HES-SO

Projet C++ : Logic Emulator

Rapport

Gilles Mottiez
17/06/2018

Table des matières

Introduction 2
But et cahier des charges 2
Développement 2
Conception globale 2
Packages 2
Patterns 3
Chargement des fichiers 4
Décodage du fichier Json 4
Génération du modèle logique 5
Calcul des sorties selon le type de portes et l'état des entrées 5
Affichage graphique 6
Modification des entrées et validation des modifications 7
Problèmes et solutions 7
Gestions des erreurs de fichiers 7
Éditeur d'entrées 8
Affichage graphique 8
Résultats 8
V0.0 8
V1.0 8
V2.0 8
Conclusion 8
Signature 9
Annexes 9

Introduction

Afin de mettre en pratique les connaissances acquises lors du 3ème et 4ème semestre du cours Inf2, nous avons dû créer le software de notre choix. Ce software se devait de répondre à certaines exigences, définies dans le cahier des charges. De plus, il devait être conçu selon des patterns et des règles fiables, étudiées lors de l'année.

Mon choix s'est porté sur un émulateur de logique numérique. J'ai choisi ce sujet car je le trouve concret et il permet d'associer 2 branches enseignées aux cours.

But et cahier des charges

Concevoir un software qui permette la simulation d'un système de logique numérique. Le fonctionnement est le suivant :

- Des fichiers Json sont écrit selon un modèle précis, qui est défini dans le document « JSON_logicProtocol.docx ».
- Le software permet de charger un de ces fichiers, de générer un modèle de ce fichier en porte logiques, d'en calculer l'état de ses portes et d'afficher une représentation graphique.
- Grâce à une fenêtre textuelle, l'utilisateur peut changer des valeurs d'entrées sans avoir à modifier tout le fichier et à le recharger complétement

Versions :

- V0.0
  - Charger un fichier Json dans le software
  - Implémentation des portes AND, OR, NOT
  - Création de la représentation graphique
  - Génération de la logique
- V1.0
  - Création de bloc logiques tels que XOR, NAND, NOR, …
  - Possibilité d'éditer le code directement à l'écran pour la modification des signaux d'entrée
- V2.0
  - Bascule D

Développement

Conception globale

Ce software a été conçu selon la théorie du cours « software engineering ». Divers patterns ont été utilisés et sont décrit ci-dessous.

Packages

- *UserInterface* : ce package contient la classe IOView qui s'occupe de l'affichage des données à l'écran, ainsi que la récupération des informations saisies par l'utilisateur
- *Controller* : centre névralgique du software, il contient la classe du même nom, qui est la machine d'état du système. Elle s'occupe de traiter les événements et de donner la voie à suivre.
- *Data* : il s'agit du modèle du système, c'est ce package qui contient toutes les données, ainsi que les méthodes qui permettent de les traiter, sur les ordres du Controller.
- *Interfaces* : regroupe toutes les interfaces de communication entre les différents packages grâce au pattern des « Port » de communication.

Patterns

- *MVC :*

Pour Model View Controller. Il s'agit d'un découpage du logiciel en fonctions primaires telles que la récolte de données, la prise de décision quant à la route à suivre, le traitement des données et enfin l'affichage des résultats.

- *Machine d'états et XFEvents :*

Elle permet de gérer de manière centralisée le déroulement du processus. Elle a l'avantage de contenir toute la gestion événementielle du logiciel en un seul endroit, ce qui garantit la robustesse. Nous utilisons un XF, qui gère de manière transparente pour le programmateur le traitement des événements. Il ne nous reste qu'à envoyer des XFEvents au bon moment, et de les traiter ensuite convenablement, grâce au pattern suivant.

