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 229 of file automate.cpp.

References m_rules.

Referenced by MainWindow::addAutomatonRules().

6.1.3.2 addRuleFile()

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

Definition at line 286 of file automate.cpp.

References loadRules().

Referenced by MainWindow::addAutomatonRuleFile().

6.1.3.3 getCellHandler()

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

Accessor of m_cellHandler.

Definition at line 281 of file automate.cpp.

References m_cellHandler.

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

6.1.3.4 getRules()

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

Return all the rules.

Definition at line 247 of file automate.cpp.

References m_rules.

6.1.3.5 loadRules()

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

Load the rules of the json given.

Returns

Return false if something went wrong

Parameters

json	JsonObject wich contains the rules

Definition at line 7 of file automate.cpp.

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

Referenced by addRuleFile(), and Automate().

6.1.3.6 run()

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

Apply the rule on the cells grid nbSteps times.

Parameters

nbSteps	number of iterations of the automate on the cell grid

Definition at line 256 of file automate.cpp.

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

6.1.3.7 saveAll()

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

Save both rules and cellHandler in the differents files.

Definition at line 222 of file automate.cpp.

References saveCells(), and saveRules().

6.1.3.8 saveCells()

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

Save cellHandler.

Definition at line 213 of file automate.cpp.

References m_cellHandler, and CellHandler::save().

Referenced by saveAll().

6.1.3.9 saveRules()

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

Save automate's rules in the file.

Returns

False if something went wrong

Definition at line 191 of file automate.cpp.

References m_rules.

Referenced by saveAll().

6.1.3.10 setRulePriority()

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

Modify the place of the rule in the priority list.

2 rules can't have the same priority rank

Parameters

rule	Rule to move
newPlace	New place of the rule

Definition at line 240 of file automate.cpp.

References m_rules.

6.1.4 Friends And Related Function Documentation

6.1.4.1 AutomateHandler

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

Definition at line 20 of file automate.h.

6.1.5 Member Data Documentation

6.1.5.1 m_cellHandler

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

CellHandler to go through.

Definition at line 18 of file automate.h.

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

6.1.5.2 m_rules

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

Rules to use on the cells.

Definition at line 19 of file automate.h.

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

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

- automate.h
- automate.cpp

6.2 AutomateHandler Class Reference

Implementation of singleton design pattern.

```
#include <automatehandler.h>
```

Public Member Functions

- Automate * getAutomate (unsigned int indexAutomate)
  Get an automate from the list according to its index.
- unsigned int getNumberAutomates () const
  Get the number of automates contained in the automate list.
- void addAutomate (Automate *automate)
  Add an automate in the automate list.
- void deleteAutomate (Automate *automate)
  Delete an automate from the automate list.

Static Public Member Functions

- static AutomateHandler & getAutomateHandler ()

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

  Delete the unique automate handler if it exists.

Private Member Functions

- AutomateHandler ()

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

  Delete all the automates contained in the automate handler.

Private Attributes

- QList< Automate * > m_ActiveAutomates

  list of existing automates

Static Private Attributes

- static AutomateHandler * m_activeAutomateHandler = nullptr

  active automate handler if existing, nullptr else

6.2.1 Detailed Description

Implementation of singleton design pattern.

Definition at line 10 of file automatehandler.h.

6.2.2 Constructor & Destructor Documentation

6.2.2.1 AutomateHandler() [1/2]

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

Construct an automate handler.

Definition at line 10 of file automatehandler.cpp.

Referenced by getAutomateHandler().

6.2.2.2 AutomateHandler() [2/2]

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

6.2.2.3 ~AutomateHandler()

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

Delete all the automates contained in the automate handler.

Definition at line 18 of file automatehandler.cpp.

References m_ActiveAutomates.

6.2.3 Member Function Documentation

6.2.3.1 addAutomate()

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

Add an automate in the automate list.

Parameters

automate	to be added to the automate list

Definition at line 78 of file automatehandler.cpp.

References m_ActiveAutomates.

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

6.2.3.2 deleteAutomate()

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

Delete an automate from the automate list.

Parameters

automate	automate to delete

Definition at line 89 of file automatehandler.cpp.

References m_ActiveAutomates.

Referenced by MainWindow::closeTab().

6.2.3.3 deleteAutomateHandler()

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

Delete the unique automate handler if it exists.

Definition at line 39 of file automatehandler.cpp.

References m_activeAutomateHandler.

6.2.3.4 getAutomate()

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

Get an automate from the list according to its index.

Parameters

indexAutomate	Index of a specific automate in the automate list

Returns

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

Definition at line 55 of file automatehandler.cpp.

References m_ActiveAutomates.

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

6.2.3.5 getAutomateHandler()

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

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

Returns

the unique running automate handler instance

Definition at line 29 of file automatehandler.cpp.

References AutomateHandler(), and m_activeAutomateHandler.

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

6.2.3.6 getNumberAutomates()

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

Get the number of automates contained in the automate list.

Returns

number of automates in the automate list

Definition at line 67 of file automatehandler.cpp.

References m_ActiveAutomates.

6.2.3.7 operator=()

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

6.2.4 Member Data Documentation

6.2.4.1 m_activeAutomateHandler

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

active automate handler if existing, nullptr else

Initialization of the static value.

Definition at line 14 of file automatehandler.h.

Referenced by deleteAutomateHandler(), and getAutomateHandler().

6.2.4.2 m_ActiveAutomates

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

list of existing automates

Definition at line 13 of file automatehandler.h.

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

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

- automatehandler.h
- automatehandler.cpp

6.3 Cell Class Reference

Contains the state, the next state and the neighbours.

`#include <cell.h>`

Public Member Functions

- Cell (unsigned int state=0)
  Constructs a cell with the state given. State 0 is dead state.
- void setState (unsigned int state)
  Set temporary state.
- void validState ()
  Validate temporary state.
- void forceState (unsigned int state)
  Force the state change.
- unsigned int getState () const
  Access current cell state.
- bool back ()
  Set the previous state.
- void reset ()
  Reset the cell to the 1st state.
- bool addNeighbour (const Cell *neighbour, const QVector< short > relativePosition)
  Add a new neighbour to the Cell.
- QMap< QVector< short >, const Cell * > getNeighbours () const
  Access neighbours list.
- const Cell * getNeighbour (QVector< short > relativePosition) const
  Get the neighbour asked. If not existent, return nullptr.
- unsigned int countNeighbours (unsigned int filterState) const
  Return the number of neighbour which have the given state.
- unsigned int countNeighbours () const
  Return the number of neighbour which are not dead (=0)

Static Public Member Functions

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

  Get the relative position, as neighbourPosition minus cellPosition.

Private Attributes

- QStack< unsigned int > m_states

  Current state.
- unsigned int m_nextState

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

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

6.3.1 Detailed Description

Contains the state, the next state and the neighbours.

Definition at line 11 of file cell.h.

6.3.2 Constructor & Destructor Documentation

6.3.2.1 Cell()

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

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

Parameters

state	Cell state, dead state by default

Definition at line 7 of file cell.cpp.

References m_states.

6.3.3 Member Function Documentation

6.3.3.1 addNeighbour()

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

Add a new neighbour to the Cell.

Parameters

relativePosition	Relative position of the new neighbour
neighbour	New neighbour

Returns

False if the neighbour already exists

Definition at line 83 of file cell.cpp.

References m_neighbours.

6.3.3.2 back()

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

Set the previous state.

Returns

Return false if we are already at the first state

Definition at line 58 of file cell.cpp.

References m_nextState, and m_states.

6.3.3.3 countNeighbours() [1/2]

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

Return the number of neighbour which have the given state.

Definition at line 110 of file cell.cpp.

References m_neighbours.

Referenced by NeighbourRule::matchCell().

6.3.3.4 countNeighbours() [2/2]

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

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

Definition at line 123 of file cell.cpp.

References m_neighbours.

6.3.3.5 forceState()

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

Force the state change.

Is equivalent to setState followed by validState

Parameters

state	New state

Definition at line 41 of file cell.cpp.

References m_nextState, and m_states.

6.3.3.6 getNeighbour()

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

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

Definition at line 103 of file cell.cpp.

References m_neighbours.

Referenced by MatrixRule::matchCell().

6.3.3.7 getNeighbours()

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

Access neighbours list.

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

Definition at line 96 of file cell.cpp.

References m_neighbours.

6.3.3.8 getRelativePosition()

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

Get the relative position, as neighbourPosition minus cellPosition.

Exceptions

QString	Different size of position vectors

Parameters

cellPosition	Cell Position
neighbourPosition	Neighbour absolute position

Definition at line 140 of file cell.cpp.

Referenced by CellHandler::foundNeighbours().

6.3.3.9 getState()

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

Access current cell state.

Definition at line 49 of file cell.cpp.

References m_states.

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

6.3.3.10 reset()

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

Reset the cell to the 1st state.

Definition at line 69 of file cell.cpp.

References m_nextState, and m_states.

6.3.3.11 setState()

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

Set temporary state.

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

Parameters

state	New state

Definition at line 20 of file cell.cpp.

References m_nextState.

6.3.3.12 validState()

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

Validate temporary state.

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

Definition at line 30 of file cell.cpp.

References m_nextState, and m_states.

6.3.4 Member Data Documentation

6.3.4.1 m_neighbours

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

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

Definition at line 37 of file cell.h.

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

6.3.4.2 m_nextState

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

Temporary state, before validation.

Definition at line 35 of file cell.h.

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

6.3.4.3 m_states

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

Current state.

Definition at line 34 of file cell.h.

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

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

- cell.h
- cell.cpp

6.4 CellHandler Class Reference

Cell container and cell generator.

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

Classes

- class iteratorT

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

Public Types

- enum generationTypes { empty, random, symetric }

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

Public Member Functions

- CellHandler (const QString filename)

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

  Construct all the cells from the json object given.
- CellHandler (const QVector< unsigned int > dimensions, generationTypes type=empty, unsigned int 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.
- unsigned int getMaxState () const

  Return the max state of the CellHandler.

- QVector< unsigned int > getDimensions () const

  Accessor of m_dimensions.
- void nextStates () const

  Valid the state of all cells.
- bool previousStates () const

  Get all the cells to their previous states.
- void reset () const

  Reset all the cells to the 1st state.
- bool save (QString filename) const

  Save the CellHandler current configuration in the file given.
- void generate (generationTypes type, unsigned int stateMax=1, unsigned short density=50)

  Replace Cell values by random values (symetric or not)
- void print (std::ostream &stream) const

  Print in the given stream the CellHandler.
- const_iterator begin () const

  Give the iterator which corresponds to the current CellHandler.
- iterator begin ()

  Give the iterator which corresponds to the current CellHandler.
- bool end () const

  End condition of the iterator.

