Rapport final

Projet C - Simulateur Portes Logiques

Raphael CASENOVE
Adrien LUITOT
Arthur TALARMAIN

Grenoble INP-Phelma

2023 - 2024

Contents

1 Gestion d'équipe

1.1 Bilan organisationnel

Au début du projet, nous avions prédéfinis une organisation de manière à ce que chacun sache ce qu'il avait à faire. Malgré cela, cette organisation a dû évoluer au cours du projet. Adrien avait été assigné à la tâche du test pendant la conception des deux lexer/parser et au simulateur par la suite. Finalement, nous avons assigné à 100% les tests à Raphaël étant donné qu'il était moins à l'aise avec la gestion des structures de données sur la partie JSON. Cette modification nous a permis d'avoir une personne attitrée aux tests ce qui a amélioré notre efficacité. Adrien s'est donc occupé de la partie JSON à l'aide d'une librairie et de la conception du simulateur. Ensuite, Arthur est resté sur sa tâche assignée qui été la partie lexing/parsing des fichiers DOT. Pour finir, Raphaël s'est concentré sur les tests unitaires et globaux de nos programmes. Cette organisation a permis de développer nos différentes parties en les testants au fur et à mesure de manière efficace. Pour la répartition du travail au cours du projet nous avons attitré 10 à chacun des membres.

1.2 Bilan temporel

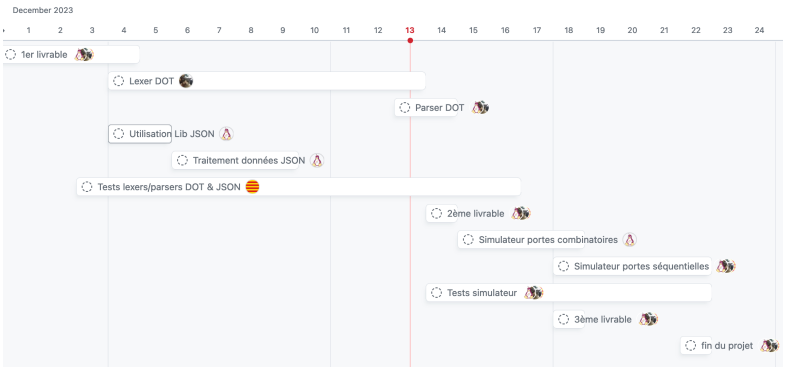

Figure 1: Gantt final du projet

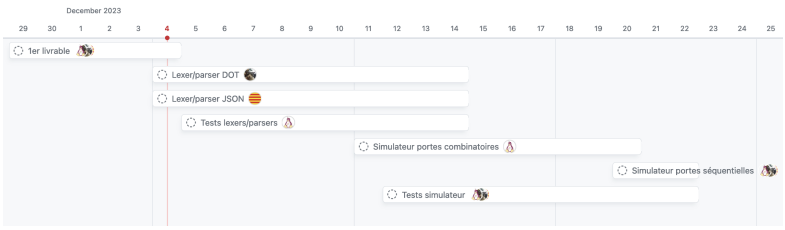

Figure 2: Gantt initial du projet

Au cours de notre projet, nous avons eu du retard sur certaines parties. On peut notamment citer la partie DOT qui a pris plus de temps que prévu à être finalisée. Ce

retard bien que non prévu à la base, nous a permis de livrer un lexer/parser de fichier DOT passant tous les tests fournis en renvoyant les erreurs de façons précise. Nous avons aussi pris ce temps pour que l'entrée du simulateur soit à 100% comme on l'attendait pour simplifier le traitement effectué par le simulateur. Ensuite, nous avons eu quelques difficultées à gérer les rebouclages et le séquentielle sur le simulateur à la fin du projet. Pour finir, nous avons dû réduire le nombre de test unitaire à la fin du projet pour que Raphaël est le temps de tester intégralement le simulateur.

2 DOT

2.1 Lexer

2.1.1 Architecture

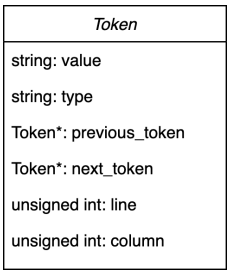

Figure 3: Objet Token

Pour lexer notre fichier nous allons créer une liste doublement chaînée d'objet de type token possédant les attributs visibles sur la figure 3. On a choisi cette architecture afin de pouvoir parcourir cette liste de token au niveau de la vérification de la syntaxe par le parser.