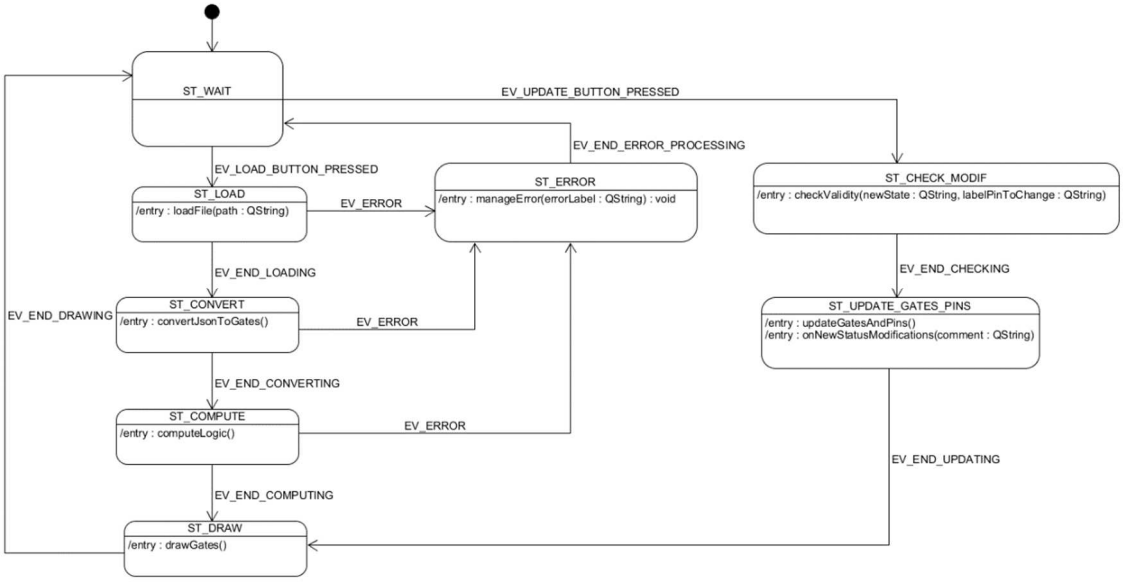

Figure 1 : machine d'états

- *Double switch pattern :*

Implémenté dans la machine d'état de la classe Controller, il permet de gérer les passages d'un état à l'autre. Il propose d'utiliser des méthodes « on transition, on entry, on exit ». Pour cela, on utilise les XFEvents qui permettent de switcher entre les états.

- *Portes de communications :*

Elles permettent l'échange de données entre les différents packages. Chaque package à sa propre porte qui permet de communiquer avec les autres portes. Les méthodes sont strictement définies dans les Interfaces.

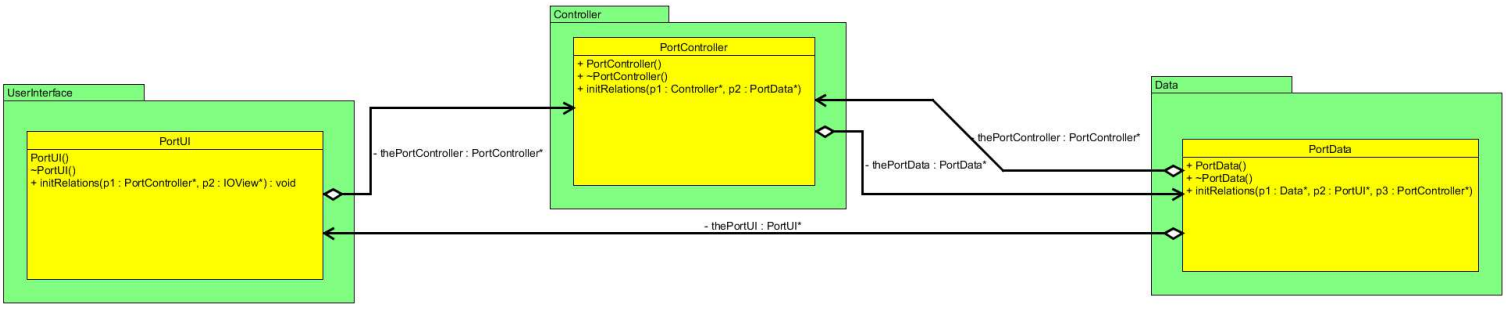

Figure 2 : portes de communication

Ce diagramme permet aussi de comprendre le MVC : une notification parvient au Controller depuis la UserInterface. Le Controller décide de quoi faire, il notifie à son tour le Model, qui va mettre à jour ses données, puis le Model renvoie au Controller ses nouvelles informations afin qu'il prenne une décision. Finalement le Controller communique au Model la direction à prendre, soit le nouvel état, et le Model s'occupe d'envoyer les données à la UserInterface pour afficher les résultats à l'écran.

