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 ~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 230 of file automate.cpp.

References m_rules.

Referenced by MainWindow::addAutomatonRules().

6.1.3.2 addRuleFile()

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

Definition at line 287 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 282 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 248 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 257 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 223 of file automate.cpp.

References saveCells(), and saveRules().

Referenced by MainWindow::~MainWindow().

6.1.3.8 saveCells()

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

Save cellHandler.

Definition at line 214 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 192 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 241 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::MainWindow(), 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::backward(), MainWindow::cellPressed(), MainWindow::changeCellValue(), MainWindow::createTab(), MainWindow::nextState(), MainWindow::reset(), MainWindow::runAutomaton(), MainWindow::saveToFile(), MainWindow::updateBoard(), and MainWindow::~MainWindow().

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::backward(), MainWindow::cellPressed(), MainWindow::changeCellValue(), MainWindow::closeTab(), MainWindow::createTab(), MainWindow::handlePlayPause(), MainWindow::MainWindow(), MainWindow::nextState(), MainWindow::openFile(), MainWindow::receiveCellHandler(), MainWindow::reset(), MainWindow::runAutomaton(), MainWindow::saveToFile(), MainWindow::setSize(), MainWindow::updateBoard(), and MainWindow::~MainWindow().

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.

Referenced by MainWindow::~MainWindow().

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 84 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 59 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 111 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 124 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.

Referenced by MainWindow::changeCellValue().

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 104 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 97 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 141 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 50 of file cell.cpp.

References m_states.

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

6.3.3.10 reset()

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

Reset the cell to the 1st state.

Definition at line 70 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 stateMax=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.
- 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.

Static Public Member Functions

- static unsigned int getMaxState ()

  Return the max state of the CellHandler.

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.

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 130 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 326 of file cellhandler.cpp.

Referenced by MainWindow::changeCellValue(), 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 319 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 335 of file cellhandler.cpp.

Referenced by MainWindow::changeCellValue(), 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 433 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 240 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 140 of file cellhandler.cpp.

References m_cells.

Referenced by MainWindow::cellPressed(), and MainWindow::changeCellValue().

6.4.5.7 getDimensions()

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

Accessor of m_dimensions.

Definition at line 154 of file cellhandler.cpp.

References m_dimensions.

Referenced by MainWindow::cellPressed(), MainWindow::changeCellValue(), 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 492 of file cellhandler.cpp.

References getListNeighboursPositionsRecursive().

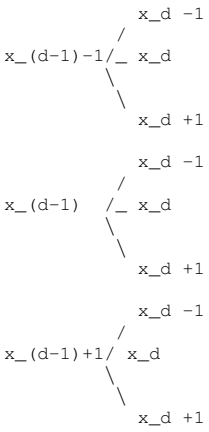
Referenced by foundNeighbours().

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:



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 533 of file cellhandler.cpp.

References m_dimensions.

Referenced by getListNeighboursPositions().

6.4.5.10 getMaxState()

```
unsigned int CellHandler::getMaxState ( )  [static]
```

Return the max state of the CellHandler.

Definition at line 147 of file cellhandler.cpp.

Referenced by MainWindow::handleTabChanged().

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 370 of file cellhandler.cpp.

References m_cells, m_dimensions, and positionIncrement().

Referenced by CellHandler().

6.4.5.12 nextStates()

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

Valid the state of all cells.

Definition at line 161 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 463 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 171 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 305 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 183 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 198 of file cellhandler.cpp.

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

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().

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

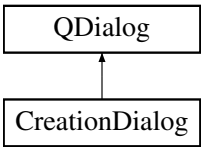
- cellhandler.h
- cellhandler.cpp

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=CellHandler::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 573 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:

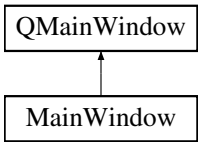
- cellhandler.h
- cellhandler.cpp

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)

  Adds a list of rules to the last Automaton.
- void addAutomatonRuleFile (QString path)

  Adds a list of rules to the last Automaton from a given file.
- void forward ()

  Show the Automaton's next state.
- void backward ()

  Show the Automaton's previous state.
- void closeTab (int n)

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

  Runs the automaton simulation. Displays a new state on the board at regular intervals, set by the user in the interface.
- void handlePlayPause ()

  Handles the press event of the play/pause button.
- void reset ()

  Resets the current Automaton, by setting its cells to their initial state.
- void cellPressed (int i, int j)

  Handles board cell press event.
- void changeCellValue ()

  Sets the selected cell's value to the one set by the user.
- void handleTabChanged ()

  Handles tab change.
- void setSize (int newCellSize)

Public Member Functions

- MainWindow (QWidget *parent=nullptr)
- virtual ~MainWindow ()

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)

  Returns the board of the n-th tab.

Static Private Member Functions

- static QColor getColor (unsigned int cellState)

  Return the color wich correspond to the cellState.

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
- QSpinBox * m_cellSetter

Simulation time step duration input.

- QTimer * m_timer

  Cell state manual modification.

- QSlider * m_zoom

  Timer running between simulation steps.

- Automate * m_newAutomate
- bool running
- QToolBar * m_toolBar
- int m_currentCellX

  Toolbar containing the buttons.

- int m_currentCellY
- unsigned int m_boardHSize = 25

  Board size settings.

- 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 AutomateHandler::addAutomate(), createActions(), createIcons(), createTab(), createTabs(), createToolBar(), AutomateHandler::getAutomateHandler(), m_tabs, m_timeStep, m_zoom, running, and updateBoard().

6.7.2.2 ~MainWindow()

```
MainWindow::~MainWindow ( )  [virtual]
```

Definition at line 45 of file mainwindow.cpp.

References AutomateHandler::getAutomate(), AutomateHandler::getAutomateHandler(), AutomateHandler::getNumberAutomates(), m_timeStep, m_zoom, and Automate::saveAll().

6.7.3 Member Function Documentation

6.7.3.1 addAutomatonRuleFile

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

Adds a list of rules to the last Automaton from a given file.

Definition at line 521 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]
```

Adds a list of rules to the last Automaton.

Definition at line 510 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 481 of file mainwindow.cpp.

References getBoard(), and m_cellSize.

Referenced by updateBoard().

6.7.3.4 backward

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

Show the Automaton's previous state.

Definition at line 595 of file mainwindow.cpp.

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

Referenced by createActions().

6.7.3.5 cellPressed

```
void MainWindow::cellPressed (
            int i,
            int j )  [slot]
```

Handles board cell press event.

Definition at line 604 of file mainwindow.cpp.

References AutomateHandler::getAutomate(), AutomateHandler::getAutomateHandler(), CellHandler::getCell(), CellHandler::getDimensions(), Cell::getState(), m_cellSetter, m_currentCellX, m_currentCellY, and m_tabs.

Referenced by createTab().

6.7.3.6 changeCellValue

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

Sets the selected cell's value to the one set by the user.

Definition at line 626 of file mainwindow.cpp.

References CellHandler::begin(), CellHandler::end(), Cell::forceState(), AutomateHandler::getAutomate(), AutomateHandler::getAutomateHandler(), getBoard(), CellHandler::getCell(), getColor(), CellHandler::getDimensions(), m_cellSetter, m_currentCellX, m_currentCellY, m_tabs, and updateBoard().

Referenced by createToolBar().

6.7.3.7 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 499 of file mainwindow.cpp.

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

Referenced by createTabs().

6.7.3.8 createActions()

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

Creates and connects QActions and associated buttons for the MainWindow.

Definition at line 95 of file mainwindow.cpp.

References backward(), forward(), handlePlayPause(), m_cellSize, m_fastBackward, m_fastBackwardBt, m_fastBackwardIcon, 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, m_zoom, openCreationWindow(), openFile(), reset(), saveToFile(), and setSize().

Referenced by MainWindow().

6.7.3.9 createBoard()

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

6.7.3.10 createIcons()

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

Creates Icons for the MainWindow.

Definition at line 65 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().

6.7.3.11 createTab()

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

Creates a new Tab with an empty board.

Definition at line 204 of file mainwindow.cpp.

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

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

6.7.3.12 createTabs()

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

Creates a QTabWidget for the main window and displays it.

Definition at line 466 of file mainwindow.cpp.

References closeTab(), handleTabChanged(), and m_tabs.

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

6.7.3.13 createToolBar()

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

Creates the toolBar for the MainWindow.

Definition at line 151 of file mainwindow.cpp.

References changeCellValue(), m_cellSetter, m_fastBackwardBt, m_fastForwardBt, m_newAutomatonBt, m_openAutomatonBt, m_playPauseBt, m_resetBt, m_saveAutomatonBt, m_timeStep, m_toolBar, and m_zoom.

Referenced by MainWindow().

6.7.3.14 forward

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

Show the Automaton's next state.

Definition at line 392 of file mainwindow.cpp.

References nextState().

Referenced by createActions().

6.7.3.15 getBoard()

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

Returns the board of the n-th tab.

Definition at line 399 of file mainwindow.cpp.

References m_tabs.

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

6.7.3.16 getColor()

```
QColor MainWindow::getColor (
            unsigned int cellState )  [static], [private]
```

Return the color wich correspond to the cellState.

Definition at line 405 of file mainwindow.cpp.

Referenced by changeCellValue(), and updateBoard().

6.7.3.17 handlePlayPause

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

Handles the press event of the play/pause button.

Definition at line 529 of file mainwindow.cpp.

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

Referenced by createActions().

6.7.3.18 handleTabChanged

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

Handles tab change.

Definition at line 662 of file mainwindow.cpp.

References CellHandler::getMaxState(), m_cellSetter, m_currentCellX, m_currentCellY, and m_tabs.

Referenced by createTabs().

6.7.3.19 nextState()

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

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

Definition at line 325 of file mainwindow.cpp.

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

Referenced by forward().

6.7.3.20 openCreationWindow

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

Opens the automaton creation window.

Definition at line 290 of file mainwindow.cpp.

References receiveCellHandler().

Referenced by createActions().

6.7.3.21 openFile

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

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

Definition at line 250 of file mainwindow.cpp.

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

Referenced by createActions().

6.7.3.22 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 303 of file mainwindow.cpp.

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

Referenced by openCreationWindow().

6.7.3.23 reset

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

Resets the current Automaton, by setting its cells to their initial state.

Definition at line 568 of file mainwindow.cpp.

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

Referenced by createActions().

6.7.3.24 runAutomaton

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

Runs the automaton simulation. Displays a new state on the board at regular intervals, set by the user in the interface.

Definition at line 556 of file mainwindow.cpp.

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

Referenced by handlePlayPause().

6.7.3.25 saveToFile

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

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

Definition at line 270 of file mainwindow.cpp.

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

Referenced by closeTab(), and createActions().

6.7.3.26 setSize

```
void MainWindow::setSize (
            int newCellSize )  [slot]
```

Definition at line 672 of file mainwindow.cpp.

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

Referenced by createActions().

6.7.3.27 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 342 of file mainwindow.cpp.

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

Referenced by backward(), changeCellValue(), MainWindow(), 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]
```

Board size settings.

Definition at line 72 of file mainwindow.h.

6.7.4.2 m_boardVSize

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

Definition at line 73 of file mainwindow.h.

6.7.4.3 m_cellSetter

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

Simulation time step duration input.

Definition at line 59 of file mainwindow.h.

Referenced by cellPressed(), changeCellValue(), createToolBar(), and handleTabChanged().

6.7.4.4 m_cellSize

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

Definition at line 74 of file mainwindow.h.

Referenced by addEmptyRow(), createActions(), createTab(), and setSize().

6.7.4.5 m_currentCellX

```
int MainWindow::m_currentCellX  [private]
```

Toolbar containing the buttons.

Definition at line 68 of file mainwindow.h.

Referenced by cellPressed(), changeCellValue(), and handleTabChanged().

6.7.4.6 m_currentCellY

```
int MainWindow::m_currentCellY  [private]
```

Definition at line 69 of file mainwindow.h.

Referenced by cellPressed(), changeCellValue(), and handleTabChanged().

6.7.4.7 m_fastBackward

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

Definition at line 40 of file mainwindow.h.

Referenced by createActions().

6.7.4.8 m_fastBackwardBt

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

Definition at line 51 of file mainwindow.h.

Referenced by createActions(), and createToolBar().

6.7.4.9 m_fastBackwardIcon

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

Icons.

Definition at line 26 of file mainwindow.h.

Referenced by createActions(), and createIcons().

6.7.4.10 m_fastForward

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

Definition at line 39 of file mainwindow.h.

Referenced by createActions().

6.7.4.11 m_fastForwardBt

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

Definition at line 50 of file mainwindow.h.

Referenced by createActions(), and createToolBar().

6.7.4.12 m_fastForwardIcon

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

Definition at line 27 of file mainwindow.h.

Referenced by createActions(), and createIcons().

6.7.4.13 m_newAutomate

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

Definition at line 64 of file mainwindow.h.

6.7.4.14 m_newAutomaton

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

Definition at line 43 of file mainwindow.h.

Referenced by createActions().

6.7.4.15 m_newAutomatonBt

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

Definition at line 54 of file mainwindow.h.

Referenced by createActions(), and createToolBar().

6.7.4.16 m_newIcon

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

Definition at line 30 of file mainwindow.h.

Referenced by createActions(), and createIcons().

6.7.4.17 m_nextState

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

Definition at line 37 of file mainwindow.h.

6.7.4.18 m_nextStateBt

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

Definition at line 48 of file mainwindow.h.

6.7.4.19 m_openAutomaton

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

Definition at line 41 of file mainwindow.h.

Referenced by createActions().

6.7.4.20 m_openAutomatonBt

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

Definition at line 52 of file mainwindow.h.

Referenced by createActions(), and createToolBar().

6.7.4.21 m_openIcon

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

Definition at line 32 of file mainwindow.h.

Referenced by createActions(), and createIcons().

6.7.4.22 m_pauseIcon

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

Definition at line 29 of file mainwindow.h.

Referenced by createIcons(), and handlePlayPause().

6.7.4.23 m_playIcon

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

Definition at line 28 of file mainwindow.h.

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

6.7.4.24 m_playPause

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

Actions.

Definition at line 36 of file mainwindow.h.

Referenced by createActions().

6.7.4.25 m_playPauseBt

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

Buttons.

Definition at line 47 of file mainwindow.h.

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

6.7.4.26 m_previousState

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

Definition at line 38 of file mainwindow.h.

6.7.4.27 m_previousStateBt

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

Definition at line 49 of file mainwindow.h.

6.7.4.28 m_resetAutomaton

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

Definition at line 44 of file mainwindow.h.

Referenced by createActions().

6.7.4.29 m_resetBt

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

Definition at line 55 of file mainwindow.h.

Referenced by createActions(), and createToolBar().

