UNIVERSITE LIBRE DE BRUXELLES
Faculté des Sciences appliquées

Année académique 2007- 2008

Evolution de design de circuit logique.

Directeur de Mémoire : H.Bersini
Superviseur : C.Philemotte

Mémoire de fin d'études
présenté par Yagoubi Khalid en
vue de l'obtention du titre
d'ingénieur civil informaticien.

Résumé

Ce travail est consacré au développement d'un framework logiciel pour l'implémentation d'algorithmes évolutionnistes dans le cadre de design de circuits logiques. La première partie de ce travail présentera les algorithmes évolutionnistes ainsi que les circuits logiques. Nous proposons ensuite un état de l'art de design de circuits évolutionnistes et finalement nous présenterons le framework qui a été développé. Nous avons tenté de valider le fonctionnement du framework ainsi que l'utilisation de telles techniques en comparant aux travaux réalisés par d'autres chercheurs.

Remerciements

Je tiens à remercier tout d'abord mes promoteurs, Hugues Bersini et Christophe Phillemote pour m'avoir encadré et conseillé durant toute l'année. Et, plus particulièrement Christophe qui a été un superviseur remarquable et qui m'a fait découvrir le monde de la recherche. Sans ses précieux conseils, sa patience et ses réponses, ce travail n'aurait jamais pu voir le jour. Je le remercie d'avoir sacrifié de son temps pour m'offrir l'aide dont j'avais besoin et d'avoir pu me faire franchir les obstacles.

Je tiens à remercier Dragomir Milojevic et Emanuel Falkenauer pour m'avoir éclairé sur certains points de mon travail et d'avoir mis en évidence certaines questions essentielles.

Je remercie également la personne qui a du relire maintes fois mon mémoire et qui a supporté mes innombrables fautes d'orthographes sans en perdre son latin.

Et finalement, je remercie ma famille qui m'a soutenue durant cette épreuve ainsi que mes amis pour m'avoir redonné le moral quand celui-ci n'était plus au rendez-vous.

Table des matières

Table des figures

Introduction

Les circuits logiques et numériques ont envahi nos vies, du téléphone mobile qui ne nous quitte jamais à l'ordinateur dernier cri que certains n'arrivent plus à quitter. Le fonctionnement de tous ces objets est basé sur de simples opérations binaires qui sont effectuées à l'aide de circuits numériques manipulant des bits.

L'électronique numérique a connu une fulgurante évolution durant ces cinquante dernières années : de la naissance du simple transistor en 1948 aux processeurs actuels qui en contiennent des milliards. Les circuits numériques se basent pour la plupart sur des portes logiques qui sont la traduction électronique des opérateurs binaires. Les techniques traditionnelles de conception de circuits logiques telles que Karnaugh, Quine Mc Cluskey, ESPRESSO, sont maîtrisées depuis des dizaines d'années. Cependant, la demande continue de circuits plus complexes et plus performants rendent ces techniques impraticables.

C'est ainsi que sont nées les techniques d'abstraction de haut niveau tels que les langages de description matériel laissant la phase de synthèse et d'optimisation à des logiciels basés sur des systèmes experts.

Ces derniers sont malheureusement souvent soumis à la propriété intellectuelle et à des secrets industriels.

Dès lors, certains chercheurs se sont mis en quête de nouvelles techniques de synthèse et d'optimisation autres que celles citées précédemment.

Ils proposèrent entre autres d'appliquer les techniques évolutionnistes au design de circuits logiques ce qui a donné naissance à une nouvelle discipline : le *design évolutionniste.*

Ce sont à la base des techniques d'optimisation inspirées du principe de l'évolution dont les mécanismes ont été décrits par Charles Darwin dans son ouvrage *l'Origine des espèces.*

Les techniques conventionnelles se basent sur une approche dite top-down. Un système complexe est subdivisé en de nombreux sous-systèmes plus faciles à résoudre.

L'approche évolutionniste est tout le contraire, tout le système recherché est généré en un seul bloc. Ce qui pourrait donc nous fournir des circuits plus optimisés, dénués de redondance.

Néanmoins, les techniques évolutionnistes se limitent à l'heure actuelle à des

circuits de quelques dizaines d'entrées à cause de l'effort computationnel qu'elles requièrent. Par contre, les techniques conventionnelles peuvent traiter des circuits possédant jusqu'à une centaine d'entrées, notamment Quine Mc Cluskey.

Cependant, certains travaux proposent d'utiliser de telles techniques comme moteur d'innovation. En effet, elles permettraient d'obtenir des designs encore inconnus par l'homme dont nous pourrions en extraire les principes et les appliquer à plus grande échelle.

Dans le cadre de ce mémoire, nous avons développé un framework logiciel pour l'implémentation des algorithmes évolutionnistes dans le cadre de design de circuits logiques. La souplesse et la modularité furent les mots d'ordre durant le développement.

Ce travail est découpé en cinq grands chapitres.

Tout d'abord, nous invitons le lecteur à découvrir le monde des algorithmes évolutionnistes afin d'avoir les bases pour la compréhension du reste du rapport. Ensuite, au deuxième chapitre, nous aborderons quelques notions des circuits logiques ainsi que les caractéristiques de quelques techniques de synthèse conventionnelles.

La fusion des deux disciplines donne naissance au principe de design évolutionniste que nous aborderons, donc, dans le troisième chapitre. Nous y présenterons les mécanismes ainsi que les travaux qui furent réalisés dans ce domaine.

Le quatrième chapitre est consacré à l'analyse et l'implémentation du framework. Nous y détaillerons les différents mécanismes ainsi que les points importants du framework. Nous présenterons également des cas d'utilisation.

Pour valider le bon fonctionnement du framework, nous avons réalisé quelques expériences que nous avons confrontées aux résultats obtenus dans la littérature concernant le sujet.

Et finalement une conclusion de mon travail sera fournie.

Chapitre 1

Algorithmes évolutionnistes

Contents

1.1 Introduction

Ce chapitre a pour but d'introduire le lecteur au monde des algorithmes évolutionnistes. En premier lieu, il est nécessaire d'expliquer ce qu'est l'optimisation et quelles sont les techniques associées à cette discipline. En second, nous dresserons une brève taxonomie des algorithmes d'optimisation pour enfin nous concentrer sur le coeur du chapitre : les algorithmes évolutionnistes.

Après avoir décrit les principes de bases des algorithmes évolutionnistes ainsi que les variantes de ce type d'algorithmes, nous nous focaliserons sur les algorithmes génétiques.

Pour terminer, nous aborderons les sujets satellites qui gravitent autour des algorithmes génétiques, notamment l'hybridation d'algorithmes.

1.2 Introduction à l'optimisation

1.2.1 Définitions

Les définitions suivantes sont extraites de [41].

Définition 1 *Variables*
Ce sont les entrées du problème étudié.

Définition 2 *Alphabet*
Ensemble des valeurs que peut prendre une variable.

Définition 3 *Solution*
Ensemble de variables instanciées dans l'alphabet du problème.

Définition 4 *Fonction objective*

Fonction mathématique associant à chaque solution une valeur. La fonction objective est basée sur un modèle exacte du problème étudié.

Définition 5 *Fonction de coût*

La fonction de coût est identique à la fonction objective à la condition près que l'on cherche à la minimiser.

Définition 6 *Espace de recherche*

Ensemble de toutes les solutions d'un problème donné.

Définition 7 *Optimum*

Valeur maximale ou minimale que peut prendre une fonction réelle.

Définition 8 *Optimum local*

Optimum défini sur un voisinage. Il existe, donc, d'autres points qui seraient plus optimum.

Définition 9 *Optimum global*

Optimum défini sur tout le domaine de définition de la fonction.

1.2.2 Les problèmes d'optimisation

Goldberg dans son ouvrage *Genetic Algorithm* [17], propose de voir un problème d'optimisation comme étant une boîte noire pourvue d'un certain nombre d'actionneurs et dont la sortie est une valeur réelle.

Les actionneurs, auxquels il est possible d'attribuer un certain nombre de valeurs, constituent les variables du problème. Ce dernier, par extension, peut posséder plusieurs sorties et, donc, produire comme sortie un vecteur de valeurs. Les problèmes auxquels s'intéresse l'optimisation sont des problèmes possédant plusieurs solutions, voire une infinité de solutions qui se différencient par la valeur de sortie.

L'optimisation a pour mission de trouver les valeurs des variables qui produisent la meilleure valeur de sortie possible.

1.2.3 Classification des problèmes

Le but de cette section est d'apporter des notions intuitives sur la complexité d'un problème afin de sensibiliser le lecteur aux problèmes d'optimisations. Nous nous basons sur les points bibliographiques[41].

La théorie de la complexité est la discipline informatique qui tente de fournir une idée de la difficulté d'un problème ou des performances d'un algorithme. Elle catégorise les problèmes en diverses classes selon leur difficulté de résolution.

Un problème de classe P est dit facile car il existe au moins un algorithme déterministe qui est prouvé pouvoir le résoudre en un temps polynomial. C'est le cas du produit de deux nombres, du tri, du plus court chemin dans un graphe, etc.

Les problèmes dit NP-difficiles n'ont pas encore à ce jour d'algorithmes déterministes capables de les résoudre en un temps polynomial. C'est le cas du problème du voyageur de commerce, de la satisfiabilité booléenne, de la planification , du bin packing, ect.

Définition 10 *Temps polynomial*

Temps de calcul est proportionnel à N^n, où N désigne le nombre de paramètres inconnus du problème, et n est une constante entière[10].

Malheureusement, les problèmes les plus pertinents dans de nombreux domaines sont des problèmes NP-Difficile : de la conception de réseaux et de la télécommunication à la réalisation d'emplois du temps en passant par le calcul de la tournée de véhicules, etc.

En générale, on se contente de solutions sous optimales obtenues par des algorithmes approchés tels que les heuristiques ou les métaheuristiques (cf.1.4 et 1.5)

L'exemple classique est celui du voyageur de commerce qui désire visiter n villes en ne passant qu'une seule fois par chacune et en minimisant le trajet parcouru. Le nombre de trajets possibles est de $\frac{(n-1)!}{2}$. Un rapide calcul nous montre que ce nombre explose rapidement avec le nombre de villes :

Nombre de villes	Nombre de trajets
5	12
10	181440
15	43 * 10^9

TAB. 1.1 – Explosion combinatoire du nombre de circuits possibles.

Déjà pour 20 villes, la résolution du problème par une méthode déterministe est impossible en terme de temps. Ceci est dû à l'explosion combinatoire dont souffre le problème du voyageur de commerce.

1.3 Techniques d'optimisation

Cette section a pour but d'introduire le lecteur aux diverses techniques d'optimisation. Il existe diverses méthodes d'optimisation que nous pouvons classer en deux groupes :

- Les méthodes ou algorithme déterministes
- Les méthodes non déterministes ou probabilistes

Un algorithme déterministe est un algorithme dans lequel l'état suivant du calcul est déterminé par l'état courant. A chaque pas d'exécution, il n'existe qu'une seule voie à prendre. Ce sont des algorithmes dans lesquels il n'existe aucun choix d'actions basé sur des probabilités.

Tandis que la deuxième classe possède des étapes dans lesquels le hasard intervient pour le choix de l'action à exécuter.

La question qui se pose consiste à déterminer quand faut-il privilégier l'utilisation des algorithmes déterministes aux dépends des algorithmes probabilistes, et vice versa.

Lorsque la structure du problème permet de mettre en évidence un lien entre la forme de la solution et la valeur de la fonction à optimiser, il peut être intéressant d'appliquer des algorithmes déterministes.

Dans le cas contraire, c'est-à-dire lorsque le problème ne permet pas de trouver une relation entre la solution et la valeur de la fonction objective ou que la dimension de l'espace de recherche est trop grande, utiliser des algorithmes dé-

terministes s'avère très difficile. En générale, cela reviendrait à une recherche exhaustive qui, dans certains cas, peut être énorme.

Nous proposons de classer les algorithmes d'optimisation selon la fig.1.1 qui est inspirée de [41].

Nous nous intéressons en particulier à l'étude des algorithmes évolutionnistes qui sont des métaheuristiques manipulant des populations de solutions. Les méthodes que nous aborderons dans la suite de ce mémoire apparaissent en couleur orange sur la figure 1.1.

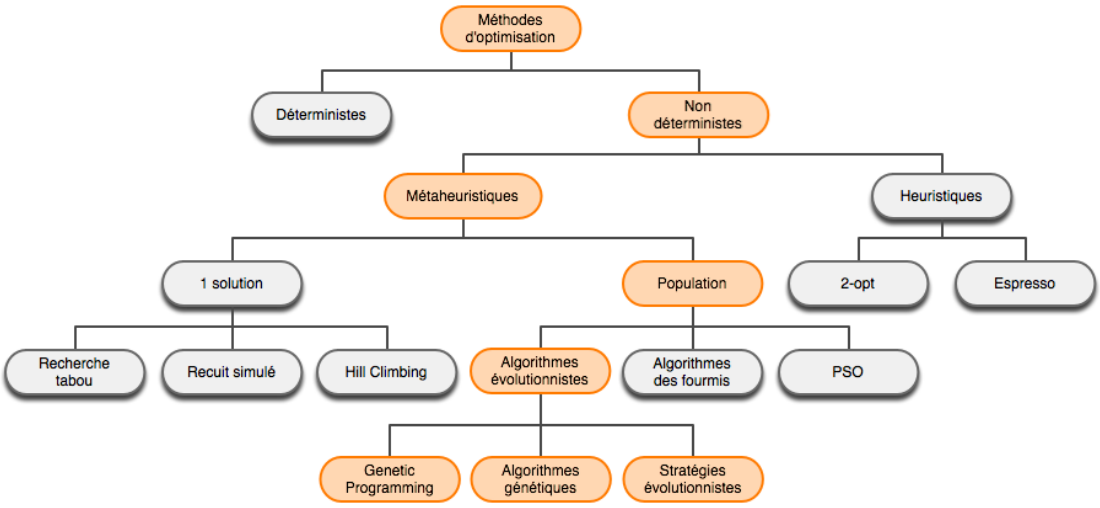

FIG. 1.1 – Classification des méthodes d'optimisation.

1.4 Les heuristiques

Les heuristiques sont une classe d'algorithmes d'optimisation qui, à chaque étape, utilisent les informations collectées précédemment afin de sélectionner la solution suivante à évaluer. Ce sont des algorithmes dits non exactes car ils ne trouvent pas nécessairement la solution optimale pour un problème donné.

Les heuristiques sont utilisées dans le but de déterminer des solutions sous optimales, voire la solution optimale, pour des problèmes d'optimisation NP-difficiles (cf.1.2.3) .

Ils dépendent fortement du problème ciblé. Une heuristique est dans la plupart des cas une technique spécifique à un type de problème donné, contrairement aux

métaheuristiques.

Sur la figure 1.1, nous avons fourni à titre d'exemple deux heuristiques :

- 2-opt : C'est l'heuristique la plus simple pour le problème du voyageur du commerce. Le principe est de permuter deux villes et d'évaluer le nouveau circuit, et ce, jusqu'à ce qu'un circuit meilleur que celui de départ soit trouvé.
- ESPRESSO : Technique d'optimisation de fonctions booléennes que nous allons revoir au point 2.4.5.

1.5 Les métaheuristiques

Les métaheuristiques sont un type d'algorithmes d'optimisation visant à résoudre des problèmes d'optimisation difficiles pour lesquels on ne connaît pas de méthodes déterministes fournissant la solution en un temps raisonnable.

Les métaheuristiques ont pour but de converger vers l'optimum globale de la fonction objectif. A chaque étape, l'algorithme se base sur les caractéristiques connues jusque là du problème pour se diriger vers une meilleure solution. Une métaheuristique peut être vue comme un mélange de plusieurs algorithmes. Elle peut, également, être considérée comme une recette que l'on adapte au problème que l'on souhaite optimiser. Nous pouvons distinguer deux catégories : les algorithmes qui se basent sur une solution qu'ils font évoluer vers une meilleure ; les algorithmes qui travaillent avec un ensemble de solutions, appelé population.

Les algorithmes évolutionnistes font partie du deuxième types de métaheuristiques et seront abordés au point suivant.

Selon [10], les métaheuristiques reposent sur un ensemble commun de concepts :

- La mémoire :
  sauvegarde l'information recueillie par l'algorithme.
- L'intensification ou l'exploitation : Tente d'utiliser les informations disponibles pour en trouver de meilleures à l'aide de recherches locales.
- La diversification ou l'exploration : Vise à accroître la quantité de ces informations en explorant de nouvelles régions de l'espace de recherche.

Nous allons nous intéresser dès à présent aux algorithmes évolutionnistes.

1.5.1 Les algorithmes évolutionnistes

Les algorithmes évolutionnistes sont des métaheuristiques. Leur particularité est de manipuler un ensemble de solutions, appelé population. Ils font partis des algorithmes dits bio-inspirés.

Un algorithme bio-inspiré est une classe d'algorithmes, plus généralement une métaheuristique, qui s'inspire des mécanismes de la nature. Les exemples sont légions mais les plus populaires demeurent les algorithmes génétiques s'inspirant de l'évolution, les algorithmes d'optimisation par colonies de fourmis et, un autre plus récent, l'algorithme par optimisation par essaim.

Comme le stipule le paragraphe précédent, un algorithme évolutionniste s'inspire du principe de l'évolution. En effet, en 1859, Charles Darwin mit en évidence le principe de l'évolution selon lequel tous les êtres vivants que la terre porte et a porté jusqu'à ce jour sont le fruit d'une suite d'opérations simples.

Les principes de la théorie de l'évolution [12] sont :

- La sélection naturelle
- La variabilité des traits de caractères
- La transmission des traits à la progéniture

Dans le cadre de problèmes d'optimisation, chaque solution peut être considérée comme un individu de la population. A chaque génération de la population, les individus subissent une pression sélective due à l'environnement dans lequel ils vivent. En biologie, on parle de fitness de l'individu. Le fitness fournit une indication sur le taux d'adaptation de l'individu à son environnement.

En optimisation, le problème considéré est l'environnement tandis que la fonction d'évaluation est la fonction de fitness. L'étape de la reproduction est remplacée par des opérateurs de croisement entre les solutions du problème. On entend par croisement le mélange de deux informations données pour en créer de nouvelles.

Biologie	Algorithmes évolutionnistes
Chromosome, génotype	Chaîne, séquence
Phénotype, individu	Solution candidate
Fitness / valeur sélective	Fonction de fitness

TAB. 1.2 – Analogie des termes en biologie et les algorithmes évolutionnistes

Les algorithmes évolutionnistes peuvent être subdivisés en trois grandes familles, les stratégies évolutionnistes, les algorithmes génétiques et la programmation génétique. Les algorithmes génétiques ont été rendus populaires par Goldberg dans son ouvrage *Algorithm genetic*. Nous allons, désormais, nous focaliser sur ces derniers, expliquer en détail les différentes étapes de l'algorithme ainsi que les opérateurs de base. Après avoir analysé les algorithmes génétiques, nous allons étudier des variantes des algorithmes génétiques et, pour terminer, diverses techniques liées aux algorithmes génétiques seront présentées.

1.6 Les algorithmes génétiques

Nous allons désormais introduire le lecteur aux algorithmes génétiques. Cette section repose sur les points bibliographiques [17][16][36].

1.6.1 Terminologie

Avant de s'attaquer aux algorithmes génétiques proprement dits, il est nécessaire d'introduire les termes utilisés dans de tels techniques.

Définition 11 *Chromosome*

Codage d'une solution au problème auquel est appliqué un algorithme génétique. Il est défini dans un alphabet donné et peut être de longueur fixée.

Définition 12 *Gène*

Brique de base du chromosome. Il peut prendre n'importe quelle valeur de l'alphabet du codage.

Définition 13 *Locus*

Position d'un gène au sein d'un chromosome.

Définition 14 *Allèle*

Différentes versions qu'un même gène peut prendre.

1.6.2 Introduction

Comme nous l'avons vu précédemment, les algorithmes génétiques font partie des algorithmes évolutionnistes. Dans le cadre d'un tel algorithme, il est nécessaire de définir un codage pour chaque solution du problème étudié. En effet, l'une des particularités des algorithmes génétiques est de manipuler une population au niveau de son codage. Ceci impose de définir une fonction de codage permettant de passer du codage (génotype) à la solution proprement dite.

L'idée de faire appel à un codage n'est pas fortuite. Il s'agit, encore une fois, du fruit de l'inspiration de la nature.

1.6.3 Principes

Nous allons à présent décrire les principes des algorithmes génétiques.
Un algorithme génétique peut être décomposé en plusieurs étapes.

- La génération de la population initiale
- L'évaluation des individus de la population
- La sélection
- Le croisement
- La mutation
- Le remplacement

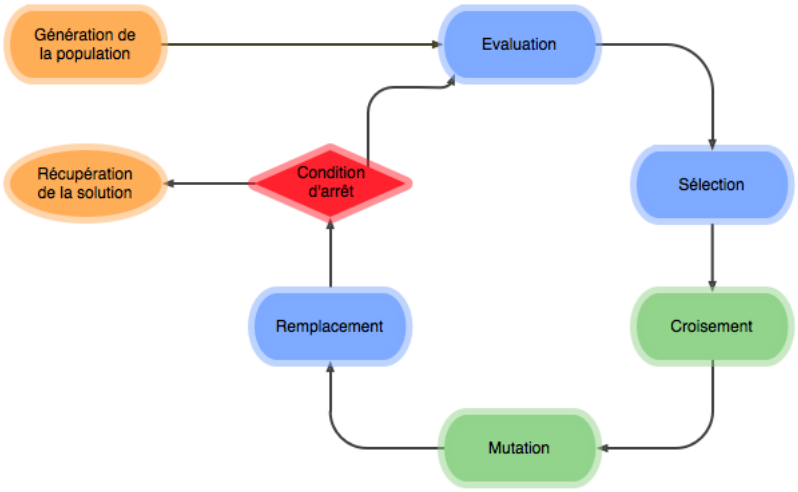

FIG. 1.2 – Principes des algorithmes génétiques.

Lors de la conception d'un algorithme génétique, il est nécessaire de définir certains points.