Private Member Functions

- bool load (const QJsonObject &json)

  Load the config file in the CellHandler.
- void foundNeighbours ()

  Set the neighbours of each cells.
- void positionIncrement (QVector< unsigned int > &pos, unsigned int value=1) const

  Increment the QVector given by the value choosen.
- QVector< QVector< unsigned int > > * getListNeighboursPositionsRecursive (const QVector< unsigned int > position, unsigned int dimension, QVector< unsigned int > lastAdd) const

  Recursive function which browse the position possibilities tree.
- QVector< QVector< unsigned int > > & getListNeighboursPositions (const QVector< unsigned int > position) const

  Prepare the call of the recursive version of itself.

Private Attributes

- QVector< unsigned int > m_dimensions

  Vector of x dimensions.
- QMap< QVector< unsigned int >, Cell *> m_cells

  Map of cells, with a x dimensions vector as key.
- unsigned int m_maxState

6.4.1 Detailed Description

Cell container and cell generator.

Generate cells from a json file.

Definition at line 20 of file cellhandler.h.

6.4.2 Member Typedef Documentation

6.4.2.1 const_iterator

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

Definition at line 94 of file cellhandler.h.

6.4.2.2 iterator

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

Definition at line 95 of file cellhandler.h.

6.4.3 Member Enumeration Documentation

6.4.3.1 generationTypes

```
enum CellHandler::generationTypes
```

Type of random generation.

Enumerator

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

Definition at line 99 of file cellhandler.h.

6.4.4 Constructor & Destructor Documentation

6.4.4.1 CellHandler() [1/3]

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

Construct all the cells from the json file given.

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

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

Parameters

filename	Json file which contains the description of all the cells

Exceptions

QString	Unreadable file
QString	Empty file
QString	Not valid file

Definition at line 25 of file cellhandler.cpp.

References foundNeighbours(), and load().

6.4.4.2 CellHandler() [2/3]

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

Construct all the cells from the json object given.

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

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

Parameters

json	Json object which contains the description of all the cells

Exceptions

QString	Not valid file

Definition at line 76 of file cellhandler.cpp.

References foundNeighbours(), and load().

6.4.4.3 CellHandler() [3/3]

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

Construct a CellHandler of the given dimension.

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

Parameters

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

Definition at line 98 of file cellhandler.cpp.

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

6.4.4.4 ~CellHandler()

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

Destroys all cells in the CellHandler.

Definition at line 131 of file cellhandler.cpp.

References m_cells.

6.4.5 Member Function Documentation

6.4.5.1 begin() [1/2]

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

Give the iterator which corresponds to the current CellHandler.

Definition at line 327 of file cellhandler.cpp.

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

6.4.5.2 begin() [2/2]

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

Give the iterator which corresponds to the current CellHandler.

Definition at line 320 of file cellhandler.cpp.

6.4.5.3 end()

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

End condition of the iterator.

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

Definition at line 336 of file cellhandler.cpp.

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

6.4.5.4 foundNeighbours()

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

Set the neighbours of each cells.

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

Definition at line 434 of file cellhandler.cpp.

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

Referenced by CellHandler().

6.4.5.5 generate()

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

Replace Cell values by random values (symetric or not)

Parameters

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

Definition at line 241 of file cellhandler.cpp.

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

Referenced by CellHandler().

6.4.5.6 getCell()

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

Access the cell to the given position.

Definition at line 141 of file cellhandler.cpp.

References m_cells.

6.4.5.7 getDimensions()

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

Accessor of m_dimensions.

Definition at line 155 of file cellhandler.cpp.

References m_dimensions.

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

6.4.5.8 getListNeighboursPositions()

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

Prepare the call of the recursive version of itself.

Parameters

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

Returns

List of positions

Definition at line 493 of file cellhandler.cpp.

References getListNeighboursPositionsRecursive().

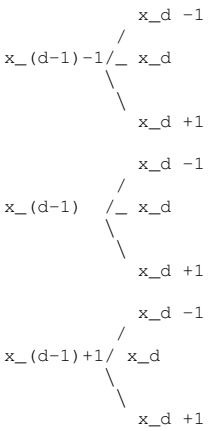
Referenced by foundNeighbours().

6.4.5.9 getListNeighboursPositionsRecursive()

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

Recursive function which browse the position possibilities tree.

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



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

Parameters

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

Returns

List of position

Definition at line 534 of file cellhandler.cpp.

References m_dimensions.

Referenced by getListNeighboursPositions().

6.4.5.10 getMaxState()

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

Return the max state of the CellHandler.

Definition at line 148 of file cellhandler.cpp.

References m_maxState.

6.4.5.11 load()

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

Load the config file in the CellHandler.

Exemple of a way to print cell states :

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

     }
}
```

Parameters

json	Json Object which contains the grid configuration

Returns

False if the Json Object is not correct

Definition at line 371 of file cellhandler.cpp.

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

Referenced by CellHandler().

6.4.5.12 nextStates()

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

Valid the state of all cells.

Definition at line 162 of file cellhandler.cpp.

References m_cells.

Referenced by Automate::run().

6.4.5.13 positionIncrement()

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

Increment the QVector given by the value choosen.

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

Parameters

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

Definition at line 464 of file cellhandler.cpp.

References m_dimensions.

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

6.4.5.14 previousStates()

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

Get all the cells to their previous states.

Definition at line 172 of file cellhandler.cpp.

References m_cells.

6.4.5.15 print()

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

Print in the given stream the CellHandler.

Parameters

stream	Stream to print into

Definition at line 306 of file cellhandler.cpp.

References begin(), and end().

6.4.5.16 reset()

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

Reset all the cells to the 1st state.

Definition at line 184 of file cellhandler.cpp.

References m_cells.

6.4.5.17 save()

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

Save the CellHandler current configuration in the file given.

Parameters

filename	Path to the file

Returns

False if there was a problem

Exceptions

QString	Impossible to open the file

Definition at line 199 of file cellhandler.cpp.

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

Referenced by Automate::saveCells().

6.4.6 Member Data Documentation

6.4.6.1 m_cells

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

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

Definition at line 135 of file cellhandler.h.

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

6.4.6.2 m_dimensions

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

Vector of x dimensions.

Definition at line 134 of file cellhandler.h.

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

6.4.6.3 m_maxState