6.7.4.30 m_resetIcon

```
QIcon MainWindow::m_resetIcon  [private]
```

Definition at line 33 of file mainwindow.h.

Referenced by createActions(), and createIcons().

6.7.4.31 m_saveAutomaton

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

Definition at line 42 of file mainwindow.h.

Referenced by createActions().

6.7.4.32 m_saveAutomatonBt

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

Definition at line 53 of file mainwindow.h.

Referenced by createActions(), and createToolBar().

6.7.4.33 m_saveIcon

```
QIcon MainWindow::m_saveIcon  [private]
```

Definition at line 31 of file mainwindow.h.

Referenced by createActions(), and createIcons().

6.7.4.34 m_tabs

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

Definition at line 22 of file mainwindow.h.

Referenced by backward(), cellPressed(), changeCellValue(), closeTab(), createTabs(), getBoard(), handleTabChanged(), MainWindow(), nextState(), openFile(), receiveCellHandler(), reset(), runAutomaton(), saveToFile(), setSize(), and updateBoard().

6.7.4.35 m_timer

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

Cell state manual modification.

Definition at line 60 of file mainwindow.h.

Referenced by handlePlayPause().

6.7.4.36 m_timeStep

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

Definition at line 58 of file mainwindow.h.

Referenced by createToolBar(), handlePlayPause(), MainWindow(), and ~MainWindow().

6.7.4.37 m_toolBar

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

Definition at line 66 of file mainwindow.h.

Referenced by createToolBar().

6.7.4.38 m_zoom

```
QSlider* MainWindow::m_zoom  [private]
```

Timer running between simulation steps.

Definition at line 62 of file mainwindow.h.

Referenced by createActions(), createToolBar(), MainWindow(), and ~MainWindow().

6.7.4.39 running

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

Definition at line 65 of file mainwindow.h.

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

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

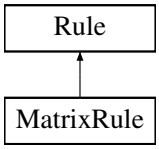
- mainwindow.h
- mainwindow.cpp

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

Protected Attributes

- QMap< QVector< short >, QVector< unsigned int > > m_matrix
  Key correspond to relative position and the value to matchable states.

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 )  [virtual]
```

Add a possible state to a relative position.

Definition at line 67 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 )  [virtual]
```

Add multiples possible states to existents states.

Definition at line 74 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 82 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  [protected]
```

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:

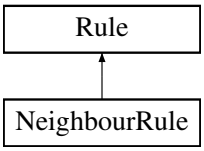
- matrixrule.h
- matrixrule.cpp

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.
- QJsonObject toJson () const

  Return a QJsonObject to save the rule.

Protected 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.

Protected 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.

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  [protected]
```

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  [protected]
```

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  [protected]
```

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().

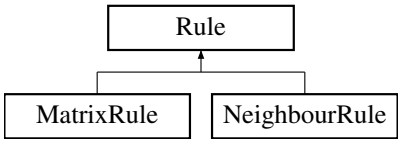
The documentation for this class was generated from the following files:

- 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 ~Rule ()
- 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.

6.10.2.2 ~Rule()

```
virtual Rule::~Rule ( )  [inline], [virtual]
```

Definition at line 22 of file rule.h.

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().

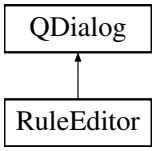
The documentation for this class was generated from the following files:

- 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 (unsigned int dimensions, 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
- QSpinBox * m_automatonNumber
- QPushButton * m_addBt
- QPushButton * m_doneBt
- QPushButton * m_removeBt
- QPushButton * m_importBt
- unsigned int m_selectedRule
- unsigned int m_dimensions

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 (
            unsigned int dimensions,
            QWidget * parent = nullptr )  [explicit]
```

Definition at line 3 of file ruleeditor.cpp.

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

6.11.3 Member Function Documentation

6.11.3.1 addRule

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

Definition at line 85 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 127 of file ruleeditor.cpp.

References fileImported().

Referenced by RuleEditor().

6.11.3.4 removeRule

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

Definition at line 108 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 113 of file ruleeditor.cpp.

References generate1DRules(), m_automatonNumber, m_dimensions, 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 21 of file ruleeditor.h.

Referenced by RuleEditor().

6.11.4.2 m_automatonNumber

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

Definition at line 19 of file ruleeditor.h.

Referenced by RuleEditor(), and sendRules().

6.11.4.3 m_currentStatesEdit

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

Definition at line 15 of file ruleeditor.h.

Referenced by addRule(), and RuleEditor().

6.11.4.4 m_dimensions

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

Definition at line 27 of file ruleeditor.h.

Referenced by RuleEditor(), and sendRules().

6.11.4.5 m_doneBt

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

Definition at line 22 of file ruleeditor.h.

Referenced by RuleEditor().

6.11.4.6 m_importBt

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

Definition at line 24 of file ruleeditor.h.

Referenced by RuleEditor().

6.11.4.7 m_lowerNeighbourBox

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

Definition at line 18 of file ruleeditor.h.

Referenced by addRule(), and RuleEditor().

6.11.4.8 m_neighbourStatesEdit

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

Definition at line 16 of file ruleeditor.h.

Referenced by addRule(), and RuleEditor().

6.11.4.9 m_outputStateBox

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

Definition at line 14 of file ruleeditor.h.

Referenced by addRule(), and RuleEditor().

6.11.4.10 m_removeBt

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

Definition at line 23 of file ruleeditor.h.

Referenced by RuleEditor().

6.11.4.11 m_rules

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

Definition at line 10 of file ruleeditor.h.

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

6.11.4.12 m_rulesListWidget

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

Definition at line 11 of file ruleeditor.h.

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

6.11.4.13 m_rulesTable

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

Definition at line 12 of file ruleeditor.h.

6.11.4.14 m_selectedRule

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

Definition at line 26 of file ruleeditor.h.

Referenced by RuleEditor().

6.11.4.15 m_upperNeighbourBox

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

Definition at line 17 of file ruleeditor.h.

Referenced by addRule(), and 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 316 of file automate.cpp.

References getRuleFromNumber().

Referenced by RuleEditor::sendRules().

7.1.1.2 getRuleFromNumber()

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

Definition at line 339 of file automate.cpp.

References MatrixRule::addNeighbourState().

Referenced by generate1DRules().

7.2 automate.cpp

```
00001 #include "automate.h"
00002
00007 bool Automate::loadRules(const QJsonArray &json)
00008 {
00009
00010     for (QJsonArray::const_iterator it = json.begin(); it != json.end(); ++it)
00011     {
00012         if (!it->isObject())
00013             return false;
00014         QJsonObject ruleJson = it->toObject();
00015
00016         if (!ruleJson.contains("type") || !ruleJson["type"].isString())
00017             return false;
00018         if (!ruleJson.contains("finalState") || !ruleJson["finalState"].isDouble())
00019             return false;
00020         if (!ruleJson.contains("currentStates") || !ruleJson["currentStates"].isArray())
00021             return false;
00022
00023         QVector<unsigned int> currentStates;
00024         QJsonArray statesJson = ruleJson["currentStates"].toArray();
00025         for (unsigned int i = 0; i < statesJson.size(); i++)
00026         {
00027             if (!statesJson.at(i).isDouble())
00028                 return false;
00029             currentStates.push_back(statesJson.at(i).toInt());
00030         }
00031
00032         if (!ruleJson["type"].toString().compare("neighbour", Qt::CaseInsensitive))
00033         {
00034             if (!ruleJson.contains("neighbourNumberMin") || !ruleJson["neighbourNumberMin"].isDouble())
00035                 return false;
00036             if (!ruleJson.contains("neighbourNumberMax") || !ruleJson["neighbourNumberMax"].isDouble())
00037                 return false;
00038
00039
00040
00041             QPair<unsigned int, unsigned int> nbrNeighbourInterval(ruleJson["neighbourNumberMin"].toInt(),
      ruleJson["neighbourNumberMax"].toInt());
00042             NeighbourRule *newRule;
00043             if (ruleJson.contains("neighbourStates"))
00044             {
00045                 if (!ruleJson["neighbourStates"].isArray())
00046                     return false;
00047                 QSet<unsigned int> neighbourStates;
00048
00049                 QJsonArray statesJson = ruleJson["neighbourStates"].toArray();
00050                 for (unsigned int i = 0; i < statesJson.size(); i++)
00051                 {
00052                     if (!statesJson.at(i).isDouble())
00053                         return false;
00054                     neighbourStates.insert(statesJson.at(i).toInt());
00055                 }
00056                 newRule = new NeighbourRule((unsigned int)ruleJson["finalState"].toInt(),
      currentStates, nbrNeighbourInterval, neighbourStates);
00057             }
00058             else
00059                 newRule = new NeighbourRule((unsigned int)ruleJson["finalState"].toInt(),
      currentStates, nbrNeighbourInterval);
00060             m_rules.push_back(newRule);
00061         }
00062         else if (!ruleJson["type"].toString().compare("matrix", Qt::CaseInsensitive))
00063         {
00064             MatrixRule *newRule = new MatrixRule((unsigned int)ruleJson["finalState"].
      toInt(), currentStates);
00065             if (ruleJson.contains("neighbours"))
00066             {
00067                 if (!ruleJson["neighbours"].isArray())
00068                     return false;
00069                 QJsonArray neighboursJson = ruleJson["neighbours"].toArray();
00070                 for (unsigned int i = 0; i < neighboursJson.size(); i++)
00071                 {
00072                     if (!neighboursJson.at(i).isObject())
00073                         return false;
00074
00075                     if (!neighboursJson.at(i).toObject().contains("relativePosition") || !neighboursJson.at
      (i).toObject()["relativePosition"].isArray())
00076                         return false;
00077                     if (!neighboursJson.at(i).toObject().contains("neighbourStates") || !neighboursJson.at(
      i).toObject()["neighbourStates"].isArray())
00078                         return false;
00079
00080                     QVector<unsigned int> neighbourStates;
00081
00082
```

```
00083                     QJsonArray statesJson = neighboursJson.at(i).toObject()["neighbourStates"].toArray();
00084                     for (unsigned int j = 0; j < statesJson.size(); j++)
00085                     {
00086                         if (!statesJson.at(j).isDouble())
00087                             return false;
00088                         neighbourStates.push_back(statesJson.at(j).toInt());
00089                     }
00090 
00091                     QVector<short> relativePosition;
00092                     QJsonArray positionJson = neighboursJson.at(i).toObject()["relativePosition"].toArray()
    ;
00093                     for (unsigned int j = 0; j < positionJson.size(); j++)
00094                     {
00095                         if (!positionJson.at(j).isDouble())
00096                             return false;
00097                         relativePosition.push_back(positionJson.at(j).toInt());
00098                     }
00099                     if (relativePosition.size() != m_cellHandler->
    getDimensions().size())
00100                         return false;
00101                     newRule->addNeighbourState(relativePosition, neighbourStates);
00102                 }
00103 
00104             }
00105             m_rules.push_back(newRule);
00106 
00107 
00108         }
00109         else
00110             return false;
00111 
00112     }
00113     return true;
00114 }
00115 
00120 Automate::Automate(QString cellHandlerFilename)
00121 {
00122     m_cellHandler = new CellHandler(cellHandlerFilename);
00123 
00124 }
00125 
00133 Automate::Automate(const QVector<unsigned int> dimensions,
    CellHandler::generationTypes type, unsigned int stateMax, unsigned int density)
00134 {
00135     m_cellHandler = new CellHandler(dimensions, type, stateMax, density);
00136 
00137 }
00138 
00144 Automate::Automate(QString cellHandlerFilename, QString ruleFilename)
00145 {
00146     m_cellHandler = new CellHandler(cellHandlerFilename);
00147 
00148     QFile ruleFile(ruleFilename);
00149     if (!ruleFile.open(QIODevice::ReadOnly | QIODevice::Text)) {
00150         qWarning("Couldn't open given file.");
00151         throw QString(QObject::tr("Couldn't open given file"));
00152     }
00153 
00154     QJsonParseError parseErr;
00155     QJsonDocument loadDoc(QJsonDocument::fromJson(ruleFile.readAll(), &parseErr));
00156 
00157     ruleFile.close();
00158 
00159 
00160     if (loadDoc.isNull() || loadDoc.isEmpty())
00161     {
00162         qWarning() << "Could not read data : ";
00163         qWarning() << parseErr.errorString();
00164         throw QString(parseErr.errorString());
00165     }
00166 
00167     if (!loadDoc.isArray())
00168     {
00169         qWarning() << "We need an array of rules !";
00170         throw QString(QObject::tr("We need an array of rules!"));
00171     }
00172 
00173     loadRules(loadDoc.array());
00174 
00175 }
00176 
00179 Automate::~Automate()
00180 {
00181     delete m_cellHandler;
00182     for (QList<const Rule*>::iterator it = m_rules.begin(); it != m_rules.end(); ++it)
00183     {
00184 
```

```
00185         delete *it;
00186     }
00187 }
00188
00192 bool Automate::saveRules(QString filename) const
00193 {
00194     QFile ruleFile(filename);
00195     if (!ruleFile.open(QIODevice::WriteOnly | QIODevice::Text)) {
00196         qWarning("Couldn't open given file.");
00197         throw QString(QObject::tr("Couldn't open given file"));
00198     }
00199
00200     QJsonArray array;
00201
00202     for (QList<const Rule*>::const_iterator it = m_rules.cbegin(); it !=
      m_rules.cend(); ++it)
00203         array.append((*it)->toJson());
00204
00205     QJsonDocument doc(array);
00206
00207     ruleFile.write(doc.toJson());
00208
00209     return true;
00210 }
00211
00214 bool Automate::saveCells(QString filename) const
00215 {
00216     if (m_cellHandler != nullptr)
00217         return m_cellHandler->save(filename);
00218     return false;
00219 }
00220
00223 bool Automate::saveAll(QString cellHandlerFilename, QString rulesFilename) const
00224 {
00225     return saveRules(rulesFilename) && saveCells(cellHandlerFilename);
00226 }
00227
00230 void Automate::addRule(const Rule *newRule)
00231 {
00232     m_rules.push_back(newRule);
00233 }
00234
00241 void Automate::setRulePriority(const Rule *rule, unsigned int newPlace)
00242 {
00243     m_rules.move(m_rules.indexOf(rule), newPlace);
00244 }
00245
00248 const QList<const Rule *> &Automate::getRules() const
00249 {
00250    return m_rules;
00251 }
00252
00257 bool Automate::run(unsigned int nbSteps) //void instead ?
00258 {
00259     for(unsigned int i = 0; i<nbSteps; ++i)
00260     {
00261         for (CellHandler::iterator it = m_cellHandler->
      begin(); it != m_cellHandler->end(); ++it)
00262         {
00263             for (QList<const Rule*>::iterator rule = m_rules.begin(); rule !=
      m_rules.end() ; ++rule)
00264             {
00265                 if((*rule)->matchCell(*it)) //if the cell matches with the rule, its state is changed
00266                 {
00267                     it->setState((*rule)->getCellOutputState());
00268                     break;
00269                 }
00270             }
00271
00272
00273         }
00274         m_cellHandler->nextStates(); //apply the changes to all the cells
       simultaneously
00275     }
00276     return true;
00277
00278 }
00279
00282 const CellHandler &Automate::getCellHandler() const
00283 {
00284     return *m_cellHandler;
00285 }
00286
00287 void Automate::addRuleFile(QString filename){
00288     QFile ruleFile(filename);
00289     if (!ruleFile.open(QIODevice::ReadOnly | QIODevice::Text)) {
00290         qWarning("Couldn't open given file.");
```

