definition at line 13 of file rule.h.

6.10.2 Constructor & Destructor Documentation

6.10.2.1 Rule()

```
Rule::Rule (
            QVector< unsigned int > currentCellValues,
            unsigned int outputState )
```

Definition at line 3 of file rule.cpp.

6.10.3 Member Function Documentation

6.10.3.1 getCellOutputState()

```
unsigned int Rule::getCellOutputState ( ) const
```

Get the rule output state that will be the next state if the cell matches the rule condition.

Definition at line 26 of file rule.cpp.

References m_cellOutputState.

6.10.3.2 matchCell()

```
virtual bool Rule::matchCell (
            const Cell * c ) const [pure virtual]
```

Verify if the cell match the rule.

Using :

```
if (rule.matchCell(&cell))
    cell.setState(rule.getCellOutputState());
```

Parameters

c	Cell to test

Implemented in NeighbourRule, and MatrixRule.

6.10.3.3 toJson()

```
QJsonObject Rule::toJson ( ) const  [pure virtual]
```

Implemented in NeighbourRule, and MatrixRule.

Definition at line 9 of file rule.cpp.

References m_cellOutputState, and m_currentCellPossibleValues.

Referenced by MatrixRule::toJson(), and NeighbourRule::toJson().

6.10.4 Member Data Documentation

6.10.4.1 m_cellOutputState

```
unsigned int Rule::m_cellOutputState  [protected]
```

Stores the output state of the cell if the condition is matched.

Definition at line 17 of file rule.h.

Referenced by getCellOutputState(), and toJson().

6.10.4.2 m_currentCellPossibleValues

```
QVector<unsigned int> Rule::m_currentCellPossibleValues  [protected]
```

Stores the possible values of the current cell as part of the rule condition.

Definition at line 16 of file rule.h.

Referenced by MatrixRule::matchCell(), NeighbourRule::matchCell(), and toJson().

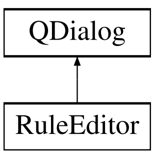
The documentation for this class was generated from the following files:

- rule.h
- rule.cpp

6.11 RuleEditor Class Reference

```
#include <ruleeditor.h>
```

Inheritance diagram for RuleEditor:



Public Slots

- void removeRule ()
- void addRule ()
- void importFile ()
- void sendRules ()

Signals

- void rulesFilled (QList< const Rule *> rules)
- void fileImported (QString path)

Public Member Functions

- RuleEditor (QWidget *parent=nullptr)

Private Attributes

- QList< const Rule * > m_rules
- QListWidget * m_rulesListWidget
- QTableWidget * m_rulesTable
- QSpinBox * m_outputStateBox
- QLineEdit * m_currentStatesEdit
- QLineEdit * m_neighbourStatesEdit
- QSpinBox * m_upperNeighbourBox
- QSpinBox * m_lowerNeighbourBox
- QPushButton * m_addBt
- QPushButton * m_doneBt
- QPushButton * m_removeBt
- QPushButton * m_importBt
- unsigned int selectedRule

6.11.1 Detailed Description

Definition at line 7 of file ruleeditor.h.

6.11.2 Constructor & Destructor Documentation

6.11.2.1 RuleEditor()

```
RuleEditor::RuleEditor (
            QWidget * parent = nullptr )  [explicit]
```

Definition at line 3 of file ruleeditor.cpp.

References addRule(), importFile(), m_addBt, m_currentStatesEdit, m_doneBt, m_importBt, m_lowerNeighbourBox, m_neighbourStatesEdit, m_outputStateBox, m_removeBt, m_rulesListWidget, m_upperNeighbourBox, removeRule(), selectedRule, and sendRules().

6.11.3 Member Function Documentation

6.11.3.1 addRule

```
void RuleEditor::addRule ( )  [slot]
```

Definition at line 66 of file ruleeditor.cpp.

References m_currentStatesEdit, m_lowerNeighbourBox, m_neighbourStatesEdit, m_outputStateBox, m_rules, m_rulesListWidget, and m_upperNeighbourBox.

Referenced by RuleEditor().

6.11.3.2 fileImported

```
void RuleEditor::fileImported (
            QString path )  [signal]
```

Referenced by importFile().

6.11.3.3 importFile

```
void RuleEditor::importFile ( )  [slot]
```

Definition at line 99 of file ruleeditor.cpp.

References fileImported().

Referenced by RuleEditor().

6.11.3.4 removeRule

```
void RuleEditor::removeRule ( )  [slot]
```

Definition at line 89 of file ruleeditor.cpp.

References m_rules, and m_rulesListWidget.

Referenced by RuleEditor().

6.11.3.5 rulesFilled

```
void RuleEditor::rulesFilled (
            QList< const Rule *> rules )  [signal]
```

Referenced by sendRules().

6.11.3.6 sendRules

```
void RuleEditor::sendRules ( )  [slot]
```

Definition at line 94 of file ruleeditor.cpp.

References m_rules, and rulesFilled().

Referenced by RuleEditor().

6.11.4 Member Data Documentation

6.11.4.1 m_addBt

```
QPushButton* RuleEditor::m_addBt  [private]
```

Definition at line 20 of file ruleeditor.h.

Referenced by RuleEditor().

6.11.4.2 m_currentStatesEdit

```
QLineEdit* RuleEditor::m_currentStatesEdit  [private]
```

Definition at line 15 of file ruleeditor.h.

Referenced by addRule(), and RuleEditor().

6.11.4.3 m_doneBt

```
QPushButton* RuleEditor::m_doneBt  [private]
```

Definition at line 21 of file ruleeditor.h.

Referenced by RuleEditor().

6.11.4.4 m_importBt

```
QPushButton* RuleEditor::m_importBt  [private]
```

Definition at line 23 of file ruleeditor.h.

Referenced by RuleEditor().

6.11.4.5 m_lowerNeighbourBox

```
QSpinBox* RuleEditor::m_lowerNeighbourBox  [private]
```

Definition at line 18 of file ruleeditor.h.

Referenced by addRule(), and RuleEditor().

6.11.4.6 m_neighbourStatesEdit

```
QLineEdit* RuleEditor::m_neighbourStatesEdit  [private]
```

Definition at line 16 of file ruleeditor.h.

Referenced by addRule(), and RuleEditor().

6.11.4.7 m_outputStateBox

```
QSpinBox* RuleEditor::m_outputStateBox  [private]
```

Definition at line 14 of file ruleeditor.h.

Referenced by addRule(), and RuleEditor().

6.11.4.8 m_removeBt

```
QPushButton* RuleEditor::m_removeBt  [private]
```

Definition at line 22 of file ruleeditor.h.

Referenced by RuleEditor().

6.11.4.9 m_rules

```
QList<const Rule*> RuleEditor::m_rules [private]
```

Definition at line 10 of file ruleeditor.h.

Referenced by addRule(), removeRule(), and sendRules().

6.11.4.10 m_rulesListWidget

```
QListWidget* RuleEditor::m_rulesListWidget [private]
```

Definition at line 11 of file ruleeditor.h.

Referenced by addRule(), removeRule(), and RuleEditor().

6.11.4.11 m_rulesTable

```
QTableWidget* RuleEditor::m_rulesTable [private]
```

Definition at line 12 of file ruleeditor.h.

6.11.4.12 m_upperNeighbourBox

```
QSpinBox* RuleEditor::m_upperNeighbourBox [private]
```

Definition at line 17 of file ruleeditor.h.

Referenced by addRule(), and RuleEditor().

6.11.4.13 selectedRule

```
unsigned int RuleEditor::selectedRule [private]
```

Definition at line 25 of file ruleeditor.h.

Referenced by RuleEditor().

The documentation for this class was generated from the following files:

- ruleeditor.h
- ruleeditor.cpp

Chapter 7

File Documentation

7.1 automate.cpp File Reference

```
#include "automate.h"
```

Functions

- QList< const Rule * > generate1DRules (unsigned int automatonNumber)
- const MatrixRule * getRuleFromNumber (int previousConfiguration, int nextState)

7.1.1 Function Documentation

7.1.1.1 generate1DRules()

```
QList<const Rule *> generate1DRules (
            unsigned int automatonNumber )
```

Definition at line 315 of file automate.cpp.

References getRuleFromNumber().

7.1.1.2 getRuleFromNumber()

```
const MatrixRule* getRuleFromNumber (
            int previousConfiguration,
            int nextState )
```

Definition at line 338 of file automate.cpp.

References MatrixRule::addNeighbourState().

Referenced by generate1DRules().

7.2 automate.cpp

```
#include "automate.h"

bool Automate::loadRules(const QJsonArray &json)
{

    for (QJsonArray::const_iterator it = json.begin(); it != json.end(); ++it)
    {
        if (!it->isObject())
            return false;
        QJsonObject ruleJson = it->toObject();

        if (!ruleJson.contains("type") || !ruleJson["type"].isString())
            return false;
        if (!ruleJson.contains("finalState") || !ruleJson["finalState"].isDouble())
            return false;
        if (!ruleJson.contains("currentStates") || !ruleJson["currentStates"].isArray())
            return false;

        QVector<unsigned int> currentStates;
        QJsonArray statesJson = ruleJson["currentStates"].toArray();
        for (unsigned int i = 0; i < statesJson.size(); i++)
        {
            if (!statesJson.at(i).isDouble())
                return false;
            currentStates.push_back(statesJson.at(i).toInt());
        }

        if (!ruleJson["type"].toString().compare("neighbour", Qt::CaseInsensitive))
        {
            if (!ruleJson.contains("neighbourNumberMin") || !ruleJson["neighbourNumberMin"].isDouble())
                return false;
            if (!ruleJson.contains("neighbourNumberMax") || !ruleJson["neighbourNumberMax"].isDouble())
                return false;



            QPair<unsigned int, unsigned int> nbrNeighbourInterval(ruleJson["neighbourNumberMin"].toInt(), ruleJson["neighbourNumberMax"].toInt());
            NeighbourRule *newRule;
            if (ruleJson.contains("neighbourStates"))
            {
                if (!ruleJson["neighbourStates"].isArray())
                    return false;
                QSet<unsigned int> neighbourStates;

                QJsonArray statesJson = ruleJson["neighbourStates"].toArray();
                for (unsigned int i = 0; i < statesJson.size(); i++)
                {
                    if (!statesJson.at(i).isDouble())
                        return false;
                    neighbourStates.insert(statesJson.at(i).toInt());
                }
                newRule = new NeighbourRule((unsigned int)ruleJson["finalState"].toInt(), currentStates, nbrNeighbourInterval, neighbourStates);
            }
            else
                newRule = new NeighbourRule((unsigned int)ruleJson["finalState"].toInt(), currentStates, nbrNeighbourInterval);
            m_rules.push_back(newRule);
        }
        else if (!ruleJson["type"].toString().compare("matrix", Qt::CaseInsensitive))
        {
            MatrixRule *newRule = new MatrixRule((unsigned int)ruleJson["finalState"].toInt(), currentStates);
            if (ruleJson.contains("neighbours"))
            {
                if (!ruleJson["neighbours"].isArray())
                    return false;
                QJsonArray neighboursJson = ruleJson["neighbours"].toArray();
                for (unsigned int i = 0; i < neighboursJson.size(); i++)
                {
                    if (!neighboursJson.at(i).isObject())
                        return false;

                    if (!neighboursJson.at(i).toObject().contains("relativePosition") || !neighboursJson.at(i).toObject()["relativePosition"].isArray())
                        return false;
                    if (!neighboursJson.at(i).toObject().contains("neighbourStates") || !neighboursJson.at(i).toObject()["neighbourStates"].isArray())
                        return false;

                    QVector<unsigned int> neighbourStates;