Chargement des fichiers

Lorsque l'utilisateur clique sur le bouton « load », un explorateur de fichier s'ouvre et l'utilisateur peut choisir un fichier Json précédemment écrit à charger. La classe QFileDialog est utilisée. Elle permet de gérer ceci très facilement et de récupérer un QString contenant le chemin du fichier voulu.

Un QFile* est ensuite créé grâce au chemin du fichier, il suffit de lire ce QFile* avec un QTextStream et sa méthode readAll() et le tour est joué.

```
QString IOView::getPath()
{
    //set the filter to get only .json files
    QString filter = "File Description (*.json)";

    //get the opened file's path
    QString filePath =  QFileDialog::getOpenFileName(
                this,
                "Open Json file",
                QDir::homePath(), filter);
    return filePath;
}
```

Décodage du fichier Json

Qt fournit 3 classes hautement utiles à la lecture d'un fichier Json. Elles se nomment :

- QJsonDocument : création d'un objet soft à partir d'un fichier Json
- QJsonObject : création d'un objet soft à partir d'un élément d'un fichier Json
- QJsonArray : création d'un tableau soft à partir d'un tableau Json

Il est ainsi relativement aisé de décomposer un fichier Json en suivant une certaine méthodologie, qui consiste à procéder de l'extérieur vers l'intérieur :

- Récupération d'un objet QJsonDocument à partir d'un QByteArray obtenu par le QString
- Récupération d'un objet QJsonObject à partir de l'objet QJsonDocument
- Récupération des attributs en utilisant l'opérateur [] d'un QJsonObject et en spécifiant le nom de la variable à récupérer ainsi que le type de variable dans lequel on veut stocker sa valeur

```
//get the name of the file
QString fileName = design["name"].toString();

//get the array of all the gates
QJsonArray gates = design["gates"].toArray();
```

Génération du modèle logique

À partir des variables récupérées du fichier Json, il faut créer des objets héritant de la classe Gate, de type AndGate, OrGate ou encore NotGate.

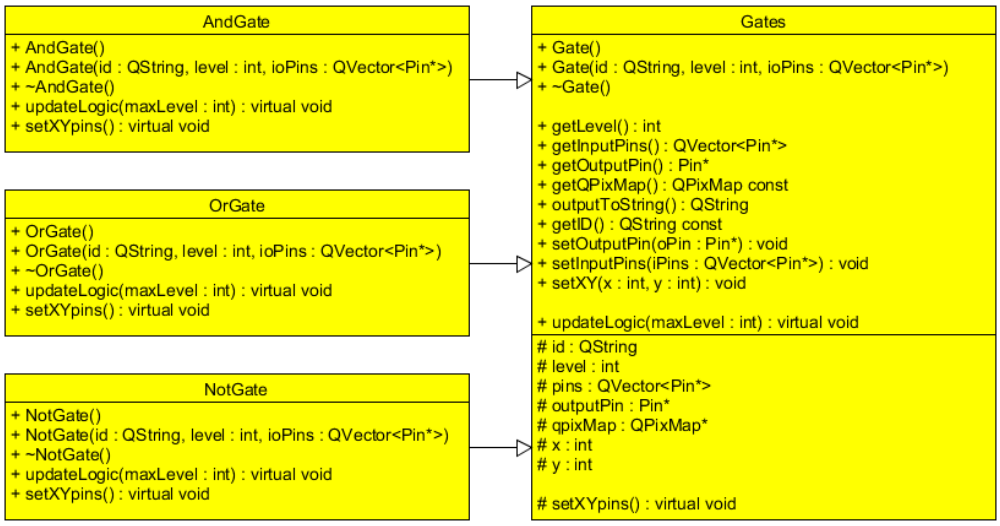

Figure 3 : héritage de la classe Gate

```
//create a logic gate
if(id.contains("AND"))
{
    AndGate* andGate = new AndGate(id, level, vPinsIO);
    vGates.push_back(andGate); //add to the global vector
}
else if(id.contains("OR"))
{
    OrGate* orGate = new OrGate(id, level, vPinsIO);
    vGates.push_back(orGate); //add to the global vector
}
else if(id.contains("NOT"))
{
    NotGate* notGate = new NotGate(id, level, vPinsIO);
    vGates.push_back(notGate); //add to the global vector
}
```