```
00291         throw QString(QObject::tr("Couldn't open given file"));
00292     }
00293
00294     QJsonParseError parseErr;
00295     QJsonDocument loadDoc(QJsonDocument::fromJson(ruleFile.readAll(), &parseErr));
00296
00297     ruleFile.close();
00298
00299
00300     if (loadDoc.isNull() || loadDoc.isEmpty())
00301     {
00302         qWarning() << "Could not read data : ";
00303         qWarning() << parseErr.errorString();
00304         throw QString(parseErr.errorString());
00305     }
00306
00307     if (!loadDoc.isArray())
00308     {
00309         qWarning() << "We need an array of rules !";
00310         throw QString(QObject::tr("We need an array of rules!"));
00311     }
00312
00313     loadRules(loadDoc.array());
00314 }
00315
00316 QList<const Rule *> generate1DRules(unsigned int automatonNumber)
00317 {
00318     if (automatonNumber > 256) throw QString(QObject::tr("Automaton number not defined"));
00319     QList<const Rule*> ruleList;
00320     unsigned short int p = 128;
00321     int i = 7;
00322     while (i >= 0) {
00323         if (automatonNumber >= p)
00324         {
00325             ruleList.push_back((Rule*)getRuleFromNumber(i, 1));
00326             //numeroBit.push_back('1');
00327             automatonNumber -= p;
00328         }
00329         else {
00330             ruleList.push_back((Rule*)getRuleFromNumber(i, 0));
00331         }
00332         i--;
00333         p = p / 2;
00334     }
00335
00336     return ruleList;
00337 }
00338
00339 const MatrixRule* getRuleFromNumber(int previousConfiguration, int nextState)
00340 {
00341     if (previousConfiguration > 7 || previousConfiguration < 0)
00342         throw QString(QObject::tr("Configuration not possible"));
00343
00344     MatrixRule* newRule;
00345     switch(previousConfiguration)
00346     {
00347         case 0:
00348             newRule = new MatrixRule(nextState, {0});
00349             newRule->addNeighbourState(QVector<short>{-1}, 0);
00350             newRule->addNeighbourState(QVector<short>{1}, 0);
00351         break;
00352     case 1:
00353         newRule = new MatrixRule(nextState, {0});
00354         newRule->addNeighbourState(QVector<short>{-1}, 0);
00355         newRule->addNeighbourState(QVector<short>{1}, 1);
00356     break;
00357     case 2:
00358         newRule = new MatrixRule(nextState, {1});
00359         newRule->addNeighbourState(QVector<short>{-1}, 0);
00360         newRule->addNeighbourState(QVector<short>{1}, 0);
00361     break;
00362     case 3:
00363         newRule = new MatrixRule(nextState, {1});
00364         newRule->addNeighbourState(QVector<short>{-1}, 0);
00365         newRule->addNeighbourState(QVector<short>{1}, 1);
00366     break;
00367     case 4:
00368         newRule = new MatrixRule(nextState, {0});
00369         newRule->addNeighbourState(QVector<short>{-1}, 1);
00370         newRule->addNeighbourState(QVector<short>{1}, 0);
00371     break;
00372     case 5:
00373         newRule = new MatrixRule(nextState, {0});
00374         newRule->addNeighbourState(QVector<short>{-1}, 1);
00375         newRule->addNeighbourState(QVector<short>{1}, 1);
00376     break;
00377     case 6:
```

```
00378         newRule = new MatrixRule(nextState, {1});
00379         newRule->addNeighbourState(QVector<short>{-1}, 1);
00380         newRule->addNeighbourState(QVector<short>{1}, 0);
00381     break;
00382     case 7:
00383         newRule = new MatrixRule(nextState, {1});
00384         newRule->addNeighbourState(QVector<short>{-1}, 1);
00385         newRule->addNeighbourState(QVector<short>{1}, 1);
00386     break;
00387     }
00388
00389     return newRule;
00390 }
00391
```

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 316 of file automate.cpp.

References getRuleFromNumber().

Referenced by RuleEditor::sendRules().

7.3.1.2 getRuleFromNumber()

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

Definition at line 339 of file automate.cpp.

References MatrixRule::addNeighbourState().

Referenced by generate1DRules().

7.4 automate.h

```
00001 #ifndef AUTOMATE_H
00002 #define AUTOMATE_H
00003 #include <QVector>
00004 #include <QList>
00005
00006 #include "cellhandler.h"
00007 #include "rule.h"
00008 #include "neighbourrule.h"
00009 #include "matrixrule.h"
00010
00011
00015 class Automate
00016 {
00017 private:
00018     CellHandler* m_cellHandler = nullptr;
00019     QList<const Rule*> m_rules;
00020     friend class AutomateHandler;
00021
00022     bool loadRules(const QJsonArray &json);
00023 public:
00024     Automate(QString filename);
00025     Automate(const QVector<unsigned int> dimensions,
      CellHandler::generationTypes type =
      CellHandler::empty, unsigned int stateMax = 1, unsigned int density = 20);
00026     Automate(QString cellHandlerFilename, QString ruleFilename);
00027     virtual ~Automate();
00028
00029     bool saveRules(QString filename) const ;
00030     bool saveCells(QString filename) const ;
00031     bool saveAll(QString cellHandlerFilename, QString rulesFilename)const ;
00032
00033     void addRuleFile(QString filename);
00034     void addRule(const Rule* newRule);
00035     void setRulePriority(const Rule* rule, unsigned int newPlace);
00036     const QList<const Rule *> &getRules() const;
00037
00038
00039
00040 public:
00041     bool run(unsigned int nbSteps = 1);
00042     const CellHandler& getCellHandler() const;
00043 };
00044
00045 QList<const Rule*> generate1DRules(unsigned int automatonNumber);
00046 const MatrixRule *getRuleFromNumber(int previousConfiguration, int nextState);
00047
00048 #endif // AUTOMATE_H
```

7.5 automatehandler.cpp File Reference

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

7.6 automatehandler.cpp

```
00001 #include "automatehandler.h"
00002
00005 AutomateHandler * AutomateHandler::m_activeAutomateHandler
       = nullptr;
00006
00007
00010 AutomateHandler::AutomateHandler()
00011 {
00012
00013 }
00014
00015
00018 AutomateHandler::~AutomateHandler()
00019 {
00020     while(!m_ActiveAutomates.empty())
00021         delete(m_ActiveAutomates.first());
00022 }
00023
00024
00029 AutomateHandler & AutomateHandler::getAutomateHandler()
00030 {
00031     if (!m_activeAutomateHandler)
00032         m_activeAutomateHandler = new AutomateHandler;
00033     return *m_activeAutomateHandler;
00034 }
00035
00036
00039 void AutomateHandler::deleteAutomateHandler()
00040 {
00041     if(m_activeAutomateHandler)
00042     {
00043         delete m_activeAutomateHandler;
00044         m_activeAutomateHandler = nullptr;
00045     }
00046 }
00047
00048
00055 Automate * AutomateHandler::getAutomate(unsigned int indexAutomate){
00056     if(indexAutomate > m_ActiveAutomates.size())
00057         return nullptr;
00058     return m_ActiveAutomates.at(indexAutomate);
00059 }
00060
00061
00067 unsigned int AutomateHandler::getNumberAutomates()const
00068 {
00069     return m_ActiveAutomates.size();
00070 }
00071
00072
00078 void AutomateHandler::addAutomate(Automate * automate)
00079 {
00080     m_ActiveAutomates.append(automate);
00081 }
00082
00083
00089 void AutomateHandler::deleteAutomate(Automate * automate)
00090 {
00091     if(m_ActiveAutomates.contains(automate))
00092     {
00093         delete automate;
00094         m_ActiveAutomates.removeOne(automate);
00095     }
00096 }
```

7.7 automatehandler.h File Reference

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

Classes

- class AutomateHandler

  Implementation of singleton design pattern.

7.8 automatehandler.h

```
00001 #ifndef AUTOMATEHANDLER_H
00002 #define AUTOMATEHANDLER_H
00003
00004 #include "automate.h"
00005
00006
00010 class AutomateHandler
00011 {
00012 private:
00013     QList<Automate*> m_ActiveAutomates;
00014     static AutomateHandler * m_activeAutomateHandler;
00015
00016     AutomateHandler();
00017     AutomateHandler(const AutomateHandler & a) = delete;
00018     AutomateHandler & operator=(const AutomateHandler & a) = delete;
00019     ~AutomateHandler();
00020
00021 public:
00022     static AutomateHandler & getAutomateHandler();
00023     static void deleteAutomateHandler();
00024
00025     Automate * getAutomate(unsigned int indexAutomate);
00026     unsigned int getNumberAutomates()const;
00027
00028     void addAutomate(Automate * automate);
00029     void deleteAutomate(Automate * automate);
00030 };
00031
00032
00033 #endif // AUTOMATEHANDLER_H
```

7.9 cell.cpp File Reference

```
#include "cell.h"
```

7.10 cell.cpp

```
00001 #include "cell.h"
00002
00007 Cell::Cell(unsigned int state):
00008     m_nextState(state)
00009 {
00010     m_states.push(state);
00011 }
00012
00020 void Cell::setState(unsigned int state)
00021 {
00022     m_nextState = state;
00023 }
00024
00030 void Cell::validState()
00031 {
00032     m_states.push(m_nextState);
00033 }
00034
00041 void Cell::forceState(unsigned int state)
00042 {
00043     m_nextState = state;
00044     m_states.pop();
00045     m_states.push(m_nextState);
00046 }
00047
00050 unsigned int Cell::getState() const
00051 {
00052     return m_states.top();
00053 }
00054
00059 bool Cell::back()
00060 {
00061     if (m_states.size() <= 1)
```

```
00062         return false;
00063     m_states.pop();
00064     m_nextState = m_states.top();
00065     return true;
00066 }
00067
00070 void Cell::reset()
00071 {
00072     while (m_states.size() > 1)
00073         m_states.pop();
00074     m_nextState = m_states.top();
00075 }
00076
00084 bool Cell::addNeighbour(const Cell* neighbour, const QVector<short> relativePosition)
00085 {
00086     if (m_neighbours.count(relativePosition))
00087         return false;
00088
00089     m_neighbours.insert(relativePosition, neighbour);
00090     return true;
00091 }
00092
00097 QMap<QVector<short>, const Cell *> Cell::getNeighbours() const
00098 {
00099     return m_neighbours;
00100 }
00101
00104 const Cell *Cell::getNeighbour(QVector<short> relativePosition) const
00105 {
00106     return m_neighbours.value(relativePosition, nullptr);
00107 }
00108
00111 unsigned int Cell::countNeighbours(unsigned int filterState) const
00112 {
00113     unsigned int count = 0;
00114     for (QMap<QVector<short>, const Cell*>::const_iterator it = m_neighbours.begin(); it !=
      m_neighbours.end(); ++it)
00115     {
00116         if ((*it)->getState() == filterState)
00117             count++;
00118     }
00119     return count;
00120 }
00121
00124 unsigned int Cell::countNeighbours() const
00125 {
00126     unsigned int count = 0;
00127     for (QMap<QVector<short>, const Cell*>::const_iterator it = m_neighbours.begin(); it !=
      m_neighbours.end(); ++it)
00128     {
00129         if ((*it)->getState() != 0)
00130             count++;
00131     }
00132     return count;
00133 }
00134
00141 QVector<short> Cell::getRelativePosition(const QVector<unsigned int> cellPosition,
      const QVector<unsigned int> neighbourPosition)
00142 {
00143     if (cellPosition.size() != neighbourPosition.size())
00144     {
00145         throw QString(QObject::tr("Different size of position vectors"));
00146     }
00147     QVector<short> relativePosition;
00148     for (short i = 0; i < cellPosition.size(); i++)
00149         relativePosition.push_back(neighbourPosition.at(i) - cellPosition.at(i));
00150
00151     return relativePosition;
00152 }
```

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

```
00001 #ifndef CELL_H
00002 #define CELL_H
00003
00004 #include <QVector>
00005 #include <QDebug>
00006 #include <QStack>
00007
00011 class Cell
00012 {
00013 public:
00014     Cell(unsigned int state = 0);
00015
00016     void setState(unsigned int state);
00017     void validState();
00018     void forceState(unsigned int state);
00019     unsigned int getState() const;
00020
00021     bool back();
00022     void reset();
00023
00024     bool addNeighbour(const Cell* neighbour, const QVector<short> relativePosition);
00025     QMap<QVector<short>, const Cell*> getNeighbours() const;
00026     const Cell* getNeighbour(QVector<short> relativePosition) const;
00027
00028     unsigned int countNeighbours(unsigned int filterState) const;
00029     unsigned int countNeighbours() const;
00030
00031     static QVector<short> getRelativePosition(const QVector<unsigned int> cellPosition,
      const QVector<unsigned int> neighbourPosition);
00032
00033 private:
00034     QStack<unsigned int> m_states;
00035     unsigned int m_nextState;
00036
00037     QMap<QVector<short>, const Cell*> m_neighbours;
00038 };
00039
00040 #endif // CELL_H
```

7.13 cellhandler.cpp File Reference

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

7.14 cellhandler.cpp

```
00001 #include <iostream>
00002 #include "cellhandler.h"
00003
00025 CellHandler::CellHandler(const QString filename)
00026 {
00027     QFile loadFile(filename);
00028     if (!loadFile.open(QIODevice::ReadOnly | QIODevice::Text)) {
00029         qWarning("Couldn't open given file.");
00030         throw QString(QObject::tr("Couldn't open given file"));
00031     }
00032
00033     QJsonParseError parseErr;
00034     QJsonDocument loadDoc(QJsonDocument::fromJson(loadFile.readAll(), &parseErr));
```