```

```
                    QJsonArray statesJson = neighboursJson.at(i).toObject()["neighbourStates"].toArray();
                    for (unsigned int j = 0; j < statesJson.size(); j++)
                    {
                        if (!statesJson.at(j).isDouble())
                            return false;
                        neighbourStates.push_back(statesJson.at(j).toInt());
                    }

                    QVector<short> relativePosition;
                    QJsonArray positionJson = neighboursJson.at(i).toObject()["relativePosition"].toArray();
                    for (unsigned int j = 0; j < positionJson.size(); j++)
                    {
                        if (!positionJson.at(j).isDouble())
                            return false;
                        relativePosition.push_back(positionJson.at(j).toInt());
                    }
                    if (relativePosition.size() != m_cellHandler->getDimensions().size())
                        return false;
                    newRule->addNeighbourState(relativePosition, neighbourStates);
                }

            }
            m_rules.push_back(newRule);


        }
        else
            return false;

    }
    return true;
}

Automate::Automate(QString cellHandlerFilename)
{
    m_cellHandler = new CellHandler(cellHandlerFilename);

}

Automate::Automate(const QVector<unsigned int> dimensions, CellHandler::generationTypes type, unsigned int stateMax, unsigned int density)
{
    m_cellHandler = new CellHandler(dimensions, type, stateMax, density);

}

Automate::Automate(QString cellHandlerFilename, QString ruleFilename)
{
    m_cellHandler = new CellHandler(cellHandlerFilename);

    QFile ruleFile(ruleFilename);
    if (!ruleFile.open(QIODevice::ReadOnly | QIODevice::Text)) {
        qWarning("Couldn't open given file.");
        throw QString(QObject::tr("Couldn't open given file"));
    }

    QJsonParseError parseErr;
    QJsonDocument loadDoc(QJsonDocument::fromJson(ruleFile.readAll(), &parseErr));

    ruleFile.close();


    if (loadDoc.isNull() || loadDoc.isEmpty())
    {
        qWarning() << "Could not read data : ";
        qWarning() << parseErr.errorString();
        throw QString(parseErr.errorString());
    }

    if (!loadDoc.isArray())
    {
        qWarning() << "We need an array of rules !";
        throw QString(QObject::tr("We need an array of rules!"));
    }

    loadRules(loadDoc.array());

}

Automate::~Automate()
{
    delete m_cellHandler;
    for (QList<const Rule*>::iterator it = m_rules.begin(); it != m_rules.end(); ++it)
    {
        delete *it;
```

```
    }
}

bool Automate::saveRules(QString filename) const
{
    QFile ruleFile(filename);
    if (!ruleFile.open(QIODevice::WriteOnly | QIODevice::Text)) {
        qWarning("Couldn't open given file.");
        throw QString(QObject::tr("Couldn't open given file"));
    }

    QJsonArray array;

    for (QList<const Rule*>::const_iterator it = m_rules.cbegin(); it != m_rules.cend(); ++it)
        array.append((*it)->toJson());

    QJsonDocument doc(array);

    ruleFile.write(doc.toJson());

    return true;
}

bool Automate::saveCells(QString filename) const
{
    if (m_cellHandler != nullptr)
        return m_cellHandler->save(filename);
    return false;
}

bool Automate::saveAll(QString cellHandlerFilename, QString rulesFilename) const
{
    return saveRules(rulesFilename) && saveCells(cellHandlerFilename);
}

void Automate::addRule(const Rule *newRule)
{
    m_rules.push_back(newRule);
}

void Automate::setRulePriority(const Rule *rule, unsigned int newPlace)
{
    m_rules.move(m_rules.indexOf(rule), newPlace);
}

const QList<const Rule *> &Automate::getRules() const
{
   return m_rules;
}

bool Automate::run(unsigned int nbSteps) //void instead ?
{
    for(unsigned int i = 0; i<nbSteps; ++i)
    {
        for (CellHandler::iterator it = m_cellHandler->begin(); it != m_cellHandler->end(); ++it)
        {
            for (QList<const Rule*>::iterator rule = m_rules.begin(); rule != m_rules.end() ; ++rule)
            {
                if((*rule)->matchCell(*it)) //if the cell matches with the rule, its state is changed
                {
                    it->setState((*rule)->getCellOutputState());
                    break;
                }
            }


        }
        m_cellHandler->nextStates(); //apply the changes to all the cells simultaneously
    }
    return true;

}

const CellHandler &Automate::getCellHandler() const
{
    return *m_cellHandler;
}

void Automate::addRuleFile(QString filename){
    QFile ruleFile(filename);
    if (!ruleFile.open(QIODevice::ReadOnly | QIODevice::Text)) {
        qWarning("Couldn't open given file.");
        throw QString(QObject::tr("Couldn't open given file"));
```

```
    }

    QJsonParseError parseErr;
    QJsonDocument loadDoc(QJsonDocument::fromJson(ruleFile.readAll(), &parseErr));

    ruleFile.close();


    if (loadDoc.isNull() || loadDoc.isEmpty())
    {
        qWarning() << "Could not read data : ";
        qWarning() << parseErr.errorString();
        throw QString(parseErr.errorString());
    }

    if (!loadDoc.isArray())
    {
        qWarning() << "We need an array of rules !";
        throw QString(QObject::tr("We need an array of rules!"));
    }

    loadRules(loadDoc.array());
}

QList<const Rule *> generate1DRules(unsigned int automatonNumber)
{
    if (automatonNumber > 256) throw QString(QObject::tr("Automaton number not defined"));
    QList<const Rule*> ruleList;
    unsigned short int p = 128;
    int i = 7;
    while (i >= 0) {
        if (automatonNumber >= p)
        {
            ruleList.push_back((Rule*)getRuleFromNumber(i, 1));
            //numeroBit.push_back('1');
            automatonNumber -= p;
        }
        else {
            ruleList.push_back((Rule*)getRuleFromNumber(i, 0));
        }
        i--;
        p = p / 2;
    }

    return ruleList;
}

const MatrixRule* getRuleFromNumber(int previousConfiguration, int nextState)
{
    if (previousConfiguration > 7 || previousConfiguration < 0)
        throw QString(QObject::tr("Configuration not possible"));

    MatrixRule* newRule;
    switch(previousConfiguration)
    {
        case 0:
            newRule = new MatrixRule(nextState, {0});
            newRule->addNeighbourState(QVector<short>{-1}, 0);
            newRule->addNeighbourState(QVector<short>{1}, 0);
        break;
    case 1:
        newRule = new MatrixRule(nextState, {0});
        newRule->addNeighbourState(QVector<short>{-1}, 0);
        newRule->addNeighbourState(QVector<short>{1}, 1);
    break;
    case 2:
        newRule = new MatrixRule(nextState, {1});
        newRule->addNeighbourState(QVector<short>{-1}, 0);
        newRule->addNeighbourState(QVector<short>{1}, 0);
    break;
    case 3:
        newRule = new MatrixRule(nextState, {1});
        newRule->addNeighbourState(QVector<short>{-1}, 0);
        newRule->addNeighbourState(QVector<short>{1}, 1);
    break;
    case 4:
        newRule = new MatrixRule(nextState, {0});
        newRule->addNeighbourState(QVector<short>{-1}, 1);
        newRule->addNeighbourState(QVector<short>{1}, 0);
    break;
    case 5:
        newRule = new MatrixRule(nextState, {0});
        newRule->addNeighbourState(QVector<short>{-1}, 1);
        newRule->addNeighbourState(QVector<short>{1}, 1);
    break;
    case 6:
        newRule = new MatrixRule(nextState, {1});
```

```
            newRule->addNeighbourState(QVector<short>{-1}, 1);
            newRule->addNeighbourState(QVector<short>{1}, 0);
        break;
        case 7:
            newRule = new MatrixRule(nextState, {1});
            newRule->addNeighbourState(QVector<short>{-1}, 1);
            newRule->addNeighbourState(QVector<short>{1}, 1);
        break;
        }

        return newRule;
}

```

7.3 automate.h File Reference

```
#include <QVector>
#include <QList>
#include "cellhandler.h"
#include "rule.h"
#include "neighbourrule.h"
#include "matrixrule.h"
```

Classes

- class Automate

Functions

- QList< const Rule * > generate1DRules (unsigned int automatonNumber)
- const MatrixRule * getRuleFromNumber (int previousConfiguration, int nextState)

7.3.1 Function Documentation

7.3.1.1 generate1DRules()

```
QList<const Rule*> generate1DRules (
            unsigned int automatonNumber )
```

Definition at line 315 of file automate.cpp.

References getRuleFromNumber().

7.3.1.2 getRuleFromNumber()

```
const MatrixRule* getRuleFromNumber (
            int previousConfiguration,
            int nextState )
```

Definition at line 338 of file automate.cpp.

References MatrixRule::addNeighbourState().

Referenced by generate1DRules().

