

Rapport LO21

Maxime Noizet, Philippe Carvalho, Claire Guyot

16 juin 2018

Table des matières

I	Contexte de l'application	2
I.1	Description des automates cellulaires	2
I.2	Les automates dans l'application	3
I.2.1	Automates cellulaires élémentaires 'revisités'	3
I.2.2	Jeu de la vie	4
I.2.3	Feu de forêt	5
I.3	Les voisinages dans l'application	6
I.3.1	Voisinage pour automates cellulaires à 1 dimension	6
I.3.2	Voisinages pour automates cellulaires à 2 dimensions	6
II	Architecture de l'application	7
II.1	Back-end de l'application	7
II.1.1	Simulation d'automates cellulaires	7
II.1.2	Construction des simulateurs	9
II.1.3	Sauvegarde et chargement	10
II.2	Front-end de l'application	12
III	Evolutivité de l'architecture	14
III.1	Adaptabilité de l'architecture	14
III.2	Ajout d'un nouvel automate	16
IV	Annexes	18
IV.1	Code source	18
IV.2	Documentation	19
IV.3	Vidéo	19
IV.4	Fichiers	20

I - Contexte de l'application

I.1 Description des automates cellulaires

Un automate cellulaire est, selon la définition donnée par Wikipédia, *une grille régulière de « cellules » contenant chacune un « état » choisi parmi un ensemble fini et qui peut évoluer au cours du temps. L'état d'une cellule au temps $t+1$ est fonction de l'état au temps t d'un nombre fini de cellules appelé son « voisinage ». À chaque nouvelle unité de temps, les mêmes règles sont appliquées simultanément à toutes les cellules de la grille, produisant une nouvelle « génération » de cellules dépendant entièrement de la génération précédente.*

Un automate cellulaire est donc caractérisé par:

- un ensemble fini d'états que peuvent prendre les cellules
- une règle de transition pour calculer les états des cellules à chaque nouvelle génération
- un voisinage, ensemble des voisines à prendre en compte pour calculer l'état suivant d'une cellule. Cependant, la règle de transition d'un automate n'est pas nécessairement basé sur un automate contrairement à la définition générale donnée par Wikipédia. Par exemple, l'automate cellulaire nommé Fourmi de Langton, n'a pas besoin d'un voisinage car la transition d'une cellule dépend de sa position par rapport à la fourmi.
- une dimension de grille

Les automates cellulaires peuvent être différenciés selon leur dimension. Les automates cellulaires 1D n'évolueront que sur une seule rangée de cellules, tandis que les automates cellulaires 2D évoluera sur une grille 2D de cellules. Il existe des automates cellulaires de dimension supérieure, bien qu'il soit plus difficile de les représenter.

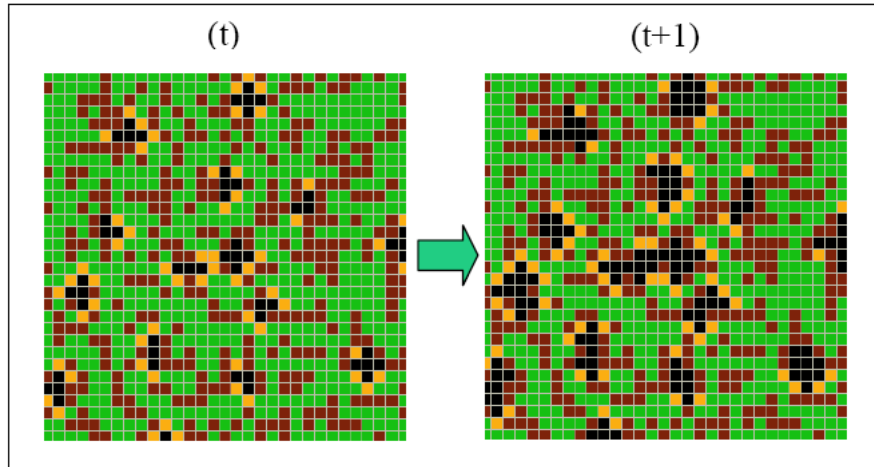


FIG. I.1 – *Automate cellulaire passant d'un instant t à un instant $t+1$*

I.2 Les automates dans l'application

Dans le cadre de notre application, il est possible de simuler 3 types d'automates:

- les automates cellulaires élémentaires
- le jeu de la vie
- le feu de forêt

I.2.1 Automates cellulaires élémentaires 'revisités'

Les automates cellulaires élémentaires sont des automates cellulaires à une dimension dans lesquels chaque cellule peut prendre 2 états. En effet chaque cellule peut être soit noire, soit blanche.

Pour passer de l'instant t à l'instant $t+1$, la règle de transition se base sur l'état de la cellule et de ses 2 voisines afin de calculer l'état de la cellule à l'instant $t+1$. Cela nous donne une règle de la forme suivante:

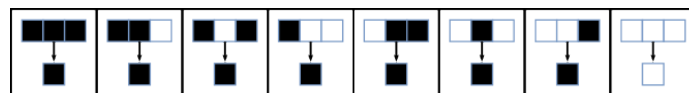


FIG. I.2 – *Règle de transition d'un automate cellulaire*

Ces automates cellulaires n'ayant que 2 états possibles et chaque cellule n'ayant que 2 voisins, la règle de transition dispose de 8 possibilités. Elle est

généralement codée par une chaîne de caractères composée de 8 caractères 0 ou 1. Par exemple, si une case noire est représentée par un 1 et une case blanche par un 0, la règle ci-dessus nous donne le code : 11111110.

Les automates cellulaires élémentaires ont donc 256 règles possibles et il suffit de convertir le code en base 10 pour trouver le numéro de la règle.