```
00035
00036     loadFile.close();
00037
00038
00039     if (loadDoc.isNull() || loadDoc.isEmpty()) {
00040         qWarning() << "Could not read data : ";
00041         qWarning() << parseErr.errorString();
00042         throw QString(parseErr.errorString());
00043     }
00044
00045     // Loadding of the json file
00046     if (!load(loadDoc.object()))
00047     {
00048         qWarning("File not valid");
00049         throw QString(QObject::tr("File not valid"));
00050     }
00051
00052     foundNeighbours();
00053
00054
00055 }
00056
00076 CellHandler::CellHandler(const QJsonObject& json)
00077 {
00078     if (!load(json))
00079     {
00080         qWarning("Json not valid");
00081         throw QString(QObject::tr("Json not valid"));
00082     }
00083
00084     foundNeighbours();
00085
00086 }
00087
00088
00098 CellHandler::CellHandler(const QVector<unsigned int> dimensions,
      generationTypes type, unsigned int stateMax, unsigned int density)
00099 {
00100     m_dimensions = dimensions;
00101     QVector<unsigned int> position;
00102     unsigned int size = 1;
00103
00104     // Set position vector to 0
00105
00106     for (unsigned short i = 0; i < m_dimensions.size(); i++)
00107     {
00108         position.push_back(0);
00109         size *= m_dimensions.at(i);
00110     }
00111
00112
00113     // Creation of cells
00114     for (unsigned int j = 0; j < size; j++)
00115     {
00116         m_cells.insert(position, new Cell(0));
00117
00118         positionIncrement(position);
00119     }
00120
00121     foundNeighbours();
00122
00123     if (type != empty)
00124         generate(type, stateMax, density);
00125
00126 }
00127
00130 CellHandler::~CellHandler()
00131 {
00132     for (QMap<QVector<unsigned int>, Cell* >::iterator it = m_cells.begin(); it !=
      m_cells.end(); ++it)
00133     {
00134         delete it.value();
00135     }
00136 }
00137
00140 Cell *CellHandler::getCell(const QVector<unsigned int> position) const
00141 {
00142     return m_cells.value(position);
00143 }
00144
00147 unsigned int CellHandler::getMaxState()
00148 {
00149     return QColor::colorNames().size()-2;
00150 }
00151
00154 QVector<unsigned int> CellHandler::getDimensions() const
00155 {
```

```
00156     return m_dimensions;
00157 }
00158
00161 void CellHandler::nextStates() const
00162 {
00163     for (QMap<QVector<unsigned int>, Cell* >::const_iterator it =
      m_cells.begin(); it != m_cells.end(); ++it)
00164     {
00165         it.value()->validState();
00166     }
00167 }
00168
00171 bool CellHandler::previousStates() const
00172 {
00173     for (QMap<QVector<unsigned int>, Cell* >::const_iterator it =
      m_cells.begin(); it != m_cells.end(); ++it)
00174     {
00175         if (!it.value()->back())
00176             return false;
00177     }
00178     return true;
00179 }
00180
00183 void CellHandler::reset() const
00184 {
00185     for (QMap<QVector<unsigned int>, Cell* >::const_iterator it =
      m_cells.begin(); it != m_cells.end(); ++it)
00186     {
00187         it.value()->reset();
00188     }
00189 }
00190
00198 bool CellHandler::save(QString filename) const
00199 {
00200     QFile saveFile(filename);
00201     if (!saveFile.open(QIODevice::WriteOnly)) {
00202         qWarning("Couldn't create or open given file.");
00203         throw QString(QObject::tr("Couldn't create or open given file"));
00204     }
00205
00206     QJsonObject json;
00207     QString stringDimension;
00208     // Creation of the dimension string
00209     for (int i = 0; i < m_dimensions.size(); i++)
00210     {
00211         if (i != 0)
00212             stringDimension.push_back("x");
00213         stringDimension.push_back(QString::number(m_dimensions.at(i)));
00214     }
00215     json["dimensions"] = QJsonValue(stringDimension);
00216
00217     QJsonArray cells;
00218     for (CellHandler::const_iterator it = begin(); it !=
      end(); ++it)
00219     {
00220         cells.append(QJsonValue((int)it->getState()));
00221     }
00222     json["cells"] = cells;
00223
00224     //json["maxState"] = QJsonValue((int)m_maxState);
00225
00226
00227     QJsonDocument saveDoc(json);
00228     saveFile.write(saveDoc.toJson());
00229
00230     saveFile.close();
00231     return true;
00232 }
00233
00240 void CellHandler::generate(CellHandler::generationTypes
      type, unsigned int stateMax, unsigned short density)
00241 {
00242     if (type == random)
00243     {
00244         QVector<unsigned int> position;
00245         for (unsigned short i = 0; i < m_dimensions.size(); i++)
00246         {
00247             position.push_back(0);
00248         }
00249         QRandomGenerator generator((float)qrand()*(float)time_t()/RAND_MAX);
00250         for (int j = 0; j < m_cells.size(); j++)
00251         {
00252             unsigned int state = 0;
00253             // 0 have (1-density)% of chance of being generate
00254             if (generator.generateDouble()*100.0 < density)
00255                 state = (float)(generator.generateDouble()*stateMax) +1;
00256             if (state > stateMax)
```

```
00257                 state = stateMax;
00258             m_cells.value(position)->forceState(state);
00259 
00260             positionIncrement(position);
00261         }
00262     }
00263     else if (type == symetric)
00264     {
00265         QVector<unsigned int> position;
00266         for (short i = 0; i < m_dimensions.size(); i++)
00267         {
00268             position.push_back(0);
00269         }
00270 
00271         QRandomGenerator generator((float)qrand()*(float)time_t()/RAND_MAX);
00272         QVector<unsigned int> savedStates;
00273         for (int j = 0; j < m_cells.size(); j++)
00274         {
00275             if (j % m_dimensions.at(0) == 0)
00276                 savedStates.clear();
00277             if (j % m_dimensions.at(0) < (m_dimensions.at(0)+1) / 2)
00278             {
00279                 unsigned int state = 0;
00280                 // 0 have (1-density)% of chance of being generate
00281                 if (generator.generateDouble()*100.0 < density)
00282                     state = (float)(generator.generateDouble()*stateMax) +1;
00283                 if (state > stateMax)
00284                     state = stateMax;
00285                 savedStates.push_back(state);
00286                 m_cells.value(position)->forceState(state);
00287             }
00288             else
00289             {
00290                 unsigned int i = savedStates.size() - (j % m_dimensions.at(0) - (
      m_dimensions.at(0)-1)/2 + (m_dimensions.at(0) % 2 == 0 ? 0 : 1));
00291                 m_cells.value(position)->forceState(savedStates.at(i));
00292             }
00293             positionIncrement(position);
00294 
00295 
00296         }
00297 
00298     }
00299 }
00300 
00305 void CellHandler::print(std::ostream &stream) const
00306 {
00307     for (const_iterator it = begin(); it != end(); ++it)
00308     {
00309         for (unsigned int d = 0; d < it.changedDimension(); d++)
00310             stream << std::endl;
00311         stream << it->getState() << " ";
00312 
00313     }
00314 
00315 }
00316 
00319 CellHandler::iterator CellHandler::begin()
00320 {
00321     return iterator(this);
00322 }
00323 
00326 CellHandler::const_iterator CellHandler::begin() const
00327 {
00328     return const_iterator(this);
00329 }
00330 
00335 bool CellHandler::end() const
00336 {
00337     return true;
00338 }
00339 
00370 bool CellHandler::load(const QJsonObject &json)
00371 {
00372     if (!json.contains("dimensions") || !json["dimensions"].isString())
00373         return false;
00374 
00375     // RegExp to validate dimensions field format : "10x10"
00376     QRegExpValidator dimensionValidator(QRegExp("([0-9]*x?)*"));
00377     QString stringDimensions = json["dimensions"].toString();
00378     int pos= 0;
00379     if (dimensionValidator.validate(stringDimensions, pos) != QRegExpValidator::Acceptable)
00380         return false;
00381 
00382     // Split of dimensions field : "10x10" => "10", "10"
00383     QRegExp rx("x");
00384     QStringList list = json["dimensions"].toString().split(rx, QString::SkipEmptyParts);
```

```
00385
00386     int product = 1;
00387     // Dimensions construction
00388     for (int i = 0; i < list.size(); i++)
00389     {
00390         product = product * list.at(i).toInt();
00391         m_dimensions.push_back(list.at(i).toInt());
00392     }
00393     if (!json.contains("cells") || !json["cells"].isArray())
00394         return false;
00395
00396     QJsonArray cells = json["cells"].toArray();
00397     if (cells.size() != product)
00398         return false;
00399
00400     QVector<unsigned int> position;
00401     // Set position vector to 0
00402     for (unsigned short i = 0; i < m_dimensions.size(); i++)
00403     {
00404         position.push_back(0);
00405     }
00406
00407
00408     // Creation of cells
00409     for (int j = 0; j < cells.size(); j++)
00410     {
00411         if (!cells.at(j).isDouble())
00412             return false;
00413         if (cells.at(j).toDouble() < 0)
00414             return false;
00415         m_cells.insert(position, new Cell(cells.at(j).toDouble()));
00416
00417         positionIncrement(position);
00418     }
00419
00420     //if (!json.contains("maxState") || !json["maxState"].isDouble())
00421     //    return false;
00422     //m_maxState = json["maxState"].toInt();
00423
00424     return true;
00425
00426 }
00427
00433 void CellHandler::foundNeighbours()
00434 {
00435     QVector<unsigned int> currentPosition;
00436     // Set position vector to 0
00437     for (unsigned short i = 0; i < m_dimensions.size(); i++)
00438     {
00439         currentPosition.push_back(0);
00440     }
00441     // Modification of all the cells
00442     for (int j = 0; j < m_cells.size(); j++)
00443     {
00444         // Get the list of the neighbours positions
00445         // This function is recursive
00446         QVector<QVector<unsigned int> > listPosition(getListNeighboursPositions(
      currentPosition));
00447
00448         // Adding neighbours
00449         for (int i = 0; i < listPosition.size(); i++)
00450             m_cells.value(currentPosition)->addNeighbour(m_cells.value(listPosition.at(i)),
      Cell::getRelativePosition(currentPosition, listPosition.at(i)));
00451         positionIncrement(currentPosition);
00452     }
00453
00454 }
00455
00463 void CellHandler::positionIncrement(QVector<unsigned int> &pos, unsigned int
      value) const
00464 {
00465     pos.replace(0, pos.at(0) + value); // adding the value to the first digit
00466
00467     // Carry management
00468     for (unsigned short i = 0; i < m_dimensions.size(); i++)
00469     {
00470         if (pos.at(i) >= m_dimensions.at(i) && pos.at(i) <
      m_dimensions.at(i)*2)
00471         {
00472             pos.replace(i, 0);
00473             if (i + 1 != m_dimensions.size())
00474                 pos.replace(i+1, pos.at(i+1)+1);
00475         }
00476         else if (pos.at(i) >= m_dimensions.at(i))
00477         {
00478             pos.replace(i, pos.at(i) - m_dimensions.at(i));
00479             if (i + 1 != m_dimensions.size())
```

```
00480                 pos.replace(i+1, pos.at(i+1)+1);
00481             i--;
00482         }
00483
00484     }
00485 }
00486
00492 QVector<QVector<unsigned int> >& CellHandler::getListNeighboursPositions
      (const QVector<unsigned int> position) const
00493 {
00494     QVector<QVector<unsigned int> > *list = getListNeighboursPositionsRecursive
      (position, position.size(), position);
00495     // We remove the position of the cell
00496     list->removeAll(position);
00497     return *list;
00498 }
00499
00533 QVector<QVector<unsigned int> >*
      CellHandler::getListNeighboursPositionsRecursive(const
      QVector<unsigned int> position, unsigned int dimension, QVector<unsigned int> lastAdd) const
00534 {
00535     if (dimension == 0) // Stop condition
00536     {
00537         QVector<QVector<unsigned int> > *list = new QVector<QVector<unsigned int> >;
00538         return list;
00539     }
00540     QVector<QVector<unsigned int> > *listPositions = new QVector<QVector<unsigned int> >;
00541
00542     QVector<unsigned int> modifiedPosition(lastAdd);
00543
00544     // "x_d - 1" tree
00545     if (modifiedPosition.at(dimension-1) != 0) // Avoid "negative" position
00546         modifiedPosition.replace(dimension-1, position.at(dimension-1) - 1);
00547     listPositions->append(*getListNeighboursPositionsRecursive(position,
       dimension -1, modifiedPosition));
00548     if (!listPositions->count(modifiedPosition))
00549         listPositions->push_back(modifiedPosition);
00550
00551     // "x_d" tree
00552     modifiedPosition.replace(dimension-1, position.at(dimension-1));
00553     listPositions->append(*getListNeighboursPositionsRecursive(position,
       dimension -1, modifiedPosition));
00554     if (!listPositions->count(modifiedPosition))
00555         listPositions->push_back(modifiedPosition);
00556
00557     // "x_d + 1" tree
00558     if (modifiedPosition.at(dimension -1) + 1 < m_dimensions.at(dimension-1)) // Avoid position
       out of the cell space
00559         modifiedPosition.replace(dimension-1, position.at(dimension-1) +1);
00560     listPositions->append(*getListNeighboursPositionsRecursive(position,
       dimension -1, modifiedPosition));
00561     if (!listPositions->count(modifiedPosition))
00562         listPositions->push_back(modifiedPosition);
00563
00564     return listPositions;
00565
00566 }
00567
00572 template<typename CellHandler_T, typename Cell_T>
00573 CellHandler::iteratorT<CellHandler_T,Cell_T>::iteratorT
      (CellHandler_T *handler):
00574         m_handler(handler), m_changedDimension(0)
00575 {
00576     // Initialisation of m_position
00577     for (unsigned short i = 0; i < handler->m_dimensions.size(); i++)
00578     {
00579         m_position.push_back(0);
00580     }
00581     m_zero = m_position;
00582 }
```

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

```
00001 #ifndef CELLHANDLER_H
00002 #define CELLHANDLER_H
00003
00004 #include <QString>
00005 #include <QFile>
00006 #include <QJsonDocument>
00007 #include <QtWidgets>
00008 #include <QMap>
00009 #include <QRegExpValidator>
00010 #include <QDebug>
00011
00012 #include "cell.h"
00013
00014
00015
00020 class CellHandler
00021 {
00022
00040     template <typename CellHandler_T, typename Cell_T>
00041     class iteratorT
00042     {
00043         friend class CellHandler;
00044     public:
00045         iteratorT(CellHandler_T* handler);
00047         iteratorT& operator++(){
00048             m_position.replace(0, m_position.at(0) + 1); // adding the value to the
      first digit
00049
00050             m_changedDimension = 0;
00051             // Carry management
00052             for (unsigned short i = 0; i < m_handler->m_dimensions.size(); i++)
00053             {
00054                 if (m_position.at(i) >= m_handler->m_dimensions.at(i))
00055                 {
00056                     m_position.replace(i, 0);
00057                     m_changedDimension++;
00058                     if (i + 1 != m_handler->m_dimensions.size())
00059                         m_position.replace(i+1, m_position.at(i+1)+1);
00060                 }
00061
00062             }
00063             // If we return to zero, we have finished
00064             if (m_position == m_zero)
00065                 m_finished = true;
00066
00067             return *this;
00068
00069         }
00071         Cell_T* operator->() const{
00072             return m_handler->m_cells.value(m_position);
00073         }
00075         Cell_T* operator*() const{
00076             return m_handler->m_cells.value(m_position);
00077         }
00078
00079         bool operator!=(bool finished) const { return (m_finished != finished); }
00080         unsigned int changedDimension() const{
00081             return m_changedDimension;
00082         }
00083
00084
```