```
unsigned int CellHandler::m_maxState  [private]
```

Definition at line 136 of file cellhandler.h.

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

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

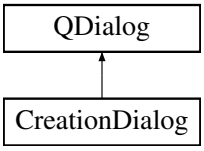
- cellhandler.h
- cellhandler.cpp

6.5 CreationDialog Class Reference

Automaton creation dialog box.

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

Inheritance diagram for CreationDialog:



Public Slots

- void processSettings ()

Signals

- void settingsFilled (const QVector< unsigned int > dimensions, CellHandler::generationTypes type=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 574 of file cellhandler.cpp.

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

6.6.3 Member Function Documentation

6.6.3.1 changedDimension()

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

Definition at line 80 of file cellhandler.h.

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

6.6.3.2 operator"!=()

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

Definition at line 79 of file cellhandler.h.

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

6.6.3.3 operator*()

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

Get the current cell.

Definition at line 75 of file cellhandler.h.

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

6.6.3.4 operator++()

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

Increment the current position and handle dimension changes.

Definition at line 47 of file cellhandler.h.

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

6.6.3.5 operator->()

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

Get the current cell.

Definition at line 71 of file cellhandler.h.

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

6.6.4 Friends And Related Function Documentation

6.6.4.1 CellHandler

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

Definition at line 43 of file cellhandler.h.

6.6.5 Member Data Documentation

6.6.5.1 m_changedDimension

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

Save the number of dimension change.

Definition at line 91 of file cellhandler.h.

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

6.6.5.2 m_finished

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

If we reach the last position.

Definition at line 89 of file cellhandler.h.

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

6.6.5.3 m_handler

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

CellHandler to go through.

Definition at line 87 of file cellhandler.h.

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

6.6.5.4 m_position

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

Current position of the iterator.

Definition at line 88 of file cellhandler.h.

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

6.6.5.5 m_zero

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

Nul vector of the good dimension (depend of m_handler)

Definition at line 90 of file cellhandler.h.

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

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

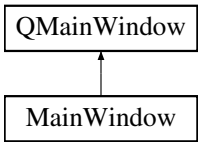
- cellhandler.h
- cellhandler.cpp

6.7 MainWindow Class Reference

Simulation window.

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

Inheritance diagram for MainWindow:



Public Slots

- void openFile ()

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

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

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

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

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

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

Public Member Functions

- MainWindow (QWidget *parent=nullptr)

Private Member Functions

- void createIcons ()

  Creates Icons for the MainWindow.
- void createActions ()

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

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

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

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

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

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

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

Private Attributes

- QTabWidget * m_tabs
- QIcon m_fastBackwardIcon

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

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

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

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

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

6.7.1 Detailed Description

Simulation window.

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

Definition at line 18 of file mainwindow.h.

6.7.2 Constructor & Destructor Documentation

6.7.2.1 MainWindow()

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

Definition at line 4 of file mainwindow.cpp.

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

6.7.3 Member Function Documentation

6.7.3.1 addAutomatonRuleFile

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

Definition at line 373 of file mainwindow.cpp.

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

Referenced by openFile(), and receiveCellHandler().

6.7.3.2 addAutomatonRules

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

Definition at line 366 of file mainwindow.cpp.

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

Referenced by openFile(), and receiveCellHandler().

6.7.3.3 addEmptyRow()

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

Add an empty row at the end of the board.

Used only in case of 1 dimension automaton

Parameters

n	Index of the board

Definition at line 341 of file mainwindow.cpp.

References getBoard(), and m_cellSize.

Referenced by updateBoard().

6.7.3.4 closeTab

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

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

Definition at line 359 of file mainwindow.cpp.

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

Referenced by createTabs().

6.7.3.5 createActions()

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

Creates and connects QActions and associated buttons for the MainWindow.

Definition at line 53 of file mainwindow.cpp.

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

Referenced by MainWindow().

6.7.3.6 createBoard()

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

6.7.3.7 createIcons()

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

Creates Icons for the MainWindow.

Definition at line 23 of file mainwindow.cpp.

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

Referenced by MainWindow().

6.7.3.8 createTab()

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

Creates a new Tab with an empty board.

Definition at line 135 of file mainwindow.cpp.

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

Referenced by openFile(), and receiveCellHandler().

6.7.3.9 createTabs()

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

Creates a QTabWidget for the main window and displays it.

Definition at line 327 of file mainwindow.cpp.

References closeTab(), and m_tabs.

Referenced by openFile(), and receiveCellHandler().

6.7.3.10 createToolBar()

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

Creates the toolBar for the MainWindow.

Definition at line 96 of file mainwindow.cpp.

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

Referenced by MainWindow().

6.7.3.11 forward

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

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

Definition at line 314 of file mainwindow.cpp.

References nextState().

Referenced by createActions().

6.7.3.12 getBoard()

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

Definition at line 319 of file mainwindow.cpp.

References m_tabs.

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

6.7.3.13 handlePlayPause

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

Definition at line 377 of file mainwindow.cpp.

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

Referenced by createActions().

6.7.3.14 nextState()

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

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

Definition at line 252 of file mainwindow.cpp.

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

Referenced by forward().

6.7.3.15 openCreationWindow

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

Opens the automaton creation window.

Definition at line 217 of file mainwindow.cpp.

References receiveCellHandler().

Referenced by createActions().

6.7.3.16 openFile

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

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

Definition at line 179 of file mainwindow.cpp.

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

Referenced by createActions().

6.7.3.17 receiveCellHandler

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

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

Definition at line 230 of file mainwindow.cpp.

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

Referenced by openCreationWindow().

6.7.3.18 reset

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

Definition at line 409 of file mainwindow.cpp.

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

Referenced by createActions().

6.7.3.19 runAutomaton

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

Definition at line 400 of file mainwindow.cpp.

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

Referenced by handlePlayPause().

6.7.3.20 saveToFile

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

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

Definition at line 199 of file mainwindow.cpp.

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

Referenced by closeTab(), and createActions().

6.7.3.21 updateBoard()

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

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

Definition at line 269 of file mainwindow.cpp.

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

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

6.7.4 Member Data Documentation

6.7.4.1 m_boardHSize

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

Toolbar containing the buttons.

Board size settings

Definition at line 66 of file mainwindow.h.

6.7.4.2 m_boardVSize

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

Definition at line 67 of file mainwindow.h.

6.7.4.3 m_cellSize

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

Definition at line 68 of file mainwindow.h.

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

6.7.4.4 m_fastBackward

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

Definition at line 40 of file mainwindow.h.

6.7.4.5 m_fastBackwardBt

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

Definition at line 51 of file mainwindow.h.

6.7.4.6 m_fastBackwardIcon

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

Icons.

Definition at line 26 of file mainwindow.h.

Referenced by createIcons().

6.7.4.7 m_fastForward

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

Definition at line 39 of file mainwindow.h.

Referenced by createActions().

6.7.4.8 m_fastForwardBt

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

Definition at line 50 of file mainwindow.h.

Referenced by createActions(), and createToolBar().

6.7.4.9 m_fastForwardIcon

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

Definition at line 27 of file mainwindow.h.

Referenced by createActions(), and createIcons().

6.7.4.10 m_newAutomate

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

Definition at line 61 of file mainwindow.h.

6.7.4.11 m_newAutomaton

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

Definition at line 43 of file mainwindow.h.

Referenced by createActions().

6.7.4.12 m_newAutomatonBt

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

Definition at line 54 of file mainwindow.h.

Referenced by createActions(), and createToolBar().

6.7.4.13 m_newIcon

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

Definition at line 30 of file mainwindow.h.

Referenced by createActions(), and createIcons().

6.7.4.14 m_nextState

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

Definition at line 37 of file mainwindow.h.

6.7.4.15 m_nextStateBt

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

Definition at line 48 of file mainwindow.h.

6.7.4.16 m_openAutomaton

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

Definition at line 41 of file mainwindow.h.

Referenced by createActions().

6.7.4.17 m_openAutomatonBt

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

Definition at line 52 of file mainwindow.h.

Referenced by createActions(), and createToolBar().

6.7.4.18 m_openIcon

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

Definition at line 32 of file mainwindow.h.

Referenced by createActions(), and createIcons().

6.7.4.19 m_pauseIcon

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

Definition at line 29 of file mainwindow.h.

Referenced by createIcons(), and handlePlayPause().

6.7.4.20 m_playIcon

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

Definition at line 28 of file mainwindow.h.

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

6.7.4.21 m_playPause

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

Actions.

Definition at line 36 of file mainwindow.h.

Referenced by createActions().

6.7.4.22 m_playPauseBt

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

Buttons.

Definition at line 47 of file mainwindow.h.

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

6.7.4.23 m_previousState

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

Definition at line 38 of file mainwindow.h.

6.7.4.24 m_previousStateBt

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

Definition at line 49 of file mainwindow.h.

6.7.4.25 m_resetAutomaton

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

Definition at line 44 of file mainwindow.h.

Referenced by createActions().

6.7.4.26 m_resetBt

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

Definition at line 55 of file mainwindow.h.

Referenced by createActions(), and createToolBar().

6.7.4.27 m_resetIcon

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

Definition at line 33 of file mainwindow.h.

Referenced by createActions(), and createIcons().

6.7.4.28 m_saveAutomaton

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

Definition at line 42 of file mainwindow.h.

Referenced by createActions().

6.7.4.29 m_saveAutomatonBt

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

Definition at line 53 of file mainwindow.h.

Referenced by createActions(), and createToolBar().

6.7.4.30 m_saveIcon

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

Definition at line 31 of file mainwindow.h.

Referenced by createActions(), and createIcons().

6.7.4.31 m_tabs

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

Definition at line 22 of file mainwindow.h.

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

6.7.4.32 m_timer

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

Simulation time step duration input.

Definition at line 59 of file mainwindow.h.

Referenced by handlePlayPause().

6.7.4.33 m_timeStep

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

Definition at line 58 of file mainwindow.h.

Referenced by createToolBar(), and handlePlayPause().

6.7.4.34 m_toolBar

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

Definition at line 63 of file mainwindow.h.

Referenced by createToolBar().

6.7.4.35 running

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

Definition at line 62 of file mainwindow.h.

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

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

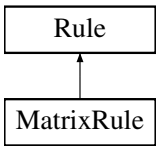
- mainwindow.h
- mainwindow.cpp

6.8 MatrixRule Class Reference

Manage specific rules, about specific values of specific neighbour.

`#include <matrixrule.h>`

Inheritance diagram for MatrixRule:



Public Member Functions

- MatrixRule (unsigned int finalState, QVector< unsigned int > currentStates=QVector< unsigned int >())
  Constructor.
- virtual bool matchCell (const Cell *cell) const
  Tells if the cell match the rule.
- void addNeighbourState (QVector< short > relativePosition, unsigned int matchState)
  Add a possible state to a relative position.
- void addNeighbourState (QVector< short > relativePosition, QVector< unsigned int > matchStates)
  Add multiples possible states to existents states.
- QJsonObject toJson () const
  Return a QJsonObject to save the rule.

Private Attributes

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

Additional Inherited Members

6.8.1 Detailed Description

Manage specific rules, about specific values of specific neighbour.

Definition at line 13 of file matrixrule.h.

6.8.2 Constructor & Destructor Documentation

6.8.2.1 MatrixRule()

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

Constructor.

Parameters

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

Definition at line 21 of file matrixrule.cpp.

6.8.3 Member Function Documentation

6.8.3.1 addNeighbourState() [1/2]

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

Add a possible state to a relative position.

Definition at line 60 of file matrixrule.cpp.

References m_matrix.

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

6.8.3.2 addNeighbourState() [2/2]

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

Add multiples possible states to existents states.

Definition at line 67 of file matrixrule.cpp.

References m_matrix.

6.8.3.3 matchCell()

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

Tells if the cell match the rule.

Parameters

cell	Cell to test

Returns

True if the cell match the rule

Implements Rule.

Definition at line 30 of file matrixrule.cpp.

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

6.8.3.4 toJson()

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

Return a QJsonObject to save the rule.

Implements Rule.

Definition at line 75 of file matrixrule.cpp.

References m_matrix, and Rule::toJson().

6.8.4 Member Data Documentation

6.8.4.1 m_matrix

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

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

Definition at line 28 of file matrixrule.h.

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

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

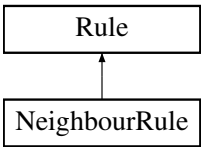
- matrixrule.h
- matrixrule.cpp

6.9 NeighbourRule Class Reference

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

`#include <neighbourrule.h>`

Inheritance diagram for NeighbourRule:



Public Member Functions

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

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

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

  Return a QJsonObject to save the rule.

Private Member Functions

- bool inInterval (unsigned int matchingNeighbours) const

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

Private Attributes

- QPair< unsigned int, unsigned int > m_neighbourInterval

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

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

Additional Inherited Members

6.9.1 Detailed Description

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

Definition at line 13 of file neighbourrule.h.

6.9.2 Constructor & Destructor Documentation

6.9.2.1 NeighbourRule()

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

Constructs a neighbour rule with the parameters.

Definition at line 95 of file neighbourrule.cpp.

References m_neighbourInterval.

6.9.2.2 ~NeighbourRule()

