POLYPUBLIE
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Génération de dictionnaires de pannes
Auteur: Author:	David Marche
Date:	1996
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Marche, D. (1996). Génération de dictionnaires de pannes [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/9001/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/9001/
Directeurs de recherche: Advisors:	Bozena Kaminska
Programme: Program:	Unspecified

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

Université de Montréal

Génération de dictionnaires de pannes

par

David MARCHE

DÉPARTEMENT DE GÉNIE ÉLECTRIQUE

ET DE GÉNIE INFORMATIQUE

ÉCOLE POLYTECHNIQUE

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES (M. Sc. A.)

(GÉNIE ÉLECTRIQUE)

DÉCEMBRE 1996

© David Marche, 1996

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-26494-7

Canada

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE

Ce mémoire intitulé:

GÉNÉRATION DE DICTIONNAIRES DE PANNES

présenté par: MARCHE David

en vue de l'obtention du diplôme de: Maîtrise ès Sciences Appliquées

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

M. BOIS Guy, Ph.D., président.

Mme. KAMINSKA Bozena, Ph.D., membre et directeur de recherche.

M. SLAMANI Mustapha, Ph.D., membre.

Dédicace

Je voudrais dédier ce travail à tous ceux qui y trouveront des informations d'utilité appréciable. L'électronique est une activité hasardeuse, je ne peux en aucun cas être tenu responsable d'accidents reliés directement ou indirectement au générateur de dictionnaires de pannes.

Remerciements

Je voudrais en premier lieu remercier ma directrice de recherche, Dr. Bozena Kaminska. Ses conseils, son soutien et la confiance qu'elle a eue en moi sont autant de facteurs qui ont fait de cette maîtrise un travail agréable. De plus je tiens à la remercier pour le support financier qu'elle m'a donné tout au long de ces deux années.

J'adresse également mes remerciements aux membres du jury pour avoir accepté de participer à l'évaluation de ce travail. Je cite particulièrement monsieur Guy Bois président du jury, ainsi que Bozena Kaminska et Mustapha Slamani membres du jury.

Je voudrais aussi remercier toute l'équipe du laboratoire de VLSI et de GRM94, ainsi que tous ceux et celles qui, de près ou de loin, ont contribué à l'achèvement de ce travail. Bonne chance à eux dans leurs travaux.

Résumé

Ce document présente le cheminement et les résultats d'une recherche sur la génération de dictionnaires de pannes.

L'importance grandissante de la fiabilité des circuits électroniques a mis l'accent sur l'insertion de tests au niveau de leur production. Le test lui-même se divise en deux étapes distinctes: la génération du test, l'application du test. La méthode la plus utilisée pour tester un circuit, est de dresser une liste des pannes qui peuvent toucher le circuit, puis de déterminer les stimuli qui, appliqués au circuit, peuvent provoquer l'apparition de ces pannes et la propagation du comportement fautif jusqu'à une sortie. Ces stimuli sont les vecteurs de test. La liste des pannes traitées constitue le dictionnaire de pannes.

Un générateur de dictionnaire de pannes est un outil qui génère la liste des pannes de façon automatique. Chaque circuit possède son dictionnaire propre et le rôle des générateurs décrits plus bas est d'automatiser la génération de ces dictionnaires.

Il existe plusieurs façons de générer un dictionnaire de pannes. La méthode de génération dépend essentiellement des informations que l'on peut fournir au générateur. En effet, un circuit peut être donné au générateur sous plusieurs formes: liste des noeuds (netlist), dessin des masques (layout), description comportementale, etc. Chacune de ces formes fournit un ensemble d'informations différent. Pour cette raison, le travail présenté ici décrit deux générateurs de dictionnaire de pannes très différents. Le premier se sert des informations contenues dans la liste des noeuds (netlist) des circuits pour générer le dictionnaire. Le second utilise plutôt les informations relatives au dessin des masques des circuits.

Selon le type de générateur utilisé, les dictionnaires résultants diffèrent. Pour cette raison, il est important de connaître la qualité des dictionnaires obtenus par les différents générateurs. Ces qualités sont directement reliées à la pertinence des pannes générées. Il est par exemple normal qu'un générateur de dictionnaire de pannes basé sur la liste des noeuds (netlist) d'un circuit fournisse un certain nombre de pannes qui peuvent s'avérer peu probables étant donné le dessin des masques choisi pour réaliser ce circuit. L'aspect du temps de génération du dictionnaire est aussi à considérer lorsque l'on s'intéresse aux performances des générateurs. Les générateurs qui se basent sur le dessin des masques sont beaucoup plus lents du fait qu'ils doivent gérer une quantité d'information beaucoup plus importante.

Ce travail a donc permis de réaliser deux générateurs de dictionnaires de pannes, l'un utilisant les informations de la liste des noeuds et l'autre du dessin des masques. Chacun des deux possède des qualités et des faiblesses. L'important est de choisir celui qui s'adapte le mieux à l'utilisation projetée.

Si les dictionnaires de pannes sont largement utilisés pour la génération de test, il est important de noter qu'ils peuvent également être très utiles pour d'autres tâches comme, par exemple, le choix et l'optimisation des designs lors de la conception des circuits.

Abstract

This document presents the methods and results of a research on fault dictionary generation.

A fault dictionary generator is very useful in electronic circuit test field. A fault dictionary is a list of fault that could affect a circuit. Therefore, each circuit has his own fault dictionary and the goal of the generators described below is to automate the generation of these dictionaries.

There is several ways of generating fault dictionaries depending of what type of information is accessible. Thus, a circuit can be described in different formats: the netlist, the layout, the behavioral description, etc. For this reason, this work has led to two different fault dictionary generators, each of them treating different circuit description type to give the dictionary. The first generator generates a dictionary according to information found in the netlist. The second generator generates a dictionary according to information found in the layout.

According to the type of the fault dictionary generator, the dictionary obtained can substantially differ. For this reason, it is important to know the quality of the dictionaries obtained. This is greatly dependant on the likelihood of occurrence of the faults listed in the dictionaries. For example a certain number of faults not likely to occur should be expected from a generator based on the netlist. On the other hand a dictionary obtained

from a layout could be more realistic but takes more time to be generated. This is why the choice of type of fault dictionary generator should be made according to both, the information available, and the performances expected.

This work has successfully led to two fault dictionary generators. The first uses the netlist's informations. The second uses the layout's informations. It is important to choose the one who is the better suited for his use. Note also that the use of fault dictionary generator is not only useful for test generation, but also during other development phases like design optimization for example.

Table des matières

Dédicace iv

Remerciements v

Résumé vi

Abstract viii

Table des matières x

Liste des figures xiv

Liste des tableaux xvi

Liste des annexes xvii

Chapitre I INTRODUCTION 1

Chapitre II REVUE DE LITTÉRATURE 5

2.1 Introduction 5

2.2 Connaissances et techniques de test 6

2.2.1 Test des circuits numériques 6

2.2.2 Test de circuits analogiques 7

2.3 Défauts et conséquences 8

2.3.1 Sources des perturbations des procédés de fabrication 8
2.3.2 Inexactitudes de fabrication reliées au procédé 9
2.3.3 Conséquences des défauts de fabrication 10
2.1.1 Modélisation des défauts 11
2.3 Les générateurs de dictionnaires de pannes 13
2.3.1 Généralités 13
2.3.1.1 Différents types de dictionnaires 13
2.3.1.2 Informations données par le dictionnaire 14
2.3.1.3 Entrées et sorties des générateurs de dictionnaires de pannes 15
2.4.1 Travaux précédents 16
Chapitre III DICTIONNAIRE EXHAUSTIF 18
3.1 Introduction. 18
3.2 Génération de dictionnaire basée sur la liste des noeuds 18
3.2.1 Méthode des noeuds 18
3.2.2 Arbre de modélisation de la liste des noeuds (netlist) 19
3.2.3 Méthode des éléments 22
3.3 Implantation informatique et résultats de simulation 25
3.3.1 Implantation informatique 25
3.3.2 Analyse de complexité 29
3.3.3 Exécution et résultats de simulation 29
3.4 Conclusion 32

Chapitre IV DICTIONNAIRE REALISTE 34

4.1 Introduction 34

4.2 Le générateur de dictionnaire réaliste 34

4.2.1 Algorithme général 34

4.3 Insertion de défauts 36

4.3.1 Algorithme d'insertion 36

4.3.2 Modélisation des défauts 37

4.4 Analyse des pannes engendrées 37

4.4.1 Interactions défauts-polygones 37

4.4.2 Analyse des pannes en CMOS 39

4.4.2.1 Exemple 40

4.5 Implantation informatique 42

4.5.1 CADENCE et le SKILL 42

4.5.2 Choix d'implantation 43

4.5.3 Modélisation des défauts 44

4.5.3.1 Forme des défauts 44

4.5.3.2 Taille des défauts 44

4.5.3.3 Insertion des défauts 45

4.5.4 Analyse des pannes engendrées 46

4.5.4.1 Exemple 47

4.5.5 Analyse de complexité 48

4.6 Résultats et mode d'emploi 50

4.7 Discussion et Conclusion 59

CONCLUSION 61

RÉFÉRENCES 63

ANNEXES 66

ANNEXE A 67

ANNEXE B 86

Liste des figures

FIGURE 2.2 Défaut causant un court-circuit au cours du procédé lithographique 12

FIGURE 2.4 Entrées et sorties des générateur de dictionnaires de pannes....................16

FIGURE 1.2 Macro et visibilité 20

FIGURE 1.3 Arbre de modélisation...21

FIGURE 1.4 Génération de la liste des noeuds d'un circuit 22

FIGURE 1.5 Différentes localisations de fautes pour un même nom de noeud fautif....23

FIGURE 1.6 Parcours de l'arbre de modélisation de la liste de noeuds 26

FIGURE 1.7 Structure d'un élément de l'arbre. 27

FIGURE 1.8 Filtre à état variable 31

FIGURE 1.9 Fonctionnement général d'un générateur de dictionnaires réalistes 35

FIGURE 1.10 Interactions défaut-polygone 38

FIGURE 1.11 Exemple de défaut et sa panne associée 41

FIGURE 1.12 Importance du type d'interaction défaut-polygones 42

FIGURE 1.13 Distribution de probabilité des défauts 45

FIGURE 1.14 Déplacement du défaut sur la surface du circuit...46

FIGURE 1.15 Exemple d'interaction défaut-polygones 48

FIGURE 1.16 Dessin des masques d'un inverseur CMOS 51

FIGURE 1.17 Inverseur CMOS 52

FIGURE 1.18 Exemple de défaut réaliste. 53

FIGURE 1.19 Dessin des masques d'un amplificateur 56

FIGURE 1.20 Chargement du générateur travaillant sur le dessin des masques 57

Liste des tableaux

TABLEAU 1 Pannes associées à chaque élément........24

TABLEAU 2 Interaction défaut-polygones-pannes en CMOS........40

Liste des annexes

ANNEXE A 75

ANNEXE B 93

Chapitre I

INTRODUCTION

Aujourd'hui, un processus de design de circuit électronique compte presque toujours une partie de test. La nécessité de distribuer un produit fiable et peu coûteux demande un test systématique des circuits produits. Ceci peut se faire à différentes étapes de la fabrication. Il faut cependant savoir que pour un circuit, plus son cycle de production avance, plus l'investissement dans ce circuit augmente. On comprend donc facilement que l'on tente de déterminer les circuits défectueux le plus tôt possible pour les écarter des étapes subséquentes de fabrication. Ceci explique que la part de travail reliée au test puisse représenter 25 à 50% de l'effort investi dans le développement d'un circuit intégré.

Le test d'un circuit peut se diviser en trois tâches distinctes: la génération du test, la validation du test, et l'application du test.

La génération de test est le travail grâce auquel on détermine les entrées à appliquer à un circuit ainsi que les sorties à observer pour détecter une faute de comportement ou de fabrication. Le test généré est un ensemble de vecteurs de tests. Chacun de ces vecteurs comporte les paramètres, qui, appliqués aux entrées du circuit sous test, permettent d'observer les pannes d'un circuit defectueux. Les vecteurs de test sont donc constitués de valeurs de tensions, courants, fréquences. Plusieurs approches peuvent être utilisées pour générer les vecteurs de tests. L'approche fonctionnelle ne s'intéresse qu'au bon fonction-

nement du circuit. Son but est de vérifier que la fonction demandée au circuit est bien réalisée. Elle ne s'occupe pas pour cela de savoir si le circuit a bien été construit. En effet, le circuit est plutôt considéré comme une boîte noire qui doit pour certaines valeurs d'entrées fournir certaines sorties. Pour de gros circuits il est impossible de tester toutes les combinaisons d'entrées possibles pour une raison évidente de coût en temps de test excessif. D'autre part, cette approche peut très bien ne pas détecter des erreurs telles qu'une consommation de courant anormale malgré un bon fonctionnement en sortie. L'approche structurelle se penche plutôt sur la qualité des éléments et des liens qui constituent le circuit. En général, chaque vecteur de test a pour but de tester un composant ou une ligne du circuit. Si tous les composants sont corrects ainsi que leurs liens, le circuit est dit sans panne et passe le test. C'est à cette étape que l'on mesure l'importance du dictionnaire de pannes pour la génération de test. En effet pour déterminer les vecteurs de test selon une approche structurelle, il faut d'abord posséder une liste des pannes susceptibles de se trouver dans le circuit à tester: c'est le dictionnaire de pannes.

La validation du test permet de valider les vecteurs de test trouvés lors de la génération du test. Cette tâche est très souvent réalisée par simulation. On peut par exemple injecter une panne dans un circuit en modifiant la description de ce circuit (liste des noeuds ou dessin des masques), puis vérifier que le vecteur de test généré provoque un comportement fautif observable.

L'application du test est l'aboutissement du travail de test. Cette tâche sera répétée sur tous les circuits à tester à certains stades du procédé de fabrication. On sera ainsi en mesure de déterminer les circuits qui présentent des pannes pour les éliminer. La qualité

des circuits distribués se trouve ainsi directement reliée à la qualité du test utilisé lors du procédé de fabrication. D'où l'importance de la génération et de la validation des tests.

Lors du design d'un circuit, la résistance aux pannes est un facteur important à considérer. Le nombre d'éléments, l'espacement entre les lignes, la disposition des polygones dans le dessin des masques, etc, sont tous des facteurs sur lesquels le concepteur peut jouer pour obtenir un circuit peu sensible aux défauts de fabrication. Le dictionnaire de fautes s'avère alors très utile puisqu'il peut fournir les endroits sensibles aux défauts ainsi que les noeuds qui risquent d'être affectés. On peut ainsi corriger la conception du circuit pour le rendre plus résistant aux défauts. Les circuits qui comporteront quand même des pannes, peuvent ensuite être éliminés par l'insertion de tests dans le cycle de production. Ces tests seront également construits grâce aux dictionnaires de pannes. Ils pourront d'abord viser les parties les plus sensibles aux défauts pour minimiser les coûts de recherche de pannes.

Ce travail décrit comment il est possible de générer automatiquement des dictionnaires de pannes. Il traite plus precisement de deux méthode de génération de tels dictionnaires. Dans une première partie, un générateur de dictionnaire de pannes qui se base sur la description schématique des circuits (liste des noeuds) est décrit. L'implémentation informatique est également présentée ainsi que certains résultats d'opération de ce premier générateur. La seconde partie décrit le fonctionnement d'un générateur de dictionnaire de pannes qui se base sur le dessin des masques des circuits. L'outil informatique développé, son algorithme et son mode d'emploi sont également commentés et des résultats sont donnés. Certaines améliorations évidentes qui pourraient être amenées à ce travail sont men-

tionnées en conclusion.

Chapitre II

REVUE DE LITTÉRATURE

2.1 Introduction

De nombreux travaux touchant, de près ou de loin, le domaine des dictionnaires de pannes ont déjà été réalisés. Ces travaux ont été d'un grand support pour ce travail. En effet ce dernier s'appuie sur beaucoup d'observations ou réalisations précédentes dans le domaine de la micro-électronique et du test de circuits intégrés (CI).

Parmi les résultats les plus intéressants, on retrouve des travaux sur l'analyse des pannes et de leurs causes, des recherches sur le test de pannes catastrophiques et analogiques, des observations sur des procédés de fabrication de CI, des statistiques sur les défauts rencontrés, leurs causes et leur effets, pour ne citer que les plus importantes références de ce travail.

La présente section décrit l'état des recherches actuelles dans les domaines reliés à la génération de dictionnaires de pannes. Une revue des informations qui ont été utilisées lors de ce travail ou qui permettent d'élargir la vision du domaine de recherche est donc fournie dans le reste de ce chapitre. Une première partie rapporte les connaissances et les techniques employées pour tester les CI. La seconde partie se penche plus particulièrement sur les sources des pannes et les connaissances relatives aux défauts de fabrication. Finalement, des travaux précédents sur la génération de dictionnaires de pannes sont présentés.

2.2 Connaissances et techniques de test

Le domaine du test a connu, ces dernières années, une croissance étonnante. Il est maintenant de mise pour tout fabricant de CI de distribuer des produits fiables, et donc testés. Rechercher une panne dans un appareil dont l'assemblage est terminé n'est plus un scénario acceptable.

Le domaine du test n'a fait qu'accentuer le fossé entre l'analogique et le numérique. En effet, les méthodes de test pour les circuits digitaux sont maintenant très avancées alors que celles qui s'appliquent aux circuits analogiques commencent à peine à voir le jour. Le regain d'intérêt pour les circuits analogiques aidant, le domaine du test de ce type de circuits vit actuellement une forte progression.

2.2.1 Test des circuits numériques

Le domaine du numérique possède plusieurs atouts qui facilitent le test des CI. Le plus important est la limitation des signaux: 0 ou 1. De cette particularité découle un avantage énorme: la simplicité du modèle de panne. Les lignes d'un circuit numérique qui fonctionne correctement peuvent prendre deux états: 0 ou 1. Une ligne est simplement fautive lorsqu'elle ne se trouve pas dans le bon état.

La méthode la plus simple pour tester un circuit porte sur sa fonctionnalité. Connaissant les sorties correctes d'un circuit, en imposant toutes les combinaisons possibles d'entrées et en vérifiant toutes les sorties obtenues, on vérifie la fonctionnalité d'un circuit. Lorsque le circuit possède un nombre d'entrées trop élevé et qu'un test complet n'est plus envisageable, une méthode courante de test est celle qui consiste à sélectionner les entrées de façons pseudo-aléatoire. On couvre ainsi, de façon aléatoire, une certaine partie de la fonctionnalité et on admet que le test est suffisant.

Les tests fonctionnels bien qu'extensivement utilisés présentent certaines lacunes. La principale étant qu'ils ne vérifient pas le coté structurel ou le fonctionnement interne du circuit. Il est très possible qu'un circuit passe un test fonctionnel avec succès alors qu'un transistor consomme un courant anormalement élevé. Une solution à ce problème a vu le jour avec le test I_{DDQ}. Il consiste non plus à observer les sorties du circuit, mais plutôt à vérifier que pour n'importe quelles entrées la consommation de courant du circuit suit un comportement normal.

Le modèle de panne le plus simple et le plus utilisé est le modèle collé-à: une ligne collé-à-0 ne peut pas être à 1; une ligne collée-à-1 ne peut pas être à 0. Sur la base de ce modèle, plusieurs algorithmes existent pour tester le circuit. Le PODEM est sûrement l'un des plus connus. On les utilise pour déterminer les vecteurs de tests nécessaires pour vérifier que toutes les lignes ne sont dans aucun des états fautifs (collé-à-1 ou collé-à-0).

2.2.2 Test de circuits analogiques

Le test de circuits analogiques présente des difficultés assez importantes. La multiplicité des paramètres à vérifier (gain, fréquence de coupure, courant, etc), le nombre presque illimité d'états possibles des lignes, la difficulté de modélisation des pannes sont autant de paramètres qui rendent le travail de test difficile.

Plusieurs méthodes ont déjà été proposées, certaines sont même déjà en application, mais beaucoup de travail reste encore à faire. Une des méthodes qui semble donner de bons résultats se base sur les sensibilités des paramètres observables par rapport aux éléments du circuit ([17]-[22]). Son principal inconvénient était, jusqu'il y a peu de temps, la difficulté de génération des sensibilités. Ce problème semble maintenant résolu selon les informations de récents articles ([8]). L'état des travaux présentés dans ces articles permet de tester des fautes analogiques et catastrophiques. Dans le cas des fautes catastrophiques

on se base non plus sur les sensibilités mais sur les gradients. On est ainsi capable de déterminer les sensibilités pour des éléments fictifs modélisant les pannes.

D'autres travaux se penchent plus particulièrement sur des types précis de pannes comme les trous (pinhole), les transistors à grille flottante ou les mauvais appariement (mismatch) entre composants ([13],[16]). Ces travaux ne sont encore qu'au stade de recherche et ne présentent pas d'application directe de leurs résultats. Certains proposent différents modèles de pannes, ou des simulations de pannes sur base de modèles statistiques.

Malgré tous ces efforts, l'unification des travaux dans le domaine du test des circuits analogiques est très loin.

2.3 Défauts et conséquences

Un certain nombre de perturbations peuvent affecter un circuit intégré. Dans tous les cas un gros effort est mis à la minimisation de leur fréquence mais on ne peut les éliminer totalement. On retrouve ces perturbations à différentes étapes du procédé de fabrication et elles peuvent dans certains cas rendre un circuit défectueux ([3]). Pour mieux mesurer les conséquences de ces perturbations, cette partie décrit les défauts les plus fréquents et leurs causes probables.

2.3.1 Sources des perturbations des procédés de fabrication

La fabrication d'un circuit intégré se décompose en un certain nombre d'étapes qui mènent finalement au circuit lui même. Les plus importantes sources de perturbations du procédé sont les suivantes:

1- Erreurs humaines et malfonctionnement de l'équipement.

2- Instabilité dans les conditions du procédé: fluctuation des conditions qui entourent le circuit intégré comme par exemple une variation de température. A cause de ces instabilités il est impossible d'obtenir deux circuits intégrés exactement identiques.

3- Instabilité des matériaux: fluctuation des paramètres physiques des composés chimiques utilisés par le procédé de fabrication. Par exemple: présence d'impuretés, variation de densité ou de viscosité des liquides utilisés ([14]).

4- Inhomogénéité du substrat et de la surface: fluctuation des propriétés du substrat dûe à des perturbations locales. Défauts localisés, dislocations ou imperfections de surface.

5- Faisceau (spot): défauts localisés principalement d'origine lithographique. L'origine du défaut peut venir de la fabrication des masques ou de leurs utilisations pendant le procédé de fabrication.

2.3.2 Inexactitudes de fabrication reliées au procédé

Un certain nombre de déformations des dessins des masques ont lieu entre le dessin et sa fabrication. Ces déformations peuvent affecter les caractéristiques électriques des circuits intégrés.

Deux types de déformations peuvent être citées:

1- Déformations géométriques: elles incluent les defauts de faisceau ainsi que les effets latéraux et verticaux . Les effets latéraux incluent les mauvais alignements de masques, la diffusion latérale, les sur ou sous-expositions lithographiques, etc. Les

effets verticaux sont des déformations en épaisseur des couches déposées, ou de la profondeur des diffusions.Les défauts de faisceau sont des défauts locaux de surplus ou de manque de matériel.

2- Déformations électriques: elles sont dues à la distribution des impuretés dans les régions conductrices ainsi qu'à la distribution des charges dans les couches d'isolation.

Toutes les déformations citées précédemment peuvent être locales ou globales. Un défaut local n'affecte le circuit intégré que dans une zone très limitée (ex: faisceau). Un défaut global affecte une grande partie du circuit intégré (ex: mauvais alignement des masques).

2.3.3 Conséquences des défauts de fabrication

Les conséquences des perturbations lors du procédé de fabrication peuvent être de deux types: paramétrique ou catastrophique.

Une faute paramétrique, aussi appellée analogique, est une déviation dans les caractéristiques électriques des composants d'un circuit. Un cas typique de faute paramétrique est une déviation d'impédance. Ces fautes sont dites douces car elle ne font que varier la sortie de façon paramétrique.

Une faute catastrophique affecte la fonctionnalité du circuit. Un cas typique de faute catastrophique est un court-circuit (short) entre deux noeuds ou un circuit ouvert (open).

La Figure 2.1 résume les différents types de défauts ainsi que leurs conséquences possibles sur le circuit intégré.

