UNIVERSITÉ DE
SHERBROOKE

Essai

Conception d’un simulateur Web de circuits logiques

Été 2020

Étudiant: Raphaël Roy

Référent : Pr. Frédéric Mailhot

1 Remerciements

La toute première partie de cet essai se doit d’être celle des remerciements, car sans les personnes que je vais citer, vous ne pourriez pas lire ces lignes.

Tout d’abord, une pensée pour ma professeure Robert-Inacio de l’Institut Supérieur de l’Électronique et du Numérique, mon école d’ingénieurs située à Toulon, dans le sud de la France. S’occupant des départs à l’étranger – ce qui n’est pas une mince affaire – elle a donné l’opportunité à plusieurs élèves, dont moi, de partir à l’Université de Sherbrooke. Je la remercie pour sa confiance en nous et ses efforts pour mettre cela en place.

Dans un second temps, il est impensable que je continue sans mentionner le rôle de mes parents. Ils ont su me soutenir, financièrement comme moralement, du début à la fin de cette dernière année d’études. J’ai la chance d’avoir des parents qui font le maximum pour me donner le meilleur, et cela inclus un double-diplôme et tout ce que cela implique.

Pour terminer, que serait ce projet sans le professeur Mailhot ? Rien. C’est lui qui me l’a proposé lors d’un rendez-vous, alors il est responsable de son lancement. De plus, Pr. Mailhot a été à mes côtés durant tout le déroulement, que ce soit pendant la préparation ou pendant le développement. Il a su me conseiller et me donner le chemin à suivre afin de venir à bout de ce projet inconnu et complexe. Alors, un grand merci à lui.

Le temps des remerciements est fini ! Place au concret.

Table des matières

Index des figures

Index des codes

2 Introduction

Dans le cadre de ma Maîtrise en Génie Électrique, j'ai passé la dernière session à développer un projet de fin d'études. Ce document est l'essai relatif à ce projet, dans lequel je vais tenter d'expliquer, avec le plus de précision possible, ce que j'ai réalisé.

Afin de décrire au mieux ce projet, l'essai est découpé en plusieurs sections que j'expose tout de suite. Tout d'abord, une mise en contexte pour voir quel est ce projet, dans quel cadre il s'inscrit, etc. Puis, une revue de littérature sur les projets similaires déjà existants ainsi que sur une lecture importante concernant le projet. Ensuite, une section imposante a le rôle de décrire ce qui a été réalisé, dans les moindres détails. Et pour terminer, une dernière partie contient des exemples d'utilisation afin de vérifier le bon fonctionnement du projet.

Notez que la troisième section, la description du projet, a une organisation spécifique. Chacune de ses parties présente une description, une liste de fonctions, de dépendances, et de ses propres parties. Ces propres parties ont aussi cette architecture, de sorte que le plan est récursif. Dès lors, vous pouvez choisir de rester à des parties générales, ou alors d'aller plus en profondeur. De plus, chaque section a une conclusion résumant ce qui a été vu.

Sans plus tarder, je vous invite à comprendre le contexte qui a permis la réalisation de ce projet.

3 Mise en contexte

Dans cette première partie je vais mettre en contexte le projet, en exposant comment et pourquoi je l'ai choisi, puis en quoi il consiste, et enfin dans quel projet plus global il s'inscrit.

Étant en dernière année de Master en Génie Électrique, de type cours sans stage, je dois effectuer un projet de fin d'études. Celui-ci s'effectue pendant la session d'été pour ma part, comme c'est ma dernière session à l'Université de Sherbrooke. Dès lors, j'ai dû trouver un professeur afin qu'il me propose un projet en lien avec mes études de développement logiciel. Le Pr Mailhot, responsable des étudiants étrangers, a répondu positivement à ma demande en me proposant de travailler sur une interface web permettant de simuler des circuits logiques. N'étant familier, ni avec le développement web, ni avec les circuits logiques, je trouvais l'occasion parfaite pour découvrir plus en profondeur ces domaines.

Plus précisément, le projet consiste à développer un interface qui va utiliser un programme pré-existant permettant de faire des simulations de circuits logiques, programme ayant été développé en été 2019, soit un an avant moi, par un autre étudiant nommé Luc Parent. Ainsi, l'interface web ne sera en effet qu'une interface à ce programme, donnant la possibilité aux utilisateurs de lancer indirectement le programme avec un simple accès internet.

L'interface ne devrait pas seulement lancer des simulations, mais afficher le circuit, les signaux d'entrée voulus par l'utilisateur, et une fois la simulation finie, les signaux de sortie voulus du circuits. Cela éviterait ainsi à la personne de devoir s'imaginer les courbes à partir des fichiers d'entrée et de sortie du programme de simulation.

En plus de représenter de façon graphique les éléments importants de la simulation, il y aurait aussi une partie serveur qui ferait toutes les opérations nécessaires, dont enregistrer les circuits, les simulations, etc.

Ce projet consiste donc à développer une interface web et un serveur, permettant de visualiser, enregistrer et lire des données nécessaires à la simulation des circuits logiques grâce à un programme pré-existant puis de voir les signaux de sortie obtenus voulus.

La partie précédente expliquait pourquoi et comment j'ai trouvé un projet de fin d'études, ainsi que le projet en soi. Mais ce projet s'inscrit dans un projet plus gros, que je vais décrire à présent. Ce projet global consiste à faire de la synthèse de circuits classiques, quantiques et hybrides. Nous sommes plusieurs à travailler sur ce projet, et nous formons le Groupe de Recherche en Optimisation Classique et Quantique (GROCQ).

Du côté synthèse de circuits classiques, on retrouve le simulateur de Luc Parent. Ce simulateur comprend un analyseur syntaxique. Celui-ci prend en entrée un fichier décrivant le circuit, et ressort le programme C++ correspondant au simulateur du circuit. L'exécutable issu de la compilation du code C++ a alors besoin d'un fichier contenant les signaux des entrées du circuit, et les fils en sortie du circuit qui nous intéressent. Pour résumer, le projet de cet étudiant permet de simuler des circuits classiques, en compilant un circuit, puis en lançant une simulation sur ce circuit.

Personnellement, je suis aussi sur la partie classique car, bien que l'interface web sera utilisée pour tout type de circuit, mon travail consiste à faire le lien entre le programme de Luc Parent, simulateur de circuits classiques, et l'interface.

D'autre part, le projet global a aussi pour but de simuler des circuits quantiques. C'est le travail d'un autre étudiant qui doit créer un simulateur semblable à celui de Luc Parent mais du côté quantique. Une des différences des circuits quantiques est que le codage de l'information se fait en qubits et non en bits. Avec cela viennent les caractéristiques quantiques qui sont très intéressantes d'une part car il est possible de représenter une quantité bien plus grande d'information, mais aussi difficiles à gérer notamment au niveau de la persistance. C'est pour cela que l'idée de circuits hybrides a émergé.

Dès lors, mon projet fait partie d'un plus grand destiné à simuler des circuits classiques, quantiques et hybrides, grâce à une interface web accessible de n'importe quel appareil disposant d'un navigateur et d'une connexion internet, et un serveur permettant la manipulation, la sauvegarde et l'accès à des données intéressantes comme les circuits ou les simulations.

Dans la partie suivante nous allons voir quels simulateurs de circuits existent déjà, et passer en revue la lecture d'un livre particulièrement utile.

4 Revue de littérature

Maintenant que le contexte du projet est connu, nous pouvons faire l'état de l'art, soit rechercher et lister les projets similaires déjà utilisables. Je vais concentrer cette étude sur les projet open-sources et gratuits.

Commençons avec le logiciel CedarLogic, qui est un logiciel libre – distribué sous la licence GPLv3 – pour Windows développé au départ par l'université de Cedarville. Il a pour but la création et la simulation de circuits logiques, de façon interactive, en choisissant et reliant des composants électroniques de plus ou moins haut niveau.

La version stable de CedarLogic est téléchargeable sur SourceForge et le développement se poursuit depuis 2020 sur GitHub.

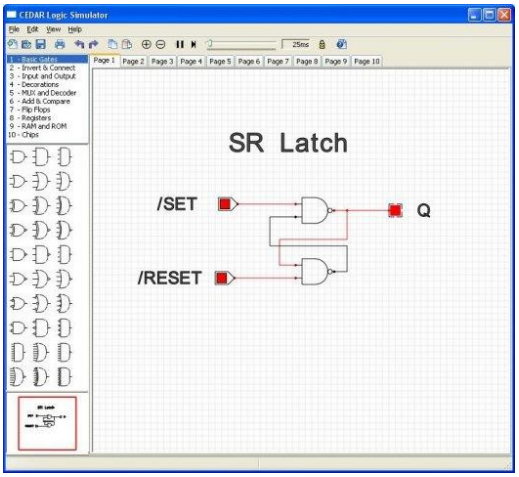

Figure 1: Capture du logiciel CEDAR LS

Sites de référence:

https://sourceforge.net/projects/cedarlogic

https://github.com/CedarvilleCS/CedarLogic

Dans la même lignée que CedarLogic, le logiciel Logism est aussi libre – distribué sous la licence GPLv2 – et utilisable sous Windows mais aussi sur Linux et Mac-OS X. Pareillement, il se présente sous forme d'interface graphique afin que l'utilisateur sélectionne les composants logiques et les relie afin d'en faire un circuit et le simuler. Il est aussi possible de créer des composants à partir de tables de vérité.

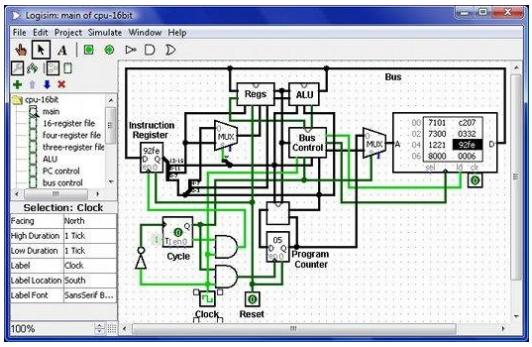

Figure 2: Capture du logiciel Logism

La dernière version de ce logiciel a été publiée en 2013.

Site de référence :

https://sourceforge.net/projects/circuit/

Un autre logiciel libre, sous la licence GPLv3, disponible sous Windows et nommé Logic Gate Simulator, est basé sur la découverte des portes logiques grâce à une interface graphique. On peut sélectionner, comme pour les précédents logiciels, des portes logiques et les relier entre elles afin d'observer les signaux résultants. La dernière version date de 2014.

Site de référence:

https://sourceforge.net/projects/gatesim

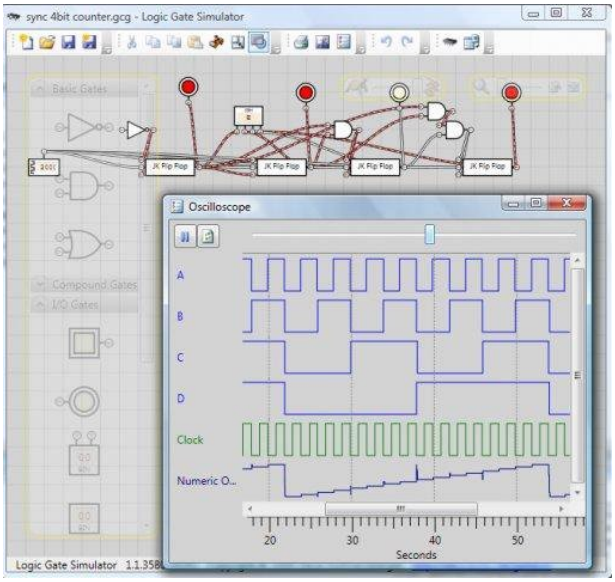

Figure 3: Capture du logiciel Logic Gate Simulator

Continuons la liste avec le logiciel libre Ngspice, visant à faire de la simulation de circuits électriques et électroniques. Il est utilisable sur Windows, BSD, Mac OS, Solaris et Linux. Toujours sous forme d'interface graphique, l'utilisateur peut choisir une variété de composants logiques et électroniques, et une variété d'analyses et de simulations. Contrairement aux autres logiciels, on ne vient pas créer un circuit en plaçant les composants mais en ligne de commande ou sous forme de fichier comme on peut voir dans la figure 4. C'est un vieux projet d'une trentaine d'années qui est toujours en activité.

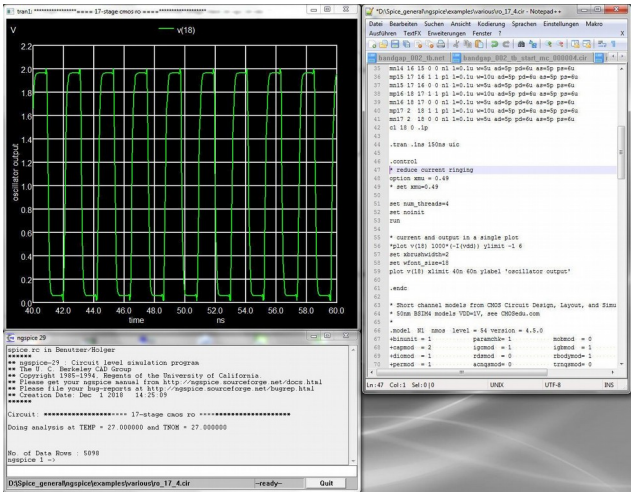

Figure 4: Capture du logiciel Ngspice

Site de référence : http://ngspice.sourceforge.net/

Pour finir sur les simulateurs de circuits classiques déjà existants, CircuitVerse est un simulateur accessible via un navigateur. Cette interface web permet d'élaborer des circuits logiques plus ou moins complexes de façon graphique, comme la plupart des logiciels vus précédemment. Le code

de cette interface web est open-source – licence MIT – et possède une bonne documentation ainsi que de nombreux exemples.

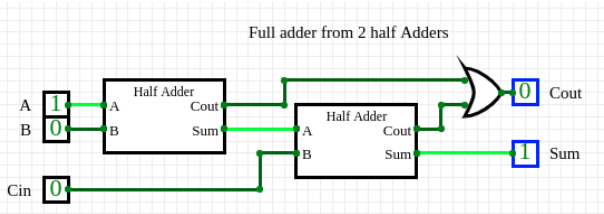

Figure 5: Exemple de circuit fait avec le simulateur CircuitVerse

Site de référence : https://circuitverse.org/

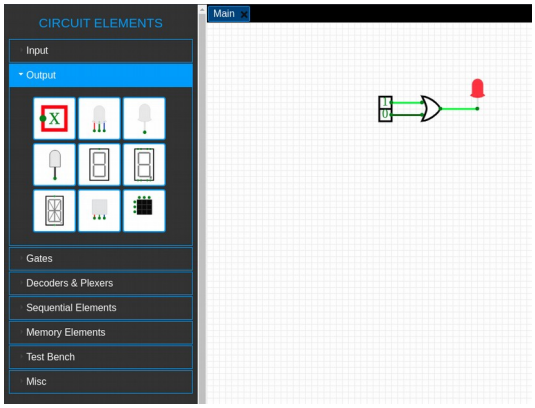

Figure 6: Capture de l'interface web CircuitVerse

Enfin, il est intéressant de présenter un simulateur de circuits quantiques, même si je ne m'occupe pas personnellement de cette partie du projet. J'ai donc trouvé un simulateur nommé Quirk qui est sous forme d'interface web, accessible via un navigateur. C'est un projet open-source distribué sous la licence Apache-2.0. Comme les simulateurs précédents, on peut placer des composants afin de créer un circuit, à la différence près que ces composants sont quantiques et non classiques. Les circuits simulés peuvent utiliser jusqu'à 16 qubits. La dernière activité de développement en date est de 2019.

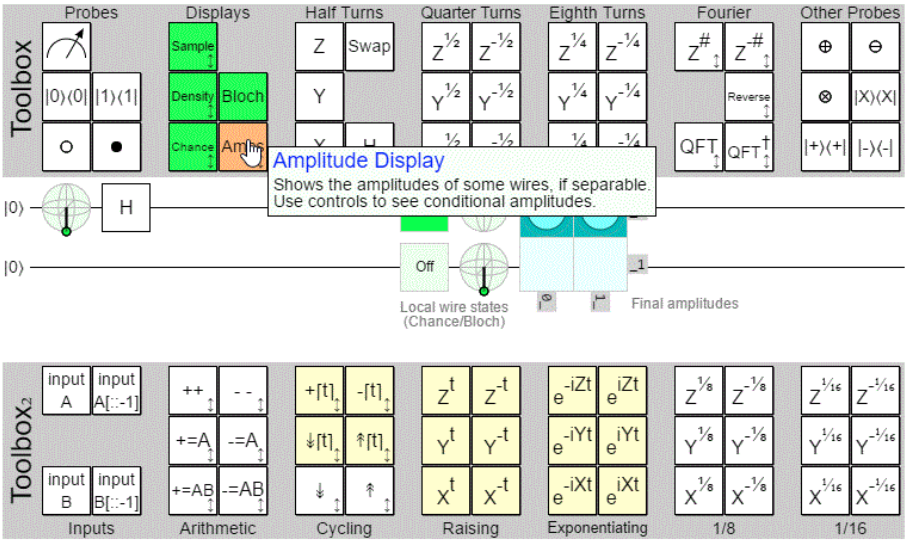

Figure 7: Capture du simulateur Quirk

Site de référence : https://github.com/Strilanc/Quirk

Nous avons vu de nombreux simulateurs de circuits logiques classiques plus ou moins complexes, allant du simple logiciel permettant la découverte et l'étude de la logique jusqu'à celui visant les professionnels. La quasi totalité permet à l'utilisateur de placer et relier les composants de façon graphique, et un seul demande la description de circuits par programmation. Or, tous présentent une interface graphique, que ce soit les divers logiciels ou les interfaces web. Pour ce qui est des simulateurs de circuits logiques quantiques, je n'en ai trouvé qu'un seul. Et aucun permettant la synthèse de circuits hybrides.

Dès lors, ce sont les interfaces web CircuitVerse et Quirk qui se rapprochent le plus du projet du GROCQ. Cependant, aucun ne combine les fonctionnalités de simulation classique et quantique en une seule interface web. Le projet du GROCQ reste donc pertinent et novateur.

Comme nous avons fait l'état de l'art dans la partie précédente, nous pouvons nous attarder sur ce qui existe en terme de technique concernant le design d'interfaces. Car, afin de créer une interface agréable et fonctionnelle il peut y avoir certains conseils ou règles à suivre. Pour cela, le Pr. Mailhot m'a proposé le livre « *The Design Of Everyday Things* » écrit par Don Norman. Cet ouvrage fournit la théorie ainsi que des exemples pratiques nous permettant de comprendre ses principes de design. En effet, ses exemples sont variés afin de s'appliquer à tout type de design : matériel, logiciel, etc.

J'ai pu extraire ce qui me semblait le plus important de ce livre, soit :

- la philosophie du design

  C'est l'état d'esprit dans lequel doit être le designer. Par exemple, on ne doit jamais blâmer l'utilisateur s'il trouve l'interface complexe, mais l'interface en soi, qu'il faut donc modifier. Un autre point est qu'il faut fournir un manuel d'utilisation compréhensible et détaillé auquel on peut se référer à tout moment. Je terminerais avec le fait d'avoir une pensée positive, ce qui veut dire accepter la critique afin d'améliorer le produit – dans mon cas l'interface – au maximum.

- les étapes de l'action et le lien avec la conscience

  Le designer doit tenir compte des étapes que l'on retrouve à chaque action de l'utilisateur. Il y en a 6, en plus du but, qui sont :

  1) planification
  2) séquençage
  3) exécution
  4) perception
  5) interprétation
  6) comparaison

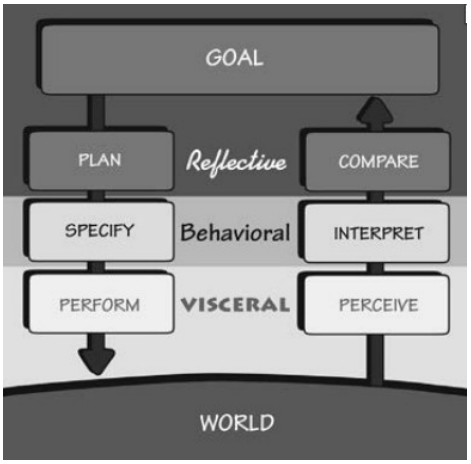

Figure 8: Étapes d'une action et lien avec les niveaux de conscience

Et comme nous pouvons le voir dans la figure ci-contre, la personne se trouve dans un état de conscience différent selon l'étape. Le but du designer est de prendre en compte ces niveaux et ce qu'ils impliquent sur le plan émotionnel afin d'améliorer l'expérience de l'utilisateur.

- les principes fondamentaux du design

Le designer doit enfin faire attention à un certain nombre de principes qui garantissent un meilleur design. On retrouve celui de la rétroaction qui vise à signaler l'utilisateur quand il effectue une action. Ou encore celui de *mapping* qui consiste à placer un composant au bon endroit. Le dernier exemple de principe serait celui de contrainte, qui veut restreindre les possibilités d'actions de la personne.

Ces points sont plus explicites dans mon rapport du cours sur les interfaces personne-système mais ils permettent de voir quelles choses il faudra que je garde en tête pendant le projet.

Ainsi, nous avons vu dans cette revue de littérature de nombreux programmes permettant la simulation de circuits logiques classiques plus ou moins complexes, et un seul voué à simuler des circuits quantiques. Il n'existe cependant, à ma connaissance, aucun simulateur de circuits hybrides. J'ai aussi évoqué un livre sur le design utile au développement de l'interface web.

5 Description du projet

Maintenant que la mise en contexte et la revue de littérature sont faites, nous allons pouvoir nous pencher sur le projet en soi. Quels outils ont été utilisés ? Quelles sont ses différentes parties ? Voici les questions auxquelles je vais répondre dans cette section. Après l'avoir lu, vous aurez compris son fonctionnement dans les moindres détails.

Notez que cette description du projet est organisée de façon récursive. C'est-à-dire que j'expose les outils et les parties du projet global, mais chaque partie contient des outils et des parties spécifiques à celle-ci, et ainsi de suite. Dès lors, il est possible de comprendre le projet de façon globale en restant à des niveaux supérieurs, mais aussi de s'intéresser à des détails particuliers en allant plus profondément dans le plan.

Les précisions étant faites, nous pouvons y aller.

5.1 Dépendances globales

Ici je vais lister les outils qui conditionnent l'utilisation du projet dans sa globalité. Cela veut dire que je n'aborderai pas ici les dépendances spécifiques aux parties.

Nous allons parler de Yarn, de Nx, de TypeScript, de Babel et de Jest. Chacune de ces technologies sera détaillée ci-dessous, par une description et une liste de ses fonctions utiles au projet, alors c'est parti.

5.1.1 Yarn

Yarn est un gestionnaire de paquets distribué sous licence BSD, dont on peut voir le code source ici, et qui permet d'installer des programmes globalement sur une machine ou spécifique à un projet grâce à un fichier local (package.json).

A chaque fois que vous choisissez d'installer ou d'enlever un paquet à un projet, Yarn vient modifier le fichier package.json, donc il ne faut pas y toucher. Notez que Yarn ressemble beaucoup au gestionnaire de paquets npm, et qu'il est possible d'utiliser npm à la place.

Ainsi, le projet contient un fichier package.json listant tous les paquets utilisés dans le projet, nécessaires seulement au développement ou pas. Pour pouvoir travailler sur le projet il faut donc les installer avec la commande `yarn install`. Pour plus d'informations sur l'utilisation de Yarn je vous renvoie à la page en question.