7.4 automate.h

```
#ifndef AUTOMATE_H
#define AUTOMATE_H
#include <QVector>
#include <QList>

#include "cellhandler.h"
#include "rule.h"
#include "neighbourrule.h"
#include "matrixrule.h"


class Automate
{
private:
    CellHandler* m_cellHandler = nullptr;
    QList<const Rule*> m_rules;
    friend class AutomateHandler;

    bool loadRules(const QJsonArray &json);
public:
    Automate(QString filename);
    Automate(const QVector<unsigned int> dimensions,
CellHandler::generationTypes type =
CellHandler::empty, unsigned int stateMax = 1, unsigned int density = 20);
    Automate(QString cellHandlerFilename, QString ruleFilename);
    virtual ~Automate();

    bool saveRules(QString filename) const ;
    bool saveCells(QString filename) const ;
    bool saveAll(QString cellHandlerFilename, QString rulesFilename)const ;

    void addRuleFile(QString filename);
    void addRule(const Rule* newRule);
    void setRulePriority(const Rule* rule, unsigned int newPlace);
    const QList<const Rule *> &getRules() const;



public:
    bool run(unsigned int nbSteps = 1);
    const CellHandler& getCellHandler() const;
};

QList<const Rule*> generate1DRules(unsigned int automatonNumber);
const MatrixRule *getRuleFromNumber(int previousConfiguration, int nextState);

#endif // AUTOMATE_H
```

7.5 automatehandler.cpp File Reference

```
#include "automatehandler.h"
```

7.6 automatehandler.cpp

```
#include "automatehandler.h"

AutomateHandler * AutomateHandler::m_activeAutomateHandler
 = nullptr;


AutomateHandler::AutomateHandler()
{

}


AutomateHandler::~AutomateHandler()
{
    while(!m_ActiveAutomates.empty())
        delete(m_ActiveAutomates.first());
}


AutomateHandler & AutomateHandler::getAutomateHandler()
{
    if (!m_activeAutomateHandler)
        m_activeAutomateHandler = new AutomateHandler;
    return *m_activeAutomateHandler;
}


void AutomateHandler::deleteAutomateHandler()
{
    if(m_activeAutomateHandler)
    {
        delete m_activeAutomateHandler;
        m_activeAutomateHandler = nullptr;
    }
}


Automate * AutomateHandler::getAutomate(unsigned int indexAutomate){
    if(indexAutomate > m_ActiveAutomates.size())
        return nullptr;
    return m_ActiveAutomates.at(indexAutomate);
}


unsigned int AutomateHandler::getNumberAutomates()const
{
    return m_ActiveAutomates.size();
}


void AutomateHandler::addAutomate(Automate * automate)
{
    m_ActiveAutomates.append(automate);
}


void AutomateHandler::deleteAutomate(Automate * automate)
{
    if(m_ActiveAutomates.contains(automate))
    {
        delete automate;
        m_ActiveAutomates.removeOne(automate);
    }
}
```

7.7 automatehandler.h File Reference

`#include "automate.h"`

Classes

- class AutomateHandler

    Implementation of singleton design pattern.

7.8 automatehandler.h

```
#ifndef AUTOMATEHANDLER_H
#define AUTOMATEHANDLER_H

#include "automate.h"


class AutomateHandler
{
private:
    QList<Automate*> m_ActiveAutomates;
    static AutomateHandler * m_activeAutomateHandler;

    AutomateHandler();
    AutomateHandler(const AutomateHandler & a) = delete;
    AutomateHandler & operator=(const AutomateHandler & a) = delete;
    ~AutomateHandler();

public:
    static AutomateHandler & getAutomateHandler();
    static void deleteAutomateHandler();

    Automate * getAutomate(unsigned int indexAutomate);
    unsigned int getNumberAutomates()const;

    void addAutomate(Automate * automate);
    void deleteAutomate(Automate * automate);
};


#endif // AUTOMATEHANDLER_H
```

7.9 cell.cpp File Reference

`#include "cell.h"`

7.10 cell.cpp

```
#include "cell.h"

Cell::Cell(unsigned int state):
    m_nextState(state)
{
    m_states.push(state);
}

void Cell::setState(unsigned int state)
{
    m_nextState = state;
}

void Cell::validState()
{
    m_states.push(m_nextState);
}

void Cell::forceState(unsigned int state)
{
    m_nextState = state;
    m_states.push(m_nextState);
}

unsigned int Cell::getState() const
{
    return m_states.top();
}

bool Cell::back()
{
    if (m_states.size() <= 1)
        return false;
```

```
    m_states.pop();
    m_nextState = m_states.top();
    return true;
}

void Cell::reset()
{
    while (m_states.size() > 2)
        m_states.pop();
    m_nextState = m_states.top();
}

bool Cell::addNeighbour(const Cell* neighbour, const QVector<short> relativePosition)
{
    if (m_neighbours.count(relativePosition))
        return false;

    m_neighbours.insert(relativePosition, neighbour);
    return true;
}

QMap<QVector<short>, const Cell *> Cell::getNeighbours() const
{
    return m_neighbours;
}

const Cell *Cell::getNeighbour(QVector<short> relativePosition) const
{
    return m_neighbours.value(relativePosition, nullptr);
}

unsigned int Cell::countNeighbours(unsigned int filterState) const
{
    unsigned int count = 0;
    for (QMap<QVector<short>, const Cell*>::const_iterator it = m_neighbours.begin(); it != m_neighbours.end(); ++it)
    {
        if ((*it)->getState() == filterState)
            count++;
    }
    return count;
}

unsigned int Cell::countNeighbours() const
{
    unsigned int count = 0;
    for (QMap<QVector<short>, const Cell*>::const_iterator it = m_neighbours.begin(); it != m_neighbours.end(); ++it)
    {
        if ((*it)->getState() != 0)
            count++;
    }
    return count;
}

QVector<short> Cell::getRelativePosition(const QVector<unsigned int> cellPosition, const QVector<unsigned int> neighbourPosition)
{
    if (cellPosition.size() != neighbourPosition.size())
    {
        throw QString(QObject::tr("Different size of position vectors"));
    }
    QVector<short> relativePosition;
    for (short i = 0; i < cellPosition.size(); i++)
        relativePosition.push_back(neighbourPosition.at(i) - cellPosition.at(i));

    return relativePosition;
}
```

7.11 cell.h File Reference

```
#include <QVector>
#include <QDebug>
#include <QStack>
```

Classes

- class Cell

Contains the state, the next state and the neighbours.

7.12 cell.h

```
#ifndef CELL_H
#define CELL_H

#include <QVector>
#include <QDebug>
#include <QStack>

class Cell
{
public:
    Cell(unsigned int state = 0);

    void setState(unsigned int state);
    void validState();
    void forceState(unsigned int state);
    unsigned int getState() const;

    bool back();
    void reset();

    bool addNeighbour(const Cell* neighbour, const QVector<short> relativePosition);
    QMap<QVector<short>, const Cell*> getNeighbours() const;
    const Cell* getNeighbour(QVector<short> relativePosition) const;

    unsigned int countNeighbours(unsigned int filterState) const;
    unsigned int countNeighbours() const;

    static QVector<short> getRelativePosition(const QVector<unsigned int> cellPosition,
const QVector<unsigned int> neighbourPosition);

private:
    QStack<unsigned int> m_states;
    unsigned int m_nextState;

    QMap<QVector<short>, const Cell*> m_neighbours;
};

#endif // CELL_H
```

7.13 cellhandler.cpp File Reference

```
#include <iostream>
#include "cellhandler.h"
```

