Titre: Title:	Validation de circuits numériques utilisant le principe du test par mutation
Auteur: Author:	Patrice Vado
Date:	1999
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Vado, P. (1999). Validation de circuits numériques utilisant le principe du test par mutation [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/8635/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/8635/
Directeurs de recherche: Advisors:	Yvon Savaria
Programme: Program:	Non spécifié

UNIVERSITÉ DE MONTREAL

VALIDATION DE CIRCUITS NUMÉRIQUES UTILISANT LE PRINCIPE DU TEST PAR MUTATION

PATRICE VADO
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE ET DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
20 AOÛT 1999

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

VALIDATION DE CIRCUITS NUMÉRIQUES UTILISANT LE PRINCIPE DE TEST PAR MUTATION

présenté par: VADO PATRICE

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

M. BOIS Guy, Ph. D., président

M SAVARIA Yvon, Ph. D., membre et directeur de recherche

M. SAWAN Mohamad , Ph. D., membre

REMERCIEMENTS

Bien qu'il soit d'habitude établi de remercier en premier le directeur d'études, j'aimerais remercier mon père, celle qui est à mes cotés et mes frères pour leurs constants encouragements.

Je tiens à remercier Monsieur Savaria qui a guidé mes recherches et qui par ses nombreux conseils éclairés m'a permis d'achever mon travail. Il a été d'un grand soutien aussi bien professionnel que personnel. Je le remercie d'avoir trouvé le temps de suivre le déroulement de mes travaux et de les avoir orientés dans le bon sens.

Je tiens à remercier Monsieur Yannick Zocarrato ainsi que l'équipe VALSYS de Grenoble. Leur appui et leur compréhension tout au long de la collaboration m'a permis d'explorer de nouvelles voies.

Je remercie Monsieur Boukari Zahir avec qui j'ai travaillé sur le simulateur Pulse et qui m'a permi de découvrir un autre aspect de la validation.

Je remercie Monsieur Paul Marriot, Ivan Kraljic, Nicolas Contandriopoulos et toute l'équipe Pulse pour leur soutien constant durant mon séjour dans l'équipe.

Enfin, merci à toute l'équipe du GRM pour leur soutien...

RESUME

Les systèmes numériques augmentent continuellement en taille et en complexité. La performance des circuits intégrés double tous les deux ans. En raison d'une trop grande complexité certaines fonctionnalitées ne sont pas vérifiées, réduisant alors la confiance dans les circuits. La complexité actuelle des circuits révèle les limitations des méthodes traditionnelles de vérification par simulation. Afin de répondre à ces limitations, les méthodes formelles de vérification tentent d'utiliser la rigueur mathématique afin de prouver l'exactitude d'un circuit. Cependant, la rigueur du formalisme ainsi que le peu d'outils performants réduisent une utilisation industrielle fréquente.

Ce mémoire propose une méthodologie de validation et d'enrichissement de vecteurs de validation de circuits numériques basée sur le test par mutation consistant à insérer des fautes spécifiques dans un langage de description matériel tel que VHDL. Le programme contenant la faute spécifique est alors appelé mutant. La validation de la méthode proposée a été réalisée sur différents bancs d'essais décrits en langage VHDL et simulés à l'aide de Synopsys. Un algorithme d'enrichissement des vecteurs de validation ainsi qu'un générateur de mutants ont été écrits en langage C. L'utilisation de cette méthode a montré qu'il était nécessaire de disposer d'un espace disque important, puisque le nombre de mutants générés est grand. Un certain nombre de concepts, tel que la contrôlabilité et l'observabilité, provenant du domaine du test matériel ont été empruntés afin d'expliquer la réduction de la couverture des mutants. Par ailleurs, une métrique appelée score de mutation a été utilisée afin de quantifier la qualité d'un jeu de vecteurs de validation.

ABSTRACT

Digital systems continuously grow in scale and functionality. In addition, the performance of integrated circuits (IC) doubles every two year. Due to the growing complexity, functionalities are not fully verified which reduce the confidence in designs. That growing complexity unravels the limitations of traditional verification methods based on functionnal simulation. To address these limitations, formal methods and tools for specifying and verifying such systems have been proposed. However, the complexity of formal notations and the practical limitations of available tools reduce their use in industry.

This master proposes a methodology based on mutation testing for validating and enriching a set of functional validation vectors for digital circuits. This methodology injects specific faults in a hardware description language description such as VHDL. A program which contains a fault is called a mutant. The validation of the proposed methodology was realized on VHDL benchmarks that were simulated with Synopsys. A mutant generator and a test suite enriching algorithm were written in the C programming language. Existing mutation testing tools were shown to require a lot of memory to analyze small functional descriptions. A metric called mutation score was used to quantify the quality of a vector set. Justification and observability concepts were applied to mutation testing to explain the reduction of the mutation score.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

INTRODUCTION

La vérification par la simulation est la méthode la plus largement répandue afin de vérifier qu'un design respecte bien les spécifications voulues. Elle fait partie intégrante de la méthodologie de conception VLSI selon la figure 0.1. La validation devient de nos jours un des goulots d'étranglement du processus de conception qui ralentit la mise en marché de puces électroniques dans un monde extrèmement compétitif et en perpétuel changement. D'autre part, certains composants défectueux ne sont découverts que chez le client, ce qui augmente considérablement les coûts de développement et entache la crédibilité des fournisseurs à qui cela arrive. En tout état de cause, tous s'accordent à dire que le processus de validation est primordial.

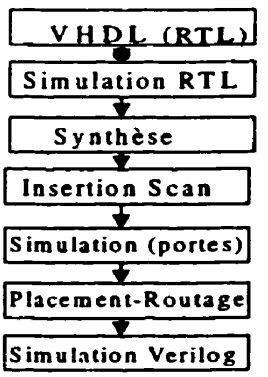


Figure 0.1: Méthodologie de conception.

Cependant, en raison de la complexité sans cesse croissante des circuits numériques, le processus de validation devient de plus en plus long et coûteux. Dans le processus traditionnel de test par simulation, le concepteur cré un jeu de test complet représentant

toutes les entrées possibles du circuit et en compare les résultats avec ceux prédits. Selon la figure 0.2, l'environnement de test est inscrit dans ce que l'on appelle un banc d'essai ou un *test bench* dans la terminologie propre au VHDL.

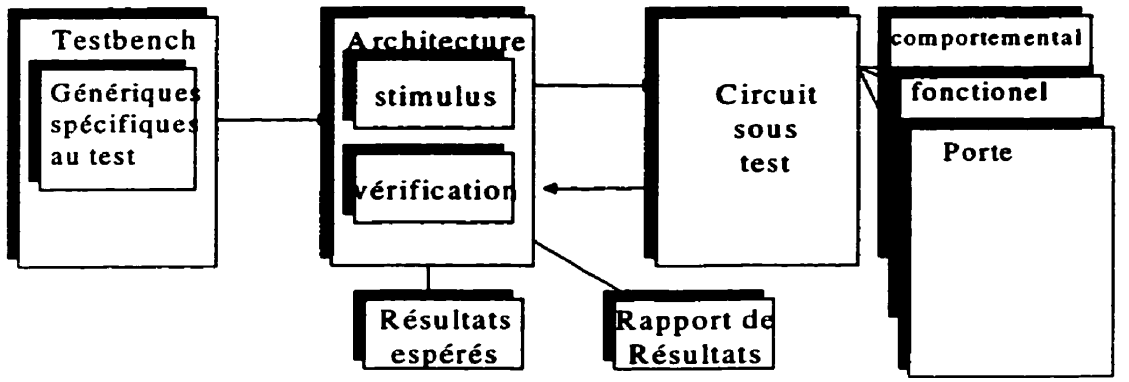


Figure 0.2: Processus de validation par simulation.

Dans le cadre de la validation de circuits industriels de plus en plus complexes, des variantes basées sur la définition de modèles de référence sont couramment préconisées. Une de ces variantes a été largement utilisée dans le cadre du projet PULSE[1] développé à l'École polytechnique. Afin de valider une puce conçue selon une architecture SIMD(single instruction multiple data), un simulateur décrivant l'architecture fonctionnelle du circuit a été développé. PULSE, dont l'architecture est présentée à la figure 0.3, est optimisé pour le traitement d'images en temps réel. Cette puce, dont la conception a été réalisée en langage VHDL, contient environ un million de transistors et a été validée à l'aide d'un simulateur écrit en langage C [2].

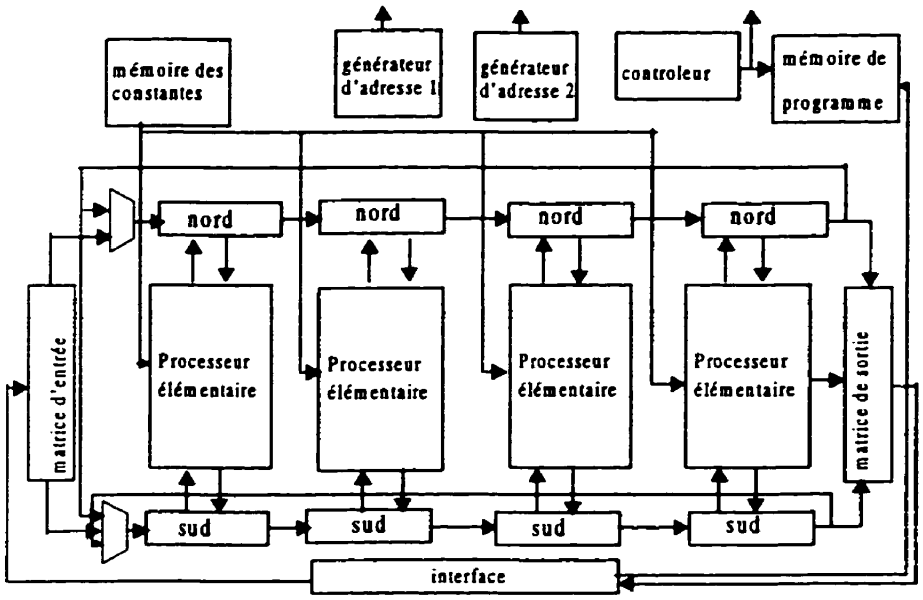


Figure 0.3: Architecture simplifiée de Pulse.

Dans le projet PULSE, le modèle C est une description fidèle de l'architecture qui représente la fonctionnalité de la puce, ainsi que les unités opératives tel que l'ALU, le décaleur et le multiplieur additionneur. Le contrôleur avec ses différents compteurs, les registres, ainsi que les canaux de communication ont eux aussi été modélisés. Un des avantages d'une méthodogie utilisant un modèle de référence, c'est que ce modèle est sensiblement moins complexe que le design en VHDL. Nous avons, dans le cas du modèle C de Pulse, modélisé le jeu d'instruction de la puce. La méthodologie de validation adoptée était la suivante. Partant d'une même séquence de vecteurs de validation, le résultat de l'exécution du programme sur le modèle VHDL est comparé avec celui du simulateur C. L'avantage de ce genre de méthode est que la description du simulateur est indépendante de l'implantation du modèle VHDL. Elle dépend seulement d'un jeu

d'instructions défini lors de la spécification. La méthodologie de validation est schématisée à la figure 0.4.

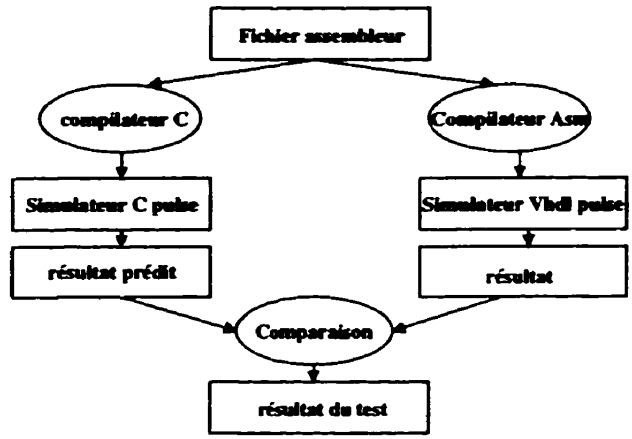


Figure 0.4: Validation Fonctionnelle.

Un des gros problèmes de la méthode par validation est que pour des circuits ayant un grand nombre d'entrées, il est impossible de tester toutes les possibilitées. En effet, pour être complète, la simulation doit étudier le comportement du circuit dans toutes les configurations possibles et pour toutes les séquences.

Par opposition à la simulation, la vérification formelle tente de prouver qu'une description structurelle possède un comportement équivalent à une description fonctionnelle à un niveau d'abstraction supérieur. Il s'agit donc de comparer des descriptions abstraites. Bien que cette méthode fasse de plus en plus d'émules, il ne faut pas oublier les problèmes qui en ralentissent une utilisation industrielle fréquente. Les principaux problèmes[3] sont au nombre de trois: d'une part l'extraction d'une description

du comportement, d'autre part la formulation d'une spécification de haut niveau et finalement, la comparaison entre la description extraite et la spécification.

Le premier problème nécessite la définition d'un modèle temporel et d'une description. Un circuit peut être décrit à différents niveaux d'abstraction, à savoir: électrique, commutateur, porte logique, registre... Le modèle temporel est continu au niveau électrique et, discret aux niveaux supérieurs. Dans ce dernier cas, un modèle de délai doit être défini: instantané pour l'étude des circuits combinatoires, unitaire correspondant à une période d'horloge pour les circuits synchrones.

Le second problème nécessite d'exprimer le comportement souhaité. La spécification correspond à la fois au but à atteindre et à la description abstraite du comportement en termes d'entrées-sorties. Cette spécification est alors exprimée sous la forme d'un automate de haut-niveau. Une spécification complète est parfois difficile à obtenir, en conséquence, seules certaines propriétées sont vérifiées. Citons par exemple la preuve qu'un système ne peut se bloquer, ou que certaines règles ou protocoles d'échange soient respectés. Cependant, le problème de complétude de la validation, ou le fait que les propriétés définies remplissent toutes les volontées du concepteur restent des questions ouvertes.

Bien que de nombreuses études et avancements aient été réalisées dans le domaine de la vérification formelle, l'utilisation de ces techniques lors de la validation amène un trop lourd formalisme mathématique pour bien des concepteurs. Par contre, la validation par la simulation traditionnelle ne permet souvent pas de valider pleinement de grands circuits.

Aussi diverses méthodes ont été proposées afin d'enrichir un jeu de vecteurs de validation amenant une meilleure confiance dans les circuits VLSI. Le test par mutation est une des techniques dérivées de la validation par simulation. Cette méthode permet d'enrichir la suite de vecteurs de validation. Il est à noter que d'autres techniques ont été proposées afin d'augmenter la confiance dans un design. Une manière valable de s'attaquer aujourd'hui aux défis inhérents à la vérification, c'est d'utiliser les analyses de couverture du code. La compagnie Summit Design inc propose un certain nombre d'outils permettant de chiffrer la couverture d'un programme HDL et la couverture des machines à états par des données de test. Ces outils permettent de montrer combien les vecteurs de validation ont pu exercer une région du circuit et en révéler les régions non testées. Par ailleurs, ils identifient les tests qui ont les plus grandes couvertures et permettent de réduire les tests redondants. Il est intéressant de remarquer que pour calculer une couverture du code, ces outils comptent combien de fois la suite de test exerce chaque ligne, chemin, expression, état et séquence. La mutation par contre ne compte pas combien de fois un vecteur a exercé une région, mais plutôt si toutes les régions ont été exercées au moins une fois.

Les différents objectifs de ce mémoire sont de trois types. Le premier consiste à étudier l'utilisation du test par mutation dans le cas de la validation de circuits numériques. Le second permet de définir un ensemble d'opérateurs de mutation permettant d'identifier d'éventuelles erreurs fonctionnelles survenant dans un langage de description matériel tel que le VHDL. Enfin, le troisième abouti à la proposition d'une nouvelle méthodologie de validation par mutation permettant l'enrichissement systématiquement d'un jeu de vecteurs de tests.

Le premier chapitre de ce mémoire consiste en une présentation des méthodes formelles ainsi que des difficultés rencontrées lors de l'utilisation de ces différentes techniques. Ce chapitre se poursuit par une revue de littérature dans le domaine du test logiciel et particulièrement sur le test par mutation.

Le second chapitre propose une adaptation du test par mutation à la validation des circuits VLSI ainsi que les résultats obtenus sur différents bancs d'essais ainsi que les problèmes inhérents au test par mutation. Aussi nous présenterons une méthodologie systématique de test par mutation dans le cas de circuits VLSI.

Enfin, le troisième chapitre présente l'algorithme de génération de mutants et d'enrichissement de vecteurs de validation, ainsi que le banc d'essai qui a permis la définition des différents opérateurs de mutation. Ce chapitre commencera donc par une redéfinition des opérateurs de mutation dans le cas de circuits VLSI et se continuera par la proposition d'un algorithme systématique de validation de circuits numériques utilisant l'outil Synopsys.

CHAPITRE 1

METHODES DE VALIDATION

1.1 Méthode Formelle

1.1.1 Introduction

Bien que la vérification par la simulation soit la méthode la plus largement utilisée dans le processus de validation de circuits VLSI, l'augmentation du niveau de complexité des circuits, ainsi que la difficulté à procéder à des tests exhaustifs, augmente le risque que certains aspects du design ne soient pas vérifiés. Une alternative basée sur des méthodes mathématiques rigoureuses a été proposée : la vérification formelle.

Cette méthode est basée sur des langages, des techniques et des outils mathématiques. La plupart de ces langages, techniques et outils, utilisent des représentations binaires plus connues sous le nom de diagrammes binaires de décision (BDD). Le BDD[4] est une simple représentation canonique de la logique binaire énumérant toutes les valeurs possibles pouvant se produire dans un circuit combinatoire. L'utilisation des méthodes formelles ne garantit pas, *a priori*, l'exactitude du design mais, elle augmente de manière significative la compréhension d'un système en y révélant les ambiguïtés et les inconsistances. Cependant, un problème majeur des méthodes formelles est l'utilisation abondante de la mémoire et du temps CPU, qui croissent parfois de manière exponentielle en fonction de la taille du circuit.

Les méthodes formelles s'appuient sur trois outils principaux afin de vérifier une spécification. On retrouve des outils de vérification d'équivalence, de vérification de modèle et des prouveurs de théorèmes. Les vérificateurs d'équivalence comparent la spécification avec un design de référence. Les vérificateurs de modèles prouvent la véracité des propriétés du design par rapport à la spécification. D'autre part, les prouveurs de théorèmes permettent à l'usager de bâtir une preuve montrant que le design respecte bien la fonction voulue.

Nous exposerons plus en détail dans la suite de ce paragraphe les vérificateurs du modèle et les prouveurs de théorèmes, deux types de vérification couramment utilisées.

Deux exigences importantes doivent être respectées dans le processus de vérification. La première est la définition d'un langage formel capable de décrire le comportement d'un système et l'expression de propositions(propriétés) bâties à partir de celui-ci. La seconde exigence est de disposer d'un calcul déductif capable de prouver toutes les propositions exprimées dans le langage. Les méthodes de vérification formelle utilisent l'approche générale suivante:

1. Écrire une spécification formelle S décrivant le comportement pour lequel le système doit être vérifié et est supposé exempt de défauts.

2. Écrire une spécification pour chaque type de primitive des composants utilisés dans la construction du système. Ces spécifications sont supposées décrire les comportements des composants réels du système.

3. Définir une expression D qui décrit le comportement du système que l'on désire prouver. La définition de D est de la forme suivante : D = P1 + ... + Pn,

où P1, ..., Pn spécifient le comportement des parties constituantes du système et le + est l'opérateur de composition qui représente l'effet de lier les composants ensemble. Les expressions P1, ..., Pn utilisées ici sont des instances des spécifications des composants primitifs définis à l'étape 2.

4. Prouver alors que le circuit est décrit par l'expression D. Ceci se fait par une preuve d'un théorème de la forme : *D satisfait S*

où *satisfait* est une relation de satisfaction des spécifications du comportement du système.

1.1.2 Vérification du modèle

La vérification du modèle est une technique qui consiste à construire un modèle fini d'un système et, à démontrer que les propriétés spécifiées sont bien respectées. En pratique, deux approches sont utilisées en vérification du modèle. La première, la vérification de modèles temporels est une technique developpée dans les années 1980 par Clarke et Emerson d'une part et par Queille et Sifakis d'autre part. Dans cette approche que nous présenterons plus loin, les spécifications sont exprimées en logique temporelle (TL) et les systèmes sont modélisés comme des systèmes à états finis qui effectuent des transitions. Dans la seconde approche, la spécification est donnée sous la forme d'une machine à états. Dans ce cas, le système est lui aussi représenté par une machine à états et, est comparé à la spécification afin de déterminer si oui ou non ses comportements sont conformes à ceux de la spécification.

Vérification de modèle temporel

Un système à états finis peut être représenté par un graphe de transition d'états étiquetés, où les étiquettes d'un état sont les valeurs des propositions atomiques de cet état (par exemple pour les valeurs des bascules). Les propriétés concernant le système sont exprimées comme des formules en logique temporelle pour lesquelles le système à transitions d'états doit être un modèle. La vérification du modèle consiste à parcourir la machine à états des transistions du système (FSM) et à vérifier s'il satisfait les formules représentant la propriété.

Arbre de calcul logique

La logique temporelle exprime l'ordre des événements dans le temps par une spécification des propriétés des opérateurs tel que "*p aura éventuellement lieu*". Il y a plusieurs versions de logique temporelle, dont une concernant le CTL[5] pour laquelle celui-ci découle de l'arbre des transitions d'états.

Donnons à titre d'explication l'exemple classique des feux de circulation (Figure 1.1).

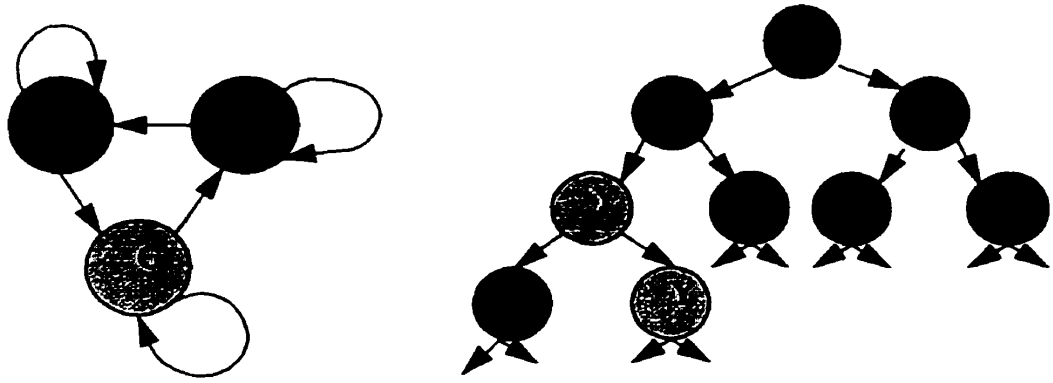


Figure 1.1: Construction d'un CTL à partir d'un automate.

Les chemins dans l'arbre de la figure 1.1 représentent toutes les possibilitées de calcul du système. Les formules en CTL font référence à l'arbre de calcul dérivé du modèle. La méthode CTL est classifiée comme une analyse des branchements logiques temporels, car ses opérateurs décrivent la structure de branchement de l'arbre. Les états du système sont des valeurs enregistrées dans des bascules. Chaque formule de la logique est soit vraie ou fausse pour un état donné. L'exactitude est évaluée à partir de l'exactitude de ces sous formules de manière récursive, jusqu'à ce qu'une proposition atomique soit vraie (1) ou fausse (0) pour un état donné. Une formule est satisfaite par un système si elle est vraie pour tous les états initiaux du système. Contrairement à la démonstration de théorèmes, la vérification du modèle est complètement automatique et relativement rapide. Cette méthode peut être utilisée afin de vérifier des spécifications partielles. Ainsi, elle peut

donner une information déterminante sur la validité d'un système, même si ce dernier n'est spécifié que partiellement. En tout état de cause, elle produit des contres exemples représentant des erreurs subtiles du design et peut ainsi être utilisé pour le déverminage. Le problème le plus important du modèle d'équivalence est l'explosion des états. En 1987 McMillan utilise les BDD et son extention MDD (Multi-valued Decision Diagram) afin de proposer une heuristique permettant de représenter les systèmes d'états de transition de manière efficace. D'autre part, il propose la minimisation de la sémantique afin d'éliminer les états non- nécessaires du modèle.

Les diagrammes de décision binaire

Les diagrammes de décision binaires (BDD) sont des graphes acycliques orientés qui permettent de représenter des fonctions booléennes. Selon Bryant, les BDD sont une représentation canonique des fonctions qui constituent une technique complémentaire à la vérification de modèle. La taille des BDD est très sensible à l'ordonnancement des variables et à la taille des BDD intermédiaires. La meilleure représentation des fonctions booléennes représentant les équations des circuits numériques est dérivée du BDD. C'est le "*Ordered Binary Decision Diagram*" (OBDD). Bien que les BDD ont été utilisés avec succès dans la vérification des FSM, ils ne constituent pas une panacée dans le cas des FSM de grandes tailles:

- les BDD demandent trop de mémoire,

- leur manipulation est très coûteuse en temps de calcul.

1.1.3 Démonstration de théorème

La démonstration de théorème est une technique dans laquelle le système et ses propriétés sont exprimés comme des formules en logique mathématique. Cette logique est donnée par un système formel, qui définit l'ensemble des axiomes et des règles d'inférence. La démonstration de théorème est le processus permettant de trouver une preuve d'une propriété en partant des axiomes pour un système spécifié. Les étapes de la preuve font appel aux axiomes, aux règles d'inférences et à la dérivation de définitions et de lemmes intermédiaires. La démonstration de théorème comprend plusieurs techniques regroupées en deux classes principales :

1. La déduction automatique qui est utilisée comme procédure générale de recherche.

2. La déduction interactive qui est beaucoup plus intéressante pour des développements formels systématiques en mathématique.

Les techniques sont :

- Boyer-Moore[6] (First-Order Logic).

- HOL (High-Oder Logic): HOL[7] utilise les notions standards de la logique des prédicats.

En plus des notions standards de la logique des prédicats, HOL est caractérisée par le fait que les variables peuvent êtres utilisées à des degrés plus élevés que ceux des fonctions et des prédicats. Ce genre de variable est appélé HOV (Higher Order Variable) et peut être quantifiée. Contrairement au modèle d'équivalence, la démonstration de théorème peut traiter les espaces à états infinis. Elle se base sur des techniques comme l'induction

structurale pour prouver ces domaines infinis. Cependant, comme elle requiert la participation humaine, le temps nécessaire à la démonstration d'un théorème est généralement beaucoup plus long que le temps de calcul dans le cas de l'équivalence du modèle et les erreurs sont beaucoup plus fréquentes.

1.1.4 Conclusion

Bien que la vérification formelle s'avère être un outil très utile dans la spécification des propriétés des systèmes numériques, elle ne peut être que complémentaire aux méthodes traditionnelles de vérification par simulation. En effet, les outils de travail ne cachent aucunement aux usagers leurs caractères formels. Par ailleurs les notations basées sur les écritures mathématiques sont généralement très complexes. Les outils et les méthodes développées aujourd'hui ne sont accessibles qu'aux spécialistes de la vérification formelle.

1.2 Test par mutation

Le test par mutation est une technique de test logiciel originalement proposée par Budd et al. 1978 [8]. Cette technique peut être décrite de la manière suivante: un grand nombre de fautes simples sont introduites, une à la fois, dans un programme sous test. Les versions modifiées, résultant de ces modifications, sont appelées mutants. Les données de test sont alors construites afin de détecter ces mutants. Lorsqu'une différence de comportement est détectée entre le mutant et la spécification de référence, le mutant est considéré tué et n'est plus réutilisé dans le processus de test. La mutation permet donc

d'élaborer un ensemble de vecteurs capables de détecter un ensemble fini et bien spécifié de fautes. Dans le but de faciliter la compréhension de cette méthode, il nous apparaît utile d'exposer les différents concepts du test logiciel.

1.2.1 Test logiciel

Le test logiciel consiste à appliquer un jeu de vecteurs de test sur un programme afin d'en révéler les défauts. Les stratégies de test se groupent en deux grandes catégories:

- structurelle(dites aussi stratégies de test boite-blanche) qui utilisent explicitement la structure du programme afin de générer les tests.

- fonctionnelle(dites aussi stratégies de test boite-noire) qui génèrent le test en n'ayant aucune information sur la structure du programme Dans ce dernier cas, on a recours à la spécification.

Le test fonctionnel examine les fonctions du programme depuis les entrées/sorties en vérifiant que pour des entrées légales on obtient des sorties correctes. Afin de quantifier la qualité du test, les stratégies structurelles cherchent à remplir certains critères relatifs à la structure du programme [9]:

- la couverture des énoncés : chacun des énoncés du programme doit être exécuté, au moins une fois.

- la couverture des branches: chaque condition de branchement binaire (IF-THEN-ELSE) doit être évaluée au moins une fois, aux valeurs booléennes VRAI et FAUX . Dans le cas

de branchement non-binaire (CASE) la condition doit être évaluée pour toutes les valeurs possibles.

- la couverture des conditions : si la condition est constituée de plusieurs conditions simples, le test des conditions consiste à évaluer toute condition simple alternativement à VRAI et FAUX.

- la couverture des chemins : tous les chemins possibles dans le programme doivent être exécutés(sensibilisés) au moins une fois.

Dans le test fonctionnel, le concepteur identifie les fonctions supposées être implémentées par le programme et teste ensuite la conformité du code avec la(les) spécification(s) de ces fonctions [10]. Malheureusement, au contraire du test structurel dans lequel beaucoup de méthodes basées en général sur la théorie des graphes ont été proposées , la majorité des méthodes de test fonctionnel sont des méthodes ad hoc et souvent manuelles.

1.2.2 Principe du test par mutation

Comme il a été dit précédemment, le test par mutation permet de quantifier la capacité d'un jeu de vecteurs fonctionnels à tester différentes fonctions dans un programme. Durant le test par mutation, des fautes simples introduites dans le programme original génèrent une multitude de programmes défectueux. Chacun de ces programmes contient une seule faute et est appelé mutant. Le test par mutation est fondé sur quatre hypothèses:

1. Le programmeur est compétent. Cette hypothèse présuppose que le programmeur écrit des programmes presque *"corrects"*. C'est-à-dire que bien qu'incorrects, les programmes écrits différeront de la "*bonne version*" par des fautes relativement simples.

2. Les fautes sont couplées [11]: on suppose ici qu' un jeu de test capable de détecter toutes les fautes simples est aussi capable de détecter les fautes plus complexes. Cette hypothèse a été justifiée à la fois de manière théorique [12] et expérimentale [13].

3. On dispose d'un ensemble valide d'opérateurs de mutation. Il s'agit d'un ensemble prédéfini d'opérateurs qui modélise toutes les fautes simples d'un programme. Ces opérateurs sont déterminés de manière tout à fait empirique et il n'existe toujours pas de méthode systématique afin de déterminer l'ensemble minimal relatif à un langage de programmation.

4. On possède un oracle : cette référence permet de vérifier systématiquement que le résultat du programme est conforme, pour un jeu de test donné.

1.2.2.1 Méthodologie.

Etant donné un programme P et un jeu de vecteurs de test T, il s'agit d'exécuter T sur P. Supposons, par ailleurs, que le résultat de T sur P soit correct; i.e. P passe le test T. Plusieurs programmes générés en appliquant de petites modifications sur P, appelées mutants, sont exécutés avec T comme stimulation. Si le résultat d'un mutant est différent de celui de P, on dit que le mutant est tué (détecté). Dans le cas contraire, il est dit vivant

(T est incapable de détecter le mutant). Si un mutant survit, le jeu de test T est insuffisant et doit être augmenté. Nous allons maintenant nous intérresser à la génération des mutants.