```
NeighbourRule::~NeighbourRule ( )
```

Definition at line 104 of file neighbourrule.cpp.

6.9.3 Member Function Documentation

6.9.3.1 inInterval()

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

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

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

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

1 : cell is alive 0 : cell is dead

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

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

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

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

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

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


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

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

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

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

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

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


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

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

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

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

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

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

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

Parameters

matchingNeighbours	Number of neighbours matching the rule condition regarding their values

Returns

True if the number of neighbours matches with the interval condition

Definition at line 84 of file neighbourrule.cpp.

References m_neighbourInterval.

Referenced by matchCell().

6.9.3.2 matchCell()

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

Checks if the input cell satisfies the rule condition.

Parameters

c	current cell

Returns

True if the cell matches the rule condition

Implements Rule.

Definition at line 115 of file neighbourrule.cpp.

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

6.9.3.3 toJson()

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

Return a QJsonObject to save the rule.

Implements Rule.

Definition at line 146 of file neighbourrule.cpp.

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

6.9.4 Member Data Documentation

6.9.4.1 m_neighbourInterval

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

Stores the rule condition regarding the number of neighbours.

Definition at line 16 of file neighbourrule.h.

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

6.9.4.2 m_neighbourPossibleValues

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

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

Definition at line 18 of file neighbourrule.h.

Referenced by matchCell(), and toJson().

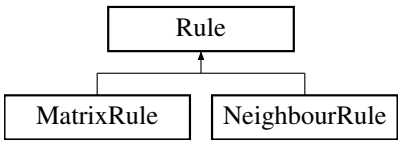
The documentation for this class was generated from the following files:

- neighbourrule.h
- neighbourrule.cpp

6.10 Rule Class Reference

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

Inheritance diagram for Rule:



Public Member Functions

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

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

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

Protected Attributes

- QVector< unsigned int > m_currentCellPossibleValues

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

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

6.10.1 Detailed Description

Definition at line 13 of file rule.h.

6.10.2 Constructor & Destructor Documentation

6.10.2.1 Rule()

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

Definition at line 3 of file rule.cpp.

6.10.3 Member Function Documentation

6.10.3.1 getCellOutputState()

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

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

Definition at line 26 of file rule.cpp.

References m_cellOutputState.

6.10.3.2 matchCell()

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

Verify if the cell match the rule.

Using :

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

Parameters

c	Cell to test

Implemented in NeighbourRule, and MatrixRule.

6.10.3.3 toJson()

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

Implemented in NeighbourRule, and MatrixRule.

Definition at line 9 of file rule.cpp.

References m_cellOutputState, and m_currentCellPossibleValues.

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

6.10.4 Member Data Documentation

6.10.4.1 m_cellOutputState

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

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

Definition at line 17 of file rule.h.

Referenced by getCellOutputState(), and toJson().

6.10.4.2 m_currentCellPossibleValues

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

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

Definition at line 16 of file rule.h.

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

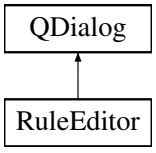
The documentation for this class was generated from the following files:

- rule.h
- rule.cpp

6.11 RuleEditor Class Reference

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

Inheritance diagram for RuleEditor:



Public Slots

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

Signals

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

Public Member Functions

- RuleEditor (QWidget *parent=nullptr)

Private Attributes

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

6.11.1 Detailed Description

Definition at line 7 of file ruleeditor.h.

6.11.2 Constructor & Destructor Documentation

6.11.2.1 RuleEditor()

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

Definition at line 3 of file ruleeditor.cpp.

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

6.11.3 Member Function Documentation

6.11.3.1 addRule

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

Definition at line 66 of file ruleeditor.cpp.

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

Referenced by RuleEditor().

6.11.3.2 fileImported

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

Referenced by importFile().

6.11.3.3 importFile

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

Definition at line 99 of file ruleeditor.cpp.

References fileImported().

Referenced by RuleEditor().

6.11.3.4 removeRule

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

Definition at line 89 of file ruleeditor.cpp.

References m_rules, and m_rulesListWidget.

Referenced by RuleEditor().

6.11.3.5 rulesFilled

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

Referenced by sendRules().

6.11.3.6 sendRules

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

Definition at line 94 of file ruleeditor.cpp.

References m_rules, and rulesFilled().

Referenced by RuleEditor().

6.11.4 Member Data Documentation

6.11.4.1 m_addBt

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

Definition at line 20 of file ruleeditor.h.

Referenced by RuleEditor().

6.11.4.2 m_currentStatesEdit

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

Definition at line 15 of file ruleeditor.h.

Referenced by addRule(), and RuleEditor().

6.11.4.3 m_doneBt

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

Definition at line 21 of file ruleeditor.h.

Referenced by RuleEditor().

6.11.4.4 m_importBt

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

Definition at line 23 of file ruleeditor.h.

Referenced by RuleEditor().

6.11.4.5 m_lowerNeighbourBox

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

Definition at line 18 of file ruleeditor.h.

Referenced by addRule(), and RuleEditor().

6.11.4.6 m_neighbourStatesEdit

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

Definition at line 16 of file ruleeditor.h.

Referenced by addRule(), and RuleEditor().

6.11.4.7 m_outputStateBox

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

Definition at line 14 of file ruleeditor.h.

Referenced by addRule(), and RuleEditor().

6.11.4.8 m_removeBt

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

Definition at line 22 of file ruleeditor.h.

Referenced by RuleEditor().

6.11.4.9 m_rules

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

Definition at line 10 of file ruleeditor.h.

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

6.11.4.10 m_rulesListWidget

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

Definition at line 11 of file ruleeditor.h.

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

6.11.4.11 m_rulesTable

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

Definition at line 12 of file ruleeditor.h.

6.11.4.12 m_upperNeighbourBox

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

Definition at line 17 of file ruleeditor.h.

Referenced by addRule(), and RuleEditor().

6.11.4.13 selectedRule

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

Definition at line 25 of file ruleeditor.h.

Referenced by RuleEditor().

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

- ruleeditor.h
- ruleeditor.cpp

Chapter 7

File Documentation

7.1 automate.cpp File Reference

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

Functions

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

7.1.1 Function Documentation

7.1.1.1 generate1DRules()

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

Definition at line 315 of file automate.cpp.

References getRuleFromNumber().

7.1.1.2 getRuleFromNumber()

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

Definition at line 338 of file automate.cpp.

References MatrixRule::addNeighbourState().

Referenced by generate1DRules().

7.2 automate.cpp

```
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         delete *it;
```

```
00185     }
00186 }
00187
00191 bool Automate::saveRules(QString filename) const
00192 {
00193     QFile ruleFile(filename);
00194     if (!ruleFile.open(QIODevice::WriteOnly | QIODevice::Text)) {
00195         qWarning("Couldn't open given file.");
00196         throw QString(QObject::tr("Couldn't open given file"));
00197     }
00198
00199     QJsonArray array;
00200
00201     for (QList<const Rule*>::const_iterator it = m_rules.cbegin(); it !=
      m_rules.cend(); ++it)
00202         array.append((*it)->toJson());
00203
00204     QJsonDocument doc(array);
00205
00206     ruleFile.write(doc.toJson());
00207
00208     return true;
00209 }
00210
00213 bool Automate::saveCells(QString filename) const
00214 {
00215     if (m_cellHandler != nullptr)
00216         return m_cellHandler->save(filename);
00217     return false;
00218 }
00219
00222 bool Automate::saveAll(QString cellHandlerFilename, QString rulesFilename) const
00223 {
00224     return saveRules(rulesFilename) && saveCells(cellHandlerFilename);
00225 }
00226
00229 void Automate::addRule(const Rule *newRule)
00230 {
00231     m_rules.push_back(newRule);
00232 }
00233
00240 void Automate::setRulePriority(const Rule *rule, unsigned int newPlace)
00241 {
00242     m_rules.move(m_rules.indexOf(rule), newPlace);
00243 }
00244
00247 const QList<const Rule *> &Automate::getRules() const
00248 {
00249    return m_rules;
00250 }
00251
00256 bool Automate::run(unsigned int nbSteps) //void instead ?
00257 {
00258     for(unsigned int i = 0; i<nbSteps; ++i)
00259     {
00260         for (CellHandler::iterator it = m_cellHandler->
      begin(); it != m_cellHandler->end(); ++it)
00261         {
00262             for (QList<const Rule*>::iterator rule = m_rules.begin(); rule !=
      m_rules.end() ; ++rule)
00263             {
00264                 if((*rule)->matchCell(*it)) //if the cell matches with the rule, its state is changed
00265                 {
00266                     it->setState((*rule)->getCellOutputState());
00267                     break;
00268                 }
00269             }
00270
00271
00272         }
00273         m_cellHandler->nextStates(); //apply the changes to all the cells
       simultaneously
00274     }
00275     return true;
00276
00277 }
00278
00281 const CellHandler &Automate::getCellHandler() const
00282 {
00283     return *m_cellHandler;
00284 }
00285
00286 void Automate::addRuleFile(QString filename){
00287     QFile ruleFile(filename);
00288     if (!ruleFile.open(QIODevice::ReadOnly | QIODevice::Text)) {
00289         qWarning("Couldn't open given file.");
00290         throw QString(QObject::tr("Couldn't open given file"));
```

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

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

7.3 automate.h File Reference

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

Classes

- class Automate

Functions

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

7.3.1 Function Documentation

7.3.1.1 generate1DRules()

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

Definition at line 315 of file automate.cpp.

References getRuleFromNumber().

7.3.1.2 getRuleFromNumber()

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

Definition at line 338 of file automate.cpp.

References MatrixRule::addNeighbourState().

Referenced by generate1DRules().

7.4 automate.h

```
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.push(m_nextState);
00045 }
00046
00049 unsigned int Cell::getState() const
00050 {
00051     return m_states.top();
00052 }
00053
00058 bool Cell::back()
00059 {
00060     if (m_states.size() <= 1)
00061         return false;
```