2.1.2 Implémentation

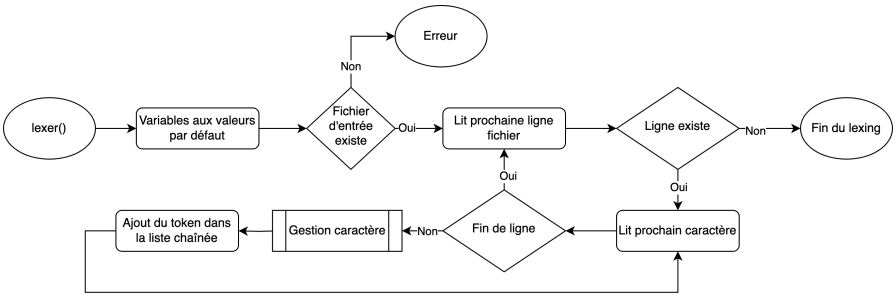

Figure 4: Schéma bloc du lexer

Pour lexer notre fichier nous allons comme représenté sur la figure 4 traiter notre fichier caractère par caractère sur chaque ligne du fichier. Selon le caractère traité, nous

effectuerons des tests afin de déterminer les différents tokens comme visible ci-dessous sur la figure 5.

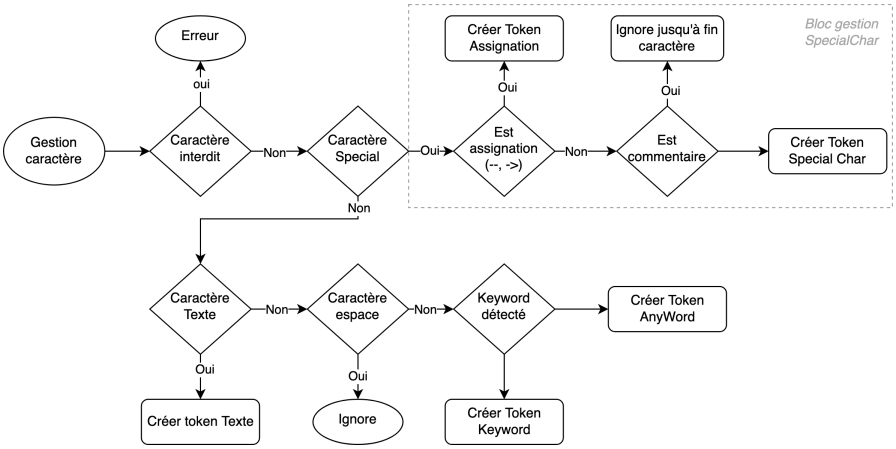

Figure 5: Schéma bloc du traitement des caratères

2.2 Parser

2.2.1 Architecture

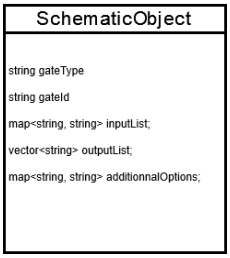

Figure 6: objet SchematicObject

En sortie de notre parser nous obtenons une map composée de pointeur de *Schematic Object* identifiables par une clé correspondant à leur ID.

2.2.2 Implémentation

Pour implémenter notre parser nous avons pris la décision de le réaliser à l'aide d'une FSM. Cette méthode nous a permis d'identifier tous les cas possibles précisément. Chaque état sera représenté par une location dans le fichier (valeur du token précédent). Ci-dessous, nous avons réalisé un schéma bloc simplifié de la FSM pour expliquer la différence d'étape de traitement entre les connexions et les instances.

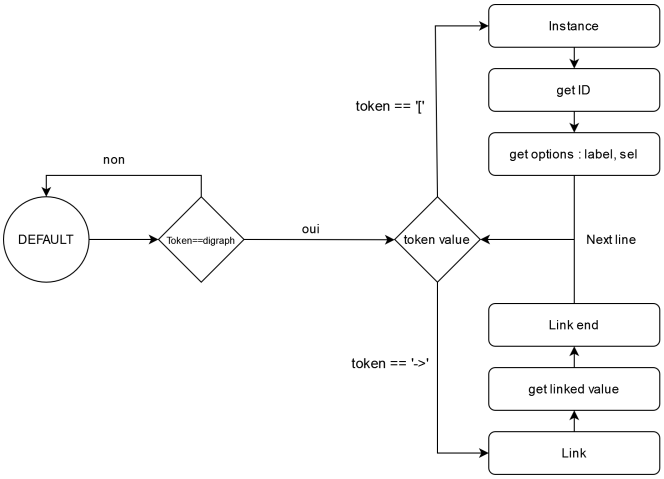

Figure 7: Schéma bloc FSM parser simplifié

3 JSON

3.0.1 Architecture

Pour le lexing et le parsing du JSON nous avons choisi d'utiliser une librairie existante. Après quelques recherches et tests, nous avons sélectionné la libraire RSJp-cpp (https://github.com/subh83/RSJp-cpp). Nous l'avons gardé notamment pour sa simplicité, son nom veut dire "Ridiculously Simple JSON Parser for C++". C'est aussi une librairie assez permissive, ce qui permet entre autre de gérer le JSON pas 100% normé de Wavedrom. Après le parsing, la librairie renvoie un tableau associatif multi-dimensionnel.

3.0.2 Implémentation

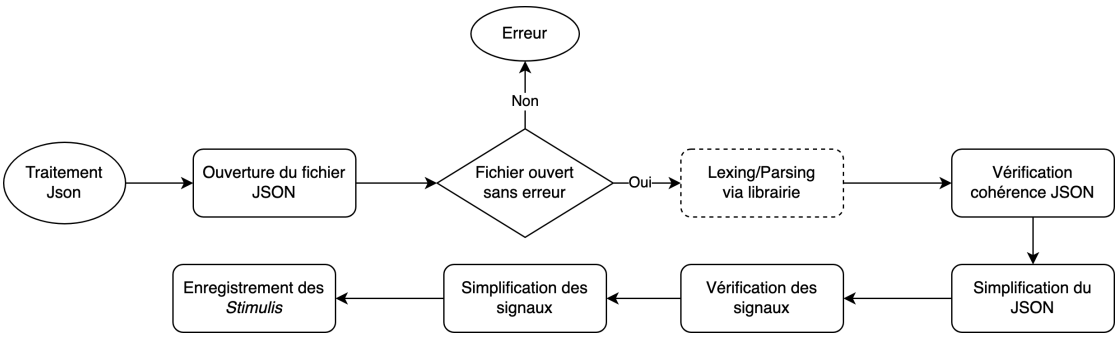

Figure 8: Schéma bloc du JSON

La partie JSON consiste donc essentiellement à du traitement de données. La figure 8 affiche un schéma bloc simplifié du déroulement du traitement du Json. Le but est de

supprimer tout le superflu inutile du visuel de Wavedrom et de préparer les signaux pour le bloc Simulateur.

4 Simulateur

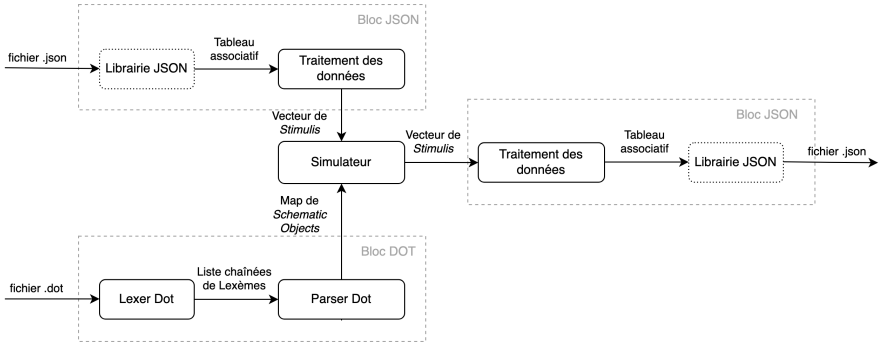

Figure 9: Schéma bloc général

Pour le simulateur nous avons conservé le système d'un objet principal avec différentes méthodes pour simplifier l'utilisation et la compréhension. Ensuite, cet objet utilise d'autres objets pour la structures des données.

Ici nous utilisons une classe mère 'Gate', qui découle vers les différentes portes (And, Not, Or...). La figure 10 montre son schéma de classe.

'input_names' est une variable static, qui permet de donner les inputs et/ou leur format pour chaque porte. Par exemple une porte Not, aura un seul input_name "i0". Une AND pourra avoir 2 inputs "i0" et "i1". Cet attribut sera surtout utiliser pour vérifier que les inputs venant du parser concorde avec les inputs attendues de la porte.
Nous utilisons aussi des wildcards : par exemple "i@" indiquera qu'il y aura autant de iX que d'inputs sur la porte, par exemple pour un AND3 ça reviendra à faire i0, i1, i2.

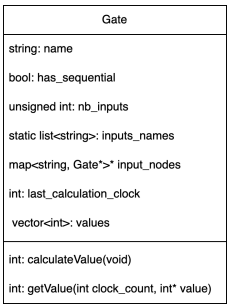

Figure 10: Schéma de classe de Gate (non-exhaustif)

La fonction 'calculateValue()' va calculer la sortie de la porte, en fonction des portes

précédentes. Cela fait une forme de récursivité qui va chercher depuis les outputs vers inputs, la valeur à chaque noeud concerné.

La fonction ‘getValue(...)‘ permet de récupérer la valeur à un numéro de clock précis. Si la valeur n’existe pas, la fonction ‘calculateValue()‘ est lancée.

5 Tests

L’étape de test est un processus indispensable à la validation d’un projet. Nous avons ainsi décidé qu’une personne serait entièrement dédiée à cette tâche. Il est important de savoir que le processus de test n’a pas beaucoup changé au cours du projet. On rappelle toutefois les étapes importantes :

- Définition structure de tests
- Confection de la base de données : fichiers true, correct et false
- Tests unitaires de toutes les fonctions, divisés en 2 parties :
  - Setup : où on renseigne les conditions nécessaires aux tests
  - Tests : réalisés avec l’appel d’une fonction “check”. Celle-ci récupère ce que renvoient les fonctions à tester grâce à la gestion d’erreur.
- Tests globaux avec affichage des résultats.
- Tests finaux : tests “structure” et “simulation” demandés

Cette tâche avançait en parallèle des 3 autres parties (dot, json et simulateur). Afin d’assurer un suivi de ces tests par tous les membres de l’équipe, les répertorier et connaître leur avancement. Nous avons créé un fichier excel répertoriant tous les tests (unitaires et globaux) des différentes parties du projet. Nous renseignons aussi leurs spécificités et leur état de validation (passed, failed ou undone).

6 Limites

6.1 Limites lexer/parser DOT

Nous avons essayé de produire un lexer/parser de fichier DOT performant, pouvant détecter toutes les erreurs de syntaxe. Les seules limites de notre parser ont été choisis car nous voulions que les vérifications soient faites au niveau du simulateur. Voici donc une liste des limites du parser :

- Signal d’entrée placé à la sortie d’une porte
- Signal de sortie en entrée d’une porte
- Nombre d’entrée d’une porte supérieur à son nombre d’entrée possible
- Signal d’entrée rentrant dans un autre signal d’entrée

6.2 Limites lexer/parser JSON

Utilisant une librairie, non dépendont de cette dernière. Elle est très complète il y a donc peu de limite. Mais sa principale qualité est également son principal défaut: la librairie est très permissive. Cette permissivité permet entre autres d'accepter le WaveJSON qui est moins normé que le JSON de base. Par exemple `wave: '10110'` est correct en WaveJSON alors que le format JSON correct serait `"wave": "10110"`.
Cette permissivité va aussi accepter un oubli de virgule par exemple et continuer de parser sans donner d'erreur. L'erreur sera donc détecter plus tard dans le traitement des données.

6.3 Limites simulateur

Le séquentiel sans rebouclage fonctionne (prise de la valeur précédente), mais pas avec rebouclage. Le rebouclage fonctionne en combinatoire.

Il n'y a aucune gestion de multi outputs, notamment pour un MUX42 par exemple (4 inputs, 2 outputs).

Pas de gestion implicite pour les portes avancées (MUX, Flip Flop...). Il faut forcément spécifier leurs inputs dans les crochets.

7 Pistes d'amélioration

7.1 JSON

Il serait possible de gérer les paramètres "data" avec un formatage particulier. Par exemple avec un signal `name: 'addr', data: 'x3, xa'`, dans 'data' le `x` correpondrait à une valeur hexadécimale, ce qui donnerait 4 signaux :

- `name: "addr[0]", wave: "10"`
- `name: "addr[1]", wave: "11"`
- `name: "addr[2]", wave: "00"`
- `name: "addr[3]", wave: "01"`

Particularité machine virtuelle Phelma Centos

Il faut activer une version récente de C++11 sur les anciennes machines virtuelles de Phelma sous Centos. Il faut télécharger le toolset 7 :

```
sudo yum update -y
sudo yum install -y devtoolset-7-gcc-c++
```

Puis compiler avec ce toolset :

```
make clean
scl enable devtoolset-7 'make'
```