Cependant, dans le cadre de notre application nous pouvons créer des automates élémentaires avec un nombre quelconque d'états possibles pour les cellules et un nombre quelconque de voisins d'où le nom d'automates cellulaires élémentaires 'revisités'.

Ainsi la règle de transition doit respecter les contraintes suivantes:

$$c_i \in [0, \text{nombre d'états} - 1]$$

$$\text{taille de la règle} = \text{nombre d'états}^{\text{nombre de voisins}}$$

Dans le voisinage, on considère que la cellule elle-même est au centre de son voisinage et en fait partie.

I.2.2 Jeu de la vie

Le jeu de la vie est un automate cellulaire à deux dimensions dans lequel chaque cellule peut prendre 2 états. En effet une cellule peut être soit vivante, soit morte. Lorsque la cellule est vivante, la case est noire et lorsqu'elle est morte, la case est blanche.

Pour définir les transitions entre chaque génération, la règle de transition prend deux paramètres:

- le nombre minimum de voisins vivants au temps t.
- le nombre maximum de voisins vivants au temps t.

La transition entre l'instant t et l'instant t+1 s'effectue ainsi:

- si la cellule est vivante à l'instant t et si le nombre de voisins vivants est entre le nombre minimum et le nombre maximum alors la cellule est vivante à l'instant t+1 sinon elle meurt à l'instant t+1 (d'isolement ou d'étouffement);
- si la cellule est morte à l'instant t et que le nombre de voisins vivants est égal au nombre maximum alors la cellule est vivante à l'instant t+1 sinon elle reste morte.

Ci-après un exemple de jeu de la vie avec un nombre minimum de voisins égal à 1 et un nombre maximum égal à 3 (à noter que cette configuration est la configuration la plus fréquemment utilisée pour un jeu de la vie).

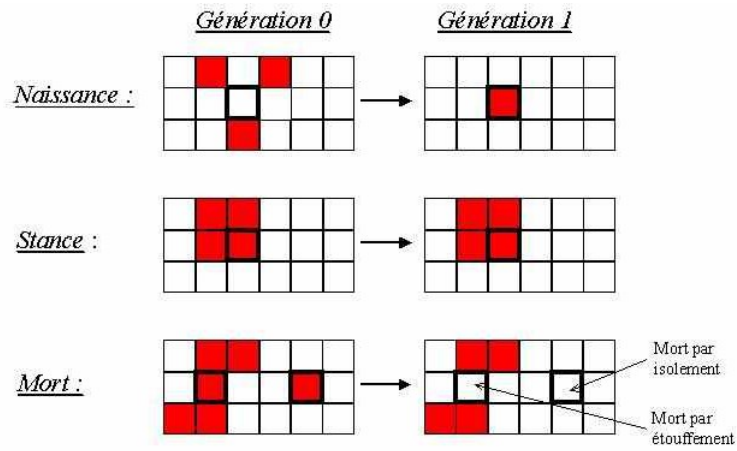


FIG. I.3 – *Exemple de jeu de la vie*

I.2.3 Feu de forêt

Le feu de forêt est un automate cellulaire à 2 dimensions qui, comme son nom l'indique, permet d'effectuer la simulation du déroulement d'un feu de forêt.

Chaque cellule peut prendre un des états suivants : Arbre, Vide, Feu ou Cendres. Le nom des états est explicite.

La règle de transition est quant à elle assez intuitive:

- une cellule vide à l'instant t reste vide à l'instant $t+1$
- une cellule cendres à l'instant t reste cendres à l'instant $t+1$
- une cellule feu à l'instant t devient cendres à l'instant $t+1$
- une cellule arbre à l'instant t :
 - devient feu à l'instant $t+1$ si l'un de ses voisins est en feu.
 - reste arbre à l'instant $t+1$ sinon.

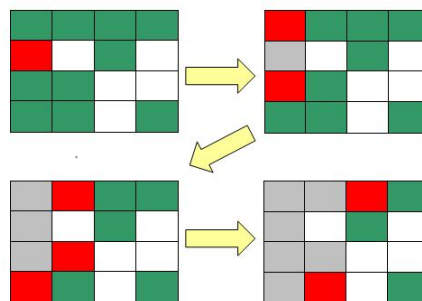


FIG. I.4 – *Exemple de feu de forêt*

I.3 Les voisinages dans l'application

Nous avons pu constater dans la partie précédente que les divers automates cellulaires implémentés dans notre application nécessitent une définition de voisinage pour les différentes règles de transition.

I.3.1 Voisinage pour automates cellulaires à 1 dimension

Ce type de voisinage est un voisinage très simple centré sur la cellule. On peut donc choisir l'ordre de ce voisinage, distance maximale possible entre la cellule et un voisin.

I.3.2 Voisinages pour automates cellulaires à 2 dimensions

Dans le cadre de notre application, nous prenons en compte deux types de voisinages différents :

- Von Neumann : croix centrée sur de la cellule
- Moore : carré centré sur de la cellule

Comme pour le voisinage des automates cellulaires à 1 dimension, il est possible de choisir l'ordre du voisinage représenté par le rayon sur la figure ci-dessous :