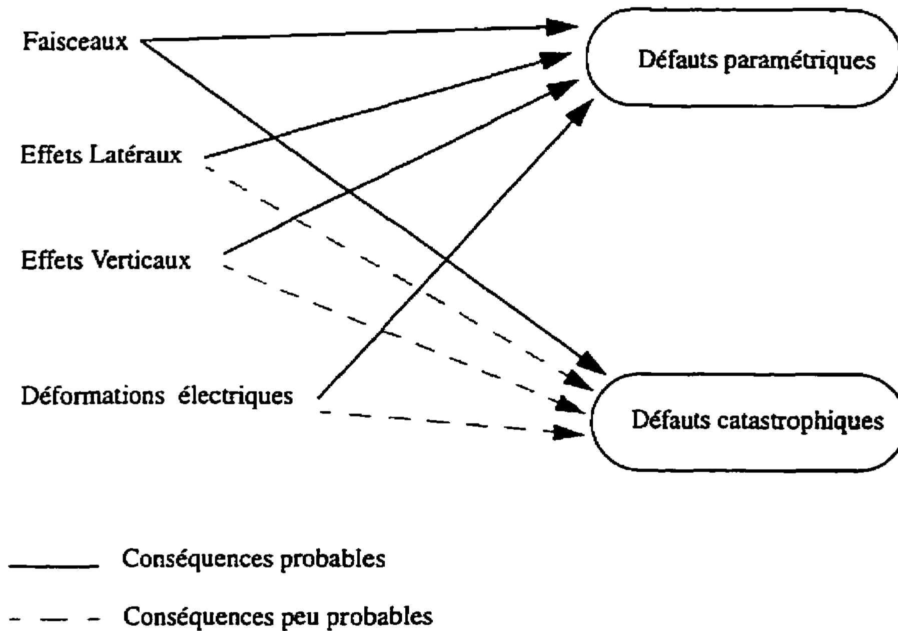


FIGURE 2.1 Défauts et leurs conséquences

Les pannes catastrophiques sont en général causées par des faisceaux. Les règles de dessin garantissent une tolérance aux autres déformations géométriques. Par exemple la largeur minimale des contacts permet une tolérance à un mauvais alignement des masques.

2.1.1 Modélisation des défauts

Le problème de la génération de dictionnaires de pannes demande une modélisation des défauts. Le travail présent ne couvre que les fautes catastrophiques, c'est pourquoi l'accent à été mis sur la modélisation des défauts causant de telles pannes: les faisceaux.

Tel que défini plus haut, un faisceau est un défaut localisé se caractérisant par un manque ou un surplus de matériaux à un endroit donné. La Figure 2.2 montre comment une

poussière sur un masque peut être à l'origine d'un tel défaut, et comment ce défaut peut entraîner une panne dans un circuit.

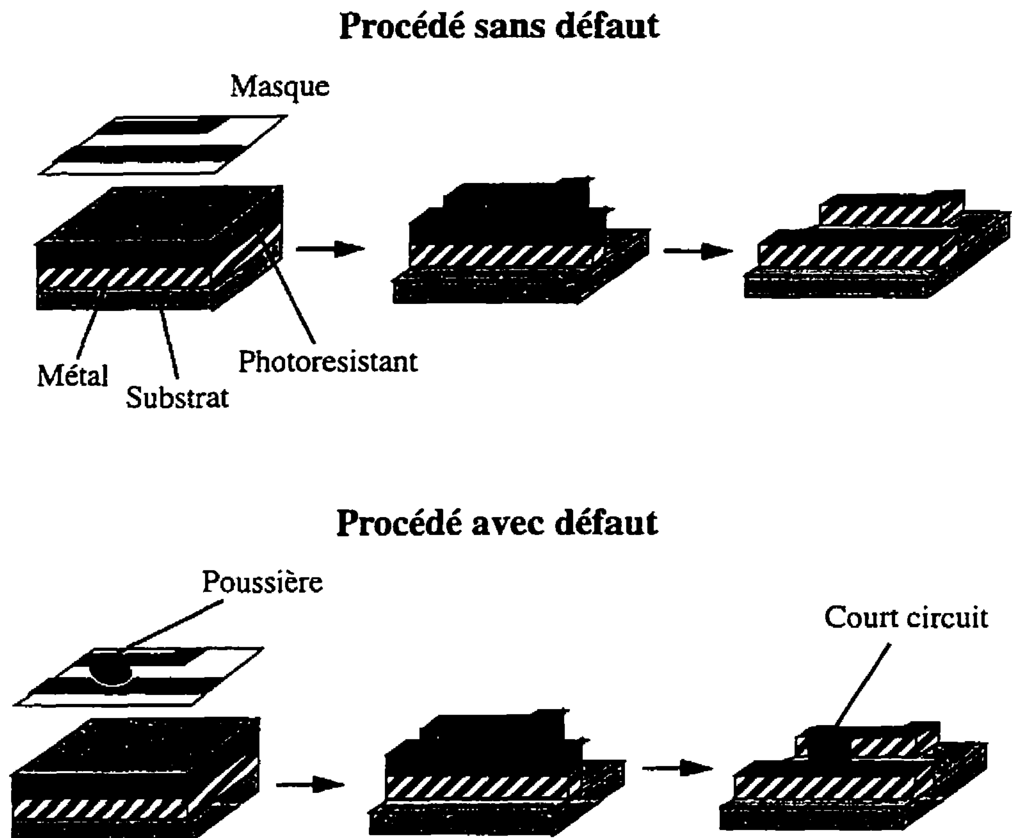


FIGURE 2.2 Défaut causant un court-circuit au cours du procédé lithographique

La façon la plus simple de modéliser de tels défauts est de considérer qu'ils sont des polygones eux-mêmes. On peut par exemple utiliser un cercle, ou un carré pour modéliser un défaut de faisceau. VLASIC ([6]) est un exemple de simulateur de pannes qui utilise un cercle comme modèle de défaut. D'autres simulateurs ([5]) préfèrent garder une forme simple comme le carré. Étant donné qu'aucune donnée ne permet de juger de la meilleure

forme de modélisation, le carré est souvent choisi pour la simplicité des fonctions d'analyse de polygones qu'il nécessite.

2.3 Les générateurs de dictionnaires de pannes

2.3.1 Généralités

2.3.1.1 Différents types de dictionnaires

Un dictionnaire de pannes doit contenir la liste des pannes qui pourraient être rencontrées dans un circuit. Cette liste permet de générer les vecteurs nécessaires au test de ce circuit. Chaque circuit possède son propre dictionnaire. Différentes versions de conception d'un même circuit peuvent également donner lieu à différents dictionnaires de fautes.

Un dictionnaire peut être exhaustif ou non. S'il est exhaustif il doit contenir toutes les pannes auxquels un circuit peut être sujet. Même les pannes les moins probables doivent en faire partie. Par exemple ce dictionnaire pourrait contenir la panne improbable suivante: tous les noeuds sont court-circuités entre eux. On peut noter que deux circuits de même topologie mais dont les dessins des masques sont différents possèdent alors le même dictionnaire de pannes. En effet, les efforts au niveau de la conception du dessin des masques ne peuvent apparaître au niveau du dictionnaire de pannes si celui-ci est exhaustif. Ceci s'explique facilement puisque ces efforts n'affectent que les probabilités des pannes sans jamais les rendre complètement impossibles. Lorsqu'on génère un dictionnaire exhaustif, on doit s'attendre à obtenir un certain nombre de pannes dont les probabilités sont tellement faibles que l'on peut raisonnablement les considérer comme impossibles. Un dictionnaire exhaustif n'est donc pas très réaliste. Sa taille dépend essentiellement du nombre de noeuds et d'éléments du circuit.

Un dictionnaire non-exhaustif donne une sélection de pannes dont la probabilité mérite que l'on y porte attention. Ce dictionnaire est un sous ensemble du dictionnaire exhaustif. Si le générateur de dictionnaire est bon il donne une liste complète de toutes les pannes qui sont susceptibles de se présenter lors de la fabrication du circuit. Cette liste est très dépendante du dessin des masques utilisé. L'analyse de ce dessin se fait en fonction des connaissances de la technologie utilisée et des statistiques des défauts rencontrés pour cette technologie. On obtient ainsi un dictionnaire réaliste. Le générateur d'un tel dictionnaire est plus complexe et le temps de génération plus important que pour un dictionnaire exhaustif.

2.3.1.2 Informations données par le dictionnaire

En général les informations que fournit un dictionnaire de pannes sont sous forme de liste de pannes. Chaque panne est décrite par son type (court-circuit, circuit ouvert) et les noeuds affectés. Dans les dictionnaires plus évolués, on retrouve également les informations suivantes: localisation de la panne sur le dessin des masques (X,Y), probabilité de la panne, la cause de la panne. Une ligne typique de dictionnaire de pannes se présente ainsi:

SHORT 10 3

ou encore

SHORT 10 3 1.5 3.2 Prob:0.13 extra polysilicon

Les deux lignes précédentes décrivent un court-circuit entre les noeuds 10 et 3. La seconde est plus précise puisqu'elle précise le type de défaut qui cause la panne (polysilicone en excès), la probabilité de rencontrer la panne ainsi que la position du défaut qui cause cette panne sur le dessin des masques (X=1.5, Y=3.2). Pour générer un dictionnaire contenant des informations aussi complètes, le générateur doit connaître les données rela-

tives au dessin des masques. La première description, plus succincte peut être obtenue à partir d'une simple analyse de la liste des noeuds.

2.3.1.3 Entrées et sorties des générateurs de dictionnaires de pannes

Les générateurs de dictionnaires de pannes les plus simples ne demandent qu'une liste des noeuds comme entrée ([11], [12]). En sortie ils fournissent le dictionnaire de pannes où chaque panne est décrite par son type (short, open) et les noeuds concernés. Certains précisent également les éléments touchés par les pannes. Par exemple *short 1 2 R1* décrit un scourt circuit entre les noeuds 1 et 2 qui touche l'élément R1.

Les générateurs de dictionnaires de pannes plus évolués se basent en général sur le dessin des masques du circuit. Ce dessin est donc l'entrée principale de ces générateurs. Il faut également leur fournir une table des défauts spécifiques à la technologie du circuit donné, et parfois les données statistiques sur les défauts (taille, distribution de probabilité, etc.).

La Figure 2.1 donne le shéma des entrées et sorties typiques des générateurs de dictionnaires de pannes.

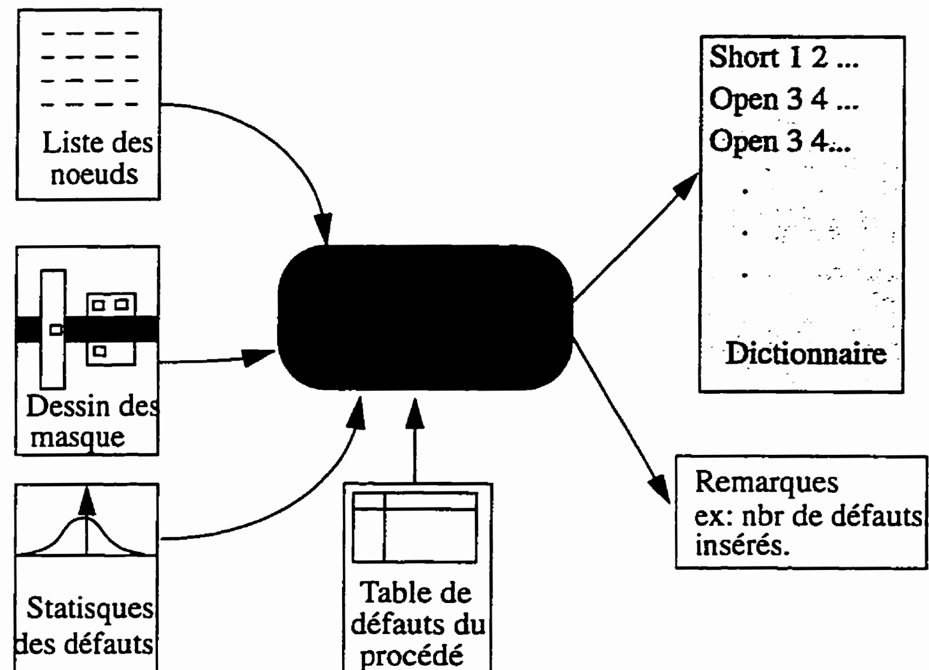


FIGURE 2.4 Entrées et sorties des générateur de dictionnaires de pannes

2.4.1 Travaux précédents

Plusieurs travaux portant sur les générateurs de dictionnaires de pannes ont déjà donné lieu à quelques articles. On peut diviser ces travaux en deux catégories principales: les générateurs qui se basent sur la liste des noeuds, les générateurs qui se basent sur le dessin des masques.

Les générateurs les plus courants sont ceux qui se basent sur la liste des noeuds([11], [12]). La simplicité de réalisation explique leur nombre. Leur fonctionnement est relativement simple puisqu'ils ne font que parcourir la liste des noeuds élément par élément en

générant les fautes qui peuvent toucher ces éléments. La difficulté de ce travail réside surtout en l'analyse syntaxique du langage de description de la liste des noeuds (ex: SPICE).

Les générateurs basés sur le dessin des masques sont beaucoup plus rares. Les informations disponibles sur ces générateurs ne sont en général pas accessibles car ce sont des outils développés par des compagnies pour leurs besoins. Ces outils ne sont donc ni publics, ni commercialisés. A notre connaissance, VLASIC [6] est le seul logiciel actuellement disponible pour générer des dictionnaires de pannes à partir de dessins de masques. Cet outil est le résultat d'un travail universitaire. Malheureusement, il est très difficile d'utiliser ce logiciel vu la pauvreté du manuel d'utilisation offert [10] et le format de dessins des masques demandé (ancienne version de Cadence). Il existe bien des articles sur certains autres logiciels tel que celui décrit dans [5] mais ces travaux ne sont encore qu'au stade de recherche.

L'état actuel de ces logiciels ne répondant pas aux exigences de notre groupe de recherche (groupe sous la direction de Mme Bozena Kaminska), ce travail vient remédier au manque d'outils disponibles pour la génération de dictionnaires de pannes. Il est malheureusement très difficile, pour des raisons d'accessibilité, de comparer les outils déjà existants. Ceci aurait pourtant permis une concentration plus rapide vers un travail de qualité.

Chapitre III

DICTIONNAIRE EXHAUSTIF

3.1 Introduction.

Un dictionnaire exhaustif doit contenir toutes les pannes qui pourraient exister dans le circuit étudié, y compris les moins probables. Aucune considération de dessin des masques n'est nécessaire. Les seules informations utiles sont les éléments qui composent le circuit ainsi que les liens qu'ils possèdent entre eux. Toutes ces informations sont contenues dans la liste des noeuds (netlist).

3.2 Génération de dictionnaire basée sur la liste des noeuds

3.2.1 Méthode des noeuds

Lorsque l'on connaît les noeuds qui composent un circuit, il est facile de générer un dictionnaire exhaustif. En effet il suffit de générer toutes les combinaisons possibles de circuits ouverts et de court-circuits entre ces noeuds. Par exemple le diviseur de tension de la figure 1 comporte trois noeuds. Ces trois noeuds peuvent être ouverts; il existe également quatre combinaisons de court-circuits possibles entre ces noeuds. Avec trois noeuds on obtient un dictionnaire de sept pannes. Ce dictionnaire est exhaustif.

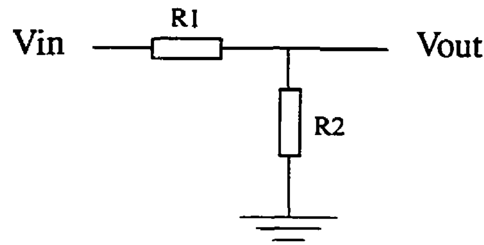


FIGURE 1.1 Diviseur de tension.

Il est possible de donner une formule générale reliant le nombre de pannes N au nombre de noeuds n du circuit:

$$N = n + \sum_{n}^{i=2} A_n^i = \sum_{n}^{i=1} A_n^i \qquad (3.1)$$

L'implantation informatique d'un tel générateur est relativement simple puisqu'il s'agit de traitement de listes. En effet l'entrée du programme est la liste des noeuds. Et la sortie est une liste de pannes. Par exemple pour le circuit de la figure 1.1 l'entrée serait la liste {gnd in out} et la sortie {open gnd, open in, open out, short {gnd in}, short {gnd out}, short {in out}, short {gnd in out}}. On obtient ainsi un dictionnaire grâce à un programme de manipulation de liste. Le travail le plus long n'est plus de générer le dictionnaire de fautes, mais la liste des noeuds du circuit.

3.2.2 Arbre de modélisation de la liste des noeuds (netlist)

Obtenir la liste des noeuds d'un circuit à partir de sa description schématique n'est pas toujours évident. Le travail qui suit vaut pour des liste de noeuds de format SPICE. Le choix de SPICE a été fait pour sa popularité et son utilisation généralisée. Il serait très facile d'étendre ce travail à d'autres logiciels (SABER, etc) mais ce n'est pas ici le but

recherché. SPICE est un logiciel de simulation de circuit électrique. Le format de description de la liste de noeuds est assez avancé et permet d'entrer de gros circuits à un effort minimum. L'utilisation de sous-circuits (subcircuit) et de macros facilite la description de circuits répétitifs. Pour des circuits numériques par exemple, on peut définir des macros pour chaque type de portes (AND, NAND, OR, etc). En plus d'éviter de redéfinir plusieurs fois des composantes similaires, on obtient des descriptions claires et lisibles. Il est également possible d'imbriquer plusieurs macros ou sous-circuits l'un dans l'autre. On peut alors jouer sur la visibilité de certains morceaux de circuit. La figure 1.2 donne un exemple de l'importance de cette propriété.

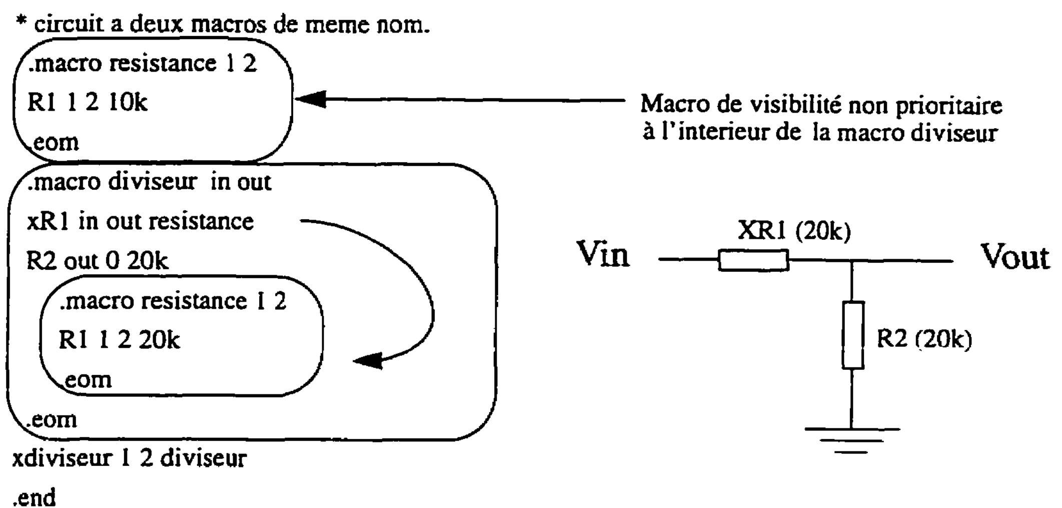


FIGURE 1.2 Macro et visibilité

Si l'utilisation des macros et des sous-circuits rend le format SPICE très pratique, il complique le parcours automatique de la liste des noeuds (netlist) et la génération de la liste des noeuds. En effet, il n'existe plus de relation précise entre la taille de la liste des noeuds (netlist) et le nombre de noeuds du circuit décrit. De plus, les références à des noeuds internes à certaines macros sont elles aussi difficiles à suivre de façons automati-

que étant donnés que deux noeuds dans différentes macros peuvent posséder le même nom. Pour toutes ces raisons, la première étape du parcours automatique d'une liste de noeud est la construction de l'arbre qui modélise le circuit. La figure 1.3 montre l'arbre de modélisation du circuit de la figure 1.2.

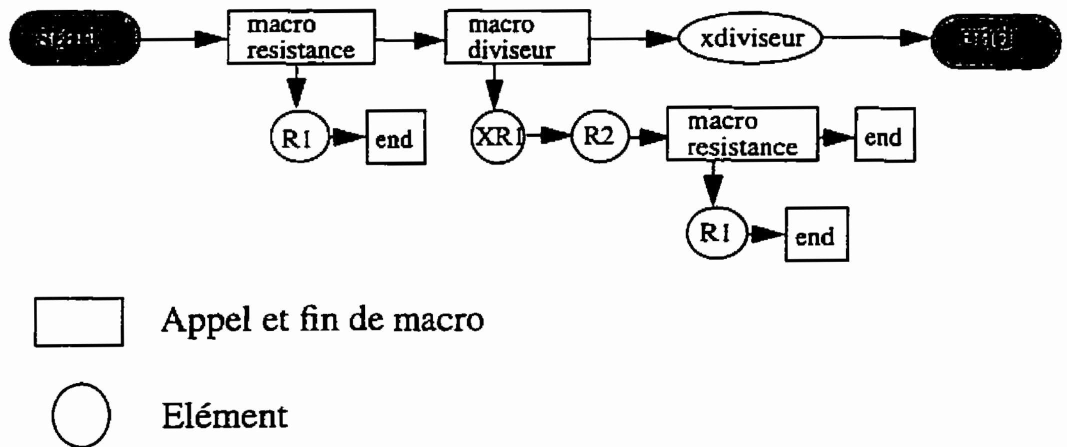


FIGURE 1.3 Arbre de modélisation

Un arbre de modélisation a une structure de liste qui peut être réalisée et gardée en mémoire grâce à l'utilisation de pointeurs et de structures lors de l'implantation informatique. Chaque élément de l'arbre peut ainsi comporter non seulement un composant du circuit, mais également toutes les valeurs et propriétés qui lui sont propres. On peut ainsi trouver dans l'arbre les éléments du circuit, les noeuds auxquels ils se connectent, la valeur des éléments (ex: valeur de la résistance, gain de la source contrôlée, etc), la visibilité des macros et sous circuits, le chemin (path) des noeuds, etc.

Lorsque l'arbre est créé, on peut parcourir le circuit comme si celui-ci était non-hiérarchique (flat) et lire les éléments l'un après l'autre, de façon simple et rapide. Il s'agit en

fait de parcourir l'arbre branche par branche. A partir de ce moment, toute analyse du circuit est grandement facilitée.

La première utilisation qui peut être faite de cet arbre est la génération de la liste des noeuds du circuit. La figure 1.4 montre comment il est facile d'obtenir cette liste. On obtient ainsi les données nécessaires à la génération du dictionnaire de pannes selon la méthode exposée dans la section 3.2.1 . Notons que les portions non utilisées du circuit ne sont pas parcourues. Dans l'exemple de la figure 1.4 la macro diviseur du premier niveau n'est donc jamais parcourue puisque cette partie du circuit n'est jamais appelée.

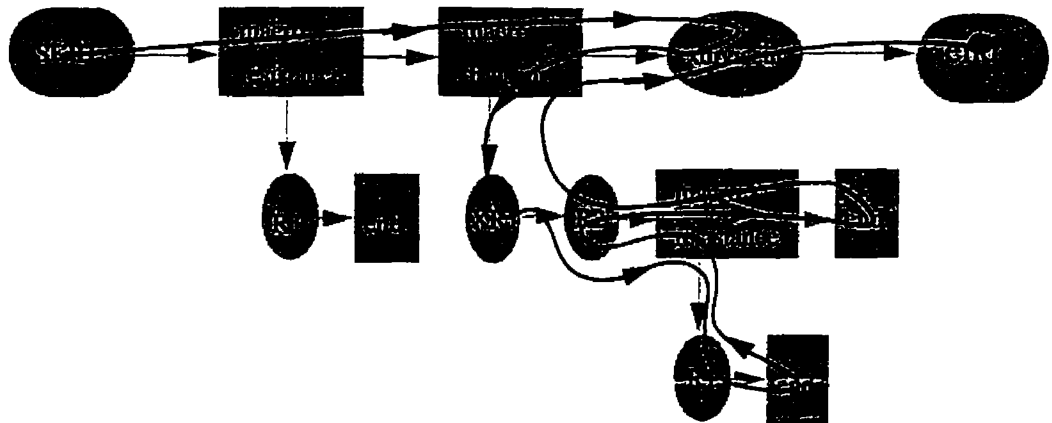


FIGURE 1.4 Génération de la liste des noeuds d'un circuit

3.2.3 Méthode des éléments

Un des désavantages de la génération de dictionnaire selon la section 3.2.1 est son manque de flexibilité. Elle ne se base que sur les noms des noeuds et fournit la liste de toutes les combinaisons de noeuds pouvant constituer une panne. On peut effectivement jouer sur certains paramètres (ex: ne pas générer la liste des courts-circuits qui impliquent plus que deux noeuds), mais il manquera toujours certaines informations importantes. Parmi

elles, le nom des éléments sujets aux pannes, leur type, etc. Ce genre d'informations peuvent s'avérer très utile, voir indispensable lors de l'utilisation d'un dictionnaire de pannes. Il est souvent tout aussi important de connaître l'élément touché que les noeuds touchés par une panne. Khaled Saab par exemple, pour générer les vecteurs de test par la méthode du gradient [7] dans le cas de circuits ouverts, doit connaître le nom des éléments touchés pour mesurer les courants dans ces éléments. D'autre part un noeud peut être ouvert à différents endroits du circuit. Le nom de l'élément auquel ce noeud se rapporte devient alors important pour localiser plus précisement la panne. Ce problème est illustré à la figure 1.5. Sur cette figure on comprend comment le nom du noeud touché peut, dans certains cas, être une information insuffisante pour situer une panne dans un circuit.