7.14 cellhandler.cpp

```
#include <iostream>
#include "cellhandler.h"

CellHandler::CellHandler(const QString filename)
{
    QFile loadFile(filename);
    if (!loadFile.open(QIODevice::ReadOnly | QIODevice::Text)) {
        qWarning("Couldn't open given file.");
        throw QString(QObject::tr("Couldn't open given file"));
    }

    QJsonParseError parseErr;
    QJsonDocument loadDoc(QJsonDocument::fromJson(loadFile.readAll(), &parseErr));

    loadFile.close();


    if (loadDoc.isNull() || loadDoc.isEmpty()) {
```

```
        qWarning() << "Could not read data : ";
        qWarning() << parseErr.errorString();
        throw QString(parseErr.errorString());
    }

    // Loadding of the json file
    if (!load(loadDoc.object()))
    {
        qWarning("File not valid");
        throw QString(QObject::tr("File not valid"));
    }

    foundNeighbours();


}

CellHandler::CellHandler(const QJsonObject& json)
{
    if (!load(json))
    {
        qWarning("Json not valid");
        throw QString(QObject::tr("Json not valid"));
    }

    foundNeighbours();

}


CellHandler::CellHandler(const QVector<unsigned int> dimensions, generationTypes type, unsigned int stateMax, unsigned int density) :
    m_maxState(stateMax)
{
    m_dimensions = dimensions;
    QVector<unsigned int> position;
    unsigned int size = 1;

    // Set position vector to 0

    for (unsigned short i = 0; i < m_dimensions.size(); i++)
    {
        position.push_back(0);
        size *= m_dimensions.at(i);
    }


    // Creation of cells
    for (unsigned int j = 0; j < size; j++)
    {
        m_cells.insert(position, new Cell(0));

        positionIncrement(position);
    }

    foundNeighbours();

    if (type != empty)
        generate(type, stateMax, density);

}

CellHandler::~CellHandler()
{
    for (QMap<QVector<unsigned int>, Cell* >::iterator it = m_cells.begin(); it != m_cells.end(); ++it)
    {
        delete it.value();
    }
}

Cell *CellHandler::getCell(const QVector<unsigned int> position) const
{
    return m_cells.value(position);
}

unsigned int CellHandler::getMaxState() const
{
    return m_maxState;
}

QVector<unsigned int> CellHandler::getDimensions() const
{
    return m_dimensions;
}

void CellHandler::nextStates() const
```

```
{
    for (QMap<QVector<unsigned int>, Cell* >::const_iterator it = m_cells.begin(); it != m_cells.end(); ++it)
    {
        it.value()->validState();
    }
}

bool CellHandler::previousStates() const
{
    for (QMap<QVector<unsigned int>, Cell* >::const_iterator it = m_cells.begin(); it != m_cells.end(); ++it)
    {
        if (!it.value()->back())
            return false;
    }
    return true;
}

void CellHandler::reset() const
{
    for (QMap<QVector<unsigned int>, Cell* >::const_iterator it = m_cells.begin(); it != m_cells.end(); ++it)
    {
        it.value()->reset();
    }
}

bool CellHandler::save(QString filename) const
{
    QFile saveFile(filename);
    if (!saveFile.open(QIODevice::WriteOnly)) {
        qWarning("Couldn't create or open given file.");
        throw QString(QObject::tr("Couldn't create or open given file"));
    }

    QJsonObject json;
    QString stringDimension;
    // Creation of the dimension string
    for (int i = 0; i < m_dimensions.size(); i++)
    {
        if (i != 0)
            stringDimension.push_back("x");
        stringDimension.push_back(QString::number(m_dimensions.at(i)));
    }
    json["dimensions"] = QJsonValue(stringDimension);

    QJsonArray cells;
    for (CellHandler::const_iterator it = begin(); it != end(); ++it)
    {
        cells.append(QJsonValue((int)it->getState()));
    }
    json["cells"] = cells;

    json["maxState"] = QJsonValue((int)m_maxState);


    QJsonDocument saveDoc(json);
    saveFile.write(saveDoc.toJson());

    saveFile.close();
    return true;
}

void CellHandler::generate(CellHandler::generationTypes type, unsigned int stateMax, unsigned short density)
{
    if (type == random)
    {
        QVector<unsigned int> position;
        for (unsigned short i = 0; i < m_dimensions.size(); i++)
        {
            position.push_back(0);
        }
        QRandomGenerator generator((float)qrand()*(float)time_t()/RAND_MAX);
        for (int j = 0; j < m_cells.size(); j++)
        {
            unsigned int state = 0;
            // 0 have (1-density)% of chance of being generate
            if (generator.generateDouble()*100.0 < density)
                 state = (float)(generator.generateDouble()*stateMax) +1;
            if (state > stateMax)
                state = stateMax;
            m_cells.value(position)->forceState(state);

            positionIncrement(position);
```

```
        }
    }
    else if (type == symetric)
    {
        QVector<unsigned int> position;
        for (short i = 0; i < m_dimensions.size(); i++)
        {
            position.push_back(0);
        }

        QRandomGenerator generator((float)qrand()*(float)time_t()/RAND_MAX);
        QVector<unsigned int> savedStates;
        for (int j = 0; j < m_cells.size(); j++)
        {
            if (j % m_dimensions.at(0) == 0)
                savedStates.clear();
            if (j % m_dimensions.at(0) < (m_dimensions.at(0)+1) / 2)
            {
                unsigned int state = 0;
                // 0 have (1-density)% of chance of being generate
                if (generator.generateDouble()*100.0 < density)
                    state = (float)(generator.generateDouble()*stateMax) +1;
                if (state > stateMax)
                    state = stateMax;
                savedStates.push_back(state);
                m_cells.value(position)->forceState(state);
            }
            else
            {
                unsigned int i = savedStates.size() - (j % m_dimensions.at(0) - (m_dimensions.at(0)-1)/2 + (m_dimensions.at(0) % 2 == 0 ? 0 : 1));
                m_cells.value(position)->forceState(savedStates.at(i));
            }
            positionIncrement(position);


        }

    }
}

void CellHandler::print(std::ostream &stream) const
{
    for (const_iterator it = begin(); it != end(); ++it)
    {
        for (unsigned int d = 0; d < it.changedDimension(); d++)
            stream << std::endl;
        stream << it->getState() << " ";

    }

}

CellHandler::iterator CellHandler::begin()
{
    return iterator(this);
}

CellHandler::const_iterator CellHandler::begin() const
{
    return const_iterator(this);
}

bool CellHandler::end() const
{
    return true;
}

bool CellHandler::load(const QJsonObject &json)
{
    if (!json.contains("dimensions") || !json["dimensions"].isString())
        return false;

    // RegExp to validate dimensions field format : "10x10"
    QRegExpValidator dimensionValidator(QRegExp("([0-9]*x?)*"));
    QString stringDimensions = json["dimensions"].toString();
    int pos= 0;
    if (dimensionValidator.validate(stringDimensions, pos) != QRegExpValidator::Acceptable)
        return false;

    // Split of dimensions field : "10x10" => "10", "10"
    QRegExp rx("x");
    QStringList list = json["dimensions"].toString().split(rx, QString::SkipEmptyParts);

    int product = 1;
    // Dimensions construction
    for (int i = 0; i < list.size(); i++)
```

```
    {
        product = product * list.at(i).toInt();
        m_dimensions.push_back(list.at(i).toInt());
    }
    if (!json.contains("cells") || !json["cells"].isArray())
        return false;

    QJsonArray cells = json["cells"].toArray();
    if (cells.size() != product)
        return false;

    QVector<unsigned int> position;
    // Set position vector to 0
    for (unsigned short i = 0; i < m_dimensions.size(); i++)
    {
        position.push_back(0);
    }


    // Creation of cells
    for (int j = 0; j < cells.size(); j++)
    {
        if (!cells.at(j).isDouble())
            return false;
        if (cells.at(j).toDouble() < 0)
            return false;
        m_cells.insert(position, new Cell(cells.at(j).toDouble()));

        positionIncrement(position);
    }

    if (!json.contains("maxState") || !json["maxState"].isDouble())
        return false;
    m_maxState = json["maxState"].toInt();

    return true;

}

void CellHandler::foundNeighbours()
{
    QVector<unsigned int> currentPosition;
    // Set position vector to 0
    for (unsigned short i = 0; i < m_dimensions.size(); i++)
    {
        currentPosition.push_back(0);
    }
    // Modification of all the cells
    for (int j = 0; j < m_cells.size(); j++)
    {
        // Get the list of the neighbours positions
        // This function is recursive
        QVector<QVector<unsigned int> > listPosition(getListNeighboursPositions(currentPosition));

        // Adding neighbours
        for (int i = 0; i < listPosition.size(); i++)
            m_cells.value(currentPosition)->addNeighbour(m_cells.value(listPosition.at(i)), Cell::getRelativePosition(currentPosition, listPosition.at(i)));
        positionIncrement(currentPosition);
    }

}

void CellHandler::positionIncrement(QVector<unsigned int> &pos, unsigned int value) const
{
    pos.replace(0, pos.at(0) + value); // adding the value to the first digit

    // Carry management
    for (unsigned short i = 0; i < m_dimensions.size(); i++)
    {
        if (pos.at(i) >= m_dimensions.at(i) && pos.at(i) < m_dimensions.at(i)*2)
        {
            pos.replace(i, 0);
            if (i + 1 != m_dimensions.size())
                pos.replace(i+1, pos.at(i+1)+1);
        }
        else if (pos.at(i) >= m_dimensions.at(i))
        {
            pos.replace(i, pos.at(i) - m_dimensions.at(i));
            if (i + 1 != m_dimensions.size())
                pos.replace(i+1, pos.at(i+1)+1);
            i--;
        }

```

```
    }
}

QVector<QVector<unsigned int> >& CellHandler::getListNeighboursPositions(const QVector<unsigned int> position) const
{
    QVector<QVector<unsigned int> > *list = getListNeighboursPositionsRecursive(position, position.size(), position);
    // We remove the position of the cell
    list->removeAll(position);
    return *list;
}

QVector<QVector<unsigned int> >* CellHandler::getListNeighboursPositionsRecursive(const QVector<unsigned int> position, unsigned int dimension, QVector<unsigned int> lastAdd) const
{
    if (dimension == 0) // Stop condition
    {
        QVector<QVector<unsigned int> > *list = new QVector<QVector<unsigned int> >;
        return list;
    }
    QVector<QVector<unsigned int> > *listPositions = new QVector<QVector<unsigned int> >;

    QVector<unsigned int> modifiedPosition(lastAdd);

    // "x_d - 1" tree
    if (modifiedPosition.at(dimension-1) != 0) // Avoid "negative" position
        modifiedPosition.replace(dimension-1, position.at(dimension-1) - 1);
    listPositions->append(*getListNeighboursPositionsRecursive(position, dimension -1, modifiedPosition));
    if (!listPositions->count(modifiedPosition))
        listPositions->push_back(modifiedPosition);

    // "x_d" tree
    modifiedPosition.replace(dimension-1, position.at(dimension-1));
    listPositions->append(*getListNeighboursPositionsRecursive(position, dimension -1, modifiedPosition));
    if (!listPositions->count(modifiedPosition))
        listPositions->push_back(modifiedPosition);

    // "x_d + 1" tree
    if (modifiedPosition.at(dimension -1) + 1 < m_dimensions.at(dimension-1)) // Avoid position out of the cell space
        modifiedPosition.replace(dimension-1, position.at(dimension-1) +1);
    listPositions->append(*getListNeighboursPositionsRecursive(position, dimension -1, modifiedPosition));
    if (!listPositions->count(modifiedPosition))
        listPositions->push_back(modifiedPosition);

    return listPositions;

}

template<typename CellHandler_T, typename Cell_T>
CellHandler::iteratorT<CellHandler_T,Cell_T>::iteratorT(CellHandler_T *handler):
        m_handler(handler), m_changedDimension(0)
{
    // Initialisation of m_position
    for (unsigned short i = 0; i < handler->m_dimensions.size(); i++)
    {
        m_position.push_back(0);
    }
    m_zero = m_position;
}
```

7.15 cellhandler.h File Reference

```
#include <QString>
#include <QFile>
#include <QJsonDocument>
#include <QtWidgets>
#include <QMap>
#include <QRegExpValidator>
#include <QDebug>
#include "cell.h"
```

Classes

- class CellHandler

  Cell container and cell generator.
- class CellHandler::iteratorT< CellHandler_T, Cell_T >

  Implementation of iterator design pattern with a template to generate iterator and const_iterator at the same time.

7.16 cellhandler.h

```
#ifndef CELLHANDLER_H
#define CELLHANDLER_H

#include <QString>
#include <QFile>
#include <QJsonDocument>
#include <QtWidgets>
#include <QMap>
#include <QRegExpValidator>
#include <QDebug>

#include "cell.h"



class CellHandler
{

    template <typename CellHandler_T, typename Cell_T>
    class iteratorT
    {
        friend class CellHandler;
    public:
        iteratorT(CellHandler_T* handler);
        iteratorT& operator++(){
            m_position.replace(0, m_position.at(0) + 1); // adding the value to the first digit

            m_changedDimension = 0;
            // Carry management
            for (unsigned short i = 0; i < m_handler->m_dimensions.size(); i++)
            {
                if (m_position.at(i) >= m_handler->m_dimensions.at(i))
                {
                    m_position.replace(i, 0);
                    m_changedDimension++;
                    if (i + 1 != m_handler->m_dimensions.size())
                        m_position.replace(i+1, m_position.at(i+1)+1);
                }

            }
            // If we return to zero, we have finished
            if (m_position == m_zero)
                m_finished = true;

            return *this;

        }
        Cell_T* operator->() const{
            return m_handler->m_cells.value(m_position);
        }
        Cell_T* operator*() const{
            return m_handler->m_cells.value(m_position);
        }

        bool operator!=(bool finished) const { return (m_finished != finished); }
        unsigned int changedDimension() const{
            return m_changedDimension;
        }



    private:
        CellHandler_T *m_handler;
        QVector<unsigned int> m_position;
        bool m_finished = false;
        QVector<unsigned int> m_zero;
        unsigned int m_changedDimension;
    };
```

```
public:
    typedef iteratorT<const CellHandler, const Cell> const_iterator;
    typedef iteratorT<CellHandler, Cell> iterator;

    enum generationTypes {
        empty,
        random,
        symetric
    };

    CellHandler(const QString filename);
    CellHandler(const QJsonObject &json);
    CellHandler(const QVector<unsigned int> dimensions, generationTypes type = empty, unsigned int stateMax = 1, unsigned int density = 20);
    virtual ~CellHandler();

    Cell* getCell(const QVector<unsigned int> position) const;
    unsigned int getMaxState() const;
    QVector<unsigned int> getDimensions() const;
    void nextStates() const;
    bool previousStates() const;
    void reset() const;

    bool save(QString filename) const;

    void generate(generationTypes type, unsigned int stateMax = 1, unsigned short density = 50);
    void print(std::ostream &stream) const;

    const_iterator begin() const;
    iterator begin();
    bool end() const;


private:
    bool load(const QJsonObject &json);
    void foundNeighbours();
    void positionIncrement(QVector<unsigned int> &pos, unsigned int value = 1) const;
    QVector<QVector<unsigned int> > *getListNeighboursPositionsRecursive(const QVector<unsigned int> position, unsigned int dimension, QVector<unsigned int> lastAdd) const;
    QVector<QVector<unsigned int> > &getListNeighboursPositions(const QVector<unsigned int> position) const;

    QVector<unsigned int> m_dimensions;
    QMap<QVector<unsigned int>, Cell* > m_cells;
    unsigned int m_maxState;
};

template class CellHandler::iteratorT<CellHandler, Cell>;
template class CellHandler::iteratorT<const CellHandler, const Cell>;

#endif // CELLHANDLER_H
```

