Clément Gresh
M1 IMPAIRS
2021-2022

Projet Langages à Objets Avancés
Sujet n°2 : simulation d'un circuit combinatoire

Table des Matières

Table des Figures

I. INTRODUCTION

Le but de ce projet de Programmation Orientée Objet est de créer en *C++* un programme permettant de simuler un circuit combinatoire à l'aide d'entrées, de sorties et de portes logiques (*AND*, *OR*, *NOT*, etc.). L'utilisatrice doit pouvoir affecter des valeurs aux entrées. Ces valeurs sont ensuite propager pas-à-pas au reste du circuit et une représentation de l'état courant de celui-ci est affichée à chaque étape.

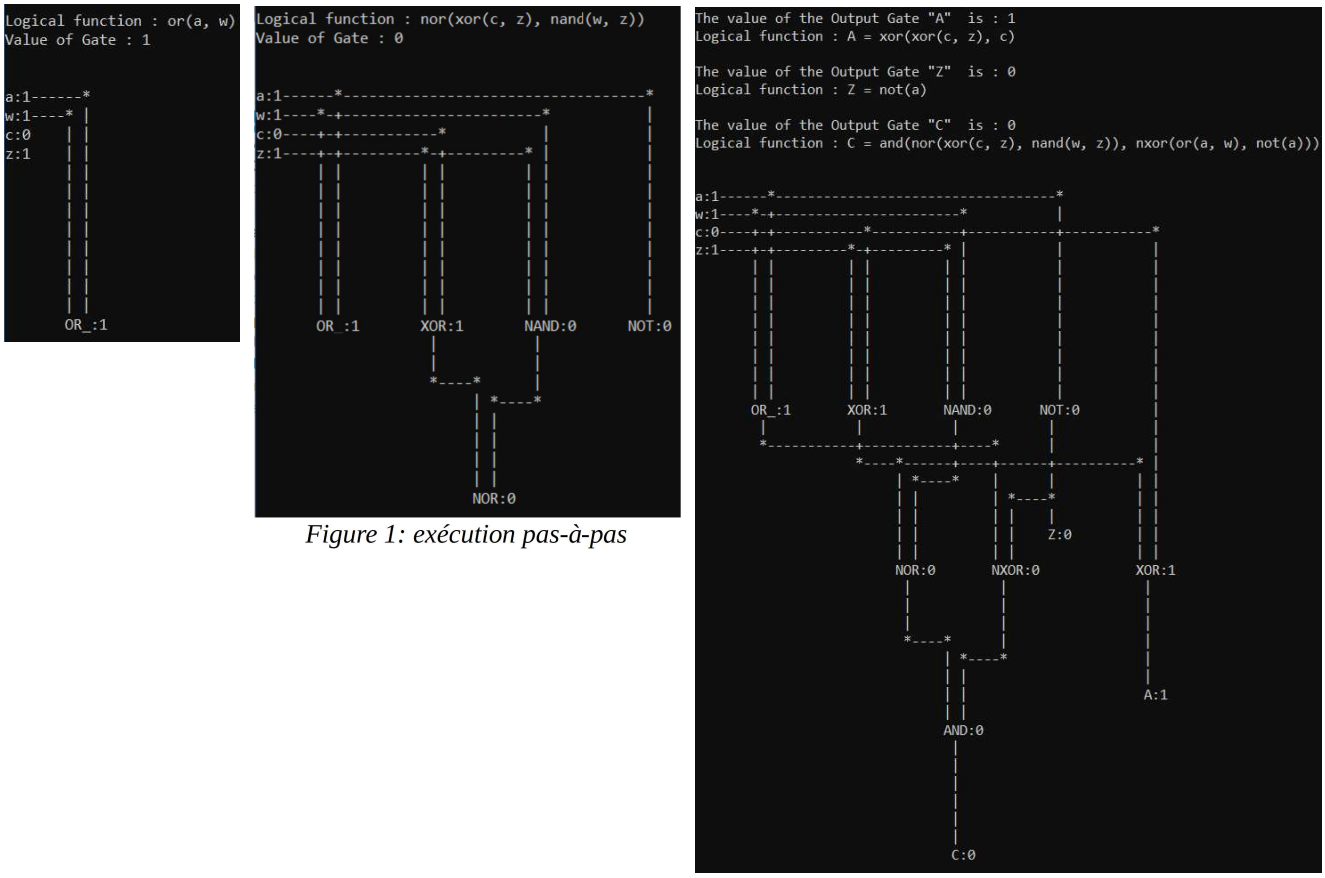

Figure 1: exécution pas-à-pas

Les fonctionnalités supplémentaires qui ont été implémentées sont :