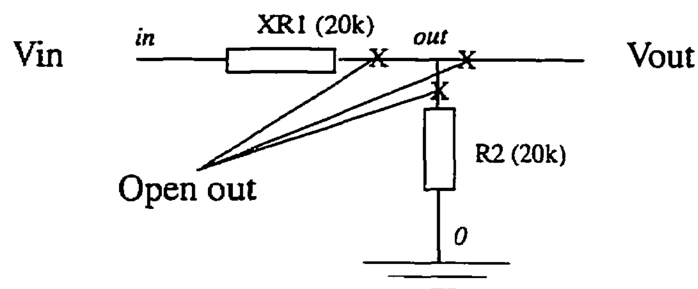


FIGURE 1.5 Différentes localisations de fautes pour un même nom de noeud fautif.

En pratique, des cas similaires à celui de la figure 1.5 se retrouvent très fréquemment. On comprend alors la nécessité de générer des dictionnaires de pannes plus complets au niveau de l'information donnée pour chacune des pannes.

Une solution à ce problème consiste à générer le dictionnaire de pannes au moment du parcours de l'arbre de modélisation du circuit traité (section 3.2.2). Chaque élément rencontré dans l'arbre est sujet à un certain nombre de pannes: ses noeuds sont ouverts, ses noeuds sont court-circuités, etc. Ces pannes sont spécifiques à chaque élément et sont présentées dans le tableau 1.

Élément / modèle	Pannes
Résistance R A B	open A open B short A B
Inductance L A B	open A open B short A B
Condensateur C A B	open A open B short A B
Source: tension ind. V A B tension dép. G A B F A B courant ind. I A B courant dép. E A B H A B	open A open B short A B
Diode D A B	open A open B short A B
Transitor BJT Q B C E	open C open B open E short BC short CE short BE
Transistor Mosfet M G S D Transistor JFET J G S D	open G open S open D short GS short SD short GD

TABLEAU 1 Pannes associées à chaque élément

L'utilisation du tableau 1 lors du parcours de l'arbre présenté à la figure 1.3 pour le circuit de la figure 1.1 donnerait le dictionnaire suivant:

1 Open xdiviseur.xR1.1 xdiviseur.xR1.R1
2 Open xdiviseur.xR1.2 xdiviseur.xR1.R1
3 Short xdiviseur.xR1.1 xdiviseur.xR1.2 xdiviseur.xR1.R1
4 Open xdiviseur.out xdiviseur.R2
5 Open xdiviseur.0 xdiviseur.R2
6 Short xdiviseur.out xdiviseur.0 xdiviseur.R2

Chaque ligne décrit une panne par son type, le noeud ouvert, ou les noeuds court-circuités et l'élément touché par cette faute. Par exemple, la ligne 3 décrit un court-circuit entre le noeud *xdiviseur.xR1.1* et le noeud *xdiviseur.xR1.2* qui affecte l'élément *xdiviseur.xR1.R1*.

Pour être exhaustif, il manque à ce dictionnaire un certain nombre de court-circuits. En effet, les seuls courts-circuits générés de cette façon sont ceux qui touchent les différents noeuds d'un même élément. Il est tout à fait possible de compléter ce dictionnaire par des combinaisons de courts-circuits issues de la méthode des noeuds.

On peut noter que le dictionnaire donné plus haut possède certaines redondances qui peuvent être éliminées tout en gardant toutes les informations utiles. Par exemple, plusieurs circuits ouverts touchent les mêmes noeuds tout en étant issus d'éléments différents. On peut obtenir un dictionnaire plus concis en éliminant ces redondances.

3.3 Implantation informatique et résultats de simulation

3.3.1 Implantation informatique

Le générateur de dictionnaire de pannes décrit dans la section 3.2.3 et utilisant l'arbre de modélisation de liste de noeuds de la section 3.2.2 à été programmé en C sur station

SUN. Le programme au complet est donné en annexe 1. Il se divise en plusieurs parties qui décrivent l'algorithme général du circuit. La figure 1.6 montre un schéma de cet algorithme.

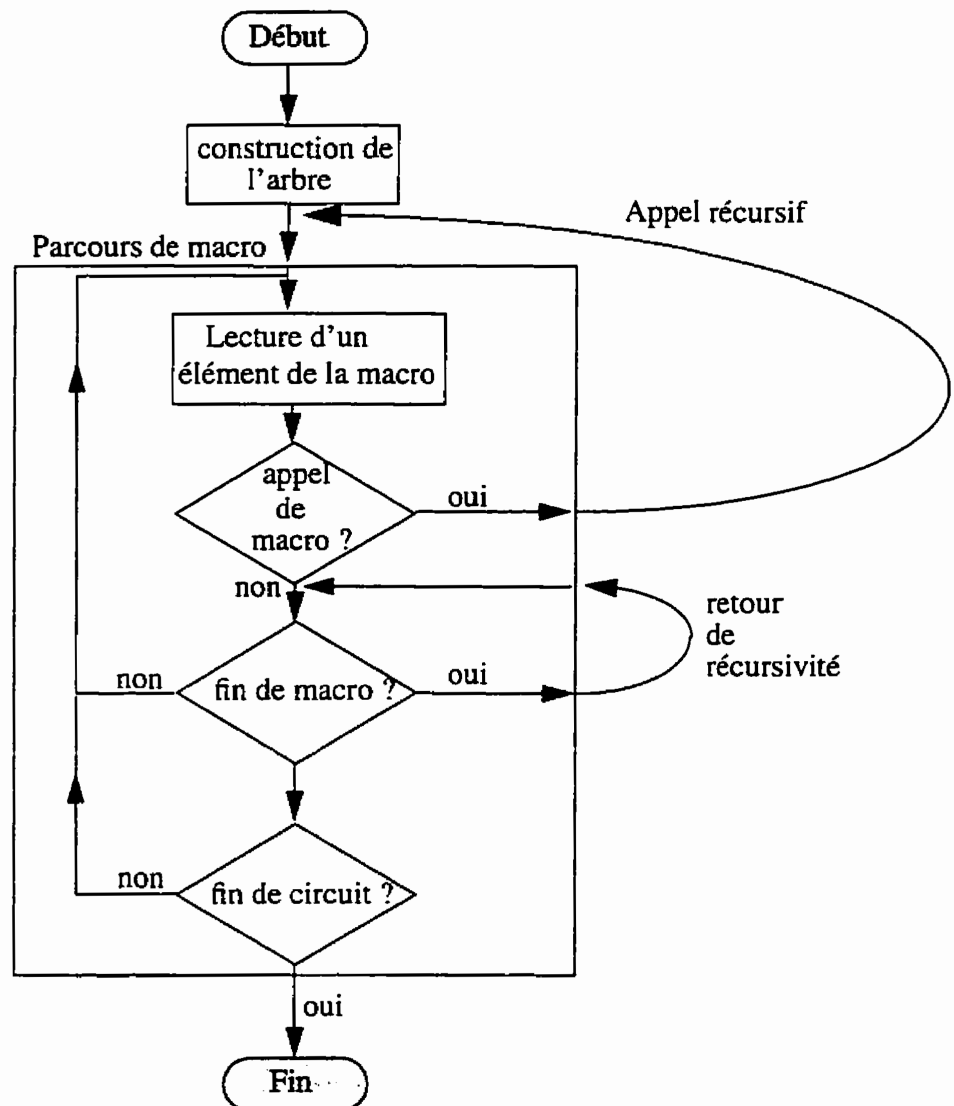


FIGURE 1.6 Algorithme général de parcours de l'arbre de modélisation de la liste de noeuds

La construction de l'arbre est la première étape indispensable à toute analyse de liste de noeuds. L'arbre est construit en mémoire. L'élément de base de l'arbre est la structure suivante (figure 1.7):

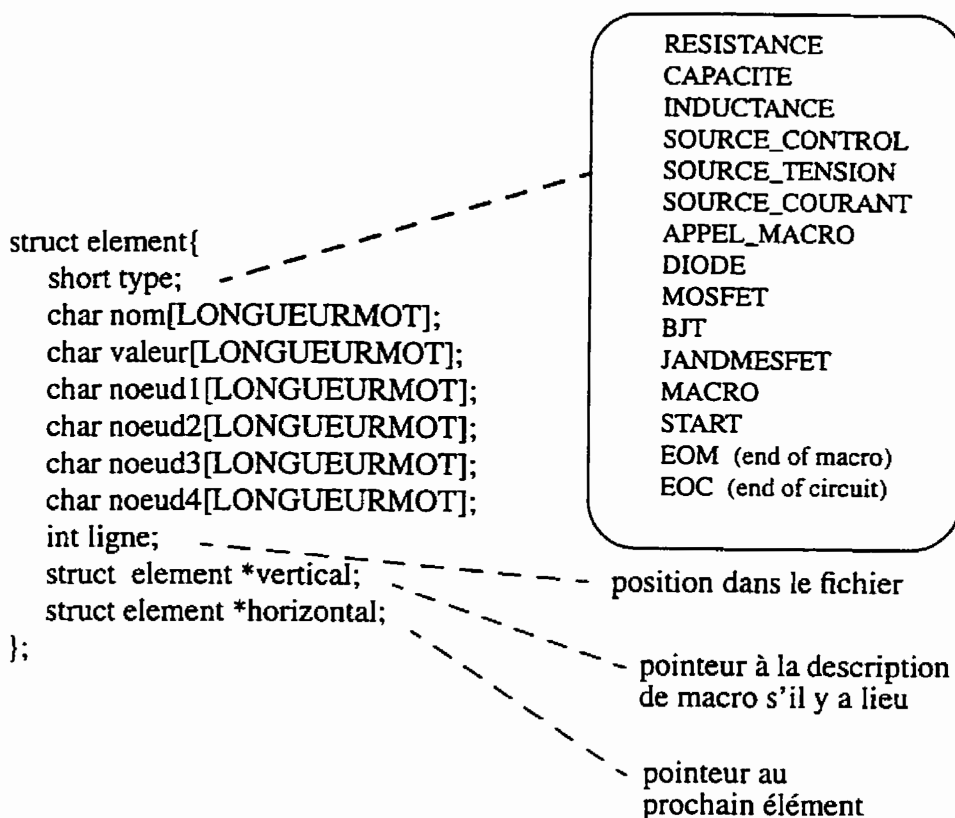


FIGURE 1.7 Structure d'un élément de l'arbre.

Grâce à cette structure chaque élément de l'arbre contient un composant du circuit, avec toutes les informations qui lui sont rattachées, et les liens de cet élément avec les autres éléments de l'arbre. Les deux derniers champs de la structure permettent de relier les éléments entre eux dans l'arbre. Un lien horizontal (*struct element *horizontal;*) mène au prochain élément ou sous-circuit de même niveau. Un lien vertical (*struct element*

**vertical;*) mène au début d'une macro de niveau supérieur. Le champ *ligne* sert à conserver la place de l'élément dans le fichier texte de la liste des noeuds.

La façon la plus simple de parcourir ce type d'arbre est par un algorithme récursif. La logique de l'algorithme qui a été utilisée est la suivante. Une procédure est utilisée pour parcourir la macro de niveau initial du circuit (niveau 0). Elle commence au premier élément de l'arbre et se déplace horizontalement d'élément en élément pour générer les pannes associées à chaque composant rencontré. Lorsqu'un appel de macro ou de sous-circuit est rencontré, la procédure s'appelle elle même en considérant la feuille du début de la macro appelée première feuille de l'arbre; c'est la récursivité; il s'agit d'un déplacement vertical dans l'arbre. Quand cette macro de niveau 1 est parcourue, l'appel récursif est terminé et le parcours du niveau 0 continue. Selon ce procédé on peut parcourir tout l'arbre en garantissant le respect des visibilités et des priorités d'appel de macro. Le parcours se termine lorsque la feuille de fin de niveau 0 est atteinte. Toutes les pannes ont alors été générées et le dictionnaire est terminé.

La dernière chose à faire lorsque le dictionnaire est complété est d'éliminer les redondances de la liste de pannes. Il existe presque toujours des redondances. En effet le simple fait que deux éléments aient un noeud en commun suffit à entraîner deux circuits ouverts du même noeud. On peut alors concaténer les deux pannes en une seule, l'information sur l'élément touché devient maintenant une liste d'éléments touchés par la même panne. On peu traiter de la même façon tous les court-circuits pour finalement obtenir un dictionnaire de pannes sans redondance.

3.3.2 Analyse de complexité

La complexité des générateurs de dictionnaire basés sur la liste des noeuds est relativement restreinte. Ceci permet de garder des temps de générations courts même lorsque la complexité des circuits traités augmente.

Le cas de la génération de dictionnaire par la méthode des noeuds est le plus simple. Il s'agit d'un traitement de liste. La complexité de cette génération est donc de type n, ou n est le nombre de noeuds du circuit.

Le cas de la génération de dictionnaire par la méthode des éléments est similaire. Cette fois, la liste traitée n'est plus une liste de noeuds, mais d'éléments. La complexité est alors de type n, ou n est le nombre d'éléments du circuit.

Dans les deux cas, la complexité reste faible ce qui permet de comprendre pourquoi ces méthodes s'étendent très bien à des circuits de grandes tailles. Ici la taille d'un circuit est définie comme étant proportionnelle au nombre de noeuds (pour la méthode des noeuds) ou d'éléments (pour la méthode des éléments) qu'il comporte.

3.3.3 Exécution et résultats de simulation

Le programme s'appelle *"faultlist"*. On doit lui fournir le nom du fichier texte contenant la liste des noeuds du circuit pour lequel on désire le dictionnaire de pannes. Par exemple, si le fichier de liste de noeuds est *"Netlist.Net"*, on génère le dictionnaire en faisant *"Faultlist Netlist.Net"*

Voici le type d'informations qui apparaissent lors de l'exécution du programme pour un diviseur de tension:

fichier du circuit:
Netlist.Net comme circuit de référence

```
nom du fichier contenant la liste intermediaire: Netlist.tmp
nom du fichier contenant la liste finale de fautes: Netlist.flt

construction de l'arbre
* Diviseur de tension
.option post
Vin in 0 dc 0
r1 in out   2k
r2 out 0 2k
.end

initialisation de la liste des macros
parcours de la macro main,
 =Vin=r1
 =r1=r2
 =r2=eoc

 elimination de la redondance...
open  in Vin
open  0 Vin
short in 0 Vin
open  in r1
open  out r1
short in out r1
open  out r2
open  0 r2
short out 0 r2
done

number of listed fault              :9
number of redundant fault eliminated :3
```

On observe bien les trois phases de l'algorithme présenté à la figure 1.6. Voici la liste des pannes obtenues lors de cette exécution:

7 open in Vin r1

8 open 0 Vin r2

9 short in 0 Vin

10 open out r1 r2

11 short in out r1

12 short out 0 r2

Il reste bien six fautes: neuf moins les trois redondances. La première ligne, par exemple, doit se comprendre ainsi: circuit ouvert au noeud *in* touchant les éléments *Vin* et *R1*. La cinquième, elle, décrit un court-circuit entre les noeuds *in* et *out* qui court-circuite l'élément *R1*.

Le programme peut très bien traiter des circuits de tailles plus importantes. Il a été utilisé pour générer le dictionnaire de pannes du circuit de la figure 1.8.

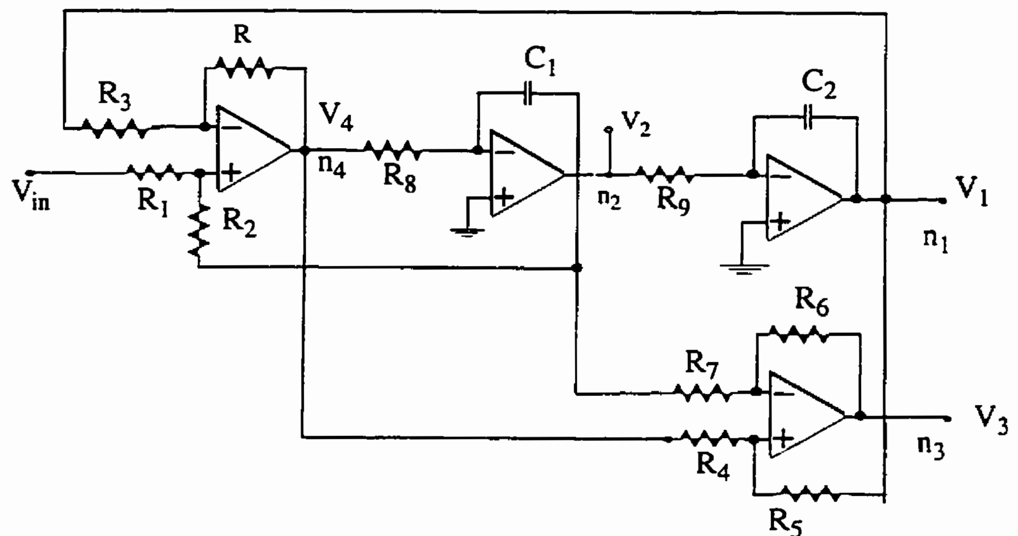


FIGURE 1.8 Filtre à état variable

Voici un extrait du dictionnaire obtenu

1 open xOPAMP4.3 Iopen2

2 short xOPAMP4.3 xOPAMP4.0 Iopen2

3 open 6 C1 R8

4 open 7 C1 R2 R7 R9

5 short 6 7 C1

6 open 8 C2 R9

7 open 2 C2 R3 R5

8 short 8 2 C2

9 open 1 R1 V1

10 open 3 R1 R2

11 short 1 3 R1

La totalité du dictionnaire obtenu pour ce filtre comporte 75 pannes dont 22 redondances:

number of listed fault :75

number of redundant fault eliminated :22

Il existe également une version du programme qui permet de spécifier le niveau maximum de recherche des pannes. On peut ainsi éviter de générer les pannes pour des éléments à l'intérieur de certaines macros qui pourraient être sans intérêt. Le programme qui permet ceci se nomme *dictio.c*.

La commande suivante génère le dictionnaire de pannes pour le circuit *state.cir* sans s'occuper des éléments de niveaux supérieurs à zéro: *dictio state.cir 0*.

On obtient ainsi un dictionnaire excluant les pannes touchant les éléments internes des macros des amplificateurs opérationnels.

3.4 Conclusion

La façon la plus simple de générer un dictionnaire de pannes est de générer toutes les combinaisons de noeuds possibles du circuit. Cette méthode présente l'avantage d'être simple et donne un dictionnaire exhaustif. Néanmoins elle ne permet pas de générer les informations importantes relatives à chaque panne telles que les éléments touchés. Ces

informations peuvent parfois être très utiles pour générer des vecteurs de test ou localiser plus précisément la panne dans le circuit. Une méthode de parcours d'arbre de modélisation de liste de noeuds à également été présentée. Dans les deux cas il s'agit de génération de dictionnaire exhaustif qui ne tient pas compte de la probabilité d'occurrence des pannes générées.

Chapitre IV

DICTIONNAIRE REALISTE

4.1 Introduction

Il est possible de générer un dictionnaire de pannes de façon plus réaliste que le font les générateurs présentés dans le chapitre III. Il faut pour cela tenir compte de plus d'informations que ne peut donner la liste des noeuds d'un circuit. Les données sur le procédé de fabrication du circuit, l'espacement des lignes, les statistiques de défauts, etc, ont toutes une influence sur la résistance aux pannes des circuits et par conséquent sur les dictionnaires de pannes qui s'y rattachent. La plupart de ces informations se trouvent dans le dessin des masques. Ce qui suit présente donc un générateur de dictionnaire réaliste qui donne une liste des pannes probables basée sur le dessin des masques des circuits.

4.2 Le générateur de dictionnaire réaliste

4.2.1 Algorithme général

Le générateur de dictionnaire réaliste se base sur la description des polygones formant le dessin des masques et le fichier de technologie d'un circuit. Il doit trouver l'ensemble des pannes susceptibles d'être entraînées par l'apparition de certains défauts.

La figure 1.9 donne un shéma général du fonctionnement d'un générateur de dictionnaire réaliste. La génération se fait de façons itérative. A chaque itération, un défaut est inséré dans le dessin des masques et l'interaction de ce défaut avec les polygones du cir-

cuit est analysée pour déduire les pannes entraînées. Lorsque tout les défauts ont été insérés, la génération du dictionnaire est complétée.

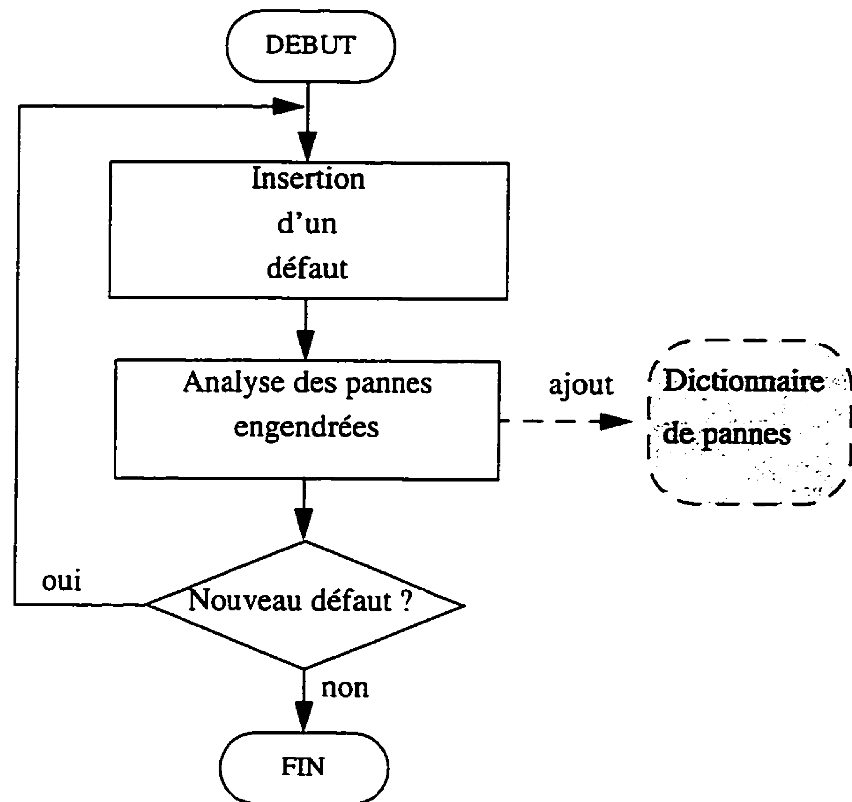


FIGURE 1.9 Fonctionnement général d'un générateur de dictionnaires réalistes

La génération du dictionnaire se divise en deux blocs principaux: un bloc d'insertion de défauts et un bloc d'analyse des pannes engendrées. Le bloc d'insertion de défauts a pour tâche d'injecter dans le dessin des masques tous les défauts qui peuvent affecter le fonctionnement du circuit. Le bloc d'analyse des pannes engendrées doit déterminer les pannes qui surviennent en présence de chaque défaut inséré.

La performance d'un générateur de dictionnaire est dépendante de l'efficacité de ces deux blocs. En effet, si l'insertion des défauts est mal faite, un certain nombre de pannes peuvent ne pas être données dans le dictionnaire. De même, une mauvaise analyse des pannes engendrées mène à un dictionnaire peu réaliste. D'autre part, les algorithmes d'insertion et d'analyse sont ceux qui déterminent la complexité de la génération du dictionnaire.