1.2.2.2 Génération des Mutants.

Plusieurs techniques ont été proposées afin de déterminer un jeu de vecteurs de test capable d'isoler des "fautes spécifiques". Une des plus difficiles et plus coûteuses tâches dans l'application de ces techniques est la génération de données de test qui est en général faite à la main. Une méthode [14] a été proposée afin de rendre la génération de ces tests, purement automatiques. Cette méthode est applicable de manière structurelle. Ces algorithmes utilisent le constraint based testing (CBT). Ces contraintes de test incluent des conditions qui permettent de tuer les programmes mutants et génèrent des données qui satisfont les contraintes. Le test de grands systèmes est composé de tests de sous-systèmes et de fonctions, si ces systèmes admettent la hiérarchie. Dans le cas d'énormes programmes non hiérarchiques, le processus de génération de vecteurs peut s'avérer intraitable.

Ces algorithmes sont assez performants s'ils sont applicables sur des "unités de programme". Une unité de programme est une sous-routine ou un ensemble de sous-routines ou de fonctions. La génération de jeu de test est en général une tâche laborieuse. En effet, afin de produire les vecteurs de test adéquats, le responsable du test est pratiquement obligé d'interagir avec le système de mutation en examinant de manière exhaustive les mutants survivants. Ce faisant, il doit ensuite construire manuellement un

jeu de vecteurs capables de les éliminer. Le processus de test par mutation commence par la construction de tous les mutants du programme. Les mutants sont générés à partir d'un jeu d'opérateurs de mutation. Les opérateurs utilisés représentent les fautes les plus répétitives faites par les concepteurs. Ces opérateurs suggèrent que les données de test couvrent tous les états, toutes les branches, les valeurs extrèmes, les domaines de perturbations et modélisent plusieurs types de fautes. Lorsque l'un de ces opérateurs est appliqué à un état, il effectue un changement simple (tout en gardant une bonne syntaxe). Le processus de test est le suivant: le programme original est testé par le jeu de test. Il en résulte des valeurs pour chacune des sorties. Un oracle (en général celui qui est en charge de la validation) se charge de vérifier que les sorties sont exactes. Si les sorties sont identiques, le programme doit être changé, puis le processus de test doit être réitéré. Dans le cas contraire, ces derniers tests sont appliqués à chacun des mutants survivants.

Après que ces derniers aient exécuté ces vecteurs avec succès, deux informations surgissent: la proportion des mutants tués, qui indique au concepteur dans quelle mesure le programme a été testé, et le nombre de mutants survivants, qui renseigne quant au nombre de fautes non testées, ou encore, quant à la faiblesse du jeu de test. A titre d'exemple, la Figure 1.2 montre le processus de mutation sur la fonction Max, qui donne le maximum de deux valeurs entières. Afin d'alléger la figure 1.2, tous les mutants sont portés sur la même figure. Autrement dit, chacune des instructions transformée par mutation représente un mutant séparé, dans lequel l'instruction mutée remplace l'instruction correspondante du programme initial. Dans le premier mutant, une valeur absolue a été injectée dans la première instruction. Le deuxième et le troisième mutants

sont les résultats d'injection de fautes dans l'opérateur relationnel de la seconde instruction. Le dernier mutant résulte de l'injection de fautes sur la variable de la troisième instruction .

```
   Function Max(integer: m, n):integer;
     begin
1              Max:= m;

--             Max := ABS(m);
2              if (n > m) then
--             if (n < m) then
--             if (n = m) then
3                  Max:= n;
--                 Max:= m;
```

Figure 1.2: la fonction Max avec quatre mutants superposés

Il est en général impossible de tuer tous les mutants, car certains changements n'ont aucun effet sur la fonctionnalité du programme original. En général, ces mutants équivalents sont identifiés soit par celui qui est en tâche de la validation soit par des heuristiques.

1.2.2.3 Les mutants équivalents

Un mutant est dit équivalent au programme original [12, 13, 15] s'il n'existe aucun vecteur de test permettant de révéler une différence de comportement entre les deux programmes. Il est en général très difficile de prouver qu'un mutant est équivalent. En effet, les raisons pour lesquelles un mutant peut être fonctionellement identique sont:

1. Le mutant est fonctionnellement équivalent au programme original. Ce mutant produira toujours la même sortie que le programme initial et ceci quelque soit le test.

2. Le mutant peut être tué, mais le jeu de test est insuffisant.

Une manière de démontrer l'équivalence est de procéder à des tests exhaustifs, évidemment, ceci n'est pas pratique pour de gros programmes. Dans la pratique, c'est au programmeur qu'incombe la tâche de déclarer un mutant équivalent. Ce problème reste toutefois une difficulté pratique et théorique devant l'automatisation du test par mutation.

1.2.2.4 Évaluation du test

Le test par mutation a été initialement proposé afin d'évaluer la capacité d'un jeu de test à exercer un programme. Les vecteurs de test peuvent être générés manuellement, aléatoirement, ou à l'aide d'un outil de génération de test. L'analyse de mutation associe une métrique (score de mutation) au jeu de test dans le but d'évaluer son efficacité. Ce score de mutation est le pourcentage de mutants non équivalents tués. Le score est donné par la formule suivante:

MS(P,T)=Mk / (Mt -Mq).

où *P* est le programme de test

T le jeu de test

Mk le nombre de mutant tué par T

Mt le nombre total de mutants générés par le programme

Mq le nombre de mutants équivalents.

Plus le nombre de mutants détectés est grand, plus l'efficacité du jeu de test est importante. Étant donné un jeu de test T, le programme P est d'abord exécuté et vérifié sur chaque vecteur qui compose T. Si le résultat est incorrect, une faute est trouvée et le

programme doit être corrigé puis le processus est relancé. Si le résultat est correct (P passe le test T), tous les vecteurs dans T sont exécutés pour tous les mutants vivants. Les mutants tués sont retirés du processus de test. Une fois tous les vecteurs de T exécutés, chacun des mutants encore vivant doit appartenir à une des deux catégories: le mutant est fonctionnellement équivalent au programme initial, ou il ne peut être tué par le jeu de test. Dans le premier cas, les deux programmes sont fonctionnellement équivalents et il n'existe aucune entrée permettant de révéler des résultats différents. Dans le deuxième cas, le jeu de test doit être renforcé par de nouveaux vecteurs. Notons que le coût du test par mutation tend à croître de façon importante, parce que chaque mutant est simulé avec l'ensemble de la suite de test et que le nombre de mutants peut devenir important. C'est pourquoi diverses techniques ont été utilisées afin de réduire le nombre de mutants générés tout en gardant un score de mutation satisfaisant. Les méthodes proposées ont été appelées mutation faible et mutation sélective.

1.2.2.5 Le test par la mutation faible

La mutation forte est une méthode très efficace, cependant elle génère une grande quantité de mutants. La méthode de test par mutation faible proposée par Howden[16,17], demande moins de tests. Elle ne considère ni l'hypothèse du programmeur compétent ni l'hypothèse du couplage des fautes. Les différences intervenant entre mutation faible et mutation forte sont que:

1. Les opérateurs de mutation dans la mutation forte sont dépendants du langage de programmation, tandis que dans la mutation faible, ils sont génériques et ne dépendent pas du langage.

2. Dans la mutation forte, il n'existe pas une méthode globale pour générer les tests qui révèlent les fautes prédéfinies par les opérateurs de mutation. Il existe toutefois un compromis dans lequel les tests peuvent être générés, si et seulement si, leur capacité de détection des fautes a été affaiblie, d'où le terme mutation faible. Howden (dans la même référence) a définit le composant comme la structure élémentaire de calcul dans le programme. Les références aux variables, les expressions arithmétiques et logiques sont des exemples de composants. Cette méthode de mutation peut être décrite de la manière suivante: si P est un programme qui contient un composant C, il existe alors une mutation C'de C et P' de P où P' correspond à P contenant C'. Dans la mutation faible, un test t est construit de manière où C et C' ont au moins une valeur différente lors de l'exécution de t par P et P', sans garantir toutefois que les résultats de P et P' soient différents. Les changements (ou les mutations des composants) dans la mutation faible affectent les composants suivants :

1. les références à une variable: il s'agit de remplacer une référence à une variable par une autre. Pour détecter ce type de mutation, il faut exécuter le programme sur des entrées pour lesquelles la variable en question se verra attribuée des valeurs différentes de celles de toutes les autres variables compatibles dans le programme.

2. les affectations d'une variable : il s'agit ici d'affecter une valeur à la mauvaise variable. Pour détecter cette mutation, il est nécessaire d'affecter une valeur différente de la valeur actuelle de la variable.

Les deux types précédents sont des mutations primitives qui contribuent dans les types de mutation suivants:

3. transformations des expressions arithmétiques : l'expression est modifiée par l'addition d'une constante, la multiplication par une constante ou bien en modifiant les coefficients qu'elle contient. La détection de l'addition d'une constante ou de la multiplication par une constante demande un seul vecteur de test, tandis que la détection des coefficients modifiés est plus complexe et demande plus d'un vecteur(Howden82).

4. transformation des relations arithmétiques : l'expression est modifiée par un opérateur relationnel incorrect ou l'addition d'une constante. La détection d'un opérateur incorrect est effectuée par l'exécution de la relation sur des données qui distinguent l'opérateur correct de tous les autres opérateurs. La détection de l'addition d'une constante est effectuée par la sélection appropriée d'un point de test.

5. transformation des expressions booléennes: les expressions booléennes sont modifiées en utilisant les opérateurs précédents sur les sous-expressions. Pour détecter la mutation, l'expression doit être testée sur toutes les valeurs dans la table de vérité de l'expression. Howden a proposé des méthodes pour contrôler la taille du jeu de test dans le cas des expression booléennes. Le test par mutation faible a plusieurs avantages sur le test

par mutation (forte). Notamment, il n'est pas nécessaire d'effectuer une exécution des mutants pour sélectionner les données de test. De plus, le nombre de vecteurs de test est souvent petit (un seul) peut parfois suffire pour la plupart des mutations. Un autre avantage majeur est la possibilité de spécifier a priori les données de test nécessaires pour que la mutation donne une sortie incorrecte. Par contre, le test par mutation faible ne garantit pas que les données choisies pour détecter une mutation donne une sortie incorrecte pour le programme tout entier.

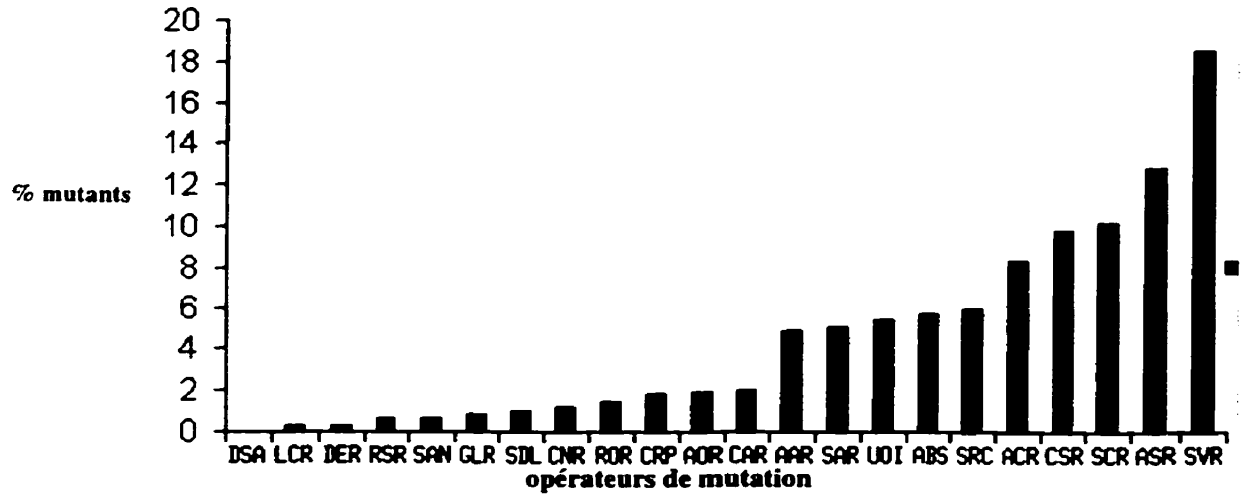
1.2.2.6 mutation sélective

Cette méthode a été proposée par Offutt et al. afin de réduire le coût de la mutation en réduisant le nombre de mutants générés. Le coût du test par mutation est essentiellement lié au nombre de mutants. Afin d'estimer la complexité du test par mutation, de nombreuses techniques empiriques ont été proposées. La première l'a été par Acree [18]. Selon ce modèle, le nombre de mutants généré est proportionnel au nombre de lignes du programme au carré. Budd [19] a affiné cette relation en estimant que le nombre de mutants étant proportionnel à O(Vals*Refs), où Vals représente le nombre d'objets de données du programme (ex : déclaration des variables et des constantes) et Refs représente le nombre de références à ces objets. Une étude statistique récente effectuée par Offutt et Lee, [20] a montré que la relation proposée par Budd est la plus représentative de la complexité réelle du test par mutation. Afin de réduire la complexité du test, une réduction des opérateurs de mutation a été proposée afin de réduire le nombre de mutants du programme. Cette approche appelé la mutation sélective, s'effectue de deux manières.

Dans la première, au lieu de considérer tous les mutants pouvant être générés par un opérateur de mutation, on sélectionne, de façon aléatoire, une portion de ces mutants. Une étude réalisée par Wong et Mathur (1995) a montré qu'en appliquant seulement 10% des mutants, on pouvait atteindre des scores de mutation très proches de ceux atteints par la mutation originale. Dans la seconde, certains opérateurs de mutation sont éliminés du processus de génération de test.

1.2.2.7 La mutation N-sélective

Dans un premier temps, il a été proposé de supprimer les opérateurs responsables de la génération du plus grand nombre de mutants [21]. Mathur[22] a proposé d'appliquer une méthode appelée la mutation 2-sélective, sur les deux opérateurs les plus coûteux (ASR et SVR). ASR consiste à remplacer chaque signal par un tableau alors que SVR consiste à remplacer chaque variable par un scalaire. L'expérience a débuté par une étape de comptage des mutants générés (à partir de 28 programmes différents). Les résultats obtenu sont présentés à la figure 1.2. Par la suite, une comparaison a été faite entre les deux méthodes de test (sélective et non-sélective). Pour effectuer cette comparaison, un jeu de vecteurs de test ayant un score de mutation de100% (pour la mutation sélective) a été créé. Puis, dans un deuxième temps, ce même jeu de vecteurs a été appliqué à la mutation non sélective et son score a été de nouveau calculé.



Afin d'éviter au maximum les aléas (dûs au fait que l'on génère les vecteurs aléatoirement), 5 jeux de tests ont été générés pour chacun des dix programmes considérés. Les valeurs ci-après sont en fait la moyenne de tout cela. Le but de cette expérimentation est de montrer qu'un jeu de vecteurs capable d'obtenir un bon score de mutation lors de la mutation sélective est aussi capable d'obtenir un bon score pour la mutation forte.

mutation 2-sélective : 99.99% (score de mutation) et 23.98% (de mutants non générés).

mutation 4-sélective : 99.84% (score de mutation) et 41.36% (de mutants non générés).

mutation 6-sélective : 99.71% (score de mutation) et 60.56% (de mutants non générés).

Le but de cette expérimentation n'est pas de mettre à jour des mutants équivalents, mais de comparer les scores de mutation obtenus pour la méthode sélective avec ceux obtenus par la mutation forte.

1.2.2.8 La mutation E-sélective

Cette nouvelle technique vise encore à diminuer le nombre de mutants générés en supprimant certains opérateurs. Les opérateurs sont classés en 3 grandes catégories:

- type R : remplacent chaque opérande du programme par toutes les autres opérandes compatibles.

- type E: modifient les expressions en remplaçant les opérateurs par d'autres de même nature (arithmétique, relationnelle ou booléenne).

- type S : modifient les tests conditionnels du programme.

L'objectif sous-jacent est de déterminer empiriquement un sous-ensemble suffisant d'opérateurs de mutation permettant une réduction de la complexité de test, tout en gardant une efficacité satisfaisante. Une étude empirique comparant les performances des mutations suivantes: ES, RE, RS, et E a été menée. Dans une mutation sélective de type XY ou de type X, on applique seulement les opérateurs de mutation compris dans le(s) type(s) mentionnés lors de la génération de test. Il a été montré que le type R génère un nombre de mutants de l'ordre de O(Vals * Refs), le type E de l'ordre de O(Refs), et le type S de l'ordre de O(Lignes). Il est à noter que le type R est le plus coûteux, car 70% des

mutants générés sont en général de ce type. Une des premières préocupations lors de l'utilisation de la mutation sélective est de déterminer la possibilité d'éliminer le type R sans avoir à réduire l'efficacité du test généré. En d'autres mots, est-il possible de procéder au processus de mutation en n'utilisant que des types -ES ou -E. Une étude empirique effectuée par Offut et al. [21] a montré qu'un jeu de test 100% adéquat par rapport à la mutation sélective-ES est suffisamment adéquat (99.54% en moyenne) par rapport à une mutation non-sélective.

Il a aussi été proposé de supprimer les opérateurs de type S dans le souci de définir un ensemble minimal suffisant d'opérateurs de mutation. Le résultat de l'application de la mutation sélective à montré que le test généré est là encore quasi-adéquat (99.5 1 % en moyenne). Par conséquent, dans la pratique, on peut considérer que la mutation sélective -E est suffisante pour générer des tests efficaces.

A titre d'exemple, les résultats d'une expérience réalisée par Irene koo [16] sont donnés ci-dessous. Ces résultats ont été obtenus en appliquant un processus identique à celui appliqué pour la mutation sélective précédente :

mutation ES-sélective : 99.54% (score de mutation) et 71.52% (de mutants non générés)

mutation RS-sélective : 97.31% (score de mutation) et 22.44% (de mutants non générés).

mutation RE-sélective : 99.97% (score de mutation) et 6.04% (de mutants non générés).

mutation E-sélective : 99.51% (score de mutation) et 77.56% (de mutants non générés).

Pour la mutation E-sélective, le score de mutation va de 98.5% à 99.5% suivant le programme, pour un pourcentage de mutants non générés allant de 37.1% à 92.12%. Par la suite, une étude a été menée dans le but de réduire encore plus le nombre de mutants générés, en supprimant un opérateur parmi les 5 (cf tableau 1.1).

Tableau 1.1 : mutation E selective

Programme	Avec les 5 opérateurs	Sans UOI	Sans ROR	Sans LCR	Sans AOR	Sans ABS
Banker	99.57	98.57	99.57	99.57	91.57	91.57
Bub	99.93	98.93	99.93	99.93	99.93	97.93
Cal	99.63	95.63	99.63	99.63	99.63	99.63
Euclid	99.00	97.30	99.30	99.30	99.30	99.30
Find	99.30	98.70	99.70	99.70	99.70	99.70
Insert	99.75	98.95	99.95	99.95	99.95	97.95
Mid	99.90	98.00	100.0 0	100.0 0	100.0 0	94.00
Quad	100.00	99.09	99.09	99.09	99.09	97.09
Trytip	99.36	98.36	98.36	99.36	99.36	98.36
Warshall	99.67	97.67	98.67	98.67	98.67	98.67

1.2.2.9 Génération de test

Dans le test par mutation, le but du concepteur est de trouver des vecteurs capables de révéler une différence de comportement. Un test est considéré efficace s'il tue au moins

un mutant. Un moyen largement utilisé dans le domaine du test par mutation afin de générer automatiquement des tests, c'est d'avoir recours à des contraintes mathématiques. En fait ces contraintes permettent de générer des vecteurs de test à partir des propriétés que les entrées du programme doivent avoir pour tuer les mutants. Ces contraintes permettent de rejeter les vecteurs inefficaces. Les contraintes doivent amener le mutant à révéler un comportement différent du programme original. Puisque le mutant est représenté par un seul changement dans le programme, l'état du mutant doit être différent de celui du programme original tout de suite après l'état muté. Cette caractéristique est une condition nécessaire afin de tuer le mutant. Cependant, cette condition n'est pas suffisante, car il peut très bien arriver que le changement de comportement ne parvienne pas à l'une des sorties du programme. Aussi, une condition suffisante est que le test cause une différence à l'une des sorties. Il est cependant très difficile de satisfaire les deux conditions, car cela demande de savoir à l'avance le chemin que va suivre le programme, ce qui est bien sûr un problème très difficile à résoudre. Pour être certain que le test tue le mutant, le mutant doit être exécuté en globalité et, sa réponse comparée avec la réponse du programme. En pratique, il est admis qu'un test remplit la condition nécessaire s'il remplit la condition suffisante sinon, le mutant produit le même résultat que le programme original et reste vivant.

CHAPITRE 2

APPLICATION DU TEST PAR MUTATION AUX CIRCUITS VLSI

2.1 Introduction

Ce chapitre fait état des investigations effectuées par l'equipe Valsys de Grenoble et de Valence, ainsi que des résultats obtenus suite à une collaboration effectuée avec notre groupe. Ces travaux sont à la base d'une proposition d'un premier algorithme de validation par mutation de circuits numériques. Tout au long de ce chapitre, nous présenterons les raisons qui ont guidé le choix d'un outil de mutation spécifique ainsi que son architecture. Nous décrirons les différents bancs d'essai sur lesquels les mutations ont été réalisées. Enfin, nous décrirons un algorithme de validation et les problèmes sous-jacents à cette méthode de validation.

Bien que le test par mutation ait été originalement proposé afin de vérifier un programme, il a par la suite été proposé d'utiliser cette méthode afin de valider une description matérielle. En effet, tout comme pour le test des logiciels, cette méthode se révèle intéressante si on travaille à partir de languages de description matériel tels que VHDL(Very High speed integrated circuit Hardware Description Language) et VERILOG. Par ailleurs, les approches fonctionnelles sont indépendantes de l'implémentation matérielle du circuit, ce qui permet d'avoir recours à un haut niveau d'abstraction capable de traiter des circuits VLSI complexes. Il n'existe pas un ensemble de principes reconnus pour la validation de circuits VLSI, qui est en général realisée de manière ad-hoc. Aussi, en se tournant vers le test logiciel, on peut, si on considère chaque description VHDL

comme un programme, utiliser les rudiments du test logiciel (voir section 1.2.1). Par ailleurs, le test par mutation se compare à bien des égards au test matériel, comme il sera demontré dans la suite de ce chapitre.

Afin de valider et d'établir les fondations de la méthode de validation par mutation de descriptions matérielles, nous avons eu recours à l'outil Mothra[23], qui est l'environnement le plus complet pour le test par mutation dans le domaine logiciel. Cet outil permet de tester des modules écrits en FORTRAN. Mothra génère automatiquement tous les mutants d'un programme en utilisant un ensemble défini d'opérateurs de mutation. L'architecture de Mothra ainsi que la définition de ces opérateurs sont présentés dans la section suivante.

2.2 Présentation de Mothra

Mothra utilise une technique de test qui est basée sur des contraintes, ce qui permet d'automatiser le processus de test par mutation en représentant sous la forme de contraintes mathématiques, les conditions pour lesquelles un mutant est tué. Mothra est constitué d'un ensemble d'outils qui permet à la fois de créer les mutants et de générer des vecteurs de test. La génération de ces vecteurs est réalisée par Godzilla. Les outils Mothra/Godzilla ont été implémentés en langage C.

2.2.1 Génération de vecteurs de validation.

Godzilla génère des vecteurs de test en tentant de trouver les propriétés que doivent avoir les entrées du programme afin de tuer les mutants. Dans le chapitre 1, il a été montré qu'il fallait que le comportement suite à la mutation soit différent de celui du programme non muté. Aussi, dans la validation par mutation, il est nécessaire que les entrées du programme engendrent un état erroné directement après l'exécution de l'état mutant. Par analogie avec le test matériel, il faut au moins que le test stimule la panne à la source en produisant une différence de comportement observable à cet endroit (condition nécessaire). Ensuite, une fois la panne stimulée, cette différence doit se propager sur un résultat observable de l'extérieur.

Godzilla décrit ces conditions sous forme d'un système de contraintes mathématiques. La condition d'accessibilité est décrite par un système de contraintes appellé " expression de chemin ". Si on représente le lieu où une panne est susceptible d'être introduite comme un état, chaque état du programme a une expression de chemin permettant de décrire la manière d'atteindre l'état mutant à partir de n'importe quel état. Une condition spécifique au type de faute modélisé par le mutant décrit à la fois la condition qui engendre le comportement défectueux du programme et la nécessité d'une exécution intermédiare incorrecte. Ces deux critères constituent une condition nécessaire, car bien qu'un état intermédiaire incorrect soit nécessaire, il n'est pas suffisant pour tuer un mutant. Afin de tuer un mutant, il faut que le programme génère des sorties incorrectes de manière à ce que l'état final du programme soit différent de celui du programme

original. Cependant, bien qu'il soit nécessaire de satisfaire la condition de suffisance, elle est pratiquement irréalisable. En effet, déterminer complètement la condition de suffisance implique de connaître à l'avance le chemin que va prendre le programme. Grâce à ce système de contraintes, Godzilla intègre plusieurs techniques de test logiciel dont le détail est donnée ci-dessous:

La couverture des états : Si on considère un état comme une combinaison des valeurs de toutes les variables dans le programme à un instant donné, chaque état du programme est exécuté au moins une fois. La couverture est réalisée directement à partir du système de contraintes appelé " l'expression de chemin ".

La couverture des branches : technique qui requière que toutes les branches du programme soit parcourues. Cette technique est respectée par le test par mutation, en effet les mutants ne peuvent être tués que si les prédicats prennent la valeur vraie puis fausse.

Domaine de perturbation : Cela consiste à partitionner les entrées du programme en domaines. Chaque vecteur compris dans un domaine suit le même chemin. La stratégie du domaine de perturbation sélectionne les vecteurs de manière à prendre les valeurs frontières du domaine. Les opérateurs de mutation forcent les vecteurs à satisfaire le domaine de perturbation en modifiant légèrement chaque expression (en ajoutant 1, en soustrayant 1 ou en prenant 10% de la valeur etc.) et en remplaçant chaque opérateur. L'architecture de Godzilla est donnée à la figure 2.1

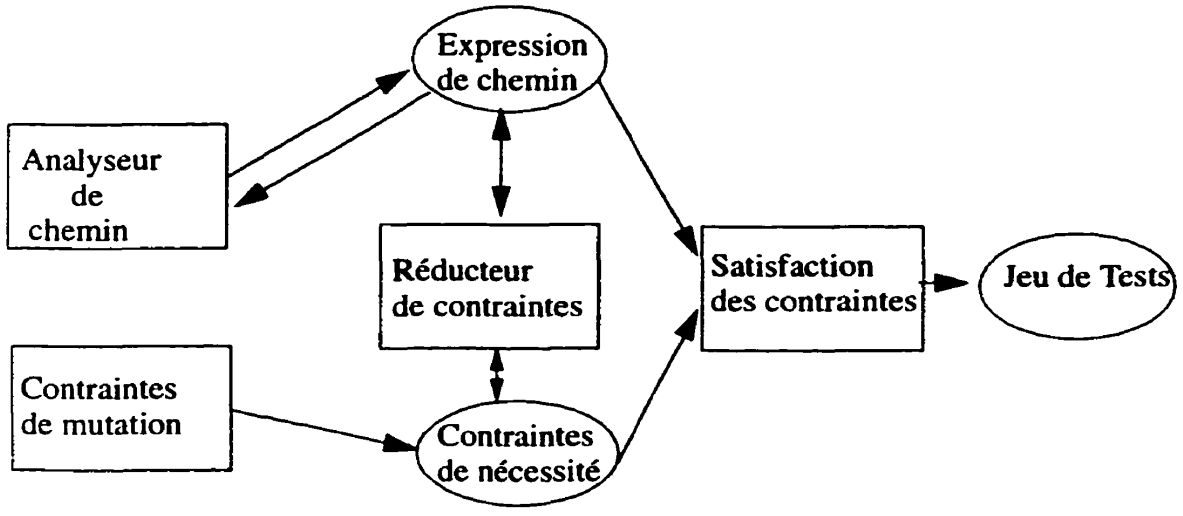


Figure 2.1: Implémentation de Godzilla

Les principales fonctions de Godzilla sont montrées ci-dessus. Cet outil intègre différentes fonctions, chacune d'entre elles a été implémentée dans différents programmes, ceux-ci communiquent par l'intermédiaire de fichiers qui sont représentés par des ellipses. Les flèches de la figure 2.1 indiquent les flots d'information du système Godzilla. Les fichiers sont accessibles par des routines communes à tous les outils, ils sont ainsi vus en faisant abstraction du type de données. Ce système permet une plus grande modularité et une plus grande extensibilité en donnant à chaque outil un accès uniforme aux routines qui créent, modifient, enregistrent et récupèrent les contraintes.

2.2.1.1 La représentation des contraintes.

Le système de contraintes de Godzilla a une structure hiérarchique composée de contraintes, d'expressions et de clauses arrangées de manière " disjonctive normale ". L'expression algébrique est l'élément de base du système de contrainte. Elle est composée de variables, de parenthèses et d'opérateurs spécifiques au FORTRAN. Une contrainte est une paire d'expression algébrique reliée par un des opérateurs suivants $\{>, <, =, \geq, \leq, \neq\}$. Les contraintes sont évaluées soit à la valeur binaire VRAI, soit à la valeur FAUSSE et elles peuvent être modifiées par l'opérateur de négation NOT ($\neg$). Une *clause* est une liste de contraintes reliées par les opérateurs logiques ET ($\wedge$) et OU ($\vee$). Une *clause conjonctive* utilise seulement le ET logique et une *clause disjonctive* seulement le OU logique. Dans le système Godzilla, toutes les contraintes sont sous la forme *normal disjonctive* (DNF), qui est en fait une série de *clauses conjonctives* reliées par des opérateurs OU. La forme DNF est utilisée par commodité durant la génération des contraintes (en effet, il suffit de satisfaire une seule *conjonctive clause*). Godzilla considère le système de contrainte comme étant un ensemble de contraintes, qui prises collectivement forment un vecteur de test.

A titre d'exemple, considérons le système de contraintes de l'équation suivante:

$$((X + Y \geq Z) \wedge (X \leq Y)) \vee (X > Z)$$

X+Y est une expression et (X+Y $\geq$ Z) est une contrainte. ((X+Y $\geq$ Z) $\wedge$ (X $\leq$ Y)) est une *clause conjonctive*, et l'expression entière est un système de contraintes. Des vecteurs de test tels que (X=3, Y=4, Z=2), (X=3, Y=4, Z=4) et (X=4, Y=1, Z=3) satisfont les con-

traintes.

2.2.1.2 L'analyseur de chemin.

L'analyseur de chemins utilise la technique de couverture des chemins parcourant ainsi tout le programme afin de construire des contraintes d'accessibilité. Pour chaque état du programme original, l'analyseur de chemins crée une contrainte telle que si le vecteur atteint cet état, la contrainte est vraie. Il faut noter qu'il vaudrait mieux avoir des contraintes inverses (si la contrainte est satisfaite, l'état est exécuté). Cependant, créer des contraintes qui garantissent l'accessibilité implique de trouver une réponse au problème de connaître à l'avance le chemin que pourrait prendre le programme. L'algorithme construisant les expressions de chemin partiel est présenté ci-dessous:

Générateur de contraintes

Variables: CPE est l'expression courante du chemin

```
        PE [ ] contient l'expression courante de chaque état.
        P est le programme.
        S et S' sont des états du programme P.
        ρ est l'expression d'un prédicat.
1 CPE = VRAI                                    -- initialisation
2 pour chaque état S de P
3       PE [ S ] = FAUX                         -- initialisation (aucun état n'a été trouvé)
4       fin
5       pour chaque état de S de P
6       PE [ S ] = PE [ S ] V CPE               -- CPE(nouvelle maniere d'atteindre S) est ajouté à l'état précédent
7       CPE = PE [ S ]                          -- le nouvel état devient l'état courant
8       Si  S est un état de contrôle de flux alors   -- si plusieurs possibilitées
              { ρ est le prédicat de S, S' est la cible de la branche. } -- condition préalable à l'évaluation de S
9             mise à jour de CPE selon le type de l'état.
10            PE [ S' ] = PE [ S' ] V ( PE [ S ]Λ  ρ )   -- ajout de l'expression du chemin et de la conjonction de
11 end                                                   -- CPE et du controle de flux
```

L'expression du chemin courant CPE (Current Path Expression) est tout d'abord initialisée à la valeur VRAI, et chaque état à la valeur FAUSSE indiquant qu'aucun

chemin menant à cet état n'a été trouvé. Pour atteindre l'état S, plusieurs actions sont entreprises. Premièrement, Le CPE est ajouté à la liste des précédentes expressions de S. En effet, le CPE représente une nouvelle voie possible pour atteindre S (ligne 6). Chaque chemin susceptible d'atteindre S est enregistré comme une nouvelle *clause disjonctive* à l'expression des chemins. Ensuite, le nouveau PE de l'état S devient le CPE (ligne 7). Chaque *clause disjonctive* dans l'expression des chemins représente une exécution différente du chemin menant à l'état S. Enfin, si S est un état de contrôle de flux (représentant différentes possibilitées, branches...), le CPE est mis à jour par une règle de modification qui dépend de S. La *clause disjonctive* et le prédicat de contrôle de flux(condition du choix multiple) sont finalement ajoutés à l'expression du chemin de l'état cible (ligne 9 et 10).