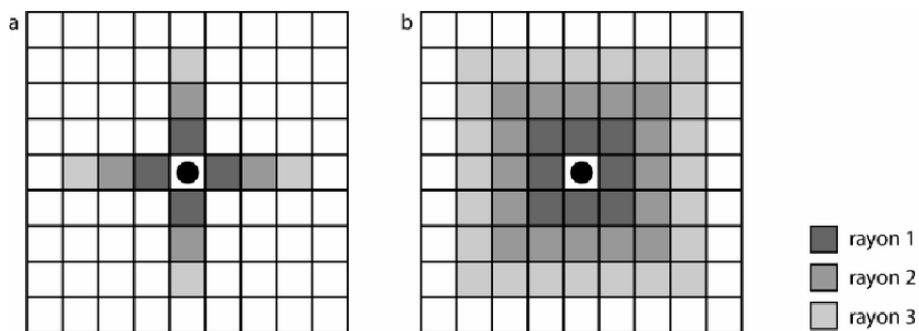


FIG. I.5 – Voisinage a) Von Neumann b) Moore

II - Architecture de l'application

II.1 Back-end de l'application

II.1.1 Simulation d'automates cellulaires

Comme expliqué précédemment, un automate cellulaire est défini par une dimension, une règle de transition, un voisinage (mais cela n'est pas obligatoire) et bien évidemment un état de départ. Nous avons donc dû effectuer plusieurs choix d'implémentation pour respecter ces contraintes.

Nous avons tout d'abord décidé de créer une classe **CellularAutomata** qui nous permet d'avoir une façade sur le back-end plus simple à utiliser. Cette classe permet ainsi de faire le lien entre la règle de transition, le voisinage, et les grilles de cellules qui sont des objets de la classe **Etat**.

Nous avons décidé de séparer la règle de transition du voisinage, car ainsi il est possible d'utiliser une même règle de transition pour différentes définitions de voisinage, et inversement. De plus nous avons choisi de stocker le voisinage de chaque cellule au niveau des cellules.

Ce choix augmente donc bien évidemment la complexité en mémoire mais il se justifie de la manière suivante: nous aurions pu choisir de passer le voisinage en paramètre de la méthode de **TransitionRule** permettant d'effectuer une transition de l'automate, cependant, certaines règles de transitions ne nécessite pas de définition de voisinage (comme par exemple la fourmi de Langton citée précédemment), il semble donc contre intuitif de forcer l'utilisateur à passer en argument une définition de voisinage qui n'a pas lieu d'exister. Ainsi, l'option qui nous restait était de stocker le voisinage de chaque cellule au niveau des cellules. De plus, cela semble logique puisqu'on ne peut pas vraiment dissocier une cellule de son voisinage.

Nous avons donc créé une classe **Cell** étant donné que l'on ne pouvait se contenter d'un tableau d'int dans la classe **Etat**.

Concernant la classe **Etat**, classe représentant donc nos grilles de cellules, nous avons également effectué divers choix d'implémentation. Tout d'abord, il est important de noter que notre classe **Etat** agrège une classe **GenerateurEtat**. Cette classe **GenerateurEtat** est en réalité une implémentation du design pattern *strategy* et nous permet donc de choisir entre divers

algorithmes de génération d'Etats, c'est-à-dire de génération des valeurs (valeurs représentant les états pris par chaque cellule) des cellules d'une grille.

La grille que représente la classe Etat est de type Cell**: il s'agit d'un tableau à deux dimensions alloué dynamiquement. Ainsi il est possible de gérer des grilles à deux dimensions ainsi que des grilles à une dimensions qui sont des cas particuliers de grille à deux dimensions. Cependant, même si cela n'était pas demandé, notre architecture ne peut malheureusement pas prendre en compte des automates à plus de deux dimensions.

Nous avons également pris la décision de faire en sorte que seul un objet de la classe Etat puisse construire des objets de la classe Cell et modifier les positions de ces objets. Ce choix est logique car une cellule ne peut exister sans la grille dont elle est un composé.

Nous avons également décidé de fournir deux `iterator` afin de fournir une méthode standard (implémentée de la même manière que la STL) afin de pouvoir parcourir séquentiellement les cases d'une grille.

La plus grande difficulté inhérente à nos choix d'implémentation est de gérer la création des sous-parties nécessaires à la création d'un automate cellulaire ainsi que leur combinaison. En effet, il n'est pas possible de combiner n'importe quel voisinage avec n'importe quelle règle de transition ou n'importe quel état. L'automate à une dimension est un cas particulier d'un automate à deux dimensions, on peut donc lui appliquer des règles de transitions pour les automates à deux dimensions, par contre on ne peut pas utiliser un voisinage à deux dimensions à cause de l'implémentation de la règle de transition élémentaire pour les automates 1D.

De plus, il est bien évidemment impossible d'appliquer une règle pour automate 1D à un automate 2D sauf si on utilise un voisinage 1D mais dans ce cas là avec notre implémentation actuelle seule la première ligne de la grille sera prise en compte.

Nous avons donc dû créer des classes permettant de gérer ces différentes sous-parties.

Vous pouvez trouver ci-après le diagramme de classes simplifié de cette partie du back-end. Le diagramme complet vous est fourni dans le dossier UML joint avec les rendus.