4.3 Insertion de défauts

4.3.1 Algorithme d'insertion

L'insertion des défauts pourrait en soit constituer un travail de recherche. La façon la plus directe d'insérer des défauts est de commencer par un coin du circuit puis de se déplacer vers le coin opposé selon un chemin qui assure que tout le circuit aura été couvert de défauts. Dans ce cas, le déplacement en abscisse et en ordonnée du défaut peut avoir un effet sur la couverture finale du circuit. D'autre part dans les zones où le dessin des masques ne présente aucun polygone, des défauts sans conséquences catastrophiques sont alors insérés inutilement.

D'autres techniques plus efficaces peuvent être utilisées. On peut, par exemple, déterminer en premier lieu les zones sensibles du circuit en procédant à une analyse de la répartition de densité des polygones, ou des distances entres eux. Une insertion plus minutieuse de défauts peut alors avoir lieu dans ces zones. On concentre ainsi l'effort de génération de dictionnaire sur les zones sensibles, tout en passant plus rapidement sur les zones plus robustes du circuit.

4.3.2 Modélisation des défauts

On peut modéliser tout les types de défauts présentés dans la section 2.3. Les défauts les plus souvent traités sont les défauts de faisceau (spot defects). On peut les représenter par un carré, un polygone, ou encore un cercle. Le choix du modèle tend à respecter la réalité. Il est très difficile de juger de la meilleure forme de modélisation pour ces défauts et de l'influence sur le dictionnaire généré.

Il est important de noter que le modèle choisi influence énormément la complexité de l'analyse des interactions entre le défaut et les polygones du circuit. Le choix du modèle doit donc tenir compte non seulement de son réalisme, mais également des performances générales que devrait avoir l'outil de génération de dictionnaire (temps de génération, complexité, etc.).

4.4 Analyse des pannes engendrées

4.4.1 Interactions défauts-polygones

Quand un défaut apparaît dans un circuit, il peut interagir de différentes façons avec les couches du circuit. Définir le type d'interaction que présente chaque défaut inséré avec les polygones qui forment le dessin des masques est la première étape de l'analyse des pannes.

Il existe quatre interactions différentes possibles. La première n'est mentionnée qu'à titre d'information puisqu'elle n'a aucune conséquence de type catastrophique sur les circuits intégrés. Il faut toutefois savoir qu'elle peut intervenir sous forme de pannes paramétriques (capacités parasites, variations de résistance, etc). En effet, un défaut, même s'il n'entre en contact avec aucun polygone du circuit, peut être la cause de fautes paramétri-

ques altérant le comportement du circuit. Le générateur présenté ici ne rend pas compte de ce type de pannes.

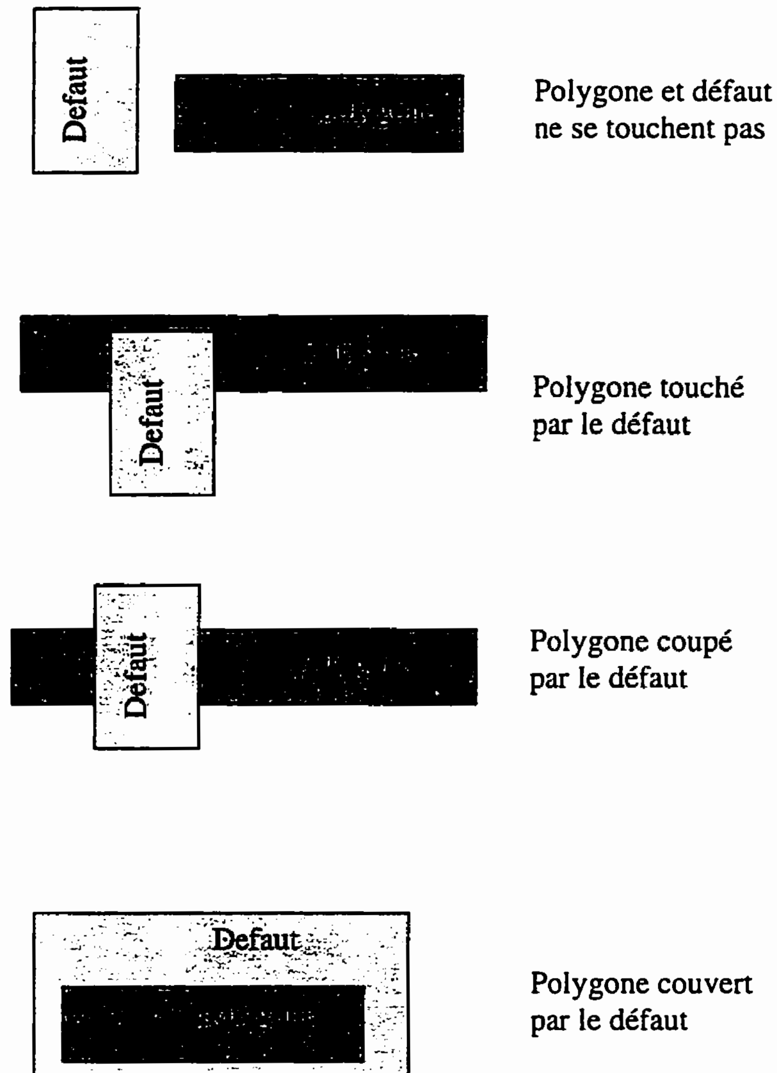


FIGURE 1.10 Interactions défaut-polygone

4.4.2 Analyse des pannes en CMOS

A partir des observations réalisées lors de la fabrication des circuits intégrés, et en se servant des connaissances sur les défauts et leurs conséquences (section 2.3), on peut créer une table de défauts et pannes pour chaque procédé de fabrication. Cette table permet de relier chaque défaut avec la ou les pannes entraînées selon la situation de leur apparition.

Ce travail a été réalisé pour la technologie CMOS. Le résultat est le tableau 2. Il permet l'analyse des défauts pour trouver les pannes associées en CMOS.

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Excès de métal	métal&métal	métal	touche	court-circuit
Manque de métal	métal contact&poly contac&polyt	métal métal métal	coupe couvre couvre	circuit-ouvert circuit-ouvert circuit-ouvert
Excès de contact	métal&poly métal&difusion	contact contact	touche touche	court-circuit court-circuit
Manque de contact	contact&poly contact&difusion	contact contact	couvre couvre	circuit-ouvert circuit-ouvert
Excès de polysilicon	poly&poly difusion difusion contact contact&difusion	poly poly poly poly poly	touche coupe couvre touche couvre	court-circuit circuit-ouvert circuit-ouvert court-circuit circuit-ouvert
Manque de polysilicone	poly contact&poly	poly poly	coupe couvre	circuit-ouvert circuit-ouvert
Excès de n+ implant	difusion p difusion p contact&dif. p	n+ n+ n+	coupe couvre couvre	circuit-ouvert circuit-ouvert circuit-ouvert

Type de défaut	Couche des polygones	Couche du défaut	Interaction défaut-polygones	Panne entraînée
Manque de n+	difusion n	n+	coupe	circuit-ouvert
	difusion n	n+	couvre	circuit-ouvert
	contact&dif. n	n+	couvre	circuit-ouvert
Manque de p+	difusion p	p+	coupe	circuit-ouvert
	difusion p	p+	couvre	circuit-ouvert
	contact&dif. p	p+	couvre	circuit-ouvert

TABLEAU 2 Interaction défaut-polygones-pannes en CMOS

Ce tableau permet de déduire les types de pannes entraînées par un défaut. Il faut pour cela connaître le type de défaut (première colonne), les couches touchées sur le dessin des masques (deuxième et troisième colonnes), et la façon dont le défaut touche les polygones (quatrième colonne).

4.4.2.1 Exemple

La première ligne du tableau se lit comme suit: un défaut d' "EXTRA METAL" qui TOUCHE un polygone de METAL et un autre polygone de METAL crée un SHORT entre ces deux polygones. Cette panne est illustrée la figure 1.11.

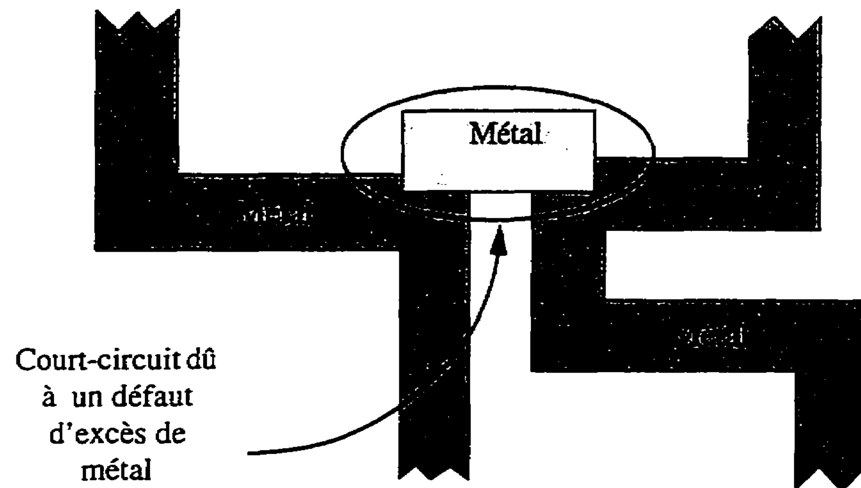


FIGURE 1.11 Exemple de défaut et sa panne associée

On peut ainsi analyser la majorité des défauts pour un circuit CMOS (tableau 2). Le type de pannes rencontrées dépend souvent du type d'interactions polygones-défauts. Un exemple de ce phénomène est donné à la figure 1.12. Bien que dans les deux cas présentés sur cette figure il y ait interaction défaut-polygone, seul un des deux défauts est la cause d'une panne catastrophique.

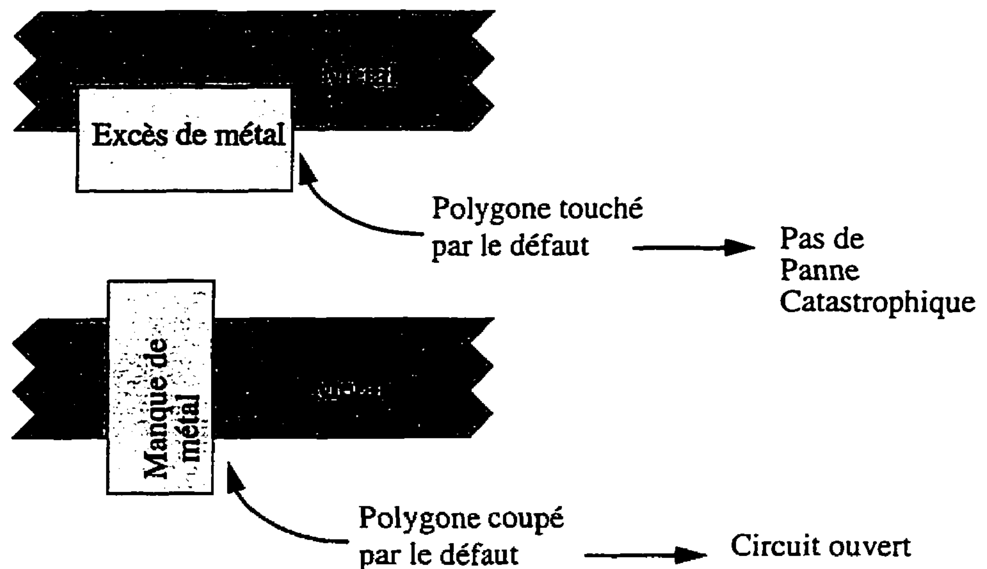


FIGURE 1.12 Importance du type d'interaction défaut-polygones

4.5 Implantation informatique

4.5.1 CADENCE et le SKILL

Le dessin des masques d'un circuit est l'ensemble des données qui décrivent les couches (métal, polysilicone, etc) formant ce circuit. Il existe plusieurs logiciels permettant de saisir les informations relatives au dessin des masques pour ensuite extraire les données nécessaires à la fabrication des masques. Le logiciel le plus couramment utilisé est sûrement CADENCE. Son interface graphique et l'ensemble des outils qu'il propose en fait un outil très pratique pour la conception de circuits.

Le premier problème de génération de dictionnaires de fautes est l'accès aux données des dessins des masques. Il existe en effet plusieurs formats de stockage de ces données. Les plus connus sont les fichier .CIF, .GIF. Quel format choisir? Comment retrouver les

données pertinentes pour le générateur de dictionnaire? Comment y accéder de façon rapide? Toutes ces questions se sont posées aux début de la conception de l'outil. La solution qui a finalement été retenue pour ce travail est la suivante: créer un outil qui travaille dans l'environnement CADENCE en utilisant les fonctions déjà existantes de ce logiciel. C'est pourquoi le générateur de dictionnaires à été réalisé en SKILL, sous l'environnement CADENCE.

Le SKILL est un langage de programmation particulier à CADENCE qui permet l'accès a sa base de données. La programmation en SKILL offre de nombreux avantages. Elle permet entre autres un accès facile et transparent aux données de n'importe quel dessin de masques. Le SKILL offre également un bon nombre de fonctions préprogrammées permettant l'analyse des interactions entre polygones. Tout le travail de gestion des données est ainsi grandement simplifié.

L'outil de génération de dictionnaires de pannes à donc été programmé en SKILL et fonctionne sous l'environnement CADENCE. Un désavantage de ce choix est que, pour le moment, l'outil ne peut fonctionner ailleurs que dans l'environnement CADENCE.

4.5.2 Choix d'implantation

La version présente du programme SKILL a demandé plusieurs choix de données et fonctions qui concernent le modèle de défaut, l'analyse d'interaction défaut-polygones, l'insertion des défauts, etc. La solution ici proposée a pour seul but la validation de la méthode de génération et ne prétend pas optimiser les performances du générateur de dictionnaires. De nombreuses améliorations peuvent être amenées à ce travail pour réduire la complexité des calculs et augmenter l'efficacité de l'outil.

4.5.3 Modélisation des défauts

4.5.3.1 Forme des défauts

Le travail présenté ici utilise un carré comme modélisation de défaut. L'avantage du carré est que l'analyse géométrique est énormément simplifiée. Les fonctions qui déterminent les interactions entre le défaut et les polygones du dessin des masques sont donc beaucoup plus rapides.

Etant donné que les connaissances actuelles des défauts ne permettent pas de juger d'un meilleur choix de modélisation, ce choix à été fait pour des raisons de performances. Pour cette raison le choix du carré est nettement avantageux puisque toutes les fonctions d'analyses d'interactions défaut-polygones s'en voient grandement simplifiées. Néanmoins, le choix de modèle pourrait être modifié à condition de maintenir les fonctions d'analyses d'interactions (touche, coupe, couvre) valables pour la nouvelle forme choisie.

4.5.3.2 Taille des défauts

La taille du défaut est un paramètre important, qui influence énormément le dictionnaire de pannes final. Un défaut très petit a peu de chance d'être la cause de pannes. Par contre un gros défaut peut causer de nombreuses pannes. Comment décider de la taille du défaut à insérer? La version actuelle du programme permet deux possibilités: l'usager donne une taille arbitraire du défaut, ou l'usager donne les paramètres de la distribution de probabilité des défauts.

La distribution de probabilité de défaut utilisée est celle de la figure 1.13. Cette distribution est le fruit d'observations de nombreux défauts en CMOS. Elle est décrite plus en détails dans [1] et [2]. Deux paramètres définissent cette distribution: χ_0 la taille du défaut

pour laquelle la densité de probabilité est maximale, et D est la valeur moyenne de la distribution tel que $P(x > \chi_0) = P(x < \chi_0) = \frac{D}{2}$.

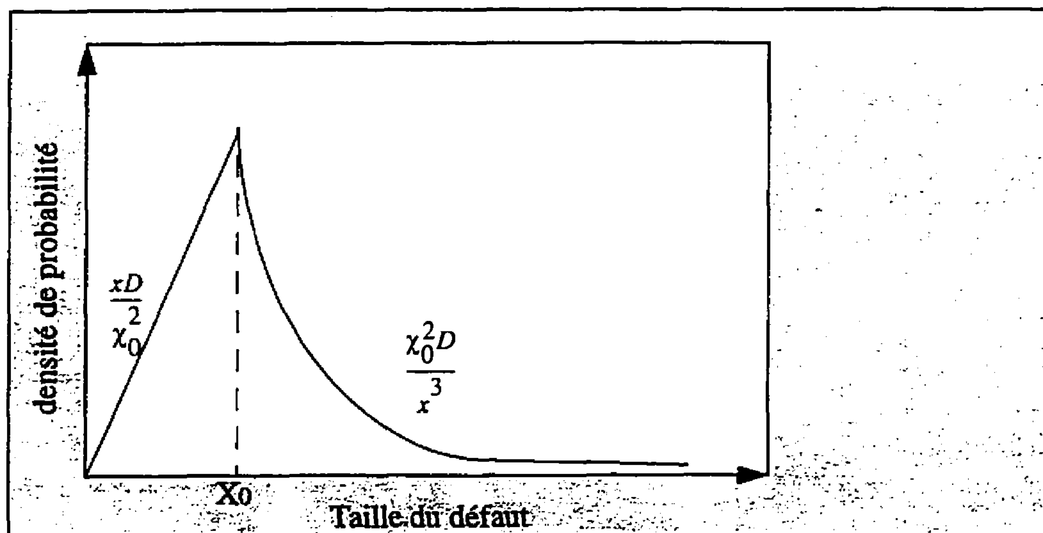


FIGURE 1.13 Distribution de probabilité des défauts

En utilisant cette distribution, la probabilité d'avoir un défaut de taille T est:

$$P(x > T) = \int_{\infty}^{T} D(x)\,dx \tag{1}$$

ou D(x) est la fonction tracée a la figure 1.13.

4.5.3.3 Insertion des défauts

Le déplacement du défaut se fait de façon itérative. Au début du programme, la fenêtre limite du dessin des masques est déterminée. Le défaut se place alors dans le coin inférieur gauche de cette fenêtre puis se déplace vers la droite. Lorsqu'il atteint la limite gauche de la fenêtre, il recommence une ligne plus haut à l'extrême droite, et ainsi de suite. Lorsque le défaut atteint le coin supérieur droit du dessin des masques, tous les

défauts ont été insérés. La figure 1.14 illustre le déplacement du défaut sur le dessin des masques.

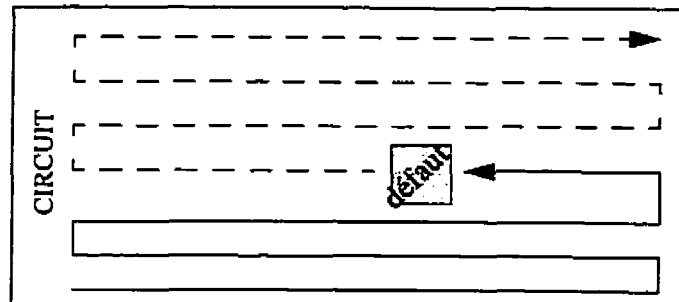


FIGURE 1.14 Déplacement du défaut sur la surface du circuit.

Cette méthode d'insertion de défauts n'est certes pas la meilleure, mais elle garantit une bonne couverture de la surface du circuit.

4.5.4 Analyse des pannes engendrées

Pour chaque défaut inséré, les pannes engendrées doivent êtres déterminées. C'est la majeure partie du programme de génération de dictionnaire qui se déroule alors.

L'analyse des pannes engendrées se fait conformément aux observations données dans la section 4.4.2. Il s'agit en fait d'automatiser le parcours du tableau 2 pour générer les pannes engendrées par un défaut.

L'analyses des pannes se fait en deux étapes principales:

1. détermination de tous les polygones touchés, coupés et couverts.
2. détermination des opens et des shorts grâce au tableau 2.

4.5.4.1 Exemple

Prenons par exemple le cas de la figure 1.11. Lors de la première étape, on obtient les listes suivantes:

-polygones touchés: A,B,C.

-polygones coupés: B.

-polygones couverts: aucun.

A partir de ces listes et en parcourant le tableau 2 ligne par ligne, il est maintenant facile de déduire les pannes suivantes:

1. excès de métal entraîne un short entre polygone A et B. (tableau 2,ligne 1)
2. manque de métal entraîne un open de B. (tableau 2,ligne 2)
3. excès de contact entraîne un short entre A, B, C. (tableau 2,ligne 3)
4. manque de contact sans conséquence. (tableau 2,ligne 4)
5. excès de poly sans conséquence. (tableau 2,ligne 5)
6. manque de poly sans conséquence. (tableau 2,ligne 6)
7. excès de n+ sans conséquence. (tableau 2,ligne 7)
8. manque de n+ sans conséquence. (tableau 2,ligne 8)
9. manque de p+ sans conséquence. (tableau 2,ligne 9)

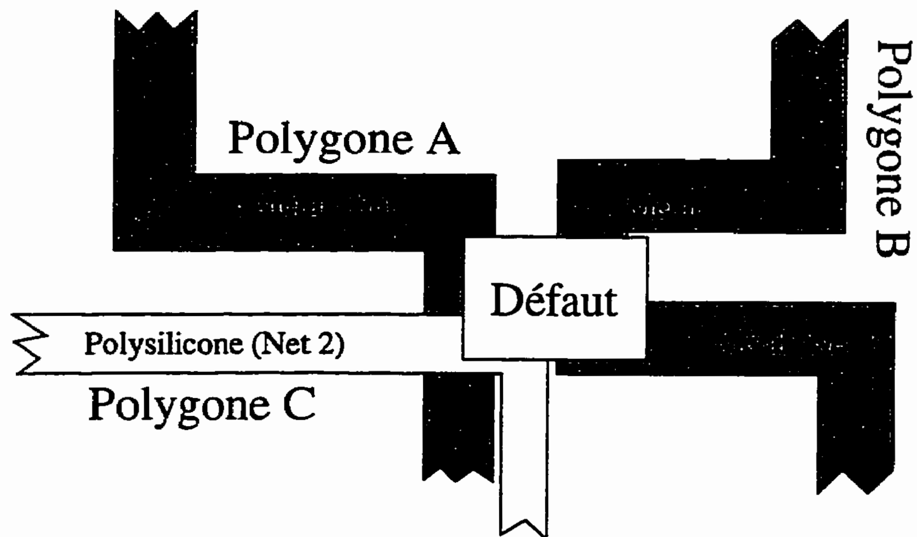


FIGURE 1.15 Exemple d'interaction défaut-polygones

Dans la base de données de CADENCE, chaque polygone possède un champ qui contient son numéro de noeud. Il est donc facile de déduire les noeuds court-circuités et ouverts une fois l'analyse précédente terminée. Le défaut de la figure 1.11 entraîne finalement les pannes suivantes: short(A,B),open (B), short(A,B,C).

Pour chaque défaut inséré dans le circuit cette procédure d'analyse de pannes doit avoir lieu. Lorsque tous les défauts ont été insérés, le dictionnaire est complété. Le temps de génération d'un dictionnaire est donc intimement lié au nombre de défauts insérés.

4.5.5 Analyse de complexité

La complexité, point de vue temps d'exécution, de la génération d'un dictionnaire de pannes à partir du dessin des masques selon la méthode présentée plus haut peut être réalisée.

L'algorithme est itératif et l'analyse de pannes se répète pour chaque défaut inséré. Le nombre de calculs effectués est donc lié au nombre de défauts (n).

Pour chaque défaut, les polygones touchés sont déterminés. Le nombre de calculs exécutés lors de cette opération est lié au nombre de sommets de tous les polygones formant le dessin des masques. La complexité de cette étape est donc liée au nombre de sommets (m).

L'ensemble de la génération du dictionnaire présente donc, dans le pire cas, une complexité de type $O(n*m)$.

Pour une taille de défaut donnée, le nombre de défauts insérés croit linéairement avec la surface du circuit. D'autre part si l'on suppose que le nombre de sommets croit également linéairement avec la surface du circuit, alors la complexité est fonction du carré de la surface du circuit. Ceci implique donc que lorsque la surface du circuit augmente, le temps de génération du dictionnaire augmente de façon quadratique en fonction de cette surface.

Etant donnée cette croissance quadratique, il faudrait, idéalement, trouver un algorithme de génération de dictionnaire de complexité linéaire. Pour cela, deux possibilités s'offrent à nous: 1) au niveau de l'insertion des défauts et 2) au niveau de l'analyse des interactions défaut-polygones.

1) Au niveau de l'insertion des défauts, il est possible de choisir un algorithme d'insertion différent qui choisit un certains nombres de défauts critiques à insérer. Le nombre n de défauts insérés ne serait donc plus directement relié à la surface du circuit. Il serait plutôt fonction de la répartition de densité de polygones sur le dessin des masques.

