Projet LO21 – P2018

AutoCell

Bertille Dubosc de Pesquidoux, Yiwen Wang
et Natan Danous

16 juin 2018

Table des matières

Introduction

Dans le cadre de l'UV LO21, « Programmation et conception orientées objet », nous avons travaillé sur des automates cellulaires. Ceux-ci sont constitués de plusieurs cellules, unités fonctionnelles placées sur une grille. Une cellule est caractérisée par son état et il existe une règle de transition qui permet de modifier au cours du temps l'état des cellules.

Le but de ce projet était de développer une application `AutoCell` qui permet de simuler et visualiser des automates cellulaire à une ou deux dimensions. L'application a été codée en C++11 et nous avons utilisé le framework Qt pour l'interface.

Le principe même des automates cellulaires, qui est l'articulation entre les états possibles, les cellules, leurs voisines et les règles de transition, oblige à adopter une approche orientée objet. C'était donc une occasion d'appliquer les pratiques de conception et de développement vues en cours. Pour comprendre comment nous avons mobilisé ces concepts dans notre code, voir la partie 2, dans laquelle nous détaillons nous architecture.

Dans la première partie de notre rapport nous présenterons le principe des automates cellulaire en donnant quelques définitions, les type d'automates existant et enfin, les automates implémentés. En seconde partie, nous exposerons nos choix pour l'architecture. En troisième partie nous nous arrêterons sur les évolutions possibles pour notre application, en particulier sur la marche à suivre pour implémenter de nouveaux automates.

1 Les automates cellulaires

1.1 Principe

Les automates cellulaires sont des objets mathématiques souvent étudiés dans le cadre de l'informatique théorique et des systèmes complexes. Ils visent à simuler le comportement d'un ensemble de cellules. Celles-ci sont disposées selon une grille et se trouvent à chaque instant dans un état bien défini. En général, un état est représenté par une valeur scalaire et le nombre d'états possibles est fini. La grille comporte une, deux, ..., n dimensions. Elle est théoriquement infinie mais souvent codée par une matrice de taille finie, éventuellement torique. Le système évolue de façon discrète, c'est-à-dire qu'une cellule dans un état E à l'instant t peut passer à l'état E' à l'instant $t+1$ sans étape intermédiaire. Cet état E' est déterminé par un ensemble de règles non ambiguës.

À partir de ce schéma général, on peut définir des catégories d'automates en choisissant la dimension de la grille, la forme du voisinage, le nombre d'états et, bien entendu, les règles de transitions

1.2 Types d'automates

Ces règles doivent notamment clarifier :

— la liste des états dans lesquels chaque cellule peut se trouver ;

— quelles cellules sont considérées pour le calcul de l'état de la cellule x à l'instant $t+1$;

— comment ce calcul est effectué.

Ces informations servent à identifier un automate et à le nommer. Pour mieux comprendre de quoi il en retourne, nous vous invitons à lire la sous-partie suivante pour des exemples d'automates.

1.3 Automates implémentés

Les automates les plus simples sont ceux dont la grille comporte une seule dimension, qui autorisent deux états pour les cellules (morte et vivante) et pour lesquels seules trois cases sont considérées : celle dont on veut calculer l'état suivant et les deux cellules adjacentes. Pour un voisinage donné, il y a $2^3 = 8$ combinaisons possibles de cellules mortes et vivantes. Il est donc possible d'implémenter tous les automates de ce type, ce que nous avons fait. L'utilisateur peut configurer sa règle, soit en donnant le numéro qui y est associé, soit en choisissant si la cellule survit ou non dans chacune des configurations possibles.

Pour les automates en deux dimensions, il devient plus difficile de donner une définition *in extenso*. Si l'on se place dans un cas similaire, où l'on prend en compte 9 cellules (celle qu'on considère et les 8 qui l'entourent), avec deux états possibles,

il y a déjà $2^9 = 512$ cas à traiter. Difficile de demander à l'utilisateur si la cellule doit survivre, naître ou mourir dans chacun de ces cas. Aussi, rien n'empêche de considérer plus de voisines (avec le deuxième rang par exemple, ce qui mène à 15 cellules qui entrent dans le calcul) ou en imaginant qu'une cellule peut prendre plus d'états.

Nous n'avons donc implémenté que deux familles d'automates en deux dimensions : le jeu de la vie et *Wireworld*.

Le jeu de la vie est un automate en deux dimensions qui autorise deux états pour chaque cellule : morte ou vivante. Les cellules considérées sont les 8 voisines directes. La règle « classique » est qu'une cellule naît au tour suivant si elle a exactement trois voisines vivantes, elle meurt si elle a moins de deux ou plus de quatre voisines vivantes et elle reste dans le même état si elle en a deux.