Mais à quoi sert Yarn par rapport au projet ? Eh bien c'est ce gestionnaire de paquet qui a permis d'installer les paquets indispensables au fonctionnement et au développement du projet, comme Nx, TypeScript, React, Nest, etc. Et certains paquets permettent de créer des projets complets déjà configurés, voir la commande `yarn create <starter-kit-package>`.

De plus, Yarn peut lancer des scripts. Ces scripts peuvent être définis manuellement ou automatiquement dans package.json. Dans celui du projet par exemple, la plupart ont été générés par Nx. Il suffit d'exécuter `yarn <nom-du-script>` pour lancer la commande.

Plus de détails seront apportés dans les sections suivantes mais pour l'instant cette brève présentation permet de bien cerner le pourquoi du comment de Yarn. Intéressons-nous maintenant à Nx, qui est la pierre angulaire du projet.

5.1.2 Nx

Qu'est-ce que Nx ? Pour résumer, c'est un outil de développement de gros projets web. Un projet web complet contient généralement une partie visible sur un navigateur, l'interface, et une partie gérant les requêtes envoyées par l'interface et l'accès aux données, le serveur. Quand j'ai commencé le projet j'ai créé ces deux parties séparément, mais à partir d'un moment je me suis rendu compte qu'il serait plus simple et logique d'avoir un projet général contenant ces deux applications. Je voulais aussi partager du code, comme des structures de données, entre les deux parties. Et bien c'est ce que permet Nx, entre autre. Pour avoir plus d'informations voici le code source et le site officiel.

Comme je l'ai dit dans la section précédente, il est possible d'installer et d'utiliser Nx avec le gestionnaire de paquets Yarn. Les développeurs de Nx ont mis à disposition un « starter-kit-package » afin de créer un projet Nx facilement, et je n'ai pas hésité à m'en servir en exécutant la commande suivante : `yarn create nx-workspace`. Il propose de créer un projet Nx vide, ou pas. Quelque soit le choix, le paquet Nx peut être utilisé à tout moment afin de créer une – nouvelle - application ou une librairie propre au projet.

Reste à savoir quels types d'application Nx propose de créer. Le choix n'est pas très large, mais les outils qui m'intéressaient étaient présents. Je veux parler ici de Nest, React et Storybook. Ces outils sont détaillés dans la partie serveur pour Nest, et interface pour les autres.

Dès lors, Nx est l'outil qui m'a permis de mettre en place mon projet, complètement configuré avec les scripts utiles au déploiement et au test des applications. Les applications créées par défaut contenaient tellement de fonctionnalités que j'ai du en supprimer certaines, notamment le rendu des tests de façon graphique de l'outil Cypress.

Il est temps de passer au code en soi, et pour cela je me dois de présenter TypeScript.

5.1.3 TypeScript

Les deux applications, que ce soit côté interface comme côté serveur, ont été développées en TypeScript. Mais qu'est-ce donc ? Comme son nom l'indique, c'est du JavaScript typé. Si vous êtes familier avec le JavaScript, vous saurez qu'une variable se définit ainsi :

```
let i ;
```

Mais quand on passe en TypeScript, on peu renseigner le type de cette variable ce qui limite son utilisation comme dans la plupart des langages de programmation :

```
let i : number ;
```

Évidemment ce n'est pas restreint aux types par défaut (number, boolean, string, etc.) car on peut créer d'autres types ou des interfaces et dire que cette fonction attend en paramètre une variable de tel type, retourne une variable de tel type, etc.

Comme pour les autres outils, c'est un paquet installé via Yarn. Bien qu'il ait été installé par le « starter-kit-package » de Nx. Aussi open-source – pourtant développé par Microsoft – dont le code source est ici et le site officiel est là.

La question qu'il est légitime de se poser est : pourquoi s'en servir ? Je répondrais par deux arguments. Le premier est que cela facilite grandement le développement. On sait quel type de variable attend cette fonction, qu'est-ce qu'elle va nous retourner, est-ce que cela correspond à ce que l'on veut, etc.

Et deuxièmement, cela réduit les possibilités et donc les risques, car s'il y a un problème de type, l'éditeur de code (VS Code pour ma part), ou le compilateur, le signaleront.

TypeScript est un framework très utile, qu'en est-il de Babel ?

5.1.4 Babel

Qu'est-ce donc encore que cet outil ? Une application de langues ? Une tour ? Rien de tout ça, c'est un compilateur de JavaScript dont le site est accessible via ce lien et le code source visible ici.

C'est lui qui va être en charge de la compilation du code, qu'il soit TypeScript ou de syntaxe très récente, en un code JavaScript compréhensible par tous les navigateurs.

Comme pour bon nombre d'outils, le paquet Babel et ses fichiers de configuration étaient déjà présents suite à la création par Nx d'une application ce qui facilite grandement les choses.

Après cette petite explication, néanmoins nécessaire, il est temps de passer à la dépendance Jest.

5.1.5 Jest

Les dépendances principales du projet global ont été exposées précédemment, mais il manque un dernier outil à développer, c'est Jest. Jest est un framework sous licence MIT pour faire du test unitaire de JavaScript. Il prend en charge de nombreux outils récents comme Babel, TypeScript, Node, React et d'autres, ce qui nous arrange bien.

Et grâce à Nx le paquet est automatiquement installé et les fichiers de configuration remplis. Il suffit ensuite d'exécuter le script de test avec `yarn test` pour exécuter tous les fichiers de test (ils ont l'extension .spec.ts) ou les tests d'une seule application avec `yarn test server` par exemple.

Cet outil est utile au projet car il permet de vérifier le bon fonctionnement des fonctions que l'on code ou que l'on a codé il y a quelques temps. Pour certaines fonctionnalités il m'était très utile de pouvoir les tester séparément du projet complet, de les isoler et de vérifier qu'elles étaient opérationnelles. De plus, je pouvais relancer les tests à tout moment – voire faire de l'intégration continue – pour être sûr que tout continue à être fonctionnel pendant le développement. Cela sert ainsi à réduire les risques et à augmenter la maintenabilité du projet pour ceux qui le reprendront.

J'ai exposé dans cette partie les dépendances globales du projet, sachant qu'il y a aussi des outils spécifiques aux parties que je vais décrire plus tard. Nous avons vu que Yarn était la dépendance la plus importante qui permettait d'installer tous les paquets nécessaires, que Nx était un paquet très utile pour la création et la gestion de gros projets contenant plusieurs applications. Puis, TypeScript, rendant possible la programmation typée en JavaScript, associé à Babel afin de compiler le tout en un code interprétable par tous les navigateurs, et enfin Jest, pour tester unitairement les fonctionnalités que l'on veut. Cette présentation étant faite, il est temps de s'attaquer aux grandes parties du projet.

5.2 Parties du projet

Les parties du projet sont présentées et expliquées dans cette partie. Un schéma aura pour rôle de résumer de façon simple le projet global et mettre en évidence le lien entre ces parties. Puis les deux grandes parties, qui sont l'interface et le serveur, seront détaillées.

5.2.1 Schéma du projet

Vous pourrez voir ci-après un schéma présentant le fonctionnement global du projet, avec une partie serveur et une partie interface utilisateur. Chaque utilisateur peut accéder à l'interface via un navigateur (Firefox, Brave, etc.) et effectuer un certain nombre d'actions en lien avec la simulation

de circuits logiques. La plupart de ces actions dépendent du serveur commun qui se charge de stocker les fichiers nécessaires et simuler ce que veut l'utilisateur. Chaque utilisateur fait donc des requêtes au serveur via l'interface, le serveur renvoie ce qui est demandé et l'interface se charge de l'afficher.

Dès lors, nous pouvons réduire le projet à deux applications : l'interface et le serveur.

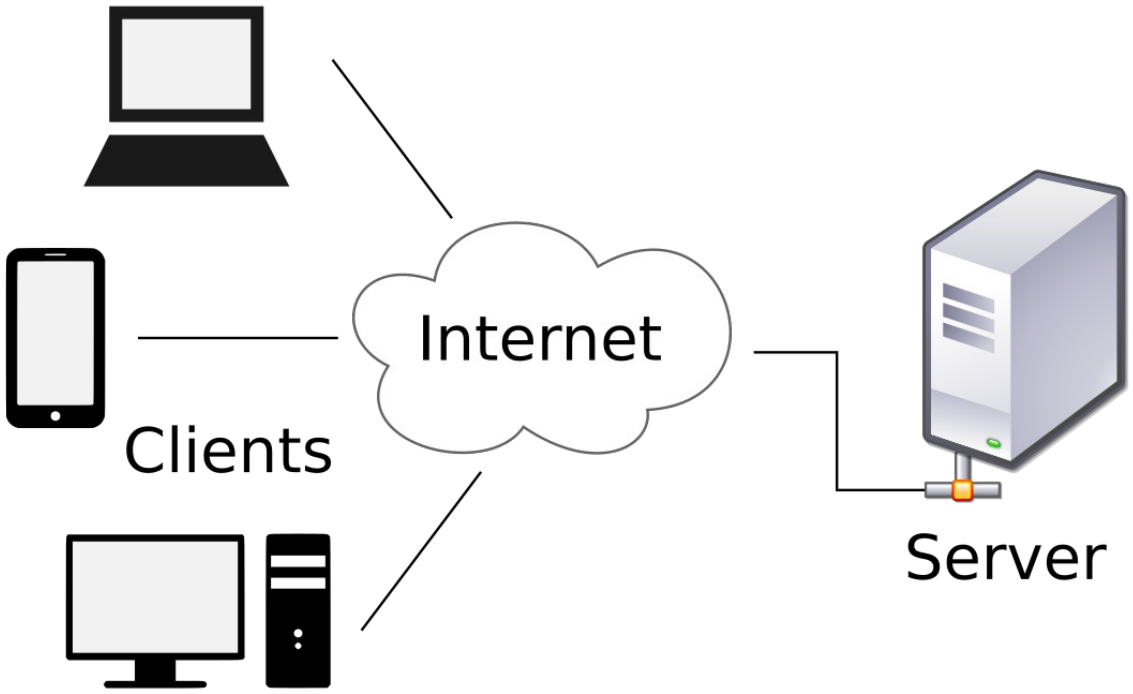

Figure 9: Schéma global du projet

Cette représentation schématique du projet global permet d'introduire les deux grosses parties qui vont suivre, et nous pouvons, dès maintenant, développer en détail l'interface.

5.2.2 Interface

Voici la section qui décrit l'application qui est chargée par un quelconque navigateur : l'interface. Une brève description sera nécessaire, ainsi que la liste de ses fonctions générales. Ensuite, nous verrons les dépendances spécifiques à l'interface, et pour finir, les différentes parties visuelles de celle-ci.

5.2.2.1 Description de l’interface

L’interface est une page web chargée par le navigateur lorsqu’il visite l’adresse exposant ce site. Par exemple quand on visite https://www.youtube.com/ notre navigateur charge l’interface exposée sur ce site.

Cette page web n’est pas statique comme une simple page HTML, en effet elle est largement programmée en JavaScript ce qui permet la gestion d’événements qui vont venir modifier le contenu de la page ou envoyer des requêtes au serveur.

Ainsi, lorsque l’utilisateur clique sur un bouton cela peut faire apparaître du texte, ou l’enlever, ou afficher un graphique, etc. Tout ça sans que la page soit rechargée. Il faut noter que le programme JavaScript – l’interface – est exécuté sur la machine du client, donc il ne faut pas qu’il soit trop gourmand en ressources.

Pour exposer l’interface localement, vous pouvez exécuter dans un terminal, en étant à la base du projet, `yarn nx serve interface` ou `nx serve interface` si Nx est installé globalement sur votre machine. Cela va automatiquement compiler le code et l’exposer sur http://localhost:4200 normalement. Il ne va pas sans dire que l’interface dépend du serveur donc il doit aussi être exposé sinon vous ne pouvez pas faire grand-chose.

Nous pouvons maintenant passer aux fonctions de l’interface.

5.2.2.2 Fonctions de l’interface

Nous venons de voir ce qu’est l’interface, mais à quoi sert-elle ? Comme son nom l’indique, c’est une interface entre l’utilisateur et le serveur. L’utilisateur peut demander au serveur, via l’interface, un certain nombre d’actions que nous verrons dans la partie serveur.

La page web, elle, peut simplement envoyer des requêtes au serveur et afficher les réponses en fonction de ce qui était demandé. Par exemple, l’interface permet de gérer les fichiers de circuits et de simulations stockés, d’afficher une simulation sous forme de graphique, de se déplacer sur celle-ci, etc. Mais il ne faut pas oublier que tout cela est intimement lié au serveur, sans ses réponses aux requêtes l’interface ne sert à rien.

Les fonctionnalités seront plus explicites lors de la description de chaque partie de l’interface. Les dépendances que nous allons traiter juste après jouent aussi leur rôle dans la compréhension de l’interface.

5.2.2.3 Dépendances de l'interface

Nous venons de voir que la fonction principale de l'interface était justement de faire l'intermédiaire entre l'utilisateur et le serveur. Cependant nous n'avons pas encore vu quels outils rendent – ou ont aidé à rendre – cela possible, et c'est ce que nous allons voir maintenant. Voici les dépendances à explorer: React, Storybook, Material-UI, Axios et WaveDrom.

5.2.2.3.1 React

React est vraiment la pierre angulaire de l'interface. C'est une librairie JavaScript open-source développée par Facebook qui se veut orientée composant. C'est-à-dire que l'on va développer des composants qui intégreront d'autres composants jusqu'à intégrer le composant principal qui est l'application.

React permet de gérer des sortes d'éléments HTML, appelé JSX. Par exemple, il est possible de déclarer une variable comme ceci :

```
const paragraph = <p>Ceci est un paragraphe</p>
```

Code 1: Déclaration d'une variable JSX

L'exemple précédent est vraiment basique, cela fonctionne avec toute sorte de balise HTML, et surtout avec nos propres balises qui sont en fait nos composants. Car je pourrais transformer la variable précédente en un composant très simple :

```
const Paragraph = () => {
        return <p>Ceci est un paragraphe</p>
}
```

Code 2: Déclaration d'un composant React basique

Je pourrais ensuite exporter puis importer et intégrer ce composant dans un autre de cette façon (même si l'on verrait deux fois le même paragraphe) comme vous pouvez le constater dans l'exemple de code suivant.

```
const Document = () => {
        return (
                <div>
                        <Paragraph/>
                        <Paragraph/>
                </div>
        )
}
```

Code 3: Déclaration d'un composant contenant d'autres composants

Ceci est vraiment la base, ensuite on peut faire intervenir des variables qui vont pouvoir changer en fonction des évènements (si l'on crée une fonction associée au clic d'un bouton par exemple) et, pourquoi pas, venir modifier le contenu du paragraphe.

Notez que les exemples ci-dessus utilisent la syntaxe la plus récente de JavaScript – celle utilisée dans le projet – pour définir des fonctions. Car ici ce sont des variables constantes qui sont en fait des fonctions. Dans les documentations officielles ce sont encore, à l'heure où j'écris cet essai, des classes qui sont utilisées largement mais cela devrait évoluer.

React permet de développer du JavaScript de façon structurée avec des composants, semblables à des objets qui ont leur propre mécanisme isolé, mais auxquels on peut passer des variables à modifier si besoin. Ceci m'amène donc à présenter un outil de développement appelé Storybook.

5.2.2.3.2 Storybook

Nous venons de voir que React organisait le code sous forme de composants, Storybook est un outils open-source qui va nous aider à développer ces composants. En effet, au lieu de visualiser les composants directement dans l'interface, nous pouvons nous créer des « stories » pour un ou plusieurs composants et les visualiser indépendamment.

Pour reprendre l'exemple du composant Paragraphe, je pourrais le voir seul sur une page vierge plutôt que l'intégrer au composant Document qui lui aussi devrait être intégré par le composant général. Aussi, si ce composant prenait des arguments en paramètre, je pourrais me créer des « stories » de ce composant pour tester des arguments différents. C'est un peu une façon de tester unitairement et visuellement des composants, car nous pourrions aussi faire de vrais tests unitaires avec Jest.

Nx offre la possibilité d'utiliser Storybook sur une libraire, donc après avoir créé la librairie nommée « ui » avec `nx g @nrwl/react:lib ui`, il ne restait plus qu'à générer sa configuration Storybook avec `nx g @nrwl/react:storybook-configuration ui`. Puis, pour déployer les

« stories » localement, la commande `nx run ui:storybook` compile ce qu'il faut et donne accès à une interface.

Bref, cet outil est vraiment le bienvenu pour tester des composants React, composants qui peuvent en intégrer d'autres déjà tout faits comme ceux de Material-UI.

5.2.2.3.3 Material-UI

Un autre dépendance importante est celle de la librairie Material-UI distribuée sous licence MIT. Elle propose des composants React variés, allant des listes aux boutons en passant par les menus. Cette librairie est le résultat de la combinaison entre React et Material Design, autre librairie proposant des composants pour toute sorte d'application. Voici le site officiel et le code source de Material-UI.

Les composants de cette librairie présentent de nombreuses caractéristiques comme des effets de clic, de transition, un choix de formes et de couleur, etc. C'est vraiment un superbe outil pour faire des interfaces utilisateur agréables et intuitives, tout en pouvant personnaliser le comportement et le style des composants. C'est d'ailleurs la librairie qui m'a permis de gérer au mieux les exigences découlant de la lecture du livre de Don Norman sur le design cité dans la section revue de littérature.

Tous les outils vus précédemment sont indispensables, mais pas autant que Axios.

5.2.2.3.4 Axios

Les dépendances que nous venons de détailler sont importantes, mais elles ne rendent pas possible la communication avec le serveur. C'est Axios qui s'en charge et très facilement. Cette librairie JavaScript s'ajoute au projet comme toute autre, avec Yarn, et il est possible de consulter le code source et la documentation ici.

En quelques lignes, Axios peut faire toute sorte de requête comme un GET, un POST ou autre, et de récupérer la réponse afin qu'elle soit traitée comme on l'entend. Cet outil nous laisse aussi le choix de faire une configuration poussée.

Dès lors, cette librairie simple est pourtant indispensable au fonctionnement du projet. Tout comme la suivante.

5.2.2.3.5 WaveDrom

Cette dernière dépendance de l'interface est tout ce pour quoi le projet est développé. Comment représenter les simulations ? Grâce à WaveDrom, une librairie open-source qui nous permet de visualiser des diagrammes électroniques complexes.

Contrairement aux autres, cet outil n'est pas disponible sur le répertoire de paquets de Yarn, il faut l'inclure à l'ancienne avec deux scripts dans index.html. Le code ci-dessous donne un aperçu du fonctionnement de WaveDrom.

```
<script type="WaveDrom">
        {
                signal : [
                                { name: "s1", wave: "0..1." },
                                { name: "s2", wave: "010.." }
                        ],
                foot : { tick : "0 10 100 150 330 " }
        }
</script>
```

Code 4: Exemple de code type traité par WaveDrom

Un objet JSON est placé entre ces balises et est traité par WaveDrom, dès lors il suffit de représenter une simulation avec la même structure. C'est ce qui a été fait.

Pour conclure sur les outils de l'interface, nous avons vu que la structure du code était faite en composants grâce à React, que ces composants sont testés indépendamment de l'application finale avec Storybook, qu'ils peuvent être déjà personnalisés en utilisant ceux de Material-UI, que les requêtes au serveur se font avec Axios, et que les simulations sont visibles graphiquement grâce à WaveDrom. Comme nous venons de décrire l'interface dans sa généralité, il est intéressant de se pencher sur ses parties.

5.2.2.4 Parties de l'interface

Ici, nous allons voir en détail chaque partie de l'interface, et les liens entre elles. Contrairement aux autres – comme celles du serveur par exemple – ces parties sont visuelles, elles sont définies dans l'espace. Et cet espace est celui offert par la fenêtre du navigateur.

Les précisions étant faites, voici les différentes sections : tout d'abord, un schéma de ces parties afin de se les représenter simplement ; puis, les parties du menu et de l'espace de travail seront abordées.

5.2.2.4.1 Schéma de l’interface

L’interface se veut intuitive, agréable et fonctionnelle. Afin de satisfaire ces exigences, elle doit présenter des parties claires et logiques qui occupent l’espace en fonction de leur rôle. Dès lors, le schéma ci-dessous montre les parties globales de l’interface.

Comme vous pouvez l’observer, il y a trois sections : la barre d’application, qui sert simplement à présenter l’interface, comme un titre ; le menu, qui sert principalement à sélectionner et paramétrer la simulation ; et l’espace de travail, qui sert à jouer et afficher la simulation.

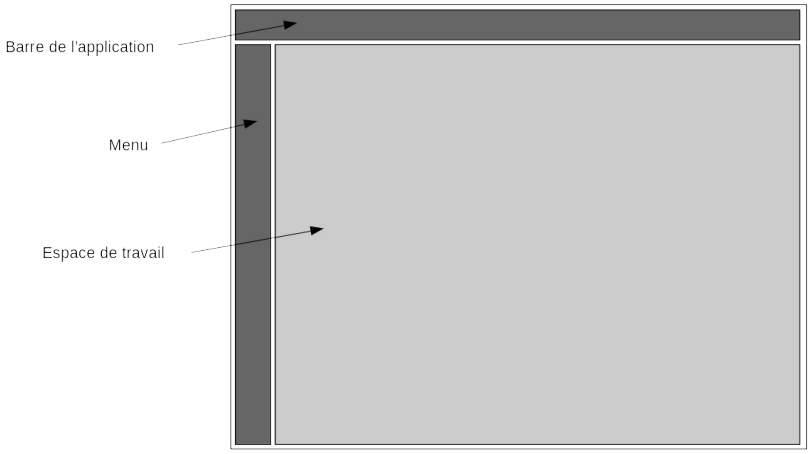

Figure 10: Schéma de l'interface

La barre d’application n’a pas besoin de plus de détails, donc seuls le menu et l’espace de travail vont être approfondis.

5.2.2.4.2 Menu

Cette section est indispensable pour comprendre la partie essentielle qu’est le menu. Pour voir en détail ce menu, une description sera utile, puis nous nous attarderons sur ses fonctions, et enfin les parties du menu pourront être abordées.

5.2.2.4.2.1 Description du menu

Le menu est une barre immobile située sur la gauche de l’interface, contenant différents onglets qui ouvrent ou ferment leur sous-menu associé quand on clique dessus. Il prend le moins de place possible afin de laisser le reste à l’espace de travail. En effet, ce sont de simples icônes qui suggèrent le contenu des sous-menus.

Quand on clique sur une icône, son sous-menu s’ouvre sur la droite et couvre l’espace de travail de sa largeur. Nous verrons le contenu des sous-menu lors de la description des parties du menu.

Attardons-nous quelques temps sur les fonctions dudit menu.

5.2.2.4.2.2 Fonctions du menu

Cette partie de l'interface est très importante car sans elle, il est impossible de faire une simulation et de la configurer. Comme les fonctions seront détaillées pour chaque partie du menu, je vais rester général.

Le menu sert à gérer les fichiers de circuits et de simulations, à en sélectionner afin de lancer une simulation, à sélectionner les fils que l'on veut voir ou pas, et à configurer l'affichage de la simulation. Il influe donc directement sur l'espace de travail comme nous le verrons par la suite.

Les parties de description et de fonctions du menu étaient brèves et concises, les détails sont énoncés dans la prochaine section.