```
00085
00086     private:
00087         CellHandler_T *m_handler;
00088         QVector<unsigned int> m_position;
00089         bool m_finished = false;
00090         QVector<unsigned int> m_zero;
00091         unsigned int m_changedDimension;
00092     };
00093 public:
00094     typedef iteratorT<const CellHandler, const Cell>
      const_iterator;
00095     typedef iteratorT<CellHandler, Cell> iterator;
00096
00099     enum generationTypes {
00100         empty,
00101         random,
00102         symetric
00103     };
00104
00105     CellHandler(const QString filename);
00106     CellHandler(const QJsonObject &json);
00107     CellHandler(const QVector<unsigned int> dimensions,
      generationTypes type = empty, unsigned int stateMax = 1, unsigned int density = 20);
00108     virtual ~CellHandler();
00109
00110     Cell* getCell(const QVector<unsigned int> position) const;
00111     static unsigned int getMaxState();
00112     QVector<unsigned int> getDimensions() const;
00113     void nextStates() const;
00114     bool previousStates() const;
00115     void reset() const;
00116
00117     bool save(QString filename) const;
00118
00119     void generate(generationTypes type, unsigned int stateMax = 1, unsigned short
      density = 50);
00120     void print(std::ostream &stream) const;
00121
00122     const_iterator begin() const;
00123     iterator begin();
00124     bool end() const;
00125
00126
00127 private:
00128     bool load(const QJsonObject &json);
00129     void foundNeighbours();
00130     void positionIncrement(QVector<unsigned int> &pos, unsigned int value = 1) const;
00131     QVector<QVector<unsigned int> > *getListNeighboursPositionsRecursive
      (const QVector<unsigned int> position, unsigned int dimension, QVector<unsigned int> lastAdd) const;
00132     QVector<QVector<unsigned int> > &getListNeighboursPositions(const
      QVector<unsigned int> position) const;
00133
00134     QVector<unsigned int> m_dimensions;
00135     QMap<QVector<unsigned int>, Cell* > m_cells;
00136 };
00137
00138 template class CellHandler::iteratorT<CellHandler, Cell>;
00139 template class CellHandler::iteratorT<const CellHandler, const Cell>
      ;
00140
00141 #endif // CELLHANDLER_H
```

7.17 creationdialog.cpp File Reference

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

7.18 creationdialog.cpp

```
00001 #include "creationdialog.h"
00002 #include <iostream>
00003
00004
00005 CreationDialog::CreationDialog(QWidget *parent)
```

```
00006 {
00007     QLabel *m_dimLabel= new QLabel(tr("Write your dimensions and their size, separated by a comma.\n"
00008                                       "For instance, '25,25 ' will create a 2-dimensional 25x25 Automaton. "));
00009     QLabel *m_densityLabel = new QLabel(tr("Density :"));
00010     QLabel *m_stateMaxLabel = new QLabel(tr("Max state :"));
00011     m_densityBox = new QSpinBox();
00012     m_densityBox->setValue(20);
00013     m_stateMaxBox = new QSpinBox();
00014     m_stateMaxBox->setValue(1);
00015 
00016     QHBoxLayout *densityLayout = new QHBoxLayout();
00017     densityLayout->addWidget(m_densityLabel);
00018     densityLayout->addWidget(m_densityBox);
00019 
00020     QHBoxLayout *stateMaxLayout = new QHBoxLayout();
00021     stateMaxLayout->addWidget(m_stateMaxLabel);
00022     stateMaxLayout->addWidget(m_stateMaxBox);
00023 
00024     m_dimensionsEdit = new QLineEdit;
00025     QRegExp rgx("([0-9]+,)*");
00026     QRegExpValidator *v = new QRegExpValidator(rgx, this);
00027     m_dimensionsEdit->setValidator(v);
00028     m_doneBt = new QPushButton(tr("Done !"));
00029 
00030     QVBoxLayout *layout = new QVBoxLayout;
00031 
00032     QGroupBox *grpBox = createGenButtons();
00033 
00034     layout->addWidget(m_dimLabel);
00035     layout->addWidget(m_dimensionsEdit);
00036     layout->addLayout(densityLayout);
00037     layout->addLayout(stateMaxLayout);
00038     layout->addWidget(grpBox);
00039     layout->addWidget(m_doneBt);
00040     setLayout(layout);
00041 
00042     connect(m_doneBt, SIGNAL(clicked(bool)), this, SLOT(processSettings()));
00043 
00044 }
00045 
00051 QGroupBox *CreationDialog::createGenButtons(){
00052     m_groupBox = new QGroupBox(tr("Cell generation settings"));
00053     m_empGen = new QRadioButton(tr("&Empty Board"));
00054     m_randGen = new QRadioButton(tr("&Random"));
00055     m_symGen = new QRadioButton(tr("&Symmetrical"));
00056 
00057     QVBoxLayout *layout = new QVBoxLayout;
00058     layout->addWidget(m_empGen);
00059     layout->addWidget(m_randGen);
00060     layout->addWidget(m_symGen);
00061 
00062     m_groupBox->setLayout(layout);
00063 
00064     return m_groupBox;
00065 }
00066 
00072 void CreationDialog::processSettings(){
00073     QString dimensions = m_dimensionsEdit->text();
00074     if(dimensions.length() == 0){
00075         QMessageBox messageBox;
00076         messageBox.critical(0,"Error","You must specify valid dimensions !");
00077         messageBox.setFixedSize(500,200);
00078     }
00079     else{
00080         CellHandler::generationTypes genType;
00081         if(m_symGen->isChecked()) genType = CellHandler::generationTypes::symetric;
00082         else if(m_randGen->isChecked()) genType = CellHandler::generationTypes::random;
00083         else genType = CellHandler::generationTypes::empty;
00084         QStringList dimList = m_dimensionsEdit->text().split(",");
00085         QVector<unsigned int> dimensions;
00086         for(int i = 0; i < dimList.size(); i++) dimensions.append(dimList.at(i).toInt());
00087 
00088         emit settingsFilled(dimensions, genType, m_stateMaxBox->value(),
      m_densityBox->value());
00089         this->close();
00090     }
00091 
00092 }
00093 
```

7.19 creationdialog.h File Reference

```
#include <QtWidgets>
```

```
#include "cellhandler.h"
```

Classes

- class CreationDialog

  Automaton creation dialog box.

7.20 creationdialog.h

```
00001 #ifndef CREATIONDIALOG_H
00002 #define CREATIONDIALOG_H
00003
00004 #include <QtWidgets>
00005 #include "cellhandler.h"
00006
00013 class CreationDialog : public QDialog
00014 {
00015     Q_OBJECT
00016
00017 public:
00018     CreationDialog(QWidget *parent = 0);
00019
00020 signals:
00021     void settingsFilled(const QVector<unsigned int> dimensions,
00022                         CellHandler::generationTypes type =
      CellHandler::generationTypes::empty,
00023                         unsigned int stateMax = 1, unsigned int density = 20);
00024
00025 public slots:
00026     void processSettings();
00027
00028 private:
00029     QLineEdit *m_dimensionsEdit;
00030     QSpinBox *m_densityBox;
00031     QSpinBox *m_stateMaxBox;
00032     QPushButton *m_doneBt;
00033
00034     QGroupBox *m_groupBox;
00035     QRadioButton *m_empGen;
00036     QRadioButton *m_randGen;
00037     QRadioButton *m_symGen;
00038
00039     QGroupBox *createGenButtons();
00040
00041
00042
00043
00044
00045
00046 };
00047
00048 #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
00012     app.setOrganizationName("LO21-project");
00013     app.setApplicationName("AutoCell");
00014
00015     MainWindow w;
00016     w.show();
00017
00018     return app.exec();
00019
00020 }
```

7.23 mainwindow.cpp File Reference

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

7.24 mainwindow.cpp

```
00001 #include "mainwindow.h"
00002 #include <iostream>
00003 #include "math.h"
00004 MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent)
00005 {
00006     createIcons();
00007     createActions();
00008     createToolBar();
00009
00010
00011     setMinimumSize(500,500);
00012     setWindowTitle("AutoCell");
00013
00014     m_tabs = NULL;
00015     running = false;
00016
```

```
00017     QSettings settings;
00018     int nbAutomate = settings.value("nbAutomate").toInt();
00019     for (unsigned int i = 0; i < nbAutomate; i++)
00020     {
00021         QString fileName = QString(".automate"+QString::number(i));
00022         try{
00023             AutomateHandler::getAutomateHandler().
      addAutomate(new Automate(QString(fileName+".atc"), QString(fileName+".atr")));
00024         if(m_tabs == NULL)
00025             createTabs();
00026         m_tabs->addTab(createTab(), "Automaton "+ QString::number(
      AutomateHandler::getAutomateHandler().getNumberAutomates()));
00027         updateBoard(AutomateHandler::getAutomateHandler().
      getNumberAutomates()-1);
00028         }
00029         catch (QString &s)
00030         {
00031             QMessageBox msgBox;
00032             msgBox.warning(0,"Error",s);
00033             msgBox.setFixedSize(500,200);
00034         }
00035         QFile fichier(QString(fileName + ".atc"));
00036         fichier.remove();
00037         fichier.close();
00038         QFile fichier2(QString(fileName + ".atr"));
00039         fichier2.remove();
00040     }
00041     m_zoom->setValue(settings.value("zoom").toInt());
00042     m_timeStep->setValue(settings.value("timestamp").toInt());
00043 }
00044
00045 MainWindow::~MainWindow()
00046 {
00047     // Saving settings for further sessions
00048     QSettings settings;
00049     settings.setValue("nbAutomate", AutomateHandler::getAutomateHandler(
      ).getNumberAutomates());
00050     settings.setValue("zoom", m_zoom->value());
00051     settings.setValue("timestamp", m_timeStep->value());
00052
00053     for (unsigned int i = 0; i < AutomateHandler::getAutomateHandler().
      getNumberAutomates(); i++)
00054     {
00055         AutomateHandler::getAutomateHandler().
      getAutomate(i)->saveAll(QString(".automate"+QString::number(i)+".atc"), QString("
      .automate"+QString::number(i)+".atr"));
00056     }
00057
00058 }
00059
00065 void MainWindow::createIcons(){
00066     QPixmap fastBackwardPm(":/icons/icons/fast-backward.svg");
00067     QPixmap fastBackwardHoveredPm(":/icons/icons/fast-backward-full.svg");
00068     QPixmap fastForwardPm(":/icons/icons/fast-forward.svg");
00069     QPixmap fastForwardHoveredPm(":/icons/icons/fast-forward-full.svg");
00070     QPixmap playPm(":/icons/icons/play.svg");
00071     QPixmap playHoveredPm(":/icons/icons/play-full.svg");
00072     QPixmap newPm(":/icons/icons/new.svg");
00073     QPixmap openPm(":/icons/icons/open.svg");
00074     QPixmap savePm(":/icons/icons/save.svg");
00075     QPixmap pausePm(":/icons/icons/pause.svg");
00076     QPixmap resetPm(":/icons/icons/reset.svg");
00077
00078     m_fastBackwardIcon.addPixmap(fastBackwardPm, QIcon::Normal, QIcon::Off);
00079     m_fastBackwardIcon.addPixmap(fastBackwardHoveredPm, QIcon::Active, QIcon::Off);
00080     m_fastForwardIcon.addPixmap(fastForwardPm, QIcon::Normal, QIcon::Off);
00081     m_fastForwardIcon.addPixmap(fastForwardHoveredPm, QIcon::Active, QIcon::Off);
00082     m_playIcon.addPixmap(playPm, QIcon::Normal, QIcon::Off);
00083     m_playIcon.addPixmap(playHoveredPm, QIcon::Active, QIcon::Off);
00084     m_pauseIcon.addPixmap(pausePm, QIcon::Normal, QIcon::Off);
00085     m_newIcon.addPixmap(newPm, QIcon::Normal, QIcon::Off);
00086     m_saveIcon.addPixmap(savePm, QIcon::Normal, QIcon::Off);
00087     m_openIcon.addPixmap(openPm, QIcon::Normal, QIcon::Off);
00088     m_resetIcon.addPixmap(resetPm, QIcon::Normal, QIcon::Off);
00089 }
00090
00095 void MainWindow::createActions(){
00096     m_fastBackward = new QAction(m_fastBackwardIcon, tr("&Previous state"),
       this);
00097     m_fastForward = new QAction(m_fastForwardIcon, tr("&Next state"), this);
00098     m_playPause = new QAction(m_playIcon, tr("Play"), this);
00099     m_saveAutomaton = new QAction(m_saveIcon, tr("Save automaton"), this);
00100     m_newAutomaton = new QAction(m_newIcon, tr("New automaton"), this);
00101     m_openAutomaton = new QAction(m_openIcon, tr("Open automaton"), this);
00102     m_resetAutomaton = new QAction(m_resetIcon, tr("Reset automaton"), this);
00103
00104     m_fastBackwardBt = new QToolButton(this);
```