À chaque fin de récupération des attributs correspondant à une Gate, un pointeur est créé selon le type de Gate et stocké dans un QVector<Gate*> vGates, de la classe Data. La puissance du polymorphisme est ainsi utilisée.

Calcul des sorties selon le type de portes et l'état des entrées

La prochaine étape est de calculer les états de chaque Gate. Pour cela, on traite d'abord les Gates avec la variable level == 0. Ce sont elles qui ont les signaux d'entrées, soit « LOG_HIGH » soit « LOG_LOW ». Chaque Pin d'entrée voit ainsi son booléen « state » mis à jour. Ensuite, la Pin de sortie est connectée à la Pin correspondante, et sa valeur est calculée selon le type de porte et l'état des entrées grâce à la méthode virtual void *updateLogic*(int maxLevel) = 0;

Ensuite, les Gates intermédiaires sont traitées selon le même processus, mais au lieu de lire des « LOG_LOW » ou « LOG_HIGH », on récupère la Pin connectée grâce à son label et à la méthode getPinFromLabel(QString labelPinToFind). L'état de cette Pin connectée est copié sur la Pin d'entrée. Puis la suite se passe de la même manière pour la Pin de sortie.

Pour la dernière Gate du circuit, la seule différence est qu'elle ne contient pas de Pin connectée.

Affichage graphique

Chaque Gate contient une QPixMap, qui n'est rien d'autre qu'une image. Dans la classe IOView, deux QGraphicsView sont présentes. Elles sont utilisées soit pour afficher le code, via une QGraphicsScene et un QGraphicsTextItem, soit pour afficher la représentation graphique du circuit, via une QGraphicsScene et des QGraphicsPixmapItem.

Chaque Gate contient une coordonnée x et une coordonnée y. Elles sont définies précisément lors du dessin de la porte selon une fonction qui se charge de placer au mieux les différentes Gates. Enfin, chaque Pin contient elle aussi un x et un y, qui sont définis par rapport à ceux de la Gate hôte de la Pin. La méthode virtual void setXYpins() = 0 est implémentée par chaque type de Gate pour coller au mieux à l'image. Les liaisons entre les Gates sont ainsi optimisées.

La méthode void drawLineBetweenP1P2(int x1, int y1, int x2, int y2, QGraphicsScene &scn, QPen &pen) permet de créer des lignes étagées entre les Gates. Elle prend comme paramètre une référence à l'objet QGraphicsScene afin de le modifier, ainsi qu'un QPen.

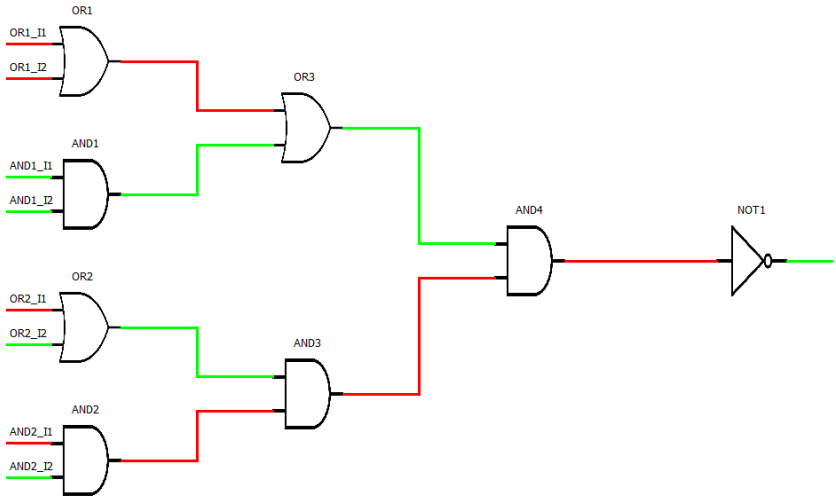

Figure 4 : rendu graphique

Interface globale :