Wireworld est un automate qui simule des circuits électriques. Chaque cellule peut être vide ou conductrice, et des électrons (composés d'une cellule tête et d'une cellule queue) circulent sur les cases conductrices.

2 Architecture

Notre architecture est séparée en deux parties. La première partie concerne le fonctionnement interne de l'application, il s'agit donc de la logique permettant la simulation des automates que l'on appelera *système*. La seconde partie concerne l'*interface* utilisateur. Celle-ci interagit avec le système en recevant et en envoyant des données au système, elle permet l'interaction avec le simulateur au travers d'une interface graphique. Le système et l'interface ont été codés en C++11, l'interface utilise le framework Qt.

2.1 Système

2.1.1 Les Automates

Tous les automates ont un fonctionnement et des « besoins » similaires. Nous avons donc une première classe *Automate* qui permet de définir les attributs de base ainsi que les méthodes à implémenter pour qu'un objet soit un automate. La classe est évidemment abstraite de par la règle de transition (méthode virtuelle pure `appliquerTransition(Grille* dep, Grille* dest)`) et la dimension qui changent en fonction de l'automate.

La classe *Automate* est ensuite héritée par deux autres classes *Automate1D* et *Automate2D*, elles aussi abstraites. Elles sont abstraites car la règle de transition change en fonction de l'automate. Elles servent en quelque sorte d'interface avec *Automate* en appelant le constructeur de celle-ci en spécifiant directement la dimension. Nous reviendrons plus tard sur l'intérêt de cette séparation.

Avec ces deux classes, nous pouvons définir les trois classes d'automates que nous avons choisis d'implémenter dans notre application : *AutomateElementaire*, *AutomateGoL* (Jeu de la Vie) et *AutomateWW* (Wireworld). Les trois classes implémentent le design pattern *singleton*, elles disposent d'autant de méthodes `static Automate* getInstance(...)` que de constructeurs (privés). Nous avons fait ce choix car l'application ne simule qu'un automate dans toute la durée de vie de l'application et nous avons besoin d'y accèder à différent moments et endroits de l'application. Dans notre cas, les méthodes `getInstance(...)` retournent un nouvel Automate seulement si l'automate demandé est différent de celui actuel (configuration différente).

2.1.2 Les États et les Cellules

La classe *Etat* est partagée entre la classe *Automate* et la classe *Cell* (Cellule). En effet, un automate a un ensemble fini d'états que peuvent prendre les cellules. Ainsi, la classe *Automate* a un membre privé `etatsPossibles` qui est un tableau d'objets *Etat*. Un objet *Cell* a un (unique) attribut de type *Etat*. On respecte ainsi sémantiquement la description des automates et des cellules.

Pour revenir à la classe *Etat*, celle-ci a deux attributs, le premier est un `int` qui contient la valeur de l'état. Celle-ci est arbitraire et le choix de la signification des valeurs est laissé à l'utilisateur de la classe. *Etat* dispose également d'un attribut `std::string desc` qui permet d'associer une courte description à la valeur d'un état. Par exemple, le constructeur par défaut de la classe *Automate* définit deux états possibles : `Etat(0, "morte"), Etat(1, "vivante")`.

2.1.3 Les Grilles

Les grilles n'étaient pas spécifiquement prévues en début de projet, mais lors de l'implémentation des règles de transition, il est vite apparu qu'une nouvelle couche d'abstraction était nécessaire. Celle-ci a également beaucoup simplifié la programmation de l'interface.

Les deux types de grilles auxquels nous avons affaire dans ce projet sont à l'image des automates : 1D et 2D. On procède de même manière, on a une première classe *Grille* qui est abstraite. Deux classes *Grille1D* et *Grille2D* héritent de celle-ci. La grille 1D alloue un tableau unidimensionnel d'objets *Cell*. La grille 2D alloue un tableau bidimensionnel d'objets *Cell*. Les grilles implémentent une partie du design pattern *prototype* avec la fonction `clone()` qui permet de cloner une grille sans connaître son type (1D ou 2D) ce qui sera utile pour le simulateur.

2.1.4 Le Simulateur

La classe *Simulateur* est le « cerveau » de notre application. Elle permet de faire évoluer une grille de cellules selon un automate en appelant la méthode `appliquerTransition (Grille* dep, Grille* dest)` de celui-ci. On garde en mémoire au minimum 2 générations, la génération actuelle et la génération précédente.