7.17 creationdialog.cpp File Reference

```
#include "creationdialog.h"
#include <iostream>
```

7.18 creationdialog.cpp

```
#include "creationdialog.h"
#include <iostream>


CreationDialog::CreationDialog(QWidget *parent)
{
    QLabel *m_dimLabel= new QLabel(tr("Write your dimensions and their size, separated by a comma.\n"
                        "For instance, '25,25 ' will create a 2-dimensional 25x25 Automaton. "));
    QLabel *m_densityLabel = new QLabel(tr("Density :"));
    QLabel *m_stateMaxLabel = new QLabel(tr("Max state :"));
    m_densityBox = new QSpinBox();
    m_densityBox->setValue(20);
```

```
    m_stateMaxBox = new QSpinBox();
    m_stateMaxBox->setValue(1);

    QHBoxLayout *densityLayout = new QHBoxLayout();
    densityLayout->addWidget(m_densityLabel);
    densityLayout->addWidget(m_densityBox);

    QHBoxLayout *stateMaxLayout = new QHBoxLayout();
    stateMaxLayout->addWidget(m_stateMaxLabel);
    stateMaxLayout->addWidget(m_stateMaxBox);

    m_dimensionsEdit = new QLineEdit;
    QRegExp rgx("([0-9]+,)*");
    QRegExpValidator *v = new QRegExpValidator(rgx, this);
    m_dimensionsEdit->setValidator(v);
    m_doneBt = new QPushButton(tr("Done !"));

    QVBoxLayout *layout = new QVBoxLayout;

    QGroupBox *grpBox = createGenButtons();

    layout->addWidget(m_dimLabel);
    layout->addWidget(m_dimensionsEdit);
    layout->addLayout(densityLayout);
    layout->addLayout(stateMaxLayout);
    layout->addWidget(grpBox);
    layout->addWidget(m_doneBt);
    setLayout(layout);

    connect(m_doneBt, SIGNAL(clicked(bool)), this, SLOT(processSettings()));

}

QGroupBox *CreationDialog::createGenButtons(){
    m_groupBox = new QGroupBox(tr("Cell generation settings"));
    m_empGen = new QRadioButton(tr("&Empty Board"));
    m_randGen = new QRadioButton(tr("&Random"));
    m_symGen = new QRadioButton(tr("&Symmetrical"));

    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(m_empGen);
    layout->addWidget(m_randGen);
    layout->addWidget(m_symGen);

    m_groupBox->setLayout(layout);

    return m_groupBox;
}

void CreationDialog::processSettings(){
    QString dimensions = m_dimensionsEdit->text();
    if(dimensions.length() == 0){
        QMessageBox messageBox;
        messageBox.critical(0,"Error","You must specify valid dimensions !");
        messageBox.setFixedSize(500,200);
    }
    else{
        CellHandler::generationTypes genType;
        if(m_symGen->isChecked()) genType = CellHandler::generationTypes::symetric;
        else if(m_randGen->isChecked()) genType = CellHandler::generationTypes::random;
        else genType = CellHandler::generationTypes::empty;
        QStringList dimList = m_dimensionsEdit->text().split(",");
        QVector<unsigned int> dimensions;
        for(int i = 0; i < dimList.size(); i++) dimensions.append(dimList.at(i).toInt());

        emit settingsFilled(dimensions, genType, m_stateMaxBox->value(),
m_densityBox->value());
        this->close();
    }

}

```

7.19 creationdialog.h File Reference

```
#include <QtWidgets>
#include "cellhandler.h"
```

Classes

- class CreationDialog

  Automaton creation dialog box.

7.20 creationdialog.h

```
#ifndef CREATIONDIALOG_H
#define CREATIONDIALOG_H

#include <QtWidgets>
#include "cellhandler.h"

class CreationDialog : public QDialog
{
    Q_OBJECT

public:
    CreationDialog(QWidget *parent = 0);

signals:
    void settingsFilled(const QVector<unsigned int> dimensions,
                        CellHandler::generationTypes type = CellHandler::generationTypes::empty,
                        unsigned int stateMax = 1, unsigned int density = 20);

public slots:
    void processSettings();

private:
    QLineEdit *m_dimensionsEdit;
    QSpinBox *m_densityBox;
    QSpinBox *m_stateMaxBox;
    QPushButton *m_doneBt;

    QGroupBox *m_groupBox;
    QRadioButton *m_empGen;
    QRadioButton *m_randGen;
    QRadioButton *m_symGen;

    QGroupBox *createGenButtons();






};

#endif // CREATEDIALOG_H
```

7.21 main.cpp File Reference

```
#include <QApplication>
#include <QDebug>
#include "cell.h"
#include "mainwindow.h"
#include "ruleeditor.h"
```

Functions

- int main (int argc, char *argv[ ])

7.21.1 Function Documentation

7.21.1.1 main()

```
int main (
            int argc,
            char * argv[] )
```

Definition at line 7 of file main.cpp.

7.22 main.cpp

```
00001 #include <QApplication>
00002 #include <QDebug>
00003 #include "cell.h"
00004 #include "mainwindow.h"
00005 #include "ruleeditor.h"
00006
00007 int main(int argc, char * argv[])
00008 {
00009     QApplication app(argc, argv);
00010     QApplication::setAttribute(Qt::AA_UseHighDpiPixmaps);
00011     MainWindow w;
00012     w.show();
00013
00014     return app.exec();
00015
00016 }
```

7.23 mainwindow.cpp File Reference

```
#include "mainwindow.h"
#include <iostream>
#include "math.h"
```