- affichage de l'expression textuelle des portes (e.g. « nand(a, e) »).
- sauvegarde d'un circuit dans un fichier.
- création d'un circuit à partir d'une expression textuelle (après avoir vérifié sa correction).

II. PORTES

1. Classe *Gate*

Toutes les catégories de portes étendent une classe de base abstraite *Gate*. Elle définit notamment une méthode virtuelle pure nommée *drawGate* qui, en combinaison avec la méthode *draw* (servant à factoriser le code en commun), doit permettre d'ajouter le nom de la porte ainsi que sa valeur courante au schéma.

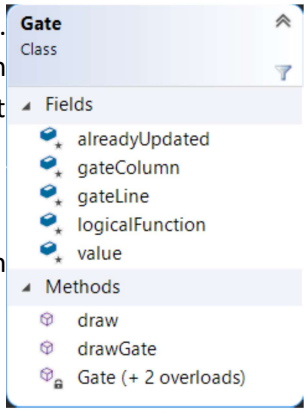

Figure 2: classe Gate

Chaque porte a des attributs *gateLine* et *gateColumn* indiquant son emplacement dans le schéma.

2. Entrées / Sorties

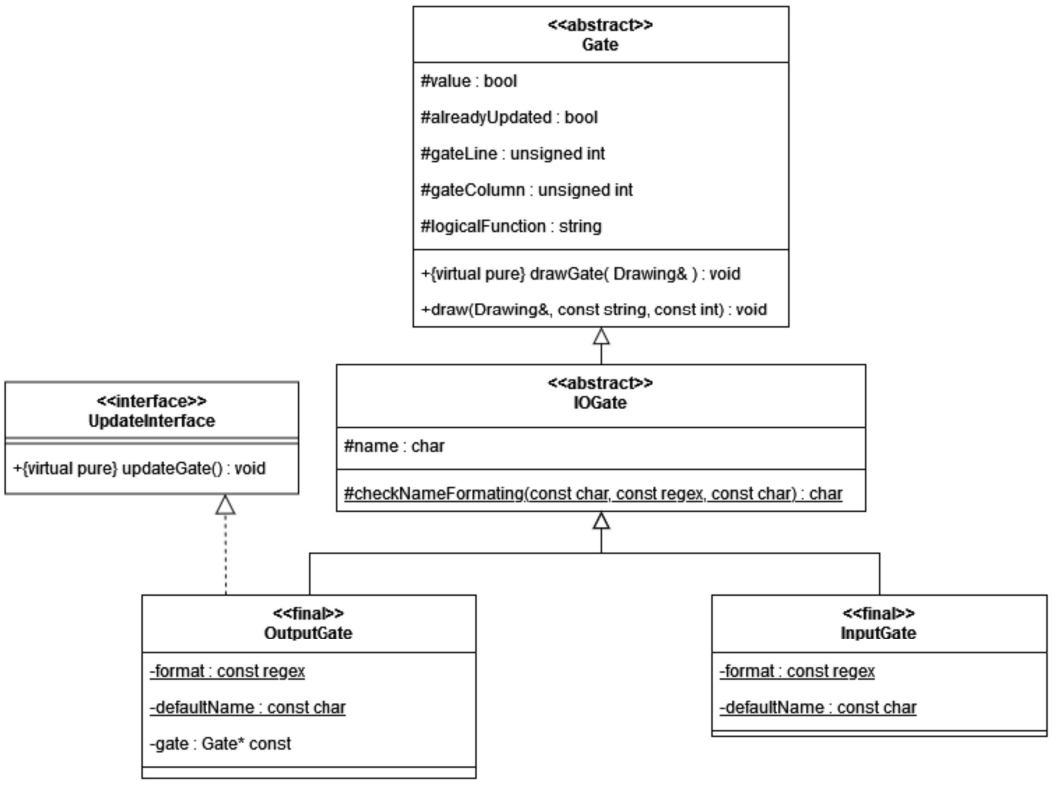

Figure 3: portes entrée / sortie : diagramme UML

Les portes entrée / sortie ont en commun un attribut *name* qu'elles héritent de la classe abstraite *IOGate*. Il doit s'agir d'une lettre minuscule pour les entrées et d'une lettre majuscule pour les sorties. Lors de la construction d'un objet, une méthode *checkNameFormating* permet de vérifier que le nom indiqué