2.2.1.3 Le résolveur de contraintes

La dernière étape du processus de génération de vecteurs de tests est de trouver des valeurs qui satisfont le système de contraintes. Godzilla travaille avec des heuristiques, et produit rapidement des vecteurs de test quand les contraintes ont une forme simple et plus lentement lorsque leur forme est plus compliquée. Tout d'abord, un domaine de valeurs possible est assigné à chaque variable. Théoriquement, ce domaine de valeur dépend du type de variable et de la machine sur laquelle le programme est utilisé. En pratique, Godzilla permet de réduire ce domaine. Par défaut, les domaines de variation des variables sont initialisés arbitrairement aux valeurs (-100, 100). Cependant, ces valeurs peuvent être modifiées dans le cas ou le besoin s'en fait sentir. Chaque contrainte dans un

système de contrainte réduit le domaine de variation d'une ou de plusieurs variables. Quand le domaine de variation d'une variable est réduit à une valeur, cette valeur fait partie du vecteur de test. La variable est alors remplacée par sa valeur dans toutes les autres contraintes. Si le domaine de variation est réduit à l'ensemble vide, la résolution de ce système de contrainte est alors impossible. Lorsque toutes les simplifications ont été faites, et qu'il reste un domaine de plusieurs valeurs, une valeur est choisie. Godzilla utilise une heuristique qui affecte la variable qui utilise le plus petit domaine de variation, en espérant que c'est cette variable qui a le moins de chance de rendre le système infaisable. Ce processus est répété jusqu'à ce que toutes les variables se soient vues assignées une valeur.

A chaque fois qu'une valeur est affectée à une variable, le domaine de variation des contraintes est réduit d'une dimension. Si la valeur est mal choisie, elle peut rendre le système infaisable. Lorsqu'un système de contraintes devient infaisable après une affectation, le processus recommence à partir du système de contraintes original. Les expériences ont montrés qu'en moyenne, le processus de réduction de contrainte trouve une solution en 4 itérations (avec un maximum de 25). Dans le cas où l'objectif n'a pas été atteint après 25 essais, on considère que le système de contraintes ne possède pas de solutions. Toutefois ce dernier cas ne s'est pas encore produit.

2.3 Opérateurs de mutation

Les opérateurs constituent le noyau du test par mutation. Ces opérateurs sont au nombre de 22 et ils identifient les modifications syntaxiques responsables des programmes mutants. Les opérateurs qui sont implémentés dans Mothra sont énumérés au tableau 2.1.

Tableau 2.1 : Opérateurs de mutation

Opérateurs de mutation	Description
AAR	Array reference for Array reference Replacement
ABS	ABSolute value insertion
ACR	Array reference for Constant Replacement
AOR	Arithmetic Operator Replacement
ASR	Array reference for Scalar variable Replacement
CAR	Constant for Array reference Replacement
CNR	Comparable array Name Replacement
CRP	Constant RePlacement
CSR	Constant for Scalar variable Replacement
DER	Do statement
DSA	Data Statement Alteration
GLR	Goto Label Replacement
LCR	Logical Connector Replacement
ROR	Relational Operator Replacement
RSR	Return Statement Replacement
SAN	Statement Analysis
SAR	Scalar variable for Array reference Replacement
SCR	Scalar for Constant Replacemen
SDL	Statement DeLetion
SVR	Scalar Variable Replacement
UOI	Unary Operator Insertion

2.4 Chaine de validation

La description de la méthodologie de test originalement proposée par l'équipe du VALSYS et modifiée par notre groupe est donnée dans la suite de ce paragraphe et est résumée à la figure 2.4.

Il faut tout d'abord réécrire la spécification VHDL en FORTRAN car MOTHRA ne produit des mutations que sur ce langage. Cette description doit être comportementale car il est très difficile d'écrire un code structurel en Fortran. Dans une seconde étape, une génération de vecteurs de test est réalisée par Mothra. La troisième étape consiste en une phase d'optimisation qui permet d'augmenter le nombre initial de vecteurs du jeu de tests, jusqu'à ce que le score de mutation soit satisfaisant. L'étape suivante est une étape de synthèse. Synopsys est utilisé pour générer une description structurelle du circuit. La dernière étape consiste à calculer la couverture de faute du jeu de vecteurs généré par MOTHRA. Pour cela, une simulation de fautes est réalisée avec l'ATPG (*automatic test pattern generator*) HILO. L'efficacité du jeu de vecteurs de validation produit par Mothra mesurée à l'aide du score de mutation(voir chapitre I) est comparée à la couverture de faute obtenue avec la suite de test appliquée à l'implémentation structurelle. L'objectif ici est de corréler le score de mutation avec une mesure dont le sens est bien établi.

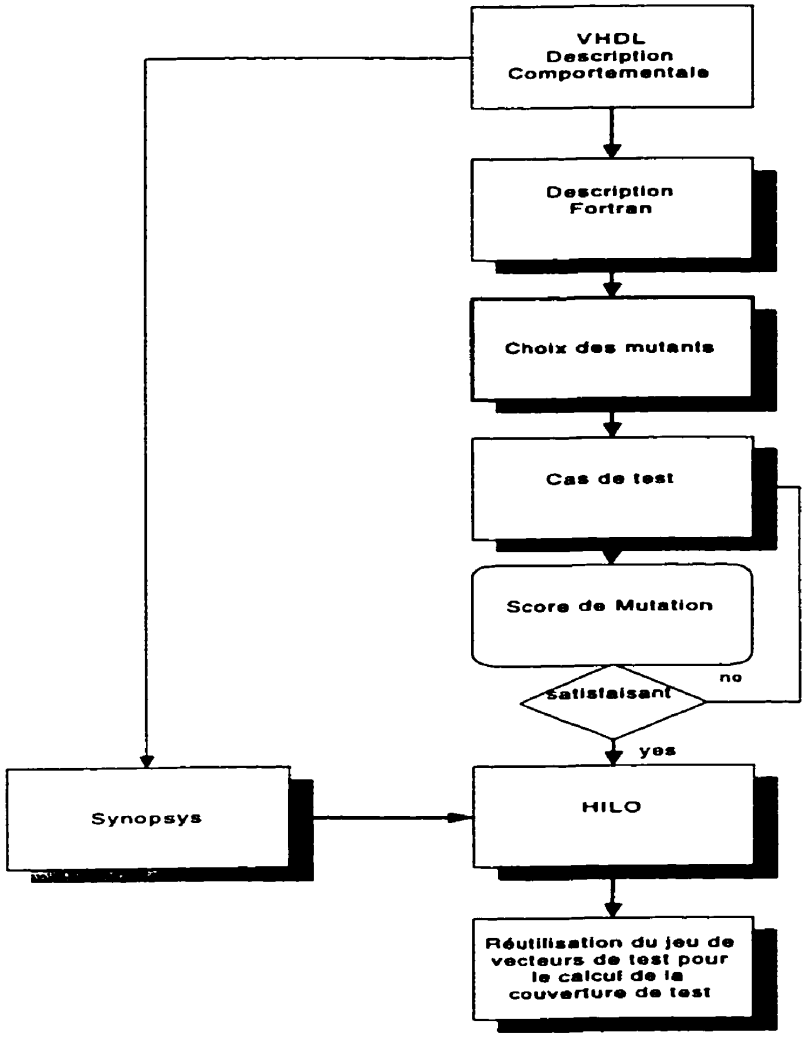


Figure 2.2: Méthode de validation de circuits numériques utilisant Mothra

2.5 Validation de la méthode

L'objectif premier de cette section est d'élargir le champ d'expérimentation de la méthode de test par mutation à des circuits de complexité plus importante que ceux réalisés à Grenoble. Pour cela, il a été décidé d'utiliser des bancs d'essai de types différents: petit séquentiel, gros séquentiel et gros combinatoire. Enfin, un circuit a spécialement été conçu afin de metttre en évidence les atouts et/ou les faiblesses du test par mutation. Les résultats de la collaboration entreprise entre Monsieur Zoccarato et notre groupe ainsi qu'une brève description de ces bancs d'essai sont donnés dans la section suivante.

2.5.1 Description des bancs d'essai.

cla : Le circuit cla est un processeur RISC 32 bits défini par Hennessy et Patterson. Ce processeur possède un jeu d'instruction simple, un seul mode d'adressage, un décodage unique de son jeu d'instruction et une architecture très simple, qui met en évidence toutes les fonctionnalités du principe RISC. Le jeu d'instruction du processeur RISC possède avant tout des instructions arithmétiques pour nombres signés et flottants. Mais aussi des caractéristiques telles que : les interruptions et exceptions, trois différents modes d'utilisation (utilisateur, superviseur et défaut) et un mode d'adressage supplémentaire. Le circuit cla permet d'effectuer des additions sur des nombres de 32 bits en calculant la retenue de manière anticipée.

Edhii: Ce circuit fait parti d'un processeur ancillaire qui insère / extrait des données dans un signal vidéo numérique exprimé selon diverses normes. Ce circuit sera présenté de manière plus explicite dans le chapitre 3. Le circuit edhii permet la réécriture des données provenant du module EDHin, dans les trames vidéo. Un explication plus détaillée de ce circuit est donnée au chapitre 3.

Sortie: Ce module, comme le précédent, fait aussi partie du module EDH du processeur ancillaire. Sa fonction est de générer les paquets de sortie.

Mux: Ce circuit a été conçu afin de mettre en évidence des lacunes possibles de certains algorithmes implémentés dans le système de mutation et les problèmes sous-jacents à la mutation. Une description plus détaillée de la fonctionnalité de ce circuit se trouve au paragraphe (2.6.4).

2.5.2 Processus de validation

La démarche adoptée pour valider les 4 circuits mentionnés précédemment est la suivante :

-Réécriture du programme VHDL en FORTRAN. Il est possible que lors de cette opération, la structure du programme ou même la structure des données soit modifiée. En effet, sachant que la définition du ET logique FORTRAN (il en est de même pour le OU logique) ne correspond pas à la définition du ET logique VHDL, il est absolument nécessaire de décomposer cet entier en autant de booléen qu'il est nécessaire, ce qui a pour conséquence d'augmenter de façon non négligeable la complexité du programme.

-Réécriture du programme VHDL en un autre programme VHDL de manière à le rendre compatible avec le programme FORTRAN qui a été généré lors de l'étape précédente. Cette phase est nécessaire car la structure des données du programme FORTRAN n'est plus compatible avec celle du programme VHDL, entraînant une incompatibilité des vecteurs générés par MOTHRA avec la synthèse du circuit réalisé par SYNOPSYS. Il faut noter qu'il existe une alternative à ce travail, elle consiste à modifier tous les vecteurs générés par MOTHRA de manière à les rendre compatibles avec la description structurelle faite par SYNOPSIS. Après que chacun des programmes sources ait été généré, une mutation non sélective suivi d'une mutation sélective ont été utilisées. Pour la mutation selective, le score de mutation a été produit à partir des mutants générés par les opérateurs aor, ror, lcr, abs et uoi.

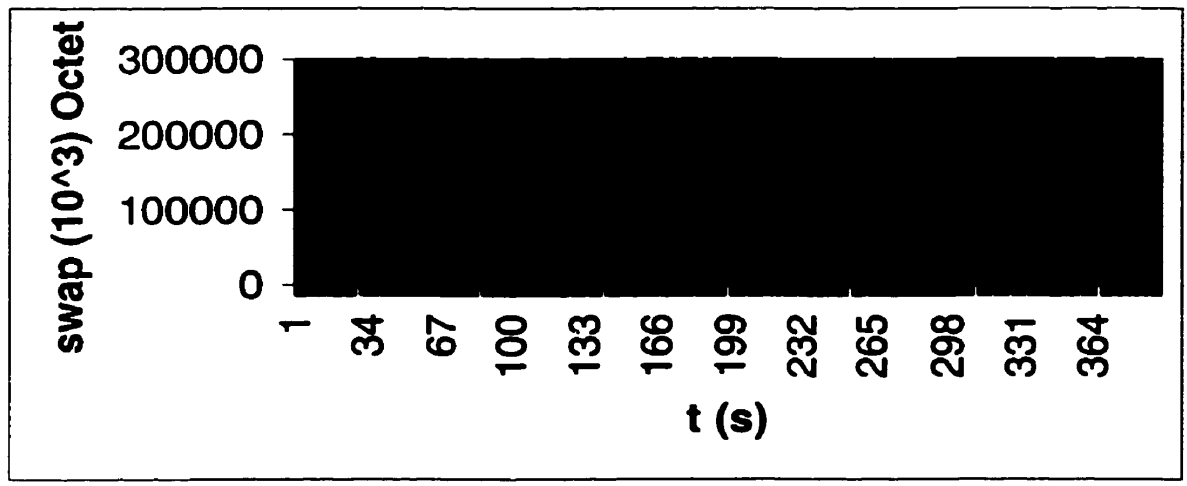
2.6 Résultats.

2.6.1 Le circuit sortie

Une des premières difficultés, inhérente au fait de traduire la spécification VHDL en FORTRAN, c'est l'abscence d'utilisation de vecteurs binaires. En effet, il est possible de définir dans l'entité VHDL des entrées binaires de longueurs différentes (2 bits, 25 bits...), chose qui n'existe pas en FORTRAN, puisque les seuls types disponibles sont les types *entiers*, *booléens*, *flottants* et *double flottants*. Afin de contourner ce problème, nous avons ajouté au début du programme FORTRAN une condition spécifiant que les vecteurs binaires ne dépassent pas leurs valeurs entières équivalentes. Ce procédé introduit des

mutants supplémentaires, cependant, afin de ne pas influencer les résultats et le score de mutation, nous ne chercherons pas à les tuer. Le deuxième problème rencontré a été un problème d'espace mémoire. En effet, lors de la satisfaction des contraintes générées par la mutation non sélective, Mothra consomme beaucoup d'espace mémoire et l'utilisation d'une machine de type 1 (65 Mo de RAM et 118 Mo de swap space) n'est pas suffisante. Nous avons donc relancé ce même processus sur une machine de type 2 (131 Mo de RAM et 257 Mo de swap space). La figure 2.5 représente la consommation de swap pour la machine de type 2 lors de la résolution des contraintes (il est clair que 118 Mo ne suffit pas). Les résultats du test par mutation sont représentés dans le tableau. Il est à noter que 3793 mutants ont été générés. Le nombre de vecteurs de validation est présenté dans le tableau 2.2.

Figure 2.3: consommation mémoire pour le circuit de sortie



Bien que la couverture de test soit élevée, le score de mutation lui ne l'est pas, ce

qui tend à révéler que les vecteurs générés par MOTHRA ne sont pas tous adéquats, que ce soit pour la mutation sélective ou non sélective. Il est cependant possible d'augmenter ce score de mutation en augmentant le jeu de test de manière à tuer les mutants existant. Cependant, cet enrichissement manuel s'avère être long et diminue l'automatisation du test par mutation. Aussi, c'est à celui qui est en charge de la validation (l'oracle) d'estimer le temps nécessaire à cet enrichissement. Par ailleurs, le circuit de sortie possède une certaine redondance, ce qui réduit le score de mutation puisque le ou les types de mutants non tués se répètent plusieurs fois. Comme il a été dit précédemment, la couverture de collage est comparée au score de mutation afin de montrer que les vecteurs fonctionnels produits peuvent aussi être utilisés pour le test matériel. A titre indicatif, nous pouvons comparer cette couverture de fautes à celle obtenue par 20 vecteurs de tests générés de façon totalement aléatoire. Le résultat obtenu par ces 20 vecteurs peut paraître très bon, mais il faut savoir que pour ce genre de circuit, il est très facile d'avoir une couverture 70 à 80%. Il est par contre beaucoup plus difficile d'obtenir les 20% restants. Concernant le score de mutation, la différence est plus importante, on peut donc dire pour ce circuit qu'il est beaucoup plus difficile de tuer un mutant que de détecter une faute de collage.

Tableau 2.2 : circuit de sortie

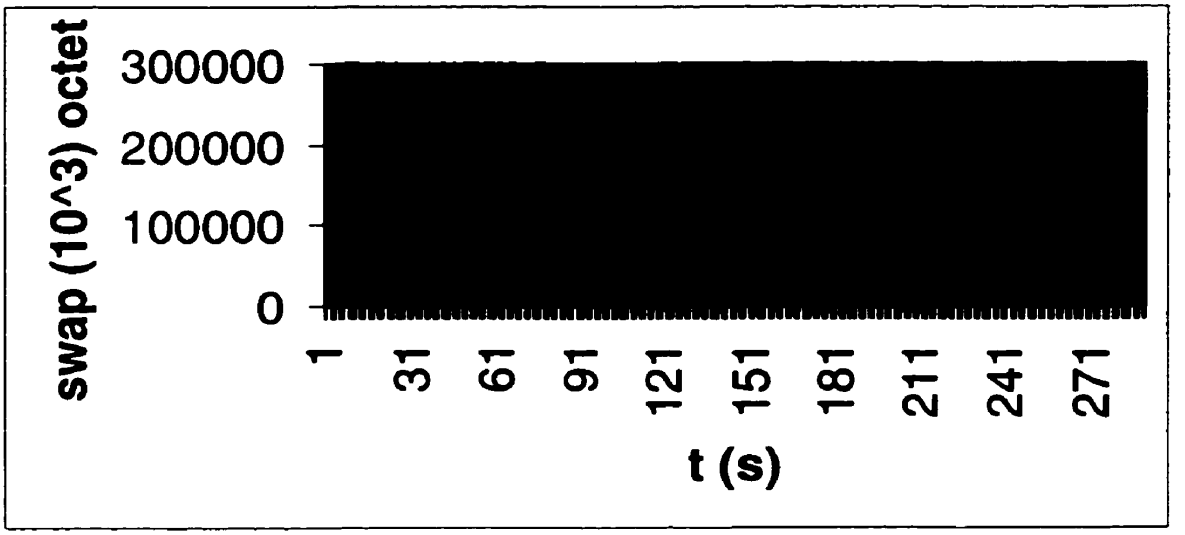
	Nombre de vecteurs	Score de mutation	Couverture de fautes
Vecteurs générés par Mothra pour la mutation non sélective	446		
Vecteurs générés par Mothra pour la mutation sélective	139		
Vecteurs générés aléatoirement	20	0.3 %	80.3 %

2.6.2 Le circuit d'entrée de l'edh

Pour ce circuit, nous avons également été confrontés au problème de la plage de variation des variables d'entrées, ainsi qu'à celui de l'espace mémoire. Cependant, cette fois-ci, le fait de changer de machine n'a pas été suffisant . En effet que ce soit pour la mutation sélective ou non sélective, les machines de type 1 et 2 ne réussissent pas à satisfaire les contraintes générées pour ce circuit. En effet, dans certains cas, le programme avorte et dans d'autres cas, bien que des mutants aient été générés, aucun vecteur ne fut construit. La consommation mémoire est donnée à la figure 2.6. Afin de pouvoir effectuer le processus de mutation, il a fallu réécrire le programme edhii de manière à ce que les contraintes générés par MOTHRA soient plus simples. La fonctionnalité du programme a bien évidemment changé, cependant nous essayons par ce " stratagème " de générer des vecteurs qui puissent tester un maximum de conditions, chemins, valeurs extrêmes et particulières, de manière à avoir une bonne couverture de fautes. Dans ce dernier cas

l'emphase a été mis sur une augmentation de la couverture de faute comparativement au score de mutation.

Figure 2.4: consommation mémoire pour le circuit d'entrée



Le tableau présente un récapitulatif des résultats obtenus. On y trouve le score de mutation et la couverture de fautes de collage pour des vecteurs générés par MOTHRA pour la mutation sélective et non sélective. Les résultats obtenus par le score de mutation, que ce soit pour la mutation sélective ou pour la mutation non sélective, sont très faibles (environ 20%). Cela s'explique par le fait que la moitié des mutants générés n'ont pas été tués. En effet, ce circuit est séquentiel et possède un état interne. Toutes les transitions partant de cet état interne ne peuvent donc pas être tuées. Les scores de mutation sont affectés par le même phénomène. Une explication plus détaillée sera donnée dans la section 2.6.4.

Tableau 2.3 : Résultats de sortie pour le circuit d'entrée de l'EDH

	Nombre de vecteurs	Score de mutation	Couverture de fautes
Vecteurs générés par Mothra pour la mutation non sélective	589		
Vecteurs générés par Mothra pour la mutation sélective	114		

2.6.3 Le circuit cla

Nous avons encore une fois été confrontés à un problème de mémoire, cependant le système de mutation n'a pas pu générer de vecteurs de test pour ce fichier, que ce soit pour la mutation sélective ou non sélective. Afin d'obtenir des résultats concluants, 8 fichiers différents ont été écrits. Chacun de ces fichiers correspond à un circuit de calcul anticipé de retenu pour des données faisant 4, 8, 12, 16 ,20, 24, 28 et 32 bits. Le seul circuit sur lequel nous pouvons appliquer la mutation sélective et non sélective est le circuit traitant des données de 4 bits. L'application du test par mutation sur ce circuit a donné les résultats se trouvant dans le tableau 2.4. Il est à noter que 4420 mutants ont été générés. Le nombre de vecteurs générés pour la mutation non sélective est de 4046. Ce nombre est énorme, puisque le nombre de vecteurs nécessaires à un test exhaustif est de $512 = 2^9$. Même si la mutation sélective réduit notablement le nombre de vecteurs générés, on peut dores et déjà dire que ce type de circuit constitue un véritable problème pour Mothra. Il apparaît à première vue que le grand nombre de variables internes contenues dans la description

VHDL de ce circuit serait responsable des difficultés qu'éprouve Mothra à le tester. Afin d'expliquer les problèmes rencontrés, une étude sur la complexité algorithmique du générateur de vecteur est rapportée dans la sous section suivante.

Tableau 2.4 : Résultats de sortie pour le circuit cla

	Nombre de vecteurs	Score de mutation	Couverture de fautes
Vecteurs générés par Mothra pour la mutation non sélective	4046		
Vecteurs générés par Mothra pour la mutation sélective	270		

2.6.4 Complexité algorithmique

Bien que Mothra soit l'outil de mutation le plus performant, il apparaît évident que ce système éprouve certaines difficultés pour la validation de circuits VLSI. Afin de proposer une amélioration à la méthode décrite dans ce chapitre, il a fallu identifier la source des difficultés éprouvées par Mothra.

Afin d'expliquer les problèmes de consommation mémoire pour certains circuits, un balayage des algorithmes de Godzilla a montré que les seuls susceptibles d'engendrer un nombre important d'itérations sont: l'algorithme de réécriture des variables internes en fonction des entrées primaires et l'algorithme de réduction du domaine de variation des variables. Ces deux algorithmes ont une complexité proportionnelle au nombre total de clauses contenues dans le système de contraintes. Il n'est cependant pas possible d'éval-

uer précisément le nombre d'itérations nécessaires, car celui-ci dépend de la structure du programme. On peut simplement dire qu'au maximum, on a une contrainte par état et au moins une clause par contrainte. Le nombre de clauses peut facilement devenir très important. Dans l'unique but de démontrer cette thèse, le nombre de contraintes du programme MID(voir en annexe B) a été augmenté progressivement. Cette augmentation s'effectue en augmentant la profondeur d'imbrication des if, tout en regardant l'évolution de la consommation mémoire. Le résultat de cette étude est montré au tableau 2.5, il a suffit de rajouter 4 if pour que la machine de type 1 n'arrive plus à générer les vecteurs de test.

Tableau 2.5 : Résultats obtenus pour la fonction MID

	Mid1	Mid2	Mid3	Mid4
Nombre de mutants	830	1219	1669	2146
Nombre de vecteurs	654	898	1285	X
Temps nécessaire à la génération des vecteurs de test	0h 43mn 3s	1h 17mn 46s	3h 44mn 44s	indeterminé
Nombre de clauses du programme	249	817	3089	12177
Taille de la swap utilisée lors de la satisfaction des contraintes	6184 Ko	22300 Ko	66208 Ko	La totalité

Une deuxième difficulté provient du nombre important de mutants équivalents augmentant ainsi la complexité du processus de validation et la tâche du concepteur. Ces mutants équivalents sont par définition des spécifications qui ont le même comportement que le programme original. Cependant, une hypothèse émise est que le nombre de mutants équivalents produits par le système de mutation pourrait être dû au fait qu'il n'y

aurait pas de propagation de la faute à l'une des sorties primaires. Afin de corroborer cette hypothèse, un programme a été construit afin de montrer que la propagation de la faute vers les sorties primaires n'est pas effectuée. En effet, la justification de la faute est faite, mais Mothra se contente " d'espérer " que les sorties primaires du circuit révèlent la présence d'une faute en générant aléatoirement des valeurs sur les entrées qui n'apparaissent pas dans les contraintes. Par opposition aux algorithmes de test matériel qui permettent de générer des vecteurs qui justifient et **cherchent à propager** les fautes. Pour analyser plus en détail la pertinence des vecteurs générés par Mothra, nous avons créé 4 circuits qui représentent un empilement de plusieurs multiplexeurs (cf. figure 2.5).

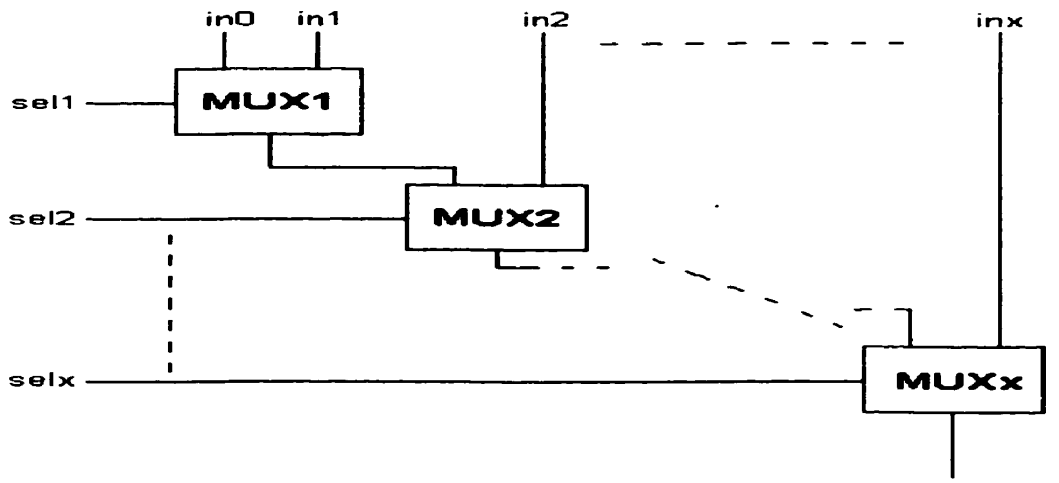


Figure 2.5: Cascade de multiplexeurs

Comme pour le circuit précédent, le nombre de vecteurs générés par Mothra est très important. Les vecteurs générés pour ce circuit ne propagent pas les fautes comme il est montré dans le tableau 2.6. Ce phénomène va en s'accentuant si la profondeur du circuit augmente. Aussi, c'est le grand nombre de vecteurs produits et non la qualité des vecteurs qui tue les mutants.

Tableau 2.6 : Cascade de Multiplexeurs

Circuits	Type de Mutation	Entrées	Nb de Vectors	Mothra	Aléatoire
				MS	MS
1 Mux	Full	3	19	100%	96.8%
1 Mux	Selective	3	3	38.7%	83.9%
2 Mux	Full	5	61	97.8%	97.8%
2Mux	Selective	5	6	86.8%	80.2%
4 Mux	Full	7	189	82.1%	81.1%
4 Mux	Selective	7	17	75.5%	57.1%
8 Mux	Full	9	670	75.3%	86.7%
8 Mux	Selective	9	34	58.7%	52.7%

2.6.5 Conclusion

La mutation sélective est loin de donner le score de mutation qui a été obtenu pour le test logiciel (57.38% contre 99.61% en moyenne), ou même celui obtenu sur les circuits d'expérimentation sans processus d'enrichissement [2] (95.02% en moyenne). Nous pouvons dire la même chose pour le taux de couverture de fautes, dans de moindre proportion cependant (74.98 % contre 95 %). La seule manière d'obtenir à la fois un bon score de mutation et une bonne couverture de fautes est de compléter manuellement le jeu de test, surtout si le circuit testé est une machine à états. En effet, Mothra est incapable de classer

les vecteurs de manière à positionner une machine à états sur un état précis. Pour une FSM, nous avons à la fois des contraintes sur les variables du programme à l'instant présent, mais aussi sur les variables du/des instant(s) précédent(s), chose qui n'est absolument pas prévue dans Mothra (cf. circuit edhii)

2.6.5.1 Étude de la puissance des mutants

Le nombre de vecteurs générés est vraiment très important (surtout pour la mutation non sélective). Un bon moyen de diminuer ce nombre consiste à utiliser la mutation sélective. De manière à ne retenir que les opérateurs les plus efficaces, une étude sur la puissance des 5 opérateurs principaux a été faite. Une fois les opérateurs les plus efficaces sélectionnés, il faut que mis les uns avec les autres, ils détectent le plus de fautes possibles. Pour chacun des 7 fichiers (sortie, Mux2, Mux2, Mux4, Mux8, Edhii et Cla4) nous avons appliqué chacun des 5 opérateurs de la mutation sélective. Tout d'abord, on effectue le lancement du processus de génération des vecteurs de test. Le nombre de mutants générés est inscrit dans la colonne 1, et le nombre de vecteurs dans la colonne 2. Le calcul du score de mutation est porté dans la colonne 3. A partir de là, 2 cas sont possibles, soit les vecteurs sont adéquats, auquel cas nous passons directement à l'étape 4, soit les vecteurs ne sont pas adéquats et nous passons à l'étape 3. Une augmentation manuelle du jeu de vecteurs est réalisée de manière à avoir des vecteurs adéquats (score de mutation de 100% (cf. colonne 5)) afin de ne pas influencer les résultats de cette étude. La colonne 4 représente le nombre initial de vecteurs plus ceux générés manuellement. Le calcul de la

couverture de fautes de collage est porté dans la colonne 6.