7.24 mainwindow.cpp

```
#include "mainwindow.h"
#include <iostream>
#include "math.h"
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent)
{
    createIcons();
    createActions();
    createToolBar();


    setMinimumSize(500,500);
    setWindowTitle("AutoCell");

    m_tabs = NULL;
    running = false;
}

void MainWindow::createIcons(){
    QPixmap fastBackwardPm(":/icons/icons/fast-backward.svg");
    QPixmap fastBackwardHoveredPm(":/icons/icons/fast-backward-full.svg");
```

```
    QPixmap fastForwardPm(":/icons/icons/fast-forward.svg");
    QPixmap fastForwardHoveredPm(":/icons/icons/fast-forward-full.svg");
    QPixmap playPm(":/icons/icons/play.svg");
    QPixmap playHoveredPm(":/icons/icons/play-full.svg");
    QPixmap newPm(":/icons/icons/new.svg");
    QPixmap openPm(":/icons/icons/open.svg");
    QPixmap savePm(":/icons/icons/save.svg");
    QPixmap pausePm(":/icons/icons/pause.svg");
    QPixmap resetPm(":/icons/icons/reset.svg");

    m_fastBackwardIcon.addPixmap(fastBackwardPm, QIcon::Normal, QIcon::Off);
    m_fastBackwardIcon.addPixmap(fastBackwardHoveredPm, QIcon::Active, QIcon::Off);
    m_fastForwardIcon.addPixmap(fastForwardPm, QIcon::Normal, QIcon::Off);
    m_fastForwardIcon.addPixmap(fastForwardHoveredPm, QIcon::Active, QIcon::Off);
    m_playIcon.addPixmap(playPm, QIcon::Normal, QIcon::Off);
    m_playIcon.addPixmap(playHoveredPm, QIcon::Active, QIcon::Off);
    m_pauseIcon.addPixmap(pausePm, QIcon::Normal, QIcon::Off);
    m_newIcon.addPixmap(newPm, QIcon::Normal, QIcon::Off);
    m_saveIcon.addPixmap(savePm, QIcon::Normal, QIcon::Off);
    m_openIcon.addPixmap(openPm, QIcon::Normal, QIcon::Off);
    m_resetIcon.addPixmap(resetPm, QIcon::Normal, QIcon::Off);
}

void MainWindow::createActions(){
    m_fastForward = new QAction(m_fastForwardIcon, tr("&fast forward"), this);
    m_playPause = new QAction(m_playIcon, tr("Play"), this);
    m_saveAutomaton = new QAction(m_saveIcon, tr("Save automaton"), this);
    m_newAutomaton = new QAction(m_newIcon, tr("New automaton"), this);
    m_openAutomaton = new QAction(m_openIcon, tr("Open automaton"), this);
    m_resetAutomaton = new QAction(m_resetIcon, tr("Reset automaton"), this);


    m_fastForwardBt = new QToolButton(this);
    m_playPauseBt = new QToolButton(this);
    m_saveAutomatonBt = new QToolButton(this);
    m_newAutomatonBt = new QToolButton(this);
    m_openAutomatonBt = new QToolButton(this);
    m_resetBt = new QToolButton(this);

    m_fastForwardBt->setDefaultAction(m_fastForward);
    m_playPauseBt->setDefaultAction(m_playPause);
    m_saveAutomatonBt->setDefaultAction(m_saveAutomaton);
    m_newAutomatonBt->setDefaultAction(m_newAutomaton);
    m_openAutomatonBt->setDefaultAction(m_openAutomaton);
    m_resetBt->setDefaultAction(m_resetAutomaton);

    m_fastForwardBt->setIconSize(QSize(30,30));
    m_playPauseBt->setIconSize(QSize(30,30));
    m_saveAutomatonBt->setIconSize(QSize(30,30));
    m_newAutomatonBt->setIconSize(QSize(30,30));
    m_openAutomatonBt->setIconSize(QSize(30,30));
    m_resetBt->setIconSize(QSize(30,30));

    connect(m_openAutomatonBt, SIGNAL(clicked(bool)), this, SLOT(openFile()));
    connect(m_newAutomatonBt, SIGNAL(clicked(bool)), this, SLOT(openCreationWindow()));
    connect(m_saveAutomatonBt, SIGNAL(clicked(bool)), this, SLOT(saveToFile()));
    connect(m_fastForwardBt, SIGNAL(clicked(bool)), this, SLOT(forward()));
    connect(m_playPauseBt, SIGNAL(clicked(bool)), this, SLOT(handlePlayPause()));
    connect(m_resetBt,SIGNAL(clicked(bool)), this,SLOT(reset()));

}

void MainWindow::createToolBar(){
    m_toolBar = new QToolBar(this);
    QLabel *timeStepLabel = new QLabel(tr("Timestep(ms)"),this);
    m_timeStep = new QSpinBox(this);
    m_timeStep->setMaximum(10000);
    m_timeStep->setValue(500);
    timeStepLabel->setFixedWidth(90);
    m_timeStep->setFixedWidth(60);
    m_toolBar->setMovable(false);

    QVBoxLayout* tsLayout = new QVBoxLayout();
    tsLayout->addWidget(timeStepLabel, Qt::AlignCenter);
    tsLayout->addWidget(m_timeStep, Qt::AlignCenter);

    QHBoxLayout *tbLayout = new QHBoxLayout(this);
    tbLayout->addWidget(m_newAutomatonBt, Qt::AlignCenter);
    tbLayout->addWidget(m_openAutomatonBt, Qt::AlignCenter);
    tbLayout->addWidget(m_saveAutomatonBt, Qt::AlignCenter);
    tbLayout->addWidget(m_resetBt, Qt::AlignCenter);
```

```
    tbLayout->addWidget(m_playPauseBt, Qt::AlignCenter);
    tbLayout->addWidget(m_fastForwardBt, Qt::AlignCenter);
    tbLayout->addLayout(tsLayout);
    tbLayout->setAlignment(Qt::AlignCenter);
    QWidget* wrapper = new QWidget(this);
    wrapper->setLayout(tbLayout);
    m_toolBar->addWidget(wrapper);
    addToolBar(m_toolBar);
}

QWidget* MainWindow::createTab(){
    QWidget *tab = new QWidget(this);
    QVBoxLayout *layout = new QVBoxLayout(this);
    QVector<unsigned int> dimensions = AutomateHandler::getAutomateHandler().getAutomate(AutomateHandler::getAutomateHandler().getNumberAutomates()-1)->getCellHandler().getDimensions();
    int boardVSize = 0;
    int boardHSize = 0;
    if(dimensions.size() > 1){
        boardVSize = dimensions[0];
        boardHSize = dimensions[1];
    }
    else{
        boardVSize = 1;
        boardHSize = dimensions[0];
    }

    QTableWidget* board = new QTableWidget(boardVSize, boardHSize, this);
        board->setFixedSize(boardHSize*m_cellSize,boardVSize*m_cellSize);
        //setMinimumSize(m_boardHSize*m_cellSize,100+m_boardVSize*m_cellSize);
        board->horizontalHeader()->setVisible(false);
        board->verticalHeader()->setVisible(false);
        board->setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
        board->setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
        board->setEditTriggers(QAbstractItemView::NoEditTriggers);
        for(unsigned int col = 0; col < boardHSize; ++col)
            board->setColumnWidth(col, m_cellSize);
        for(unsigned int row = 0; row < boardVSize; ++row) {
            board->setRowHeight(row, m_cellSize);
            for(unsigned int col = 0; col < boardHSize; ++col) {
                board->setItem(row, col, new QTableWidgetItem(""));
                board->item(row, col)->setBackgroundColor("white");
                board->item(row, col)->setTextColor("black");
            }
        }
     QScrollArea *scrollArea = new QScrollArea(this);
     scrollArea->setWidget(board);
     layout->setContentsMargins(0,0,0,0);
     layout->addWidget(scrollArea);
     tab->setLayout(layout);
     return tab;
}

void MainWindow::openFile(){
    QString fileName = QFileDialog::getOpenFileName(this, tr("Open Cell file"), ".",
                                                    tr("Automaton cell files (*.atc)"));
    if(!fileName.isEmpty()){
        AutomateHandler::getAutomateHandler().addAutomate(new Automate(fileName));
        if(m_tabs == NULL) createTabs();
        m_tabs->addTab(createTab(), "Automaton "+ QString::number(AutomateHandler::getAutomateHandler().getNumberAutomates()+1));
        updateBoard(AutomateHandler::getAutomateHandler().getNumberAutomates()-1);

        RuleEditor* ruleEditor = new RuleEditor();
        connect(ruleEditor, SIGNAL(fileImported(QString)),this,SLOT(addAutomatonRuleFile(QString)));
        connect(ruleEditor, SIGNAL(rulesFilled(QList<const NeighbourRule*>)), this, SLOT(addAutomatonRules(QList<const NeighbourRule*>)));
        ruleEditor->show();
    }
}

void MainWindow::saveToFile(){
    if(AutomateHandler::getAutomateHandler().getNumberAutomates() > 0){
        QString fileName = QFileDialog::getSaveFileName(this, tr("Save Automaton cell configuration"),
                                                        ".", tr("Automaton Cells file (*.atc"));
        AutomateHandler::getAutomateHandler().
```

```
getAutomate(m_tabs->currentIndex())->saveCells(fileName+".atc");

    }
    else{
        QMessageBox msgBox;
        msgBox.critical(0,"Error","Please create or import an Automaton first !");
        msgBox.setFixedSize(500,200);
    }
}

void MainWindow::openCreationWindow(){
    CreationDialog *window = new CreationDialog(this);
    connect(window, SIGNAL(settingsFilled(QVector<uint>, CellHandler::generationTypes,uint,uint)),
                this, SLOT(receiveCellHandler(QVector<uint>, CellHandler::generationTypes,uint,uint)));
    window->show();
}

void MainWindow::receiveCellHandler(const QVector<unsigned int> dimensions,
                                    CellHandler::generationTypes type,
                                    unsigned int stateMax, unsigned int density){
    AutomateHandler::getAutomateHandler().addAutomate(new Automate(dimensions, type, stateMax, density));

    if(m_tabs == NULL) createTabs();
    QWidget* newTab = createTab();
    m_tabs->addTab(newTab, "Automaton "+ QString::number( AutomateHandler::getAutomateHandler().getNumberAutomates()));
    m_tabs->setCurrentWidget(newTab);
    updateBoard(AutomateHandler::getAutomateHandler().getNumberAutomates()-1);

    RuleEditor* ruleEditor = new RuleEditor();
    connect(ruleEditor, SIGNAL(fileImported(QString)),this,SLOT( addAutomatonRuleFile(QString)));
    connect(ruleEditor, SIGNAL(rulesFilled(QList<const Rule*>)), this, SLOT( addAutomatonRules(QList<const Rule*>)));
    ruleEditor->show();

}

void MainWindow::nextState(unsigned int n){
    if(AutomateHandler::getAutomateHandler().getNumberAutomates()== 0){
        QMessageBox msgBox;
        msgBox.critical(0,"Error","Please create or import an Automaton first !");
        msgBox.setFixedSize(500,200);
    }
    else{

        AutomateHandler::getAutomateHandler().getAutomate(m_tabs->currentIndex())->run(n);
        updateBoard(m_tabs->currentIndex());
    }
}

void MainWindow::updateBoard(int index){
    if(AutomateHandler::getAutomateHandler().getNumberAutomates()== 0){
        QMessageBox msgBox;
        msgBox.critical(0,"Error","Please create or import an Automaton first !");
        msgBox.setFixedSize(500,200);
    }
    else{

        const CellHandler* cellHandler = &( AutomateHandler::getAutomateHandler().getAutomate(index)->getCellHandler());
        QVector<unsigned int> dimensions = cellHandler->getDimensions();
        QTableWidget* board = getBoard(index);
        if(dimensions.size() > 1){
            int i = 0;
            int j = 0;
            for (CellHandler::const_iterator it = cellHandler->begin(); it != cellHandler->end() && it.changedDimension() < 2; ++it){
                if(it.changedDimension() > 0){
                    i = 0;
                    j++;
                }
                board->item(i,j)->setBackgroundColor(QColor::colorNames().at(it->getState()));
                i++;
            }
        }
        else{ // dimension = 1
            if (board->rowCount() != 1)
                addEmptyRow(index);
            int i = board->rowCount() -1;
            int j = 0;
```

```
            for (CellHandler::const_iterator it = cellHandler->begin(); it != cellHandler->end() && it.changedDimension() < 1; ++it){
                board->item(i,j)->setBackgroundColor(QColor::colorNames().at(it->getState()));
                j++;
            }
            if (board->rowCount() == 1)
                addEmptyRow(index);

        }

    }

}

void MainWindow::forward(){
    //nextState(m_timeStep->value());
    nextState(1);
}

QTableWidget* MainWindow::getBoard(int n){
    return m_tabs->widget(n)->findChild<QTableWidget *>();
}

void MainWindow::createTabs(){
    m_tabs = new QTabWidget(this);
    m_tabs->setMovable(true);
    m_tabs->setTabsClosable(true);
    setCentralWidget(m_tabs);
    connect(m_tabs, SIGNAL(tabCloseRequested(int)), this, SLOT(closeTab(int)));
}

void MainWindow::addEmptyRow(unsigned int n)
{
    QTableWidget *board = getBoard(n);
    board->setFixedHeight(board->height() + m_cellSize);
    unsigned int row = board->rowCount();
    board->insertRow(row);
    board->setRowHeight(row, m_cellSize);
    for(unsigned int col = 0; col < board->columnCount(); ++col) {
        board->setItem(row, col, new QTableWidgetItem(""));
        board->item(row, col)->setBackgroundColor("white");
        board->item(row, col)->setTextColor("black");
    }
}

void MainWindow::closeTab(int n){
    m_tabs->setCurrentIndex(n);
    saveToFile();
    AutomateHandler::getAutomateHandler().deleteAutomate(AutomateHandler::getAutomateHandler().getAutomate(n));
    m_tabs->removeTab(n);
}

void MainWindow::addAutomatonRules(QList<const Rule *> rules){
    for(int i =0 ; i < rules.size();i++)
    {
        AutomateHandler::getAutomateHandler().getAutomate(AutomateHandler::getAutomateHandler().getNumberAutomates()-1)->addRule(rules.at(i));
    }
}

void MainWindow::addAutomatonRuleFile(QString path){
    AutomateHandler::getAutomateHandler().getAutomate(AutomateHandler::getAutomateHandler().getNumberAutomates()-1)->addRuleFile(path);
}

void MainWindow::handlePlayPause(){
    if(AutomateHandler::getAutomateHandler().getNumberAutomates()== 0){
        QMessageBox msgBox;
        msgBox.critical(0,"Error","Please create or import an Automaton first !");
        msgBox.setFixedSize(500,200);
    }
    else{
        if(running){
            m_playPauseBt->setIcon(m_playIcon);
            delete m_timer;
        }
        else {
            m_playPauseBt->setIcon(m_pauseIcon);
            m_timer = new QTimer(this);
            connect(m_timer, SIGNAL(timeout()), this, SLOT(runAutomaton()));
            m_timer->start(m_timeStep->value());
        }
        running = !running;
```

```
    }


}

void MainWindow::runAutomaton(){
    if(running){
        AutomateHandler::getAutomateHandler().getAutomate(m_tabs->currentIndex())->run();
        QCoreApplication::processEvents();
        updateBoard(m_tabs->currentIndex());
        QCoreApplication::processEvents();
    }
}

void MainWindow::reset(){
    if(AutomateHandler::getAutomateHandler().getNumberAutomates()== 0){
        QMessageBox msgBox;
        msgBox.critical(0,"Error","Please create or import an Automaton first !");
        msgBox.setFixedSize(500,200);
    }
    else{
        QTableWidget *board = getBoard(m_tabs->currentIndex());
        board->setRowCount(1);
        board->setFixedHeight(m_cellSize);

        AutomateHandler::getAutomateHandler().getAutomate(m_tabs->currentIndex())->getCellHandler().reset();
        updateBoard(m_tabs->currentIndex());
    }
}
```