2) Au niveau de l'analyse des interactions défaut-polygones, il existe des méthodes qui permettent de venir à bout de cette tâche en minimisant la complexité. L'utilisation de représentation matricielle (ex: le "quadtree" est une liste, chainée selon deux dimensions, des polygones formant le circuit) permet de limiter la recherche des polygones touchés aux voisins immédiats du défaut inséré. Pour chaque défaut, une certain nombre constant de polygones sont inspectés pour déterminer s'ils sont recouverts ou non. On ramène alors la complexité pour tout le circuit à O(n) (n*constante). Il faut cependant ajouter la complexité de génération du "quadtree" qui est de O(m*log(m)). On obtient donc au total une complexité de O(n)+O(m*log(m)).

Le travail présent laisse donc une grande place à l'amélioration, en particulier au niveau de l'implantation informatique et des analyses d'interaction défaut-polygones.

4.6 Résultats et mode d'emploi

Un exemple très simple de génération de dictionnaires de pannes peut être exécuté sur un inverseur. La figure 1.16 montre son dessin des masques.

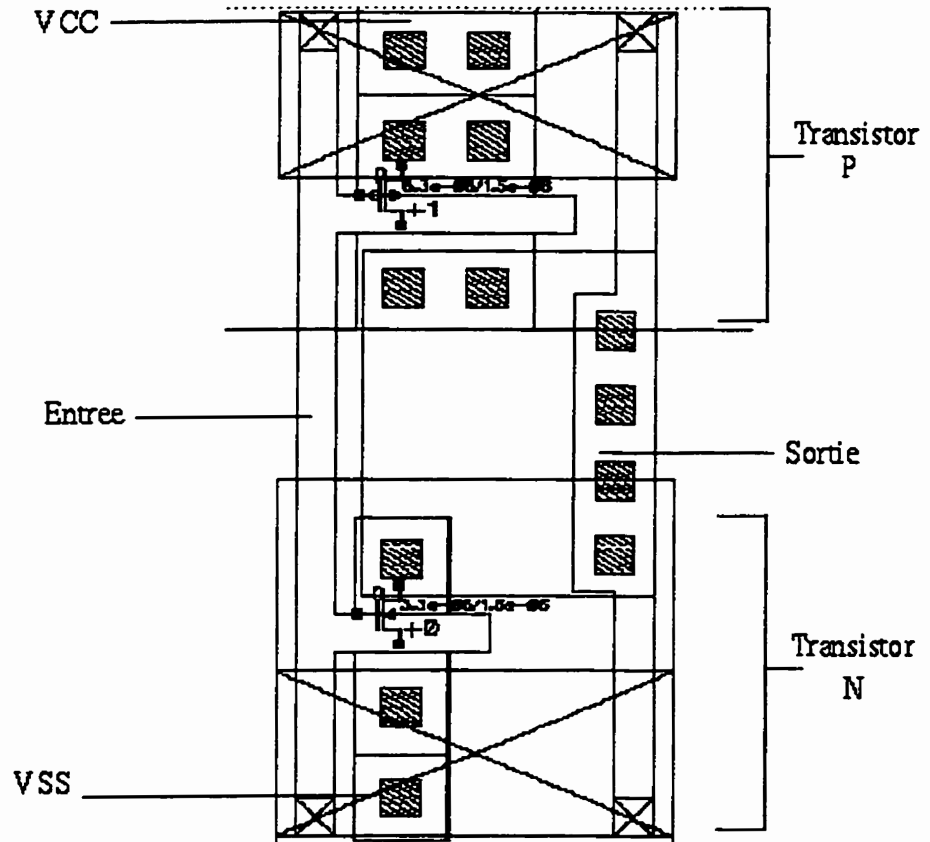


FIGURE 1.16 Dessin des masques d'un inverseur CMOS

La figure 1.17 donne la topologie ainsi que la liste des noeuds de cet inverseur tel que généré par CADENCE.

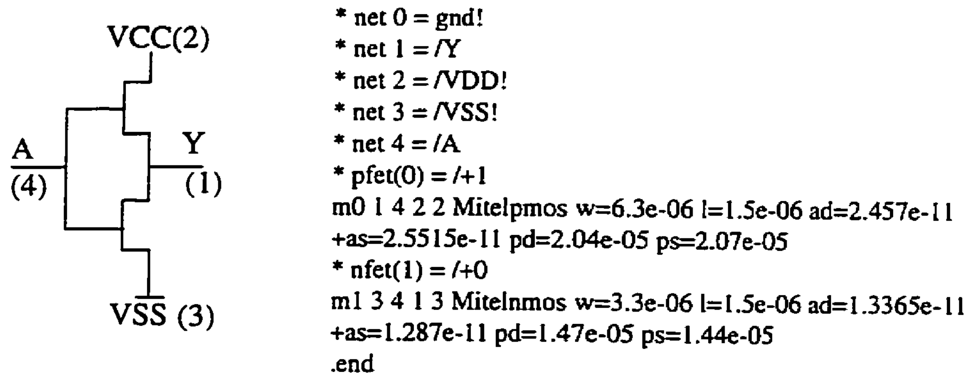


FIGURE 1.17 Inverseur CMOS

Il est facile de générer le dictionnaire exhaustif des pannes:

1 open A

2 open Y

3 open VCC

4 open VSS

5 short A VCC

6 short A Y

7 short A VSS

8 short VCC Y

9 short VCC VSS

10 short Y VSS

11 short A VCC Y

12 short A Y VSS

13 short A VCC VSS

14 short VCC Y VSS

15 short A VCC Y VSS

Il existe donc quinze différentes pannes possibles pour cet inverseur. La plupart sont très improbables puisqu'elles ne peuvent être provoquées que par d'énormes défauts. C'est le cas de la quinzième par exemple. Pour que cette panne ait lieu, et en se fiant au dessin des masques de la figure 1.16, il faudrait un défaut qui recouvre presque entière-

ment la surface de l'inverseur. On comprend alors l'importance de générer un dictionnaire sélectif donnant une liste de pannes plus réaliste.

La figure 1.18 montre un exemple de taille de défaut réaliste entraînant une panne probable. La panne entraînée par la présence d'un défaut d'*Excès de Contact* à cet endroit entraîne un court-circuit entre l'entrée et la sortie de l'inverseur. (c.f. ligne 4 tableau 2)

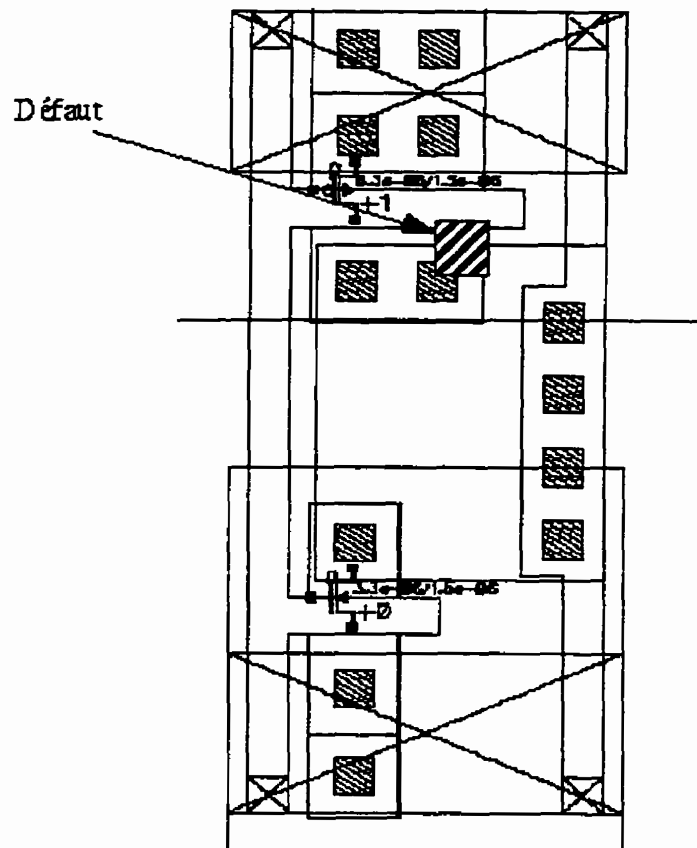


FIGURE 1.18 Exemple de défaut réaliste.

Si maintenant on déplace le défaut sur toute la surface du circuit, on obtient le dictionnaire de fautes réalistes suivant:

1. open net("Y") poly defect: ((2210.0 -1895.0) (2390.0 -1715.0))

2. open net("Y") poly defect: ((2210.0 -1745.0) (2390.0 -1565.0))
3. open net("Y") poly defect: ((2210.0 -1595.0) (2390.0 -1415.0))
4. open net("Y") poly defect: ((2210.0 -1445.0) (2390.0 -1265.0))
5. open net("A") poly defect: ((1160.0 -1295.0) (1340.0 -1115.0))
6. open net("A") poly defect: ((1310.0 -1295.0) (1490.0 -1115.0))
7. open net("A") poly defect: ((1460.0 -1295.0) (1640.0 -1115.0))
8. open net("A") poly defect: ((1610.0 -1295.0) (1790.0 -1115.0))
9. open net("Y") poly defect: ((2210.0 -1295.0) (2390.0 -1115.0))
10. open net("Y") poly defect: ((2210.0 -1145.0) (2390.0 -965.0))
11. open net("Y") poly defect: ((2210.0 55.0) (2390.0 235.0))
12. short net("Y" "A") poly/poly-contact defect: ((2060.0 205.0) (2240.0 385.0))
13. open net("Y") poly defect: ((2210.0 205.0) (2390.0 385.0))
14. open net("A") poly defect: ((1160.0 355.0) (1340.0 535.0))
15. open net("A") poly defect: ((1310.0 355.0) (1490.0 535.0))
16. open net("A") poly defect: ((1460.0 355.0) (1640.0 535.0))
17. open net("A") poly defect: ((1610.0 355.0) (1790.0 535.0))
18. open net("A") poly defect: ((1760.0 355.0) (1940.0 535.0))
19. open net("A") poly defect: ((1910.0 355.0) (2090.0 535.0))
20. short net("Y" "A") poly/poly-contact defect: ((2060.0 355.0) (2240.0 535.0))
21. open net("Y") poly defect: ((2210.0 355.0) (2390.0 535.0))
22. short net("Y" "A") poly/poly-contact defect: ((2060.0 505.0) (2240.0 685.0))
23. open net("Y") poly defect: ((2210.0 505.0) (2390.0 685.0))
24. open net("Y") poly defect: ((2210.0 655.0) (2390.0 835.0))
25. open net("Y") poly defect: ((2210.0 805.0) (2390.0 985.0))
26. open net("Y") poly defect: ((2210.0 955.0) (2390.0 1135.0))

Parmi ces 26 pannes, de nombreuses redondances apparaissent. Ces redondances sont inutiles en ce qui concerne le type de la panne. Par contre elles apportent une certaine information quant à la probabilité d'apparition d'une panne. C'est pourquoi lors de l'élimination de la redondance un certain poids, correspondant au nombre de fois que cette panne est apparue, est attribué à chacune des pannes.

Le dictionnaire obtenu après élimination de la redondance est le suivant:

1 open net("Y") Weight: 13.000000

2 open net("A") Weight: 10.000000
3 short net("Y" "A") Weight: 3.000000

Le dictionnaire final comporte donc trois pannes réalistes différentes. La panne la plus souvent rencontrée est l'entrée ouverte. Ceci s'explique par le fait que c'est le polygone le plus long et peu large. Un plus grand nombre de défauts (missing poly) peuvent donc le toucher. La panne la moins probable est un short entre l'entrée et la sortie. En effet peu de défauts peuvent joindre ces deux polygones qui ne se rapprochent qu'en peu d'endroits. Notons enfin que les défauts injectés ont été de 2 types différents (matériel en excès ou en manque) sur 9 couches (c.f. tableau 2) et ont emprunté 112 positions différentes. Un total de 2016 défauts ont donc été injectés dans le circuit de l'inverseur lors de l'exécution du programme de génération de dictionnaires de fautes réalistes.

Pour des circuits de taille plus importante, le processus de génération est exactement le même. Le circuit de la figure 1.19 est un amplificateur opérationnel visualisé sous CADENCE.

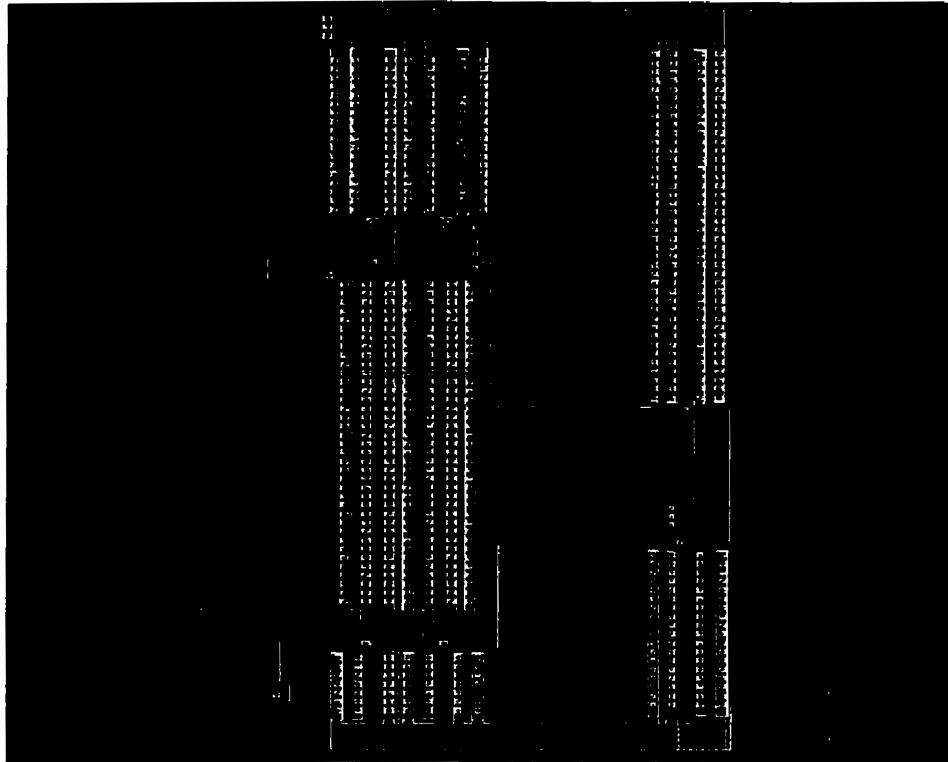


FIGURE 1.19 Dessin des masques d'un amplificateur

Si nous désirons obtenir un dictionnaire de fautes pour ce circuit en insérant des défauts de taille 1000 par 1000 (unité CADENCE mesurable avec l'éditeur de dessin des masques), il faut alors se placer dans la fenêtre principale de CADENCE et charger le programme de génération de dictionnaires de fautes. Ceci est illustré à la figure 1.20. Si tout

se passe bien, la commande est exécutée et la lettre "t" (true) apparaît dans la boite de dialogue pour indiquer que la commande est réussie.

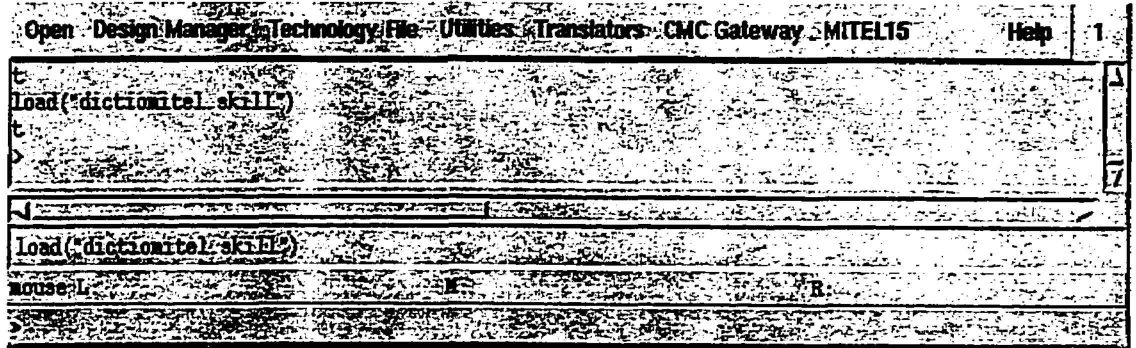


FIGURE 1.20 Chargement du générateur travaillant sur le dessin des masques

On peut alors générer le dictionnaire par la commande *defectList("mylib" "amp0" "0.0" 800.0 800.0 1000).*

On demande ainsi de générer un dictionnaire de pannes sur le circuit "amp0" de la librairie "mylib" (version "0.0") en insérant un défaut de taille 1000 par 1000 et en le déplaçant par incrément de 800 en abscisse et 800 en ordonnée.

L'exécution de cette commande entraîne l'insertion de 17818 défauts (2 types sur 9 couches à 989 emplacements différents). Le dictionnaire de fautes contient alors 3249 fautes qui se trouvent dans le fichier *error.list*. L'élimination de la redondance entraîne un dictionnaire de taille beaucoup plus raisonnable puisqu'il ne reste alors que 17 fautes différentes.

Ce dictionnaire est donné ci dessous:

1 short net("7" "8") Weight: 12.000000
2 open net("7") Weight: 303.000000
3 short net("7" "4") Weight: 1.000000
4 short net("6" "7") Weight: 13.000000

5 open net("8") Weight: 117.000000
6 open net("4") Weight: 498.000000
7 open net("6") Weight: 47.000000
8 short net("4" "8") Weight: 3.000000
9 short net("3" "4") Weight: 1.000000
10 open net("3") Weight: 91.000000
11 open net("1") Weight: 13.000000
12 open net("5") Weight: 95.000000
13 open net("2") Weight: 12.000000
14 short net("6" "8") Weight: 6.000000
15 short net("5" "4") Weight: 2.000000
16 short net("6" "5") Weight: 2.000000
17 short net("8" "5") Weight: 2.000000

Le poids donné permet un classement des pannes les plus souvent rencontrées. La panne numéro six (open net("4")) par exemple, a été la conséquence de l'insertion de 498 défauts différents. Ceci se comprend aisément quand on sait que le noeud 4 est une longue résistance étroite. De nombreux défauts sont donc susceptibles de couper cette résistance.

Les poids peuvent également être utilisés pour améliorer le design en le rendant plus résistant aux défauts. On peut par exemple décider de l'espacement de certains polygones en fonction des poids des court-circuits qui pourraient les relier.

Dans l'exemple ci-dessus le dictionnaire à été généré en fixant la taille du défaut de façon arbitraire. Une seconde option de génération existe. (c.f. "Forme des défauts" 4.5.3.1 p.44). Elle consiste à fournir les paramètres de la distribution de la taille des défauts pour que le générateur s'occupe du choix des tailles. Dans ce cas, le poids de chaque défaut est sa probabilité en fonctions des paramètres de la distribution fournie. Lors de l'élimination de la redondance, les probabilités des pannes redondantes sont ajoutées entre elles. La probabilité finale des pannes n'est donc plus une valeur comprise entre 0 et 1, mais un poids donné à chaque pannes.

Pour l'amplificateur de la figure 1.19, un dictionnaire a été généré en utilisant la distribution de la figure 1.13 avec comme paramètre, les valeurs suivantes: 1000.0 comme taille ou la densité de probabilité est la plus élevée, 0.5 comme densité de probabilité moyenne. Après avoir chargé le programme "mitellPrinc.skill", le dictionnaire se génère par la commande suivante: defectList("mylib" "amp0" "extracted" "0.0" 800.0 800.0 1000.0 0.5 2). On demande ainsi de générer le dictionnaire pour la cellule amp0 de la librairie mylib (version 0.0) en déplaçant des défauts par incrément de 800 en abscisse et de 800 en ordonnée. Pour chaque emplacement, des tailles de défauts sont générées en utilisant une distribution de valeur moyenne 0.5 et de densité maximale égale à 1000. On obtient alors un dictionnaire similaire, après insertion de 17903 défauts dans le layout.

4.7 Discussion et Conclusion

Une méthode de génération de dictionnaires de pannes à partir du dessin des masques à été validée: il est possible de générer un dictionnaire de pannes en se basant sur le dessin des masques des circuits. On obtient alors, au prix d'un effort d'analyse supérieur, un dictionnaire plus réaliste qu'en se basant sur la liste des noeuds.

Une façon de réaliser un tel générateur de dictionnaire consiste à le faire en SKILL dans l'environnement CADENCE. On profite ainsi des fonctions d'accès aux données des dessins des masques. Par contre, le générateur ne peut alors fonctionner que sous l'environnement CADENCE. D'autre part un programme en C (par exemple) serait beaucoup plus rapide. De nombreuses fonctions pourraient être retravaillées pour minimiser la complexité de ce logiciel. Entre autres celles d'insertion et d'analyse des interactions entre polygones.

Le modèle carré de défaut qui à été utilisé est discutable du point de vue du réalisme. Le débat étant ouvert, rien n'empêche de changer le modèle utilisé. Il faut toutefois

s'assurer que les fonctions d'interactions défaut-polygones s'appliquent au nouveau modèle choisi. Il faut également considérer le compromis qu'il peut exister entre les performances de l'outil et le modèle de défaut choisi.

L'analyse des pannes se base sur une table de défauts spécifiques à la technologie. Cette table est en fait le fichier de technologie et détermine les pannes entraînées par les différents défauts. Ici le travail à été réalisé pour la technologie CMOS de Mitell. Pour traiter des circuits de différentes technologies, il faut à chaque fois générer la table d'analyse des pannes associées à ces technologies. Une génération automatique de cette table pourrait venir compléter ce travail de façon intéressante.

Il est important de noter que cet outil n'est pas seulement utile lors de la génération de tests et plus précisément des vecteurs de tests, mais peut très bien servir lors de la phase de design pour minimiser le risque de panne à la fabrication. On peut notamment situer les zones, ou les éléments à risque pour ensuite les modifier et rendre le circuit plus résistant aux défauts.

CONCLUSION

Dans ce document, deux méthodes de génération de dictionnaires de pannes ont été décrites avec leurs avantages et leurs inconvénients. Un premier générateur de dictionnaire de fautes à été réalisé en C. Il utilise la liste des noeuds des circuits pour donner un dictionnaire exhaustif des pannes qui peuvent affecter le circuit. Un grand nombre des pannes générées de cette façon sont très peu probables .

Un deuxième générateur à été réalisé en SKILL sous l'environnement CADENCE. Il utilise le dessin des masques des circuits pour générer un dictionnaire réaliste des pannes probables. Pour atteindre ce but, il doit insérer des défauts sur toute la surface du circuit et déduire les pannes entraînées pour chacun d'entre eux. Le dictionnaire ne comporte donc que des pannes pouvant découler de défauts insérés. Deux versions du programme ont été faites. L'une permet de donner une taille de défaut fixe. L'autre permet de définir la distribution de probabilité des défauts qui devra être utilisée par le générateur. Dans les deux cas, les implantations logicielles ne font que valider la méthode et ne pourraient être commercialisées sans minimiser la complexité des algorithmes.

Les deux générateurs (exhaustif et réaliste) aboutissent à des dictionnaires très différents après des analyses différentes. Le premier donne un dictionnaire complet en peu de temps. Le second donne un dictionnaire très réaliste. Ou encore: le premier génère un dictionnaire comportant beaucoup de pannes sans intérêt (car peu probables) et le second demande un temps d'analyse très élevé (insertion et analyse de nombreux défauts).

Le choix du générateur doit se faire surtout en fonction des données disponibles (liste des noeuds et/ou dessin des masques), et du niveau auquel on veut travailler (schématique

et/ou physique). Dans les deux cas, il est possible non seulement de générer des tests avec les dictionnaires obtenus, mais d'améliorer les désigns des circuits traités en les rendant plus résistants aux défauts.

RÉFÉRENCES

[1] C.H.Stapper, “Modelling of Integrated Circuit Defect Sensitivities”, IBM J. Res. Develop. , Vol. 27, N0. 6, nov. 83, 549-557.

[2] C.H.Stapper, “LSI Yield Modelling and Process Monitoring”, IBM J.Res.Develop., may 1976, 228-233.

[3] B.Courtois, “Failure Mechanisms, Fault Hypotheses And Analytical Testing of LSI-NMOS (HMOS) Circuits”,

[4] Michel Rivier, “Random Yield Simulation Applied to Physical Circuit Design”, YIELD MODELLING and DEFECT TOLERANCE in VLSI, edited by Will Moore, wojciech Maly and Andrzej Strojwas, p.111-119.

[5] F.J.Ferguson and J.P.Shen. “A CMOS Fault Extractor for Inductive Fault Analysis.”, IEEE transaction on Computer-Aided Design of Intergrated Circuits and Systems 7, 11 (novembre 1988), 1181-1194.