respecte la contrainte correspondante et d'utiliser un nom par défaut dans le cas contraire. On définit donc une expression régulière (REGEX) ainsi qu'un nom par défaut pour chaque classe sous la forme d'attributs statiques.

```
OutputGate::OutputGate(const char name, Gate* const gate) :
    IOGate{ IOGate::checkNameFormating(name, OutputGate::format, OutputGate::defaultName) },
    gate{ gate } {}
```

Figure 4: construction d'une OutputGate

Un objet de classe *OutputGate* doit également pouvoir mettre à jour en fonction de son entrée :

- son emplacement dans le schéma (ligne et colonne).
- sa fonction logique.
- sa valeur.

Cette classe implémente donc l'interface *UpdateInterface* qui l'oblige à redéfinir la méthode *updateGate*.

3. Portes logiques

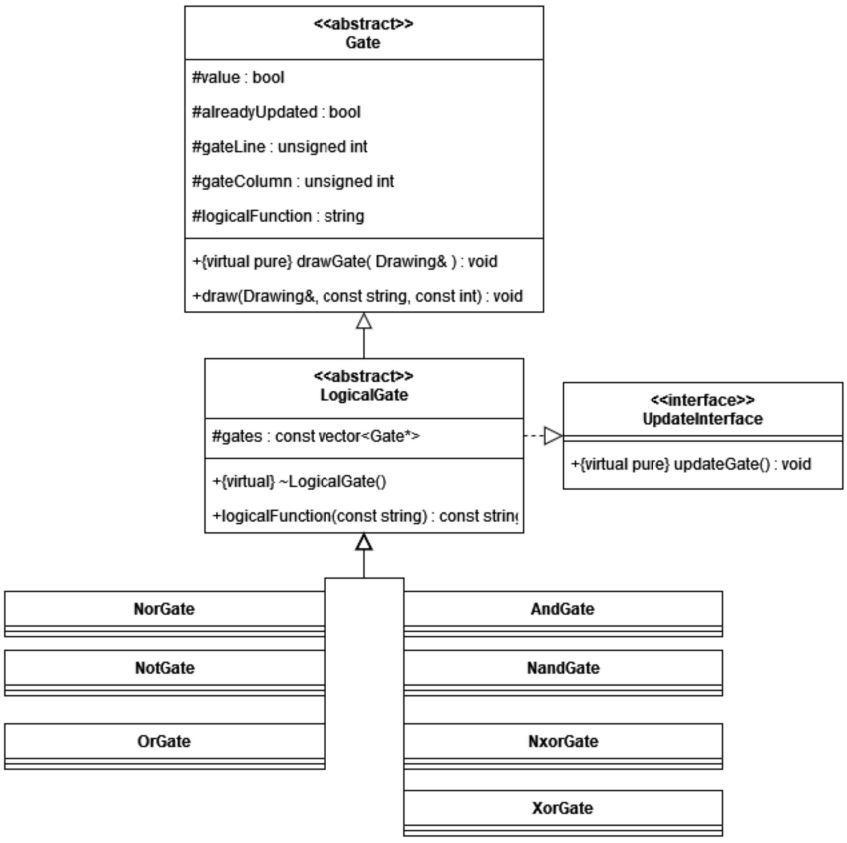

Figure 5: portes logiques : diagramme UML

La classe *LogicalGate* définit un attribut *gates*. Il permet à chaque porte logique de stocker des pointeurs vers ses entrées. Puisqu'il s'agit d'un vecteur, cela permet la plupart du temps de ne pas avoir à définir de traitement différencié selon que la porte logique a une ou deux entrées (il suffit de parcourir le vecteur).