5.2.2.4.2.3 Parties du menu

Les fonctions du menu énumérées précédemment découlent des parties de celui-ci. En effet, nous avons une partie réservées aux fichiers de circuits, une autre aux fichiers de simulations, une troisième gérant les signaux de la simulation, et une dernière pour sa configuration. Chacune sera détaillée ci-dessous, juste après avoir vu le schéma des parties du menu.

5.2.2.4.2.3.1 Schéma des parties du menu

Comme pour chaque section sur les parties, un schéma est le bienvenu pour se les représenter simplement. Dès lors, la figure ci-dessous montre le menu et ses différentes parties qui sont les circuits, les simulations, les signaux, la configuration et les paramètres.

Les paramètres ne sont pas implémentés donc cette partie ne sera pas détaillée contrairement aux autres. Concernant le sous-menu, c'est le contenu de chaque onglet que je vais présenter juste après, et comme il change en fonction de l'onglet sélectionné il ne constitue pas une partie en soi.

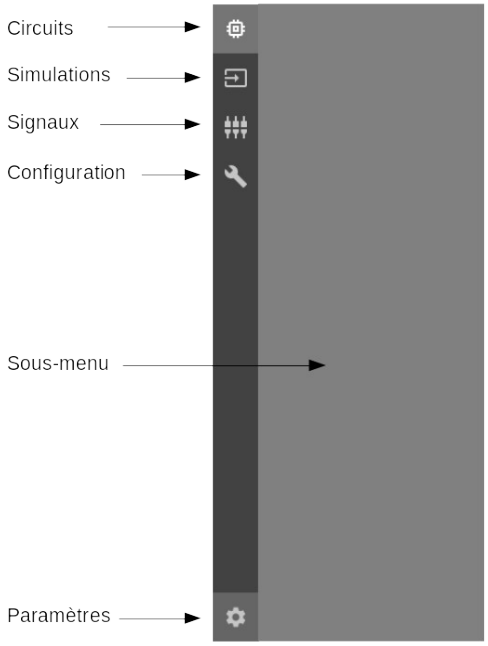

Figure 11: Schéma du menu de l'interface

5.2.2.4.2.3.2 Circuits

Lorsqu'on clique sur l'onglet Circuits, son sous-menu s'ouvre et laisse l'utilisateur faire plusieurs choses :

- uploader des fichiers de circuits sur le serveur
- lister ces fichiers
- rechercher parmi ces fichiers
- modifier le nom de ces fichiers
- supprimer ces fichiers
- sélectionner un de ces fichier pour une simulation

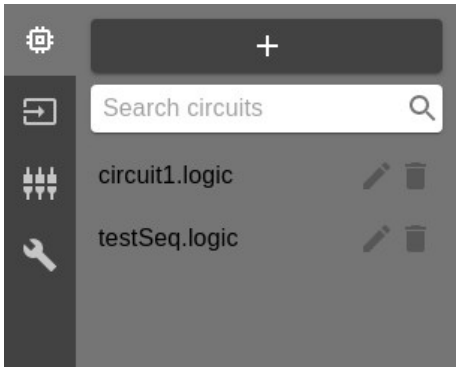

Figure 12: Capture d'écran du sous-menu Circuits

Ainsi, cet onglet permet de gérer les fichiers de circuits (se terminant par l'extension .logic) sur le serveur, et d'entamer le processus de simulation en sélectionnant le circuit sur lequel on veut effectuer la simulation.

Ayant voulu rendre l'interface la plus intuitive possible, le bouton « + » sert à uploader des fichiers de circuits en ouvrant notre explorateur de fichiers pour sélectionner ceux qu'on veut, tant qu'ils ont l'extension .logic. Il est important de noter que quand on upload un fichier sur le serveur, il est stocké sous un nom *hashé* (b18f8d03883e8e4f9c76349cb0fe849b par exemple) et son nom original et son emplacement sont enregistrés dans la base de données.

La barre de recherche laisse l'utilisateur chercher les fichiers qui contiennent l'expression entrée. Par exemple si j'écris « uit » puis clique sur l'icône de recherche (ou appuie sur Entrée), seul circuit1.logic s'affichera.

De plus, chaque fichier listé est un composant qui contient un bouton pour renommer le fichier (son nom dans la base de données), et un pour le supprimer (du serveur et de la base de données). Ces fonctionnalités sont accessibles quand le fichier est sélectionné, soit quand on a cliqué dessus.

Enfin, comme je viens de le dire, un circuit est sélectionné quand on clique sur celui-ci. L'application sait à ce moment quel fichier de circuit utiliser pour la simulation.

Dès lors, cette partie du menu concerne les fichiers descriptifs de circuits, de leur gestion sur le serveur à leur sélection pour simulation. Elle est très similaire à celle des simulations.

5.2.2.4.2.3.3 Simulations

Le sous-menu Simulations ressemble beaucoup au sous-menu Circuits. En effet, ils utilisent les mêmes composants comme nous pouvons le voir sur la figure ci-après.

Les fonctionnalités sont les mêmes, la seule chose qui change est que l'on gère les fichiers de simulations, qui eux ont l'extension .simu.

Nous pouvons donc uploader des fichiers de simulation en cliquant sur le bouton « + », rechercher parmi ces fichiers grâce à la barre de recherche, et puis sélectionner, éditer et supprimer un de ces fichiers.

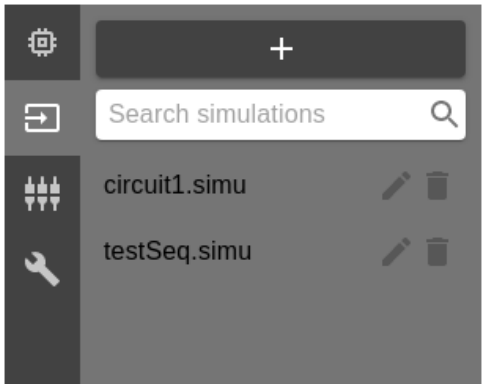

Figure 13: Capture d'écran du sous-menu Simulations

De la même façon que pour les circuits, quand on sélectionne un fichier de simulation, l'application sait que c'est ce fichier qui servira à la simulation.

La seule différence est que la sélection d'une simulation implique son affichage sous forme de graphique WaveDrom dans l'espace de travail, mais nous verrons cela dans la partie appropriée.

Nous venons de voir les deux parties du menu indispensables à la simulation de circuits logiques, car sans sélection de circuit et de simulation, nous ne pouvons pas faire grand-chose. La partie suivante n'est pas inutile cependant.

5.2.2.4.2.3.4 Signaux

Nous allons voir maintenant la partie Signaux du menu. Vous remarquerez qu'elle est vide quand on arrive sur l'interface. En effet, ce sous-menu est fait pour lister les signaux de la simulation actuelle, donc si aucune simulation n'est sélectionnée, il n'y a aucun signal à afficher.

Lorsque l'on clique sur une simulation, elle est chargée, tout comme ses signaux. Alors, ils sont listés dans le sous-menu Signaux comme nous pouvons le voir dans la capture d'écran de gauche.

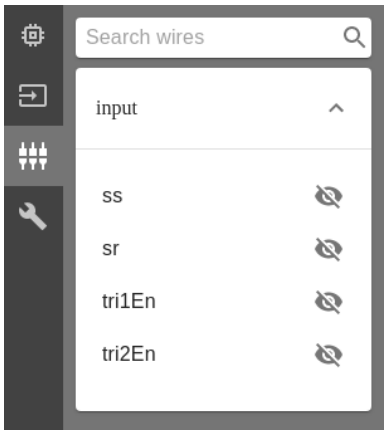

Figure 14: Capture d'écran du sous-menu Signaux associé

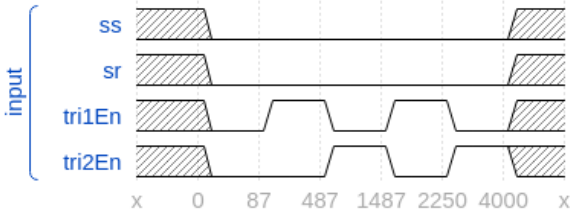

Figure 15: Capture d'écran du WaveDrom de la simulation sélectionnée

Dans les captures précédentes, il est seulement question des signaux d'entrée de la simulation, pas de ceux observés durant son exécution. Pour voir ces signaux il faut sélectionner un circuit et exécuter la simulation.

Ce sous-menu laisse la possibilité à l'utilisateur de rechercher des signaux, de la même façon qu'il peut rechercher des circuits ou des simulations.

Il peut aussi choisir de cacher des signaux du WaveDrom en cliquant sur l'icône représentant un œil barré. Cela peut être utile s'il y a beaucoup de signaux ou si seulement certains nous intéressent.

Dès lors, la partie Signaux sert à cacher ou non les signaux de la simulation, exécutée ou non. Le sous-menu que je vais présenter après permet aussi de jouer avec l'affichage de la simulation.

5.2.2.4.2.3.5 Configuration

L'onglet Configuration apporte d'autres fonctionnalités à l'utilisateur rendant l'affichage de la simulation plus poussé.

En effet, ce sous-menu permet de sélectionner une intervalle. Il est ainsi possible de dire au serveur : « je veux voir la simulation du temps 100 au temps 1000 ». Ce qui est bienvenu si la simulation est très longue.

La valeur *shift* est liée à l'intervalle. Comme nous le verrons dans la partie Espace de travail, il est possible d'aller à l'intervalle suivante ou précédente. La valeur représente donc le temps de déplacement. Pour reprendre l'exemple précédent, si *shift* vaut 30, alors l'intervalle suivante commencera à 130 et se terminera à 1030.

Pour conclure, les parties Signaux et Configuration modifient l'affichage graphique de la simulation sur l'espace de travail.

Nous arrivons à la fin de la présentation du menu de l'interface, où l'on a décrit cette partie, résumé ses fonctions, et détaillé chaque sous-menu. Vous avez dû vous rendre compte de l'utilité de chaque sous-menu, que ce soit Circuits, Simulations, Signaux ou Configuration.

L'autre grande partie de l'interface est l'espace de travail, et nous allons nous y attaquer dans la prochaine section.

5.2.2.4.3 Espace de travail

La deuxième grande partie de l'interface est l'espace de travail. Comme pour le menu, nous allons nous attarder sur cet espace en commençant par le décrire, puis en expliquant ses fonctions, et enfin en détaillant ses parties.

5.2.2.4.3.1 Description de l’espace de travail

L’espace de travail est la partie de l’interface qui occupe le plus de place, même si elle est partiellement recouverte par le menu quand on clique sur un onglet.

Elle est découpée en trois parties, qui sont, de celle du haut à celle du bas : le statut de la sélection, le graphique de la simulation, et le Player. Nous verrons chacune d’entre elles après avoir listé les fonctions.

Maintenant que nous avons brièvement décris l’espace de travail, passons à ses fonctions.

5.2.2.4.3.2 Fonctions de l’espace de travail

Cette partie de l’interface est tout aussi indispensable que le menu et nous allons voir pourquoi. Le but de l’interface est de lancer et visualiser des simulations, mais ce n’est pas le menu qui s’en charge, c’est l’espace de travail.

En effet, nous pouvons lister ses fonctionnalités de la sorte :

- montrer quels fichiers ont été sélectionnés
- afficher la représentation graphique de la simulation
- exécuter la simulation et se déplacer par intervalles

Chaque point correspond à une partie de l’espace de travail, il faut se pencher sur la section suivante pour avoir plus de détails.

5.2.2.4.3.3 Parties de l’espace de travail

L’espace de travail ayant fait l’objet d’une description et d’une liste de ses fonctions, il ne reste qu’à voir ses parties. Et comme il a été mentionné plus haut, il y en a trois : le statut de la sélection, le graphique de la simulation et le Player.

Ne perdons pas de temps pour les présenter avec un schéma, puis individuellement.

5.2.2.4.3.3.1 Schéma de l’espace de travail

Le shéma ci-dessous encadre simplement les parties de l’espace de travail sur une capture d’écran, une fois un circuit et une simulation sélectionnés. Nous retrouvons le statut de la sélection, le graphique de la simulation et le Player.

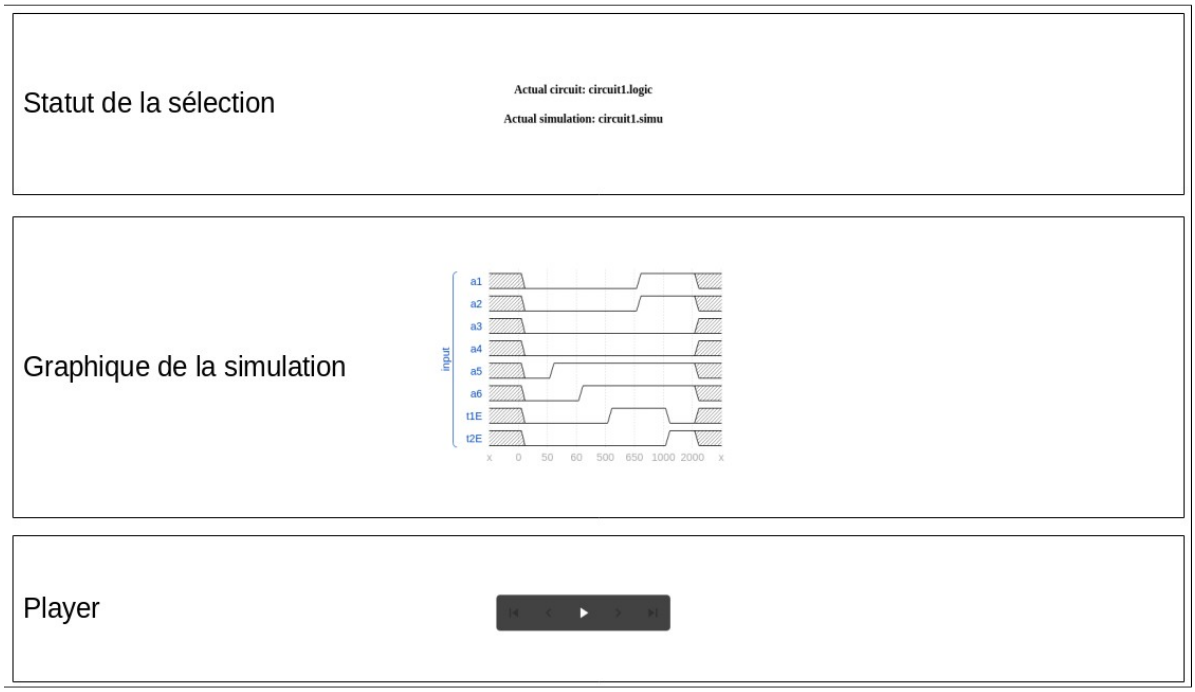

Figure 16: Schéma de l'espace de travail de l'interface

Passons maintenant à la description de chaque partie.

5.2.2.4.3.3.2 Statut de la sélection

Cette première partie de l’espace de travail, qui se situe en haut, expose les noms du circuit et de la simulation sélectionnés.

Il permet simplement à l’utilisateur de savoir si son clic sur un des fichiers a bien été pris en compte, et de savoir quel circuit et quelle simulation sont utilisés sans avoir à ouvrir le menu.

Sa présentation est simple et fonctionnelle, comme les autres parties.

5.2.2.4.3.3.3 Graphique de la simulation

Nous voici à la partie principale de l’espace de travail, celle qui affiche le graphique de la simulation.

Quand la variable JSON contenant les signaux de la simulation est reçue par l’interface, elle est passée à cette partie qui fait appel à WaveDrom pour la traiter. Dès que cette variable change, le graphique est mis à jour.

Ainsi, quand la simulation est exécutée, la variable est modifiée – elle contient en plus les signaux observés – et le graphique affiche alors le résultat. Ou, quand on cache un signal, il est retiré de cette variable et donc le nouveau graphique ne contient plus ce signal. De même pour les intervalles.

Sa fonction est simple, mais indispensable, tout comme celle du Player.

5.2.2.4.3.3.4 Player

Dans le schéma de l'espace de travail, le Player se trouve en bas. Il contient plusieurs boutons qui vont effectuer des requêtes au serveur et modifier le graphique de la simulation.

La plus importante est celle de l'exécution, lancée quand on clique sur le bouton *Play*. Ce bouton est cliquable car un circuit et une simulation ont été sélectionnés. En effet, il n'est possible de lancer une simulation que si un circuit et une simulation sont choisis.

Les autres qui sont inactifs sur le schéma gèrent le déplacement par intervalle. Comme toute la simulation est affichée, il n'y a pas de déplacement possible. Mais si l'on sélectionne une intervalle allant de 0 à 100 à partir du menu, il sera possible de se déplacer vers la droite.

Les boutons aux extrémités servent à aller au tout début ou à la toute fin de la simulation. Alors que les autres – sauf le bouton du milieu – ont pour fonction d'aller à l'intervalle précédente ou suivante. J'en avais déjà parlé dans la partie Configuration du menu, notamment avec la valeur *shift*.

À ce point, toutes les parties de l'interface ont été vues. À commencer par le menu et ses différents sous-menu, dont Circuits, Simulations, Signaux et Configuration. Et celles de l'espace de travail, contenant le statut de la sélection, le graphique de la simulation et le Player.

Durant cette présentation de l'interface, le serveur a été mentionné quelques fois, notamment lors de la gestion de fichiers de circuits et de simulations. D'ailleurs, ces détails sur l'interface ne nous dit pas comment tout cela fonctionne, nous n'avons vu que la peinture, maintenant il faut ouvrir le capot.

5.2.3 Serveur

Après avoir détaillé longuement l'interface, nous pouvons passer à la deuxième partie du projet : le serveur. Qu'est-ce que le serveur, à quoi sert-il, quelles parties le composent… Les réponses à ces questions se trouvent ici.

Nous commencerons par une description du serveur, puis nous verrons quelles sont ses fonctions, il sera aussi nécessaires de lister ses dépendances, et enfin ses différentes parties.

5.2.3.1 Description du serveur

Le serveur est un service, une application, s'exécutant sans arrêt sur une machine qui peut recevoir des requêtes venant de l'internet, et répondant en fonction. Ce service peut utiliser toute sorte d'autres services, allant de la base de données à des programmes quelconques. C'est vraiment le cœur d'une application web complexe car sans le serveur, l'interface n'a pas lieu d'être.

Pour reprendre l'exemple de la description de l'interface à propos du site https://www.youtube.com/, ce qui s'affiche dans notre navigateur est l'interface, mais les données affichées (comme l'utilisateur, les vidéos, les suggestions, etc.) viennent du serveur.

Dans le cas de ce projet, l'interface fait des requêtes au serveur en fonction des actions de l'utilisateur, et le serveur renvoie les données appropriées que l'interface affiche à sa façon.

Afin de générer et exposer ce serveur, il suffit d'exécuter `yarn nx serve server` qui va déployer l'application du serveur sur http://localhost:8080. D'autre part, le serveur fonctionne de pair avec une base de données MySQL qu'il faut aussi déployer, cette fois-ci avec docker, comme nous le verrons dans les parties du serveur.

Pour résumer, il faut entrer l'adresse de déploiement de l'interface dans la barre de recherche du navigateur (voir description de l'interface), l'interface enverra des requêtes sur l'adresse du serveur, et attendra les réponses.

Le langage utilisé pour développer ce serveur est détaillé dans les dépendances, mais avant cela il est intéressant de se pencher ses fonctionnalités.

5.2.3.2 Fonctions du serveur

La réponse à la question 'Qu'est-ce que le serveur ?' se trouve dans la section précédente, mais celle à la question 'Que fait-il concrètement ?' se trouve ici.

Comme nous l'avons vu, le serveur répond aux requêtes qu'on lui envoie, et ces requêtes peuvent être de toute sorte, un POST contenant une variable JSON, un simple GET, ou encore un DELETE sur une certaine adresse… Bref, le serveur répond à tout cela mais de manière différente.

Si la requête n'est pas connue, le serveur ne retournera rien, ou alors une erreur. Pareil si la requête est prise en charge, mais qu'une erreur survient (un fichier inexistant, une simulation inconnue, etc.).

Une requête est traitée si jamais le code correspondant a été développé. Par exemple, si je fais un GET à l'adresse http://adresse-serveur/simulations, il faut qu'il y ait une fonction dans le code du serveur qui s'en charge spécifiquement.

Dès lors, la fonction principale du serveur est de recevoir, traiter, et répondre à des requêtes.

Maintenant, le serveur a aussi des fonctionnalités secondaires, que nous allons énumérer rapidement. Tout d'abord, une fonction de stockage. On peut y déposer des fichiers de circuits et de simulations comme expliqué dans la partie de l'interface. Cette fonction de stockage dépend évidemment de celle des requêtes, car ce sont des requêtes contenant des fichiers qui vont permettre leur enregistrement sur le serveur.

Il ne faut pas oublier le rôle important de la base de données. En effet, quand on upload un fichier, une nouvelle entrée est créée contenant l'identifiant, le nom et le chemin de stockage de ce fichier. C'est un peu comme un registre qui renseigne sur le contenu du serveur.

Le serveur a aussi la fonctionnalité de simulation, rendue possible grâce au simulateur développé par Luc Parent. Ce programme est présent sur le serveur, et il est appelé selon les requêtes.

Et enfin, la fonction d'extraction sert à lire les fichiers de simulation, avant et après simulation, pour former une variable JSON contenant les signaux et l'abscisse de temps. C'est cette variable qui est traitée par WaveDrom et affichée sous forme graphique sur l'interface.

Nous venons d'aborder brièvement les fonctionnalités du serveur, elles seront plus détaillées dans les parties. En attendant, passons à ses dépendances.

5.2.3.3 Dépendances du serveur

Maintenant que nous savons ce qu'est la partie serveur et à quoi elle sert, il reste à savoir qu'est-ce qui rend cela possible. Et pour savoir cela, nous devons lister quels sont les outils dont dépend le serveur.

Notez que ce sont les dépendances du serveur en général, les dépendances de chaque partie seront vues dans leur partie respective. En effet, nous traitons ici ce qui permet au serveur d'assurer sa fonction principale, qui est de traiter des requêtes.

Dès lors, ceci est possible grâce un seul outil, et pas des moindres, Nest.

5.2.3.3.1 Nest

Dans cette section nous allons nous attarder sur l'outil Nest. Ce framework Node.js distribué sous licence MIT, dont le code est visible ici, sert à développer des applications serveur.

Rappelons rapidement que Node.js est un environnement d'exécution JavaScript asynchrone orienté événement adapté à la gestion de requêtes. Pour prendre un exemple assez spécial, une fonction synchrone peut être comparée au fait de rester devant la machine à laver jusqu'à ce que ce soit fini, tandis qu'une fonction asynchrone lancerait la machine et passerait à autre chose.

Dans le code ci-dessous, la fonction `findOne` « promet » de retourner une `Simulation`. Donc l'environnement lance la fonction et passe à autre chose.

```
findOneSimulation(id: string): Promise<Simulation> {
        return this.simulations_repository.findOne(id);
}
```

Code 5: Fonction Node non-bloquante

Cependant, il est possible d'attendre le résultat d'une `Promise` comme dans l'exemple ci-après. Pour cela il faut préciser que la fonction est asynchrone avec `async` et placer `await` devant l'instruction concernée. Cela revient à attendre devant la machine à laver. C'est très utile si jamais il est nécessaire de récupérer les `Promise` de façon séquentielle. On ne voudrait pas lancer la sécheuse si jamais les vêtements sont encore dans la machine à laver.

```
async findOneSimulation(id: string): Promise<Simulation> {
        return await this.simulations_repository.findOne(id);
}
```

Code 6: Fonction Node bloquante

Comme vous pouvez le voir, le développement est similaire à celui de l'interface. On retrouve les types de TypeScript et la syntaxe JavaScript, cela est très pratique car il est possible de partager du code entre le *frontend* et le *backend*, surtout grâce à Nx.