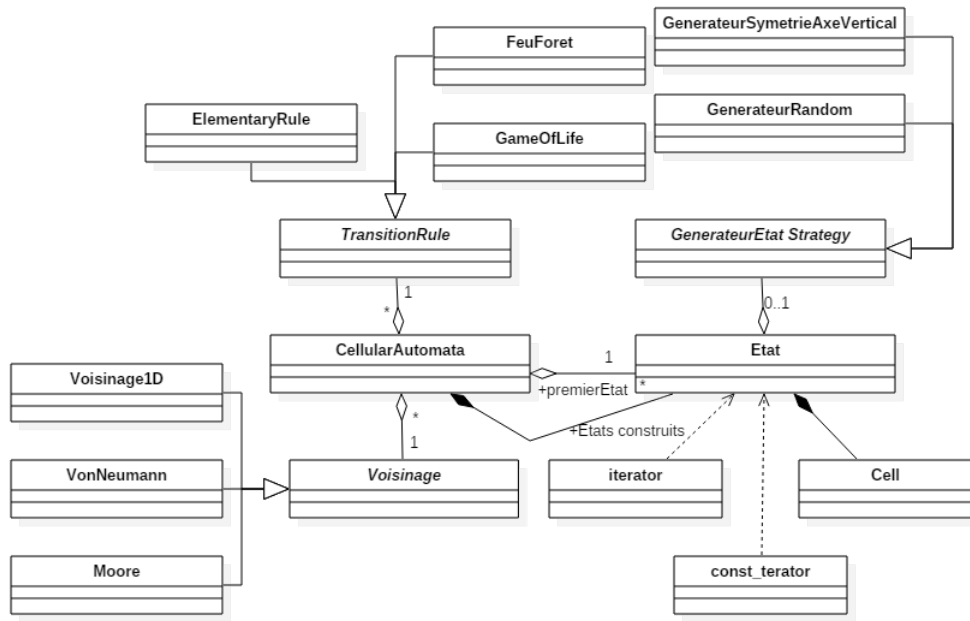


FIG. II.1 – UML simplifié de la partie simulateur du back-end

II.1.2 Construction des simulateurs

Afin de pouvoir construire convenablement les sous-parties d'un automate cellulaire, sans avoir de fuite de mémoire, nous avons décidé de créer deux classes CABuilder1D et CABuilder2D.

Comme leur nom l'indique, CABuilder1D permet de construire et de gérer toutes les sous-parties nécessaires à la construction d'un automate cellulaire à une dimension et CABuilder2D permet de construire et de gérer toutes les sous-parties nécessaires à la construction d'un automate cellulaire à deux dimensions.

Ces deux classes sont des singletons afin de faciliter leur accès à n'importe quel emplacement dans le code.

Ces deux classes héritent d'une classe CABuilder, non instanciable. Cette classe permet de généraliser les parties communes à CABuilder1D et CABuilder2D, c'est-à-dire les attributs, les accesseurs sur les sous-parties des automates cellulaires, ainsi que les méthodes permettant de construire les différents générateurs d'états qui peuvent être utilisés pour générer les objets de la classe Etat.

Cependant, notre implémentation actuelle ne permet que de construire un automate cellulaire 1D et un automate cellulaire 2D simultanément. Pour améliorer notre implémentation, nous pourrions utiliser le conteneur 'map' de la STL du C++ afin de pouvoir associer une clé de type string à chaque sous-partie créée pour un simulateur. Ainsi chaque sous-partie créée sera accessible par son nom et il sera possible de la réutiliser facilement pour un autre automate cellulaire. De plus, il ne sera plus nécessaire de supprimer les sous-parties déjà existantes lorsqu'on voudra en recréer une nouvelle, contrairement à notre implémentation actuelle. De plus, associée à un menu déroulant sur l'interface, il serait facile à l'utilisateur de sélectionner une sous-partie déjà créée afin de la réutiliser.

Vous pouvez voir sur la figure II.2 ci-après l'UML simplifié des classes CABuilder. Le diagramme complet vous est fourni dans le dossier UML joint avec les rendus.

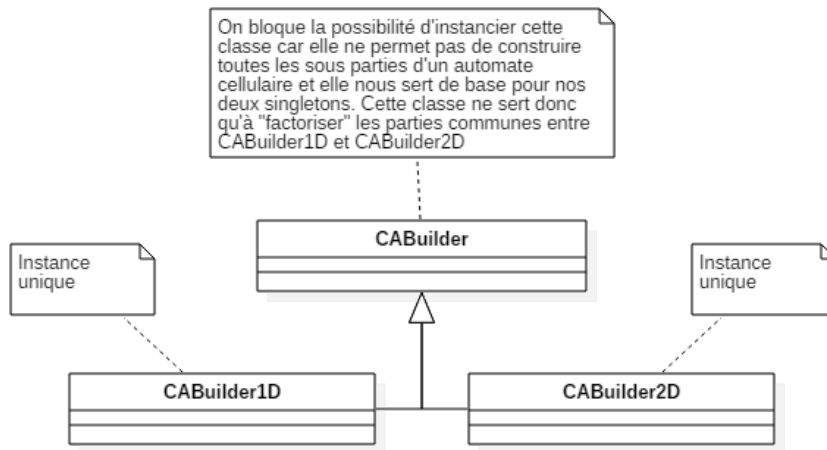


FIG. II.2 – UML simplifié des CABuilder, classes permettant la construction des sous parties du simulateur

II.1.3 Sauvegarde et chargement

La gestion de la sauvegarde et du chargement est gérée dans deux fichiers, `fichier.cpp` et `sauvegarde.cpp` (et leurs headers).

Dans `sauvegarde.cpp`, on définit la fonction `sauvegarde()`, dont les paramètres sont une référence vers l'automate à sauvegarder, la dimension de l'automate envoyé et le type de sauvegarde/chargement voulu, et la fonction `chargement()`, qui prend les mêmes paramètres sauf une référence sur

pointeur pour l'automate à charger.

En effet, on différencie deux types d'enregistrements : l'enregistrement d'un état, qui s'effectue dans le format `.bn` que nous avons conçu, et l'enregistrement d'une configuration qui s'effectue en `.csv`. Un fichier `.bn` se présente comme une succession de chiffres, représentant les états des cases de l'automate, séparés par des virgules entre chaque case et des points-virgules pour représenter les différentes lignes. Un fichier `.csv` se présente comme une suite d'instructions séparées par des virgules, qui sont placées dans un certain ordre afin d'enregistrer tous les attributs de la configuration. Les fichiers (en particulier les configurations) n'ayant pas les mêmes propriétés entre 1D et 2D, il est également nécessaire de différencier la dimension dans ces fonctions.