Une méthode *logicalFunction* permet également de mettre à jour simplement les fonctions logiques. Les classes enfant n'ont alors plus qu'à indiquer le nom de la porte qu'elles représentent.

On pourra noter la présence d'un destructeur virtuel au sein de la classe *LogicalGate*. Le but est d'éviter les problèmes, notamment de fuite de mémoire, lors de la destruction de portes logiques dont on ne connaît pas le type exact (e.g. détruire un objet *NotGate* alors qu'il est typé comme *LogicalGate* dans le code correspondant).

Enfin, on oblige chaque classe enfant de *LogicalGate* à redéfinir les méthodes *updateGate* et *drawGate* (respectivement via *UpdateInterface* et *Gate*) car les résultats de celles-ci (respectivement la valeur de la porte et le nom affiché sur le schéma) dépendent du type de porte que la classe représente.

```
void AndGate::updateGate() {
    // Updating the logical function corresponding to this gate
    std::string function = this->logicalFunction("and");
    this->setLogicalFunction(function);

    // Updating the value of the gate
    this->setValue(
        this->getGates().at(0)->getValue() && this->getGates().at(1)->getValue()
    );
}

void AndGate::drawGate(Drawing& d) { this->draw(d, "AND", 1); }
```

Figure 6: classe AndGate : redéfinition des méthodes abstraites

III. CIRCUIT

1. Classe *Circuit*

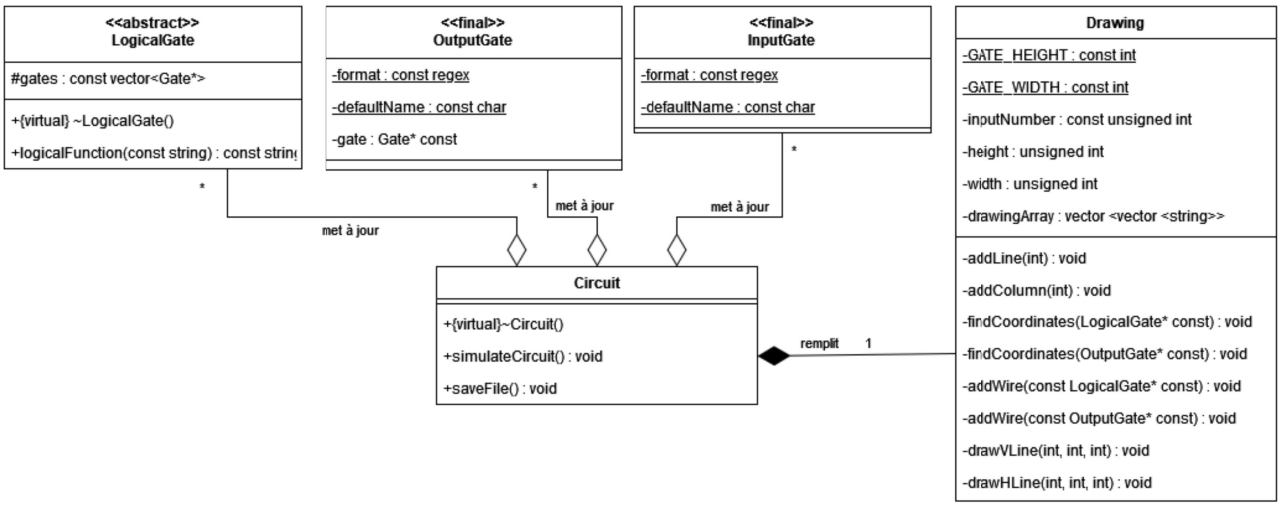

Figure 7: circuit : diagramme UML

Un objet de classe *Circuit* contient des listes de portes d'entrée, de portes logiques et de portes de sortie sous la forme de trois vecteurs de pointeurs, ainsi qu'un schéma *via* un objet de classe *Drawing*. Son destructeur a pour charge de détruire toutes les portes utilisées et une méthode *saveFile* permet, une fois le circuit complété, d'enregistrer les expressions textuelles des sorties ainsi que le schéma dans un fichier.