- La fonction de codage et de décodage d'une solution
- Les fonctions d'évaluations d'une solution
- Les opérateurs de croisement
- Les opérateurs de mutation
- La stratégie de remplacement

Sur la figure 1.2, nous avons tenté de mettre en évidence les principes généraux des algorithmes génétiques. En bleu, nous avons les étapes liées à la pression sélective. En vert, nous avons les étapes liées à la variabilité et à la transmission des traits de caractère.

1.6.4 Le codage

A chaque problème, il faut définir la manière de coder une solution, la manière de manipuler la solution et comment tester la validité de la solution. La première étape de l'adaptation d'un algorithme génétique est de définir comment coder les solutions au problème étudié.

Une solution peut être décomposée en plusieurs parties. Par exemple, les villes qu'un voyageur de commerce doit visiter, les différentes étapes d'une planification, etc.

Dans la littérature[17], nous pouvons rencontrer trois grandes familles de codage d'une solution :

Codage binaire

C'est le codage qui a été largement utilisé par Holland et Goldberg dans leurs travaux. Il est utilisé dans le Simple Genetic Algorithm. La solution est codée selon une chaîne de bits.

Codage à caractère multiple.

Chaque gène peut prendre un certain nombre de symboles définis dans un alphabet donné. C'est notamment le codage utilisé dans le cadre du design de circuits par des méthodes évolutionnistes que nous décrirons au chapitre 3.

Codage sous forme d'arbre.

Les solutions sont codées sous la forme d'arbre. C'est le codage utilisé notamment dans la programmation génétique (cf.1.7).

Il faut noter qu'il existe d'autres types de codages qui sont spécifiques au problème ciblé par l'algorithme génétique.

1.6.5 La génération de la population

La génération de la population est la première étape dans un algorithme génétique. Elle consiste, comme son nom l'indique, à la création d'une population de départ. La distribution de la population doit être la plus uniforme possible.

C'est une étape importante en ce sens que la qualité de la population initiale conditionne énormément la convergence de l'algorithme vers une solution au problème donné.

1.6.6 L'évaluation de la population

Comment parvenir à différencier deux individus ? Comme tout algorithme d'optimisation, les algorithmes génétiques possèdent une fonction d'évaluation, appelée fonction de fitness. Celle-ci associe à chaque individu un score qui est proportionnel à sa qualité.

La conception de la fonction de fitness est essentielle pour un algorithme génétique. En effet, elle est dépendante du problème, du but recherché ainsi que de l'expertise du concepteur. De plus, la plupart des décisions d'actions au sein de l'algorithme s'effectuent en fonction du fitness.

1.6.7 La sélection

Les méthodes de sélection sont nombreuses dont les plus connues sont : la roulette, la sélection par rang, la sélection par tournoi et l'élitisme. L'idée la plus intuitive de la sélection est de ne garder que les meilleurs individus de la population mais ceci peut facilement conduire soit à la non convergence de l'algorithme soit à un optimum locale. Par conséquent, il peut être intéressant de garder une certaine diversité dans la population et donc de prendre aussi quelques individus

moyens, voire mauvais, car leur génotype peut contenir une partie de la solution recherchée.

La sélection par roulette

La sélection de la roulette ou wheel se base sur le principe de la roue de la fortune. On associe à chaque individu un quartier de disque dont l'angle de secteur est proportionnel à son fitness. Tout comme le jeu dont la méthode s'inspire, une "bille" est lancée et le chromosome sur lequel elle s'arrête sera sélectionné.

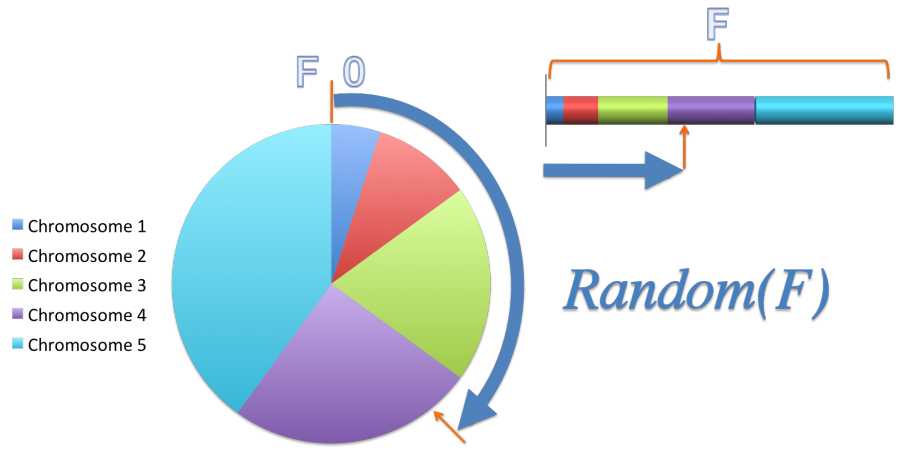

FIG. 1.3 – Principe de la sélection par roue de la fortune. Figure réalisée avec PowerPoint

La sélection par rang

La sélection par rang est une variante de la roulette où les individus sont dans un premier temps triés selon leur fitness. Ensuite, un rang est attribué à chaque individu. Le plus mauvais reçoit le rang 1 et le meilleur individu reçoit quant à lui l'ordre N si la population a une taille N. A partir de là, on applique de nouveau le principe de la roulette à cette différence près que les proportions qu'occuperont les individus seront calculées à partir du rang et non plus du fitness. Cette technique permet d'éviter qu'un individu ayant un fitness trop important par rapport aux autres puisse être trop souvent sélectionné au dépourvu de tous les autres.

La sélection par tournoi

La sélection par tournoi groupe les individus par paire de manière aléatoire. Le meilleur des deux individus sera sélectionné selon une probabilité définie comme paramètre de l'algorithme. Afin d'éviter de sélectionner trop souvent les mauvais individus, il est important que la probabilité de prendre le meilleur soit supérieure à 50%.

La sélection stochastique universelle

Le sélection stochastique universelle, ou SUS, est une méthode de sélection élimant le biais de la sélection par roulette.

A chaque individu de la population est associé un segment d'une longueur égale à son fitness. L'ensemble des segments est placé de manière contiguë pour former une ligne de longueur égale à la somme des fitness, comme dans la sélection par roulette.

La différence réside, ici, en l'ajout à intervalles réguliers d'un ensemble de pointeurs situés au dessus de la concaténation des segments. Le nombre de pointeurs doit être égal au nombre d'individus à sélectionner. La distance les séparant équivaut, ainsi, à l'inverse du nombre d'individus à sélectionner. La position du premier pointeur est donnée par un nombre aléatoire compris entre 0 et 1/N.

L'élitisme

L'élitisme permet d'éviter que les meilleurs individus d'une population à une génération donnée ne soit détruits par les opérateurs de croisement et de mutation dans la génération suivante. Le principe est de recopier le meilleur ou un certain nombre d'entre eux sans aucune modification dans la nouvelle génération.

1.6.8 Le croisement

Le principe du croisement est très simple : les génotype de deux individus, appelés parents, sont mélangés afin de produire d'autres individus, appelés enfants. Le croisement est effectué dans l'espoir que les enfants soient meilleurs que leurs parents et de faire évoluer ainsi la population vers des solutions de plus en plus

optimales. Cette étape a donc lieu après la sélection. Il existe différents opérateurs de croisement :

- Croisement en un point

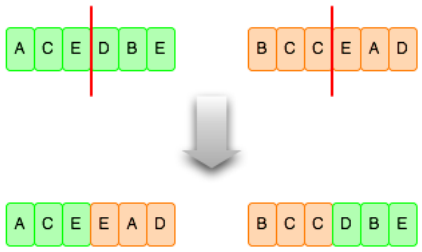

FIG. 1.4 – Croisement en 1 point

- Croisement en deux points

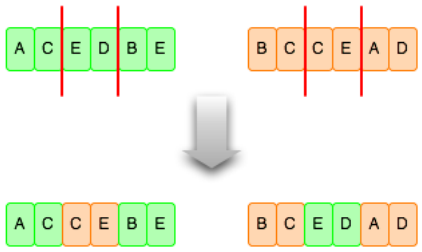

FIG. 1.5 – Croisement en 2 points

Il est nécessaire de rappeler que pour chaque type de problème, il existe un certain nombre d'opérateurs de croisement spécifiques.

La mutation

La mutation empêche l'algorithme de converger vers un extrema local en introduisant des modifications aléatoires à une petite proportion des individus. Cela augmente la diversité des solutions mais peut aussi détruire de bons individus. Si le taux de probabilité de mutation est trop élevé, l'algorithme dégénère en une recherche aléatoire. Cette caractéristique ne présente aucun intérêt dans notre cas.

FIG. 1.6 – Principe de la mutation.

Après la mutation, l'algorithme reprend le cycle à partir de l'évaluation avec la nouvelle population produite.

1.6.9 Le théorème des schématas

Nous allons fournir une explication du théorème des schématas basée sur les sources bibliographiques [17] et [16].

Le théorème des schématas fût énoncé par Holland dans les années 70. Il tente d'expliquer le fonctionnement des algorithmes génétiques d'une manière plus mathématique.

Néanmoins, l'algorithme génétique qu'a proposé Holland dans son ouvrage n'est plus vraiment d'actualité.

Depuis, de nombreuses variations furent proposées au point où les algorithmes actuels diffèrent considérablement de l'algorithme de Holland. Néanmoins, la théorie des schématas constitue au demeurant un bon point de départ pour comprendre leur fonctionnement.

La question que s'est posée Holland et qui a été reprise par Goldberg est :

Existe-t-il un lien entre toutes les solutions considérées comme meilleures ?

En d'autres termes, quelles sont les similitudes entre deux bonnes solutions. C'est ainsi que sont nés les schémas qui ne sont rien d'autres que des patrons de similitudes pour les chaînes de codage.

Qu'est ce qu'un schéma ? Pour comprendre ce qu'est un schéma, il est plus pratique de se placer dans un exemple concret.

Pour simplifier, supposons que les solutions d'un problème donné soient codées dans un alphabet binaire, noté A. Pour définir un schéma, il est nécessaire de compléter l'alphabet de codage par un symbole pouvant prendre n'importe quelle valeur de l'alphabet concerné. Ce symbole est souvent appelé wildcare ou don't care et est noté *.

Un schéma est représenté par une chaîne, codée dans l'alphabet A, dans lequel on retrouve des don't care. Un schéma décrit un ensemble de chaînes ayant en commun les symboles autres que des wildcares.

Dans l'espace des solutions à n dimensions, un schéma définit un hyperplan. Soit l'ensemble des séquences binaires à 3 bits, nous pouvons définir l'espace de recherche à l'aide d'un cube dont les sommets sont les solutions candidates. Un schéma représente l'ensemble des sommets formant un côté du cube ou celui formant une face du cube (voir fig.1.7).

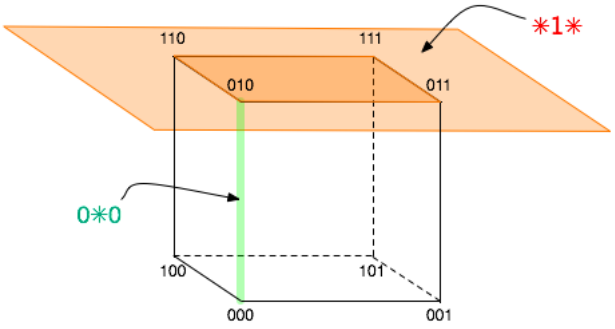

FIG. 1.7 – Illustration des schémas

Il est essentiel de définir quelques notions concernant les schémas.

Définition 15 *L'ordre d'un schéma, noté* $o(H)$*, est le nombre de positions fixées.*

Définition 16 *La longueur utile est la distance séparant les première et dernière positions fixées. Elle est notée* $\delta(H)$.

Définition 17 *Toute chaîne qui est incluse dans un schéma est appelée une instance ou un échantillon du schéma.*

Comportement des schémas. Selon Holland, l'algorithme génétique ne manipule pas les schémas mais, d'une manière ou d'une autre, il garanti la propagation de bons schémas.

Le théorème des schémas Lors d'un croisement, chaque schéma transmis à un enfant reçoit une copie supplémentaire dans la population. Un schéma peut être

détruit si le site de croisement tombe entre les gènes qui le définissent. Ainsi, la destruction d'un schéma est proportionnelle à la longueur du schéma.

Théorème 1 *Les schématas courts et d'ordre inférieur voient leur nombre d'instances augmenter de manière exponentielle au cours des générations.*

$$m(H,t+1) \geqslant m(H,t)\frac{f(H)}{\mathcal{F}}\left[1-\rho_c\frac{\delta(H)}{L-1}-\rho_m o(H)\right]$$

ρ_c Probabilité de croisement

ρ_m Probabilité de mutation

L Longueur d'un chromosome

Selon Goldberg, le croisement laisse le schéma intact s'il ne le coupe pas et peut le détruire dans le cas contraire. Par conséquent, les schémas de courte longueur utile sont conservés par le croisement et reproduits en quantité par l'opérateur de reproduction. La survie d'un schéma face à l'opérateur de mutation dépend du taux de mutations accordé au sein d'une population.

1.6.10 L'hypothèse des building blocs

L'hypothèse des *building blocs* ou *BBH* avance l'idée que l'algorithme génétique fonctionnerait s'il trouvait les bons building blocs à l'aide de la mutation, du croisement et de la sélection. Ensuite, il les combine à l'aide du croisement pour produire des solutions complètes.

Le BBH est basé sur la supposition que les bonnes propriétés d'un parent sont rassemblées dans un bloc relativement petit à divers endroits de la chaîne de codage. Ces blocs peuvent être regroupés ou fusionnés par le croisement. Autrement dit, le BBH suppose qu'une meilleure solution est assemblée à partir des parties de ses parents[17].

1.7 La programmation génétique

La programmation génétique est une variante des algorithmes génétiques. Sa particularité est que les solutions sont codées sous la forme d'arbres. John R. Koza est considéré comme le père de la programmation génétique. Il l'appliqua à des programmes en LISP, langage utilisé en intelligence artificielle[34] [35].

1.8 Les stratégies évolutionnistes

Une stratégie évolutionniste est un algorithme évolutionniste qui ressemble aux algorithmes génétiques mais dont l'importance du crossover est quasi nulle, voir inexistante. Les individus sont généralement codés sous la forme de vecteurs de nombres réels. Les opérateurs de mutation et de sélection sont les acteurs principaux tandis que le croisement est beaucoup moins commun. La mutation modifie un élément du vecteur d'une solution aléatoirement selon une distribution normal N(xi, si). Et pour finir, tout comme les autres algorithmes évolutionnistes, ces différentes étapes sont itérées tant que les conditions d'arrêt ne sont pas atteintes. Chaque itération est appelée génération.

1.9 Les algorithmes mémétiques

Tout comme énoncé précédemment, les algorithmes évolutionnistes souffrent de la convergence de la population vers une solution unique. Il est nécessaire de garder une certaine diversité afin d'échapper à cet optimum local. Comme nous avons déjà vu les techniques de sharing et de scaling, nous allons introduire maintenant le concept d'hybridation d'algorithmes.

Les techniques de recherche locale offrent une bonne exploitation d'une zone donnée tandis qu'un algorithme évolutionniste permet une grande exploration de l'espace de recherche. La question qui se pose est pourquoi ne pas profiter de l'apport des deux familles de techniques. La réponse réside dans le concept des algorithmes mémétiques où l'étape de mutation est remplacée par un algorithme local. Le but de ce dernier est d'augmenter la diversité dans la population tout en cherchant de meilleures solutions, voire de permettre d'échapper à un optimum local.

1.10 Conclusion

Dans ce chapitre, nous avons présenté différents algorithmes évolutionnistes. Nous nous sommes, ensuite, concentrés sur les algorithmes génétiques, algorithmes que nous utiliserons ultérieurement. Nous allons maintenant aborder le deuxième domaine de ce mémoire : les circuits logiques.

Chapitre 2

Circuits Logiques

Contents

2.1 Introduction

Les circuits logiques constituent un des types de circuits les plus répandus. Nous pouvons en trouver dans les ordinateurs, les téléphones portables, les jouets et autres appareils électroniques. Il est évident que la technologie utilisée ainsi que leur forme et leur taille n'ont cessé d'évoluer durant ces cinquante dernières années.

Dans cette seconde partie du mémoire, il apparaît pertinent de traiter du second thème, à savoir celui des circuits logiques. Dans un premier temps, nous définirons les concepts de base de la logique booléenne. Nous allons, ensuite, expliquer ce qu'est un circuit logique, en décrire les différentes sortes pour finalement concentrer notre attention sur les circuits logiques combinatoires. Quelques techniques de design conventionnelles seront également décrites dans ce chapitre.

2.2 L'algèbre de Boole

En informatique et en électronique numérique, un bit est considéré comme l'unité de travail. Un bit peut prendre soit un état haut soit un état bas. Il existe différents opérateurs pour manipuler des bits dont les règles sont définies par l'algèbre de Boole. Une variable logique est une variable qui peut contenir un bit. Nous définissons une fonction logique comme étant une fonction composée d'opérations logiques entre plusieurs variables et possédant comme valeur une variable logique. Toutes les fonctions logiques peuvent être exprimées à partir des trois opérateurs de base : OR , AND, NOT.

2.3 Les portes logiques

Chaque opération logique définie dans l'algèbre de Boole a été traduite en circuit électronique sous forme de portes logiques. Les portes logiques de base sont au nombre de 3 : AND, OR, et NOT. En combinant ces différentes portes, nous pouvons former des circuits logiques. Un circuit logique est, donc, la traduction électronique d'une fonction logique dont les variables sont les entrées du circuit et la valeur en est la sortie.

Selon [40], les portes les plus courantes dans le monde informatique sont les portes NOR et NAND. Cela est dû aux portes CMOS qui se sont avérées les composants électroniques les plus efficaces pour la réalisation de circuits.

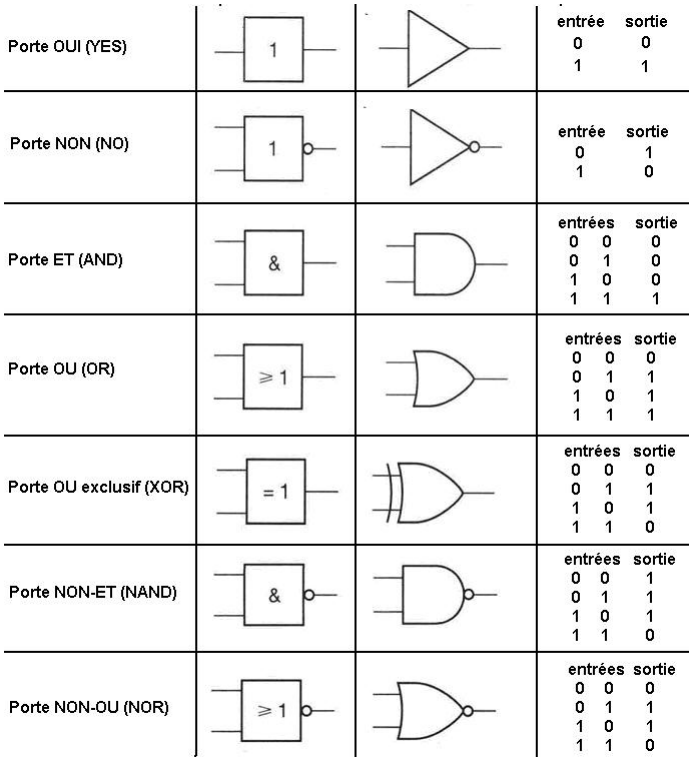

FIG. 2.1 – Portes logiques de base

2.4 Les circuits logiques

Les circuits logiques sont un assemblage de portes logiques et de mémoires qui permettent de stocker les données. Nous pouvons les classer en deux sous catégories : les circuits combinatoires et les circuits séquentiels.

Un circuit combinatoire est un circuit dont la valeur des sorties est calculée à partir des entrées uniquement ; tandis que, pour un circuit séquentiel, la valeur de sortie est calculée à partir de la valeur de sortie passée ainsi que des entrées à l'instant donné.

Dans le cadre de ce mémoire, nous nous focaliserons sur les circuits logiques combinatoires. La raison de ce choix est principalement due à la complexité d'un circuit séquentiel, notamment à cause de leur caractère rétroactif. Un circuit logique peut posséder plusieurs sorties dont chacune correspond à une fonction logique.

En générale, chaque fonction logique peut être traduite en circuit logique combinatoire et vice-versa. Chaque porte logique d'un circuit est l'équivalent d'un opérateur de la fonction logique.

2.4.1 Spécification

Un circuit logique peut être décrit de différentes manières selon le niveau d'abstraction désiré. Un circuit logique peut être décrit par

Le cahier des charges Il s'agit de la première étape du processus de conception d'un circuit logique, il définit le comportement de ce dernier de manière verbale.

La table de vérité La table de vérité est un tableau indiquant pour chaque entrée la sortie correspondante. Tout circuit logique en possède une. Néanmoins, cette description possède des limites contraignantes dès que le nombre d'entrées ou de sorties augmente. En effet, un rapide calcul démontre que, si le nombre d'entrées du circuit vaut N, alors le nombre de lignes de la table vaudra : 2ˆN. De plus, le nombre de colonnes est fixé par le nombre de sorties du circuit, noté m. Chaque case de la table de vérité contient soit un 1 ou un 0. Donc, pour représenter une table de vérité, nous avons besoin de 2ˆN * m cases.

Après avoir défini le cahier des charges et effectué une analyse, il est nécessaire de s'intéresser au hardware.

2.4.2 Représentation de fonctions logiques

A partir de la table de vérité, nous pouvons extraire une forme algébrique de la fonction logique. Cependant, nous ferons remarquer que la forme algébrique n'est pas unique[26].

Les formes canoniques booléens

Forme canonique standard

La forme canonique standard est unique mais peut s'exprimer sous deux formes :

- La somme des mintermes
- Le produit des maxtermes

La forme canonique non standard

Si la forme canonique standard est simplifiée ou étendue à l'aide des axiomes de l'algèbre de Boole, nous obtenons la forme canonique non standard.

Une table de vérité peut posséder une infinité de formes canoniques non standards.

Les formes canoniques ne comportent que des portes OU, ET et NOT.

Les formes canoniques Reed-Muller

Ce sont des formes booléennes qui font appel aux portes XOR, AND, NOT.