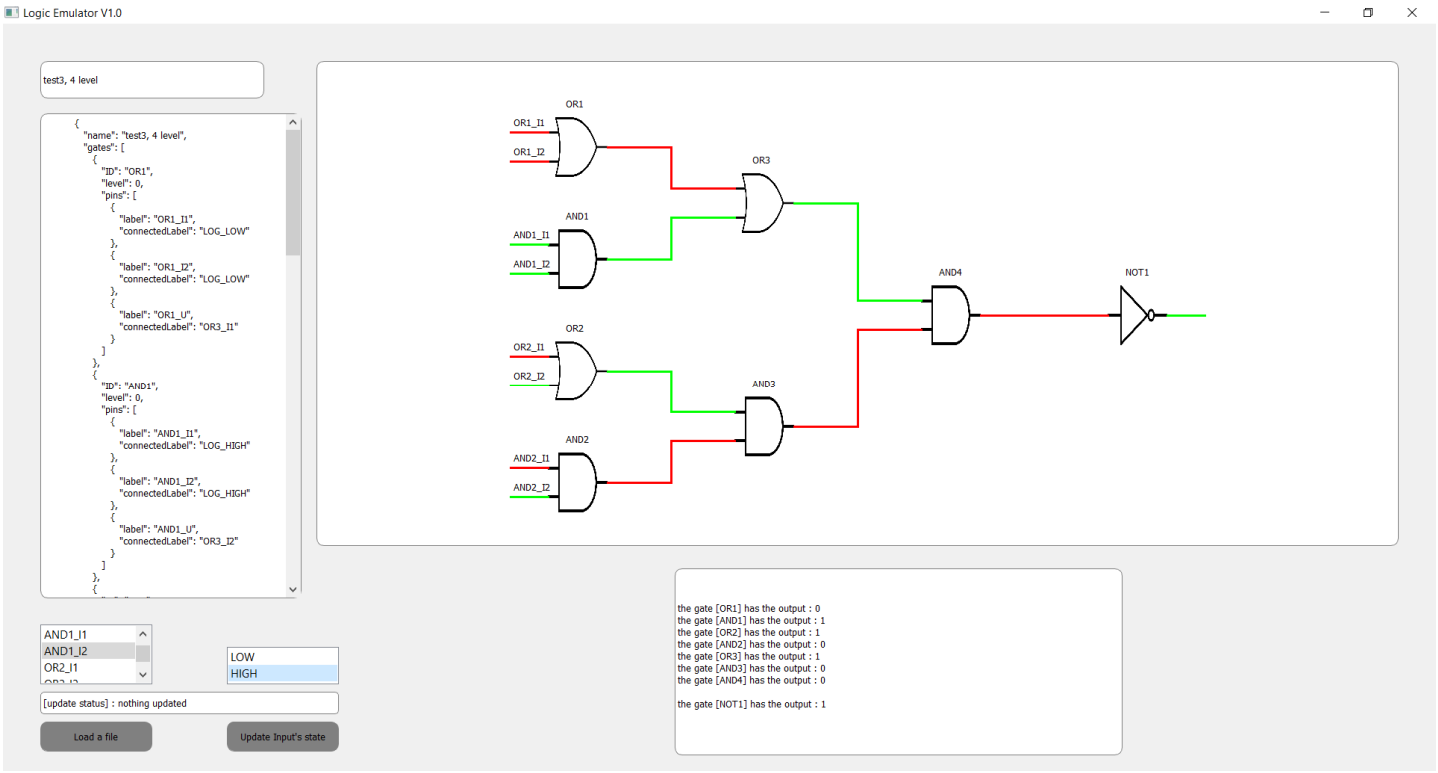

Figure 5 : interface utilisateur

Modification des entrées et validation des modifications

L'utilisateur a la possibilité de modifier l'état des entrées sans modifier le fichier Json lui-même. Pour cela, lorsque le fichier est chargé pour la première fois, une QStringList est remplie avec tous les labels des Pins d'entrées. Ils sont ensuite affichés à l'écran dans une QListWidget. En sélectionnant un de ces labels, puis en choisissant le nouvel état souhaité, le modèle logique est mis à jour et l'affichage est rafraîchi.