D'ailleurs Nx propose la création et la gestion d'applications Nest – raison pour laquelle Nx a été choisi – et c'est pour cela qu'il est simple de déployer l'application avec la commande `nx serve` comme vu dans la description de l'interface ou du serveur.

Pour revenir sur la fonctionnalité principale du serveur, soit traiter des requêtes, le code 7 est un exemple qui permet de mieux comprendre comment cela se fait.

Ce code permet de traiter les requêtes GET à l'adresse http://adresse-serveur/simulations/***id*** avec **id** un nombre identifiant une simulation. Ici, le traitement dépend de la base de données donc nous verrons cela plus tard.

Ainsi, Nest est adapté aux applications côté serveur de par son environnement Node.js, mais aussi pour les nombreuses fonctionnalités intégrées. Pour plus d'informations, le site officiel contient une très bonne documentation, ou alors les exemples sur le répertoire github.

```
@Controller('simulations')
export class SimulationsController {
	@Get(':id')
	async findOne(@Param('id') id: number) {
		return this.simulations_service.findEntity(id);
	}
}
```

Code 7: Code traitant les requêtes GET sur http://adresse-serveur/simulations/id-simulation

Le traitement des requêtes ne dépend que de Nest, mais passons aux parties du serveur pour voir les outils spécifiques et leur rôle.

5.2.3.4 Parties du serveur

Le serveur est une partie complexe du projet, elle est responsable de la récupération des requête, de leur traitement, et de l'envoi de la réponse correspondante. Cela se fait grâce à Nest, mais le traitement n'a pas encore été détaillé.

Nous allons le faire en abordant les différentes parties dudit serveur. Chaque partie étant essentielle au fonctionnement global, il est pourtant possible de les séparer selon leur rôle.

Ainsi, un schéma servira à voir et comprendre ces parties et leurs interactions, puis nous verrons chaque partie individuellement, soit le contrôleur, la base de données, l'extracteur et le simulateur.

5.2.3.4.1 Schéma du serveur

Ici, le but est de représenter les différentes parties du serveur et les interactions qu'elles ont entre elles. Le schéma ci-après servira à cela, et une brève explication suivra.

La flèche à double-sens entre Internet et le contrôleur représente la requête venant d'Internet et la réponse du serveur.

C'est le contrôleur qui capte les requêtes et fait ce qui est demandé. Il a accès à toutes les fonctionnalités et donc parties du serveur. Il peut ajouter ou supprimer des fichiers, gérer la base de données, ou encore appeler l'extracteur et le simulateur.

L'extracteur et le simulateur se servent des fichiers, car ils ont pour fonction respective d'extraire les simulations sous forme de variable JSON et de lancer une simulation sur un circuit. Le simulateur altère cependant les fichiers en stockant le résultat.

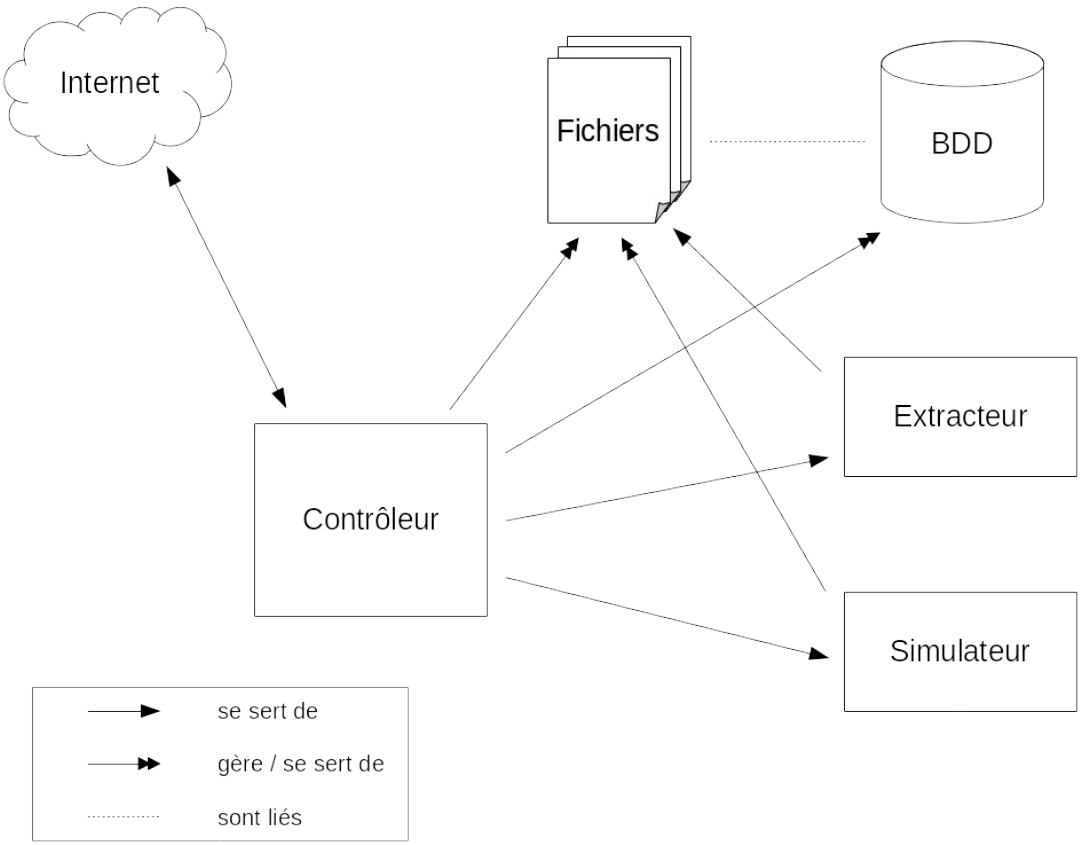

Figure 17: Schéma du serveur

Enfin, la base de données et les fichiers n'interagissent pas entre eux mais sont liés car le nom et l'emplacement de chaque fichier est enregistré dans la base. Dès lors, quand un utilisateur renomme un fichier, c'est le champ dans la base de données qui est modifié.

Pour plus de détails, chaque partie est développée ci-après.

5.2.3.4.2 Contrôleur

Nous voici dans la section détaillant la partie contrôleur du serveur. Afin d'en savoir plus sur celle-ci, je vous propose une description générale de cette partie, puis une section sur ses fonctions, et enfin ses dépendances.

5.2.3.4.2.1 Description du contrôleur

Comme expliqué antérieurement, le contrôleur est une sorte de chef d'orchestre qui va recevoir les requêtes et exécuter la fonction appropriée qui se servira de telle ou telle autre partie du serveur. On pourrait dire que le contrôleur est « à la surface ».

Dans le schéma ci-contre – généré par la commande `yarn doc:server` – le contrôleur est plus détaillé. Nous avons le module principal, AppModule, qui se sert de CircuitsModule et SimulationsModule.

Concrètement, le module principal se sert de deux modules : un pour gérer les simulations, et un pour gérer les circuits.

C'est eux qui contiennent le code relatif au traitement des requêtes. Ainsi, c'est l'un des deux modules qui prendra en charge une requête.

A présent, nous devrions voir quelles sont les fonctions du contrôleur.

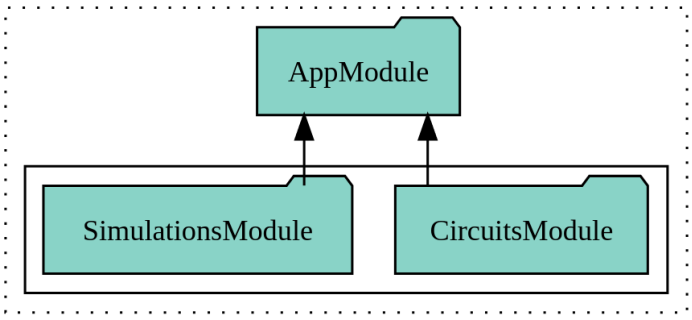

Figure 18: Schéma du contrôleur du serveur

5.2.3.4.2.2 Fonctions du contrôleur

Le contrôleur du serveur a pour fonction principale celle de réception de requête et de réponse. Sans cette partie, impossible de dialoguer avec le serveur et de lui faire exécuter des tâches.

En effet, chaque module contient une fonction par requête possible. Par exemple, la fonction `findOne` ci-dessous provenant de CircuitsModule est chargée de retourner le résultat de la fonction `findEntity` quand le serveur reçoit la requête GET à l'adresse http://adresse-serveur/circuits/***id***.

```
@Controller('circuits')
export class CircuitsController {

	@Get(':id')
	findOne(@Param('id') id: string): Promise<Circuit> {
		return this.circuits_service.findEntity(id);
	}
}
```

Code 8: Code traitant les requêtes GET sur http://adresse-serveur/s/id-circuit

Pour plus de précision, le descripteur `@Controller('circuits')` signale que la classe `CircuitsController` utilisée par CircuitsModule prend en charge les requêtes à l'adresse http://adresse-serveur/circuits/. Puis, le descripteur `@Get(':id')` au dessus de la fonction précise que c'est elle qui doit s'exécuter quand l'adresse est du type http://adresse-serveur/circuits/***id*** avec ***id*** étant n'importe quelle valeur.

L'autre fonction du contrôleur – en fait des modules gérant les circuits et les simulations – est de stocker les fichiers reçus. Il a été dit plusieurs fois que l'on pouvait uploader des fichiers de circuits et de simulations sur le serveur, c'est ici que cela se passe.

Dans le bout de code ci-après issu de CircuitsModule – circuits.controller.ts – nous pouvons voir que la fonction `uploadCircuitFiles` est précédée du descripteur `@Post()` donc elle est appelée lors d'une requête POST à l'adresse http://adresse-serveur/circuits. C'est le descripteur `@UseInterceptors` qui est responsable de récupérer les fichiers et de les enregistrer.

Il utilise un `FilesInterceptor` qui autorise un maximum de 20 fichiers et qui les stocke à l'adresse `dest`. Notez que cette adresse est fixe, mais quand la fonctionnalité de connexion sera implémentée, `user1` devra être remplacée par le nom d'utilisateur.

Ainsi, la fonction n'est appelée qu'une fois les fichiers enregistrés, d'où le paramètre `UploadedFiles`. Le contenu de celle-ci n'est pas détaillé ici car seule la fonction d'upload nous intéressait.

```
@Post()
@UseInterceptors(FilesInterceptor('file', 20, {
        dest: 'simulator/home/user1/circuitCreator/data'
}))
async uploadCircuitFiles(@UploadedFiles() files: Express.Multer.File[]) { /* ... */ }
```

Code 9: Descripteurs responsables de l'upload de fichiers

Pour conclure, le contrôleur sert principalement à recevoir et répondre à des requêtes, qui parfois demandent l'upload des fichiers envoyés.

Mais de quoi a besoin le contrôleur pour effectuer ces tâches ? Je vous invite à lire la section suivant pour cela.

5.2.3.4.2.3 Dépendances du contrôleur

La partie contrôleur du serveur gère les requêtes venant d'internet et peut uploader les fichiers envoyés. Pour cela, le contrôleur n'a besoin d'aucun outil spécifique, tout est déjà fourni par Nest.

En effet, les descripteur que nous avons vu précédemment, permettant d'effectuer ces tâches, sont déjà inclus dans le framework. Il suffit simplement de les importer au début du fichiers contenant les fonctions. Je vous renvoie au code du projet pour cela.

Cette petite section sur les dépendances du contrôleur étant finie, nous pouvons résumer la partie contrôleur. Nous avons vu que c'est elle qui, grâce à ses modules, traitait les requêtes et envoyait le résultat. Ces requêtes pouvaient impliquer l'upload de fichiers de circuits ou de simulations. Et que cela ne dépendait que de Nest.

Mais nous avons vu dans le schéma du serveur que le contrôleur gérait d'autres parties, comme la base de données. La section suivante traite justement de cela.

5.2.3.4.3 Base de données

La base de données, une partie essentielle du serveur. Qu'est-ce que c'est ? A quoi sert-elle ? De quoi a-t-elle besoin pour fonctionner ? Nous tenterons de répondre à ces questions simplement dans cette section. Pour cela, nous commencerons par décrire la base de données, puis nous continuerons en listant ses fonctions, et pour finir nous examinerons ses dépendances.

5.2.3.4.3.1 Description de la base de données

La base de données est un service déployé sur le serveur permettant d'ajouter, modifier et supprimer des données. Ces données sont organisées en tables – des tableaux en fait – dont les champs sont préalablement définis.

La base de données du serveur contient deux tables, une pour les circuits et une autre pour les simulations. Les voici :

Circuits	id	name	path	simulator_path
	…	…	…	…

Simulations	id	name	path	result_path
	…	…	…	…

La première s'occupe des fichiers relatifs aux circuits, tandis que la deuxième, des fichiers relatifs aux simulations.

Vous vous demandez peut-être comment le contrôleur a-t-il accès à cette base de données, et si vous ne vous posez pas la question je vais quand même y répondre. Les schémas ci-dessous reprennent les modules du contrôleur et affichent leur lien avec les services CircuitsService et SimulationsService.

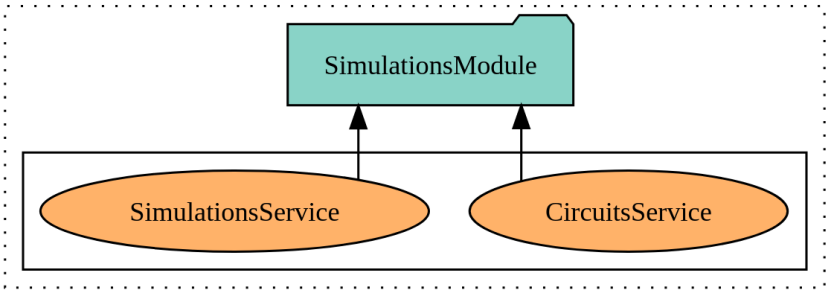

Figure 19: Schéma de SimulationsModule

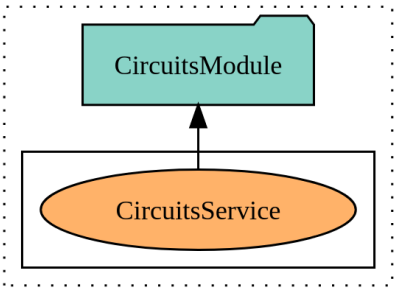

Figure 20: Schéma de CircuitsModule

Ce sont ces services qui contiennent des fonctions gérant la base de données. Plus précisément, SimulationsService accède à la table des simulations tandis que CircuitsService touche à celle des circuits.

Vous pouvez d'ailleurs voir dans le schéma de gauche que le module s'occupant des simulations se sert des deux services étant donné qu'il a besoin d'accéder aux fichiers de simulations et de circuits pour exécuter des simulations.

La base de données est donc un service nécessaire au fonctionnement du serveur, et nous allons préciser dans la section suivante ses fonctions.

5.2.3.4.3.2 Fonctions de la base de données

La description de la base de données a donné un aperçu de ce qui y est stocké, c'est-à-dire les informations relatives aux fichiers de simulations et de circuits. Dès lors, il est possible de déduire la fonction de cette partie du serveur, qui est de mémoriser les fichiers stockés dans le serveur.

Pour mieux comprendre, quand on upload un fichier décrivant un circuit, une nouvelle ligne est ajoutée à la table Circuits. Cette ligne contient l'identifiant généré automatiquement, le nom du fichier, et son emplacement.

Or, le champ `simulator_path` reste vide. Cette donnée contiendra l'emplacement du simulateur relatif à ce circuit une fois compilé. C'est cet exécutable qui sera utilisé pour effectuer des simulations sur ce circuit.

Pour la table Simulation, c'est la même chose lors de l'upload. Mais cette fois, le champ qui reste vide s'appelle `result_path`. Il est destiné à stocker l'emplacement du résultat de cette simulation, qui est mis à jour à chaque simulation.

Dès lors, le contrôleur se sert de la base de données pour connaître quels fichiers existent, où ils se trouvent, aussi pour savoir s'il doit compiler le simulateur d'un circuit, ou s'en servir directement… Bref, c'est cette partie qui contient les informations des fichiers, si indispensable au bon fonctionnement du serveur.

La question qu'il reste à se poser concernant la base de données est : quelles sont ses dépendances ?

5.2.3.4.3.3 Dépendances de la base de données

Nous venons de voir à quoi servait la base de données, mais nous n'avons pas encore d'indice sur ce qui la fait tourner. Elle dépend de deux outils, l'un extérieur à Nest, l'autre compris dans Nest. Je veux parler ici de Docker et de TypeORM.

5.2.3.4.3.3.1 Docker

Docker est un environnement d'exécution open-source permettant de déployer des applications sur n'importe quel environnement. Une application peut être un environnement Linux, Java, Node, une base de données, etc. On peut même se créer notre propre application à partir de celles disponibles. Les applications sont en fait appelées *images*.

Ces *images* peuvent être déployées par Docker sur tout type de machine grâce à des *containers*. C'est une façon d'isoler l'application de l'environnement hôte, tout en utilisant ses ressources. Le schéma ci-dessous issu du site officiel permet de mieux le visualiser, et de faire la comparaison avec les machines virtuelles.

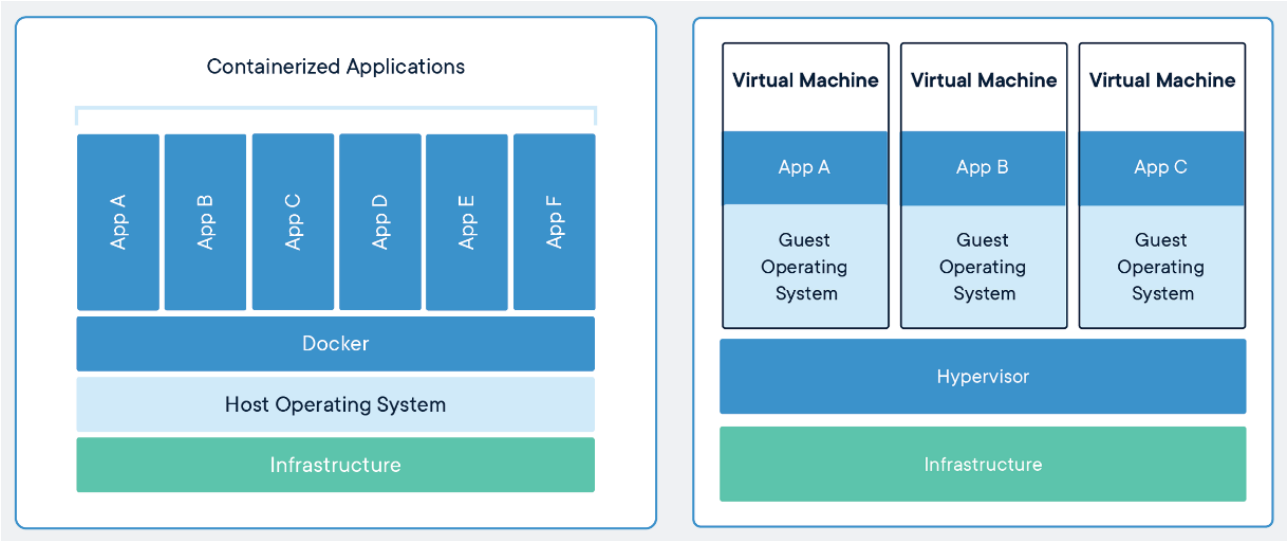

Figure 21: Schéma comparant Docker et machine virtuelle

Après avoir installé Docker, nous avons accès à un programme en ligne de commande qui nous permet de gérer des *containers*, des *images*, et tout ce qui en dépend. Ainsi, l'exécution de `docker container ls` dans un terminal va lister les *containers* présents, et `docker images`, les *images*.

Je vous ai dit qu'il est possible de créer ses propres *images*, l'utilisation de plusieurs *images* en même temps est aussi permise. Pour cela, on utilise des *services*. Ces *services* ont de nombreux paramètres de configuration, notamment l'*image* à déployer. Cette configuration peut se faire dans un fichier docker-compose.yml. Celui utilisé dans le projet est visible ci-contre.

```yaml
version: "3"

services:
    db:
        container_name: db
        image: mysql:5
        restart: always
    environment:
        MYSQL_ROOT_PASSWORD: root
        MYSQL_DATABASE: test
    ports:
        - "3306:3306"
```

Code 10: Fichier docker-compose.yml pour déployer la base de données

Vous pouvez observer qu'il n'y a qu'un seul service, celui de la base de données, configuré d'une certaine façon. Il utilise une image MySQL déposée en ligne, sur le Docker Hub.

Cette configuration va de pair avec le module Nest associé que nous verrons dans la prochaine partie.

Dès lors, pour déployer ce service il faut simplement lancer `docker-compose`. Cet exécutable va lire le fichier docker-compose.yml dans le dossier courant, et faire appel à Docker pour le mettre en route. Il sera possible d'arrêter ce service en faisant un Ctrl+c, ou avec `docker-compose stop`. Sinon, il reste l'exécutable `docker` et ses nombreuses options.

Pour conclure, Docker est un outils indispensable pour déployer facilement tout type d'applications, et dans notre cas une base de données MySQL. Dans la section suivante nous allons nous intéresser à l'interaction entre cette base de données et le serveur.

5.2.3.4.3.3.2 TypeORM

TypeORM est un ORM (*object-related mapping*) permettant l'échange de données entre frameworks JavaScript et base données comme MySQL, MariaDB, Postgres, etc. Le code source de cette dépendance est visible ici, et comme pour la plupart des dépendances, c'est un paquet qui s'installe avec `yarn` ou `npm`. Dès lors, les données du serveur Nest, qui est basé sur Node et donc sur JavaScript, sont gérées par TypeORM. Il est aussi important de relever que Nest permet l'utilisation de cet outil, notamment sous forme de module. En effet, voici la configuration du module principal du serveur que nous avons évoqué dans la description du contrôleur.

Vous remarquerez les deux modules SimulationsModule et CircuitsModule que nous avons déjà vu. Celui de la base de données est un module TypeORM, dont la configuration va de pair avec le service Docker vu précédemment.

Le lien entre base de données et serveur étant fait, il reste à voir comment se créent les tables, et comment des données sont insérées.

Cela se fait à l'aide de décorateurs, par exemple, le code 12 constitue la table Simulation. C'est en fait une simple classe avec des variables internes, mais les décorateurs spécifiques à TypeORM permettent la conversion de l'objet en une table.

```
@Module({
	imports: [
		TypeOrmModule.forRoot({
			type: 'mysql',
			host: 'localhost',
			port: 3306,
			username: 'root',
			password: 'root',
			database: 'test',
			autoLoadEntities: true,
			synchronize: true,
		}),
		SimulationsModule,
		CircuitsModule
	]
})
export class AppModule {}
```

Code 11: Descripteur du module principal AppModule

```
@Entity()
export class Simulation {
	@PrimaryGeneratedColumn()
	id: number;

	@Column()
	name: string;

	@Column()
	path: string;

	@Column()
	result_path: string;
}
```

Code 12: Classe TypeORM de la table Simulation

Concernant la gestion des données, ce sont les services SimulationsService et CircuitsService qui s'en occupent. D'ailleurs, dans plusieurs exemples du contrôleur, les fonctions utilisaient des méthodes de ces services.

Par exemple, lors de l'upload d'un fichier de simulation, le contrôleur fait appel à la fonction create de SimulationsService qui est montré dans le code 13.