```
00062     m_states.pop();
00063     m_nextState = m_states.top();
00064     return true;
00065 }
00066
00069 void Cell::reset()
00070 {
00071     while (m_states.size() > 2)
00072         m_states.pop();
00073     m_nextState = m_states.top();
00074 }
00075
00083 bool Cell::addNeighbour(const Cell* neighbour, const QVector<short> relativePosition)
00084 {
00085     if (m_neighbours.count(relativePosition))
00086         return false;
00087
00088     m_neighbours.insert(relativePosition, neighbour);
00089     return true;
00090 }
00091
00096 QMap<QVector<short>, const Cell *> Cell::getNeighbours() const
00097 {
00098     return m_neighbours;
00099 }
00100
00103 const Cell *Cell::getNeighbour(QVector<short> relativePosition) const
00104 {
00105     return m_neighbours.value(relativePosition, nullptr);
00106 }
00107
00110 unsigned int Cell::countNeighbours(unsigned int filterState) const
00111 {
00112     unsigned int count = 0;
00113     for (QMap<QVector<short>, const Cell*>::const_iterator it = m_neighbours.begin(); it !=
      m_neighbours.end(); ++it)
00114     {
00115         if ((*it)->getState() == filterState)
00116             count++;
00117     }
00118     return count;
00119 }
00120
00123 unsigned int Cell::countNeighbours() const
00124 {
00125     unsigned int count = 0;
00126     for (QMap<QVector<short>, const Cell*>::const_iterator it = m_neighbours.begin(); it !=
      m_neighbours.end(); ++it)
00127     {
00128         if ((*it)->getState() != 0)
00129             count++;
00130     }
00131     return count;
00132 }
00133
00140 QVector<short> Cell::getRelativePosition(const QVector<unsigned int> cellPosition,
      const QVector<unsigned int> neighbourPosition)
00141 {
00142     if (cellPosition.size() != neighbourPosition.size())
00143     {
00144         throw QString(QObject::tr("Different size of position vectors"));
00145     }
00146     QVector<short> relativePosition;
00147     for (short i = 0; i < cellPosition.size(); i++)
00148         relativePosition.push_back(neighbourPosition.at(i) - cellPosition.at(i));
00149
00150     return relativePosition;
00151 }
```

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     m_maxState(stateMax)
00100 {
00101     m_dimensions = dimensions;
00102     QVector<unsigned int> position;
00103     unsigned int size = 1;
00104
00105     // Set position vector to 0
00106
00107     for (unsigned short i = 0; i < m_dimensions.size(); i++)
00108     {
00109         position.push_back(0);
00110         size *= m_dimensions.at(i);
00111     }
00112
00113
00114     // Creation of cells
00115     for (unsigned int j = 0; j < size; j++)
00116     {
00117         m_cells.insert(position, new Cell(0));
00118
00119         positionIncrement(position);
00120     }
00121
00122     foundNeighbours();
00123
00124     if (type != empty)
00125         generate(type, stateMax, density);
00126
00127 }
00128
00131 CellHandler::~CellHandler()
00132 {
00133     for (QMap<QVector<unsigned int>, Cell* >::iterator it = m_cells.begin(); it !=
      m_cells.end(); ++it)
00134     {
00135         delete it.value();
00136     }
00137 }
00138
00141 Cell *CellHandler::getCell(const QVector<unsigned int> position) const
00142 {
00143     return m_cells.value(position);
00144 }
00145
00148 unsigned int CellHandler::getMaxState() const
00149 {
00150     return m_maxState;
00151 }
00152
00155 QVector<unsigned int> CellHandler::getDimensions() const
00156 {
00157     return m_dimensions;
00158 }
00159
00162 void CellHandler::nextStates() const
```

```
00163 {
00164     for (QMap<QVector<unsigned int>, Cell* >::const_iterator it =
      m_cells.begin(); it != m_cells.end(); ++it)
00165     {
00166         it.value()->validState();
00167     }
00168 }
00169
00172 bool CellHandler::previousStates() const
00173 {
00174     for (QMap<QVector<unsigned int>, Cell* >::const_iterator it =
      m_cells.begin(); it != m_cells.end(); ++it)
00175     {
00176         if (!it.value()->back())
00177             return false;
00178     }
00179     return true;
00180 }
00181
00184 void CellHandler::reset() const
00185 {
00186     for (QMap<QVector<unsigned int>, Cell* >::const_iterator it =
      m_cells.begin(); it != m_cells.end(); ++it)
00187     {
00188         it.value()->reset();
00189     }
00190 }
00191
00199 bool CellHandler::save(QString filename) const
00200 {
00201     QFile saveFile(filename);
00202     if (!saveFile.open(QIODevice::WriteOnly)) {
00203         qWarning("Couldn't create or open given file.");
00204         throw QString(QObject::tr("Couldn't create or open given file"));
00205     }
00206
00207     QJsonObject json;
00208     QString stringDimension;
00209     // Creation of the dimension string
00210     for (int i = 0; i < m_dimensions.size(); i++)
00211     {
00212         if (i != 0)
00213             stringDimension.push_back("x");
00214         stringDimension.push_back(QString::number(m_dimensions.at(i)));
00215     }
00216     json["dimensions"] = QJsonValue(stringDimension);
00217
00218     QJsonArray cells;
00219     for (CellHandler::const_iterator it = begin(); it !=
      end(); ++it)
00220     {
00221         cells.append(QJsonValue((int)it->getState()));
00222     }
00223     json["cells"] = cells;
00224
00225     json["maxState"] = QJsonValue((int)m_maxState);
00226
00227
00228     QJsonDocument saveDoc(json);
00229     saveFile.write(saveDoc.toJson());
00230
00231     saveFile.close();
00232     return true;
00233 }
00234
00241 void CellHandler::generate(CellHandler::generationTypes
      type, unsigned int stateMax, unsigned short density)
00242 {
00243     if (type == random)
00244     {
00245         QVector<unsigned int> position;
00246         for (unsigned short i = 0; i < m_dimensions.size(); i++)
00247         {
00248             position.push_back(0);
00249         }
00250         QRandomGenerator generator((float)qrand()*(float)time_t()/RAND_MAX);
00251         for (int j = 0; j < m_cells.size(); j++)
00252         {
00253             unsigned int state = 0;
00254             // 0 have (1-density)% of chance of being generate
00255             if (generator.generateDouble()*100.0 < density)
00256                 state = (float)(generator.generateDouble()*stateMax) +1;
00257             if (state > stateMax)
00258                 state = stateMax;
00259             m_cells.value(position)->forceState(state);
00260
00261             positionIncrement(position);
```

```
00262         }
00263     }
00264     else if (type == symetric)
00265     {
00266         QVector<unsigned int> position;
00267         for (short i = 0; i < m_dimensions.size(); i++)
00268         {
00269             position.push_back(0);
00270         }
00271 
00272         QRandomGenerator generator((float)qrand()*(float)time_t()/RAND_MAX);
00273         QVector<unsigned int> savedStates;
00274         for (int j = 0; j < m_cells.size(); j++)
00275         {
00276             if (j % m_dimensions.at(0) == 0)
00277                 savedStates.clear();
00278             if (j % m_dimensions.at(0) < (m_dimensions.at(0)+1) / 2)
00279             {
00280                 unsigned int state = 0;
00281                 // 0 have (1-density)% of chance of being generate
00282                 if (generator.generateDouble()*100.0 < density)
00283                     state = (float)(generator.generateDouble()*stateMax) +1;
00284                 if (state > stateMax)
00285                     state = stateMax;
00286                 savedStates.push_back(state);
00287                 m_cells.value(position)->forceState(state);
00288             }
00289             else
00290             {
00291                 unsigned int i = savedStates.size() - (j % m_dimensions.at(0) - (
      m_dimensions.at(0)-1)/2 + (m_dimensions.at(0) % 2 == 0 ? 0 : 1));
00292                 m_cells.value(position)->forceState(savedStates.at(i));
00293             }
00294             positionIncrement(position);
00295 
00296 
00297         }
00298 
00299     }
00300 }
00301 
00306 void CellHandler::print(std::ostream &stream) const
00307 {
00308     for (const_iterator it = begin(); it != end(); ++it)
00309     {
00310         for (unsigned int d = 0; d < it.changedDimension(); d++)
00311             stream << std::endl;
00312         stream << it->getState() << " ";
00313 
00314     }
00315 
00316 }
00317 
00320 CellHandler::iterator CellHandler::begin()
00321 {
00322     return iterator(this);
00323 }
00324 
00327 CellHandler::const_iterator CellHandler::begin() const
00328 {
00329     return const_iterator(this);
00330 }
00331 
00336 bool CellHandler::end() const
00337 {
00338     return true;
00339 }
00340 
00371 bool CellHandler::load(const QJsonObject &json)
00372 {
00373     if (!json.contains("dimensions") || !json["dimensions"].isString())
00374         return false;
00375 
00376     // RegExp to validate dimensions field format : "10x10"
00377     QRegExpValidator dimensionValidator(QRegExp("([0-9]*x?)*"));
00378     QString stringDimensions = json["dimensions"].toString();
00379     int pos= 0;
00380     if (dimensionValidator.validate(stringDimensions, pos) != QRegExpValidator::Acceptable)
00381         return false;
00382 
00383     // Split of dimensions field : "10x10" => "10", "10"
00384     QRegExp rx("x");
00385     QStringList list = json["dimensions"].toString().split(rx, QString::SkipEmptyParts);
00386 
00387     int product = 1;
00388     // Dimensions construction
00389     for (int i = 0; i < list.size(); i++)
```

```
00390     {
00391         product = product * list.at(i).toInt();
00392         m_dimensions.push_back(list.at(i).toInt());
00393     }
00394     if (!json.contains("cells") || !json["cells"].isArray())
00395         return false;
00396
00397     QJsonArray cells = json["cells"].toArray();
00398     if (cells.size() != product)
00399         return false;
00400
00401     QVector<unsigned int> position;
00402     // Set position vector to 0
00403     for (unsigned short i = 0; i < m_dimensions.size(); i++)
00404     {
00405         position.push_back(0);
00406     }
00407
00408
00409     // Creation of cells
00410     for (int j = 0; j < cells.size(); j++)
00411     {
00412         if (!cells.at(j).isDouble())
00413             return false;
00414         if (cells.at(j).toDouble() < 0)
00415             return false;
00416         m_cells.insert(position, new Cell(cells.at(j).toDouble()));
00417
00418         positionIncrement(position);
00419     }
00420
00421     if (!json.contains("maxState") || !json["maxState"].isDouble())
00422         return false;
00423     m_maxState = json["maxState"].toInt();
00424
00425     return true;
00426
00427 }
00428
00434 void CellHandler::foundNeighbours()
00435 {
00436     QVector<unsigned int> currentPosition;
00437     // Set position vector to 0
00438     for (unsigned short i = 0; i < m_dimensions.size(); i++)
00439     {
00440         currentPosition.push_back(0);
00441     }
00442     // Modification of all the cells
00443     for (int j = 0; j < m_cells.size(); j++)
00444     {
00445         // Get the list of the neighbours positions
00446         // This function is recursive
00447         QVector<QVector<unsigned int> > listPosition(getListNeighboursPositions(
      currentPosition));
00448
00449         // Adding neighbours
00450         for (int i = 0; i < listPosition.size(); i++)
00451             m_cells.value(currentPosition)->addNeighbour(m_cells.value(listPosition.at(i)),
      Cell::getRelativePosition(currentPosition, listPosition.at(i)));
00452         positionIncrement(currentPosition);
00453     }
00454
00455 }
00456
00464 void CellHandler::positionIncrement(QVector<unsigned int> &pos, unsigned int
      value) const
00465 {
00466     pos.replace(0, pos.at(0) + value); // adding the value to the first digit
00467
00468     // Carry management
00469     for (unsigned short i = 0; i < m_dimensions.size(); i++)
00470     {
00471         if (pos.at(i) >= m_dimensions.at(i) && pos.at(i) <
      m_dimensions.at(i)*2)
00472         {
00473             pos.replace(i, 0);
00474             if (i + 1 != m_dimensions.size())
00475                 pos.replace(i+1, pos.at(i+1)+1);
00476         }
00477         else if (pos.at(i) >= m_dimensions.at(i))
00478         {
00479             pos.replace(i, pos.at(i) - m_dimensions.at(i));
00480             if (i + 1 != m_dimensions.size())
00481                 pos.replace(i+1, pos.at(i+1)+1);
00482             i--;
00483         }
00484
```

```
00485     }
00486 }
00487
00493 QVector<QVector<unsigned int> >& CellHandler::getListNeighboursPositions
      (const QVector<unsigned int> position) const
00494 {
00495     QVector<QVector<unsigned int> > *list = getListNeighboursPositionsRecursive
      (position, position.size(), position);
00496     // We remove the position of the cell
00497     list->removeAll(position);
00498     return *list;
00499 }
00500
00534 QVector<QVector<unsigned int> >*
      CellHandler::getListNeighboursPositionsRecursive(const
      QVector<unsigned int> position, unsigned int dimension, QVector<unsigned int> lastAdd) const
00535 {
00536     if (dimension == 0) // Stop condition
00537     {
00538         QVector<QVector<unsigned int> > *list = new QVector<QVector<unsigned int> >;
00539         return list;
00540     }
00541     QVector<QVector<unsigned int> > *listPositions = new QVector<QVector<unsigned int> >;
00542
00543     QVector<unsigned int> modifiedPosition(lastAdd);
00544
00545     // "x_d - 1" tree
00546     if (modifiedPosition.at(dimension-1) != 0) // Avoid "negative" position
00547         modifiedPosition.replace(dimension-1, position.at(dimension-1) - 1);
00548     listPositions->append(*getListNeighboursPositionsRecursive(position,
       dimension -1, modifiedPosition));
00549     if (!listPositions->count(modifiedPosition))
00550         listPositions->push_back(modifiedPosition);
00551
00552     // "x_d" tree
00553     modifiedPosition.replace(dimension-1, position.at(dimension-1));
00554     listPositions->append(*getListNeighboursPositionsRecursive(position,
       dimension -1, modifiedPosition));
00555     if (!listPositions->count(modifiedPosition))
00556         listPositions->push_back(modifiedPosition);
00557
00558     // "x_d + 1" tree
00559     if (modifiedPosition.at(dimension -1) + 1 < m_dimensions.at(dimension-1)) // Avoid position
       out of the cell space
00560         modifiedPosition.replace(dimension-1, position.at(dimension-1) +1);
00561     listPositions->append(*getListNeighboursPositionsRecursive(position,
       dimension -1, modifiedPosition));
00562     if (!listPositions->count(modifiedPosition))
00563         listPositions->push_back(modifiedPosition);
00564
00565     return listPositions;
00566
00567 }
00568
00573 template<typename CellHandler_T, typename Cell_T>
00574 CellHandler::iteratorT<CellHandler_T,Cell_T>::iteratorT
      (CellHandler_T *handler):
00575         m_handler(handler), m_changedDimension(0)
00576 {
00577     // Initialisation of m_position
00578     for (unsigned short i = 0; i < handler->m_dimensions.size(); i++)
00579     {
00580         m_position.push_back(0);
00581     }
00582     m_zero = m_position;
00583 }
```