2.4.3 Implémentation des circuits logiques

Les circuits logiques peuvent être implémentés grâce à des portes logiques électroniques à base de technologies CMOS ou TTL[40]. Pour chaque porte logique, il existe un circuit intégré comportant quelques portes du même type.

Depuis que la technologie permet d'effectuer des prouesses en miniaturisation, des circuits logiques programmables ont vu le jour. Un circuit logique programmable est un circuit intégré qui peut être reprogrammé après sa fabrication. Il en existe plusieurs familles tels que les FPGA , les EPLD et les PLA. Pour plus d'informations concernant ces termes nous invitons le lecteur à consulter [40].

Les FPGA sont basés sur des cellules de mémoire utilisées aussi bien pour le routage que pour les blocs logiques. En général, un bloc logique est composé d'une LUT[1] et d'une bascule. Chaque LUT permet d'implémenter une équation logique de 4 à 6 entrées et une sortie.

Les CPLD regroupent plusieurs types de circuits logiques reprogrammables. Ils sont constitués d'un réseau de portes ET et OU afin d'implémenter des équations logiques.

Quelque soit la technologie utilisée, la simplification d'une fonction logique mène souvent à des économies en termes de portes logiques et donc de matériau (voir fig.2.2). La synthèse logique est la discipline qui traduit une spécification de circuit en un ensemble de portes logiques interconnectées.

[1] LUT : Look-Up-Table ou table associative

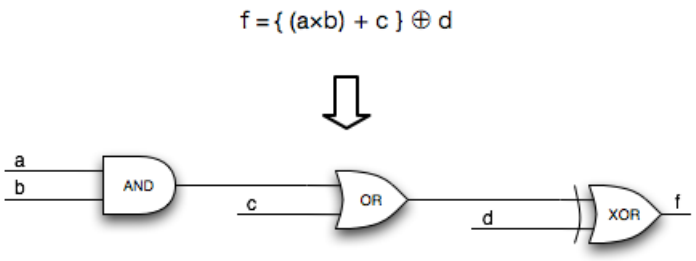

FIG. 2.2 – Traduction d'une fonction logique en circuit

2.4.4 La synthèse de circuits logiques

La synthèse est le processus qui décrit le passage de la description haut niveau d'un circuit électronique vers son implémentation dans une technologie donnée. Il existe donc différents niveau de synthèse de circuits, nous ne nous intéresserons qu'à la synthèse logique.

Dans le cadre des circuits logiques, la synthèse consiste à la réalisation du circuit à l'aide de portes logiques à partir de la spécification textuelle ou bien de sa table de vérité.

En plus, de pouvoir fournir le design hardware d'un circuit logique, la synthèse a pour but l'optimisation du circuit.

Mais que cherche-t-on a optimiser dans un circuit logique ? Cette section a pour but d'introduire le sujet d'une manière naïve. Le paramètre fondamental et "logique" pour un circuit logique est bel et bien le nombre de portes contenues qu'il contient. Le nombre de portes logiques influence énormément les paramètres suivant d'un circuit :

- Le prix du circuit
- La consommation
- La taille du circuit
- Le délai de réponse

Hormis le nombre de portes, la disposition des portes ou des composants au sein d'un circuit logique est également importante.

Nous allons nous intéresser aux techniques de synthèse qui, à partir d'une table de vérité, produisent une fonction logique tout en essayant d'optimiser le nombre d'opérateurs dans la fonction. Ceci se traduit en terme de circuit par une économie

de composants.

2.4.5 Techniques de synthèse et d'optimisation de circuits logiques

Cette section abordera trois techniques conventionnelles de synthèse de circuits logiques. Nous donnerons une brève description de chacune d'elles ainsi qu'une discussion sur leur limitation. Cette section se base sur les points bibliographiques [26][40].

Nous faisons remarquer qu'à l'heure actuelle, la fonctionnalité d'un circuit est définie dans un langage de description matériel tel que le VHDL et la synthèse est laissée au main de logiciels basés sur des systèmes experts. Néanmoins, ces techniques de synthèse capable de gérer diverses contraintes sont soumis à des secrets professionnels.

La table de Karnaugh

La table de Karnaugh n'est rien d'autre qu'une disposition spéciale de la table de vérité. C'est une table à deux dimensions. Chaque colonne diffère de sa voisine d'un seul littéral. L'intersection d'une ligne et d'une colonne donne la valeur pour l'entrée donnée. Le but de la table de Karnaugh est de transformer la table de vérité d'un circuit logique en une équation logique sous la forme de sommes de produits. Il est plus simple d'expliquer la table de Karnaugh par un exemple que par de longues phrases. Le problème majeur de Karnaugh est qu'il s'agit, non seulement, d'une méthode graphique mais aussi d'une méthode dont l'efficacité dépend fortement de l'expertise du designer. Par ailleurs, Karnaugh se révèle impraticable quand le nombre d'entrées est plus grand que 6.

Quine McCluskey

L'algorithme de Quine-McCluskey, connu également sous l'appellation de la méthode des impliquants premiers, est une méthode de minimisation des fonctions booléennes. Son fonctionnement est identique à celui de la table de Karnaugh mais elle est plus adaptée pour une exécution sur ordinateur. La raison principale

est qu'elle offre une manière déterministe de savoir si la forme minimale d'une fonction booléenne est atteinte.

L'algorithme comporte deux étapes : la recherche des impliquants premiers et la recherche des impliquants premiers dits essentiels.

Le temps d'exécution de Quine-McCluskey augmente de manière exponentielle en fonction du nombre de variables et, donc, du nombre d'entrées du circuit correspondant. Les fonctions possédant un trop grand nombre de variables ne peuvent être simplifiées par une méthode déterministe car il s'agit un problème dit NP-HARD [11]. Il est, dès lors, nécessaire de faire appel à des heuristiques telles que l'algorithme ESPRESSO.

ESPRESSO

Espresso est une heuristique permettant la réduction de la complexité des circuits composés de portes logiques. Espresso fût développé chez IBM par Richard L. Rudell[22]. Pour comprendre le fonctionnement de cette heuristique, nous invitons le lecteur intéressé à ce reporter aux points bibliographiques [22].

2.5 Circuits de base

Cette section a pour but de décrire quelques circuits de base qui sont généralement utilisés comme blocs de base dans de plus larges circuits. Il s'agit des circuits dits arithmétiques. On distingue l'additionneur, le multiplieur ainsi que le testeur de parité. Ces circuits ne peuvent être implémentés sous forme de mémoire associative au sein d'un circuit reconfigurable en raison de l'explosion combinatoire du problème qu'il traite. En effet, les nombres sont encodés sur 32 bits au sein d'un ordinateur. Or, afin de multiplier ou d'additionner deux nombres de 32 bits, il est nécessaire de disposer de 2 x 32 entrées et 32 sorties dans le circuit. L'utilisation d'une mémoire associative nécessiterait donc 2^64 * 32 bits de mémoire, ce qui équivaut à 2^36 giga-octets. Quelque soit le système et encore moins s'il s'agit d'un FPGA, il est impossible de disposer d'une quantité de mémoire aussi importante. Enfin, ces circuits logiques sont souvent câblés et sont toujours sujets à diverses optimisations.

L'additionneur

Un additionneur est un circuit qui, comme son nom l'indique, effectue la somme de nombres binaires. On parle en générale d'additionneurs n bits. La brique de base d'un additionneur n bits est l'additionneur 1 bit. C'est un circuit que nous retrouvons dans les circuits logiques effectuant des multiplications, des divisions, etc. [6]

L'additionneur 1 bit se décline en deux types : le half adder et le full adder. Le half adder possède deux entrées, les bits dont on veut calculer la somme, et deux sorties, le bit de report et la somme.

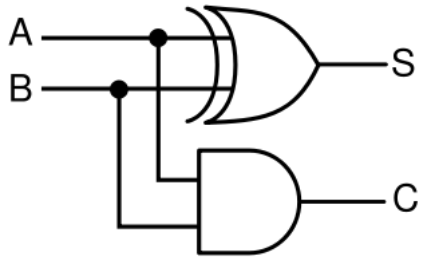

FIG. 2.3 – Schéma du semi-additionneur 1 bit.Source : Wikipédia

Le full adder, quant à lui, prend en charge un bit supplémentaire d'entrée qui correspond au report entrant. Ce dernier joue un rôle majeur pour la constitution d'additionneurs plus grands. En effet, pour créer un additionner de N bits, il suffit d'interconnecter N fulladders en veillant à connecter le carry out de chaque circuit au carry in de l'étage suivant (voir fig.2.5). C'est le principe du *ripplecarry*.

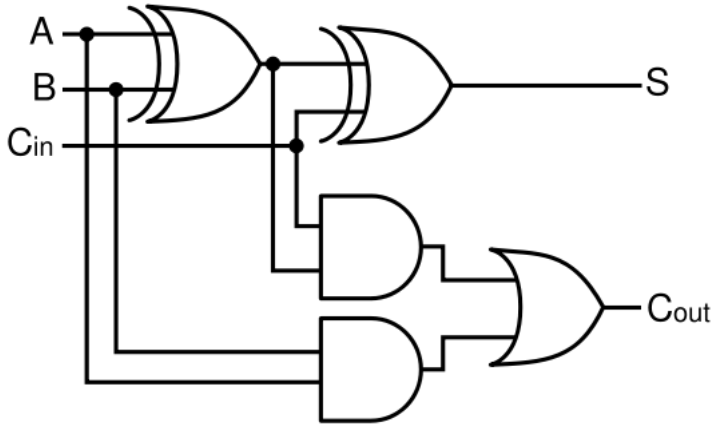

FIG. 2.4 – Schéma du full-additionneur 1 bit.Source : Wikipédia

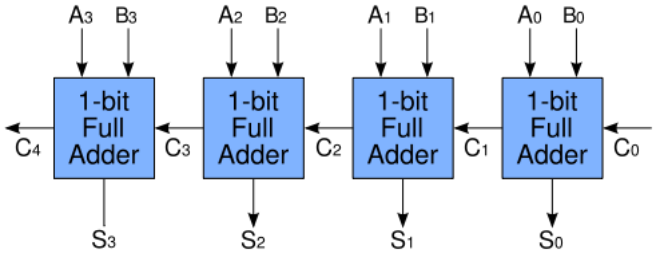

FIG. 2.5 – Additionneur 4 bits obtenu par le principe du ripple-carry :Wikipédia

Le multiplieur

Le multiplieur est un circuit permettant d'effectuer la multiplication de deux nombres binaires. Il est constitué d'additionneurs comme nous pouvons le constater sur la figure ci-dessous 2.6.

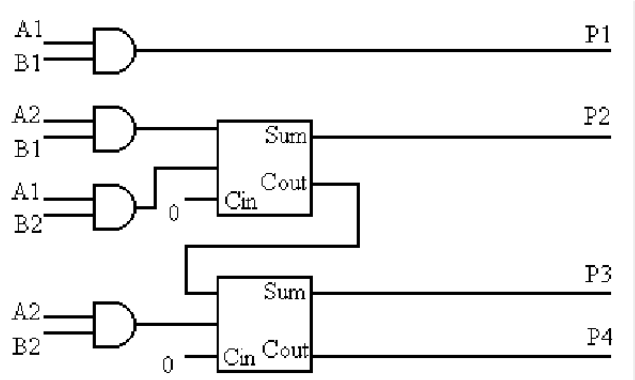

FIG. 2.6 – Multiplicateur à 2 bits [23].

Le parity check circuit

Les informations contenues dans cette section proviennent de [30] .Le testeur de parité est l'un des circuits les plus importants en théorie de l'information. En effet, l'algorithme le plus simple de détection d'erreurs est le test de parité. Le principe consiste en l'ajout d'un bit de parité à la fin de la séquence binaire dont on veut s'assurer de l'intégrité. Ce bit est mis à 1 lorsque le nombre de 1 dans la séquence est un nombre impaire et vaut 0 dans le cas contraire. Cette tâche peut facilement être réalisée à l'aide d'un circuit combinatoire.

Malheureusement, le circuit le plus simple pour réaliser cette tâche a été breveté le 22 juin 2000 par Satoshi Owada. Comme nous pouvons le voir sur la figure ci-dessous 2.7, le circuit qu'il a développé est composé essentiellement de portes OU Exclusives. Il compte un certain nombre d'étages dont le nombre de portes XOR est divisé par deux à chaque fois jusqu'au dernier étage qui ne comporte plus qu'une seule porte XOR. Dès lors, afin de valider une séquence binaire de n bits, avec n pair, il est nécessaire d'utiliser n/2 portes XOR auxquelles on interconnecte chaque sources aux bits à valider.

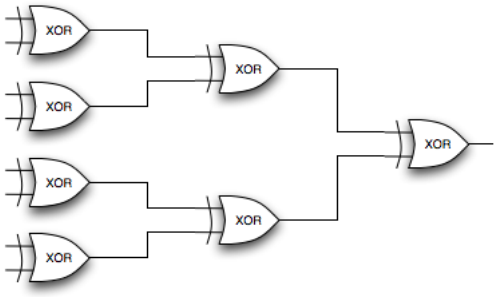

FIG. 2.7 – Parity Check Circuit à 4 bits. Inspiré de[30]

2.6 Conclusion

Dans ce chapitre, nous avons tout d'abord introduit les notions de base de l'algèbre de Boole pour ensuite présenter les différentes portes logiques existantes. Nous avons dès lors fourni une brève introduction aux circuits logiques ainsi qu'aux diverses techniques de synthèse et de simplification conventionnelles. Et pour finir, quelques circuits de base ont été présentés.

Nous allons désormais nous intéresser aux principes du design évolutionniste.

Chapitre 3

Evolution de design de circuits logiques

Contents

3.1 Introduction

Le premier chapitre était dédié aux algorithmes évolutionnistes et aux techniques d'optimisation. Le second quant à lui traitait des circuits logiques mais il n'a encore jamais été question de combiner les deux domaines.

Ce chapitre servira donc d'état de l'art dans l'application des techniques évolutionnistes au design des circuits logiques. En effet, les concepteurs humains de circuits logiques conçoivent souvent les circuits d'une manière top-down et utilisent les techniques décrites au point 2.4.4 pour effectuer la synthèse.

En effet, comme vu au chapitre 2, certains circuits sont très répandus mais, malheureusement, beaucoup d'entre eux sont brevetés tel que le parité check.

3.2 Algorithmes évolutionnistes et innovation

Au chapitre 1, les algorithmes évolutionnistes ont été présentés comme étant des techniques d'optimisation. Dorénavant, nous allons leur greffer une seconde dimension, l'innovation.

Selon l'article [14], les algorithmes évolutionnistes sont utilisés pour améliorer les profiles aérodynamiques des carrosseries de voitures, des fuselages et des ailes d'avions. Seules les grandes entreprises pouvaient se permettre ce genre de techniques grâce à l'accès à des superordinateurs capables de manipuler de très grands génotypes, pouvant atteindre des milliers de générations.

Dorénavant, avec la puissance de calcul grandissante des systèmes informatiques, ce qui, dans le temps nécessitait des mois de calcul, n'en requiert plus que quelques jours. C'est la raison pour laquelle les techniques évolutionnistes sont de plus en plus utilisées dans de nombreux domaines.

L'équipe de Koza, considérée comme père de la programmation génétique, est parvenue à développer une antenne Wi-Fi pour un client qui ne voulait pas payer une licence Cisco Systems.

Les algorithmes évolutionnistes seraient un moteur d'innovation permettant même dans certains cas de briser des brevets sans les enfreindre.

Miller montra dans un de ces travaux que l'idée précédente a aussi du sens dans le cadre des circuits logiques (voir . [23]).

Ce mémoire se base donc sur les deux dimensions des techniques évolutionnistes, l'optimisation et l'innovation dans le design des circuits logiques.

3.3 L'evolvable hardware

Cette section s'inspire du point bibliographique [18] et tentera de mettre en évidence l'importance des techniques évolutionnistes pour les circuits électroniques.

L'evolvable hardware ou EH concerne le matériel capable de modifier sa structure et son comportement dynamiquement en interagissant avec le monde extérieur. Cette discipline fait appel aux algorithmes génétiques pour générer les nouvelles configurations. Le but est d'obtenir des circuits capables de se configurer afin de corriger des éventuelles erreurs causées par l'environnement parfois imprévisible.

L'exemple classique est celui de la sonde spatiale ou du robot explorateur dont les circuits sont soumis à des conditions extrêmes souvent dues aux températures ou aux bombardements de particules. Afin de compenser les erreurs de fonctionnement qui risquent d'apparaître dans de tels conditions, le circuit doit pouvoir s'auto-reconfigurer afin de retrouver son comportement d'origine. C'est la principale raison pour laquelle la NASA s'intéresse de près à ces techniques [4]. L'evolvable hardware est souvent rendu possible grâce aux logiques programmables présentées dans le chapitre 2.

Des algorithmes évolutionnistes ainsi que diverses techniques d'intelligence artificielle sont mis en oeuvre afin de trouver la configuration la plus adéquate à la situation.

Selon [18], l'evolvable hardware se subdivise en deux sous catégories :

- Extrinsèque
- Intrinsèque

La différence entre ces catégories réside dans la manière de calculer la fonction de fitness. Dans le cas de l'évolution intrinsèque, l'évaluation du fitness s'effectue directement sur hardware. Ceci est rendu possible grâce aux logiques programmables (voir chapitre 2) . Toutes les solutions sont implémentées et testées physiquement. Le principal problème est la vitesse de reprogrammation qui peut être très longue et peut occasionner des dégâts aux systèmes.

Tandis que l'évolution extrinsèque se base sur un simulateur externe à hardware. Tout le processus d'évolution se déroule au sein d'un module destiné. Seule la meilleure solution de la dernière génération est implémentée physiquement.

L'evolvable hardware est à l'intersection de nombreuses disciplines telles que nous pouvons le voir sur la figure ci-dessus.

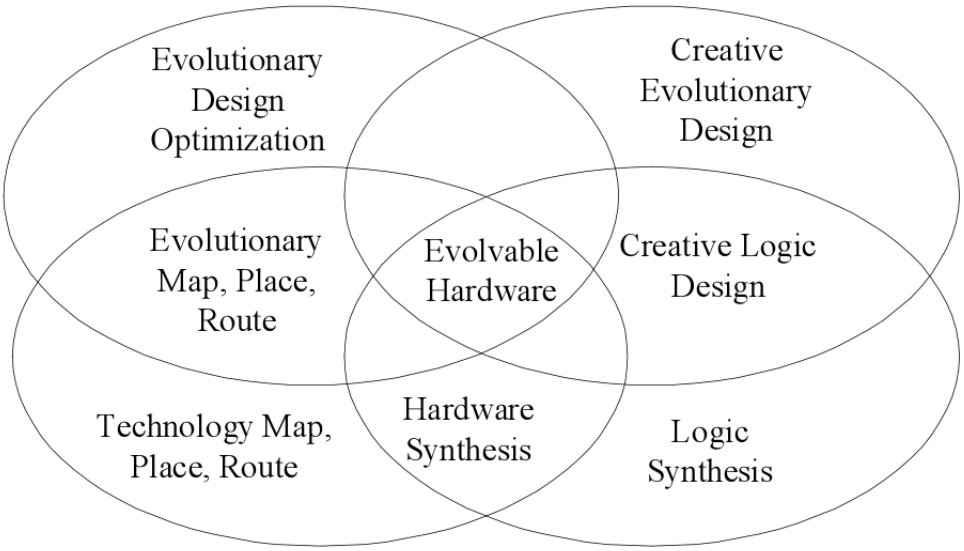

FIG. 3.1 – Domaines impliqués dans l'evolvable hardware. [18]

Notre travail est consacré au processus d'évolution intervenant dans la méthode extrinsèque. Ainsi, toutes les expérimentations ont lieu dans un simulateur. La suite du mémoire se focalise sur l'évolution de design de circuits logiques. Comme annoncé au chapitre 2, malgré l'avènement des logiques reprogrammables qui se basent sur des LUT, les circuits arithmétiques ne peuvent être détrônés et sont toujours à l'heure actuelle câblés [23] soit à l'aide de portes logiques soit à l'aide d'un certain nombre de LUT dédiées.

3.4 Principe du design évolutionniste

Le design évolutionniste a pour but de concevoir des circuits en faisant appel à des algorithmes évolutionnistes. Chaque circuit est codé sous forme de chromosomes. L'algorithme commence par générer une population de circuits aléatoires

qui seront par la suite évalués selon plusieurs critères.

Dans le cadre d'un circuit logique, il est nécessaire que le circuit que l'on désire synthétiser réponde correctement à la fonctionnalité recherchée. De plus, le circuit généré doit pouvoir répondre à certaines contraintes telles que le nombre de maximum portes, la consommation, la taille du circuit, le prix...

Comme la plupart de ces contraintes sont liées, on peut souvent se contenter de n'optimiser que quelques paramètres.

Dans le cadre de ce mémoire, nous nous focaliserons sur la fonctionnalité et le nombre de portes du circuit.

3.5 Espace de représentation

Cette section est une discussion concernant la représentation des circuits logiques que nous avons abordé au point 2.4.2.

Dans l'article [23], les auteurs mettent en évidence une problématique assez intéressante.

Comme nous avons vu au point 2.4, un circuit logique est l'implémentation électronique d'une table de vérité et il existe différentes techniques de synthèse logique.

Nous allons décrire le raisonnement que Miller et d'autres ont tenu concernant l'espace de représentation des fonctions logiques.

Les expression canoniques booléennes que nous avons abordées au chapitre 2, n'utilisent que les portes AND, OR et NOT tandis que les expressions Reed-Muller ne sont composées que des portes XOR, AND et NOT.

Mais l'espace des circuits logiques ne se limitent pas qu'à ces deux ensembles. Qu'en est-il du reste de l'espace des représentations des fonctions logiques (voir fig.3.2) ?