```
00105     m_fastForwardBt = new QToolButton(this);
00106     m_playPauseBt = new QToolButton(this);
00107     m_saveAutomatonBt = new QToolButton(this);
00108     m_newAutomatonBt = new QToolButton(this);
00109     m_openAutomatonBt = new QToolButton(this);
00110     m_resetBt = new QToolButton(this);
00111
00112     m_fastBackwardBt->setDefaultAction(m_fastBackward);
00113     m_fastForwardBt->setDefaultAction(m_fastForward);
00114     m_playPauseBt->setDefaultAction(m_playPause);
00115     m_saveAutomatonBt->setDefaultAction(m_saveAutomaton);
00116     m_newAutomatonBt->setDefaultAction(m_newAutomaton);
00117     m_openAutomatonBt->setDefaultAction(m_openAutomaton);
00118     m_resetBt->setDefaultAction(m_resetAutomaton);
00119
00120     m_fastBackwardBt->setIconSize(QSize(30,30));
00121     m_fastForwardBt->setIconSize(QSize(30,30));
00122     m_playPauseBt->setIconSize(QSize(30,30));
00123     m_saveAutomatonBt->setIconSize(QSize(30,30));
00124     m_newAutomatonBt->setIconSize(QSize(30,30));
00125     m_openAutomatonBt->setIconSize(QSize(30,30));
00126     m_resetBt->setIconSize(QSize(30,30));
00127
00128
00129     m_zoom = new QSlider(Qt::Horizontal);
00130     m_zoom->setValue(m_cellSize);
00131     m_zoom->setMinimum(4);
00132     m_zoom->setMaximum(100);
00133     m_zoom->setFixedWidth(100);
00134
00135
00136     connect(m_openAutomatonBt, SIGNAL(clicked(bool)), this, SLOT(
      openFile()));
00137     connect(m_newAutomatonBt, SIGNAL(clicked(bool)), this, SLOT(
      openCreationWindow()));
00138     connect(m_saveAutomatonBt, SIGNAL(clicked(bool)), this, SLOT(
      saveToFile()));
00139     connect(m_fastForwardBt, SIGNAL(clicked(bool)), this, SLOT(
      forward()));
00140     connect(m_fastBackwardBt, SIGNAL(clicked(bool)), this, SLOT(
      backward()));
00141     connect(m_playPauseBt, SIGNAL(clicked(bool)), this, SLOT(
      handlePlayPause()));
00142     connect(m_resetBt,SIGNAL(clicked(bool)), this,SLOT(reset()));
00143     connect(m_zoom, SIGNAL(valueChanged(int)), this, SLOT(setSize(int)));
00144
00145 }
00146
00151 void MainWindow::createToolBar(){
00152     m_toolBar = new QToolBar(this);
00153     QLabel *timeStepLabel = new QLabel(tr("Timestep(ms)"),this);
00154     m_timeStep = new QSpinBox(this);
00155     m_timeStep->setMaximum(10000);
00156     m_timeStep->setValue(500);
00157     timeStepLabel->setFixedWidth(90);
00158     m_timeStep->setFixedWidth(60);
00159     m_toolBar->setMovable(false);
00160
00161     QLabel *cellSetterLabel = new QLabel(tr("Cell value"));
00162     m_cellSetter = new QSpinBox(this);
00163     connect(m_cellSetter, SIGNAL(valueChanged(int)),this, SLOT(
      changeCellValue()));
00164     QLabel *zoomLabel = new QLabel(tr("Zoom"),this);
00165     QVBoxLayout* zoomLayout = new QVBoxLayout();
00166     zoomLayout->addWidget(zoomLabel, Qt::AlignCenter);
00167     zoomLayout->addWidget(m_zoom, Qt::AlignCenter);
00168
00169     QVBoxLayout* tsLayout = new QVBoxLayout();
00170     tsLayout->addWidget(timeStepLabel, Qt::AlignCenter);
00171     tsLayout->addWidget(m_timeStep, Qt::AlignCenter);
00172
00173     QVBoxLayout* csLayout = new QVBoxLayout();
00174     csLayout->addWidget(cellSetterLabel, Qt::AlignCenter);
00175     csLayout->addWidget(m_cellSetter, Qt::AlignCenter);
00176
00177     QHBoxLayout *tbLayout = new QHBoxLayout(this);
00178     tbLayout->addLayout(zoomLayout);
00179     tbLayout->addWidget(m_newAutomatonBt, Qt::AlignCenter);
00180     tbLayout->addWidget(m_openAutomatonBt, Qt::AlignCenter);
00181     tbLayout->addWidget(m_saveAutomatonBt, Qt::AlignCenter);
00182     tbLayout->addWidget(m_fastBackwardBt, Qt::AlignCenter);
00183     tbLayout->addWidget(m_playPauseBt, Qt::AlignCenter);
00184     tbLayout->addWidget(m_fastForwardBt, Qt::AlignCenter);
00185     tbLayout->addWidget(m_resetBt, Qt::AlignCenter);
00186     tbLayout->addLayout(tsLayout);
00187     tbLayout->addLayout(csLayout);
00188
```

```
00189
00190
00191     tbLayout->setAlignment(Qt::AlignCenter);
00192     QWidget* wrapper = new QWidget(this);
00193     wrapper->setLayout(tbLayout);
00194     m_toolBar->addWidget(wrapper);
00195     addToolBar(m_toolBar);
00196
00197
00198 }
00199
00204 QWidget* MainWindow::createTab(){
00205     QWidget *tab = new QWidget(this);
00206     QVBoxLayout *layout = new QVBoxLayout(this);
00207     QVector<unsigned int> dimensions = AutomateHandler::getAutomateHandler
      ().getAutomate(AutomateHandler::getAutomateHandler().
      getNumberAutomates()-1)->getCellHandler().getDimensions();
00208     int boardVSize = 0;
00209     int boardHSize = 0;
00210     if(dimensions.size() > 1){
00211         boardVSize = dimensions[0];
00212         boardHSize = dimensions[1];
00213     }
00214     else{
00215         boardVSize = 1;
00216         boardHSize = dimensions[0];
00217     }
00218
00219     QTableWidget* board = new QTableWidget(boardVSize, boardHSize, this);
00220         board->setFixedSize(boardHSize*m_cellSize,boardVSize*
      m_cellSize);
00221         //setMinimumSize(m_boardHSize*m_cellSize,100+m_boardVSize*m_cellSize);
00222         board->horizontalHeader()->setVisible(false);
00223         board->verticalHeader()->setVisible(false);
00224         board->setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
00225         board->setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
00226         board->setEditTriggers(QAbstractItemView::NoEditTriggers);
00227         for(unsigned int col = 0; col < boardHSize; ++col)
00228             board->setColumnWidth(col, m_cellSize);
00229         for(unsigned int row = 0; row < boardVSize; ++row) {
00230             board->setRowHeight(row, m_cellSize);
00231             for(unsigned int col = 0; col < boardHSize; ++col) {
00232                 board->setItem(row, col, new QTableWidgetItem(""));
00233                 board->item(row, col)->setBackgroundColor("white");
00234                 board->item(row, col)->setTextColor("black");
00235             }
00236         }
00237      QScrollArea *scrollArea = new QScrollArea(this);
00238      scrollArea->setWidget(board);
00239
00240      layout->setContentsMargins(0,0,0,0);
00241      layout->addWidget(scrollArea);
00242      tab->setLayout(layout);
00243      connect(board, SIGNAL(cellClicked(int,int)), this, SLOT(cellPressed(int,int)));
00244      return tab;
00245 }
00246
00250 void MainWindow::openFile(){
00251     QString fileName = QFileDialog::getOpenFileName(this, tr("Open Cell file"), ".",
00252                                                     tr("Automaton cell files (*.atc)"));
00253     if(!fileName.isEmpty()){
00254         AutomateHandler::getAutomateHandler().
      addAutomate(new Automate(fileName));
00255         if(m_tabs == NULL) createTabs();
00256         m_tabs->addTab(createTab(), "Automaton "+ QString::number(
      AutomateHandler::getAutomateHandler().getNumberAutomates()+1));
00257         updateBoard(AutomateHandler::getAutomateHandler().
      getNumberAutomates()-1);
00258
00259         RuleEditor* ruleEditor = new RuleEditor(
      AutomateHandler::getAutomateHandler().getAutomate(
      AutomateHandler::getAutomateHandler().getNumberAutomates()-1)->
      getCellHandler().getDimensions().size(), this);
00260         connect(ruleEditor, SIGNAL(fileImported(QString)),this,SLOT(
      addAutomatonRuleFile(QString)));
00261         connect(ruleEditor, SIGNAL(rulesFilled(QList<const NeighbourRule*>)), this, SLOT(
      addAutomatonRules(QList<const NeighbourRule*>)));
00262         ruleEditor->show();
00263     }
00264 }
00265
00266
00270 void MainWindow::saveToFile(){
00271     if(AutomateHandler::getAutomateHandler().getNumberAutomates() > 0){
00272         QString automatonFileName = QFileDialog::getSaveFileName(this, tr("Save Automaton cell
       configuration"),
00273                                                     ".", tr("Automaton Cells file (*.atc"));
```

```
00274         AutomateHandler::getAutomateHandler().
      getAutomate(m_tabs->currentIndex())->saveCells(automatonFileName+".atc");
00275         QString ruleFileName = QFileDialog::getSaveFileName(this, tr("Save Automaton rules"),
00276                                                             ".", tr("Automaton Rules file (*.atr"));
00277         AutomateHandler::getAutomateHandler().
      getAutomate(m_tabs->currentIndex())->saveRules(ruleFileName+".atr");
00278     }
00279     else{
00280         QMessageBox msgBox;
00281         msgBox.critical(0,"Error","Please create or import an Automaton first !");
00282         msgBox.setFixedSize(500,200);
00283     }
00284 }
00285
00290 void MainWindow::openCreationWindow(){
00291     CreationDialog *window = new CreationDialog(this);
00292     connect(window, SIGNAL(settingsFilled(QVector<uint>,
      CellHandler::generationTypes,uint,uint)),
00293             this, SLOT(receiveCellHandler(QVector<uint>,
      CellHandler::generationTypes,uint,uint)));
00294     window->show();
00295 }
00296
00303 void MainWindow::receiveCellHandler(const QVector<unsigned int> dimensions,
00304                                     CellHandler::generationTypes type,
00305                                     unsigned int stateMax, unsigned int density){
00306     AutomateHandler::getAutomateHandler().
      addAutomate(new Automate(dimensions, type, stateMax, density));
00307
00308     if(m_tabs == NULL) createTabs();
00309     QWidget* newTab = createTab();
00310     m_tabs->addTab(newTab, "Automaton "+ QString::number(
      AutomateHandler::getAutomateHandler().getNumberAutomates()));
00311     m_tabs->setCurrentWidget(newTab);
00312     updateBoard(AutomateHandler::getAutomateHandler().
      getNumberAutomates()-1);
00313
00314     RuleEditor* ruleEditor = new RuleEditor(
      AutomateHandler::getAutomateHandler().getAutomate(
      AutomateHandler::getAutomateHandler().getNumberAutomates()-1)->
      getCellHandler().getDimensions().size(), this);
00315     connect(ruleEditor, SIGNAL(fileImported(QString)),this,SLOT(
      addAutomatonRuleFile(QString)));
00316     connect(ruleEditor, SIGNAL(rulesFilled(QList<const Rule*>)), this, SLOT(
      addAutomatonRules(QList<const Rule*>)));
00317     ruleEditor->show();
00318
00319 }
00320
00325 void MainWindow::nextState(unsigned int n){
00326     if(AutomateHandler::getAutomateHandler().getNumberAutomates()== 0){
00327         QMessageBox msgBox;
00328         msgBox.critical(0,"Error","Please create or import an Automaton first !");
00329         msgBox.setFixedSize(500,200);
00330     }
00331     else{
00332
00333         AutomateHandler::getAutomateHandler().
      getAutomate(m_tabs->currentIndex())->run(n);
00334         updateBoard(m_tabs->currentIndex());
00335     }
00336 }
00337
00342 void MainWindow::updateBoard(int index){
00343     if(AutomateHandler::getAutomateHandler().getNumberAutomates()== 0){
00344         QMessageBox msgBox;
00345         msgBox.critical(0,"Error","Please create or import an Automaton first !");
00346         msgBox.setFixedSize(500,200);
00347     }
00348     else{
00349
00350         const CellHandler* cellHandler = &(
      AutomateHandler::getAutomateHandler().
      getAutomate(index)->getCellHandler());
00351         QVector<unsigned int> dimensions = cellHandler->getDimensions();
00352         QTableWidget* board = getBoard(index);
00353         if(dimensions.size() > 1){
00354             int i = 0;
00355             int j = 0;
00356             for (CellHandler::const_iterator it = cellHandler->
      begin(); it != cellHandler->end() && it.changedDimension() < 2; ++it){
00357                     if(it.changedDimension() > 0){
00358                         i = 0;
00359                         j++;
00360                     }
00361                     board->item(i,j)->setBackgroundColor(getColor(it->getState()));
00362                     i++;
```

```
00363             }
00364         }
00365         else{ // dimension = 1
00366             if (board->rowCount() != 1)
00367                 addEmptyRow(index);
00368             int i = board->rowCount() -1;
00369             int j = 0;
00370             for (CellHandler::const_iterator it = cellHandler->
      begin(); it != cellHandler->end() && it.changedDimension() < 1; ++it){
00371                     board->item(i,j)->setBackgroundColor(getColor(it->getState()));
00372                     j++;
00373             }
00374             if (board->rowCount() == 1)
00375                 addEmptyRow(index);
00376
00377             // Go to bottom
00378             QScrollArea *scrool = static_cast<QScrollArea*>(m_tabs->currentWidget()->layout()->itemAt
      (0)->widget());
00379
00380             scrool->verticalScrollBar()->setSliderPosition(scrool->verticalScrollBar()->maximum());
00381
00382         }
00383
00384     }
00385
00386 }
00387
00392 void MainWindow::forward(){
00393     nextState(1);
00394 }
00395
00399 QTableWidget* MainWindow::getBoard(int n){
00400     return m_tabs->widget(n)->findChild<QTableWidget *>();
00401 }
00402
00405 QColor MainWindow::getColor(unsigned int cellState)
00406 {
00407     if (cellState > QColor::colorNames().size() -2)
00408         return Qt::black;
00409     switch (cellState)
00410     {
00411     case 0:
00412         return Qt::white;
00413     case 1:
00414         return Qt::black;
00415     case 2:
00416         return Qt::red;
00417     case 3:
00418         return Qt::green;
00419     case 4:
00420         return Qt::blue;
00421     case 5:
00422         return Qt::yellow;
00423     case 6:
00424         return QColor(170, 110, 40); // brown
00425     case 7:
00426         return QColor(145,30, 180); // purple
00427     case 8:
00428         return QColor(245,130,48); // orange
00429     case 9:
00430         return Qt::cyan;
00431     case 10:
00432         return Qt::magenta;
00433     case 11:
00434         return QColor(210, 245, 60); // Lime
00435     case 12:
00436         return QColor(250, 190, 190); // pink
00437     case 13:
00438         return QColor(0,128,128); // Teal
00439     case 14:
00440         return QColor(230, 190, 255); // Lavender
00441     case 15:
00442         return QColor(255, 250, 200); // beige
00443     case 16:
00444         return QColor(128, 0,0); // Maroon
00445     case 17:
00446         return QColor(170, 255, 195); // Mint
00447     case 18:
00448         return QColor(128, 128, 0); // Olive
00449     case 19:
00450         return QColor(255, 215, 180); // Coral
00451     case 20:
00452         return QColor(0,0,128); // Navy
00453     case 21:
00454         return Qt::gray;
00455
00456
```