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     unsigned int getMaxState() const;
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     unsigned int m_maxState;
00137 };
00138
00139 template class CellHandler::iteratorT<CellHandler, Cell>;
00140 template class CellHandler::iteratorT<const CellHandler, const Cell>
      ;
00141
00142 #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     MainWindow w;
00012     w.show();
00013
00014     return app.exec();
00015
00016 }
```

7.23 mainwindow.cpp File Reference

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

7.24 mainwindow.cpp

```
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
00023 void MainWindow::createIcons(){
00024     QPixmap fastBackwardPm(":/icons/icons/fast-backward.svg");
00025     QPixmap fastBackwardHoveredPm(":/icons/icons/fast-backward-full.svg");
```

```
00026     QPixmap fastForwardPm(":/icons/icons/fast-forward.svg");
00027     QPixmap fastForwardHoveredPm(":/icons/icons/fast-forward-full.svg");
00028     QPixmap playPm(":/icons/icons/play.svg");
00029     QPixmap playHoveredPm(":/icons/icons/play-full.svg");
00030     QPixmap newPm(":/icons/icons/new.svg");
00031     QPixmap openPm(":/icons/icons/open.svg");
00032     QPixmap savePm(":/icons/icons/save.svg");
00033     QPixmap pausePm(":/icons/icons/pause.svg");
00034     QPixmap resetPm(":/icons/icons/reset.svg");
00035
00036     m_fastBackwardIcon.addPixmap(fastBackwardPm, QIcon::Normal, QIcon::Off);
00037     m_fastBackwardIcon.addPixmap(fastBackwardHoveredPm, QIcon::Active, QIcon::Off);
00038     m_fastForwardIcon.addPixmap(fastForwardPm, QIcon::Normal, QIcon::Off);
00039     m_fastForwardIcon.addPixmap(fastForwardHoveredPm, QIcon::Active, QIcon::Off);
00040     m_playIcon.addPixmap(playPm, QIcon::Normal, QIcon::Off);
00041     m_playIcon.addPixmap(playHoveredPm, QIcon::Active, QIcon::Off);
00042     m_pauseIcon.addPixmap(pausePm, QIcon::Normal, QIcon::Off);
00043     m_newIcon.addPixmap(newPm, QIcon::Normal, QIcon::Off);
00044     m_saveIcon.addPixmap(savePm, QIcon::Normal, QIcon::Off);
00045     m_openIcon.addPixmap(openPm, QIcon::Normal, QIcon::Off);
00046     m_resetIcon.addPixmap(resetPm, QIcon::Normal, QIcon::Off);
00047 }
00048
00053 void MainWindow::createActions(){
00054     m_fastForward = new QAction(m_fastForwardIcon, tr("&fast forward"), this)
    ;
00055     m_playPause = new QAction(m_playIcon, tr("Play"), this);
00056     m_saveAutomaton = new QAction(m_saveIcon, tr("Save automaton"), this);
00057     m_newAutomaton = new QAction(m_newIcon, tr("New automaton"), this);
00058     m_openAutomaton = new QAction(m_openIcon, tr("Open automaton"), this);
00059     m_resetAutomaton = new QAction(m_resetIcon, tr("Reset automaton"), this);
00060
00061
00062     m_fastForwardBt = new QToolButton(this);
00063     m_playPauseBt = new QToolButton(this);
00064     m_saveAutomatonBt = new QToolButton(this);
00065     m_newAutomatonBt = new QToolButton(this);
00066     m_openAutomatonBt = new QToolButton(this);
00067     m_resetBt = new QToolButton(this);
00068
00069     m_fastForwardBt->setDefaultAction(m_fastForward);
00070     m_playPauseBt->setDefaultAction(m_playPause);
00071     m_saveAutomatonBt->setDefaultAction(m_saveAutomaton);
00072     m_newAutomatonBt->setDefaultAction(m_newAutomaton);
00073     m_openAutomatonBt->setDefaultAction(m_openAutomaton);
00074     m_resetBt->setDefaultAction(m_resetAutomaton);
00075
00076     m_fastForwardBt->setIconSize(QSize(30,30));
00077     m_playPauseBt->setIconSize(QSize(30,30));
00078     m_saveAutomatonBt->setIconSize(QSize(30,30));
00079     m_newAutomatonBt->setIconSize(QSize(30,30));
00080     m_openAutomatonBt->setIconSize(QSize(30,30));
00081     m_resetBt->setIconSize(QSize(30,30));
00082
00083     connect(m_openAutomatonBt, SIGNAL(clicked(bool)), this, SLOT(
    openFile()));
00084     connect(m_newAutomatonBt, SIGNAL(clicked(bool)), this, SLOT(
    openCreationWindow()));
00085     connect(m_saveAutomatonBt, SIGNAL(clicked(bool)), this, SLOT(
    saveToFile()));
00086     connect(m_fastForwardBt, SIGNAL(clicked(bool)), this, SLOT(
    forward()));
00087     connect(m_playPauseBt, SIGNAL(clicked(bool)), this, SLOT(
    handlePlayPause()));
00088     connect(m_resetBt,SIGNAL(clicked(bool)), this,SLOT(reset()));
00089
00090 }
00091
00096 void MainWindow::createToolBar(){
00097     m_toolBar = new QToolBar(this);
00098     QLabel *timeStepLabel = new QLabel(tr("Timestep(ms)"),this);
00099     m_timeStep = new QSpinBox(this);
00100     m_timeStep->setMaximum(10000);
00101     m_timeStep->setValue(500);
00102     timeStepLabel->setFixedWidth(90);
00103     m_timeStep->setFixedWidth(60);
00104     m_toolBar->setMovable(false);
00105
00106     QVBoxLayout* tsLayout = new QVBoxLayout();
00107     tsLayout->addWidget(timeStepLabel, Qt::AlignCenter);
00108     tsLayout->addWidget(m_timeStep, Qt::AlignCenter);
00109
00110     QHBoxLayout *tbLayout = new QHBoxLayout(this);
00111     tbLayout->addWidget(m_newAutomatonBt, Qt::AlignCenter);
00112     tbLayout->addWidget(m_openAutomatonBt, Qt::AlignCenter);
00113     tbLayout->addWidget(m_saveAutomatonBt, Qt::AlignCenter);
00114     tbLayout->addWidget(m_resetBt, Qt::AlignCenter);
```