Une gestion d'erreur a été mise en place au cas où l'utilisateur appuie sur le bouton « Update Input's state » sans avoir préalablement choisi un label.

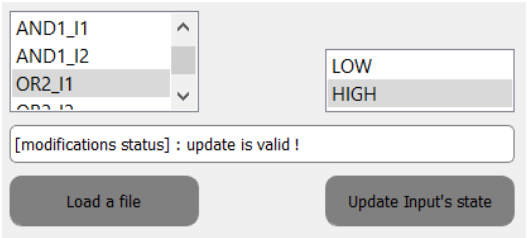

Figure 7 : éditeur d'entrées

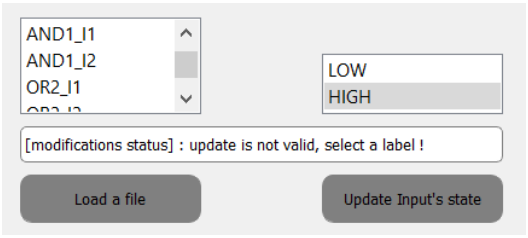

Figure 6 : appui sans sélection du label

Problèmes et solutions

Gestions des erreurs de fichiers

La source du problème est évidente : les fichiers sont écrits par un utilisateur, il se pourrait qu'ils contiennent des erreurs de syntaxe, faciles à détecter grâce aux classes QJson, mais aussi des erreurs dans la conception même du circuit logique, c'est-à-dire un non-respect du protocole défini. Cela

peut entrainer des NULL pointer exceptions, car certaines Pin* n'ont pas encore eu de mémoire allouée et on tente de les utiliser.

Pour contrer cette problématique, j'ai mis en place des états de gestion d'erreur en testant systématiquement les éléments sensibles, que ce soit lors de la conversion du fichier Json ou lors du calcul. Plusieurs fichiers Json volontairement incorrects ont été créés dans le seul but de valider la fiabilité.

Éditeur d'entrées

Mon idée initiale était de pouvoir éditer directement le code affiché à l'écran, de le traiter avec des QRegularExpression et de comparer avec le code initial. Mais je me suis vite rendu compte que ce n'était pas raisonnable de procéder comme suit : il y a un nombre d'erreur de modifications potentiellement énorme, et réussir à les traiter toutes de manière fine m'a semblé impossible, en tout cas dans le temps imparti. C'est pourquoi je me suis décidé pour la version expliquée ci-dessus avec les QListWidget de labels.

Affichage graphique

Les portes logiques peuvent avoir plus que 2 entrées. Si la partie logique et calcul de mon logiciel les accepte, la partie graphisme ne les prend pas en charge. Il m'aurait été trop long d'implémenté une fonction pour les dessiner.

Il n'y a pas de zoom automatique, je n'ai pas eu le temps d'adapter les images en fonction de leur nombre.

L'interface globale ne peut que peu évoluer d'une taille d'écran à une autre, toutes les dimensions sont hardcodées. À nouveau, il s'agit là d'un choix purement temporel, j'ai préféré me concentrer sur des fonctions plus intéressantes à implémenter que de faire du graphisme.

Résultats

V0.0

Totalement fonctionnelle.

V1.0

Éditeur d'entrées parfaitement fonctionnel, mais les blocs de composants Nand, Nor, … n'ont pas été implémentés.

V2.0

Aucune implémentation.

Conclusion

Créer une interface graphique prend énormément de temps si l'on veut un résultat convaincant. De plus, il est compliqué de créer des méthodes graphiques qui s'adaptent à plusieurs paramètres pouvant radicalement changer.

Lors de l'utilisation de pointeurs, il faut être prudents et tester systématiquement la validité de celui-ci, sous peine de se voir renvoyer une belle erreur lors de l'exécution.

Signature

Collonges, le 17 juin 2018 Gilles Mottiez

Annexes

- Protocole Json
- Diagrammes UML
- Projet Qt creator