Tableau 2.7 : Puissance des opérateurs

	UOI					
	Nb mutants	Nb vect-eurs	M. S.	Nb2 vect.	M. S. 2	Stuck at
Sortie	260	80	45.8%	122	100%	96.6%
Mux1	3	2	100%	2	100%	71.4%
Mux2	6	4	83.3%	5	100%	83.3%
Mux4	14	10	85.7%	12	100%	84.1%
Mux8	28	20	71.4%	23	100%	84.5%
Edhii	116	51	24.1%	57	55.2%	80.8%
Cla4	163	163	80.4%	176	100%	100%

	ROR					
	Nb mutants	Nb vect-eurs	M. S.	Nb2 vect.	M. S. 2	Stuck at
Sortie	420	19	23.3%	112	100%	93.3%
Mux1	7	1	0%	3	100%	57.1%
Mux2	14	2	28.6%	5	100%	58.3%
Mux4	28	4	35.7%	9	100%	93.2%
Mux8	56	8	39.3%	14	100%	83.3%
Edhii	259	34	6.2%	45	55.6%	79.3%
Cla4	0	X	X	X	X	X

	LCR					
	Nb mutants	Nb vect-eurs	M. S.	Nb2 vect.	M. S. 2	Stuck at

Sortie	287	0	0%	55	100%	82.3%
Mux1	0	X	X	X	X	X
Mux2	0	X	X	X	X	X
Mux4	0	X	X	X	X	X
Mux8	0	X	X	X	X	X
Edhii	112	0	0%	8	50%	51.4%
Cla4	469	95	75.5%	107	100%	100%

	ABS					
	Nb mutants	Nb vecteurs	M. S.	Nb2 vect.	M. S. 2	Stuck at
Sortie	150	42	28.7%	110	100%	87.1%
Mux1	0	X	X	X	X	X
Mux2	0	X	X	X	X	X
Mux4	0	X	X	X	X	X
Mux8	0	X	X	X	X	X
Edhii	48	20	16.7%	28	66.7%	78.3%
Cla4	12	12	100%	12	100%	97.5%

Bien que certains mutants permettent d'obtenir des scores de mutation intéressants, on ne peut affirmer qu'un opérateur est plus important que les autres. En effet, l'opérateur AOR donne de mauvais résultats car aucun des programmes testés ne contient d'opérateurs arithmétiques. Un meilleur moyen d'étudier la puissance des opérateurs est d'en montrer la capacité à découvrir des fautes de collages en fonction du nombre de fois où on le trouve dans le programme ou, le nombre de fois où les vecteurs couvrent une région du design. Le test par mutation semble être une solution au problème de la validation. En effet, cette méthode fournit un modèle qui permet de mesurer à quel point le circuit a été validé . Il est également intéressant de noter que les mêmes vecteurs sont

capables de très bien tester le circuit final si toutefois l'on s'est assuré de leur capacité à tuer les mutants par le score de mutation. Afin de répondre à plusieurs des limitations identifiées pour un test par mutation avec les outils existants, des opérateurs propres au langage VHDL, un générateur de mutants ainsi qu'une nouvelle méthodologie de test par mutation sont exposés dans le chapitre suivant.

CHAPITRE 3

MÉTHODE SYSTÉMATIQUE D'ENRICHISSEMENT DE VECTEURS FONCTIONNELS

3.1 Introduction

3.1.1 Limitations de Mothra pour la validation.

Dans le chapitre précédent, des expériences ont été menées afin de montrer les différents pièges pouvant découler du test par mutation. Le banc d'essai concernant la cascade de multiplexeurs (cf 2.6.7) a été choisi afin d'exposer les similaritées existant entre la génération pseudo aléatoire du modèle de panne *bloquée-à 1 ou 0* et la suite de test produite par Mothra. L'outil semble présumer que les pannes se propagent naturellement à l'une ou l'autre des sorties primaires. Cette hypothèse peut s'avérer incorrecte avec des circuits ayant un problème potentiel d'observabilité. Par exemple, dans le cas des multiplexeurs, ce phénomène a considérablement réduit la couverture de pannes ainsi que le score de mutation. En effet si les entrées de commande de la cascade sont stimulées avec des vecteurs aléatoires, la probabilité qu'une panne puisse être observée décroit exponientellement avec la distance mesurée en nombre de multiplexeurs de la panne à la sortie. Cette expérience, dont les résultats sont rapportés au tableau 2.6, a confirmé pour ce banc d'essai que les tests générés par Mothra pouvaient dans certains cas n'être guère meilleurs que ceux produits par une génération exhaustive.

Soulignons que dans certains cas, le nombre de vecteurs de test produits par le système de mutation exède le nombre de vecteurs requis pour une validation exhaustive. Ce problème n'est pas dû à une limitation du test par mutation, mais à une limitation de Mothra qui utilise un procédé de génération de test limité. Afin de répondre à ces limitations, un certain nombre de solutions sont proposées dans la section suivante.

3.1.2 Redéfinition de la méthode de validation.

Nous avons identifié trois limites importantes avec la méthode proposée au chapitre 2. La première découle des traductions du VHDL au FORTRAN imposées par MOTHRA. La seconde provient de la difficulté à détecter les mutants équivalents. La troisième découle de l'hypothèse implicite que les différences de comportements se propagent naturellement à un noeud de sortie observable, sans avoir recours à des actions explicites.

La méthode proposée dans la suite de ce chapitre est d'utiliser le test par mutation comme un moyen de mesurer l'efficacité d'une suite de tests définie lors de la validation. Avec cette méthode, nous supposons que les suites de test proviennent de celui qui est en charge de la validation. L'objectif est de détecter les parties de la fonctionnalité où une ou plusieurs erreurs de spécification passeraient inaperçues à travers la suite de test. Les mutants qui ne peuvent être détectés permettent alors d'identifier les imperfections de la suite de vecteurs de validation. Cela guide le responsable de la validation vers les régions du design insuffisamment testées. Considérant la difficulté à déterminer précisément les mutants équivalents, nous avons mis l'emphase sur la possibilité d'éviter de produire des

mutants qui pourraient l'être. Finalement, afin d'empêcher le manque de fidélité dans la description fonctionnelle associée à la traduction du VHDL au FORTRAN, nous avons implémenté un outil qui réalise les mutations directement à partir de descriptions VHDL. Par ailleurs, il a été montré dans le chapitre 1.2.2.8 qu'un ensemble bien défini d'opérateurs de mutation (mutation sélective) était suffisant afin de développer une bonne qualité de test. De plus, le choix de ce type de mutation réduit sensiblement le nombre de mutants, ce qui réduit sensiblement la tâche du concepteur. Dans la suite de ce chapitre, nous présenterons le banc d'essais qui nous a guidé dans le choix des opérateurs de mutation et l'implémentation du générateur de mutants.

3.2 Description du banc d'essai

3.2.1 Processeur ancillaire.

La puce à données ancillaires possède deux modes de fonctionnement: MUX et DEMUX. A ces deux modes se rajoutent plusieurs sous mode de fonctionnement : normal, bypass, tones, cbar, tones et cbar. Brièvement, le sous mode bypass n'altère en rien le signal vidéo d'entrée. Les sous modes cbar et tones sont des sous modes de test . Le premier modifie le domaine actif de l'image alors que le second insère ou extrait(tout dépendant du mode MUX/DEMUX) des échantillons audio de 1 KHz. En mode MUX, la puce doit encoder plusieurs types de données reçues sériellement en paquets ancillaires et les insérer dans l'espace HANC. Les types de données insérées ou extraites de la trame

vidéo sont: audio, time code, RS422 ou GPS. Les paquets ancillaires sont formés de la manière suivante:

Anc Data Flag	Anc Data Flag	Anc Data Flag	DID	DBN	DC	User Data Words	CS

Toutes les cellules du tableau sont des mots de 10 bits. Les *ancillary Data Flags* servent à identifier un paquet ancillaire. Le DID (*Data Identification*) renseigne quant à la nature des données présentes dans le paquet. Le DBN (*Data Block Number*) sert à numéroter les paquets de 1 à 255. Le DC (*Data Count*)précise le nombre de *User Data Words* que contient le paquet. Les *User Data Words* contiennent les données sérielles transformées en mots de 10 bits selon un standard défini pour chaque type de données. Le CS (*checksum*) est un mot servant à la détection d'erreurs dans le paquet.

Le signal vidéo entre dans la puce par le module EDH. Ce dernier est responsable de l'identification du standard vidéo en présence. Notons que la puce est faite pour traiter les standards vidéos 4:2:2, 525 lignes, 27MHz; 4:2:2, 625 lignes, 27 MHz; 4:2:2, 525 lignes, 36 MHz; 4:2:2, 625 lignes, 36 MHz et HDTV. Le module d'entrée doit également reconnaître les signaux de synchronisation qui permettent de se situer dans le standard vidéo. Il est enfin responsable de la détection des erreurs présentes dans ce même signal d'entrée.

En mode MUX, la puce est responsable du multiplexage des nouveaux paquets ancillaires présents dans le signal vidéo d'entrée.

En mode DEMUX, le contrôleur reconnaît les paquets ancillaires pertinents et inscrit les USERS DATA WORDS dans une RAM. Dans les deux modes de fonctionnement, le module EDH est responsable de l'encodage des fanions révélant des erreurs dans le signal vidéo d'entrée, ainsi que de l'encodage de certains mots dont il fait le calcul, mots qui seront utiles aux modules qui recevront ce signal.

3.2.1.1 Module EDH

Le module EDH (*error detection and handling*) permet la génération de paquets ancillaires. L'écriture de ces paquets se fait à des lignes définies par le standard video. Ces standards sont au nombre de cinq. A savoir:

- 4:2:2 525 lignes à 27 MHz
- 4:2:2 525 lignes à 36 MHz
- 4:2:2 625 lignes à 27 MHz
- 4:2:2 625 lignes à 36 MHz
- HDTV

Le module EDH doit aussi être en mesure d'écrire dans les trames vidéos les données provenant du bus ou du module EDHin. Le choix d'une de ces deux entrées est effectué par le contrôleur du système. Enfin, afin de détecter des erreurs de transmission, le module EDH doit être capable de générer des codes de vérification (CRCC).

3.2.1.2 Implémentation du CRCCs

L'encodage redondant est une méthode de détection d'erreur qui étend l'information à un nombre de bits supérieur à celui de la donnée initiale. En général, plus le nombre de bits redondants utilisés est grand, plus les chances de détecter une erreur dans la transmission seront grandes. Les normes de télévision numérique prévoient que le codage de cette redondance est effectué grâce à un CRC (cyclic redundancy codes). Les CRCC(cyclic redundancy code checkers) vérifient les différences entre les données transmises et les données originales. Lorsqu'on utilise le CRCC afin de vérifier une trame vidéo, cette trame est traitée comme un long mot binaire qui est alors divisé par un générateur. Cette division produit un "reste" qui est transmis avec la donnée. A la reception, la donnée est divisée par le même générateur et le reste est comparé à celui qui a été envoyé. La comparaison permet ainsi de détecter une erreur qui se serait éventuellement produite durant la transmission. En général les calculs du CRC sont réalisés à l'aide d'une LFSR (linear feedback shift register). Cependant, cette méthode demanderait dans notre cas l'utilisation d'une horloge trop rapide. Par conséquent, l'utilisation d'un CRC permettant le calcul parallèle a été préconisé. L'algorithme qui permet de synthétiser ce CRCC est basé sur la même approche que la LFSR. Le générateur CRC-16 a ainsi été choisi et est donné en annexe.

3.2.1.3 Implémentation du module EDH

Le module EDH a été réalisé de manière hiérarchique. Il est constitué de cinq sous-modules:

1. crc active picture: permet les calculs du crc dans la région active picture.

2. crc fullfield picture: permet les calcul du crc dans la region fullfield.

3. edhin: permet la réécriture des données provenant du module EDHin dans les trames vidéo.

4. edhout: génère le paquet ancillaire

5. control: controleur qui commande les deux modules crc.

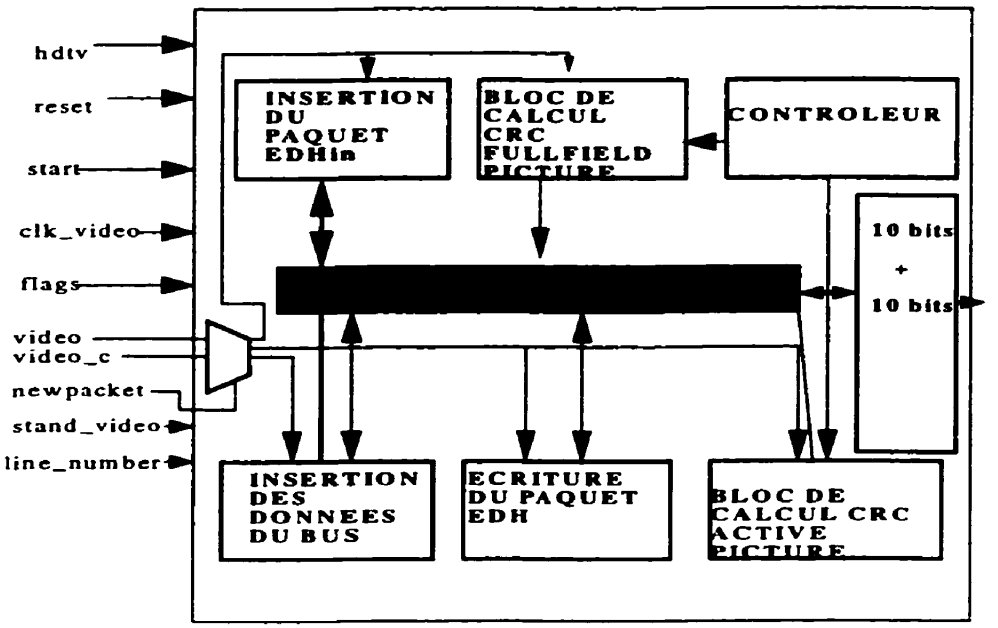
ENTITE DU MODULE EDH OUT

L'entité correspond à la boite noire du sous-système EDH out.

Les signaux d'entrées sont les suivants:

1. *hdtv* (pour le signal HDTV)
2. *reset* (permettant d'initialiser le module)
3. *start*
4. *clk_video* (horloge)
5. *newpacket* (permettant l'insertion de nouveaux paquets)
6. *ues, ida, idh, eda, edh* (flags d' erreurs)
7. *video* (signal exterieur codé sur 10 bits)
8. *video_c*(signal video provenant du bus)
9. *stand_vidéo* (standard vidéo)
10. *line_number*
11. Le signal de sortie est qout qui est codé sur 20 bits .

Figure 3.1: Architecture de l'EDH



La description de chacun des modules a été réalisée de manière comportementale à l'aide du langage VHDL. La description VHDL est donnée en annexe. Chacun des modules sera considéré comme un composant.

3.3 Opérateurs de mutation

Afin de pouvoir réaliser la mutation, nous avons adopté la mutation sélective. Nous avons donc défini un ensemble d'opérateurs spécifiques que nous exposerons dans la suite de cette section. En effet, le langage VHDL étant différent du Fortran, certaines

fonctionnalités n'ont plus à être prise en compte. On peut citer par exemple les opérateurs agissant sur les tableaux qui sont :AAR, ACR, ASR, CAR, CNR et SAR , les instructions do (DER) et goto (GLR), les instructions d'altération de données (DSA), l'instruction return (RSR), les instructions d'analyse des états (SAN) et enfin les instructions de remplacement des constantes (SRC). La génération des mutants est réalisée par un ensemble de programmes de mutation. Ces programmes sont spécifiques au langage VHDL et ils permettent de couvrir tous les chemins, conditions, valeurs limites et régions de perturbation d'un design. Un grand défi a été d'éviter la génération d'un nombre important de mutants équivalents, qui augmentent le temps de test et réduisent le score de mutation, tout en compliquant la tâche du concepteur. En effet, certains opérateurs utilisés au chapitre 2 créent systématiquement des opérateurs équivalents. On peut par exemple citer l'opérateur ABS (insertion de valeurs absolues) qui remplace les expressions et sous-expressions par leur valeurs absolues. Cependant, ces valeurs sont identiques aux valeurs initiales dans le cas de vecteurs VHDL non signé (unsigned). Un autre exemple vient de la traduction en Fortran de vecteurs binaires fixes. La modélisation de ces vecteurs, inexistants en Fortran, produit quelquefois des entiers ou réels correspondant à la taille maximale de ce vecteur, augmentant ainsi la limite désirée. Citons par exemple l'opérateur UOI, qui augmente la valeur supérieure des valeurs entières. Quand l'intervalle est changé, il est possible qu'aucun vecteur de validation ne détecte le mutant correspondant, car la limite peut être trop haute.

Les opérateurs VHDL que nous proposons furent définis lors de la conception du processeur ancillaire(cf 3.2.1.1). Ces opérateurs représentent les erreurs les plus communes pouvant être faites par les concepteurs dans une spécification VHDL.

3.3.0.1 Programmes de mutation

Un résumé des opérateurs est donné au tableau 3.1. Une description de ces opérateurs est effectuée ci-dessous.

Tableau 3.1 : Opérateurs de mutation

OPERATEUR	FONCTION
CNR: *comparable array name replacement*	Chaque tableau est remplacé par un tableau de même type et de même dimension présent dans la description VHDL.
CSR: *constant for scalar variable replacement*	Chaque variable et signal présent dans le programme VHDL est remplacé par une constante du même type.
GRP: *generic replacement*	Simule les mauvaises connexions des modèles. Cet opérateur est assez intéressant pour les designs de type structurels.
SUR: *signed unsigned replacement*	Teste les vecteurs binaires signés et non-signés.
VSAR: *variable and signal replacement*	Teste les mauvaises assignations des variables et signaux. Modélise aussi des erreurs de synchronisation et le mauvais séquencement des actions
SVIR: signal and variable initialisation	modélise les mauvaises initialisations des registres et des variables
CLR: *constant limit replacement*	Teste les limites supérieurs et inférieures des différents registres. Le même processus est effectué pour les variables. La région de perturbation est elle aussi simulée.
SSR: *state sequencement replacement*	Teste la séquence des états dans une machine à états.
COR: *conditional operator replacement*	Substitution de toutes les conditions possibles.
LOR: loop operator replacement	Change la longeur des boucles
LCR: *logical replacement*	Chaque opérateur logique est remplacé par les autres.
LOR: *loop operator replacement*	Change la longeur des boucles
LER: *level replacement*	Teste le niveau de sensibilité des circuits (haut et bas)

CLR: une des méthodes les plus couramment utilisées lors de la validation d'un circuit VLSI est le test des valeurs limites. En effet on définit des vecteurs qui testent les bornes supérieures et inférieures et enfin une valeur intermédiaire pour chaque variable. Afin de reproduire cette méthodologie, on simule le domaine de perturbation de test. Ce programme teste aussi les conditions aux limites des différents registres présents dans le design. Afin de réaliser cette modélisation, chacune des constantes déterminant les limites est incrémentée puis décrémentée. Chaque modification correspond à un programme unique.

CNR: afin de modéliser l'écriture de données incorrectes dans les registres, cet opérateur assigne le contenu d'un registre aux autres registres de même type présent dans la spécification. Chaque modification est unique et correspond à un programme. La procédure est la suivante: chaque registre est remplacé par un scalaire du même type présent dans la spécification. Chaque registre est remplacé par une constante du même type présente dans la spécification. Afin de ne pas créer de mutant équivalent, on ne replace jamais un registre par lui même.

CSR: chaque variable ou signal est remplacée par une constante du même type.

GRP: Il a été introduit afin de tester les connexions erronnées. Les valeurs des génériques sont incrémentées. Il est intéressant aussi bien pour les designs de type structurel comme il est montré à la figure 3.4, que pour les descriptions fonctionnelles.

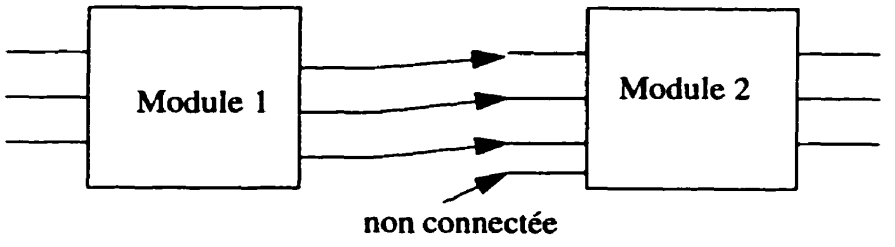


Figure 3.2: Design de type structurel

SUR: teste les valeurs signées et non-signées. Cet opérateur change la bibliothèque signé à non signé et vice versa. L'utilité de cet opérateur est apparue lors de l'utilisation d'un sous ensemble vectoriel par rapport à un vecteur plus grand. En effet, dans la validation du processeur ancillaire, nous avons utilisé des nombres signés. A titre d'exemple, le port d'entrée contenait un vecteur de N bits(data). Le N^{ieme} bit avait la valeur 0. Cependant, il n'était pas prévu qu'un sous ensemble de ce vecteur *data* soit lui aussi signé. Pour mieux comprendre l'impact d'une telle erreur, considérons le code suivant:

use IEEE.std_logic_signed.all;

avec:

data :in std_logic_vector(67 downto 0);-- bus de donnees

cur_tamp:=conv_integer(data(3 downto 0));

La fonction conv_integer permet de transformer en entier un vecteur binaire. Le résultat de simulation donné par synopsys[20] est donné à la figure 3.5

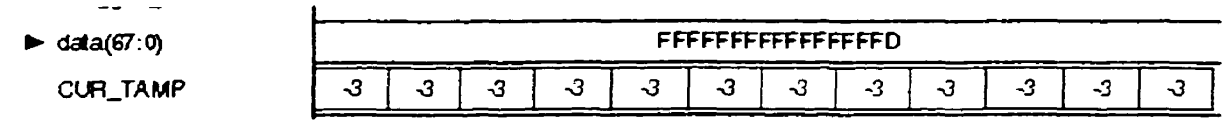


Figure 3.3: Simulation du fichier original à l'aide de Synopsys.

La valeur desirée était 13 mais en raison du fait que le sous ensemble soit signé on a obtenu la valeur -3. Suite à une mutation:

--use IEEE.std_logic_signed.all;

use IEEE.std_logic_unsigned.all; --mutation

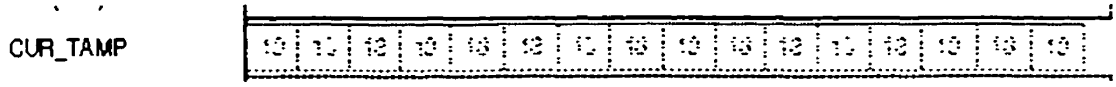


Figure 3.4: Résultat de SUR.

Cette mutation a permis d'identifier l'ambiguité sur le signe d'un sous ensemble d'un vecteur et a permis d'apporter la correction: *cur_tamp:=abs(conv_integer(data(3 downto 0)));*

VSAR: Cet opérateur permet de mettre en évidence une erreur dans l'assignation des variables et des signaux. Cet opérateur complémente les valeurs binaires des signaux. Il a été introduit afin de modéliser les mauvaises initialisations. Cela peut se traduire par une différence de séquencement dans la machine à état ou par une erreur de synchronisation dans le design. En effet, supposons qu'une séquence d'actions attend l'arrivée d'un signal de synchronisation. Si ce signal arrive trop tôt ou trop tard, on aura un mauvais fonctionnement de la machine. Le programme de mutation a été réalisé en tenant compte des particularités suivantes:

cas des signaux ou variables binaires :

On remplace un seul bit par le complément.

ex: a<='0'; devient a<='1';

ex: cet opérateur a été utilisé sur le module EDH out du processeur ancillaire. Le programme original était:

```
case parite is
        when 1=> P:='0';
        when 3 => P:='0';
        when 5 =>P:='0';
        when others =>P:='1';
end case;
```

où le signal parité, suivant sa valeur, affectait une valeur binaire à une variable P. Les resultats *qact* (*qact0, qact1, qact2)* et *qoutf* à la sortie du module EDH sont donnés à la figure 3.7.:

qact0(19:0)	00000	52000			
qact1(19:0)	00000	44000			
qact2(19:0)	00000	A2000			
qoutf(19:0)	30000	FFC00	00000	38000	00000

Figure 3.5: Simulation obtenue sans VSAR

Une mutation a été faite sur le programme initial:

```
case parite is
        when 1=> P:='1';  --mutation
        when 3 => P:='0';
        when 5 =>P:='0';
        when others =>P:='1';
end case;
```

La simulation de ce mutant a donné:

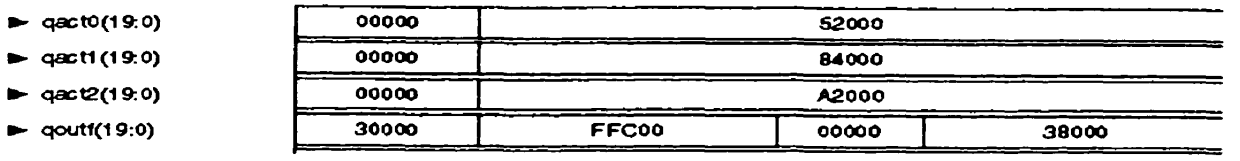


Figure 3.6: mutation avec VSAR

Bien que les valeurs dans les divers registres soient différentes, il apparaît clair que la valeur à la sortie est la même. Une étude plus poussée a montré que l'erreur produite est due à un module de sortie (multiplexeur) qui choisit une entrée indépendante des "qact". Ce qui correspond à l'hypothèse de propagation implicite du test par mutation qui n'est pas toujours vérifiée.

Dans le cas de valeurs entières

On incrémente puis décrémente la valeur de 1.

SAR: modélise l'assignation à des registres incorrects. Chacun des signaux est remplacé par tous les autres signaux de même type dans le programme. Cela se traduit, du point de vue fonctionnel, à affecter le contenu de registres à d'autres.

ex: on veut a<= data (4 downto 0) et on écrit a<= data2(4 downto 0).

cet opérateur a été testé sur le programme edh:

```
for i in 17 downto 12 loop
      qact0(i)<=r3(i-12);
end loop;
      qact0(18) <= P;
      qact0(19) <= not(P);
      qact0(11 downto 0)<="000000000000";
```

Un exemple de mutation a consisté à changer le vecteur qact0 en qact1:

```
for i in 17 downto 12 loop
        qact1(i)<=r3(i-12);
end loop;
        qact1(18) <= P;
        qact1(19) <= not(P);
        qact1(11 downto 0)<="000000000000";
```

Le résultat a montré un comportement différent. Cependant, le résultat aurait été autre si les données dans r3 avaient été toutes égales à 0 ou à 1. Le mutant aurait alors été un mutant équivalent. On peut aussi provoquer une mauvaise synchronisation dans le cas ou l'erreur se produit sur un signal de synchronisation.

SVIR: Cet opérateur modélise une mauvaise initialisation des registres ou variables

Dans le cas binaire

On complémente toutes les valeurs du vecteur binaire.

Dans le cas de valeurs entières

On incrémente puis décrémente de 1 afin de créer une légère perturbation.

SSR: Le but de ce mutant est de changer l'ordre de séquencement dans une machine à état. On remplace chacun des états présents par tous les autres états.

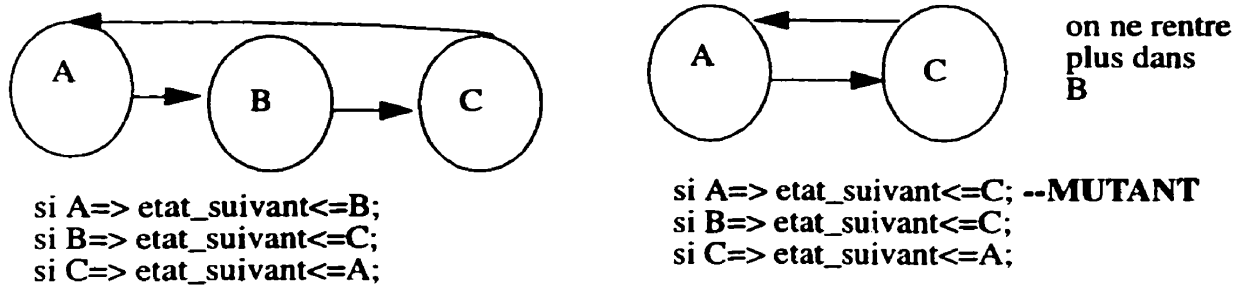


Figure 3.7: mutation avec SSR

Cependant, un problème d'espace disque peut se poser si le nombre d'états dans le programme initial est important et si le nombre de lignes de code est lui aussi important.

ASR: cet opérateur identifie les effets d'un changement de séquencement des actions dans un état et donc la dépendance des données.Cet opérateur est utilisé dans les descriptions fonctionnelles de type comportementale où les actions dans chaque état de la machine à états se réalisent de manière séquentielle. Pour ce faire, l'ordre des actions est changé dans l'état.

LCR: Chaque opérateur logique and, or, equ, not equ, xor, xnor, nand est remplacé par chacun des autres opérateurs.

COR: Change les conditions des *if* et *case* par chacune des autres conditions.

Dans le cas d'un if

if cond1 then action 1 else action 2	devient	if cond1 then action 2 else action 1

si la condition est une valeur binaire on la remplace par son complément. Dans le cas de valeures non binaires, on remplace chacun des opérateurs <, >, ==, >=, <= par chacun des autres.

Dans le cas d'un case

On remplace les valeurs des when par les autres. Cependant, afin d'éviter de donner 2 choix à l'outil de simulation on intervertit les actions. On échange donc le *when* ... de l'action 2 avec le *when* ... de l'action 1, ensuite le *when* ... de l'action 3 avec le *when* ... de l'action 1, le *when* ... de l'action 3 avec le *when* ... de l'action 2 et ainsi de suite. Cependant, aucun échange n'est effectué avec le *when others,* car le langage VHDL spécifie que cette condion doit être la dernière.

ex:

when "00" => action 0		when "00" => action 1
when "01" => action 1	devient	when "01" => action 0
when "10" => action 2		when "10" => action 2
when "11" => action 3		when "11" => action 3

3.4 Implémentation des programmes de mutation

Dans cette section, nous présentons l'implémentation des programmes typiques de mutation. Ici, on appèle opérateur l'objet dans la spécification VHDL sur lequel est mené le processus de mutation. Afin de pouvoir effectuer la mutation, chaque état dans lequel peut se trouver l'opérateur est encodé. Dans le but de faciliter l'encodage, un diagramme représentant les états de l'opérateur est déterminé. Une matrice représentant les états et les

transitions de ce diagramme est alors créée. A titre d'exemple la figure 3.10 montre les transistions d'états pour l'opérateur AOR, qui remplace chacun des opérateurs +, -, * et \ par chacun des autres. La matrice décrivant le diagramme est aussi donnée dans le tableau 3.2. Donnons à titre d'exemple une mutation survenant sur l'opérateur "-". L'état de départ étant l'état 0 du diagramme d'état, quand on rencontre l'opérateur "-" dans la spécification VHDL, on se déplace de l'état 0 à l'état 1. Ensuite, si le caractère suivant est différent de "-" alors on retourne à l'état 0 où l'on procède à la mutation. Dans le cas contraire, on est en présence d'un commentaire, auquel cas on ne procède à aucune mutation. Ce cas est codé au moyen de la matrice et est représenté à la première ligne. Si l'on se trouve à l'état 0 et la transition est 0 (représenté entre parenthèse et qui correspond à l'opérateur "-") on passe à l'état 1. Si l'on se trouve à l'état 1 et la transition est encore 0, on passe à l'état 2 où aucune mutation ne doit être efectuée. Dans le cas contraire le fait de passer de l'état 1 à l'état 0 (quelque soit le caractère et autre que "-") permet de réaliser la mutation.