[6] H.Walker and S.W.director, “VLASIC: A catastrophic Fault Yield Simulator for Integrated Circuits”. IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems CAD-5, 4 (Octobre 1986), 541-556.

[7] Khaled Saab, David Marche, Bozena Kaminska, Naim B.Hamida, Guy Quesnel, “LIMSOFT: Automated Tool for Test Vector Génération”, International Mixed Signal Testing Workshop, june 1995, Grenoble.

[8] J. Chojean and J.Izydorcky, "The Time domain Sensitivity Computation Using SPICE2. The Linear Network Case", Proceedings Intern. AMSE conference "Signal & System", Cetinje (Yugoslavia), Sep. 3-5, 1990, Vol 3, pp. 113-123.

[9] Stephan W. Director and Ronald A. Rohrer, "The Generalized Adjoint Network and Network Sensitivities", IEEE Trans. on Circuit and Theory, Vol. CT-16, no 3, August 1969, pp. 318-323.

[10] D. Walker, "VLASIC System User Manual Release 1.3", CAmegie-Mellon Uni. Pittsburgh USA., June 1990.

[11] J.S. Augusto, C.F.B. Almeida, "Automatic Fault Dictionnary Construction and Test Node Selection for Analogue and Mixed Fault Diagnosis with DC measurments.", ?, ?.

[12] Pascal Cauneger, Claude Abraham, " Achieving Simulation-BAsed Test Program Verification and Fault Simulation Capabilities for Mixed-Signal Systems", iee trans. on CAD, sept 1995, p.469-477.

[13] Marek Syrzycki, "Modeling of Gate Oxide Short in MOS Transistors", ieee trans. on CAD, vol. 8, no. 3, mars 1989.

[14] Ravi, "Imperfection and impurities in Semi-conductor Silicon",Wiley-Interscience.

[15] Linda Milor, V. Visvanathan, "Detection of Catastrophic Faults in analog Integrated Circuits",iee trans. on CAD, fevrier 89.

[16] Christopher Michael, Mohammed Ismail, “Statistical Modeling of Device Mismatch for Analog MOS Integrated Circuits”, ieee Journal of solid-state circuits, vol. 27, no. 2, fevrier 1992.

[17] Slamani M., Kaminska B.,”Soft Large Deviation and Hard Fault Multifrequency Analysis in Analog Circuits, IEEE Design and Test of computers, to appear in 1995.

[18] Ben Hamida N. and Kaminska B., “Multiple Fault Analog Circuit Testing by Sensitivity Analysis”, J. of Electronic Testing, JETTTA, Kluwer Academic Publ., no 4, Nov. 1993, pp. 331-343.

[19] Slamani M., Kaminska B., “Analog Circuit Fault Diagnosis Based on Sensitivity Computation and Functional Testing”, IEEE Design and Test of Computers, March 1992, pp. 30-39.

[20] Slamani M., Kaminska B., “An Integrated Approach for Analog Circuit Testing with a Minimum Number of Detected Parameters”, IEEE International Test Conference, Washington D.C., Oct. 1994, pp. 631-640

[21] Slamani M., Kaminska B., “Multifrequency testability Analysis for Analog Circuit”, 12th IEEE VLSI Test Symposium, Cherry Hill, April 25-28, 1994, pp. 54-59.

[22] Ben Hamida N. and Kaminska B., “Analog Circuit Testing Based on Sensitivity Computation”, IEEE International Test Conference, Baltimore, Oct. 1993, pp. 652-661.

ANNEXES

ANNEXE A

code du générateur

de

dictionnaire exhaustif

(basé sur le Netlist)

```
/****************************************
** Programme de génération        **
**     de dictionnaire de faute    **
**     a partir du netlist       **
**   et du fichier des .print pour  **
**     le calcul des gradient      **
** David Marche juillet 95       **
****************************************/
/* chaque element est la source d'un certain nombre de faute:
reference: J.S.Augusto, C. F. B. Almeida,
"Automatic Fault Dictionary Construction and Test Node Selection
 for Analogue and Mixed Fault Diagnosis with DC Measurements"

voici les fautes repertorie et donnees par ce programme:

elements passif...

R short,open
L short,open
Cshort, open

transistors...

Q (bjt)BCshort,CEshort,BEshort,Bopen,Copen,Eopen
M (mos) DGshort,DSshort,GSshort,Gopen,Dopen,Sopen
J (J&M) DGshort,DSshort,GSshort,Gopen,Dopen,Sopen

diodes...

D (dio) short, open

  Une liste de fautes est ainsi generee.
  Pour obtenir la liste finale, on elimine la redondance dans les fautes

*/




#include <stdio.h>
#include <string.h>
#include <stdlib.h>
/*#include <alloc.h>*/


#define FALSE 0
#define TRUE 1
#define NUL          ((char)0)
#define LONGUEURMOT     10
#define LONGUEURMAX     100
#define RESISTANCE      1   /* r... */
#define CAPACITE        2   /* c... */
```

```
#define INDUCTANCE     3   /* l... */
#define SOURCE_CONTROL  5   /* e...,f...,g...,h... */
#define SOURCE_TENSION  6   /* v... */
#define SOURCE_COURANT  7   /* i... */
#define APPEL_MACRO    10  /* x... */
#define DIODE          14  /* d... */
#define MOSFET         15  /* m... */
#define BJT            16  /* q... */
#define JANDMESFET     17  /* j... */
#define MACRO  20  /* .macro ... */
#define START          30  /* start of macro or circuit */
#define EOM            40  /* end of macro   (.eom)*/
#define EOC            50  /* end of circuit (.end)*/

struct macro_mere{
char *nom;
struct macro_mere *suivant;
struct macro_mere *precedent;

};

struct macro_mere *derniere_macro;
struct macro_mere *premiere_macro;
struct element{
short type;
     char nom[LONGUEURMOT];
     char valeur[LONGUEURMOT];
     char noeud1[LONGUEURMOT];
     char noeud2[LONGUEURMOT];
     char noeud3[LONGUEURMOT];
     char noeud4[LONGUEURMOT];
     int ligne;
struct element *vertical;
     struct element *horizontal;
};

void repositionner(FILE *fichier,int position);
void warning(char *lineptr);
void error(char *lineptr);
char *strrev(char *chaine);
struct element *construire_arbre(FILE *fichiercir);
struct element *remonter_arbre(struct element *feuille_courante);
structelement *dernier_element_macro(struct element *feuille_courante);
char *determine_macro_appele(char *linebfr);
void ajouter_macro(char* nom);
void retirer_macro();
void parcourir(struct element *feuille,FILE *fichiertemp, int niveaulimit);
struct element *goto_macro(struct element *feuille_courante);
void affiche_macro(char *macro);
void creer_la_liste(); /* pour les macro*/

char *prendre_mot(int numero, char *linebfr);
char *proprietaire();

void   eliminer_redondance(FILE *fichiertemp,FILE *fichierfault, FILE *fichierprint, FILE *fichiermat);
```

```
short same_fault(char *linebfr,char *fault);
void add_fault(char *linebfr,FILE *fichierfault);
void print_fault(char *linebfr, FILE *fichierfault, FILE *fichierprint, FILE *fichiermat);
short new_fault(char *fault,int faultcount,FILE *fichiertemp);

void transistor_mosfet(struct element *feuille,FILE *fichiertemp);
void transistor_jandmesfet(struct element *feuille,FILE *fichiertemp);
void transistor_bjt(struct element *feuille,FILE *fichiertemp);
void diode(struct element *feuille, FILE *fichiertemp);

void element_macro(struct element *feuille,FILE *fichiertemp);

void r(struct element *feuille,FILE *fichiertemp);
void c(struct element *feuille,FILE *fichiertemp);
void l(struct element *feuille,FILE *fichiertemp);
void g(struct element *feuille,FILE *fichiertemp);
void h(struct element *feuille,FILE *fichiertemp);
void f(struct element *feuille,FILE *fichiertemp);
void e(struct element *feuille,FILE *fichiertemp);
void i(struct element *feuille,FILE *fichiertemp);
void v(struct element *feuille,FILE *fichiertemp);

int niveaucnt;

void main(int argc, char *argv[])
{
 char nomfichiercircuit[15];
 int  niveaulimit;
 char  *nomfichierfault;
 char  *nomfichiertemp;
 char * nomfichierprint;
 char * nomfichiermat;
 FILE  *fichiercir;
 FILE  *fichierfault;
 FILE  *fichiertemp;
 FILE  *fichierprint;
 FILE  *fichiermat;
 struct element *debut_arbre;

 /* verifier si le fichier du netlist a ete fourni */
 if (argc<1) error("!!! specify circuit and limit level of element inspection!!!");

 /* definition du fichier de reference*/
 printf("fichier du circuit:");
 strcpy(nomfichiercircuit,argv[1]);
 printf("\n %s comme circuit de référence \n",nomfichiercircuit);

 /* definition du niveau limit d'inspection de macro*/
 printf("niveau limit d'inspection des macros:");
 sscanf(argv[2],"%i",&niveaulimit);
 printf("\n %i comme niveau limite d'inspection \n",niveaulimit);


 /* determination du nom du fichier de faute temporaire */
 /*  (avant elimination de la redondance)            */
```

```
nomfichiertemp=(char *)malloc(15);
strcpy(nomfichiertemp,nomfichiercircuit);
nomfichiertemp =strtok(nomfichiertemp,".");
nomfichiertemp=strcat(nomfichiertemp,".tmp");
printf("nom du fichier contenant la liste intermediaire: %s\n", nomfichiertemp);



/* determination du nom du fichier de faute */
nomfichierfault=(char *)malloc(15);
strcpy(nomfichierfault,nomfichiercircuit);
nomfichierfault =strtok(nomfichierfault,".");
nomfichierfault=strcat(nomfichierfault,".flt");
printf("nom du fichier contenant la liste finale de fautes: %s\n", nomfichierfault);

/* determination du nom du fichier de .print */
nomfichierprint=(char *)malloc(15);
strcpy(nomfichierprint,nomfichiercircuit);
strtok(nomfichierprint,".");
strcat(nomfichierprint,".pri");
printf("nom du fichier contenant la liste finale de .print: %s\n", nomfichiermat);

/* determination du nom du fichier des noms des matrices: .mat */
nomfichiermat=(char *)malloc(15);
strcpy(nomfichiermat,nomfichiercircuit);
strtok(nomfichiermat,".");
strcat(nomfichiermat,".ele");
printf("nom du fichier contenant la liste finale de matrices: %s\n", nomfichiermat);



/*ouvrir les fichiers*/
fichiercir  = fopen(nomfichiercircuit,"rt");
fichiertemp = fopen(nomfichiertemp   ,"wt");

/*construire l'arbre du circuit */
printf("constructionde l'arbre \n");
debut_arbre=construire_arbre(fichiercir);

/*ouvrir la liste des macro*/
printf("initialisation de la liste des macros \n");
creer_la_liste();

/*parcourir l'arbre en generant les fautes possibles rencontrees*/
niveaucnt=0;
parcourir(debut_arbre,fichiertemp,niveaulimit);
fprintf(fichiertemp,"this is the end my friend");



/*fermer les fichier*/
fclose(fichiercir);
fclose(fichiertemp);
```

```
 /*elimination des redondances du fichier intermediaire */
 fichiertemp = fopen(nomfichiertemp ,"rt");
 fichierfault= fopen(nomfichierfault,"wt");
 fichierprint= fopen(nomfichierprint  ,"wt");
 fichiermat  = fopen(nomfichiermat    ,"wt");

 eliminer_redondance(fichiertemp,fichierfault,fichierprint,fichiermat);
 fclose(fichiertemp);
 fclose(fichierfault);
 fclose(fichierprint);
 fclose(fichiermat);
 }


struct element *construire_arbre(FILE *fichiercir)
{
 char *linebfr;
 char *mot1;
 int line;

 struct element *debut_arbre;
 struct element *feuille;
 struct element *feuille_courante;


 linebfr=(char *)malloc(LONGUEURMAX);


 /* creer le premier element de l'arbre */
 feuille=(struct element*) malloc(sizeof(struct element));
 feuille->type=MACRO;
 strcpy(feuille->nom,"main");
 feuille->ligne=0;
 feuille->horizontal=NULL;
 feuille->vertical=NULL;
 debut_arbre=feuille;
 feuille_courante=feuille;

 feuille=(struct element*) malloc(sizeof(struct element));
 feuille_courante->vertical=feuille;
 feuille->type=START;
 strcpy(feuille->nom,"TREESTART");
 feuille->horizontal=feuille;
 feuille->vertical=feuille_courante;
 feuille_courante=feuille;

 niveaucnt=1;
 line=0;
 do {
  line++;
  fgets(linebfr,LONGUEURMAX,fichiercir); /* lecture d'une ligne */
  printf("%s",linebfr);
  /*ajouter l'element dans l'arbre*/
  mot1=prendre_mot(1,linebfr);
  if ((*mot1=='r')||
```

```
(*mot1=='R')||
(*mot1=='c')||
(*mot1=='C')||
(*mot1=='l')||
(*mot1=='L')||
(*mot1=='d')||
    (*mot1=='D')||
(*mot1=='m')||
    (*mot1=='M')||
(*mot1=='j')||
    (*mot1=='J')||
(*mot1=='q')||
    (*mot1=='Q')||
    (*mot1=='x')||
(*mot1=='X')||
(*mot1=='g')||
(*mot1=='G')||
(*mot1=='f')||
(*mot1=='F')||
(*mot1=='e')||
(*mot1=='E')||
(*mot1=='h')||
(*mot1=='H')||
(*mot1=='v')||
(*mot1=='V')||
(*mot1=='i')||
(*mot1=='I'))
   {
    feuille=(struct element*) malloc(sizeof(struct element));
    feuille->horizontal=feuille_courante->horizontal;
    feuille_courante->horizontal=feuille;
    feuille->vertical=NULL;


    switch (*mot1){
     case 'r':
 case 'R': feuille->type=RESISTANCE;
  strcpy(feuille->noeud1,prendre_mot(2,linebfr));
  strcpy(feuille->noeud2,prendre_mot(3,linebfr));
  strcpy(feuille->valeur,prendre_mot(4,linebfr));break;
 case 'c':
 case 'C': feuille->type=CAPACITE;
          strcpy(feuille->noeud1,prendre_mot(2,linebfr));
  strcpy(feuille->noeud2,prendre_mot(3,linebfr));
  strcpy(feuille->valeur,prendre_mot(4,linebfr));break;
 case 'l':
 case 'L': feuille->type=INDUCTANCE;
   strcpy(feuille->noeud1,prendre_mot(2,linebfr));
  strcpy(feuille->noeud2,prendre_mot(3,linebfr));
  strcpy(feuille->valeur,prendre_mot(4,linebfr));break;
     case 'm':
 case 'M': feuille->type=MOSFET;
          strcpy(feuille->noeud1,prendre_mot(2,linebfr));
  strcpy(feuille->noeud2,prendre_mot(3,linebfr));
  strcpy(feuille->noeud3,prendre_mot(4,linebfr));
```

```
strcpy(feuille->noeud4,prendre_mot(5,linebfr));
strcpy(feuille->valeur,prendre_mot(6,linebfr));break;

case 'j':
case 'J': feuille->type=JANDMESFET;
        strcpy(feuille->noeud1,prendre_mot(2,linebfr));
 strcpy(feuille->noeud2,prendre_mot(3,linebfr));
 strcpy(feuille->noeud3,prendre_mot(4,linebfr));
 strcpy(feuille->noeud4,prendre_mot(5,linebfr));
 strcpy(feuille->valeur,prendre_mot(6,linebfr));break;

    case 'd':
case 'D': feuille->type=DIODE;
        strcpy(feuille->noeud1,prendre_mot(2,linebfr));
 strcpy(feuille->noeud2,prendre_mot(3,linebfr));
 strcpy(feuille->valeur,prendre_mot(4,linebfr));break;
case 'q':
case 'Q': feuille->type=BJT;
        strcpy(feuille->noeud1,prendre_mot(2,linebfr));
 strcpy(feuille->noeud2,prendre_mot(3,linebfr));
 strcpy(feuille->noeud3,prendre_mot(4,linebfr));
 strcpy(feuille->noeud4,prendre_mot(5,linebfr));
 strcpy(feuille->valeur,prendre_mot(6,linebfr));break;
case 'g':
case 'G': feuille->type=SOURCE_CONTROL;
 strcpy(feuille->noeud1,prendre_mot(2,linebfr));
 strcpy(feuille->noeud2,prendre_mot(3,linebfr));
        strcpy(feuille->noeud3,prendre_mot(5,linebfr));
 strcpy(feuille->noeud4,prendre_mot(6,linebfr));
 strcpy(feuille->valeur,prendre_mot(7,linebfr));break;
case 'f':
case 'F': feuille->type=SOURCE_CONTROL;
 strcpy(feuille->noeud1,prendre_mot(2,linebfr));
 strcpy(feuille->noeud2,prendre_mot(3,linebfr));
        strcpy(feuille->noeud3,prendre_mot(5,linebfr));
 strcpy(feuille->noeud4,prendre_mot(6,linebfr));
 strcpy(feuille->valeur,prendre_mot(6,linebfr));break;
case 'e':
case 'E': feuille->type=SOURCE_CONTROL;
 strcpy(feuille->noeud1,prendre_mot(2,linebfr));
 strcpy(feuille->noeud2,prendre_mot(3,linebfr));
        strcpy(feuille->noeud3,prendre_mot(5,linebfr));
 strcpy(feuille->noeud4,prendre_mot(6,linebfr));
 strcpy(feuille->valeur,prendre_mot(7,linebfr));break;
case 'h':
case 'H': feuille->type=SOURCE_CONTROL;
 strcpy(feuille->noeud1,prendre_mot(2,linebfr));
 strcpy(feuille->noeud2,prendre_mot(3,linebfr));
        strcpy(feuille->noeud3,prendre_mot(5,linebfr));
 strcpy(feuille->noeud4,prendre_mot(6,linebfr));
 strcpy(feuille->valeur,prendre_mot(6,linebfr));break;
case 'v':
case 'V': feuille->type=SOURCE_TENSION;
        strcpy(feuille->noeud1,prendre_mot(2,linebfr));
 strcpy(feuille->noeud2,prendre_mot(3,linebfr));
```

```
 strcpy(feuille->valeur,prendre_mot(5,linebfr));break;
 case 'i':
     case 'I': feuille->type=SOURCE_COURANT;
             strcpy(feuille->noeud1,prendre_mot(2,linebfr));
  strcpy(feuille->noeud2,prendre_mot(3,linebfr));
  strcpy(feuille->valeur,prendre_mot(5,linebfr));break;
      case 'x':
     case 'X': feuille->type=APPEL_MACRO;
  strcpy(feuille->valeur,determine_macro_appele(linebfr));
             strcpy(feuille->noeud1,strtok(linebfr," "));      /*nom de l'element*/
  strcpy(feuille->noeud2,strtok(NULL,feuille->valeur)); /*noeud de l'element*/
      break;

      }
     strcpy(feuille->nom,mot1);
     feuille->ligne=line;
     feuille_courante=feuille;
     }

   if ((strcmp(mot1,".macro")==0)||(strcmp(mot1,".MACRO")==0)
||(strcmp(mot1,".subckt")==0)||(strcmp(mot1,".SUBCKT")==0))
     {
     feuille=(struct element*) malloc(sizeof(struct element));
     feuille->horizontal=feuille_courante->horizontal;
     feuille_courante->horizontal=feuille;
     feuille->type=MACRO;
     feuille->ligne=line;
     strcpy(feuille->nom,prendre_mot(2,linebfr));
     feuille_courante=feuille;

     feuille=(struct element*) malloc(sizeof(struct element));
     feuille->type=START;
     strcpy(feuille->nom,"start");
     feuille->ligne=line;
     feuille->vertical=feuille_courante;
     feuille_courante->vertical=feuille;
     feuille->horizontal=feuille;
     feuille_courante=feuille;
     }

  if ((strcmp(mot1,".eom")==0)||(strcmp(mot1,".EOM")==0)||
(strcmp(mot1,".ends")==0)||(strcmp(mot1,".ENDS")==0))
     {
     feuille=(struct element*) malloc(sizeof(struct element));
     feuille->horizontal=feuille_courante->horizontal;
     feuille_courante->horizontal=feuille;
     feuille->type=EOM;
     strcpy(feuille->nom,"eom");
     feuille->ligne=line;
     feuille_courante=feuille;
     feuille_courante=remonter_arbre(feuille_courante);
     feuille_courante=dernier_element_macro(feuille_courante);
     }
 } while ((strcmp(mot1,".end\n")!=0)
&&(strcmp(mot1,".end")!=0));
```

```
  feuille=(struct element*) malloc(sizeof(struct element));
  feuille->horizontal=feuille_courante->horizontal;
  feuille_courante->horizontal=feuille;
  feuille->type=EOC;
  strcpy(feuille->nom,"eoc");
  feuille->ligne=line;
  return(debut_arbre);
 }

/* revenir a la macro de niveau superieur */
struct element *remonter_arbre(struct element *feuille_courante)
{

feuille_courante=(feuille_courante->horizontal)->vertical;
     return(feuille_courante);
}

/* aller aux dernier element de la macro */
struct element *dernier_element_macro(struct element *feuille_courante)
{
   if ((feuille_courante->horizontal)->type!=START)
   {
    do{
      feuille_courante=feuille_courante->horizontal;
     }while ((feuille_courante->horizontal)->type!=START);
   }
   return(feuille_courante);
}



void parcourir(struct element *feuille,FILE *fichiertemp,int niveaulimit)
{

 printf("parcours de la macro %s,\n",feuille->nom);
 feuille=feuille->vertical;
 feuille=feuille->horizontal;

 do {
  printf(" =%s=",feuille->nom);
  switch (feuille->type){


case APPEL_MACRO   : niveaucnt++;
   ajouter_macro(feuille->nom);
        if (niveaucnt<=niveaulimit)
parcourir(goto_macro(feuille),fichiertemp,niveaulimit);
   retirer_macro();
   if (niveaucnt>niveaulimit)
 element_macro(feuille,fichiertemp);
                    printf("\nretour a la macro superieure\n");

case MACRO         : feuille=feuille->horizontal;break;
```

```
    case DIODE       : diode(feuille, fichiertemp);
   feuille=feuille->horizontal;
                  break;

case MOSFET       : transistor_mosfet(feuille,fichiertemp);
   feuille=feuille->horizontal;
    break;

case JANDMESFET   : transistor_jandmesfet(feuille,fichiertemp);
   feuille=feuille->horizontal;
                  break;


case BJT          : transistor_bjt(feuille,fichiertemp);
   feuille=feuille->horizontal;
        break;

case RESISTANCE   : r(feuille,fichiertemp);feuille=feuille->horizontal;break;

case CAPACITE     : c(feuille,fichiertemp);feuille=feuille->horizontal;break;

case INDUCTANCE   : l(feuille,fichiertemp);feuille=feuille->horizontal;break;

case SOURCE_CONTROL: switch (*(feuille->nom)){
          case 'g':
 case 'G':g(feuille,fichiertemp);break;
case 'h':
case 'H':h(feuille,fichiertemp);break;
case 'f':
 case 'F':f(feuille,fichiertemp);break;
case 'e':
case 'E':e(feuille,fichiertemp);break;
          }
                 feuille=feuille->horizontal;break;

case SOURCE_TENSION: v(feuille,fichiertemp);
   feuille=feuille->horizontal;break;

    case SOURCE_COURANT: i(feuille,fichiertemp);
   feuille=feuille->horizontal;break;
}
  printf("%s\n",feuille->nom);

 } while ( (feuille->type!=START)
       &&(feuille->type!=EOM)
       &&(feuille->type!=EOC) );
 niveaucnt--;
 printf("\n ");
}

void r(struct element *feuille,FILE *fichiertemp)
{
 fprintf(fichiertemp,"open %s%s %s\nopen %s%s %s\nshort %s%s %s%s %s\n",
proprietaire(),feuille->noeud1,feuille->nom,
proprietaire(),feuille->noeud2,feuille->nom,
```

```
proprietaire(),feuille->noeud1,proprietaire(),feuille->noeud2,feuille->nom);
}

void element_macro(struct element *feuille,FILE *fichiertemp)
{
 char *noeuds;
 short nbr_noeuds;
 char *mot;
 short i;

 noeuds=(char *)malloc(LONGUEURMAX);
 mot=(char *)malloc(LONGUEURMOT);

 strcpy(noeuds,feuille->noeud2);
 strcat(noeuds," nomorenoeud");
 nbr_noeuds=0;
 mot=strtok(noeuds," ");
 do{
  fprintf(fichiertemp,"open  %s%s %s\n",proprietaire(),mot,feuille->nom);
  nbr_noeuds++;
  mot=strtok(NULL," ");
 }while(strcmp(mot,"nomorenoeud")!=0);

 free(noeuds);
 free(mot);
}

void c(struct element *feuille,FILE *fichiertemp)
{
 fprintf(fichiertemp,"open  %s%s %s\nopen  %s%s %s\nshort %s%s %s%s %s\n",
proprietaire(),feuille->noeud1,feuille->nom,
proprietaire(),feuille->noeud2,feuille->nom,
proprietaire(),feuille->noeud1,proprietaire(),feuille->noeud2,feuille->nom);
}

void l(struct element *feuille,FILE *fichiertemp)
{
 fprintf(fichiertemp,"open  %s%s %s\nopen  %s%s %s\nshort %s%s %s%s %s\n",
proprietaire(),feuille->noeud1,feuille->nom,
proprietaire(),feuille->noeud2,feuille->nom,
proprietaire(),feuille->noeud1,proprietaire(),feuille->noeud2,feuille->nom);
}

void g(struct element *feuille,FILE *fichiertemp)
{
 fprintf(fichiertemp,"open  %s%s %s\nopen %s%s %s\nshort %s%s %s%s %s\n",
proprietaire(),feuille->noeud1,feuille->nom,
proprietaire(),feuille->noeud2,feuille->nom,
proprietaire(),feuille->noeud1,proprietaire(),feuille->noeud2,feuille->nom);
}

void h(struct element *feuille,FILE *fichiertemp)
{
 fprintf(fichiertemp,"open  %s%s %s\nopen  %s%s %s\nshort %s%s %s%s %s\n",
proprietaire(),feuille->noeud1,feuille->nom,
```

```
proprietaire(),feuille->noeud2,feuille->nom,
proprietaire(),feuille->noeud1,proprietaire(),feuille->noeud2,feuille->nom);
}

void f(struct element *feuille,FILE *fichiertemp)
{
 fprintf(fichiertemp,"open %s%s %s\nopen %s%s %s\nshort %s%s %s%s %s\n",
proprietaire(),feuille->noeud1,feuille->nom,
proprietaire(),feuille->noeud2,feuille->nom,
proprietaire(),feuille->noeud1,proprietaire(),feuille->noeud2,feuille->nom);
}

void e(struct element *feuille,FILE *fichiertemp)
{
 fprintf(fichiertemp,"open %s%s %s\nopen %s%s %s\nshort %s%s %s%s %s\n",
proprietaire(),feuille->noeud1,feuille->nom,
proprietaire(),feuille->noeud2,feuille->nom,
proprietaire(),feuille->noeud1,proprietaire(),feuille->noeud2,feuille->nom);
}

void i(struct element *feuille,FILE *fichiertemp)
{
 fprintf(fichiertemp,"open %s%s %s\nopen %s%s %s\nshort %s%s %s%s %s\n",
proprietaire(),feuille->noeud1,feuille->nom,
proprietaire(),feuille->noeud2,feuille->nom,
proprietaire(),feuille->noeud1,proprietaire(),feuille->noeud2,feuille->nom);
}

void v(struct element *feuille,FILE *fichiertemp)
{
 fprintf(fichiertemp,"open %s%s %s\nopen %s%s %s\nshort %s%s %s%s %s\n",
proprietaire(),feuille->noeud1,feuille->nom,
proprietaire(),feuille->noeud2,feuille->nom,
proprietaire(),feuille->noeud1,proprietaire(),feuille->noeud2,feuille->nom);
}

void transistor_mosfet(struct element *feuille, FILE *fichiertemp)
{
 fprintf(fichiertemp,"open %s%s G.%s\nopen %s%s D.%s\nopen %s%s S.%s\n",
proprietaire(),feuille->noeud1,feuille->nom,
proprietaire(),feuille->noeud2,feuille->nom,
proprietaire(),feuille->noeud3,feuille->nom);
 fprintf(fichiertemp,"short %s%s %s%s DG.%s\nshort %s%s %s%s DS.%s\nshort %s%s %s%s GS.%s\n",
proprietaire(),feuille->noeud1,proprietaire(),feuille->noeud2,feuille->nom,
proprietaire(),feuille->noeud1,proprietaire(),feuille->noeud3,feuille->nom,
proprietaire(),feuille->noeud2,proprietaire(),feuille->noeud3,feuille->nom);
}

void transistor_jandmesfet(struct element *feuille, FILE *fichiertemp)
{
 fprintf(fichiertemp,"open %s%s G.%s\nopen %s%s D.%s\nopen %s%s S.%s\n",
proprietaire(),feuille->noeud1,feuille->nom,
proprietaire(),feuille->noeud2,feuille->nom,
proprietaire(),feuille->noeud3,feuille->nom);
 fprintf(fichiertemp,"short %s%s %s%s DG.%s\nshort %s%s %s%s DS.%s\nshort %s%s %s%s GS.%s\n",
```

```
proprietaire(),feuille->noeud1,proprietaire(),feuille->noeud2,feuille->nom,
proprietaire(),feuille->noeud1,proprietaire(),feuille->noeud3,feuille->nom,
proprietaire(),feuille->noeud2,proprietaire(),feuille->noeud3,feuille->nom);
}

void transistor_bjt(struct element *feuille, FILE *fichiertemp)
{
  fprintf(fichiertemp,"open  %s%s B.%s\nopen  %s%s C.%s\nopen  %s%s E.%s\n",
proprietaire(),feuille->noeud1,feuille->nom,
proprietaire(),feuille->noeud2,feuille->nom,
proprietaire(),feuille->noeud3,feuille->nom);
  fprintf(fichiertemp,"short %s%s %s%s BC.%s\nshort %s%s %s%s BE.%s\nshort %s%s %s%s CE.%s\n",
proprietaire(),feuille->noeud1,proprietaire(),feuille->noeud2,feuille->nom,
proprietaire(),feuille->noeud1,proprietaire(),feuille->noeud3,feuille->nom,
proprietaire(),feuille->noeud2,proprietaire(),feuille->noeud3,feuille->nom);
}

void diode(struct element *feuille, FILE *fichiertemp)
{
  fprintf(fichiertemp,"open  %s%s %s\nopen  %s%s %s\nshort %s%s %s%s %s\n",
proprietaire(),feuille->noeud1,feuille->nom,
proprietaire(),feuille->noeud2,feuille->nom,
proprietaire(),feuille->noeud1,proprietaire(),feuille->noeud2,feuille->nom);
}

char *proprietaire()
{
  struct macro_mere *macro;
  char* proprio;

  proprio=(char *)malloc(LONGUEURMAX);
  macro=(struct macro_mere*) malloc(sizeof(struct macro_mere));

  strcpy(proprio,"");
  macro=premiere_macro;
  if (premiere_macro!=derniere_macro)
    {
    do{
      macro=macro->suivant;
      strcat(proprio,macro->nom);
      strcat(proprio,".");
    }while (macro!=derniere_macro);
    }
  return(proprio);
}


char *prendre_mot(int numero, char *linebfr)
{
  char *mot;
  int i;
  char *ptr;

  mot=(char *)malloc(LONGUEURMAX);
  /*printf("ligne %s\n",linebfr);*/
```

```
  strcpy(mot,linebfr);
  ptr=strtok(mot," ");
  i=1;
  if ((ptr!=NULL)&&(i<numero)){
   do{
    i++;
    ptr=strtok(NULL," ");
    }while ((i<numero)&&(ptr!=NULL));
  }
  if (strstr(ptr,"\n")) strtok(ptr,"\n");

  if ((ptr!=NULL)&&(strcmp(ptr,"\n")!=0)) strcpy(mot,strtok(ptr,"\n"));
  else strcpy(mot,"");
  return(mot);
 }
 struct element *goto_macro(struct element *feuille)
 {

  char *nom;

  nom=(char *)malloc(LONGUEURMOT);
  strcpy(nom,feuille->valeur);


  printf("\nrecherche de la macro %s\n",feuille->valeur);

  feuille=dernier_element_macro(feuille);
  feuille=feuille->horizontal->horizontal;
   do{
     if (feuille->type==START)
    {
    feuille=feuille->vertical;
    feuille=dernier_element_macro(feuille);
    feuille=feuille->horizontal->horizontal;
    }
   else feuille=feuille->horizontal;
  }while (   (strcmp(feuille->nom,nom)!=0)
       && (feuille->type!=EOC));
 if (feuille->type==EOC) error("macro not found");
 else return(feuille);




}



void creer_la_liste() /* pour les macro*/
{
struct macro_mere *fin_liste;
fin_liste=(struct macro_mere*) malloc(sizeof(struct macro_mere));
fin_liste->nom = "main";
premiere_macro=fin_liste;
derniere_macro=fin_liste;
```

```
}

void ajouter_macro(char* name)
{
struct macro_mere *macro;
macro=(struct macro_mere*) malloc(sizeof(struct macro_mere));
macro->nom=name;
derniere_macro->suivant=macro;
macro->precedent=derniere_macro;
derniere_macro=macro;
}

void  retirer_macro()
{
struct macro_mere *macro;
macro=(struct macro_mere*) malloc(sizeof(struct macro_mere));
macro=derniere_macro;
derniere_macro=macro->precedent;
derniere_macro->suivant=NULL;
free(macro);
}


void affiche_macro(char *macro)
{
 printf("macro: %s:\n",macro);
}


char *strrev(char *chaine)
{
 int i;
 int longueur;
 char *revstring;
 char *caractere;

 revstring=(char *)malloc(LONGUEURMAX);

 strcpy(revstring,"");
 longueur=strlen(chaine);
  for (i=longueur-1;i>=0;i--){
   caractere=chaine+i;
   revstring=strncat(revstring,caractere,1);
 }
 return(revstring);
}

void warning(char *lineptr)
{
printf("** Warning **");
printf(lineptr);
}

void error(char *lineptr)
{
```

```
printf("** Error **");
printf(lineptr);
exit(0);
}

void repositionner(FILE *fichier,int position)
{
 char *linebfr;
 int  actualposition;

 linebfr=(char *)malloc(LONGUEURMAX);

 rewind(fichier);
 do
 {
  fgets(linebfr,LONGUEURMAX,fichier);
  actualposition=ftell(fichier);
 }while(actualposition<position);
}

char *determine_macro_appele(char *linebfr)
{
 char *lineptr;
 char *macro_demande;
 char *parametre;

 lineptr=(char *)malloc(LONGUEURMAX);

 strcpy(lineptr,linebfr);
 if (strchr(lineptr,'=')!=NULL)
  {
  strtok(lineptr,"=");
  strcpy(lineptr,strrev(lineptr));
  strtok(lineptr," ");
  macro_demande=strrev(strtok(NULL," "));
  }
 else
  {
  strtok(lineptr,"\n");
  strcpy(lineptr,strrev(lineptr));
  macro_demande=strrev(strtok(lineptr," "));
  }
 return(macro_demande);
}

void   eliminer_redondance(FILE *fichiertemp,FILE *fichierfault, FILE *fichierprint, FILE *fichiermat)
{
 int faultcount;
 int redondancecount;
 int i;

 char *linebfr;
 char *fault;

 printf("\n elimination de la redondance...\n");
```

```
 linebfr=(char *) malloc(LONGUEURMAX);
 fault =(char *) malloc(LONGUEURMAX);

 faultcount=0;
 redondancecount=0;
 fgets(linebfr,LONGUEURMAX,fichiertemp); /* lecture d'une faute */
 do{

  printf("%s",linebfr);
  faultcount++;
  if (new_fault(linebfr,faultcount,fichiertemp)){
   print_fault(linebfr,fichierfault,fichierprint,fichiermat);
   strcpy(fault,linebfr);
   do{
    fgets(linebfr,LONGUEURMAX,fichiertemp);
    if (same_fault(linebfr,fault)) {add_fault(linebfr,fichierfault);
redondancecount++;}
   }while (strcmp(linebfr,"this is the end my friend")!=0);
   fprintf(fichierfault,"\n");
   rewind(fichiertemp);
   for(i=1;i<=faultcount;i++) { fgets(linebfr,LONGUEURMAX,fichiertemp); }
  }
  fgets(linebfr,LONGUEURMAX,fichiertemp); /* lecture d'une faute */

 }while (strcmp(linebfr,"this is the end my friend")!=0);

 printf("done\n");
 printf("number of listed fault              :%d\n",faultcount);
 printf("number of redundant fault eliminated :%d\n",redondancecount);

}

short same_fault(char *linebfr,char *fault)
{
 if (strcmp(prendre_mot(1,fault),"short")==0){
  if (strcmp(prendre_mot(1,linebfr),"short")==0){
   if (strcmp(prendre_mot(2,linebfr),prendre_mot(2,fault))==0){
if (strcmp(prendre_mot(3,linebfr),prendre_mot(3,fault))==0) return (TRUE);
   }
  }
 }
 if (strcmp(prendre_mot(1,fault),"open")==0){
  if (strcmp(prendre_mot(1,linebfr),"open")==0){
   if (strcmp(prendre_mot(2,linebfr),prendre_mot(2,fault))==0) return (TRUE);
  }
 }
 return(FALSE);
}

void add_fault(char *linebfr,FILE *fichierfault)
{
 if (strcmp(prendre_mot(1,linebfr),"short")==0) fprintf(fichierfault," %s",prendre_mot(4,linebfr));
 if (strcmp(prendre_mot(1,linebfr),"open")==0)  fprintf(fichierfault," %s",prendre_mot(3,linebfr));
```

```
}

void print_fault(char *linebfr, FILE *fichierfault, FILE *fichierprint, FILE *fichiermat)
{
 if (strcmp(prendre_mot(1,linebfr),"short")==0){
   fprintf(fichierfault,"%s %s %s %s",prendre_mot(1,linebfr)
     ,prendre_mot(2,linebfr)
,prendre_mot(3,linebfr)
,prendre_mot(4,linebfr));
   fprintf(fichierprint,".print vr(xcircuit.%s,xcircuit.%s)  vi(xcircuit.%s,xcircuit.%s)\n"
,prendre_mot(2,linebfr),prendre_mot(3,linebfr)
,prendre_mot(2,linebfr),prendre_mot(3,linebfr));
   fprintf(fichiermat ,"S%sxx%s\n",prendre_mot(2,linebfr),prendre_mot(3,linebfr));
   }
 if (strcmp(prendre_mot(1,linebfr),"open")==0){
   fprintf(fichierfault,"%s %s %s",prendre_mot(1,linebfr)
     ,prendre_mot(2,linebfr)
,prendre_mot(3,linebfr));
   fprintf(fichierprint,".print ir(xcircuit.%s,xcircuit.%s)  ii(xcircuit.%s,xcircuit.%s)\n"
,prendre_mot(3,linebfr),prendre_mot(3,linebfr)
,prendre_mot(3,linebfr),prendre_mot(3,linebfr));
   fprintf(fichiermat ,"O%s\n",prendre_mot(3,linebfr));
   }
}


short new_fault(char *fault,int faultcount,FILE *fichiertemp)
{
 char *linebfr;
 int compteur;
 int i;

 compteur=0;
 linebfr=(char *)malloc(LONGUEURMAX);


 if (faultcount==1) return(TRUE);
 rewind(fichiertemp);
 do{
   compteur++;
   fgets(linebfr,LONGUEURMAX,fichiertemp);
 }while((compteur<faultcount)&&(!(same_fault(fault,linebfr))));
 rewind(fichiertemp);
 for(i=1;i<=faultcount;i++) { fgets(linebfr,LONGUEURMAX,fichiertemp); }
 if (compteur==faultcount) return(TRUE);
 return(FALSE);
}
```