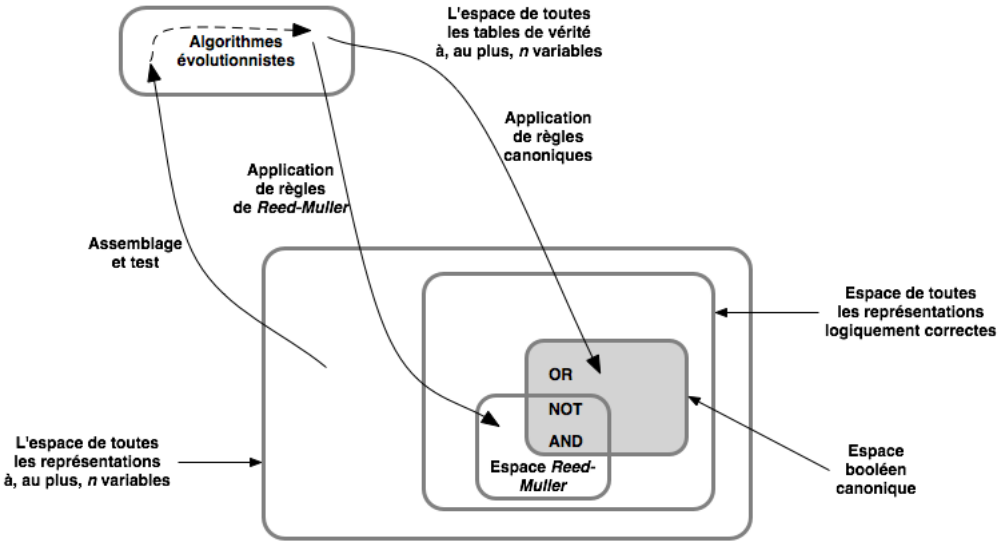

FIG. 3.2 – Espace de représentation des fonctions logiques. [23]

L'idée intuitive pour l'exploration d'un espace de recherche est selon [23], la recherche exhaustive en utilisant le principe du *assemble and test*[1]. C'est pour cette raison qu'il proposa d'utiliser des algorithmes évolutionnistes pour explorer l'espace des représentations plus efficacement.

3.6 Limites du design évolutionniste

Avant d'entrer dans le vif du sujet, il est nécessaire de clarifier certains points avant de présenter les diverses techniques de la littérature et leurs résultats.

Comme le souligne Coello dans son article [11], bien que les techniques conventionnelles explorent mal l'espace de recherche, elles s'avèrent largement plus performantes que les algorithmes génétiques.

En effet, certaines techniques sont capables de synthétiser et d'optimiser de manière approchée des circuits possédant jusqu'à une centaine d'entrées tandis que les algorithmes évolutionnistes sont limités à de petites tables.

[1] Assemble and test peut-être traduit par Assembler et essayer.

3.7 Le codage de circuit

Comme l'algorithme génétique manipule le codage de la solution et non la solution elle-même, il nous apparaît important de commencer par décrire les différents modèles de génotypes définis pour les circuits logiques.

Les modèles concernant le génotype d'un circuit logique ne sont pas légions dans la littérature concernant le sujet. En effet, il existe principalement deux types de modélisation possibles avec leur variantes.

Un circuit logique peut être vu soit comme un programme soit comme un schéma.

Nous allons dans la suite de cette section expliquer les différences fondamentales entre les deux visions et présenter les modélisations qui en découlent.

3.7.1 Le codage en matrice

Sushil J. Louis, un chercheur de l'université de l'Indiana, introduisit l'idée d'utiliser les algorithmes évolutionnistes comme outil de design [21]. Il appliqua notamment ses idées à des circuits logiques.

Dans son modèle, le circuit est vu comme un tableau dans lequel on place les portes aux différentes cases. Les entrées de chaque portes sont connectées aux sorties des portes situées dans les colonnes précédentes.

Dans son modèle, il utilisa seulement trois types de portes logique de base, AND, OR et NOT, auxquelles il ajouta la pseudo porte *WIRE*.

Les portes WIRE sont de simples fils qui recopie leur entrée vers la sortie.

De travaux en travaux, le modèle de Louis a subi quelques adaptations. C'est ainsi que Miller et al. qui travaillaient alors avec des circuits sur FPGA, ont proposé un codage et, donc, un génotype très proche du phénotype et, donc, du circuit physique.

Leur modèle est aussi une matrice composée de blocs logiques de base à l'instar des FPGA[23].

L'avantage qu'apportent les FPGA est d'élargir l'ensemble des portes logiques de base à toutes celles qu'un bloc logique d'un FPGA peut implémenter.

Les entrées du circuit ainsi que les sorties de chaque porte se voient attribuer un indice.

Chaque gène du chromosome définit une porte logique du circuit. Il se présente sous la forme d'un tableau d'entiers contenant le type de portes ainsi que les indices des portes qui lui servent d'entrées.

Comme nous pouvons le voir sur la fig.3.3 , le mapping entre le génotype et le phénotype est intuitif.

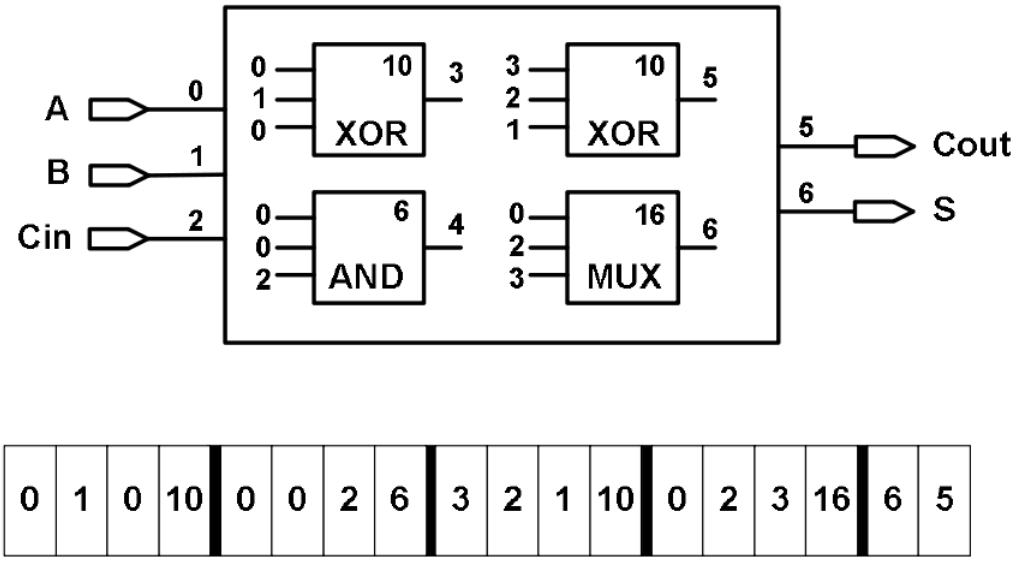

FIG. 3.3 – Représentation sous forme de matrice [8]

Ce modèle est encore d'actualité et a récemment été utilisé en 2008 par des chercheurs allemands pour valider un nouvel opérateur ainsi qu'une fonction de fitness qu'ils proposent.

3.7.2 Le codage en arbre

Avec ce codage, le circuit est modélisé sous la forme d'un arbre à plusieurs racines. Le nombres de racines dépend du nombre de sorties que possède le circuit logique. En partant d'une racine, on construit l'arbre de manière à ce qu'à chaque noeud soit placée l'opération logique et que chaque branche représente une connexion vers un autre opérateur logique. Les extrémités de l'arbre, appelées feuilles, sont les entrées du circuits.

Ce type de codage se prête bien à la programmation génétique qui est, par définition, spécialisée dans la manipulation d'une telle structure (cf.1.7) .

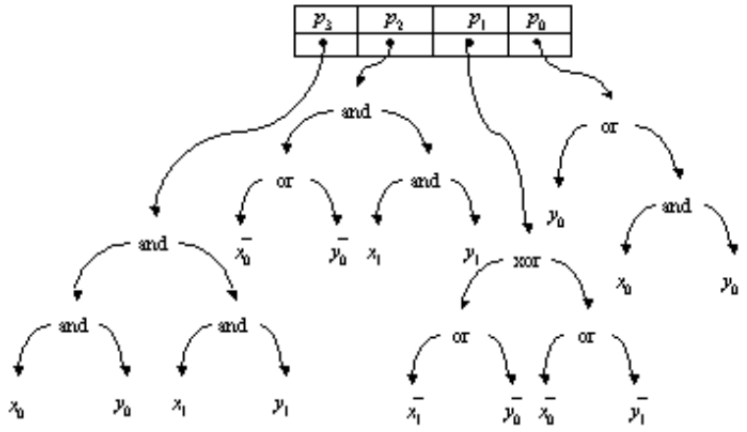

FIG. 3.4 – Représentation en arbre [28].

3.7.3 Discussion

Une étude comparative [28] à été menée sur les deux modèles présentés précédemment. Il en est sorti que le modèle sous forme de matrice est plus performant que celui en arbre.

L'expérience s'est déroulée sur quatre circuits de tests. Chaque processus d'évolution a été réalisé 100 fois pour chacun des circuits.

Il s'est avéré que les circuits obtenus étaient comparables en termes de taille et de temps de réponse. Néanmoins, le modèle en arbre nécessitait dix fois plus de générations pour trouver une solution. De plus, le gain en terme de performances est de 8.5 fois plus grand dans le modèle en matrice.

Nous avons donc choisi d'utiliser une représentation sous forme de matrice dans la modélisation du circuit au sein du framework.

Bien entendu, il n'est pas exclu que d'autres représentations ou des variantes puissent être implémentées.

3.8 Evaluation

Nous allons dès à présent étudier les différentes fonctions de fitness qui ont été développées dans divers travaux concernant le sujet de ce mémoire.

L'une des questions fondamentales que nous devons nous poser lors de la conception d'un algorithme évolutionniste est comment évaluer la qualité d'une solution.

L'outil mathématique est la fonction de fitness. Elle doit être adaptée au problème étudié.

Nous allons présenter quelques fonctions de fitness que nous avons rencontrées au cours de nos recherches.

La fonction de fitness la plus utilisée dans les travaux passés procèdent en deux étapes :

- L'évaluation de la fonctionnalité du circuit logique.
- L'évaluation du nombre de portes contenues dans le circuit logique.

Le but recherché est de maximiser la fonctionnalité et de minimiser le nombre de portes contenues dans le circuit.

Si nous considérons le modèle sous forme de matrice 3.7.1, la minimisation du nombre de portes se traduit par une maximisation du nombre de fils dans le circuit.

La fonction de fitness ci-dessous a été utilisée dans les travaux [8] et [11].

$$\text{Fit ness}(C) = \sum_{i=1}^{m} \sum_{j=1}^{2n} f_s(i,j) + W \text{ avec } f_s(i,j) = \begin{cases} 0 \text{ si } \hat{o}_{i,j} \neq o_{i,j} \\ 1 \text{ si } \hat{o}_{i,j} = o_{i,j} \end{cases}$$

où

- $\hat{o}_{i,j}$ est la valeur de la jème sortie de la ième combinaison du circuit à tester
- $o_{i,j}$ est la valeur de la jème sortie de la ième combinaison du circuit référence

$\sum_{i=1}^{m} \sum_{j=1}^{2n} f_s(i,j)$ représente la somme du nombre de sorties correctes pour toutes les combinaisons possibles d'entrées.

Si le circuit est fonctionnelle, on rajoute W qui est le nombre de fils dans le circuit.

3.9 Les opérateurs

Les algorithmes évolutionnistes se basent sur une série d'opérateurs qui agissent sur les génotypes mais ils ne spécifie à aucun moment le fonctionnement de ces opérateurs.

Dans notre cas, l'algorithme génétique doit posséder un certain nombre d'opérateurs génétiques afin de pouvoir manipuler les génotypes. Il est, donc, nécessaire de concevoir des opérateurs génétiques en tenant compte de la nature du problème étudié ainsi que du codage utilisé au sein de l'algorithme.

Cette section présente les différents opérateurs de croisement mis en oeuvre dans la littérature. La première partie abordera les opérateurs liés à la modélisation sous forme d'arbre et l'autre sous forme de vecteurs.

3.9.1 Croisement

Nous allons présenter un certain nombre d'opérateurs de croisement qui ont été imaginés aux cours des travaux passés.

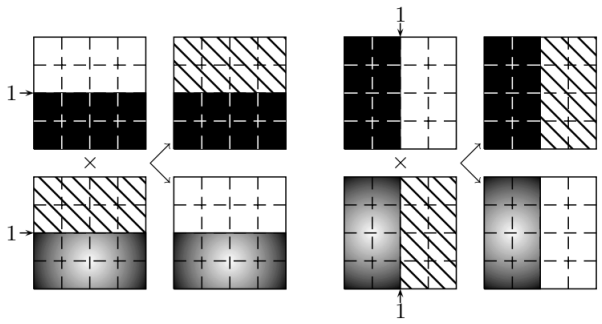

FIG. 3.5 – Croisement en 1 point pour le modèle sous forme de matrice [28].

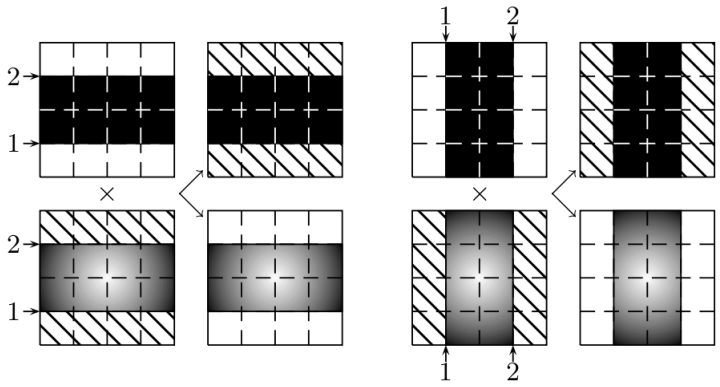

FIG. 3.6 – Croisement en 2 points pour le modèle en matrice [28].

3.9.2 Mutation

Ci-dessous se trouve une liste d'opérateurs de mutation qui ont été définis pour les circuits logiques.

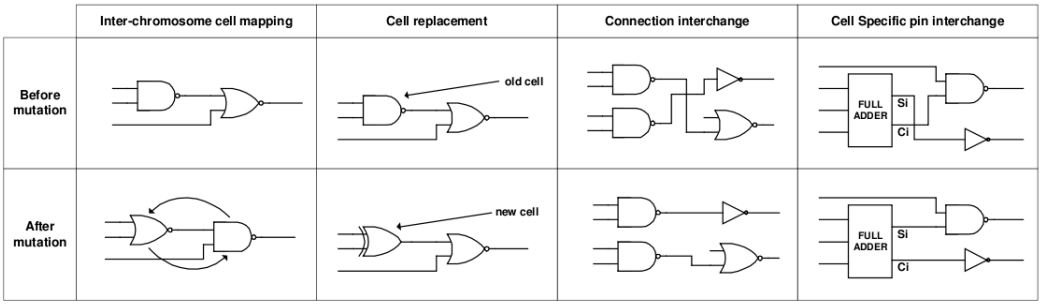

FIG. 3.7 – Opérateurs de mutation pour circuits logiques [7].

3.10 Résultats des travaux antérieurs

Cette section a pour mission de rappeler quelques résultats clés obtenus au cours de travaux antérieurs dans ce domaine.

Comme annoncé précédemment, Louis [21] proposa d'utiliser les algorithmes évolutionnistes dans le cadre du design de structures. Il l'appliqua notamment aux

circuits logiques et c'est au cours de ces travaux qu'il proposa le modèle sous forme de matrice.

Miller reprit l'idée et proposa d'utiliser les algorithmes évolutionnistes comme moteur d'innovation [23]. Il réussit à les optimiser et obtint de nouveaux designs pour les circuits logiques arithmétiques du chapitre 2. La figure 3.8 présente le circuit d'un additionneur 1 bit obtenu par des techniques évolutionnistes. Nous pouvons remarquer la présence d'un multiplexeur (MUX) à 2 entrées. Le circuit est totalement différent et plus econome en terme de portes que le circuit conventionnel voif fig.2.4. Le MUX est considéré comme une porte de base tout car comme nous avons vu au chapitre 2, les logiques programmables sont composées de blocs logiques pouvant synthétiser n'importe quel fonction logique de 4 à 6 entrées.

Sur la figure 3.9 se trouve le schéma d'un multiplieur 2 bits obtenu par Miller à l'aide de techniques évolutionnistes.

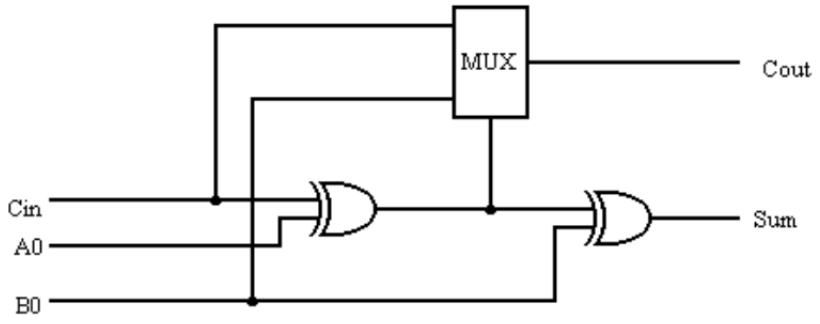

FIG. 3.8 – Full-adder 1 bit obtenu par évolution par Miller [23].

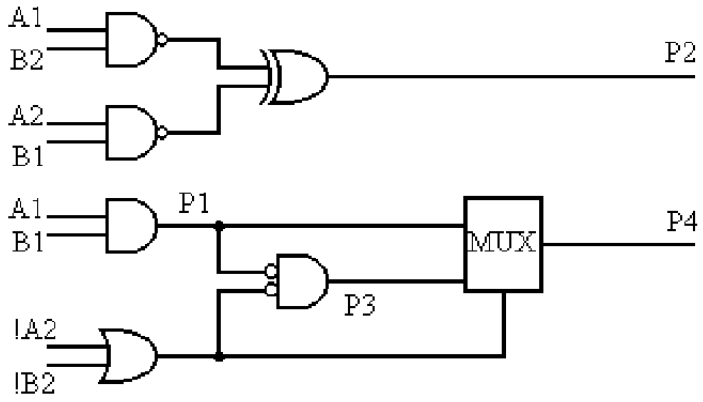

FIG. 3.9 – Multiplier 2 bits obtenu par évolution par Miller [23].

Un travail utilisant une optimisation multi-objective fût proposé par Coello [11]. La population est subdivisée en autant de sorties qu'en possède le circuit logique. Chaque sous-population subit un processus d'évolution dont la fonction fitness évalue une seule sortie.

Une étude comparative avec des techniques conventionnelles ainsi que les algorithmes génétiques standards montra clairement que les techniques évolutionnistes dépassent largement les designs humains. Par exemple, pour un testeur de parité à 3 bits, Coello a pu obtenir des circuits contenant au moins deux portes de moins (cf. fig. 5.13).

Les algorithmes évolutionnistes ont également été utilisés dans le cadre de la synthèse sur FPGA comme c'est le cas de Housell qui effectua des expériences en faisant évoluant un additionneur 1 bit, des multiplieurs ainsi que des circuits de reconnaissance de patterns (voir [7]).

Plus récemment, un groupe de chercheurs allemands a publié un article (voir [8]) dans lequel il présentait un nouvel opérateur de croisement ainsi qu'une nouvelle fonction de fitness. Les résultats obtenus montrent une amélioration comparés à ceux obtenus avec les opérateurs décrits au point 3.9.1

3.11 Conclusion

Dans ce chapitre, nous avons défini ce qu'est le design évolutionniste ainsi que l'intérêt d'une telle technique. Nous avons ensuite discuté des limites d'un tel processus.

Une présentation des différentes représentations et opérateurs utilisés pour les circuits logiques dans le cadre de techniques évolutionnistes a été donnée.

Et pour finir, nous avons publiés quelques résultats clés des expériences menées dans ce domaine par d'autres chercheurs.

Avec ce chapitre, nous avons terminé la phase théorique de ce mémoire. Les chapitres suivants entameront l'aspect pratique avec une présentation du framework logiciel qui a été développé ainsi que des résultats des expériences menées grâce au framework.

Chapitre 4

Framework logiciel

Contents

4.1 Introduction

Maintenant que la théorie n'a plus de secrets pour nous, nous pouvons aborder la réalisation proprement dite de mon mémoire.

En effet, il ne faut pas perdre de vue le but de ce mémoire, à savoir le développement d'un framework logiciel afin d'expérimenter des techniques évolutionnistes sur le design de circuits logiques.

Avant de s'attaquer au coeur du problème, un état de l'art s'impose. Nous y traiterons des différents framework manipulant les algorithmes évolutionnistes. Ensuite, il sera question de l'analyse et de la mise sur papier de l'architecture du framework et pour finir, nous allons présenter la méthode de développement d'un algorithme grâce au framework.

4.2 Etat de l'art

Cette section présentera brièvement les différents frameworks logiciels existants pour l'implémentation d'algorithmes évolutionnistes.

L'évaluation et la comparaison de ces frameworks logiciels sortent du cadre de ce mémoire.

Les frameworks pour manipuler des algorithmes évolutionnistes sont légion. Les plus connus sont EO, Galib, Beagle, Paradiseo, Matlab, etc.

La particularité de ces frameworks est qu'ils sont très généralistes. Pour pouvoir les utiliser, il est souvent nécessaire d'effectuer des adaptations à plusieurs niveaux.

4.3 Motivation

Comme nous l'avons fait remarquer précédemment, les frameworks logiciels existants sont très généralistes.

Le fait de repartir sur de nouvelles bases permet, d'une part, d'avoir une nouvelle approche et, d'autre part, de spécialiser le framework au design de circuits logiques.

4.4 Choix technologiques

Les algorithmes génétiques sont d'une simplicité exceptionnelle dans leur conception. Malheureusement, le revers de la médaille est la phase d'évaluation qui dépend fortement du problème ciblé.

Dans notre cas, étant donné que nous manipulons des circuits logiques, l'effort d'évaluation subit une explosion combinatoire lorsque le nombre de portes augmente.

En effet, chaque couple entrée/sortie doit être testé et comparé au circuit de référence. Or, ce nombre est exponentielle au nombre d'entrées.

Cette réflexion a influencé énormément le reste du framework, notamment le choix du langage de programmation.

Java et Python ont été rapidement écartés à cause du fait que leur code source est interprété, en raison du passage au travers une machine virtuelle et de l'impossibilité d'optimisation de la mémoire.

Le premier essai d'implémentation du framework a été réalisé en C#.Net mais, une prise de conscience du besoin de puissance et d'optimisation en tout genre, cette technologie a très vite été abandonnée.