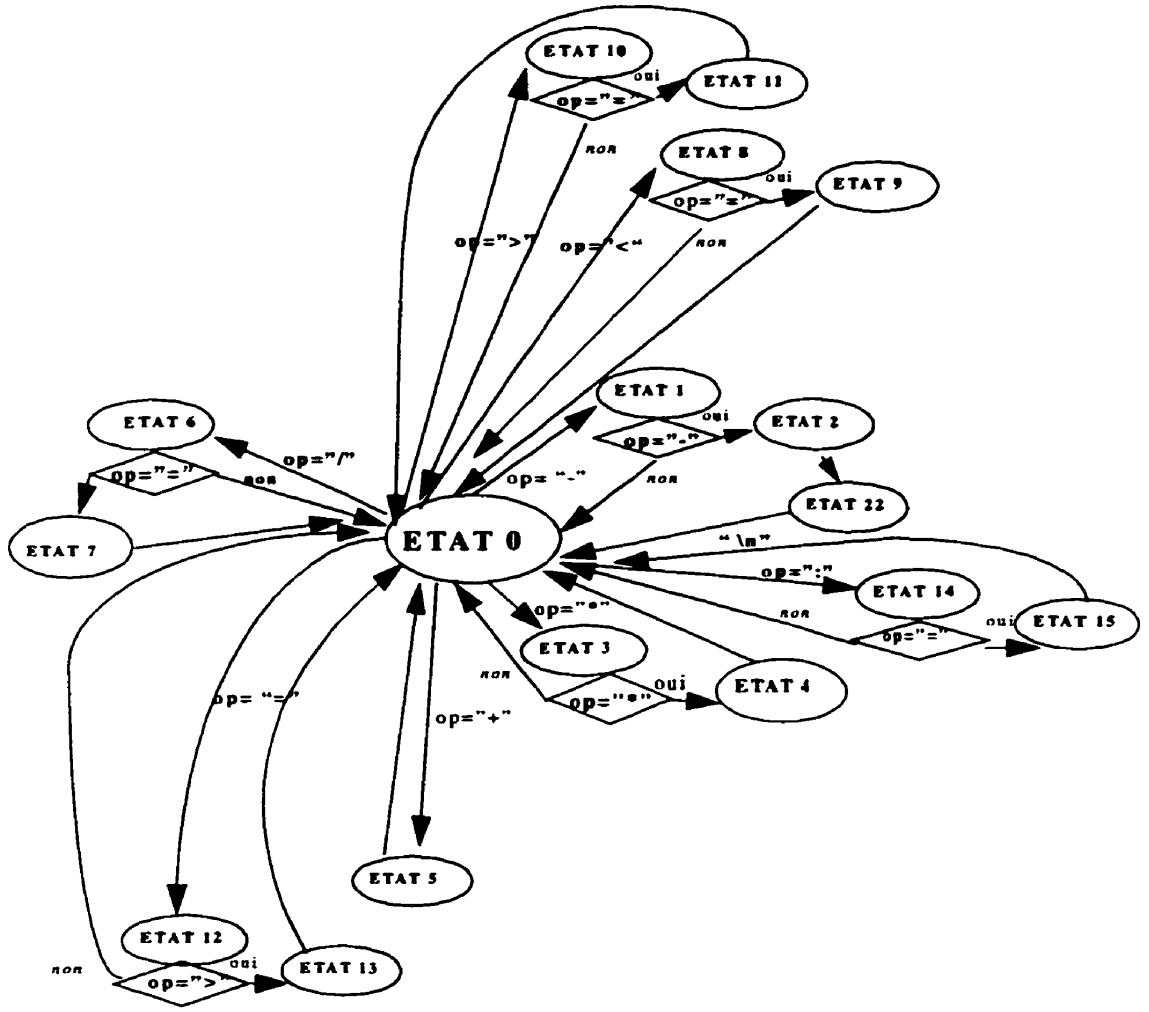


Figure 3.8: Diagramme d'état pour la mutation AOR

Tableau 3.2 : Codage des états pour la mutation AOR

états / trans	0	1	2	22	3	4	5
(0) op="-"	1	2	0	0	0	0	0
(1) op="+"	5	0	0	0	0	0	0
(2) op="*"	3	0	0	0	4	0	0
(3) op="/"	6	0	0	0	0	0	0
(4) op="<"	8	0	0	0	0	0	0
(5) op=">"	10	0	0	0	0	0	0
(6) op="="	12	2	0	0	0	0	0
(7) op=":"	14	0	0	0	0	0	0
(8) op="\n"	0	0	0	0	0	0	0

états / trans	6	7	8	9	10	11	12
(0) op="-"	0	0	0	0	0	0	0
(1) op="+"	0	0	0	0	0	0	0
(2) op="*"	0	0	0	0	0	0	0
(3) op="/"	0	0	0	0	0	0	0
(4) op="<"	0	0	0	0	0	0	0
(5) op=">"	0	0	0	0	0	0	13
(6) op="="	7	0	9	0	11	0	0
(7) op=":"	0	0	0	0	0	0	0
(8) op="\n"	0	0	0	0	0	0	0

états trans	12	13	14	15
(0) op="-"	0	0	0	0
(1) op="+"	0	0	0	0
(2) op="*"	0	0	0	0
(3) op="/"	0	0	0	0
(4) op="<"	0	0	0	0
(5) op=">"	13	0	0	0
(6) op="="	0	0	15	0
(7) op=":"	0	0	0	0
(8) op="\n"	0	0	0	0

Tous les programmes de mutation ont été décrits en utilisant cette méthodologie. Dépendant de l'opérateur rencontré dans la description comportementale, le programme de mutation approprié sera invoqué. Dans la suite, nous expliquerons sous forme de pseudo code la manière par laquelle chacun des types de mutation est effectuée.

3.4.1 Algorithmes de mutation

Cette section présente en détail les implantations des différents programmes de mutation. Ces implantations ont été réalisées en langage C et sont présentés en annexe. La première étape consiste à encoder chacun des états dans lequel peut se trouver l'opérateur afin de pouvoir entamer le processus de mutation au bon endroit. Cette matrice permet de passer de l'ancien état à l'état courant. En effet, afin que la mutation puisse s'effectuer, il faut absolument que l'état précédent et l'état courant soient représentés dans l'un des états

du diagramme d'état et en accord avec le séquencement établi par ce diagramme. Prenons l'exemple du programme de mutation AOR de la figure 3.8. La mutation ne s'effectuera dans le cas de l'opérateur "-" que si l'état courant est 0 et l'ancien état est 1. La seconde étape consiste à ouvrir le fichier sous test et à effectuer la mutation suivant l'opérateur rencontré. Le détail de ces implantations est donné ci-dessous.

CLR: Le pseudo-code est donné à la figure 3.9. La première étape consiste à encoder les états. Ensuite, si l'opérateur rencontré dans le fichier original est un entier, on crée le mutant en incrémentant la valeur limite de 1 et en recopiant le reste du fichier. Aucune autre modification n'est apportée pour ce mutant. En effet chaque mutant correspond à une panne de conception unique. Un autre mutant, c'est-à-dire une autre spécification VHDL, est créé en décrémentant la valeur limite inférieure de 1. Dans le cas de "range", un mutant supplémentaire est créé en décrementant(si différent de 0) la limite inférieure.

```
--Coder les états
int diagr_états[TRANSITION][State]={
                    /*next_state */
          /* transitions* 1,0,0,0,0,0,0,0,0,9,11,11,0,
                          0,0,0,0,0,0,0,0,0,9,11,11,0, // matrice de codage des états
                          0,0,0,0,0,0,0,0,9,9,11,11,0,
                        0,0,0,0,0,0,0,0,0,0,11,12,0};
if(( M0 = fopen("bench.vhdl" ,"r"))!=NULL)        // ouverture du fichier sous test
next_state= diagr_etat[transition][state];
switch (state)
case operator state:
While(!=EOF(bench))                          // on parcourt le fichier dans sa globalité
     {
          If (op==(integer|| boolean)) // selon que la variable à muter est un entier
            Open the mutant i      // ouverture du mutant
            Add +1 for the mutant i  // la variable est incrémentée
            Close the mutant i          // on ferme le Fichier mutant
            Sub -1 for the mutant i++ // un autre mutant est créé en décrémentant la
variable
            Close the mutant i++
         Else(op=="range") // si la variable à muter est un intervalle
          Increase the limit  // on augmente les limites inférieures et supérieures
          Put in next mutant i++ // mutants correspondant à ces nouvelles limites
          Close the next mutant i++
         Decrease the limit  // on diminue les limites inférieures et supérieures
         Put in next mutant i++
          Close the next mutant i++  }
```

Figure 3.9: Pseudo code pour la mutation CLR

CNR:

Le pseudo-code est donné à la figure 3.10. La première étape consiste à encoder les états. Ensuite, si l'opérateur rencontré dans le fichier original est un tableau on ajoute ce tableau à ceux du même type. Ensuite on créé les mutants en remplaçant le tableau d'un type par tous les autres du même type.

```
CNR: each array is replaced by all same type and dimension array in the vhdl
program.
.
Coder les états       //définition de la matrice d'états
Ouvrir la spec VHDL
While(bench!=EOF) // parcourir la spécification VHDL
{
 Si un tableau existe
{
Pour tout le fichier
  Ajouter le tableau a ceux du même type existant;
}
 If(dimension (array(i))=dimension(array(i+1))) // si 2 registres de même type
   {
    Remplacer array(i) par array(i+1); //remplacer un registre par un
    I++;                                 // autre du même type
}
}
}
```

Figure 3.10: Pseudo code pour la mutation CNR

CSR:

Le pseudo-code est donné à la figure 3.11. La première étape consiste à encoder les états. Ensuite, si l'opérateur rencontré dans le fichier original est une variable, on l'ajoute à la liste des variables du même type. Le même processus est réalisé pour toutes les constantes présentes dans la spécification VHDL. Enfin, pour toutes les constantes de type identique à ceux des variables, on crée des mutants en remplaçant les variables par des constantes. Chaque remplacement cré un mutant unique.

```
Coder les états de l'opérateur
Pour tout le Fichier
{
 If(variable)
 Mettre dans buffer_variable
End if;
If(constant)
 Mettre dans const_buffer;
End if;
Pour toutes les constant (i)
If(type(buffer_variable(i))=type(buffer_const(i+1)))
 Replacer buffer_variable(i) by buffer_const(i+1);
End if;
}
```

Figure 3.11: Pseudo code pour la mutation CSR

GRP

Chaque mutant est créé en incrémentant les valeurs génériques de un. Le processus est réitéré mais en décrémentant cette fois la valeur de un.

SUR

Le pseudo-code est donné à la figure 3.12. La première étape consiste là aussi à encoder les états. Ensuite, un mutant est créé suivant la librairie *signed* ou *unsigned*. Si la librairie est signed (ou unsigned) le mutant est créé en remplacant cette librairie par unsigned (signed). Chaque remplacement crée un mutant unique.

```
If (library signed)
 Modifier cette librairie par unsigned
Else
 Modifier par signed
End if;
```

Figure 3.12: Pseudo code pour la mutation SUR

SSR:

Le pseudo-code est donné à la figure 3.13. Après que le codage des états ait été effectué, chaque mutant est créé en effectuant une modification unique dans le sequencement de la machine à état. La modification consiste à remplacer un des états de la machine à états par un autre.

```
Coder ler états de l'opétaeur
if ( state machine==true)
 {
  mettre les états dans tamp_state;
 while(tamp_state-1!=null)
  {
    while(original( next_state)!=newstate)
   replacer next_state par new state;
  }
end if;
```

Figure 3.13: Pseudo code pour la mutation SSR

LCR:

Le pseudo-code est donné à la figure 3.14. Après que le codage des états ait été effectué chaque mutant est créé en remplacant les opérateurs logiques présents dans le programme VHDL par tous les autres opérateurs logiques xor, and, or, nand et xnor. Chaque

remplacement constitue un mutant unique. Cependant afin de ne pas produire de mutants équivalent on ne remplace jamais l'opérateur par lui même.

```
If(op==and)        // si l'opérateur est un and
{
 fopen(mutant i);      // création d'un mutant
replacer op par xor;
fermer mutant i;
replacer op par or
fermer mutant i++;
replacer op par xnor
fermer mutant i++;
replacer op par nand;
fermer mutant i++;
}
elsif(op ==xnor)  // si l'opérateur est un xnor
... .
Elsif(op==or)     // si l'opérateur est un or
...
else(nand)        // si l'opérateur est un nand
...
end;
```

Figure 3.14: Pseudo code pour la mutation LCR

VSAR:

Le pseudo-code est donné à la figure 3.15. Après que le codage des états ait été effectué, chaque mutant est créé en remplaçant les opérateurs binaires présents dans le programme VHDL par leur complément. Chaque remplacement constitue un mutant unique.

```
Coder les états des opérateurs
If(op==binary)      // si c'est un vecteur
 Trouver la  dimension
  For all dimension
   Créer mutant
    Complementer un bit
  Metrre le complement dans le mutant
 End for
Else
{
 Decrease the operator for mutant i;
Increase for mutant i+1;
}
end if;
```

Figure 3.15: Pseudo code pour la mutation VSAR

3.5 Implémentation C

Cette mise en oeuvre traduit en langage C les différents programmes de mutation donnés dans la section 3.4.1. L'ensemble des programmes est donné en appendice. Tout d'abord, le fichier d'entrée qui est le programme sous test ou plus particulièrement le programme VHDL est ouvert en écriture à partir de l'instruction *M0 = fopen("fichier-sous-test.vhdl","r")*. Ensuite une variable nommée *C* est définie afin de positionner un pointeur sur les différents caractères constituants le programme VHDL. Suivant les caractères rencontrés, le pointeur C positionnera les transitions de la matrice de codage à des valeurs définies par cette dernière. Prenons par exemple le cas de l'opérateur LCR, qui selon que le caractère soit a, n, d, o, r et x mettra la transition respectivement aux valeurs 0, 1, 2, 3, 4 et 5. Cependant, puisque le langage VHDL ne fait aucune différence entre les majuscules et les minuscules, la valeur de la transition sera la même pour une minuscule et une majuscule. Pour ce dernier cas les instructions seront de la forme:

```
switch(C)
{
        case 'A':
        case 'a': transition = 0;
      break;
}
```

Le programme VHDL est parcouru dans sa globalité par l'instruction *while ((C=fgetc(M0))!=EOF)* qui permet au pointeur C de s'incrémenter tant que la fin du fichier n'a pas été atteinte. Ensuite le couple d'instructions

```
ancien_etat=etat_courant;
etat_courant= diagr_etat_lcr[transition][etat_courant];
switch(etat_courant)
    case ....:
        switch(ancien_etat)
            case .....:
```

permet dans le cas où on se trouve dans un des diagrammes d'états définis comme dans la section 3.4 d'éffectuer la mutation. Chacun des mutants qui sont en fait d'autres programmes VHDL sont alors construit au moyen des opérations suivantes:

```
M1 =fopen("tampon_or_lcr", "w");
            compteur_lcr ++;
            sprintf(lcr_chaine1,"Mut_lcr%d,compteur_lcr);
```

Le compteur permet de calculer le nombre de mutants créés. Puisque chaque mutant correspond à une unique faute, il faut qu'un programme VHDL soit créé à chaque présence d'un état suceptible d'étre muté dans la version originale. Cela est réalisé de la manière suivante. On commence par pointer l'endroit dans le programme où la mutation est réalisée au moyen de l'instruction *pos = ftell(M0);*

On recopie dans un fichier tampon toute la fin du programme original VHDL et ceci depuis la position du pointeur donnée par la variable *pos.* Ceci est réalisé par les instructions suivantes

```
while((c=fgetc(M0))!=EOF)
                        {
                          fputc(c,M1);
                          if(c=='\r')
                            fputc('\n',M1);
                        }
                      fclose(M1);
```

La même opération est effectuée pour la copie du début du fichier au moyen des instructions données à la page suivante

```
fseek(M0,0,0);
                    while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos))
                      {
                        for(i=0;i<=4;i++)
                          {
                            fputc(c,tamp_chaine[i]);
                          }
                      }
```

L'instruction *fseek* permet de replacer le pointeur au début du fichier original VHDL, alors que les autres permettent d'écrire le début du code VHDL dans les programmes mutants identifiés par *tamp_chaine*. On procède ensuite à la mutation proprement dite. Citons par exemple dans le cas de l'opérateur SVIR où le remplacement de == par >, <, >= et /= s'effectue au moyen des instructions

```
fputs("<=",tamp_chaine[0]);
fputc('>',tamp_chaine[1]);
fputs(">=",tamp_chaine[2]);
fputs("/=",tamp_chaine[3]);
fputc('<',tamp_chaine[4]);
```

Finalement la copie de la fin du fichier qui se trouve dans le tampon mentionné précedemment est ajoutée à la suite de chacun des mutants au moyen des instructions

```
M1 = fopen("mut_tampon_cor","r");
while((c=fgetc(M1))!=EOF)
  {
    for(i=0;i<=4;i++)
      fputc(c,tamp_chaine[i]);
  }
//fin de la reecriture
```

M1 correspond au pointeur du tampon. Finalement il ne faut pas oublier qu'une mutation peut se réaliser à plusieurs endroits dans le programme sous test et que chacune de ces mutations doit résulter en un programme unique. Aussi, cela est réalisé en remettant le pointeur *C* à la position courante (*fseek(M0, pos , 0)*). Le fait que le pointeur soit remis à la position précédant la position dans chacun des *ancien_etat* permet, suite à la découverte d'un nouvel état mutant, d'ouvrir un nouveau mutant et de procéder à une nouvelle mutation. Chacun des programmes de mutation a été réalisé de la manière expliquée précedemment. La production de mutants ne demande pas beaucoup de temps. Le temps de calcul dépend de la taille du programme VHDL à tester et surtout du nombre de mutations ou de variables sur lesquels doivent s'effectuer la mutation. Une grosse contrainte vient aussi du fait que la mutation peut parfois fournir un nombre important de mutants ce qui implique un espace disque très important. Ceci peut devenir probématique si une description VHDL initiale contient 50000 lignes de code, comme c'est le cas pour la puce multiprocesseurs PULSE ou le processeur à données ancillaires.

3.6 Algorithme final de mutation

L'algorithme de la figure 3.16 présente en détail le processus de génération des mutants, ainsi que la méthodologie adoptée pour la validation de circuits numériques par mutation.

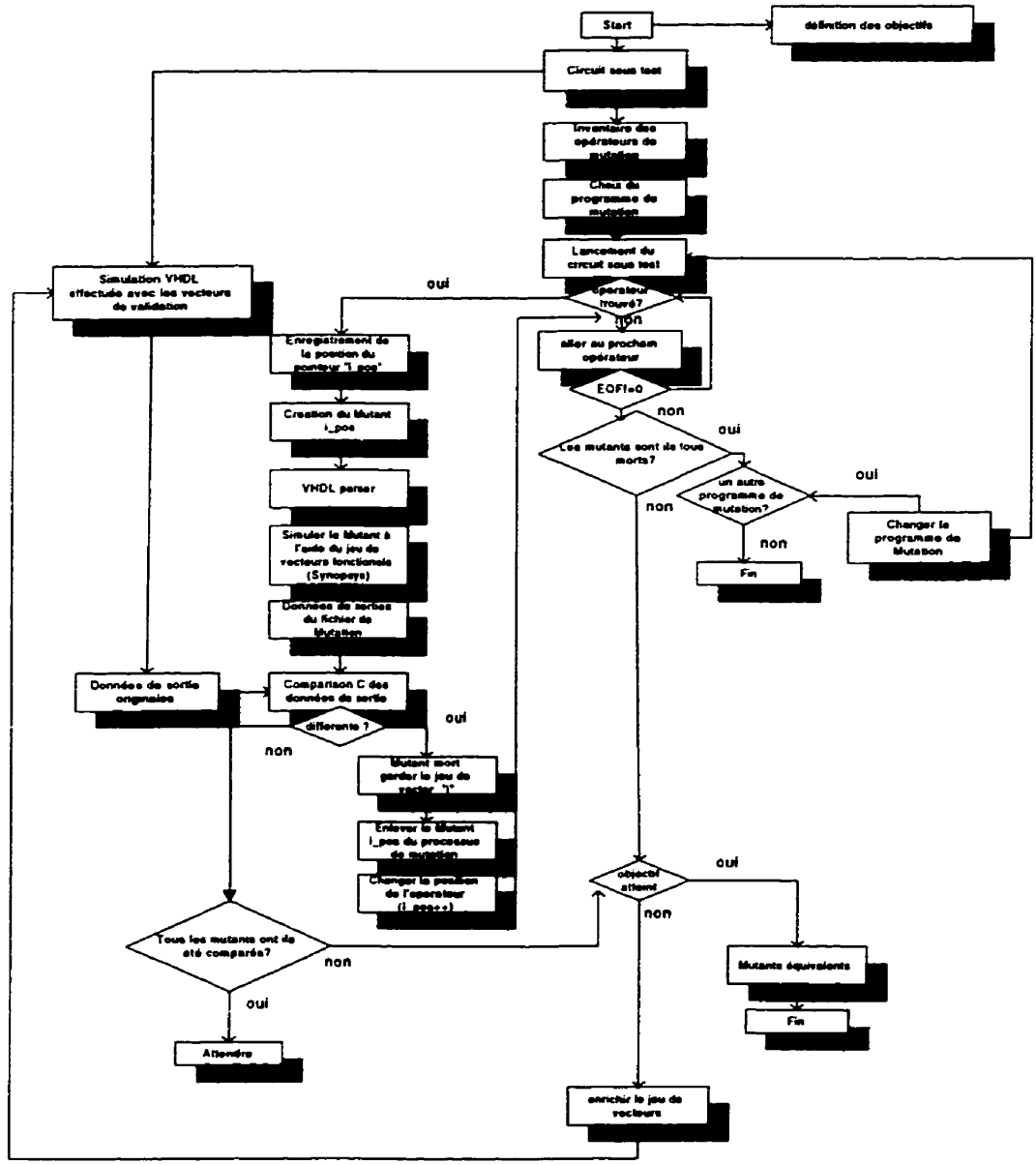


Figure 3.16: Test par mutation et génération de mutants

Tout d'abord, une définition des objectifs est effectuée. Cette définition est importante, puisqu'elle détermine le nombre de fois ou le processus de mutation est réitéré afin de faire échouer les mutants restants. La première étape consiste à simuler la version originale du programme VHDL avec les vecteurs fonctionnels fournis par le concepteur et à enregistrer la sortie dans une base de données. Cette simulation fonctionnelle est réalisée à l'aide du simulateur VSS de Synopsys[20]. La seconde étape consiste à amorcer le processsus de mutation. Un inventaire des différents opérateurs se trouvant dans la spécification du programme permet de choisir parmi les différents programmes de mutation. Après qu'un programme ait été choisi, le processus de mutation est lancé. Le pointeur de position est mis au début du programme VHDL. Dès qu'un opérateur est trouvé, la position du pointeur est enregistrée. Le Mutant spécique à cet opérateur est alors créé en transformant l'opérateur et en recopiant de manière identique le reste du programme. Afin d'être exempt de fautes syntaxiques, le mutant est vérifié à l'aide de Synopsis. Le mutant est à son tour exécuté avec le même jeu de vecteur de validation qu'à la première étape. La sortie du mutant est enregistrée dans une base de donnée des sorties mutantes. Une comparaison des bases de données de sorties mutantes et du fichier original est entreprise. Si la comparaison révèle une différence de comportement entre les deux bases, le mutant est considéré tué et est retiré du processus de validation. Il faut remarquer qu'il est aussi judicieux d'arrêter la simulation dès qu'une différence de comportement est enregistrée. Cependant cet objectif nécessiterait de concevoir notre simulateur, ce qui ne s'avérait pas nécessaire pour les petits circuits d'essai testés. Le pointeur de position est

alors incrémenté jusqu'à ce qu'un autre opérateur de mutation soit trouvé. Lorsqu'un nouvel opérateur est trouvé, le processus de génération de mutants est réinitialisé. Ce nouvel opérateur est alors muté suivant les programmes de mutation donnés dans la section 3.4. Le nouveau mutant est alors simulé avec le même jeu de vecteurs de validation et une comparaison des bases de données de sortie du nouveau fichier mutant et du fichier original est réalisée. Si la comparaison montre une différence de comportement entre ces différentes bases de données de sorties, le mutant et ses résultats de simulation sont encore effacés du processus de validation. Dans le cas contraire, le jeu de vecteurs de validation doit être augmenté. Après enrichissement du jeu de vecteurs, le fichier original est simulé à travers les nouveaux vecteurs et le résultat est gardé dans une nouvelle base de données de sortie. Les mutants non tués sont eux aussi exécutés à travers le nouveau jeu de vecteurs. Le processus de validation est alors réitéré jusqu'à ce que le but fixé au début du processus soit atteint ou qu'il n'y ait aucun mutant vivant. Il est à noter que la génération des mutants est réalisée tant que toutes la description VHDL n'a pas été couverte. Le score de mutation est alors calculé à partir de la formule donnée au paragraphe 1.2.2.4.

3.7 Résultats

Les améliorations apportés par ce nouvel algorithme en comparaison de ceux rapportés au chapitre 2 sont significatives. Une étude menée sur quatre opérateurs a permis de montrer que certains aspects négatifs de la méthode originale ont pu être

résolus. L'étude a portée sur les opérateurs UOI, COR, LCR et ABS définis dans Mothra. Les résultats de cette étude sont présentés aux tableaux 3.3.

UOI

	Mut Nb	Vect Nb	MS	Vect2 Nb	Mut2 Nb	MS2
out	260	80	45.8%	8	6	100%
Mux1	3	2	100%	2	1	100%
Mux2	6	4	83.3%	4	2	100%
Mux4	14	10	85.7%	4	4	100%
Edh	116	51	24.1%	32	67	97%

COR

	Mut Nb	Vect Nb	MS	Vect2 Nb	Mut2 Nb	MS2
out	420	19	23.3%	32	14	93%
Mux1	7	1	0%	N/A	N/A	N/A
Mux2	14	2	28.6%	N/A	N/A	N/A
Mux4	28	4	35.7%	N/A	N/A	N/A
Edh	259	34	6.2%	32	14	93%

N/A : not applicable

LCR

	Mut Nb	Vect Nb	MS	Vect2 nb	Mut2 nb	MS2
out	287	0	0%	32	14	43%
Mux4	0	N/A	N/A	N/A	0	N/A
Edh	112	0	0%	32	14	43%

N/A : not applicable

ABS

	Mut Nb	Vect Nb	MS	Vect2 Nb	Mut2 nb	MS2
Out	150	42	28.7%	N/A	N/A	N/A
Edh	48	20	16.7%	N/A	N/A	N/A

N/A : not applicable

Tableau 3.3 : Comparaison des algorithmes de validation par mutation

Le nombre de mutants MutNb, le nombre de vecteurs de validation ainsi que le score de mutation produit par Mothra, MS, sont rapportés dans ces tableaux. Des résultats

similaires obtenus avec l'algorithme présenté à la figure 3.18 sont rapportés sous les acronymes suivant Mut2Nb, Vect2Nb et MS2. Un premier résultat très important est une réduction significative du nombre de mutants. Cette réduction se traduit par une diminution de la tâche du concepteur dans le cas de mutants équivalents. Cette réduction est due à la fois au fait que la mutation s'effectue directement sur le programme écrit en VHDL et au fait qu'aucune mutation n'est entreprise sur les valeurs de l'entité. Afin de supporter ces observations, prenons l'exemple du multiplexeur de l'exemple 3.11 tiré du processeur ancillaire.

exemple 3.6.1

```
case sel1 is
when '0'=>
    qout <= in1;
when others =>
    qout <= in2;
end case;
```

```
case sel1 is
when '1'=>     --mutation
      qout <= in1;
when others =>
      qout <= in2;
end case;
```

Dans cet exemple un seul mutant a été généré avec l'algorithme présenté dans ce chapitre, comparativement à trois avec Mothra pour l'opérateur UOI. Cette différence est due au fait que la seule mutation effectuée est réalisée sur la valeur binaire de sel1. Aucune mutation n'est réalisée sur les valeurs des signaux présents dans l'entité. En effet, le but de la méthode est de valider un design aussi, nous ne touchons aucunement au contenu des signaux présent dans l'entité. Il ne faut pas oublier que ce rôle est donné aux vecteurs de validation. La complexité de Mothra est donnée par:

$$Mutnb = nb$$

est réduite à

$$Mutnb2 = nb - E + (C-1)$$

où

nb est le nombre de signaux ou variables présents dans la spécification VHDL.

E représente tous les signaux ou variables présents dans l'entité.

C représente le nombre de conditions (case or if).

Remarquons que C est en général plus petit que nb car le nombre de signaux augmente avec le nombre de conditions. Comme il a été mentionné au début de ce chapitre, la réduction du nombre de mutants est aussi due au fait qu'on ne fait plus de traduction de VHDL en FORTRAN . Dans le but de montrer l'impact de la traduction, citons par exemple le module de sortie de l'EDH qui est composé de plusieurs vecteurs binaires. Chacun des vecteurs doit être décomposé en booléen. Prenons le cas de l'exemple donné ci dessous.

exemple 3.4.2

```
case stand_video is
    when "000"
```

le vecteur est remplacé par

((std1.eq.0)and(std2.eq.0)and.(std3.eq.0)).

Sachant que dans le cas de l'opérateur COR, Mothra remplace chacune des occurences LT, LE, GT, GE, EQ, NE par toutes les autres, le nombre de mutants se trouve considérablement augmenté. Les mutants survivants après tout le processus de validation sont en grande partie dûs à une non propagation de la faute à l'une des sorties primaires. Cela résulte en un regroupement de fonctionnalitées provenant d'un module qui semble indépendant. Une grande difficulté qui a surgit a été le test du niveau de validité d'un

signal. En effet ce test produit souvent un mutant équivalent, car la fonctionnalité, si l'on fait abstraction du moment de validité, est souvent la même. Le gain amené dans la méthode d'enrichissement des vecteurs de validation est très important, car elle permet d'identifier des parties de design incomplètement testées plutôt que d'avoir recours à un générateur aléatoire de vecteurs comme il est fait dans Mothra. Le recours à ce générateur aléatoire ne nous permet pas de savoir ce que l'on a testé.

CONCLUSION

Dans ce mémoire, nous avons introduit le concept du test par vérification formelle ainsi que les méthodes traditionnelles de vérification par simulation. Nous avons montré les difficultés rencontrées lors de l'utilisation de ces deux méthodes. Bien que conceptuellement différentes, ces méthodes ont pour but commun de prouver qu'un circuit donné respecte bien les spécifications du concepteur. Nous avons montré les limitations des méthodes de simulations et des méthodes formelles.

Le premier chapitre débutait par une présentation des méthodes formelles ainsi que des outils utilisés afin de prouver l'exactitude d'un circuit. Ce chapitre montrait à la fois l'utilité de ces méthodes et leurs limitations. Ce chapitre s'est poursuivi par une présentation du test logiciel et plus particulièrement du test par mutation. Les principes du test par mutation, la génération des mutants, ainsi que la classification de ces mutants ont été exposés afin de permettre à l'usager de comprendre la méthode de validation par mutation. Finalement, une description des différents types de mutation a permis de motiver le choix de la méthode de mutation sélective.

Le deuxième chapitre a présenté une méthodologie de validation de circuits numériques utilisant le principe du test par mutation. Ce chapitre a commencé par la définition du système de mutation Mothra ainsi que la chaîne de validation. Nous avons fait ressortir des similitudes importantes entre le test par mutation et le test matériel. En effet, il est primordial qu'une justification suivie d'une propagation soit réalisée afin de détecter une

erreur de conception. Le test par mutation a tendance à produire un nombre important de mutants aussi, l'utilisation de la mutation sélective a permi de réduire ce nombre de mutant tout en gardant un score de mutation important. Il est également intéressant de constater que les vecteurs produits par le système ont pu être utilisés afin de procéder à un test matériel.