```
00115     tbLayout->addWidget(m_playPauseBt, Qt::AlignCenter);
00116     tbLayout->addWidget(m_fastForwardBt, Qt::AlignCenter);
00117     tbLayout->addLayout(tsLayout);
00118
00119
00120
00121
00122     tbLayout->setAlignment(Qt::AlignCenter);
00123     QWidget* wrapper = new QWidget(this);
00124     wrapper->setLayout(tbLayout);
00125     m_toolBar->addWidget(wrapper);
00126     addToolBar(m_toolBar);
00127
00128
00129 }
00130
00135 QWidget* MainWindow::createTab(){
00136     QWidget *tab = new QWidget(this);
00137     QVBoxLayout *layout = new QVBoxLayout(this);
00138     QVector<unsigned int> dimensions = AutomateHandler::getAutomateHandler
      ().getAutomate(AutomateHandler::getAutomateHandler().
      getNumberAutomates()-1)->getCellHandler().getDimensions();
00139     int boardVSize = 0;
00140     int boardHSize = 0;
00141     if(dimensions.size() > 1){
00142         boardVSize = dimensions[0];
00143         boardHSize = dimensions[1];
00144     }
00145     else{
00146         boardVSize = 1;
00147         boardHSize = dimensions[0];
00148     }
00149
00150     QTableWidget* board = new QTableWidget(boardVSize, boardHSize, this);
00151         board->setFixedSize(boardHSize*m_cellSize,boardVSize*
      m_cellSize);
00152         //setMinimumSize(m_boardHSize*m_cellSize,100+m_boardVSize*m_cellSize);
00153         board->horizontalHeader()->setVisible(false);
00154         board->verticalHeader()->setVisible(false);
00155         board->setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
00156         board->setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
00157         board->setEditTriggers(QAbstractItemView::NoEditTriggers);
00158         for(unsigned int col = 0; col < boardHSize; ++col)
00159             board->setColumnWidth(col, m_cellSize);
00160         for(unsigned int row = 0; row < boardVSize; ++row) {
00161             board->setRowHeight(row, m_cellSize);
00162             for(unsigned int col = 0; col < boardHSize; ++col) {
00163                 board->setItem(row, col, new QTableWidgetItem(""));
00164                 board->item(row, col)->setBackgroundColor("white");
00165                 board->item(row, col)->setTextColor("black");
00166             }
00167         }
00168      QScrollArea *scrollArea = new QScrollArea(this);
00169      scrollArea->setWidget(board);
00170      layout->setContentsMargins(0,0,0,0);
00171      layout->addWidget(scrollArea);
00172      tab->setLayout(layout);
00173      return tab;
00174 }
00175
00179 void MainWindow::openFile(){
00180     QString fileName = QFileDialog::getOpenFileName(this, tr("Open Cell file"), ".",
00181                                                     tr("Automaton cell files (*.atc)"));
00182     if(!fileName.isEmpty()){
00183         AutomateHandler::getAutomateHandler().
      addAutomate(new Automate(fileName));
00184         if(m_tabs == NULL) createTabs();
00185         m_tabs->addTab(createTab(), "Automaton "+ QString::number(
      AutomateHandler::getAutomateHandler().getNumberAutomates()+1));
00186         updateBoard(AutomateHandler::getAutomateHandler().
      getNumberAutomates()-1);
00187
00188         RuleEditor* ruleEditor = new RuleEditor();
00189         connect(ruleEditor, SIGNAL(fileImported(QString)),this,SLOT(
      addAutomatonRuleFile(QString)));
00190         connect(ruleEditor, SIGNAL(rulesFilled(QList<const NeighbourRule*>)), this, SLOT(
      addAutomatonRules(QList<const NeighbourRule*>)));
00191         ruleEditor->show();
00192     }
00193 }
00194
00195
00199 void MainWindow::saveToFile(){
00200     if(AutomateHandler::getAutomateHandler().getNumberAutomates() > 0){
00201         QString fileName = QFileDialog::getSaveFileName(this, tr("Save Automaton cell configuration"),
00202                                                         ".", tr("Automaton Cells file (*.atc"));
00203         AutomateHandler::getAutomateHandler().
```

```
    getAutomate(m_tabs->currentIndex())->saveCells(fileName+".atc");
00204
00205     }
00206     else{
00207         QMessageBox msgBox;
00208         msgBox.critical(0,"Error","Please create or import an Automaton first !");
00209         msgBox.setFixedSize(500,200);
00210     }
00211 }
00212
00217 void MainWindow::openCreationWindow(){
00218     CreationDialog *window = new CreationDialog(this);
00219     connect(window, SIGNAL(settingsFilled(QVector<uint>,
    CellHandler::generationTypes,uint,uint)),
00220             this, SLOT(receiveCellHandler(QVector<uint>,
    CellHandler::generationTypes,uint,uint)));
00221     window->show();
00222 }
00223
00230 void MainWindow::receiveCellHandler(const QVector<unsigned int> dimensions,
00231                                     CellHandler::generationTypes type,
00232                                     unsigned int stateMax, unsigned int density){
00233     AutomateHandler::getAutomateHandler().
    addAutomate(new Automate(dimensions, type, stateMax, density));
00234
00235     if(m_tabs == NULL) createTabs();
00236     QWidget* newTab = createTab();
00237     m_tabs->addTab(newTab, "Automaton "+ QString::number(
    AutomateHandler::getAutomateHandler().getNumberAutomates()));
00238     m_tabs->setCurrentWidget(newTab);
00239     updateBoard(AutomateHandler::getAutomateHandler().
    getNumberAutomates()-1);
00240
00241     RuleEditor* ruleEditor = new RuleEditor();
00242     connect(ruleEditor, SIGNAL(fileImported(QString)),this,SLOT(
    addAutomatonRuleFile(QString)));
00243     connect(ruleEditor, SIGNAL(rulesFilled(QList<const Rule*>)), this, SLOT(
    addAutomatonRules(QList<const Rule*>)));
00244     ruleEditor->show();
00245
00246 }
00247
00252 void MainWindow::nextState(unsigned int n){
00253     if(AutomateHandler::getAutomateHandler().getNumberAutomates()== 0){
00254         QMessageBox msgBox;
00255         msgBox.critical(0,"Error","Please create or import an Automaton first !");
00256         msgBox.setFixedSize(500,200);
00257     }
00258     else{
00259
00260         AutomateHandler::getAutomateHandler().
    getAutomate(m_tabs->currentIndex())->run(n);
00261         updateBoard(m_tabs->currentIndex());
00262     }
00263 }
00264
00269 void MainWindow::updateBoard(int index){
00270     if(AutomateHandler::getAutomateHandler().getNumberAutomates()== 0){
00271         QMessageBox msgBox;
00272         msgBox.critical(0,"Error","Please create or import an Automaton first !");
00273         msgBox.setFixedSize(500,200);
00274     }
00275     else{
00276
00277         const CellHandler* cellHandler = &(
    AutomateHandler::getAutomateHandler().
    getAutomate(index)->getCellHandler());
00278         QVector<unsigned int> dimensions = cellHandler->getDimensions();
00279         QTableWidget* board = getBoard(index);
00280         if(dimensions.size() > 1){
00281             int i = 0;
00282             int j = 0;
00283             for (CellHandler::const_iterator it = cellHandler->
    begin(); it != cellHandler->end() && it.changedDimension() < 2; ++it){
00284                     if(it.changedDimension() > 0){
00285                         i = 0;
00286                         j++;
00287                     }
00288                     board->item(i,j)->setBackgroundColor(QColor::colorNames().at(it->getState()));
00289                     i++;
00290             }
00291         }
00292         else{ // dimension = 1
00293             if (board->rowCount() != 1)
00294                 addEmptyRow(index);
00295             int i = board->rowCount() -1;
00296             int j = 0;
```

```
00297             for (CellHandler::const_iterator it = cellHandler->
      begin(); it != cellHandler->end() && it.changedDimension() < 1; ++it){
00298                     board->item(i,j)->setBackgroundColor(QColor::colorNames().at(it->getState()));
00299                     j++;
00300             }
00301             if (board->rowCount() == 1)
00302                 addEmptyRow(index);
00303
00304         }
00305
00306     }
00307
00308 }
00309
00314 void MainWindow::forward(){
00315     //nextState(m_timeStep->value());
00316     nextState(1);
00317 }
00318
00319 QTableWidget* MainWindow::getBoard(int n){
00320     return m_tabs->widget(n)->findChild<QTableWidget *>();
00321 }
00322
00327 void MainWindow::createTabs(){
00328     m_tabs = new QTabWidget(this);
00329     m_tabs->setMovable(true);
00330     m_tabs->setTabsClosable(true);
00331     setCentralWidget(m_tabs);
00332     connect(m_tabs, SIGNAL(tabCloseRequested(int)), this, SLOT(closeTab(int)));
00333 }
00334
00341 void MainWindow::addEmptyRow(unsigned int n)
00342 {
00343     QTableWidget *board = getBoard(n);
00344     board->setFixedHeight(board->height() + m_cellSize);
00345     unsigned int row = board->rowCount();
00346     board->insertRow(row);
00347     board->setRowHeight(row, m_cellSize);
00348     for(unsigned int col = 0; col < board->columnCount(); ++col) {
00349         board->setItem(row, col, new QTableWidgetItem(""));
00350         board->item(row, col)->setBackgroundColor("white");
00351         board->item(row, col)->setTextColor("black");
00352     }
00353 }
00354
00359 void MainWindow::closeTab(int n){
00360     m_tabs->setCurrentIndex(n);
00361     saveToFile();
00362     AutomateHandler::getAutomateHandler().
      deleteAutomate(AutomateHandler::getAutomateHandler().
      getAutomate(n));
00363     m_tabs->removeTab(n);
00364 }
00365
00366 void MainWindow::addAutomatonRules(QList<const Rule *> rules){
00367     for(int i =0 ; i < rules.size();i++)
00368     {
00369         AutomateHandler::getAutomateHandler().
      getAutomate(AutomateHandler::getAutomateHandler().
      getNumberAutomates()-1)->addRule(rules.at(i));
00370     }
00371 }
00372
00373 void MainWindow::addAutomatonRuleFile(QString path){
00374     AutomateHandler::getAutomateHandler().
      getAutomate(AutomateHandler::getAutomateHandler().
      getNumberAutomates()-1)->addRuleFile(path);
00375 }
00376
00377 void MainWindow::handlePlayPause(){
00378     if(AutomateHandler::getAutomateHandler().getNumberAutomates()== 0){
00379         QMessageBox msgBox;
00380         msgBox.critical(0,"Error","Please create or import an Automaton first !");
00381         msgBox.setFixedSize(500,200);
00382     }
00383     else{
00384         if(running){
00385             m_playPauseBt->setIcon(m_playIcon);
00386             delete m_timer;
00387         }
00388         else {
00389             m_playPauseBt->setIcon(m_pauseIcon);
00390             m_timer = new QTimer(this);
00391             connect(m_timer, SIGNAL(timeout()), this, SLOT(runAutomaton()));
00392             m_timer->start(m_timeStep->value());
00393         }
00394         running = !running;
```