Notre choix s'est porté alors sur le C++ qui malgré sa complexité de développement offre des possibilités sans limites au programmeur averti.

De plus, moyennant une compilation et quelques modifications, il est tout à fait possible de porter une application C++ vers d'autres plate-formes.

4.5 Analyse

4.5.1 Cas d'utilisation

La première étape de l'analyse consiste en la définition des cas d'utilisation, use cases en anglais, auxquels doit répondre le framework.

L'utilisateur est défini comme la personne qui conçoit un algorithme à l'aide du framework en vue de réaliser des expérimentations de design évolutionniste.

- Paramétrisation de l'algorithme
- Récupération des données
- Simulation de circuits
- Définition de la fonctionnalité
- Modification de l'algorithme aisée
- Définition de nouveaux opérateurs

4.5.2 Décomposition

Le framework peut être décomposé en différentes parties.

- Le codage des circuits logiques
- La modélisation et la simulation des circuits logiques
- Les opérateurs
- Les étapes de l'algorithmes
- Les statistiques
- La configuration

4.6 Modélisation et codage

Cette section traite des structures de données manipulées par les algorithmes évolutionnistes.

Comme nous avons pu le voir au chapitre I, un algorithme génétique agit sur trois plans différents :

- Le bagage génétique
- L'individu
- La population

Une population est composée d'individus qui possèdent un certain nombre de chromosomes composés de gènes.

Dans le cadre de ce travail, nous supposons qu'un individu ne possède qu'un seul chromosome.

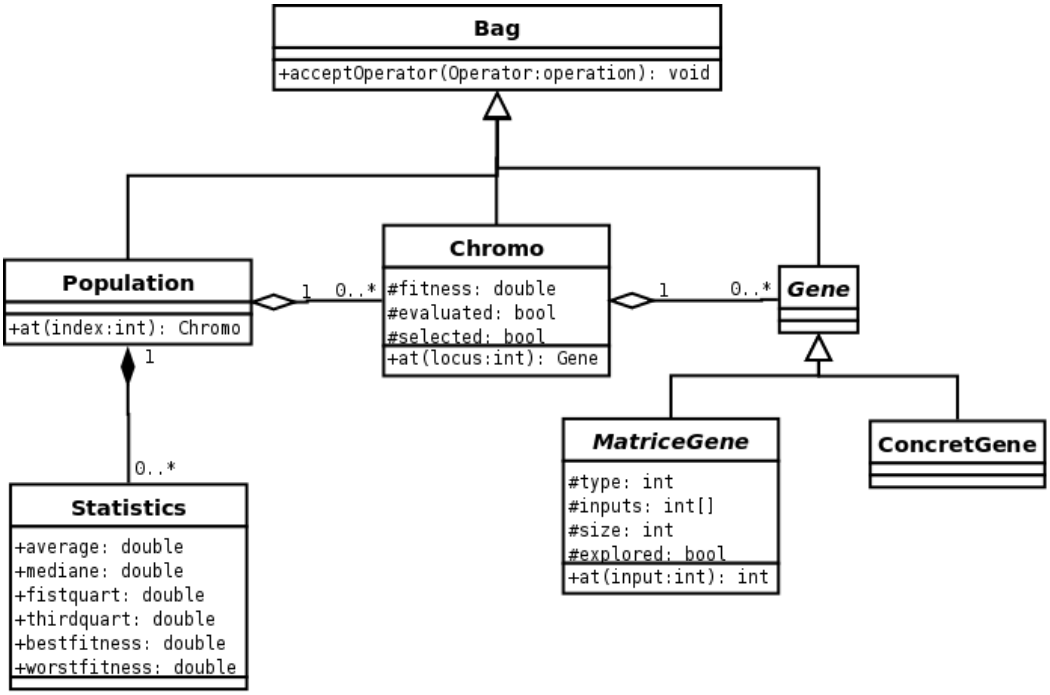

FIG. 4.1 – Diagramme UML des structures de données employées dans le framework.

Nous avons donc reproduit le mécanisme décrit ci-dessus sur base des principes de l'orienté objet.

Chaque entité du point précédent est modélisée par une classe. Elles héritent toutes de la classe *Bag*.

La population possède un objet de type *Statistics* qui est la structure de stockage des données statistiques que manipule le framework.

Le codage qui a été mis en place pour tester le framework et pour la réalisation des expériences est inspiré de la représentation sous forme de matrice du point

3.7.1.

Nous avons opté pour un codage sous la forme d'un vecteur contenant des groupements d'entiers. Chaque code (chromosome) est composé d'objets de type Gene. Chaque gène est quant à lui composé de deux parties :

– Le type de la porte.

– Les indices des portes qui servent d'entrée à la porte concernée.

Le type de porte est défini selon le codage du tableau ci-dessous.

Code	Fonction
0	AND
1	OR
2	XOR
3	NAND
4	NOR
5	XNOR
6	NOT
7	MUX
8	OUT

TAB. 4.1 – Codage employé

Même la sortie est codée sous la forme d'une porte. Nous rappelons que le mécanisme qui a été implémenté permet aussi d'ajouter des types autres que ceux répertoriés dans le tableau ci-dessus.

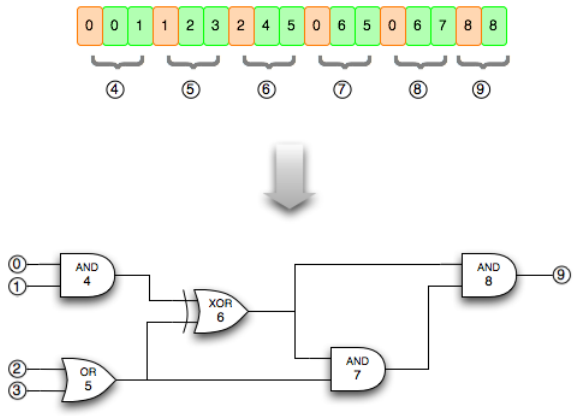

FIG. 4.2 – Codage/Décodage utilisé au sein du framework

4.7 Gestion de l'algorithme

Comment introduire une modularité dans l'algorithme ? La réponse se trouve dans le chapitre 1.

En effet, le caractère séquentiel des étapes d'un algorithme évolutionniste a inspiré énormément l'implémentation du framework.

Nous allons tenter de relever les points communs de cette famille d'algorithmes.

- Ils manipulent une population
- Une suite d'étapes distinctes constitue un cycle de traitement
- Chaque étape effectue un traitement spécifique à la population courante

Cette analyse, nous a poussé à faire appel à un design pattern peu courant : le *Chain-of-responsibility [15].* Nous allons fournir une description du patron adapté à notre cas.

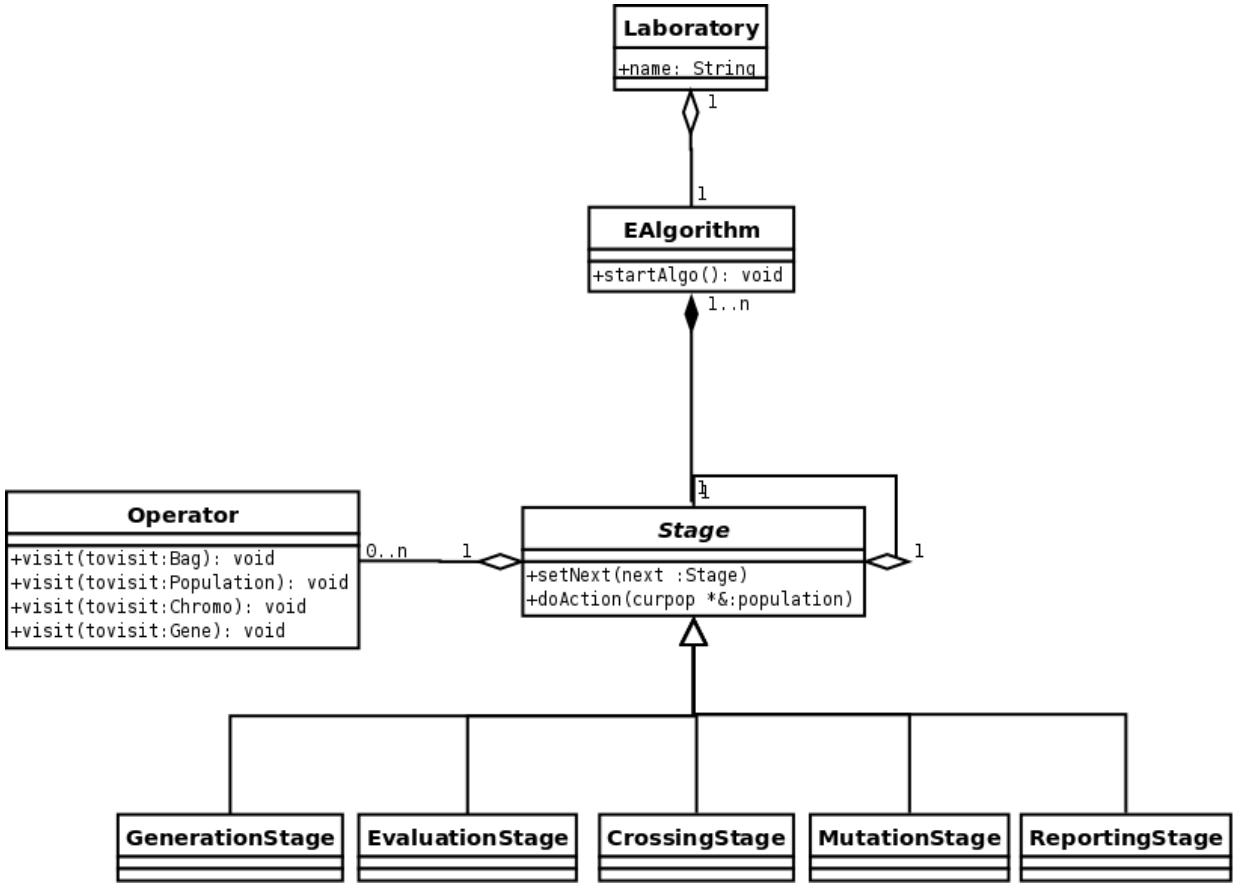

FIG. 4.3 – Modélisation d'un algorithme évolutionniste.

L'application du patron nécessite que chaque étape hérite d'une classe commune appelée ici *Stage* voir fig.4.3.

Les *Stages* sont des classes qui effectuent un traitement sur une population.

Ils sont reliés entre eux sous la forme d'une liste liée. Ceci permet d'insérer des traitements dans un algorithme donné. Tous les Stage implémentent la méthode doAction() qui reçoit comme paramètre un pointeur vers la population en cours de traitement.

A la fin de ce dernier, le Stage fait appel à la méthode du Stage suivant. Chaque Stage possède un jeu d'opérateurs qui peuvent être ajoutés dynamiquement ou à l'initialisation de l'algorithme.

Nous avons implémenté différents Stage de base pour un algorithme évolutionniste. De plus comme, la structure d'un Stage est indépendante du modèle

choisi pour le génotype, ils peuvent sans problèmes être réutilisés. Les Stages mis en place sont :

- GenerationStage : s'occupe de la génération de circuit et fait appel au générateur de chromosomes. Ce dernier est dépendent du modèle choisi pour le génotype.
- EvaluationStage : effectue un parcourt de la population et évalue les différentes solutions en faisant appel aux différents opérateurs d'évaluation.
- SelectionStage
- CrossingStage
- MutatingStage
- ReportingStage : constitue l'étape d'analyse statistique de la population. Elle est aussi responsable d'arrêter l'algorithme dès que les conditions d'arrêts sont atteintes.
- AdaptatingStage : consiste à considérer que, suivant l'analyse obtenue par le *Reporting*, l'algorithme puisse modifier ses paramètres afin, par exemple, d'augmenter le taux de mutation. Cependant, cette idée n'a pu être implémentée faute de temps.

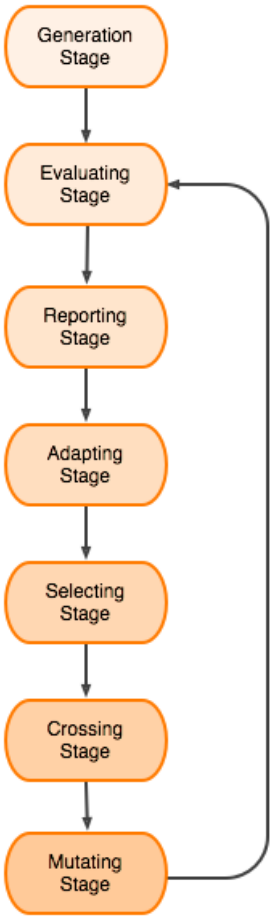

FIG. 4.4 – Exemple de mise en place des différents Stages.

Les expériences

Il ne faut pas perdre de vue l'un des buts principaux de ce mémoire, à savoir l'expérimentation. En plus d'offrir un framework pour l'implémentation des algorithmes évolutionnistes, il est nécessaire de permettre une certaine souplesse pour le design des expériences que l'utilisateur souhaite réaliser.

Nous avons décrit jusqu'ici les briques de base pour l'implémentation d'un algorithme évolutionniste en terme de générations. Mais, lorsque nous réalisons une expérience avec des algorithmes de cette sorte, il est nécessaire de pouvoir effectuer des statistiques sur un certain nombre de répétitions appelées *run*. C'est pour cela que nous avons implémenté la classe *laboratory* dont le but est d'exécu-

ter une instance de la classe *EAlgorithm* autant de fois que cela est défini dans le fichier de configuration 4.11.2.

4.8 Les opérateurs

Comme vu au chapitre I, les différents opérateurs que manipule un algorithme génétique dépendent fortement de la forme du génotype et donc du codage.

Ainsi, il est nécessaire de faciliter l'implémentation des opérateurs de mutation et de croisement en fonction du modèle introduit par l'utilisateur du framework.

Au delà de cette dimension, l'ajout de nouveaux opérateurs de manière dynamique ou avant la compilation peut s'avérer être un atout.

Dans un algorithme génétique, nous pouvons rencontrer divers opérateurs tels que :

- Opérateurs de mutation
- Opérateurs de croisement
- Opérateurs d'évaluation

Nous avons introduit l'opérateur d'évaluation car nous supposons que l'utilisateur doit pouvoir définir de nouvelles fonctions de fitness et, par extension, lui permettre de faire une optimisation à plusieurs étages.

Les opérateurs ont été implémentés selon le patron visiteur dérivé à notre manière.

Tous les opérateurs héritent de la classe Operator et redéfinissent la méthode visit() selon le type de données qu'ils doivent traiter.

Comme nous pouvons le voir sur la figure ci-dessous (fig.4.5), nous avons laissé libre cours au concepteur de l'algorithme de définir également des opérateurs de population. L'idée est de pouvoir concevoir des algorithmes tels que celui utilisé par Coello dans son approche multi-objective.

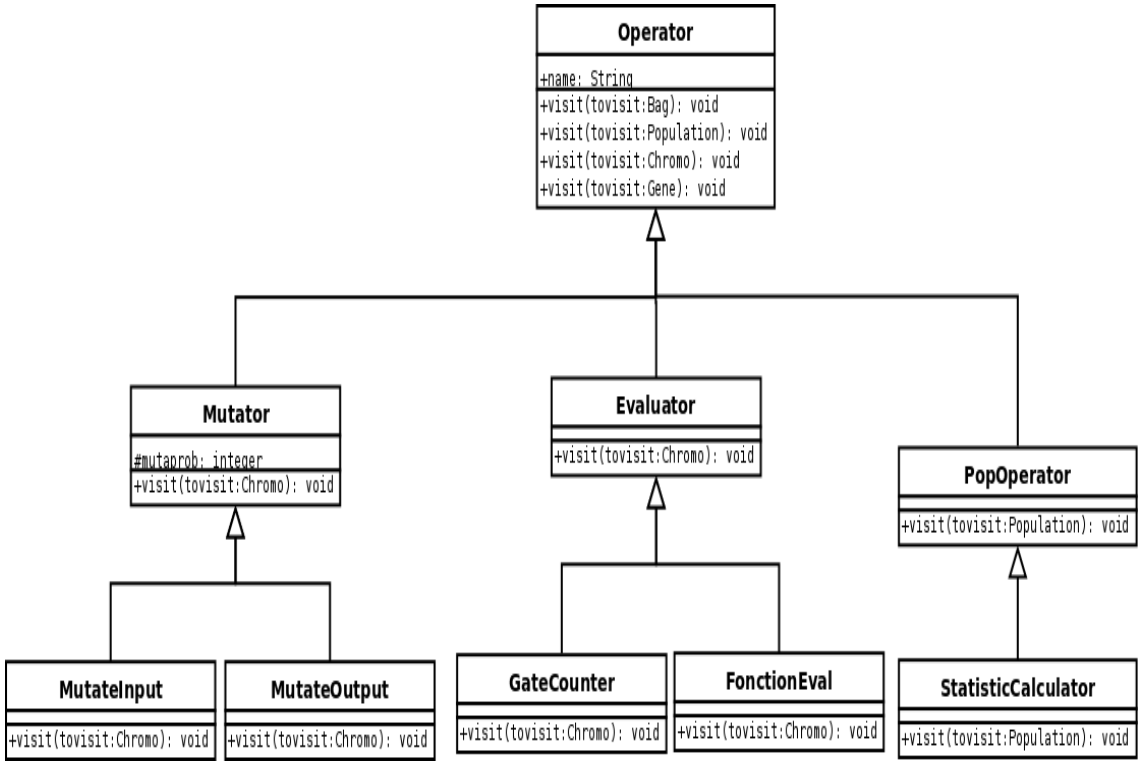

FIG. 4.5 – Diagramme UML de la gestion des opérateurs

4.8.1 Opérateurs de mutation

Nous avons implémenté trois types d'opérateurs de mutation.

- *ChangeType* : sélectionne au hasard un gène d'un chromosome et change son type.

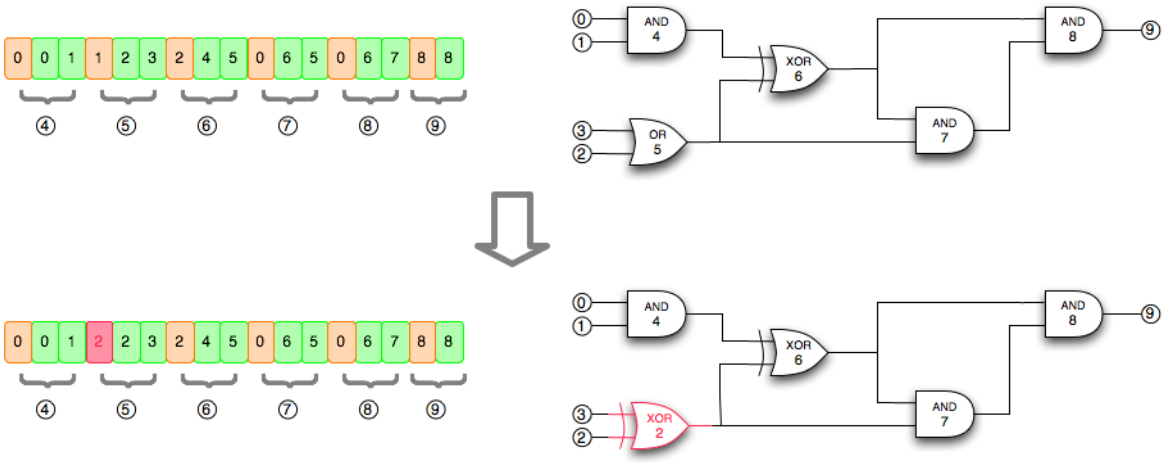

FIG. 4.6 – Opérateur mutation : changement de type d'une porte.

- *ChangeOutput* : reconnecte une sortie prise au hasard à une autre porte.

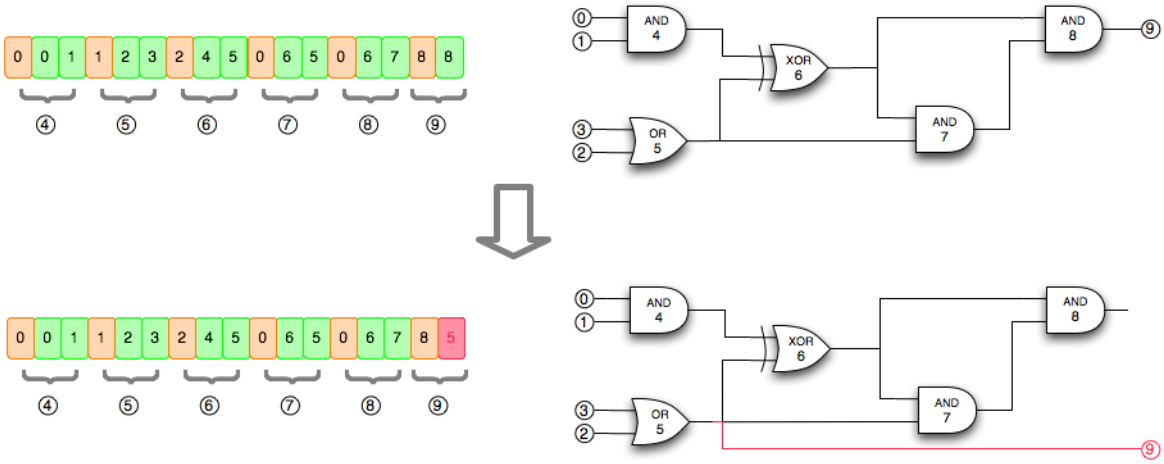

FIG. 4.7 – Opérateur mutation : changement de la sortie du circuit.

- *ChangeInput* : modifie les entrées d'une porte prise au hasard sur le circuit.