Le troisième chapitre présentait le banc d'essai qui a permis de déterminer les opérateurs de mutation. Ces opérateurs sont loin d'être exhaustifs et peuvent être augmentés. La première amélioration apportée dans ce chapitre a été la définition d'opérateurs spécifiques au VHDL et la définition d'un générateur de mutants travaillant directement sur une spécification VHDL. En effet, il a été montré que la traduction de VHDL en FORTRAN présentée dans le chapitre deux introduisait une augmentation du nombre de mutants. Par ailleurs, cette traduction ne permet pas de dire que l'on a validé la spécification initiale du concepteur, mais plutôt que l'on a validé une version en principe équivalente et non exempte d'erreurs. Bien que le choix de la mutation sélective permette de réduire de manière significative le nombre de mutants, diminuant ainsi la tâche du concepteur lors de la validation, ce nombre demeure important. Le processus d'enrichissement peut se révéler long et pénible si l'objectif est d'obtenir un score de mutation de l'ordre de 100%. En effet, un grand nombre de mutants peuvent être des mutants équivalents. Par ailleurs, il est très difficile de prouver qu'un mutant est équivalent, car il faudrait dans certains cas procéder à un test exhaustif ce qui n'est généralement pas possible. Bien que cette méthode nous paraisse intéressante vis à vis du problème de la validation, il est important de souligner les limitations qui lui interdisent

une utilisation immédiate et fréquente sur des circuits de tailles importantes. Le principale problème est la systématisation du jeu de test devant être effectué afin de tuer les mutants survivants. En effet, systématiser ce processus revient à réaliser un algorithme permettant la propagation d'erreurs à l'une des sorties. Cependant cela consisterait à connaître à l'avance le chemin que devrait prendre les données de test ce qui n'est évidemment pas faisable puisque dans le cas de la validation, on parle d'une fonctionnalité. On pourrait aussi générer un nombre important de vecteurs dans le seul but de tuer les mutants survivants. Malheureusement, le processus de test pourrait augmenter car il faudrait ensuite identifier la fonctionnalité de chacun des vecteurs efficaces. Le second problème vient de la génération du nombre de mutants. Bien que le choix des mutants soit effectué avant de lancer la génération des mutants (voir chapitre 3) et que le nombre de mutants soit considérablement réduit (tableau 3.3), il est bien évident que ce nombre demeure encore trop important. Il a été suggéré d'utiliser la mutation sélective afin d'éliminer les opérateurs de mutation responsables de la génération du plus grand nombre de mutants. Dans le chapitre 3 une alterenative a été utilisé. Plutôt que d'éliminer uniquement les opérateurs de mutation responsables du plus grand nombre de mutants, il est aussi intéressant de les choisir sélectivement en fonction de leur capacité à tester le circuit. Le nombre de mutants demeure quand même très important. Il serait donc interressant de montrer une certaine redondance chez certains opérateurs et donc de les éliminer du processus du test par mutation. Le choix des opérateurs de mutation pourrait aussi être réaliser en fonction du nombre de fois où l'état responsable de la mutation apparaît dans le code VHDL. Enfin, il est très difficile de prouver qu'un mutant est équivalent. En effet

afin de réduire le processus de validation, un grand nombre de mutants survivants sont déclarés équivalents alors qu'ils peuvent être tués manuellement. Pour conclure, il est à souligner qu'en plus de tenter d'enrichir une suite de test et donc, guarantir une meilleure validation, le test par mutation tente non seulement de mettre à jour d'éventuelles erreurs dans un circuit, mais aussi leurs abscences.

Il pourrait être intéressant dans des travaux futurs de tenter d'augmenter l'ensemble des opérateurs de mutation. En effet les opérateurs définis dans ce mémoire sont loin d'être exhaustif. L'auteur pourrait alors constituer un véritable dictionnaire de bogues. Le phénomène de redondance pourrait être aussi prouvé afin de réduire le nombre de mutants. Enfin une forme d'ATPG (automatic test pattern generator) propre à la vérification pourrait être réalisé afin de tuer les mutants survivants et réduire les "faux" mutants équivalents. On pourrait dans ce dernier cas définir un générateur aléatoire dont la plage de variation des valeurs d'entrées seraient donnée par l'utilisateur.

BIBLIOGRAPHIE

[1] P Marriot, I. Kraljic and Y. Savaria. Parallel Ultra Large Scale Engine, SIMD Architecture For Real-time Digital Signal Processing Applications. ICCD 98.

[2] Z. Boukari et P. Vado. Conception d'un simulateur C pour une puce multiprocesseurs.

[3] C. Berthet. Vérification Automatique de Circuits de Transistors VLSI. Thèse de Philosophae Doctor. Informatique et recherche opérationnelle. 1987

[4] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers, C-35(8):677-691, August 1986.

[5] E.M. Clark, E. A Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specification. ACM transactions on Programming Language and system, 8(2):244-263(April 1986).

[6] M. Fitting. First Order Logic and Automated Theorem Proving. Spinger-Verlag, 1990.

[7] Gordon, M. & Melham, T. F. (1993), Introduction to HOL: A Theorem proving environment for higher order logic, Cambridge University press.

[8] Timothy A. Budd, Richard J. Lipton, Frederick G. Sayward, and Richard A. DeMillo. The Design of a Prototype mutation System for Program Testing. In Proceedings of the National Computer Conference, pages 623-627, Anaheim, CA, June 5-8 1978. The Association for Computing Machinery, AFIPS Press, Montvale, NJ. Vol. 47.

[9] Myers G J, The Art of Software Testing, Wiley, NY, 1979.

[10] Howden W E functional program testing, IEEE Transactions Software Eng section 6(2) 162-169, 1980.

[11] R. A DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. IEEE Computer, 11(4):34-41, April 1978.

[12] W. M Craft. Detecting equivalent mutants using compiler optimization techniques. Master's thesis, Departement of Computer Science, Clemson University, Clemson SC, 1989. Technical Report 91-128.

[13] A. Jefferson Offut and William Michael Craft. Using Compiler Optimization Techniques to Detect Equivalent Mutants. Technical Report 92-102, Departement of Information and Software Sysrem Engineering, Goerge Mason University, Fairfax, VA, November 1992.

[14] R. A. DeMillo, A. J. Offut. Constraint-Based Automatic Test Data Generation. IEEE Transactions on Software Engineering, 17(9): 900-910, September 1991.

[15] D. Baldwin and Frederick G. Sayward. Heuristics for Determining Equivalence of Program mutations. Research Report 276, Departement of Computer Science, Yale University, New Haven, CT, 1979.

[16] I. Koo. mutation Testing and Three Variation.November 29, 1996.

[17] W. E Howden. Weak mutation Testing and Completeness of Test sets. IEEE Transactions of Software Engineering, vol. SE-8.NO. 4, 731-379 July 1982.

[18] A. T. Acree. mutation Analysis. Technical report GIT- ICS-79/08, School of information and Computer Science, Georgia Institute of technology, Atlanta GA, september 1979.

[19] T. A Budd. mutation Analysis of Program Test Data. PhD thesis, Yale University, New Haven CT, 1980.

[20] A. J. Offut. Investigations of the software testing coupling effect. ACM Transactions on Software Engineering Methodology, 1(1):3-18, January 1992.

[21] A.J. Offut, G. Rothermel, and C. Zapf. An Experimental Evaluation of Selective mutation. International Conference on Software Engineering, 1993. 100-107. 1993.

[22] A. P. Mathur. Performance, Effectiveness, and Reliability Issues in Software Testing. IEEE Proceedings of 15th Annual International Computer Software & Application Conference. 604-605, 1991.

[23] R.A. DeMillo and E. H Spafford, "The Mothra Software Testing Environement", Proceedings of the 11th Nasa Software Engineering Laboratory Workshop, Goddard Space Center, December 1986.

ANNEXE A

Processeur ancillaire

Description des états de la FSM,

Afin d'implanter ce système il a fallu construire une machine à état.Cette machine décrit le comportement du module EDH.

L'état initial permet l'initialisation des toutes les variables internes. Les états *lec_vidéo* permettent de détecter un paquet ancillary suivant la séquence 3ff 000 000.

L'état *eav_sav* permet d'attribuer au compteur la valeur correspondante au standard vidéo.

L'état *compter* permet a la machine de positionner son compteur de mot à la bonne adresse afin d'ecrire les information dans les trames vidéo.

Le module EDH commence ensuite à former le paquet ancillary. Les états *adf* écrient la séquence 000 3ff 3ff. Bien entendu, ces valeurs se verront augmenter à 20 bits en positionnant les bits les moins significatifs à zéro.

L'état *did* permet d'écrire le mot 1f4 (+10 bits de 0 sur les bits les moins significatifs)

L'état *dbn* écrit 200(+10 bits de 0 sur les bits les moins significatifs)

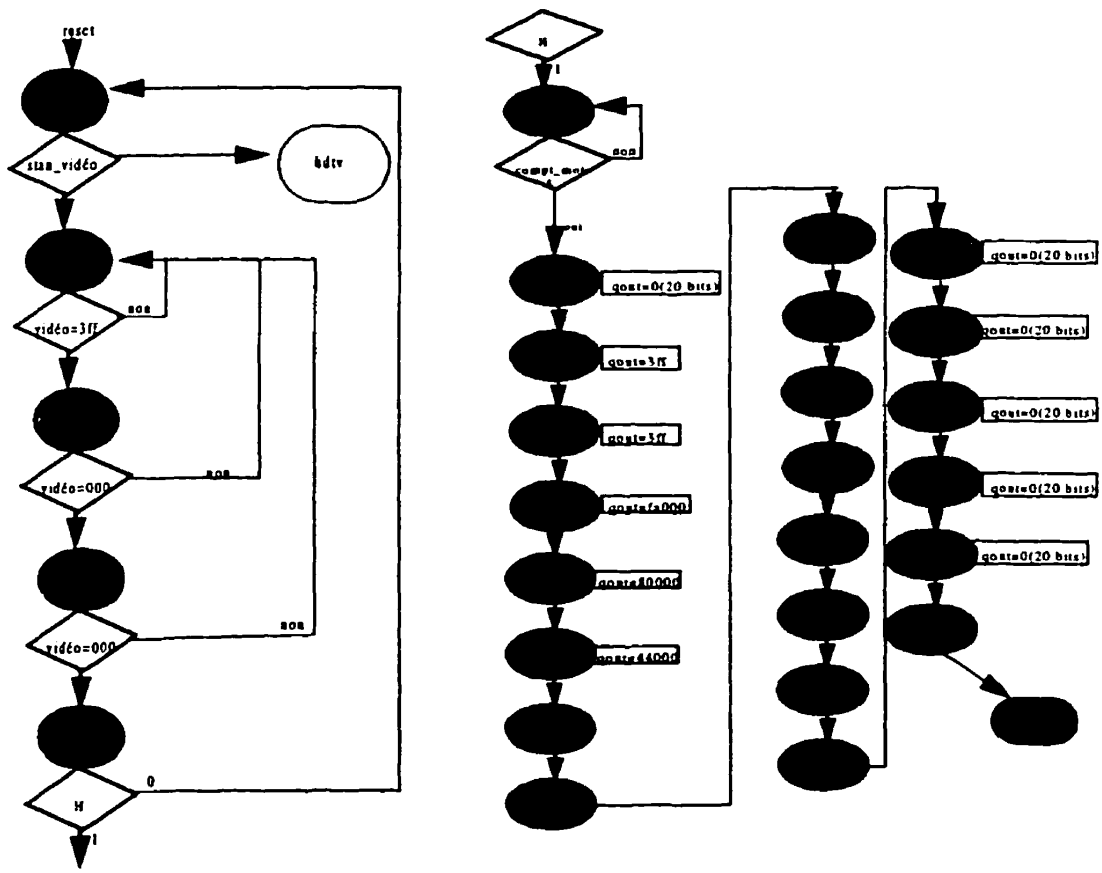
L'état *dc* écrit 100(+10 bits de 0 sur les bits les moins significatifs)

Les états *aedf* doivent écrire les mots selon la séquence suivante: not(p) p ues ida idh eda edh 0 0 (+ 10 bits de 0 sur les bits les moins significatifs). Le bit p est un bit de parité alors que les autres(ues, eda...) sont des flags qui proviennent du module EDH in.

Les états *rw* ecrivent une séquence de 0.

Enfin, un checksum des users data words est réalisé.

MACHINE A ETATS (pour la génération du paquet ancillaire)

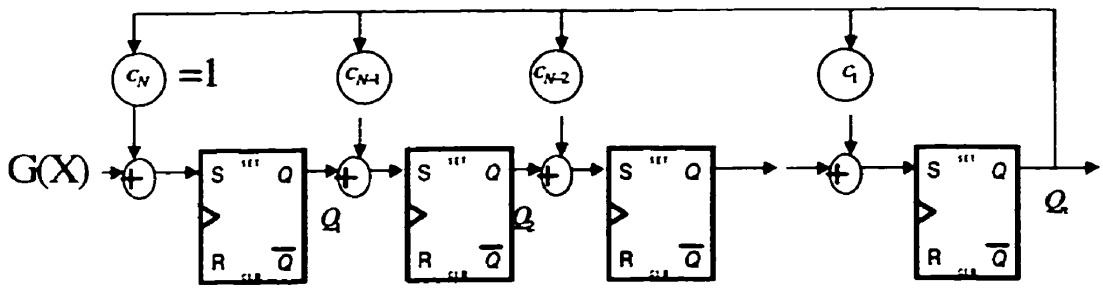


Implémentation du CRCCs

C'est une technique de compression qui effectue la division du polynome :

$$P(x) = 1 + \sum_{i=1}^{n} C_i x^{n-i}$$

Le reste de la division se trouve alors dans les registres constituants la LFSR. Cette LFSR (linear feedback shift register) un polynome d'entrée G(X) divise le polynome P(X) censé représenté la LFSR suivant le calcul: G(X)/ P(X)= Q(X)+R(X)/P(X). Le reste R(X) est alors contenu dans le registre et le quotient Q(X) est constitué par les données de sorties. Le CRCC a été implanté à l'aide du générateur polynomial CITT 16 où le shéema général est donnée ci-dessous.



Les Cn sont des connexions quand ces coefficients sont égaux à 1. Dans le cas contraire ils doivent être omis. Les + sont des portes xor.

ANNEXE B : FONCTION MID

```
        Integer Function Mid (X, Y, Z)

        integer x, y, z
1       mid = z
2       if (y .LT. z) then
3       if (x .LT. y) then
4       mid = y
5       else if (x .LT. z) then
6       mid = x
7       end if
8       else
9       if (x .GT. y) then
Δ               if (x .LT. y) then
10      mid = y
11      else if (x .GT. z) then
12      mid = x
Δ                       mid = y
13      end if
14      end if
        return
```

ANNEXE C.

//operateur de mutation pour le cas des if pour l'opérateur COR

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define TRANSITION 15
#define ETAT_COURANT 23
void main()
{
unsigned long etat_courant;
unsigned long ancien_etat;
char tata[5][20];
 long c;
long i;
long transition;
long pos;
int compteur_cor;
int pos2;
int pos3;
int pos4;
int pos5;
FILE *M0=NULL;  /*definition du pointeur file pour le fichier d'entrâe*/
FILE *M1=NULL;
FILE *tamp_chaine[5];
FILE *tampon0=NULL;
FILE *tampon1=NULL;
FILE *tampon2=NULL;
FILE *tampon3=NULL;
FILE *tampon4=NULL;

/* cette marice represente les etats suivants du diagramme if*/
int diagr_etat_IF[TRANSITION][ETAT_COURANT]={
                    /*etat_suivant */
          /* transitions*/      1, 0, 2, 5, 0, 2, 2, 9, 0, 2, 12, 2, 2, 15, 2, 2, 2, 2, 2, 19,2, 2, 0,
                                0, 2, 2, 5, 0, 2, 2, 9, 0, 2, 12, 2, 2, 15, 2, 2, 2, 2, 2, 19,2, 2, 0,
                                0,0,13,5,0,2, 2, 9, 0, 2, 12, 2, 2, 15, 2, 2, 2, 2, 2, 0,2, 2, 0,
                                0, 0, 3, 5, 0, 2, 7, 9, 0, 2, 11, 2, 2, 14, 2, 2, 2, 2, 2, 0,2, 2, 0,
                                0, 0, 2, 4, 0, 2, 2, 8, 0, 2, 12, 2, 2, 15, 2, 2, 2, 2, 2, 0,2, 2, 0,
                                0, 0, 6, 5, 0, 2, 2, 9, 0, 2, 12, 2, 2, 15, 2, 2, 2, 2, 2, 0,2, 2, 0,
                                0, 0,10, 5, 0, 2, 2, 9, 0, 2, 12, 2, 2, 15, 2, 2, 2, 2, 2, 0,2, 2, 0,
                                0, 0, 0, 5, 0, 0, 2, 9, 0, 2 ,12, 2, 2, 15, 2, 2, 2, 2, 2, 0,2, 2, 0,
                                0, 0, 2, 3, 0, 2, 2, 9, 0, 2, 12, 2, 2, 13, 2, 2, 2, 2, 19,0,2, 2, 0,
                                0, 0,16, 5, 0, 2, 2, 9, 0, 2, 12 ,2, 2, 15, 2, 2, 2, 2, 2, 0,2,22, 0,
                                0, 0, 2, 5, 0, 2, 2, 9, 0, 2, 12, 2, 2, 15, 2, 2,17, 2, 2, 0,2, 2, 0,
                                0, 0, 2, 5, 0, 2, 2, 9, 0, 2, 12, 2, 2, 15, 2, 2, 2,18, 2, 0,2, 2, 0,
                                0, 0,20, 5, 0, 2, 2, 9, 0, 2, 12, 2, 2, 15, 2, 2, 2, 2, 2, 0,2, 2, 0,
                                0, 0, 2, 5, 0, 2, 2, 9, 0, 2, 12, 2, 2, 15, 2, 2, 2, 2, 2, 0,21,2, 0,
                                0, 0, 2, 5, 0, 2, 2, 9, 0, 2, 12, 2, 2, 15, 2, 2, 2, 2, 2, 0,2, 2, 0};
```

```
etat_courant=0;
ancien_etat=0;
pos=0;
pos2=0;
pos3=0;
pos4=0;
pos5=0;
compteur_cor=0;

if(( M0 = fopen("edhii.vhdl","r"))==NULL)/* ouverture du fichier du premier mutant*/
  printf("ERREUR:le fichier source ne peut etre ouvert \n");
else
 {
  while ((c=fgetc(M0))!=EOF)
   {
    switch(c)
     {
     case 'I':
     case 'i':transition=0;
       break;
     case 'F':
     case 'f':transition=1;
       break;
     case '>':transition=2;
       break;
     case '=':transition=3;
       break;
     case '\':transition=4;
       break;
     case '/':transition=5;
       break;
     case '<':transition= 6;
       break;
     case ';':transition= 7;
       break;
     case ' ':transition= 8;
       break;
     case 'E':
     case 'e':transition=9;
       break;
     case 'N':
     case 'n':transition=10;
       break;
        case 'D':
     case 'd':transition=11;
       break;
      case 'T':
```

```
        case 't':transition=12;
          break;
        case 'H':
        case 'h':transition=13;
          break;
       default:transition=14;
          break;
        }
      ancien_etat=etat_courant;
   etat_courant= diagr_etat_IF[transition][etat_courant];

switch(etat_courant)
{
case 2:pos =ftell(M0);
break;
case 5:switch (ancien_etat)
{
     case 3:
           M1 = fopen("mut_tampon_cor","w");
           for(i=0;i<=4;i++)
            {
             sprintf(tata[i],"Mut_cor%d.vhdl",compteur_cor);
             compteur_cor++;
             tamp_chaine[i] = fopen(tata[i],"w");  //ouverture des 5 mutants cor
            }
           fputc(c,M1);
           // ecriture de la fin du fichier
           while((c=fgetc(M0))!=EOF)
            {
             fputc(c,M1);
             if(c=='\r')
              fputc('\n',M1);
            }
           fclose(M1);
           //ajout du debut du fichier dans les fichiers mutants
           fseek(M0,0,0);
           while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos))
            {
             for(i=0;i<=4;i++)
              {
               fputc(c,tamp_chaine[i]);
              }
            }
           //fin d'ajout
           fputs("<=",tamp_chaine[0]);
           fputc('>',tamp_chaine[1]);
           fputs(">=",tamp_chaine[2]);
           fputs("/=",tamp_chaine[3]);
           fputc('<',tamp_chaine[4]);
           //on reecrit la fin du fichier
           M1 = fopen("mut_tampon_cor","r");
```

```
            while((c=fgetc(M1))!=EOF)
              {
               for(i=0;i<=4;i++)
                fputc(c,tamp_chaine[i]);
              }
            //fin de la reecriture
            for(i=0;i<=4;i++)
             fclose (tamp_chaine[i]);
            fclose(M1);
            fseek(M0,pos,0); //on remet a la position courante
            break;
                default:
                 break;
}
     break;
case 7:pos2=ftell(M0);
       break;

case 9:switch (ancien_etat)
       {
       case 7:
        M1 = fopen("mut_tampon_cor","w");
        for(i=0;i<=4;i++)
          {
           sprintf(tata[i],"Mut_cor%d.vhdl",compteur_cor);
           compteur_cor++;
           tamp_chaine[i] = fopen(tata[i],"w");  //ouverture des 5 mutants cor
          }
        fputc(c,M1);
        // ecriture de la fin du fichier
        while((c=fgetc(M0))!=EOF)
         {
          fputc(c,M1);
          if(c=='\r')
           fputc('\n',M1);
         }
        fclose(M1);

        //ajout du debut du fichier dans les fichiers mutants
        fseek(M0,0,0);
        while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos))
         {
          for(i=0;i<=4;i++)
           {
            fputc(c,tamp_chaine[i]);
           }
         }

        //fin d'ajout
        fputc('=',tamp_chaine[0]);
        fputc('>',tamp_chaine[1]);
```

```
          fputs(">=",tamp_chaine[2]);
          fputs("<=",tamp_chaine[3]);
          fputc('<',tamp_chaine[4]);


          //on reecrit la fin du fichier
          M1 = fopen("mut_tampon_cor","r");
          while((c=fgetc(M1))!=EOF)
           {
            for(i=0;i<=4;i++)
             {
              fputc(c,tamp_chaine[i]);
             }
           }
          //fin de la reecriture
          for(i=0;i<=4;i++)
           fclose (tamp_chaine[i]);
          fclose(M1);
          fseek(M0,pos2,0); //on remet a la position courante
          break;
         default:
          break;
         }
       break;
case 10: pos3=ftell(M0);
        break;
case 11:switch (ancien_etat)
        {
        case 10:
         M1 = fopen("mut_tampon_cor","w");
         for(i=0;i<=4;i++)
          {
           sprintf(tata[i],"Mut_cor%d.vhdl",compteur_cor);
           compteur_cor++;
           tamp_chaine[i] = fopen(tata[i],"w");  //ouverture des 5 mutants cor
          }
         //  fputc(c,M1);

         // ecriture de la fin du fichier
         while((c=fgetc(M0))!=EOF)
          {
           fputc(c,M1);
           if(c=='\r')
            fputc('\n',M1);
            }
           fclose(M1);

           //ajout du debut du fichier dans les fichiers mutants
           fseek(M0,0,0);
           while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos))
            for(i=0;i<=4;i++)
```

```
                {
                 fputc(c,tamp_chaine[i]);
                }

              //fin d'ajout
              fputc('=',tamp_chaine[0]);
              fputc('>',tamp_chaine[1]);
              fputs(">=",tamp_chaine[2]);
              fputs("/=",tamp_chaine[3]);
              fputs("<",tamp_chaine[4]);


              //on reecrit la fin du fichier
              M1 = fopen("mut_tampon_cor","r");
              while((c=fgetc(M1))!=EOF)
               {
                for(i=0;i<=4;i++)
                 fputc(c,tamp_chaine[i]);
               }
              //fin de la reecriture
              for(i=0;i<=4;i++)
               fclose (tamp_chaine[i]);
              fclose(M1);
              fseek(M0,pos3,0); //on remet a la position courante
              break;
            default:
             break;
            }
           break;
case 12:switch (ancien_etat)
           {
           case 10:
            M1 = fopen("mut_tampon_cor","w+");
            for(i=0;i<=4;i++)
             {
              sprintf(tata[i],"Mut_cor%d.vhdl",compteur_cor);
              compteur_cor++;
              tamp_chaine[i] = fopen(tata[i],"w");  //ouverture des 5 mutants cor
             }
            fputc(c,M1);

            // ecriture de la fin du fichier
            while((c=fgetc(M0))!=EOF)
             {
              fputc(c,M1);
              if(c=='\r')
               fputc('\n',M1);
             }
            fclose(M1);

            //ajout du debut du fichier dans les fichiers mutants
```

```
        fseek(M0,0,0);
        while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos))
         for(i=0;i<=4;i++)
           {
            fputc(c,tamp_chaine[i]);
           }

        //fin d'ajout
        fputc('=',tamp_chaine[0]);
        fputc('>',tamp_chaine[1]);
        fputs(">=",tamp_chaine[2]);
        fputs("/=",tamp_chaine[3]);
        fputs("<=",tamp_chaine[4]);


        //on reecrit la fin du fichier
        M1 = fopen("mut_tampon_cor","r");
        while((c=fgetc(M1))!=EOF)
         {
          for(i=0;i<=4;i++)
            fputc(c,tamp_chaine[i]);
         }
        //fin de la reecriture
        for(i=0;i<=4;i++)
         fclose (tamp_chaine[i]);
        fclose(M1);
        fseek(M0,pos3,0); //on remet a la position courante
        break;
       default:
        break;
       }
      break;


case 14: pos4=ftell(M0); ;
       switch (ancien_etat)
        {
        case 13:
         M1 = fopen("mut_tampon_cor","w+");
          for(i=0;i<=4;i++)
           {
            sprintf(tata[i],"Mut_cor%d.vhdl",compteur_cor);
            compteur_cor++;
            tamp_chaine[i] = fopen(tata[i],"w");  //ouverture des 5 mutants cor
           }
          fputc(c,M1);

          // ecriture de la fin du fichier
          while((c=fgetc(M0))!=EOF)
           {
            fputc(c,M1);
```

```
            if(c=='\r')
              fputc('\n',M1);
            }
          fclose(M1);

          //ajout du debut du fichier dans les fichiers mutants
          fseek(M0,0,0);
          while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos))
           for(i=0;i<=4;i++)
            {
             fputc(c,tamp_chaine[i]);
            }

          //fin d'ajout
          fputc('=',tamp_chaine[0]);
          fputc('>',tamp_chaine[1]);
          fputs("<=",tamp_chaine[2]);
          fputs("/=",tamp_chaine[3]);
          fputc('<',tamp_chaine[4]);


          //on reecrit la fin du fichier
          M1 = fopen("mut_tampon_cor","r");
          while((c=fgetc(M1))!=EOF)
           {
            for(i=0;i<=4;i++)
              fputc(c,tamp_chaine[i]);
           }
          //fin de la reecriture
for(i=0;i<=4;i++)
           fclose (tamp_chaine[i]);
          fclose(M1);
          fseek(M0,pos4,0); //on remet a la position courante
          break;
        default:
         break;
        }
      break;

    case 15:pos5=ftell(M0);
     switch (ancien_etat)
      {
      case 13:
       M1 = fopen("mut_tampon_cor","w+");

        for(i=0;i<=4;i++)
         {
          sprintf(tata[i],"Mut_cor%d.vhdl",compteur_cor);
          compteur_cor++;
          tamp_chaine[i] = fopen(tata[i],"w");  //ouverture des 5 mutants cor
         }
```

```
                fputc(c,M1);

                // ecriture de la fin du fichier
                while((c=fgetc(M0))!=EOF)
                  {
                   fputc(c,M1);
                   if(c=='\r')
                     fputc('\n',M1);
                  }
                fclose(M1);

                //ajout du debut du fichier dans les fichiers mutants
                fseek(M0,0,0);
                while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos))
                 for(i=0;i<=4;i++)
                   {
                    fputc(c,tamp_chaine[i]);
                   }

                //fin d'ajout
                fputc('=',tamp_chaine[0]);
                fputs("<=",tamp_chaine[1]);
                fputs(">=",tamp_chaine[2]);
                fputs("/=",tamp_chaine[3]);
                fputc('<',tamp_chaine[4]);


                //on reecrit la fin du fichier
                M1 = fopen("mut_tampon_cor","r");
                while((c=fgetc(M1))!=EOF)
                 {
                   for(i=0;i<=4;i++)
                     fputc(c,tamp_chaine[i]);
                 }
                //fin de la reecriture
                for(i=0;i<=4;i++)
                  fclose (tamp_chaine[i]);
                fclose(M1);
                fseek(M0,pos5,0); //on remet a la position courante
                break;
              default:
                break;
              }
            break;

          default:
            break;
          }
        }
      }
    }
```

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define TRANSITION 13
#define ETAT_COURANT 20

//operateur de mutation pour le cas des if


void main()
{
unsigned long etat_courant;
unsigned long ancien_etat;
char tata[5][20];

long c;
long i;
long transition;
long pos;
int compteur_cor;


FILE *M0=NULL;  /*definition du pointeur file pour le fichier d'entrâe*/
FILE *M1=NULL;
FILE *tamp_chaine[5];

FILE *tampon0=NULL;
FILE *tampon1=NULL;
FILE *tampon2=NULL;
FILE *tampon3=NULL;
FILE *tampon4=NULL;

int diagr_etat_IF[TRANSITION][ETAT_COURANT]={
                    /*etat_suivant */
          /* transitions*/1, 0, 2, 5, 0, 0, 0, 9, 0, 2, 12, 0, 0, 15, 0, 0, 2, 2, 2, 0,
                    0, 2, 2, 5, 0, 0, 0, 9, 0, 2, 12, 0, 0, 15, 0, 0, 2, 2, 2, 0,
                     0, 0,13, 5, 0, 0, 0, 9, 0, 2, 12, 0, 0, 15, 0, 0, 2, 2, 2, 0/* cette marice represente les etats
suivants du diagramme if*/
                    0, 0, 3, 5, 0, 0, 7, 9, 0, 2, 11, 0, 0, 14, 0, 0, 2, 2, 2, 0,
                    0, 0, 2, 4, 0, 0, 0, 8, 0, 2, 12, 0, 0, 15, 0, 0, 2, 2, 2, 0,
                    0, 0, 6, 5, 0, 0, 0, 9, 0, 2, 12, 0, 0, 15, 0, 0, 2, 2, 2, 0,
                    0, 0,10, 5, 0, 0, 0, 9, 0, 2, 12, 0, 0, 15, 0, 0, 2, 2, 2, 0,
                    0, 0, 2, 0, 0, 0, 0, 0, 0, 2 , 0 ,0, 0,  0, 0, 0, 2, 0, 2, 0,
                    0, 0, 2, 3, 0, 0, 0, 7, 0, 2,  0, 0, 0, 13, 0, 0, 2, 2, 19,0,
                    0, 0,16, 5, 0, 0, 2, 9, 0, 2, 12 ,0, 0, 15, 0, 0, 2, 2, 2, 0,
                    0, 0, 2, 5, 0, 0, 2, 9, 0, 2, 12, 0, 0, 15, 0, 0,17, 2, 2, 0,
                    0, 0, 2, 5, 0, 0, 2, 9, 0, 2, 12, 0, 0, 15, 0, 0, 2,18, 2, 0,
                    0, 0, 2, 5, 0, 0, 2, 9, 0, 2, 12, 0, 0, 15, 0, 0, 2, 2, 2, 0};
```

//operateur de mutation pour le cas des if pour l'opérateur COR (si nombre non binaire)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define TRANSITION 13
#define ETAT_COURANT 20
void main()
{
unsigned long etat_courant;
unsigned long ancien_etat;
char tata[5][20];

long c;
long i;
long transition;
long pos;
int compteur_cor;
FILE *M0=NULL;  /*definition du pointeur file pour le fichier d'entrée*/
FILE *M1=NULL;
FILE *tamp_chaine[5];
FILE *tampon0=NULL;
FILE *tampon1=NULL;
FILE *tampon2=NULL;
FILE *tampon3=NULL;
FILE *tampon4=NULL;

int diagr_etat_IF[TRANSITION][ETAT_COURANT]={
                        /*etat_suivant */
          /* transitions*/      1, 0, 2, 5, 0, 0, 0, 9, 0, 2, 12, 0, 0, 15, 0, 0, 2, 2, 2, 0,
                                0, 2, 2, 5, 0, 0, 0, 9, 0, 2, 12, 0, 0, 15, 0, 0, 2, 2, 2, 0,
                                0, 0,13, 5, 0, 0, 0, 9, 0, 2, 12, 0, 0, 15, 0, 0, 2, 2, 2, 0,
                                0, 0, 3, 5, 0, 0, 7, 9, 0, 2, 11, 0, 0, 14, 0, 0, 2, 2, 2, 0,
                                0, 0, 2, 4, 0, 0, 0, 8, 0, 2, 12, 0, 0, 15, 0, 0, 2, 2, 2, 0,
                                0, 0, 6, 5, 0, 0, 0, 9, 0, 2, 12, 0, 0, 15, 0, 0, 2, 2, 2, 0,
                                0, 0,10, 5, 0, 0, 0, 9, 0, 2, 12, 0, 0, 15, 0, 0, 2, 2, 2, 0,
                                0, 0, 2, 0, 0, 0, 0, 0, 0, 2 , 0 ,0, 0,  0, 0, 0, 2, 0, 2, 0,
                                0, 0, 2, 3, 0, 0, 0, 7, 0, 2,  0, 0, 0, 13, 0, 0, 2, 2, 19,0,
                                0, 0,16, 5, 0, 0, 2, 9, 0, 2, 12 ,0, 0, 15, 0, 0, 2, 2, 2, 0,
                                0, 0, 2, 5, 0, 0, 2, 9, 0, 2, 12, 0, 0, 15, 0, 0,17, 2, 2, 0,
                                0, 0, 2, 5, 0, 0, 2, 9, 0, 2, 12, 0, 0, 15, 0, 0, 2,18, 2, 0,
                                0, 0, 2, 5, 0, 0, 2, 9, 0, 2, 12, 0, 0, 15, 0, 0, 2, 2, 2, 0};

etat_courant=0;
ancien_etat=0;
pos=0;
compteur_cor=0;

 if(( M0 = fopen("text.vhdl","r"))==NULL)/* ouverture du fichier du premier mutant*/
   printf("ERREUR:le fichier source ne peut etre ouvert \n");
 else
```