```
00395     }
00396
00397
00398 }
00399
00400 void MainWindow::runAutomaton(){
00401     if(running){
00402         AutomateHandler::getAutomateHandler().
      getAutomate(m_tabs->currentIndex())->run();
00403         QCoreApplication::processEvents();
00404         updateBoard(m_tabs->currentIndex());
00405         QCoreApplication::processEvents();
00406     }
00407 }
00408
00409 void MainWindow::reset(){
00410     if(AutomateHandler::getAutomateHandler().getNumberAutomates()== 0){
00411         QMessageBox msgBox;
00412         msgBox.critical(0,"Error","Please create or import an Automaton first !");
00413         msgBox.setFixedSize(500,200);
00414     }
00415     else{
00416         QTableWidget *board = getBoard(m_tabs->currentIndex());
00417         board->setRowCount(1);
00418         board->setFixedHeight(m_cellSize);
00419
00420         AutomateHandler::getAutomateHandler().
      getAutomate(m_tabs->currentIndex())->getCellHandler().reset();
00421         updateBoard(m_tabs->currentIndex());
00422     }
00423 }
```

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     QTimer* m_timer;
00060
00061     Automate* m_newAutomate;
00062     bool running;
00063     QToolBar *m_toolBar;
00064
00066     unsigned int m_boardHSize = 25;
00067     unsigned int m_boardVSize = 25;
00068     unsigned int m_cellSize = 30;
00069
00070     void createIcons();
00071     void createActions();
00072     void createToolBar();
00073     void createBoard();
00074     QWidget* createTab();
00075     void createTabs();
00076
00077     void addEmptyRow(unsigned int n);
00078     void updateBoard(int index);
00079     void nextState(unsigned int n);
00080     QTableWidget* getBoard(int n);
00081
00082
00083 public:
00084     explicit MainWindow(QWidget *parent = nullptr);
00085
00086
00087 signals:
00088
00089 public slots:
00090     void openFile();
00091     void saveToFile();
00092     void openCreationWindow();
00093     void receiveCellHandler(const QVector<unsigned int> dimensions,
00094                             CellHandler::generationTypes type =
      CellHandler::generationTypes::empty,
00095                             unsigned int stateMax = 1, unsigned int density = 20);
00096     void addAutomatonRules(QList<const Rule *> rules);
00097     void addAutomatonRuleFile(QString path);
00098     void forward();
00099     void closeTab(int n);
00100     void runAutomaton();
00101     void handlePlayPause();
00102     void reset();
00103
00104 };
00105
00106 #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             matched = false;
00046             break;
00047         }
00048         if (! it.value().contains(cell->getNeighbour(it.key())->getState()))
00049         {
00050             matched = false;
00051             break;
00052         }
00053     }
00054
00055     return matched;
00056 }
00057
00060 void MatrixRule::addNeighbourState(QVector<short> relativePosition, unsigned
      int matchState)
00061 {
00062     m_matrix[relativePosition].push_back(matchState);
00063 }
00064
00067 void MatrixRule::addNeighbourState(QVector<short> relativePosition,
      QVector<unsigned int> matchStates)
00068 {
00069     for (QVector<unsigned int>::const_iterator it = matchStates.begin(); it != matchStates.end(); ++it)
00070         m_matrix[relativePosition].push_back(*it);
00071 }
00072
00075 QJsonObject MatrixRule::toJson() const
00076 {
00077     QJsonObject object(Rule::toJson());
00078
00079     object.insert("type", QJsonValue("matrix"));
00080
00081     QJsonArray neighbours;
00082     for (QMap<QVector<short>,  QVector<unsigned int> >::const_iterator it =
      m_matrix.begin(); it != m_matrix.end(); ++it)
00083     {
00084         QJsonObject aNeighbour;
00085         QJsonArray relativePosition;
00086         for (unsigned int i = 0; i < it.key().size(); i++)
00087         {
00088             relativePosition.append(QJsonValue((int)it.key().at(i)));
00089         }
00090         aNeighbour.insert("relativePosition", relativePosition);
00091
00092         QJsonArray neighbourStates;
00093         for (unsigned int i = 0; i < it.value().size(); i++)
00094         {
00095             neighbourStates.append(QJsonValue((int)it.value().at(i)));
00096         }
00097         aNeighbour.insert("neighbourStates", neighbourStates);
00098
00099         neighbours.append(aNeighbour);
00100     }
00101     object.insert("neighbours", neighbours);
00102
00103     return object;
00104 }
```

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         void addNeighbourState(QVector<short> relativePosition, unsigned int matchState);
00021         void addNeighbourState(QVector<short> relativePosition, QVector<unsigned int>
      matchStates);
00022
00023         QJsonObject toJson() const;
00024
00025
00026 private:
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 private:
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     virtual 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
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(QWidget *parent)
00004 {
00005     selectedRule = -1;
00006     QHBoxLayout *hlayout = new QHBoxLayout();
00007     m_rulesListWidget = new QListWidget(this);
00008     QLabel *rulesLabel = new QLabel("Rules ",this);
00009     QVBoxLayout *rulesListLayout = new QVBoxLayout();
00010     rulesListLayout->addWidget(rulesLabel);
00011     rulesListLayout->addWidget(m_rulesListWidget);
00012     hlayout->addLayout(rulesListLayout);
00013
00014     QGridLayout *rulesInputLayout = new QGridLayout();
```

```
00015
00016     rulesInputLayout->addWidget(new QLabel("Current cell values :",this),0,0);
00017     m_currentStatesEdit = new QLineEdit(this);
00018     QRegExp rgx("([0-9]+,)*");
00019     QRegExpValidator *v = new QRegExpValidator(rgx, this);
00020     m_currentStatesEdit->setValidator(v);
00021     rulesInputLayout->addWidget(m_currentStatesEdit,0,1);
00022
00023     rulesInputLayout->addWidget(new QLabel("Neighbour number lower bound :",this),1,0);
00024     m_lowerNeighbourBox = new QSpinBox(this);
00025     rulesInputLayout->addWidget(m_lowerNeighbourBox,1,1);
00026
00027     rulesInputLayout->addWidget(new QLabel("Neighbour number upper bound :",this),2,0);
00028     m_upperNeighbourBox = new QSpinBox(this);
00029     rulesInputLayout->addWidget(m_upperNeighbourBox,2,1);
00030
00031     rulesInputLayout->addWidget(new QLabel("Neighbour values :",this),3,0);
00032     m_neighbourStatesEdit = new QLineEdit(this);
00033     m_neighbourStatesEdit->setValidator(v);
00034     rulesInputLayout->addWidget(m_neighbourStatesEdit,3,1);
00035
00036     rulesInputLayout->addWidget(new QLabel("Output state :",this),4,0);
00037     m_outputStateBox = new QSpinBox(this);
00038     rulesInputLayout->addWidget(m_outputStateBox,4,1);
00039
00040     hlayout->addLayout(rulesInputLayout);
00041     QVBoxLayout* mainLayout = new QVBoxLayout();
00042     QHBoxLayout* buttonLayout = new QHBoxLayout();
00043
00044     m_addBt = new QPushButton("Add Rule",this);
00045     m_importBt = new QPushButton("Import Rule file",this);
00046     m_doneBt = new QPushButton("Done !",this);
00047     m_removeBt = new QPushButton("Remove Rule",this);
00048     buttonLayout->addWidget(m_importBt);
00049     buttonLayout->addWidget(m_addBt);
00050     buttonLayout->addWidget(m_removeBt);
00051     buttonLayout->addWidget(m_doneBt);
00052
00053     mainLayout->addLayout(hlayout);
00054     mainLayout->addLayout(buttonLayout);
00055     setLayout(mainLayout);
00056
00057     connect(m_addBt, SIGNAL(clicked(bool)), this, SLOT(addRule()));
00058     connect(m_importBt, SIGNAL(clicked(bool)), this, SLOT(importFile()));
00059     connect(m_doneBt, SIGNAL(clicked(bool)), this, SLOT(sendRules()));
00060     connect(m_removeBt, SIGNAL(clicked(bool)), this, SLOT(removeRule()));
00061
00062 }
00063
00064
00065
00066 void RuleEditor::addRule(){
00067     unsigned int outputState = m_outputStateBox->value();
00068     QVector<unsigned int> currentCellValues;
00069     QStringList valList = m_currentStatesEdit->text().split(",");
00070     for(int i = 0; i < valList.size(); i++) currentCellValues.append(valList.at(i).toInt());
00071
00072     QPair<unsigned int, unsigned int> neighbourInterval;
00073     neighbourInterval.first = m_lowerNeighbourBox->value();
00074     neighbourInterval.second = m_upperNeighbourBox->value();
00075
00076     QSet<unsigned int> neighbourValues;
00077     valList = m_neighbourStatesEdit->text().split(",");
00078     for(int i = 0; i < valList.size(); i++) neighbourValues << valList.at(i).toInt();
00079
00080     m_rules.append(new NeighbourRule(outputState,currentCellValues,neighbourInterval,
      neighbourValues));
00081
00082     QString listLabel = m_currentStatesEdit->text()+" -> "+QString::number(
      m_outputStateBox->value())
00083                         +" if "+QString::number(m_lowerNeighbourBox->value())+" to "+
00084                         QString::number(m_upperNeighbourBox->value())+" neighbours are
       in states "+
00085                         m_neighbourStatesEdit->text();
00086     m_rulesListWidget->addItem(listLabel);
00087 }
00088
00089 void RuleEditor::removeRule(){
00090     m_rules.removeAt(m_rulesListWidget->currentRow());
00091     delete m_rulesListWidget->takeItem(m_rulesListWidget->currentRow());
00092 }
00093
00094 void RuleEditor::sendRules(){
00095     emit rulesFilled(m_rules);
00096     this->close();
00097 }
00098
```

```
00099 void RuleEditor::importFile(){
00100     QString fileName = QFileDialog::getOpenFileName(this, tr("Open Rule file"), ".",
00101                                                     tr("Automaton rule files (*.atr)"));
00102     if(!fileName.isEmpty()){
00103         emit fileImported(fileName);
00104         this->close();
00105     }
00106 }
```

7.45 ruleeditor.h File Reference

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

Classes

- class RuleEditor

7.46 ruleeditor.h

```
00001 #ifndef RULEEDITOR_H
00002 #define RULEEDITOR_H
00003 #include <QtWidgets>
00004 #include "neighbourrule.h"
00005
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
00020     QPushButton* m_addBt;
00021     QPushButton* m_doneBt;
00022     QPushButton* m_removeBt;
00023     QPushButton* m_importBt;
00024
00025     unsigned int selectedRule;
00026
00027
00028 public:
00029     explicit RuleEditor(QWidget *parent = nullptr);
00030
00031 signals:
00032     void rulesFilled(QList<const Rule*> rules);
00033     void fileImported(QString path);
00034
00035 public slots:
00036     void removeRule();
00037     void addRule();
00038     void importFile();
00039     void sendRules();
00040
00041
00042 };
00043
00044 #endif // RULEEDITOR_H
```

Index