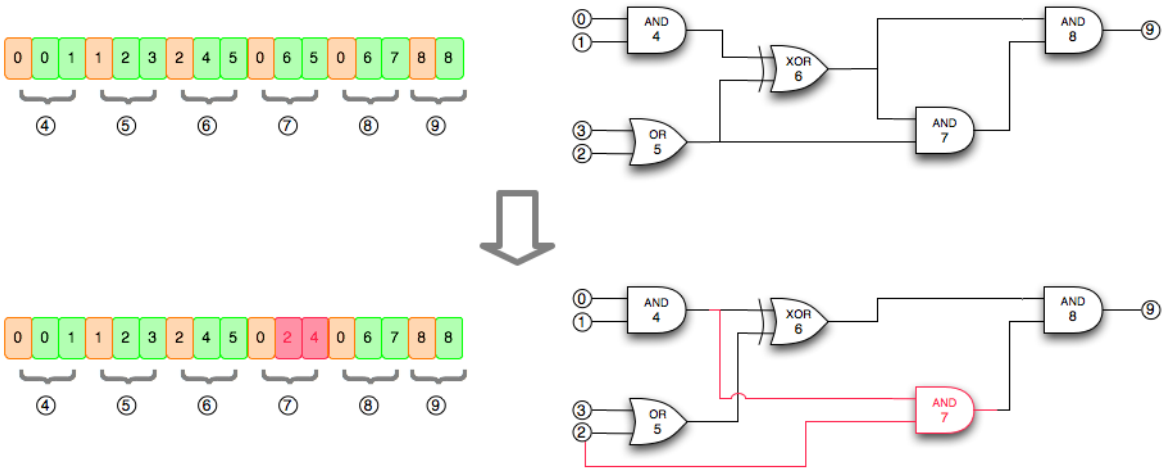

FIG. 4.8 – Opérateur mutation : changement des entrées d'une porte.

4.8.2 Opérateurs de croisement

Nous avons implémenter l'opérateur de croisement le plus simple : le croisement en un point. Sur la figure 4.9, nous pouvons observer l'effet du croisement aussi bien sur les chromosomes que sur les circuits.

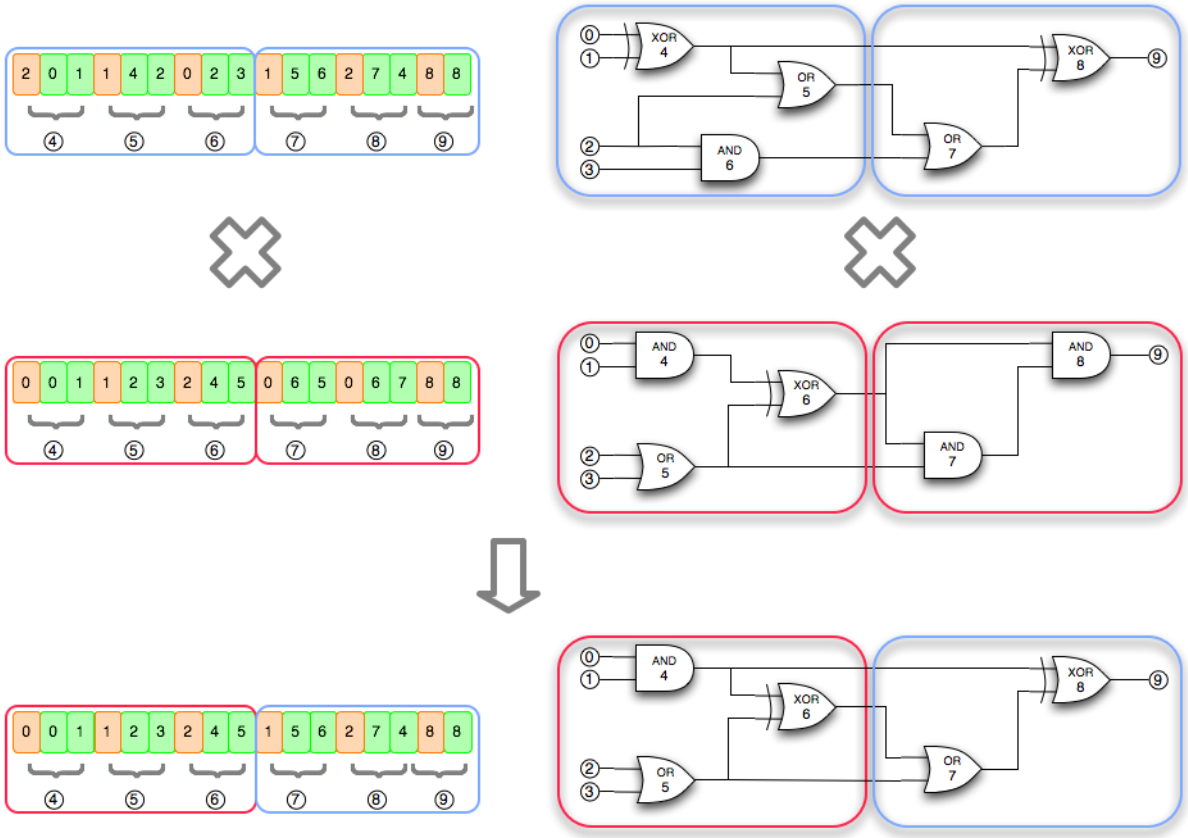

FIG. 4.9 – Opérateur croisement en un point implémenté au sein du framework.

4.9 La simulation

L'évaluation des circuits est réalisée pour l'instant grâce à un simulateur qui a été conçu dès le début du mémoire.

Le simulateur est rudimentaire en ce sens qu'il ne peut évaluer que des circuits combinatoires et que, pour l'instant, il ne permet pas de simuler des circuits à grand nombre d'entrées.

Il se décompose en deux parties :

- Le comparateur de circuits
- La modélisation d'un circuit

Le comparateur de circuit a pour mission d'évaluer un circuit par rapport à une table de vérité ou un circuit donné comme référence.

Grâce au gestionnaire de configuration, le simulateur précharge la table de vérité de référence en mémoire. Néanmoins, suite à la discussion du point 2.5, cette technique n'est plus possible lorsque le nombre d'entrées du circuit est trop grand.

Chaque chromosome qui doit être évalué est décodé par le simulateur en un circuit.

4.9.1 La modélisation d'un circuit

Un circuit est composé de portes logiques interconnectées entre elles. Pour modéliser un circuit, nous avons opté pour une approche orientée objet.

Les portes logiques

Une porte logique est modélisée sous la forme d'une boîte noire possédant un certain nombre d'entrées et de sorties. Nous nous sommes limités dans le cadre de ce mémoire aux portes logiques à une sortie.

Chaque type de portes logiques possède sa propre classe qui définit :

- Le nombre d'entrées
- Les valeurs des entrées
- Le nom du type
- La fonction logique qui calcule la sortie en fonction des entrées

Nous avons implémenté les différentes portes logiques de base.

Le circuit logique

Un circuit logique est modélisé par une classe possédant comme attributs :

- Un certain nombre d'entrées
- Un certain nombre de sorties
- Une fonction de calcul de l'état de sortie
- La liste des portes dont elle est l'entrée

4.9.2 Le simulateur

Le simulateur est composé d'une part d'une classe *Timer* qui génère les différentes combinaisons des entrées. Nous avons reproduit le mécanisme d'un comp-

teur électronique. Plusieurs portes de type *Toggle* sont interconnectées entre elles dont le comportement est d'inverser leur état de sortie quand l'entrée passe d'un état bas à un état haut.

Un objet de la classe *Timer* est connecté aux circuit et lui fourni toutes les combinaisons des entrées une par une. Nous comparons ensuite les valeurs obtenues à celles attendues.

4.10 Spécification du circuit recherché.

La fonctionnalité du circuit recherché peut être introduite de deux manières différentes : soit sous forme d'une table de vérité soit sous la forme d'un circuit de référence.

La table de vérité est lu sur un fichier contenant les valeurs que le circuit doit posséder pour chaque combinaison. Elle est lu par le singleton *Configurator*.

Si la table de vérité est trop grande il est possible d'introduire le code correspondant à un circuit existant en instanciant chaque gène suivant le codage définit plus haut.

4.11 Utilitaires

Cette section décrit les différentes classes de gestion. Elle n'interviennent pas directement dans un algorithme mais sont amplement utilisées pour gérer diverses tâches telles que la gestion des types de portes, la génération des nombres pseudo-aléatoires ainsi que la gestion des paramètres de l'algorithme.

Leur caractéristique est qu'elle peuvent être appelées à partir de n'importe quel endroit du code. C'est pour cela que nous avons décidé de faire appel au pattern *Singleton* qui garanti l'unicité. Comme nous pouvons le voir sur la figure 4.10, les trois classes qui doivent posséder une instance unique durant toute l'exécution héritent de la classe *Singleton*.

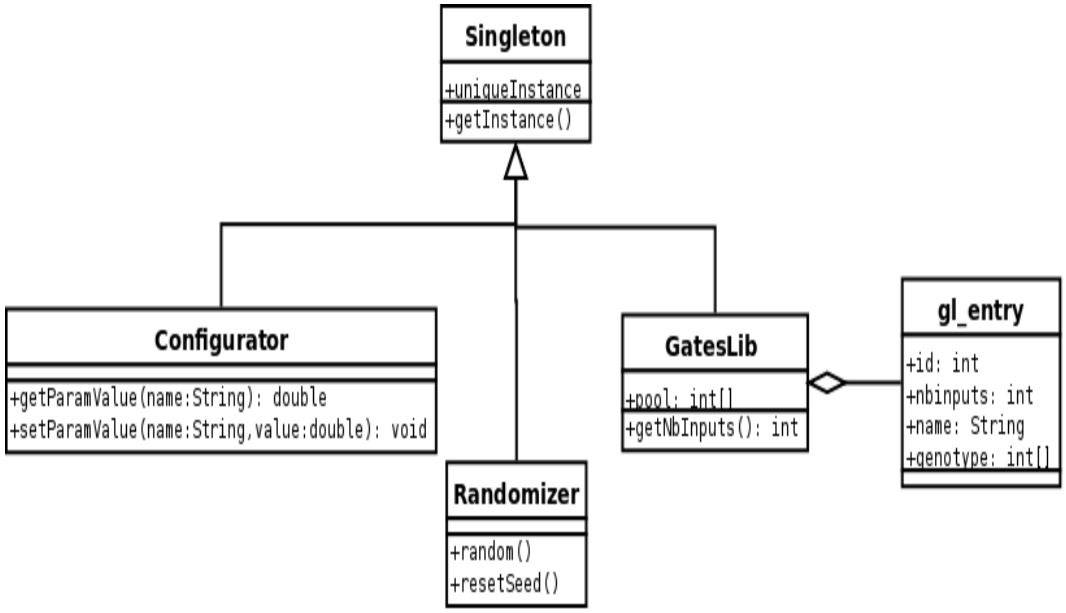

FIG. 4.10 – Classes de gestion des types de portes, de la génération de nombres aléatoires et de la configuration des paramètres.

4.11.1 La librairie des portes logiques

La gestion des portes logiques est importante en ce sens qu'il est nécessaire de définir les portes logiques que l'utilisateur souhaite utiliser dans son circuit.

L'utilisateur doit aussi pouvoir introduire des portes dont il fournit soit la table de vérité soit la fonction logique. Chaque porte logique possède un certain nombre d'entrées mais est limitée à une entrée pour l'instant.

Cette tâche est laissée à la classe GatesLib. Cette dernière hérite de la classe Singleton afin d'assurer l'unicité de l'objet et de centraliser les données concernant les portes logiques.

Nous proposons comme idée future de permettre au framework de charger dynamiquement les caractéristiques des différents types de portes logiques ou de circuits logiques à partir d'une base de données.

De plus, une des idées concernant le fonctionnement de l'algorithme génétique est de pouvoir repérer de bons patterns dans les circuits d'une population et d'en faire des building blocs pour la génération suivante.

Il serait donc intéressant de pouvoir introduire de nouvelles portes logiques

dynamiquement dans la base de données.

4.11.2 La gestion de la configuration

Les algorithmes évolutionnistes possèdent un grand nombre de paramètres de configuration disséminés un peu partout dans leur processus. De plus, l'efficacité de l'algorithme dépend fortement de ces paramètres. Il est, donc, nécessaire de pouvoir les modifier aisément au lancement ou même en cours d'exécution. Pour répondre à ce besoin, un mécanisme de gestion des paramètres a été développé.

Le principe est de regrouper tous les paramètres au sein d'une classe dont le rôle est de :

- Lire le fichier de configuration
- Initialiser les paramètres
- Offrir une interface pour accéder et modifier les paramètres au reste du programme
- Respecter l'unicité des paramètres

Une classe singleton a été implémentée. Elle se construit autour d'un tableau associatif dont la clé est le nom du paramètre et l'élément, la valeur du paramètre.

Le fichier de configuration

Le fichier configuration contient les différents paramètres et leur valeur. Dans le format qui est utilisé à l'heure actuelle, chaque paramètre est défini par une ligne contenant son nom suivi de sa valeur.

Bien entendu, certains de ces paramètres sont obligatoires tels que la taille de la population, le nombre de générations et les probabilités de mutation et de croisement.

Ci-dessous, se trouve un exemple de fichier de configuration.

```
maxrun 100
maxgeneration 6000
mutationprob 30
populationsize 50
nbgatesmax 15
nbinputs 4
```

```
nboutputs 4
pcelites 2
pcselected 50
goodfitness 80
goodelement 100
SA_initTemp 100
SA_downcoef 90
SA_nbiteration 10
SA_finalTemp 1
```

4.11.3 La génération de nombres aléatoires

Dans la conception d'un algorithme génétique, il ne faut pas négliger son caractère stochastique. Le hasard intervient à tous les niveaux d'un algorithme génétique, de la génération de la population aux choix des opérateurs à appliquer, en passant par la sélection.

Malheureusement, les générateurs de nombres aléatoires ne sont pas parfaits dans le monde informatique. On parle le plus souvent de générateurs de nombres pseudo-aléatoires qui est un algorithme qui génère une séquence de nombres presque aléatoires. Il est difficile d'obtenir un réel générateur de nombres aléatoires sans passer par la physique.

Dans le cadre des algorithmes génétiques, un générateur pseudo-aléatoire convient très bien à condition qu'il ne soit pas biaisé.

Etant donné que le choix du langage de programmation fût le C++, il semble naturel de vouloir utiliser la fonction rand().

Malheureusement, ce n'est pas un bon générateur de nombres aléatoires(cf.[32]).

Nous avons décidé de faire appel à la GSL, GNU Scientific Library, qui comme son nom l'indique, est une librairie scientifique très complète contenant la plus part des outils mathématiques et, plus particulièrement, des générateurs de nombres pseudo aléatoires performants[38].

Dans le cadre ce mémoire, nous nous sommes limités à l'utilisation des générateurs pseudo-aléatoires fournis par la GSL. Néanmoins, nous avons effectué une expérience pour s'assurer de l'uniformité du générateur. Bien que la GSL fût adoptée dans le cadre de ce mémoire, il serait intéressant de laisser libre cours au

choix de l'utilisateur dans l'éventualité qu'un algorithme plus performant soit implémenté. Pour garder cette flexibilité et une évolution possible du mécanisme de génération de nombres aléatoires, nous avons décidé d'implémenter le générateur à l'aide d'un pattern stratégies. Tous les générateurs de nombres aléatoires doivent se conformer à une interface commune. Comme le générateur est le même durant le déroulement de l'algorithme et à tous les niveaux d'exécution, il hérite aussi de la classe Singleton pour assurer ce caractère unique.

4.12 Traitement des résultats

4.12.1 La gestion des solutions

Afin de pouvoir récupérer les solutions que génère l'algorithme génétique, nous avons mis en place un mécanisme d'exportation vers des formats de fichiers divers. A l'aide d'un pattern stratégie, il est tout à fait possible pour une tierce personne de coder son propre exporter vers d'autres formats.

Pour l'instant, seul un exporter vers un format Graphviz a été codé. Graphviz est un utilitaire permettant de réaliser facilement des graphes de réseaux et autres [3].

Mais, le but visé est de réaliser un exporter vers des formats plus proches des simulateurs de circuits existants sur le marché.

L'exemple classique serait un exporter vers des fichiers VHDL ce qui permettrait, d'une part, de pouvoir simuler le circuit sur un simulateur plus sophistiqué et, d'autre part, une implémentation hardware sur un FPGA ou autre logique reprogrammable en vue d'essais physiques.

4.12.2 Les statistiques

Il serait intéressant de pouvoir recueillir diverses statistiques concernant le processus d'évolution.

Nous avons, d'une part, les statistiques concernant l'évolution de la population et, d'autre part, les statistiques de performance.

Il serait intéressant d'avoir une idée de la répartition du fitness au sein d'une population donnée.

Afin de recueillir les données statistiques concernant une population, nous avons implémenté une structure de données comportant plusieurs champs tels que :

- Le fitness du meilleur individu
- Le 3ème quartile
- La médiane
- Le 1er quartile
- Le plus mauvais des fitness
- La moyenne

A la fin de chaque run, un fichier est généré comportant pour chaque génération, les statistiques précédentes. Le fichier est conforme au format de GNUPLOT, qui est un outil libre d'affichage de graphes voir [2]. C'est grâce à cet outil que nous avons généré les graphes du chapitre concernant les expérimentations.

4.12.3 La journalisation

Afin de s'assurer du bon fonctionnement d'une expérience, nous avons implémenté un mécanisme de débogage.

L'utilisateur peut définir le niveau de détail des d'informations affichées à la console durant l'exécution d'un algorithme.

Pour cela, nous avons fait appel à la puissance des commandes de préprocesseur.

L'utilisateur doit simplement définir une macro DEBUG_LEVEL X, où X est le niveau de debug, avant la fonction principale du programme.

Nous avons défini quatre macros d'affichage de niveau hiérarchique définies grâce aux macros PRINTX(...), où x représente le niveau de détail.

4.13 Problèmes rencontrés

Fuites de mémoire

L'un des problèmes qui s'est très vite manifesté concerne la fuite de mémoire. Ceci était dû à l'oubli de la libération des ressources à certains endroits du code.

Une exécution de l'algorithme implémenté consommait énormément de mémoire après quelques *run*s.

Nous avons résolu le problème grâce à des outils de profilage de code et de détection de mauvaise gestion de la mémoire tel que *Valgrind*.

Malheureusement, nous n'avons pas encore à l'heure actuelle implémenté un mécanisme efficace qui pourrait automatiquement gérer la gestion de la mémoire pour éviter à l'utilisateur de devoir y penser.

4.14 Mise en oeuvre du framework

Nous allons expliquer comment utiliser le framework pour implémenter des algorithmes évolutionnistes.

4.14.1 Algorithme génétique

Pour implémenter un algorithme génétique, nous avons besoin de faire appel à tous les *Stages* décrits au point 4.7.

Comme dit précédemment, un algorithme se présente sous la forme d'une classe héritant de *EAlgorithm*

Le code principal de l'algorithme se trouve dans la méthode ci-dessous.

```
void startAlgo()
{
      //Faire appel à l'initialiseur pour charger la table de vérité
      this->cfg->loadThruthTable(``table.dat'');

      //Création des Stages de l'algorithme
      GenerationStage * generation = new GenerationStage();
      CrossingStage * crossing= new CrossingStage();
      EvaluationStage * evaluation= new EvaluationStage();
      MutationStage * mutation= new MutationStage();
      ReportingStage * reporting= new ReportingStage();

      //Création des opérateurs
```

```
        Evaluator * fonctioneval= new Evaluator();
        GateCounter * gatecounter= new GateCounter();
        Mutator * mutateType= new MutateType();
        Mutator * mutateOutput= new MutateOutput();
        Mutator * mutateInput=new MutateInput();

        //Ajout des opérateurs aux stages
        mutation->addMutator(mutateType);
        mutation->addMutator(mutateOutput);
        mutation->addMutator(mutateInput);

        evaluation->addEvaluator(fonctioneval);
        evaluation->addEvaluator(gatecounter);

        //Création de l'algorithme
        generation->setNext(evaluation);
        evaluation->setNext(reporting);
        reporting->setNext(selection);
        selection->setNext(crossing);
        crossing->setNext(mutation);
        mutation->setNext(evaluation);

        generation->doAction(this->currentpopulation);
        deletePopulation();

}
```

Comme nous venons de le voir, la construction d'un algorithme évolutionniste est aisée.

4.14.2 Hybridation

Si l'utilisateur désire hybrider l'algorithme ci-dessus, il lui suffira de coder son *Stage* à lui et de l'injecter dans la chaîne de traitement.

Soit un nouveau *Stage* de recuit simulé dénommé *SAStage*, le code ci-dessus devient :

```
...
     SAStage * recuitSimule= new SAStage(...) ;
     //Creation de l'algorithme
      generation->setNext(evaluation);
      evaluation->setNext(reporting);
      reporting->setNext(selection);
      selection->setNext(crossing);
      crossing->setNext(recuitSimule);
      recuitSimule->setNext(evaluation);

     //La mutation n'est plus nécessaire
     // mutation->setNext(evaluation);
...
```

4.15 Travaux futurs

Bien qu'à l'heure actuelle le framework est opérationnel, il subsiste encore des points d'ombres.

Cette section pointe du doigt les parties qui requièrent des améliorations.

Format de sortie Il serait nécessaire de pouvoir exporter les circuits obtenus vers des formats supportés par les simulateurs du marché. Nous avons déjà proposé un exportateur vers des fichiers VHDL qui peuvent être implémentés sur un FPGA pour des essais physiques.

Reconnaissance de pattern Une autre idée est d'essayer de reconnaître des patterns au sein d'un circuit afin de former de nouvelle porte logique. On entend par pattern un ensemble de portes interconnectées entre elles.

Un début d'implémentation a été réalisé du côté de la modélisation du circuit logique mais rien en ce qui concerne la détection des patterns.

Simulation La simulation compare la fonctionnalité d'un circuit à une table de vérité stockée en mémoire. Cette technique devient impraticable lorsque le nombre d'entrées augmente. Il ne serait pas exclu de faire appel à un simulateur externe.

Interface graphique Le développement d'une interface graphique a été amorcé en python, mais vu le temps accordé, nous avons dû nous en passer. L'idée était que l'utilisateur puisse interconnecter de manière graphique les blocs, *Stage et Operator*, constituant l'algorithme qu'il conçoit.

4.16 Conclusion

Après avoir mis défini le cahier des charges du framework, nous avons proposé une découpe pour l'implémentation. Chaque partie a été implémentée en faisant appel à des techniques de conception orientées objet. C'est ainsi que l'utilisation de design patterns, ou patrons de conception, ont permit d'obtenir un framework modulaire.

Nous avons également présenté les opérateurs et le modèle que nous avons implémenté pour réaliser nos expériences. En guise de conclusion, nous avons fourni deux exemples d'algorithmes évolutionnistes codés à l'aide de notre framework.