```
{
 while ((c=fgetc(M0))!=EOF)
  {
   switch(c)
    {
    case 'I':
    case 'i':transition=0;
     break;
    case 'F':
    case 'f':transition=1;
     break;
    case '>':transition=2;
     break;
    case '=':transition=3;
     break;
    case '\':transition=4;
     break;
    case '/':transition=5;
     break;
    case '<':transition= 6;
     break;
    case ';':transition= 7;
     break;
    case ' ':transition= 8;
     break;
    case 'E':
    case 'e':transition=9;
     break;
    case 'N':
    case 'n':transition=10;
     break;
    case 'D':
    case 'd':transition=11;
     break;
    default:transition=12;
     break;
    }
   ancien_etat=etat_courant;
etat_courant= diagr_etat_IF[transition][etat_courant];

      switch(etat_courant)
       {

          case 9:switch (ancien_etat)
       {
case 7:
        M1 = fopen("mut_tampon_cor","w+");
        pos =ftell(M0);
        for(i=0;i<=4;i++)
         {   printf("comp %d \n",compteur_cor);
          sprintf(tata[i],"Mut_cor%d.vhdl",compteur_cor);
```

```
                compteur_cor++;
                tamp_chaine[i] = fopen(tata[i],"w");  //ouverture des 5 mutants cor
               }
             fputc(c,M1);

             // ecriture de la fin du fichier
             while((c=fgetc(M0))!=EOF)
              {
               fputc(c,M1);
               if(c=='\r')
                fputc('\n',M1);
              }
             fclose(M1);

             //ajout du debut du fichier dans les fichiers mutants
             fseek(M0,0,0);
            while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos-3)) //-3 afin de ne pas reecrire le meme caractere
              for(i=0;i<=4;i++)
               {
                fputc(c,tamp_chaine[i]);
               }

             //fin d'ajout
           fputc('=',tamp_chaine[0]);
             fputc('>',tamp_chaine[1]);
             fputs(">=",tamp_chaine[2]);
             fputs("<=",tamp_chaine[3]);
             fputc('<',tamp_chaine[4]);

            //on reecrit la fin du fichier
             M1 = fopen("mut_tampon_cor","r");
             while((c=fgetc(M1))!=EOF)
              {
               for(i=0;i<=4;i++)
                fputc(c,tamp_chaine[i]);
              }
             //fin de la reecriture
              for(i=0;i<=4;i++)
               fclose (tamp_chaine[i]);
             fclose(M1);
             fseek(M0,pos,0); //on remet a la position courante
             break;
default:
break;
}
 default:
       break;
          }
         }
    }
}
```

```
//operateur de mutation pour le cas des case  pour l'opérateur COR  et SSR
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define TRANSITION 11
#define ETAT_COURANT 15
#define TRANSITION1 7
#define ETAT_COURANT1 8
void main()
{
unsigned char etat_courant;
unsigned char etat_courant1;
char cor_chaine[10];
char tabcase[120][12];
unsigned pos[120];
unsigned i_max[120];
int c;
int flag;
int i;
int j_max;
int j;
int posf;
int i_tab;
int i_tab2;
int transition;
int transition1;
long compteur_cor;
long posM2;//position du pointeur
long postampon;
long i_pos;
long i_pos2;
long i_posmax;

FILE *M0=NULL;  /*definition du pointeur file pour le fichier d'entrÈe*/
FILE *M1=NULL;
FILE *tampon;
int diagr_etat_case[TRANSITION][ETAT_COURANT]={
            /*etat_suivant */
/* transitions*/                1,0,0,0,5,5,0,0,0,0,0,0,0,0,0,
                                0,2,0,0,5,5,0,0,0,0,0,0,0,0,0,
                                6,0,3,0,5,5,0,0,0,0,0,0,13,0,0,
                                0,0,0,4,5,5,7,0,0,0,0,0,0,0,0,
                                0,0,0,0,4,14,0,0,9,0,0,0,0,0,0,
                                0,0,0,0,5,5,0,8,0,0,0,0,0,0,0,
                                0,0,0,0,5,5,0,0,0,10,0,0,0,0,0,
                                0,0,0,0,5,5,0,0,0,0,11,0,0,0,0,
                                0,0,0,0,5,5,0,0,0,0,0,12,0,0,0,
                                0,0,0,0,5,14,0,0,0,0,0,0,0,0,0,
                                0,0,0,0,5,5,0,0,0,0,0,0,0,0,0};
```

```
int diagr_etat_pos[TRANSITION1][ETAT_COURANT1]={
            /*etat_suivant */
/* transitions*/                1,0,0,0,5,5,0,0,
                                0,2,0,0,5,5,0,0,
                                0,0,3,0,5,5,0,0,
                                0,0,0,4,5,5,0,0,
                                0,0,0,0,5,6,0,0,
                                0,0,0,0,5,5,7,0,
                                0,0,0,0,5,5,0,0};

etat_courant=0;
etat_courant1=0;
i=0;
i_posmax=0;
j_max=0;
j=0;
i_pos=0;
i_tab=0;
i_tab2=0;
compteur_cor=0;
posM2=0;
postampon=0;
posf=0;
 if(( M0 = fopen("mux2.vhdl" ,"r"))==NULL)/* ouverture du fichier du premier mutant*/
   printf("ERREUR:le fichier source ne peut etre ouvert \n");
 else{
   printf("fichier mux2.vhdl ouvert \n");
   while ((c=fgetc(M0))!=EOF){
    switch(c)
{
case 'W':
case 'w':transition=0;
break;
case 'H':
case 'h':transition=1;
break;
case 'E':
case 'e':transition=2;
break;
case 'N':
case 'n':transition=3;
break;
case ' ':transition=4;
break;
case 'D':
case 'd':transition=5;
break;
case 'C':
case 'c':transition=6;
break;
case 'A':
```

```
case 'a':transition=7;
break;
case 'S':
case 's':transition=8;
break;
case '=':transition=9;
break;
default:transition=10;
break;
}
etat_courant= diagr_etat_case[transition][etat_courant];
switch(etat_courant)
{
      case 0:
        i=0;
        break;
      case 1:
      case 2:
      case 3:
      case 4:
        break;
      case 5:
        tabcase[j][i]=c;
        i_max[j]=i;
        i++;
        break;
      case 6:
      case 7:
      case 8:
      case 9:
      case 10:
      case 11:
      case 12:
        break;
      case 13:
        for(i=0;i<j_max-1;i++)
          {
          flag=0;
           i_tab++;
              sprintf(cor_chaine,"Mut_mux%d.vhdl",compteur_cor);
              compteur_cor++;
              M1 = fopen("mut_tampon","w");
              tampon = fopen(cor_chaine,"w");
              if(i_tab<=j_max)
                fputs(tabcase[i_tab],M1);
              fseek(M0,pos[i],0);
                while((c=fgetc(M0))!=EOF)
                {
                   switch(c)
                    {
                    case 'W':
```

```
 case 'w':transition1=0;
  break;
 case 'H':
 case 'h':transition1=1;
  break;
 case 'E':
 case 'e':transition1=2;
  break;
 case 'N':
 case 'n':transition1=3;
  break;
 case '=':transition1=4;
  break;
 case '>':transition1=5;
  break;
default:transition1=6;
  break;
 }

etat_courant1= diagr_etat_pos[transition1][etat_courant1];
switch(etat_courant1)
 {
 case 0:
 case 1:
 case 2:
 case 3:fputc(c,M1);
  break;
 case 4:
  fputc(c,M1);
  fputc(' ',M1);
  if (flag<=1)
    {
     posf=ftell(M0);
     i_tab2=i_tab-1;
     fputs(tabcase[i],M1);
     fputc(' ',M1);
     flag++;
    }
  break;
 case 5:
 case 6:
  break;
 case 7:
  fputs("=>",M1);
// ajout de la suite de la fin du texte
  while((c=fgetc(M0))!=EOF)
    {
     fputc(c,M1);
     if(c=='\r')
       fputc('\n',M1);
  // fin ajout
```

```
                    }
                    fclose(M1);
                   break;
                  default:
                   break;
                  }
                 flag=0;

              }
                      //ajout du debut du texte avant la mutation
               fseek(M0,0,0);
               while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos[i]-i_max[i]-2)) // -2 a cause de =>
                {
                 fputc(c,tampon);
                 if(c=='\r')
                  fputc('\n',tampon);
                }
               fclose(tampon);
               //fin d'ajout

              //on reecrit la fin du fichier
               tampon = fopen(cor_chaine,"a");
               M1 = fopen("mut_tampon","r");
               while((c=fgetc(M1))!=EOF)
                {
                 fputc(c,tampon);
                }
               //  fin de la reecriture

               fclose (tampon);
               fclose(M1);
          }
        fclose(M0);
        fclose(M1);
        break;
      case 14:
        pos[i_pos]=ftell(M0);
        i_pos++;
        j_max=j;
        j++;
        break;
       default:
        break;
}
   }
 }
fclose(M0);
}
```

```
//operateur de mutation AOR

//fonction general permettant le changement des opÈrateurs arithmetiques +, /, *, -, >, >= et <

#include <stdio.h>
#define TRANSITION 9
#define ETAT_COURANT 17
main()
{
unsigned char etat_courant;
unsigned char ancien_etat;
unsigned char tata[4][10];
int c,compt;
int pos;
int compteur_aor;
int i;
int transition;
 FILE *M0=NULL;  /*definition du pointeur file pour le fichier d'entrÈe*/
 FILE *M1=NULL;
 FILE *M2=NULL;
 FILE *M3=NULL;
 FILE *M4=NULL;
FILE *tamp_chaine[4];

 int diagr_etat[TRANSITION][ETAT_COURANT]={
            /*etat_suivant */
/* transitions*/                  1,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                                  5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                                  3,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,
                                  6,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                                  8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                                  10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                                  12,2,0,0,0,0,0,7,9,0,0,0,0,0,0,15,0,
                                  14,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                                  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

etat_courant=0;
compt=0;
pos=0;
compteur_aor=0;

 if(( M0 = fopen("test.txt" ,"r"))==NULL)/* ouverture du fichier du premier mutant*/
   printf("ERREUR:le fichier source ne peut etre ouvert \n");
 else{
   printf("fichier ouvert \n");

while ((c=fgetc(M0))!=EOF){
   switch(c)
{
   case '-':transition=0;
   break;
```

```
  case '+':transition=1;
   break;
  case '*':transition=2;
   break;
  case '/':transition=3;
   break;
  case '<':transition=4;
   break;
  case '>':transition=5;
   break;
  case '=':transition=6;
   break;
  case ':':transition=7;
   break;
  case '\0':
   break;
  default:transition=8;
   break;
  }
ancien_etat=etat_courant;
etat_courant= diagr_etat[transition][etat_courant];


 switch(etat_courant)
 {
 case 0: switch (ancien_etat)
        {
   case 1:
         M1 = fopen("mut_tampon_aor","w");
         fputc(c,M1);
         pos =ftell(M0);
         for(i=0;i<=2;i++)
          {
           sprintf(tata[i],"Mut_aor%d.vhdl",compteur_aor);
           compteur_aor++;
           tamp_chaine[i] = fopen(tata[i],"w");  //ouverture des 5 mutants cor
          }

         // ecriture de la fin du fichier
         while((c=fgetc(M0))!=EOF)
          {
           fputc(c,M1);
           if(c=='\r')
            fputc('\n',M1);
          }
         fclose(M1);

         //ajout du debut du fichier dans les fichiers mutants
         fseek(M0,0,0);
         while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos-2))
          for(i=0;i<=2;i++)
```

```
            {
             fputc(c,tamp_chaine[i]);
            }
          //fin d'ajout

          fputc('+',tamp_chaine[0]);
          fputc('/',tamp_chaine[1]);
          fputc('*',tamp_chaine[2]);

         //on reecrit la fin du fichier
          M1 = fopen("mut_tampon_aor","r");
          while((c=fgetc(M1))!=EOF)
           {
            for(i=0;i<=2;i++)
              fputc(c,tamp_chaine[i]);
           }
        //fin de la reecriture

          for(i=0;i<=2;i++)
           fclose (tamp_chaine[i]);
          fclose(M1);
          fseek(M0,pos,0); //on remet a la position courante
          break;
case 3:
          M1 = fopen("mut_tampon_aor","w");
          fputc(c,M1);
          pos =ftell(M0);
          for(i=0;i<=2;i++)
           {
            sprintf(tata[i],"Mut_aor%d.vhdl",compteur_aor);
            compteur_aor++;
            tamp_chaine[i] = fopen(tata[i],"w");
           }

          // ecriture de la fin du fichier
          while((c=fgetc(M0))!=EOF)
           {
            fputc(c,M1);
            if(c=='\r')
              fputc('\n',M1);
           }
          fclose(M1);

          //ajout du debut du fichier dans les fichiers mutants
          fseek(M0,0,0);
          while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos-2))
           for(i=0;i<=2;i++)
            {
             fputc(c,tamp_chaine[i]);
            }
          //fin d'ajout
```

```
 fputc('-',tamp_chaine[0]);
 fputc('/',tamp_chaine[1]);
 fputc('+',tamp_chaine[2]);

//on reecrit la fin du fichier
 M1 = fopen("mut_tampon_aor","r");
 while((c=fgetc(M1))!=EOF)
  {
   for(i=0;i<=2;i++)
    fputc(c,tamp_chaine[i]);
  }
//fin de la reecriture

 for(i=0;i<=2;i++)
  fclose (tamp_chaine[i]);
 fclose(M1);
 fseek(M0,pos,0); //on remet a la position courante
 break;
case 5:
 M1 = fopen("mut_tampon_aor","w");
 fputc(c,M1);
 pos =ftell(M0);
 for(i=0;i<=2;i++)
  {
   sprintf(tata[i],"Mut_aor%d.vhdl",compteur_aor);
   compteur_aor++;
   tamp_chaine[i] = fopen(tata[i],"w");
  }

 // ecriture de la fin du fichier
 while((c=fgetc(M0))!=EOF)
  {
   fputc(c,M1);
   if(c=='\r')
    fputc('\n',M1);
  }
 fclose(M1);

 //ajout du debut du fichier dans les fichiers mutants
 fseek(M0,0,0);
 while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos-2))
  for(i=0;i<=2;i++)
   {
    fputc(c,tamp_chaine[i]);
   }
 //fin d'ajout

 fputc('-',tamp_chaine[0]);
 fputc('/',tamp_chaine[1]);
 fputc('*',tamp_chaine[2]);
```

```
 //on reecrit la fin du fichier
  M1 = fopen("mut_tampon_aor","r");
  while((c=fgetc(M1))!=EOF)
    {
     for(i=0;i<=2;i++)
       fputc(c,tamp_chaine[i]);
    }
//fin de la reecriture

  for(i=0;i<=2;i++)
   fclose (tamp_chaine[i]);
  fclose(M1);
  fseek(M0,pos,0); //on remet a la position courante
  break;

case 6:
  M1 = fopen("mut_tampon_aor","w");
  fputc(c,M1);
  pos =ftell(M0);
  for(i=0;i<=2;i++)
    {
     sprintf(tata[i],"Mut_aor%d.vhdl",compteur_aor);
     compteur_aor++;
     tamp_chaine[i] = fopen(tata[i],"w");
    }

  // ecriture de la fin du fichier
  while((c=fgetc(M0))!=EOF)
    {
     fputc(c,M1);
     if(c=='\r')
       fputc('\n',M1);
    }
  fclose(M1);

  //ajout du debut du fichier dans les fichiers mutants
  fseek(M0,0,0);
  while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos-2))
   for(i=0;i<=2;i++)
     {
      fputc(c,tamp_chaine[i]);
     }
  //fin d'ajout

  fputc('-',tamp_chaine[0]);
  fputc('+',tamp_chaine[1]);
  fputc('*',tamp_chaine[2]);

 //on reecrit la fin du fichier
  M1 = fopen("mut_tampon_aor","r");
```

```
 while((c=fgetc(M1))!=EOF)
  {
   for(i=0;i<=2;i++)
    fputc(c,tamp_chaine[i]);
  }
//fin de la reecriture

 for(i=0;i<=2;i++)
  fclose (tamp_chaine[i]);
 fclose(M1);
 fseek(M0,pos,0); //on remet a la position courante
 break;

case 8:
 M1 = fopen("mut_tampon_aor","w");
 fputc(c,M1);
 pos =ftell(M0);
 sprintf(tata[0],"Mut_aor%d.vhdl",compteur_aor);
 compteur_aor++;
 tamp_chaine[0] = fopen(tata[0],"w");

 // ecriture de la fin du fichier
 while((c=fgetc(M0))!=EOF)
  {
   fputc(c,M1);
   if(c=='\r')
    fputc('\n',M1);
  }
 fclose(M1);

 //ajout du debut du fichier dans les fichiers mutants
 fseek(M0,0,0);
 while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos-2))
  fputc(c,tamp_chaine[0]);
 //fin d'ajout

 fputc('>',tamp_chaine[0]);

//on reecrit la fin du fichier
 M1 = fopen("mut_tampon_aor","r");
 while((c=fgetc(M1))!=EOF)
  fputc(c,tamp_chaine[0]);
//fin de la reecriture

 fclose (tamp_chaine[0]);
 fclose(M1);
 fseek(M0,pos,0); //on remet a la position courante
 break;

case 10:
 M1 = fopen("mut_tampon_aor","w");
```

```
            fputc(c,M1);
            pos =ftell(M0);
            sprintf(tata[0],"Mut_aor%d.vhdl",compteur_aor);
            compteur_aor++;
            tamp_chaine[0] = fopen(tata[0],"w");

            // ecriture de la fin du fichier
            while((c=fgetc(M0))!=EOF)
              {
               fputc(c,M1);
               if(c=='\r')
                 fputc('\n',M1);
              }
            fclose(M1);

            //ajout du debut du fichier dans les fichiers mutants
            fseek(M0,0,0);
            while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos-2))
             fputc(c,tamp_chaine[0]);
            //fin d'ajout

            fputc('<',tamp_chaine[0]);

           //on reecrit la fin du fichier
            M1 = fopen("mut_tampon_aor","r");
            while((c=fgetc(M1))!=EOF)
              fputc(c,tamp_chaine[0]);
          //fin de la reecriture

            fclose (tamp_chaine[0]);
            fclose(M1);
            fseek(M0,pos,0); //on remet a la position courante
            break;
          default:
            break;


          }
   default:
    break;
   }
}
 }
}
```

```
//operateur de mutation CNR  et CSR

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define TRANSITION 9
#define ETAT_COURANT 10
#define TRANSITION3 4
#define ETAT_COURANT3 2
#define TRANSITION4 3
#define ETAT_COURANT4 2
#define TRANSITION5 5
#define ETAT_COURANT5 6


/* inventaire des signaux du programme source */
void main()
{
unsigned char etat_courant;
unsigned char ancien_etat;
unsigned char etat_courant2;
unsigned char etat_courant3;
unsigned char ancien_etat3;
unsigned char etat_courant4;
unsigned char ancien_etat4;
unsigned char etat_courant5;
unsigned char ancien_etat5;
char tabsig[120][120];
char tabname[120][120];
char tabtype[120][120];
long compteur_type;
char number1[10];
int c,c1,c2,i,i_name,j_name,i3,j3,i3_max,j,j_type,tamp,flag_name,i_max,j_max,f,compteur,i_ident;
int transition,transition3,transition4;
int transition5;
int compt_mutant;
int flag_compare;
long compteur_type1;
long flag_white;
long pos;
long posf;
long posM2;//position du pointeur
long postampon;
char word[1][80];
char tata[10];
FILE *M0=NULL;  /*definition du pointeur file pour le fichier d'entrée*/
FILE *M1=NULL;
FILE *M2=NULL;
FILE *M3=NULL;
FILE *M4=NULL;
FILE *M5=NULL;
```

```
FILE *M6=NULL;
FILE *M7=NULL;
FILE *M8=NULL;
FILE *M9=NULL;
FILE *tampon;
int diagr_etat_name[TRANSITION3][ETAT_COURANT3]={
        /*etat_suivant */
/* transitions*/        1,0,
                        0,0,
                        0,1,
                        0,0};
int diagr_etat_compteur[TRANSITION4][ETAT_COURANT4]={
        /*etat_suivant */
/* transitions*/        1,1,
                        0,0,
                        0,0};

int diagr_etat[TRANSITION][ETAT_COURANT]={
        /*etat_suivant */
/* transitions*/        1,0,0,0,0,0,0,8,8,0,
                        0,2,0,0,0,0,0,8,8,0,
                        0,0,3,0,0,0,0,8,8,0,
                        0,0,0,4,0,0,0,8,8,0,
                        0,0,0,0,5,0,0,8,8,0,
                        0,0,0,0,0,6,0,8,8,0,
                        0,0,0,0,0,0,7,8,8,0,
                        0,0,0,0,0,0,0,8,9,0,
                        0,0,0,0,0,0,0,8,8,0};


int diagr_etat_word[TRANSITION5][ETAT_COURANT5]={

        /*etat_suivant */
/* transitions*/        1,4,2,0,0,0,
                        1,5,2,0,0,0,
                        1,2,2,0,0,0,
                        1,1,3,0,0,0,
                        1,1,2,0,0,0};

etat_courant=0;
etat_courant2=0;
etat_courant3=0;
etat_courant4=0;
etat_courant5=0;
i=0;
i_name=0;
j_name =0;
j=0;
i_max=0;
f=0;
tamp=0;
```

```
compteur=0;
j_max=0;
i_ident=0;
flag_compare=0;
pos=0;
posM2=0;
postampon=0;

if(( M0 = fopen("gene.vhdl" ,"r"))==NULL)/* ouverture du fichier du premier mutant*/
  printf("ERREUR:le fichier source ne peut etre ouvert \n");
else{
  printf("fichier INV_SIG_TAB ouvert \n");
  M1 = fopen("INV_SIG_TAB","w"); /*ouverture du fichier inventaire*/
while ((c=fgetc(M0))!=EOF){
  switch(c)
{
case 'S':
case 's':transition=0;
break;
case 'I':
case 'i':transition=1;
break;
case 'G':
case 'g':transition=2;
break;
case 'N':
case 'n':transition=3;
break;
case 'A':
case 'a':transition=4;
break;
case 'L':
case 'l':transition=5;
break;
case ' ':transition=6;
break;
case ';':transition=7;
break;
default:transition=8;
break;
}
ancien_etat=etat_courant;
etat_courant= diagr_etat[transition][etat_courant];
switch(etat_courant)
{
case 7: switch (ancien_etat)
{
case 6:
fputc('s',M1);
fputc('i',M1);
fputc('g',M1);
```

```
 fputc('n',M1);
 fputc('a',M1);
 fputc('l',M1);
 fputc(' ',M1);
 break;
      }
   break;

   case 8: switch(ancien_etat)
     {
     case 7:fputc(c,M1);
break;
     case 8:fputc(c,M1);
break;
     }
   break;

   case 9: switch(ancien_etat)
     {
     case 8:
fputc(c,M1);
fputc('\n',M1);
break;
     }
   break;
   default:
     break;
}
}
fclose(M1);
fclose(M0);
 }


/*******************les signaux sont mis dans un tableau**************************/

if(( M1 = fopen("INV_SIG_TAB" ,"r"))==NULL)/* ouverture du fichier du premier mutant*/
   printf("ERREUR:le fichier source ne peut etre ouvert \n");
 else{

  M2 = fopen("TAB_NAME","w");//nom et type des signaux
   printf("fichier TAB_NAME  ouvert \n");
    while ((c1=fgetc(M1))!=EOF){
     switch(c1)
{
case ':':transition3 =0;
  transition4 =0;
  break;
case ';':transition3 =1;
  break;
case '=':transition4 =1;
```

```
  transition3 =3;
  break;
 default: transition3 =2;
  transition4 =2;
  break;
 }
     ancien_etat3=etat_courant3;
     etat_courant3= diagr_etat_name[transition3][etat_courant3];
    if (c1=='\n'){

i++;
j=0;
i_name++;
j_name=0;
j_type=0;
compteur=0;
i_max=i;
tabsig[i][j]=c1;

}
    else {

  /* permet de controller les variables rentrees dans les tab*/

 ancien_etat4=etat_courant4;
 etat_courant4= diagr_etat_name[transition4][etat_courant4];
 if(c1==':') {/* permet de mettre aussi le type */
  compteur++;}

 if(c1==';'){
  compteur++;
  fputc('\n',M2);
 }

 if((j>5)&&(compteur==0)){ /* on a directement le nom des signaux*/
  tabname[i_name][j_name]=c1;
  fputc(tabname[i_name][j_name],M2);
  j_name++;
 }

/* on a directement le type du signal dans tabtype */
  if((j>5)&&(compteur==1)){
   switch(etat_courant3)
{
case 1:
 switch(ancien_etat3)
  {
  case 1:
  case 0:
if(c1!=':') //test
{
```

```
tabtype[i_name][j_type]=c1; /*attention les car : et ' ' sont dedans*/
fputc(tabtype[i_name][j_type],M2);
j_type++;
}

    break;
   default:
    break;
   }
 break;
default:
 break;
}
   }


   if((j>5)&&(compteur==2)){  /*pour l'affichage */
    compteur++;
   }

 j++;
 j_max=j;}

   }
 }
/*++++++++++++++++++++++++dernier signal mis dans un tableau++++++++++++++++++++++++*/

M3 = fopen("TAB_FINAL","w");
fputs( tabtype[i_max-1],M3);

fclose(M3);
fclose(M2);
fclose(M1);


/*---------------------------------reconnaissance du meme type-----------------------------*/

 i3=0;
 flag_name=1;

 if(( M1 = fopen("TAB_NAME" ,"r"))==NULL)/* ouverture du fichier du premier mutant*/
   printf("ERREUR:le fichier source ne peut etre ouvert \n");
 else{
   M2 = fopen("TEST","w"); /* que les noms des variables*/
   printf("fichier TEST ouvert \n");
   while ((c2=fgetc(M1))!=EOF){
    if((c!='\n')&& (flag_name==1)){
fputs(tabname[i3],M2);
flag_name=0;
    }
if (c2=='\n'){
```

```
 i3++;
 fputc('\n',M2);
 flag_name=1;}
 i3_max=i3;
  }
 }
fclose(M1);
fclose(M2);

/*CONVERSION DU FICHIER TEXTE EN CHAINE DE CARACTERES*/
M1 = fopen("gene_chaine_word.vhdl","w");
M0 = fopen("gene.vhdl" ,"rb+");

while((c=fgetc(M0))!=EOF){

 switch(c){
 case ' ':transition5=0;
 break;
 case ';':transition5=1;
 break;
 case '(':transition5=2;
 break;
 case ')':transition5=3;
 break;
 default:transition5=4;
 break;
 }
 ancien_etat5=etat_courant5;
 etat_courant5= diagr_etat_word[transition5][etat_courant5];

 switch(etat_courant5){
 case 0:if(c!='\r') //doit peut etre enleve
 fputc(c,M1);
 if(c=='\r')
 fputc('\n',M1);
break;
 case 1:switch (ancien_etat5){
case 0:
default:if(c!='\r')
fputc(c,M1);
else
fputc('\n',M1);
break;
}
 break;
 case 2:fputc(c,M1);
 break;
 case 3:fputc(c,M1);
fputc('\0',M1);
break;
 case 4:
```

```
 case 5:fputc(c,M1); //test on doit peut etre intervertir l'ordre
 fputc('\0',M1);
 break;
 default:
 break;
 }


// fin de la generation du fichier chaine de carateres

}


fclose(M1);
fclose(M0);

/*CONVERSION DU FICHIER TEXTE EN CHAINE DE CARACTERES*/
M1 = fopen("gene_chaine_word.vhdl","w");
M0 = fopen("gene.vhdl" ,"rb+");
while((c=fgetc(M0))!=EOF){

 switch(c){
 case ' ':transition5=0;
 break;
 case ';':transition5=1;
 break;
 case '(':transition5=2;
 break;
 case ')':transition5=3;
 break;
 default:transition5=4;
 break;
 }
 ancien_etat5=etat_courant5;
 etat_courant5= diagr_etat_word[transition5][etat_courant5];

 switch(etat_courant5){
 case 0:if(c!='\r') //doit peut etre enleve
 fputc(c,M1);
break;
 case 1:switch (ancien_etat5){
case 0:
default:if(c!='\r')
fputc(c,M1);
break;
}
 break;
 case 2:fputc(c,M1);
 break;
 case 3:fputc(c,M1);
fputc('\0',M1);
```

```
 break;
  case 4:
  case 5:fputc(c,M1); //test on doit peut etre intervertir l'ordre
  fputc('\0',M1);
  break;
  default:
  break;
  }


// fin de la generation du fichier chaine de carateres

}


fclose(M1);
fclose(M0);


/*-------------------------------------------------------------------------------*/


compteur_type=0;
tamp=1;
flag_white=0;
j=0;
M3 = fopen("TEST2","w"); //fichier contenant les signaux de meme type
M0 = fopen("gene_chaine_word.vhdl","rb+");
M1 = fopen("WORD_TAB","w");

for(i3=0;i3<=i3_max-1;i3++)
 {compt_mutant=1;
     for(j3=0;j3<=i3_max-1;j3++)
  if(j3!=i3)
  {
  if ((strcmp(tabtype[i3],tabtype[j3])==0)&&(compt_mutant==1))
  {
  compt_mutant++;
  compteur_type1++;
  sprintf(number1,"Mut_sig%d",compteur_type1);
  i_ident=j3;
  fputs(tabname[i3],M3);
  fputc('\n',M3);
  j=0;

  while((c=fgetc(M0))!=EOF)
  {
  if(c!='\0'){
  word[0][j]=c;
  j++;
  if(c=='\r')
```

```
fputc('\n',M1);
}
else
{
if(c=='\r')
fputc('\n',M1);
else
{
word[0][j]='\0';
fputs(word[0],M1);
fputc('\0',M1);
j=0;
}
if ((strcmp("data(35 downto 4)",word[0])==0))
{
printf("comparaison \n");
pos =ftell(M0); printf("la position du pointeur est %d \n",pos);
compteur_type++;
M2 = fopen("mut_tampon","w");
sprintf(tata,"Mut_sig%d",compteur_type);
tampon = fopen(tata,"w");
fputs("REMPLACEMENT",M2);
posM2=ftell(M2);
while((c=fgetc(M0))!=EOF)
{
fputc(c,M2);
if(c=='\r')
fputc('\n',M2);
}
fclose(M2);

//ajout
fseek(M0,0,0);
while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=posf)){
fputc(c,tampon);
if(c=='\r')
fputc('\n',tampon);
}
postampon=ftell(tampon);
fclose(tampon);
//fin d'ajout

//on reecrit la fin du fichier
tampon = fopen(tata,"a");
M2 = fopen("mut_tampon","r");
while((c=fgetc(M2))!=EOF)
{
fputc(c,tampon);
}
//fin de la reecriture
```

```
  fclose (tampon);
  fclose(M2);
  fseek(M0,pos,0); //on remet a la position courante