Tout d'abord, remarquez le type de paramètre attendu, c'est un DTO. Cela veut dire qu'une vérification a lieu avant d'entrer dans la fonction (voir code 14). Puis, une simulation est enregistrée dans la base de données avec les champs remplis. C'est la fonction save de TypeORM qui fait cela automatiquement. Plus besoin de requêtes SQL complexes !

```
async create(createSimulationDto: CreateSimulationDto) {
        const simulation = new Simulation();
        simulation.name = createSimulationDto.name
        simulation.path = createSimulationDto.path;
        simulation.result_path = '';
        await this.simulations_repository.save(simulation);
}
```

Code 13: Fonction ajoutant une simulation dans la base de données

Pour résumer, nous avons vu qu'un module TypeORM était responsable de la liaison avec le service Docker MySQL, et que les services servaient à créer et altérer des tables dans cette base de données.

Nous arrivons à la fin de la partie base de données du serveur. Comme vous avez pu le constater, nous n'avons pas simplement parlé de la base de données, mais aussi de sa relation avec le contrôleur. En effet, la base de données se résume à un service déployé par Docker, donc elle n'effectue aucune actions, elle reçoit les instructions du contrôleur grâce à TypeORM.

En tout cas, cette base de données est fondamentale. Elle enregistre les données relatives aux fichiers de circuits et de simulations, sans quoi le serveur ne peut pas exécuter de simulations. Toutefois, une fois ces fichiers stockés, il faut pouvoir extraire leur contenu, ce qui nous amène à la prochaine partie du serveur, l'extracteur.

5.2.3.4.4 Extracteur

Le serveur contient aussi une partie extracteur, dont il est possible de retrouver les liens avec les autres parties dans le schéma. Afin de bien la détailler, nous allons dans un premier temps décrire cette partie, puis aborder ses fonctions, et enfin présenter ses dépendances.

5.2.3.4.4.1 Description de l'extracteur

L’extracteur est un service utilisé par le contrôleur des simulations afin de représenter une simulation sous forme de variable WaveDrom, utilisable par l’interface. Cette partie du serveur responsable de la création et de la manipulation de variables à partir de de fichiers de simulations. Il est utilisé par le contrôleur afin de remplir et envoyer une variable JSON contenant la simulation voulue, c’est cette variable qui sera affichée en graphique grâce à WaveDrom.

Pour cela, il faut faire un POST contenant une variable JSON à l’adresse http://adresse-serveur/simulations/extract décrivant l’extraction à effectuer, et le contrôleur fera appel aux fonctions de l’extracteur nécessaires.

Voici le DTO responsable de la validation de cette variable :

```
export class GetSimulationDto {

        @IsNotEmpty()
        @IsNumber()
        @Min(0)
        id_simu: number;

        @IsOptional()
        @IsNumber()
        @Min(0)
        id_circuit: number;

        @IsOptional()
        @DependsOnIfTrue("id_circuit", { message: "'id_circuit' is missing" })
        @IsBoolean()
        result: boolean;

        @IsOptional()
        @IntervalChecker({ message: "'interval' object is not correct" })
        interval: Interval;

        @IsOptional()
        @IsString({ each: true })
        wires: string[];
}
```

Code 14: DTO de la variable décrivant l’extraction à effectuer

Les nombreux descripteurs conditionnent la variable reçue. Elle doit obligatoirement contenir l’identifiant de la simulation, alors que les autres champs sont optionnels. Cependant, si l’on souhaite aussi avoir le résultat, l’identifiant du circuit est indispensable. On peut aussi demander une intervalle de cette simulation, le descripteur associé se charge alors de vérifier cet objet. Et pour

terminer, le champ wires, utile pour sélectionner certains signaux, ne doit contenir que des chaînes de caractères.

C'est donc cette variable qui conditionne la création de la variable WaveDrom faite par l'extracteur. Nous allons découvrir avec précision les fonctionnalités de celui-ci dans la section d'après.

5.2.3.4.4.2 Fonctions de l'extracteur

L'extracteur a un attirail de fonctions qui permettent la création et la manipulation de la variable demandée. La première étant la lecture de fichier de simulations dans le but de contenir cette simulation dans une variable JSON. Pour mieux comprendre cela, le code 15 montre à quoi ressemble un fichier de simulation (voir l'essai de Luc Parent pour plus de détails à ce sujet).

```
START_TIME 0
END_TIME 2000
CIRCUIT_NAME bloc1
WATCHLIST s1

EVENT a1 F 0
EVENT a2 F 0

EVENT a1 T 50
EVENT a2 T 50

EVENT a1 F 650

EVENT a2 F 1000
```

Code 15: Exemple de contenu de fichier de simulation

La variable issue de l'extraction ressemblerait au code 16. Le champ foot représente l'abscisse tandis que le champ signal contient les signaux de simulation. Vous pouvez observer le '-' et le '+', cela montre que toute la simulation est représentée. Entre, il y a le temps de début et fin, et celui de chaque évènement.

Pour les signaux, ils sont décrits par un champ name et un champ wave. Ce deuxième champ est une chaîne de caractères, dont chaque caractère correspond à un état du fil à un évènement. Par exemple, à l'évènement '-', l'état des deux signaux est à 'x', soit inconnu. Au temps 1000, a1 est à 0 et a2 est aussi à 1. En effet, l'état '.' correspond à l'état de l'évènement précédent.

```
{
        signal: [ { name: 'a1', wave: 'x010.x' },
                  { name: 'a2', wave: 'x01.0x' } ],
        foot: { tick: '- 0 50 650 1000 2000 + ' }
}
```

Code 16: Variable WaveDrom issue de l'extraction

Une fois que l'extracteur a cette variable basique, il l'enregistre dans une variable afin de ne pas traiter le fichier à chaque fois.

Il était important de voir ce passage de fichier à variable, c'est la base de toute extraction. D'ailleurs, cette opération a aussi lieu sur le fichier résultat de simulation.

Outre cette fonctionnalité basique, l'extracteur peut combiner plusieurs de ces variables. Cela sert à assembler la variable de la simulation et celle de son résultat. L'extracteur produit donc une variable dont l'abscisse contient les évènements des signaux d'entrée, et ceux des signaux observés. Les signaux d'entrée et observés sont alors contenus dans la liste des signaux, et leur `wave` est adaptée de sorte que leurs états correspondent au nouvel abscisse.

L'extracteur peut aussi sélectionner une intervalle de cette variable. Pour cela, il attend un objet contenant le champ *start* et/ou le champ *end*. En effet, il est possible qu'un des champs soit vide. Si *start* est vide, l'intervalle ira du début à la valeur *end* ; au contraire, si *end* est vide, elle ira de la valeur de *start* à la fin.

D'autre part, comme il a été mentionné dans la description, certains fils peuvent êtres choisis. Pour cela, l'extracteur attend une liste contenant les noms de ces signaux. L'opération consistera à créer une nouvelle variable de type WaveDrom ne contenant que ceux dont le nom correspond. L'abscisse, qui peut contenir des évènements d'autres signaux, n'est pas modifié pour autant.

Nous continuons cette énumération de fonctionnalités avec celle responsable de l'organisation en groupes. Concrètement, elle consiste à modifier la variable WaveDrom afin de séparer les signaux d'entrée de ceux observés. Pour cela, ils sont regroupés en listes dont le premier élément est le nom du groupe : « input » ou « output ». Quand, du côté interface, WaveDrom analysera cette variable, il affichera une accolade permettant de mettre ces groupes en évidence.

Nous venons de voir toutes les fonctions d'extraction. Il a été dit précédemment que l'extracteur enregistrait la variable correspondant au fichier de simulation. Le contrôleur lui impose aussi d'enregistrer celle envoyée à l'interface, soit celle ayant possiblement subi toutes les modifications. Ainsi, il joue aussi le rôle de mémoire temporaire.

Pour conclure, l'extracteur possède de nombreuses fonctions permettant l'extraction de simulations à partir de fichiers et la manipulation des variables résultantes. Pour faire cela, l'extracteur a-t-il besoin d'outils additionnels ? La section d'après répond à cette interrogation.

5.2.3.4.4.3 Dépendances de l'extracteur

L'extracteur, étant une partie du serveur Nest, possède déjà de nombreux outils. Rappelons que Nest est basé sur Node, qui propose beaucoup de fonctions accessibles par un simple import. C'est

le cas de la librairie *fs* – pour *file system* – qui nous offre des fonctions relatives au fichiers système. L'extracteur s'en sert pour aller lire les fichiers de simulations et de résultat de simulation.

La manipulation des variables n'a aucune dépendances, c'est simplement du traitement d'objets JSON.

Donc, cette partie du serveur est relativement autonome pour effectuer les tâches qui lui sont attribuées.

Nous voilà à la fin de la section concernant la partie extracteur du serveur. Pour résumer, le contrôleur fait appel à ce service afin d'extraire le contenu des fichiers relatifs à la simulation, et afin de modifier et enregistrer temporairement ces variables de type WaveDrom issues de l'extraction. Ces modifications peuvent être la combinaison de la variable contenant les signaux d'entrée et celle contenant les signaux observés lors de l'exécution, la sélection ou le regroupement de signaux ou encore la réduction par intervalles.

Dans cette section, le résultat de la simulation a été évoqué plusieurs fois, mais nous ne savons pas encore comment il est obtenu. C'est la partie simulateur du serveur, qui fait suite à celle-ci, qui s'en occupe.

5.2.3.4.5 Simulateur

Nous voilà au niveau du moteur du projet, le simulateur. De façon à bien comprendre son rôle et son fonctionnement, nous commencerons cette section par une brève description, puis nous nous attarderons sur les fonctions du simulateur, et pour finir, sur ses dépendances.

5.2.3.4.5.1 Description du simulateur

Le simulateur est la partie du serveur responsable de la simulation, tout le projet est bâtis dessus. En effet, comme je l'ai déjà mentionné, ce projet fait suite à celui de Luc Parent. Cet étudiant ayant développé ce simulateur, la suite était de donner accès ce programme à des utilisateurs via une interface web, avec la possibilité de stocker ses fichiers. Notez que la gestion d'utilisateurs n'existe pas encore. Ainsi, la partie simulateur du serveur contient le programme de Luc, et comment il est exploité par le serveur.

Pour rappel, il fonctionne avec deux fichiers : un décrivant le circuit (.logic), et un décrivant la simulation (.simu). Concrètement, le fichier .logic est examiné par la partie Java du programme, qui écrit le circuit en langage C++. Ce code peut alors être compilé, ce qui donne l'exécutable, le simulateur. Ensuite, on peut donner le fichier .simu à cet exécutable, qui affichera les évènements du ou des fils observés.

Ce programme a été modifier afin d'être adapté à la structure voulue sur le serveur. En effet, comme il est censé être utilisé par divers internautes, le code commun et celui spécifique à chaque utilisateur est séparé. Les fichiers uploadés sont donc stockés dans la partie utilisateur. La figure ci-après, détaillant la structure des dossiers du simulateur, montre bien cela.

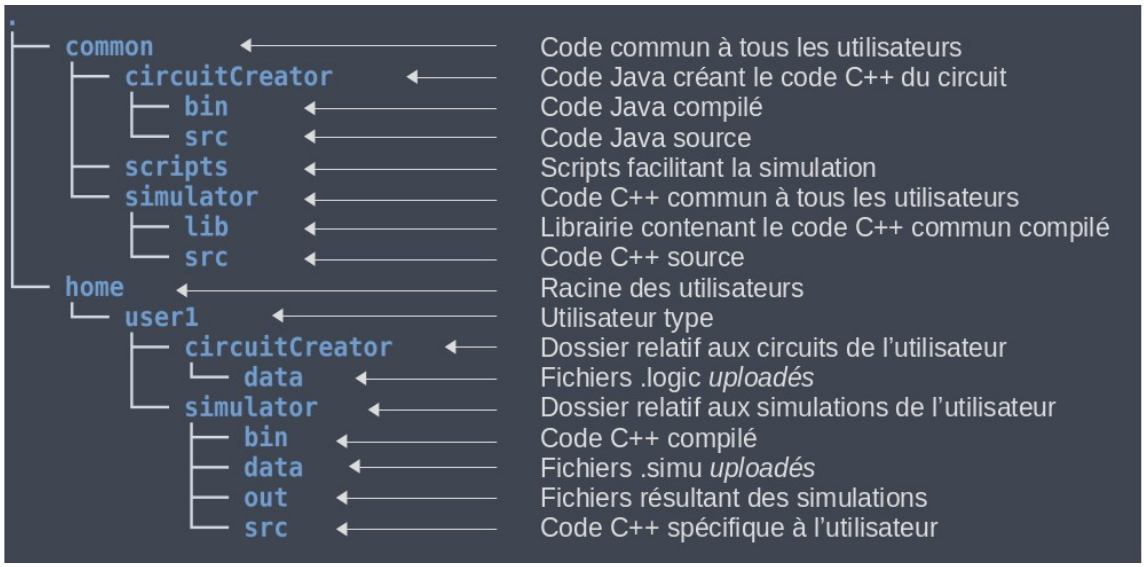

Figure 22: Structure du simulateur présent sur le serveur

Afin de rendre la compilation et l'exécution de simulateurs plus simples, deux petits scripts ont été implémentés. La partie contrôleur exécute donc ces scripts afin de lancer des simulations.

Dès lors, la partie simulateur crée des simulateurs de circuits à l'aide de fichiers .logic, et les exécute avec des fichiers de simulations .simu, à l'aide de scripts. Ces fonctions sont détaillées ci-après.

5.2.3.4.5.2 Fonctions du simulateur

La première fonction de la partie simulateur est la création de simulateurs de circuits. Pour cela, il faut fournir un fichier décrivant l'architecture du circuit au code Java. Celui-ci créera un fichier Circuit.cpp et un fichier Circuit.h dans home/user/simulator/src.

Vous pourrez observer un exemple très simple d'un fichier circuit ci-contre. Remarquez la déclaration d'éléments logiques imbriqués, tels que la porte OR

```
LIB lib1{

        bloc1(a1,a2;s1){
                LOGIC
                orGate(a1,a2;s1);
        }

        orGate(a,b;s){
                LOGIC
                s = a+b;
        }
}
```

Code 17: Exemple de fichier .logic

utilisée par le bloc1. Ce fichier permet de décrire divers élément logiques du circuit, pour en savoir plus, voir l'essai de Luc.

Une fois le code C++ créé par le code Java commun, il ne reste plus qu'à le compiler afin d'obtenir le simulateur. Cette compilation fait appel à la librairie C++ contenant le code commun, préalablement compilée et située dans common/simulator/lib.

```
START_TIME 0
END_TIME 2000
CIRCUIT_NAME bloc1
WATCHLIST s1

EVENT a1 F 0
EVENT a2 F 0

EVENT a1 T 50
EVENT a2 T 50

EVENT a1 F 650

EVENT a2 F 1000
```

Code 18: Exemple de fichier .simu

Le résultat de cette compilation est un exécutable se trouvant dans home/user/simulator/bin. C'est le simulateur du circuit pris pour exemple ci-dessus. Il reste à donner un fichier de simulation à cet exécutable. Celui du code 18 convient au circuit par exemple.

En effet, le circuit observé s'appelle bloc1, et il est défini dans le fichier .logic. De plus, il n'y a qu'un fil à observer pendant cette simulation, s1, qui est la sortie du bloc1. Nous avons ensuite les évènements des fils a1 et a2, qui sont les entrées du circuit comme le montrent les paramètres du bloc1.

Le programme afficherait alors ce qui est contenu dans le code 19. Si vous observez bien, cela correspond bien à la sortie de la porte OR, avec un délai de 100.

L'autre fonction de la partie serveur est obligatoirement d'utiliser ce programme. Pour cela, deux scripts sont disponibles à tous les – futurs – utilisateurs dans common/scripts. Ce sont des scripts sh, exécutables dans un environnement GNU/Linux.

Le script create_simulator.sh est responsable de la création du code C++ du simulateur et de sa compilation. Il prend comme argument l'utilisateur et le nom du fichier .logic, de façon à aller chercher le fichier, puis de créer et compiler les fichiers C++ au bon endroit.

```
START_TIME 0
END_TIME 2000
EVENT s1 F 100
EVENT s1 T 150
EVENT s1 F 1100
```

Code 19: Exemple de sortie du simulateur

Le deuxième script, simulate_save.sh, s'occupe d'exécuter le simulateur du circuit. Pour cela, il a besoin de l'utilisateur, du nom du fichier exécutable, et du nom du fichier de simulation. Ainsi, il peut exécuter le simulateur avec tel ou tel fichier de simulation. Ce script sert aussi à enregistrer le résultat dans un fichier qui sera placé dans home/user/out, et dont le nom sera le même que celui de simulation.

La dernière fonction à voir est celle qui lie le contrôleur à ces scripts. Sans perdre de temps, laissez-moi vous montrer les deux fonctions du contrôleur en lien avec la partie simulateur.

```
createAndSaveSimulator(circuit: Circuit) {
        const circuit_filename = circuit.path.split('/').pop();
        execSync(`cd simulator/common/scripts && ./create_simulator.sh user1 ${circuit_filename}`);
        const simulator_path = `simulator/home/user1/simulator/bin/${circuit_filename}`;
        if (fs.existsSync(simulator_path)) {
                circuit.simulator_path = simulator_path;
                this.circuits_service.update(circuit); // save circuit simulator in database
        } else throw new InternalServerErrorException("Error in circuit simulator creation");
}

executeAndSaveSimulation(circuit: Circuit, simulation: Simulation) {
        const circuit_filename = circuit.path.split('/').pop();
        const simulation_filename = simulation.path.split('/').pop();
        execSync(`cd simulator/common/scripts && ./simulate_save.sh user1 ${circuit_filename} $
                {simulation_filename}`);
        const result_path = `simulator/home/user1/simulator/out/${simulation_filename}`;
        if (fs.existsSync(result_path)) {
                simulation.result_path = result_path;
                this.simulations_service.update(simulation); // save result simulation in database
        } else throw new InternalServerErrorException("Error in executing and saving simulation");
}
```

Code 20: Fonctions du contrôleur relatives au simulateur

Dans chacune de ses fonctions, une commande système est exécutée grâce à la fonction execSync. L'exécution des scripts se fait donc de façon synchrone, car il est souhaitable d'attendre la fin de la commande pour passer à autre chose, au cas où une erreur se produisait.

Ensuite, que ce soit pour l'exécutable ou le résultat de la simulation, le fichier est enregistré dans la base de donnée au champ approprié, s'il existe. Concernant l'exécutable, c'est dans le champ simulator_path de la table circuit, et pour le résultat, result_path de la table simulation.

Nous voyons clairement ici le lien du contrôleur avec la base de données et le simulateur.

Pour conclure, la partie simulateur a trois fonctionnalités : le simulateur, les scripts utilisant ce simulateur, et leur exécution par le contrôleur. Or, pour être opérationnel, cette partie a quelques dépendances importantes que nous abordons juste après.

5.2.3.4.5.3 Dépendances du simulateur

La partie simulateur est parfaitement fonctionnelle, mais dans l'environnement approprié. Dès lors, nous allons prendre un peu de temps pour voir ses dépendances dans cette section. Nous verrons qu'il a besoin d'un environnement GNU/Linux, Java et C++.

5.2.3.4.5.3.1 GNU/Linux

GNU/Linux est un système d'exploitation libre créé dans les années 90, dont le noyau (Linux) a été – et continue à l'être – développé majoritairement par Linus Torvald. Ce noyau est utilisé par GNU, un ensemble de programmes basés développés et maintenus par Richard Stallman et le projet GNU. L'association de ces deux projet a donné naissance à un système d'exploitation open-source (même si les *drivers* restent propriétaires) puissant et configurable, utilisé sur la majorité des serveurs Web et maintenant par de nombreux particuliers.

Il a été mentionné dans les sections d'avant que des scripts sh étaient exécutés par le contrôleur. Leur exécution ne marche que sur un système GNU/Linux. En effet, le lancement de ces scripts et les commandes qui y sont exécutées sont issues de programmes GNU. Ainsi, l'environnement de base est une système d'exploitation GNU/Linux, qu'il soit natif, dans une machine virtuelle ou dans un *container* Docker.

Cet environnement rend l'exécution des scripts possibles, mais le programme de Luc Parent a aussi ses propres dépendances.

5.2.3.4.5.3.2 Java

Java, langage orienté objet, est exécutable sur tout système d'exploitation car il a sa propre machine virtuelle, la Java Virtual Machine (JVM). Ainsi, tout système ayant cette JVM d'installée – et les librairies nécessaires - peut lancer un programme Java. Cet environnement Java est contenu dans OpenJDK, la version libre du Java Development Kit d'Oracle.

La partie du programme de Luc qui crée les fichiers C++ à partir d'un fichier .logic est un exécutable Java. En effet, la compilation du code se fait avec l'outil javac, tandis que son exécution requiert la commande java. Voir le script create_simulator.sh pour vérifier cela.

De plus, le créateur de simulateurs requiert la librairie ANTLR. Ce générateur de parseur qui rend l'analyse syntaxique des fichiers .logic possible. Tout cela est décrit avec précision dans l'essai de M. Parent. Le chemin de cette librairie doit être inclus dans la variable d'environnement CLASSPATH traitée par Java.

Dès lors, la partie du simulateur qui crée le code des simulateurs demande qu'un environnement Java soit présent et configuré de sorte que l'analyse syntaxique des fichiers décrivant les circuits soit faite par ANTLR. Mais le programme de Luc ne se limite pas à cela, car le code créé est du code C++, et doit être compilé.

5.2.3.4.5.3.3 C++

C++ est un langage de programmation assez flexible que j'aime placer entre le C et le Java. C'est la programmation procédurale et la rapidité du C combinée à la programmation orientée objet du Java. Comme le C, ce langage doit être compilé et n'utilise pas de machine virtuelle.

C'est la seconde partie du simulateur, celle qui exécute les simulations, qui est codée en C++. En effet, le code C++ commun et celui généré par le code Java doivent être compilés afin de créer le simulateur du circuit. Pour cela, il existe un Makefile pour le code commun et un pour les fichiers générés. Le code commun est compilé une seule fois en une librairie, tandis que le code spécifique au circuit est compilé à chaque fois qu'il est généré. Vous pouvez vérifier cela dans le script create_simulator.sh, où la commande make est lancée dans le dossier où se trouvent le code C++ créé. Ce qu'il faut retenir, c'est que cette partie a besoin de commandes de compilation de code C++, soit make et gcc.

En termes de conclusion, la partie du simulateur se charge de la création de code C++ grâce à un analyseur syntaxique qui lit les fichiers .logic, puis compile ce nouveau code en exécutable, qui lui, lit un fichier de simulation afin de lancer une simulation sur le simulateur du circuit. Cela est automatisé par des scripts, qui sont appelés par le contrôleur. Tout cela nécessite un environnement GNU/Linux avec Java, librairie ANTLR, et outils C++.

A ce point, toutes les parties du serveur ont été vues. Pour rappel, la partie contrôleur traite les requêtes venant d'internet et fait appel aux autres parties, qui sont les services gérant la base de données, stockant des informations sur les fichiers ; l'extracteur, qui crée des variables en lisant les fichiers de simulations et les modifie ; et enfin le simulateur, qui se charge d'exécuter et enregistrer des simulations.

Sachez que nous sommes aussi à la fin des sections sur les grandes parties du projet, qui sont l'interface et le serveur. Vous avez pu comprendre que l'une était accessible sur tout navigateur internet, tandis que l'autre était en arrière plan, en attente d'instructions, de requêtes. Ces deux parties forment ainsi la totalité du projet, soit un simulateur Web de circuits logiques. La prochaine section traite de la mise en production de ce projet.