Désormais, il est nécessaire de valider le fonctionnement du framework. Nous allons donc procéder à la description des expériences qui ont été menées et dresser un détaillé des résultats obtenus.

Chapitre 5

Expérimentations

Contents

5.1 Introduction

Ce chapitre regroupe les résultats des différentes expériences que nous avons réalisées.

La première expérience a pour objectif de valider le fonctionnement du framework et a pour objet l'évolution de l'additionneur 1 bit.

Ensuite, nous comparerons différentes variantes d'algorithmes génétiques d'un point de vue statistique.

Nous tenterons également de tester ce que produirait l'évolution du design d'un parity check circuit (voir chapitre 2).

En guise de conclusion, une liste d'expériences réalisables dans le future sera fournie.

5.2 Remarques préliminaires

Cette section décrira le schéma suivi par les différentes expériences.

Pour chaque expérience réalisée, nous fournissons un descriptif détaillé contenant le but de l'expérience, le type d'algorithme utilisé, la valeur des paramètres et la définition des données à récolter.

Ensuite, nous présentons les graphiques ainsi qu'une description des résultats obtenus.

Une discussion de chaque résultat sera fournie avant de conclure.

Remarque 1 Le taux de mutation est exprimé en pourcentage d'individus par population dont un seul de leur gène subit une seule des modifications possibles.

5.3 Expérience 1

5.3.1 But de l'expérience :

Le but de cette expérience consiste à valider le bon fonctionnement du framework en faisant évoluer un additionneur 1 bit.

Nous avons choisi ce circuit car il a été la cible d'algorithmes évolutionnistes à plusieurs reprises dans la littérature [23]. Cela permet de vérifier si nous obtenons les mêmes résultats en terme de design que les travaux passés.

5.3.2 Paramètres

Sujet : Evolution de l'additionneur 1 bit.

Type d'algorithme : Algorithme génétique.

Paramètres de l'algorithme :

- Taux de mutation : 30%
- Probabilité de croisement : 100%
- Taille de la population : 50
- Nombre de runs : 100
- Nombre de générations : 2000
- Nombre maximum de portes : 15
- Pourcentage d'élites dans la population : 2 %
- Sélection de 50% des individus

Portes logiques utilisées :

- OR
- AND
- MUX
- XOR
- NOT

Opérateurs de mutation :

- Mutation de la porte de sortie

- Mutation des entrées d'une porte donnée
- Mutation du type de porte

Fonction de fitness :

- Evaluation selon la fonction de fitness du point 3.8
- Selon la formule choisie pour le calcul du fitness, un circuit est à 100% opérationnel lorsque son fitness est d'au moins 16. Cette valeur est ensuite augmentée par le nombre de portes non utilisées.

Stratégie de sélection : Sélection par tournoi

Données à récolter :

- Description statistique de la distribution du fitness du meilleur individu au travers des runs par génération.
- Evolution du fitness sur le meilleur des runs
- Meilleures solutions trouvées

5.3.3 Résultats

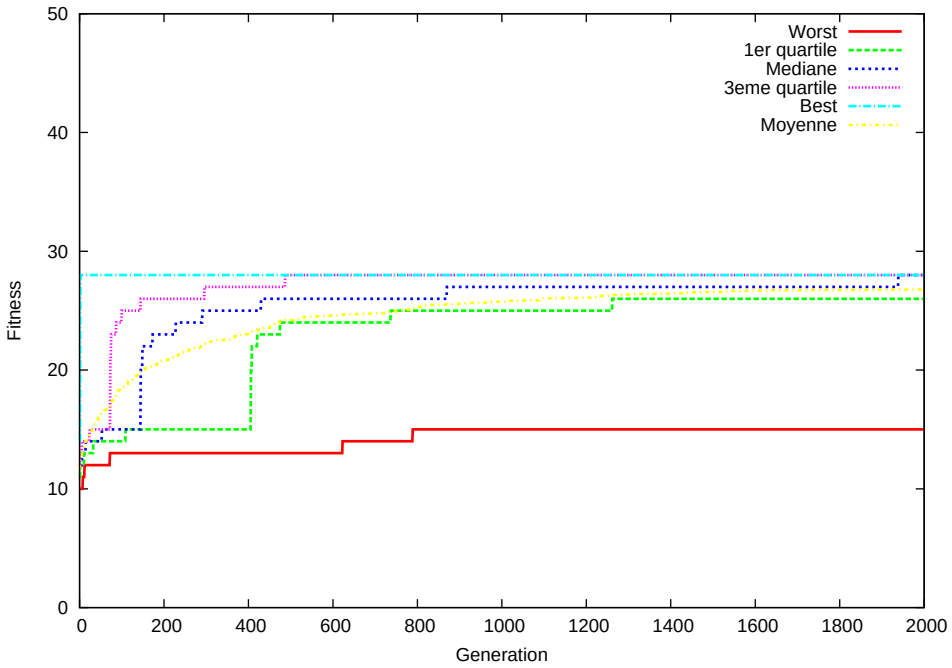

FIG. 5.1 – Description statistique de la distribution du fitness du meilleur individu au travers des runs par génération.

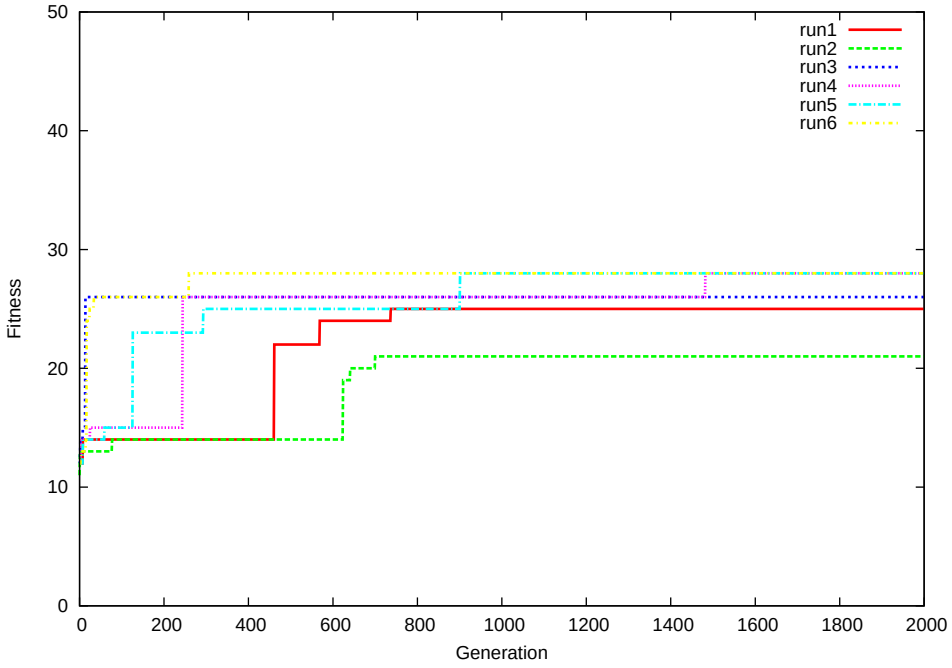

FIG. 5.2 – Evolution du meilleur fitness à travers les générations pour un échantillon de 6 runs.

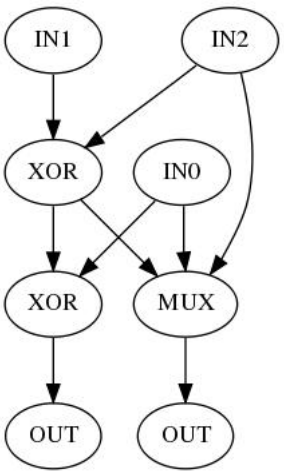

FIG. 5.3 – Full-adder 1 bit obtenu par évolution à l'aide du framework

5.3.4 Description

Nous observons sur la figure 5.1 que le quart des 30 runs effectués ont pu aboutir sur un optimum en un peu plus de 400 générations.

Ce qui, par estimation, équivaut à 400 * 50 évaluations. Au delà de 1200 générations, une solution optimale est découverte par la plupart des runs. Dans le cas du meilleur des runs, une solution qui semble être optimale est trouvée dés la première génération. Néanmoins, la plupart des runs obtiennent une solution fonctionnelle dont le fitness est supérieur ou égal à 16.

La figure 5.2 nous montre l'évolution du meilleur fitness en fonction des générations pour plusieurs runs. Nous remarquons, effectivement, une augmentation de la qualité de la solution. En ce qui concerne le circuit obtenu, nous pouvons remarquer qu'il diffère de l'additionneur traditionnel.

Il est composé de trois portes logiques.

5.3.5 Discussion

Comme la fonction de fitness calcule le nombre de bits de sorties correctes, un circuit fonctionnel possède un fitness de 2*2, soit de 16, lorsqu'il s'agit de l'additionneur 1 bit à 3 entrées et 2 sorties.

Dans le cas où le circuit est fonctionnel, nous additionnons le nombre de portes non utilisées. Dans ce cas-ci, comme le nombre de portes maximum est de 15, le fitness maximum que l'on peut obtenir est de 16 + 15, soit 31.

Nous avons fait remarquer au point précédent que le circuit obtenu est inhabituel à cause de la présence d'un multiplexeur. Comme nous pouvons le voir sur la figure 5.3, le circuit ne comporte plus que trois portes tandis que le circuit conventionnel, bien connu des électroniciens, en compte 5 (cf. chapitre 2).

Il est clair que ce résultat n'est pas intuitif pour un concepteur humain en raison de la présence du MUX et qu'aucune des techniques conventionnelles n'aurait pu le mettre en évidence.

Nous retrouvons en fait le même design que les travaux [23][8][24].

L'expérience s'est avérée très concluante en ce sens que, d'une part, l'algorithme génétique converge dans la plupart des exécutions et, d'autre part, que le circuit obtenu est le même que celui retrouvé par nos prédécesseurs.

5.4 Expérience 2

5.4.1 But de l'expérience :

Le but de cette expérience est de valider l'utilisation des algorithmes génétiques pour le design de circuits logiques. Nous avons repris pour cible l'additionneur 1 bit.

L'expérience s'est déroulée en trois étapes :

- L'évolution du circuit à l'aide d'un algorithme génétique
- Une recherche aléatoire
- Un algorithme génétique sans étape de croisement et sans élite

Pour chacun des essais, nous avons utilisé la même population de départ.

5.4.2 Paramètres

Sujet : Etude de l'intérêt des algorithmes génétiques.

Etape 1

Type d'algorithme :

- Algorithme génétique

Paramètres de l'algorithme :

- Taux de mutation : 30%
- Probabilité de croisement : 100%
- Taille de la population : 50
- Nombre de runs : 30
- Nombre de générations : 2000
- Nombre maximum de portes : 15
- Pourcentage d'élites dans la population : 2 %
- Pression sélective : 50 %

Portes logiques utilisées :

- OR
- AND

- MUX
- XOR

Opérateurs de mutation :

- Mutation de la porte de sortie
- Mutation des entrées d'une porte donnée
- Mutation du type de porte

Fonction de fitness :

- Evaluation selon la fonction de fitness du point 3.8

Stratégie de sélection : Sélection par tournoi

Données à récolter :

- Description statistique de la distribution du fitness du meilleur individu au travers des runs par génération.

Etape 2

Type d'algorithme Algorithme génétique sans croisement et avec mutation à 100% et élitisme.

Etape 3

Type d'algorithme Recherche aléatoire

Etape 4

Type d'algorithme Algorithme génétique sans croisement et avec mutation à 100% et sans élitisme

5.4.3 Résultats

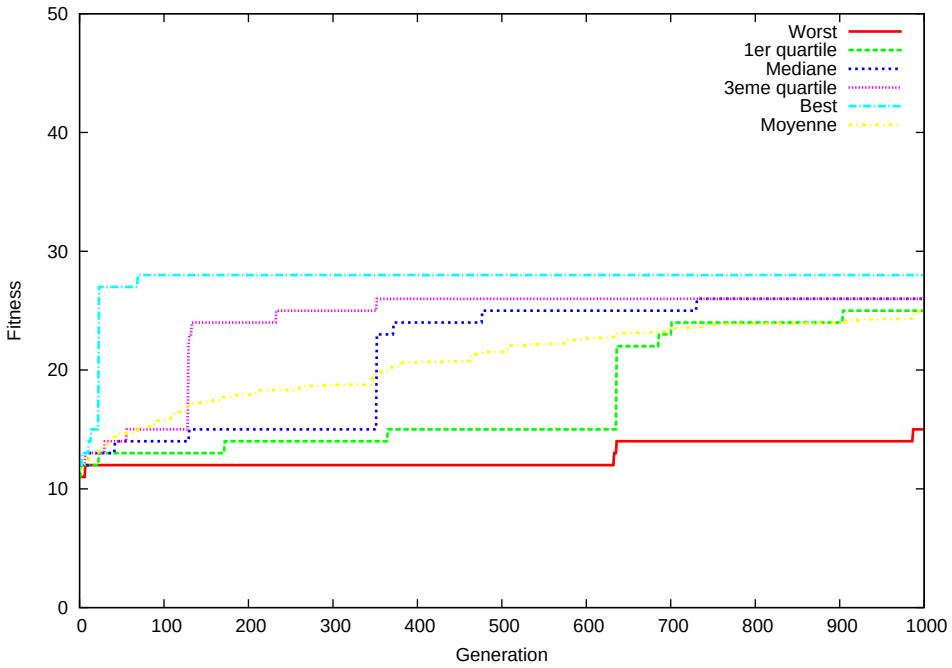

FIG. 5.4 – Algorithme génétique complet

Description L'algorithme a rapidement obtenu une très bonne solution. Néanmoins, comme l'évolution du 3ème quartile l'indique, la plupart des runs n'ont pas pu trouver la meilleure des solutions.

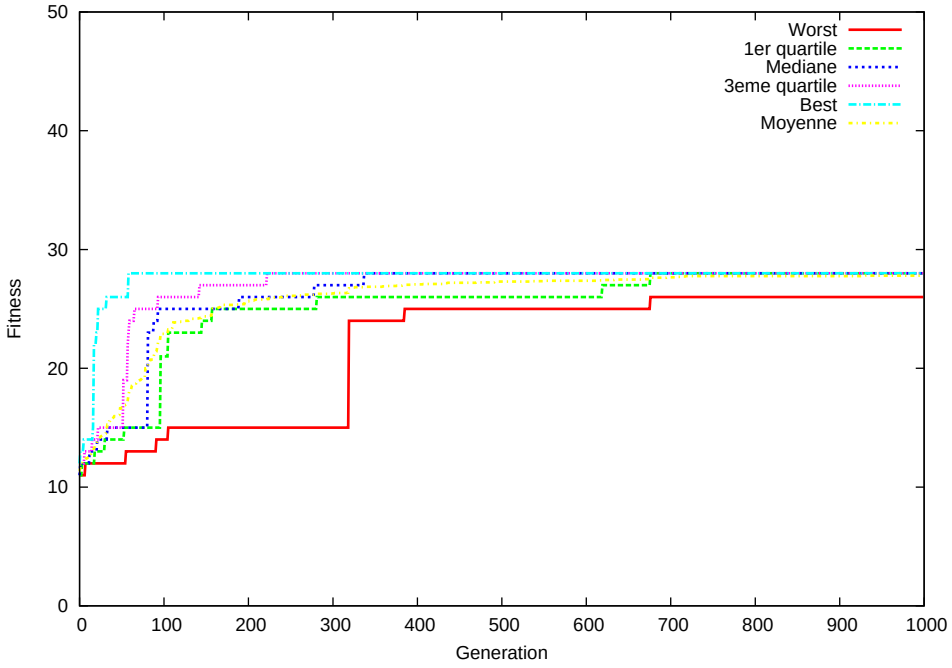

FIG. 5.5 – Algorithme génétique sans croisement avec mutation à 100%

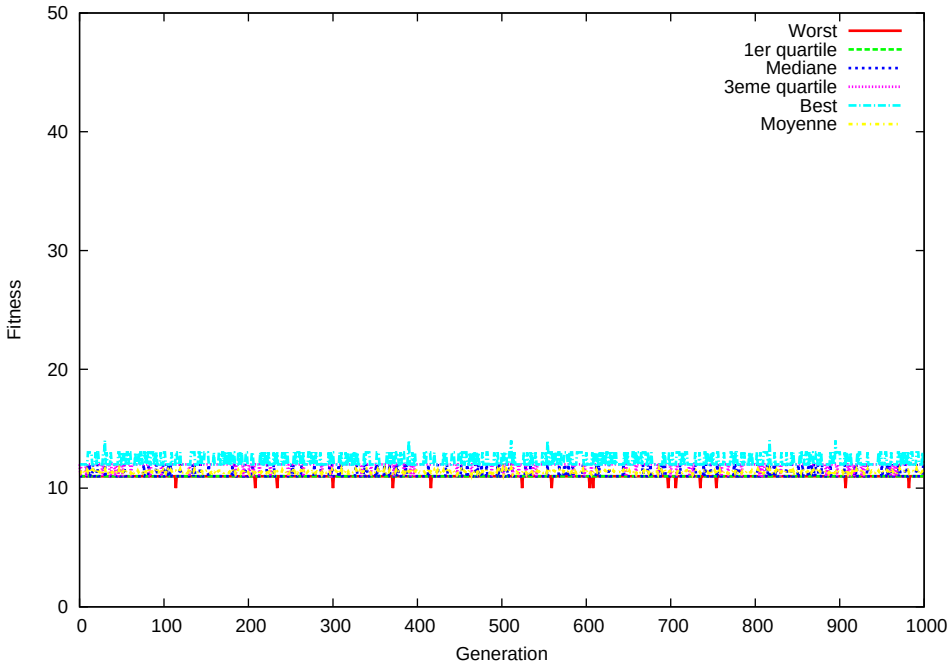

FIG. 5.6 – Recherche aléatoire

Description Sur la figure 5.6, nous n'observons aucune amélioration du fitness tandis qu'aucune solution fonctionnelle n'est générée.

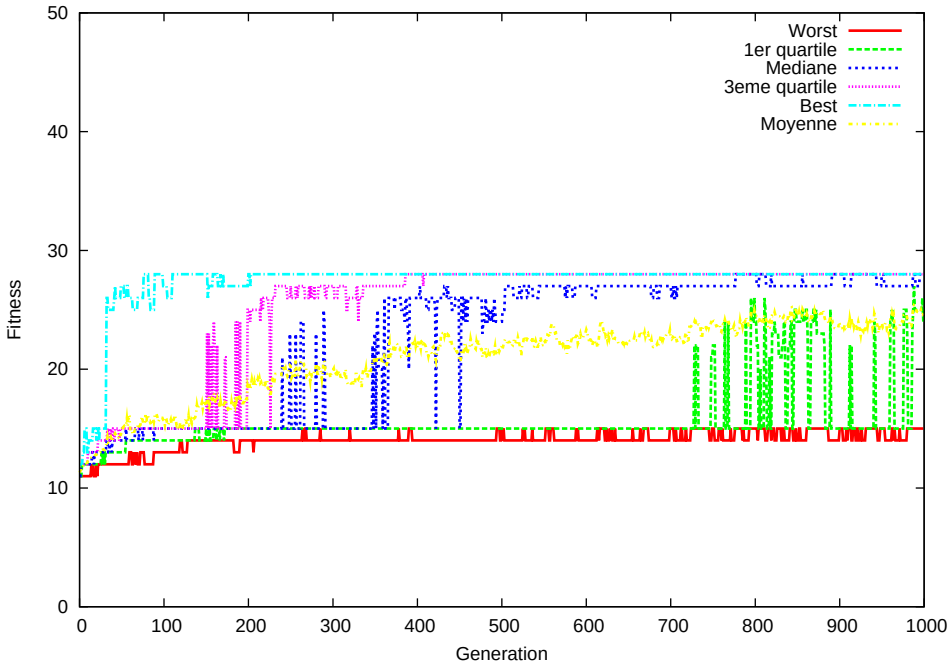

FIG. 5.7 – Algorithme génétique sans croisement et sans élitisme et mutation à 100%

Description La courbe d'évolution du fitness du meilleur individu est très perturbée mais semble se stabiliser dans certains cas.

5.4.4 Discussion

Le premier point à aborder est le fait que l'algorithme génétique complet semble être moins performant qu'un algorithme sans croisement avec seulement un taux de mutation de 100%.

Le nombre de runs obtenant une bonne solution est plus grand dans le deuxième cas.

Une des explications que nous proposons est que l'opérateur de croisement ne génère pas de bonnes solutions.

Mais, pour être plus rigoureux, il est nécessaire de définir une métrique entre

les solutions et d'étudier l'influence de différents opérateurs de croisement sur l'évolution du meilleur fitness.

De plus, il est clair que la recherche aléatoire est beaucoup moins performante que les techniques évolutionnistes car nous n'avons même pas pu trouver un circuit opérationnel sur une même fenêtre de génération et de runs donnés.

5.5 Expérience 3

5.5.1 But de l'expérience :

Comme indiqué dans le chapitre II, le parity check circuit est soumis à un brevet. Nous avons donc décidé de faire évoluer un tel circuit afin de tester ce que les techniques évolutionnistes peuvent nous fournir.

5.5.2 Paramètres

Sujet : Evolution du parity check circuit.

Type d'algorithme : Algorithme génétique.