  }
  else
  posf=ftell(M0);
  }
  }
  }
  }
}
fclose(M1);
fclose(M0);
}
```

```
//operateur de mutation CLR (cas d'un range)

// constant limit replacement : test des valeurs limites dans  le cas d'un range
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define TRANSITION 7
#define ETAT_COURANT 7
void main()
{
unsigned char etat_courant;
unsigned char ancien_etat;
unsigned char tata[10];
int c;
int transition;
int pos;
int compteur_clr;

FILE *M0=NULL;  /*definition du pointeur file pour le fichier d'entrÈe*/
FILE *M1=NULL;
FILE *M2=NULL;
FILE *tampon=NULL;
int diagr_etat[TRANSITION][ETAT_COURANT]={
                    /*etat_suivant */
          /* transitions*/      1,0,0,0,0,0,0,
                                0,2,0,0,0,0,0,
                                0,0,3,0,0,0,0,
                                0,0,0,4,0,0,0,
                                0,0,0,0,5,0,0,
                                0,0,0,0,0,6,0,
                                0,0,0,0,0,0,0};

etat_courant=0;
pos=0;
compteur_clr=0;

 if(( M0 = fopen("gene_test.vhdl" ,"r"))==NULL)/* ouverture du fichier du premier mutant*/
   printf("ERREUR:le fichier source ne peut etre ouvert \n");
 else
 {
 while ((c=fgetc(M0))!=EOF)
 {
 switch(c)
 {
 case 'R':
 case 'r':transition=0;
 break;
 case 'A':
 case 'a':transition=1;
 break;
 case 'N':
```

```
case 'n':transition=2;
break;
case 'G':
case 'g':transition=3;
break;
case 'E':
case 'e':transition=4;
break;
case ' ':transition=5;
break;
default:transition=6;
break;
}
ancien_etat=etat_courant;
etat_courant= diagr_etat[transition][etat_courant];
switch(etat_courant)
{
case 0:
break;
case 1:
case 2:
case 3:
case 4:
case 5:
break;
case 6:
M1 = fopen("mut_tampon","w");
pos =ftell(M0);
compteur_clr++;
sprintf(tata,"Mut_clr%d",compteur_clr);
tampon = fopen(tata,"w");
fputc(c,M1);
fputc('1',M1);
fputc('+',M1);

// ecriture de la fin du fichier
while((c=fgetc(M0))!=EOF)
{
fputc(c,M1);
if(c=='\r')
fputc('\n',M1);
}
fclose(M1);
// fin ecriture

//ajout du debut du fichier
fseek(M0,0,0);
while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos))
fputc(c,tampon);
fclose(tampon);
//fin d'ajout
```

```
//on reecrit la fin du fichier
tampon = fopen(tata,"a");
M1 = fopen("mut_tampon","r");
while((c=fgetc(M1))!=EOF)
{
fputc(c,tampon);
}
//fin de la reecriture

fclose (tampon);
fclose(M1);
fseek(M0,pos,0); //on remet a la position courante
break;
default:
break;
}
}
fclose(M1);
fclose(M0);
}
}
```

//operateur de mutation CLR (cas d'une constante)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define TRANSITION 11
#define ETAT_COURANT 13
void main()
{
unsigned char etat_courant;
unsigned char ancien_etat;
unsigned char tata[10];
unsigned char clr_m_chaine[10];
int c;
int transition;
int pos;
int pos_minus;
int compteur_clr;
int flag;

FILE *M0=NULL; /*definition du pointeur file pour le fichier d'entrÈe*/
FILE *M1=NULL;
FILE *M2=NULL;
FILE *tampon=NULL;
FILE *tampon_minus=NULL;
int diagr_etat[TRANSITION][ETAT_COURANT]={
                    /*etat_suivant */
          /* transitions*/     1,0,0,0,0,0,0,0,0,9,11,11,0,
                               0,2,0,0,0,0,0,0,0,9,11,11,0,
                               0,0,3,0,0,0,7,0,0,9,11,11,0,
                               0,0,0,4,0,0,0,0,0,9,11,11,0,
                               0,0,0,0,5,0,0,8,0,9,11,11,0,
                               0,0,0,0,0,6,0,0,0,9,11,11,0,
                               0,0,0,0,0,0,0,0,0,10,11,11,0,
                               0,0,0,0,0,0,0,0,0,9,11,11,0,
                               0,0,0,0,0,0,0,0,9,9,11,11,0,
                               0,0,0,0,0,0,0,0,0,0,11,12,0};

etat_courant=0;
pos=0;
pos_minus=0;
compteur_clr=0;
flag=0;

if(( M0 = fopen("gene_test.vhdl" ,"r"))==NULL)/* ouverture du fichier du premier mutant*/
  printf("ERREUR:le fichier source ne peut etre ouvert \n");
else
{
while ((c=fgetc(M0))!=EOF)
{
```

```
switch(c)
{
case 'C':
case 'c':transition=0;
break;
case 'O':
case 'o':transition=1;
break;
case 'N':
case 'n':transition=2;
break;
case 'S':
case 's':transition=3;
break;
case 'T':
case 't':transition=4;
break;
            case 'A':
case 'a':transition=5;
break;
            case '=':transition=6;
                  break;
             case ' ':transition=8;
                  break;
            case ';':transition=9;
                  break;
default:transition=7;
break;
}
ancien_etat=etat_courant;
etat_courant= diagr_etat[transition][etat_courant];
switch(etat_courant)
{
case 0: flag=0;
                  break;
case 1:
case 2:
case 3:
case 4:
case 5:
            case 6:
            case 7:
            case 8:
            case 9:
break;
case 10:
M1 = fopen("tampon_consp","w");
pos =ftell(M0);
compteur_clr++;
sprintf(tata,"Mut_clr_const%d",compteur_clr);
tampon = fopen(tata,"w");
```

```
fputc('1',M1);
fputc('+',M1);

// ecriture de la fin du fichier
while((c=fgetc(M0))!=EOF)
{
fputc(c,M1);
if(c=='\r')
fputc('\n',M1);
}
fclose(M1);
// fin ecriture

//ajout du debut du fichier
fseek(M0,0,0);
while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos))
                {
fputc(c,tampon);
                }

fclose(tampon);

//fin d'ajout

//on reecrit la fin du fichier
tampon = fopen(tata,"a");
M1 = fopen("tampon_consp","r");
while((c=fgetc(M1))!=EOF)
{
fputc(c,tampon);
}

//fin de la reecriture

fclose (tampon);
fclose(M1);
fseek(M0,pos,0); //on remet a la position courante
break;

        case 11:pos_minus =ftell(M0);
                if(c!='0')
                  flag++;
          break;

        case 12:if(flag!=0)
          {printf("je rentre2 \n");
                {
                 // pos_minus =ftell(M0);
                 M2 = fopen("tampon_consm","w");
                 compteur_clr++;
                 sprintf(clr_m_chaine,"Mut_clr_const%d",compteur_clr);
```

```
                    tampon_minus= fopen(clr_m_chaine,"w");
                    fputc('-',M2);
                    fputc('1',M2);
                    fputc(c,M2);

                }

               // ecriture de la fin du fichier
while((c=fgetc(M0))!=EOF)
{
fputc(c,M2);
if(c=='\r')
fputc('\n',M2);
}
fclose(M2);
// fin ecriture

                //ajout du debut du fichier
fseek(M0,0,0);
while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos_minus))
                 fputc(c,tampon_minus);
                fclose(tampon_minus);

//fin d'ajout

                //on reecrit la fin du fichier
tampon_minus = fopen(clr_m_chaine,"a");
M2 = fopen("tampon_consm","r");
                while((c=fgetc(M2))!=EOF)
{
fputc(c,tampon_minus);
}
//fin de la reecriture

                fclose (tampon_minus);
                fclose(M2);
fseek(M0,pos,0); //on remet a la position courante
            }
break;
default:
break;
}
}
     fclose(M2);
fclose(M1);
fclose(M0);
}
}
```

```
//operateur de mutation SVIR

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define TRANSITION 17
#define ETAT_COURANT 22
void main()
{
unsigned char etat_courant;
unsigned char ancien_etat;
unsigned char svir_chaine[10];
unsigned char clr_m_chaine[10];
long binaire[31];
int c;
int i;
int i_max;
int transition;
int pos;
int pos_minus;
int compteur_clr;
int flag;
int flag_binaire;

FILE *M0=NULL;  /*definition du pointeur file pour le fichier d'entrÈe*/
FILE *M1=NULL;
FILE *M2=NULL;
FILE *tampon=NULL;
FILE *tampon_minus=NULL;


int diagr_etat_svir[TRANSITION][ETAT_COURANT]={
                        /*etat_suivant */
          /* transitions*/      1,0,0,0,0,0,0,0,8,10,10,0,0,0,0,0,0,0,0,0,0,
                                0,2,0,0,0,0,0,0,8,10,10,0,0,0,0,0,0,0,0,0,0,
                                0,0,3,0,0,0,7,0,8,10,10,0,0,0,0,0,0,0,0,0,0,
                                0,0,0,4,0,0,0,0,8,10,10,0,0,0,0,0,0,0,0,0,0,
                                0,0,0,0,5,0,0,8,8,10,10,0,0,0,0,0,0,0,19,0,0,
                                0,0,0,0,0,6,0,0,8,10,10,12,0,0,15,0,0,0,0,0,0,
                                0,0,0,0,0,0,0,0,8,10,10,0,0,14,0,0,0,0,0,0,0,
                                0,0,0,0,0,0,0,0,8,10,10,0,0,0,0,16,0,0,0,0,0,
                                0,0,0,0,0,0,0,0,8,10,10,0,0,0,0,0,17,0,0,0,0,
                                0,0,0,0,0,0,0,0,8,10,10,0,0,0,0,0,0,8,0,0,8,
                                0,0,0,0,0,0,0,0,8,10,10,0,0,0,0,0,0,0,19,0,0,
                                0,0,0,0,0,0,0,0,8,10,10,0,0,0,0,0,0,0,0,20,0,
                                0,0,0,0,0,0,0,0,9,10,10,0,0,0,0,0,0,0,0,0,0,
                                0,0,0,0,0,0,0,0,8,10,10,0,0,0,0,0,0,0,0,0,0,
                                0,0,0,0,0,0,0,0,8,10,101,0,0,0,0,0,0,0,0,0,0};
```

```
etat_courant=0;
pos=0;
pos_minus=0;
compteur_clr=0;
flag_binaire=0;
flag=0;
i=0;
i_max=0;

if(( M0 = fopen("gene_test.vhdl" ,"r"))==NULL)/* ouverture du fichier du premier mutant*/
  printf("ERREUR:le fichier source ne peut etre ouvert \n");
else
{
while ((c=fgetc(M0))!=EOF)
{
switch(c)
{
case 'C':
case 'c':transition=0;
break;
case 'O':
case 'o':transition=1;
break;
case 'N':
case 'n':transition=2;
break;
case 'S':
case 's':transition=3;
break;
case 'T':
case 't':transition=4;
break;
        case 'A':
case 'a':transition=5;
break;
        case 'R':
case 'r':transition=6;
break;
        case 'I':
case 'i':transition=7;
break;
        case 'B':
case 'b':transition=8;
break;
        case 'L':
case 'l':transition=9;
break;
        case 'E':
case 'e':transition=10;
break;
```

```
            case 'Y':
 case 'y':transition=11;
 break;
            case 'P':
 case 'p':transition=12;
 break;
            case ':':transition=13;
                 break;
            case '=':transition=14;
                 break;
            case ';':transition=15;
                 break;
 default:transition=16;
 break;
 }
 ancien_etat=etat_courant;
 etat_courant= diagr_etat_svir[transition][etat_courant];
 switch(etat_courant)
 {
 case 0: //flag=0;
             //break;
 case 1:
 case 2:
 case 3:
 case 4:
 case 5:
            case 6:
            case 7:
            case 8:
            case 9:break;
            case 10:if(c=="")
             {
                 flag_binaire=1;
                 if(c!="")
                  {
                   binaire[i++]=c;
                   i_max=i;
                  }

            }
 break;
 case 101:
 M1 = fopen("tampon_svir","w");
 pos =ftell(M0);
 compteur_clr++;

 sprintf(svir_chaine,"Mut_svir%d",compteur_clr);
 tampon = fopen(svir_chaine,"w");
                if(flag_binaire==0)
                  {
                   fputc('1',M1);
```

```
                    fputc('+',M1);
                   }
                  else
                   {
                    for(i=0;i++;i<=i_max)
                     if(binaire[i]==0)
                      fputc('1',M1);
                     else
                      fputc('0',M1);
                   }

// ecriture de la fin du fichier
while((c=fgetc(M0))!=EOF)
{
fputc(c,M1);
if(c=='\r')
fputc('\n',M1);
}
fclose(M1);
// fin ecriture

//ajout du debut du fichier
fseek(M0,0,0);
while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos))
                  {
fputc(c,tampon);
                  }

fclose(tampon);

//fin d'ajout

//on reecrit la fin du fichier
tampon = fopen(tata,"a");
M1 = fopen("tampon_consp","r");
while((c=fgetc(M1))!=EOF)
{
fputc(c,tampon);
}

//fin de la reecriture

fclose (tampon);
fclose(M1);
fseek(M0,pos,0); //on remet a la position courante
break;

          case 11:pos_minus =ftell(M0);
                  if(c!='0')
                   flag++;
           break;
```

```
            case 12:if(flag!=0)
              {
                    {
                     // pos_minus =ftell(M0);
                     M2 = fopen("tampon_consm","w");
                     compteur_clr++;
                     sprintf(clr_m_chaine,"Mut_clr_const%d",compteur_clr);
                     tampon_minus= fopen(clr_m_chaine,"w");
                     fputc('-',M2);
                     fputc('1',M2);
                     fputc(c,M2);
                    }
                   // ecriture de la fin du fichier
while((c=fgetc(M0))!=EOF)
{
fputc(c,M2);
if(c=='\r')
fputc('\n',M2);
}
fclose(M2);
// fin ecriture
                    //ajout du debut du fichier
fseek(M0,0,0);
while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos_minus))
                     fputc(c,tampon_minus);
                   fclose(tampon_minus);

//fin d'ajout
                    //on reecrit la fin du fichier
tampon_minus = fopen(clr_m_chaine,"a");
M2 = fopen("tampon_consm","r");
                   while((c=fgetc(M2))!=EOF)
{
fputc(c,tampon_minus);
}
//fin de la reecriture

                   fclose (tampon_minus);
                   fclose(M2);
fseek(M0,pos,0); //on remet a la position courante
              }
break;
default:
break;
}
}
     fclose(M2);
fclose(M1);
fclose(M0);
}
}
```

```
//operateur de mutation LCR

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define TRANSITION 9
#define ETAT_COURANT 16
void main()
{
unsigned char etat_courant;
unsigned char lcr_chaine1[10];
unsigned char lcr_chaine2[10];
unsigned char lcr_chaine3[10];
unsigned char lcr_chaine4[10];
int c;
int transition;
int pos;
int compteur_lcr;
FILE *M0=NULL;  /*definition du pointeur file pour le fichier d'entrée*/
FILE *M1=NULL;
FILE *M2=NULL;
FILE *M3=NULL;
FILE *M4=NULL;
FILE *tampon1=NULL;
FILE *tampon2=NULL;
FILE *tampon3=NULL;
FILE *tampon4=NULL;


int diagr_etat_lcr[TRANSITION][ETAT_COURANT]={
                        /*etat_suivant */
            /* transitions*/       1,0,0,0,5,0,0,0,0,0,0,0,0,0,0,0,
                                   4,2,0,0,0,6,0,0,11,0,0,0,0,0,0,0,
                                   0,0,3,0,0,0,7,0,0,0,0,0,0,0,0,0,
                                   14,0,0,0,0,0,0,0,9,0,0,12,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,10,0,0,13,0,15,0,
                                   8,0,0,0,0,0,0,0,0,0,C,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
etat_courant=0;
pos=0;
compteur_lcr=0;

 if(( M0 = fopen("gene.txt" ,"r"))==NULL)/* ouverture du fichier du premier mutant*/
   printf("ERREUR:le fichier source ne peut etre ouvert \n");
 else
   {
    while ((c=fgetc(M0))!=EOF)
     {
      switch(c)
        {
        case 'A':
```

```
 case 'a':transition=0;
  break;
 case 'N':
 case 'n':transition=1;
  break;
 case 'D':
 case 'd':transition=2;
  break;
 case 'o':
 case 'O':transition=3;
  break;
 case 'R':
 case 'r':transition=4;
  break;
 case 'X':
 case 'x':transition=5;
  break;
 default:transition=6;
  break;
 }
etat_courant= diagr_etat_lcr[transition][etat_courant];
switch(etat_courant)
 {
 case 0:
 case 1:
 case 2:
  break;
 case 3:
  M1 = fopen("tampon_or_lcr","w");
  M2 = fopen("tampon_xor_lcr","w");
  M3 = fopen("tampon_xnor_lcr","w");
  M4 = fopen("tampon_nand_lcr","w");
  compteur_lcr++;
  sprintf(lcr_chaine1,"Mut_lcr%d",compteur_lcr);
  tampon1 = fopen(lcr_chaine1,"w");
  compteur_lcr++;
  sprintf(lcr_chaine2,"Mut_lcr%d",compteur_lcr);
  tampon2 = fopen(lcr_chaine2,"w");
  compteur_lcr++;
  sprintf(lcr_chaine3,"Mut_lcr%d",compteur_lcr);
  tampon3 = fopen(lcr_chaine3,"w");
  compteur_lcr++;
  sprintf(lcr_chaine4,"Mut_lcr%d",compteur_lcr);
  tampon4 = fopen(lcr_chaine4,"w");
  fputs("or",M1);
  fputs("xor",M2);
  fputs("xnor",M3);
  fputs("nand",M4);
  pos =ftell(M0);

// ecriture de la fin du fichier
```

```
        while((c=fgetc(M0))!=EOF)
          {
           fputc(c,M1);
           fputc(c,M2);
           fputc(c,M3);
           fputc(c,M4);
           if(c=='\r')
            {
             fputc('\n',M1);
             fputc('\n',M2);
             fputc('\n',M3);
             fputc('\n',M4);
            }
          }
        fclose(M1);
        fclose(M2);
        fclose(M3);
        fclose(M4);
// fin ecriture

//ajout du debut du fichier
        fseek(M0,0,0);
        while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos-3)) // pos -3 a cause de and
          {
           fputc(c,tampon1);
           fputc(c,tampon2);
           fputc(c,tampon3);
           fputc(c,tampon4);
          }
        fclose(tampon1);
        fclose(tampon2);
        fclose(tampon3);
        fclose(tampon4);

//fin d'ajout

//on reecrit la fin du fichier
        tampon1 = fopen(lcr_chaine1,"a");
        tampon2 = fopen(lcr_chaine2,"a");
        tampon3 = fopen(lcr_chaine3,"a");
        tampon4 = fopen(lcr_chaine4,"a");
        M1 = fopen("tampon_or_lcr","r");
        M2 = fopen("tampon_xor_lcr","r");
        M3 = fopen("tampon_xnor_lcr","r");
        M4 = fopen("tampon_nand_lcr","r");

        while((c=fgetc(M1))!=EOF)
          {
           fputc(c,tampon1);
          }
        while((c=fgetc(M2))!=EOF)
```

```
            {
             fputc(c,tampon2);

            }
           while((c=fgetc(M3))!=EOF)
            {
             fputc(c,tampon3);

            }
           while((c=fgetc(M4))!=EOF)
            {
             fputc(c,tampon4);

            }
//fin de la reecriture
          fclose (tampon1);
          fclose (tampon2);
          fclose (tampon3);
          fclose (tampon4);
          fclose(M1);
          fclose(M2);
          fclose(M3);
          fclose(M4);
          fseek(M0,pos,0); //on remet a la position courant
          break;
         case 4:
         case 5:
         case 6:
          break;
         case 7:
          M1 = fopen("tampon_or_lcr","w");
          M2 = fopen("tampon_xor_lcr","w");
          M3 = fopen("tampon_xnor_lcr","w");
          M4 = fopen("tampon_and_lcr","w");
          compteur_lcr++;
          sprintf(lcr_chaine1,"Mut_lcr%d",compteur_lcr);
          tampon1 = fopen(lcr_chaine1,"w");
          compteur_lcr++;
          sprintf(lcr_chaine2,"Mut_lcr%d",compteur_lcr);
          tampon2 = fopen(lcr_chaine2,"w");
          compteur_lcr++;
          sprintf(lcr_chaine3,"Mut_lcr%d",compteur_lcr);
          tampon3 = fopen(lcr_chaine3,"w");
          compteur_lcr++;
          sprintf(lcr_chaine4,"Mut_lcr%d",compteur_lcr);
          tampon4 = fopen(lcr_chaine4,"w");

          fputs("or",M1);
          fputs("xor",M2);
          fputs("xnor",M3);
          fputs("and",M4);
```

```
pos =ftell(M0);

// ecriture de la fin du fichier
while((c=fgetc(M0))!=EOF)
 {
  fputc(c,M1);
  fputc(c,M2);
  fputc(c,M3);
  fputc(c,M4);
  if(c=='\r')
   {
    fputc('\n',M1);
    fputc('\n',M2);
    fputc('\n',M3);
    fputc('\n',M4);
   }
 }
fclose(M1);
fclose(M2);
fclose(M3);
fclose(M4);
// fin ecriture

//ajout du debut du fichier
fseek(M0,0,0);
while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos-4)) // pos -4 a cause de nand
 {
  fputc(c,tampon1);
  fputc(c,tampon2);
  fputc(c,tampon3);
  fputc(c,tampon4);
 }

fclose(tampon1);
fclose(tampon2);
fclose(tampon3);
fclose(tampon4);

//fin d'ajout

//on reecrit la fin du fichier
tampon1 = fopen(lcr_chaine1,"a");
tampon2 = fopen(lcr_chaine2,"a");
tampon3 = fopen(lcr_chaine3,"a");
tampon4 = fopen(lcr_chaine4,"a");
M1 = fopen("tampon_or_lcr","r");
M2 = fopen("tampon_xor_lcr","r");
M3 = fopen("tampon_xnor_lcr","r");
M4 = fopen("tampon_and_lcr","r");

while((c=fgetc(M1))!=EOF)
```

```
            {
             fputc(c,tampon1);
            }
           while((c=fgetc(M2))!=EOF)
            {
             fputc(c,tampon2);

            }
           while((c=fgetc(M3))!=EOF)
            {
             fputc(c,tampon3);

            }
           while((c=fgetc(M4))!=EOF)
            {
             fputc(c,tampon4);

            }
//fin de la reecriture

           fclose (tampon1);
           fclose (tampon2);
           fclose (tampon3);
           fclose (tampon4);
           fclose(M1);
           fclose(M2);
           fclose(M3);
           fclose(M4);
           fseek(M0,pos,0); //on remet a la position courante
           break;
         case 8:
         case 9:
          break;
         case 10:
          M1 = fopen("tampon_and_lcr","w");
          M2 = fopen("tampon_or_lcr","w");
          M3 = fopen("tampon_xnor_lcr","w");
          M4 = fopen("tampon_nand_lcr","w");
          compteur_lcr++;
          sprintf(lcr_chaine1,"Mut_lcr%d",compteur_lcr);
          tampon1 = fopen(lcr_chaine1,"w");
          compteur_lcr++;
          sprintf(lcr_chaine2,"Mut_lcr%d",compteur_lcr);
          tampon2 = fopen(lcr_chaine2,"w");
          compteur_lcr++;
          sprintf(lcr_chaine3,"Mut_lcr%d",compteur_lcr);
          tampon3 = fopen(lcr_chaine3,"w");
          compteur_lcr++;
          sprintf(lcr_chaine4,"Mut_lcr%d",compteur_lcr);
          tampon4 = fopen(lcr_chaine4,"w");
```

```
fputs("and",M1);
fputs("or",M2);
fputs("xnor",M3);
fputs("nand",M4);
pos =ftell(M0);

// ecriture de la fin du fichier
while((c=fgetc(M0))!=EOF)
 {
  fputc(c,M1);
  fputc(c,M2);
  fputc(c,M3);
  fputc(c,M4);
  if(c=='\r')
   {
    fputc('\n',M1);
    fputc('\n',M2);
    fputc('\n',M3);
    fputc('\n',M4);
   }
 }
fclose(M1);
fclose(M2);
fclose(M3);
fclose(M4);
// fin ecriture

//ajout du debut du fichier
fseek(M0,0,0);
while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos-3)) // pos -3 a cause de xor
 {
  fputc(c,tampon1);
  fputc(c,tampon2);
  fputc(c,tampon3);
  fputc(c,tampon4);
 }

fclose(tampon1);
fclose(tampon2);
fclose(tampon3);
fclose(tampon4);
//fin d'ajout

//on reecrit la fin du fichier
tampon1 = fopen(lcr_chaine1,"a");
tampon2 = fopen(lcr_chaine2,"a");
tampon3 = fopen(lcr_chaine3,"a");
tampon4 = fopen(lcr_chaine4,"a");
M1 = fopen("tampon_and_lcr","r");
M2 = fopen("tampon_or_lcr","r");
M3 = fopen("tampon_xnor_lcr","r");
```

```
        M4 = fopen("tampon_nand_lcr","r");

        while((c=fgetc(M1))!=EOF)
         {
          fputc(c,tampon1);

         }
        while((c=fgetc(M2))!=EOF)
         {
          fputc(c,tampon2);

         }
        while((c=fgetc(M3))!=EOF)
         {
          fputc(c,tampon3);

         }
        while((c=fgetc(M4))!=EOF)
         {
          fputc(c,tampon4);

         }


//fin de la reecriture

        fclose (tampon1);
        fclose (tampon2);
        fclose (tampon3);
        fclose (tampon4);
        fclose(M1);
        fclose(M2);
        fclose(M3);
        fclose(M4);
        fseek(M0,pos,0); //on remet a la position courante
        break;

       case 11:
       case 12:
        break;
       case 13:
        M1 = fopen("tampon_or_lcr","w");
        M2 = fopen("tampon_xor_lcr","w");
        M3 = fopen("tampon_and_lcr","w");
        M4 = fopen("tampon_nand_lcr","w");
        compteur_lcr++;
        sprintf(lcr_chaine1,"Mut_lcr%d",compteur_lcr);
        tampon1 = fopen(lcr_chaine1,"w");
        compteur_lcr++;
        sprintf(lcr_chaine2,"Mut_lcr%d",compteur_lcr);
        tampon2 = fopen(lcr_chaine2,"w");
```

```
compteur_lcr++;
sprintf(lcr_chaine3,"Mut_lcr%d",compteur_lcr);
tampon3 = fopen(lcr_chaine3,"w");
compteur_lcr++;
sprintf(lcr_chaine4,"Mut_lcr%d",compteur_lcr);
tampon4 = fopen(lcr_chaine4,"w");

fputs("or",M1);
fputs("xor",M2);
fputs("and",M3);
fputs("nand",M4);
pos =ftell(M0);

// ecriture de la fin du fichier
while((c=fgetc(M0))!=EOF)
 {
  fputc(c,M1);
  fputc(c,M2);
  fputc(c,M3);
  fputc(c,M4);
  if(c=='\r')
   {
    fputc('\n',M1);
    fputc('\n',M2);
    fputc('\n',M3);
    fputc('\n',M4);
   }
 }
fclose(M1);
fclose(M2);
fclose(M3);
fclose(M4);
// fin ecriture

//ajout du debut du fichier
fseek(M0,0,0);
while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos-4)) // pos -4 a cause de xnor
 {
  fputc(c,tampon1);
  fputc(c,tampon2);
  fputc(c,tampon3);
  fputc(c,tampon4);
 }

fclose(tampon1);
fclose(tampon2);
fclose(tampon3);
fclose(tampon4);

//fin d'ajout
```

```
//on reecrit la fin du fichier
tampon1 = fopen(lcr_chaine1,"a");
tampon2 = fopen(lcr_chaine2,"a");
tampon3 = fopen(lcr_chaine3,"a");
tampon4 = fopen(lcr_chaine4,"a");
M1 = fopen("tampon_or_lcr","r");
M2 = fopen("tampon_xor_lcr","r");
M3 = fopen("tampon_and_lcr","r");
M4 = fopen("tampon_nand_lcr","r");

while((c=fgetc(M1))!=EOF)
 {
  fputc(c,tampon1);

 }
while((c=fgetc(M2))!=EOF)
 {
  fputc(c,tampon2);

 }
while((c=fgetc(M3))!=EOF)
 {
  fputc(c,tampon3);

 }
while((c=fgetc(M4))!=EOF)
 {
  fputc(c,tampon4);

 }
//fin de la reecriture

fclose (tampon1);
fclose (tampon2);
fclose (tampon3);
fclose (tampon4);
fclose(M1);
fclose(M2);
fclose(M3);
fclose(M4);
fseek(M0,pos,0); //on remet a la position courante
break;
case 14:
break;
case 15:
M1 = fopen("tampon_xnor_lcr","w");
M2 = fopen("tampon_xor_lcr","w");
M3 = fopen("tampon_and_lcr","w");
M4 = fopen("tampon_nand_lcr","w");
compteur_lcr++;
sprintf(lcr_chaine1,"Mut_lcr%d",compteur_lcr);
```

```
 tampon1 = fopen(lcr_chaine1,"w");
 compteur_lcr++;
 sprintf(lcr_chaine2,"Mut_lcr%d",compteur_lcr);
 tampon2 = fopen(lcr_chaine2,"w");
 compteur_lcr++;
 sprintf(lcr_chaine3,"Mut_lcr%d",compteur_lcr);
 tampon3 = fopen(lcr_chaine3,"w");
 compteur_lcr++;
 sprintf(lcr_chaine4,"Mut_lcr%d",compteur_lcr);
 tampon4 = fopen(lcr_chaine4,"w");

 fputs("xnor",M1);
 fputs("xor",M2);
 fputs("and",M3);
 fputs("nand",M4);
 pos =ftell(M0);

 // ecriture de la fin du fichier
 while((c=fgetc(M0))!=EOF)
  {
   fputc(c,M1);
   fputc(c,M2);
   fputc(c,M3);
   fputc(c,M4);
   if(c=='\r')
    {
     fputc('\n',M1);
     fputc('\n',M2);
     fputc('\n',M3);
     fputc('\n',M4);
    }
  }
fclose(M1);
fclose(M2);
fclose(M3);
fclose(M4);
// fin ecriture

//ajout du debut du fichier
fseek(M0,0,0);
while(((c=fgetc(M0))!=EOF)&&(ftell(M0)<=pos-2)) // pos -2 a cause de or
 {
  fputc(c,tampon1);
  fputc(c,tampon2);
  fputc(c,tampon3);
  fputc(c,tampon4);
 }

fclose(tampon1);
fclose(tampon2);
fclose(tampon3);
```

```
            fclose(tampon4);
            //fin d'ajout

            //on reecrit la fin du fichier
            tampon1 = fopen(lcr_chaine1,"a");
            tampon2 = fopen(lcr_chaine2,"a");
            tampon3 = fopen(lcr_chaine3,"a");
            tampon4 = fopen(lcr_chaine4,"a");
            M1 = fopen("tampon_xnor_lcr","r");
            M2 = fopen("tampon_xor_lcr","r");
            M3 = fopen("tampon_and_lcr","r");
            M4 = fopen("tampon_nand_lcr","r");

            while((c=fgetc(M1))!=EOF)
             {
              fputc(c,tampon1);
             }
            while((c=fgetc(M2))!=EOF)
             {
              fputc(c,tampon2);

             }
            while((c=fgetc(M3))!=EOF)
             {
              fputc(c,tampon3);

             }
            while((c=fgetc(M4))!=EOF)
             {
              fputc(c,tampon4);

             }
            //fin de la reecriture

            fclose (tampon1);
            fclose (tampon2);
            fclose (tampon3);
            fclose (tampon4);
            fclose(M1);
            fclose(M2);
            fclose(M3);
            fclose(M4);
            fseek(M0,pos,0); //on remet a la position courante
            break;
           default:
            break;
           }
         }
        fclose(M0);
       }
      }
```