5.3 Production

Tout ce que nous avons vu jusqu'à présent est basé sur l'environnement de développement. Pour rappel, afin de tester le projet localement (sur localhost), il faut déployer le service de la base de donnée avec `docker-compose` sur le port 3306 , puis déployer le serveur avec `yarn nx serve`

`server` sur le port 8080 , et finalement mettre en service l'interface avec `yarn nx serve interface` sur le port 4200.

Mais cela reste un environnement local, la finalité est que ce projet soit déployée sur internet et donc accessible à tous. Pour cela, il faut que tout un chacun puisse entrer l'adresse de l'interface dans la barre de recherche de son navigateur, et accéder à celle-ci. Dans le cas de l'environnement de développement, l'interface est accessible via http://localhost:4200, c'est tout. Tenter d'y accéder depuis un appareil sur le même réseau local en cherchant http://ip-local-machine-dev:4200, avec ip-local-machine-dev du style 192.168.0.x, ne fonctionnera pas. Alors y accéder depuis le Web, n'en parlons pas. Le seul moyen serait d'ouvrir le port 4200. A ce moment, tout le monde pourrait accéder à l'interface, mais cela reste un environnement de développement.

Pour passer en mode production, il faut de nouveau faire appel à Nx, plus pour déployer mais pour compiler les applications cette fois. En effet, `yarn nx buid interface` et `yarn nx build server` vont générer le code de production dans le dossier dist/ du projet. Ce sera ensuite à nous d'exposer le code de l'interface via un serveur Web et de gérer le lancement du serveur via Node, ce que faisait Nx avec la commande serve.

C'est là qu'intervient de nouveau Docker. Deux fichiers sont présents à la racine du projet : Dockerfile.interface et Dockerfile.server. Ils servant à générer des *images* Docker qui pourront être mise en service par la suite. Ils contiennent les instructions pour compiler et déployer leur code respectif. Par exemple, celui de l'interface récupère une *image* d'un serveur Web NGINX qui est configuré de sorte que le code React compilé de l'interface soit exposé sur le port 80. Concernant l'*image* du serveur, elle-aussi compile son code, déploie le serveur avec une commande node, et l'expose sur le port 8080.

Or, le serveur a de nombreuses dépendances qu'il faut satisfaire. Il se base donc sur une autre *image* dont le fichier s'appelle Dockerfile.env. La dépendance de base de la partie serveur du projet est un environnement GNU/Linux, cela explique la première ligne, qui est `FROM debian`. Ensuite, pour compiler le serveur, nous avons besoin de Yarn et de Node. Ces deux paquets sont récupérés des dépôts Debian via `apt`, leur gestionnaire de paquet.

Puis, comme dans l'environnement de développement, les paquets Yarn listés dans package.json doivent être téléchargés, et c'est Yarn qui s'en occupe.

Les dernières dépendances sont Java et les outils C++, sans lesquels le simulateur ne fonctionnerait pas. Vous remarquerez aussi la variable d'environnement CLASSPATH contenant le chemin vers la librairie ANTLR.

Bien, à ce stade le code de production de l'interface et du serveur peuvent être déployées sur n'importe quelle machine qui a Docker et une connexion internet. Il manque cependant le service de la base de données.

Afin, de déployer tous ces services d'un coup, un autre fichier docker-compose existe. C'est docker-compose.prod.yml. Vous pouvez y observer le service de la base de données, celui de l'interface et celui du serveur. Un autre s'appelle traefik, il joue le rôle de routeur entre internet et les services du projet. Pour plus d'informations sur cet outil, veuillez visiter ce lien.

Cet environnement de production n'est pas encore totalement fonctionnel. En effet, l'adresse du serveur est écrite dans le code de l'interface, mais elle devrait dépendre de l'environnement. En développement, c'est localhost, alors qu'en production, elle devrait être modifiée automatiquement.

D'autre part, il faudrait se pencher sur la persistance de la base de données en utilisant un volume défini par exemple.

Enfin, si l'environnement de production était opérationnel, le projet pourrait être déployée sur une machine dont le port 80 serait ouvert. Ainsi, tout le monde pourrait accéder à la merveilleuse interface depuis un navigateur, uploader des fichiers et exécuter des simulations. Cette machine pourrait être une de l'université ou un serveur privé virtuel (VPS). Ce serait aussi pratique d'acheter un nom de domaine redirigeant vers l'adresse de l'interface plutôt que de taper l'adresse IP directement dans le navigateur.

Après de nombreuses pages, nous voici arrivés à la fin de la description du projet. Comme vous avez pu le constater, après avoir décrit le projet et présenté ses dépendances globales, nous sommes passé à ses différentes parties. Celles-ci étant l'interface et le serveur, et pour chacun d'entre elles, une description, une liste de dépendances et le détails de leurs parties respectives a été donné. Et ainsi de suite jusqu'à arriver aux plus petites parties qui avaient besoin d'être présentées.

Maintenant que le projet a été décrit dans son intégralité, il est important de vérifier qu'il est fonctionnel. La partie suivante sert justement à cela.

6 Test des fonctionnalités

Dans la mesure où tout le projet a été présenté dans la section précédente, il ne nous reste plus qu'à vérifier que les fonctionnalités sont bien présentes. Pour cela, nous allons nous placer du côté utilisateur, qui fait diverses actions depuis l'interface. Dès lors, nous allons nous pencher sur les fonctions d'upload, de sélection, de modification, et de suppression de fichiers. Puis, sur le lancement d'une simulation, la sélection d'une intervalle de celle-ci, le déplacement de cette intervalle, et la sélection de signaux.

6.1 Upload de fichiers

Une fois la base de donnée, le serveur et l’interface déployée, nous accédons à l’interface via un navigateur. Pour uploader des fichiers, il faut ouvrir l’onglet Circuits ou Simulations et cliquer sur le bouton « + ». Si le projet est vierge, aucun fichier ne devrait être listé, comme le montre la capture d’écran ci-dessous.

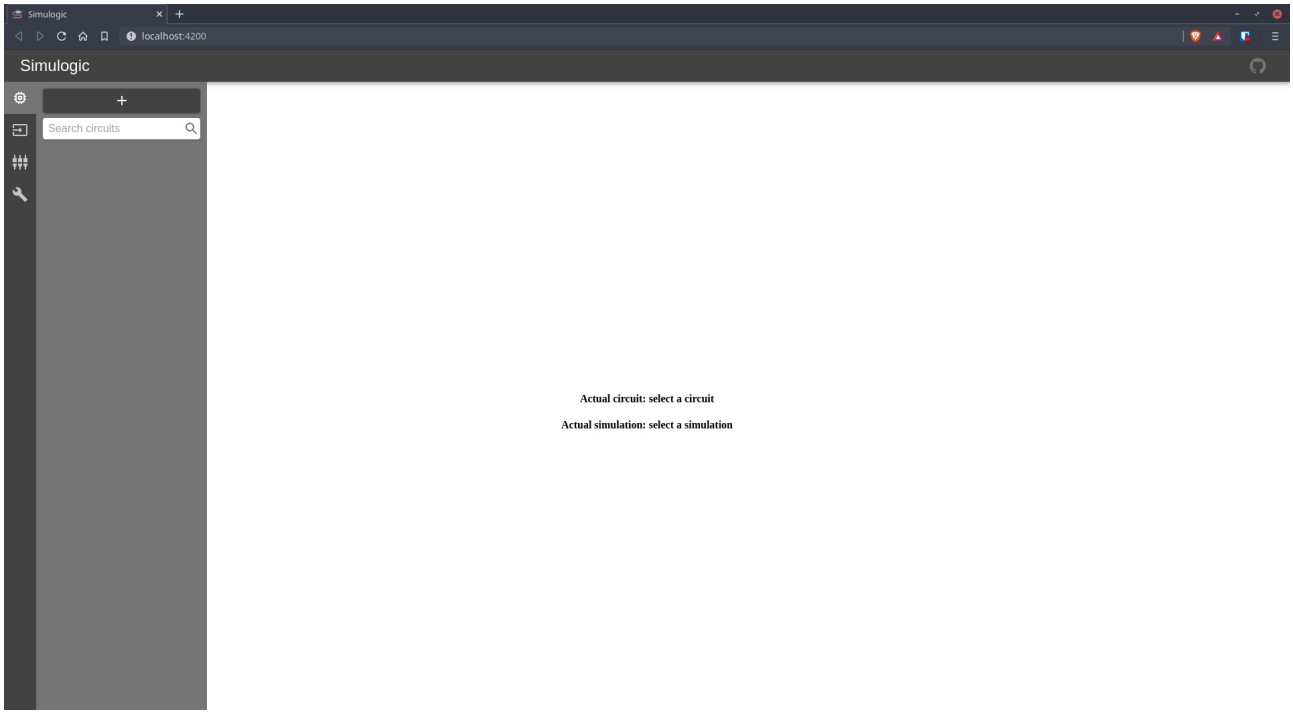

Figure 23: Interface avec l'onglet Circuits sélectionné et aucun fichier uploadé

Maintenant, lorsqu’on clique sur le bouton d’upload, votre explorateur de fichiers devrait s’ouvrir, vous demandant de sélectionner un fichier .logic (.simu pour les fichiers de l’onglet Simulations). Voir la capture d’écran suivante, où l’on choisit d’uploader deux fichiers d’un coup.

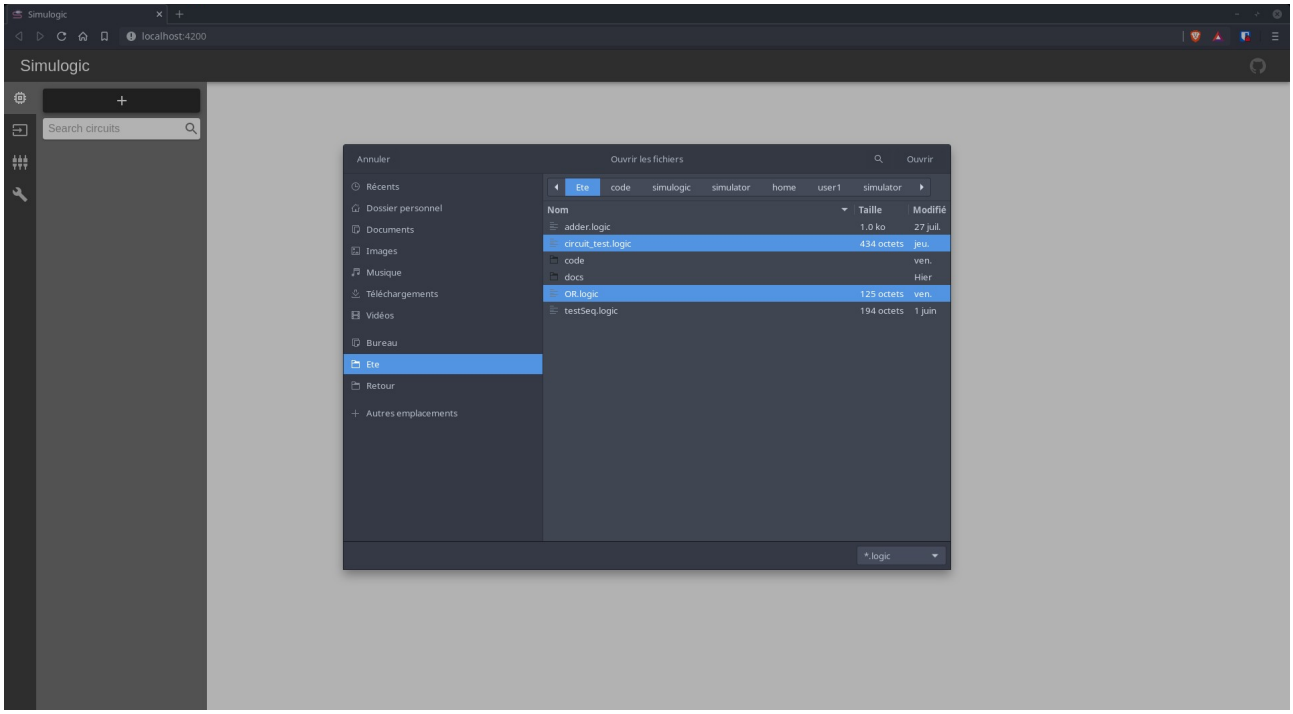

Figure 24: Sélection des fichiers .logic à uploader

Les nouveaux fichiers, s'ils ont bien l'extension .logic, sont alors listés dans l'onglet Circuits :

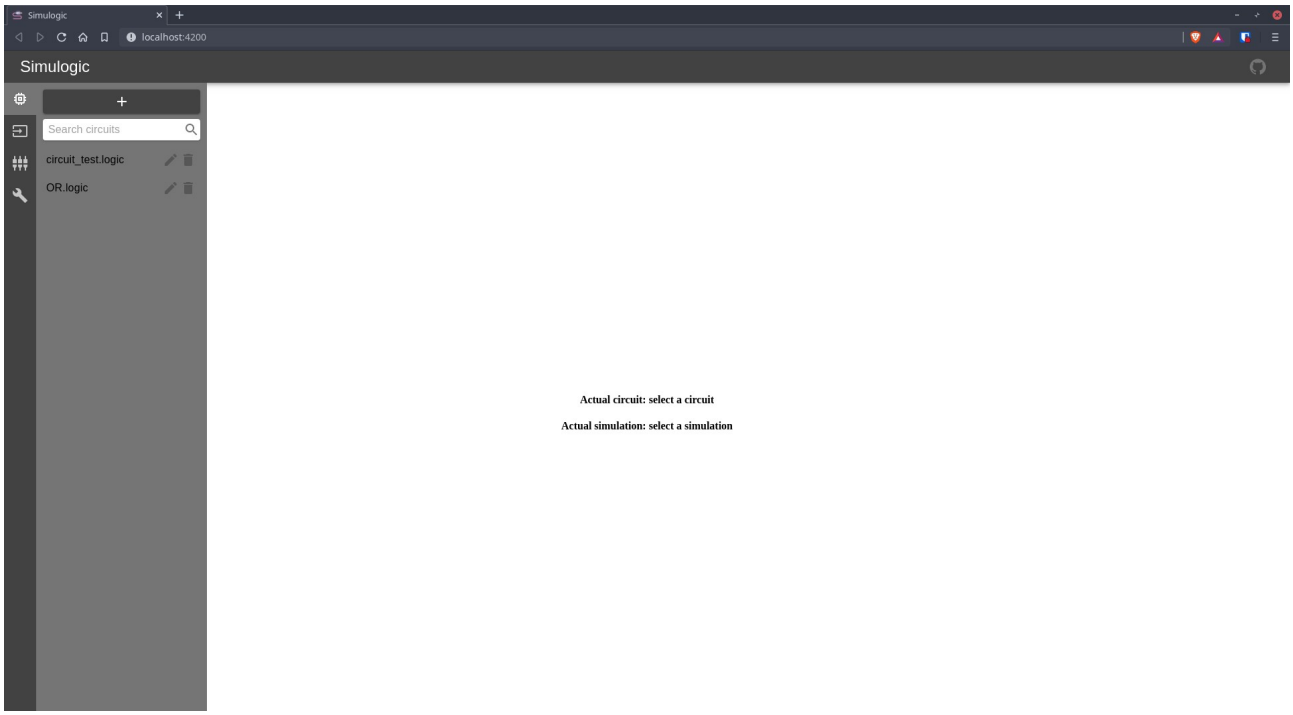

Figure 25: Onglet Circuits liste les fichiers .logic uploadés

C’est exactement le même processus pour les fichiers de simulations, sauf que c’est l’onglet Simulations – juste en dessous – qui s’en charge. La seule différence est l’extension qui est .simu.

Voilà pour l’upload des fichiers, nous pouvons ensuite vérifier la fonctionnalité de leur sélection.

6.2 Sélection de fichiers

Pour pouvoir effectuer une simulation, le simulateur utilise un fichier décrivant une simulation et un autre décrivant le circuit soumis à cette simulation. Comment se fait cette sélection de fichiers alors ?

En restant sur l’onglet Circuits, il faut simplement cliquer sur un des fichiers pour le sélectionner. Il apparaîtra dans une couleur plus foncée, et le champ « Actual circuit » de l’espace de travail affichera son nom, comme vous pouvez le vérifier dans la figure 26.

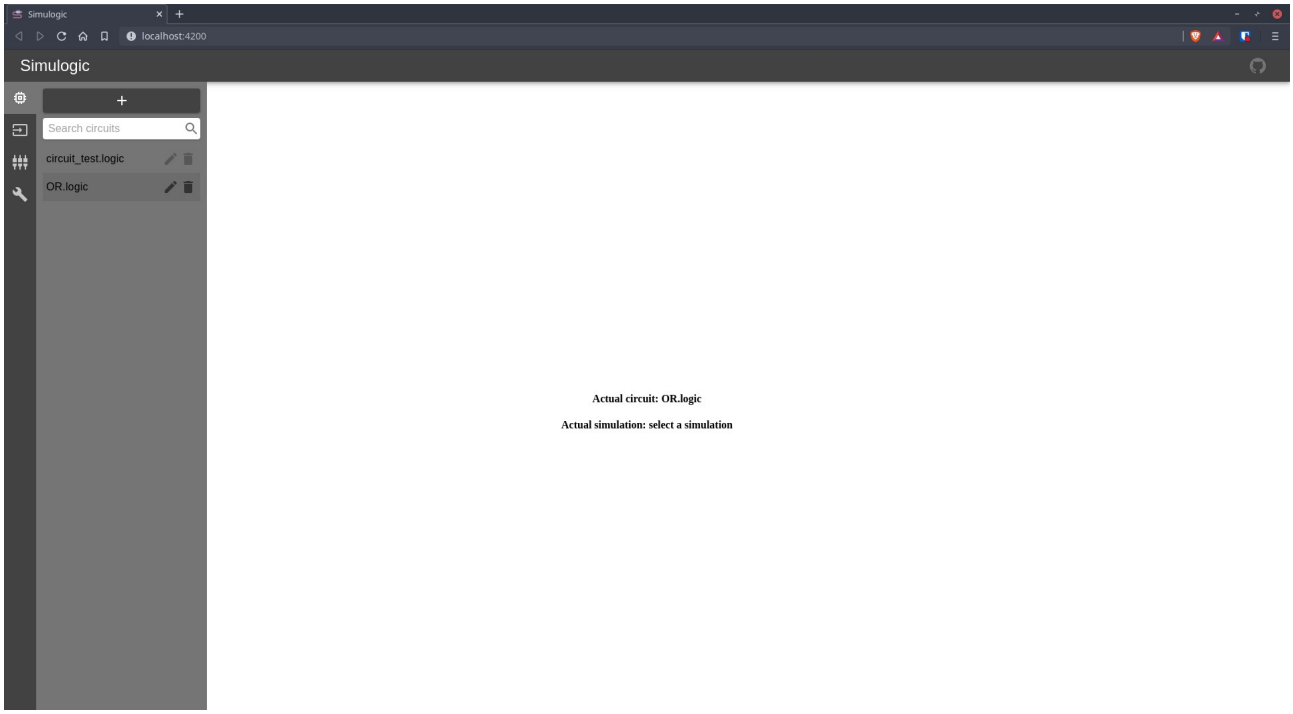

Figure 26: Sélection d'un fichier dans l'onglet Circuits

Le processus est le même pour les fichiers de simulation, mais en plus, l'extracteur lit le fichier et renvoie la variable résultante afin qu'elle soit affichée par WaveDrom :

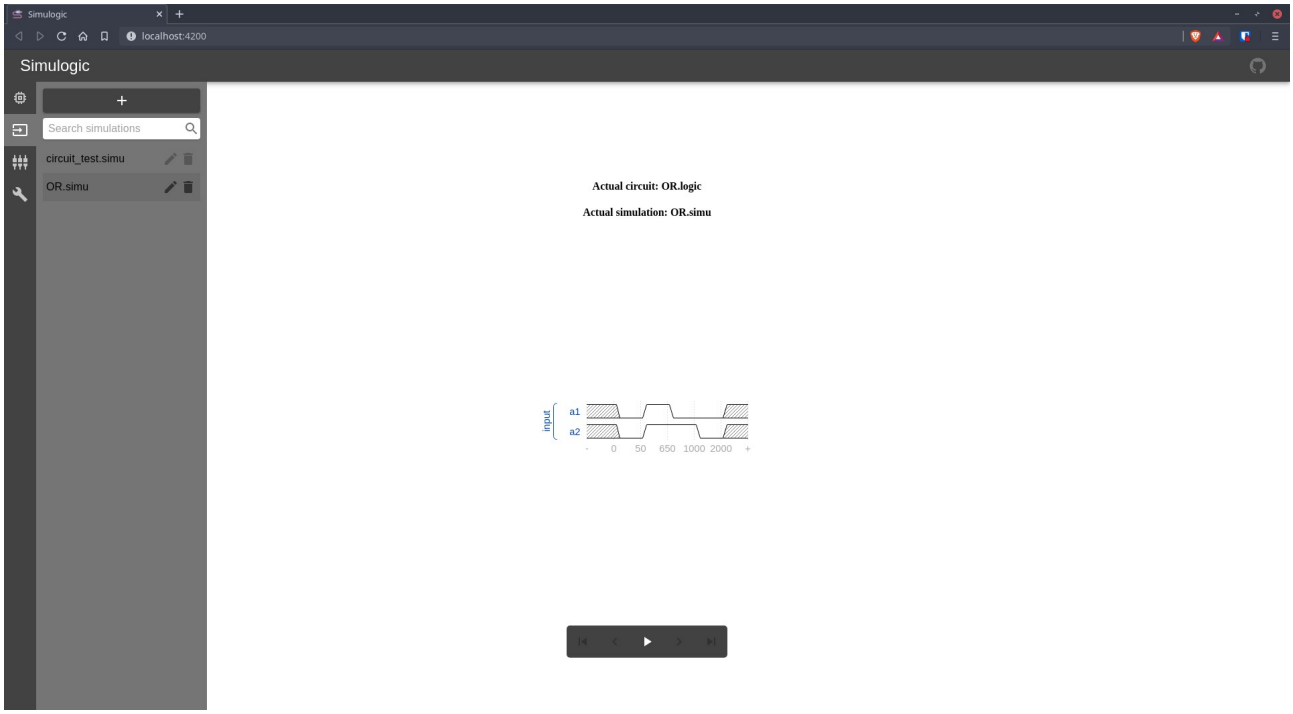

Figure 27: Sélection d'un fichier dans l'onglet Simulations

Vous remarquerez l'apparition du graphique WaveDrom et du Player juste en dessous. Oui, le graphique est minuscule, cela est dû à la simplicité du fichier de simulation.

La fonctionnalité de sélection de ces fichiers étant vue, qu'en est-il de leur modification ?

6.3 Modification de fichiers

La modification de fichier consiste simplement à le renommer. Afin de changer son nom, il faut d'abord avoir sélectionné le fichier, puis cliquer sur l'icône crayon juste à la droite du nom. Un champ de texte apparaît alors :

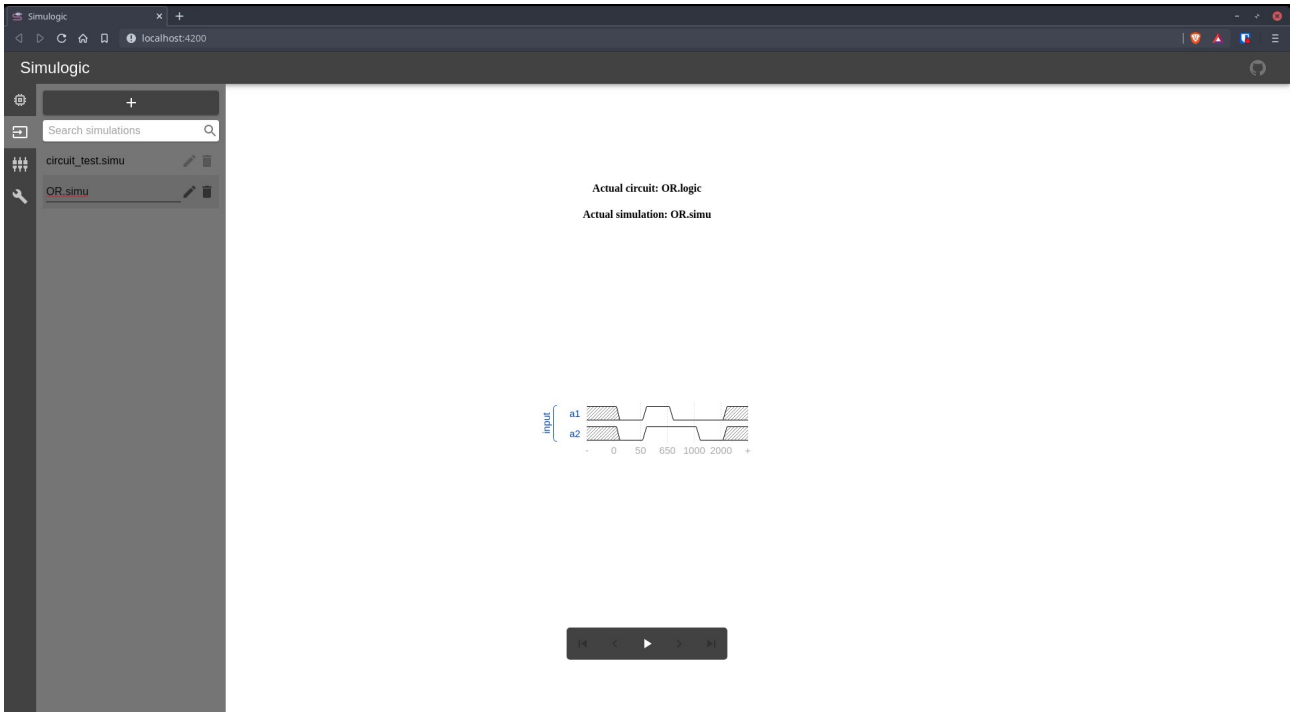

Figure 28: Modification du nom d'un fichier

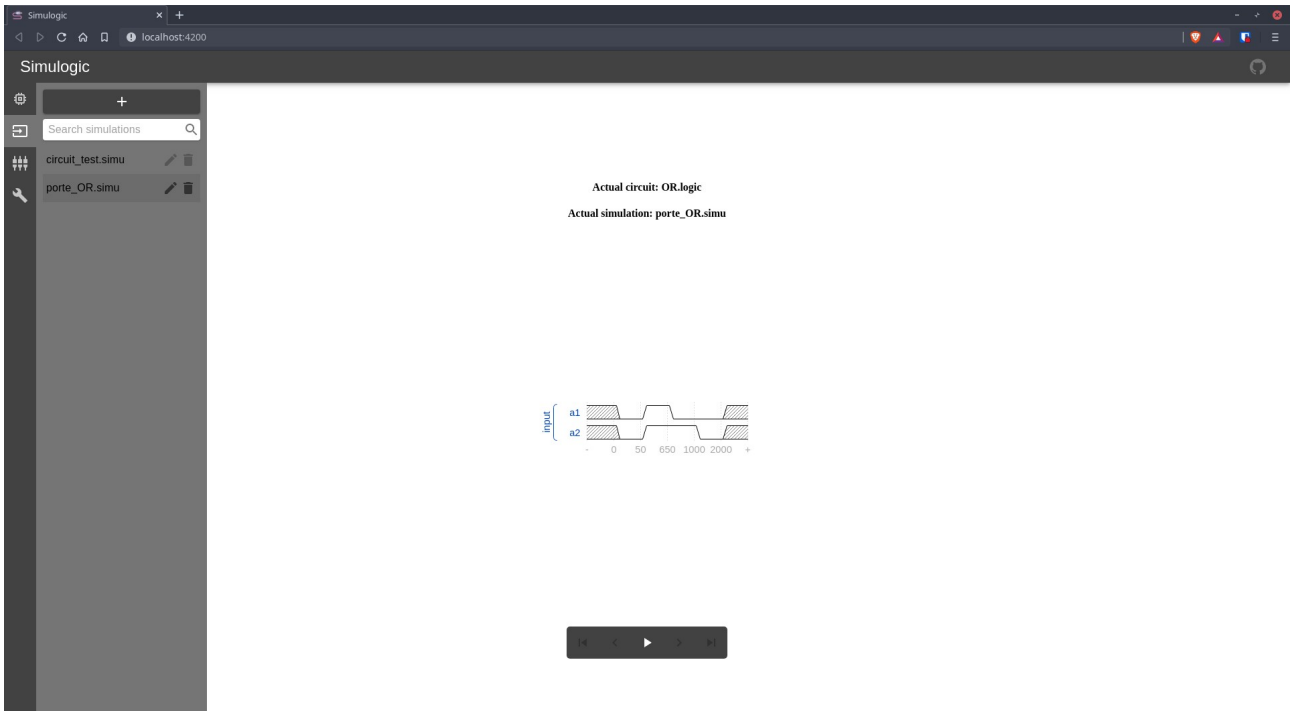

Figure 29: Fichier de simulation sélectionné renommé

Après avoir écrit le nouveau nom et tapé Entrer, le champ est modifié. En réalité, c'est le nom enregistré dans la base de données qui a été altéré.

Bien, il nous reste à voir, concernant la gestion des fichiers, la suppression.

6.4 Suppression de fichiers

L’utilisateur a peut-être uploadé un mauvais fichier, ou ne veut plus s’en servir, il souhaite alors le supprimer. Il n’a qu’à cliquer sur l’icône poubelle à droite du bouton de modification pour le faire disparaître du serveur.

Dans cet exemple, nous allons supprimer les fichiers circuit_test. En sélectionnant celui de simulation, et en cliquant sur le bouton supprimer, il disparaît de la liste et comme plus aucun fichier de simulation n’est sélectionné, le graphique et le Player ne sont plus visibles :

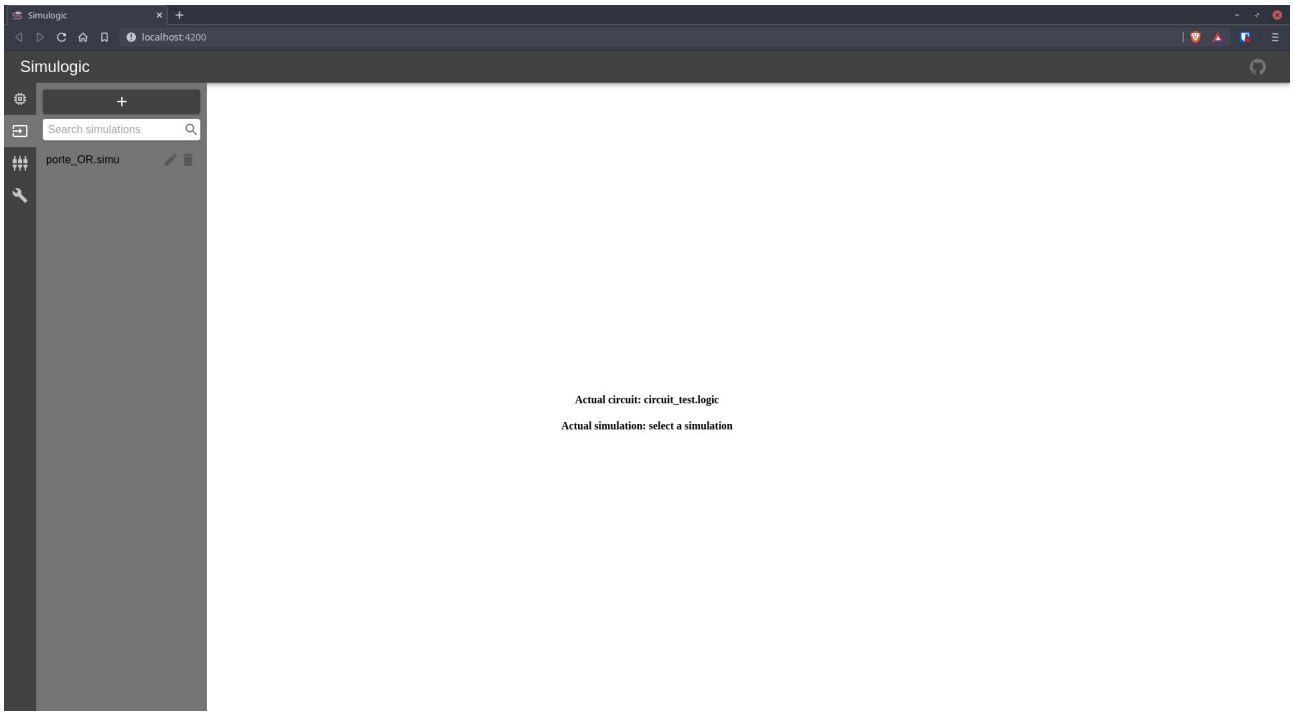

Figure 30: Interface suite à la suppression de la simulation sélectionnée

Pour supprimer le fichier circuit_test.logic, il faut se rendre dans l’onglet Circuits, et également cliquer sur l’icône en question.

Les fonctionnalités de gestion des fichiers ont été passées en revue, mais pas encore celle d’exécution de simulation.

6.5 Lancement de simulation

Quand un fichier de circuit et un de simulation sont sélectionnés, le bouton Play du Player est activé. En effet, il apparaît en blanc, comme dans la figure 27. Quand on clique dessus, la simulation est lancée. Puis quand la variable contenant les signaux d'entrée et ceux observés durant la simulation arrive à l'interface, elle est directement affichée. Voyez par vous-même dans l'image ci-après.

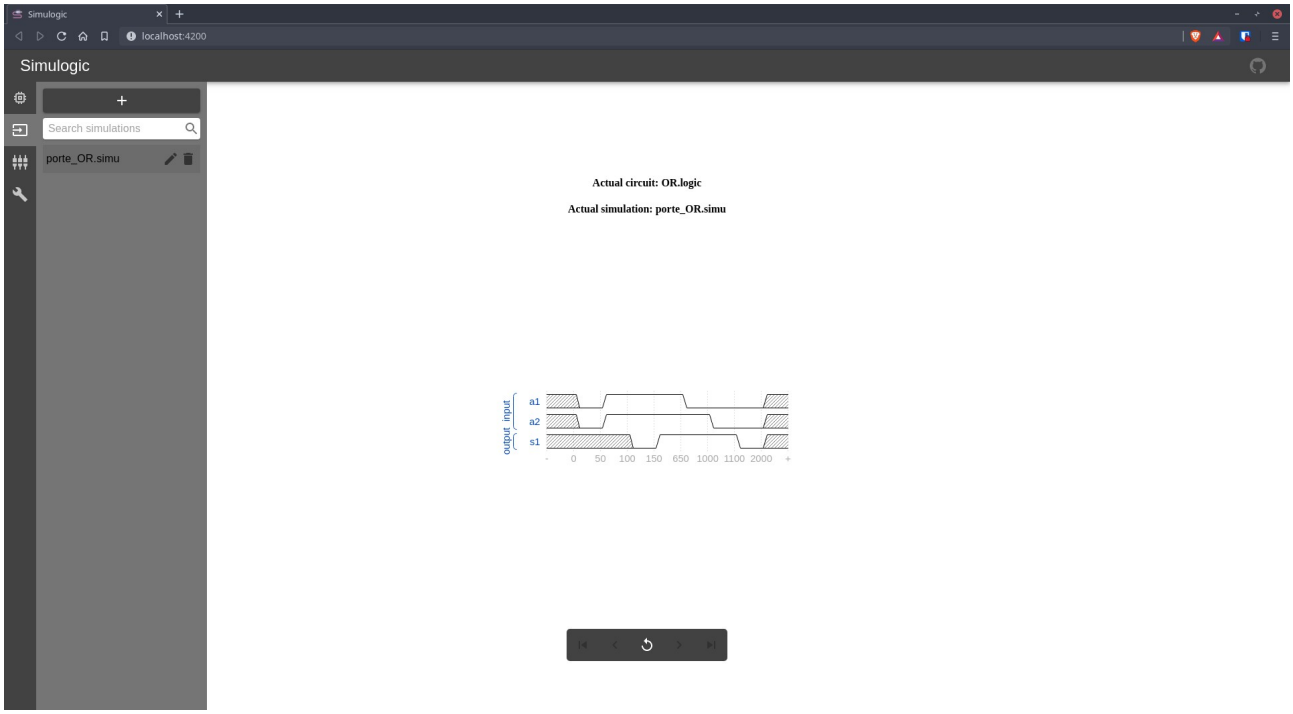

Figure 31: Résultat de l'exécution d'une simulation

Le bouton Play est remplacé par le bouton Reset qui revient à la variable de simulation initiale, celle ne contenant pas le groupe *output*.

Le lancement de simulation étant opérationnel, la section suivante s'attarde sur la fonctionnalité de sélection d'intervalle.

6.6 Sélection d'intervalle

L'utilisateur peut choisir de ne voir qu'une partie du graphique, pour cela il peut se rendre dans l'onglet Configuration, le dernier bouton du menu. Plusieurs champs y sont présent, c'est un formulaire donc il faut tous les remplir si l'on veut configurer totalement le comportement d'affichage du graphique. Dans notre cas, seule la sélection d'intervalle nous importe.

Les captures d’écran ci-dessous montrent l’état de l’interface avant et après l’envoi du formulaire.

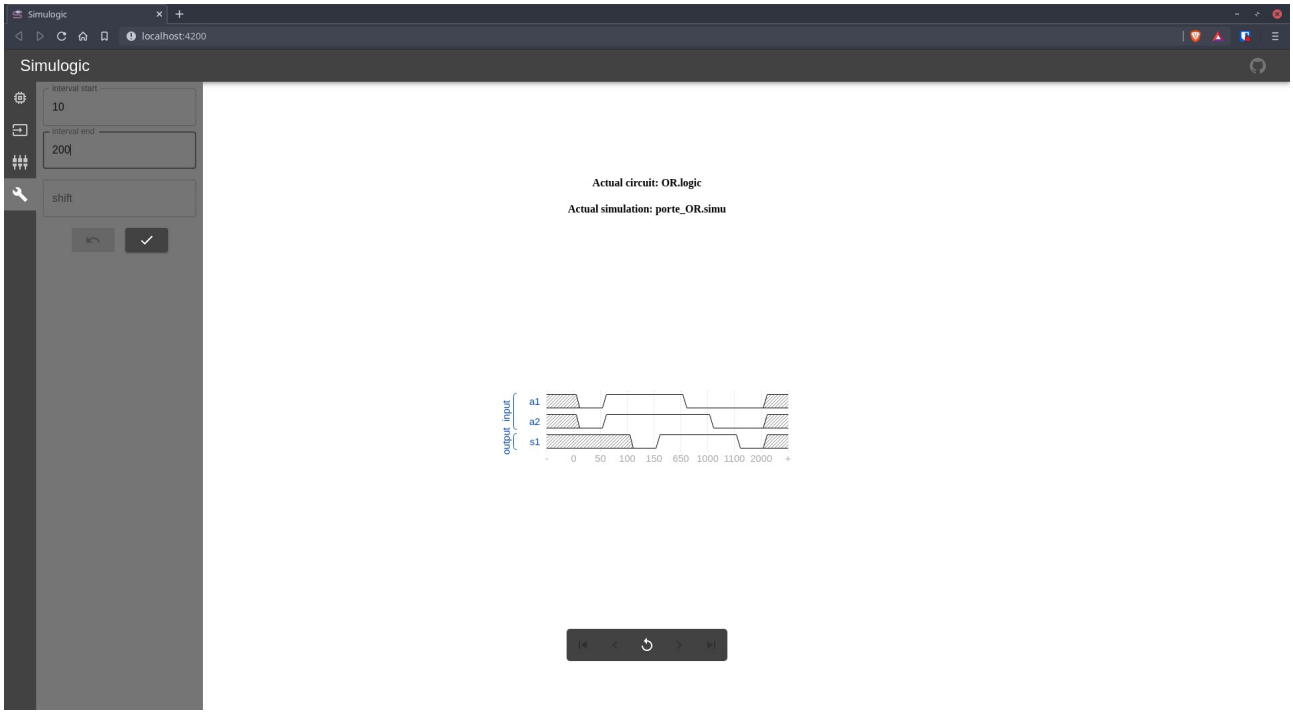

Figure 32: Interface avec les champs de sélection de l'intervalle remplis

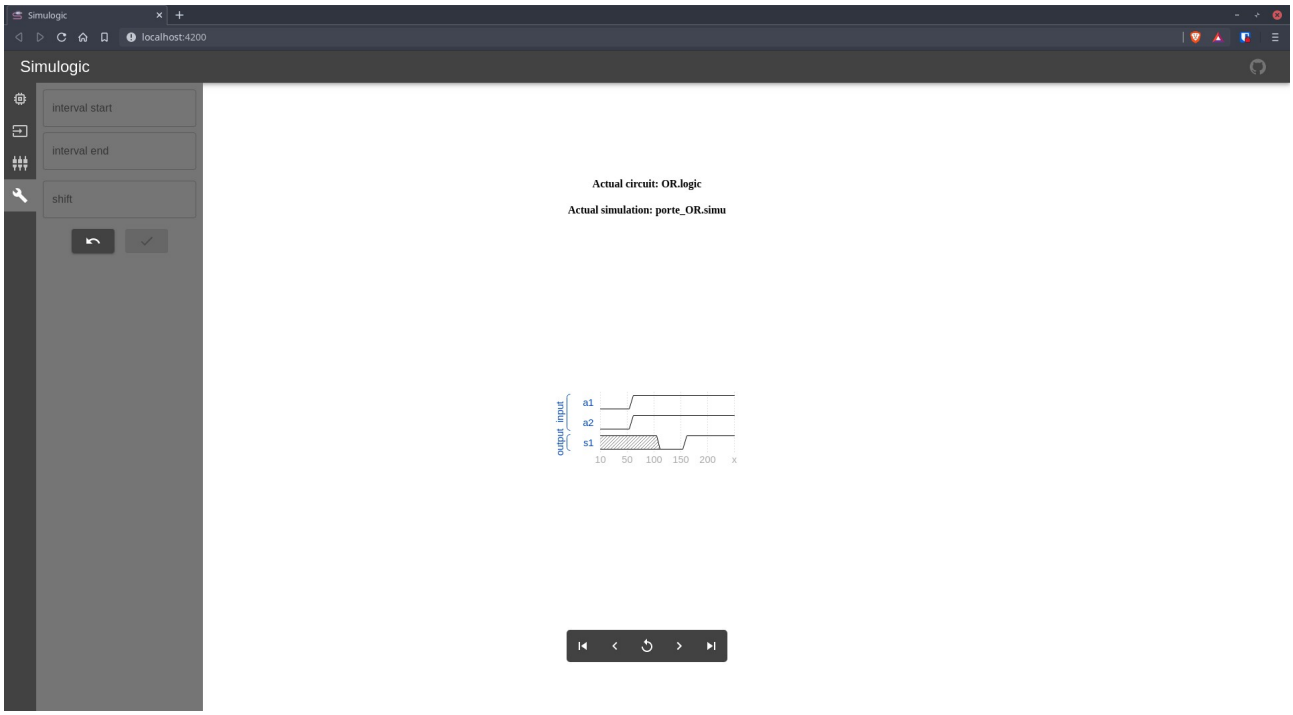

Figure 33: Interface une fois le formulaire de sélection d'intervalle envoyé

Le bouton Retour du formulaire s’active afin de revenir à la configuration initiale, le graphique ne présente que les évènements présents dans l’intervalle, et les boutons de déplacement de l’intervalle s’activent.

C’est tout pour la sélection d’intervalle, passons à son déplacement.

6.7 Déplacement d’intervalle

Quand une intervalle de la simulation a été sélectionnée via le formulaire dans l’onglet Configurations, le Player a la capacité de faire glisser cet intervalle pour afficher les évènements précédents ou suivants.

Reprenons l’état de l’exemple précédent avec une intervalle de 10 à 200. Quand on clique sur le bouton juste à droite du bouton Play, la nouvelle intervalle part de 200 et va jusqu’à 390. Comme il n’y a aucun évènement dans cet intervalle le graphique est presque vide.

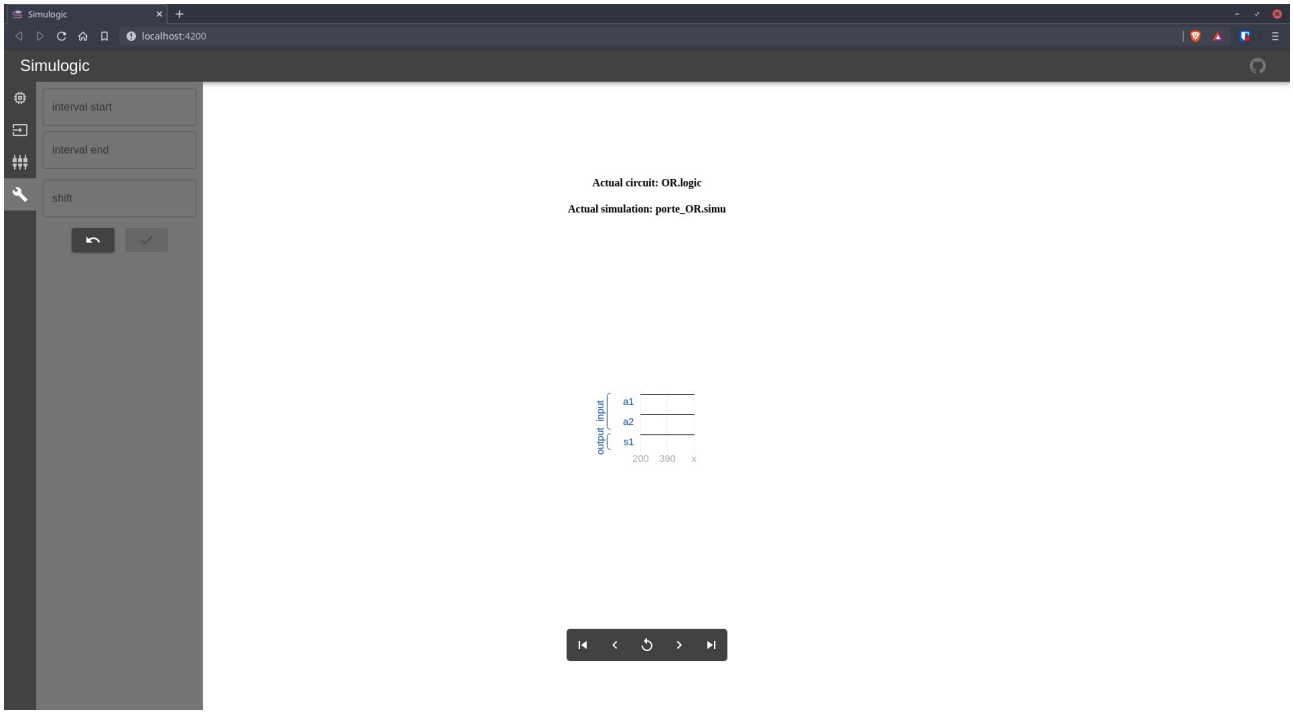

Figure 34: Interface suite à un shift vers la droite

Cette opération est appelée *shift*, qu’il soit vers la droite comme ci-dessus, ou vers la gauche. Si la valeur *shift* dans le formulaire est laissée vide dans le formulaire, alors la fin de l’intervalle (ou le début) devient le début (ou la fin) de la nouvelle intervalle.

Par contre, si l’on entre un nombre dans le champ *shift*, comme 100 par exemple :

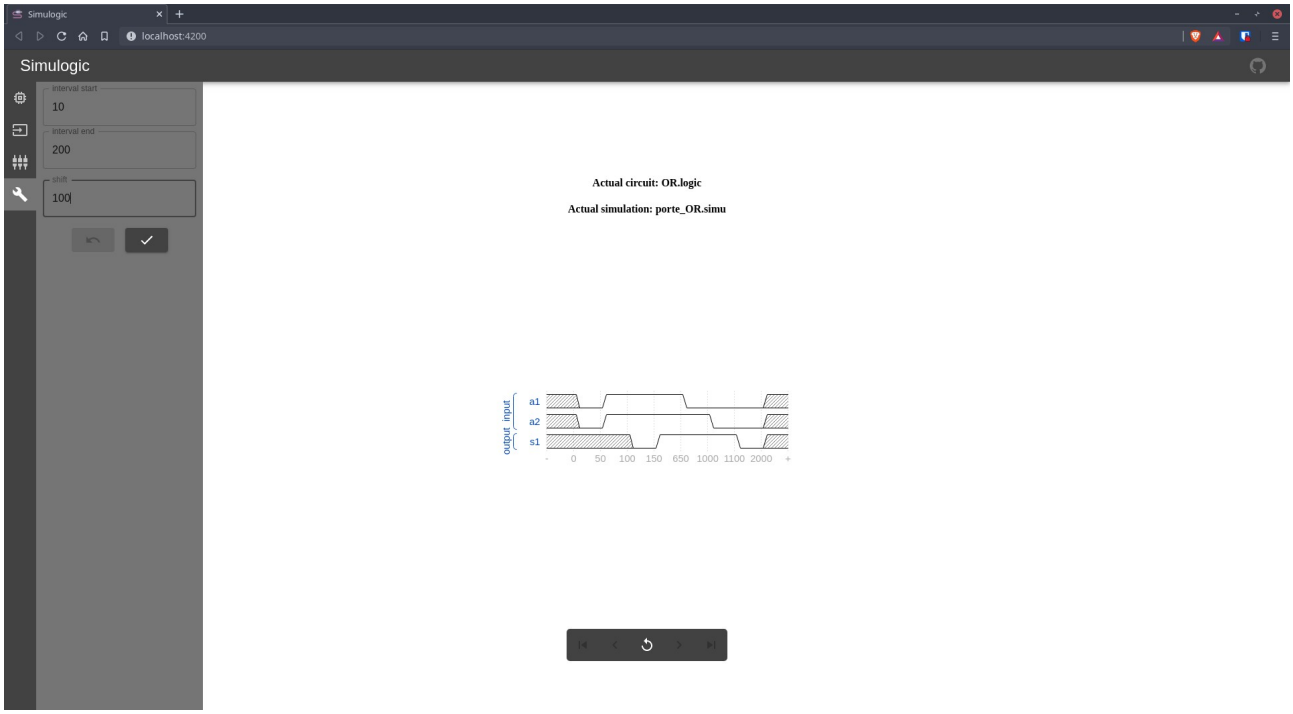

Figure 35: Interface avant l'envoi du formulaire complet de configuration

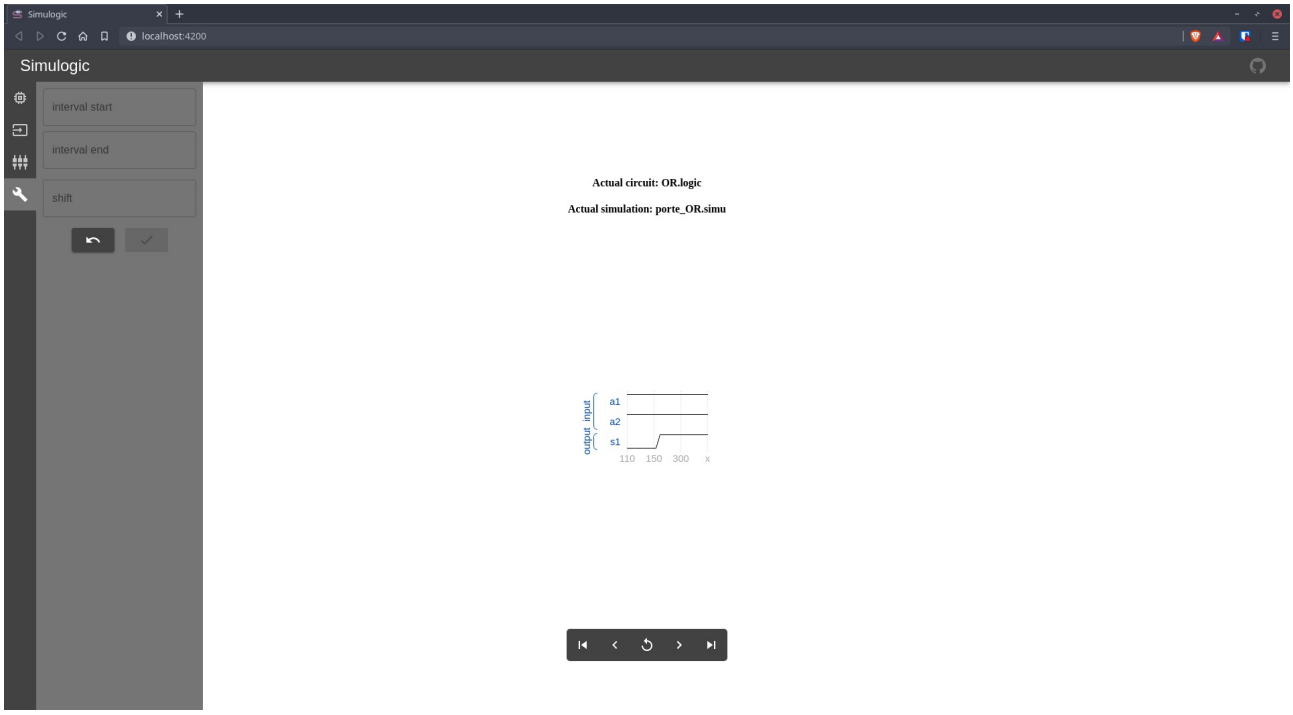

Figure 36: Interface après un shift de 100 vers la droite

Le graphique obtenu après l'envoi du formulaire est le même que celui de la figure 30. Mais quand on fait un *shift* vers la droite, le graphique est différent. En effet, l'intervalle se déplace de 100 cette fois-ci. Et donc, la nouvelle intervalle va de 110 à 300.

D’autre part, les boutons aux extrémités du p*layer* servent à déplacer l’intervalle au tout début ou à la toute fin de la simulation. Cette action s’appelle *full shift*. Donc si l’on reprend l’état précédent de l’interface et que l’on clique sur le bouton complètement à gauche, voici le résultat :

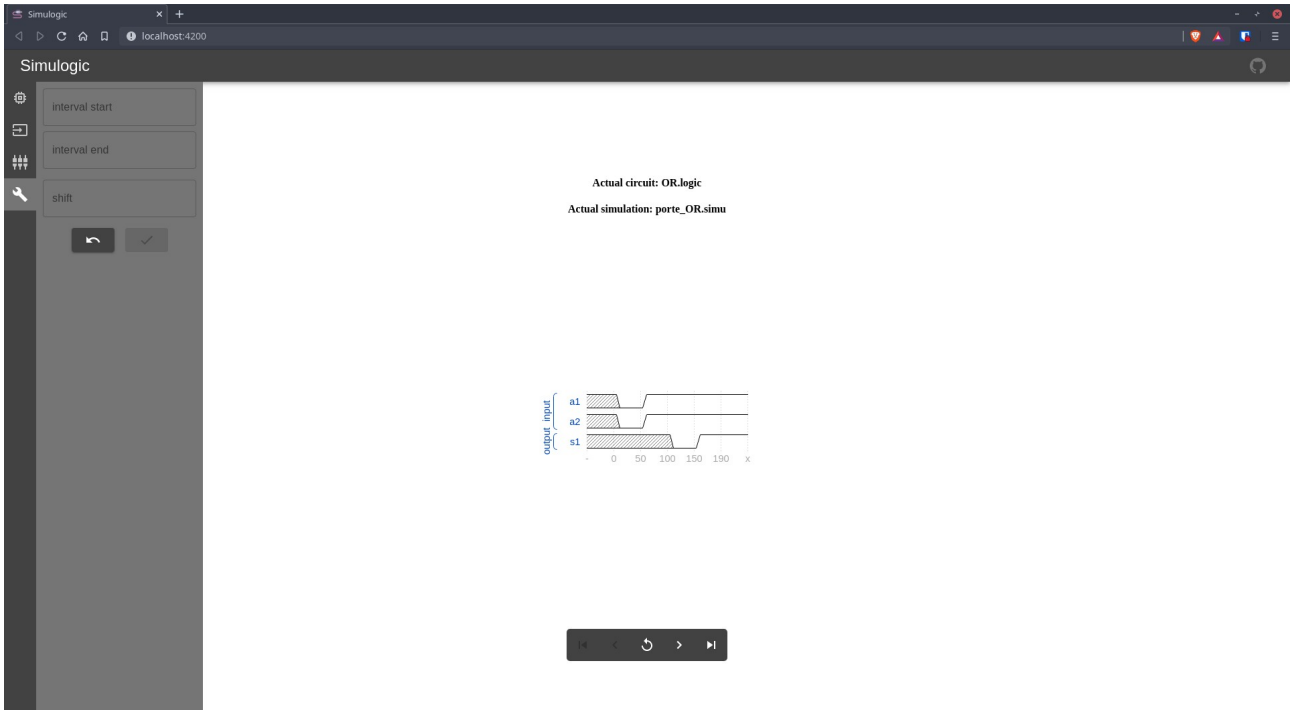

Figure 37: Interface après un full shift vers la gauche

Comme le début de la simulation est atteint, les boutons de *shift* vers la gauche sont désactivés. Après un *full shift* vers la droite, ce seront les boutons de *shift* vers la droite qui seront incliquables.

Comme vous avez pu le constater, ces opérations sont bien fonctionnelles. La prochaine fonctionnalité à tester est celle de sélection de signaux.

6.8 Sélection de signaux

Nous n’avons pas encore touché à l’onglet Signaux, qui se trouve juste au dessus de Configuration. Il sert à voir quels signaux sont présents dans le graphique, et offre la possibilité d’en cacher certains.

Voici son contenu dans l’état précédent de l’interface :

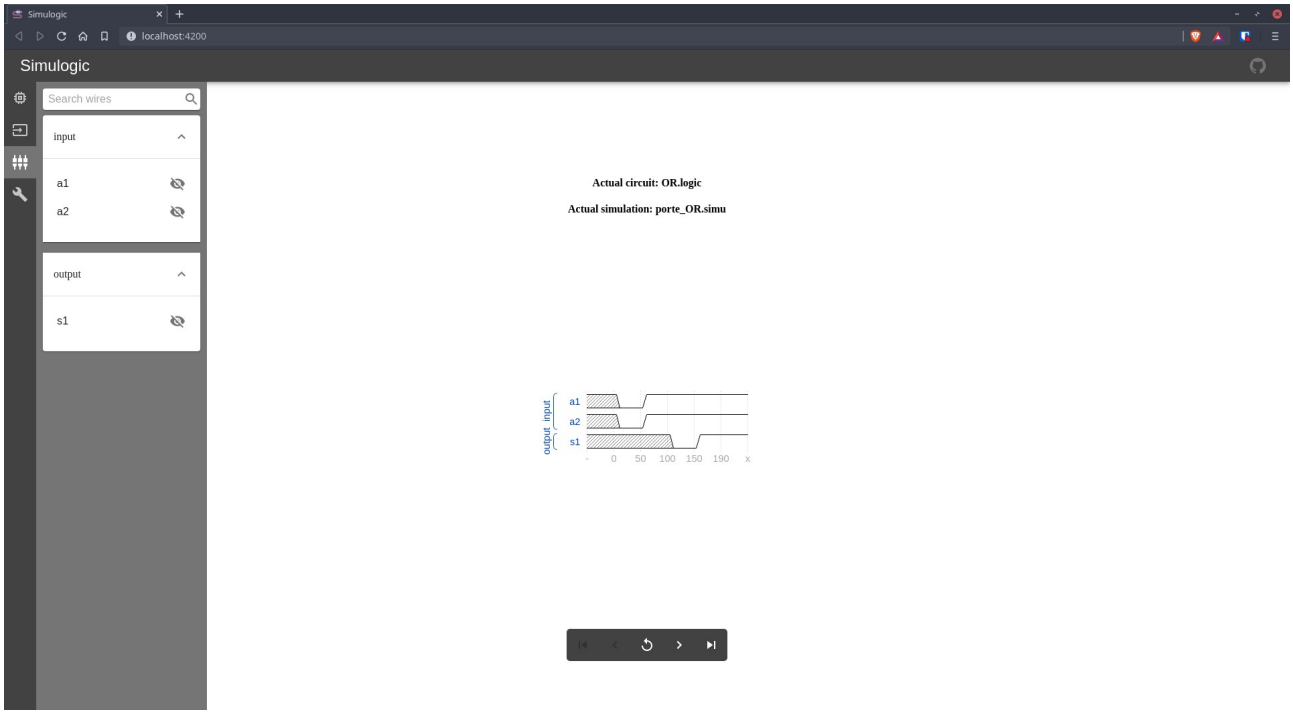

Figure 38: Interface avec l'onglet Signaux ouvert

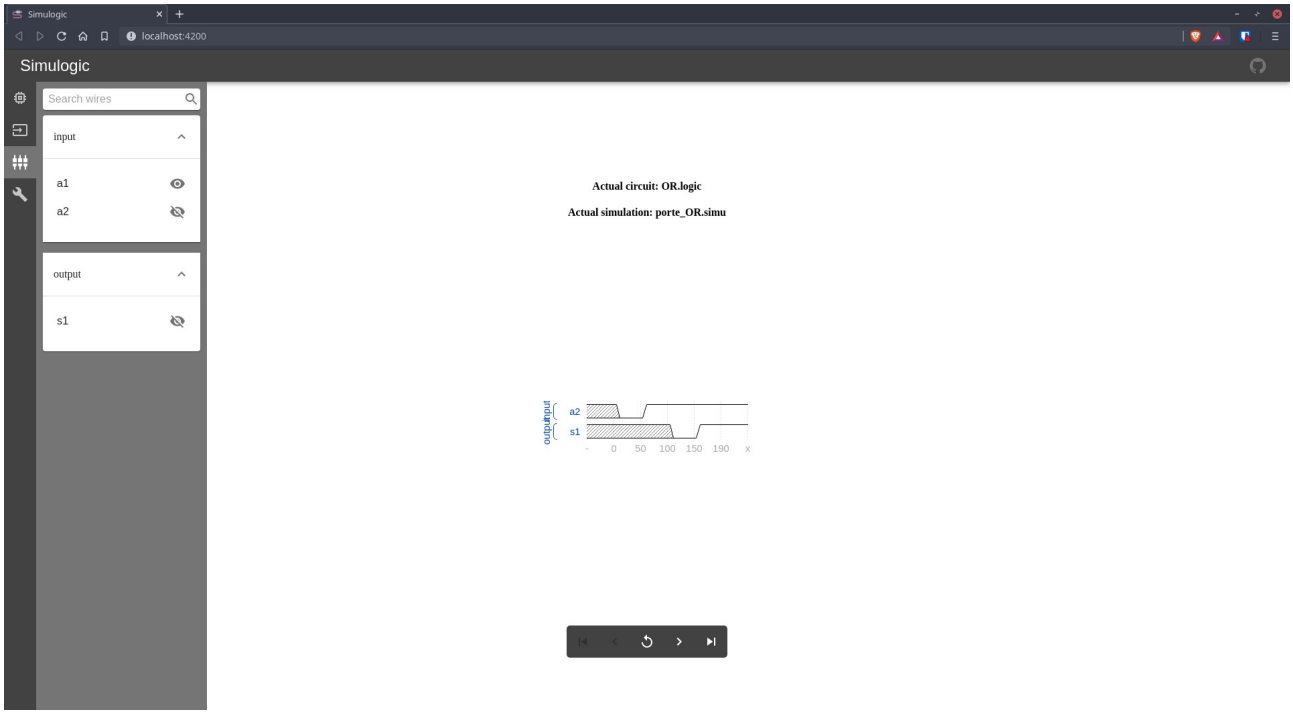

Figure 39: Interface avec le signal a1 caché

Les icônes à droite de chaque signal donnent un aperçu de leur action, cacher le signal. Après avoir cliqué sur celle du signal a1, il disparaît du graphique et son icône est changé pour celle représentant l'action d'affichage du signal.

La dernière fonctionnalité à explorer est celle de recherche, je vous renvoie donc à la section suivante.

6.9 Recherche

Vous avez dû remarquer les champs de recherche présents dans les onglets Circuits, Simulations et Signaux. Leur fonction est celle de rechercher des circuits ou des simulations dans la base données, ou des signaux dans la simulation actuelle.

Tout d'abord, concernant la recherche de signaux, reprenons l'état de l'interface où tous les signaux sont visibles. Quand on cherche l'expression « 1 », seuls les signaux a1 et s1 apparaissent dans le résultat de la recherche.

Figure 40: Résultat de la recherche de signaux contenant "1"

Faire une recherche d'expression vide fera réapparaître tous les signaux visibles. Ainsi, si certains signaux étaient cachés, ils ne seront listés ni dans les résultats de la recherche, ni dans la liste de tous les signaux. Il faudra cliquer sur le bouton Reset (et sur Play si c'étaient des signaux cachés du résultat) pour les faire réapparaître dans la liste.

En ce qui concerne la recherche de circuits et de simulations, le principe est le même sauf que la recherche a lieu dans la base de données. Prenons comme acquis que plusieurs fichiers .logic et .simu ont été uploadés comme vous pouvez le constater dans les images ci-après.

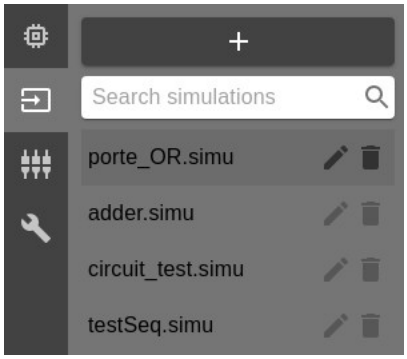

Figure 41: Onglet Simulations montrant plusieurs fichiers .simu uploadés

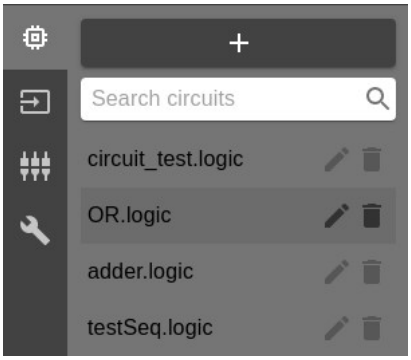

Figure 42: Onglet Circuits montrant plusieurs fichiers .logic uploadés

Lorsqu’on va chercher l’expression «test», il faut s’attendre à ce que seuls soient affichés ceux contenant cette chaîne de caractères. Et effectivement, c’est le cas d’après les deux captures d’écran suivantes.

Figure 43: Onglet Simulations après la recherche "test"

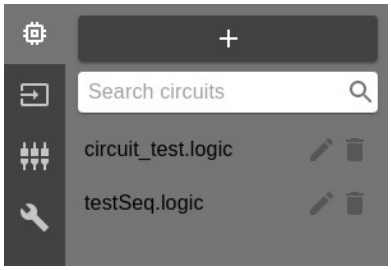

Figure 44: Onglet Circuits après la recherche "test"

Dans cette partie nous avons passé en revue les fonctionnalités du projet via l’interface utilisateur. Nous avons vu qu’il était possible d’uploader, sélectionner, modifier et supprimer des fichiers de circuits et de simulations ; d’exécuter une simulation, de n’afficher qu’une intervalle et de la déplacer, de choisir les fils visibles, et enfin de rechercher des circuits, des simulations, ou des signaux.

7 Conclusion

Pour conclure, dans cette définition de projet d’essai nous avons tout d’abord expliqué le contexte, c’est-à-dire qu’il s’agit de définir mon projet de fin de Maîtrise, ayant lieu à l’université de Sherbrooke, qui consiste à développer une interface Web permettant de simuler des circuits logiques classiques. La mise en contexte a aussi permis de comprendre que mon travail s’inscrivait dans un projet plus large, lui-même visant à simuler des circuits quantiques et hybrides.

Nous avons ensuite passé du temps à voir quels projets similaires existaient déjà, et il en ressort qu’il y a de nombreux logiciels de simulation de circuits logiques classiques, peu de circuits quantiques, et aucun mêlant les deux. Mon projet a donc des concurrents, mais pas le projet global car aucun ne propose les fonctionnalités dont il disposera. Il était aussi important de citer ma lecture du livre de Don Norman, « The Design Of Everyday Things », qui m’a apporté des connaissances sur le design et dont je me suis servi durant l’élaboration de l’interface.

Puis, la description du projet a été l’occasion de voir en profondeur chacune de ses parties, dont chacune était détaillée par une description, une présentation de ses fonctions, ses dépendances, et ses propres parties. En effet, les deux grandes applications du projet sont l’interface et le serveur. L’interface peut être découpée en deux, le menu et l’espace de travail. Le serveur, lui, est plus

complexe. Sa partie contrôleur traitant les requêtes peut faire appel à la base de données, l'extracteur ou encore le simulateur. La plupart de ces modules dépendent d'outils spécifiques, mentionnés précédemment, en plus des dépendances de leur parent.

Et pour terminer, la section de test des fonctionnalités a récapitulé, en image, toutes les actions que peut effectuer l'utilisateur sur l'interface, gérées en arrière plan par le serveur. On retrouve la gestion des fichiers de circuits et de simulations, l'exécution d'une simulation, la sélection d'une intervalle et son déplacement, l'affichage ou non de signaux de la simulation en cours, et la recherche de circuits, de simulations, et de signaux.

Si nous voulions avoir un aperçu du futur du projet, je pourrais dire qu'il reste à voir la partie mise en production qui n'est pas tout à fait aboutie, et la gestion d'utilisateurs qui est manquante. De plus, la gestion des erreurs seraient la bienvenue, avec peut-être des *pop-up* contenant une indication du problème. Une interface peut toujours être plus intuitive, alors pourquoi ne pas la pousser au maximum.

Pour conclure cet essai, le projet de développement du simulateur Web de circuits logiques est arrivé à son terme. Toutes les fonctionnalités qui permettent la simulation depuis un navigateur sont opérationnelles grâce à une interface minimale et intuitive, et à un serveur simple et rapide. Toute personne souhaitant continuer ce projet devrait avoir la compréhension nécessaire suite à la lecture de ce document.