Selon les paramètres, les fonctions initialisent un objet `fichier` en utilisant le constructeur de l'une de ses classes filles (voir la figure II.3). La fonction gère également la sélection du nom du fichier à sauvegarder/charger. Si on veut sauvegarder/charger un état, c'est `fichierEtat1D` ou `fichierEtat2D` qui est appelé selon le type d'automate. Si l'on veut sauvegarder/charger la configuration d'un automate, c'est dans ce cas `fichierConfig1D` ou `fichierConfig2D`.

Nous avons décidé d'implémenter le polymorphisme ici, car il suffit ensuite d'appeler la méthode `save()` ou `load()` et de détruire l'objet. Ces méthodes s'occupent d'ouvrir le fichier dont le nom a été donné dans le constructeur, de sauvegarder ou de charger dans/à partir de l'automate, puis de refermer le fichier dans le destructeur de l'objet.

La sauvegarde d'un état 1D et 2D est relativement simple : il suffit de parcourir l'ensemble des cases du tableau et d'enregistrer leur état. Si on arrive au bout d'une ligne, on enregistre un point-virgule et sinon une virgule.

Les configurations 1D et 2D sont enregistrées de la même manière : on enregistre d'abord les voisinages en faisant appel aux méthodes `getType()` et `getOrdre()`. On enregistre ensuite la transition avec `getTransition()`. Ces méthodes renvoient toutes les informations demandées sous forme d'un string.

Le chargement est une tâche plus complexe, car elle nécessite, d'une part, de pouvoir faire face aux erreurs dans le fichier (corruption), et d'autre part de modifier correctement le simulateur.

Le chargement d'un état lit un fichier caractère par caractère et vérifie l'alternance des nombres et des symboles. Le chargement d'état 1D s'arrête à la lecture d'un point virgule, le chargement 2D continue jusqu'à la fin du document. Si les nombres lus sont plus élevés que le nombre d'états actuel de l'automate, on charge l'état le plus élevé possible (par exemple, si l'automate

a trois états, mais qu'on lit un 4 dans le document, on charge un 3 dans la case lue). On construit l'état de départ ainsi lu à l'aide des méthodes de la classe CABuilder1D ou CABuilder2D, puis on met à jour l'état de départ de l'automate avec celui construit.

Le chargement d'une configuration lit le document mot par mot, jusqu'à détection d'une virgule (ou fin du fichier). L'ordre des mots attendus est strict, et si le mot lu n'est pas celui attendu le chargement échoue. Le chargement met à jour l'objet CABuilder, et reconstruit l'automate (d'où l'intérêt du pointeur sur référence) à la fin de la fonction.

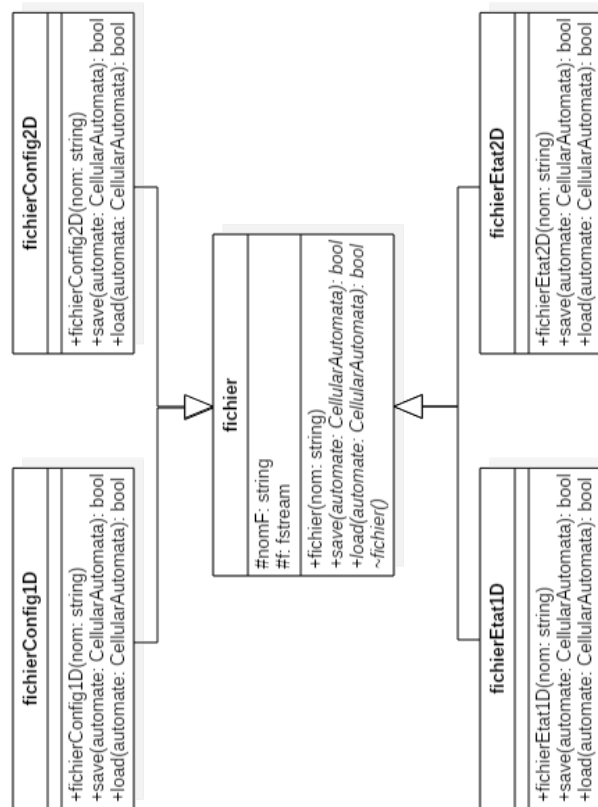


FIG. II.3 – UML des classes utilisées pour la sauvegarde

II.2 Front-end de l'application

L'interface de l'application est gérée par un ensemble de classes afin de gérer d'une part la forme de l'interface, et d'autre part le comportement de celle-ci.

Tout d'abord, la classe `MainWindow` permet de construire, grâce à un *QStackedWidget*, la fenêtre principale de l'application ainsi qu'un menu déroulant permettant à l'utilisateur de choisir entre *Automate 1D* ou *Automate 2D*. Selon son choix, une sous-fenêtre va afficher l'interface d'un automate 1D ou bien 2D.

L'interface de cette sous-fenêtre est gérée par la classe `SousFenetre`. Celle-ci fournit l'interface (et non le comportement) permettant de sauvegarder/charger le contexte de la simulation, que ce soit d'un automate 1D ou bien 2D, avec `loadContexte()` et `loadContexte()`.

Le comportement est géré par les classes `fenetreConfig` et `FenetreAutomate`, et leurs classes filles.

La première gère l'affichage de la fenêtre de configuration de l'automate (à gauche de la grille) et permet à l'utilisateur de rentrer les données de configuration de l'automate selon son type, grâce à ses classes filles `fenetreElementaryRule`, `fenetreGameOfLife` et `fenetreFeuForet`. Ainsi, l'utilisateur peut choisir, selon l'automate, l'ordre du voisinage, la règle de transition, le nombre d'états possibles pour les cellules, etc. Cette classe permet également d'afficher les informations concernant le fonctionnement de l'automate choisi, et de sauvegarder/charger le contexte d'un automate selon son type.

La classe `FenetreAutomate` permet d'afficher la sous-fenêtre entière dans la `MainWindow`: la `fenetreConfig` à gauche et le reste de l'interface (la grille, les boutons de sauvegarde, les boutons de sélection, etc.). Le comportement de ces widgets dépend du type d'automate et agira de manière différente que l'on soit dans la classe `fenetre1D` ou bien dans `fenetre2D`.

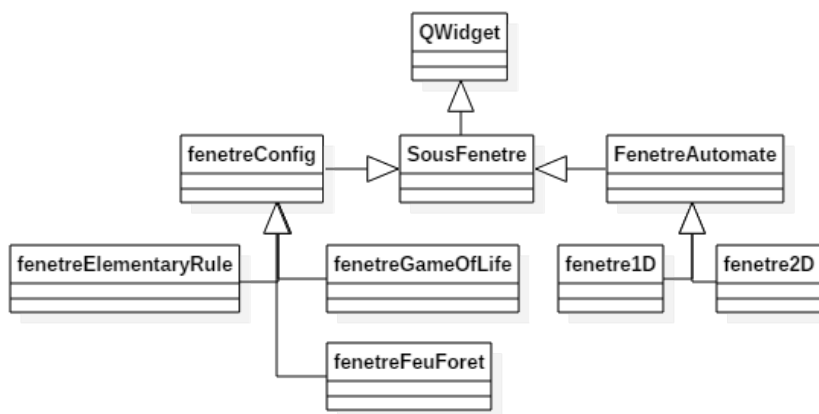


FIG. II.4 – UML simplifié de l'interface

III - Evolutivité de l'architecture

Dans notre projet, comme demandé, nous avons conçu notre architecture pour qu'elle puisse évoluer dans l'avenir, c'est-à-dire qu'on puisse y effectuer des modifications ou des améliorations le plus facilement possible. Cela signifie que nous avons dû concevoir une architecture qui, d'une part, permette de séparer les différentes parties de l'application (interface et simulation) afin de pouvoir modifier l'une sans impacter l'autre, et d'autre part permette de concevoir correctement l'application afin de pouvoir ajouter et modifier les automates et les générateurs d'états sans modifier le code du simulateur : nous avons fait en sorte que celui-ci puisse s'adapter à n'importe quel automate ou générateur.

III.1 Adaptabilité de l'architecture

L'interface est totalement indépendante du simulateur : l'adresse de l'automate est stockée dans le pointeur `m_simulateur` dans `FenetreAutomate`. On utilise ensuite différentes méthodes pour extraire les informations de l'automate (`afficherDernierEtat()`) ou le modifier (`contruireAutomate()`). Ces méthodes n'influent pas sur le fonctionnement de l'automate mais permettent simplement de le manipuler : il est possible d'écrire de nouvelles méthodes pour manipuler l'automate.

Pour simuler un nouvel automate, il suffit de définir, comme dit plus haut, son nombre d'états, sa dimension, sa règle de transition et son voisinage.

La règle de transition doit être définie dans une classe dérivée de `TransitionRule`. Le nombre d'états quant à lui est défini lors de la création de `CellularAutomata`. Il faut bien évidemment ensuite affecter un état de départ qui ne possède aucune cellule dont la valeur dépasse le nombre d'états possibles pour celle-ci. La dimension quant à elle dépend de la règle de transition utilisée comme expliqué dans la section II.1.1.

Il faut également ajouter les informations nécessaires pour la sauvegarde (méthode `getTransition()`) et le chargement (en créant de nouveaux points de choix dans la méthode `load()`). Ces informations pourront ensuite être utilisées par toutes les autres méthodes de l'automate tout en songeant à mettre à jour l'énumération `etat` si le nombre d'états est trop grand, afin de gérer les couleurs qui seront affichées par l'interface sinon la cou-

leur par défaut pour des états sera la couleur blanche et le numéro de l'état sera affiché dans la case. Il faut ensuite mettre à jour la méthode `afficherDernierEtat()` pour gérer l'affichage de ces nouvelles couleurs.

```

if(mot == "transition")
{
    a=0,b=0;
    f.getline(st,TAILLE_BUF,');
    mot = st;
    if(mot != "2D")
    {
        QMessageBox::critical(nullptr,"Erreur chargement","Le fichier est une configuration 1D.");
        f.close();
        return false;
    }
    f.getline(st,TAILLE_BUF,'); //GameOfLife ou FeuForet
    mode = st;
    if(mode == "GameOfLife")
    {
        f.getline(st,TAILLE_BUF,'); //m_minVoisinsVivants
        mot = st;
        if(mot=="")
        {
            QMessageBox::critical(nullptr,"Erreur chargement","Le fichier semble corrompu.");
            f.close();
            return false;
        }
        a=std::stoi(mot);
        f.getline(st,TAILLE_BUF,'); //m_maxVoisinsVivants
        mot = st;
        if(mot=="")
        {
            QMessageBox::critical(nullptr,"Erreur chargement","Le fichier semble corrompu.");
            f.close();
            return false;
        }
        b=std::stoi(mot);
        m.BuildGameOfLife(a,b);
        nbEtats = 2;
    }
    else if(mode == "FeuForet")
    {
        m.BuildFeuForet();
        nbEtats = 4;
    }
    else
    {
        QMessageBox::critical(nullptr,"Erreur chargement","Le fichier semble corrompu.");
        f.close();
        return false;
    }
}

```

FIG. III.1 – *Chargement de la transition dans la méthode `load()` de fichier-Config2D*

Le voisinage peut être ajouté de la même manière dans une classe héritée de `Voisinage`, que l'on décrit dans la méthode `definirVoisinage()`. On doit, de même, en fournir une description pour la sauvegarde dans `getType()` et rajouter des points de choix dans `load()`. On peut aussi utiliser un voisinage déjà existant.

Enfin, la génération d'un état est gérée par la classe `GenerateurEtat` (sauf pour la génération manuelle qui est entièrement gérée entre l'inter-

face et le simulateur). Cette classe abstraite permet de générer des tableaux aléatoires dont la génération est définie par ses classes dérivées. Il s'agit donc d'une implémentation du design pattern *strategy*, où chaque classe fille de `GenerateurEtat` représente un algorithme de calcul des valeurs des cellules d'un objet `Etat`. Il suffit ensuite d'appeler la méthode `BuildEtatDepart()` pour obtenir le tableau. Il est entièrement possible de retirer ou d'ajouter des méthodes de génération en ajoutant des classes dérivées de `GenerateurEtat` et de les définir dans la méthode `GenererEtat()`. Il suffira ensuite de rendre ce choix possible dans l'interface en modifiant les méthodes `construireEtat()`.

III.2 Ajout d'un nouvel automate

Pour ajouter l'automate `FeuForet`, nous avons d'abord dû définir ses caractéristiques : sa règle de transition, sa dimension, son nombre d'états et son voisinage. Ce dernier est défini dans une classe héritée de `Voisinage` selon son type (Moore ou Von Neumann). Il est bien entendu possible d'ajouter un type de voisinage, qu'il faut définir comme décrit plus haut.

Les trois premières caractéristiques sont définies dans la classe `FeuForet` héritée de `TransitionRule`. On y définit la méthode `TransitionCellule()`, qui définit la règle de transition d'une cellule en fonction de ses voisins (et donc en fonction du voisinage choisi). Selon l'état de la cellule, on modifie ou non son état. On vérifie que les cases vertes n'ont aucun voisin rouge, et le cas échéant la case devient rouge. On définit également la méthode `getTransition()` qui permet d'envoyer une chaîne de caractères décrivant les caractéristiques de l'automate pour la sauvegarde. Cet automate n'a pas de caractéristiques particulières, on envoie simplement son nom et sa dimension.

Cet algorithme peut être appliqué à un automate 1D, c'est pourquoi nous ne définissons pas sa dimension dans la classe. Elle est toutefois définie dans la méthode `getTransition()`, et `load()` empêche donc de charger un tel automate dans un automate 1D. De même, l'interface ne permet pas de sélectionner cette règle en 1D. Ce sont des restrictions de l'IHM et de la sauvegarde : il est tout à fait possible de les modifier pour permettre l'utilisation de cet algorithme en 1D (tout comme le jeu de la vie). Le nombre d'états et la dimension ne sont pas non plus définis ici : c'est à l'IHM et à la sauvegarde/chargement de s'occuper de leur définition, car cet algorithme pourrait également être appliqué à d'autres dimensions. Le nombre d'états est fixe : étant donné qu'il est intégré à la règle de transition, nous n'en avons besoin qu'à la création de l'automate.

Il faut ensuite ajouter l'algorithme à la sélection de l'IHM. Il faut, pour cela, créer une nouvelle classe `fenetreFeuForet` qui hérite de `fenetreConfig` (la classe que nous utilisons pour laisser l'utilisateur configurer l'automate de son choix, représenté par une classe héritée de `fenetreConfig`). Cette classe contient ses propres attributs pour définir le voisinage et redéfinit les méthodes `constructionAutomate()`, `loadContexte()` et `saveContexte()`. La première méthode permet de régler le constructeur de simulateur, et les deux dernières méthodes sauvegardent/chargent le contexte de la fenêtre lors de la fermeture/ouverture du programme. Il suffit ensuite d'ajouter l'option `FeuForet` à la boîte de sélection de l'automate `m_choixAutomate` et d'ajouter une instance de la classe `fenetreFeuForet` au *QStackedWidget* `m_automates`.

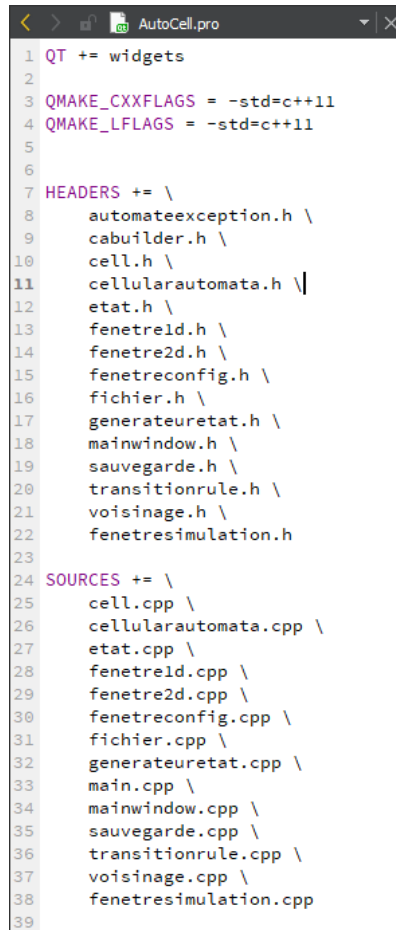
Enfin, il faut pouvoir gérer la sauvegarde et le chargement corrects de l'automate. La sauvegarde est déjà correctement gérée en complétant la méthode évoquée plus tôt. Le chargement doit être géré dans la méthode `load()` de la classe `fichierConfig2D`. Il faut ajouter un point de choix lors de la lecture du type (décrit dans les commentaires). Il faut ensuite continuer de lire le document, en testant si les données lues sont bien les mêmes que celles que l'on doit avoir enregistré. Ici, on n'a rien à lire de plus, contrairement à `GameOfLife` qui nécessite d'enregistrer plus d'informations.

IV - Annexes

IV.1 Code source

Le code source est divisé en deux catégories de fichiers : les fichiers d'entête `.h` et les fichiers sources `.cpp`. Ils ont tous été implémentés à l'aide de `Qt Creator` et utilisent des objets propres à cet éditeur. Il est donc nécessaire de faire fonctionner l'application sur cette interface.

Voici les instructions de compilation nécessaires au bon fonctionnement de notre application :

A screenshot of a Qt Creator project file (.pro) for a project named 'AutoCell.pro'. The file contains compilation instructions for Qt widgets, C++11 flags, and lists of header and source files. The lines are numbered from 1 to 39.

```
1 QT += widgets
2
3 QMAKE_CXXFLAGS = -std=c++11
4 QMAKE_LFLAGS = -std=c++11
5
6
7 HEADERS += \
8     automateexception.h \
9     cabuilder.h \
10    cell.h \
11    cellularautomata.h \
12    etat.h \
13    fenetre1d.h \
14    fenetre2d.h \
15    fenetreconfig.h \
16    fichier.h \
17    generateuretat.h \
18    mainwindow.h \
19    sauvegarde.h \
20    transitionrule.h \
21    voisinage.h \
22    fenetresimulation.h
23
24 SOURCES += \
25     cell.cpp \
26     cellularautomata.cpp \
27     etat.cpp \
28     fenetre1d.cpp \
29     fenetre2d.cpp \
30     fenetreconfig.cpp \
31     fichier.cpp \
32     generateuretat.cpp \
33     main.cpp \
34     mainwindow.cpp \
35     sauvegarde.cpp \
36     transitionrule.cpp \
37     voisinage.cpp \
38     fenetresimulation.cpp
39
```

FIG. IV.1 – Instructions de compilation contenues dans le fichier `.pro`

IV.2 Documentation

Nous avons généré une documentation du code source à l'aide de **Doxygen**. Cette documentation se situe dans le dossier **html** du projet. Pour lancer la documentation dans un navigateur, il faut ouvrir le fichier **index.html**. La page affichée doit alors ressembler à ceci :

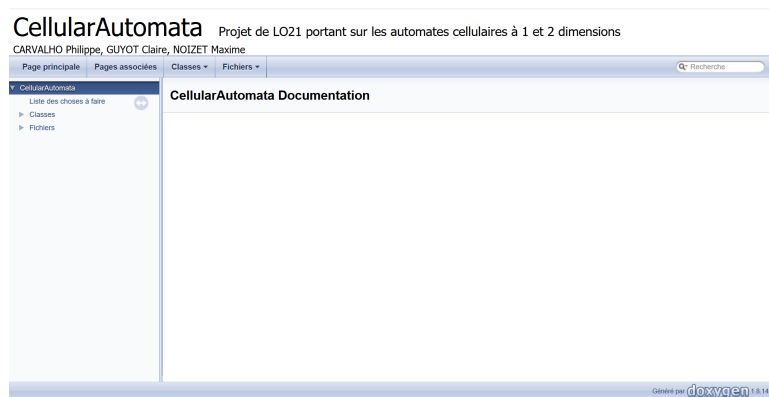


FIG. IV.2 – *Visuel de la page principale de la documentation Doxygen*

IV.3 Vidéo

Nous avons filmé une vidéo explicative de l'application, et de la manière dont utiliser les différentes fonctionnalités implémentées, depuis l'interface graphique. En lançant l'application, l'interface doit ressembler à ceci :

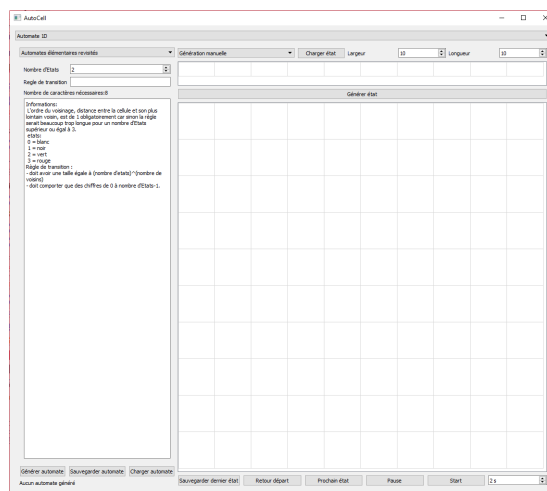


FIG. IV.3 – *Interface graphique de l'application*

IV.4 Fichiers

Les fichiers d'exemple se trouvent dans le dossier "Fichiers". Ce dossier contient cinq fichiers de configuration :

- config_1D_2etats.csv,
- config_1D_3etats.csv,
- config_1D_4etats.csv,
- config_2D_gameoflife.csv,
- config_2D_feuforet.csv.

Il contient également cinq fichiers d'états :

- etat1D_2etats.bn
- etat1D_3etats.bn
- etat1D_4etats.bn
- etat2D_1.bn,
- etat2D_feuforet.bn.