2.2 Interface

De la même manière que tous les automates partagent des caractéristiques communes, les interfaces de automates partagent des éléments communs. En effet, pour tous les automates, un système d'état a été mis en place. Il permet de garder un état global de l'interface. Ce système d'état contient entre autres les dimensions de la grille, la taille des cellules, l'état actuel et un booléen `paused` qui nous indique si le simulateur est en pause ou non. Ces deux derniers attributs nous permettent de gérer le système de lecture de l'automate. Spécifiquement, si l'utilisateur décide dans un premier temps de faire une lecture simple puis de mettre en pause et de faire une lecture pas à pas, celle-ci reprendra depuis la dernière génération simulée. Ainsi l'utilisateur a une prise en main naturelle du système de lecture.

Au niveau de l'architecture, nous avons fait le choix d'avoir une classe abstraite *AutomateView* de laquelle, les trois classes d'interfaces héritent. Au niveau de l'in-

terface, l'organisation est simple, *MainWindow* contient un *QStackedWidget* qui lui-même instancie 3 *Widgets* correspondant à nos trois classes d'interface.

2.3 Sauvegarde et importation

La sauvegarde d'un automate se fait grâce à la fonction `save(QFile* f, ...)` présente dans *AutomateView*. Les automates sont sauvegardés dans des fichiers `.xml`. Tout est sauvegardé : la configuration, la grille de départ mais également la grille actuelle, avec les états de toutes les cellules. L'élément racine de ces fichiers `.xml` est `<automate>`. On fait la distinction entre les automates grâce à l'attribut `type=""` qui peut être `elementaire, gol` ou `ww`. L'importation traverse le fichier `.xml` et remplit l'interface avec ces données. Nous avons utilisé deux classes Qt pour écrire et parser les fichiers `.xml` : *QXmlStreamWriter* et *QXmlStreamReader*.

Pour la sauvegarde du contexte de l'application, il suffit d'appeler les trois fonctions de sauvegarde des automates dans le destructeur de *MainWindow*. On enregistre ces trois fichiers dans le dossier `QStandardPaths::DataLocation` qui correspond au dossier des données de l'application (la location de ce dossier dépend du système d'exploitation de l'utilisateur). On crée le dossier si il n'existe pas. Pour l'importation, il suffit de regarder pour chacun des automates si il existe un fichier de configuration dans le dossier des données qui leur correspond et de l'importer si c'est le cas.

3 Évolution

Afin d'ajouter un automate, il suffit de faire hériter celui-ci de la classe abstraite d'automate qui lui corresponde : *Automate1D* (resp. *Automate2D*) si il est en une dimension (resp. deux dimensions). Il faut ensuite implémenter la fonction `appliquerTransition` correspondante et mettre en place le design pattern *singleton* de la même manière qu'il a été fait sur les autres automates implémentés (faire correspondre les `getInstance(...)` avec les constructeurs). Si l'automate requiert plus de deux états, il est possible de faire appel à un constructeur où l'on peut passer en argument un tableau des états possibles.

Au niveau de l'interface, il suffit de créer une nouvelle classe qui hérite de *AutomateView*. Il faut ensuite ajouter au `stackedWidget` un nouveau pointeur de cette View. Enfin, il faut implémenter au minimum toutes les fonctions virtuelles de *AutomateView*. L'interface graphique est réalisée au travers de fichiers `.ui` et il est nécessaire de garder certaines conventions sur le nommage des widgets d'entrée utilisateur (`inputTailleCell, inputSpeed,` etc...) (voir les constructeurs de *ElementaireView* ou *GoLView*).

Pour le format de sauvegarde, il faut au moins un élément racine : `<automate type="typeAutomate">`. Un axe d'amélioration aurait pu être de créer une DTD.

Conclusion

Dans ce rapport, nous avons exposé l'architecture et l'évolution possible de notre application `Autocell`.

Pour conclure, ce projet nous a permis de mettre en œuvre des *design patterns* et des bonnes pratiques de programmation orientée objet. Nous avons donc gagné à la fois en rigueur et en créativité.

Ce fut aussi pour certains membres du groupe la première occasion de réaliser un tel projet, de réaliser une application du début à la fin et d'utiliser le *framework* Qt.

Au-delà de l'aspect purement technique du développement de fonctions et d'interfaces, il a aussi fallu se mettre d'accord sur une architecture et donc débattre des patrons de conception à notre disposition. Nous avons eu l'obligation, pour pouvoir faire un choix éclairé en tant que groupe, de comprendre plus en profondeur les possibilités que recèle le langage C++ et de nous l'approprier.