La mise à jour pas-à-pas du circuit se fait en plusieurs étapes grâce à la méthode *simulateCircuit* :

- réinitialisation du circuit : on met l'attribut *alreadyUpdated* des portes logiques à faux et on efface le schéma.
- initialisation des entrées : l'utilisatrice attribue des valeurs aux entrées qui sont ensuite ajoutées au schéma.
- actualisation des portes logiques : on parcourt la liste des portes logiques et, pour chacune, si ses entrées ont déjà été mises à jour, on l'actualise et on l'ajoute au schéma qu'on affiche. Si l'ensemble des portes logiques est balayé plus de fois qu'il n'y a de portes, c'est que le circuit ne peut terminer et une exception est lancée.
- mise à jour des sorties et de leurs expressions logiques. Si l'entrée d'une des sorties n'est contenue ni dans la liste des portes d'entrée, ni dans celle des portes logiques, le circuit ne peut terminer et une exception est lancée.
- affichage du circuit complet.

2. Affichage : classe *Drawing*

L'affichage est possible grâce à l'attribut *drawingArray* de la classe *Drawing*. Il s'agit d'une matrice rectangulaire, i.e. un vecteur de vecteurs de *string*. Chaque case contient un caractère (un espace blanc par défaut) qui peut être modifié via la méthode *draw*. Ces caractères sont ensuite simplement imprimés dans l'ordre, par exemple *via* la surcharge de l'opérateur d'affichage <<, pour afficher le dessin représentant le circuit logique.

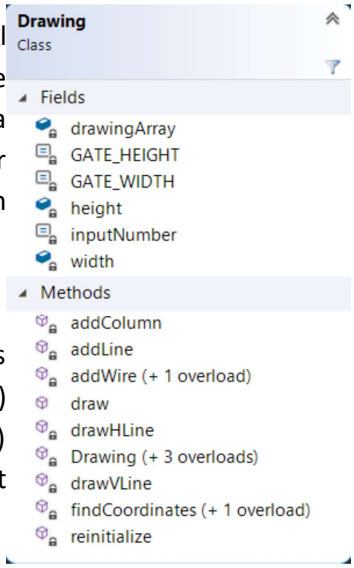

Figure 8: classe Drawing

Lors de la construction, *drawingArray* est initialisé avec un nombre de lignes égal au nombre de portes d'entrée du circuit. La méthode *addColumn* (resp. *addLine*) permet ensuite d'ajouter des colonnes (resp. des lignes) de la hauteur (resp. largeur) du tableau. Les constantes de classe *GATE_WIDTH* et *GATE_HEIGHT* permettent quant-à-elles de paramétrer l'espacement entre les portes.

Les méthodes *findCoordinates* servent à déterminer les coordonnées d'une porte logique ou d'une porte de sortie. L'ordonnée est calculée comme étant un cran au-dessus de l'ordonnée de l'entrée la plus haute. Concernant l'abscisse :

- si l'une des entrées est une *InputGate*, on ajoute des colonnes au schéma et on y place la porte. Cela permet d'éviter qu'un « fils » puisse être tracé là où se trouve déjà une porte.
- sinon, il est calculé comme étant la moyenne des abscisses des entrées. On le décale ensuite d'un cran vers la droite tant que la porte se trouve à une place déjà occupée.

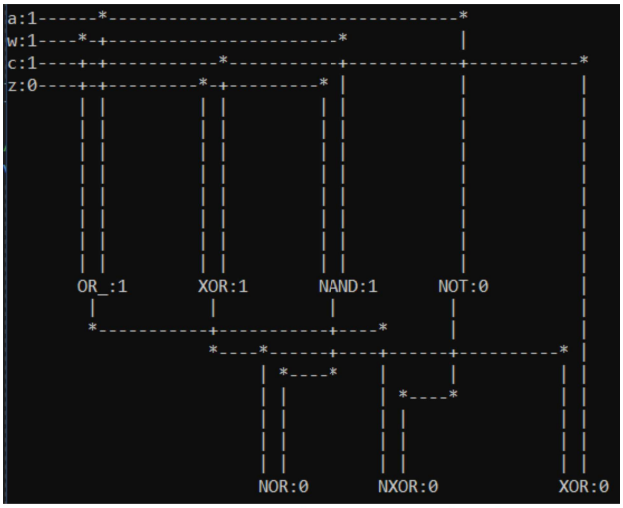

Figure 9: placement des portes sur le schéma

Les méthodes *addWire* permettent quant-à-elles de tracer les « fils » entre une porte et ses entrées. Une partie du code est factorisé via les méthodes *drawVLine* et drawHLine qui servent respectivement à

tracer des lignes verticales et horizontales. On s'assure également que si la première entrée est une *InputGate* ou bien qu'elle se trouve sur la droite de la porte, alors c'est « l'entrée » de droite de celle-ci qui est utilisée (afin de limiter le nombre de fils se croisant).