Paramètres de l'algorithme :

- Taux de mutation : 30%
- Probabilité de croisement : 100%
- Taille de la population : 50
- Nombre de runs : 100
- Nombre de générations : 2000
- Nombre maximum de portes : 15
- Pourcentage d'élites dans la population : 2 %
- Nombre d'individus sélectionnés : 50 %

Portes logiques utilisées :

- OR
- AND
- MUX

- XOR
- NOT

Opérateurs de mutation :

- Mutation de la porte de sortie
- Mutation des entrées d'une porte donnée
- Mutation du type de porte

Fonction de fitness :

- Evaluation selon la fonction de fitness du point 3.8

Stratégie de sélection : Sélection par tournoi

Données à récolter :

- Description statistique de la distribution du fitness du meilleur individu au travers des runs par génération.
- Evolution du fitness sur le meilleur des runs
- Meilleures solutions trouvées

5.5.3 Résultats

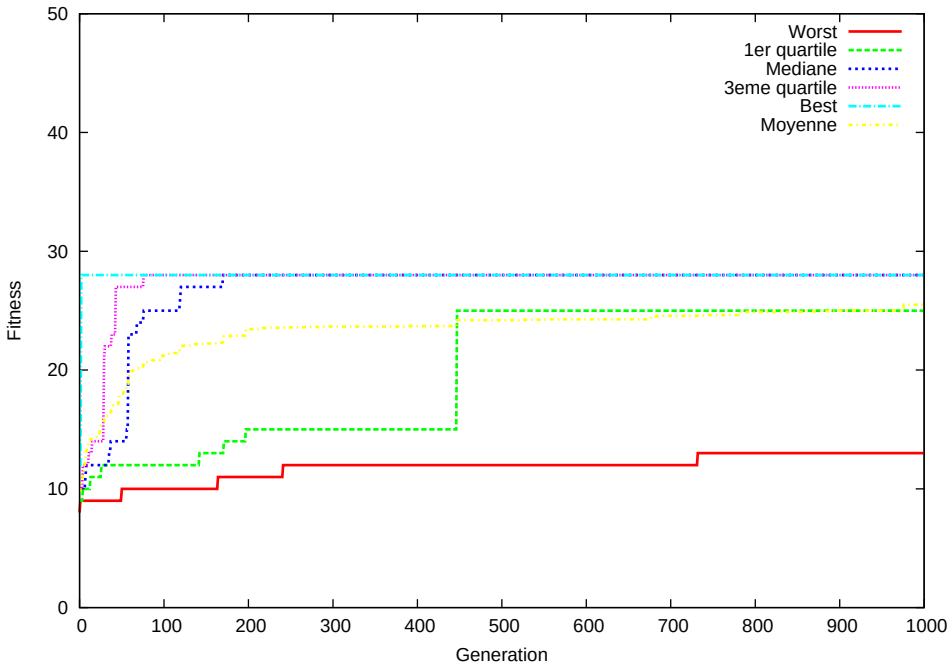

FIG. 5.8 – Description statistique de la distribution du fitness du meilleur individu au travers des runs par génération.

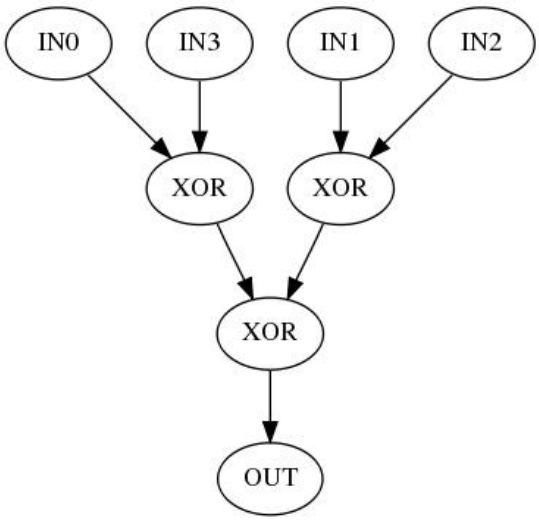

FIG. 5.9 – Parity check circuit obtenu par évolution

5.5.4 Description

La figure 5.8 nous indique que l'algorithme se révèle efficace aussi bien en termes de convergence que de solution trouvée.

La courbe du 4ème quartile montre que plus d'un quart des runs a convergé vers une bonne solution en moins de 100 générations et que la moitié se sont avérés concluants.

5.5.5 Discussion

La figure ci-dessus représente l'un des circuit obtenu. Nous pouvons remarquer qu'il s'agit en fait du circuit qui est breveté 2.5.

De plus, nous avons pu retrouver des circuits fonctionnels, voire optimisés, pour la plus part des runs.

5.6 Expérience 4

5.6.1 But de l'expérience :

Cette expérience a pour but de trouver un circuit pour réaliser le test de parité sur 3 bits et de comparer ce que nous obtenons en terme de nombre de portes par rapport aux travaux de Coello principalement. En effet, Coello a testé sa technique de multioptimisation que nous avons vue au chapitre 3, sur le testeur de parité à 3 bits. Il a comparé le circuit qu'il a obtenu avec un design obtenu par Karnaugh et un design obtenu par Quine McCluskey.

Nous allons donc présenter les résultats obtenus ainsi que quelques statistiques intéressantes.

5.6.2 Paramètres

Sujet : Evolution du testeur de parité 3 bits.

Type d'algorithme : Algorithme génétique.

Paramètres de l'algorithme :

- Taux de mutation : 30%
- Probabilité de croisement : 100%
- Taille de la population : 50
- Nombre de runs : 100
- Nombre de générations : 2000
- Nombre maximum de portes : 15
- Pourcentage d'élites dans la population : 2 %
- Sélection de 50% des individus

Portes logiques utilisées :

- OR
- AND
- XOR
- NOT

Opérateurs de mutation :

- Mutation de la porte de sortie
- Mutation des entrées d'une porte donnée
- Mutation du type de porte

Fonction de fitness :

- Evaluation selon la fonction de fitness du point 3.8.
- Dans ce cas-ci, un circuit est à 100% opérationnel lorsque son fitness est d'au moins 8. Cette valeur est ensuite augmentée par le nombre de portes non utilisées.

Stratégie de sélection : Sélection par tournoi

Données à récolter :

- Description statistique de la distribution du fitness du meilleur individu au travers des runs par génération.
- Meilleures solutions trouvées
- Nombre de fois que la meilleure solution a été trouvée

5.6.3 Résultats

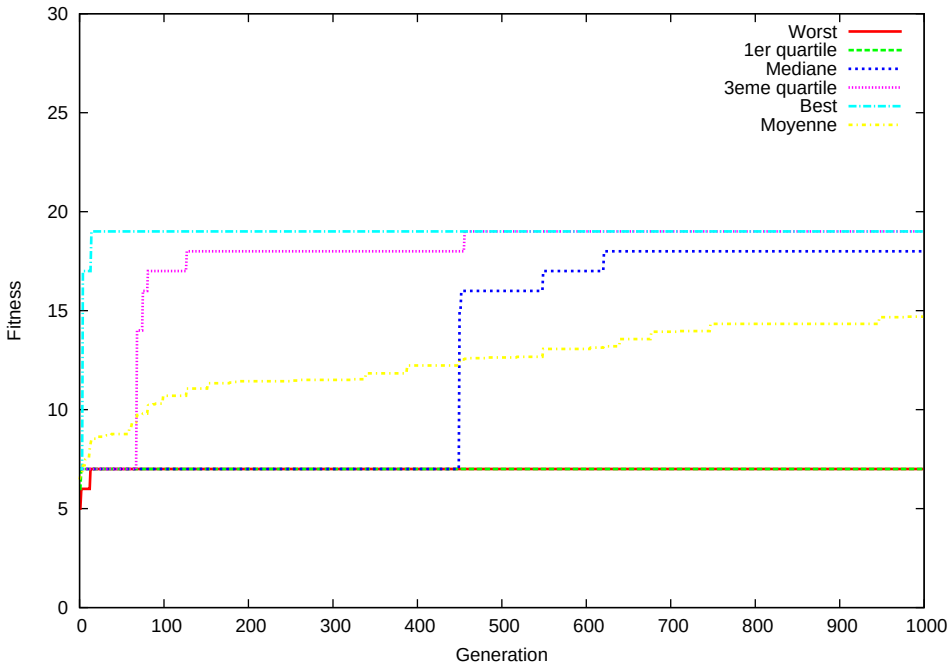

FIG. 5.10 – Description statistique

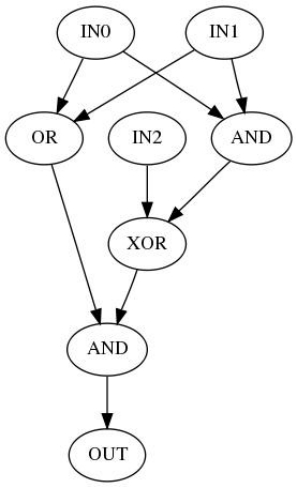

FIG. 5.11 – 3 bits parity check circuit obtenu par évolution

5.6.4 Description

Nous pouvons observer sur la figure 5.10 que toutes les courbes de statistiques se dépassent rapidement la valeur 8 qui est le fitness pour un circuit opérationnel.

La valeur maximale atteinte par la fonction de fitness dans ce cas est de 19 ce qui correspond à un circuit ne contenant que 15-(19-8) cf.3.8 , soit 4, portes.

Ce circuit est celui de la figure 5.11.

L'algorithme l'a découvert après seulement 3 générations dans le meilleur des runs tandis qu'un quart des runs l'obtient en un peu plus de 400 générations.

Au pire des cas, tous les runs trouvent un circuit fonctionnel mais dont le nombre de portes vaut 15 qui est le nombre de portes maximum permis.

5.6.5 Discussion

Le circuit obtenu est le même que celui que Coello a trouvé [11]. Il est nécessaire de remarquer que les deux dernières portes chez Coello sont inversées, nous avons recalculé la table de vérité à partir du circuit et il s'est avéré qu'il s'agit d'une erreur probable et que le circuit aurait dû être identique à celui que nous obtenons avec notre framework 5.11.

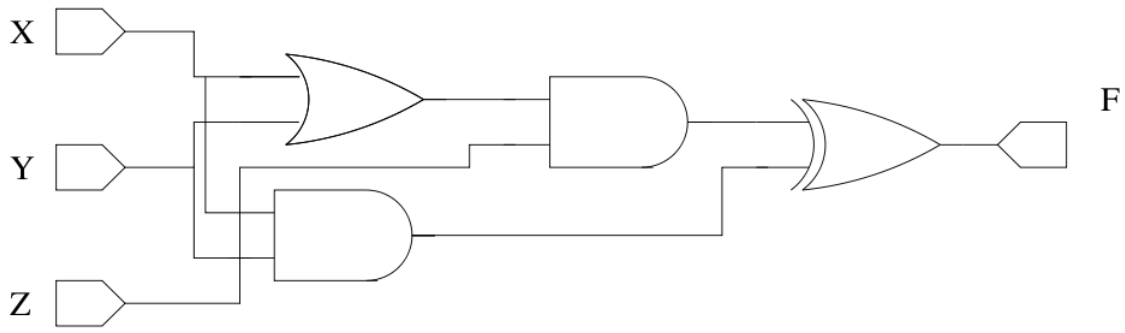

FIG. 5.12 – 3 bit parity check trouvé par Coello. [11]

Nous nous basons sur les résultats obtenus dans [11] pour effectuer une comparaison. Sur la figure 5.13, nous retrouvons quatre designs obtenus par deux techniques conventionnelles et deux techniques évolutionnistes. HD1 et HD2 sont des designs obtenus par respectivement Karnaugh (cf.2.4.5) et Quine Mc Cluskey

NGA et MGA sont deux algorithmes génétiques que Coello à imaginé. Le premier est un simple algorithme génétique utilisant un alphabet de taille N. Pour le second, nous renvoyons le lecteur vers [11] et 3.10.

Nous observons sur la figure 5.13 que les circuits obtenus avec des techniques de design évolutionniste fournissent des circuits avec moins de portes que les autres.

MGA	**NGA**	**HD 1**	**HD 2**
$F = (X + Y)Z \oplus (XY)$	$F = Z(X + Y) \oplus (XY)$	$F = Z(X \oplus Y) + Y(X \oplus Z)$	$F = X'YZ + X(Y \oplus Z)$
4 gates	4 gates	5 gates	6 gates
2 ANDs, 1 OR, 1 XOR, 1 NOT	2 ANDs, 1 OR, 1 XOR	2 ANDs, 1 OR, 2 XORs	3 ANDs, 1 OR, 1 XOR, 1 NOT

FIG. 5.13 – Comparaison de la meilleure solution obtenue par deux techniques évolutionnistes, MGA et NGA, et les deux techniques conventionnelles, Karnaugh (HD1) et Quine Mc Cluskey (HD2) [11].

5.7 Conclusion

Nous avons pu réaliser et présenter dans ce document quatre expériences d'évolution de design de circuits logiques.

Il est clair que nous avons choisi des circuits simples. La raison est qu'en simulant sur des petits circuits, nous pouvons très vite tester différentes stratégies qui par extension pourraient être appliquées à de plus grands circuits.

Comme nous le soulignons dans certaines des discussions précédentes, de nombreuses expériences sont encore à réaliser.

Voici donc une liste d'expériences que nous avons imaginées mais qui nécessite, dans certains cas, de modifier le framework :

- Comparaison des différents opérateurs de croisement
- Comparaison des résultats avec d'autres modélisations notamment le modèle en arbre.
- Etude d'autres fonctions de fitness
- Evaluation des performances du framework

Par ailleurs, les expériences que nous avons présentées montrent que le framework est tout à fait capable de retrouver les même résultats que la littérature et même retrouver un circuit qui a été breveté.

D'une part, cela confirme le fonctionnement du framework et, d'autre part, l'intérêt de l'utilisation des algorithmes évolutionnistes appliqués aux circuits logiques. Il est clair que vu le nombre d'expériences réalisées, nous ne pouvons conclure que le framework peut à l'heure actuelle permettre de réaliser une découverte dans le domaine. Pour cela, il serait nécessaire de continuer son développement et de réaliser d'autres expériences qui pourront démontrer son efficacité en terme de performances ainsi qu'en modularité.

Conclusion

Dans le cadre de ce mémoire, nous avons décidé, dans un premier temps, d'étudier les différentes techniques évolutionnistes existantes. Après avoir acquis de nombreuses connaissances dans ce domaine, nous nous sommes attaqués aux circuits logiques. Pour le développement du framework logiciel, nous nous sommes basés sur la plupart des travaux concernant le design évolutionniste afin d'en extraire les principes.

Ainsi, nous avons débuté par l'implémentation d'une première version sous la plate-forme .NET. Mais, celle-ci fût rapidement abandonnée en raison d'un besoin grandissant de performances. Dès lors, le langage C++ nous sembla être un choix judicieux en terme de performances car, d'une part, il offre un code compilé et optimisé pour une machine cible et, d'autre part, car il permet de gérer la mémoire à sa guise. Cet avantage s'est révélé être un couteau à double tranchant. En effet, les algorithmes évolutionnistes manipulent énormément de données sur de longues périodes de calcul, il s'avère strictement nécessaire de libérer toutes les ressources mémoire non utilisées. Chose qui a été oubliée au départ. Le code du framework a, donc, subi une importante phase de refactoring qui consistait à améliorer sa modularité ainsi qu'à minimiser sa consommation en terme de ressources mémoire. Afin d'y parvenir, nous avons fait appel à des outils d'analyse de l'utilisation de la mémoire tels que Valgrind.

Par ailleurs, de nombreux design patterns ont été implémentés lors la réalisation du framework. Citons, entre autres, l'application du pattern Chain-of-responsibility pour la gestion des algorithmes évolutionnistes. La structure de ces derniers est modélisée sous la forme d'une liste liée d'objets appelés Stage. Chaque objet Stage transmet à l'objet suivant la population courante afin qu'elle puisse subir les traitements adéquats. Cette structure permet de changer statiquement ou dynamiquement le flot d'exécution de l'algorithme. Les différents opérateurs répondent à un pattern visiteur afin d'offrir une souplesse en vue la création éventuelle de nouveaux traitements.

Afin de tester le framework, nous avons réalisé quelques expériences dont les buts étaient, d'une part, de valider le fonctionnement du framework et, d'autre part, de tester différents algorithmes évolutionnistes sur divers circuits. L'outil que nous avons développé s'est avéré très flexible pour la réalisation des différentes

expériences et nous avons pu aboutir sur des designs identiques à ceux obtenus durant des travaux passés. Remarquons, en outre, qu'un simulateur de circuits logiques a été réalisé afin d'évaluer la fonctionnalité des circuits obtenus.

Néanmoins, il est évident que le développement d'un tel framework n'est pas fini, nous avons donc proposé une série d'améliorations possibles à la fin du quatrième chapitre.

Ce travail a été pour moi un premier contact avec le monde de la recherche alors que j'étais habitué à des réalisations dont le déroulement et le but étaient clairement définis par un cahier de charges. Ici, j'ai dû faire preuve de rigueur et d'imagination pour toutes les démarches que j'ai entreprises. De plus, cette nouvelle approche m'a permis de marier deux de mes domaines d'intérêt : l'informatique et l'électronique.

Enfin, pour conclure ces longs mois de recherche, je terminerai sur une note optimiste : selon moi, les algorithmes évolutionnistes appliqués au design de circuits logiques constituent un domaine très prometteur malgré les innombrables essais et erreurs. Non seulement les ordinateurs gagnent de jour en jour en puissance et en performances et balayeront, ainsi, la contrainte concernant le besoin de performances qui est inhérent à ce domaine ; mais j'ai pu, personnellement, reproduire à petite échelle et sans trop de difficultés certains résultats déjà obtenus par d'autres chercheurs, ce qui constitue, à mon sens, un espoir pour leur avenir.

Bibliographie

[1] Galib. a c++ library of genetic algorithm components.

[2] Gnuplot.

[3] Graphviz.

[4] Nasa/esa conference on adaptive hardware and systems, 2007.

[5] Jean-Claude Bajard. Circuits logiques. Cours donné IUT de Montpellier.

[6] T. Beielstein, J. Dienstuhl, C. Feist, and M. Pompl. Circuit design using evolutionary algorithms, 2002.

[7] Hounsell B.I. and Arslan T. A novel genetic algorithm for the automated design of performancedriven digital circuits. In *Evolutionary Computation*, volume 1, pages 601–608, 2000.

[8] Peter Bungert, Sanaz Mostaghim, Harmut Schmeck, and Jurgen Branke. *Architecture of Computing Systems ARCS 2008*, chapter Design of Gate Array Circuits Using Evolutionary Algorithms, pages 38–50. Lecture Notes in Computer Science. Springer Berlin, 2008.

[9] Christian Casteyde. Cours de c/c++, 2002.

[10] Maurice Clerc and Patrick Siarry. Une nouvelle métaheuristique pour l'optimisation difficile : la méthode des essaims particulaires. *J3eA*, 3, 2004.

[11] Carlos A. Coello Coello and Arturo Hernandez Aguirre. Design of combinational logic circuits through an evolutionary multiobjective optimization approach. *Artif. Intell. Eng. Des. Anal. Manuf.*, 16(1) :39–53, 2002.

[12] Charles Darwin. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life (L'Origine des espèces)*. John Murray, Londres, 1872.

[13] DOLPHIN.

[14] Economist.com. Don't invent, evolve, 2007.

[15] E.Freeman, K.Sierra, and B.Bates. *Design Patterns*. Tete la première. O'Reilly, 2005.

[16] Emanuel Falkenauer. *Genetic Algorithms and Grouping Problems*. John Wiley Sons, 1997.

[17] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[18] Timothy G. W. Gordon and Peter J. Bentley. On evolvable hardware.

[19] Peter J.Ashenden. *The designer's Guide to VHDL*. Morgan Kaufmann, 1995.

[20] Eric Fimbel Bruno De Kelper. Classification d'algorithmes et heuristiques, 11. Université du Quebec Cours 9.

[21] Sushil J. Louis. *Genetic Algorithms as a Computational Tool for Design*. PhD thesis, Department of Computer Science Indiana University, 1993.

[22] Richar L.Rudell. *MULTIPLE-VALUED LOGIC MINIMIZATION FOR PLA SYNTHESIS*. PhD thesis, University of California, 1986.

[23] J.F. Miller, T. Kalganova, N. Lipnitskaya, and D. Job. The genetic algorithm as a *Discovery Engine* : Strange circuits and new principles. In *Proc. of the AISB Symposium on Creative Evolutionary Systems (CES'99)*, pages 65–74, Edingurgh, UK, 1999. The Society for the Study of Artificial Intelligence and Simulation of Behaviour (AISB).

[24] Julian F. Miller, Dominic Job, and Vesselin K. Vassilev. Principles in the evolutionary design of digital circuits - part i. *Genetic Programming and Evolvable Machines*, 1(1-2) :7–35, 2000.

[25] Julian F. Miller, Dominic Job, and Vesselin K. Vassilev. Principles in the evolutionary design of digital circuits - part II. *Genetic Programming and Evolvable Machines*, 1(3) :259–288, 2000.

[26] Dragomir Milojevic. Circuits logiques et numériques, 2007. Cours donné au BA3 de la faculté de polytechnique. ULB.

[27] Eshel Ben-Jacob Nadav Raichman, Ronen Segev. Evolvable hardware : genetic search in a physical realm. *Elsevier*, 2002.

[28] Nadia Nedjah and Luiza de Macedo Mourelle. Evolutionary digital circuit design using genetic programming. In Nadia Nedjah, Ajith Abraham, and Luiza de Macedo Mourelle, editors, *Genetic Systems Programming : Theory and Experiences*, volume 13 of *Studies in Computational Intelligence*, pages 149–174. Springer, Germany, 2006. Forthcoming.

[29] Nketsa. *Circuits logiques programmables : Mémoires, PLD, CPLD et FPGA*. Ellipses Marketing, 1998.

[30] OWADA. Us patent 6027243 - parity check circuit, 2000.

[31] Marc Pirlot. Introduction aux métaheuristiques. Faculté de Polytechnique de Mons.

[32] William Press, Saul Teukolsky, William Vetterling, and Brian Flannery. *Numerical Recipes in C*. Cambridge University Press, Cambridge, UK, 2nd edition, 1992.

[33] John R.Koza. Genetic programming : A paradigm for genetically breeding populations of computer programs to solve problems. Technical report, Stanford University Computer Science Department, 1990.

[34] John R.Koza. *Genetic Programming : On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[35] Peter Seibel. *Practical Common Lisp*. Apress, 2005.

[36] Patrick Siarry. *Métaheuristiques pour l'optimisation difficile*. Eyrolles, 2003.

[37] Geneura Team. Eo evolutionary computation framework.

[38] The GSL Team. Manuel en ligne de la gsl.

[39] Floyd Thomas-L. *Fondements d'électronique : Circuits c.c. Circuits c.a. Composants et applications*. Reynald Goulet, 2006.

[40] Floyd Thomas-L. *Systèmes numériques*. Reynald Goulet, 2006.

[41] Thomas Weise. *Global Optimization Algorithms Theory and Application*. 2008.