```
00457     }
00458
00459     return QColor((Qt::GlobalColor)(cellState +2));
00460 }
00461
00466 void MainWindow::createTabs(){
00467     m_tabs = new QTabWidget(this);
00468     m_tabs->setMovable(true);
00469     m_tabs->setTabsClosable(true);
00470     setCentralWidget(m_tabs);
00471     connect(m_tabs, SIGNAL(tabCloseRequested(int)), this, SLOT(closeTab(int)));
00472     connect(m_tabs, SIGNAL(currentChanged(int)), this, SLOT(
      handleTabChanged()));
00473 }
00474
00481 void MainWindow::addEmptyRow(unsigned int n)
00482 {
00483     QTableWidget *board = getBoard(n);
00484     board->setFixedHeight(board->height() + m_cellSize);
00485     unsigned int row = board->rowCount();
00486     board->insertRow(row);
00487     board->setRowHeight(row, m_cellSize);
00488     for(unsigned int col = 0; col < board->columnCount(); ++col) {
00489         board->setItem(row, col, new QTableWidgetItem(""));
00490         board->item(row, col)->setBackgroundColor("white");
00491         board->item(row, col)->setTextColor("black");
00492     }
00493 }
00494
00499 void MainWindow::closeTab(int n){
00500     m_tabs->setCurrentIndex(n);
00501     saveToFile();
00502     AutomateHandler::getAutomateHandler().
      deleteAutomate(AutomateHandler::getAutomateHandler().
      getAutomate(n));
00503     m_tabs->removeTab(n);
00504 }
00505
00510 void MainWindow::addAutomatonRules(QList<const Rule *> rules){
00511     for(int i =0 ; i < rules.size();i++)
00512     {
00513         AutomateHandler::getAutomateHandler().
      getAutomate(AutomateHandler::getAutomateHandler().
      getNumberAutomates()-1)->addRule(rules.at(i));
00514     }
00515 }
00516
00521 void MainWindow::addAutomatonRuleFile(QString path){
00522     AutomateHandler::getAutomateHandler().
      getAutomate(AutomateHandler::getAutomateHandler().
      getNumberAutomates()-1)->addRuleFile(path);
00523 }
00524
00529 void MainWindow::handlePlayPause(){
00530     if(AutomateHandler::getAutomateHandler().getNumberAutomates()== 0){
00531         QMessageBox msgBox;
00532         msgBox.critical(0,"Error","Please create or import an Automaton first !");
00533         msgBox.setFixedSize(500,200);
00534     }
00535     else{
00536         if(running){
00537             m_playPauseBt->setIcon(m_playIcon);
00538             delete m_timer;
00539         }
00540         else {
00541             m_playPauseBt->setIcon(m_pauseIcon);
00542             m_timer = new QTimer(this);
00543             connect(m_timer, SIGNAL(timeout()), this, SLOT(runAutomaton()));
00544             m_timer->start(m_timeStep->value());
00545         }
00546         running = !running;
00547     }
00548
00549
00550 }
00551
00556 void MainWindow::runAutomaton(){
00557     if(running){
00558         AutomateHandler::getAutomateHandler().
      getAutomate(m_tabs->currentIndex())->run();
00559         QCoreApplication::processEvents();
00560         updateBoard(m_tabs->currentIndex());
00561         QCoreApplication::processEvents();
00562     }
00563 }
00564
00568 void MainWindow::reset(){
```

```
00569     if(AutomateHandler::getAutomateHandler().getNumberAutomates()== 0){
00570         QMessageBox msgBox;
00571         msgBox.critical(0,"Error","Please create or import an Automaton first !");
00572         msgBox.setFixedSize(500,200);
00573     }
00574     else{
00575         //QTableWidget *board = getBoard(m_tabs->currentIndex());
00576         //board->setRowCount(1);
00577         //board->setFixedHeight(m_cellSize);
00578 
00579         AutomateHandler::getAutomateHandler().
      getAutomate(m_tabs->currentIndex())->getCellHandler().reset();
00580         if (AutomateHandler::getAutomateHandler().getAutomate(
      m_tabs->currentIndex())->getCellHandler().getDimensions().size() == 1)
00581         {
00582             QTableWidget *board  = getBoard(m_tabs->currentIndex());
00583             board->setRowCount(0);
00584             board->setFixedHeight(0);
00585         }
00586         updateBoard(m_tabs->currentIndex());
00587     }
00588 }
00589 
00590 
00595 void MainWindow::backward(){
00596     AutomateHandler::getAutomateHandler().
      getAutomate(m_tabs->currentIndex())->getCellHandler().previousStates();
00597     updateBoard(m_tabs->currentIndex());
00598 }
00599 
00604 void MainWindow::cellPressed(int i, int j){
00605     QVector<unsigned int> coord;
00606 
00607     m_currentCellX = i;
00608     m_currentCellY = j;
00609     const CellHandler* cellHandler = &(
      AutomateHandler::getAutomateHandler().
      getAutomate(m_tabs->currentIndex())->getCellHandler());
00610     if(cellHandler->getDimensions().size() > 1){
00611         coord.append(i);
00612         coord.append(j);
00613         m_cellSetter->setValue(cellHandler->getCell(coord)->
      getState());
00614     }
00615     else{
00616         coord.append(j);
00617         m_cellSetter->setValue(cellHandler->getCell(coord)->
      getState());
00618     }
00619 }
00620 
00621 
00626 void MainWindow::changeCellValue(){
00627     if(AutomateHandler::getAutomateHandler().getNumberAutomates()== 0){
00628         QMessageBox msgBox;
00629         msgBox.critical(0,"Error","Please create or import an Automaton first !");
00630         msgBox.setFixedSize(500,200);
00631     }
00632     else{
00633         if(m_currentCellX > -1 && m_currentCellY > -1){
00634             const CellHandler* cellHandler = &(
      AutomateHandler::getAutomateHandler().
      getAutomate(m_tabs->currentIndex())->getCellHandler());
00635             QVector<unsigned int> coord;
00636             if(cellHandler->getDimensions().size() > 1){
00637                 coord.append(m_currentCellX);
00638                 coord.append(m_currentCellY);
00639                 cellHandler->getCell(coord)->forceState(
      m_cellSetter->value());
00640                 updateBoard(m_tabs->currentIndex());
00641             }
00642             else{
00643                 coord.append(m_currentCellY);
00644                 cellHandler->getCell(coord)->forceState(
      m_cellSetter->value());
00645                 QTableWidget *board = getBoard(m_tabs->currentIndex());
00646                 int i = 0;
00647                 int j = 0;
00648                 for (CellHandler::const_iterator it = cellHandler->
      begin(); it != cellHandler->end() && it.changedDimension() < 1; ++it){
00649                         board->item(i,j)->setBackgroundColor(getColor(it->getState()));
00650                         j++;
00651                 }
00652             }
00653 
00654         }
00655     }
```

```
00656 }
00657
00662 void MainWindow::handleTabChanged(){
00663     if(m_tabs->currentIndex() >= 0){
00664         m_cellSetter->setMaximum(CellHandler::getMaxState());
00665         m_currentCellX = -1;
00666         m_currentCellY = -1;
00667     }
00668
00669 }
00670
00671
00672 void MainWindow::setSize(int newCellSize)
00673 {
00674     m_cellSize = newCellSize;
00675     if(AutomateHandler::getAutomateHandler().getNumberAutomates()!= 0)
00676     {
00677         for (unsigned int i = 0; i < m_tabs->count(); i++)
00678         {
00679             QTableWidget* board = getBoard(i);
00680             if (m_cellSize < 10)
00681                 board->setShowGrid(false);
00682             else
00683                 board->setShowGrid(true);
00684             for (unsigned int row = 0; row < board->rowCount(); row++)
00685                 board->setRowHeight(row, m_cellSize);
00686             for (unsigned int col = 0; col < board->columnCount(); col++)
00687                 board->setColumnWidth(col, m_cellSize);
00688             board->setFixedSize(board->columnCount()*m_cellSize, board->rowCount()*
    m_cellSize);
00689         }
00690     }
00691 }
```

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

```
00001 #ifndef MAINWINDOW_H
00002 #define MAINWINDOW_H
00003
00004 #include <QMainWindow>
00005 #include <QtWidgets>
00006 #include "cellhandler.h"
00007 #include "automate.h"
00008 #include "creationdialog.h"
00009 #include "automatehandler.h"
00010 #include "ruleeditor.h"
00011
00018 class MainWindow : public QMainWindow
00019 {
00020     Q_OBJECT
00021
```

```
00022     QTabWidget *m_tabs; //Tabs for the main window
00023     //QVector<Automate *> m_automatons; //QVector containing a pointer to each tab's Automaton
00024
00026     QIcon m_fastBackwardIcon;
00027     QIcon m_fastForwardIcon;
00028     QIcon m_playIcon;
00029     QIcon m_pauseIcon;
00030     QIcon m_newIcon;
00031     QIcon m_saveIcon;
00032     QIcon m_openIcon;
00033     QIcon m_resetIcon;
00034
00036     QAction *m_playPause;
00037     QAction *m_nextState;
00038     QAction *m_previousState;
00039     QAction *m_fastForward;
00040     QAction *m_fastBackward;
00041     QAction *m_openAutomaton;
00042     QAction *m_saveAutomaton;
00043     QAction *m_newAutomaton;
00044     QAction *m_resetAutomaton;
00045
00047     QToolButton *m_playPauseBt;
00048     QToolButton *m_nextStateBt;
00049     QToolButton *m_previousStateBt;
00050     QToolButton *m_fastForwardBt;
00051     QToolButton *m_fastBackwardBt;
00052     QToolButton *m_openAutomatonBt;
00053     QToolButton *m_saveAutomatonBt;
00054     QToolButton *m_newAutomatonBt;
00055     QToolButton *m_resetBt;
00056
00057
00058     QSpinBox *m_timeStep;
00059     QSpinBox *m_cellSetter;
00060     QTimer* m_timer;
00061
00062     QSlider *m_zoom;
00063
00064     Automate* m_newAutomate;
00065     bool running;
00066     QToolBar *m_toolBar;
00067
00068     int m_currentCellX;
00069     int m_currentCellY;
00070
00072     unsigned int m_boardHSize = 25;
00073     unsigned int m_boardVSize = 25;
00074     unsigned int m_cellSize = 30;
00075
00076     void createIcons();
00077     void createActions();
00078     void createToolBar();
00079     void createBoard();
00080     QWidget* createTab();
00081     void createTabs();
00082
00083     void addEmptyRow(unsigned int n);
00084     void updateBoard(int index);
00085     void nextState(unsigned int n);
00086     QTableWidget* getBoard(int n);
00087
00088     static QColor getColor(unsigned int cellState);
00089
00090
00091 public:
00092     explicit MainWindow(QWidget *parent = nullptr);
00093     virtual ~MainWindow();
00094
00095 signals:
00096
00097 public slots:
00098     void openFile();
00099     void saveToFile();
00100     void openCreationWindow();
00101     void receiveCellHandler(const QVector<unsigned int> dimensions,
00102                             CellHandler::generationTypes type =
      CellHandler::generationTypes::empty,
00103                             unsigned int stateMax = 1, unsigned int density = 20);
00104     void addAutomatonRules(QList<const Rule *> rules);
00105     void addAutomatonRuleFile(QString path);
00106     void forward();
00107     void backward();
00108     void closeTab(int n);
00109     void runAutomaton();
00110     void handlePlayPause();
00111     void reset();
```

```
00112     void cellPressed(int i, int j);
00113     void changeCellValue();
00114     void handleTabChanged();
00115     void setSize(int newCellSize);
00116
00117 };
00118
00119 #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
00030 bool MatrixRule::matchCell(const Cell *cell) const
00031 {
00032     // Check cell state
00033     if (!m_currentCellPossibleValues.contains(cell->
      getState()))
00034     {
00035         return false;
00036     }
00037
00038     // Check neighbours
00039     bool matched = true;
00040     // Rappel : QMap<relativePosition, possibleStates>
00041     for (QMap<QVector<short>,  QVector<unsigned int> >::const_iterator it =
      m_matrix.begin() ; it != m_matrix.end(); ++it)
00042     {
00043         if (cell->getNeighbour(it.key()) != nullptr) // Border management
00044         {
00045             if (! it.value().contains(cell->getNeighbour(it.key())->getState()))
00046             {
00047                 matched = false;
00048                 break;
00049             }
00050         }
00051         else
00052         {
00053             if (!it.value().contains(0))
00054             {
00055                 matched = false;
00056                 break;
00057             }
00058         }
00059
00060     }
00061
00062     return matched;
00063 }
00064
00067 void MatrixRule::addNeighbourState(QVector<short> relativePosition, unsigned
      int matchState)
00068 {
00069     m_matrix[relativePosition].push_back(matchState);
00070 }
00071
00074 void MatrixRule::addNeighbourState(QVector<short> relativePosition,
      QVector<unsigned int> matchStates)
00075 {
00076     for (QVector<unsigned int>::const_iterator it = matchStates.begin(); it != matchStates.end(); ++it)
00077         m_matrix[relativePosition].push_back(*it);
00078 }
00079
00082 QJsonObject MatrixRule::toJson() const
00083 {
00084     QJsonObject object(Rule::toJson());
00085
00086     object.insert("type", QJsonValue("matrix"));
00087
00088     QJsonArray neighbours;
00089     for (QMap<QVector<short>,  QVector<unsigned int> >::const_iterator it =
      m_matrix.begin(); it != m_matrix.end(); ++it)
00090     {
00091         QJsonObject aNeighbour;
00092         QJsonArray relativePosition;
00093         for (unsigned int i = 0; i < it.key().size(); i++)
00094         {
00095             relativePosition.append(QJsonValue((int)it.key().at(i)));
00096         }
00097         aNeighbour.insert("relativePosition", relativePosition);
00098
00099         QJsonArray neighbourStates;
00100         for (unsigned int i = 0; i < it.value().size(); i++)
00101         {
00102             neighbourStates.append(QJsonValue((int)it.value().at(i)));
00103         }
00104         aNeighbour.insert("neighbourStates", neighbourStates);
```

```
00105
00106         neighbours.append(aNeighbour);
00107     }
00108     object.insert("neighbours", neighbours);
00109
00110     return object;
00111 }
```

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

```
00001 #ifndef MATRIXRULE_H
00002 #define MATRIXRULE_H
00003
00004 #include <QVector>
00005 #include <QMap>
00006 #include "cell.h"
00007 #include "rule.h"
00008
00009 QVector<unsigned int> fillInterval(unsigned int min, unsigned int max);
00010
00013 class MatrixRule : public Rule
00014 {
00015     public:
00016         MatrixRule(unsigned int finalState, QVector<unsigned int> currentStates =
      QVector<unsigned int>());
00017
00018
00019         virtual bool matchCell(const Cell* cell) const;
00020         virtual void addNeighbourState(QVector<short> relativePosition, unsigned int
      matchState);
00021         virtual void addNeighbourState(QVector<short> relativePosition, QVector<unsigned
       int> matchStates);
00022
00023         QJsonObject toJson() const;
00024
00025
00026 protected:
00027
00028         QMap<QVector<short>,  QVector<unsigned int> > m_matrix;
00029 };
00030
00031
00032
00033 #endif // MATRIXRULE_H
```

7.31 neighbourrule.cpp File Reference