3. Création : *namespace Parser*

Le *namespace Parser* contient une méthode *userInput* qui permet, à l'aide d'une chaîne de caractères et d'une expression régulière, de demander à l'utilisatrice d'entrer des informations et de renouveler cette demande tant que l'entrée ne correspond pas au format attendu.

Mais ce *namespace* contient avant tout des variables et méthodes permettant de créer un circuit logique à partir d'une expression textuelle telle que « *A = and(a, or(c, d))* ». Lorsque l'utilisatrice en entre une, un certain nombre de vérifications sont opérées pour s'assurer qu'elle peut être transformée en un circuit. Des expressions régulières, ainsi que les erreurs associées, sont regroupées dans des vecteurs permettant de découper l'expression textuelle en morceaux pour la vérifier. Par exemple, le début de l'expression doit être un unique caractère majuscule suivi d'un signe égal. Ou encore, toute porte autre que « not » doit avoir deux paramètres entourés de parenthèses et séparés par une virgule. Ces vérifications sont faites de manière récursive sur chaque porte et, le cas échéant, chacun de ses paramètres.

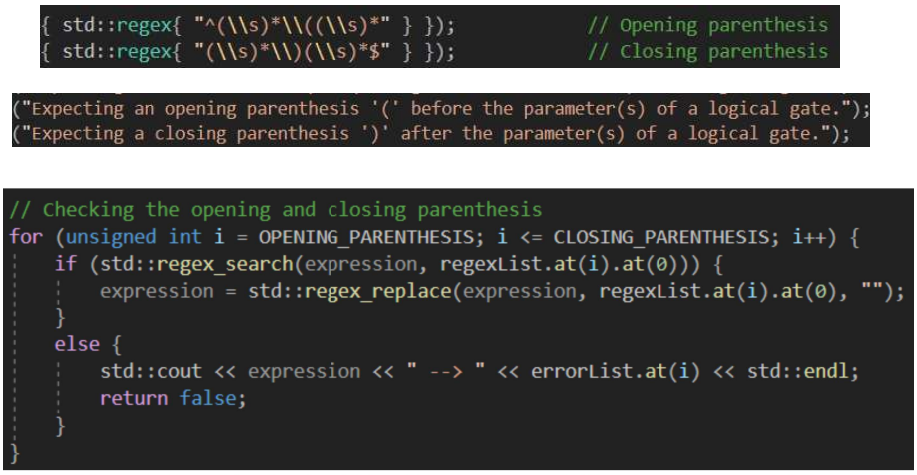

Figure 10: méthode checkGateExpression : vérification des parenthèses

Une fois ces vérifications faites, un traitement récursif similaire est appliqué pour créer les entrées, sorties et portes logiques. Les pointeurs correspondants sont stockés dans des vecteurs qui sont ensuite utilisés pour créer le circuit.

IV. CONCLUSION

Ce projet a été formateur concernant la programmation orientée objet en *C++*, de l'héritage aux références constantes en passant par les nuances de la construction et de la destruction d'objets.

Je tiens à remercier les professeur.e.s du module de Langages à Objets Avancés, Mme Micheli et MM. Yunès, Andriambolamalala et Jurski, pour leur aide tout au long du semestre et leurs réponses à mes nombreuses questions. Je voudrais aussi remercier mes camarades de classe sans lesquel.le.s il est probable que je n'aurais pu mener à bien ce projet.

De nombreuses autres fonctionnalités pourraient encore être implémentées : créer un circuit à partir du chargement d'un fichier, autoriser plus de deux entrées pour les portes AND et OR, permettre à l'utilisatrice de créer un circuit avec plusieurs sorties à partir d'une expression textuelle, etc.