ANNEXE B

code SKILL du générateur

de

dictionnaire réaliste

(basé sur le Layout)

note: fonctionne sous l'environnement

Cadence

Programme principal: dictiomitel.skill

```
;programme principal d'analyse du layout pour la generation des hard fault.
; DefectList est la procedure qui genere le dictionnaire de faute.
; les ergument de la procedure sont:
;-le nom de la librarie
;-le nom de la cellule
;-le nom de la vue ("extracted")
; -la version (ex: 0.0)
; -le pas de deplacement du defaut en abscisse
;-le pas de deplacement du defaut en ordonnee
;-le diametre du defaut
;
; le 4 premier argument definissent la cellule sur laquelle l'insertion de defaut est faite.
; les deux suivant definissent les deplacement du defaut donc la precision de la couverture.
; les deux suivant definissent la distribution de densite deprobabilite des defauts selon leurs tailles.
; le dernier definis le nombre de taille differentes de defaut insere a chaque position.
; ce dernier defini donc aussi la precision de la couverture de la simulation.
; La taille des defauts se repartie de facons homogene entre une taille valant le double
; de celle ou la densite de prob. est maximale et 0.

;chargement des procedures
load("addDefect.skill")
load("mitellDefectAnalisys.skill")
load("getDefectSizeList.skill")
load("computeDefectProb.skill")
load("mitellPinholeFaults.skill")


;procedure de generation du dictionnaire de faute.

procedure( defectList( library cell version
deltadefect_x deltadefect_y
DefectSize
"tttfff")
  if( progn(
;ouverture de la cellule
printf("opening cell...")
      view="extracted"                          ; analisys performed on extracted view
cellule=dbOpenCellView(library cell view version "a") ; opening cell
printf("successfull\n")

;ouverture du fichier de sortie: error.list
printf("opening error.list file...")
errorfile=outfile("error.list" "w")             ; declaring dictionnary Output File
printf("successfull\n")

;recherche de la fenetre qui contient la cellule.
printf("searching cell bounding box...")
cellbBox=cellule->bBox                          ; Get cell Bounding Box
printf("successfull\n")
```

```
printf("cell bounding box: %L\n" cellbBox)
cellBox.xinf=nth(0 nth(0 cellbBox))
cellBox.yinf=nth(1 nth(0 cellbBox))
cellBox.xsup=nth(0 nth(1 cellbBox))
cellBox.ysup=nth(1 nth(1 cellbBox))
plist('cellBox)


;generation des defauts et analyse des consequences sur le netlist
printf("starting defect generation...\n")

defect_number=0                ;0 defauts inseres pour le moment
deltadefect.x=deltadefect_x      ;deplacement du defaut en x
deltadefect.y=deltadefect_y      ;deplacement du defaut en y
defectxysize='(1 1)             ;taille du defaut par defaut
defectlayer="text"              ;couche du defaut (utile a la visualisation)
xdefect=cellBox.xinf             ;coordonnees initiales du defaut en x
ydefect=cellBox.yinf  ;coordonnees initiales du defaut en x


defectsize_list=append(list(DefectSize) list(DefectSize))  ;sizes of square defect.
printf("defect sizes: %L\n" defectsize_list )

;boucle de parcours de la boite (bBox) contenant la cellule.

while( (ydefect <= cellBox.ysup)
      progn(while( (xdefect<=cellBox.xsup)
        progn( printf("defect data: -pos: ( %f, %f)\n" xdefect ydefect )
             addDefect( cellule defectlayer `(,xdefect ,ydefect) defectsize_list)
        mitellDefectAnalisys(cellule defaut 1.0)
        dbDeleteObject(defaut)
        defect_number=defect_number+1
        xdefect=xdefect+deltadefect.x
                 )
                 )
   xdefect=cellBox.xinf
   ydefect=ydefect+deltadefect.y
           )
   )

fprintf(errorfile "this is the end my friend")      ;end of file marker for dictionnary
close( errorfile)    ;closing the dictionnary

;resultat des insertions et bilan de la simulation

printf("successfull\n");
hiGetAttention()                          ; beep

printf("\n*** %d defect inserted in cell %s ***\n" defect_number
  cellule->cellName)    ; summary of dictionnary generation.

;looking for pinholes faults

printf("\n\nlooking for pinholes short\n")
mitellPinholeFaults(library cell view version )
```

```
) ;endprogn     ; end of dictionnary generation and pinholes simulation
  printf("End of fault dictionnary generation")
  ) ;endif
) ;enprocedure                    ;that's all.
```

Programme d'insertion des défaut: addDefect.skill

```
; procedure qui ajoutes un defaut (rectangulaire)
; il faut specifier la cellule, la couche, la taille du defaut ,
; et la position (coin inferieur gauche du defaut)

procedure( addDefect(cellule couche pos taille "dtll")

 defaut=dbCreateRect( cellule~>cellView
     couche
     list(pos mapcar('plus pos taille))
)
```