7.25 mainwindow.h File Reference

```
#include <QMainWindow>
#include <QtWidgets>
#include "cellhandler.h"
#include "automate.h"
#include "creationdialog.h"
#include "automatehandler.h"
#include "ruleeditor.h"
```

Classes

- class MainWindow

    Simulation window.

7.26 mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QtWidgets>
#include "cellhandler.h"
#include "automate.h"
#include "creationdialog.h"
#include "automatehandler.h"
#include "ruleeditor.h"

class MainWindow : public QMainWindow
{
    Q_OBJECT

    QTabWidget *m_tabs; //Tabs for the main window
    //QVector<Automate *> m_automatons; //QVector containing a pointer to each tab's Automaton
```

```
    QIcon m_fastBackwardIcon;
    QIcon m_fastForwardIcon;
    QIcon m_playIcon;
    QIcon m_pauseIcon;
    QIcon m_newIcon;
    QIcon m_saveIcon;
    QIcon m_openIcon;
    QIcon m_resetIcon;

    QAction *m_playPause;
    QAction *m_nextState;
    QAction *m_previousState;
    QAction *m_fastForward;
    QAction *m_fastBackward;
    QAction *m_openAutomaton;
    QAction *m_saveAutomaton;
    QAction *m_newAutomaton;
    QAction *m_resetAutomaton;

    QToolButton *m_playPauseBt;
    QToolButton *m_nextStateBt;
    QToolButton *m_previousStateBt;
    QToolButton *m_fastForwardBt;
    QToolButton *m_fastBackwardBt;
    QToolButton *m_openAutomatonBt;
    QToolButton *m_saveAutomatonBt;
    QToolButton *m_newAutomatonBt;
    QToolButton *m_resetBt;


    QSpinBox *m_timeStep;
    QTimer* m_timer;

    Automate* m_newAutomate;
    bool running;
    QToolBar *m_toolBar;

    unsigned int m_boardHSize = 25;
    unsigned int m_boardVSize = 25;
    unsigned int m_cellSize = 30;

    void createIcons();
    void createActions();
    void createToolBar();
    void createBoard();
    QWidget* createTab();
    void createTabs();

    void addEmptyRow(unsigned int n);
    void updateBoard(int index);
    void nextState(unsigned int n);
    QTableWidget* getBoard(int n);


public:
    explicit MainWindow(QWidget *parent = nullptr);


signals:

public slots:
    void openFile();
    void saveToFile();
    void openCreationWindow();
    void receiveCellHandler(const QVector<unsigned int> dimensions,
                            CellHandler::generationTypes type =
CellHandler::generationTypes::empty,
                            unsigned int stateMax = 1, unsigned int density = 20);
    void addAutomatonRules(QList<const Rule *> rules);
    void addAutomatonRuleFile(QString path);
    void forward();
    void closeTab(int n);
    void runAutomaton();
    void handlePlayPause();
    void reset();

};

#endif // MAINWINDOW_H
```

7.27 matrixrule.cpp File Reference

```
#include "matrixrule.h"
```

Functions

- QVector< unsigned int > fillInterval (unsigned int min, unsigned int max)
  Returns a vector fill of the integers between min and max (all included)

7.27.1 Function Documentation

7.27.1.1 fillInterval()

```
QVector<unsigned int> fillInterval (
            unsigned int min,
            unsigned int max )
```

Returns a vector fill of the integers between min and max (all included)

Returns

Interval

Parameters

min	Minimal value (included)
max	Maximal value (included)

Definition at line 8 of file matrixrule.cpp.

7.28 matrixrule.cpp

```
00001 #include "matrixrule.h"
00002
00008 QVector<unsigned int> fillInterval(unsigned int min, unsigned int max)
00009 {
00010     QVector<unsigned int> interval;
00011     for (unsigned int i = min; i <= max ; i++)
00012         interval.push_back(i);
00013
00014     return interval;
00015 }
00016
00021 MatrixRule::MatrixRule(unsigned int finalState, QVector<unsigned int> currentStates)
      :
00022     Rule(currentStates, finalState)
00023 {
00024 }
00025
```

```
bool MatrixRule::matchCell(const Cell *cell) const
{
    // Check cell state
    if (!m_currentCellPossibleValues.contains(cell->getState()))
    {
        return false;
    }

    // Check neighbours
    bool matched = true;
    // Rappel : QMap<relativePosition, possibleStates>
    for (QMap<QVector<short>,  QVector<unsigned int> >::const_iterator it = m_matrix.begin() ; it != m_matrix.end(); ++it)
    {
        if (cell->getNeighbour(it.key()) == nullptr) // Border management
        {
            matched = false;
            break;
        }
        if (! it.value().contains(cell->getNeighbour(it.key())->getState()))
        {
            matched = false;
            break;
        }
    }

    return matched;
}

void MatrixRule::addNeighbourState(QVector<short> relativePosition, unsigned int matchState)
{
    m_matrix[relativePosition].push_back(matchState);
}

void MatrixRule::addNeighbourState(QVector<short> relativePosition, QVector<unsigned int> matchStates)
{
    for (QVector<unsigned int>::const_iterator it = matchStates.begin(); it != matchStates.end(); ++it)
        m_matrix[relativePosition].push_back(*it);
}

QJsonObject MatrixRule::toJson() const
{
    QJsonObject object(Rule::toJson());

    object.insert("type", QJsonValue("matrix"));

    QJsonArray neighbours;
    for (QMap<QVector<short>,  QVector<unsigned int> >::const_iterator it = m_matrix.begin(); it != m_matrix.end(); ++it)
    {
        QJsonObject aNeighbour;
        QJsonArray relativePosition;
        for (unsigned int i = 0; i < it.key().size(); i++)
        {
            relativePosition.append(QJsonValue((int)it.key().at(i)));
        }
        aNeighbour.insert("relativePosition", relativePosition);

        QJsonArray neighbourStates;
        for (unsigned int i = 0; i < it.value().size(); i++)
        {
            neighbourStates.append(QJsonValue((int)it.value().at(i)));
        }
        aNeighbour.insert("neighbourStates", neighbourStates);

        neighbours.append(aNeighbour);
    }
    object.insert("neighbours", neighbours);

    return object;
}
```

7.29 matrixrule.h File Reference

```
#include <QVector>
#include <QMap>
#include "cell.h"
```

```
#include "rule.h"
```

Classes

- class MatrixRule

  Manage specific rules, about specific values of specific neighbour.

Functions

- QVector$<$ unsigned int $>$ fillInterval (unsigned int min, unsigned int max)

  Returns a vector fill of the integers between min and max (all included)

7.29.1 Function Documentation

7.29.1.1 fillInterval()

```
QVector<unsigned int> fillInterval (
            unsigned int min,
            unsigned int max )
```

Returns a vector fill of the integers between min and max (all included)

Returns

Interval

Parameters

min	Minimal value (included)
max	Maximal value (included)

Definition at line 8 of file matrixrule.cpp.

7.30 matrixrule.h

```
#ifndef MATRIXRULE_H
#define MATRIXRULE_H

#include <QVector>
#include <QMap>
#include "cell.h"
#include "rule.h"

QVector<unsigned int> fillInterval(unsigned int min, unsigned int max);

class MatrixRule : public Rule
```

```
{
    public:
        MatrixRule(unsigned int finalState, QVector<unsigned int> currentStates = QVector<unsigned int>());


        virtual bool matchCell(const Cell* cell) const;
        void addNeighbourState(QVector<short> relativePosition, unsigned int matchState);
        void addNeighbourState(QVector<short> relativePosition, QVector<unsigned int> matchStates);

        QJsonObject toJson() const;


private:

        QMap<QVector<short>,  QVector<unsigned int> > m_matrix;
};



#endif // MATRIXRULE_H
```

7.31 neighbourrule.cpp File Reference

`#include "neighbourrule.h"`

7.32 neighbourrule.cpp

```
#include "neighbourrule.h"

bool NeighbourRule::inInterval(unsigned int matchingNeighbours)const
{
    if(matchingNeighbours >= m_neighbourInterval.first && matchingNeighbours<= m_neighbourInterval.second)
        return true;
    else
        return false;
}

NeighbourRule::NeighbourRule(unsigned int outputState, QVector<unsigned int> currentCellValues, QPair<unsigned int, unsigned int> intervalNbrNeighbour, QSet<unsigned int> neighbourValues)
 :
        Rule(currentCellValues, outputState), m_neighbourInterval(intervalNbrNeighbour), m_neighbourPossibleValues(neighbourValues)
{
    if (m_neighbourInterval.second == 0)
        throw QString(QObject::tr("Low value of the number of neighbour interval can't be 0"));
    if (m_neighbourInterval.first > m_neighbourInterval.second)
        throw QString(QObject::tr("The interval must be (x,y) with x <= y"));
}

NeighbourRule::~NeighbourRule()
{

}

bool NeighbourRule::matchCell(const Cell *c)const
{
    unsigned int matchingNeighbours = 0;
    if (!m_currentCellPossibleValues.contains(c->getState()))
        return false;

   // QSet<unsigned int> set;
    //QSet<unsigned int> m_neighbourPossibleValues;
    //set<<3<<2<<5<<9;
    QSet<unsigned int>::const_iterator i = m_neighbourPossibleValues.constBegin();
    if (i == m_neighbourPossibleValues.constEnd()) // All possibles values (except 0)
    {
        matchingNeighbours = c->countNeighbours();
    }
    else
```

```
    {
        while (i != m_neighbourPossibleValues.constEnd()) {
            //std::cout<<*i;
            matchingNeighbours += c->countNeighbours(*i);
            ++i;
        }
    }
    if(!inInterval(matchingNeighbours))
        return false; //the rule cannot be applied to the cell

    return true; //the rule can be applied to the cell

}

QJsonObject NeighbourRule::toJson() const
{
    QJsonObject object(Rule::toJson());

    object.insert("type", QJsonValue("neighbour"));
    object.insert("neighbourNumberMin", QJsonValue((int)m_neighbourInterval.first));
    object.insert("neighbourNumberMax", QJsonValue((int)m_neighbourInterval.second));

    QJsonArray neighbourState;
    for (QSet<unsigned int>::const_iterator it = m_neighbourPossibleValues.begin()
; it != m_neighbourPossibleValues.end(); ++it)
    {
        neighbourState.append(QJsonValue((int)*it));
    }
    object.insert("neighbourStates", neighbourState);

    return object;
}
```

7.33 neighbourrule.h File Reference

```
#include <QPair>
#include <QSet>
#include "cell.h"
#include "rule.h"
```

Classes

- class NeighbourRule

  Contains the rule condition and the output state if that condition is satisfied The rule modifies a cell depending on the number of its neighbours belonging to a range.

7.34 neighbourrule.h

```
#ifndef NEIGHBOURRULE_H
#define NEIGHBOURRULE_H

#include <QPair>
#include <QSet>
#include "cell.h"
#include "rule.h"

class NeighbourRule : public Rule
{
private:
    QPair<unsigned int , unsigned int> m_neighbourInterval;
    //ATTENTION check that first is lower than second
    QSet<unsigned int> m_neighbourPossibleValues;
    bool inInterval(unsigned int matchingNeighbours)const;
    //bool load(const QJsonObject &json);
public:
    NeighbourRule(unsigned int outputState, QVector<unsigned int> currentCellValues,
QPair<unsigned int, unsigned int> intervalNbrNeighbour, QSet<unsigned int> neighbourValues = QSet<unsigned int>());
```

```
    ~NeighbourRule();
    bool matchCell(const Cell * c)const;

    virtual QJsonObject toJson() const;
};

#endif // NEIGHBOURRULE_H
```

7.35 presentation.md File Reference

7.36 presentation.md

```
\page Presentation
# What is AutoCell
The purpose of this project is to create a Cellular Automate Simulator.

\includedoc CellHandler
```

7.37 README.md File Reference

7.38 README.md

```
\mainpage

To generate the Documentation, go in Documentation directory and run `make`.

It will generate html doc (in `output/html/index.html`) and latex doc (pdf output directely in
 Documentation directory (`docPdf.pdf`).
```

7.39 rule.cpp File Reference

```
#include "rule.h"
```

7.40 rule.cpp

```
#include "rule.h"

Rule::Rule(QVector<unsigned int> currentCellValues, unsigned int outputState):
        m_currentCellPossibleValues(currentCellValues), m_cellOutputState(outputState)
{

}

QJsonObject Rule::toJson() const
{
    QJsonObject object;
    object.insert("finalState", QJsonValue((int)m_cellOutputState));

    QJsonArray currentStates;
    for (unsigned int i = 0; i < m_currentCellPossibleValues.size(); i++)
    {
        currentStates.append(QJsonValue((int)m_currentCellPossibleValues.at(i)))
;
    }
    object.insert("currentStates", currentStates);

    return object;
}

unsigned int Rule::getCellOutputState()const
{
    return m_cellOutputState;
}

```

7.41 rule.h File Reference

```
#include <QVector>
#include <QJsonObject>
#include <QJsonArray>
#include "cell.h"
```

Classes

- class Rule

7.42 rule.h

```
#ifndef RULE_H
#define RULE_H

#include <QVector>
#include <QJsonObject>
#include <QJsonArray>
#include "cell.h"


class Rule
{
protected:
    QVector<unsigned int> m_currentCellPossibleValues;
    unsigned int m_cellOutputState;
public:
    Rule(QVector<unsigned int> currentCellValues, unsigned int outputState);

    virtual QJsonObject toJson() const = 0;

    virtual bool matchCell(const Cell * c)const = 0;
    unsigned int getCellOutputState() const;

};

#endif // RULE_H
```

7.43 ruleeditor.cpp File Reference

```
#include "ruleeditor.h"
```

7.44 ruleeditor.cpp

```
#include "ruleeditor.h"

RuleEditor::RuleEditor(QWidget *parent)
{
    selectedRule = -1;
    QHBoxLayout *hlayout = new QHBoxLayout();
    m_rulesListWidget = new QListWidget(this);
    QLabel *rulesLabel = new QLabel("Rules ",this);
    QVBoxLayout *rulesListLayout = new QVBoxLayout();
    rulesListLayout->addWidget(rulesLabel);
    rulesListLayout->addWidget(m_rulesListWidget);
    hlayout->addLayout(rulesListLayout);

    QGridLayout *rulesInputLayout = new QGridLayout();
```

```
    rulesInputLayout->addWidget(new QLabel("Current cell values :",this),0,0);
    m_currentStatesEdit = new QLineEdit(this);
    QRegExp rgx("([0-9]+,)*");
    QRegExpValidator *v = new QRegExpValidator(rgx, this);
    m_currentStatesEdit->setValidator(v);
    rulesInputLayout->addWidget(m_currentStatesEdit,0,1);

    rulesInputLayout->addWidget(new QLabel("Neighbour number lower bound :",this),1,0);
    m_lowerNeighbourBox = new QSpinBox(this);
    rulesInputLayout->addWidget(m_lowerNeighbourBox,1,1);

    rulesInputLayout->addWidget(new QLabel("Neighbour number upper bound :",this),2,0);
    m_upperNeighbourBox = new QSpinBox(this);
    rulesInputLayout->addWidget(m_upperNeighbourBox,2,1);

    rulesInputLayout->addWidget(new QLabel("Neighbour values :",this),3,0);
    m_neighbourStatesEdit = new QLineEdit(this);
    m_neighbourStatesEdit->setValidator(v);
    rulesInputLayout->addWidget(m_neighbourStatesEdit,3,1);

    rulesInputLayout->addWidget(new QLabel("Output state :",this),4,0);
    m_outputStateBox = new QSpinBox(this);
    rulesInputLayout->addWidget(m_outputStateBox,4,1);

    hlayout->addLayout(rulesInputLayout);
    QVBoxLayout* mainLayout = new QVBoxLayout();
    QHBoxLayout* buttonLayout = new QHBoxLayout();

    m_addBt = new QPushButton("Add Rule",this);
    m_importBt = new QPushButton("Import Rule file",this);
    m_doneBt = new QPushButton("Done !",this);
    m_removeBt = new QPushButton("Remove Rule",this);
    buttonLayout->addWidget(m_importBt);
    buttonLayout->addWidget(m_addBt);
    buttonLayout->addWidget(m_removeBt);
    buttonLayout->addWidget(m_doneBt);

    mainLayout->addLayout(hlayout);
    mainLayout->addLayout(buttonLayout);
    setLayout(mainLayout);

    connect(m_addBt, SIGNAL(clicked(bool)), this, SLOT(addRule()));
    connect(m_importBt, SIGNAL(clicked(bool)), this, SLOT(importFile()));
    connect(m_doneBt, SIGNAL(clicked(bool)), this, SLOT(sendRules()));
    connect(m_removeBt, SIGNAL(clicked(bool)), this, SLOT(removeRule()));

}



void RuleEditor::addRule(){
    unsigned int outputState = m_outputStateBox->value();
    QVector<unsigned int> currentCellValues;
    QStringList valList = m_currentStatesEdit->text().split(",");
    for(int i = 0; i < valList.size(); i++) currentCellValues.append(valList.at(i).toInt());

    QPair<unsigned int, unsigned int> neighbourInterval;
    neighbourInterval.first = m_lowerNeighbourBox->value();
    neighbourInterval.second = m_upperNeighbourBox->value();

    QSet<unsigned int> neighbourValues;
    valList = m_neighbourStatesEdit->text().split(",");
    for(int i = 0; i < valList.size(); i++) neighbourValues << valList.at(i).toInt();

    m_rules.append(new NeighbourRule(outputState,currentCellValues,neighbourInterval,
neighbourValues));

    QString listLabel = m_currentStatesEdit->text()+" -> "+QString::number(
m_outputStateBox->value())
                        +" if "+QString::number(m_lowerNeighbourBox->value())+" to "+
                        QString::number(m_upperNeighbourBox->value())+" neighbours are
 in states "+
                        m_neighbourStatesEdit->text();
    m_rulesListWidget->addItem(listLabel);
}

void RuleEditor::removeRule(){
    m_rules.removeAt(m_rulesListWidget->currentRow());
    delete m_rulesListWidget->takeItem(m_rulesListWidget->currentRow());
}

void RuleEditor::sendRules(){
    emit rulesFilled(m_rules);
    this->close();
}
```

```
void RuleEditor::importFile(){
    QString fileName = QFileDialog::getOpenFileName(this, tr("Open Rule file"), ".",
                                                    tr("Automaton rule files (*.atr)"));
    if(!fileName.isEmpty()){
        emit fileImported(fileName);
        this->close();
    }
}
```

7.45 ruleeditor.h File Reference

```
#include <QtWidgets>
#include "neighbourrule.h"
```

Classes

- class RuleEditor

7.46 ruleeditor.h

```
#ifndef RULEEDITOR_H
#define RULEEDITOR_H
#include <QtWidgets>
#include "neighbourrule.h"


class RuleEditor : public QDialog
{
    Q_OBJECT
    QList<const Rule*> m_rules;
    QListWidget* m_rulesListWidget;
    QTableWidget* m_rulesTable;

    QSpinBox* m_outputStateBox;
    QLineEdit* m_currentStatesEdit;
    QLineEdit* m_neighbourStatesEdit;
    QSpinBox* m_upperNeighbourBox;
    QSpinBox* m_lowerNeighbourBox;

    QPushButton* m_addBt;
    QPushButton* m_doneBt;
    QPushButton* m_removeBt;
    QPushButton* m_importBt;

    unsigned int selectedRule;


public:
    explicit RuleEditor(QWidget *parent = nullptr);

signals:
    void rulesFilled(QList<const Rule*> rules);
    void fileImported(QString path);

public slots:
    void removeRule();
    void addRule();
    void importFile();
    void sendRules();


};

#endif // RULEEDITOR_H
```

Index