```
#include "neighbourrule.h"
```

7.32 neighbourrule.cpp

```
00001 #include "neighbourrule.h"
00002
00084 bool NeighbourRule::inInterval(unsigned int matchingNeighbours)const
00085 {
00086     if(matchingNeighbours >= m_neighbourInterval.first && matchingNeighbours<=
      m_neighbourInterval.second)
00087         return true;
00088     else
00089         return false;
00090 }
00091
00095 NeighbourRule::NeighbourRule(unsigned int outputState, QVector<unsigned int>
      currentCellValues, QPair<unsigned int, unsigned int> intervalNbrNeighbour, QSet<unsigned int> neighbourValues)
       :
00096         Rule(currentCellValues, outputState), m_neighbourInterval(intervalNbrNeighbour),
      m_neighbourPossibleValues(neighbourValues)
00097 {
00098     if (m_neighbourInterval.second == 0)
00099         throw QString(QObject::tr("Low value of the number of neighbour interval can't be 0"));
00100     if (m_neighbourInterval.first > m_neighbourInterval.second)
00101         throw QString(QObject::tr("The interval must be (x,y) with x <= y"));
00102 }
00103
00104 NeighbourRule::~NeighbourRule()
00105 {
00106
00107 }
00108
00115 bool NeighbourRule::matchCell(const Cell *c)const
```

```
00116 {
00117     unsigned int matchingNeighbours = 0;
00118     if (!m_currentCellPossibleValues.contains(c->
      getState()))
00119         return false;
00120 
00121    // QSet<unsigned int> set;
00122     //QSet<unsigned int> m_neighbourPossibleValues;
00123     //set<<3<<2<<5<<9;
00124     QSet<unsigned int>::const_iterator i = m_neighbourPossibleValues.constBegin();
00125     if (i == m_neighbourPossibleValues.constEnd()) // All possibles values (except
       0)
00126     {
00127         matchingNeighbours = c->countNeighbours();
00128     }
00129     else
00130     {
00131         while (i != m_neighbourPossibleValues.constEnd()) {
00132             //std::cout<<*i;
00133             matchingNeighbours += c->countNeighbours(*i);
00134             ++i;
00135         }
00136     }
00137     if(!inInterval(matchingNeighbours))
00138         return false; //the rule cannot be applied to the cell
00139 
00140     return true; //the rule can be applied to the cell
00141 
00142 }
00143 
00146 QJsonObject NeighbourRule::toJson() const
00147 {
00148     QJsonObject object(Rule::toJson());
00149 
00150     object.insert("type", QJsonValue("neighbour"));
00151     object.insert("neighbourNumberMin", QJsonValue((int)m_neighbourInterval.first));
00152     object.insert("neighbourNumberMax", QJsonValue((int)m_neighbourInterval.second));
00153 
00154     QJsonArray neighbourState;
00155     for (QSet<unsigned int>::const_iterator it = m_neighbourPossibleValues.begin()
      ; it != m_neighbourPossibleValues.end(); ++it)
00156     {
00157         neighbourState.append(QJsonValue((int)*it));
00158     }
00159     object.insert("neighbourStates", neighbourState);
00160 
00161     return object;
00162 }
```

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

```
00001 #ifndef NEIGHBOURRULE_H
00002 #define NEIGHBOURRULE_H
00003 
```

```
00004 #include <QPair>
00005 #include <QSet>
00006 #include "cell.h"
00007 #include "rule.h"
00008
00013 class NeighbourRule : public Rule
00014 {
00015 protected:
00016     QPair<unsigned int , unsigned int> m_neighbourInterval;
00017     //ATTENTION check that first is lower than second
00018     QSet<unsigned int> m_neighbourPossibleValues;
00019     bool inInterval(unsigned int matchingNeighbours)const;
00020     //bool load(const QJsonObject &json);
00021 public:
00022     NeighbourRule(unsigned int outputState, QVector<unsigned int> currentCellValues,
      QPair<unsigned int, unsigned int> intervalNbrNeighbour, QSet<unsigned int> neighbourValues = QSet<unsigned int>());
00023     ~NeighbourRule();
00024     bool matchCell(const Cell * c)const;
00025
00026     QJsonObject toJson() const;
00027 };
00028
00029 #endif // NEIGHBOURRULE_H
```

7.35 presentation.md File Reference

7.36 presentation.md

```
00001 \page Presentation
00002 # What is AutoCell
00003 The purpose of this project is to create a Cellular Automate Simulator.
00004
00005 \includedoc CellHandler
```

7.37 README.md File Reference

7.38 README.md

```
00001 \mainpage
00002
00003 To generate the Documentation, go in Documentation directory and run `make`.
00004
00005 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

```
00001 #include "rule.h"
00002
00003 Rule::Rule(QVector<unsigned int> currentCellValues, unsigned int outputState):
00004         m_currentCellPossibleValues(currentCellValues), m_cellOutputState(outputState)
00005 {
00006
00007 }
00008
00009 QJsonObject Rule::toJson() const
00010 {
00011     QJsonObject object;
00012     object.insert("finalState", QJsonValue((int)m_cellOutputState));
00013
00014     QJsonArray currentStates;
00015     for (unsigned int i = 0; i < m_currentCellPossibleValues.size(); i++)
00016     {
00017         currentStates.append(QJsonValue((int)m_currentCellPossibleValues.at(i)))
    ;
00018     }
00019     object.insert("currentStates", currentStates);
00020
00021     return object;
00022 }
00023
00026 unsigned int Rule::getCellOutputState()const
00027 {
00028     return m_cellOutputState;
00029 }
00030
```

7.41 rule.h File Reference

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

Classes

- class Rule

7.42 rule.h

```
00001 #ifndef RULE_H
00002 #define RULE_H
00003
00004 #include <QVector>
00005 #include <QJsonObject>
00006 #include <QJsonArray>
00007 #include "cell.h"
00008
00009
00013 class Rule
00014 {
00015 protected:
00016     QVector<unsigned int> m_currentCellPossibleValues;
00017     unsigned int m_cellOutputState;
00018 public:
00019     Rule(QVector<unsigned int> currentCellValues, unsigned int outputState);
00020
00021     virtual QJsonObject toJson() const = 0;
00022     virtual ~Rule(){}
00032     virtual bool matchCell(const Cell * c)const = 0;
00033     unsigned int getCellOutputState() const;
00034
00035 };
00036
00037 #endif // RULE_H
```

7.43 ruleeditor.cpp File Reference

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

7.44 ruleeditor.cpp

```
00001 #include "ruleeditor.h"
00002
00003 RuleEditor::RuleEditor(unsigned int dimensions, QWidget *parent) : QDialog(parent),
      m_dimensions(dimensions)
00004 {
00005     QGridLayout *rulesInputLayout = new QGridLayout();
00006     QHBoxLayout *hlayout = new QHBoxLayout();
00007     if (m_dimensions > 1)
00008     {
00009         m_selectedRule = -1;
00010
00011         m_rulesListWidget = new QListWidget(this);
00012         QLabel *rulesLabel = new QLabel("Rules ",this);
00013         QVBoxLayout *rulesListLayout = new QVBoxLayout();
00014         rulesListLayout->addWidget(rulesLabel);
00015         rulesListLayout->addWidget(m_rulesListWidget);
00016         hlayout->addLayout(rulesListLayout);
00017
00018         rulesInputLayout->addWidget(new QLabel("Current cell values :",this),0,0);
00019         m_currentStatesEdit = new QLineEdit(this);
00020         QRegExp rgx("([0-9]+,)*");
00021         QRegExpValidator *v = new QRegExpValidator(rgx, this);
00022         m_currentStatesEdit->setValidator(v);
00023         rulesInputLayout->addWidget(m_currentStatesEdit,0,1);
00024
00025         rulesInputLayout->addWidget(new QLabel("Neighbour number lower bound :",this),1,0);
00026         m_lowerNeighbourBox = new QSpinBox(this);
00027         rulesInputLayout->addWidget(m_lowerNeighbourBox,1,1);
00028
00029         rulesInputLayout->addWidget(new QLabel("Neighbour number upper bound :",this),2,0);
00030         m_upperNeighbourBox = new QSpinBox(this);
00031         rulesInputLayout->addWidget(m_upperNeighbourBox,2,1);
00032
00033         rulesInputLayout->addWidget(new QLabel("Neighbour values :",this),3,0);
00034         m_neighbourStatesEdit = new QLineEdit(this);
00035         m_neighbourStatesEdit->setValidator(v);
00036         rulesInputLayout->addWidget(m_neighbourStatesEdit,3,1);
00037
00038         rulesInputLayout->addWidget(new QLabel("Output state :",this),4,0);
00039         m_outputStateBox = new QSpinBox(this);
00040         rulesInputLayout->addWidget(m_outputStateBox,4,1);
00041     }
00042     else
00043     {
00044         rulesInputLayout->addWidget(new QLabel("Automaton number :",this),0,0);
00045         m_automatonNumber = new QSpinBox(this);
00046         m_automatonNumber->setMaximum(255);
00047         m_automatonNumber->setMinimum(0);
00048         rulesInputLayout->addWidget(m_automatonNumber,0,1);
00049     }
00050
00051     hlayout->addLayout(rulesInputLayout);
00052     QVBoxLayout* mainLayout = new QVBoxLayout();
00053     QHBoxLayout* buttonLayout = new QHBoxLayout();
00054
00055     if (dimensions > 1)
00056     {
00057         m_addBt = new QPushButton("Add Rule",this);
00058         m_importBt = new QPushButton("Import Rule file",this);
00059         m_removeBt = new QPushButton("Remove Rule",this);
00060         buttonLayout->addWidget(m_importBt);
00061         buttonLayout->addWidget(m_addBt);
00062         buttonLayout->addWidget(m_removeBt);
00063     }
00064     m_doneBt = new QPushButton("Done !",this);
00065
00066
00067     buttonLayout->addWidget(m_doneBt);
00068
00069     mainLayout->addLayout(hlayout);
00070     mainLayout->addLayout(buttonLayout);
00071     setLayout(mainLayout);
```

```
00072
00073     if (dimensions > 1)
00074     {
00075         connect(m_addBt, SIGNAL(clicked(bool)), this, SLOT(addRule()));
00076         connect(m_importBt, SIGNAL(clicked(bool)), this, SLOT(
      importFile()));
00077         connect(m_removeBt, SIGNAL(clicked(bool)), this, SLOT(
      removeRule()));
00078     }
00079     connect(m_doneBt, SIGNAL(clicked(bool)), this, SLOT(sendRules()));
00080
00081 }
00082
00083
00084
00085 void RuleEditor::addRule(){
00086     unsigned int outputState = m_outputStateBox->value();
00087     QVector<unsigned int> currentCellValues;
00088     QStringList valList = m_currentStatesEdit->text().split(",");
00089     for(int i = 0; i < valList.size(); i++) currentCellValues.append(valList.at(i).toInt());
00090
00091     QPair<unsigned int, unsigned int> neighbourInterval;
00092     neighbourInterval.first = m_lowerNeighbourBox->value();
00093     neighbourInterval.second = m_upperNeighbourBox->value();
00094
00095     QSet<unsigned int> neighbourValues;
00096     valList = m_neighbourStatesEdit->text().split(",");
00097     for(int i = 0; i < valList.size(); i++) neighbourValues << valList.at(i).toInt();
00098
00099     m_rules.append(new NeighbourRule(outputState,currentCellValues,neighbourInterval,
      neighbourValues));
00100
00101     QString listLabel = m_currentStatesEdit->text()+" -> "+QString::number(
      m_outputStateBox->value())
00102                         +" if "+QString::number(m_lowerNeighbourBox->value())+" to "+
00103                         QString::number(m_upperNeighbourBox->value())+" neighbours are
       in states "+
00104                         m_neighbourStatesEdit->text();
00105     m_rulesListWidget->addItem(listLabel);
00106 }
00107
00108 void RuleEditor::removeRule(){
00109     m_rules.removeAt(m_rulesListWidget->currentRow());
00110     delete m_rulesListWidget->takeItem(m_rulesListWidget->currentRow());
00111 }
00112
00113 void RuleEditor::sendRules(){
00114     if (m_dimensions == 1)
00115     {
00116         QList<const Rule*> ruleList = generate1DRules(
      m_automatonNumber->value());
00117         for (const Rule* rule : ruleList) // C++11
00118         {
00119             m_rules.append(rule);
00120         }
00121
00122     }
00123     emit rulesFilled(m_rules);
00124     this->close();
00125 }
00126
00127 void RuleEditor::importFile(){
00128     QString fileName = QFileDialog::getOpenFileName(this, tr("Open Rule file"), ".",
00129                                                     tr("Automaton rule files (*.atr)"));
00130     if(!fileName.isEmpty()){
00131         emit fileImported(fileName);
00132         this->close();
00133     }
00134 }
```

7.45 ruleeditor.h File Reference

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

Classes

- class RuleEditor

7.46 ruleeditor.h

```
00001 #ifndef RULEEDITOR_H
00002 #define RULEEDITOR_H
00003 #include <QtWidgets>
00004 #include "neighbourrule.h"
00005 #include "automate.h"
00006
00007 class RuleEditor : public QDialog
00008 {
00009     Q_OBJECT
00010     QList<const Rule*> m_rules;
00011     QListWidget* m_rulesListWidget;
00012     QTableWidget* m_rulesTable;
00013
00014     QSpinBox* m_outputStateBox;
00015     QLineEdit* m_currentStatesEdit;
00016     QLineEdit* m_neighbourStatesEdit;
00017     QSpinBox* m_upperNeighbourBox;
00018     QSpinBox* m_lowerNeighbourBox;
00019     QSpinBox* m_automatonNumber;
00020
00021     QPushButton* m_addBt;
00022     QPushButton* m_doneBt;
00023     QPushButton* m_removeBt;
00024     QPushButton* m_importBt;
00025
00026     unsigned int m_selectedRule;
00027     unsigned int m_dimensions;
00028
00029
00030 public:
00031     explicit RuleEditor(unsigned int dimensions, QWidget *parent = nullptr);
00032
00033 signals:
00034     void rulesFilled(QList<const Rule*> rules);
00035     void fileImported(QString path);
00036
00037 public slots:
00038     void removeRule();
00039     void addRule();
00040     void importFile();
00041     void sendRules();
00042
00043
00044 };
00045
00046 #endif // RULEEDITOR_H
```

Index