Programme d'analyse de pannes : mitellDefectAnalysis.skill

```
; Analyse des consequence d'un defaut sur le netlist de la cellule
; valable pour la technologie mitel

load("getCutList.skill")    ; charger les procedure de determination des formes
load("getCoverList.skill")  ; coupees et recouvertes.
load("getIntersectionList.skill")
load("selectLayer.skill")
load("eliminateRedundancy.skill")


procedure( mitellDefectAnalisys( cellule defaut prob "ddf")
 ;liste des object touche par le defaut.
 overlap_list=dbGetTrueOverlaps( cellule~>cellView  defaut~>bBox) ;overlap
 cover_list=getCoverList(overlap_list defaut)  ;covers
 cut_list=getCutList(overlap_list defaut)  ;cuts

 ;------------METAL2 DEFECT--------------------

 ;extra metal2 defect
 short_dirty_list=selectLayer("metal2" overlap_list)
 short_clean_list=eliminateRedundancy(short_dirty_list~>net~>name)
 if( length(short_clean_list)>1
    fprintf( errorfile "short net%L weight: %f metal2 defect: %L\n" short_clean_list prob defaut~>bBox)
  )
```

```
;missing metal2 defect
open_dirty_list=selectLayer("contact" cover_list)
open_clean_list=eliminateRedundancy(open_dirty_list->net->name)
if( length(open_clean_list)>0
  fprintf( errorfile "open net%L weight: %f contact defect: %L\n" open_clean_list prob defaut->bBox)
 )

open_dirty_list=append( selectLayer("metal2" cut_list)
selectLayer("metal2" cover_list))
open_clean_list=eliminateRedundancy(open_dirty_list->net->name)
if( length(open_clean_list)>0
  fprintf( errorfile "open net%L metal2 defect: %L\n" open_clean_list prob defaut->bBox)
 )

;--------------METAL1 DEFECT ----------------

;extra metal1 defect
short_dirty_list=selectLayer("metal1" overlap_list)
short_clean_list=eliminateRedundancy(short_dirty_list->net->name)
if( length(short_clean_list)>1
  fprintf( errorfile "short net%L weight: %f metal1 defect: %L\n" short_clean_list prob defaut->bBox)
 )

;missing metal1 defect
open_dirty_list=selectLayer("contact" cover_list)
open_clean_list=eliminateRedundancy(open_dirty_list->net->name)
if( length(open_clean_list)>0
  fprintf( errorfile "open net%L weight: %f contact defect: %L\n" open_clean_list prob defaut->bBox)
 )

;missing metal1 defect
open_dirty_list=append( selectLayer("metal1" cut_list)
selectLayer("metal1" cover_list))
open_clean_list=eliminateRedundancy(open_dirty_list->net->name)
if( length(open_clean_list)>0
  fprintf( errorfile "open net%L weight: %f metal1 defect: %L\n" open_clean_list prob defaut->bBox)
 )

;---------------POLY1i DEFECT---------------

;extra poly1 defect
short_dirty_list=append( selectLayer("poly1i" overlap_list)
  selectLayer("contact" overlap_list))
short_clean_list=eliminateRedundancy(short_dirty_list->net->name)
if( length(short_clean_list)>1
  fprintf(errorfile "short net%L weight: %f poly/poly-contact defect: %L\n" short_clean_list prob
defaut->bBox)
 )
```

```
;extra poly1 defect
newtrans_dirty_list=append( selectLayer("psd" cut_list)
   selectLayer("nsd" cut_list) )
newtrans_clean_list=eliminateRedundancy(newtrans_dirty_list->net->name)
if( length(newtrans_clean_list)>0
   progn( fprintf(errorfile "open net%L weight: %f newtrans defect: %L\n" newtrans_clean_list   prob
defaut->bBox)
      connexion=setof(shape overlap_list (shape->layerName=="poly1i")
        ||(shape->layerName=="contact"))
          connexion=eliminateRedundancy(connexion->net->name)
          if( length(connexion)>0
fprintf(errorfile "PS: gate of newtrans connected to net %L\n" connexion)
             fprintf(errorfile "PS: gate of newtrans floating\n")
      )
        )
   )

;extra poly1 defect
misstrans_dirty_list=append( selectLayer("psd" cover_list)
   selectLayer("nsd" cover_list) )
misstrans_clean_list=eliminateRedundancy(misstrans_dirty_list->net->name)
if( length(misstrans_clean_list)>0
   if( length(misstrans_clean_list)>1
     fprintf(errorfile "short net%L weight: %f misstrans defect: %L\n" misstrans_clean_list   prob defaut->bBox)
     fprintf(errorfile "open net%L weight: %f misstrans defect: %L\nPS: but probable short with gate anyway\n"
misstrans_clean_list   prob defaut->bBox)
   )
 )


;extra poly1 defect
contact_list=eliminateShapeRedundancy(selectLayer("contact" cover_list))
open_list=()
if( dtpr( contact_list)                    ; si la liste des contact touche n'est pas vide
   progn( psd_contact=()
        nsd_contact=()
        foreach( contact contact_list
      progn(nsd_contact=selectLayer("nsd" dbGetTrueOverlaps(cellule contact->bBox))
     psd_contact=selectLayer("psd" dbGetTrueOverlaps(cellule contact->bBox))
     open_list=append(open_list nsd_contact)
     open_list=append(open_list psd_contact)
 )
            )
       )
 )
open_list=eliminateRedundancy(open_list->net->name)
if( dtpr(open_list)
   fprintf(errorfile "open net%L weight: %f poly defect: %L\n" open_list   prob defaut->bBox)
 )



;missing poly1 defect
open_dirty_list=append( selectLayer("poly1i" cut_list)
              selectLayer("poly1i" cover_list))
```

```
open_clean_list=eliminateRedundancy(open_dirty_list~>net~>name)
if( length(open_clean_list)>0
   fprintf(errorfile "open net%L weight: %f poly defect: %L\n" open_clean_list   prob defaut~>bBox)
  )


;missing poly1 defect
contact_list=eliminateShapeRedundancy(selectLayer("contact" cover_list))
open_list=()
if( dtpr( contact_list)                    ; si la liste des contact touche n'est pas vide
   progn( poly_contact=()
        foreach( contact contact_list
      progn(poly_contact=selectLayer("poly1i" dbGetTrueOverlaps(cellule contact~>bBox))
     open_list=append(open_list poly_contact)
 )
              )
        )
  )
open_list=eliminateRedundancy(open_list~>net~>name)
if( length(open_list)>0
   fprintf(errorfile "open net%L weight: %f poly defect: %L\n" open_list   prob defaut~>bBox)
  )
;----------------POLY2i DEFECT-----------------

;extra poly defect
short_dirty_list=append( selectLayer("poly2i" overlap_list)
  selectLayer("contact" overlap_list))
short_clean_list=eliminateRedundancy(short_dirty_list~>net~>name)
if( length(short_clean_list)>1
   fprintf(errorfile "short net%L weight: %f poly/poly-contact defect: %L\n" short_clean_list   prob
defaut~>bBox)
  )


;extra poly defect
newtrans_dirty_list=append( selectLayer("psd" cut_list)
   selectLayer("nsd" cut_list) )
newtrans_clean_list=eliminateRedundancy(newtrans_dirty_list~>net~>name)
if( length(newtrans_clean_list)>0
   progn( fprintf(errorfile "open net%L weight: %f newtrans defect: %L\n" newtrans_clean_list   prob
defaut~>bBox)
   connexion=setof(shape overlap_list (shape~>layerName=="poly2i")
     ||(shape~>layerName=="contact"))
        connexion=eliminateRedundancy(connexion~>net~>name)
        if( length(connexion)>0
fprintf(errorfile "PS: gate of newtrans connected to net %L\n" connexion)
           fprintf(errorfile "PS: gate of newtrans floating\n")
    )
       )
  )

;extra poly defect
misstrans_dirty_list=append( selectLayer("psd" cover_list)
   selectLayer("nsd" cover_list) )
misstrans_clean_list=eliminateRedundancy(misstrans_dirty_list~>net~>name)
```

```
if( length(misstrans_clean_list)>0
   if( length(misstrans_clean_list)>1
     fprintf(errorfile "short net%L weight: %f misstrans defect: %L\n" misstrans_clean_list  prob defaut->bBox)
     fprintf(errorfile "open net%L weight: %f misstrans defect: %L\nPS: but probable short with gate anyway\n"
misstrans_clean_list  prob defaut->bBox)
    )
  )


;extra poly defect
contact_list=eliminateShapeRedundancy(selectLayer("contact" cover_list))
open_list=()
if( dtpr( contact_list)                    ; si la liste des contact touche n'est pas vide
   progn( psd_contact=()
        nsd_contact=()
        foreach( contact contact_list
      progn(nsd_contact=selectLayer("nsd" dbGetTrueOverlaps(cellule contact->bBox))
     psd_contact=selectLayer("psd" dbGetTrueOverlaps(cellule contact->bBox))
     open_list=append(open_list nsd_contact)
    open_list=append(open_list psd_contact)
 )
           )
      )
 )
open_list=eliminateRedundancy(open_list->net->name)
if( dtpr(open_list)
   fprintf(errorfile "open net%L weight: %f poly defect: %L\n" open_list  prob defaut->bBox)
 )



;missing poly defect
open_dirty_list=append( selectLayer("poly2i" cut_list)
               selectLayer("poly2i" cover_list))
open_clean_list=eliminateRedundancy(open_dirty_list->net->name)
if( length(open_clean_list)>0
   fprintf(errorfile "open net%L weight: %f poly defect: %L\n" open_clean_list  prob defaut->bBox)
 )

;missing poly defect
contact_list=eliminateShapeRedundancy(selectLayer("contact" cover_list))
open_list=()
if( dtpr( contact_list)                    ; si la liste des contact touche n'est pas vide
   progn( poly_contact=()
        foreach( contact contact_list
      progn(poly_contact=selectLayer("poly2i" dbGetTrueOverlaps(cellule contact->bBox))
     open_list=append(open_list poly_contact)
 )
           )
      )
 )
open_list=eliminateRedundancy(open_list->net->name)
if( length(open_list)>0
   fprintf(errorfile "open net%L weight: %f poly defect: %L\n" open_list  prob defaut->bBox)
 )
```

```
;----------------VIA DEFECT------------------

;missing VIA defect
open_dirty_list=selectLayer("contact" cover_list)
open_clean_list=eliminateRedundancy(open_dirty_list->net->name)
if( length(open_clean_list)>0
   fprintf(errorfile "open net%L weight: %f via defect: %L\n" open_clean_list   prob defaut->bBox)
  )

;---------------CONTACT DEFECT---------------

;missing CONTACT defect
open_dirty_list=selectLayer("contact" cover_list)
open_clean_list=eliminateRedundancy(open_dirty_list->net->name)
if( length(open_clean_list)>0
   fprintf(errorfile "open net%L weight: %f contact defect: %L \n" open_clean_list   prob defaut->bBox)
  )

;--------------nsd DEFECT------------------(n diffusion?)

;missing nsd implant defect
open_dirty_list=append( selectLayer("nsd" cut_list)
selectLayer("nsd" cover_list))
open_clean_list=eliminateRedundancy(open_dirty_list->net->name)
if( length(open_clean_list)>0
   fprintf(errorfile "open net%L weight: %f nsd defect: %L\n" open_clean_list   prob defaut->bBox)
  )

;extra nsd implant defect
open_dirty_list=append( selectLayer("psd" cut_list)
selectLayer("psd" cover_list))
open_clean_list=eliminateRedundancy(open_dirty_list->net->name)
if( length(open_clean_list)>0
   fprintf(errorfile "open net%L weight: %f psd defect: %L\n" open_clean_list   prob defaut->bBox)
  )

;extra nsd defect
contact_list=eliminateShapeRedundancy(selectLayer("contact" cover_list))
open_list=()
if( dtpr( contact_list)                    ; si la liste des contact touche n'est pas vide
   progn( poly_contact=()
         foreach( contact contact_list
       progn(poly_contact=selectLayer("psd" dbGetTrueOverlaps(cellule contact->bBox))
     open_list=append(open_list poly_contact)
 )
            )
      )
 )
open_list=eliminateRedundancy(open_list->net->name)
if( length(open_list)>0
   fprintf(errorfile "open net%L weight: %f nsd defect: %L\n" open_list   prob defaut->bBox)
   ;printf("open list due to extra n+ implant cutting contact with psd: %L \n" open_list)
```

```
 )

;missing nsd defect
contact_list=eliminateShapeRedundancy(selectLayer("contact" cover_list))
open_list=()
if( dtpr( contact_list)                    ; si la liste des contact touche n'est pas vide
   progn( poly_contact=()
        foreach( contact contact_list
      progn(poly_contact=selectLayer("nsd" dbGetTrueOverlaps(cellule contact->bBox))
     open_list=append(open_list poly_contact)
 )
             )
       )
 )
open_list=eliminateRedundancy(open_list->net->name)
if( length(open_list)>0
   fprintf(errorfile "open net%L weight: %f nsd defect: %L\n" open_list   prob defaut->bBox)
   ;printf("open list due to missing p+ implant cutting contact with nsd: %L \n" open_list)
 )

;---------------psd DEFECT--------------------(p diffusion?)



;missing PDIFF implant defect
open_dirty_list=append( selectLayer("psd" cut_list)
selectLayer("psd" cover_list))
open_clean_list=eliminateRedundancy(open_dirty_list->net->name)
if( length(open_clean_list)>0
   fprintf(errorfile "open net%L weight: %f psd defect: %L\n" open_clean_list   prob defaut->bBox)
 )

;missing PDIFF defect
contact_list=eliminateShapeRedundancy(selectLayer("contact" cover_list))
open_list=()
if( dtpr( contact_list)                    ; si la liste des contact touche n'est pas vide
   progn( poly_contact=()
        foreach( contact contact_list
      progn(poly_contact=selectLayer("psd" dbGetTrueOverlaps(cellule contact->bBox))
     open_list=append(open_list poly_contact)
 )
             )
       )
 )
open_list=eliminateRedundancy(open_list->net->name)
if( length(open_list)>0
   fprintf(errorfile "open net%L weight: %f psd defect: %L\n" open_list   prob defaut->bBox)
   ;printf("open list due to missing p+ implant cutting contact with psd: %L \n" open_list)
 )



;---------------NO MORE DEFECT--------------------
```

```
  ;end of analisys beep
  ;hiGetAttention()
 )
```

Programme de generation de valeur de taille de défaut : getDefectSizeList.skill

```
; procedure de generation de la liste des tailles de defaut.


procedure( getDefectSizeList( X0 Dbar n "ffx")
 if( progn( wmax=X0*2
   wmin=wmax/n
   w=wmin
       deltaw=wmax/n;
       w_list=list(list(wmin wmin))
       defectsizeprob_list=list(computeDefectProb(w X0 Dbar))
   for( i 1 n-1
         w=w+deltaw
```

```
   w_list=append(w_list list(list(w w)))
  prob=computeDefectProb(w X0 Dbar)
            defectsizeprob_list=append(defectsizeprob_list list(prob))
           )

       )
    w_list
   )
 )
```

Programme d'analyse de "pinhole" : mitellPinholefaults.skill

```
;procedure qui donne toutes les fautes possibles dues a des pinholes.
;ces shorts peuvent avoir lieux entre metal(1/2)-poly, metal(1/2)-diffusion,
;metal1-metal2

procedure( eliminateFalseShort( net1 net2_list "tl")
 if( (!member(net1 net2_list))
    net2_list
```

```
progn(
clean_list=()
for(i 1 length(net2_list)
if( (nth(i-1 net2_list)!=net1)
clean_list=append(clean_list list(nth(i-1 net2_list)))
)
)
clean_list
)
)
)



procedure( getCroiseList(forme overlap_list "dl")
 if( progn( nbr_overlap=length(overlap_list)
        touch_list=()
        box=forme->bBox
        if( (forme->objType=="polygon")
           forme1point_list=forme->points
           forme1point_list=list(list( nth(0 nth(0 box)) nth(1 nth(0 box)) )
                     list( nth(0 nth(0 box)) nth(1 nth(1 box)) )
                     list( nth(0 nth(1 box)) nth(1 nth(1 box)) )
                     list( nth(0 nth(1 box)) nth(1 nth(0 box)) )
                     )
        )

        for( i 1 nbr_overlap
           box=nth(i-1 overlap_list)->bBox
           if( (nth(i-1 overlap_list)->objType=="polygon")
              forme2point_list=nth(i-1 overlap_list)->points
              forme2point_list=list(list( nth(0 nth(0 box)) nth(1 nth(0 box)) )
                        list( nth(0 nth(0 box)) nth(1 nth(1 box)) )
                        list( nth(0 nth(1 box)) nth(1 nth(1 box)) )
                        list( nth(0 nth(1 box)) nth(1 nth(0 box)) )
                        )
           )
           intersection_list=eliminateRedundancy(getIntersectionList( forme2point_list forme1point_list))
           if( (length(intersection_list)>=1) ; si il y a un points d'intersection
              touch_list=append(touch_list list(nth(i-1 overlap_list)))
           )
        )
      )
    touch_list
  )
)

procedure( membre( element liste "dl")
 if( progn( nbr_ele=length(liste)
        result=nil
    for( i 1 nbr_ele
 shape=nth(i-1 liste)
 if( ((element->bBox==shape->bBox)    &&
     (element->points==shape->points) &&
    (element->layerName==shape->layerName))
```

```
                result=t
        )

)
        )
    result
    )
)

procedure( eliminateShapeRedundancy( shape_list "l")
 if( progn( nbr_element=length(shape_list)
        clean_list=()
        for(i 1 nbr_element
            if( membre( nth(i-1 shape_list) clean_list)==nil
                clean_list=append(clean_list list(nth(i-1 shape_list)))
        )

            )
        )
    clean_list
    )
)

procedure( writeToFile( net1 net2_list "tl")
 if( (length(net2_list)>0)
    for( i 1 length(net2_list)
        nets=append(list(net1) list(nth(i-1 net2_list)))
    fprintf(errorfile "short %L prob: 0.0001 pinhole\n" nets)
        )
    )
)




procedure( mitellPinholeFaults( library cell view version "tttt")

 if( progn(
;ouverture de la cellule
      printf("opening cell...")
      cellule=dbOpenCellView(library cell view version "a") ; ***
      printf("successfull\n")

;ouverture du fichier de sortie: PHerror.list
      printf("opening PHerror.list file...")
      errorfile=outfile("PHerror.list" "w")
      printf("successfull\n")

metal2_list=eliminateShapeRedundancy(selectLayer("metal2" cellule->shapes))
metal1_list=eliminateShapeRedundancy(selectLayer("metal1" cellule->shapes))
poly1_list=eliminateShapeRedundancy(selectLayer("poly1i" cellule->shapes))
```

```
poly2_list=eliminateShapeRedundancy(selectLayer("poly1i" cellule->shapes))
ndiff_list=eliminateShapeRedundancy(selectLayer("psd" cellule->shapes))
     pdiff_list=eliminateShapeRedundancy(selectLayer("nsd" cellule->shapes))

diff=append(ndiff_list pdiff_list)

;metal2-(metal1/Poly/diffusion) pinhole short
printf("looking for metal2-metal1/Poly/diffusion pinhole short\n")
   for( i 1 length(metal2_list)
 overlap_list=dbGetTrueOverlaps( cellule->cellView nth(i-1 metal2_list)->bBox)
 croise_list =getCroiseList(nth(i-1 metal2_list) overlap_list)
 croise_list=eliminateRedundancy(croise_list->net->name)
      short_list=eliminateFalseShort( nth(i-1 metal2_list)->net->name croise_list)
      ;printf(" pinhole short with net %s :%L\n" nth(i-1 metal2_list)->net->name short_list)
      if( length(short_list)>0 writeToFile(nth(i-1 metal2_list)->net->name short_list))
 )


;metal1-(Poly/diffusion) pinhole short
printf("looking for metal1-poly/diffusion pinhole short\n")
   for( i 1 length(metal1_list)
 first_list=dbGetTrueOverlaps( cellule->cellView nth(i-1 metal1_list)->bBox)
      overlap_list=append( append( append(selectLayer("poly1i" first_list) selectLayer("poly2i" first_list))
        selectLayer("ndiff" first_list)) selectLayer("pdiff" first_list))
 croise_list =getCroiseList(nth(i-1 metal1_list) overlap_list)
      croise_list=eliminateRedundancy(croise_list->net->name)
      short_list=eliminateFalseShort( nth(i-1 metal1_list)->net->name croise_list)
      ;printf(" pinhole short with net %s :%L\n" nth(i-1 metal1_list)->net->name short_list)
      if( length(short_list)>0 writeToFile(nth(i-1 metal1_list)->net->name short_list))
 )
 fprintf(errorfile "this is the end my friend")
     close(errorfile)
   )
   printf("pinholes short search done\n\n")
  )
)
```

Programme de recherche des polygones coupés : getCutList.skill

```
; a partir d'une liste de forme touche par un defaut,
; cette fonction determine celle qui sont coupe par le defaut.


procedure( getCutList(overlap_list defaut "ld")
 if( progn( nbr_overlap=length(overlap_list)
        cut_list=()
   defect.xinf=nth(0 nth(0 defaut->bBox))
   defect.yinf=nth(1 nth(0 defaut->bBox))
   defect.xsup=nth(0 nth(1 defaut->bBox))
       defect.ysup=nth(1 nth(1 defaut->bBox))
        defectpoint_list=list(list(defect.xinf defect.yinf)
  list(defect.xinf defect.ysup)
  list(defect.xsup defect.ysup)
  list(defect.xsup defect.yinf))
        for( i 1 nbr_overlap
           box=nth(i-1 overlap_list)->bBox
      if( (nth(i-1 overlap_list)->objType=="polygon")
              formepoint_list=nth(i-1 overlap_list)->points
              formepoint_list=list(list( nth(0 nth(0 box)) nth(1 nth(0 box)) )
  list( nth(0 nth(0 box)) nth(1 nth(1 box)) )
  list( nth(0 nth(1 box)) nth(1 nth(1 box)) )
  list( nth(0 nth(1 box)) nth(1 nth(0 box)) )
 )
           )

           intersection_list=eliminateRedundancy(getIntersectionList( defectpoint_list formepoint_list))
           ;printf("couche %s\n forme %L" nth(i-1 overlap_list)->layerName formepoint_list)
           ;printf("%L\n" intersection_list)
           if( (length(intersection_list)>=4) ; si il y a quatre points d'intersection
              cut_list=append(cut_list list(nth(i-1 overlap_list)))
            )
)
      )
   cut_list
  )
)
```

Programme de recherche des polygones touchés : getOverlapObject.skill

```
; fonction qui fournie la liste des objets touche par le defaut.
; cette list est: overlap_list
; N.B:toutes les couches sont prises en compte.


procedure( getOverlapObject(cellule defaut "dd")

 overlap_list=dbGetTrueOverlaps( cellule~>cellView
        defaut~>bBox)
)
```

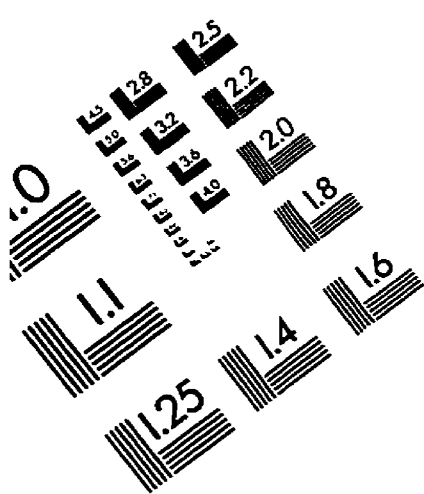
Programme de recherche des polygones couverts : getOverlapObject.skill

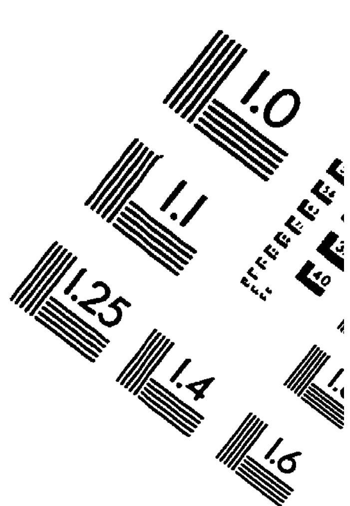
```
; a partir d'une liste de forme touche par un defaut,
; cette fonction determine celle qui sont entierement couverte par le defaut.


procedure( getCoverList(overlap_list defaut "ld")
 if( progn( nbr_element=length(overlap_list)
         cover_list=()
    defect.xinf=nth(0 nth(0 defaut->bBox))
    defect.yinf=nth(1 nth(0 defaut->bBox))
    defect.xsup=nth(0 nth(1 defaut->bBox))
        defect.ysup=nth(1 nth(1 defaut->bBox))
         for( i  1 nbr_element
  forme.xinf=nth(0 nth(0 nth( i-1 overlap_list->bBox)))
  forme.yinf=nth(1 nth(0 nth( i-1 overlap_list->bBox)))
  forme.xsup=nth(0 nth(1 nth( i-1 overlap_list->bBox)))
  forme.ysup=nth(1 nth(1 nth( i-1 overlap_list->bBox)))

                 if( ( (defect.xinf<=forme.xinf)&&(defect.yinf<=forme.yinf)
    &&(defect.xsup>=forme.xsup)&&(defect.ysup>=forme.ysup))
                    cover_list=append( cover_list list( nth( i-1 overlap_list)))
   )
          )
      )
    then cover_list
   )
)
```

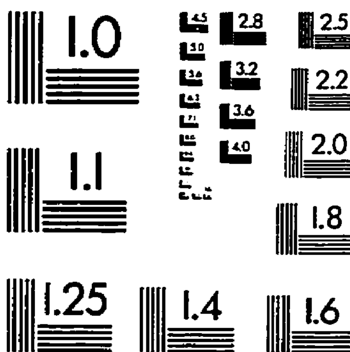
Programme de calcul de probabilité des défauts : computeDefectProb.skill

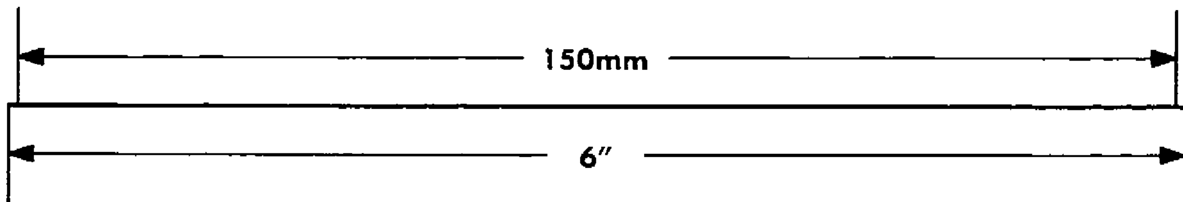
```
; calcule la probabilite d'avoir un defaut de diametre w
; pour avoir une distribution de probabilite quii donne prob(0)=1
; il faut que Dbar=(2*X0)/(2+X0^2)


procedure( computeDefectProb( w X0 Dbar "fff")
 if( progn(
   winf=w
   wsup=w
   if( (w>X0) winf=X0 wsup=X0)
   probsup=(X0*X0*Dbar)/(2*wsup*wsup);
        probinf=(Dbar/(X0*X0*X0))*(X0-winf)


  )
    probinf+probsup
   )
)
```

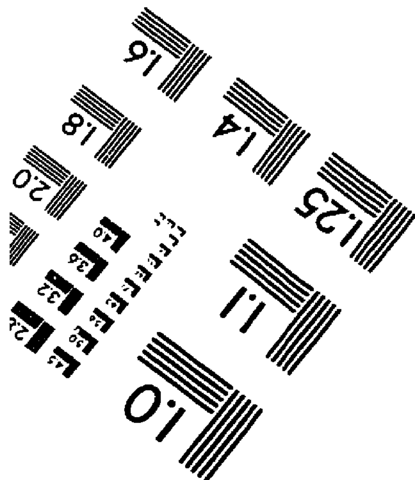
IMAGE EVALUATION
TEST TARGET (QA-3)

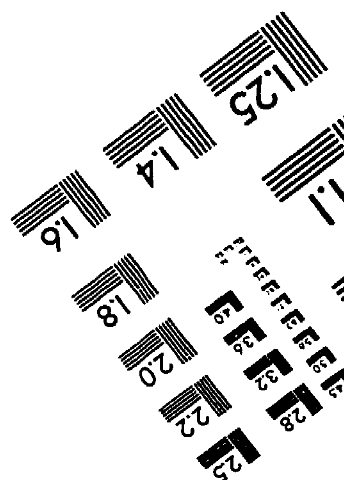












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved