Synthèse Automatique de Contrôleurs avec Contraintes de Sûreté de Fonctionnement

Raphaël Rochet

Thèse

Présentée par

Raphaël ROCHET

Pour obtenir le grade de
Docteur de l'Institut National Polytechnique de Grenoble
(arrêté ministériel du 30 Mars 1992)
(Spécialité **Informatique**)

Synthèse Automatique de Contrôleurs avec Contraintes de Sûreté de Fonctionnement

---

Date de soutenance : 10 Septembre 1996

Composition du Jury :

Messieurs	Guy MAZARE	Président
	Jean-Dominique DECOTIGNIE	Rapporteur
	Christian LANDRAULT	Rapporteur
	Régis LEVEUGLE	
	Mihail NICOLAIDIS	
	Jean-Francois AGAESSE	

Thèse préparée au laboratoire de Conception de Systèmes Intégrés à l'INPG.

Abstract

-- ◊ - ◊◊◊ - ◊ --

CAD Tools are increasingly used by designers, and integrated circuits synthesized with these tools are now extensively used in critical applications. Unfortunately, these tools classically don't take into account the dependability constraints. So we studied and automated the synthesis flows of several Finite State Machine (FSM) architectures with error detection or fault tolerance capabilities. FSMs are the part of integrated circuits implemented to control all the operations performed.

The architectures with error detection capabilities are based on the principle of control flow checking by signature monitoring. The signature is a piece of information representative of the state sequence followed by the FSM. Control flow checking aims at detecting illegal state sequences. The architectures with fault tolerance capabilities are based on the use of error correcting codes during state encoding. If a fault occurs in the implemented FSM, the resulting error may be corrected due to the error correcting code properties.

The analysis of the synthesis results obtained for a lot of examples shows the interest of the synthesis algorithms implemented and of the resulting architectures. In particular, the studied architectures are more difficult to implement than those used during manual implementations, but are more efficient in terms of silicon area overhead for a similar dependability level. So, with the developed tool, designers can obtained new area / dependability trade-offs, which are better suited for industrial control process applications and non-military critical applications.

Key words : Automatic synthesis, Finite state machines, Error detection, Fault tolerance, Control flow checking, Error correcting codes.

Remerciements

-- ◊ - ◊◊◊ - ◊ --

Mes plus grands remerciements à Régis Leveugle, pour avoir dirigé mes recherches, ainsi que pour son aide, sa rigueur et sa motivation qui ont été un soutien constant tout au long de ces dernières années.

Je remercie également :

Monsieur Guy Mazaré, Professeur et directeur de l'ENSIMAG, de me faire l'honneur de présider ce jury.

Monsieur Jean-Dominique Decotignie, Professeur à l'Ecole Polytechnique Fédérale de Lausanne, et Monsieur Christian Landrault, directeur de Recherches CNRS au LIRMM, pour avoir accepté d'être rapporteurs de mon travail, ce malgré le court délai accordé pour rédiger le rapport et malgré la période estivale.

Monsieur Mihail Nicolaïdis, directeur de Recherches CNRS au TIMA, pour avoir accepté de faire partie du jury, pour son amitié et sa bonne humeur.

Monsieur Jean-François Agaesse, chef de projet à la société Thomson-CSF, pour avoir accepté d'être membre de ce jury, et pour sa participation constructive à l'évaluation de l'outil développé au cours de cette thèse.

Je tiens aussi à remercier tous les membres et anciens membres du laboratoire CSI pour leur collaboration et tous ceux qui ont contribué de près ou de loin à l'élaboration de ce travail. Je remercie tout particulièrement Olivier Husson pour sa participation aux travaux exposés dans cette thèse, et Viet, Vincent, Philippe, Xavier, Bobach (dit Roberto), Marie-Claude, Adel, Bernard, Alexandre, David, José, Thierry, Pierre-Yves, Anne, et tant d'autres, pour leur amitié et leur sympathie.

My
~~The~~ Tao Of Love

Vangelis

Résumé

-- ◊ - ◊◊◊ - ◊ --

Cette thèse propose de nouvelles méthodes de synthèse automatique des contrôleurs internes aux circuits numériques. Elles permettent en particulier d'intégrer, directement au niveau du contrôleur, des dispositifs de détection d'erreurs ou de tolérance aux fautes.

En ce qui concerne la détection d'erreurs, quatre flots de synthèse ont été implantés. Deux d'entre eux utilisent la méthode classique de duplication et comparaison, tandis que les deux autres sont basés sur la vérification d'un flot de contrôle par analyse de signature. La signature est une information permettant de caractériser la séquence parcourue d'états du contrôleur. La vérification du flot de contrôle correspond à la détection des séquences illégales d'états.

En ce qui concerne la tolérance aux fautes, quatre flots ont été implantés. Deux d'entre eux utilisent la méthode classique de triplement et vote majoritaire, tandis que les deux autres sont basés sur l'utilisation d'un code correcteur d'erreurs lors du codage du contrôleur. Une erreur survenant dans le code de l'état courant peut ainsi être corrigée en utilisant les propriétés du code correcteur choisi.

L'analyse des résultats de synthèse de nombreux exemples montre l'intérêt des nouvelles méthodes de détection et de tolérance proposées, et des algorithmes de synthèse implantés. Ainsi, ces méthodes et ces algorithmes permettent, entre autres, de définir de nouveaux compromis coût / sûreté de fonctionnement, en réduisant sensiblement le coût matériel de la redondance implantée. L'automatisation des traitements permet de plus de réduire le coût de conception lié à l'amélioration de la sûreté de fonctionnement des contrôleurs, en particulier lorsque des techniques plus pointues sont préférées à la redondance massive. Enfin, du fait de leur faible surface pour un niveau de sûreté de fonctionnement similaire aux implantations basées sur la duplication et le triplement, les alternatives proposées à la redondance massive sont particulièrement bien adaptées aux applications de contrôle de processus industriel et aux applications critiques civiles.

Mots clés : Synthèse automatique, Contrôleurs, Détection d'erreurs, Masquage d'erreurs, Vérification d'un flot de contrôle, Codes correcteurs d'erreurs.

TABLE DES MATIÈRES

Chapitre II - Détection et Tolérance dans les Contrôleurs..... 31

Chapitre III - Synthèse de Contrôleurs avec Détection d'Erreurs 69

LISTE DES DÉFINITIONS ET THÉORÈMES

Définitions :

Propriétés :

Conditions :

Théorèmes :

LISTE DES TABLEAUX ET FIGURES

Tableaux :

Figures :

Figures (suite) :

INTRODUCTION

Nous nous intéressons dans ce document à la ***synthèse automatique de contrôleurs avec contraintes de sûreté de fonctionnement.*** Dans ce titre, les termes « sûreté de fonctionnement » désignent la « propriété permettant aux utilisateurs de placer une confiance justifiée dans le service délivré » par un système [Lapr 88]. On peut, de plus, définir de manière succinte un contrôleur comme étant la partie d'une puce électronique implantée pour définir, séquencer, toutes les opérations effectuées par le circuit intégré numérique. Enfin, la synthèse automatique d'un contrôleur est un processus permettant de générer une description structurelle à base d'éléments logiques simples, à partir d'une spécification fonctionnelle du dit contrôleur. La synthèse automatique de contrôleurs avec contraintes de sûreté de fonctionnement concerne donc la génération, à partir d'une spécification fonctionnelle des contrôleurs, d'une description structurelle bas niveau intégrant des dispositifs permettant d'accroître le niveau de confiance des utilisateurs dans le service délivré par le circuit implanté.

L'étude conjointe de la sûreté de fonctionnement et de la synthèse automatique est motivée par les deux remarques suivantes.

En premier lieu, l'utilisation d'outils de synthèse, maintenant généralisée, lors de la conception des circuits permet, certes, une meilleure productivité et des implantations optimisées, mais réduit considérablement le contrôle du concepteur sur la structure finale du circuit synthétisé. Modifier « à la main » la description générée par la synthèse automatique pour y introduire des dispositifs particuliers est dès lors très difficile et se heurte à des limites très strictes.

En second lieu, deux évolutions antagonistes s'accentuent depuis une décennie. Ainsi, d'un coté, les ASICs (« Application Specific Integrated Circuits » : circuits intégrés pour applications spécifiques) sont de plus en plus utilisés dans des applications critiques :

- soit mettant en jeu des vies humaines comme l'avionique, le nucléaire, ou plus simplement les transports terrestres par l'intermédiaire des dispositifs de déclenchement des éléments de sécurité des voitures, par exemple,
- soit mettant en jeu d'importants investissements pour lesquels la gène occasionnée par le dysfonctionnement d'un système peu s'avérer particulièrement coûteuse, comme dans le domaine bancaire ou certains processus industriels.

D'un autre coté, l'évolution des technologies, à savoir l'intégration de plus en plus importante et la réduction des tensions de fonctionnement, conduit à une augmentation des

probabilités de défaillance des systèmes [Duba 88]. Plus précisément, ces évolutions conduisent à une augmentation de la probabilité des défaillances liées à des fautes survenant de manière temporaire, alors que celles-ci sont déjà à l'origine de quatre-vingt pourcent des défaillances des systèmes électroniques ([Iyer 86]).

Des solutions d'amélioration de la sûreté de fonctionnement des circuits intégrés, par détection ou par masquage d'erreurs, existent, mais il s'agit principalement de techniques mettant en oeuvre des processus suffisamment simples pour être mis en oeuvre manuellement. La solution la plus souvent utilisée est la redondance massive, c'est-à-dire la duplication avec comparaison pour la détection d'erreurs, ou le triplement avec vote majoritaire pour le masquage d'erreurs.

L'intérêt d'outils automatiques de conception prenant en compte des contraintes de sûreté de fonctionnement est donc double. D'abord ils permettraient d'intégrer les préoccupations de sûreté de fonctionnement dans le flot de conception communément utilisé (gain de productivité). Ensuite et surtout, ils autoriseraient la synthèse d'architectures plus difficiles à implanter de façon optimisée, et donc jamais envisagées lors d'implantations manuelles, mais utilisant des techniques de détection ou de masquage d'erreurs plus pointues. L'enjeu peut être soit de réduire le coût matériel des dispositifs implantés pour un niveau de sûreté de fonctionnement équivalent à celui des implantations manuelles, soit de définir de nouveaux compromis entre sûreté de fonctionnement et coût matériel, mieux adaptés à certains types d'applications industrielles.

Nous présentons dans ce document les études ayant conduit à la réalisation d'un tel outil de conception. Cet outil inclut plusieurs flots de synthèse permettant d'obtenir la détection ou le masquage des erreurs dans les contrôleurs. Nos efforts de recherche se sont d'abord portés sur cette partie des circuits intégrés parce qu'elle constitue un bloc particulièrement critique, dont les dysfonctionnements sont difficiles à gérer de manière externe. En effet, les erreurs de séquencement (évolutions erronées de l'état du contrôleur) résultent dans des comportements du circuit difficilement prévisibles par une analyse de défaillance. La propagation des erreurs vers les sorties primaires du circuit ne peut donc pas toujours être évitée : une défaillance critique doit donc être envisagée. Le fait que la propagation des erreurs vers les sorties primaires puisse prendre un temps important tend à agraver le problème, en réduisant les possibilités de détection externe. De plus, [Duba 88] indique que les erreurs résultant de fautes temporaires ont une forte probabilité d'être mémorisées par ce type de bloc, et donc d'avoir un effet artificiellement prolongé sur l'ensemble du système.

Dans la suite, le chapitre deux présente un ensemble non exhaustif, mais toutefois assez représentatif, des différentes techniques de détection et de masquage d'erreurs développées jusqu'ici pour la logique combinatoire et les machines séquentielles. Il présente en particulier les notions de vérification d'un flot de contrôle pour la détection d'erreurs, ainsi que les techniques basées sur l'utilisation de codes correcteurs d'erreurs pour le masquage d'erreurs.

Les chapitres trois et quatre présentent nos travaux proprement dits, en s'appuyant sur les concepts exposés au chapitre deux.

Le chapitre trois est consacré aux flots de synthèse automatisés pour la génération de contrôleurs détectant les erreurs, ainsi qu'à l'analyse des résultats obtenus pour un grand nombre d'exemples universitaires et industriels. La technique utilisée pour la détection est basée sur la vérification d'un flot de contrôle et consiste à vérifier le bon séquencement des états du contrôleur ([Leve 90b]). L'intérêt de ce type de technique par rapport à la duplication est de définir un autre compromis entre coût matériel et sûreté de fonctionnement, mieux adapté, par exemple, aux applications de contrôle des processus industriels peu critiques. La lecture des paragraphes II.1.1.3, II.1.4.1 et II.1.4.3 est indispensable aux personnes non familiarisées avec la notion de

vérification d'un flot de contrôle, pour une bonne compréhension des travaux décrits dans ce chapitre.

Le chapitre quatre présente, ensuite, les flots de synthèse automatisés pour la génération de contrôleurs masquant les erreurs simples, ainsi que les résultats obtenus pour une centaine d'exemples universitaires et industriels. La technique utilisée pour le masquage d'erreurs dans le contrôleur est basée sur l'utilisation de codes correcteurs d'erreurs simples lors du codage des états du dit contrôleur selon les principes exposés dans [Arms 61]. Le but visé par ce type de technique est l'obtention d'un niveau de sûreté de fonctionnement équivalent, du point de vue des fautes simples, à celui pouvant être obtenu avec les techniques basées sur le triplement du bloc à rendre plus sûr, mais à un coût matériel moindre.

Mais avant d'entrer dans ces différentes considérations, le premier chapitre introduit plus complètement les notions de contrôleurs, de synthèse de contrôleurs et de sûreté de fonctionnement.

CHAPITRE I - CONTEXTE ET DÉFINITIONS

Le sujet de cette thèse étant la synthèse automatique de contrôleurs avec contraintes de sûreté de fonctionnement, ce premier chapitre introduit brièvement le contexte de l'étude, la synthèse automatique de manière générale et le système ASYL développé au laboratoire CSI. Les notions de base de la synthèse de contrôleurs et de la sûreté de fonctionnement nécessaires à la bonne compréhension de la suite du document sont aussi présentées.

I.1. NOTIONS GÉNÉRALES

Ce paragraphe introduit la notion de contrôleur et situe la synthèse de contrôleurs parmi les différents flots de synthèse possibles des circuits. Il donne en particulier un bref aperçu de la place de la synthèse de contrôleurs dans l'outil ASYL.

I.1.1. Contrôleurs dans les circuits VLSI

Quatre types de blocs constituent généralement les circuits VLSI numériques. Il s'agit des contrôleurs, des chemins de données, des blocs compilés et de la glu logique.

La glu logique sert le plus souvent à implanter de petites fonctions pour générer des signaux annexes, simples et non structurés. Il s'agit généralement de quelques portes logiques interconnectées.

Les blocs compilés comme les ROM, RAM, multiplieurs et autres sont des blocs à structure très régulière. Les ROM vont par exemple servir dans les contrôleurs microprogrammés, les RAM peuvent être utilisées pour le stockage des poids dans les réseaux de neurones, etc ... Leur intérêt est leur forte densité d'implantation qui permet de réduire le coût en surface du circuit.

Un bloc (souvent) compilé un peu particulier est le chemin de données. Un chemin de données effectue des opérations logiques, arithmétiques ou de stockage sur des données et fournit le résultat de ces opérations ainsi qu'un compte rendu donnant une information qualitative sur le

déroulement des opérations et leur résultat. La taille des données traitées par un chemin de données est généralement fixe. Un chemin de données est essentiellement composé de 3 types d'éléments : les éléments de mémorisation contenant les données (registres), les opérateurs (Unités Arithmétiques et Logiques, ...), les éléments d'interconnexion (bus, multiplexeurs).

Enfin le dernier type de bloc est le contrôleur ou séquenceur. Ce bloc contrôle, séquence, les opérations, flots de données, mémorisations ... effectués par le circuit. Par exemple, c'est lui qui va déclencher l'opération o au temps t puis positionner les signaux nécessaires au transfert du résultat vers un registre r1 via un bus b1 au temps t+1. Ainsi, le contrôleur positionne au bon moment les signaux de contrôle nécessaires à la coordination interne et externe des autres blocs du circuit pour la réalisation du service demandé.

Un circuit peut comporter un ou plusieurs contrôleurs. Plusieurs contrôleurs peuvent aussi communiquer entre eux par l'intermédiaire de signaux de synchronisation. On parle alors de contrôleurs communicants. Dans la suite nous ne nous intéresserons qu'au cas de contrôleurs simples, non communicants. L'extension aux contrôleurs communicants des techniques développées ne pose, a priori, pas de problèmes particuliers, mais reste à étudier.

I.1.2. Flot général de synthèse

I.1.2.1. Définitions

Partant d'une description à un certain niveau d'abstraction d'un circuit, la synthèse est un processus consistant à générer une description de plus bas niveau sans modifier la fonctionnalité de ce circuit.

On différencie plusieurs types de synthèse, selon le niveau d'abstraction de la spécification d'entrée du circuit. Ainsi on parle de *Synthèse de Haut Niveau*, de *Synthèse RTL*, de *Synthèse de Contrôleurs*, de *Synthèse Logique* et de *Synthèse Structurelle* (Fig. I.1).

La synthèse de haut niveau (High Level Synthesis : HLS) consiste à générer une description niveau transfert de registre (Register Transfer Level : RTL), ou même plus bas niveau, à partir d'une description algorithmique du circuit [Gajs 92, West 93]. Une description algorithmique d'un circuit est une description utilisant des variables, sans lien direct avec toute réalisation matérielle précise, et laissant à l'outil de synthèse, ou au concepteur dans le cas d'une synthèse manuelle, la charge de choisir les opérateurs et d'ordonner, « séquencer », les opérations de façon la plus optimale possible. Ainsi, de manière générale, la synthèse de haut niveau commence par la génération d'un Graphe de Flot de Données (GFD), qui consiste en la liste, ou plutôt la succession sous forme de graphe, des opérations décrites par la description algorithmique. Les arcs entre les différents noeuds du graphe décrivent les dépendances entre les données et les opérations. Le GFD est divisé en pas de contrôle (« c-step » en anglais), qui permettent de dater chaque opération. Généralement, une opération, compte tenu des dépendances données-opérations, peut intervenir, au choix, durant plusieurs pas de contrôle. Ainsi, l'étape d'ordonnancement va fixer pour chaque opération, le pas de contrôle durant lequel elle sera effectuée, en fonction du degré de liberté autorisé pour cette opération, représenté par le nombre de pas de contrôle où elle peut effectivement être exécutée, et en fonction du coût matériel qu'implique la décision d'effectuer cette opération durant tel ou tel pas de contrôle. Enfin, la dernière étape consiste à « instancier » les opérations, c'est-à-dire à choisir les unités fonctionnelles qui seront chargées de réaliser telle ou telle opération durant tel ou tel pas de contrôle, à choisir les registres chargés de mémoriser le résultat de ces opérations, et à générer le contrôleur chargé de commander le tout.

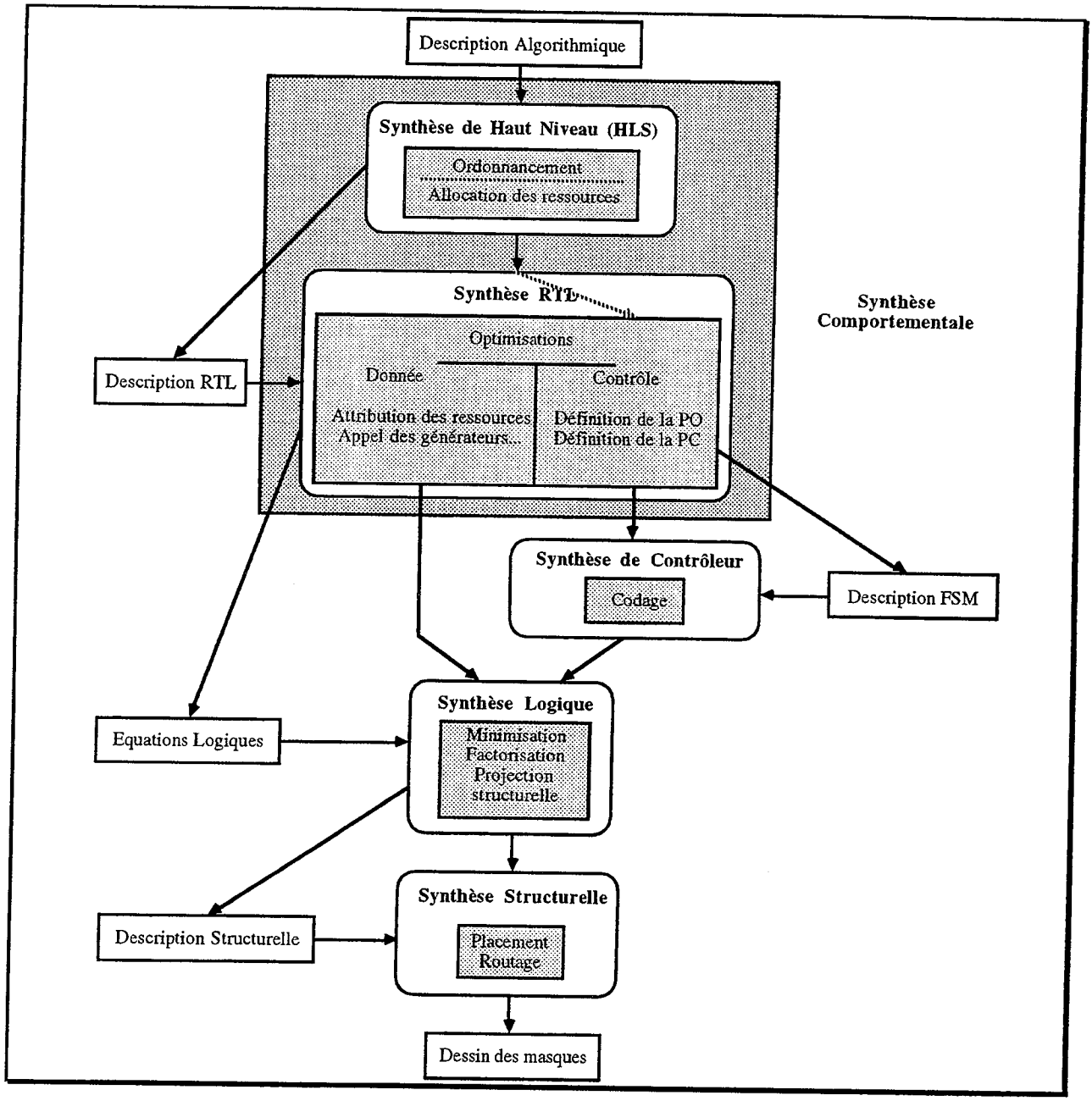

Figure I.1 - Flot général de synthèse [Safi 95].

La synthèse RTL, elle, consiste à générer une description au niveau logique à partir d'une description au niveau RTL [Carl, West 93, Mign 92]. Dans une description de niveau RTL, les opérations à effectuer à chaque cycle fonctionnel (le circuit est modélisé par un processus indéfiniment répété) sont clairement définies, et les traitements sur les données sont définis en terme d'opérations sur des registres [Peng 94]. Une description au niveau logique, quant à elle, est une description sous forme d'équations Booléennes ou de tables de vérité (ce qui revient au même), ou encore une description structurelle pour les chemins de données. La synthèse RTL peut être guidée par le contrôle ou par les données. Dans le second cas, l'approche est généralement incrémentale, les opérations spécifiées étant traduites sur un ensemble de cellules au niveau du bit et les éléments réalisant le contrôle des opérations n'étant pas identifiés explicitement. A l'inverse, une synthèse orientée par le contrôle extrait explicitement les spécifications distinctes d'un contrôleur (partie contrôle, ou PC) et d'un chemin de données (partie opérative, ou PO). Comme il a déjà été mentionné, le contrôleur, généralement spécifié comme une machine d'état finis

(« Finite State Machine » : FSM), est chargé du séquencement des opérations alors que la partie opérative, généralement implantée sous la forme d'un chemin de données, est chargée des opérations sur les données et de leur mémorisation.

La synthèse logique permet de générer une description au niveau structurel, généralement une liste d'interconnexion d'éléments physiques de base décrits dans une bibliothèque, à partir d'équations Booléennes [Sica 88, Abou 92, Sako 93]. La synthèse de contrôleurs peut être considérée comme une extension de cette synthèse logique. Elle consiste à générer les équations Booléennes correspondant à la description de la machine d'états finis fournie par la synthèse RTL orientée par le contrôle, ou à tout graphe d'états spécifié par le concepteur.

La synthèse structurelle permet de générer le dessin des masques du circuit qui sera envoyé en fonderie pour la phase de fabrication, à partir d'une liste d'interconnexion d'éléments physiques et de la bibliothèque correspondante. Dans le cas où on utilise une bibliothèque, elle comprend principalement deux étapes : le placement qui consiste à placer de manière optimale les éléments physiques de base, puis le routage qui correspond à la matérialisation de la liste des interconnexions fournie par la synthèse logique. Le but des diverses optimisations de ces deux étapes est principalement de deux ordres : réduire la surface de silicium nécessaire pour la fabrication du circuit, et réduire la longueur des interconnexions entre les éléments de base.

Dans la suite, par abus de langage, l'expression *Synthèse de Contrôleurs* désignera à la fois la synthèse de contrôleurs proprement dite et la synthèse logique qui s'en suit. Nous focaliserons notre discours sur cette synthèse de contrôleurs, la synthèse comportementale (de haut niveau et RTL) ainsi que la synthèse structurelle étant annexes à notre sujet.

I.1.2.2. Le système ASYL

Le système ASYL développé au laboratoire CSI est un outil de synthèse comportementale. Du fait des différents types de synthèse existant, les outils de synthèse comportementale se distinguent en fonction de deux caractéristiques majeures : le niveau d'abstraction de la spécification initiale et l'approche utilisée lors de la synthèse. Des outils comme MIMOLA [Maew 86], HAL [Paul 86], YSC [Camp 90], CATHEDRAL II [Note 89, Zege 90], AMICAL [Park 92, Park 93], ou le Behavioral Compiler de SYNOPSYS [SYNO] sont des outils de synthèse de haut niveau. Ils sont par essence orientés par le contrôle. D'un autre coté se trouvent les outils de synthèse RTL comme ALLIANCE [ALLI, Grei 93], COMPASS [COMP, Mahm 92], AUTOLOGIC [MENT], ou le Design Compiler de SYNOPSYS [SYNO]. Le système ASYL [ASYL, Ramp 91, Mign 92, Belh 95] se classe dans cette deuxième catégorie. Il offre la possibilité de faire soit une synthèse orientée par les données, soit une synthèse orientée par le contrôle.

Aujourd'hui, la plupart des outils de synthèse comportementale acceptent en entrée une description dans un langage de haut niveau standard (comme VHDL [VHDL 92] ou VERILOG [Thom 91]), ou plus exactement dans un sous-ensemble de ces langages. En effet, toute description VHDL, par exemple, n'est pas forcément synthétisable. A l'heure actuelle, la synthèse orientée par le contrôle d'ASYL accepte uniquement les descriptions écrites avec un sous-ensemble VHDL, les descriptions VERILOG n'étant acceptées que par la synthèse orientée par les données. Dû aux spécificités de la synthèse RTL orientée par le contrôle, le sous-ensemble VHDL accepté pour ce type de synthèse est plus restreint que celui accepté par la synthèse orientée par les données [ASYL]. Il s'agit de la description d'un processus synchrone unique décrivant les états successifs du circuit et les opérations monocycles effectuées dans chaque état. Les structures de type boucles sont interdites, et la construction VHDL doit obligatoirement comporter un registre,

déclaré par l'utilisateur comme registre d'état permettant de stocker l'état courant du circuit, et une liste descriptive des opérations effectuées dans chaque état. Le gabarit de la spécification acceptée pour le circuit est donné en figure I.2.

```
use work.ASYL_RTL.all;
-- Le fichier de paquetage ASYL_RTL contient des attributs et des fonctions

entity <nom_d'entité> is
<déclaration des ports>, <spécification des attributs>
end <nom_d'entité>;

architecture <nom_d'architecture> of <nom_d'entité> is
<déclaration des signaux>, <spécification des attributs>
begin
    -- assignation concurrente des signaux
    <nom_de_port> <= <expression>;
    <nom_de_port> <= <expression_1> when <condition_1> else ...
                        <expression_N-1> when <condition_N-1> else
                            <expression_N>;
    -- <expression> est une constante, un booléen , une expression arithmétique ou un appel
    -- de fonction pour un signal d'entrée ou un signal interne (excepté pour le registre d'état),
    -- <condition> peut être une expression booléenne.
    process (<nom_d'horloge>, <nom_du_raz>) -- process séquentiel
    <déclaration des variables>
    begin
        if <nom_du_raz> = '1' then ...
            -- Remise A Zero asynchrone , les expressions logiques sont acceptées
        elsif <nom_d'horloge>'EVENT and <nom_d'horloge> = '1' then
            case <nom_de_l'état> is
                when <valeur_1> =>
                    <nom_de_signal> <= <nom_de_fonction>
                                        (<nom_de_signal>, <nom_de_variable>,...);
                    <nom_de_l'état> <= <valeur_2>;
                when <valeur_2> =>
                    <nom_de_signal> <= <nom_de_signal>
                                        <opérateur_VHDL> <nom_de_signal>;
                    -- L'opérateur VHDL est soit projeté sur un élément de la
                    -- bibliothèque, soit resynthétisé
                when <valeur_3> =>
                    <nom_de_variable> <= <nom_de_fonction> (<nom_de_signal>,...);
                    <nom_de_signal> <= <nom_de_fonction> (<nom_de_variable>,...);
                    -- Description d'opérations chaînées avec des variables intermédiaires
                when <valeur_4> =>
                    if <nom_de_prédicat> = '1' then
                    -- expressions booléennes
                        <nom_de_signal> <= <nom_de_fonction> (<nom_de_signal>,...);
                    end if;
                    --Opérations et affectations du registre d'état peuvent être conditionnelles.
                ...
            end case;
        end if;
    end process;
end <nom_d'architecture>;
```

Figure I.2 - Gabarit de la spécification VHDL acceptée par ASYL pour la synthèse RTL orientée par le contrôle [ASYL].

Etant intéressés par les contrôleurs, nous laisserons de coté la synthèse orientée par les données où les fonctions de contrôle ne sont pas clairement identifiées. Nous verrons par contre un peu plus en détail la synthèse orientée par le contrôle qui repose sur une architecture PC-PO. L'architecture PC-PO est une architecture classique, utilisée par exemple dans [Rama 92] et

[Gajs 88]. Le circuit synthétisé automatiquement est constitué de deux blocs : le contrôleur et le chemin de données (Fig. I.3). Le contrôleur reçoit les signaux de contrôle externes et les comptes rendus en provenance du chemin de données. En fonction de ces signaux, il détermine l'état suivant du circuit et les signaux de contrôle associés qui seront envoyés au chemin de données pour commander ses éléments (opérateurs, registres, multiplexeurs, portes trois états, ...). Le chemin de données effectue les traitements et calculs commandés sur les données présentent dans les registres ou en entrée des opérateurs, et fournit en sortie les données résultantes et les comptes rendus qui serviront au calcul de l'état suivant par le contrôleur.

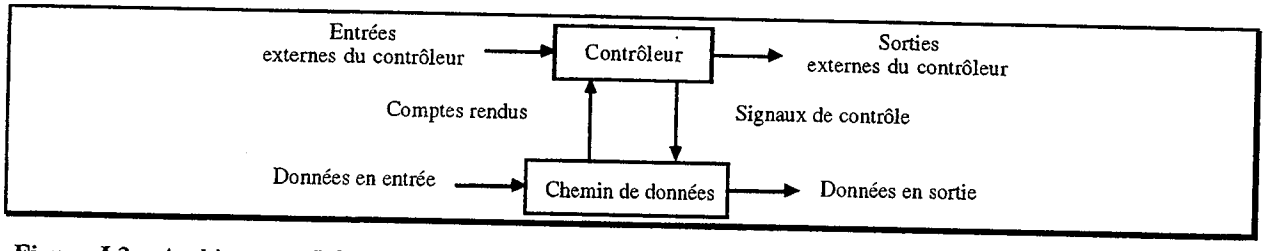

Figure I.3 - Architecture PC-PO.

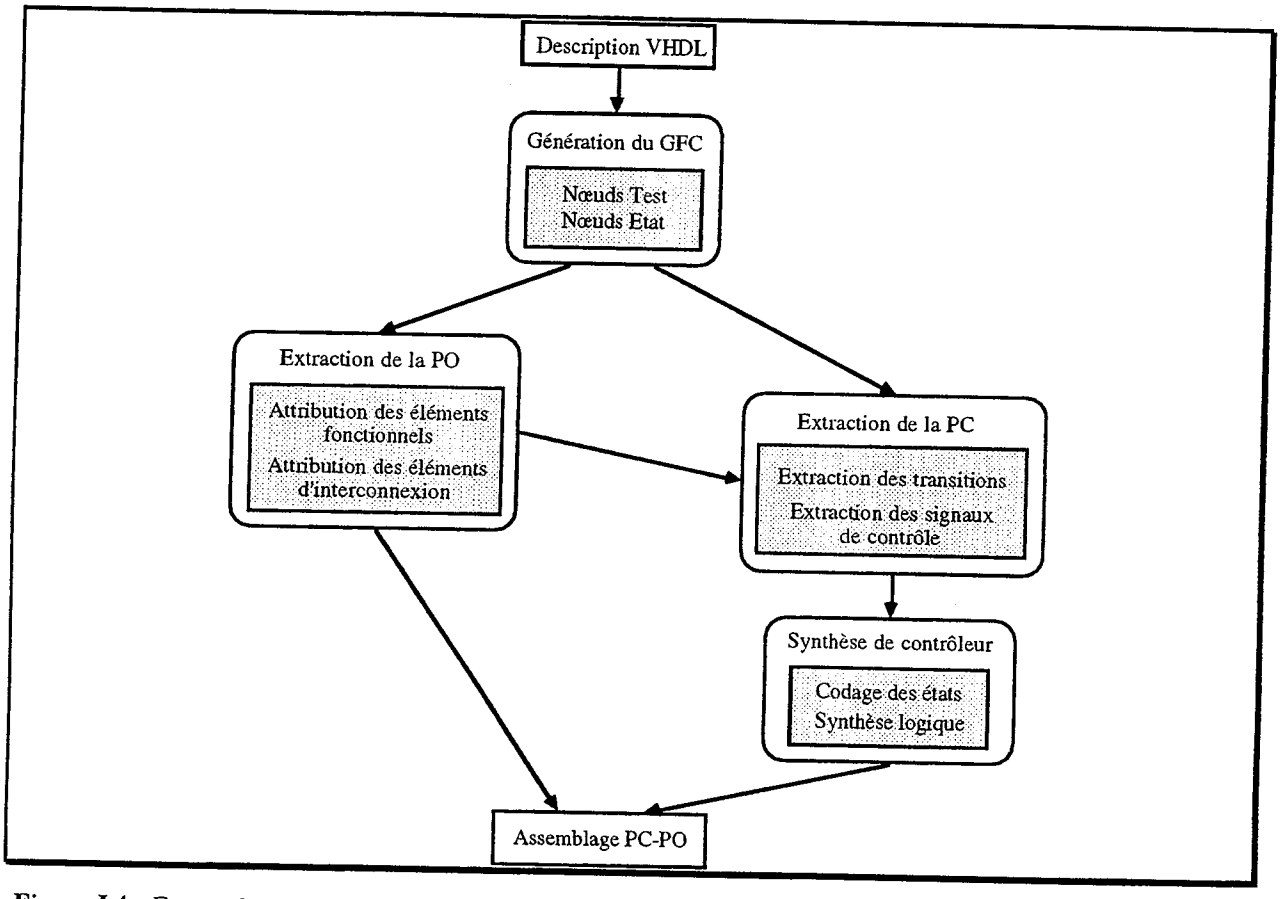

Figure I.4 - Etapes de la synthèse RTL orientée par le contrôle [Safi 95].

Lors d'une synthèse RTL orientée par le contrôle, les étapes effectuées par ASYL sont les suivantes (Fig. I.4). ASYL commence par créer une structure particulière appelée Graphe de Flot de Contrôle (GFC, [Aho 88]) à partir de la description VHDL qui lui est fournie. Ensuite la synthèse proprement dite débute par l'extraction de la partie opérative [Safi 95]. Une fois la PO extraite, elle est sauvegardée dans un fichier au format EDIF [EDIF 88, EDIF 89] qui comporte donc la liste des ressources fonctionnelles utilisées (éléments de mémorisation, opérateurs) et la liste des interconnexions entre ces éléments fonctionnels. Trois types d'architectures sont

possibles pour la PO : soit des architectures à base de multiplexeurs, aussi appelées architectures à topologie aléatoire, soit des architectures à base de bus, aussi appelées architectures à topologie linéaire, soit des architectures mixtes où le meilleur compromis est recherché entre l'utilisation de multiplexeurs et l'utilisation de bus ou segments de bus, en fonction des contraintes de surface et de vitesse. La synthèse se poursuit par l'extraction de la PC [Safi 95]. Ceci consiste en l'élaboration de la spécification d'un contrôleur unique, synchrone, destiné à commander la partie opérative générée. L'extraction de la PC est réalisée en deux étapes : extraction des transitions et extraction des signaux de contrôle. Ensuite les procédures de synthèse de contrôleurs sont appelées afin de générer la liste d'interconnexions de portes logiques correspondant au contrôleur. Enfin, une dernière étape crée le fichier final décrivant les interconnexions entre la PO et la PC.

Pour ce qui nous concerne, outre la synthèse PC-PO, ASYL offre aussi la possibilité de synthétiser directement un contrôleur (sans génération de partie opérative), à partir d'un graphe d'états décrit en VHDL. L'utilisateur peut donc synthétiser un contrôleur seul. Un exemple de contrôleur seul spécifié en VHDL est fourni au paragraphe I.2. qui décrit de façon détaillée la synthèse de contrôleurs dans ASYL.

I.1.3. Contrôleurs et machines d'états finis

Cette section fournit une définition formelle des contrôleurs, ainsi que l'architecture utilisée par la synthèse et les cibles technologiques possibles.

I.1.3.1. Définitions

Un contrôleur est une forme étendue de machine d'états finis. On définit ci-dessous les machines de Moore et de Mealy.

- **Définition I.1 : Machine d'états finis**

*Une **machine d'états finis** est définie par un quintuplet $(I, O, S, \delta, \omega)$ où :*

I est un ensemble fini d'entrées,
O est un ensemble fini de sorties,
S est un ensemble fini d'états,

δ est une application de $I \times S \rightarrow S$, appelée fonction d'état suivant,
ω est l'une des deux applications suivantes :
$\omega : S \rightarrow O$ *dans le cas des automates de Moore,*
$\omega : I \times S \rightarrow O$ *dans le cas des automates de Mealy,*
ω est appelée la fonction de sortie.

A un instant donné, la machine (ou automate) se trouve dans un état s, appelé état courant. Elle évoluera en fonction des entrées vers un état s', appelé état suivant. L'évolution de l'état s vers l'état s' définit une transition.

- **Définition I.2 : Transition**

*Une **transition** est définie par :*
*- un triplet (s',i,s) vérifiant $s' = \delta(i,s)$ dans le cas d'un automate de **Moore**,*

*- par un quadruplet (s',o,i,s) vérifiant (s',o) = δ (i,s) × ω(i,s), pour les automates de **Mealy** ,*

où o ∈ O, s ∈ S, s' ∈ S et i ∈ I . i est appelé la condition sur les entrées permettant la transition (s', i, s) ou (s', o, i, s). o est la sortie associée à la transition s'→ s conditionnée par i dans le cas des automates de Mealy.

Dans ces définitions, les entrées, états et sorties sont des éléments symboliques sans aucune référence à un codage binaire. La synthèse va coder cet automate afin de transformer les fonctions δ et ω en fonctions Booléennes pouvant être implantées à l'aide de semiconducteurs.

- **Définition I.3 : Machine d'états finis codée**

*Une **machine d'états finis codée** est définie par un quintuplet (I, O, Y, δ, ω) où :*

I est une variable Booléenne générale dont chaque bit est appelé variable d'entrée,
O est une variable Booléenne générale dont chaque bit est appelé variable de sortie,
Y est une variable Booléenne générale dont chaque composante est appelée variable interne. Y représente les états de la machine.

δ est une fonction Booléenne de $I \times Y \rightarrow Y$, appelée équation des variables internes.
ω est l'une des deux fonctions Booléennes suivantes :
$\omega : Y \rightarrow O$ dans le cas des automates de Moore,
$\omega : I \times Y \rightarrow O$ dans le cas des automates de Mealy,
et est souvent appelée équation des sorties.

La condition i d'une transition est alors notée sous la forme d'un monôme canonique représentant une valeur de la variable Booléenne I, c'est-à-dire les valeurs logiques des entrées de l'automate. De même, pour les sorties, o est représenté par un monôme définissant les valeurs logiques des sorties.

La notion de contrôleur est une extension de la notion de machine d'états finis où la condition de transition sur les entrées ne se limite plus à un monôme canonique mais peut être multiple. En d'autres termes, pour un contrôleur, une transition est définie par un triplet (s', i, s) ou un quadruplet (s', o, i, s), où i peut être une somme de monômes et ainsi représenter plusieurs valeurs de la variable Booléenne I . i est alors appelé **prédicat**. Dans la suite, par abus de langage, « machine (ou automate) d'états finis » sera employé comme un synonyme de contrôleur.

Remarque : on ne considérera que les contrôleurs synchrones déterministes. Le terme synchrone signifie que le temps est discrétisé et que les événements résultant du fonctionnement de l'automate peuvent-être datés. On peut alors écrire la fonction d'état suivant et la fonction de sortie de la manière suivante (s(t), i(t) et o(t) représentent respectivement l'état, les entrées et les sorties de la machine à un instant t donné):

$s(t+1) = \delta(i(t), s(t))$,
$o(t) = \omega(s(t))$ pour un automate de **Moore**,
$o(t) = \omega(i(t), s(t))$ pour un automate de **Mealy.**

Le terme déterministe signifie que pour un état et une valeur d'entrée donnés, il y a un seul état suivant. Cela se traduit par une contrainte portant sur les prédicats P_i des transitions issues d'un même état s, à savoir que les prédicats sont exclusifs 2 à 2 ($P_i * P_j = 0$). De plus, en principe, un contrôleur déterministe est totalement spécifié, c'est-à-dire que la somme des prédicats Pi est

une tautologie ($\Sigma P_i = 1$). Toutefois, dans la suite, les contrôleurs pourrons ne pas être totalement spécifiés, le choix de l'état successeur pour les valeurs d'entrée non spécifiées étant laissé à la discrétion du processus de synthèse. Le degré de liberté ainsi obtenu lors de la synthèse peut être utilisé à des fins d'optimisation.

Le mode de représentation le plus classique des automates est le graphe de contrôle, appelé aussi graphe d'états ou graphe de transitions.

- **Définition I.4 : Graphe de contrôle**

Un graphe de contrôle G représentant un contrôleur C est un graphe orienté G(V, T) où :
- *V est l'ensemble des noeuds du graphe associé bijectivement à l'ensemble S des états du contrôleur C,*
- *T est l'ensemble des arcs associé bijectivement à l'ensemble des transitions du contrôleur C.*

Chaque noeud du graphe est étiqueté avec le nom symbolique de l'état qu'il représente, tandis que les entrées et les sorties sont représentées par leurs valeurs codées. Ainsi, chaque arc de G est étiqueté avec le prédicat de la transition correspondante, sauf lorsque ce prédicat est une tautologie. De plus, dans le cas des automates de ***Moore****, chaque noeud est étiqueté par un vecteur des variables de sorties égales à 1 pour l'état correspondant du contrôleur C. Dans le cas des automates de* ***Mealy****, chaque arc est étiqueté par un vecteur des variables de sortie égales à 1 pour la transition correspondante.*

La figure I.5 montre la représentation sous forme de graphe de contrôle de deux automates ayant la même fonctionnalité, un de Moore et un de Mealy.

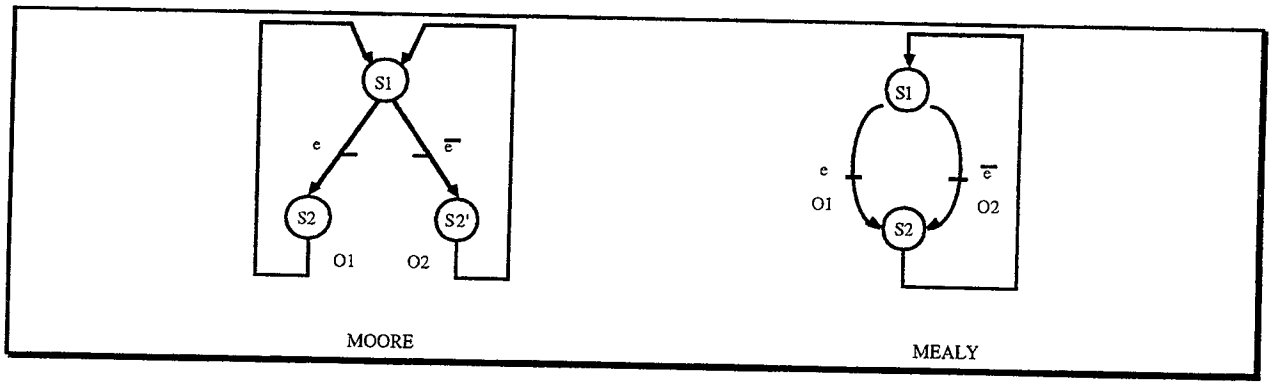

Figure I.5 - Graphe de contrôle.

Afin de simplifier la terminologie, les arcs du graphe de contrôle seront appelés indifféremment transition du graphe ou du contrôleur, et les noeuds du graphe de contrôle seront appelés indifféremment états du graphe ou du contrôleur.

I.1.3.2. Architecture générale

L'architecture de base, dénommée « simplex », classiquement utilisée pour implanter les contrôleurs est représentée en figure I.6. Cette architecture comporte un registre, appelé registre d'états, permettant de mémoriser l'état courant de la machine. Les fonctions δ et ω sont implantées à l'aide de logique combinatoire multicouche. La logique combinatoire permettant de calculer δ sera appelée « logique de calcul de l'état suivant » ou plus simplement « logique d'état suivant ».

La logique combinatoire permettant de calculer ω sera appelée « logique de calcul des sorties » ou plus simplement « logique de sortie ». Le temps est discrétisé par l'intermédiaire d'un signal d'horloge. Un signal d'initialisation (reset) permet de forcer un état prédéfini en début d'opération.

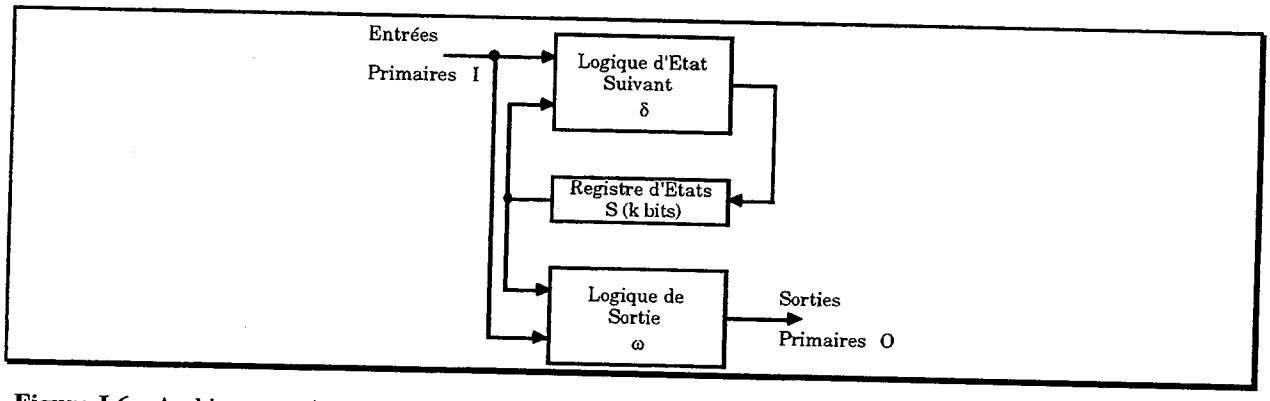

Figure I.6 - Architecture simplex (blocs fonctionnels).

Chaque état est représenté par un vecteur binaire, c'est-à-dire par une valeur de la variable Booléenne générale Y mentionnée dans la définition I.3. Ces vecteurs sont appelés codes d'états et chaque bascule du registre d'état permet de mémoriser la valeur d'une des variables internes, à savoir un des bits des codes d'états. Le nombre de bits minimum nécessaire pour coder tous les états de la machine sera appelé k (k = partie_entière_supérieure($\log_2$(|S|)) où |S| est le nombre d'états du contrôleur).

Différentes cibles technologiques peuvent être utilisées avec l'outil ASYL. On peut citer les cellules standard (prédiffusé et précaractérisé) [Sako 93], les FPGA (Xilinx, Actel, ...) [Bess 92] et les CPLD (PAL, PLA, ...) [Gerb 92]. Les contrôleurs peuvent aussi être synthétisés sur ROM [Gerb 94]. Dans la suite nous nous limiterons aux cellules standard et équivalents.

I.2. ETAPES DE LA SYNTHÈSE DE CONTRÔLEURS DANS ASYL

Le paragraphe précédent a introduit la notion de contrôleurs et a situé la synthèse de contrôleurs parmi les différents flots de synthèse existant, et en particulier les différents flots de synthèse existant dans l'outil ASYL. Cette seconde partie a pour but de détailler la synthèse de contrôleurs dans ASYL, les travaux présentés aux chapitres III et IV y étant directement liés. Seule l'étape de codage des états (paragraphe I.2.2.) fera l'objet d'une description un peu plus large incluant d'autres outils de synthèse.

I.2.1. Spécification initiale

Comme nous l'avons déjà mentionné, le système ASYL accepte deux types de spécification conduisant à la synthèse de contrôleurs : soit une spécification niveau RTL conduisant à une synthèse PC-PO, soit une description niveau RTL d'un contrôleur seul [ASYL].

Le gabarit de spécification VHDL accepté par ASYL pour la synthèse PC-PO a été donné en figure I.2 (paragraphe I.1.2.2.). Le format de description VHDL d'un contrôleur seul est identique, à ceci près que les opérations sur les registres sont exclues, évidemment, si on veut éviter l'extraction d'une partie opérative. La description VHDL d'un contrôleur comporte donc principalement un registre d'état, un processus ayant pour liste de sensibilité le signal d'horloge et

le signal d'initialisation, et une structure « case » sur la valeur du registre d'état permettant de lister, pour chaque état, les signaux à positionner (Fig. I.7).

```
use work.ASYL_RTL.all;

entity Controleur is

    port (  Reset     :    IN Bit;
            Horloge   :    IN Bit;
            Entrée    :    IN Bit;
            Sortie    :    OUT Bit);
    attribute WIR_TYP of Sortie     :    signal is OUT_CONTROL;

end Controleur;

architecture Exemple of Controleur is

    type TypeEtat is ( s1, s2, s3, s4 );
    signal Etat : TypeEtat;
    attribute WIR_TYP of Etat : signal is STATE_REG;

begin
    process ( Horloge, Reset )
    begin
        if Reset = '1' then
            Etat <= s1;
        elsif Horloge'EVENT and Horloge = '1' then
            case Etat is
                when s1 =>
                    Sortie <= '0';
                    if Entrée = '0'  then     Etat <= s2;
                                     else     Etat <= s3;
                    end if;
                when s2 =>
                    Sortie <= '0';
                    if Entrée = '0'  then     Etat <= s4;
                                     else     Etat <= s2;
                    end if;
                when s3 =>
                    Sortie <= '1';
                    Etat <= s1;
                when s4 =>
                    Sortie <= '1';
                    Etat <= s1;
            end case;
        end if;
    end process;
end Exemple;
```

Figure I.7 - Exemple de description VHDL d'un contrôleur.

I.2.2. Flot de synthèse d'un contrôleur

La synthèse de contrôleurs se déroule en cinq étapes : le codage des états, la génération des équations, la minimisation des équations, la factorisation et la projection structurelle sur la cible technologique choisie.

I.2.2.1. Le codage

Le codage des entrées et des sorties est très souvent implicite et fourni par le graphe de contrôle. Par contre, coder les états d'un automate, c'est-à-dire associer injectivement à chaque état $s \in S$ une valeur de Y, se fait explicitement. Si on appelle N le nombre de variables internes, ceci implique que $2^N \geq |S|$, ou encore $N \geq \log_2(|S|)$, la valeur minimale de N étant k (k est défini au paragraphe I.1.3.2).

Dans la suite, on notera y_j la valeur de la $j^{ième}$ composante de la variable Booléenne Y à l'instant t, et Y_j la valeur de la $j^{ième}$ composante de la variable Booléenne Y à l'instant t+1. On notera aussi C(s) le code de l'état s et $C_j(s)$ le $j^{ième}$ bit du code de s.

Différents types de codages sont implantés dans ASYL comme le codage de Gray, le codage de Johnson, le codage séquentiel, ... Les deux plus connus et utilisés sont le codage 1 parmi n et le codage compact optimisé [Duff 91, Belh 93, Gerb 94]. Le codage 1 parmi n est un codage où une seule des composantes de Y vaut 1, et ceci pour tous les codes d'états du contrôleur. Ceci signifie que N = |S|. Le codage compact est un codage où N atteint sa plus petite valeur, c'est-à-dire k. Les algorithmes de codage seront davantage détaillés en section I.3.

I.2.2.2. Génération des équations Booléennes

On appelle $M(s)$ le monôme canonique associé au code de l'état s. Par exemple, si le code de l'état s est C(s) = 00110 (N = 5), alors $M(s) = \overline{y_5} \cdot y_4 \cdot y_3 \cdot y_2 \cdot \overline{y_1}$.

On appelle $E(s,i)$ l'expression Booléenne associée à une transition (s', i, s) ou (s', o, i, s). $E(s,i)$ est égale au produit du monôme $M(s)$ et du prédicat i de cette transition (dont s est l'état source). Par exemple, si I, la variable Booléenne générale représentant les entrées du contrôleur, comprend deux composantes i_1 et i_2, et si N = 2, une transition d'état source s de code C(s) = 01 et de prédicat $i = i_1 \cdot \overline{i_2} + \overline{i_1} \cdot i_2$, donne l'expression $E(s,i) = \overline{y_2} \cdot y_1 \cdot (i_1 \cdot \overline{i_2} + \overline{i_1} \cdot i_2)$.

Dans le cas d'un automate de Moore, l'équation d'une variable interne y_j est la somme des expressions Booléennes $E(s,i)$ de toutes les transitions $(s', i, s) \in T$ telles que $C_j(s') = 1$:

$$Y_j = \sum_{\{(s', i, s) \in T / Cj(s') = 1\}} E(s,i)$$

L'équation d'une variable de sortie o_j pour un automate de Moore est la somme de tous les monômes $M(s)$, $s \in S$, tels que $(\omega(s))_j = 1$:

$$o_j = \sum_{\{s \in S / (\omega(s))j = 1\}} M(s)$$

Dans le cas d'un automate de Mealy, l'équation d'une variable interne y_j est la somme des expressions Booléennes $E(s,i)$ de toutes les transitions $(s', o, i, s) \in T$ telles que $C_j(s') = 1$:

$$Y_j = \sum_{\{(s', o, i, s) \in T / Cj(s') = 1\}} E(s,i)$$

L'équation d'une sortie o_j pour un automate de Mealy est la somme des expressions Booléennes $E(s,i)$ de toutes les transitions (s', o, i, s) telles que $(\omega(i,s))_j = 1$:

$$o_j = \sum_{\{(s', o, i, s) \in T / (\omega(i,s))j = 1\}} E(s,i)$$

Prenons par exemple l'automate de Moore décrit par la figure I.7. Affectons respectivement les codes 00, 01, 10 et 11 aux états s1, s2, s3, et s4. Appelons y_2 la variable interne représentée par le bit de poids fort des codes d'états (bit le plus à gauche) et y_1 la variable représentée par le bit de poids faible (bit le plus à droite). Appelons o_1 l'unique sortie et i_1 l'unique entrée. On a alors :

$$Y_2 = \overline{y_2} \cdot \overline{y_1} \cdot i_1 + y_2 \cdot y_1 \cdot \overline{i_1}$$

$$Y_1 = \overline{y_2} \cdot \overline{y_1} \cdot \overline{i_1} + \overline{y_2} \cdot y_1 \cdot i_1 + y_2 \cdot y_1 \cdot \overline{i_1}$$

$$o_1 = y_2 \cdot \overline{y_1} + y_2 \cdot y_1$$

I.2.2.3. Minimisation, factorisation et projection structurelle

La minimisation consiste à simplifier les équations des variables internes et des sorties en utilisant les propriétés de l'algèbre de Boole [Kunt 68, Bray 85, Sica 88] . La minimisation peut se faire localement ou globalement. La minimisation locale consiste à minimiser le nombre de littéraux de chaque équation prise indépendamment des autres (un littéral correspond à une occurrence de variable Booléenne simple directe ou complémentée). La minimisation globale consiste à minimiser le nombre de littéraux globalement sur l'ensemble des fonctions en maximisant le nombre de monômes communs aux différentes équations. Dans l'exemple de la figure I.7, une minimisation locale générera les équations suivantes :

$$Y_2 = \overline{y_2} \cdot \overline{y_1} \cdot i_1 + \overline{y_2} \cdot y_1 \cdot \overline{i_1}$$

$$Y_1 = \overline{y_2} \cdot \overline{i_1} + \overline{y_2} \cdot y_1$$

$$o_1 = y_2$$

La factorisation consiste à mettre en évidence des facteurs, ou plus généralement des sous-fonctions, communs aux différentes équations [Bray 82, Abou 92, Sako 93]. Ces sous-fonctions communes à plusieurs équations ne seront alors implantées qu'une seule fois. Par exemple, pour le contrôleur de la figure I.7, les équations peuvent se réécrire sous la forme :

$$Y_2 = sf_1 \cdot (\overline{y_1} \cdot i_1 + y_1 \cdot \overline{i_1})$$

$$Y_1 = sf_1 \cdot (\overline{i_1} + y_1)$$

$$o_1 = \overline{y_2}$$

$$sf_1 = \overline{y_2}$$

Nous aurions aussi pu créer une sous-fonction sf_2 égale à $\overline{i_1}$, mais $\overline{i_1}$ ne pouvant être mis en facteur, la création de sf_2 est moins intéressante que celle de sf_1 : les contraintes qu'elle implique au niveau du routage n'est pas forcément avantageux par rapport à l'optimisation du nombre de portes logique utilisées.

Enfin la projection structurelle consiste à rechercher les éléments de la bibliothèque de cellules choisie, qui permette d'implanter physiquement les fonctions Booléennes ainsi obtenues [Sako 93]. Dans l'exemple de la figure I.7, la sous fonction ($\overline{y_1} \cdot i_1 + y_1 \cdot \overline{i_1}$) de l'équation d'Y_2 pourra être implantée, entre autres, soit avec des inverseurs et des portes logiques ET et OU, soit directement avec une porte OU Exclusif, selon les éléments existant dans la bibliothèque cible.

Il résulte de ce qui précède que certaines sous-fonctions sont partagées entre les différentes fonctions synthétisées. Dans le cas des contrôleurs, la figure I.6 doit donc être modifiée, si on veut non plus représenter l'architecture simplex du point de vue des bloc fonctionnels, mais du point de vue de l'implantation physique. En effet, de la logique est partagée entre les différentes fonctions de calcul de l'état suivant, entre les différentes fonctions de calcul des sorties, mais aussi entre les fonctions de calcul de l'état suivant et les fonctions de calcul des sorties. Ainsi, si du point de vue fonctionnel, le bloc de calcul de l'état suivant et le bloc de calcul des sorties sont bien séparés, du

point de vue de l'implantation physique réalisée par ASYL, de la logique étant partagée, ces deux blocs fonctionnels sont difficilement séparables. La figure I.8 donne une illustration de l'architecture simplex plus proche de l'implantation physique.

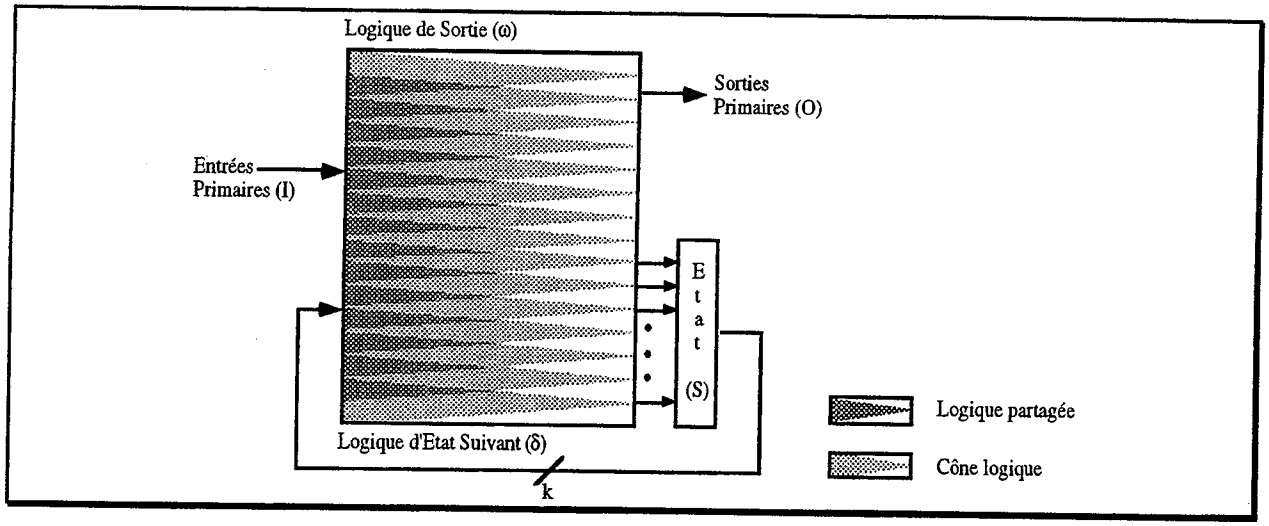

Figure I.8 - Architecture simplex (implantation physique).

I.2.2.4. Contraintes de surface et de vitesse

On sent assez intuitivement que la factorisation, en réduisant le nombre de fois où une même sous-fonction est implantée, tend à réduire la surface de silicium nécessaire à l'implantation du circuit final [Abou 92, Sako 93]. En fait, la factorisation peut aussi jouer sur la vitesse finale du circuit. En limitant le nombre de niveaux de factorisation, on peut limiter le nombre de couches de logique nécessaires pour l'implantation des fonctions, et donc accélérer le temps de calcul de ces fonctions. Mais réduire la surface et augmenter la vitesse sont généralement antagoniques. Lors de la factorisation, l'utilisateur d'ASYL peut donc préciser s'il veut une optimisation en vitesse ou une optimisation en surface. Selon la contrainte choisie (surface ou vitesse), sans entrer dans le détail, le processus de factorisation cherchera donc soit à minimiser la profondeur de factorisation, soit à maximiser le nombre de facteurs communs.

La projection structurelle peut elle aussi être réalisée différemment selon que l'utilisateur désire réduire la surface du circuit ou en augmenter les performances temporelles. Pour cela, les algorithmes vont jouer sur le choix des éléments de base existant dans la bibliothèque cible [Sako 93].

Les procédures de minimisation et de codage d'ASYL ne permettent pas à l'utilisateur de préciser la contrainte de synthèse. Toutefois, nous verrons que le choix du code des états a un impact direct sur la surface et la rapidité du circuit synthétisé.

I.2.3. Codage compact optimisé

I.2.3.1. Impact du codage sur les équations

Considérons l'automate de Moore donné par la figure I.7. Si les codes 10, 01, 11 et 00 sont affectés respectivement aux états s1, s2, s3 et s4, on obtient les équations minimisées suivantes composées d'un total de 14 littéraux :

$$Y_2 = \overline{y_2} \cdot \overline{y_1} + y_2.(y_1+i_1)$$

$$Y_1 = y_2.\overline{y_1} + \overline{y_2}.y_1.i_1$$
$$o_1 = \overline{y_2}.\overline{y_1} + y_2.y_1$$

Par contre, si on choisit les codes 00, 01, 11 et 10, on obtient de nouvelles équations minimisées comprenant 9 littéraux :

$$Y_2 = \overline{y_2}.(\overline{y_1}.i_1 + y_1.\overline{i_1})$$
$$Y_1 = \overline{y_2}.(i_1 + \overline{y_1})$$
$$o_1 = y_2$$

On constate donc que le deuxième codage conduit à des équations dont la complexité est beaucoup plus réduite en terme de nombre de littéraux. En réduisant le nombre d'opérateurs Booléens et le nombre d'interconnexions, le choix judicieux des codes d'états permet ainsi de réduire sensiblement la surface finale du circuit et d'augmenter ses performances temporelles.

I.2.3.2. Algorithme de codage compact optimisé

Les procédures de codage compact optimisé classiques comportent toutes clairement deux étapes. La première correspond à une reconnaissance, dans le graphe d'états, de situations susceptibles de conduire, par un traitement adéquat, à des optimisations. La seconde étape correspond à l'immersion dans un hypercube, c'est-à-dire à l'affectation d'un code binaire à chaque état, guidée par les situations trouvées lors de la première étape.

Tous les outils existants (KISS [DeMi 84], MUSTANG [Deva 88], NOVA [Vill 89], JEDI [Lin 89], ASYL [Sauc 90], ...) commencent par une analyse du graphe d'états pour définir un ensemble de ces situations. Un exemple courant d'une telle situation est l'existence, dans le graphe d'état, d'un ensemble d'états s_i ayant un même successeur pour une même condition sur les entrées. Une telle situation conduit en effet à avoir une expression Booléenne, somme des monômes $M(s_i)$, qui apparaît dans les équations de toutes les variables internes égales à 1 dans le code de l'état suivant. Placer les états s_i sur une même face de l'hypercube, c'est-à-dire leur affecter des codes binaires adjacents, peut permettre de réduire cette expression Booléenne à un simple monôme. En fait, à chaque situation identifiée dans le graphe est associée une contrainte sur l'ensemble des états concernés, représentée par une « contrainte (ou un groupe) d'adjacence » dans le cas de KISS, NOVA et ASYL, ou des « (valeurs d') attractions » dans le cas de JEDI et MUSTANG. Evidemment, des situations beaucoup plus complexes peuvent être identifiées dans le graphe [Sauc 90]. Le problème est alors d'affecter aux états des codes binaires respectant au mieux les contraintes d'adjacence, en commençant par les situations les plus susceptibles de conduire à de fortes optimisations.

KISS utilise un algorithme de codage dit « par colonne » que nous n'aborderons pas plus parce qu'il ne garantit pas, contrairement aux autres outils cités ici, le nombre minimal de variables internes. De plus il cible uniquement les implantations en logique deux couches. MUSTANG et JEDI placent les états d'une même situation dans un proche voisinage dans l'hypercube, en affectant des codes proches (en terme de distance de Hamming) aux états ayant une forte attraction entre eux. Du fait des algorithmes utilisés, ceci correspond à une minimisation globale des distances de Hamming entre les différents codes d'états. L'algorithme utilisé par NOVA revient à associer à chaque groupe d'adjacence un cube, de telle sorte que les relations d'intersection et d'inclusion entre groupes soient respectées dans le cube. Cela ressemble à ce qui est fait dans ASYL, à ceci près que NOVA génère exhaustivement tous les cubes jusqu'à trouver le bon, tandis qu'ASYL va construire le cube au vu des contraintes d'adjacences à respecter. Pour ce faire,

l'affectation des codes binaires aux états est basée sur une théorie dite des cubes intersectants, et prend en compte globalement les intersections entre les contraintes d'adjacence. De plus, elle tend à favoriser une minimisation locale des distances par rapport à une minimisation globale. Il a en effet été démontré que cela minimisait la surface de routage lors du dessin des masques [Sauc 90, Duff 91].

L'impact d'un codage compact optimisé est loin d'être négligeable. Ainsi, comparé au codage aléatoire (résultat moyen de cinq codages aléatoires), la procédure de codage compact optimisé d'ASYL permet un gain moyen de 32% en surface et 15% en chemin critique (résultats obtenus sur un ensemble de 32 exemples internationaux) [Duff 91]. Lorsqu'on compare le codage compact optimisé au codage 1 parmi n, il faut différencier les contrôleurs de complexité moyenne de ceux plus conséquents. Ainsi, lorsqu'on travaille sur cellules standard, le codage compact optimisé permet un gain moyen en surface de silicium d'environ 35% pour les machines de moins de 20 états (résultats obtenus sur 16 exemples), et d'environ 12% pour les machines de 20 à 80 états (résultats obtenus sur 14 exemples). Par contre, pour les machines de plus de 80 états, le codage 1 parmi n offre un gain moyen de 26% par rapport au codage compact optimisé (résultats obtenus sur 11 exemples, [Belh 93]). Toutefois, lorsque le nombre d'états est important, il faut noter que le codage 1 parmi n peut poser des problèmes de distribution d'horloge nécessitant un effort de conception important.

Dans la suite, les termes « codage classique » désigneront la procédure de codage compact optimisé d'ASYL brièvement décrite ci-dessus.

I.3. SÛRETÉ DE FONCTIONNEMENT

Nous avons vu dans ce qui précède les notions de contrôleurs et de synthèse de contrôleurs. Les contraintes d'optimisation classiquement retenues lors de la synthèse sont la surface et la vitesse du circuit synthétisé. Plus récemment sont apparues les préoccupations de consommation et de sûreté de fonctionnement. Le présent document portant sur l'introduction de la sûreté de fonctionnement comme contrainte d'optimisation lors de la synthèse des contrôleurs, cette partie présente les différents concepts de la sûreté de fonctionnement indispensables à la compréhension des chapitres suivants. Plus précisément, le paragraphe I.3.1. est essentiellement une présentation condensée des concepts de sûreté de fonctionnement exposés dans [Cour 91, Lapr 88], et le paragraphe I.3.2. présente un rapide état de l'art des outils de conception existant qui prennent en compte la sûreté de fonctionnement.

I.3.1. Concepts de base

I.3.1.1. Introduction [Cour 91, Lapr 88]

Comme déjà mentionné dans l'introduction, *la* **sûreté de fonctionnement** *d'un système informatique est la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre [Lapr 88].*

Dans notre cas le système informatique se limitera à un ASIC (« Application Specific Integrated Circuit » : circuit intégré pour application spécifique), et plus particulièrement à la partie contrôle dans cet ASIC. La sûreté de fonctionnement comporte différents attributs, à savoir la **disponibilité**, la **fiabilité**, la **sécurité-innocuité** et la **sécurité-confidentialité** [Cour 91]. La disponibilité se rapporte à la propriété du système d'être prêt à l'utilisation. La fiabilité se

rapporte à l'absence de discontinuité dans le service délivré. La sécurité-innocuité correspond à la propriété de non-occurrence de défaillances catastrophiques, c'est-à-dire de défaillances dangereuses pour le système informatique lui-même ou pour l'entité utilisatrice (cette entité peut-être une carte électronique, une personne humaine, une société, etc ...). Enfin la sécurité-confidentialité correspond à la sûreté de fonctionnement perçue du point de vue de la prévention d'accès ou de manipulations non-autorisées des informations.

Une **défaillance** est définie comme la non-conformité du service délivré au service spécifié, la spécification *étant une description agréée de la fonction ou du service attendu du système* [Cour 91]. Une **erreur** est *la partie de l'état du système qui est susceptible d'entraîner une défaillance. La cause adjugée ou supposée d'une erreur est une* **faute**. *Une erreur est donc la manifestation d'une faute dans le système, alors qu'une défaillance est l'effet d'une erreur sur le service* [Lapr 88].

Les méthodes permettant d'améliorer la sûreté de fonctionnement peuvent être classées en quatre catégories complémentaires : les méthodes de **prévention** des fautes (l'empêchement des fautes), les méthodes de **tolérance** aux fautes (l'empêchement des défaillances malgré les fautes), les méthodes d'**élimination** des fautes (réduction du nombre et de la sévérité des fautes), les méthodes de **prévision** des fautes (estimation de leur présence, de leur création et de leurs conséquences). La prévention et la tolérance permettent l'obtention de la sûreté de fonctionnement, tandis que l'élimination et la prévision permettent la validation de la sûreté de fonctionnement.

En résumé, fautes, erreurs et défaillances constituent les entraves à la sûreté de fonctionnement. La prévention, la tolérance, l'élimination et la prévision des fautes constituent les moyens pour la sûreté de fonctionnement. Enfin, la disponibilité, la fiabilité, la sécurité-innocuité et la sécurité-confidentialité constituent les attributs de la sûreté de fonctionnement. Mesurer la sûreté de fonctionnement se fait donc par l'évaluation de ses différents attributs.

Pour les attributs de la sûreté de fonctionnement, les travaux présentés dans ce document concernent la disponibilité, la fiabilité et la sécurité-innocuité uniquement. La sécurité-confidentialité ne sera donc plus considérée dans la suite. Pour ce qui est des moyens, nous nous sommes essentiellement intéressés à la tolérance des fautes, soit par l'intermédiaire de la détection d'erreurs qui vise plus particulièrement l'amélioration de la disponibilité et de la sécurité-innocuité, soit par l'intermédiaire du masquage d'erreurs.

I.3.1.2. Fautes, erreurs et défaillances [Lapr 88]

On distingue deux caractéristiques importantes pour les fautes, à savoir leur origine et leur persistance.

Lorsqu'on considère l'origine des fautes du point de vue de la cause phénoménologique, on est amené à distinguer les **fautes physiques** des **fautes humaines**. On distingue aussi les **fautes internes** des **fautes externes** lorsqu'on s'intéresse à l'origine des fautes du point de vue des frontières du système. Enfin, on distingue les **fautes de conception** des **fautes opérationnelles**, lorsqu'on oppose la phase de conception du système à sa phase d'exploitation.

La persistance conduit à distinguer les **fautes permanentes** des **fautes temporaires**, c'est-à-dire les fautes d'une durée limitée dans le temps. Les fautes temporaires externes, c'est-à-dire résultant d'interférences ou d'interactions du système avec son environnement, sont aussi appelées **fautes transitoires**. Les fautes temporaires internes sont appelées **fautes intermittentes**.

En ce qui concerne les erreurs, nous avons vu qu'une erreur est susceptible de conduire à une défaillance. Ceci dépend de deux facteurs principaux. Tout d'abord, une défaillance est définie du point de vue de l'utilisateur et dépend donc du taux d'erreurs acceptable consciemment ou inconsciemment défini par celui-ci. Ensuite, la structure, la composition du système peut empêcher qu'une erreur conduise à une défaillance. La redondance (matérielle, temporelle, ...) peut en effet empêcher la dégénérescence d'une erreur en défaillance. La redondance peut-être intentionnelle (pour tolérer les fautes) ou non intentionnelle. En pratique, sauf cas extrêmement rare, il existe toujours une redondance non intentionnelle et celle-ci peut avoir, dans certains cas, le même effet, mais inattendu, qu'une redondance intentionnelle.

En ce qui concerne les défaillances, il est à noter qu'un système ne défaille pas toujours de la même façon. On est amené à distinguer deux modes de défaillance, à savoir le **mode de défaillance contrôlée** lorsque le mode de défaillance a été spécifié (prévu dans la spécification du système) et par opposition, le **mode de défaillance incontrôlée**. De plus, on peut différencier les défaillances selon leur sévérité, à savoir les **défaillances bénignes** et les **défaillances catastrophiques**. Une défaillance est dite catastrophique lorsque le coût (humain, économique, ...) de ses conséquences est très supérieur aux bénéfices retirés de la délivrance d'un service approprié (c'est-à-dire conforme à la spécification).

Une faute est dite **active** lorsqu'elle produit une erreur, sinon elle est dite **dormante**. Une erreur est par nature temporaire et peut être **latente** ou **détectée**. On dit qu'une erreur peut propager [Lapr 88], et en propageant créer d'autres erreurs. Le but des techniques de masquage d'erreurs est d'empêcher cette propagation et ainsi d'éviter la défaillance du système, tandis que les techniques de détection d'erreurs vont, au contraire, forcer et canaliser cette propagation afin d'empêcher l'erreur de rester latente et de propager de manière incontrôlée ou non perçue.

Les travaux présentés dans ce document concernent essentiellement les fautes physiques, opérationnelles, internes ou externes et permanentes ou temporaires ... et évidemment les erreurs et défaillances qui peuvent en résulter.

Les causes pouvant conduire à des fautes internes ou externes sont nombreuses. Nous n'en donnerons ici que quelques exemples.

Un exemple de cause de faute interne permanente dans un circuit intégré est un défaut dans le silicium (impureté, défaut de structure, etc ...) qui peut conduire à un court-circuit. Un phénomène d'électromigration dans les couches métalliques peut conduire à un circuit ouvert. Des dérives de la tension de seuil peuvent aussi intervenir à la suite d'une polarisation de molécules dans l'oxyde de grille d'un transistor, entraînant un mauvais fonctionnement logique [ASTE 91].

L'exemple le plus connu de cause de fautes externes sont les radiations spatiales (vent solaire, éruptions solaires, rayonnement galactique et extragalactique, etc ...) qui provoquent un bombardement incessant des dispositifs spatiaux en électrons, protons, ions lourds, etc ... Ces bombardements peuvent provoquer des aléas logiques (SEU : « Single Event Upset ») à savoir le basculement d'un bistable. Ce phénomène n'est pas destructif et correspond en fait à une faute transitoire. Il n'est pas rare puisque le nombre moyen de SEU recensé peut aller de 10 SEU par an dans les composants mémoire du satellite TIROS N jusqu'à 10 SEU/jour pour UOSAT 2. Ces chiffres sont montés jusqu'à 70 SEU/jour pour ETS 5 durant l'éruption solaire d'octobre 1989 [Bour 91].

Les fautes pouvant revêtir un nombre de formes très important selon l'événement qui en est à l'origine, différents modèles ont été définis afin d'en faciliter le traitement. Nous venons de voir

les SEU (par nature transitoires), mais il existe aussi les collages à 0 (« stuck-at 0 ») qui correspondent à un court-circuit d'une équipotentielle et de la masse, les collages à 1 (« stuck-at 1 »), les courts-circuits entre deux équipotentielles (« bridge »), etc ... qui sont des modèles de fautes opérationnelles (physiques) internes ou externes, permanentes ou temporaires.

I.3.1.3. Tolérance aux fautes [Lapr 88]

La tolérance aux fautes peut être mise en oeuvre soit par le **traitement d'erreurs** soit par le **traitement de fautes.**

Le traitement d'erreurs peut prendre deux formes : soit le **recouvrement d'erreur**, qui correspond à la substitution, après coup, d'un état exempt d'erreur à un état erroné, soit la **compensation d'erreur**, si l'état erroné comprend suffisamment de redondance pour permettre de délivrer un service approprié malgré l'erreur.

Le traitement de fautes s'effectue en deux étapes : tout d'abord le **diagnostic** de faute qui cherche à identifier la cause de l'erreur, puis la **passivation** de la faute incriminée, c'est-à-dire le traitement qui consiste à empêcher toute activation ultérieure de la faute en retirant les composants fautifs du processus d'exécution.

Les travaux décrits dans ce document correspondent à des techniques de traitement d'erreurs, soit par recouvrement pour les techniques originales de **détection d'erreurs** décrites au Chapitre III, soit par compensation pour les nouvelles techniques de **masquage d'erreurs** dans les contrôleurs décrites au Chapitre IV.

Des techniques classiques de détection d'erreurs utilisent par exemple des **codes détecteurs d'erreurs** (souvent utilisés pour les mémoires) comme le code de parité, ou encore la duplication/comparaison. La **duplication** consiste à implanter un même élément en deux exemplaires identiques et à en comparer les sorties à chaque cycle fonctionnel.

Des techniques classiques de masquage d'erreurs utilisent par exemple des **codes correcteurs d'erreurs** ou encore la **redondance modulaire triple** (TMR : « Triple Modular Redundancy »). La redondance modulaire triple correspond à l'implantation en trois exemplaires identiques (du point de vue fonctionnel) d'un élément, et au vote majoritaire sur leurs sorties.

Le paragraphe suivant présente succinctement différents outils de conception utilisant ces techniques simples de tolérance des fautes pour l'amélioration de la fiabilité du composant ou du système synthétisé.

I.3.2. Outils de conception prenant en compte la sûreté de fonctionnement

Quelques outils d'automatisation et d'aide à la conception ont été développés pour l'amélioration de la sûreté de fonctionnement. Ces outils concernent les systèmes au niveau carte, ou au niveau circuit, ou encore au niveau des éléments de base des circuits.

I.3.2.1. Au niveau carte

Lorsqu'on considère le système au niveau carte, on peut citer deux outils développés à l'Université Carnegie Mellon.

Le premier, ASSURE ([Edmo 90]), est un outil d'analyse et de synthèse pour la sûreté de fonctionnement. Beaucoup d'outils pour l'analyse de la sûreté de fonctionnement des systèmes niveau carte existaient avant lui (Nasa [Stif 79], Duke University [Gies 83], Carnegie Mellon University [Elki 83a, Elki 83b, Clun 87], ...), mais ASSURE est certainement le premier à autoriser la synthèse automatique des architectures préconisées. ASSURE est en fait un module ajouté à l'outil de conception MICON [Birm 88]. Le module de synthèse de MICON, appelé M1, accepte en entrée un ensemble de spécifications du système à synthétiser (nom du processeur, quantité de mémoire, vitesse, ...) ainsi qu'une liste de contraintes (surface de la carte, consommation, ...). Il fournit une description de la carte résultante. ASSURE, lui, est un module d'analyse pour la sûreté de fonctionnement qui étudie le système synthétisé par le module M1, et compare ses caractéristiques de sûreté de fonctionnement (principalement des valeurs relatives à la fiabilité), aux caractéristiques exigées par le concepteur (niveau de fiabilité spécifié par différentes mesures dans un fichier utilisé en entrée par le module ASSURE). Si les contraintes de sûreté de fonctionnement imposées par le concepteur ne sont pas respectées, ASSURE va proposer une nouvelle architecture de carte et utiliser le module M1 pour la synthèse du système modifié. ASSURE va procéder ainsi, par étapes, jusqu'à l'obtention d'un système répondant aux contraintes de sûreté de fonctionnement, où jusqu'à ce que toutes les techniques d'amélioration de la sûreté de fonctionnement connues par ASSURE aient été épuisées (échec). Au cours de ce processus itératif incrémental, ASSURE commence par appliquer, les unes après les autres, toutes les techniques d'évitement des fautes à sa disposition (changement de boîtier, changement de niveau de qualité de fabrication, ...) puis ensuite, si les contraintes ne sont toujours pas satisfaites, les techniques de tolérance des fautes (triplement et vote majoritaire, utilisation de codes correcteurs d'erreurs pour les mémoires, ...).

Le second outil développé, SIDECAR ([Youn 91]), est une amélioration de l'outil précédent. Différents points gênants ont, en effet, été mis en évidence dans le fonctionnement d'ASSURE. Les principaux sont les suivants :

- Le premier inconvénient est que l'outil ASSURE ne prend en compte qu'un seul paramètre à la fois lors de son processus d'optimisation (sûreté de fonctionnement ou surface ou consommation ...). En d'autres termes, il va essayer d'améliorer la sûreté de fonctionnement, par exemple, sans tenir compte du coût des techniques mises en oeuvre.
- Ensuite, l'évaluation de l'amélioration apportée ne peut se faire qu'après resynthèse du système par le module M1 de MICON. Ceci signifie que l'espace des solutions explorées est réduit du fait du temps de synthèse nécessaire à l'évaluation de chaque solution.
- Enfin, en fonctionnant de manière itérative, par amélioration successive de la fiabilité, ASSURE ne donne pas toujours la solution la moins coûteuse respectant les contraintes de sûreté de fonctionnement. Ce problème est aggravé par le fait qu'ASSURE applique d'abord toutes les techniques d'évitement avant d'envisager les techniques de tolérance des fautes et qu'il ne revient pas sur les décisions prises.

SIDECAR procède donc comme suit. Le concepteur débute par la synthèse du système spécifié. Ensuite SIDECAR explore les solutions d'évitement et de tolérance et fournit au concepteur une liste de possibilités avec le gain attendu (dont la métrique est fournie par l'utilisateur) et le coût, la responsabilité de la décision restant à la charge du concepteur. Une fois les modifications à apporter choisies, la synthèse du nouveau système est effectuée. Ceci permet un parcours plus efficace de l'ensemble des solutions, en prenant en compte les problèmes de coût, consommation, surface, etc ... et non plus seulement l'amélioration à « tout prix » de la fiabilité.

I.3.2.2. Au niveau circuit

Nous venons de voir deux exemples d'outils de synthèse de systèmes niveau carte. Lorsqu'on considère le système au niveau circuit, de nombreuses propositions ont été faites pour l'amélioration de la sûreté de fonctionnement lors de la synthèse de haut niveau.

Ainsi, [Karr 92a] propose une méthode de réduction du coût matériel de la redondance modulaire au cours de la synthèse de haut niveau. Cette méthode a été intégrée à un ensemble de programmes de synthèse comportementale fonctionnant sur SUN 3/50. L'idée est la suivante. Soit G un graphe de flot de données décrivant l'ordonnancement de diverses opérations. Une solution possible pour tolérer les fautes dans ce flot de données est de l'implanter en trois exemplaires identiques et de faire un vote majoritaire sur les résultats fournis. Ceci correspond à un masquage d'erreur par la technique classique du TMR. Toutefois, il est possible de profiter du degré de liberté autorisé dans certains graphes au niveau de l'ordre des opérations réalisées. Cette liberté est liée aux dépendances existant, ou plutôt n'existant pas, entre les opérations effectuées dans le graphe de flot de données, et aux propriétés de distributivité, associativité, et commutativité. Ainsi, en réorganisant les opérations des trois exemplaires du flot de données, il est possible de les implanter avec moins de matériel (deux additionneurs au lieu de trois par exemple). Cependant, cette technique est intéressante du point de vue des fautes transitoires, mais ne garantit pas totalement le masquage des erreurs consécutives aux fautes permanentes, ou consécutives à des fautes transitoires se prolongeant sur plusieurs pas de contrôle, du fait du partage de matériel entre les trois flots de données.

[Ragh 91, Karr 92b, Karr 93, Blou 95] proposent, eux, des outils de synthèse de haut-niveau avec insertion de points de recouvrement pour l'obtention de la tolérance des fautes après détection d'erreurs. [Ragh 91] n'adresse que le problème de l'insertion de points de recouvrement dans le GFD, et la génération du chemin de données et du microprogramme correspondant, sans tenir compte du problème de la détection d'erreurs déclenchant le processus de recouvrement. Avec leur outil, l'utilisateur peut fixer trois contraintes à la procédure d'insertion des points de recouvrement : le nombre de registres maximum autorisé, le temps maximum de recouvrement autorisé et le nombre d'essais de recouvrement maximum autorisé. De plus, l'algorithme d'ordonnancement des opérations prend en compte, tout au long du processus d'ordonnancement, le facteur de coût des opérations non encore ordonnancées. La fonction de priorité définissant l'ordre dans lequel les opérations vont être considérées lors de l'ordonnancement peut être facilement redéfinie.

[Karr 92b] et plus récemment [Karr 93] précisent, eux, le mode de détection des erreurs qui se fait à travers la duplication et la comparaison. Outre la duplication, la différence principale entre l'outil décrit dans [Ragh 91] et celui décrit dans [Karri 92b] est que le premier effectue l'ordonnancement et l'insertion des points de recouvrement en deux étapes distinctes, tandis que le second le fait conjointement. [Karri 92b] montre que cette deuxième solution permet une meilleure optimisation du coût matériel et temporel de l'insertion des points de recouvrement.

[Karri 93] reprend les travaux décrits dans [Hwan 91] et décrit une formulation mathématique du problème de l'ordonnancement, de l'allocation des unités fonctionnelles et des registres avec prise en compte du choix des points de recouvrement et de la duplication pour la détection et le recouvrement d'erreurs. L'intérêt de cette formalisation est de permettre une optimisation globale des différentes étapes de synthèse. Les algorithmes correspondants ont étés implantés et constituent un outil plus complet que [Ragh 91] et [Karr 92b] en prenant en particulier en compte le coût de la duplication/comparaison (évaluation du nombre de comparateurs à implanter). Enfin, dernière remarque, l'outil décrit par [Ragh 91] tente de minimiser le temps d'exécution en fonction du coût matériel maximum autorisé par le concepteur. Dans [Karr 92b],

l'outil tente au contraire de minimiser le coût matériel en fonction du temps d'exécution maximum autorisé par le concepteur. Ceci indique l'ordre de priorité accordé à l'optimisation du coût matériel et du coût temporel lors de l'insertion des points de recouvrement. Dans [Karri 93], les deux approches ont été formalisées, avec en plus l'élaboration d'algorithmes pour l'ordonnancement et l'insertion séparés de points de recouvrement (comme dans [Ragh 91]) et pour l'ordonnancement et l'insertion conjoints de points de recouvrement (comme dans [Karr 92b]). Comme dans [Karr 92b], les résultats tendent à montrer que l'insertion conjointe à l'ordonnancement donne de meilleurs résultats.

Enfin, [Blou 95] montre que les algorithmes proposés dans [Ragh 91] et [Karr 92b] ne sont pas optimaux et propose deux nouvelles approches, une pour la minimisation du coût matériel, et une pour la minimisation du coût temporel. Ces deux approches sont basées sur la recherche d'un sous graphe particulier dans un graphe dit de coût des points de recouvrement, qui modélise le coût du placement d'un point de recouvrement à la fin de tel ou tel pas de contrôle. Les résultats publiés dans [Blou 95] montrent la plus grande efficacité de ces nouveaux algorithmes par rapport à ceux décrits dans [Ragh 91] et [Karr 92b]. Toutefois, ces algorithmes effectuent l'insertion des points de recouvrement sur un graphe de flot de données déjà ordonnancé, et les auteurs font remarquer, après analyse des résultats, qu'ils pourraient encore gagner en efficacité s'ils étaient combinés à l'étape d'ordonnancement. Ceci est fait dans [Ohm 96].

Une approche légèrement différente est exposée dans [Nara 95]. Dans les cinq articles précédemment cités, le choix des points de recouvrement est effectué avec, entre autres, pour but la minimisation du coût temporel en l'absence de fautes. Dans [Nara 95], au contraire, le but recherché par les algorithmes mis en place est la minimisation du coût temporel du recouvrement en présence d'une faute transitoire. En fait, dans l'état actuel de leurs recherches, les auteurs cherchent à placer le minimum de points de recouvrement possible pour un temps maximum de recouvrement spécifié. Lorsque le temps de recouvrement maximum spécifié est trop faible, l'outil indique qu'il n'existe pas de solutions.

Une autre solution étudiée pour l'amélioration de la sûreté de fonctionnement des circuits synthétisés à partir d'une description de haut niveau est la détection d'erreurs par test continu. Ainsi, [Grim 94] décrit un outil de synthèse de haut-niveau, appelé GAUT-AT, permettant de générer des chemins de données avec dispositifs de détection d'erreurs. Cet outil est dédié aux applications de traitement du signal. L'obtention d'opérateurs avec test continu est basée sur l'utilisation du code double-rail, du code de Berger, du code de parité ou encore du code à résidu. Le codage des données et l'adjonction des éléments nécessaires à l'autotestabilité se fait après ordonnancement et instanciation du graphe de flot de données. Le choix du code utilisé pour telle ou telle zone fonctionnelle du chemin de donnée est effectué en fonction des contraintes en surface ou du taux de détection d'erreurs défini par le concepteur. Cet outil ne traite pas le problème d'insertion de points de recouvrement, mais propose par contre des alternatives à la classique duplication avec une analyse précise de l'impact de telle ou telle méthode de détection d'erreurs sur la sûreté de fonctionnement et son coût matériel au niveau du chemin de données.

Dans le même esprit, mais à un plus bas niveau, [Hamd 94] propose lui aussi un outil de génération automatique de parties opératives à test continue. Les deux codes retenus pour les unités arithmétiques et les unités arithmétiques et logiques sont le code double rail et la prédiction de parité. L'originalité de cet outil par rapport au précédent tient essentiellement dans le fait suivant. Lors de la synthèse d'un chemin de données avec GAUT-AT, l'implantation est effectuée en utilisant les éléments classiques de la bibliothèque de cellules standard spécifiée. Avec l'outil décrit dans [Hamd 94], la bibliothèque de cellules standard utilisée a été modifiée : des cellules spécifiques à la détection d'erreurs ont été définies, ajoutées à la bibliothèque, et sont utilisées lors de l'implantation du chemin de données, ce qui permet un gain en surface non négligeable, en

particulier pour les implantations à base de prédiction de parité (gain supérieur à 10%). Outre l'outil implanté, l'intérêt de cette étude est de montrer que la prise en compte des problèmes de détection d'erreurs par les fondeurs, qui définissent les bibliothèques de cellules standard, est non-seulement possible, mais qui plus est souhaitable.

I.3.2.3. Au niveau des éléments de base des circuits

Outre les outils de synthèse de systèmes au niveau carte ou au niveau circuit, l'amélioration de la sûreté de fonctionnement a aussi été envisagée au niveau des éléments de base, et en particulier des PLAs (« Programmable Logique Array » : matrice de logique programmable).

Ainsi, [Tork 91] décrit un outil appelé PROTECT, de plus bas niveau que les outils précédant, permettant la génération automatique de PLAs auto-contrôlables. Pour cela, l'outil code les sorties du PLA en utilisant soit le code de Berger, soit un code m parmi n, soit un code double-rail. La procédure de codage des sorties avec le code de Berger est assez complexe et ne donne pas forcément un résultat. Toutefois [Tork 91] précise que les cas d'échec sont particulièrement rares, et que le code de Berger donne souvent le meilleur résultat.

I.3.2.4. Conclusion

Nous venons de voir de manière plus ou moins détaillée différents outils visant à l'amélioration de la sûreté de fonctionnement au niveau carte, au niveau circuit ou encore au niveau des éléments de base des circuits. On peut encore citer des outils moins en rapport avec le sujet de ce document, comme par exemple celui décrit dans [Papa 91] destiné à la génération, au niveau RTL, de chemins de données avec dispositifs de test hors ligne (« Built In Self Test » : à autotest intégré), à partir d'une description algorithmique.

Chacun de ces outils offre un intérêt certain à son propre niveau. Toutefois, aucun de ces outils ne cible la détection ou le masquage des erreurs de séquencement, c'est-à-dire le passage erroné d'un contrôleur d'un état à un autre, directement au niveau du circuit. La détection ou le masquage de telles erreurs, grâce aux outils cités ci-dessus, ne peut intervenir que par « un heureux effet de bord » des techniques d'amélioration de la sûreté de fonctionnement utilisées. Du fait de la difficulté à gérer les erreurs de séquencement par des dispositifs externes aux circuits, il est indispensable d'en tenir compte directement au niveau des contrôleurs, et si possible avec des dispositifs de tolérance des fautes au coût réduit. C'est tout le sujet de l'outil ASYL-SdF que nous avons développé.

I.4. OBJECTIFS ET VUE D'ENSEMBLE DU TRAVAIL RÉALISÉ

Comme nous venons de le voir, il n'existait pas jusqu'à récemment d'outils de synthèse de contrôleurs intégrant la notion de sûreté de fonctionnement. Plus généralement, aucun des outils cités au paragraphe I.3. ne permet la synthèse de circuits séquentiels intégrant à la fois dans la partie contrôle et la partie opérative un ensemble cohérent de dispositifs pour l'amélioration de la sûreté de fonctionnement, à un coût moindre que les classiques duplication et triplement. Nous nous sommes donc attachés ces dernières années à développer un outil de synthèse, ASYL-SdF (Aide à la SYnthèse Logique pour la Sûreté de Fonctionnement), dont l'objectif général est de mettre à la disposition des concepteurs des flots de synthèse proposant des solutions originales à la détection et au masquage d'erreurs (ces solutions devant éviter les redondances inutiles entre la PC et la PO). La sûreté de fonctionnement des contrôleurs étant certainement la moins bien représentée au niveau des outils de conception existants, et à nos yeux la plus prioritaire, nous avons

commencé par développer différents flots de synthèse des contrôleurs intégrant les préoccupations de sûreté de fonctionnement.

A l'heure actuelle, l'outil ASYL-SdF, qui est un élément du système ASYL commercialisé par la société IST, est dédié à la synthèse de circuits séquentiels décrits en VHDL, pour une implantation sur cellules standard ou équivalent. Durant les premières étapes de la synthèse, une partie contrôle et un chemin de données sont extraits de la description RTL du circuit. Ensuite, la partie contrôle peut être implantée avec des dispositifs de détection ou de masquage d'erreurs, en ciblant plus particulièrement les erreurs de séquencement. L'implantation de dispositifs équivalents au niveau du chemin de données n'est pas encore disponible mais est en cours de développement. De même les techniques de tolérance des fautes élaborées pour les contrôleurs implantés sur cellules standard sont en train d'être adaptées aux contrôleurs sur ROM. La figure I.9 donne une vue d'ensemble des possibilités de l'outil actuel.

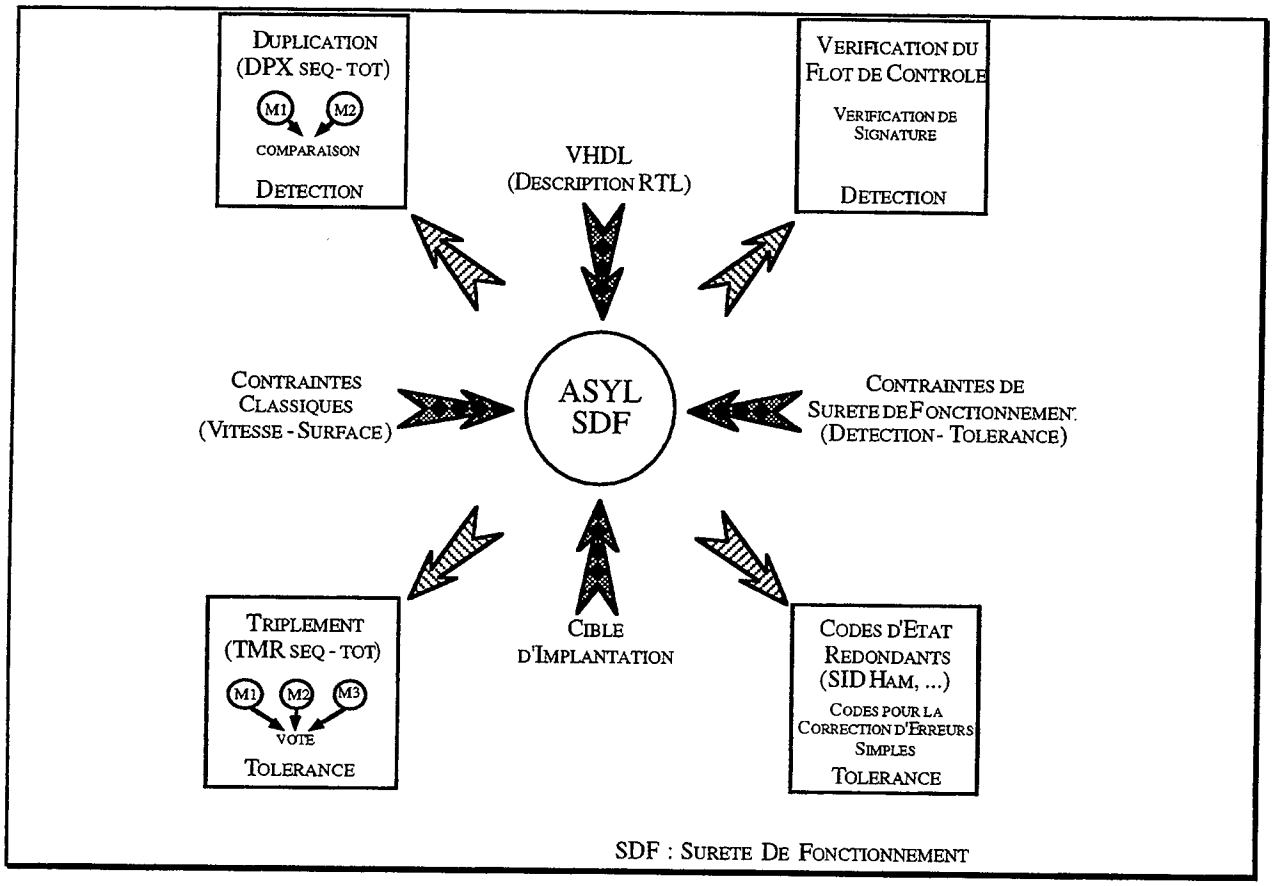

Figure I.9 - L'outil ASYL-SdF.

Sept architectures pour la tolérance des fautes dans les machines d'états finis sont disponibles. Quatre sont dédiées à la détection d'erreurs et trois au masquage d'erreurs.

Sur les quatre architectures dédiées à la détection, deux utilisent la duplication/comparaison. Elles sont appelées DPX Tot et DPX Seq (DPX pour DuPleX, Tot pour Total et Seq pour Séquencement). DPX Tot correspond à une duplication complète de la machine d'états finis, la comparaison se faisant sur les sorties primaires. DPX Seq correspond à la duplication de la logique de séquencement (logique d'état suivant plus registre d'état) uniquement, la comparaison se faisant sur les sorties des registres d'état (Fig. I.10). Les deux autres architectures sont appelées CFC Ajs et CFC NoAjs. CFC signifie « Control Flow Checking ». Ainsi, ces deux architectures

sont basées sur la vérification du flot de contrôle, et plus précisément sur la vérification du flot de contrôle par analyse de signature. L'une, CFC Ajs, utilise une technique d'ajustement explicite. L'autre, CFC NoAjs, est basée sur l'utilisation d'ajustements implicites de la signature du flot de contrôle. Ces deux architectures font l'objet du chapitre III.

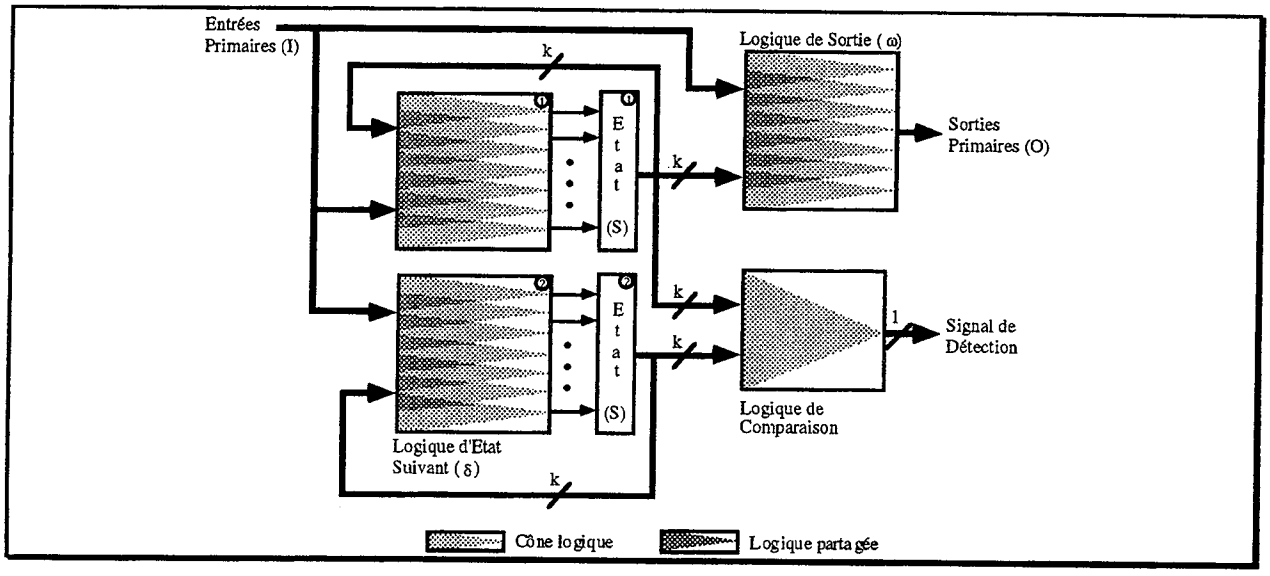

Figure I.10 - Architecture DPX Seq.

Sur les trois architectures dédiées au masquage d'erreurs, deux utilisent le triplement et le vote majoritaire. Elles sont appelées TMR Tot et TMR Seq. Comme son nom l'indique, l'architecture TMR Tot correspond au triplement de tout le contrôleur, le vote se faisant sur les sorties primaires. TMR Seq correspond au triplement de la logique de séquencement uniquement, le vote se faisant sur les sorties des registres d'état (Fig. I.11). Enfin, la dernière architecture, appelée architecture SID (« Single Independent Decoder » : décodeur indépendant unique), est basée sur l'utilisation de codes correcteurs d'erreurs durant la phase de codage des états du contrôleur. Cette architecture fait l'objet du chapitre IV.

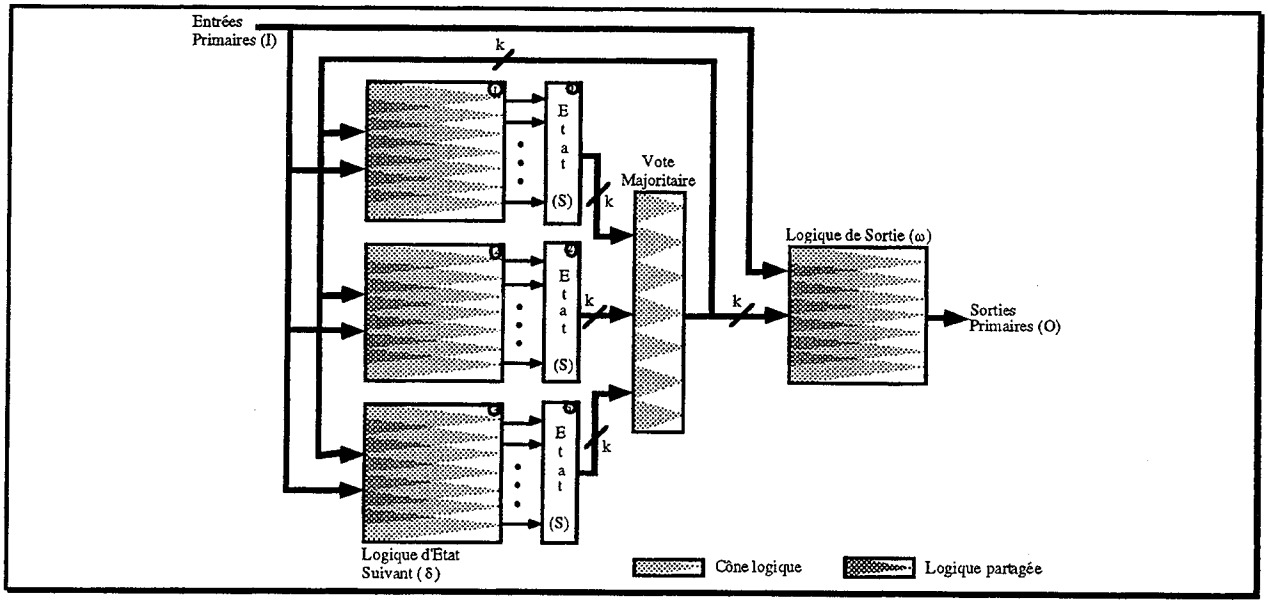

Figure I.11 - Architecture TMR Seq.

Il va de soit que le flot de synthèse de l'implantation simplex décrit au paragraphe I.2.2 est légèrement modifié pour l'implantation des architectures DPX et TMR. Pour les implantations DPX Tot et TMR Tot, les modifications sont mineures, puisqu'il suffit de générer normalement une machine simplex, puis de l'instancier deux ou trois fois (selon l'architecture), de générer le comparateur ou le voteur, et d'interconnecter les trois ou quatre blocs obtenus (un traitement post-interconnexion permet de régler les problèmes d'amplification). Pour les architectures DPX Seq et TMR Seq, le flot de synthèse est plus sensiblement différent puisque, après le codage des états et la génération des équations, les fonctions de calcul de l'état suivant sont différenciées des fonctions de calcul des sorties et subissent des traitements séparés. Ceci permet la génération d'un bloc de séquencement qui sera doublé ou triplé, et d'un bloc de sortie qui n'est, lui, instancié qu'une seule fois.

Les architectures CFC et SID permettent d'implanter une redondance plus fine et mieux ciblée, et ainsi d'obtenir des coûts en surface souvent moindres que celui des architectures DPX et TMR. Ceci se fait au prix d'un flot de synthèse beaucoup plus complexe, qui explique que ce type d'architectures, bien que très souvent plus intéressantes que la duplication ou le triplement, n'aient pour ainsi dire jamais été utilisées lors d'implantations manuelles. C'est le principal intérêt des outils de synthèse automatique pour la sûreté de fonctionnement, à savoir de proposer aux concepteurs de nouveaux compromis surface/vitesse/sûreté inenvisageables lors d'implantations manuelles. Dans les chapitres suivant, nous allons donc étudier les flots de synthèse des architectures CFC (chapitre III) et SID (chapitre IV) avec bien sûr des résultats concrets sur des exemples internationaux et industriels de contrôleurs. Mais avant de voir nos propres propositions en matière de sûreté de fonctionnement des contrôleurs, le chapitre II présente un état de l'art, non exhaustif, des approches ayant été proposées jusqu'ici.

CHAPITRE II - DÉTECTION ET TOLÉRANCE DANS LES CONTRÔLEURS

La littérature scientifique est assez prolixe en ce qui concerne les méthodes de détection et de tolérance. On peut en trouver un résumé assez général, mais toutefois très synthétique, dans [Peer 93]. Afin de conserver une taille raisonnable, le présent chapitre se limite à la présentation d'un état de l'art non exhaustif, qui donne une vue d'ensemble des méthodes proposées pour la détection d'erreurs et la tolérance aux fautes dans les contrôleurs. Il introduit, en particulier, les notions de Vérification d'un Flot de Contrôle (VFC) par analyse de signature et de masquage d'erreurs à base de codes correcteurs d'erreurs, notions qui sont utilisées aux chapitres III et IV.

II.1. MÉTHODES DE DÉTECTION

Dans un soucis de clarté, ces méthodes sont regroupées selon trois points : la détection d'erreurs par codage (paragraphe II.1.2), la vérification d'un flot de contrôle par codage (paragraphe II.1.3) et la vérification d'un flot de contrôle par analyse de signature (II.1.4). Il s'agit de méthodes de test en ligne, à opposer au test hors ligne en ce sens que le processus de vérification est concurrent au fonctionnement normal du dispositif sous test. Le paragraphe II.1.1 précise les notions de détection d'erreurs, de vérification d'un flot de contrôle, et de contrôleur microprogrammé utilisées dans cette section. Nous ne détaillerons pas les approches « fail-safe » (sûres vis à vis des défaillances), ou les méthodes de détection dans la partie opérative (respectivement [Hala 79], [Seng 77], [Nico 89], ..., et [Russ 91], [Orto 92], [Saxe 95], ...).

II.1.1. Définitions

II.1.1.1. Contrôleurs microprogrammés

Bien que les contrôleurs microprogrammés ne soient pas le sujet premier de ce document, l'importance des méthodes de détection d'erreurs mises au point pour ce type d'implantation exige quelques explications. Le présent paragraphe donne donc quelques informations informelles quant

au mode de fonctionnement des contrôleurs microprogrammés, pour une meilleure compréhension des analyses développées dans les paragraphes suivants.

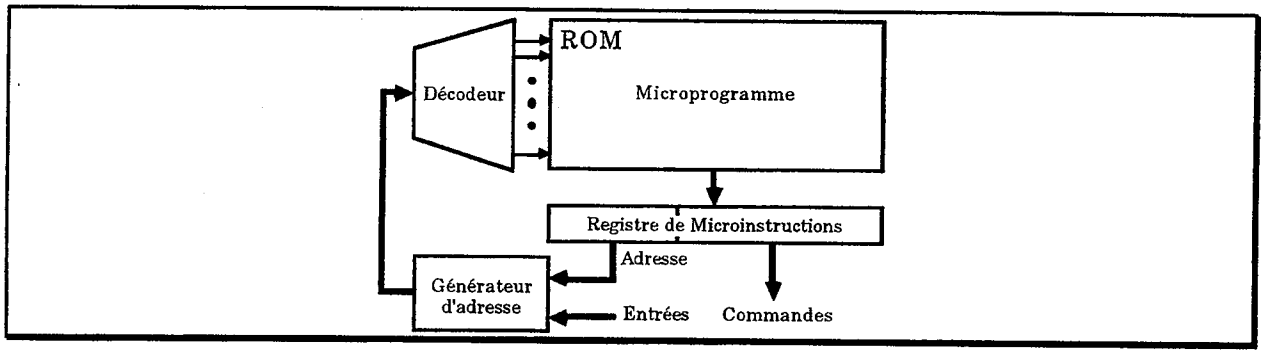

Figure II.1 - Architecture de base des contrôleurs microprogrammés.

Dans sa forme la plus simple, un contrôleur microprogrammé comprend un bloc permettant de stocker le microprogramme (généralement une ROM - « Read Only Memory » : mémoire à lecture seule), un registre de microinstructions où est stockée la microinstruction courante, et un générateur d'adresses permettant de générer, à partir de la microinstruction courante et des entrées, l'adresse de la microinstruction suivante. Un décodeur d'adresse en entrée de la ROM permet d'activer l'unique ligne de la ROM qui correspond à l'adresse décodée (chaque ligne contient une microinstruction). Une microinstruction est composée de deux champs, un champ pour l'adresse de la microinstruction suivante (qui peut être un champ codé en cas d'éclatement ou de branchement), un champ pour les sorties, que l'on peut aussi appeler micro-code opératoire ou encore champ de commandes. Bien sûr, en pratique, ce schéma est généralement beaucoup plus complexe, ne serait-ce que pour prendre en compte les appels à des sous-programmes avec empilement du contexte. Le nombre de champs de la microinstruction peut aussi être plus important. La figure II.1 montre l'architecture de base des contrôleurs microprogrammés.

II.1.1.2. Détection d'erreurs

Les méthodes de détection d'erreurs, dans l'acception considérée au paragraphe II.1.2, consistent à détecter toute erreur résultant de l'occurrence d'une faute dans la logique d'état suivant, la logique de séquencement (logique d'état suivant plus registre d'état), ou/et la logique de sortie. Il s'agit principalement de l'approche « auto-contrôlable » (« self-checking »). Les fautes prises en compte lors de l'étude et de la réalisation du dispositif de détection sont souvent élément d'un ensemble restreint de fautes prédéfinies. Cibler un ensemble de fautes particulier permet de définir, par simulation, l'ensemble des erreurs à détecter, et donc d'obtenir des taux de couverture, pour l'ensemble de fautes considéré, généralement remarquables. Toutefois, l'efficacité de ces méthodes pour un autre ensemble de fautes que celui spécifié reste inconnue.

Les techniques décrites dans les paragraphes suivants et entrant dans cette catégorie n'intègrent pas la notion de flot de contrôle proprement dite. La différenciation avec les méthodes de VFC est donc le plus souvent liée à la démarche de l'auteur de la méthode, la plupart des méthodes de détection ayant pour effet de bord la vérification du flot de contrôle à un degré plus ou moins important.

II.1.1.3. Vérification d'un flot de contrôle

La vérification d'un flot de contrôle recouvre un concept à la fois plus large et plus restreint que celui décrit au paragraphe précédent ; plus large parce que beaucoup de méthodes de

vérification d'un flot de contrôle ne se limitent pas à un ensemble de fautes prédéfinies (collage ou autre) ; plus restreint parce que le but n'est pas de détecter les erreurs résultant de toutes les fautes pouvant se produire, mais seulement de celles conduisant à un flot de contrôle différent de celui spécifié par le concepteur. La démarche s'appuie généralement non pas sur un modèle de fautes, mais sur un modèle d'erreurs : quelles sont les erreurs que l'on considère graves, indépendamment de leur cause mais plutôt du point de vue de leurs conséquences possibles, et comment les détecter avant qu'elles ne propagent.

Un flot de contrôle peut être représenté, indépendamment de la nature de l'élément contrôlé, par un graphe orienté G = {N, A}, appelé Graphe de Flot de Contrôle (GFC). N est l'ensemble des noeuds du graphe et A est l'ensemble des arcs représentant les transitions autorisées entre les noeuds de N.

A chaque noeud sont associés une information permettant de l'identifier de façon unique dans le GFC, et une information indiquant quelle opération doit être effectuée sur ce noeud (éventuellement aucune).

A chaque arc sont associés un noeud d'origine No, un noeud destination Nd, un prédicat de transition (qui détermine à quelle condition la transition est autorisée), et une information indiquant quelle opération doit être effectuée lors de la transition (éventuellement aucune).

Un GFC définit en fait différentes relations mathématiques entre les noeuds, les prédicats, et les opérations du flot à contrôler. En particulier, l'ensemble des arcs permet de définir la relation δ de succession entre noeuds, telle que δ (No, P) = Nd si et seulement si il existe un arc de A de prédicat P ayant No comme noeud origine et Nd comme noeud destination.

Plusieurs définitions sont rattachées à la notion de GFC.

- **Définition II.1 : Divergence**

*Une **divergence** est un ensemble d'arcs de A de même origine mais de destinations différentes.*

- **Définition II.2 : Convergence**

*Une **convergence** est un ensemble d'arcs de A de même destination mais d'origines différentes.*

- **Définition II.3 : Point de jonction**

*Un **point de jonction** est un noeud destination d'une convergence.*

- **Définition II.4 : Bloc linéaire**

*Un **bloc linéaire** est un sous graphe orienté du GFC possédant les caractéristiques suivantes :*

- *le seul point d'entrée est le premier noeud du bloc,*
- *le seul point de sortie est le dernier noeud du bloc,*
- *aucune divergence et aucun point de jonction n'existe à l'intérieur du bloc.*

Les prédicats associés aux arcs d'une divergence sont supposés tous différents et non intersectants (pas de parallélisme dans le graphe). Si le graphe est entièrement spécifié, ils forment de plus une tautologie. La figure II.2c illustre ces différentes notions.

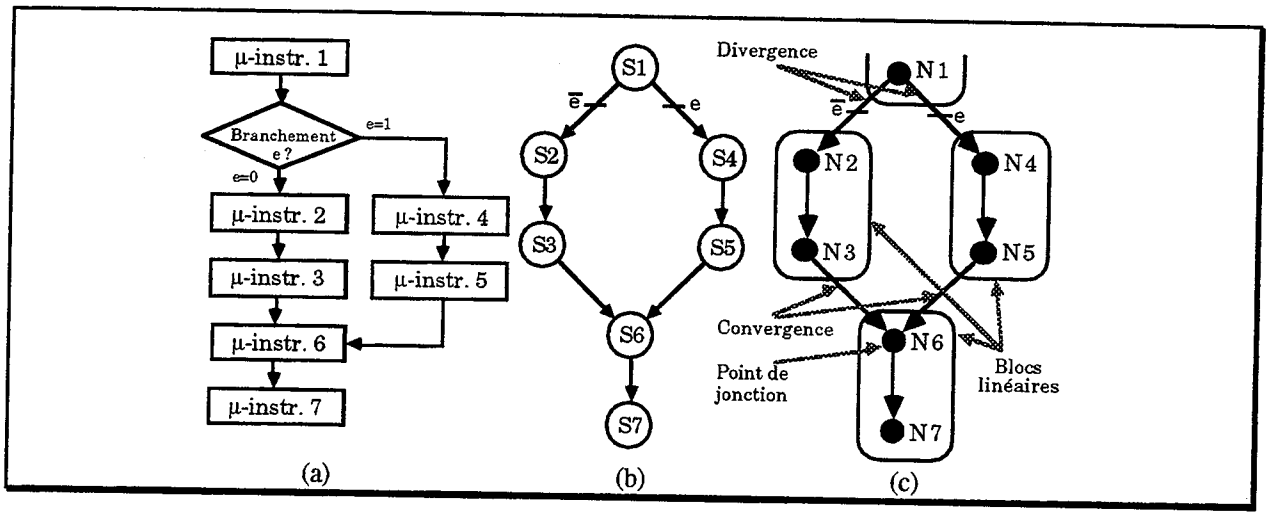

Figure II.2 - Exemple d'organigramme de contrôleur microprogrammé (a), de graphe de contrôle (b), et de GFC associé (c).

La vérification d'un flot de contrôle fait souvent intervenir la notion de chemin (au sens classique de la théorie des graphes).

- **Définition II.5 : Correction d'un chemin**

Soit un chemin C = {Cn, Ca} où Cn représente l'ensemble des noeuds du chemin et Ca l'ensemble des transitions entre ces noeuds. C est dit correct vis à vis d'un GFC G = {N, A} ssi $Cn \subseteq N$ *et* $\forall i \in Ca$, *d'origine* No_i, *de destination* Nd_i, *et de prédicat* P_i, *on a* $\delta(No_i, P_i) = Nd_i$, *où* δ *est la relation de succession définie par G. Dans le cas contraire, le chemin est dit incorrect.*

- **Définition II.6 : Légalité d'un chemin**

Soit un chemin C = {Cn, Ca} où Cn représente l'ensemble des noeuds du chemin et Ca l'ensemble des transitions entre ces noeuds. C est dit légal vis à vis d'un GFC G = {N, A} si et seulement si :

- $Cn \subseteq N$,
- $\forall i \in Ca$, $\exists P_j$, *un prédicat, tel que* $(P_j = P_i$ *ou* $P_j \neq P_i)$ *et* $\delta(No_i, P_j) = Nd_i$, *où* δ *est la relation de succession définie par G, et i a pour origine* No_i, *destination* Nd_i, *et prédicat* P_i.

Dans le cas contraire, le chemin est dit illégal.

Il découle des définitions précédentes que tout chemin correct est légal, mais qu'un chemin légal peut être incorrect, c'est-à-dire qu'il existe au moins un arc du chemin tel qu'aucun arc du GFC, de même origine et de même destination, n'a le même prédicat (il existe forcément au moins un arc de même origine et de même destination puisque le chemin est légal vis à vis du GFC). La figure II.3 montre un exemple de chemin illégal, de chemin légal incorrect et de chemin correct vis à vis du GFC décrit par la figure II.2.

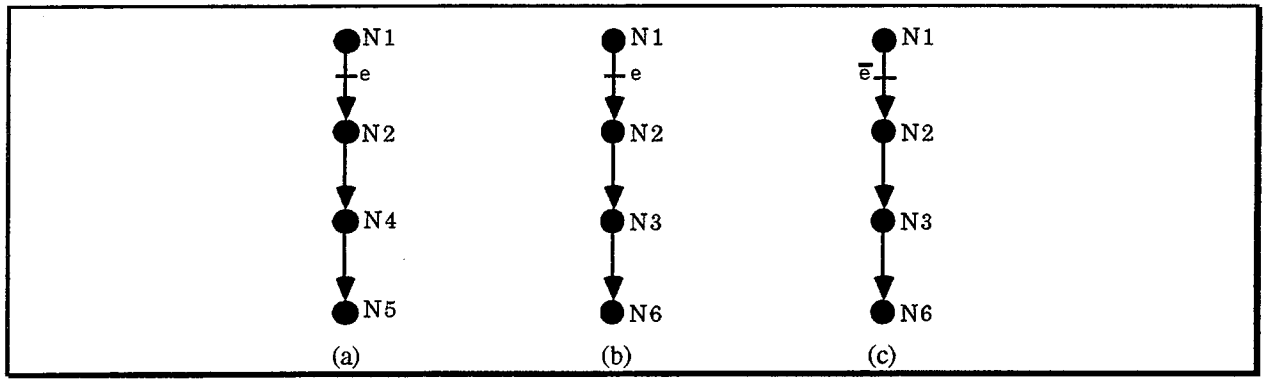

Figure II.3 - Exemple de chemin illégal (a), légal incorrect (b) et correct (c) vis à vis du GFC de la figure II.2.

La vérification du flot de contrôle d'un élément consiste à s'assurer de la correspondance (plus ou moins exacte selon le degré de vérification) entre le flot théoriquement attendu et le flot réellement obtenu. Le flot théoriquement attendu est représenté par le GFC obtenu à partir de la spécification comportementale de l'élément. Le flot réellement obtenu est généralement modélisé, en temps réel, sous la forme de chemins, par l'observation de l'élément lors de son fonctionnement normal. Une vérification complète consiste à s'assurer que :

- le chemin établi en temps réel est correct vis à vis du GFC de référence,
- les opérations associées aux noeuds (ou arcs) sont les mêmes dans le chemin et le GFC,
- les opérations commandées à, ou par, l'élément sont bien exécutées complètement.

Le premier type de vérification concerne le séquencement. Toutefois, vérifier la correction d'un chemin vis à vis du graphe de référence est souvent délicat et difficile, ou tout du moins coûteux, à mettre en oeuvre. La plupart des méthodes de vérification du flot de contrôle se limitent donc à la détection de l'illégalité des chemins du flot réel. Le deuxième type de vérification consiste à s'assurer qu'une opération n'est pas remplacée par une autre, sans modification du séquencement. Enfin, le troisième type de vérification concerne non plus le contrôle proprement dit mais la prise en compte et l'exécution de ce qui est commandé. Ce genre de vérification est très rarement considéré ([Khan 89a; Khan 89b]).

Lorsque l'élément dont on vérifie le flot de contrôle est une machine d'états, le GFC est obtenu soit à partir du graphe de contrôle s'il s'agit d'un contrôleur câblé (chapitre I), soit à partir de l'organigramme du microprogramme s'il s'agit d'un contrôleur microprogrammé. Un exemple de GFC associé à un graphe de contrôle et à un microprogramme est donné figure II.2.

Si le contrôleur est défini par un graphe d'états, chaque noeud du GFC correspond à un état et les arcs du GFC sont les arcs présents dans le graphe d'états, les prédicats restant identiques. L'identificateur d'un noeud du GFC est le code de l'état correspondant dans le graphe d'états. L'opération à effectuer est représentée par la valeur des sorties émises, soit sur les noeuds s'il s'agit d'un automate de Moore, soit sur les transitions s'il s'agit d'un automate de Mealy.

Si le contrôleur est défini par un microprogramme, à chaque microinstruction est associé un noeud du GFC. L'identificateur du noeud est l'adresse de la microinstruction et l'opération à effectuer au niveau du noeud est le code opératoire de cette microinstruction (champ sortie). Le séquencement entre les microinstructions est représenté par les arcs du GFC. Les prédicats associés aux arcs correspondent aux conditions de branchement ou d'éclatement dans l'organigramme. La hiérarchisation du GFC est introduite lorsque le microprogramme fait appel à des sous-programmes.

II.1.2. Détection par codage

La détection d'erreurs dans les contrôleurs commençant souvent par une détection dans leurs blocs combinatoires, nous commencerons par étudier quelques méthodes de détection d'erreurs dans la logique combinatoire, même si celles-ci n'ont jamais été appliquées explicitement au cas particulier des machines d'états finis. Les deux sections suivantes considèrent respectivement la détection d'erreurs dans les contrôleurs câblés et les contrôleurs microprogrammés.

II.1.2.1. Logique combinatoire

La technique la plus souvent utilisée pour détecter les erreurs dans la logique combinatoire est de coder ses sorties avec un code de parité ou un code de Berger. Sous sa forme la plus simple, le code de parité correspond à l'adjonction d'un bit au vecteur de sorties de la logique. La valeur de ce bit est choisie afin d'obtenir une parité constante au niveau du vecteur binaire ainsi formé. La fonction logique de ce bit de parité peut être par exemple construite de manière à ce que le bit de parité ait pour valeur logique 1 si le nombre des autres sorties à 1 est pair, et 0 sinon. Une parité plus complète est la parité multiple. Il s'agit de coder sur r bits (les bits de redondance) le nombre de 1 modulo 2^r. Le code de Berger suit à peu près le même mécanisme. Les r bits ajoutés codent le complément à 1 du nombre de bits de sorties à 1 (ou à 0). Moyennant quelques modifications de ces principes de codage ou de la structure de la logique, le circuit résultant peut détecter un grand nombre d'erreurs résultant de fautes de collage, voire de fautes de retard ou de courts-circuits entre équipotentielles. Le code de parité permet de détecter toutes les erreurs simples dans le vecteur codé, le code de parité multiple toutes les erreurs multiples unidirectionnelles non multiples de 2^r, et le code de Berger toutes les erreurs multiples unidirectionnelles.

[Fuji 84], [De 92, schéma 2], [Sogo 93a], ou encore [De 94, schéma 3], utilisent une technique combinée de codage de parité et de groupement des sorties.

Dans [Fuji 84], le bloc de logique combinatoire, appelons le C, auquel on veut ajouter le dispositif de détection, n'est pas modifié. Pour obtenir la détection d'erreurs, les sorties de C sont séparées en plusieurs groupes, auxquels sont adjoints un bit de parité par groupe. Un bloc (combinatoire) de prédiction de parité est ajouté pour chacun de ces bits, afin de calculer la parité attendue de chaque groupe de sorties à partir des entrées du bloc C. Pour chaque groupe de sorties, un bloc de logique génère le bit de parité obtenu. Enfin, un comparateur auto-contrôlable vérifie la cohérence entre bits de parités attendus et obtenus. Les règles présidant au partitionnement de l'ensemble des bits de sorties du bloc C n'est pas précisé. Le type de fautes visé par cette méthode est l'ensemble des collages. L'intérêt de grouper les sorties est d'augmenter l'efficacité du dispositif de détection. Fujiwara annonce ainsi un taux de détection de 91 à 100% selon le nombre de groupes et la structure exacte du dispositif de détection. Toutefois, l'augmentation du nombre de groupes (c'est-à-dire de bits de parité), augmente aussi le coût de la méthode. Les résultats donnés en terme de nombre de portes logiques montrent cependant un coût matériel généralement plus faible que la duplication (de 75 à 190% contre 145 à 280% pour la duplication [Fuji 84]). Une estimation de l'efficacité de la détection en terme de taux de détection d'erreurs plutôt qu'en terme de taux de couverture de faute est aussi indiquée (entre 88 et 100% de taux de détection), mais semble malheureusement ne pas tenir compte des erreurs pouvant survenir au niveau des bits de parité.

Le schéma 2 dans [De 92] et le schéma 3 dans [De 94] reprennent l'idée du partitionnement des bits de sortie, mais cette fois implantent chaque groupe séparément (sans logique partagée entre deux groupes), avec de la logique multi-couches ([De 92]) ou deux-couches ([De 94]). Un bit de parité est associé au vecteur formé par le premier bit de chaque groupe, un autre au deuxième bit de chaque groupe, et ainsi de suite. Ainsi implanté, une faute simple dans la logique de calcul

d'un groupe sera détectée même si elle provoque une erreur multiple au niveau des sorties. Un algorithme de partitionnement des sorties (pour un nombre de groupes fixé) est fourni. Les essais effectués par les auteurs montrent que la solution la moins coûteuse matériellement consiste à partitionner l'ensemble des bits de sortie en autant de groupes qu'il y a de bits, et donc à n'implanter qu'un seul bit de parité. Les coûts matériels sont indiqués en nombre de littéraux. Le coût du bloc de vérification de la parité n'est pas compris. [De 92] indique un coût moyen de 67% et [De 94] un coût moyen de 61%.

Dans [Sogo 93a], Sogomonyan propose un algorithme de partitionnement des sorties en fonction du modèle de fautes considéré (introduction de la notion de sorties faiblement indépendantes : « weakly independent outputs »). Deux sorties seront élément du même groupe si toute faute simple, de l'ensemble des fautes considéré, ne les rend pas simultanément erronées. Un bit de parité est implanté pour chaque groupe (les groupes sont disjoints). Un bit de parité de l'ensemble des sorties est aussi implanté. La logique de calcul des bits de parité est séparée de la logique de calcul des sorties.

[Sogo 93b], [Göss 93] et [Toub 93] étudient des méthodes visant à modifier, de manière très fine, la structure de la logique combinatoire considérée.

Ainsi, [Sogo 93b] et [Göss 93] reprennent et précisent la notion de sorties faiblement indépendantes. Dans ces deux articles, la logique de calcul des sorties est modifiée (duplication de certaines portes), de manière à ce qu'un collage ne provoque pas plus d'une sortie erronée. Ainsi, l'adjonction d'un seul bit de parité est suffisante pour détecter tous les collages, tout en ayant tout de même de la logique partagée entre les différentes fonctions de calcul des sorties (y compris le bit de parité).

[Toub 93] applique le même principe de duplication de certaines portes, mais avec un degré de liberté plus important puisque la logique peut être modifiée en fixant le nombre ou la parité du nombre de sorties erronées du fait d'un collage dans la logique. L'intérêt est de pouvoir préciser, par exemple, comme contrainte de synthèse, non pas qu'au plus une sortie peut être erronée en cas de faute simple, mais que le nombre de sorties erronées doit être impair, ce qui sera tout aussi bien détecté par le bit de parité.

Le schéma 1 dans [De 92] et le schéma 2 dans [De 94], étudient, eux, l'utilisation du code de Berger. Le principe utilisé est le suivant. Le code de Berger permet de détecter les erreurs multiples unidirectionnelles. Il est donc proposé de synthétiser la logique de telle manière que tout collage se traduise forcément par une variation des sorties erronées uniquement de 0 à 1, ou uniquement de 1 à 0 (en prohibant l'utilisation de logique inverseuse, par exemple). Cette technique donne en moyenne un surcoût de 22% en logique multi-couches (nombre de littéraux, [De 92]) et un surcoût de 112% en logique deux-couches ([De 94]). Ces chiffres ne comprennent pas le coût du bloc de vérification de la cohérence du code. A titre indicatif, [De 94] annonce un surcoût moyen de 162% pour la duplication. Nous verrons que ce principe est aussi utilisé dans [Jha 91] pour les contrôleurs câblés.

II.1.2.2. Contrôleurs câblés

La détection d'erreurs dans les contrôleurs utilise souvent des techniques mises au point pour la détection d'erreurs dans la logique combinatoire, en particulier lorsqu'il s'agit de détecter des erreurs au niveau des sorties primaires du contrôleur. La détection d'erreurs au niveau de la logique d'état suivant pose toutefois parfois des problèmes particuliers qui amènent les auteurs à proposer des solutions originales.

A la base, le problème étudié dans [Osma 73] est le même que celui étudié dans [Göss 93], [Sogo 93b] ou encore [Toub 93], à savoir : comment détecter efficacement les erreurs en sortie d'un bloc combinatoire par l'utilisation d'un bit de parité, tout en autorisant le partage de logique entre les différentes fonctions Booléennes implantées ? Le concept utilisé par Osman est plus complexe à mettre en oeuvre que celui de Touba ou Sogomonyan, mais a le mérite d'avoir été publié vingt ans plus tôt. Le principe utilisé par Osman est de décomposer une fonction Booléenne f à n variables de telle manière que $f_n = z_1g_1+...+z_mg_m$ où :

- les g_i sont de la forme $g_i = m_{1i}+...+m_{ti}$, chaque m_{ji} étant un monôme de n variables,
- les $G_i = \{m_{1i}, ..., m_{ti}\}$ forment une partition des 2^n monômes de n variables.

L'intérêt d'une telle décomposition est de pouvoir partager les fonctions g_i entre plusieurs fonctions f_n différentes. En contrôlant la parité en sortie des g_i et en sortie des f_n, comme indiqué figure II.4, on peut détecter tous les collages simples provoquant une erreur. L'application aux contrôleurs est donnée figure II.4, et permet de détecter les erreurs dans la logique de séquencement et dans la logique de sortie. Le coût matériel n'a pas été évalué par Osman. Nous verrons que cette technique peut aussi être appliquée aux implantations TMR.

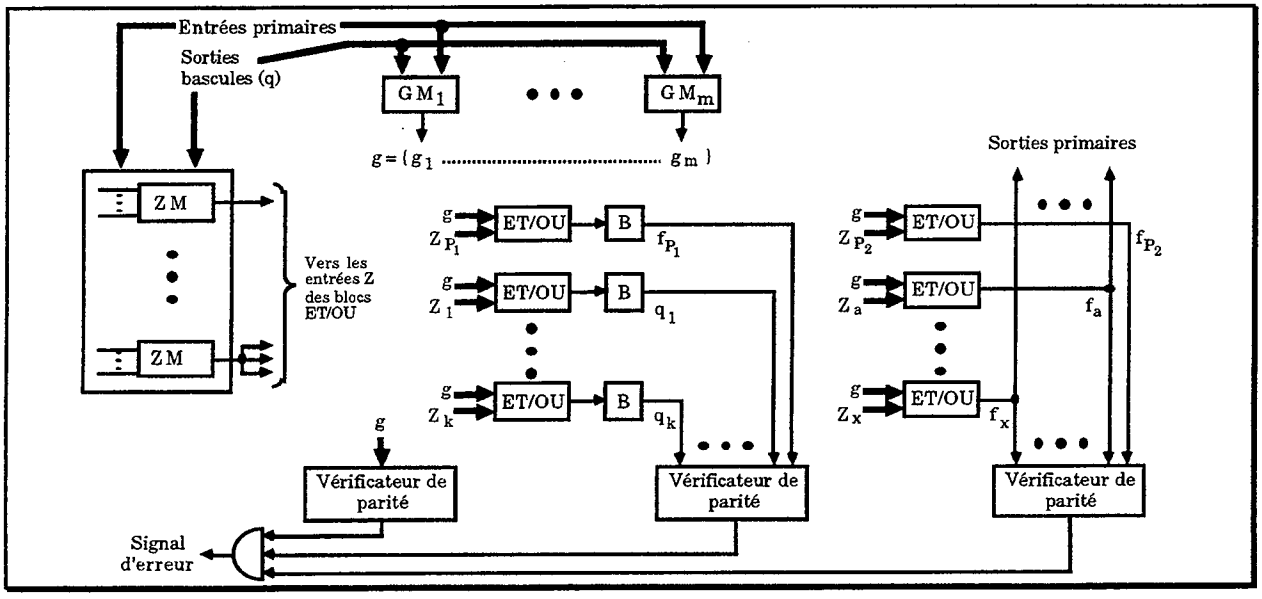

Figure II.4 - Contrôleur à base de parité d'après [Osma 73].

Les autres méthodes de détection d'erreurs dans les contrôleurs détaillées ici agissent directement sur le choix de la procédure de codage des états. Ainsi, [Micz 83] et [Özgü 77] privilégient le codage 1 parmi n. [Özgü 77] propose des techniques de détection pour les machines synchrones et asynchrones. Pour les machines synchrones (celles qui nous intéressent ici), Özgüner montre qu'en codant les états en 1 parmi n et en implantant la logique d'état suivant avec de la logique deux-couches, la première couche formée uniquement de portes ET et la deuxième uniquement de OU, tout collage est détectable car il modifie le nombre de 1 du code d'état calculé. Miczo, dans un article très succinct, reprend la même idée ([Micz 83]) et indique qu'en implantant un vérificateur de parité en sortie du registre d'états, tout collage simple dans la logique d'état suivant est détecté, ainsi que certains collages multiples ou transitoires.

Dans [Jha 91] et [Jha 93], les états sont codés avec un codage m parmi n, et les sorties avec un code de Berger. La logique d'état suivant et de sortie est implantée sans portes inverseuses afin d'éviter qu'une faute ne produise des erreurs bidirectionnelles. Les vérificateurs sont totalement auto-contrôlable (TSC : « Totally Self-Checking »). Les résultats montrent que le codage 2 parmi n donne généralement les meilleurs résultats. Les auteurs ont aussi étudié le coût matériel et le

niveau de détection pour différents exemples lorsque la logique est implantée avec des portes inverseuses et non-inverseuses (ce qui permet une meilleure optimisation des équations Booléennes). Si du point de vue du coût matériel (en nombre de littéraux), les meilleurs résultats sont obtenus avec le codage 1 parmi n, du point de vue du taux de détection des collages simples (qui dans ce cas n'est pas forcément 100%) , les meilleurs résultats sont obtenus avec le codage 2 parmi n (entre 87 et 100%). Le codage 1 parmi n donne, en effet, des résultats entre 52 et 99% et le codage 3 parmi n des résultats entre 83 et 100%.

[Pare 91], [Bolc 95a], et [Bolc 95b] agissent eux sur la distance de Hamming entre le code de l'état courant et le code de l'état suivant. La technique détaillée dans [Pare 91] utilise un moniteur (machine d'états finis simplifiée) et un « comparateur » qui permet de vérifier à tout instant la cohérence entre l'état du moniteur et l'état du contrôleur. Le nombre d'états du moniteur est très faible et induit une partition, ici implicite, de l'ensemble des états du contrôleur. Parekhji montre qu'en codant les états du contrôleur avec un code m parmi n, en respectant une distance de Hamming égale à 2 entre le code d'un état et celui de ses successeurs, il est possible de détecter les erreurs consécutives à des collages simples ou à des fautes de retard simples. Le moniteur comporte alors 2 états si le graphe de contrôle contient uniquement des cycles de longueur paire (en nombre d'états traversés), ou 3 états si le graphe de contrôle contient uniquement des cycles de longueur impaire. Cette méthode permet une réduction moyenne du coût matériel de 16% par rapport à la duplication ([Pare 93]). Le détail du coût de la modification du graphe de contrôle, du coût du codage particulier des états du contrôleur et du coût du moniteur est donné dans [Pare 93]. Les résultats sont donnés en nombre de portes ET et OU.

L'idée utilisée dans [Bolc 95a] et [Bolc 95b] est un peu la même, à savoir imposer une distance de Hamming égale à 2 entre le code d'un état et celui de ses successeurs. Toutefois, la mise en oeuvre de la détection est très différente de celle de [Pare 91] (pas de moniteur) et ne cible que les collages simples. [Bolc 95b] utilise en plus un code de Berger pour détecter les erreurs consécutives à des collages dans la logique de sortie. Le dispositif de détection compare l'entrée et la sortie du registre d'états, afin de vérifier que le code d'état courant est bien à distance 2 du code d'état suivant. L'état suivant n'étant pas mémorisé dans un registre au moment de la comparaison, l'intervalle de temps de validité du signal d'erreur résultant de la comparaison peut être très court, voire difficile à définir. Une solution simple est soit d'utiliser un registre supplémentaire (registre d'état précédent), soit de choisir une fréquence d'horloge intentionnellement supérieure de quelques unités de temps au chemin critique de la logique d'état suivant (quelques nanosecondes en technologie CMOS). En ce qui concerne la logique d'état suivant (comparateur non compris), le coût matériel du codage distance 2 comparé à un codage réalisé avec l'outil NOVA est en moyenne de 26% (en nombre de littéraux) pour un taux de détection moyen de 98,35% (de 88,75% à 100% selon les exemples). Notons que le protocole de simulation de fautes semble exclure de ces chiffres les fautes indécelables au niveau des sorties de la logique d'état suivant. De plus, aucune information n'est fournie à propos des vecteurs de simulation utilisés. Le taux de détection peut être amélioré en codant les états de telle sorte que le passage du code d'état courant au code d'état suivant nécessite un changement bidirectionnel des deux bits modifiés (un de 0 vers 1 et l'autre de 1 vers 0), et en implantant la logique d'état suivant de manière à éviter les erreurs bidirectionnelles. En ce qui concerne la logique de sortie, les auteurs proposent deux méthodes de synthèse : le codage des sorties avec un code de Berger soit après avoir fixé les phi-Booléens des fonctions Booléennes de calcul des sorties, soit avant, auquel cas la valeur des phi-Booléens est choisie non seulement pour permettre une bonne optimisation des équations Booléennes de sortie, mais aussi pour permettre une bonne optimisation des équations Booléennes des bits de redondance. La seconde méthode donne a priori de meilleurs résultats, en terme de coût matériel, que la première. Elle semble, toutefois, difficile à mettre en oeuvre dès que l'exemple traité est un peu important en terme de nombre de sorties primaires, du fait de la complexité algorithmique (temps de calcul) et

du coût mémoire des algorithmes à implanter. Le coût matériel de l'utilisation de codes de Berger n'est pas détaillé.

En dernière remarque sur la méthode qui vient d'être présentée, il faut noter que celle-ci vise principalement la détection des fautes permanentes (collages simples), les fautes transitoires pouvant ne pas être détectées. En effet, la synthèse normale de la logique d'état suivant ne garantit pas une détection immédiate des fautes simples (ceci en dehors du fait que le taux de détection n'est pas de 100%). Ce point est mentionné dans [Bolc 95a]. De plus, [Vosk 96] indique que si on appelle q le nombre de successeurs d'un état, n le nombre de variables d'états et d la distance de Hamming imposée entre le code d'un état et ses successeurs, alors on a :

$$q = \left\lfloor \frac{2n}{d} \right\rfloor$$

Aussi, le nombre de bascules nécessaires pour cette méthode peut devenir très important lorsque la complexité des exemples implantés augmente.

II.1.2.3. Contrôleurs microprogrammés

Quelques méthodes de détection ont aussi été proposées pour les contrôleurs microprogrammés. Elles consistent généralement en un codage du champ adresse, du champ sorties ou des deux champs des microinstructions.

[Beus 70] propose l'utilisation d'un bit de parité pour le champ adresse et un bit de parité pour le champ sorties. Ces 2 bits supplémentaires sont stockés avec les microinstructions du microprogramme. Afin de vérifier que le décodage de l'adresse en entrée de la ROM est correct, la vérification de la parité est effectuée après décodage. Le vérificateur de parité reçoit donc en entrée le bit de parité et les sorties du décodeur. Afin d'éviter d'avoir à vérifier les sorties du décodeur (non disponibles avec les technologies actuelles), une solution est de stocker en ROM deux bits supplémentaires, égaux à 01 si la microinstruction est à une adresse dont le nombre de bits à un est pair, et 10 sinon. Le vérificateur de parité reçoit alors en entrée le bit de parité du champ adresse, et les deux bits ajoutés. L'efficacité du dispositif est alors la même, du point de vue des erreurs simples, que celle du dispositif précédent. Enfin, une troisième possibilité proposée est de diviser la ROM et son décodeur d'adresses en deux, une moitié permettant de stocker les microinstructions dont l'adresse a un nombre pair de bits à 1, et l'autre celles dont l'adresse a un nombre impair de bits à 1. Ceci implique une modification des fonctions logiques du décodeur, et un dessin particulier de la ROM : pour une telle implantation, l'utilisation d'un générateur de ROM classique n'est pas possible. L'auteur indique qu'un tel schéma ne nécessite qu'un bit supplémentaire au lieu de deux (les deux bits de parité restent nécessaires). Afin d'augmenter le pouvoir de détection de ces différentes solutions, Beusher propose l'utilisation d'un code parité multiple modulo 4 (deux bits de parité au lieu d'un pour chaque champ). Si on applique ce codage à la première solution évoquée par Beusher, cela implique de grouper les sorties du décodeur en fonction du résidu de la division entière de l'adresse par 4.

Les travaux de [Cook 73] utilisent explicitement de vieilles technologies (vieilles du point de vue informatique et micro-électronique, s'entend), puisqu'il s'agit de ROM LSI (« Large Scale Integration » : intégration à grande échelle), le reste du circuit étant à base de composants discrets (« Small-Scale Integration » : intégration à petite échelle). Toutefois les idées exprimées dans cet article ne sont pas obsolètes pour autant. Dans la solution proposée par Cook, le champ adresse des microinstructions est protégé par un bit de parité, et les deux champs commandes, du format de microinstruction considéré dans l'article, sont codés avec un code m parmi n (4 parmi 8 pour être précis). La cohérence du code 4 parmi 8 est vérifiée après décodage des commandes : en fonction des commandes activées, le mot de code est reconstitué et vérifié, ce qui permet de

s'assurer que non seulement il n'y a pas eu de fautes en ROM et dans le registre de microinstruction, mais aussi qu'il n'y a pas eu de fautes lors du décodage. En ce qui concerne l'adresse, en fait deux bits de parité sont stockés dans la microinstruction : la parité de l'adresse stockée dans la microinstruction (P_F) et la parité de l'adresse de la microinstruction elle-même (P_N). La comparaison entre le bit P_F de la microinstruction précédente, et le bit P_N de la microinstruction courante permet de détecter une erreur simple en sortie de ROM, dans le champ adresse du registre microinstruction, ou certaines erreurs en sortie du décodeur d'adresses. Cette technique évite de recalculer la parité, mais semble difficile à mettre en oeuvre en cas de branchement ou d'éclatement. Enfin, afin de tenir compte du fait que, en sortie d'une ROM, la probabilité d'avoir deux bits erronés côte à côte est plus importante que celle d'avoir deux bits erronés dispersés, Cook propose d'entrelacer les bits d'adresse et les bits de commande. Le coût des solutions proposées n'est pas évalué.

[Wong 83] présente un contrôleur microprogrammé, basé sur l'Am2910, intégrant des dispositifs de détection d'erreurs. L'Am2910 contient une mémoire de 4Ko (adresses sur 12 bits), un multiplexeur 4:1 qui permet de choisir entre un registre/compteur (R), une entrée externe (D) venant de la ROM, le compteur microprogramme (PC) et une pile (F) comme source de l'adresse de la prochaine microinstruction (Y). La pile permet l'appel et le retour des sous-programmes, ainsi que l'exécution d'instructions de boucles. Le registre R permet de contrôler le décompte des boucles. Un bloc PLA (« Programmable Logic Array » : matrice de logique programmable) permet de calculer les commandes de ces différents éléments en décodant les microinstructions (Fig. II.5).

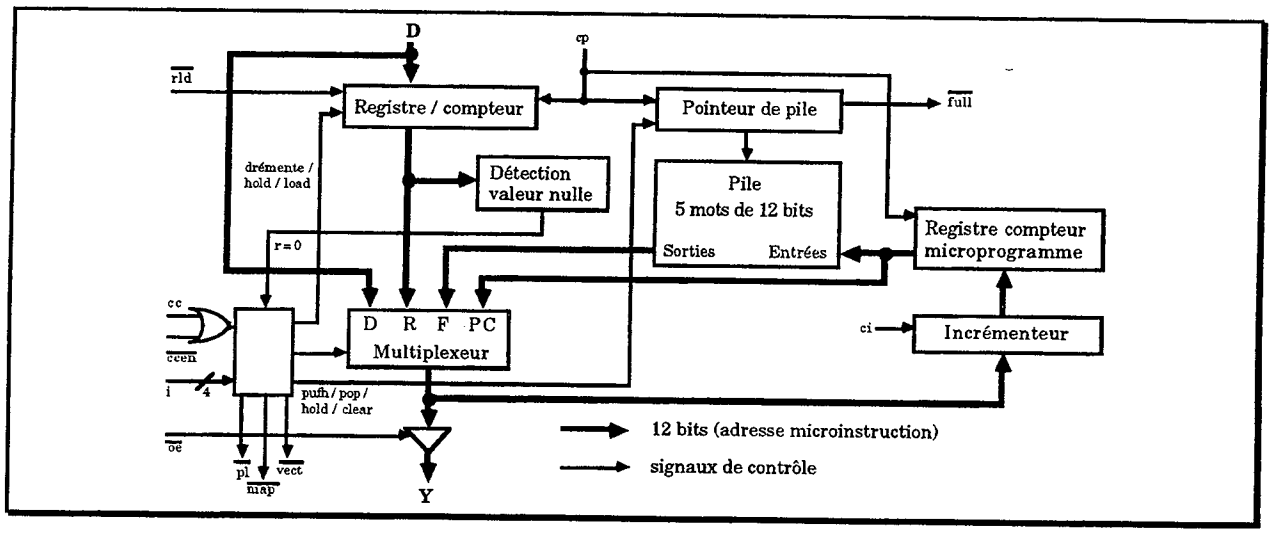

Figure II.5 - Architecture de l'Am2910.

Différentes modifications sont apportées à cette architecture originale. Les plus importantes sont les suivantes. Les mots de la ROM sont codés avec un code de Berger et vérifiés dans le circuit. Les registres R et PC sont dupliqués. 5 bits sont ajoutés à la pile pour stocker les bits de redondance du code de Berger. Lorsque le registre R_1, la pile ou PC_1 est déposé sur le bus d'adresse Y, les bits de redondance stockés dans la pile ou calculés à partir de R_2 ou PC_2 sont déposés sur un bus parallèle qui alimente, avec Y, un vérificateur de Berger. En fait, le code de Berger n'utilise que 4 bits (12 se code sur 4 bits), mais Wong utilise 3 bits qui permettent de détecter les mauvaises sélections de registres (R, PC, l'entrée externe et les 5 registres de la pile). Ces trois bits sont ajoutés, additionnés avec propagation de retenue, aux bits de redondance du code de Berger. Les sorties du PLA sont protégées par un code de Berger modifié, le nombre de 1 en sortie du PLA étant toujours restreint (seulement deux bits de redondance). Ces dispositifs

permettent de protéger le contrôleur contre toutes les erreurs simples, plus un grand nombre d'erreurs multiples. Le coût de la modification du PLA est de 8%. Le coût de la protection des registres R et PC représente 19,6% du circuit (ROM non comprise). En fait, les éléments les plus coûteux sont les générateurs et vérificateurs des bits de redondance qui représentent 27% de la surface finale (toujours ROM non comprise). Cette étude est intéressante parce qu'elle est basée sur un contrôleur réel. Qui plus est le circuit a été placé/routé. Il est à noter que l'auteur s'est appuyé non pas sur un modèle de fautes, mais sur une analyse fonctionnelle (« modèle d'erreurs ») pour déterminer les dispositifs de détection à implanter.

Une autre étude basée sur un processeur réel est présentée dans [Nico 90]. Il s'agit cette fois d'intégrer des dispositifs de détection et de test dans un contrôleur MC68000. La complétude et la complexité de l'étude nous oblige à n'en rapporter ici que les grandes lignes. La détection est obtenue par l'utilisation du code m parmi n pour plusieurs champs de la microinstruction ainsi que des bits de parité. Le générateur d'adresses et le décodeur d'adresses sont protégés à l'aide d'un codage m parmi n des adresses et la regénération de l'adresse codée après le décodeur d'adresses, l'adresse regénérée étant alors comparée au mot d'adresse en entrée du décodeur. Nicolaïdis indique un surcoût de 23% pour les dispositifs de détection (ROM, vérificateurs et comparateur compris). Ce type d'implantation, moyennant quelques règles précises de conception, permet de détecter toutes les fautes simples dans le décodeur d'adresses, la matrice de la ROM et les différents PLA [Nico 90].

II.1.3. VFC par codage

La section précédente a présenté différentes méthodes de détection d'erreurs. Elles s'appuient quasi systématiquement sur des techniques de codage. La présente section montre que certaines de ces techniques permettent aussi la vérification d'un flot de contrôle. La vérification d'un flot de contrôle par codage est généralement basée sur deux principes : l'utilisation de codes de convolution (« convolutional codes ») et l'utilisation de clefs. Les paragraphes suivants donnent des exemples de méthodes utilisant ces deux principes.

II.1.3.1. Contrôleurs câblés

Un exemple de vérification d'un flot de contrôle basé sur les codes de convolution est donné dans [Holm 88] et [Holm 91].

Un code de convolution a la particularité de coder non pas des mots mais des séquences de mots. Pour un code (n, k, m), k représente le nombre de bits d'informations d'un mot, n le nombre de bits du mot une fois codé et m la longueur de la séquence de mots. La détection d'erreurs se fait, par exemple, par l'utilisation d'un LFSR modifié (« Linear Feedback Shift Register » : registre à décalage, à rebouclage linéaire) qui vérifie que les séquences de mots qu'il reçoit en entrée sont bien des séquences du code. Le principe est de coder les états de la machine avec un code de convolution, de façon à ce que les séquences d'états autorisées dans le GFC forment des séquences du code. Cette méthode permet de détecter les chemin illégaux (pas de vérification des prédicats), mais le codage des états afin d'obtenir une bonne détection d'erreurs semble difficile à obtenir ([Holm 88]). En effet, pour éviter le masquage d'erreurs, il faut coder les états de la machine de telle sorte que les chemins légaux forment des séquences du code, bien sûr, mais il faut aussi que tous les chemins illégaux (ou au moins une bonne partie) ne forment pas une séquence du code.

Afin de remédier à ce problème, dans [Holm 91], Holmquist introduit une partition des états de la machine. Plutôt que de coder directement les états, une clef est associée à chaque ensemble d'états formant la partition, cette clef étant choisie de telle manière que les chemins légaux vis à vis du GFC forment une séquence d'un code de convolution. Ainsi, durant le fonctionnement normal de la machine, un bloc combinatoire calcule la clef correspondant à l'ensemble d'états dont est élément l'état courant, selon la partition choisie. Ensuite, cette clef est envoyée au vérificateur du code de convolution choisi pour vérifier si elle s'inscrit dans une séquence autorisée. Cette solution est intéressante mais apporte une restriction majeure au schéma décrit dans [Holm 88]. En effet, une partition permettant de détecter tous les chemins illégaux comprend autant d'ensembles qu'il y a d'états. Aussi, afin d'obtenir des coûts non prohibitifs, Holmquist propose de définir la partition des états en fonction d'un modèle de fautes, et non plus en fonction d'un modèle d'erreurs. D'un autre coté, cette technique est facilement applicable aux contrôleurs microprogrammés.

II.1.3.2. Contrôleurs microprogrammés

L'application aux contrôleurs microprogrammés de la méthode présentée au paragraphe II.1.3.1, décrite dans [Holm 91], est très simple. Plutôt que de recalculer la clef à chaque cycle, il suffit de la stocker directement au niveau de chaque microinstruction. Toutefois, l'utilisation de clefs, aussi appelées signatures imposées, pour la vérification du flot de contrôle des contrôleurs microprogrammés n'est pas nouvelle.

Elle est apparue en 1982, avec [Namj 82b] et [Iyen 82], puis a été approfondie dans [Guha 84], [Stae 85], [Iyen 85], [Guha 87] et [Bail 88].

Différents schémas ont été définis. Ainsi [Namj 82b] propose de stocker deux clefs au niveau de chaque microinstruction. L'une, appelons-la C_c, correspond à la microinstruction courante, l'autre, C_s, à la microinstruction suivante. Le bloc de détection compare donc simplement la clef C_c stockée dans la microinstruction courante, à la clef C_c stockée dans la microinstruction précédente. Les clefs sont soit un numéro, choisi aléatoirement pour certains noeuds, ou choisi en fonction de la structure du GFC pour d'autres (schéma 3 de l'article), soit une forme compressée de la microinstruction (schéma 4). Ainsi, il est possible de détecter certaines erreurs de séquencement, ou certaines erreurs de séquencement plus certaines erreurs de bits dans la microinstruction.

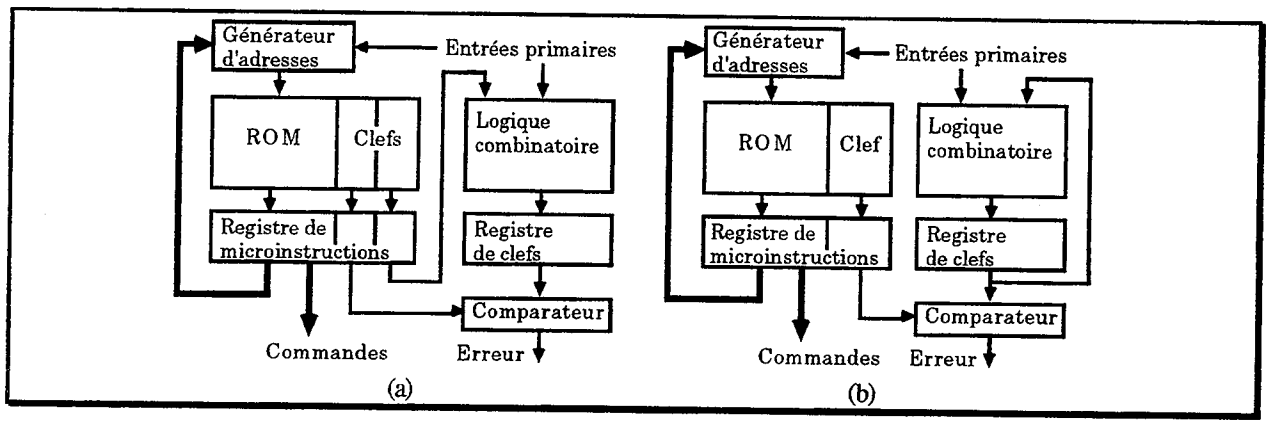

Figure II.6 - VFC à base de clefs : clef double (a), clef unique (b) [Iyen 82, Iyen 85].

Dans [Iyen 82] et [Iyen 85], le schéma général est le suivant. Une simulation de fautes est tout d'abord employée pour déterminer, au niveau comportemental, les transitions erronées à détecter entre noeuds du GFC. La recherche de la détection de ces transitions revient à définir des

relations nécessaires de distinction entre certaines classes de noeuds du GFC (partitions). La prise en compte d'un modèle de fautes, comme pour [Holm 91], permet d'éviter l'attribution d'une clef différente à chaque microinstruction, c'est-à-dire de créer une partition des noeuds formée uniquement de singletons. Le nombre de bits nécessaires pour stocker la (ou les) clef(s) est ainsi grandement réduit. Iyengard propose l'utilisation d'une ou de deux clefs. Lorsque deux clefs sont utilisées, la première correspond à une partition des noeuds sources, et la seconde à une partition des noeuds destinations. Ainsi, la première correspond à la microinstruction courante, et la seconde permet au moniteur de calculer, en fonction des entrées du contrôleur (prédicats), la clef à vérifier au cycle suivant (Fig. II.6a). Lorsqu'une seule clef est utilisée, le coût mémoire peut être diminué, mais au prix d'une augmentation de la complexité du moniteur et d'un partitionnement des noeuds plus difficile (Fig. II.6b). Ces techniques permettent la détection de certains chemins illégaux et de certains chemins incorrects.

Les travaux décrits dans [Stae 85] sont très proches de ceux décrits dans [Iyen 82] et [Iyen 85], bien que les algorithmes de partitionnement des noeuds du GFC soient différents. [Guha 84] et [Guha 87] proposent des méthodes de réduction du coût des moniteurs à une clef, mais au prix de la diminution du taux de détection. Ainsi, des résultats présentés dans [Guha 87] montrent une diminution du coût de la clef et du moniteur associé de 17%, pour une diminution du taux de couverture des collages simples de 3%. Toutefois, [Bail 88] montre que le coût de ces méthodes reste élevé en étant souvent proche du coût de la duplication.

Une méthode originale de détection des chemins incorrects avec vérification des opérations (commandes de la microinstruction) est décrite dans [Berr 90]. Par certains aspects, cette méthode s'apparente au schéma 4 de [Namj 82b].

La détection est basée sur l'utilisation d'un code produit de parité. Pour être codé, un vecteur est « replié » sur lui même de façon à former une matrice de L lignes par C colonnes. Les bits de redondance correspondent alors à la parité de chaque ligne, chaque colonne et la parité du vecteur complet. Les meilleurs résultats en terme de coût sont donc obtenus lorsque L = C. Le code produit de parité permet de détecter toutes les erreurs multiples non multiples de 4. Si le nombre de bits erronés est multiple de 4, l'erreur ne sera pas détectée s'ils forment, 4 par 4, des rectangles dans la matrice de codage.

Une fois les microinstructions codées, les bits de parité sont séparés en deux champs : un champ est stocké dans la microinstruction à laquelle il correspond (on parle de champ de redondance local), et l'autre champ est stocké dans les microinstructions qui précèdent (dans le GFC) la microinstruction à laquelle il correspond (on parle de champ de redondance récurrente).

Le GFC est supposé ne pas contenir de divergences de plus de deux arcs (pas d'éclatements, uniquement des branchements avec au plus deux successeurs). Ceci peut impliquer une modification de l'organigramme du microprogramme, et donc du GFC. Des noeuds doivent être ajoutés pour éliminer les éclatements. Lorsque des états sont ajoutés, le respect de la synchronisation originale nécessite l'implantation de plusieurs signaux d'horloge avec un mécanisme de commutation entre eux.

Le format de microinstruction résultant est le suivant : un champ M contient le masque des entrées concernées par la transition future, un champ V qui correspond à la valeur de ces entrées pour un branchement pris (prédicat de la transition correspondant à un saut d'adresse), un champ sorties OF si la condition de transition est fausse, un champ de sorties OV si la condition de transition est vrai, deux champs CV et CF qui correspondent à l'indicateur de commutation d'horloge, deux champs SV et SF qui correspondent à l'adresse de l'état suivant, deux champs RV et RF qui correspondent à la redondance récurrente et un champ R qui correspond à la redondance locale.

Le vérificateur de parité s'occupe donc de reconstituer les bits de parité de la microinstruction courante par concaténation du champ R de cette microinstruction et du champ RF ou RV de la

microinstruction précédente. Pour un exemple étudié dans l'article, la probabilité de masquage d'erreurs est comprise entre 0,1 et 1,5%, selon que sont considérées les erreurs de séquencement ou les erreurs de bits (dans la ROM ou la logique qui l'entoure). Le coût général est mal défini, mais le coût mémoire est évalué entre 10 et 35%. Enfin, le coût en performance est de 50% du fait de la commutation d'horloge.

II.1.4. VFC par analyse de signature

A part quelques exceptions ([Beus 70], [Namj 82b], [Wong 83], [Holm 88] et [Berr 90]), nous n'avons vu jusqu'à présent que des techniques de détection d'erreurs et de vérification d'un flot de contrôle basées sur des modèles de fautes plutôt que sur une analyse fonctionnelle de l'élément à protéger (vérification comportementale basée sur un modèle d'erreurs). Nous avons déjà émis quelques réserves quant à l'utilisation des modèles de fautes. Nous précisons ici nos raisons.

Partir d'un modèle de fautes a, entre autres, l'inconvénient d'obliger le concepteur à considérer des ensembles de fautes souvent restreints. Il existe alors un risque non négligeable de ne pas considérer, dans l'ensemble des fautes à détecter, la faute qui va réellement survenir. L'intérêt des vérifications comportementales est de ne pas se soucier de la cause (faute) mais de l'effet néfaste (erreur). Une fois définies les propriétés importantes à vérifier, la confiance dans la vérification faite peut être (intuitivement) plus forte. En fait, se restreindre à un ensemble prédéfini de fautes n'est pas une mauvaise chose en soit : elle permet en effet de mieux cibler la redondance et donc, en principe, de réduire les coûts matériels. Ce qui est plus contestable, c'est le choix de l'ensemble de fautes à considérer.

La vérification d'un flot de contrôle par analyse de signature présentée ici est basée sur un modèle d'erreurs (généralement la détection des chemins illégaux). Elle permet ainsi de s'affranchir du problème du choix du modèle de fautes, sans pour autant empêcher une analyse, a posteriori, du taux de couverture pour un modèle donné.

II.1.4.1. Notions de base

La vérification d'un flot de contrôle par analyse de signature a d'abord été étudiée pour la vérification du bon séquencement des programmes d'application et des microprogrammes ([Namj 82a], [Namj 82b], [Srid 82]). Par la suite, elle a évidemment continué à être étudiée pour le test en ligne des microprogrammes (paragraphe II.1.4.2), mais c'est avant tout au niveau système qu'elle a acquis ses lettres de noblesse : [Namj 83], [Shen 83], [Eife 84], [Mahm 85], [Jay 86], [Wilk 87], [Wilk 88], [Delo 89], [Wilk 90], [Mich 91], [Leve 93d], [Ohls 95] n'est qu'une maigre liste des publications dans le domaine. Dans la suite, nous nous intéresserons essentiellement à la vérification d'un flot de contrôle des contrôleurs microprogrammés et cablés. Des résumés des techniques de vérification d'un flot de contrôle au niveau système sont disponibles dans [Leve 90b], [Roch 91] ou encore [Mich 93].

Notion de signature.

Que ce soit pour le test en ligne d'un contrôleur microprogrammé ou câblé, la signature est une information permettant de caractériser le flot de contrôle.

- **Définition II.7 : Signature d'un chemin**

Une signature est une information résultant d'un processus de compaction, permettant de caractériser le chemin suivi dans le graphe du flot de contrôle. L'information compactée est une information pertinente associée à chaque noeud du chemin.

L'information compactée pour le calcul de la signature peut être soit le code de la microinstruction ou de l'état traversé selon qu'il s'agit respectivement de la vérification d'un contrôleur microprogrammé ou d'un contrôleur câblé.

Processus de compaction.

La compaction est un processus séquentiel avec perte d'information. Etant donné que nous nous intéressons ici aux circuits numériques binaires, la signature est représentée par un vecteur binaire, l'information compactée est représentée par des vecteurs binaires (souvent un par noeud du chemin parcouru), et la fonction de compaction est une fonction Booléenne. Les fonctions de compaction généralement utilisées pour le calcul de la signature sont la somme modulo 2 bit à bit et la division polynomiale.

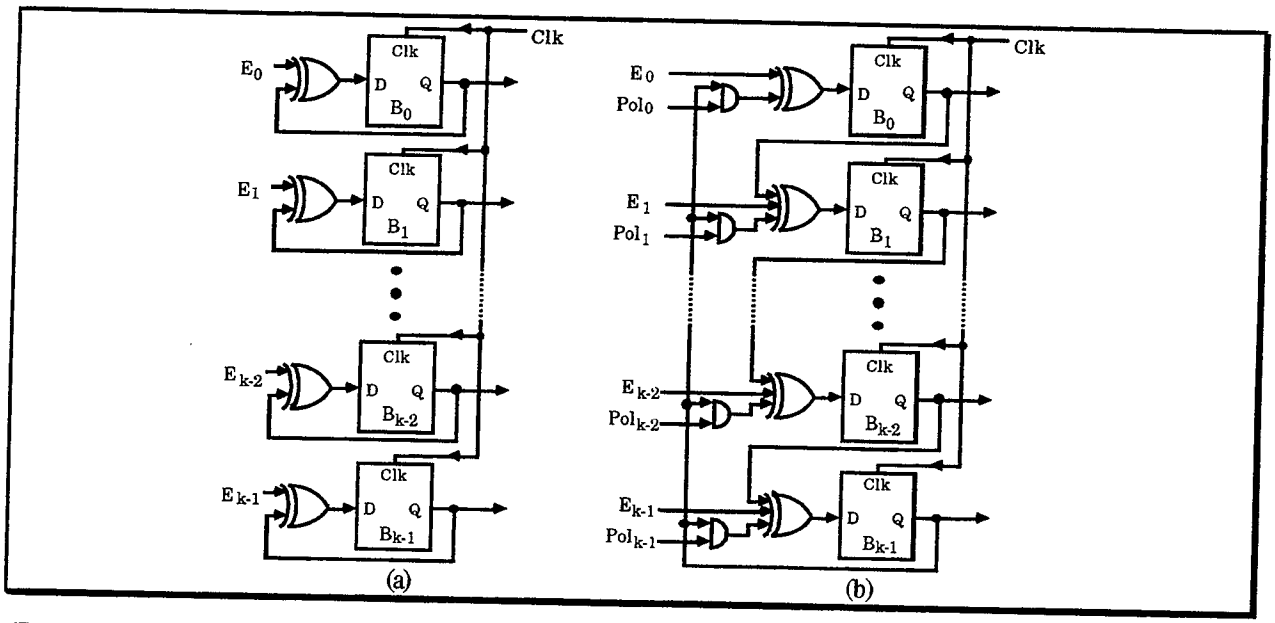

Figure II.7 - Fonctions de compactions : OU-Exclusifs (a), MISR à OU-Exclusifs internes (b).

La somme modulo 2 est réalisée à l'aide de portes logiques OU-Exclusif (portes XOR : « eXclusive OR ») entre la signature déjà calculée, c'est-à-dire la signature du chemin déjà parcouru, et le code microinstruction ou le code d'état associé au prochain noeud traversé. La figure II.7a montre un tel dispositif. Dans cet exemple, la signature et le mot associé à chaque noeud sont codés sur k bits. Les équations Booléennes qui régissent ce dispositif sont les suivantes (t représente la discrétisation du temps) :

$$b_i(t+1) = b_i(t) \oplus e_i(t+1) \qquad \text{pour } 0 \le i \le k\text{-}1 \qquad (1)$$

Les $e_i(t+1)$ représentent l'information à compacter associée au prochain noeud à traverser. Les $b_i(t)$ représentent le contenu des bascules b_i à l'instant t, c'est-à-dire la signature du flot de contrôle avant la traversée du noeud représenté par les entrées $e_i(t+1)$. Les $b_i(t+1)$ représentent la

signature à l'instant $t+1$, c'est-à-dire la signature du chemin suivi par le flot de contrôle, le noeud représenté par les $e_i(t+1)$ compris.

La division polynomiale est généralement réalisée à l'aide d'un MISR (« Multiple Input Shift Register » : registre à décalage à entrées multiples, à rebouclage linéaire). Un exemple de MISR, dit à OU-Exclusif interne à polynôme diviseur programmable, est représenté en figure II.7b. Etant donné que l'on travail en binaire, les polynômes sont représentés par des vecteurs binaires. Par exemple, le polynôme x^3+x+1 est représenté par le vecteur 1011. Dans la figure II.7b, le polynôme diviseur est représenté par le vecteur $Pol_{k-1}Pol_{k-2}...Pol_1Pol_0$. Les entrées $e_{k-1}e_{k-2}...e_1e_0$ représentent l'information à compacter. Les sorties des bascules $b_{k-1}b_{k-2}...b_1b_0$ représentent la signature du chemin déjà parcouru (résultat des divisions polynomiales précédentes).

Les équations Booléennes qui régissent le MISR de la figure II.7b sont les suivantes (t représente la discrétisation du temps) :

$$b_i(t+1) = b_{i-1}(t) \oplus (Pol_i \,.\, b_{k-1}(t)) \oplus e_i(t+1) \qquad \text{pour } 1 \le i \le k-1 \qquad (2)$$
$$b_0(t+1) = e_0(t+1) \oplus (Pol_0 \,.\, b_{k-1}(t))$$

Le polynôme diviseur est supposé constant. Les $e_i(t+1)$, $b_i(t)$, et $b_i(t+1)$ ont la même signification que dans l'équation (1).

Le choix du polynôme diviseur a son importance. En effet, la probabilité de masquage d'erreurs d'un MISR n'est pas nulle, et un choix judicieux du polynôme peut la réduire de manière sensible. Les différentes études montrent l'importance du choix d'un polynôme primitif premier pour obtenir les meilleurs résultats en terme d'efficacité de la vérification du flot de contrôle [Leve 90b].

Un avantage de la division polynomiale par rapport à la somme modulo 2 bit à bit est de ne pas être commutative. Ceci signifie qu'un chemin n'a pas la même signature selon qu'il est parcouru du premier noeud jusqu'au dernier, ou du dernier jusqu'au premier. Une étude détaillée des fonctions de compaction est fournie dans [Leve 90b], au chapitre I.

Vérification de la signature.

Avant d'aller plus avant dans la présentation du processus de vérification, il nous faut définir la notion de signature avant et après un noeud.

- **Définition II.8 : Signature avant et après un noeud**

Soit $b_i(t)$ le vecteur binaire représentant la signature à l'instant t, $b_i(t+1)$ le vecteur représentant la signature à l'instant t+1 et $e_i(t+1)$ le mot associé au prochain noeud n traversé par le flot de contrôle.

*On appelle **signature avant n**, la signature représentée par le vecteur $b_i(t)$, et **signature après n**, la signature représentée par le vecteur $b_i(t+1)$.*

Le principe de la vérification de signature est le suivant. Lors de la conception de l'élément à vérifier, les signatures des chemins du GFC sont calculées. Nous verrons que le nombre de chemins étant important, les différentes techniques de VFC s'arrangent pour que la signature après un noeud soit constante dans le GFC, c'est-à-dire que tous les chemins légaux (ou corrects, selon

le niveau de vérification), conduisant à ce noeud aient la même signature. Dans le cas contraire, le coût de la vérification serait prohibitif. Ceci a été formalisé dans [Leve 90b] et sera détaillé au paragraphe II.1.4.3.

Une fois la signature après chaque noeud fixée et calculée, plusieurs points de contrôle, ou points de vérification, sont choisis selon différents critères : minimisation de la latence de détection, du coût matériel, de la probabilité de masquage. Les signatures après les noeuds points de contrôle sont mémorisées : elles sont appelées « **signatures de référence** ». Lors de l'exécution du programme d'application, ou lors du fonctionnement normal du contrôleur microprogrammé ou câblé, la signature du flot de contrôle est calculée en temps réel. Lorsque le flot de contrôle traverse un point de vérification, la signature est comparée à la signature de référence en ce point. Si les deux signatures sont différentes, c'est que le flot de contrôle n'est pas conforme à celui attendu spécifié par le GFC. En d'autres termes, le flot de contrôle à l'exécution a emprunté un chemin illégal vis à vis du GFC de référence. Si la signature calculée en temps réel est identique à la signature de référence, c'est que le flot de contrôle est conforme à celui attendu, avec une probabilité plus ou moins forte. Cette probabilité est fonction de la méthode de vérification par analyse de signature utilisée, et de la probabilité de masquage d'erreurs de la fonction de compaction choisie.

L'unicité des signatures après chaque noeud du GFC est généralement obtenue par l'utilisation d'ajustements.

- **Définition II.9 : Ajustement**

*Un **ajustement** est un vecteur binaire permettant de modifier artificiellement la signature courante tout en lui conservant ses propriétés de détection d'erreurs.*

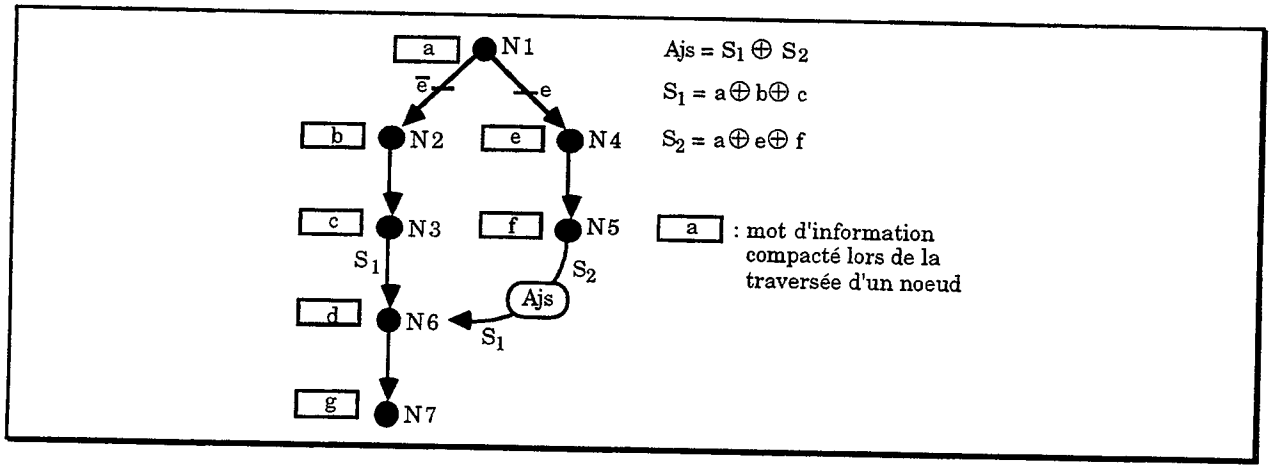

Figure II.8 - Ajustement dans un GFC pour une fonction de compaction à base de OU-Exclusifs.

La figure II.8 donne un exemple d'ajustement. La fonction de compaction considérée est la somme modulo 2 bit à bit. La signature S_2 est ajustée lors de la transition entre les noeuds *f* et *d*. Ceci signifie que le dispositif de vérification détecte qu'il s'agit d'une transition avec ajustement, génère ou récupère (si elle est mémorisée) la valeur d'ajustement correspondant, et effectue la somme modulo 2 entre S_2 et *Ajs* de manière à obtenir une signature avant *d* égale à S_1 en l'absence d'erreur. Ainsi, si pendant le fonctionnement normal de l'élément vérifié, la signature S'_2 après le noeud *f* est erronée, la signature S' calculée par l'ajustement de S'_2 est égale à $S'_2 \oplus Ajs$, à savoir $S'_2 \oplus S_1 \oplus S_2$. Etant donné que S'_2 est erronée, c'est-à-dire différente de S_2, S' est égale à $S_1 \oplus \xi$,

avec $\xi = S'_2 \oplus S_2$ différent du vecteur nul. S' est donc différente de S_1 et l'erreur mémorisée dans S'_2 reste détectable.

Le schéma général de vérification présenté ci-dessus est, évidemment, susceptible d'être modifié de manière plus ou moins importante selon la technique de vérification choisie.

II.1.4.2. Contrôleurs microprogrammés

Dans [Namj 82b], deux méthodes de vérification du flot de contrôle, par analyse de signature, des contrôleurs microprogrammés sont exposées. Il s'agit de deux méthodes très proches des méthodes PSA et PSA généralisée décrites dans [Namj 82a] pour les programmes d'application. La fonction de compaction est la somme modulo 2 bit à bit.

Le schéma 1 de [Namj 82b] s'appuie sur le principe suivant. L'implantation des dispositifs de test en ligne commence par une étude du GFC afin d'en localiser les blocs linéaires. Une fois les blocs linéaires identifiés, leur signature est calculée. On obtient donc une signature de référence par bloc. Cette signature est stockée au début du bloc auquel elle correspond. Un champ supplémentaire *s* d'une longueur de 1 bit est ajouté aux microinstructions, de manière à pouvoir différencier les microinstructions normales, des microinstructions utilisées pour stocker les signatures de référence des blocs. Lorsque la microinstruction contient une signature (début de bloc), un dispositif (appelé « masque de signature » dans [Namj 82b]) permet de charger le registre de microinstruction avec une microinstruction NOP (« No Operation » : pas d'opération). La signature calculée en temps réel est testée à la fin de chaque bloc, ou plutôt juste avant le début du bloc suivant, lorsque le bit *s* identifie une nouvelle microinstruction contenant une signature de référence. Etant donné que la signature de référence est la première valeur du bloc à être compactée lors du calcul de la signature en temps réel, à la fin du bloc, il suffit de vérifier que la signature calculée est nulle (Fig. II.9 et Fig. II.10a).

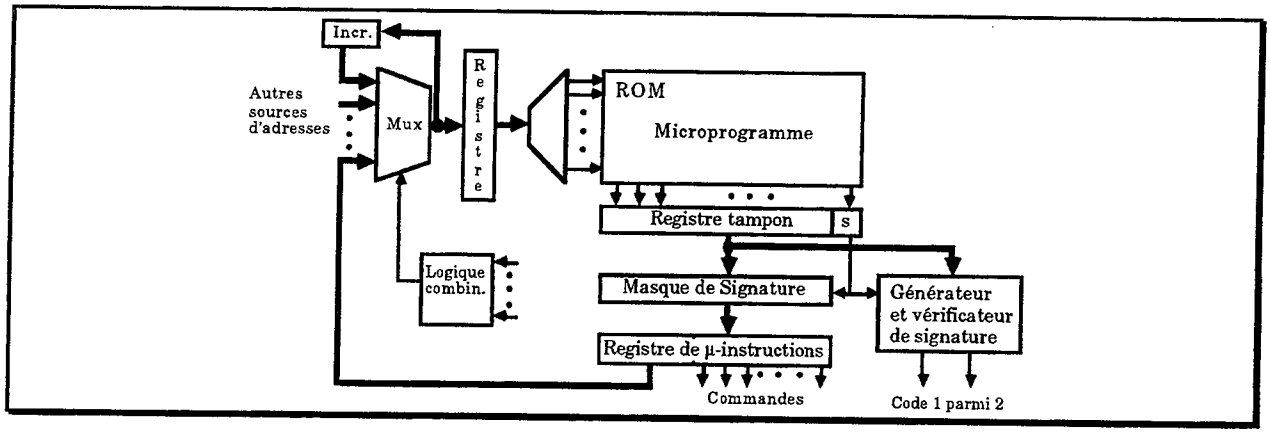

Figure II.9 - Implantation du schéma 1 de [Namj 82b].

Le schéma 2 (Fig. II.10b) tente d'améliorer cette première solution en réduisant le coût mémoire et en améliorant le pouvoir de détection. Le coût mémoire du schéma un est de 1 bit par microinstruction, plus un mot mémoire par bloc (la taille d'un bloc est généralement de quelques instructions seulement). La méthode détecte les erreurs de séquencement à l'intérieur d'un bloc, et les erreurs de bits dans le champ commandes (détection des erreurs d'opération). Par contre elle ne permet pas de détecter les erreurs de séquencement entre blocs, c'est-à-dire le saut erroné d'une fin de bloc A vers le début d'un bloc B, alors que A et B ne sont reliés par aucun arc dans le GFC. Aussi, le schéma 2 ne travaille plus sur la notion de blocs linéaires, mais sur la notion de chemins, un chemin traversant généralement plusieurs blocs linéaires. Un algorithme fourni permet de segmenter le GFC en plusieurs chemins non forcément disjoints. La signature de référence de

chaque chemin est calculée et stockée au début du chemin. Lorsque plusieurs chemins ont le même point d'entrée, c'est-à-dire débutent sur le même noeud, des ajustements sont ajoutés en différents points des chemins concernés, afin qu'ils aient tous la même signature. La signature calculée en temps réel lors du fonctionnement normal du contrôleur est comparée à 0 à la fin de chaque chemin. Un champ de deux bits s1 et s2 est ajouté à chaque microinstruction afin de différencier les microinstructions normales, les microinstructions contenant une signature de référence, les microinstructions contenant un ajustement et la dernière microinstruction des chemins.

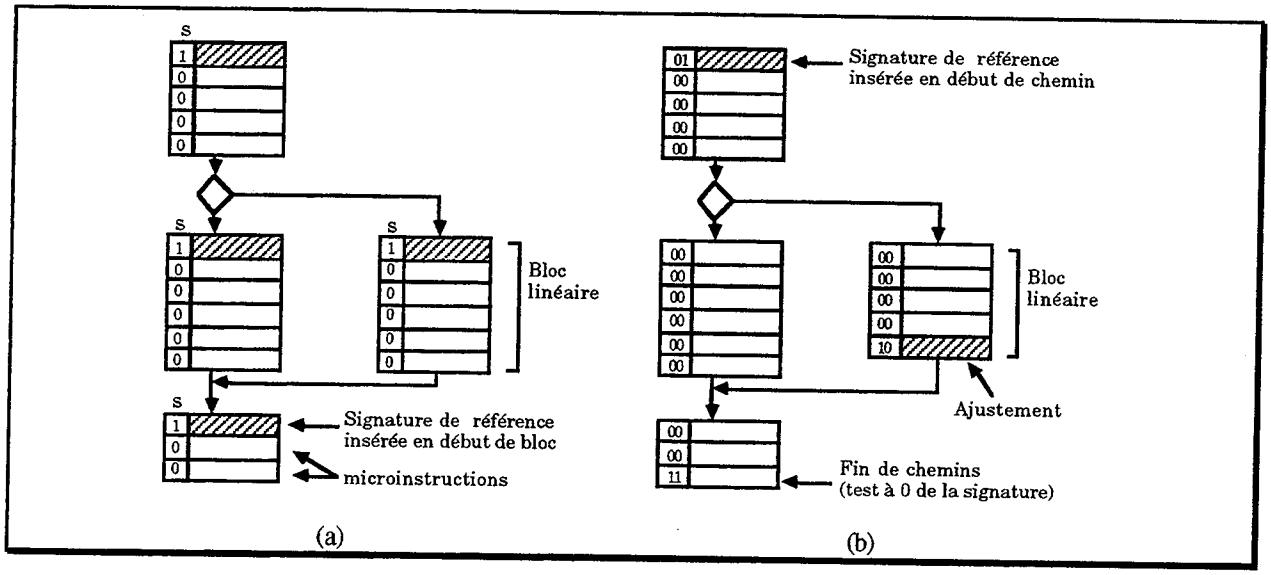

Figure II.10 - Insertion de signatures de référence [Namj 82b]-schéma 1 (a), de signatures de référence et d'ajustements [Namj 82b]-schéma 2 (b).

Le coût matériel de ce deuxième schéma est de 2 bits par microinstruction, mais le nombre de mots mémoires à ajouter pour les signatures de référence et les ajustements est généralement moindre que celui du schéma 1. Le pouvoir de détection est amélioré puisque la signature prend en compte un certain nombre de séquencements entre les blocs.

[Srid 82] propose une technique dite de « synchronisation », qui correspond en fait à une forme particulière d'ajustement de la signature. Pour ce faire, un champ C de k bits est ajouté à chaque microinstruction. k est généralement égal au nombre de bits nécessaire pour coder toutes les adresses de la ROM. Si k est choisi inférieur à ce nombre, le risque de masquage d'erreurs est augmenté. Seule la valeur de ce champ est compactée. Pour les microinstructions correspondant à des noeuds origines d'une convergence dans le GFC, la valeur de C est choisie de manière à obtenir une même signature après tous les noeuds origines d'une même convergence (ajustement). La valeur de C est choisie égale à l'adresse de la microinstruction courante pour les autres noeuds. Des points de vérification explicites sont indiqués par un champ de 1 bit ajouté à toutes les microinstructions. La fonction de compaction est la division polynomiale. Afin de faciliter le test de la signature calculée en temps réel, les valeurs du champ C sont choisies de telle sorte que la signature de référence soit nulle ou tout à 1 aux points de contrôle. L'inconvénient de cette méthode par rapport à celles proposées par Namjoo est de ne vérifier que le séquencement. L'intérêt est d'éviter l'insertion d'instructions NOP, et donc de réduire le coût en performances.

Un deuxième schéma permettant de compacter toute la microinstruction dans la signature est proposé dans [Srid 82]. Il consiste en l'insertion, dans le microprogramme, de microinstructions dédiées à l'ajustement de la signature. Ce schéma est très proche du schéma 2 décrit dans [Namj 82b], mais ne nécessite pas l'ajout de deux bits aux microinstructions, ni le stockage de la signature de référence en début de chemin. Toutefois cette dernière caractéristique semble

impliquer un nombre plus important d'ajustements. Nous verrons qu'une méthode similaire au premier schéma décrit dans [Srid 82] est utilisée dans [Anto 90] pour l'obtention de la tolérance aux fautes par détection/recouvrement (paragraphe II.2.1.1).

Notons que la réinitialisation de la signature à une valeur identique pour tous les blocs ([Namj 82b] schéma 1), ou pour tous les chemins ([Namj 82b] schéma 2, [Srid 82]), augmente très sensiblement la probabilité de masquage d'erreur, du fait d'une forte corrélation entre les signatures de référence ([Wilk 87]). Pour une meilleur détection, il est préférable d'avoir une répartition uniforme des valeurs des signatures de référence dans le GFC.

Une autre approche consiste à associer une signature à chaque routine. La routine peut correspondre à un microprogramme complet correspondant à une macroinstruction ([Dura 83], Mahm 85]), ou à un simple sous microprogramme ([Tung 86]).

Dans [Mahm 85], une signature de référence est associée à chaque microprogramme correspondant à une macroinstruction. La fonction de compaction est la division polynomiale. La signature de référence est calculée par compaction des microinstructions du microprogramme. La signature calculée en temps réel est comparée à la signature de référence à la fin du microprogramme, c'est-à-dire à la fin de l'exécution de la macroinstruction. Les signatures de référence sont stockées dans une deuxième ROM. Des ajustements sont employés afin d'obtenir une signature unique pour tout le microprogramme malgré les boucles et les branchements (convergences et divergences dans le GFC), c'est-à-dire malgré le nombre important de chemins que peut emprunter le flot de contrôle dans un microprogramme. Un bit spécifique est ajouté à chaque microinstruction afin de différencier les ajustements des instructions normales. Comme dans les deuxièmes schémas de [Namj 82b] et [Srid 82], l'ajustement de la signature signifie une perte de performance (exécution d'une microinstruction NOP). Afin de la limiter, comme [Srid 82], [Mahm 85] propose l'ajout d'un champ spécifique aux microinstructions pour le calcul de la signature. Cette méthode vérifie la légalité des chemins et les commandes émises.

Dans [Dura 83], la fonction de compaction est aussi la division polynomiale, mais deux MISR sont utilisés : un pour le champ adresse et un pour le champ commandes. Les signatures de référence sont stockées dans une seconde ROM. Toutefois, contrairement à [Mahm 85], aucun ajustement n'est effectué. Les microinstructions de branchement sont détectées et la signature de référence de la macroinstruction est sélectionnée en fonction de la valeur des bits de condition (un microprogramme a donc généralement plusieurs signatures de référence selon le chemin suivi). Ainsi, [Dura 83] vérifie la correction des chemins, ainsi que les commandes émises. Le coût mémoire est de 8% sur l'exemple traité, mais le coût total n'est pas évalué.

Dans [Tung 86], les signatures de référence sont calculées pour chaque sous-programme. Seule la légalité des chemins est vérifiée. La fonction de compaction est la division polynomiale. Un peu comme dans [Namj 82b], une valeur est stockée au début de chaque sous-microprogramme. Cette valeur est toutefois le résultat de la différence entre la signature de référence du sous-programme et l'adresse de celui-ci (technique de hachage). Un bit associé à chaque microinstruction permet de distinguer les deux types de microinstructions. Le processus de vérification est alors le suivant : à l'exécution, la signature de référence est obtenue par la somme de la valeur stockée dans la première microinstruction et l'adresse d'appel du sous-microprogramme. La signature courante calculée en temps réel est comparée à la signature de référence à la fin du sous-microprogramme.

II.1.4.3. Contrôleurs câblés

L'application aux contrôleurs câblés de la vérification d'un flot de contrôle par analyse de signature a été introduite à la fin des années 80 par [Leve 89a], [Leve 89b], [Leve 89c],

[Delo 89a], [Leve 90a], ou encore [Leve 90c]. La méthode proposée a, en particulier, été précisément formalisée dans [Leve 90b].

Méthode proposée dans [Leve 90b].

L'idée de la méthode proposée dans [Leve 90b] est la suivante. Le GFC est déduit du graphe d'états du contrôleur en considérant chaque état comme un noeud du GFC et chaque transition comme un arc du GFC (voir figure II.1, paragraphe II.1.1.3). La fonction de compaction proposée est la division polynomiale. L'information compactée pour chaque noeud, c'est-à-dire pour chaque état, est le code binaire de cet état. La signature calculée est donc bien représentative du chemin suivi par le flot de contrôle.

Toutefois, comme au niveau système ou pour les contrôleurs microprogrammés, la multiplicité des chemins permettant d'atteindre un noeud du GFC entraîne la multiplicité des signatures après chaque noeud. La vérification du flot de contrôle sans recherche d'optimisations a toutes les chances de conduire à un coût matériel prohibitif. La seule méthode proposée dans les paragraphes précédants et ayant choisi cette stratégie est [Dura 83]. Afin de réduire le coût de la vérification, R. Leveugle propose l'introduction d'une propriété d'invariance forcée au niveau du graphe d'états.

- **Propriété II.1 : Invariance forcée**

La signature d'une séquence de codes d'états obtenue par division polynomiale, avec un polynôme diviseur donné, en suivant un chemin légal d'un GFC, est invariante en sortie de chaque état de ce GFC.

Ainsi, la signature invariante obtenue après un point de vérification est utilisée comme signature de référence. R. Leveugle montre qu'une condition nécessaire et suffisante pour obtenir la propriété d'invariance dans un GFC est la suivante.

- **Condition nécessaire et suffisante II.1 : Identité des signatures**

Les signatures des séquences de codes d'états obtenues par division polynomiale, avec un polynôme diviseur donné, en suivant des chemins légaux du GFC, sont identiques après tous les états ayant un successeur commun.

Le respect de cette condition nécessaire et suffisante peut se faire soit par l'utilisation de techniques d'ajustement, comme c'est généralement le cas au niveau système ou pour les contrôleurs microprogrammés ([Namj 82b], [Mahm 85], [Tung 86], ...), soit par le choix judicieux des valeurs à compacter ([Srid 82], [Anto 90]). Appliqué aux contrôleurs câblés, cela signifie choisir les codes des états de telle manière que la condition II.1 soit respectée.

- **Définition II.10 : S-codage**

*On appelle « **S-codage** » d'un graphe de contrôle un codage des états rendant la signature des codes des états invariante après chaque état.*

Toutefois, contrairement à [Srid 82] où une même valeur du champ compacté peut être utilisée dans plusieurs microinstructions, ici un code binaire donné ne peut être, bien évidemment, associé qu'à un seul état du graphe de contrôle. R. Leveugle a mis en évidence le fait que certaines structures, dans un GFC donné, pouvaient conduire à des situations non S-codables.

- **Définition II.11 : SC-graphe**

*On appelle « **SC-graphe** » un graphe de contrôle dont la structure autorise un S-codage.*

- **Définition II.12 : NSC-graphe**

*On appelle « **NSC-graphe** » un graphe de contrôle n'étant pas un SC-graphe.*

La mise en évidence des situations non S-codables, et le S-codage d'un SC-graphe s'appuie sur les deux théorèmes qui suivent. En particulier, le théorème II.2 montre qu'il est préférable de ne pas utiliser un polynôme diviseur divisible par x+1 : ceci renforce l'intérêt déjà mentionné de l'utilisation d'un polynôme diviseur premier primitif. Dans la suite, on appelle **état rebouclé** un état qui est son propre successeur, c'est-à-dire un état tel qu'il existe, dans le graphe de contrôle, un arc à la fois entrant et sortant pour cet état.

- **Théorème II.1 :**

Lors d'un S-codage, trois paramètres sont à déterminer pour chaque état : la signature avant l'état, la signature après et le code de l'état. Dans le cas d'un MISR à OU-Exclusifs internes, si le polynôme diviseur n'est pas divisible par x, alors le choix de la valeur de deux de ces paramètres définit une valeur et une seule pour le troisième paramètre.

- **Théorème II.2 :**

Si le polynôme diviseur du MISR à OU Exclusif internes employé lors du S-codage, sur k bits, d'un SC-graphe est divisible par x+1, sans l'être par x, alors seuls 2^{k-1} des 2^k codes possibles peuvent être utilisés pour un état rebouclé.

La démonstration de ces théorèmes est faite dans [Leve 90b]. Le théorème II.2 résulte de la condition II.1 appliquée au cas particulier des états rebouclés. Pour ce type d'états, les signatures avant et après un état doivent être identiques, ce qui, du fait du théorème II.1, impose une contrainte forte sur le choix du code d'état lorsqu'un polynôme diviseur divisible par x+1 est utilisé.

R. Leveugle propose deux algorithmes de reconnaissance des SC-graphes et une technique (non systématique) de modification du graphe de contrôle pour l'obtention d'un SC-graphe, fonctionnellement équivalent, à partir d'un NSC-graphe (sans dégradation de performance due à l'ajout de cycles d'attente). Ceci s'appuie sur la formalisation suivante :

- **Définition II.13 : E-équivalence**

*Deux états sont dits « **E-équivalents** » dans un graphe de structure donnée si cette structure et le respect de la propriété II.1 conduisent à imposer la même signature en entrée des deux états quelque soient le polynôme diviseur premier primitif et les conditions initiales de codage choisis.*

- **Définition II.14 : S-équivalence**

*Deux états sont dits « **S-équivalents** » dans un graphe de structure donnée si cette structure et le respect de la propriété II.1 conduisent à imposer la même signature en sortie*

des deux états quelque soient le polynôme diviseur premier primitif et les conditions initiales de codage choisis.

- **Définition II.15 : C-équivalence**

*Deux états sont dits « **C-équivalents** » dans un graphe de structure donnée si cette structure et le respect de la propriété II.1 conduisent à imposer le même code pour les deux états quelque soient le polynôme diviseur premier primitif et les conditions initiales de codage choisis.*

- **Théorème II.3 :**

Deux états sont C-équivalents dans un graphe si et seulement si ils sont à la fois E-équivalents et S-équivalents.

La démonstration de ce théorème est faite dans [Leve 90b].

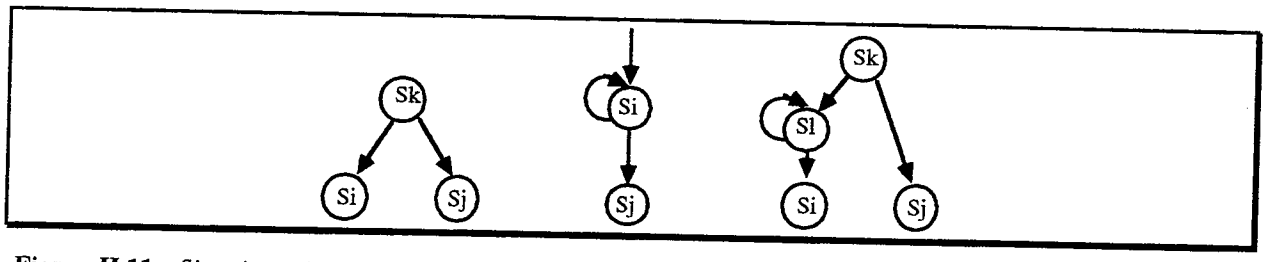

Figure II.11 - Situations simples conduisant à une E-équivalence des états Si et Sj.

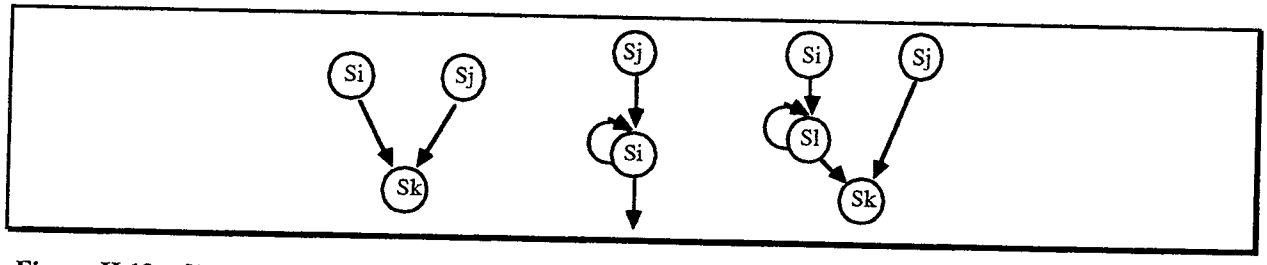

Figure II.12 - Situations simples conduisant à une S-équivalence des états Si et Sj.

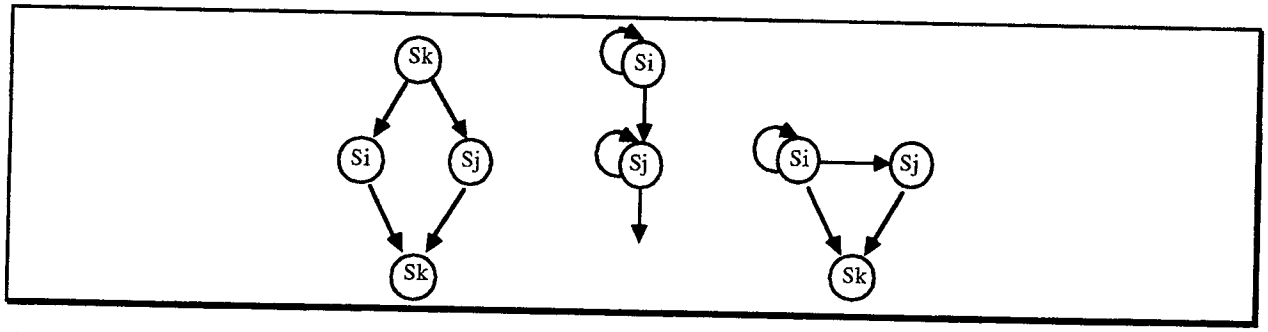

Figure II.13 - Situations simples conduisant à une C-équivalence des états Si et Sj.

Les figures II.11, II.12 et II.13 montrent des exemples de situations conduisant respectivement à une E-équivalence, à une S-équivalence et à une C-équivalence des états Si et Sj.

Les algorithmes de reconnaissance des SC-graphes effectuent un traitement global sur le graphe considéré. Ils reposent sur l'identification des classes de E et S-équivalences, puis sur la recherche des intersections entre ces deux types de classes afin de déterminer des classes d'états

C-équivalents. Le graphe est S-codable lorsque la cardinalité de toutes ses classes de C-équivalences est 1.

La technique utilisée pour supprimer les C-équivalences est de répliquer certains états, ou plus exactement de les scinder en un ensemble fonctionnellement équivalent de plusieurs états. Dans la figure II.14, le scindement de l'état S1 (figure II.14b) supprime la C-équivalence entre les états S2 et S3, mais crée un graphe uniquement fonctionnellement équivalent : le test de l'entrée e est avancé d'un cycle. Dans le cadre d'une modification systématique sans informations particulières sur les degrés de liberté temporels disponibles dans le graphe, ce type de scindement correspondant à une partition des successeurs de l'état scindé n'est pas utilisable. Une autre solution est de scinder l'état S4 en partitionnant l'ensemble de ses prédécesseurs (figure II.14c). Avec une telle modification, le graphe généré est fonctionnellement *et* temporellement équivalent au graphe initiale, ce qui évite d'avoir besoin d'informations sur les degrés de liberté temporels du graphe. Dans la suite nous ne considérerons que ce deuxième type de scindement. La modification d'un NSC-graphe en un SC-graphe équivalent du point de vue fonctionnel et temporel est souvent appelée **réparation** d'un NSC-graphe. La définition II.16 précise la notion d'états cousins résultant d'un tel scindement.

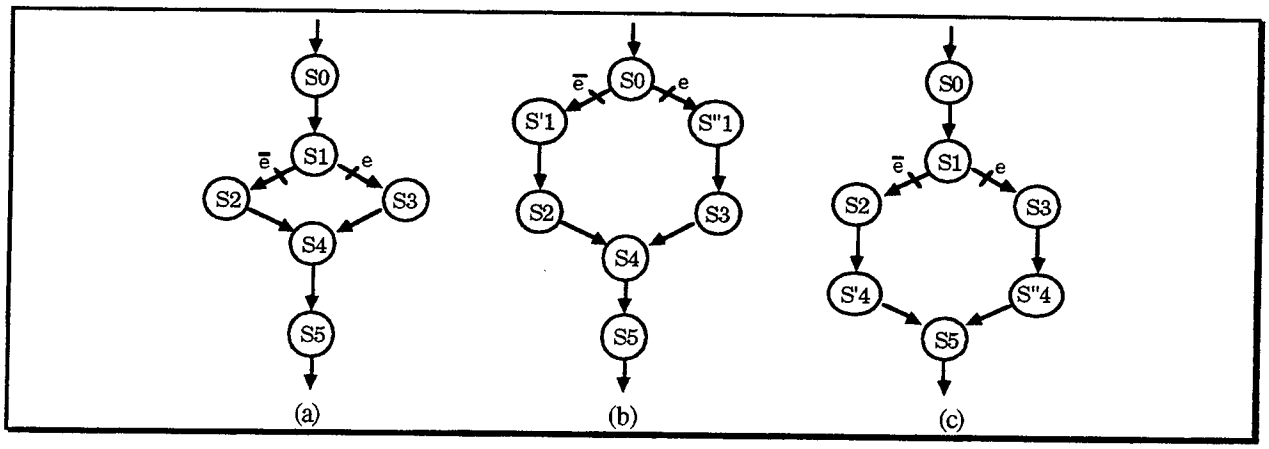

Figure II.14 - Exemple de modification de NSC-graphe : graphe initial (a), graphe modifié fonctionnellement équivalent (b), et graphe modifié fonctionnellement et temporellement équivalent (c).

- **Définition II.16 : Etats cousins**

Lors de la réparation d'un NSC-graphe G, on appelle ***états cousins*** *d'un état S du graphe d'origine G, les états issus du scindement de l'état S, chacun d'entre eux ayant les mêmes successeurs que S et ayant pour prédécesseurs un sous-ensemble des prédécesseurs de S. Les états cousins d'un état S forment une partition des états prédécesseurs de S.*

Dans la figure II.14c, les états S'4 et S''4 issus du scindement de S4 sont des états cousins de S4. L'ensemble {S2, S3} des prédécesseurs de S4 est partitionné en deux singletons. Scinder S4 en deux états S'4 et S''4 permet de supprimer la relation de S-équivalence des états S2 et S3, et donc la relation de C-équivalence.

L'architecture proposée pour la vérification du flot de contrôle est illustrée par la figure II.15. Les éléments à ajouter au contrôleur pour permettre son test en ligne continu sont : le MISR pour le calcul de la signature, le comparateur et le bloc de calcul des références pour les comparaisons, et enfin une bascule D de mémorisation du signal d'erreur.

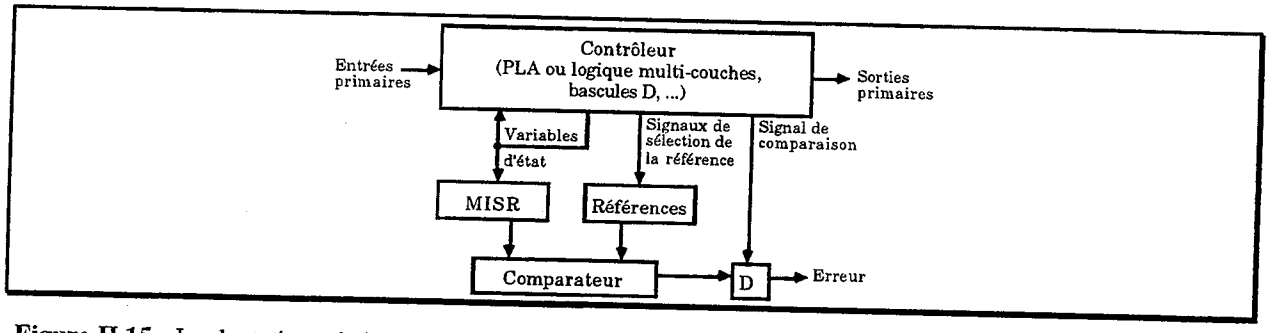

Figure II.15 - Implantation générale de la vérification de la signature d'après [Leve 90b].

Ainsi, outre la formalisation de la vérification de la signature appliquée aux contrôleurs câblés, R. Leveugle a proposé deux algorithmes de reconnaissance des SC-graphes, un programme d'aide au S-codage compact optimisé, et une technique de modification des graphes, reposant sur des règles systématiques, mais ne garantissant pas une réparation complète (intervention du concepteur parfois nécessaire).

Méthode proposée dans [Robi 92a].

Dès 1990, S. Robinson propose une méthode un peu différente de celle de R. Leveugle pour l'analyse de signature dans les contrôleurs câblés ([Robi 90], [Robi 92b], [Robi 92a]). Ses recherches aboutiront à un outil de reconnaissance des SC-graphes, de réparation systématique des NSC-graphes et de codage automatique des SC-graphes ([Robi 92a]), appelé I-Tool.

```
Pour tous les états Si du graphe d'origine G Faire
        Pour tous les états Sd successeurs de Si Faire
                Ajouter l'état Sd.i au nouveau graphe Gmax
                Pour tous les états Sp prédécesseurs de Si Faire
                        Si Si.p ∉ Gmax Alors
                                Ajouter l'état Si.p au nouveau graphe Gmax
                        Fin Si
                        Définir l'état Sd.i comme le successeur de Si.p
                Fin Pour
        Fin Pour
Fin Pour
```

Figure II.16 - Algorithme d'expansion maximale des graphes ([Robi 92a]).

La méthode utilisée pour la réparation compte deux étapes. La première étape consiste en une expansion maximale du graphe. Ceci signifie qu'à partir d'un graphe G, S. Robinson construit un graphe modifié Gmax tel que tout état S de G est scindé en Np états cousins dans Gmax, où Np est le nombre de prédécesseurs de S dans G. Démontrer que Gmax est S-codable est quasi immédiat, à partir du moment où on constate que les états cousins d'un même état S de G ne sont pas E-équivalents. S. Robinson a ainsi démontré dans [Robi 90] la faisabilité de la réparation pour tous les NSC-graphes. La figure II.16 montre l'algorithme d'expansion maximale développé par S. Robinson.

La seconde étape consiste en une fusion d'états dits compatibles vis à vis de la condition de S-codage. Il apparaît que les seules fusions possibles sont certaines fusions entre cousins. Pour réaliser ces fusions, une liste Lf de toutes les fusions envisageables est construite à l'aide des cousins obtenus. Pour une famille de Nc états cousins, on ajoutera ainsi à la liste Nc(Nc-1) couples. L'algorithme de fusion considère alors chacune des fusions potentielles les unes après les autres : si la fusion ne rend pas le nouveau graphe non S-codable, elle est effectuée, sinon elle est abandonnée. Ceci pose évidemment un problème d'ordonnancement. En effet, l'ordre des fusions

n'est pas indifférent : chaque fusion induit un certain nombre de contraintes sur les fusions réalisées par la suite. Le choix du meilleur ordre étant d'une complexité exponentielle, S. Robinson utilise une heuristique pour fixer l'ordre dans lequel il va envisager les fusions, et ainsi obtenir un optimum local de la fonction déterminant le nombre d'états du SC-graphe résultant. L'obtention du SC-graphe, équivalent au NSC-graphe d'origine, ayant le nombre minimum d'états ne peut être garanti. L'annexe II.1 fournit un exemple de réparation d'un NSC-graphe.

Le codage du SC-graphe s'appuie sur un processus de vérification un peu différent de celui utilisé dans [Leve 90b]. En effet, le code des états est défini de telle sorte que les signatures de référence puissent être dérivées directement de certains bits du dernier code d'état compacté. Ceci évite d'avoir à stocker les signatures de référence. La signature calculée en temps réel est considérée correcte si ses bits d'indice impair sont à 0, et ses bits d'indice pair sont égaux aux bits d'indice pair du dernier code d'état compacté. Par exemple, la signature de référence après un état codé 1011 est 0001 (les bits sont indicés de 0 à 3, de droite à gauche). Le premier avantage est de pouvoir effectuer une comparaison à chaque cycle d'horloge. Le second est de permettre un codage simple des SC-graphes en respectant la condition II.1. Le premier inconvénient est d'augmenter le nombre de bits nécessaires au codage des états par rapport au nombre minimum k défini au chapitre I. Le second est de créer une corrélation directe entre l'information compactée et les signatures de référence, ce qui peut avoir des conséquences sur la probabilité de masquage d'erreurs.

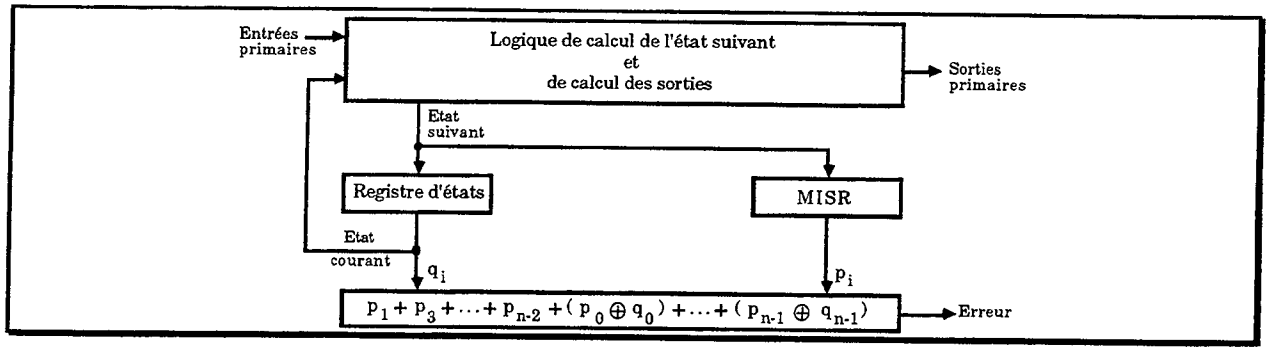

Figure II.17 - Implantation générale de la vérification de la signature d'après [Robi 92a].

La figure II.17 montre l'architecture proposée par S. Robinson pour ce processus de vérification de la signature.

Méthode proposée dans [Esch 92].

B. Escherman propose de profiter de dispositifs de test hors ligne intégrés pour effectuer la vérification du flot de contrôle. Il remarque que certaines architectures de test intégré utilisent un LFSR pour la génération de vecteurs de test hors ligne, ce même LFSR étant utilisé comme registre d'état en mode normal. L'idée est d'utiliser les propriétés de ce LFSR pour permettre la vérification de NSC-graphes sans scindement d'états (le graphe d'origine et celui implanté ont le même nombre d'états).

La méthode est la suivante. B. Escherman commence par une analyse du graphe pour repérer les situations rendant le graphe non S-codable. Plutôt que de scinder les états afin de rendre le graphe S-codable comme dans [Leve 90b] et [Robi 92a], B. Escherman propose de supprimer, ou couper, les transitions gênantes. Ensuite un S-codage compact non optimisé est effectué sur le

graphe modifié. L'implantation des transitions supprimées sera réalisée non plus par la logique de calcul d'état suivant, mais par le LFSR. Le mode de fonctionnement est donc le suivant.

Pour une transition normale, la logique de calcul d'état suivant calcule le code de l'état, qui est mémorisé par le LFSR qui fonctionne comme un registre d'état normal. Le code de l'état suivant est aussi fourni en entrée du MISR qui calcule la signature.

Par contre, pour une transition supprimée, la logique de calcul d'état suivant calcule une valeur d'ajustement fournie en entrée du MISR. Le LFSR fonctionne comme un générateur de vecteurs de test, et fournit en entrée des bascules du registre d'état (à savoir lui-même), le code de l'état suivant. B. Escherman obtient donc ainsi la propriété d'invariance des signatures, en fournissant au MISR une valeur d'ajustement lorsque le code de l'état ne permet pas de la respecter. La figure II.18 montre l'architecture résultant de ce mode de fonctionnement.

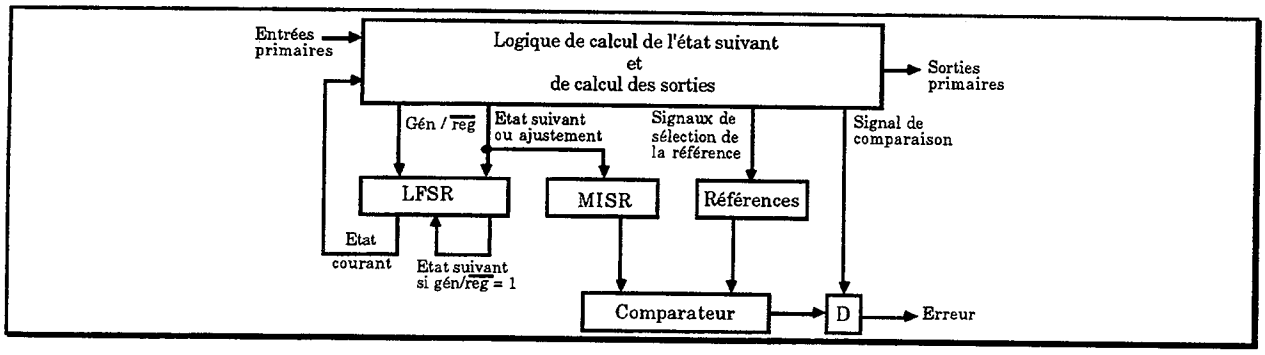

Figure II.18 - Implantation générale de la vérification de la signature d'après [Esch 92].

Une contrainte du S-codage est de s'assurer que si une transition Si → Sj est supprimée, le code de Sj est le successeur de Si dans la séquence de vecteurs générée par le LFSR. B. Escherman constate que le codage du graphe est difficile. De plus le S-codage obtenu n'est pas optimisé, et peut donc conduire à une augmentation non négligeable de la logique d'état suivant. Cette dernière doit de plus calculer les valeurs d'ajustement, et le signal permettant de faire basculer le LFSR du mode « registre d'état » dans le mode « générateur de vecteurs de test ». Toutefois, du fait du S-codage, le nombre d'ajustements est réduit, et le graphe d'états implanté comporte le même nombre d'états que le graphe d'origine. Malheureusement, B. Escherman ne fournit pas d'évaluation du coût matériel de la méthode. Le coût des deux autres méthodes [Leve 90] et [Robi 92a] est étudié au chapitre III.

II.1.5. Etude comparative

Les principales propriétés des méthodes précédemment étudiées sont résumées dans les tableaux II.1 et II.2.

Le tableau II.1 donne un résumé des méthodes de détection (au sens défini au paragraphe II.1.1.2). La colonne « Paragraphe » indique le paragraphe dans lequel la méthode a été citée. La colonne « Cible » indique son domaine d'application : la logique combinatoire (« log. »), les contrôleurs câblés (« câb. ») ou microprogrammés (« µpgr. »). La colonne « Modèle » indique le modèle de fautes ou d'erreurs utilisé. La signification des colonnes « Code », « Dispositif de détection », et « Détection » est triviale. Enfin, la colonne « Coût » indique le coût de la méthode du point de vue matériel (« mat. ») et temporel (« perf. »). Ces coûts sont soit dérivés de résultats trouvés dans la littérature scientifique, soit estimés en fonction des modifications de l'élément sous test dues à la méthode utilisée.

Tableau II.1 - Propriétés des méthodes de détection d'erreurs.

Méthode	Para-graphe	Cible	Modèle	Code	Dispositif de détection	Détection	Coût mat.	Coût perf.
[Fuji 84]	II.1.2.1	log.	collages	parité	vérificateur de parité	fautes simples	e	m
[De 92, sch. 2]	II.1.2.1	log.	collages	parité	vérificateur de parité	fautes simples	e	m
[Sogo 93a]	II.1.2.1	log.	ensemb. de f. prédéf.	parité	vérificateur de parité	fautes simples	m	m
[Sogo 93b, Göss 93]	II.1.2.1	log.	collages	parité	vérificateur de parité	fautes simples	m	m
[Toub 93]	II.1.2.1	log.	collages	parité	vérificateur de parité	fautes simples	m	m
[De 92, sch. 1]	II.1.2.1	log.	collages	Berger	vérificateur de Berger	fautes simples	e	e
[Osma 73]	II.1.2.2	câb.	collages	parité	3 vérificateurs de parité	fautes simples et qq. multiples	e	e
[Özgü 77]	II.1.2.2	câb.	collages	1 parmi n	vérificateur de parité	fautes s. sorties, état, entrées	e	f
[Micz 83]	II.1.2.2	câb.	collages	1 parmi n	vérificateur de parité	fautes simples et qq. multiples	m	f
[Jha 91, 93]	II.1.2.2	câb.	collages	m parmi n, Berger	vérificateur m parmi n, Berger	fautes simples et qq. multiples	e	f
[Pare 91]	II.1.2.2	câb.	collages, retards	m parmi n	moniteur	fautes simples	m	f
[Bolc 95a, 95b]	II.1.2.2	câb.	collages	distance de Hamming = 2	vérificateur de distance	fautes simples	m	m
[Beus 70]	II.1.2.3	µpgr.	erreurs	parité	vérificateur de parité	erreurs simples	m	f
[Cook 73]	II.1.2.3	µpgr.	collages	parité, 4 parmi 8	vér. parité et 4 parmi 8	fautes simples	e	m
[Wong 83]	II.1.2.3	µpgr.	erreurs	Berger, duplication	vérificateur de Berger	erreurs simples et qq. multiples	f	m
[Nico 90]	II.1.2.3	µpgr.	collages	m parmi n, parité	vér. m parmi n, parité	fautes simples	f	m

f : faible, m : moyen, e : élevé ; log. : logique, câb. : câblé, µpgr. : microprogrammé ; imp. : signature imposée, dér. : signature dérivée ; qq. : quelques.

Le tableau II.2 résume les principales caractéristiques des méthodes de vérification d'un flot de contrôle. La signification des différentes colonnes est la même que pour le tableau II.1. Toutefois, la colonne « Sign / code » précise le type de signature ou de codage utilisé. Il est à noter que certaines méthodes de vérification d'un flot de contrôle par codage sont en fait des cas particuliers de vérification par analyse de signature. Ainsi, l'utilisation de codes de convolution permet d'obtenir une signature considérée correcte si elle vérifie un invariant intrinsèque au code utilisé. De même, l'utilisation de clefs permet en fait de réaliser une analyse de signature où la signature n'est pas dérivée, comme c'est souvent le cas, mais imposée.

L'état de l'art présenté dans cette section est volontairement limité. Nous aurions pu citer encore [Diaz 74], [Diaz 75], et beaucoup d'autres, mais un chapitre n'y suffirait pas. Les méthodes exposées ici ont été choisies afin de donner un éventail assez général des techniques publiées, tout en insistant sur celles qui nous concernent directement en raison des travaux présentés au chapitre III. La section suivante est encore plus sélective.

Tableau II.2 - Propriétés des méthodes de vérification d'un flot de contrôle.

Méthode	Para-graphe	Cible	Modèle	Sign. / code	Dispositif de détection	Détection	Coût mat.	Coût perf.
[Holm 88]	II.1.3.1	câb.	erreurs de séquenc.	imp. (code de convolution)	vér. de code de convolution	chemins illégaux	e	e
[Holm 91]	II.1.3.1 II.1.3.2	câb., µpgr.	collages	imp. (code à circonv., clef)	comparateur	chemins illégaux dus aux collages	e	f
[Namj 82b, sch. 4]	II.1.3.2	µpgr.	erreurs	imposée (2 clefs)	comparateur	ch. illégaux et qq. erreurs d'opérat.	e	f
[Iyen 82 et 85]	II.1.3.2	µpgr.	collages	imposée (1ou 2 clefs)	moniteur	ch. incorrects dus aux collages	e	f
[Stae 85]	II.1.3.2	µpgr.	collages	imposée (2 clefs)	moniteur	ch. incorrects dus aux collages	e	f
[Guha 84 et 87]	II.1.3.2	µpgr.	collages	imposée (1 clefs)	moniteur	ch. incorrects dus aux collages	e	f
[Berr 90]	II.1.3.2	µpgr.	erreurs	produit de parité	vérificateur de parité	ch. incorrects, erreurs s. et m.	e	e
[Namj 82b, sch. 1]	II.1.4.1	µpgr.	erreurs	dérivée (somme modulo 2)	signature à 0	qq. ch. illégaux, erreurs d'opérat.	e	e
[Namj 82b, sch. 2]	II.1.4.1	µpgr.	erreurs	dérivée (somme modulo 2)	signature à 0	ch. illégaux, erreurs d'opérat.	m	e
[Srid 82]	II.1.4.1	µpgr.	erreurs	dérivée (somme modulo 2)	signature à 0	chemins illégaux	m	m
[Dura 83]	II.1.4.1	µpgr.	erreurs	dérivée (division polynomiale)	signature de référence	ch. incorrects, erreurs d'opérat.	e	m
[Tung 86]	II.1.4.1	µpgr.	erreurs	dérivée (division polynomiale)	signature de référence	ch. illégaux, erreurs d'opérat.	m	e
[Mahm 85]	II.1.4.1	µpgr.	erreurs	dérivée (division polynomiale)	signature de référence	ch. illégaux, erreurs d'opérat.	m	e
[Leve 90b]	II.1.4.2	câb.	erreurs	dérivée (division polynomiale)	signature de référence	chemins illégaux	f	m
[Robi 92a]	II.1.4.2	câb.	erreurs	dérivée (division polynomiale)	comparaison à l'état courant	chemins illégaux	f	m
[Esch 92]	II.1.4.2	câb.	erreurs	dérivée (division polynomiale)	signature de référence	chemins illégaux	m	m

f : faible, m : moyen, e : élevé ; câb. : câblé, µpgr. : microprogrammé ; imp. : signature imposée, dér. : signature dérivée ; qq. : quelques, ch. : chemins, *s.* : simples, m. : multiples, opérat. : opérations.

II.2. MÉTHODES DE TOLÉRANCE

Différentes méthodes ont été proposées pour la tolérance aux fautes dans les contrôleurs, toutefois le sujet a moins été étudié que la détection d'erreurs, ou que la tolérance aux fautes dans la partie opérative (voir respectivement la section II.1, et [Elli 90], [Russ 91], [Orto 92], [Hsu 93], [Hsu 94], [Tahi 95], ...). Notons tout de même que si [Elli 90] n'adresse que la tolérance aux fautes dans la partie opérative (essentiellement par utilisation de codes correcteurs d'erreurs), la partie contrôle est protégée. En effet, le séquencement des opérations est assuré par deux parties contrôle différentes et fonctionnant en parallèle. Il ne s'agit toutefois pas, à proprement parler, d'une technique de duplication : l'une de ces deux PC assure le séquencement des opérations sur les bits d'information des données, tandis que l'autre assure le séquencement des opérations sur les bits de redondance des données manipulées par la PO. Ceci permet la détection d'un grand nombre d'erreurs dans les deux blocs de contrôle.

Le principe qui nous intéresse le plus, pour la tolérance dans la partie contrôle, est le masquage à base de codes correcteurs d'erreurs. Toutefois, quelques propositions autres seront tout de même mentionnées comme la détection / recouvrement, l'utilisation d'informations passées, ou encore la tolérance par duplication et codage. Quelques travaux portant sur le masquage d'erreurs à base de TMR seront aussi cités.

II.2.1. Tolérance sans codes correcteurs d'erreurs

II.2.1.1. Détection / recouvrement

La détection / recouvrement est très souvent nommée mais généralement seul l'aspect détection est étudié. [Anto 90] propose une méthode améliorée du premier schéma exposé dans [Srid 82] de vérification d'un flot de contrôle par analyse de signature, et étudie explicitement la mise en place d'un dispositif de recouvrement associé.

La signature est le résultat de la compaction d'un champ particulier ajouté à chaque microinstruction. La valeur de ce champ est choisie aléatoirement, sauf pour les microinstructions correspondant aux noeuds destination des divergences ou aux noeuds origine des convergences dans le GFC. Cette valeur est choisie de telle sorte que la signature soit différente après chaque noeud destination d'une même divergence, et identique après chaque noeud origine d'une même convergence (ajustement). Un traitement particulier est aussi appliqué pour les boucles et les sous-programmes. La fonction de compaction est la division polynomiale. Les signatures de référence des chemins sont calculées de manière à être nulles. Pour les éclatements, un champ supplémentaire permet de coder les bits de condition, et ainsi d'assurer la vérification de la correction des chemins.

Le processus de recouvrement est le suivant : un point de recouvrement est placé au niveau de chaque point de contrôle de la signature. Si la signature en un point N_i est correcte, les principaux registres sont sauvegardés dans des registres de recouvrement, puis l'exécution du microprogramme continu. Si la signature est incorrecte, c'est qu'une erreur est survenue, et l'état de la machine est réinitialisé avec les valeurs sauvegardées dans les registres de recouvrement à l'avant dernier point de contrôle N_{i-1}. L'exécution du microprogramme est donc reprise au dernier point de contrôle traversé avec succès. Le recouvrement peut demander plusieurs cycles, et donc entraîner une certaine perte en performance, sans compter, évidemment, qu'il faut recommencer l'exécution des microinstructions comprises entre N_i et N_{i-1}. Le placement des points de recouvrement, et donc des points de contrôle, est fonction de l'action des microinstructions. Une analyse permet en effet de déterminer les microinstructions ayant une action irréversible, ou trop difficile à annuler. Un point de recouvrement doit impérativement être placé au niveau de toutes les microinstructions de ce type [Anto 90]. L'auteur précise que le nombre de microinstructions de ce type est généralement faible, ce qui induit une latence de détection, et un coût en performance en cas de recouvrement assez important. Il préconise donc le placement de points de recouvrement supplémentaires. Afin de tolérer les fautes dans les registres de recouvrement, [Anto 90] propose leur triplement associé à un vote majoritaire. Le coût matériel de ces dispositifs est très variable selon la taille du microprogramme vérifié (de 20% à plus de 100% de surcoût).

II.2.1.2. Duplication plus codage

Dans [Seng 77], la méthode proposée s'appuie sur la duplication et le code parité.

En premier lieu, A. Sen Gupta préconise la séparation des cônes logiques dans la logique de calcul d'état suivant. Ceci signifie que les fonctions de calcul de deux variables d'état différentes ne partagent pas de portes logiques. Ainsi, une faute simple ne peut produire qu'une erreur simple au niveau du registre d'état (sauf les courts-circuits entre cônes, par exemple).

Ensuite la logique de séquencement est dupliquée, et un bit de parité est ajouté à l'un des blocs obtenus. Soit M_1 et M_2 les deux blocs de logique de séquencement, Z le bloc de logique de sortie et M^* le bloc de prédiction du bit de parité. Seules les sorties de M_1 sont comparées au bit de

parité fourni par M^*. En l'absence de faute, le bloc Z reçoit en entrée les sorties de M_1, et M_2 calcule un état suivant à partir de l'état courant calculé dans M_1. M^* prédit la parité de l'état suivant en fonction de l'état courant de M_1.

Si une faute simple survient dans M_2, la machine continue à fonctionner normalement. Si la faute est transitoire, M_2 reprendra un séquencement normal puisque ses entrées sont les sorties de M_1. Si une faute simple se produit dans M_1, elle est détectée grâce au bit généré par M^*. Le bloc Z reçoit alors en entrée les sorties de M_2. M_2 calcule l'état suivant à partir de son propre état courant. M_1 reçoit en entrée non plus son propre état courant mais l'état courant de M_2. M^* reçoit en entrée l'état courant de M_2. Si une faute simple se produit dans M^*, le fonctionnement est le même que si M_1 est fautive.

Cette architecture permet de tolérer les fautes simples dans la logique de séquencement (logique d'état suivant plus registre d'état). L'auteur n'a pas évalué le coût matériel et temporel de la méthode. Toutefois, le coût matériel d'une telle implantation est a priori compris entre celui de la duplication et celui de la triplication. Le coût temporel, quant à lui, semble nettement supérieur à celui de la triplication (traversée de multiplexeurs plus vérification de la parité).

II.2.1.3. Modification de la structure de la logique

Dans [Bogl 95], le masquage d'erreur est obtenu dans la logique combinatoire multi-couches par l'introduction d'une redondance minimale au cours du processus d'optimisation, en utilisant le degré de liberté autorisé par les phi-Booléens. A la base, le masquage est obtenu en utilisant les propriétés de masquage inhérentes à certaines portes logiques (ici les portes NOR), et à la duplication de certaines sous-fonctions.

Cette méthode est intéressante pour deux raisons. D'abord parce qu'elle agit à un grain très fin (niveau portes), en utilisant les possibilités d'optimisations offertes par les phi-Booléens, ce qui peut laisser augurer un coût réduit du masquage. Ensuite parce que le masquage utilisé empêche une erreur de propager à travers plus d'une couche de logique. Ainsi, le circuit combinatoire peut tolérer plusieurs fautes simultanées, à condition quelles ne se produisent pas dans le même voisinage logique (c'est-à-dire n'affectent pas les mêmes sous-fonctions).

Le gros inconvénient est de ne pas être testable. En effet, le test d'un tel bloc combinatoire semble difficile à réaliser puisque la logique masque systématiquement les erreurs. Connaître les capacités de tolérance du bloc obtenu après fabrication est alors un problème particulièrement complexe. En l'état actuel des choses, la méthode pourrait donc avant tout être utilisée pour améliorer le rendement, mais l'auteur n'a pas encore fait d'études dans ce sens. Pour être réellement intéressante, il serait aussi important d'étendre la méthode à un ensemble plus large de portes logiques, afin d'éviter le surcoût inhérent à l'utilisation exclusive de portes NOR.

II.2.1.4. Informations passées

Dans [Tohm 71], une méthode originale propose de masquer les erreurs simples en définissant des états composites incluant des informations passées. Ainsi, les états et les entrées sont stockés un cycle d'horloge supplémentaire. Un état composite est représenté par l'état courant, l'état précédent (l'état mémorisé), et les entrées précédentes (entrées mémorisées). L'état suivant est calculé à partir de cet état composite et des entrées courantes.

Soient Sr_i et Sr_j les états précédents de deux états composites, X_i et X_j leurs entrées précédentes, et S_i et S_j leurs états courants. Y. Thoma montre que la correction des erreurs simples

est possible si la distance de Hamming $d(Sr_i, Sr_j) \geq 2$ ou $d(S_i, S_j) \geq 2$ lorsque $d(X_i, X_j) = 0$, et si $d(Sr_i, Sr_j) + d(S_i, S_j) \geq 2$ lorsque $d(X_i, X_j) = 1$. L'algorithme de codage des états prend donc en compte ces conditions.

La méthode a les mêmes capacités de masquage que celle de [Arms 61] présentée au paragraphe II.2.2.2. Toutefois, le nombre de variables ajoutées aux fonctions de calcul de l'état suivant risque d'augmenter sensiblement leur complexité. Y. Thoma ne donne pas d'évaluation précise du coût matériel de sa méthode. Il indique simplement qu'elle utilise moins de bascules que celle de [Arms 61] lorsque le nombre de variables d'état est faible. Il semble, toutefois, ne pas tenir compte des bascules nécessaires à la mémorisation des entrées précédentes dans cette évaluation.

II.2.1.5. Masquage à base de TMR

Différents travaux ont été réalisés pour les architectures à base de TMR. Ainsi, [Osma 73] propose une architecture TMR avec partage de logique, un peu comme pour la duplication (paragraphe II.1.2.2, où sont définies les notations utilisées ici). Pour ce faire, plutôt que de tripler la logique de calcul d'une fonction f à n variables, la fonction f et son complément sont implantées sous la forme $f_n = z_1g_1+...+z_mg_m$, ce qui permet de partager entre ces deux fonctions les modules G_i ($1 \leq i \leq m$). Un troisième bloc implanté normalement calcule la fonction f. Une porte majorité effectue le vote entre les deux exemplaires de f et le complément de f inversé. Cette architecture peut être facilement étendue à plusieurs fonctions différentes, avec partage des modules G_i entre elles. Le circuit résultant tolère les fautes simples (au plus un module fautif à la fois parmi les modules G_i, Z_i et le troisième bloc).

D'autres études ont été effectuées pour l'approche TMR. Nous pouvons par exemple citer l'amélioration de la testabilité par une modification de la structure du TMR dans [Stro 93] (sans ajout de « scan path »), ou encore la réalisation d'un bus de vote majoritaire permettant une implantation peu coûteuse et facilement reconfigurable d'un vote majoritaire entre N fonctions. Ainsi ce bus est particulièrement bien adapté aux architectures NMR et DNMR (« Dynamic N-Modular Redundant » : redondance modulaire N-aire à reconfiguration dynamique).

II.2.2. Masquage à base de codes correcteurs d'erreurs

Nous avons vu différentes possibilités pour la tolérance aux fautes. Toutefois, les méthodes les plus prometteuses du point de vue du coût matériel sont certainement celles basées sur l'utilisation de codes correcteurs d'erreurs, et en particulier celles basées sur le codage des états du contrôleur avec un tel code ([Arms 61]).

II.2.2.1. Méthodes autres que [Arms 61]

[Mand 72] montre qu'il est possible d'utiliser un compteur modulo T pour détecter, ou masquer, les erreurs dans la logique de séquencement d'un contrôleur, dont le graphe d'état ne comporte que des cycles d'une longueur multiple de T (longueur en nombre d'états traversés). Ainsi, il est possible d'utiliser un compteur modulo 2 pour détecter les erreurs avec un code de parité. Pour la correction d'erreurs, un code correcteur d'erreurs simples peut être utilisé. Les bits de redondance constituent alors le code des états du compteur, et les bits d'information le code des états de la machine. Lorsqu'un tel code est utilisé, T est fonction du nombre d'états de la machine et du code correcteur d'erreurs choisi. Ce type d'implantation d'une machine M et d'un compteur M' nécessite généralement la modification du graphe de contrôle de M, de manière à faire en sorte

que ses cycles aient une longueur multiple de T. De plus, le codage des états de M doit correspondre à la séquence des bits de redondance générée par le compteur M'. En pratique, ceci peut s'avérer difficile à réaliser.

II.2.2.2. Méthode de [Arms 61] et dérivées

En 1961, D. B. Armstrong et D. K. Ray-Chaudhuri proposent, dans [Arms 61] et [RayC 61], une architecture tolérant les fautes à base de codes correcteurs d'erreurs, comme une alternative, moins coûteuse, au TMR. [Arms 61] concerne plus l'aspect architectural, et [RayC 61] formalise l'aspect codage. Ces travaux vont être à l'origine de différentes études publiées principalement dans [Russ 65], [Fran 66], [Reed 70], [Meye 71], [Lars 72], [Seng 81], et [Wang 84]. Nous ne détaillerons ici que certaines d'entre elles pour introduire certaines notions indispensables à la compréhension du chapitre IV.

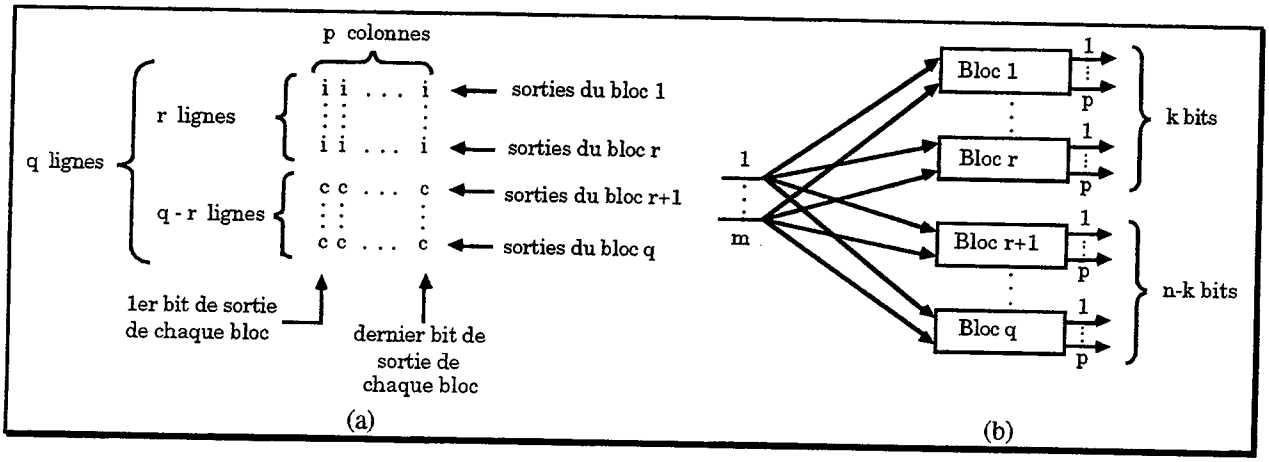

Figure II.19 - Organisation des bits de contrôle (a), et de la logique implantée (b), dans [Arms 61].

Dans [Arms 61], D. B. Armstrong constate que l'architecture TMR utilise 2*k bits de redondance pour k bits d'information. Il propose donc d'utiliser un code correcteur d'erreurs défini dans [RayC 61] pour réduire le nombre de bits de redondance, afin de réduire le coût des dispositifs de masquage d'erreurs, voire d'augmenter la fiabilité du système obtenu.

Etant donné qu'à l'époque sa réflexion s'appuie sur des technologies à base de composants discrets, il vise avant tout la modularité, afin d'améliorer la maintenabilité par le changement de la carte sur laquelle est implanté un module défectueux. Ce souci se traduit par l'architecture suivante : la logique de calcul de l'état suivant est divisée en r blocs électriquement indépendants (sans logique partagée). Chaque bloc comporte p bits de sortie. Si k est le nombre de fonctions booléennes calculées par la logique combinatoire, p*r = k. Appelons n le nombre total de bits (bits d'information plus bits de contrôle). La logique de calcul des n-k bits de redondance est implantée sous la forme de q-r blocs. q est le nombre total de blocs logiques. Chacun des q-r blocs de redondance calcule p bits de contrôle. L'organisation de la redondance est un peu la même que celle utilisée dans [De 92], mais cette fois avec un code correcteur d'erreurs plutôt qu'un code parité : l'ensemble des premiers bits de sortie des q-r blocs de redondance correspond aux q-r bits de contrôle du mot formé par les premiers bits des r blocs de calcul des bits d'information (Fig. II.19) ; l'ensemble des seconds bits de sortie des q-r blocs de redondance correspond aux q-r bits de contrôle du mot formé par les seconds bits des r blocs de calcul des bits d'information ; et ainsi de suite ... Armstrong propose d'implanter le bloc de correction des codes d'état en sortie du registre d'état, et en entrée des q blocs de calcul de l'état suivant et de la logique de sortie. La même technique peut être appliquée à la logique de calcul des sorties.

D. B. Armstrong remarque que l'utilisation d'un code de Hamming pour coder les p colonnes de la matrice définie par la figure II.19 est excessive. En effet, un code de Hamming ([Hamm 50]) permettrait de corriger un bit erroné dans chaque colonne, même si les bits erronés ne sont pas éléments de la même ligne. Or, du fait de l'implantation sous forme de r blocs indépendants, une faute simple ne peut provoquer que des bits erronés dans une seule et même ligne. D. B. Armstrong et D. K. Ray-Chaudhuri ont donc mis a profit cette propriété pour définir un code correcteur d'erreurs nécessitant généralement moins de bits de contrôle. Ainsi, si r = 3 et p = 2, le nombre de bits de contrôle nécessaire avec le code de D. K. Ray-Chaudhuri est 4 (q = 5), tandis qu'il est égal à 6 avec un code de Hamming (q = 6).

[Russ 65], [Fran 66], et [Meye 71] reprennent l'idée de Armstrong, c'est-à-dire coder les états de la machine avec un code correcteur d'erreurs. Mais plutôt que d'utiliser un bloc de correction séparé, ils « distribuent » la correction au niveau des fonctions de calcul d'état suivant. Ceci signifie que les fonctions de calcul de l'état suivant sont modifiées de manière à réaliser la correction en même temps que le calcul de l'état suivant. De plus, ils ne groupent pas les sorties de la logique d'état suivant comme dans [Arms 61], mais utilisent une architecture où p=1, c'est-à-dire où les cônes logiques des fonctions de calcul de l'état suivant sont séparés (cette séparation n'est pas explicitement mentionnée dans [Fran 66]).

[Russ 65] applique cette architecture aux compteurs. Il est le premier à indiquer qu'un codage assurant une distance de Hamming supérieure ou égale à 3 entre les codes des états du compteur permet de tolérer les erreurs simples. A la base, le principe est très simple : une erreur simple modifie un code d'état en créant un code erroné à distance 1 du code correct. Etant donné que tous les codes corrects sont à distance 3 les uns des autres, il suffit de substituer le code d'état correct le plus proche, du point de vue de la distance de Hamming, au code d'état erroné pour corriger l'erreur.

[Fran 66] étend ce principe aux machines séquentielles en général en expliquant qu'il suffit de considérer les codes à distance 1 d'un code d'état correct, comme des états associés à cet état. Les fonctions de calcul de l'état suivant vont alors non seulement calculer les transitions Si → Sj, où Si et Sj sont deux états de la machine, mais aussi les transitions Si_e → Sj où Si_e représente les états associés à Si (c'est-à-dire les codes à distance 1 du code de l'état Si).

Toutefois, c'est dans [Meye 71] qu'est formalisée cette notion d'états associés. J. F. Meyer explique, du point de vue théorique, comment réaliser une machine (R,t)-tolérante. R représente les mots du code correcteur d'erreurs choisis pour coder les états de la machine, et t la capacité de tolérance de la machine : t = 1 signifie que la machine tolère les erreurs simples, t = 2 signifie que la machine tolère les erreurs simples et doubles, etc ... Pour qu'une machine M, de m états, soit (R,t)-tolérante, R doit être un ensemble de m mots d'un code correcteur d'erreurs, la distance minimale entre deux mots étant égale à 2t+1. Pour rendre M (R,t)-tolérante, il modifie, après codage, le graphe de contrôle de M en associant à chaque état Si de M un ensemble d'états associés As(Si). Les états éléments de As(Si) ont les mêmes successeurs que Si, et pas de prédécesseur (seul Si ∈ As(Si) a des prédécesseurs). As(Si) est défini par deux conditions :

- L'ensemble des codes des états éléments de As(Si) doit contenir tous les codes à une distance inférieure ou égale à t du code de Si,
- Soit n le nombre de variables d'états, la fonction de $\{0,1\}^n$ dans R, qui fait correspondre au code d'un état associé un code élément de R, doit être bijective.

La figure II.20 montre le graphe de contrôle d'une machine M de 4 états et le graphe, après codage, résultant de la modification appliquée par Meyer pour rendre M (R,1)-tolérante (correction des erreurs simples). R est l'ensemble {01101, 00000, 10110, 11011}, représentant les codes distance 3 des 4 états de M. La machine implantée est la machine spécifiée par le graphe modifié.

[Reed 70] et [Lars 72] proposent une architecture mixte. Plutôt que de modifier les fonctions de calcul de l'état suivant comme dans [Russ 65], [Fran 66] et [Meye 71], ou de confier la correction à un unique bloc séparé comme dans [Arms 61], les bits du code d'état sont corrigés individuellement en entrée de chaque cône logique de calcul de l'état suivant où ils sont utilisés. Ceci signifie que la logique de correction d'un bit du code d'état sera répliquée autant de fois qu'il y a de cônes qui utilisent ce bit pour leurs calculs. Le code correcteur d'erreurs utilisé est le code Reed-Muller modifié, défini dans [Reed 70]. L'intérêt de ce code est de pouvoir effectuer une correction individuelle des bits d'information pour un coût matériel très faible (2 portes XOR 2 entrées plus une porte majorité). L'inconvénient est de doubler systématiquement le nombre de bits du registre d'état : le nombre de bits de redondance est égal au nombre de bits d'information. Dans [Lars 72], la même idée est appliquée au TMR, c'est-à-dire que les portes majorité corrigeant les bits des registres d'état sont distribuées en entrée de chaque bloc répliqué.

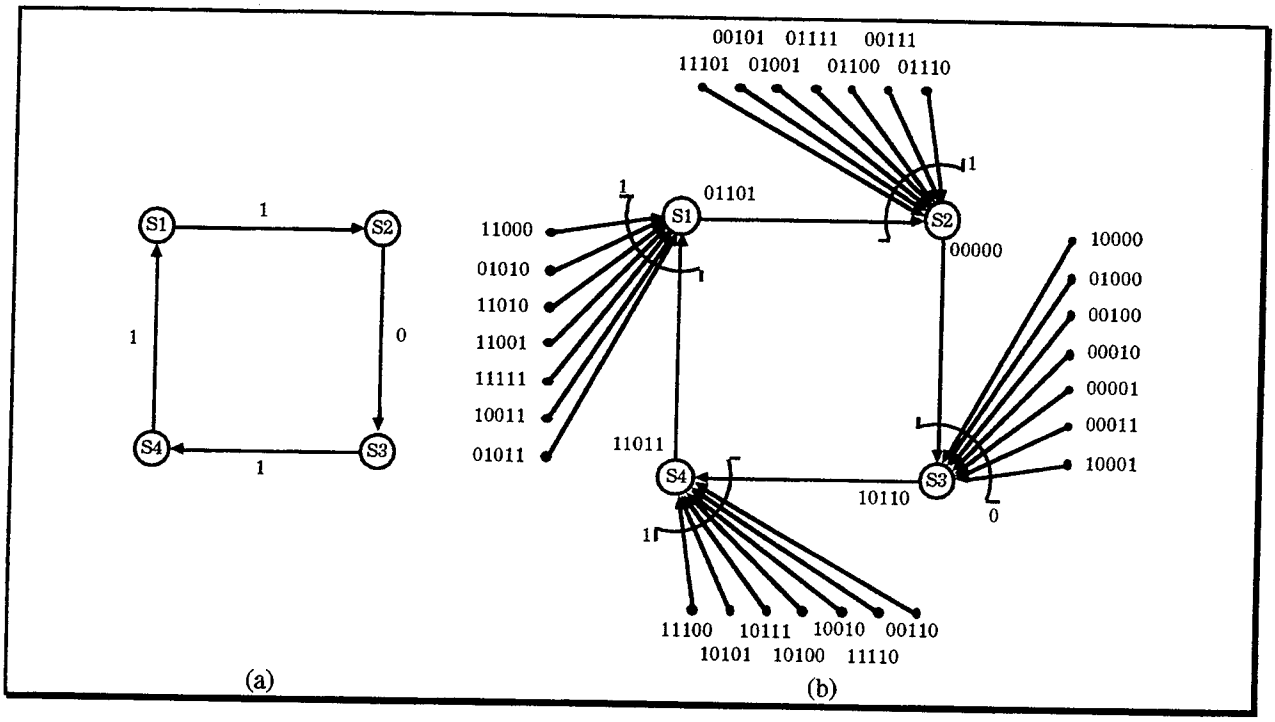

Figure II.20 - Réalisation d'une machine (R,1)-tolérante [Meye 71] : graphe d'origine (a), graphe modifié (b).

[Seng 81] reprend les idées exprimées dans [Arms 61] et [Wang 84] celles exprimées dans [Reed 70].

La plupart de ces articles proposent leurs techniques de tolérance, mais ne donnent aucune indication quant à leur coût matériel. D. B. Armstrong indique que son architecture est potentiellement moins coûteuse que le TMR uniquement en mentionnant le fait que le nombre de bascules utilisées est moindre. De plus, pour ce qui est de la fiabilité, il utilise la modularité de son architecture pour prendre en compte la possibilité de changer un module défectueux (ce qui n'est plus possible avec les technologies d'aujourd'hui). Seul [Russ 65] indique que pour 125 compteurs implantés, le surcoût dû au masquage d'erreurs par distribution de la correction est de 400 à 500%. Nous verrons toutefois qu'une architecture proche de celle présentée dans [Arms 61] et [Seng 81] conduit à des coûts matériels bien moindres.

II.2.3. Etude comparative

Le tableau II.3 donne un résumé des caractéristiques des méthodes de tolérance aux fautes et de masquage d'erreurs abordées précédemment. La signification des colonnes est semblable à celle des tableaux II.1 et II.2 du paragraphe II.1.5. Ainsi, la colonne « Paragraphe » indique le paragraphe dans lequel la méthode a été citée. La colonne « Cible » indique son domaine d'application : la logique combinatoire (« log. »), les contrôleurs câblés (« câb. ») ou microprogrammés (« µpgr. »). La colonne « Modèle » indique si elle est basée sur un modèle de fautes ou d'erreurs. La colonne « Architecture » rappelle l'architecture utilisée pour l'obtention de la tolérance. La signification des colonnes « Dispositif de correction », et « Correction » est trivial. Enfin, la colonne « Coût » indique le coût de la méthode du point de vue matériel (« mat. ») et temporel (« perf. »). Comme pour les méthodes de détection, ces coûts sont soit dérivés de résultats trouvés dans la littérature scientifique, soit estimés en fonction des modifications nécessaires pour rendre l'élément tolérant.

Tableau II.3 - Propriétés des méthodes de tolérance aux fautes.

Méthode	Para-graphe	Cible	Modèle	Architecture	Dispositif de correction	Correction	Coût	
							mat.	perf.
[Anto 90]	II.2.1.1	µpgr.	collages	VFC par signat. + recouvrement	recouvrement	fautes simples	e	e
[Seng 77]	II.2.1.2	câb.	erreurs	duplication + parité	basculement d'un module à l'autre	erreurs simples	m	e
[Bogl 95]	II.2.1.3	log.	collages	dupl. niv. porte, modif struct. log.	intrinsèque à la logique	fautes multiples	f	m
[Tohm 71]	II.2.1.4	câb.	erreurs	informations passées	fonctions log. modifiées	erreurs simples	e	e
[Osma 77]	II.2.1.5	câb.	collages	décomposition logique, TMR	portes majorités	fautes simples	e	m
[Mand 72]	II.2.2.1	câb.	erreurs	code correcteur d'erreurs	moniteur	erreurs simples	e	m
[Arms 61, RayC 61, Seng 81]	II.2.2.2	câb.	erreurs	code correcteur d'erreurs	décodeur	erreurs simples	f	e
[Russ 65, Fran 66, Meye 71]	II.2.2.2	câb.	erreurs	code correcteur d'erreurs	fonctions log. modifiées	erreurs simples	e	e
[Reed 70, Lars 72, Wang 84]	II.2.2.2	câb.	erreurs	code correcteur d'erreurs	décodeurs distribués	erreurs simples	m	e

f : faible, m : moyen, e : élevé ; log. : logique(s), câb. : câblé, µpgr. : microprogrammé ; qq. : quelques.

En fait les techniques développées dans [Arms 61] et [Seng 81] sont les plus prometteuses. Le chapitre IV étudie une architecture qui en est directement dérivée. La méthode évoquée dans [Bogl 95] est aussi très intéressante et est certainement une méthode d'avenir parce qu'en agissant au niveau portes, elle travaille à un grain très fin, ce qui permet de minimiser le coût et maximiser l'efficacité. Toutefois, nous avons déjà mentionné les problèmes de testabilité qu'elle engendre, et qui pour l'instant sont loin d'être résolus.

CHAPITRE III - SYNTHÈSE DE CONTRÔLEURS AVEC DÉTECTION D'ERREURS

Nous avons vu dans le chapitre précédent différentes techniques de détection d'erreurs et de vérification d'un flot de contrôle. Rares sont celles qui sont automatisées, et les articles publiés présentent généralement peu de résultats concrets. Nous nous sommes donc intéressés à l'automatisation de la synthèse d'architectures pour la vérification d'un flot de contrôle dans les contrôleurs. Nous rappelons que les notions de base de la vérification d'un flot de contrôle sont définies au paragraphe II.1.1.3. Les notions de base de l'analyse de signature sont présentées au paragraphe II.1.4.1. Enfin, la formalisation de la vérification d'un flot de contrôle par analyse de signature appliquée aux contrôleurs câblés est succinctement abordée au paragraphe II.1.4.3.

Avant d'entrer dans le détail des travaux effectués dans ce domaine, afin d'être le plus précis possible, nous introduisons au paragraphe III.1 une nouvelle classification des différentes méthodes de vérification d'un flot de contrôle par analyse de signature appliquées aux contrôleurs câblés.

III.1. AJUSTEMENTS ET RÉFÉRENCES IMPLICITES OU EXPLICITES

Nous introduisons ici une nouvelle classification des techniques de vérification d'un flot de contrôle par analyse de signature, en différenciant les techniques à base d'ajustements implicites ou explicites, et les techniques à base de signatures de référence implicites ou explicites.

Définition III.1 : Ajustements implicites ou explicites

*On appelle méthode de vérification d'un flot de contrôle par analyse de signature à base d'**ajustements implicites**, une méthode utilisant la technique du S-codage pour garantir la propriété d'invariance II.1.*

*Une méthode utilisant des valeurs d'ajustement (définition II.9) pour garantir la propriété d'invariance II.1, est dite à base d'**ajustements explicites**.*

Définition III.2 : Références implicites ou explicites

*On appelle méthode de vérification d'un flot de contrôle par analyse de signature à base de **références implicites**, une méthode utilisant l'existence d'une propriété propre aux signatures des chemins légaux.*

*Une méthode stockant ou re-calculant les signatures de référence aux points de comparaison est dite à base de **références explicites**.*

La méthode proposée dans [Leve 90b], et présentée au paragraphe II.1.4.3, est une méthode à base d'ajustements implicites et de références explicites. On parle d'ajustements implicites parce que le S-codage permet de s'affranchir de l'utilisation de valeurs d'ajustement proprement dites. En fait, chaque code d'état est un ajustement en soit de la signature, qui permet d'obtenir la propriété d'invariance de la signature dans le GFC. On parle de références explicites parce que la valeur des signatures de référence est explicitement recalculée.

La méthode présentée dans [Robi 92a] est une méthode à base d'ajustements et de références implicites. On parle d'ajustements implicites parce que, comme pour [Leve 90b], le S-codage permet d'éviter l'utilisation d'ajustements proprement dits. Il s'agit de plus d'une méthode à base de références implicites parce que la signature calculée en temps réel est vérifiée, non pas en la comparant à une valeur de référence précalculée, mais en vérifiant une propriété particulière des signatures de référence, à savoir ici une propriété de corrélation entre le code de l'état courant et la signature après cet état.

Enfin, la méthode décrite dans [Esch 92] est une solution mixte à base d'ajustements implicites et explicites, avec des références explicites. Il s'agit d'une solution mixte parce qu'une valeur d'ajustement explicite est générée par la logique de calcul de l'état suivant lorsqu'une transition « coupée » est rencontrée (voir paragraphe II.1.4.3), mais elle s'appuie aussi sur un ajustement implicite de la signature du fait du S-codage des états du graphe.

Dans la suite, nous présentons le flot de synthèse automatisé pour une méthode à base d'ajustements implicites et de références explicites selon les principes énoncés dans [Leve 90b]. Ensuite la section III.3 présente le flot de synthèse d'une architecture à base d'ajustements et de références explicites. Une étude détaillée de la probabilité de masquage d'erreurs est présentée dans la section III.4 pour une méthode à base d'ajustements implicites / références implicites, une à base d'ajustements implicites / références explicites et une à base d'ajustements explicites / références explicites. Enfin, la section III.5 présente les résultats d'implantations pour ces trois méthodes en termes de surface après placement / routage, performances, et taux de couverture de fautes, ainsi qu'une comparaison avec la duplication de la logique de séquencement (Fig. I.10).

III.2. AJUSTEMENTS IMPLICITES / RÉFÉRENCES EXPLICITES

Le principe utilisé par les implantations à base d'ajustements implicites et de références explicites est celui présenté dans [Leve 90b]. Comme cela a déjà été mentionné, les codes des états sont utilisés pour le calcul de la signature. La signature calculée en temps réel est comparée à des valeurs de référence en différents points de contrôle. L'ajustement implicite de la signature est

obtenu par S-codage (paragraphe II.1.4.3). Une modification du graphe d'états peut être nécessaire afin de le rendre S-codable.

III.2.1. Architecture

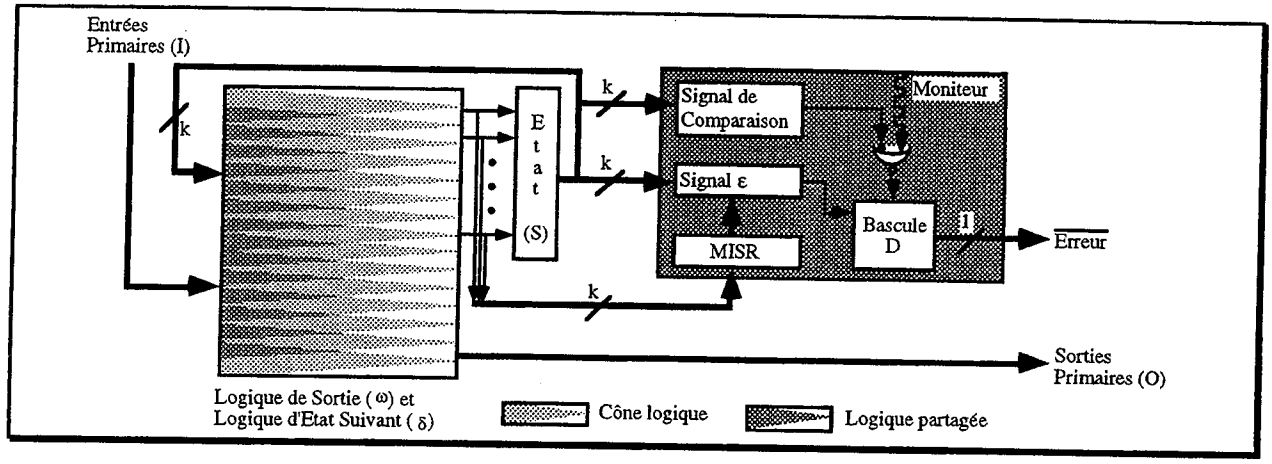

Figure III.1 - Architecture CFC NoAjs.

L'architecture adoptée, appelée CFC NoAjs (« Control Flow Checking, No Adjustment » : vérification d'un flot de contrôle sans ajustement), est décrite par la figure III.1. Dans cette architecture, les entrées du MISR sont connectées aux sorties de la logique de calcul de l'état suivant, comme le registre d'états. Ceci implique une synchronisation différente de celle proposée dans [Leve 90b], où les entrées du MISR sont connectées aux sorties du registre d'états (Fig. II.15), ces deux registres travaillant en opposition de phase. Dans l'architecture décrite ici, le registre d'états et le MISR sont synchronisés sur le même front d'horloge (Fig. III.2). L'intérêt est de permettre une implantation plus simple (du point de vue du flot de synthèse) des dispositifs de vérification de la signature. Ce mode de synchronisation implique que le MISR doit être initialisé avec la valeur de la signature de référence après l'état d'initialisation du contrôleur.

Le signal de comparaison est calculé à partir de l'état courant, c'est-à-dire des sorties du registre d'états. Ainsi, si dans un graphe de contrôle, deux états, disons S4 et S11, sont choisis comme points de contrôle, alors la fonction Booléenne implantée pour le signal de comparaison est égale à 1 lorsque le registre d'état contient le code de l'état S4 ou le code de l'état S11, et est égale à 0 sinon. Aucun φ-Booléen n'est accepté pour cette fonction très simple, à l'exception des codes inutilisés (combinaisons binaires non affectées à un état du contrôleur).

Les signaux, dans la figure II.15, de sélection et de calcul de la signature de référence, ainsi que le comparateur, sont ici implantés en un seul et même bloc logique. Celui-ci génère le signal ε en fonction des sorties du MISR et du registre d'états. ε est connecté en entrée de la bascule de mémorisation du signal d'erreur. Sa fonction logique est construite de la manière suivante. Si l'état courant est un point de contrôle, le signal est à 1 si la signature courante (la signature calculée en temps réel) est identique à la signature de référence en ce point. Si elle est différente, le signal est à 0. Il en est de même pour tous les points de contrôle. Par contre, l'utilisation d'un signal de comparaison permet d'introduire des φ-Booléens pour tous les états qui ne sont pas des points de contrôle : étant donné que le signal de comparaison est inactif, la valeur d'ε importe peu.

A titre d'exemple, supposons que S4 et S11 soient des points de contrôle de code 01 et 11. Soit 00 la signature de référence après S4 et 10 la signature de référence après S11. La fonction de calcul d'ε sera égale à 1 si le code de l'état courant est 01 et que la signature courante est 00. Si le code d'état courant est 01, mais que la signature courante est 01, 10 ou 11, la fonction est égale à

0. De même pour le code d'état 11. Si la signature courante est 10, la fonction est égale à 1. Sinon, si la signature courante est 00, 01 ou 11, la fonction est égale à 0. Pour les codes d'états 00 et 01 qui ne sont pas des points de comparaison (voire qui ne sont pas des codes d'états tout court), la fonction de calcul d'ε est φ, quelque soit la valeur de la signature courante.

Le nombre de points de contrôle étant généralement réduit ([Leve 90b],[Leve 90d]), ceci permet d'obtenir une fonction de calcul d'ε très peu coûteuse en terme de nombre de portes logiques. Il est à noter que du fait de la fonction logique d'ε, le signal d'erreur en sortie de la bascule est actif à 0 (signal $\overline{\text{Erreur}}$).

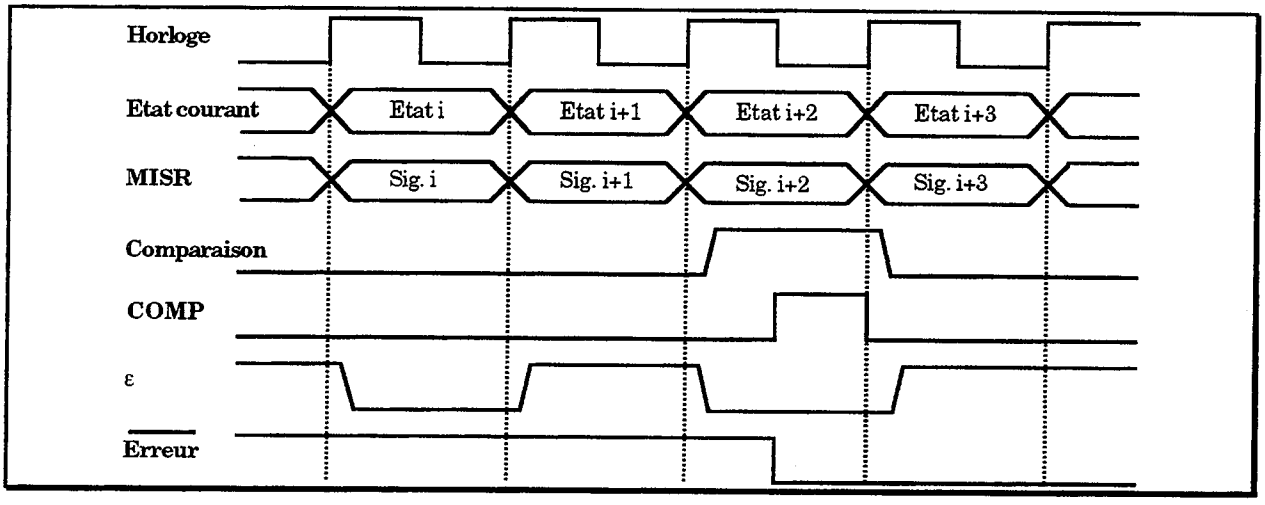

Figure III.2 - Synchronisation pour la vérification de la signature sans Ajs.

La bascule de mémorisation du signal d'erreur est une bascule D travaillant en opposition de phase par rapport au MISR et au registre d'états. Aussi, le signal COMP connecté sur son entrée de synchronisation est le résultat d'un Non OU (NOR), entre le signal de comparaison inversé et le signal d'horloge (pour être exact, cela dépend du type des bascules utilisées). Cela signifie que le signal de comparaison et ε doivent être calculés en moins d'une demi-période, ce qui est généralement le cas du fait de la simplicité des fonctions de calcul de ces deux signaux. L'utilisation d'une bascule de mémorisation travaillant en phase avec le MISR provoquerait des comparaisons parasites.

Dans le cas où le signal de comparaison ou/et le signal ε nécessitent plus d'une demi-période pour être calculés, on peut, par exemple, utiliser deux bascules fonctionnant en phase avec le registre d'états et le MISR (le signal d'horloge est directement connecté sur leur entrée de synchronisation). L'une reçoit en entrée le signal de comparaison, l'autre le signal ε. Le signal d'erreur est alors le résultat d'un OU entre la sortie inversée de la bascule du signal de comparaison et la sortie non inversée de la bascule d'ε.

III.2.2. Modification du graphe

L'une des étapes de la synthèse lorsqu'on veut implanter une architecture comme celle décrite par la figure III.1 est la modification du graphe de contrôle pour le rendre S-codable. Comme nous l'avons déjà dit, S. Robinson a montré que tout NSC-graphe est réparable ([Robi 90]), et a automatisé cette réparation avec la mise au point du logiciel I-Tool ([Robi 92a]).

Plutôt que de procéder comme lui à une expansion maximale du graphe puis à des fusions successives pour la réparation des graphes, nous avons cherché, de notre coté, à procéder à la

réparation par touches successives, sans passer par une phase d'expansion, en ne scindant que les états nécessitant de l'être.

III.2.2.1. Définitions

Nous avons vu au chapitre II les notions de signature avant et après un noeud (définition II.8), les notions de E, S et C-équivalence entre états (définitions II.13, II.14 et II.15), ainsi que la notion d'états cousins (définition II.16). Nous introduisons ici quelques notions supplémentaires nécessaires à la compréhension de notre discours.

- **Définition III.3 : Propagation de la signature**

Soit une signature s associée à un arc A. Cette signature est dite ***propagée*** *à un arc B si le respect de la propriété d'invariance II.1 impose que la signature associée à l'arc B soit également s.*

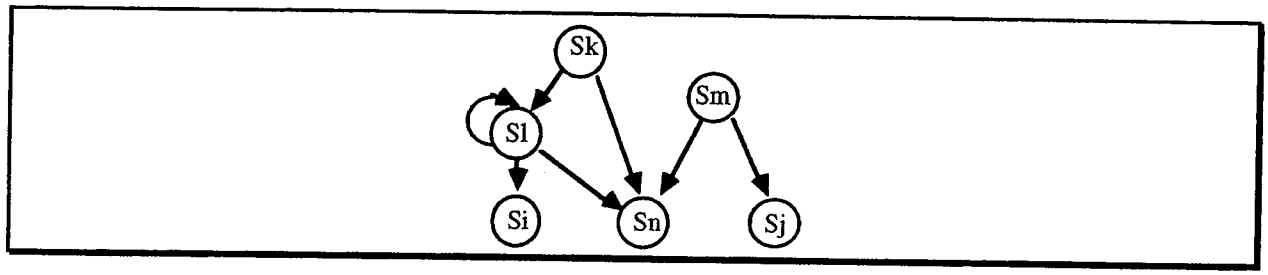

Figure III.3 - Propagation de la signature avant deux états E-équivalents Si et Sj.

La figure III.3 donne un exemple de propagation de la signature pour les arcs entrants de deux états E-équivalents. Dans cet exemple, les états Si et Sj sont E-équivalents, bien qu'ils n'aient pas de prédécesseurs commun, du fait de la propagation de la signature entre l'arc Sl→Si et l'arc Sm→Sj, par l'intermédiaire des états Sl, Sn et Sm. En effet, du fait de la propriété d'invariance forcée, les états Sl et Sm ayant un successeur commun Sn, la signature est la même après ces deux états. Ce qui signifie que la signature avant Si et Sj est identique.

Lorsqu'on considèrera, dans ce document, la propagation de la signature avant deux états E-équivalents, par abus de langage, on parlera de **propagation en E-équivalence**, comme c'est le cas pour la figure III.3. Lorsqu'on s'intéressera à la propagation de la signature après deux états S-équivalents, on parlera de **propagation de la signature en S-équivalence**.

Lors de la réparation d'un graphe, le but est de fractionner les classes d'états S-équivalents et / ou E-équivalents, jusqu'a obtenir des classes d'états C-équivalents (appelées aussi classes de C-équivalence) singletons. La notion de propagation de la signature en E et S-équivalence est donc intéressante, car elle permet de caractériser les états à scinder pour supprimer la E-équivalence de deux états (fractionnement d'une classe de E-équivalence), ou pour supprimer la S-équivalence entre deux états (fractionnement d'une classe de S-équivalence). Si les deux états en question sont C-équivalents, cela revient à supprimer leur C-équivalence (les classes de C-équivalence sont l'intersection des classes de S et E-équivalence).

Afin de visualiser plus facilement cette notion de propagation en E et S-équivalence, nous introduisons une nouvelle représentation qui met en évidence les classes d'équivalence dans un graphe.

- **Définition III.4 : Graphe de représentation d'une classe de E-équivalence**

Soit G le graphe de contrôle d'une machine M. On appelle ***graphe de représentation d'une classe de E-équivalence*** G_e, *un graphe dont l'ensemble des noeuds contient :*

- les états E-équivalents S_e,

- les états S_p prédécesseurs des états S_e dans G, ayant au moins deux états successeurs éléments de l'ensemble des états S_e.

L'ensemble des arcs de G_e contient tous les arcs reliant les états S_p aux états S_e dans G. Les états S_e sont représentés par des cercles, et les états S_p sont représentés par des rectangles. Si un état est à la fois élément de l'ensemble des états S_e et de l'ensemble des états S_p, il sera représenté deux fois (une fois par un cercle, et une fois par un rectangle).

- **Définition III.5 : Graphe de représentation d'une classe de S-équivalence**

Soit G le graphe de contrôle d'une machine M. On appelle ***graphe de représentation d'une classe de S-équivalence*** G_s, *un graphe dont l'ensemble des noeuds contient :*

- les états S-équivalents S_e,

- les états S_s successeurs des états S_e dans G, ayant au moins deux états prédécesseurs éléments de l'ensemble des états S_e.

L'ensemble des arcs de G_s contient tous les arcs reliant les états S_e aux états S_s dans G. Les états S_e sont représentés par des cercles, et les états S_s sont représentés par des rectangles. Si un état est à la fois élément de l'ensemble des états S_e et de l'ensemble des états S_s, il sera représenté deux fois (une fois par un cercle, et une fois par un rectangle).

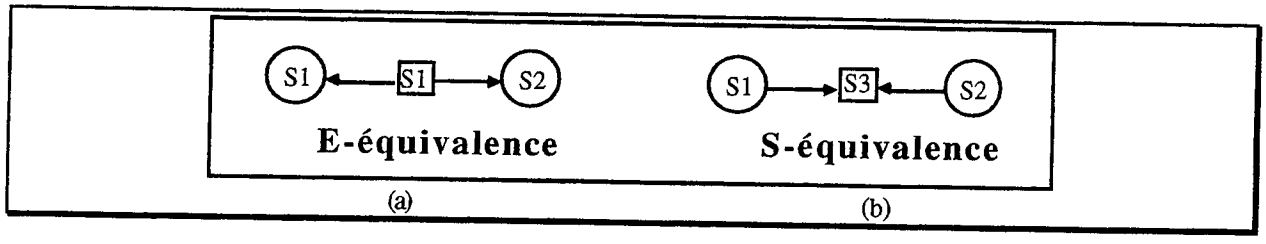

Figure III.4 - Représentation de la E-équivalence (a), et de la S-équivalence (b), entre états.

La figure III.4a donne un exemple de représentation de deux états E-équivalents S1 et S2, du fait d'un même prédécesseur commun S1. Notons que S1 est un état rebouclé puisqu'il est son propre successeur. La figure III.4b donne un exemple de représentation de deux états S-équivalents S1 et S2, du fait d'un même successeur commun S3.

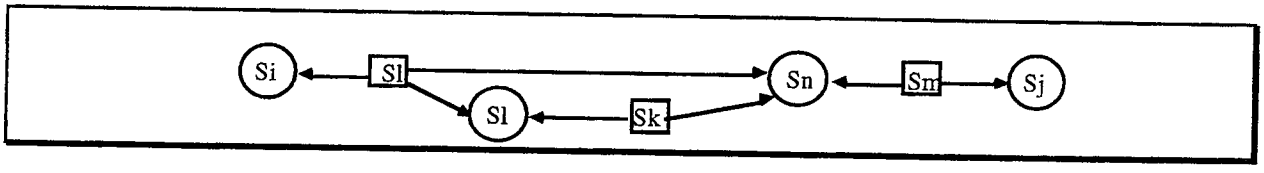

Figure III.5 - Représentation de la E-équivalence pour l'exemple de la figure III.3.

La figure III.5 donne la représentation de la E-équivalence des états Si, Sl, Sn et Sj de l'exemple de la figure III.3. La propagation de la signature en E-équivalence entre les états Si et Sj par l'intermédiaire des états Sl, Sn et Sm apparaît immédiatement. Ceci nous permet d'introduire

les notions de E-équivalence locale, de E-équivalence distante et de manière duale, de S-équivalence locale, et de S-équivalence distante.

- **Définition III.6 : E-équivalence locale**

*La **E-équivalence** entre deux états est dite **locale,** s'il existe au moins un prédécesseur commun aux deux états concernés.*

Dans le cas contraire, la E-équivalence est liée à une **propagation à distance** de la signature : il n'existe pas de prédécesseur commun aux deux états.

- **Définition III.7 : S-équivalence locale**

*La **S-équivalence** entre deux états est dite **locale,** s'il existe au moins un successeur commun aux deux états concernés.*

Ainsi, dans l'exemple de la figure III.5, les états Si et Sj sont des états E-équivalents du fait d'une propagation à distance de la signature, tandis que les états Si et Sl sont des états E-équivalents localement.

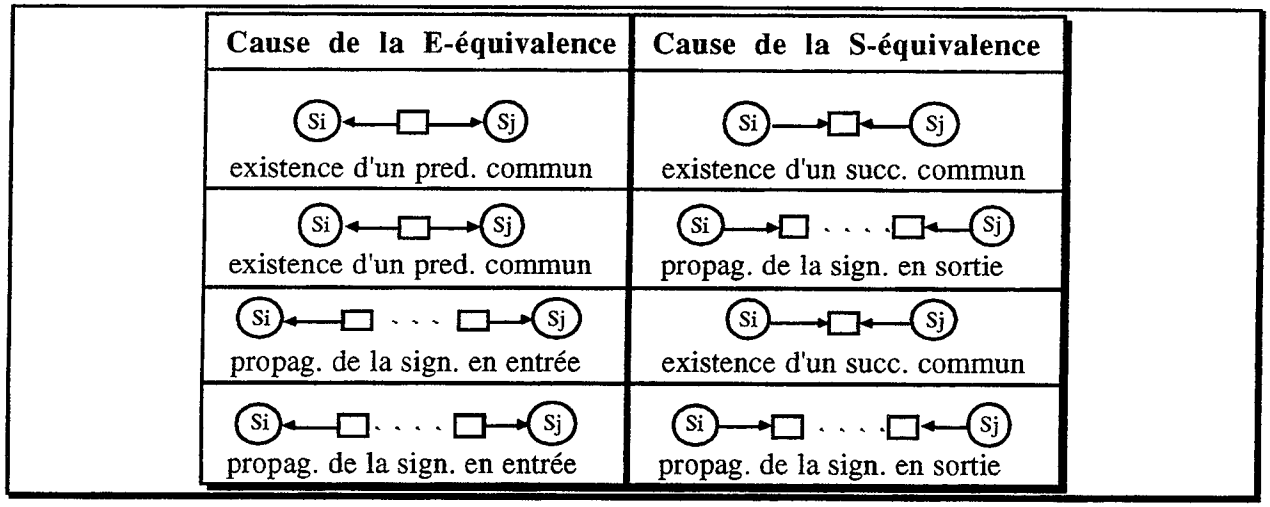

Figure III.6 - Les quatre causes de C-équivalence entre deux états Si et Sj.

Ceci nous permet de différencier quatre cas de C-équivalence, résumés par la figure III.6. Parmi ces quatre cas, on définit en particulier la notion de **C-équivalence locale** qui correspond au premier cas du tableau de la figure III.6.

- **Définition III.8 : C-équivalence locale**

*La **C-équivalence** entre deux états est dite **locale,** s'il existe au moins un prédécesseur et un successeur commun aux deux états concernés.*

La nouvelle représentation introduit la notion de chaîne de propagation de la signature.

- **Définition III.9 : Chaîne de propagation**

*On appelle **chaînes de propagation de la signature** entre deux états Si et Sj dans un graphe $G_{e/s}$ de représentation d'une classe de E ou de S-équivalence, les chaînes, au sens de*

la théorie des graphes, reliant Si et Sj dans $G_{e/s}$ où tous les arcs sont remplacés par des arêtes, et les états par des sommets.

Dans la figure III.5, par exemple, il existe deux chaînes de propagation de la signature entre Si et Sj : la chaîne formée des états {Si, Sl, Sn, Sm, Sj} et celle formée des états {Si, Sl, Sl, Sk, Sn, Sm, Sj}.

Supprimer la relation de C-équivalence entre deux états revient donc à scinder l'un des états de chacune des chaînes de propagation de la signature en E-équivalence entre ces deux états, ou de chacune des chaînes de propagation de la signature en S-équivalence entre ces deux états. Dans la suite, on utilisera l'expression « **couper une chaîne de propagation** » pour désigner l'action de scinder un des états de la chaîne.

- **Théorème III.1 :**

Si on considère uniquement les scindements conduisant à un graphe fonctionnellement et temporellement équivalent au graphe d'origine (partitionnement de l'ensemble des prédécesseurs de l'état scindé, paragraphe II.1.4.3, figure II.14), alors couper une chaîne de propagation en E-équivalence entre deux états C-équivalents Si et Sj en scindant Si, Sj ou l'un des prédécesseurs de Si ou Sj ne supprime pas la relation de C-équivalence.

Démonstration :

Supposons que l'on scinde Si (ou Sj), alors Si est remplacé par deux états cousins Si' et Si'', chacun ayant les mêmes successeurs que Si et comme prédécesseurs, un sous-ensemble des prédécesseurs de Si. L'un des deux états cousins ainsi créé va donc « hériter » du prédécesseur de Si élément de la chaîne de propagation de la signature entre Si et Sj. Les relations de E et de S-équivalences sont donc recréées entre Sj et l'un des états cousins de Si. Le raisonnement est le même si Si(ou Sj) est scindé en plus de deux états. La relation de C-équivalence n'a donc été que déplacée.

Maintenant supposons que l'on scinde Sx, un prédécesseur de Si (ou Sj) élément de la chaîne de propagation en E-équivalence. On va créer Sx' et Sx'' ayant les mêmes successeurs que Sx et pour prédécesseurs un sous-ensemble des prédécesseurs de Sx. Etant donné que Sx' et Sx'' ont les mêmes successeurs que Sx, au lieu de couper la chaîne existant, on en a créé une supplémentaire, et la relation de C-équivalence n'est pas rompue.

III.2.2.2. Présentation générale de la réparation effectuée

L'algorithme de réparation des NSC-graphes implanté comporte deux phases : la première réalise l'élimination des C-équivalences locales, et la seconde s'occupe de la suppression des C-équivalences distantes.

L'algorithme utilisé pour la détermination des classes de C-équivalence est l'un des deux algorithmes présentés dans [Leve 90b]. Sa complexité est $O(N^2)$, ou N est le nombre d'états.

Dans la suite, lors du scindement d'un état, la notation utilisée sera la suivante : si on scinde un état Si ayant pour prédécesseurs les états Sv, Sw, Sx, Sy et Sz, alors on appellera Si.xyz l'état cousin de Si ayant pour prédécesseurs les états Sx, Sy et Sz, et on appellera Si.vw celui ayant comme prédécesseurs les états Sv et Sw. Toutefois, par commodité d'écriture, l'état ayant le plus

de prédécesseurs sera souvent noté Si, en particulier lorsque seuls deux états cousins sont créés, l'un des deux n'héritant que d'un seul prédécesseur.

III.2.2.3. Suppression des C-équivalences locales

Dans la suite, nous appellerons **classe de C-équivalence locale**, un ensemble d'états C-équivalents localement, c'est-à-dire ayant tous, dans leur ensemble, au moins un prédécesseur et un successeur communs. Un même état peut-être élément de plusieurs classes de C-équivalences locales, comme c'est le cas pour l'état Sj dans la figure III.7a. En effet, dans le cas de la figure III.7a, Si et Sk ne sont pas C-équivalents localement, car ils n'ont pas de prédécesseur commun.

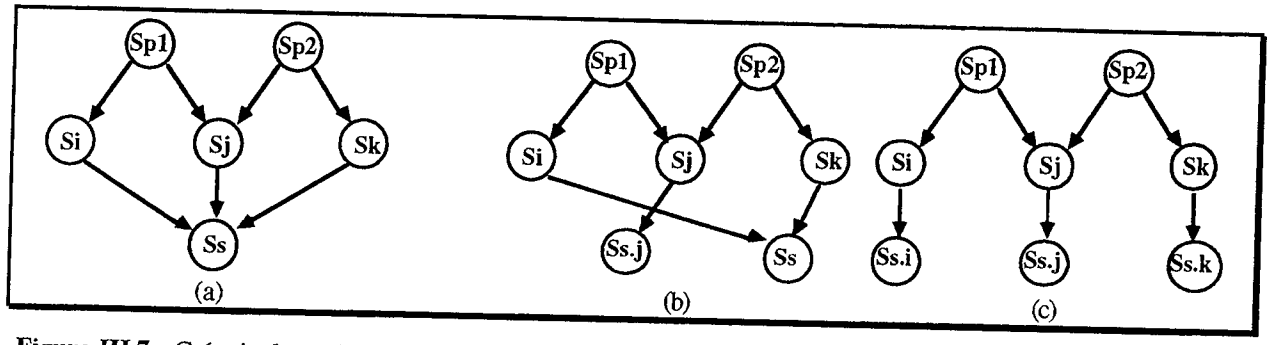

Figure III.7 - C-équivalence locale des paires d'états (Si, Sj) et (Sj, Sk) (a), résolution 1 (b), résolution 2 (c).

L'enjeu de l'algorithme implanté est de résoudre uniquement les C-équivalences locales, et de ne créer que des états indispensables à la résolution de ces C-équivalences. Ce type de C-équivalences est en effet plus simple à résoudre du fait du théorème III.1. Ainsi, soient n états S1, ..., Sn, tous C-équivalents localement 2 à 2, c'est-à-dire ayant tous en commun au moins un prédécesseur et au moins un successeur. Soient Ssc les successeurs communs à tous ces états, alors le théorème III.1 nous assure que la résolution de cette C-équivalence locale ne peut se faire que par la création des états Ssc.1, ..., Ssc.n.

Toutefois, lorsqu'on envisage non plus une seule classe de C-équivalence locale, mais plusieurs, le choix des états à scinder est toujours unique, mais la façon de les scinder, c'est-à-dire le nombre d'états cousins créés et la partition choisie des états prédécesseurs de l'état scindé peut avoir une grande influence sur le nombre d'états nécessaires à la résolution des C-équivalences locales considérées.

La figure III.7a montre un exemple de deux classes de C-équivalence locale : (Si, Sj) et (Sj, Sk). Une résolution séparée de ces deux C-équivalences locales donne la solution de la figure III.7c. Pourtant, Si et Sk n'étant pas C-équivalents localement, rien n'oblige la création des états Ss.i et Ss.k. En fait, un état Ss.ik suffit (Fig. III.7b). Notons qu'avec la solution de la figure III.7b, Si et Sk restent C-équivalents (mais à distance), contrairement à ce qui résulte de la solution de la figure III.7c. On pourrait donc penser que la solution de la figure III.7c est préférable. Toutefois, supprimer la C-équivalence distante (Si, Sk) peut aussi se faire en scindant Sj en deux états Sj.p1 et Sj.p2 dans la figure III.7b. Sans informations sur les autres C-équivalences distantes du graphe, rien ne permet d'affirmer que l'une des deux solutions est préférable. L'avantage de la résolution des C-équivalences locales de la figure III.7b par rapport à celle de la figure III.7c est donc double : en premier lieu elle permet de laisser un degré de liberté supplémentaire à la résolution des C-équivalences distantes de la phase II, et en second lieu, elle ne crée que l'état indispensable à la résolution des C-équivalences locales (Si, Sj) et (Sj, Sk).

```
Recherche des C-équivalences locales
Créer un tableaux Tab de liste-d'incompatibilité vides (une liste vide par état du graphe)
Pour tout couple d'états (S1, S2) de la liste des C-équivalences locales, Faire
        Ajouter S2 à la liste-d'incompatibilité de S1 dans le tableau Tab
        Ajouter S1 à la liste-d'incompatibilité de S2 dans le tableau Tab
Fin Pour
Construction du tableau Pred des prédécesseurs de chaque état
Pour chaque état Si du Graphe Faire
        Construire le tableau TabSi des incompatibilités entre les prédécesseurs de Si à l'aide de Tab et Pred
        Si l'état Si possède des prédécesseurs incompatibles Alors
                Trier TabSi dans l'ordre décroissant de la taille des liste-d'incompatibilité
                Partition-des-prédécesseurs-de-Si = liste-vide d'ensembles
                Pour chaque prédécesseur Sp de Si dans l'ordre de TabSi Faire
                        Sous-ensemble-courant = premier de la Partition-des-prédécesseurs-de-Si
                        Tant que (Sous-ensemble-courant ≠ ensemble-vide) et (Sp non placé) Faire
                                Si Sp est compatible avec les états du Sous-ensemble-courant Alors
                                        Placer Sp dans Sous-ensemble-courant
                                Fin Si
                                Sous-ensemble-courant = suivant de la Partition-des-prédécesseurs-de-Si
                        Fin Tant que
                        Si Sp n'a pu être placé Alors
                                On crée un nouveau sous-ensemble contenant uniquement Sp
                                Ajouter ce sous-ensemble à la liste d'ensembles Partition-des-prédécesseurs-de-Si
                        Fin Si
                Fin Pour
                Ajouter la Partition-des-prédécesseurs-de-Si au tableau des Partitions-des-prédécesseurs
        Fin Si
Fin Pour
Modifier le graphe en fonction du tableau des partitions
```

Figure III.8 - Algorithme de suppression des C-équivalences locales.

Le principe de l'algorithme implanté est donc le suivant. On commence par créer une table dite **d'incompatibilité** qui fournit, pour chaque état du graphe, la liste des états qui lui sont C-équivalents localement. Dans l'exemple de la figure III.7a, la liste d'incompatibilité de Sj est {Si, Sk}. Celle de Si est {Sj} et celle de Sk est {Sj}. Ensuite, pour chaque état successeur d'états incompatibles, on partitionne l'ensemble de ses états prédécesseurs de manière à ce que :

- deux états incompatibles ne soient pas dans le même sous-ensemble,
- le nombre de sous-ensembles constitués par la partition soit minimal.

Enfin, lorsque les partitions ont été constituées pour tous les états du graphe, on modifie le graphe en fonction de ces partitions : un état donné est scindé en fonction de la partition de l'ensemble de ses prédécesseurs (un état cousin est créé pour chaque sous-ensemble de prédécesseurs). La figure III.8 résume le traitement effectué.

Trouver la partition des prédécesseurs d'un état comportant le nombre minimum de sous-ensembles est un problème NP-complet. En effet, supposons que l'on construise un graphe non orienté, tel que chaque état prédécesseur est représenté par un sommet, et que deux sommets sont reliés par une arête si les états correspondants sont incompatibles. Il s'agit alors d'un problème de la théorie des graphes bien connu, à savoir trouver la partition des sommets d'un graphe non orienté telle que chaque sous-ensemble de la partition forme un stable dans le graphe, et que le nombre de ces stables est minimum. L'heuristique choisie ici pour résoudre ce problème est la suivante : placer en premier les états les plus contraignants, c'est-à-dire considérer en premier dans le graphe non orienté les sommets de degré maximum. Cette heuristique se montre en effet assez efficace tant que le nombre de sommets avec le même degré n'est pas trop important. Elle offre, de plus, une faible complexité algorithmique.

Tel que l'algorithme est implanté, la modification du graphe de contrôle se fait en une fois à la fin. Une solution, algorithmiquement plus coûteuse, mais pouvant paraître plus efficace, serait

de modifier, lors de l'obtention de la partition des prédécesseurs de chaque état, le graphe et le tableau Tab des incompatibilités. Toutefois, nous montrons ici que la modification du graphe à la fin de l'algorithme n'a pas de conséquences sur la structure du graphe de contrôle obtenu.

- **Théorème III.2 :**

La modification du graphe à la fin de l'algorithme de la figure III.8 est sans conséquence sur la structure du graphe obtenu, par rapport à une modification lors de chaque obtention de la partition des prédécesseurs d'un état.

Démonstration :

1)- Deux états cousins sont par construction compatibles. Mais s'ils sont éléments d'un ensemble de prédécesseurs comportant des états incompatibles, lors de la partition de cet ensemble, ils doivent impérativement être placés dans le même sous-ensemble (deux états cousins doivent avoir les mêmes successeurs). Si la modification du graphe était faite au fur et à mesure, cette remarque aurait deux conséquences :

- les états cousins doivent être traités tous en même temps lors de la construction des sous-ensembles d'une partition,
- leur rang dans le tableau trié des listes d'incompatibilités TabSi (cf. figure III.8) doit être le même que celui de leur état cousin d'origine s'il n'avait pas été scindé, si on veut respecter l'heuristique de partitionnement choisie.

Ceci signifie que même si la modification du graphe se faisait au fur et à mesure, les états cousins devraient de toutes façon être considérés comme un seul et même état non scindé lors de la construction des partitions.

2)- Par construction, le scindement d'un état ne peut pas rendre incompatibles des états compatibles.

3)- Si deux états incompatibles, ou plus, deviennent compatibles du fait d'un scindement, cela signifie que la C-équivalence locale a été supprimée entre ces états : tous leurs successeurs communs ont déjà été parcourus et traités par l'algorithme. Aucun de ces états n'ayant plus de successeurs communs, la modification du tableau Tab, dû à la suppression de leur C-équivalence locale, n'a pas de conséquences sur le contenu des tableaux TabSi des états Si restant à traiter.

Ces trois points démontrent que le scindement d'un état n'a pas de conséquence sur les partitions restant à construire, et donc que le graphe de contrôle obtenu est le même, que la modification se fasse au fur et à mesure ou à la fin du traitement.

La complexité de l'algorithme de la figure III.8 est la suivante. Si on appelle N le nombre d'états du graphe de contrôle, la complexité de la rechercher des C-équivalences locales est $O(N^2)$. La complexité de la création du tableau Tab est $O(N^2)$. La complexité de la construction du tableau Pred est $O(N^2)$. La complexité de la construction du tableau TabSi est N*(N/Taille d'un mot mémoire) car les listes d'incompatibilités sont construites à l'aide d'un ET logique entre des champs de bits. Dans la plupart des cas, ceci correspond à un algorithme en O(N), mais dans l'absolu la complexité est $O(N^2)$. La complexité du tri de TabSi est $O(N*\log_2(N))$. La complexité du placement de Sp est O(N) : on peut avoir N sous-ensembles, et un sous-ensemble peut contenir au plus N éléments, mais le total de la somme des cardinalités des sous-ensembles est au plus N. La complexité de l'algorithme complet est donc $O(N^3)$.

III.2.2.4. Suppression des C-équivalences non locales

L'algorithme proposé ici permet de supprimer les C-équivalences distantes. Après la suppression des C-équivalences locales par la phase I, il reste 4 types de C-équivalences à supprimer. Nous les appellerons dans la suite C-équivalences de Type II, Type III, Type III Bis et Type IV. Une C-équivalence de Type II correspond à deux états E-équivalents localement et S-équivalents à distance (Fig. III.6, cas 2). Une C-équivalence de Type III correspond à deux états E-équivalents à distance et S-équivalents localement (Fig. III.6, cas 3). Enfin, une C-équivalence de Type IV correspond à deux états E-équivalents et S-équivalents à distance (Fig. III.6, cas 4). Le Type III Bis est un cas particulier du Type III où les deux états C-équivalents sont cousins.

Principe de base de l'algorithme.

Nous présentons ici une méthode dite des états instables. Le principe est d'élire et créer l'état cousin jugé le plus susceptible de permettre l'obtention d'un graphe S-codable comportant peu d'états. La liste des classes de C-équivalence est alors mise à jours et l'élection est renouvelée tant que la cardinalité d'une classe est supérieure à 1.

Le principe de l'élection est le suivant. Chaque classe de E-équivalence ou de S-équivalence est considérée comme une structure dans laquelle chaque état possède une énergie potentielle dépendante des liaisons engagées avec ses plus proches voisins. On obtient la susceptibilité de la structure à se rompre au voisinage de chaque état en déterminant les potentiels des différents états : l'état le plus susceptible à engendrer une rupture sera celui dont le potentiel est le plus élevé.

Pour une classe de E-équivalence (resp. de S-équivalence) donnée, on calcule le potentiel en E-équivalence (resp. en S-équivalence) de tous les états de la classe, si et seulement si cette classe possède deux états C-équivalents. On procède en deux tours à l'élection du couple d'états (Sx, Sy), vérifiant la relation de succession $\delta(Sx)=Sy$, et correspondant au maximum du produit des potentiels de Sx en S-équivalence et de Sy en E-équivalence. Cette élection détermine la création de l'état Sy.x.

La fonction de calcul du potentiel de chaque état est la suivante : un état sera d'autant moins stable qu'il possédera de nombreuses "liaisons", c'est-à-dire de nombreuses E-équivalences locales (ou S-équivalences locales) avec les autres états de sa classe de E-équivalence (resp. de S-équivalence). Pour un état Sx, on définit les fonctions énergies potentielles Pe pour une classe de E-équivalence et Ps pour une classe de S-équivalence par :

Pe (Sx) = nombre de E-équivalences locales dans lequel intervient Sx,

Ps (Sx) = nombre de S-équivalences locales dans lequel intervient Sx.

Durant le premier tour de l'élection, la liste des états possédant le potentiel maximum de leur classe est construite (au minimum un état par classe de E et S-équivalence). Parmi ces états, sont retenus pour le second tour les deux états possédant les potentiels les plus élevés en E-équivalence, et les deux états possédant les potentiels les plus élevés en S-équivalence, soit au total quatre états (e1, e2, s1, s2). Pour trancher les cas d'égalité lors de la sélection de ces quatre états et lors du choix final du second tour, on définit que :

- les états déjà élus à une élection précédente ne peuvent en aucun cas être élus à nouveau (remplacement d'un état par lui-même),
- un état C-équivalent est prioritaire devant un état « normal » pour une élection en S-équivalence et le contraire dans le cas de la E-équivalence (en raison du théorème III.1).

Si des ex-aequo existent encore, les deux états sélectionnés seront choisis arbitrairement. Une fois les deux états en E-équivalence et les deux états en S-équivalence sélectionnés, le choix final du second tour est réalisé de la façon suivante :

- si la relation de succession est vérifiée entre un état sélectionné en S-équivalence et un sélectionné en E-équivalence, on détermine le potentiel global **P** = Ps * Pe. Dans ce cas la paire d'états élue sera celle pour laquelle **P** est maximum,
- si aucun couple ne satisfait la relation de succession, alors on choisit un état pour lequel Ps est maximum et on choisit parmi ses successeurs celui pour lequel Pe est maximum. Par défaut, les potentiels non évalués valent zéro. Si ces états ont tous un Pe nul, un tirage arbitraire est effectué.

Une fois le couple élu, l'état cousin correspondant est créé. Les classes de C-équivalence sont modifiées en conséquence et une nouvelle élection est effectuée si nécessaire. La figure III.9 décrit l'organigramme général de la méthode.

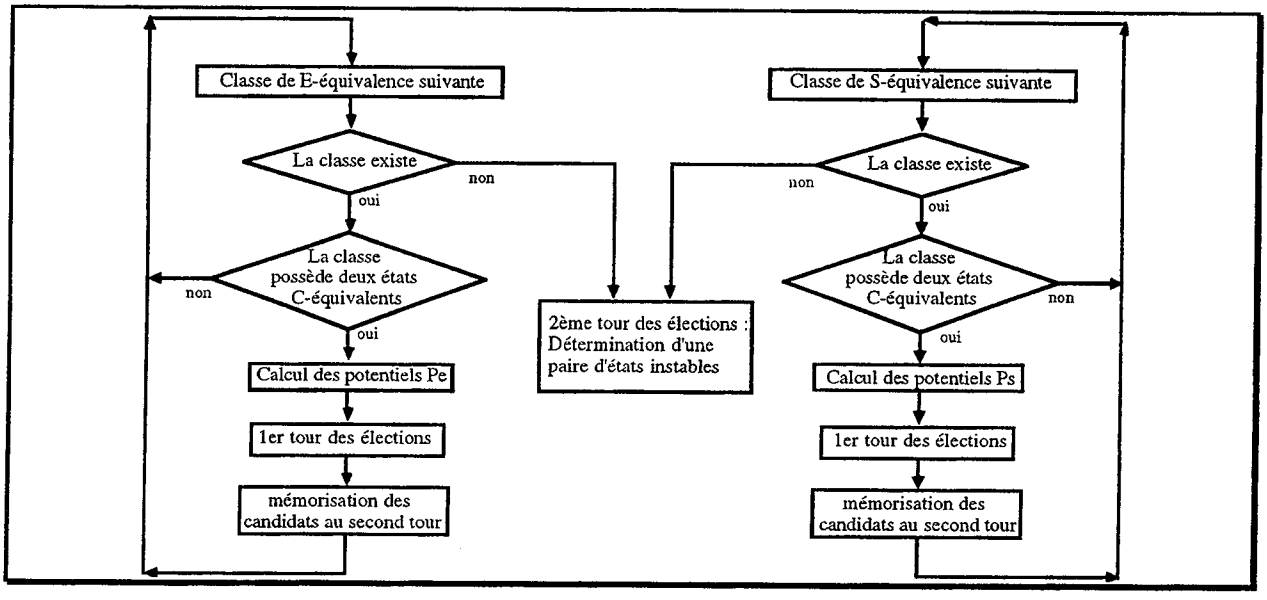

Figure III.9 - Mécanisme d'élection d'un état instable.

Algorithme implanté.

Le principe de base est expliqué ci-dessus. Quelques remarques doivent toutefois être ajoutées en ce qui concerne le fonctionnement de l'algorithme implanté.

Les C-équivalences de Type II imposent de rompre la chaîne de propagation de la signature en E-équivalence (théorème III.1). Les C-équivalences de Type III peuvent être résolues soit en E-équivalence, soit en S-équivalence. Les C-équivalences de Type III Bis peuvent être résolues uniquement en E-équivalence (deux états cousins doivent avoir les mêmes successeurs). Les C-équivalences de Type IV peuvent être résolues selon de multiples combinaisons. Le choix de la solution la plus efficace en terme du nombre d'états final du SC-graphe est donc plus délicat.

L'algorithme implanté différencie trois cas :

cas 1 - au delà de 50 états C-équivalents dans une des classes de C-équivalence du graphe, on utilise la méthode des états instables en ne considérant que la classe d'états C-équivalents de cardinalité maximum.

cas 2 - au delà de 20 états C-équivalents dans une des classes de C-équivalence du graphe, on utilise la méthode des états instables en ne considérant que la classe d'états C-équivalents de cardinalité maximum, mais en tenant compte d'informations supplémentaires sur la structure du graphe lors du calcul des potentiels de chaque état.

cas 3 - avec un cardinal maximum des classes de C-équivalence inférieur à 20 états, une analyse des conflits est réalisée sur l'ensemble des classes de C-équivalence. De plus la détermination des potentiels tient compte des mêmes informations supplémentaires que dans le cas 2.

La nouvelle fonction de calcul des potentiels proposée dans les cas 2 et 3 consiste en une recherche des chaînes de propagation en E-équivalence et en S-équivalence entre deux états C-équivalents donnés, puis à une augmentation des potentiels pour les états de ces chaînes. Pour une situation pouvant être rompue soit par la création d'un état, soit par la création de deux, l'augmentation des potentiels effectuée tendra à favoriser le premier cas. La recherche de toutes les chaînes entre chaque paire d'états C-équivalents dans le graphe (cas 3), ou dans la classe de C-équivalence de plus grande cardinalité (cas 2), étant trop complexe du point de vu algorithmique, la recherche est limitée aux chaînes d'une longueur inférieure ou égale à une certaine valeur. Les résultats présentés au paragraphe III.5.1 ont été obtenus avec une recherche des chaînes d'une longueur inférieure ou égale à 3 (complexité en $O(N^3)$). Il va de soit que les chaînes en E-équivalence ne sont pas recherchées pour les C-équivalences de Type II, ainsi que les chaînes en S-équivalence pour les C-équivalences de Type III Bis.

La complexité totale de l'algorithme de cette deuxième phase est $O(N^5)$.

III.2.3. Flot de synthèse

L'ensemble du flot de synthèse a été intégré dans l'outil ASYL-SdF. Le flot de synthèse débute par la modification du graphe de contrôle si celui-ci n'est pas S-codable. Ensuite, le SC-graphe est codé. L'algorithme de S-codage utilisé a été défini par R. Leveugle.

Cet algorithme prend en compte deux types de contraintes : les contraintes impératives, liées au respect de la propriété d'invariance forcée, et les contraintes facultatives, liées aux situations d'optimisation décrites au paragraphe I.2.3.2. L'algorithme donne donc d'abord la priorité au respect de la propriété d'invariance des signatures, puis, si c'est possible, cherche à satisfaire en plus les contraintes d'adjacence les plus susceptibles de conduire à de fortes optimisations de la logique. L'utilisateur peut imposer certaines contraintes supplémentaires : le nombre de variables d'état (par défaut égal à k, la partie entière supérieure du $\log_2$(nb. états)), le code de certains états, la signature après ou avant certains états, le polynôme diviseur, les points de contrôle, etc ... Si l'utilisateur ne précise pas le polynôme diviseur, le programme implanté le choisi premier primitif. La syntaxe du fichier fourni à ASYL-SdF pour la spécification de ces contraintes utilisateur est donnée en annexe III.1.

L'organigramme de l'algorithme implanté est fourni en figure III.10. Le principe est le suivant. Le programme commence par répercuter au niveau de l'ensemble du graphe les conséquences des choix de l'utilisateur. En particulier, le choix de certaines signatures et de certains codes d'états peu imposer la valeur d'autres codes d'états ou d'autres signatures avant et après certains états, du fait des contraintes obligatoires liées à la propriété d'invariance forcée. Ensuite, l'algorithme effectue un parcours en profondeur des codages possibles : un code d'état est choisi, et si aucune contradiction n'apparaît avec la propriété d'invariance, le code d'un autre état

est choisi et ainsi de suite. Lorsqu'une impossibilité apparaît, un retour arrière dans l'arbre des possibilités de codage est effectué afin d'essayer les autres codages. Cet algorithme n'est évidemment pas polynomial. Il a toutefois l'intérêt de fournir une aide efficace au concepteur. De nombreux exemples de contrôleurs ont en effet été codés sans intervention humaine. Celle-ci est parfois nécessaire pour gagner du temps : l'algorithme n'étant pas polynomial et n'utilisant pas d'heuristique (parcours systématique), la recherche d'un codage satisfaisant peut être très long. Pour la plupart des exemples ayant nécessité une intervention humaine, une solution intéressante, du point de vue des contraintes d'adjacence satisfaites, a été obtenue en quelques heures en jouant sur les contraintes utilisateurs. Du fait de cette limite de l'algorithme, seule une vingtaine d'exemples ont été synthétisés avec l'implantation CFC NoAjs, contre plus d'une soixantaine pour l'implantation CFC Ajs. L'investissement en temps, bien que relativement faible pour quelques exemples, est apparu excessif par rapport à l'intérêt d'augmenter la liste des résultats.

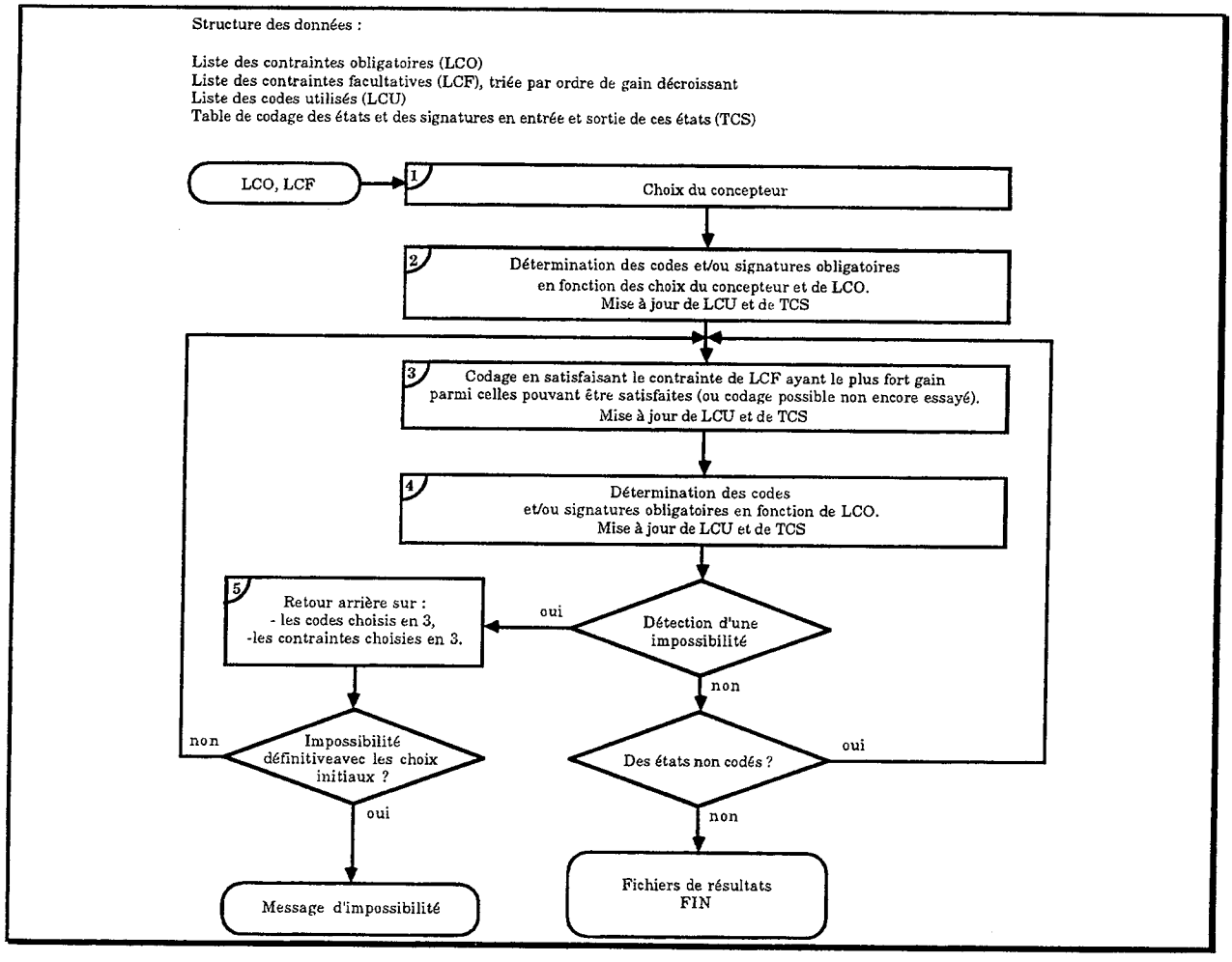

Figure III.10 - Algorithme de S-codage implanté dans ASYL-SdF.

Le S-codage fournit deux résultats : la liste des codes d'états et la liste des signatures après les états. Une fois le codage effectué, si l'utilisateur n'a pas imposé de points de contrôle, deux états sont choisis en fonction de leur nombre d'arcs entrants et sortants, de manière à placer les points de contrôle sur des points de jonction importants. Ensuite les fonctions logiques de génération du signal de comparaison et du signal ε sont générées, ainsi que les fonctions de calcul de l'état suivant et de calcul des sorties. Elles sont minimisées, factorisées et implantées avec de la logique combinatoire. Le MISR correspondant au polynôme diviseur sélectionné ou imposé, le registre d'états et la bascule de mémorisation du signal d'erreur sont construits et le tout est

interconnecté comme l'indique la figure III.1. L'utilisateur peut choisir deux types de MISR : soit un MISR à OU-Exclusifs internes soit un MISR à OU-Exclusifs externes. Le mode de fonctionnement de ces deux types de MISR est le même, seules changent les équations de calcul de la signature suivante (ce qui implique un S-codage différent du contrôleur).

III.3. AJUSTEMENTS EXPLICITES / RÉFÉRENCES EXPLICITES

La section précédente présente le flot de synthèse d'une architecture basée sur l'utilisation d'ajustements implicites et de références explicites. Le S-codage compact optimisé des états du contrôleur pour l'ajustement implicite de la signature est un problème dont la complexité a été mentionnée dans [Leve 90b] et [Esch 92].

Une solution à ce problème de codage est d'utiliser des ajustements explicites. En ajustant explicitement la signature, il est en effet possible d'utiliser directement les procédures classiques de codage compact optimisé (paragraphe 1.2.3.2). La solution présentée ici est nettement différente de celle proposée dans [Esch 92], puisqu'ici les ajustements explicites sont utilisés pour éviter la modification du graphe et le S-codage, tandis que dans [Esch 92], ceux-ci sont utilisés pour éviter uniquement la modification du graphe de contrôle. Le codage classique ne prenant pas en compte les contraintes liées à la propriété d'invariance forcée, il va de soit que la méthode utilisée ici nécessite un ajustement plus fréquent de la signature que celle proposée dans [Esch 92].

III.3.1. Architecture

Le principe de l'architecture adoptée, appelée CFC Ajs (« Control Flow Checking with Adjustments » : vérification d'un flot de contrôle avec ajustements), est décrit par la figure III.11. Comme pour l'architecture CFC NoAjs, le MISR et le registre d'états fonctionnent en phase, le signal de comparaison est calculé à partir de l'état courant, et un bloc de logique génère le signal ε en fonction des sorties du MISR et du registre d'états.

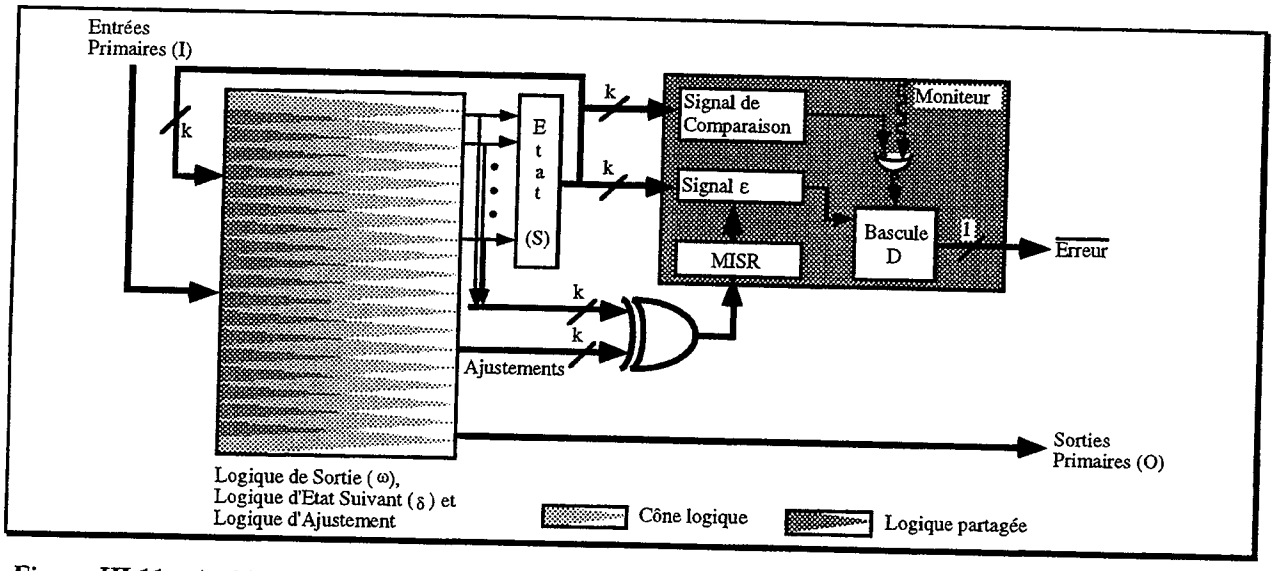

Figure III.11 - Architecture CFC Ajs (principe).

Toutefois, les entrées du MISR ne sont pas directement connectées aux sorties du bloc de calcul de l'état suivant. L'ajustement de la signature se fait, en fait, par l'ajustement du code d'état suivant, c'est-à-dire par l'ajustement du vecteur binaire à compacter. L'ajustement se fait par

l'intermédiaire d'une somme modulo 2 bit à bit entre le code de l'état suivant, donc, et la valeur d'ajustement. Afin d'autoriser l'utilisation de valeurs φ-Booléennes lors de la génération et de la minimisation des fonctions de calcul des valeurs d'ajustement, un signal α est généré. Ce signal est à 1 lorsque la signature doit être ajustée et à 0 sinon. Le signal α est connecté à l'entrée de sélection d'une série de multiplexeurs 2 vers 1. L'une des entrées de ces multiplexeurs est connectée aux sorties de la logique de calcul de l'état suivant, et l'autre aux sorties des portes OU-Exclusif réalisant la somme modulo 2 bit à bit. Les entrées de ces portes sont connectées aux sorties de la logique de calcul de l'état suivant et de calcul des ajustements. Les entrées du MISR sont connectées aux sorties des multiplexeurs.

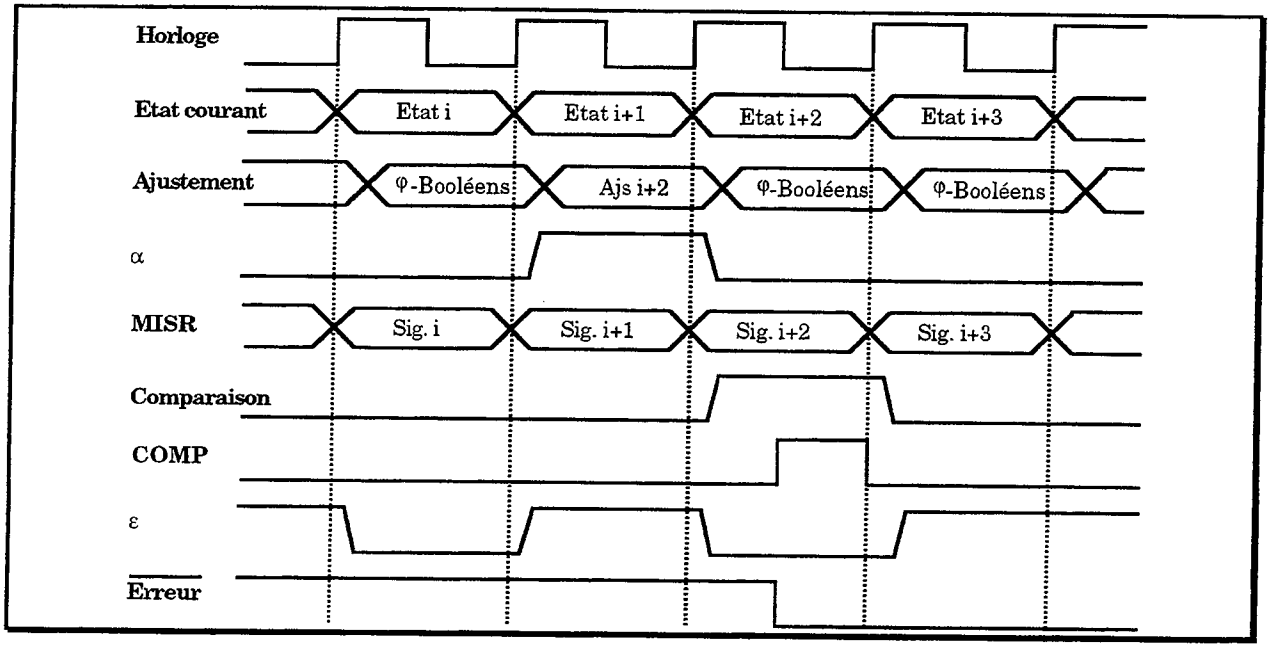

Figure III.12 - Synchronisation pour la vérification de la signature avec Ajs.

La synchronisation est identique à celle de l'architecture CFC NoAjs. La figure III.12 en résume le principe. Comme pour l'architecture CFC NoAjs, la bascule de mémorisation du signal d'erreur fonctionne en opposition de phase par rapport au MISR et au registre d'état, et est activée grâce au signal de comparaison. Le signal α et les valeurs d'ajustement sont calculés à partir du code de l'état courant et des entrées primaires, c'est-à-dire de la même manière que les fonctions de calcul de l'état suivant.

III.3.2. Placement des Ajustements

Pour synthétiser l'architecture présentée au paragraphe précédent, il est d'abord nécessaire de déterminer à quels moments sera ajustée la signature, c'est-à-dire au niveau de quelles transitions seront placés les ajustements dans le GFC. Le placement des ajustements est un problème délicat mais déjà bien étudié dans le cadre de la vérification d'un flot de contrôle niveau système [Wart 90], [Wilk 91], [Wilk 93]. Un algorithme optimal a, en particulier, été proposé par K. Wilken en 1993 ([Wilk 93]) et est présenté au paragraphe III.3.2.1. Le paragraphe III.3.2.2 présente l'application de cet algorithme aux contrôleurs, et le paragraphe III.3.2.3 présente l'algorithme simplifié implanté.

III.3.2.1. Algorithme de [Wilk 93]

L'algorithme présenté dans [Wilk 93] est inspiré d'un algorithme décrit dans [Gabo 88], et permettant de trouver la pseudo-forêt étendue maximale (« Maximum Spanning Pseudoforest ») dans un graphe non orienté pondéré. Une pseudo-forêt est un graphe non forcément connexe, composé de différents sous-graphes, chaque sous-graphe étant un arbre ou un pseudo-arbre. Un pseudo-arbre est un graphe connexe contenant un et un seul cycle. Une pseudo-forêt étendue est une pseudo-forêt contenant le plus grand nombre possible d'arêtes. Une pseudo-forêt étendue maximale est une pseudo-forêt étendue telle que la somme des poids associés à chacune de ses arêtes est maximale.

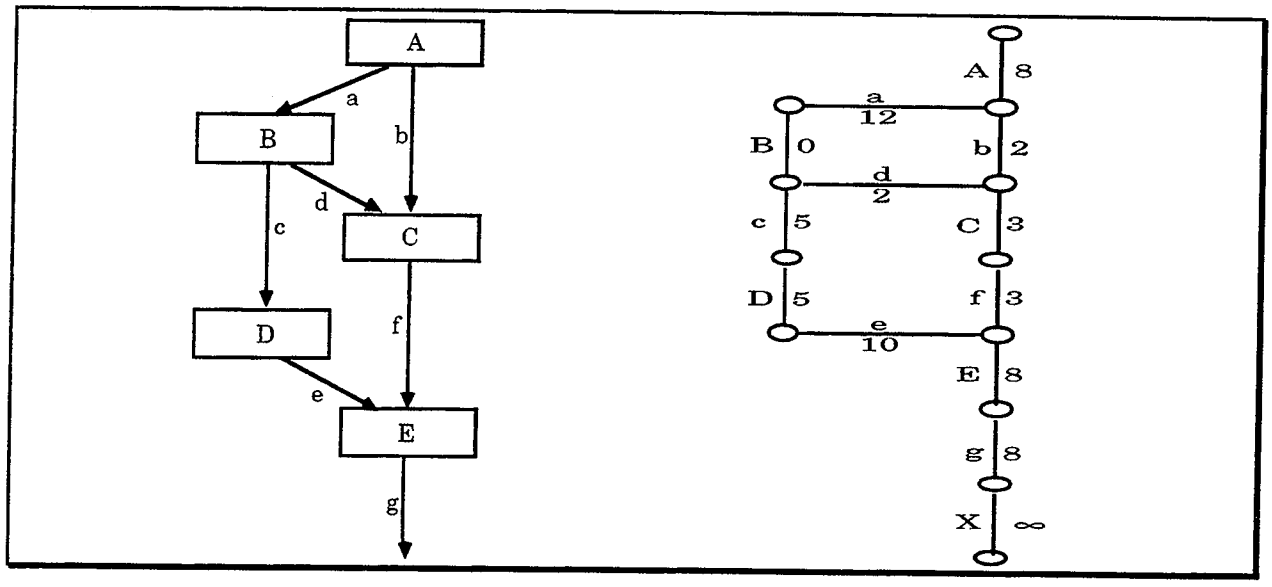

Figure III.13 - Génération du graphe non orienté pour le placement des ajustements : niveau système.

Dans [Wilk 93], le placement des ajustements se fait en deux étapes. La première étape consiste à générer un graphe non orienté pondéré à partir de l'organigramme du programme considéré. Un exemple de transformation est donné en figure III.13. Dans le graphe non-orienté, chaque instruction i de l'organigramme est représentée par deux sommets si_1 et si_2 reliés par une arête. De même, un arc reliant une instruction i à une instruction j dans l'organigramme, sera représenté par une arête reliant les sommets si_2 et sj_1. La pondération associée à chaque arête du graphe non orienté représente le coût en performance estimé du placement d'un ajustement au niveau de l'instruction, ou de l'arc, de l'organigramme correspondant à l'arête considérée.

La deuxième étape consiste à placer effectivement les ajustements. Pour cela, K. Wilken utilise l'algorithme détaillé par la figure III.14. Au début de l'algorithme, tous les sommets sont coloriés en bleu et toutes les arêtes en rouge-clair. Un composant est défini comme étant un sous-graphe connexe. Au début de l'algorithme, chaque sommet constitue donc un composant bleu.

Dans la condition $\alpha^D \neq 1$, α représente la racine du polynôme diviseur et D la somme :

$$\sum_{i=1}^{max} d_i$$

max est le nombre d'arêtes (si_1, si_2) du cycle, c'est-à-dire le nombre d'arêtes du cycle correspondant à une instruction de l'organigramme du programme d'application. Les d_i sont calculés en prenant en compte des informations liées au sens du flot de contrôle, et donc à l'organigramme du programme. d_i est égal à 1 si, en suivant le cycle, l'instruction i est traversée dans le sens du flot ; d_i est égal à -1 sinon. Dans l'exemple de la figure III.13, si on considère le

cycle (c, D, e, f, C, d), alors $d_D = 1$ et $d_C = -1$, ou $d_D = -1$ et $d_C = 1$, selon le sens de parcours du cycle. Dans les deux cas $\alpha^D = 1$.

K. Wilken a prouvé que lorsque $\alpha^D \neq 1$ pour un cycle donné, il est possible de trouver des valeurs de signature avant et après les instructions éléments du cycle, telles qu'aucun ajustement n'est nécessaire, et ce sans modifier l'information compactée, c'est-à-dire le code opératoire des instructions. Ainsi, si pour le cycle (a, B, d, b) de la figure III.13, on a $\alpha^D \neq 1$, alors cela signifie qu'il est possible de trouver des valeurs de signature avant et après l'instruction B telle que la signature des chemins (A, B, C) et (A, C), dans l'organigramme, soit la même. Ceci signifie que si on choisit les bonnes valeurs pour les signatures, il n'est pas nécessaire d'ajuster l'une ou l'autre des signatures de ces deux chemins. K. Wilken précise que ceci ne s'applique qu'à la division polynomiale qui génère un code cyclique, mais n'est pas adaptable à la somme modulo 2 pour laquelle la signature est indépendante de l'ordre de compaction de l'information.

A la fin de l'algorithme, c'est l'ensemble des arêtes rouges (claires ou sombres) qui détermine l'endroit (ou plutôt le moment) où sera effectué l'ajustement de la signature : chaque arête rouge correspond à un ajustement.

```
Tant que l'ensemble des composants bleus ayant des arêtes incidentes rouge-claires n'est pas vide Faire
        Sélectionner un composant T bleu ayant le plus petit nombre d'arêtes incidentes rouge-claires
        Sélectionner une arête rouge-claire a, incidente à T et de poids maximum
        Choix entre trois cas :
                Cas où a connecte T à un autre composant S bleu Faire
                        Colorier a en bleu
                        T et S deviennent un seul et même composant bleu contenant a
                Fin Cas
                Cas où a connecte T à lui-même Faire
                        Si α^D ≠ 1 pour le cycle formé par a dans T Alors
                                Colorier T et a en vert
                                a devient une des arêtes de T
                        Sinon
                                Colorier a en rouge sombre
                        Fin Si
                Fin Cas
                Cas où a connecte T à un composant S vert Faire
                        Colorier T et a en vert
                        T et S deviennent un seul et même composant vert contenant a
                Fin Cas
        Fin Choix
Fin Tant que
```

Figure III.14 - Algorithme de placement des ajustements d'après [Wilk 93].

K. Wilken indique que cet algorithme est optimal du point de vue du nombre d'ajustements placés, et du point de vue du coût de ces ajustements (somme des pondérations). La complexité de cet algorithme est O(n+m) où n représente le nombre de sommets et m le nombre d'arêtes du graphe considéré. Dans le cas de K. Wilken, $m \leq 2n$ et la complexité est donc O(n). De plus, la complexité de l'algorithme de calcul des signatures avant et après les instructions d'un cycle tel que $\alpha^D \neq 1$ est $O(n*\log_2(n))$. Le nombre de ce type de cycles pouvant être n, la complexité du calcul des signatures est $O(n^2*\log_2(n))$ ([Wilk 93]).

III.3.2.2. Application aux contrôleurs

Appliquer l'algorithme de placement des ajustements, décrit dans le paragraphe précédant, au cas des contrôleurs, nécessite un aménagement de la première étape. Dans le cas des contrôleurs, deux points diffèrent du cas des programmes d'application.

Le premier point est qu'il est hors de question de placer des ajustements au niveau des états du graphe de contrôle, comme c'est le cas pour les instructions du programme d'application. En

effet, ajuster la signature au niveau des états signifierait ajouter des états spécifiques à l'ajustement de la signature, ce qui modifierait les caractéristiques temporelles du contrôleur résultant. Seuls les ajustements sur les transitions doivent donc être autorisés.

Le second point est que, dans le cas des contrôleurs, il nous est pour l'instant très difficile de prévoir le coût en performance du placement d'un ajustement à tel ou tel endroit dans le graphe, et donc d'effectuer une pondération des arêtes du graphe non-orienté auquel est appliqué l'algorithme. Le coût en performance d'un ajustement est en effet lié à la valeur et au placement des autres ajustements, à la valeur des codes des états du graphe et des sorties, qui vont influencer les phases de minimisation, factorisation, projection structurelle et routage des fonctions logiques chargées de calculer les valeurs d'ajustement, du code de l'état suivant et des sorties.

La génération du graphe non orienté, à partir du graphe de contrôle, résultant de ces deux remarques est la suivante. A chaque état du graphe de contrôle est associé un sommet du graphe non orienté, et chaque arc est remplacé par une arête. Un exemple est fourni en figure III.15. L'algorithme de la deuxième phase est alors quasi identique à celui de la figure III.14, à ceci près que les termes « Sélectionner une arête rouge-claire *a*, incidente à *T* et de poids maximum » deviennent « Prendre une arête rouge-claire *a* incidente à *T* ». De plus, le calcul de l'exposant D dans la condition $\alpha^D \neq 1$ est légèrement différent :

- $D = \sum_{i=1}^{max} d_i$
- max est égal au nombre de sommets du cycle
- d_i est égal à 1 si, en suivant le cycle, l'état S_i est traversé dans le sens du flot, à -1 si l'état S_i est traversé dans le sens inverse du flot, et à 0 sinon.

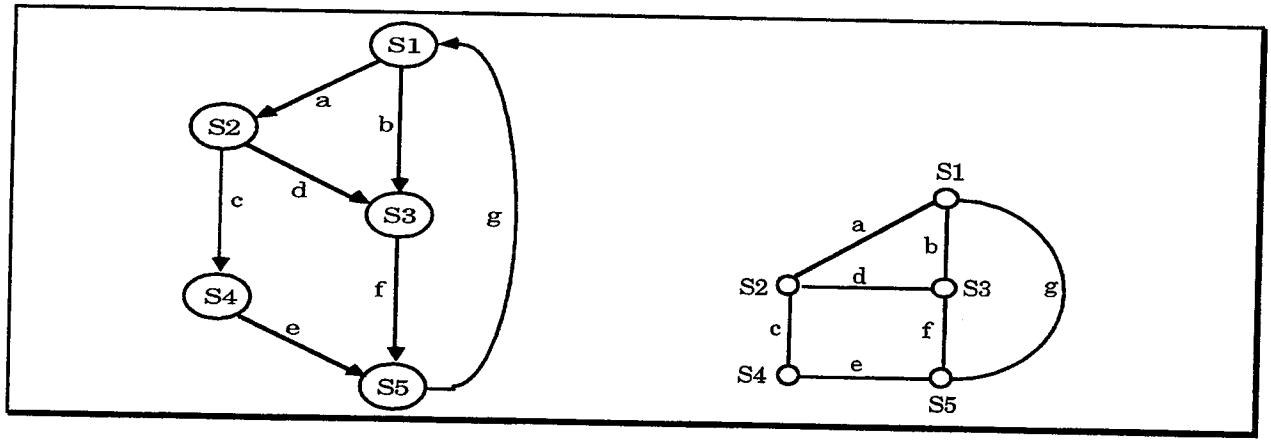

Figure III.15 - Génération du graphe non orienté pour le placement des ajustements : niveau contrôleur.

Dans l'exemple de la figure III.15, si on considère le cycle (c, S4, e, S5, f, S3, d, S2), alors $d_{S4} = 1$, $d_{S5} = 0$, $d_{S3} = -1$ et $d_{S2} = 0$, ou $d_{S4} = -1$, $d_{S5} = 0$, $d_{S3} = 1$ et $d_{S2} = 0$, selon le sens de parcours du cycle. Dans les deux cas $\alpha^D = 1$. $d_{S5} = 0$ et $d_{S2} = 0$ car les arêtes du cycle incidentes aux sommets S5 et S2 dans le graphe non-orienté correspondent à des arcs soit tous entrants, soit tous sortants dans le graphe de contrôle : il n'y a pas d'indication du sens de traversée de ces deux états par le flot de contrôle vis à vis du sens de parcours du cycle dans le graphe non-orienté.

Ces différentes modifications de l'algorithme de la figure III.14 ne sont en fait que des adaptations à la nouvelle structure du graphe non-orienté généré. La complexité de l'algorithme résultant est aussi O(n+m). Toutefois, dans le cas des contrôleurs, étant donné qu'un état peut avoir plus de deux successeurs, $m \leq n^2$. La complexité exprimée uniquement en fonction du

nombre de sommets du graphe est donc $O(n^2)$. La complexité du calcul des signatures reste inchangée.

III.3.2.3. Algorithme simplifié

Lorsque le graphe d'états du contrôleur contient des états rebouclés, il est possible de s'affranchir du calcul de la racine du polynôme, et par la même occasion de simplifier le calcul des signatures avant et après chaque état (assez complexe pour les cycles tels que $\alpha^D \neq 1$ contenant un grand nombre d'états), tout en obtenant un nombre minimal d'ajustements. L'absence de pondération nous permet en effet de ne considérer que le problème du nombre d'ajustements de la signature à effectuer. Cela nous a incité à implanter un algorithme simplifié de placement des ajustements qui fournit soit un nombre minimal d'ajustements, soit un nombre très proche, selon la structure du graphe de contrôle considéré.

Algorithme par recherche d'un arbre étendu

Dans la suite nous appelons $A_{\alpha^D \neq 1}$ l'algorithme de la figure III.14 modifié selon les indications du paragraphe III.3.2.2, et A_{arbre} un algorithme optimal quelconque de recherche d'un arbre étendu dans les graphes connexes. On appelle « arbre étendu » d'un graphe connexe G, un arbre contenant tous les sommets de G et le nombre maximum d'arêtes. On considère dans la suite que A_{arbre} colorie les noeuds et les arêtes de l'arbre étendu en bleu, et colorie les arêtes de G non élément de l'arbre étendu en rouge.

- **Théorème III.3 :**

Soit G un graphe connexe. L'application de l'algorithme $A_{\alpha^D \neq 1}$ à G se termine avec uniquement des composants verts et des arêtes rouges si et seulement si il existe au moins un cycle dans le graphe G tel que $\alpha^D \neq 1$.

Démonstration :

L'implication « si l'application de l'algorithme $A_{\alpha^D \neq 1}$ à G se termine avec uniquement des composants verts et des arêtes rouges alors il existe au moins un cycle dans le graphe G tel que $\alpha^D \neq 1$ » est évidente. La suite de la démonstration s'intéresse à l'implication inverse, et plus précisément à sa contraposée, à savoir « si l'application de $A_{\alpha^D \neq 1}$ à G génère autre chose que des composants verts et des arêtes rouges, alors G n'est pas connexe ou ne contient pas de cycle tel que $\alpha^D \neq 1$ ».

De manière générale, l'exécution de $A_{\alpha^D \neq 1}$ peut générer des composants vertes, des composants bleus et des arêtes rouges. Les deux cas à envisager pour démontrer cette contraposée sont donc : l'algorithme génère un seul composant bleu et l'algorithme génère plusieurs composants dont un au moins est bleu.

1)- Supposons que l'algorithme se termine avec un seul composant T bleu. L'algorithme étant terminé, cela signifie qu'il n'existe aucune arête rouge-claire incidente à T. Le graphe étant connexe, cela signifie aussi que toutes les arêtes ont été parcourues, et qu'aucune d'entre elles ne connectait un composant à lui-même tel que $\alpha^D \neq 1$. G ne contient donc pas de cycle tel que $\alpha^D \neq 1$.

2)- Supposons que l'algorithme se termine avec un composant bleu T et un ou plusieurs autres composants (bleus ou verts, peu importe). L'algorithme étant terminé, cela signifie

qu'il n'existe aucune arête rouge-claire incidente à T. Or, par construction, une arête rouge sombre, bleue ou verte ne peut relier qu'un composant à lui-même. Puisqu'il n'existe ni arête rouge, ni arête verte, ni arête bleue reliant T à un au moins des autres composants, cela signifie que G n'est pas connexe, ce qui est contraire à l'hypothèse de départ.

Ceci démontre que si l'algorithme se termine avec au moins un composants bleu, soit G n'est pas connexe, soit G ne contient pas de cycle tel que $\alpha^D \neq 1$. On démontre ainsi l'implication contraposée : « si G contient au moins un cycle tel que $\alpha^D \neq 1$, alors l'application de l'algorithme $A_{\alpha^D \neq 1}$ à G se termine avec uniquement des composants verts et des arêtes rouges ».

- **Théorème III.4 :**

Soit un graphe connexe G contenant au moins un cycle tel que $\alpha^D \neq 1$. Si N est le nombre d'arêtes rouges résultant de l'application de l'algorithme $A_{\alpha^D \neq 1}$ à G, alors le nombre d'arêtes rouges résultant de l'application de l'algorithme A_{arbre} à G est inférieur ou égal à N+1.

Démonstration :

G étant connexe et contenant au moins un cycle tel que $\alpha^D \neq 1$, l'algorithme $A_{\alpha^D \neq 1}$ se termine avec uniquement des composants verts et des arêtes rouges (théorème III.3). On appelle P le nombre de composants. Chaque composant vert contient par construction un cycle et un seul (pseudo-arbres). Comme le graphe G est connexe, il existe dans G au moins P-1 arêtes rouges reliant les P composants entre eux. On appelle r le nombre d'arêtes rouges supplémentaires et N = (P-1)+r le nombre total d'arêtes rouges obtenues en appliquant $A_{\alpha^D \neq 1}$ à G.

Un algorithme simple pour trouver un sous-graphe sans cycle connexe (un arbre) dans G contenant tous les noeuds de G est d'appliquer $A_{\alpha^D \neq 1}$ à G, puis de colorier une arête de chacun des P composants en rouge afin de supprimer le cycle que contient chacun des composants, et enfin de colorier en vert les P-1 arêtes rouges reliant les P composants entre eux. Le nombre d'arêtes rouges résultant est alors N+P-(P-1), c'est-à-dire N+1. L'algorithme A_{arbre} étant optimal, il fournira un nombre d'arêtes rouges inférieur ou égal à N+1, ce qui termine la démonstration.

Du fait du théorème III.4, l'algorithme suivant donne un nombre minimal ou presque minimal d'ajustements. On commence par appliquer A_{arbre} au graphe G, puis on parcourt toutes les arêtes rouges à la recherche d'un cycle tel que $\alpha^D \neq 1$.

Si on le trouve, l'arête correspondante et tout l'arbre étendu sont coloriés en vert et forment un pseudo-arbre. Le nombre d'ajustements résultant de cet algorithme est minimal puisqu'il est inférieur ou égal à celui obtenu avec l'algorithme $A_{\alpha^D \neq 1}$ (en fait égal puisque $A_{\alpha^D \neq 1}$ est optimal).

Dans le cas ou aucun cycle tel que $\alpha^D \neq 1$ n'est trouvé, le nombre d'ajustements obtenu est au plus supérieur d'une unité à celui obtenu avec $A_{\alpha^D \neq 1}$. Dans le cas particulier où G ne contient aucun cycle tel que $\alpha^D \neq 1$, l'algorithme ci-dessus et $A_{\alpha^D \neq 1}$ donnent le même nombre d'ajustements (ils créent tous les deux un arbre étendu).

Intuitivement, on peut remarquer que si au moins un cycle tel que $\alpha^D \neq 1$ existe dans le graphe connexe G, le parcours de toutes les arêtes rouges nous permet de le trouver à coup sûr, ce qui implique que l'algorithme présenté ci-dessus est optimal. Toutefois, cette propriété n'étant pas utile à la suite de notre discours, nous ne le démontrerons pas ici.

Algorithme implanté

L'algorithme présenté ci-dessus est, en particulier, intéressant pour les graphes contenant des états rebouclés, parce que vérifier qu'il est possible de ne pas ajuster la signature sur l'arc rebouclant d'un état rebouclé est particulièrement simple (Corollaire III.1). Il en est de même pour le calcul de la valeur de la signature à imposer avant et après un état rebouclé, si on veut éviter l'ajustement sur l'arc rebouclant. Nous rappelons ici que le respect de la propriété d'invariance II.1 impose une signature identique avant et après un état rebouclé, si aucun ajustement n'est effectué sur la transition rebouclante.

- **Corollaire III.1 : Corollaire au théorème II.2**

Un état rebouclé ne nécessite pas d'ajustement de la signature sur l'arc rebouclant si le nombre de bits à 1 du code de l'état rebouclé ou le nombre de bits à 1 du polynôme diviseur est pair (on ignore ici le bit toujours à 1 indiquant le degré du polynôme).

Démonstration :

Soient e le code d'état et Pol le polynôme diviseur. Dans la suite, on ignore le bit de poids fort Pol_k du polynôme diviseur, qui est toujours à 1. Soient b(t) la signature avant le compactage de e et b(t+1) la signature après le compactage de e. Les équations d'un MISR à OU-Exclusifs internes sont les suivantes (paragraphe II.1.4.1) :

$$b_i(t+1) = b_{i-1}(t) \oplus (Pol_i \,.\, b_{k-1}(t)) \oplus e_i(t+1) \qquad \text{pour } 1 \le i \le k-1$$
$$b_0(t+1) = e_0(t+1) \oplus (Pol_0 \,.\, b_{k-1}(t))$$

Eviter d'ajuster la signature signifie imposer la même signature avant et après l'état rebouclé. Si on appelle x le bit de poids fort $b_{k-1}(t) = b_{k-1}(t+1)$ de la signature, on a :

$$x = e_0 \oplus e_1 \oplus \ldots \oplus e_{k-1} \oplus (Pol_0 \,.\, x) \oplus (Pol_1 \,.\, x) \oplus \ldots \oplus (Pol_{k-1} \,.\, x) \qquad (3)$$

Du fait de la formule (3), il existe quatre cas différents :

- Soit e et Pol comportent un nombre pair de 1, auquel cas x=0 est une solution,
- Soit e comporte un nombre pair de 1 et Pol un nombre impair, auquel cas x=0 et x=1 sont deux solutions,
- Soit e comporte un nombre impair de 1 et Pol un nombre pair, auquel cas x=1 est une solution,
- Soit e et Pol comportent un nombre impair de 1, auquel cas il n'y a pas de solution.

```
Compter le nombre de bits à 1 du polynôme diviseur
Répéter
        Choisir un état reboucler et compter le nombre de bits à 1
Jusqu'à trouver une solution ou épuiser la liste des états rebouclés
Mémoriser le résultat de la recherche d'un état rebouclé satisfaisant
Appliquer A_arbre au graphe G
```

Figure III.16 - Algorithme de placement des ajustements implanté.

L'algorithme implanté résultant de ces diverses simplifications est décrit par la figure III.16. La complexité de la recherche d'un état rebouclé satisfaisant est $O(n*\log_2(n))$. L'algorithme de recherche d'un arbre étendu est un algorithme classique de la théorie des graphes de parcours des arêtes du graphe. Sa complexité est $O(n^2)$. L'algorithme de la figure III.16 ne nécessite pas le

calcul de la racine du polynôme et simplifie le calcul des signatures avant et après chaque état dont la complexité est désormais $O(n*\log_2(n))$. Il s'agit simplement de calculer la signature avant et après l'état rebouclé choisi, puis de parcourir les autres états pour calculer les autres signatures à l'aide des équations du MISR choisi.

Du point de vue du nombre d'ajustements nécessaires, si on appelle N_{Ajs} le nombre minimal fourni par l'algorithme $A_{\alpha^D \neq 1}$, l'algorithme de la figure III.16 nécessite N_{Ajs} ajustements de la signature si le graphe contient un état rebouclé satisfaisant où si il ne contient pas de cycles tel que $\alpha^D \neq 1$, et $N_{Ajs}+1$ sinon.

III.3.3. Flot de synthèse

Comme pour l'architecture CFC NoAjs, l'ensemble du flot de synthèse a été intégré dans l'outil ASYL-SdF. Le flot de synthèse débute par un codage classique des états du graphe de contrôle. Ensuite les ajustements sont placés avec l'algorithme décrit en figure III.16. Une fois le placement des ajustements effectué, la signature après chaque état est calculée. Ensuite les valeurs d'ajustement sont calculées pour chaque transition coloriée en rouge par l'algorithme de placement, en utilisant les équations du MISR choisi, et en fonction de trois paramètres : la signature après l'état d'origine de la transition, le code de l'état destination de la transition et la signature après cet état.

Ce premier traitement fournit la liste des codes d'états, la liste des signatures après les états, la liste des valeurs d'ajustement et leur emplacement dans le graphe d'état. Comme pour l'architecture CFC NoAjs, si l'utilisateur n'a pas imposé de points de contrôle, deux états sont choisis en fonction de leur nombre d'arcs entrants et sortants. Ensuite les fonctions logiques de génération du signal de comparaison et du signal ε sont générées, ainsi que les fonctions de calcul de l'état suivant, de calcul des ajustements, de calcul du signal d'ajustement et de calcul des sorties. Elles sont minimisées, factorisées et implantées avec de la logique combinatoire. Le MISR correspondant au polynôme diviseur sélectionné ou imposé, le registre d'états et la bascule de mémorisation sont construits et le tout est interconnecté selon le principe illustré par la figure III.11, comme indiqué au paragraphe III.3.1. Comme pour l'architecture CFC NoAjs, l'utilisateur peut choisir deux types de MISR : soit un MISR à OU-Exclusifs internes soit un MISR à OU-Exclusifs externes (ce qui implique un calcul différent des valeurs d'ajustement).

III.4. EVALUATION THÉORIQUE DES PROBABILITÉS DE DÉTECTION

Les deux sections précédentes présentent les architectures CFC Ajs et CFC NoAjs et leurs flots de synthèse respectifs. Nous étudions dans la suite leur efficacité théorique qui dépend de différents paramètres, à savoir la probabilité de masquage d'erreurs, la distribution de la valeur des signatures de référence dans le graphe de contrôle et l'existence ou non de fautes structurellement indétectables du fait de la méthode de vérification de signature employée.

Les architectures CFC Ajs et CFC NoAjs représentent respectivement les méthodes de vérification d'un flot de contrôle par analyse de signature du type Ajustement Explicites / Références Explicites, et Ajustement Implicites / Références Explicites. Nous nous intéresserons aussi dans la suite à la méthode de Références et Ajustements implicites proposée par S. Robinson dans [Robi 92a]. Les méthodes de vérification de la signature utilisées par ASYL-SdF pour les architectures CFC Ajs et CFC NoAjs, et la méthode proposée par S. Robinson,

seront respectivement désignées dans la suite par les termes ASYL-SdF Ajs, ASYL-SdF NoAjs et I-Tool.

Dans la suite, nous utiliserons la lettre ω pour désigner le nombre de bits de la signature pour les méthodes ASYL-SdF NoAjs et ASYL-SdF Ajs. La lettre υ désignera le nombre de bits de la signature pour la méthode I-Tool. Par construction υ est pair ([Robi 92a]). En plus de ces deux lettres, $\mathbf{b_i}$ désignera le $i+1^{ième}$ bit de la signature courante, $\mathbf{e_i}$ désignera le $i+1^{ième}$ bit de l'état courant et **G** désignera le graphe de contrôle d'une machine **M**. Enfin, nous désignerons par $\mathbf{S_i}$ et $\mathbf{S_j}$ deux états de G tels que la transition $\mathbf{S_i \rightarrow S_j}$ n'existe pas dans G.

La probabilité de masquage d'erreurs intrinsèque aux MISR lorsqu'on utilise un polynôme diviseur primitif est $1 / 2^{nb.\ bits}$, où « nb. bits » est le nombre de bascules du MISR ([Leve 90b]). Elle ne sera pas détaillée plus avant.

III.4.1. Probabilités de masquage

Nous étudions ici la probabilité de non détection d'une transition erronée $S_i \rightarrow S_j$, sans masquage par le processus de compaction. Les valeurs des signatures de référence sont supposées uniformément distribuées sur le graphe G.

Si durant le fonctionnement normal de la machine M, une faute, ou une combinaison de fautes, résulte en une transition erronée $S_i \rightarrow S_j$, cette transition illégale (au sens des chemins illégaux définis au paragraphe II.1.1.3) ne sera pas détectée si la signature courante calculée après l'état S_i est égale à la signature de référence avant l'état S_j. Ceci est bien entendu directement lié à la distribution des signatures de référence, aussi appelée corrélation entre les signatures dans [Wilk 87].

Quand les méthodes ASYL-SdF sont considérées, le nombre de signatures possibles après l'état S_i est 2^{ω}. Donc, si on suppose les signatures uniformément distribuées dans le graphe G, la probabilité de masquage d'un tel événement est $\mathbf{P_A = 2^{-\omega}}$.

Lorsqu'on s'intéresse à la méthode I-Tool, le nombre de signatures possibles après S_i est $2^{\upsilon/2}$ (tous les mots de υ bits avec les bits d'indice impair égaux à 0, c'est-à-dire toutes les valeurs non erronées possibles de signature). La probabilité de masquage est donc $\mathbf{P_I = 2^{-\upsilon/2}}$.

Considérons maintenant un chemin illégal un peu plus compliqué, avec m transitions erronées.

Nous nous intéressons d'abord au cas des méthodes ASYL-SdF Ajs et NoAjs. On suppose que le dernier état du chemin illégal est un point de comparaison de la signature. On appelle $S_i \rightarrow S_j$ la dernière transition erronée du chemin illégal avant le point de comparaison. Si la signature courante (calculée en temps réel) avant l'état S_i est incorrecte, c'est-à-dire différente de la signature de référence en ce point, les transitions erronées étant survenues avant $S_i \rightarrow S_j$ sont susceptibles d'être détectées. Toutefois, le chemin illégal ne sera pas détecté si la signature courante après compaction du code de l'état S_i est égale à la référence avant l'état S_j. De même, si la signature courante avant S_i est correcte, le chemin illégal sera tout de même détecté si la signature courante après la compaction du code de S_i est différente de la signature de référence avant l'état S_j. Ceci signifie que si on suppose les fautes à l'origine des erreurs de transition indépendantes, la probabilité de masquage d'un tel chemin illégal dépend uniquement de la dernière transition erronée $S_i \rightarrow S_j$ et est $\mathbf{P_{Ap} = 2^{-\omega}}$.

Lorsqu'on considère la méthode I-Tool, la probabilité de ne pas détecter la première transition erronée du chemin illégal est $2^{-\upsilon/2}$. Ceci est lié au fait que la méthode I-Tool vérifie la

signature à chaque cycle (chaque état est un point de comparaison). Sachant que la première transition erronée n'a pas été détectée, la probabilité de ne pas détecter la deuxième transition illégale est $2^{-\upsilon/2}$. Ainsi de suite jusqu'à la dernière transition erronée : sachant que la m-1^{ième} transition erronée n'a pas été détectée, la probabilité de ne pas détecter la dernière transition erronée est $2^{-\upsilon/2}$. La probabilité de ne détecter aucune de ces erreurs est donc $\mathbf{P_{Ip} = 2^{-m\upsilon/2}}$.

Avant de comparer υ et ω dans le paragraphe suivant, il nous faut étudier un peu plus en détail le cas particulier de la méthode ASYL-SdF Ajs.

Supposons qu'une faute ϕ_1 ne modifie que les bits de la valeur d'ajustement, elle sera détectée avec une probabilité égale à 1 si aucune autre faute ne survient avant qu'un point de comparaison ne soit atteint. Si par contre une autre faute ϕ_2 survient, la probabilité de ne pas détecter ϕ_1 et ϕ_2 est $\mathbf{P_{Ajs1} = 2^{-\omega}}$. P_{Ajs1} est en effet la probabilité que ϕ_2 corrige la signature modifiée par ϕ_1 si ϕ_2 ne modifie que les bits d'ajustement.

La logique de calcul des ajustements et de l'état suivant étant implantés ensemble (avec de la logique partagée), une faute ϕ_2 dans la logique d'ajustement peut modifier à la fois les bits de la valeur d'ajustement et les bits du code d'état suivant. Quand ϕ_2 modifie le code d'état suivant ou le code d'état suivant et la valeur d'ajustement, P_{Ajs1} est égale à la probabilité de ne pas détecter ϕ_2, c'est-à-dire P_A ou P_{Ajs2} (selon que ϕ_2 modifie uniquement le code d'état ou le code d'état et la valeur d'ajustement).

P_{Ajs2} est définie comme suit. Si une faute ϕ modifie à la fois le code d'état suivant et les bits d'ajustement, la probabilité de ne pas détecter ϕ est $\mathbf{P_{Ajs2} = 2^{-\omega}}$. En effet, P_{Ajs2} est la probabilité que la signature courante calculée après l'occurrence de ϕ soit égale à la signature de référence après l'état atteint par erreur.

Ce qui précède montre que l'utilisation d'ajustements explicites ne modifie pas sensiblement la probabilité de masquage d'erreurs par rapport à l'utilisation d'ajustements implicites. Par contre, la vérification de la signature à chaque cycle semble avoir un impact plus important. Toutefois, la comparaison des probabilités P_{Ap} et P_{Ip} nécessite de comparer les valeurs de υ et ω qui sont souvent assez différentes. De plus, l'utilisation d'un S-codage et de références implicites plutôt qu'explicites n'est pas sans conséquences sur la distribution des signatures dans le graphe d'état, ou sur certaines causes structurelles de masquage. C'est ce qui est étudié dans les deux paragraphes suivants.

III.4.2. Distribution et largeur de signature

Pour les 44 exemples étudiés dans [Robi 92a] et la vingtaine d'exemples de graphes réparés et codés (ou S-codés) avec ASYL-SdF et I-Tool, $\omega \leq \upsilon \leq 2(\omega-1)$. Aussi, nous avons $2^{-\omega+1} \leq P_I \leq 2^{-\omega/2}$, soit :

$$2P_A \leq P_I \leq \sqrt{P_A} \qquad \text{(avec } \omega \geq 2\text{)}$$

De même, $2^{-m\omega+m} \leq P_{Ip} \leq 2^{-m\omega/2}$, c'est-à-dire :

$$(2P_{Ap})^m \leq P_{Ip} \leq \sqrt{P_{Ap}{}^m} \qquad \text{(avec } \omega \geq 2\text{)} \quad (4)$$

Du point de vue de la distribution des valeurs des signatures de référence sur le graphe, le respect de la propriété d'invariance forcée et la procédure de codage ne garantissent pas une distribution uniforme de la signature, ce qui peut modifier de manière importante les probabilités de masquage.

A titre d'exemple, pour le contrôleur ex4 dont le graphe de contrôle est fourni en annexe III.2, le nombre de transitions illégales indétectables entre deux états du graphe est 0 si les signatures sont uniformément distribuées, 3 si la distribution des signatures respecte la propriété d'invariance forcée II.1, 5 pour la méthode ASYL-SdF Ajs, 17 pour la méthode ASYL-SdF NoAjs et 33 pour la méthode I-Tool.

Si pour les méthodes ASYL-SdF ce problème n'a pas encore été bien formalisé, que ce soit au niveau du placement des ajustements ou au niveau du S-codage, I-Tool en tient compte lors du S-codage. Toutefois, sur les 44 exemples traités dans [Robi 92a], 33 ont un nombre de signatures différentes maximum vis à vis de la propriété d'invariance forcée, et 11 ont un nombre minimum de signatures différentes. La minimisation du nombre de signatures différentes est utilisée par S. Robinson pour réduire le nombre de variables internes, le but étant de réduire le coût matériel de la méthode. L'inconvénient est que la mauvaise distribution des valeurs de référence qui en résulte réduit fortement les probabilités de détection d'erreurs. La figure III.17 montre, pour un même exemple, la valeur des codes d'états et des signatures de référence pour un S-codage qui maximise le nombre de signatures différentes (Fig. III.17a), et pour un S-codage qui, au contraire, le minimise (Fig. III.17b). Les transitions erronées indétectables résultant de cette réduction du nombre de signatures différentes sont indiquées en pointillés.

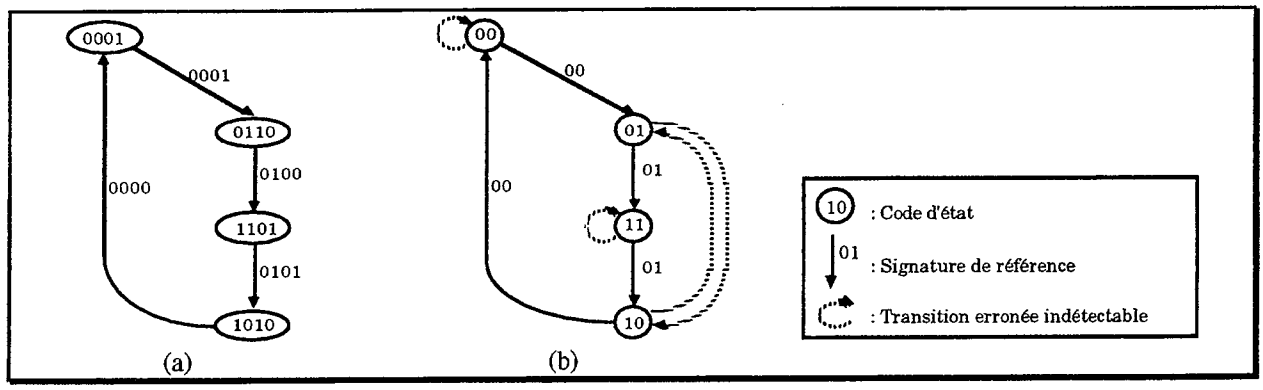

Figure III.17 - Méthode I-Tool : S-codage qui maximise (a) ou minimise (b) le nombre de références différentes.

L'inégalité (4) montre que P_{Ap} est inférieure à P_{Ip} lorsque m est égal à 1, et est supérieure lorsque m est supérieur à 2. Toutefois ce résultat est à nuancer en fonction de la distribution des valeurs des signatures de référence sur le graphe, distribution qui dépend du respect de la propriété d'invariance et des algorithmes de S-codage ou de placement des ajustements. En particulier, la technique de réduction du nombre de signatures différentes utilisée par I-Tool tend à augmenter sensiblement la probabilité de masquage, du fait de l'augmentation du nombre de transitions erronnées indétectables.

III.4.3. Fautes structurellement indétectables

Le paragraphe III.4.1 précise la probabilité de détection d'une erreur essentiellement du point de vue du moment où la faute qui en est la cause devient active, ce qui se traduit au niveau du graphe d'états par ses conséquences topologiques (cheminement selon une transition inexistante dans le graphe). Nous nous intéressons ici aux caractéristiques structurelles (lieu d'occurence de la faute à l'origine de l'erreur), et non plus temporelles des erreurs. Plus particulièrement, nous nous intéressons aux fautes dont les conséquences sont indétectables, de par la mise en défaut structurelle du dispositif de vérification de la signature. Dans l'absolu, la non-détection des erreurs résultant de ce type de fautes est indépendante du moment auquel ces dernières sont activées.

III.4.3.1. Mise en défaut structurelle pour ASYL-SdF

Pour les méthodes ASYL-SdF, la vérification de la signature se fait lors de la rencontre de points de comparaison. Cela signifie qu'un chemin illégal ne peut, sauf heureux hasard, être détecté que s'il contient un point de comparaison. En d'autres termes, toute faute permanente qui empêche le calcul du code d'un état point de comparaison est structurellement indétectable.

Par exemple, si seulement un point de comparaison est implanté avec le code d'état « 000 », un collage à 1 d'une des entrées du registre d'état ne peut être détecté. Différentes solutions peuvent être proposées. Par exemple, un deuxième point de comparaison peut être implanté avec le code d'état « 111 ». Ou encore un compteur peut être implanté pour vérifier que le signal de comparaison est actif au moins une fois tous les n cycles, n étant déterminé en fonction du graphe de contrôle et de la latence de détection maximale imposée par le choix des points de comparaison. Mais évidemment, ces différentes solutions ont un coût matériel.

III.4.3.2. Mise en défaut structurelle pour I-Tool

Lorsqu'on considère la méthode I-Tool, la signature est vérifiée à chaque cycle d'horloge. Malheureusement, l'utilisation de références implicites induit une corrélation entre la signature de référence après un état et le code de cet état. Pour une implantation I-Tool, cette corrélation se traduit par la fonction logique du comparateur de vérification de la signature détaillée ci-dessous :

$$Icmp = b_1 + b_3 + ... + b_{\upsilon-1} + (b_0 \oplus e_0) + ... + (b_{\upsilon-2} \oplus e_{\upsilon-2})$$

Ici, b_i représente le $i+1^{ième}$ bit de la signature courante (signature calculée en temps réel), e_i représente le $i+1^{ième}$ bit du code d'état courant, + représente l'opérateur logique OU et $\oplus$ l'opérateur logique OU-Exclusif.

Ceci signifie que la signature courante est considérée correcte si et seulement si :

$b_i = 0$ lorsque i est impair,
$b_i = e_i$ lorsque i est pair.

Aussi, si un MISR à OU-Exclusifs internes est utilisé comme c'est la cas dans [Robi 92a], la signature courante est considérée correcte si et seulement si :

$$\begin{aligned} b_i(t) &= Pol_i \,.\, b_{\upsilon-1}(t\text{-}1) \oplus e_i \oplus b_{i-1}(t\text{-}1) = 0 && \text{pour i impair} \\ b_i(t) &= Pol_i \,.\, b_{\upsilon-1}(t\text{-}1) \oplus e_i \oplus b_{i-1}(t\text{-}1) = e_i && \text{pour i pair} \end{aligned} \qquad (5)$$

$b(t)$ représente la signature calculée en temps réel après l'état de code e, $b(t\text{-}1)$ représente la signature calculée en temps réel avant l'état de code e et Pol_i représente le $i+1^{ième}$ bit du polynôme diviseur.

Etant donné que υ est pair, si $b(t\text{-}1)$ est correcte, $b_{\upsilon-1}(t\text{-}1)$ est égal à 0. La nouvelle signature $b(t)$ est donc considérée correcte si et seulement si :

$e_i = b_{i-1}(t\text{-}1)$ pour les valeurs impaires de i,

quelque soit la valeur de e_i pour les valeurs paires de i, puisque (5) est toujours vrai.

Ceci signifie que toutes les fautes permanentes modifiant uniquement les bits d'indice pair du code d'état sont indétectables quelque soit la valeur des entrées de la logique de calcul de l'état

suivant. De même, les fautes transitoires ne modifiant que les bits d'indice pair tout le temps où elles sont actives sont indétectables. Ce deuxième type de fautes ne contient pas seulement des fautes structurellement indétectables au sens propre du terme, puisque pour certaines de ces fautes transitoires, la non détection dépend de leur durée et de la valeur des entrées de la logique de calcul d'état suivant au moment où elles se produisent.

III.4.4. Conclusion

L'inégalité (4) tend à montrer que les méthodes ASYL-SdF Ajs et NoAjs ont un pouvoir de détection plus important que la méthode I-Tool lorsque la fréquence d'occurrence des fautes transitoires, ou d'activation des fautes permanentes, est faible, c'est-à-dire lorsque m=1. C'est généralement le cas pour les applications visées par ce type de technique de vérification d'un flot de contrôle. Par contre, lorsque la fréquence d'occurrence des fautes est plus importante, c'est-à-dire lorsque $m \geq 2$, la méthode I-Tool a une probabilité de masquage d'erreurs plus faible.

Une mauvaise distribution des valeurs de référence tend à augmenter les probabilités de masquage d'erreurs, comme cela est indiqué pour l'exemple ex4. Toutefois, le plus gênant est certainement les fautes structurellement indétectables, en particulier pour la méthode I-Tool où une solution à ce problème semble difficile à mettre en place. Une étude expérimentale succincte des taux de couverture est fournie dans la section suivante et donne une idée (approximative) de l'impact de ces différentes probabilités de masquage pour les fautes permanentes.

III.5. RÉSULTATS EXPÉRIMENTAUX

Cette section présente les résultats expérimentaux obtenus pour une vingtaine d'exemples en ce qui concerne les résultats en surface et vitesse, une centaine en ce qui concerne les algorithmes de réparation des NSC-graphes et quelques exemples pour l'évaluation des taux de couverture, des collages à 0 et à 1, par simulation de fautes.

La synthèse des différents exemples a été effectuée avec l'outil ASYL-SdF en utilisant une bibliothèque de cellules standard CMOS 1,2µm de VLSI Technology. Le placement et le routage ont été effectués avec les outils COMPASS, et la simulation de fautes avec l'outil SUNRISE. Pour le résultat des implantations utilisant la méthode de S. Robinson, la réparation des NSC-graphes et le S-codage ont été réalisés avec l'outil I-Tool[1]. La synthèse proprement dite de l'architecture correspondante a été intégrée dans la version non commerciale d'ASYL-SdF pour nous permettre d'analyser les avantages et les inconvénients des méthodes à base d'ajustements et de références implicites.

Les résultats en surface, chemin critique, et taux de couverture ont été obtenus, pour les architectures CFC Ajs et NoAjs, en implantant automatiquement seulement deux points de comparaison. Les solutions proposées au paragraphe III.4.3.1 pour la réduction du nombre de fautes structurellement indétectables n'ont pas été appliquées lors de ces implantations.

III.5.1. Modification du graphe

Un tableau complet des résultats obtenus est donné en annexe III.3. Quelques résultats sont fournis dans le tableau III.1. Les colonnes de la partie « Graphe d'origine » du tableau

[1] Nous remercions S. Robinson pour nous avoir aimablement fourni l'outil I-Tool.

fournissent le nombre d'états, le nombre de transition, le nombre de classes de C-équivalence et la cardinalité de la plus grande classe pour le graphe d'origine. Les colonnes de la partie « Résultats ASYL-SdF » fournissent le nombre d'états du graphe réparé avec l'outil ASYL-SdF, le nombre d'états ajoutés par la réparation, et le nombre de transitions du graphe réparé. Enfin, la partie « I-Tool » donne des informations sur les résultats obtenus avec la méthode de S. Robinson : la colonne « Gmax » donne le nombre d'états du graphe résultant de l'expansion maximale, et la colonne « Delta » donne le résultat de la différence entre le nombre d'états obtenu avec ASYL-SdF et le nombre d'états obtenu avec I-Tool (ASYL-SdF moins I-Tool). Pour l'exemple bbsse, cela signifie que ASYL-SdF a créé un SC-graphe comportant 34 états, tandis que I-Tool a créé un SC-graphe comportant 37 états.

Tableau III.1 - Réparation des NSC-graphes.

Nom	Graphe d'origine					Résultats ASYL-SdF			I-Tool	
	Type	Etats	Trans.	Classes	Card. Max.	Etats	Delta	Trans.	Gmax	Delta
bbit3	Moore	504	1279	481	3	527	23	1389	774	-
bbsse	Mealy	16	56	4	13	34	18	141	42	-3
bbtas	Mealy	6	24	1	6	10	4	40	12	0
dk15	Mealy	4	32	1	4	12	8	96	12	0
gestionfifo	Mealy	7	13	4	4	12	5	29	13	0
imec1	Moore	101	327	68	15	148	47	494	158	0
imec2	Moore	409	1383	247	6	795	386	2947	859	0
s1	Mealy	20	107	1	20	64	44	363	80	-4
s820	Mealy	25	232	1	25	104	79	886	107	3
scf	Mealy	121	166	120	2	122	1	167	151	-
seqbis	Mealy	141	489	74	60	229	88	941	265	5

Les résultats peuvent être très variables d'un exemple à l'autre. Par exemple, bbit3 ne nécessite la création que de 23 états pour un graphe d'origine comportant 504 états, tandis que s820 nécessite la création de 79 états pour un graphe d'origine n'en comportant que 25. Ceci est très lié à la complexité (ou plutôt à la connectivité) du NSC-graphe. Une estimation a priori de l'intérêt du coût de la réparation peut être de calculer le rapport : d_c = nombre de transitions / nombre d'états. Une méthode un peu plus précise est de déterminer la cardinalité de la plus grande classe de C-équivalence : plus cette cardinalité est proche du nombre d'états du graphe, et plus la réparation sera coûteuse. Toutefois, le temps nécessaire à la réparation d'un NSC-graphe est généralement de l'ordre de la seconde ou de la minute, ce qui permet de savoir très rapidement si une implantation à base de vérification d'un flot de contrôle par ajustement implicite a des chances d'être intéressante ou non par rapport à la duplication (du point de vue du coût matériel des dispositifs de détection).

Un autre point intéressant est que la résolution des C-équivalences locales a permis de réparer totalement plus de 35% des graphes testés. Ceci confirme l'intérêt d'appliquer un traitement particulier à ce type de C-équivalences, lors de la réparation.

Enfin, les résultats obtenus avec ASYL-SdF sont très semblables à ceux obtenus avec l'outil I-Tool. Tous les résultats n'ont pu être obtenus avec I-Tool en raison d'une différence de format de spécification des contrôleurs. L'investissement en temps pour traduire la spécification d'un format à l'autre peut être assez important pour les gros contrôleurs, et n'a pas été jugé indispensable étant donné la quantité de résultats déjà disponible. Le nombre d'états du graphe résultant de l'expansion maximale a été calculé, ce qui explique qu'un résultat « Gmax » soit mentionné même lorsque le nombre d'états final du graphe réparé par I-Tool n'est pas disponible.

Une étude croisée a aussi été menée afin de qualifier la qualité de la réparation non plus vis à vis du nombre d'états ajoutés, mais plutôt vis à vis de la qualité structurelle du graphe généré,

c'est-à-dire de la facilité à synthétiser efficacement le graphe réparé. Nous entendons par étude croisée, le S-codage et la synthèse avec la méthode I-Tool d'un graphe réparé avec ASYL-SdF, et vice versa, le S-codage et la synthèse pour l'architecture CFC NoAjs de graphes réparés avec I-Tool.

La figure III.18 donne une synthèse des résultats. Il s'agit du gain en surface dû à la réparation avec ASYL-SdF par rapport à la réparation avec I-Tool, et ce pour les deux S-codages. Pour la figure III.18a, nous avons donc effectué une synthèse avec la méthode I-Tool de graphes réparés avec ASYL-SdF et avec I-Tool. Le résultat indiqué est calculé de la manière suivante : $(R_{I\text{-}Tool} - R_{Asyl}) / R_{I\text{-}Tool}$ où $R_{I\text{-}Tool}$ est le résultat obtenu en réparant le graphe avec l'outil I-Tool, et R_{Asyl} est le résultat obtenu en réparant le graphe avec ASYL-SdF. Nous avons effectué de même pour la figure III.18b, mais avec une implantation CFC NoAjs.

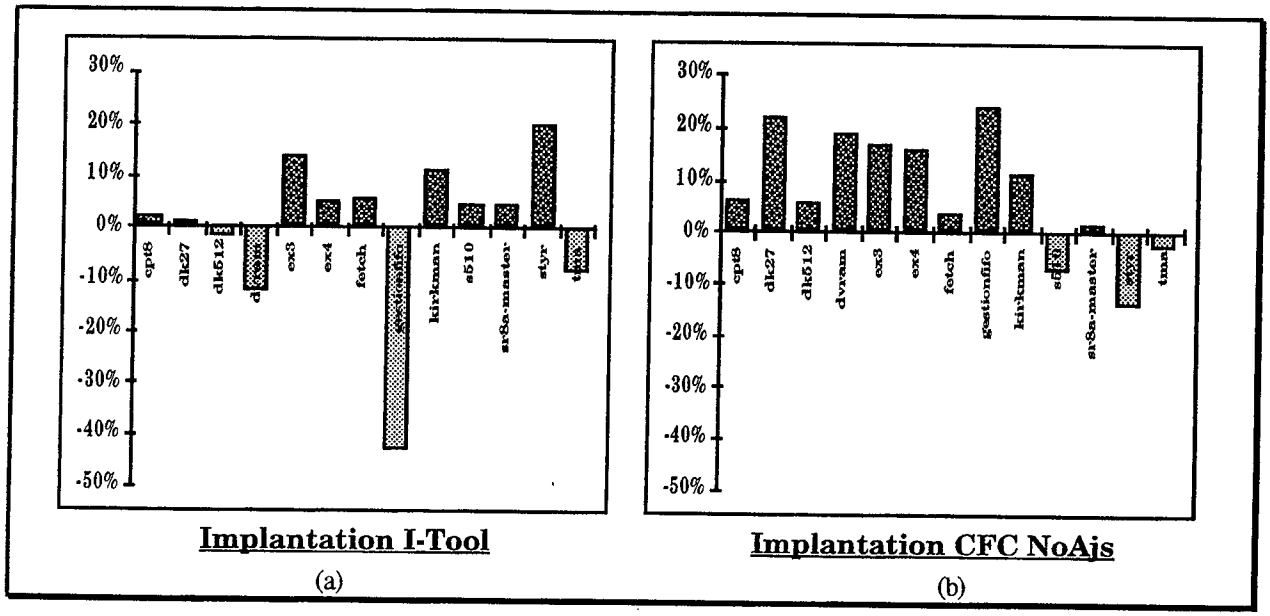

Figure III.18 - Gain en surface dû à la réparation du graphe avec ASYL-SdF comparée à la réparation avec I-Tool : lors d'une implantation avec l'architecture I-Tool (a) et CFC NoAjs (b).

Cette figure montre clairement le gain en surface généralement lié à la réparation effectuée par ASYL-SdF. Toutefois, ce gain semble ne pas provenir seulement de meilleures caractéristiques structurelles, et ces résultats sont à relativiser. En effet, les procédures de S-codage utilisées par ASYL-SdF et I-Tool sont sensibles à l'ordre des états en entrée. Selon l'ordre d'énumération des états dans la spécification du contrôleur, le résultat du S-codage n'est pas forcément le même. Ceci est illustré par l'exemple ex4 dont le graphe de contrôle d'origine est un SC-graphe, c'est-à-dire qu'il ne nécessite pas de réparation. Aussi, la différence entre le graphe fourni par I-Tool et celui fourni par ASYL-SdF n'est pas structurelle, mais provient uniquement de l'ordre dans lequel les états et les transitions sont énumérés (cet ordre est modifié même lorsque le graphe n'a pas à être réparé). Ceci indique que le gain en surface dû à la réparation des graphes effectuée par ASYL-SdF, par rapport à celle effectuée par I-Tool, peut non seulement provenir de meilleures caractéristiques structurelles, mais aussi de l'ordre dans lequel ASYL-SdF présente les transitions du graphe réparé.

III.5.2. Résultats en surface

Les résultats complets en surface sont donnés en annexe III.5. Cela représente une soixantaine d'exemples pour les architectures DPX Seq (Fig. I.10, section I.4) et CFC Ajs et une

vingtaine d'exemples pour les implantations CFC NoAjs et I-Tool. Un échantillon des exemples les plus intéressants est donné dans les tableaux III.2 et III.3.

Tableau III.2 - Caractéristiques des exemples du tableau III.3.

Nom	Nb. Etats / Nombre de bits k	Etats ajoutés (I-tool / ASYL-SdF)	Nombre de bits ajoutés (I-tool / ASYL-SdF)	Trans. / Ajust. (graphe initial)
cpt8	8 / 3	14 / 14	3 / 2	37 / 32%
dk27	7 / 3	3 / 3	1 / 1	14 / 36%
dk512	15 / 4	4 / 4	2 / 1	30 / 50%
dvram	35 / 6	6 / 6	2 / 0	47 / 28%
ex3	10 / 4	13 / 9	2 / 1	37 / 49%
ex4	14 / 4	0 / 0	0 / 0	21 / 24%
fetch	26 / 5	1 / 1	3 / 0	35 / 29%
gestionfifo	7 / 3	5 / 5	1 / 1	13 / 38%
kirkman	17 / 5	1 / 1	1 / 0	370 / 1%
s510	47 / 6	5 / 5	4 / 0	77 / 38%
sr8a-master	11 / 4	9 / 9	2 / 1	24 / 42%
styr	30 / 5	56 / 58	5 / 2	166 / 38%
tma	20 / 5	6 / 8	1 / 0	44 / 43%

Le tableau III.2 donne les caractéristiques principales des exemples du tableau III.3. La première colonne indique le nombre d'états du graphe d'origine et le nombre de variables d'état correspondant. La seconde colonne indique le nombre d'états ajoutés par la réparation du graphe pour I-Tool et ASYL-SdF. La troisième colonne donne le nombre de variables d'état ajoutées par la réparation du graphe pour ASYL-SdF, et le nombre de variables d'état ajoutées par la réparation et le S-codage pour I-Tool. Le S-codage utilisé par ASYL-SdF n'ajoute par de bits au code d'état étant donné qu'il s'agit d'un S-codage compact optimisé. Enfin, la dernière colonne indique le nombre de transitions dans le graphe d'origine, et le pourcentage de transitions nécessitant un ajustement.

Tableau III.3 - Résultats en surface : surcoût par rapport à l'architecture simplex.

Nom	DPX Seq	CFC Ajs	CFC NoAjs	I-Tool	Meilleur cas	Moniteur
cpt8	82%	78%	175%	49%	ASYL/I-tool (46%)	72%
dk27	130%	172%	147%	128%	ASYL/I-tool (125%)	148%
dk512	107%	121%	74%	57%	I-tool/I-tool (57%)	61%
dvram	83%	57%	74%	68%	I-tool/I-tool (68%)	39%
ex3	116%	173%	109%	124%	ASYL/I-tool (108%)	79%
ex4	103%	84%	42%	45%	ASYL/I-tool (37%)	51%
fetch	93%	88%	87%	105%	ASYL/ASYL (87%)	56%
gestionfifo	103%	118%	151%	120%	I-tool/I-tool (120%)	132%
kirkman	44%	41%	20%	49%	ASYL/ASYL (20%)	36%
s510	96%	88%	67%	13%	ASYL/I-tool (7%)	23%
sr8a-master	115%	109%	154%	117%	ASYL/I-tool (106%)	89%
styr	65%	91%	537%	176%	ASYL/I-tool (120%)	26%
tma	84%	112%	131%	53%	I-tool/I-tool (53%)	51%

Le tableau III.3 indique le surcoût de ces différentes architectures par rapport à l'architecture simplex (Fig. I.8, paragraphe I.2.2.3). Les zones grisées indiquent le meilleur résultat en surface pour chaque exemple, pour une synthèse effectuée uniquement avec ASYL pour les implantations CFC Ajs et CFC NoAjs, et une synthèse effectuée uniquement avec I-Tool pour l'implantation I-Tool. La colonne « meilleur cas » donne le résultat de synthèses croisées. Elle indique quelle est la combinaison « réparation / architecture ciblée » qui a donné le meilleur résultat en surface pour les deux architectures basées sur les ajustements implicites. Ainsi, la mention « ASYL/I-Tool » indique que le meilleur résultat entre une implantation CFC NoAjs et une implantation I-Tool a été obtenu en réparant le graphe avec ASYL-SdF et en le S-codant avec I-Tool (lorsque l'exemple est S-codé avec ASYL-SdF, il s'agit d'une implantation CFC NoAjs, et lorsqu'il est S-codé avec I-Tool, il s'agit d'une implantation I-Tool). La colonne « moniteur » indique le coût du moniteur

(MISR, signal ε, etc ...), utilisé par l'architecture CFC NoAjs, par rapport à l'implantation simplex.

On constate que la vérification d'un flot de contrôle peut être peu coûteuse en surface, comme c'est le cas pour les exemples s510 et kirkman, ou au contraire d'un coût prohibitif. Cela dépend de différents paramètres. Le premier est évidemment le coût en nombre d'états de la réparation du graphe, ou en nombre de transitions du placement des ajustements. Ensuite intervient la taille du contrôleur d'origine. Pour les petits contrôleurs, le coût des dispositifs de détection, et en particulier du MISR, est relativement très important. On peut d'ailleurs constater que pour un exemple comme ex4, le coût du moniteur a été compensé par un gain en surface au niveau de la logique de calcul d'état suivant et de calcul des sorties. Enfin, il apparaît que malgré le nombre important de bits nécessaires au S-codage effectué par I-Tool, les implantations résultantes ont des surfaces souvent intéressantes. Si tous les résultats que nous avons obtenus, et détaillés dans ce document, pour les architectures CFC Ajs et NoAjs correspondent à des implantations utilisant le nombre minimum de bits pour le codage des états, l'algorithme de S-codage implanté dans ASYL-SdF permet toutefois un codage sur un plus grand nombre de bits. Ceci permet d'augmenter le champ d'exploration des S-codages possibles, et de trouver dans certains cas une solution respectant mieux les contraintes d'adjacence. Ainsi, en autorisant une meilleure optimisation des équations Booléennes, un tel codage sur un nombre non minimum de bits peut permettre un gain en surface par rapport à une implantation basée sur un codage compact (et ce malgré le nombre plus important de bascules utilisées pour le registre d'état et le MISR). Toutefois, n'ayant pas d'heuristique pour la recherche d'un codage intéressant sur un nombre quelconque de bits, l'étude de telles solutions prend beaucoup de temps et n'a pour l'instant pas été effectuée.

III.5.3. Chemins critiques

Le tableau III.4 présente les résultats du point de vue du chemin critique. La colonne « simplex » donne le chemin critique, en nanosecondes, de l'implantation simplex. Les colonnes suivantes indiquent la valeur de l'accroissement du chemin critique en nanosecondes, pour chaque architecture comparée à l'implantation simplex.

Tableau III.4 - Résultats en vitesse : augmentation du chemin critique par rapport à l'architecture simplex en nanosecondes.

Nom	simplex chemin critique	DPX Seq	CFC Ajs	CFC NoAjs	I-Tool
cpt8	9,6	1,5	3,6	3,3	1,1
dk27	4,2	0,4	2,9	1,2	1,1
dk512	10,7	2,1	3,4	2,1	1,6
dvram	9,7	1,6	3,4	1,9	2,1
ex3	8,2	1,8	3,9	2,1	2,3
ex4	7	1,5	3	0	1,8
fetch	9,4	2,2	3,1	1,1	1,4
gestionfifo	4,7	0,8	2,2	1,1	0,8
kirkman	9,4	0,7	0,2	0,1	0,2
s510	14,6	3,1	4	2,1	0,1
sr8a-master	6	0,9	2,9	1,5	1,5
styr	14,8	2,8	4,6	5,9	4,1
tma	10	1	3,1	1,9	1,1

On constate que l'accroissement du chemin critique est généralement assez faible quelque soit l'architecture considérée. Plus précisément, la plus coûteuse du point de vue temporelle est l'architecture CFC Ajs, principalement du fait de la présence des multiplexeurs et des portes

OU-Exclusif en entrée du MISR (ce qui est lié à la synchronisation choisie). La très faible augmentation du chemin critique pour l'exemple kirkman est due au fait que celui-ci correspond, dans l'implantation simplex, à un chemin entre le registre d'états et une sortie primaire. Pour l'exemple ex4 et l'implantation CFC NoAjs, la non augmentation du chemin critique provient du fait que le codage obtenu par la procédure de S-codage a permis d'améliorer le chemin critique de la logique de calcul de l'état suivant. Ceci compense le délai induit par l'augmentation de la sortance, au niveau des sorties de cette même logique, due à l'implantation du MISR. Enfin, l'augmentation indiquée pour l'architecture CFC NoAjs, pour les très petits exemples dk27 et gestionfifo, correspond en fait au double du temps de calcul du signal ε moins le chemin critique de l'implantation simplex. Le temps de calcul du signal ε étant plus important que la demi-période d'horloge imposée par le chemin critique réel, on considère que le chemin critique de l'implantation est fixé par ε. Ce sont les deux seuls cas où le calcul du signal ε ne peut pas se faire en un demi-cycle pour une fréquence maximum.

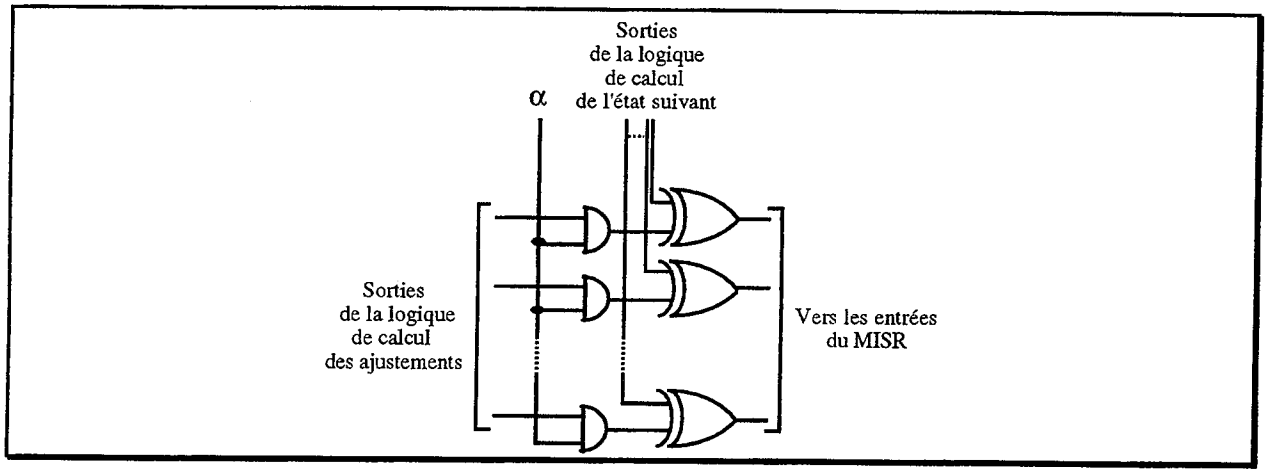

Fig. III.19 - Alternative d'implantation pour le dispositif d'ajustement.

Une possibilité d'amélioration de l'implantation en vue d'une légère optimisation du chemin critique de l'architecture CFC Ajs est la suivante : plutôt que d'implanter l'ajustement de la signature à base de multiplexeurs comme indiqué au paragraphe III.3.1, il est possible d'utiliser des portes ET comme indiqué en figure III.19. Ce type d'implantation permet en effet de réduire la sortance des sorties de la logique de calcul de l'état suivant, et d'éviter l'utilisation de multiplexeurs. Ce schéma d'implantation n'a pas encore été intégré au flot de synthèse de l'architecture CFC Ajs. Son impact sur le chemin critique n'a donc pas encore été chiffré.

III.5.4. Taux de couverture

Les tableaux III.5 et III.6 présentent les résultats des simulations de fautes effectuées. Les fautes simulées sont les collages à 0 et à 1 niveau porte. Le protocole de simulation est le suivant.

L'ensemble des vecteurs d'entrée utilisés a été généré de manière à faire emprunter par le contrôleur l'ensemble des transitions spécifiées dans son graphe de contrôle ([Kara 91]). Ceci nous permet de tester tous les cas possibles de mauvais cheminement dû a des fautes permanentes, tout en limitant le nombre de vecteurs, et donc le temps de simulation.

Le taux de couverture a été obtenu en observant uniquement le signal d'erreur généré par les dispositifs de vérification de la signature (ou de comparaison dans le cas de la duplication). Dans le cas du tableau III.5, le taux indiqué prend en compte les collages pouvant se produire dans toute la

logique, y compris la logique de calcul des sorties. Etant donné que nous nous intéressons ici essentiellement aux erreurs de séquencement, les résultats du tableau III.6 ne prennent en compte que les fautes pouvant se produire dans la logique de séquencement et les dispositifs de détection.

Tableau III.5 - Taux de couverture pour toute la logique (% de fautes détectées).

Nom	DPX Seq	CFC Ajs	CFC NoAjs	I-tool
dvram	74,46	57,02	57,58	12,82
gestionfifo	64,81	11,58	50,00	20,38
kirkman	45,57	19,30	30,26	12,72
s510	87,22	57,83	20,92	13,06

Tableau III.6 - Taux de couverture pour la logique de séquencement et la logique de détection.

Nom	DPX Seq	CFC Ajs	CFC NoAjs	I-tool
dvram	97,65	76,25	77,66	40,09
gestionfifo	93,38	28,53	67,74	51,94
kirkman	97,36	67,52	75,95	77,98
s510	95,89	66,14	38,00	35,07

La simulation de fautes concerne uniquement des fautes permanentes, et fait donc totalement abstraction de la latence de détection : une faute peut être détectée au bout d'un cycle, 10, ou même 100 cycles. De ce fait, elle est assez révélatrice de l'impact des fautes structurellement indétectables.

Les résultats, en particulier ceux du tableau III.6, confirment que la corrélation entre les signatures de référence et les codes d'états courants, pour l'implantation I-Tool, a un impact non négligeable sur l'efficacité de la détection, comparé à l'impact des fautes structurellement indétectables des implantations CFC Ajs et CFC NoAjs. Le mauvais résultat obtenu pour l'exemple gestionfifo avec l'implantation CFC Ajs semble dû à un mauvais choix des états points de comparaison. Leurs codes sont, en effet, à distance 1 l'un de l'autre, ce qui est très pénalisant du point de vue des collages pour un petit exemple tel que celui-ci.

III.6. CONCLUSION

Ce chapitre a présenté les architectures pour la vérification d'un flot de contrôle dont le flot de synthèse a été implanté dans ASYL-SdF. La synthèse des architectures CFC Ajs et CFC NoAjs est totalement automatisée. Toutefois, lors de la synthèse des implantations CFC NoAjs, le S-codage nécessite encore parfois une intervention humaine afin de réduire le temps de codage. Les algorithmes implantés ont été détaillés et leur efficacité démontrée.

L'automatisation de la synthèse de ce type d'implantations a permis d'effectuer une meilleure caractérisation des avantages et des inconvénients de chaque architecture. Les implantations I-Tool sont, par exemple, souvent moins coûteuses en surface que les implantations CFC Ajs et CFC NoAjs, mais au prix d'une moindre efficacité de la détection. De manière plus générale, les résultats obtenus montrent que la vérification d'un flot de contrôle peut être une réelle alternative à la duplication, pour certains types de contrôleurs, et lorsqu'on cherche un compromis entre l'efficacité de la détection et le coût matériel.

Le chapitre suivant présente les travaux effectués dans le domaine du masquage d'erreurs dans les contrôleurs câblés. De même que le présent chapitre pour la détection d'erreurs, il détaille le flot de synthèse de nouvelles implantations permettant de définir des compromis mieux adaptés à tel ou tel type d'application.

CHAPITRE IV - SYNTHÈSE DE CONTRÔLEURS MASQUANT LES ERREURS

Après les aspects détection du chapitre précédent, nous nous intéressons ici aux architectures pour le masquage d'erreurs.

Nous avons vu dans le premier chapitre qu'une solution classique pour le masquage des erreurs simples est la redondance massive, à savoir le triplement avec vote majoritaire. Deux architectures ainsi basées sur le TMR, appelées TMR Tot et TMR Seq, sont présentées dans la section I.4.

Le chapitre II présente les travaux de D.B. Armstrong ([Arms 61]) qui constituent une alternative aux implantations TMR. Cette approche affine le grain de la redondance implantée pour le masquage d'erreurs, en utilisant, en particulier, un code correcteur d'erreurs simples lors du codage des états du contrôleur. Cette implantation plus fine de la redondance permet d'espérer obtenir des implantations matériellement plus économiques, pour un niveau de fiabilité équivalent aux implantations TMR. L'inconvénient est que ce type d'implantations utilise un flot de synthèse plus complexe, devant nécessairement être automatisé pour être accessible au monde industriel.

Le présent chapitre étudie deux flots de synthèse d'une architecture dérivée des travaux présentés au paragraphe II.2.2.2. Cette architecture est appelée architecture SID pour « Single Independent Decoder », c'est-à-dire « décodeur indépendant unique ». Il s'agit en effet d'une architecture basée sur l'utilisation d'un bloc de logique séparé et non répliqué pour la correction des codes d'états erronnés. Les deux flots de synthèse, intégrés à l'outil ASYL, se différencient principalement par la procédure de codage utilisée. Ils ont été évalués du point de vue de la surface, de la vitesse et de la fiabilité des implantations obtenues. Aussi, la section IV.1 présente l'architecture utilisée et les flots de synthèse implantés. La section IV.2 présente le protocole d'évaluation de la fiabilité utilisé pour la comparaison des différentes implantations. Enfin, la section IV.3 donne un résumé des résultats, en comparant en particulier les implantations obtenues avec les deux flots de synthèse aux implantations simplex et TMR.

IV.1. ARCHITECTURE SID

Cette section présente l'architecture SID dérivée des travaux présentés au paragraphe II.2.2.2, ainsi que les flots de synthèse implantés. Le paragraphe IV.1.1 indique rapidement les raisons de notre choix architectural.

IV.1.1. Choix architectural

Les travaux effectués par Armstrong, Frank, Reed, (etc ...) sont principalement basés sur la remarque suivante. Dans le cas du TMR, le nombre total de codes d'états correspondant à un état donné (codes erronés tolérables plus le code valide) est $\tau = 2^{2k}$ si le vote majoritaire est indépendant pour chaque bit du code (une porte majorité par bit du code d'état), $\tau = 3.2^k-2$ si le vote majoritaire se fait globalement au niveau du code (k est défini au paragraphe I.1.3.2). En revanche, lorsqu'un code correcteur d'erreurs simples est utilisé, ce nombre est réduit à $\sigma = n+1$, où n est le nombre de bits du code correcteur d'erreurs ($n \leq 2.k$ si $k > 2$). σ étant très inférieur à τ, on peut espérer qu'une implantation basée sur l'utilisation de codes correcteurs d'erreurs conduira à une redondance mieux ciblée et moins coûteuse qu'avec le TMR.

Les travaux présentés au paragraphe II.2.2.2 montrent toutefois que l'utilisation de codes correcteurs d'erreurs simples, lors du codage des états de la machine, peut conduire à des implantations très diverses. Nous avons vu que [Arms 61] propose une implantation utilisant un décodeur unique, tandis que [Reed 70] préfère implanter autant de décodeurs qu'il y a de fonctions logiques dans le bloc de calcul de l'état suivant. [Meye 71] propose, pour sa part, de distribuer le processus de correction directement au niveau des fonctions de calcul de l'état suivant. L'intérêt des implantations proposées par Meyer et Reed est d'éviter que le bloc de décodage constitue un bloc critique, comme c'est le cas pour l'implantation proposée par Armstrong, c'est-à-dire d'éviter qu'une faute simple dans le bloc de décodage ne puisse être tolérée. Toutefois, une étude publiée dans [Leve 93a] a montré que la distribution du décodage dans le cas de Meyer est très coûteuse en nombre de transistors, du fait de la complexité des fonctions Booléennes de calcul d'état suivant modifiées. Le coût de ce type d'implantation est en effet en moyenne sept fois supérieur à celui de l'architecture simplex pour les contrôleurs, ce qui concorde avec les chiffres annoncés par Frank pour les compteurs. Plus précisément, sur 80 exemples synthétisés, le surcoût en nombre de transistors par rapport à l'architecture simplex est supérieur à 500% pour 78% des exemples, et supérieur à 900% pour 28% d'entre eux (annexe IV.1). Enfin, même si elle n'a pas été étudiée en particulier, l'option proposée par Reed semble, elle aussi, coûteuse, du fait de la réplication des fonctions de décodage. Néanmoins, l'utilisation du code Reed-Müller modifié, très bien adapté à ce type d'implantation, peut laisser penser que le coût de la redondance est moindre que dans l'option proposée par Meyer.

Pour notre part, nous nous sommes essentiellement intéressés à la réduction du coût en surface des dispositifs de masquage d'erreurs. Nous avons donc choisi une architecture avec un seul bloc de décodage, qui correspond à l'architecture proposée par Armstrong où les fonctions logiques sont séparées en blocs disjoints, avec une fonction par bloc ($p = 1$ dans la figure II.19). Deux procédures de codage ont été proposées pour cette architecture. La première consiste à suivre les mêmes principes que le codage d'états classique, c'est-à-dire essayer de satisfaire des contraintes d'optimisation, afin de minimiser l'ensemble des fonctions de calcul de l'état suivant, en n'utilisant toutefois que des mots du code correcteur d'erreurs choisi (paragraphe VI.1.3). La seconde, elle, consiste à ajouter des bits de contrôle à des codes d'états obtenus par l'intermédiaire de la procédure de codage classique (paragraphe VI.1.4).

VI.1.2. Architecture synthétisée

L'architecture SID (Fig. IV.1) est fondée sur les principes suivants. Pour permettre la tolérance d'un bit erroné dans le registre d'état, un code correcteur d'erreurs simples, ayant une distance de Hamming minimale égale à trois, est utilisé lors du codage de la machine d'états finis. Ensuite, afin de garantir aussi la tolérance d'une faute simple dans le bloc de calcul de l'état suivant, l'implantation évite tout partage de logique entre les différentes fonctions qui le composent. Enfin, la tolérance proprement dite est obtenue par l'implantation du bloc dit de décodage, qui corrige les codes d'états erronés. Ce bloc est connecté en sortie du registre d'état, en entrée de la logique de sortie et en entrée de la logique d'état suivant.

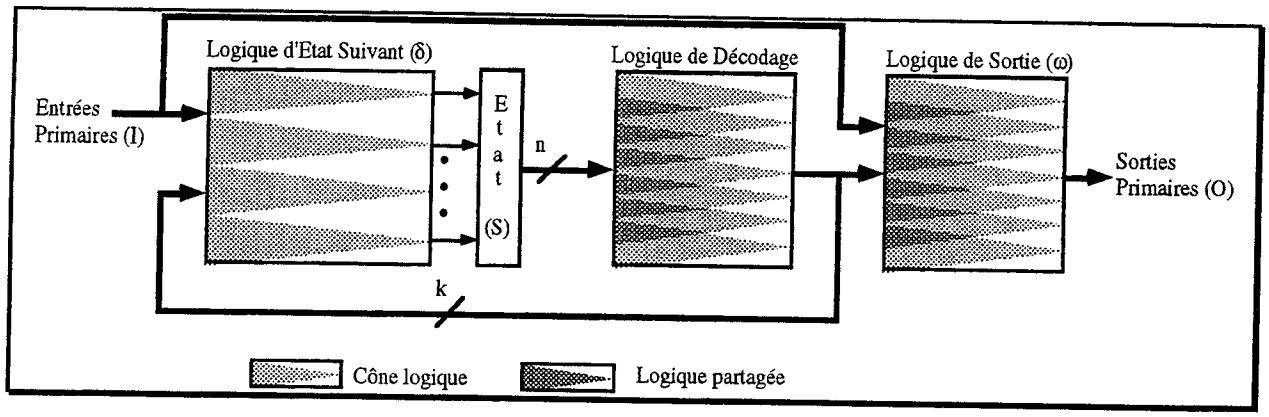

Figure IV.1 - Architecture SID (« Single Independent Decoder » : décodeur indépendant unique).

L'architecture SID permet donc de tolérer les erreurs simples dans la logique de séquencement. Comme pour l'implantation TMR Seq, elle ne tolère pas nécessairement les fautes se produisant dans le décodeur et la logique de sortie. Enfin, comme pour l'architecture TMR Seq et contrairement à l'architecture TMR Tot, le circuit ne nécessite pas de réinitialisation pour le recouvrement d'erreurs dues à des fautes transitoires.

Du fait de l'utilisation d'un code correcteur d'erreurs simples, le nombre de bits du registre d'états est n ($n \leq 2.k$ si $k > 2$). Le nombre d'entrées du bloc de décodage est donc n. Le nombre de bits nécessaires en sortie du décodeur est k, car seulement k bits sont nécessaires pour coder tous les états de la machine. Toutefois le nombre de sorties du bloc de décodage sera soit égal à k, soit élément de l'intervalle [k,n] selon le flot de synthèse choisi.

IV.1.3. Flot de synthèse dit « Global »

Premier flot de synthèse implanté pour l'architecture SID, le flot dit « Global » fut présenté dans [Leve 93a]. Il repose sur la procédure de codage « Global », qui cherche avant tout à minimiser le coût de la redondance au niveau de la logique d'état suivant. L'algorithme de codage a été décrit de manière détaillée par R. Leveugle dans [Leve 93b]. Nous n'en rappellerons ici que les principes.

IV.1.3.1. Algorithme de codage

Lorsqu'on considère un codage d'état utilisant un code correcteur d'erreurs simples, c'est-à-dire un code dont la distance de Hamming minimale entre chaque mot est égale à 3, les procédures de codage classique ne peuvent pas être réutilisées directement. Par exemple, prenons le cas de quatre états éléments d'une même contrainte d'adjacence. Un algorithme classique de codage

essaierait de placer ces quatre états sur un même 2-cube, ce qui correspond à une distance maximale égale à 2 entre leurs codes binaires. Aussi, si on veut assurer une optimisation globale des n bits du code d'état, il faut réécrire la procédure de codage de manière à assurer différemment la minimisation locale des distances, en respectant la contrainte de distance minimale égale à 3 entre deux codes d'états.

		Code
1:	0	1
2:	7	2
3:	30	3
4:	25	4
5:	43	5
6:	44	6
7:	53	7
8:	50	8
9:	102	9
10:	97	10
11:	120	11
12:	127	12
13:	77	13
14:	74	14
15:	83	15
16:	84	16

(a)

Code	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	3	4	3	4	3	4	3	4	3	4	7	4	3	4	3
2	3	0	3	4	3	4	3	4	3	4	7	4	3	4	3	4
3	4	3	0	3	4	3	4	3	4	7	4	3	4	3	4	3
4	3	4	3	0	3	4	3	4	7	4	3	4	3	4	3	4
5	4	3	4	3	0	3	4	3	4	3	4	3	4	3	4	7
6	3	4	3	4	3	0	3	4	3	4	3	4	3	4	7	4
7	4	3	4	3	4	3	0	3	4	3	4	3	4	7	4	3
8	3	4	3	4	3	4	3	0	3	4	3	4	7	4	3	4
9	4	3	4	7	4	3	4	3	0	3	4	3	4	3	4	3
10	3	4	7	4	3	4	3	4	3	0	3	4	3	4	3	4
11	4	7	4	3	4	3	4	3	4	3	0	3	4	3	4	3
12	7	4	3	4	3	4	3	4	3	4	3	0	3	4	3	4
13	4	3	4	3	4	3	4	7	4	3	4	3	0	3	4	3
14	3	4	3	4	3	4	7	4	3	4	3	4	3	0	3	4
15	4	3	4	3	4	7	4	3	4	3	4	3	4	3	0	3
16	3	4	3	4	7	4	3	4	3	4	3	4	3	4	3	0

(b)

Figure IV.2 - Les 16 premiers mots du code (a) et la matrice des distances associée (b).

Ceci a conduit R. Leveugle à définir un code correcteur d'erreurs simples spécifique, dont les mots sont ordonnés, et ayant les propriétés suivantes [Leve 93b] :

P1 : La distance minimale entre deux mots du code est 3.
P2 : Le nombre total de bits, n, nécessaire pour représenter les mots du code est le minimum connu ([Pete 72]), parmi les codes correcteurs d'erreurs simples, pour le nombre de mots générés.
P3 : La distance de Hamming entre deux mots successifs du code est 3.
P4 : La somme des distances entre 4 ou 8 mots successifs est minimale si le dernier d'entre eux est, dans la séquence des mots du code, à une position multiple de 4 ou 8 respectivement.

Les seize premiers mots du code résultant sont fournis en figure IV.2a. Dans cette figure, la valeur 7 du deuxième mot du code représente le vecteur binaire « 000...0111 ». La figure IV.2b donne la matrice des distances de Hamming pour ces seize premiers mots. On peut constater qu'ils respectent bien les quatre propriétés citées ci-dessus.

Après génération du nombre requis de mots du code spécifique, la liste des contraintes d'adjacence est créée et ordonnée de manière décroissante selon leur susceptibilité d'apporter un gain d'optimisation. Ensuite la liste des états est ordonnée, en fonction des contraintes d'adjacence et des propriétés de distance des mots du code, de telle sorte que la somme des distances entre les codes d'états éléments d'une même contrainte soit minimisée. Le $i^{ème}$ mot de la liste des mots du code est alors affecté au $i^{ème}$ état de la liste ordonnée des états.

Afin d'utiliser au mieux les propriétés P3 et P4, le codage Global trie la liste des états en recherchant en premier lieu les inclusions de contraintes. Par exemple, la contrainte d'adjacence sur les états (e1, e3) est incluse dans la contrainte (e1, e2, e3, e6), et elles seront toutes deux remplacées par une contrainte ordonnée (e1, e3, e2, e6). En second lieu, l'algorithme va rechercher les intersections. Par exemple, les contraintes ordonnées (e1, e3, e2, e6) et (e3, e1, e4, e5) obtenues après recherche des inclusions aboutiront à la liste d'états (e5, e4, e1, e3, e2, e6). Le

programme que nous avons implanté respecte ces deux phases, à savoir, dans un premier temps, la recherche des inclusions et l'ordonnancement des états des contraintes en fonction de ces inclusions, puis, dans un second temps, la recherche des intersections avec permutation éventuelle.

IV.1.3.2. Génération du bloc de décodage

Une fois le codage Global effectué, la liste des codes binaires à distance 1 des codes d'états générés est construite. Il s'agit de la liste des codes d'états erronés à tolérer. Si on appelle n le nombre de bits des codes d'états et S le nombre d'états du contrôleur, le nombre de codes ainsi généré est n*S. Les équations du bloc de décodage sont alors définies par la table de vérité suivante : tout code erroné à distance 1 d'un code valide est corrigé en ce code valide. Prenons par exemple le cas d'un contrôleur ayant deux états S_1 et S_2 codés 000 et 111. La liste des codes erronés pour S_1 est 100, 010 et 001. La liste des codes erronnés pour S_2 est 011, 101 et 110. Dans la table de vérité du décodeur de ce contrôleur, la valeur associée en sortie à 000, 001, 010 et 100 est 000, comme indiqué par le tableau IV.1.

Tableau IV.1 - Table de vérité d'un décodeur pour deux états (codage Global).

Entrées du décodeur			Sorties du décodeur		
e_1	e_2	e_3	s_1	s_2	s_3
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
1	0	0	0	0	0
0	1	1	1	1	1
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

Dans le cas où un code ne correspond ni à un code d'état, ni à un code à distance 1 d'un code d'état, la valeur qui lui est associée en sortie est φ-Booléenne, car l'occurence d'un tel code en entrée du décodeur correspond à un cas de dépassement des capacités de masquage d'erreurs du code correcteur utilisé (code correcteur d'erreurs simples).

IV.1.3.3. Flot de synthèse

Une fois la table de vérité du décodeur générée, les équations de décodage proprement dites sont générées, ainsi que les équations de calcul de l'état suivant et de calcul des sorties. Afin d'éviter toute logique partagée entre les différentes fonctions de calcul de l'état suivant, chacune d'entre elles est minimisée, factorisée et décomposée structurellement indépendamment des autres. Par contre, les fautes n'étant pas nécessairement tolérées dans les blocs de décodage et de sortie, ceux-ci peuvent être, séparément, implantés avec de la logique partagée afin de réduire leur taille.

Etant donné que le code correcteur d'erreurs utilisé pour le codage Global n'est pas séparable, c'est-à-dire qu'on ne peut pas séparer les bits d'information des bits de contrôle, le bloc de décodage est généré avec n entrées et n sorties. Toutefois, du fait des simplifications logiques effectuées par la minimisation, les blocs de calcul de l'état suivant et des sorties n'utilisent souvent qu'un sous-ensemble des bits du code d'état pour effectuer leurs calculs. Une procédure permet donc d'éliminer les fonctions inutilisées du bloc de décodage, mais le nombre de fonctions restantes est souvent supérieur à k (minimum requis pour coder tous les états de la machine).

IV.1.4. Flot de synthèse dit « de Hamming »

L'analyse des premiers résultats obtenus avec le codage Global a montré que le bloc de décodage était pour une grande part responsable du coût de l'architecture SID ainsi synthétisée ([Roch 93a] et paragraphe IV.3.1). Nous avons donc cherché à mieux contrôler la structure et la synthèse de ce bloc. Un second flot de synthèse a été étudié et implanté.

IV.1.4.1. Choix du code correcteur d'erreurs

Les codes correcteurs d'erreurs ne manquent pas. Rien que dans ce document, outre le code correcteur d'erreurs simples dont l'algorithme de génération a été défini par R. Leveugle, nous avons cité les codes de Hamming ([Hamm 50]), de Ray-Chaudhuri ([RayC 61]), et de Reed-Müller modifié ([Reed 70]).

Ce qui nous intéresse, c'est un code de préférence séparable afin de pouvoir garantir facilement un nombre minimum de k bits en sortie du bloc de décodage. Il nous faut de plus un code correcteur d'erreurs simples utilisant un nombre minimum de bits de redondance afin de limiter le coût des fonctions de calcul des bits de contrôle à implanter. Enfin, il nous faut un code dont les fonctions de décodage soient les plus simples possible afin d'être à même de bien contrôler la structure et la synthèse du bloc de décodage.

Le code qui nous est apparu le mieux adapté à ces diverses exigences est le code de Hamming. Les autres codes ont différents inconvénients. Par exemple, le code de Reed-Müller permet une implantation très peu coûteuse du bloc de décodage, mais double malheureusement le nombre de bits des mots du code (pour k bits d'information, le code de Reed-Müller utilise k bits de contrôle). Inversement, pour le code de Ray-Chaudhuri, le nombre de bits de contrôle est minimal, toutefois le décodeur est plus complexe à implanter.

IV.1.4.2. Algorithme de codage

L'utilisation du code de Hamming permet de limiter la taille du décodeur. Toutefois, limiter cette taille n'est pas tout. Il est aussi nécessaire d'éviter un surcoût trop important au niveau de la logique de calcul de l'état suivant et de la logique de calcul des sorties. Pour cela, la procédure de codage a été définie comme suit.

Dans une première étape, un codage distance 1 des états est effectué avec la procédure de codage optimisé classique ([Duff 91]). Ceci permet, pour les k bits d'information, de mieux satisfaire les contraintes d'adjacence que dans le cas des n bits du code non séparable lorsqu'on effectue un codage Global. Ceci conduit à une meilleure optimisation du bloc de calcul des sorties, et de k, parmi n, fonctions de calcul de l'état suivant.

Dans une seconde phase, en utilisant une matrice génératrice du code de Hamming ([Hamm 50]), les bits de contrôle sont accolés aux codes d'états générés par le codage classique, afin d'obtenir un code de Hamming correcteur d'erreurs simples. Cette phase correspond à une simple multiplication de vecteurs par une matrice. Avec une telle approche, les fonctions du bloc d'état suivant qui calculent les bits de contrôle ne sont pas pris en compte durant l'optimisation du codage des états.

IV.1.4.3. Génération du bloc de décodage

Si les n-k fonctions de calcul des bits de contrôle ne sont pas optimisées, après le codage, toutefois, les équations du bloc de décodage peuvent être générées en utilisant les propriétés du code de Hamming. Ceci permet de limiter le coût de la logique de décodage et ainsi, a priori, de compenser le surcoût d'implantation des fonctions de calcul des bits de contrôle dû au manque d'optimisation .

Par exemple, soit H la matrice de contrôle de parité du code de Hamming pour k égal à 3 :

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Soient $c_1c_2c_3d_1d_2d_3$ un code d'état où c_i représente un bit de contrôle et d_i un bit d'information. Le décodage est effectué de la façon suivante :

1°)- Multiplication de H par le vecteur du code d'état, pour générer un syndrome $s_1s_2s_3$, identique bit à bit à la colonne de H qui a détecté l'erreur (Tab. IV.2). Les fonctions logiques qui calculent les bits du syndrome sont particulièrement simples. Pour k égal à 3, nous avons par exemple : $s_1 = c_1 \oplus d_2 \oplus d_3$, $s_2 = c_2 \oplus d_1 \oplus d_3$ et $s_3 = c_3 \oplus d_1 \oplus d_2$, où $\oplus$ est l'opération OU-Exclusif.

Tableau IV.2 - Interprétation de la valeur du syndrome.

s1	s2	s3	signification	s1	s2	s3	signification
0	0	0	pas d'erreur	0	1	1	d1 erroné
0	0	1	c3 erroné	1	0	1	d2 erroné
0	1	0	c2 erroné	1	1	0	d3 erroné
1	0	0	c1 erroné	1	1	1	plus d'une erreur

2°)- Inversion du bit erroné du code d'état, s'il y en a un, en fonction de la valeur du syndrome.

Le décodeur résultant est très simple, et ses équations Booléennes sont faciles à générer. La figure IV.3 indique la structure du décodeur pour k égal à 3. Notons que seuls les bits d'information sont corrigés. Les erreurs sur les bits de contrôle sont ignorées afin de réduire la complexité de la logique.

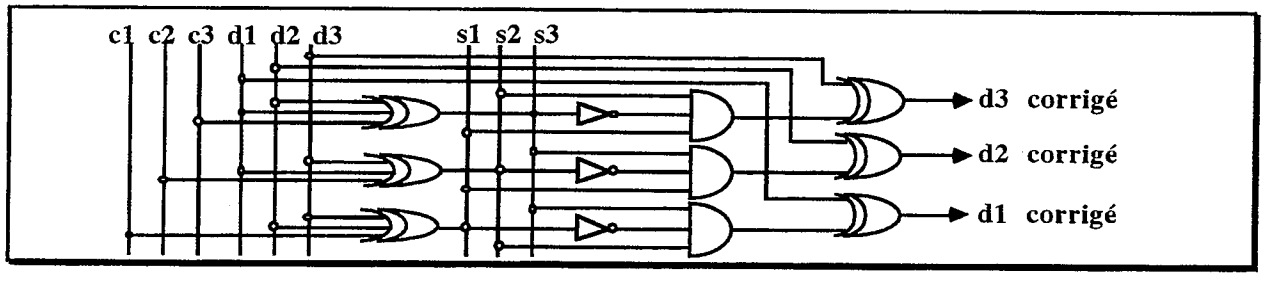

Figure IV.3 - Exemple de décodeur pour k=3.

IV.1.4.4. Flot de synthèse

Après la génération du bloc de décodage, les équations de calcul des sorties et du code d'état suivant sont générées, minimisées, et factorisées comme décrit pour le flot de synthèse Global. Les n équations de calcul du code d'état suivant calculent les k bits du code distance 1 plus les n-k

bits de redondance, à partir du code corrigé de l'état courant (k bits). Enfin, contrairement au flot de synthèse Global, le décodeur est implanté directement, sans minimisation ni factorisation.

IV.1.5. Test de fin de fabrication

Dans le cas d'une machine d'états finis tolérant les fautes simples, un test de fin de fabrication réussi peut être obtenu soit parce qu'il n'y a pas de fautes dans la machine, soit parce que la faute est tolérée. Pour distinguer ces deux possibilités, un dispositif de test et un signal de correction optionnels peuvent être implantés automatiquement au cours de la synthèse de l'architecture SID.

Le signal de correction est positionné lorsqu'un code erroné est corrigé. Ainsi, si durant le test hors-ligne le signal de correction est positionné, cela signifie que le test a échoué. Toutefois, cela ne suffit pas, car le signal de correction ne permet pas de déterminer si le circuit sera capable de tolérer une faute durant son fonctionnement normal.

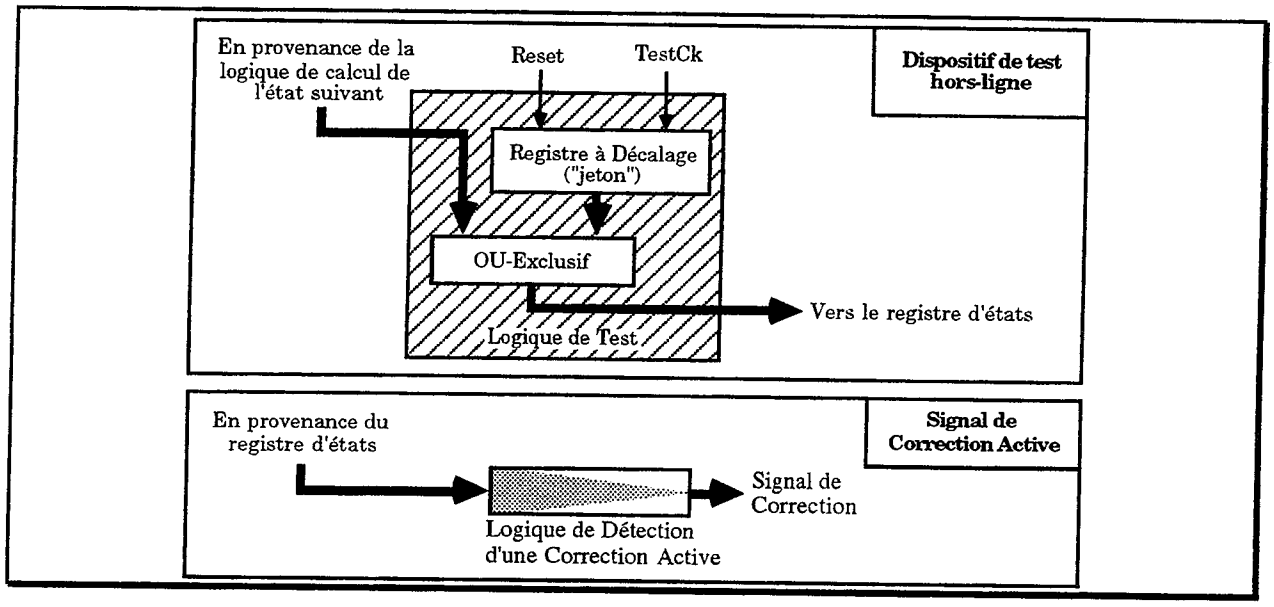

Figure IV.4 - Principe des dispositifs de test intégrés.

Pour cela, le dispositif intégré spécifique de test hors-ligne permet au concepteur de tester la logique redondante en injectant des « fautes » pendant le test. Ce dispositif est composé d'un registre à décalage dont les sorties sont connectées à des portes OU-Exclusif. Ces portes reçoivent aussi en entrée les sorties de la logique de calcul d'état suivant, et sont connectées en sortie aux entrées des bascules du registre d'état (Fig. IV.4). En mode normal, les sorties du registre à décalage sont toutes à « 0 ». Durant le test, l'horloge TestCk est utilisée pour déplacer un jeton (un bit à « 1 » parmi les bits à « 0 »), afin d'injecter une faute en différents points du registre d'état (un seul bit à la fois). Ainsi, un test complet des capacités de tolérance de la machine peut être effectué. De plus, il a été montré que l'implantation de ces dispositifs de test n'introduit aucun mode de défaillance conduisant à un comportement imprévisible du circuit, et que l'occurrence d'une faute dans ces dispositifs durant le fonctionnement normal du circuit est tolérée de la même manière que les fautes dans la logique de calcul de l'état suivant ou dans le registre d'état ([Leve 93c]).

IV.2. EVALUATION DE LA FIABILITÉ

Avant de présenter l'ensemble des résultats obtenus, il est nécessaire de préciser la méthode employée pour comparer la sûreté de fonctionnement des différentes implantations. L'étude qui suit concernera les fautes permanentes.

En ce qui concerne les fautes transitoires, deux remarques peuvent être faites. En premier lieu, comme nous l'avons déjà fait remarquer, les architectures SID et TMR Seq ne nécessitent pas de réinitialisation pour revenir à un état cohérent après occurrence d'une faute transitoire simple, contrairement aux implantations TMR Tot et simplex. En second lieu, le nombre de points mémoires pour l'implantation SID est $n \leq 2.k$, contre 3.k pour une implantation TMR. L'architecture SID est donc moins sensible aux perturbations dues à l'environnement, telles celles provoquées par des ions lourds (« upset »). Ceci laisse prévoir un bon comportement de l'architecture SID vis-à-vis des fautes transitoires simples.

IV.2.1. Métrique

Différentes caractéristiques peuvent être considérées pour évaluer la sûreté de fonctionnement, comme la fiabilité ou la sécurité-innocuité en cas de défaillance (paragraphe I.3.1).

La sécurité-innocuité ne peut pas être précisément évaluée sans informations sur le système complet. Or cette étude a été effectuée en utilisant des exemples dont seule la description de la machine d'états finis était disponible. En conséquence, seule la fiabilité sera considérée dans la suite. Toutefois, notons que le masquage des erreurs de séquencement est importante pour l'amélioration de la sécurité d'un système. En effet, le comportement d'un système après une erreur de séquencement est difficilement prévisible, et une défaillance critique doit donc être envisagée au niveau circuit, diminuant d'autant la sécurité-innocuité du système. Une telle hypothèse de pire cas peut être évitée lorsqu'une implantation tolérant les fautes est utilisée.

Pour ce qui est de la fiabilité, celle-ci peut elle-même être caractérisée par différentes mesures. Etant donné que la machine d'états finis fait partie d'un circuit intégré, il s'agit d'un système non réparable. Nous étudierons donc le temps moyen jusqu'à la première défaillance (« Mean Time to First Failure » : MTFF), et le temps de mission (« Mission Time » : MT). Plus précisément, le MTFF est le temps moyen jusqu'à défaillance en partant de l'état initial (sans faute) du système. Cette mesure est très sensible à la surface du circuit et il est bien connu que l'implantation simplex donne souvent le meilleur résultat. Aussi, afin de donner une évaluation plus significative de la fiabilité, le temps de mission, qui représente le temps de fonctionnement attendu pour un niveau de fiabilité donné, a aussi été calculé. Tous les calculs ont été effectués sous l'hypothèse de taux de défaillance constants et de distributions exponentielles.

IV.2.2. Calcul des taux de défaillance

La procédure d'évaluation sélectionnée durant cette étude pour prévoir les taux de défaillance est le modèle détaillé pour les circuits CMOS VLSI décrit dans la documentation MIL-HDBK-217F éditée par le département de la défense des Etats Unis [Mili 91]. D'après ce modèle, le taux de défaillance d'un circuit est :

$$\lambda_p = \lambda_{BD}\pi_{MFG}\pi_T\pi_{CD} + \lambda_{BP}\pi_E\pi_Q\pi_{PT} + \lambda_{EOS} \text{ défaillances / } 10^6 \text{ heures} \quad (6)$$

où λ_{BD} est le taux de défaillance de base du cœur du circuit, π_{MFG} le facteur de correction pour le processus de fabrication, π_T le facteur de correction pour la température, π_{CD} le facteur de complexité du coeur, λ_{BP} le taux de défaillance de base lié au boîtier, π_E le facteur de correction pour l'environnement, π_Q le facteur de qualité, π_{PT} le facteur de correction dû au type de boîtier, et λ_{EOS} est le taux de défaillance par surcharge électrique. π_{CD} est défini comme $\pi_{CD} = 0,64*(A/0,21)(2/Xs)^2 + 0,36$ où A est la surface du cœur du circuit en cm^2 et Xs la taille caractéristique de la technologie en µm.

Ici nous considérons la machine d'états finis comme une partie d'un circuit plus important, et nous voulons calculer le taux de défaillance uniquement dû à cette partie, de manière à pouvoir départager les différentes implantations étudiées. Dans la formule (6), les deuxième et troisième termes concernent uniquement les caractéristiques du boîtier, de l'environnement, du processus de fabrication et des interfaces d'entrée/sortie, qui ne sont pas dépendantes de l'implantation choisie pour la machine d'états finis. Aussi, seul le premier terme est significatif pour la comparaison des différentes architectures, et le taux de défaillance d'un bloc de logique sans dispositifs de masquage (simplex) peut s'exprimer sous la forme :

$$\lambda = \lambda_{BD}*\pi_{MFG}*\pi_T*0,64*(A/0,21)(2/Xs)^2 + \lambda_{BD}*\pi_{MFG}*\pi_T* 0,36 \quad (7)$$

Le taux de défaillance de base du coeur du circuit λ_{BD} est égal à 0,16 pour de la logique. π_{MFG} est égal à 2 pour un processus de fabrication inconnu. Enfin, considérant les dispositifs CMOS, il est raisonnable de supposer que la température de jonction du dispositif, et donc π_T, est quasi indépendante de l'implantation choisie pour la machine d'états finis. En conséquence, seul le premier terme de la formule (7) varie selon le flot de synthèse choisi, et plus précisément A. Le taux de défaillance dû uniquement au bloc de la machine d'états peut donc être calculé :

$$\lambda = \lambda_{BD}*\pi_{MFG}*\pi_T*0,64*(A/0,21)(2/Xs)^2 \quad (8)$$

où A est la surface d'implantation du bloc. Avec cette procédure d'évaluation, le taux de défaillance d'un bloc sans dispositif de masquage d'erreurs peut donc s'exprimer comme une somme des taux de défaillance de ses différents éléments. Cette caractéristique sera utilisée dans la suite.

IV.2.3. Procédure d'évaluation de la fiabilité

Le calcul des différentes caractéristiques de fiabilité des 5 types d'implantation (SID Global, SID Hamming, TMR Seq, TMR Tot et simplex) a été effectué en utilisant des modèles de Markov à temps continu. Deux modèles ont été définis. La figure IV.5 montre le plus simple des deux. Il peut être utilisé pour chacune des implantations étudiées. Avec ce modèle, la transition $1 \rightarrow 2$ est empruntée quand une faute se produit mais est tolérée du fait du mécanisme de tolérance implanté. La transition $1 \rightarrow 3$ est empruntée quand une faute se produit dans un bloc critique de la logique et ne peut donc pas être tolérée. Enfin, la transition $2 \rightarrow 3$ est empruntée quand les capacités de tolérance de l'implantation sont dépassées.

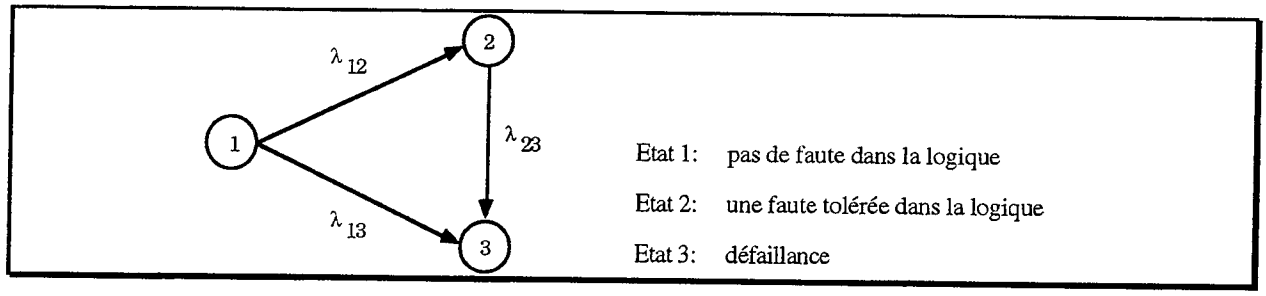

Figure IV.5 - Structure du modèle de Markov générique.

Les taux de défaillance associés à chaque transition du modèle ont été calculés avec la formule (8) présentée dans la section précédente et en fonction de la part de logique dans laquelle l'occurrence d'une faute conduit à valider la transition considérée. Le tableau IV.3 donne, pour chaque type d'implantation et chaque transition, la part de logique concernée. Il est à noter qu'ici λ_{23} est surestimé dans le cas de l'architecture SID : la logique générant le bit erroné du code d'état n'a, en fait, pas à être prise en compte lors du calcul. En effet, toute faute multiple dans un même cône logique du bloc de séquencement est tolérée par l'architecture SID. Cependant, la surface du cône considéré dépend à la fois de la machine d'états finis implantée, du flot de synthèse choisi et du bit erroné. En conséquence, toute la logique de séquencement est prise en considération (pire cas). Enfin, plus important, λ_{13} est calculé en considérant que toute faute dans la logique de décodage conduit à défaillance.

Tableau IV.3 - Logique considérée pour chacune des transitions du modèle Générique.

Architecture	λ_{12}	λ_{13}	λ_{23}
Simplex	néant	toute la logique	néant
TMR Tot	logique triplée de séquencement et de sortie	portes majorités	2/3 de la logique triplée de séquencement et de sortie, portes majorités
TMR Seq	logique triplée de séquencement	portes majorités, logique de sortie	2/3 de la logique triplée de séquencement, portes majorités, logique de sortie
SID Global et Hamming	logique de séquencement	logique de décodage, logique de sortie	toute la logique

Du fait de la structure du bloc de décodage, et en particulier du bloc de décodage généré par le flot de synthèse Global, certaines fautes dans ce bloc produisent des erreurs équivalentes à des erreurs de bit dans le registre d'états, et sont donc tolérées du fait des propriétés du code correcteur d'erreurs utilisé. Afin de prendre en compte ce fait, nous avons défini un autre modèle de Markov, spécifique à l'implantation SID, décrit par la figure IV.6.

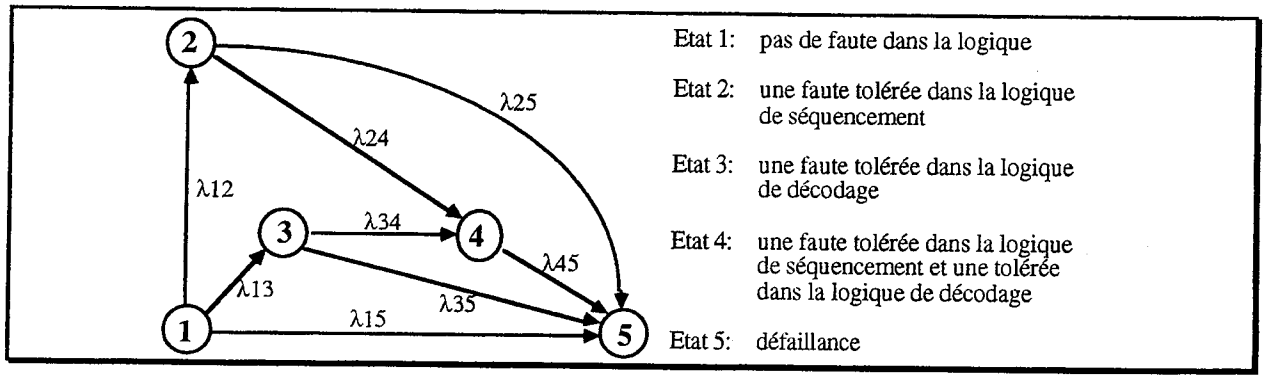

Figure IV.6 - Structure du modèle de Markov spécifique.

Pour ce modèle, nous avons les définitions suivantes.

- **Définition IV.1 : Coefficient C1**

On appellera C1 le taux de couverture obtenu par simulation de fautes, avec le jeu de vecteurs de test défini par la liste des codes d'états non erronés de la machine d'états finis, et appliqué au bloc de décodage.

- **Définition IV.2 : Coefficient C2**

On appellera C2 le taux de couverture obtenu par simulation de fautes, avec le jeu de vecteurs de test défini par la liste des codes d'états non erronés et la liste des codes d'états erronés dus à une faute simple dans la logique de calcul de l'état suivant ou le registre d'état, et appliqué au bloc de décodage.

Cela signifie que si C1 a pour valeur 0,3, 70% des fautes (prises en compte lors de la simulation) dans le bloc de décodage sont tolérées lorsque le bloc de séquencement ne contient pas lui-même de fautes actives. De même, si C2 a pour valeur 0,9, cela signifie que 10% des fautes dans le bloc de décodage sont tolérées si le bloc de séquencement ne contient pas de fautes actives, ou si celles-ci ne produisent qu'un seul bit erroné dans le code d'état.

Soit λ_{Seq} (respectivement λ_{Dec} et λ_{Out}) le taux de défaillance dû à la logique de séquencement (respectivement la logique de décodage et la logique de calcul des sorties). Les taux de défaillance relatifs à chaque transition du modèle spécifique s'expriment alors de la manière suivante :

$\lambda_{12} = \lambda_{Seq}$ $\quad\quad$ $\lambda_{25} = \lambda_{Seq} + \lambda_{Out} + \lambda_{Dec}*C2$

$\lambda_{13} = \lambda_{Dec}*(1-C1)$ $\quad\quad$ $\lambda_{34} = \lambda_{Seq}*(1-C2)$

$\lambda_{15} = \lambda_{Out} + \lambda_{Dec}*C1$ $\quad\quad$ $\lambda_{35} = \lambda_{Seq}*C2 + \lambda_{Dec} + \lambda_{Out}$,

$\lambda_{24} = \lambda_{Dec}*(1-C2)$ $\quad\quad$ $\lambda_{45} = \lambda_{Seq} + \lambda_{Dec} + \lambda_{Out}$

Ceci signifie que la transition $1 \rightarrow 2$ correspond à l'occurrence d'une faute tolérée dans la logique de séquencement. La transition $1 \rightarrow 3$ correspond à l'occurrence d'une faute tolérée dans la logique de décodage. La transition $1 \rightarrow 5$ correspond à l'occurrence d'une faute dans la logique de sortie ou à l'occurrence d'une faute non tolérée dans la logique de décodage bien qu'aucune faute ne se soit encore produite dans la logique de séquencement (coefficient C1). La transition $2 \rightarrow 4$ correspond à l'occurrence d'une faute tolérée dans la logique de décodage alors qu'une faute s'est déjà produite (et est tolérée) dans la logique de séquencement (coefficient 1-C2). La transition $2 \rightarrow 5$ correspond à l'occurrence d'une faute supplémentaire dans la logique de séquencement, ou à l'occurrence d'une faute dans la logique de sortie, ou encore à l'occurrence d'une faute non tolérée dans le décodeur alors qu'une faute est déjà tolérée dans le bloc de séquencement (coefficient C2), etc ...

Ce deuxième modèle de Markov sera utilisé dans la suite afin d'étudier l'impact que peuvent avoir les coefficients C1 et C2 sur l'évaluation de la fiabilité des implantations SID comparées aux implantations simplex et TMR.

IV.3. RÉSULTATS EXPÉRIMENTAUX

Environ une centaine d'exemples de machines d'états finis ont été synthétisés en utilisant l'outil ASYL-SdF, incluant des jeux de test MCNC et des exemples industriels. Ceci correspond à

un ensemble très représentatif de machines ayant de 4 à 257 états et de 10 à 3598 transitions dans la spécification initiale (annexe IV.2). Ces exemples ont été implantés en utilisant une bibliothèque de cellules standard CMOS 1,2µm de VLSI Technology (vsc370). La synthèse a été effectuée en utilisant les options d'optimisation en surface d'ASYL. Les surfaces et les chemins critiques après placement et routage ont été obtenus avec l'outil COMPASS. La procédure de codage utilisée lors des synthèses simplex et TMR est la procédure de codage classique [Sauc 90] décrite au paragraphe I.2.3.2.

Les taux de défaillance, pour les différentes architectures, ont été calculés sous l'hypothèse de conditions de fabrication et d'utilisation similaires, avec la taille caractéristique de la technologie (Xs) égale à 1,2µm, et un facteur de température égal à 0,84 (température de jonction de 80°C). Les caractéristiques de fiabilité ont été obtenues en utilisant les modèles de Markov présentés précédemment et l'outil SURF-2 du LAAS-CNRS [Béou 93]. Notons que, du fait des simplifications apportées à la procédure de calcul des taux de défaillance (paragraphe IV.2.2), les différences entre les résultats concernant la fiabilité des cinq types d'implantation sont plus significatives que les valeurs absolues.

IV.3.1. Impact du codage sur le coût et la fiabilité de l'architecture SID

Cette section présente les résultats comparés obtenus pour l'implantation SID avec les flots de synthèse Global et de Hamming. L'étude comparée des flots de synthèse simplex, SID et TMR est présentée dans la section suivante.

IV.3.1.1. Résultats en surface

Lorsque le flot de synthèse que nous avons appelé « de Hamming » est utilisé, du fait des propriétés du code, nous avons vu que la structure du bloc de décodage était très régulière. Il en résulte une propriété très intéressante, à savoir le fait que la surface du bloc de décodage croît très lentement lorsque le nombre d'états augmente. Ceci est illustré par la figure IV.7 qui montre la surface du bloc de décodage, en fonction du nombre d'états de la machine et en fonction du flot de synthèse choisi pour l'architecture SID.

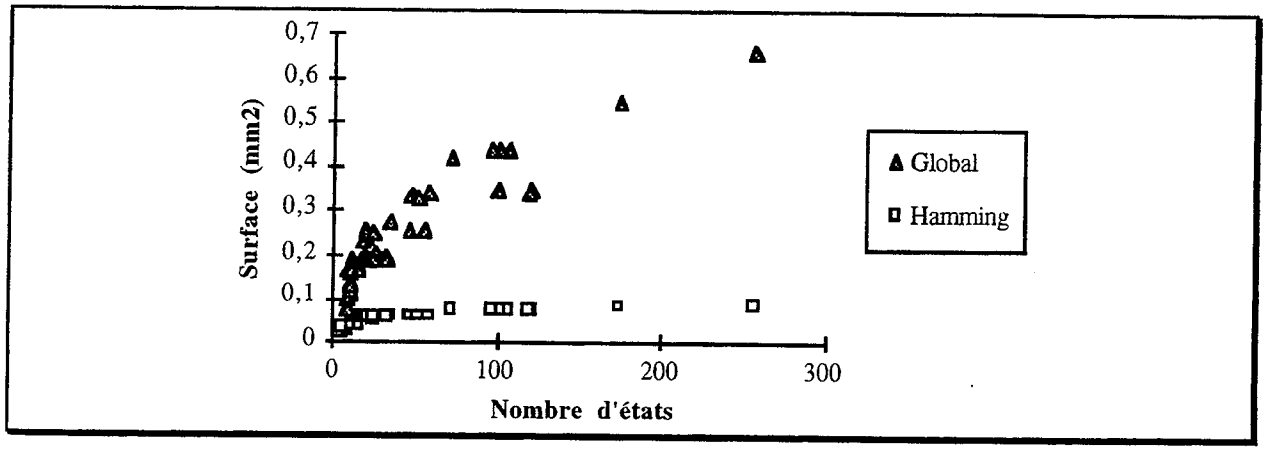

Figure IV.7 - Surface de la logique de décodage pour les deux flots de synthèse de l'architecture SID.

Contrairement à la logique de décodage du code de Hamming, le bloc de décodage obtenu avec le flot de synthèse Global n'a pas de structure particulière, et sa surface finale est très sensible au nombre de codes d'états à décoder. Ainsi le codage de Hamming donne très rapidement un meilleur résultat que le codage Global lorsque le nombre d'états croît. La différence est

particulièrement importante lorsque le nombre d'états est grand. De manière inverse, et du fait même de la rigidité de sa structure, le bloc de décodage de Hamming conduit à de moins bons résultats que ceux obtenus avec le codage Global lorsque le nombre d'états est petit. Mais là, la différence est beaucoup moins marquée.

Du point de vue de la logique de calcul de l'état suivant, les résultats obtenus avec les deux flots de synthèse sont en moyenne très proches. Avec le flot de synthèse de Hamming, les équations de calcul des bits de contrôle ne sont pas optimisées, mais l'efficacité de la procédure de codage classique sur les k bits d'information compense. Comparées à un codage aléatoire distance 3, les procédures de codage de Hamming et de codage Global donnent une logique de séquencement plus petite en surface dans plus de 80% des cas. Le gain est supérieur à 10% pour environ 60% des exemples (jusqu'à 62% pour le codage Global et l'exemple Cpt100). Quelques résultats représentatifs sont donnés dans le tableau IV.4.

Tableau IV.4 - Surface de la logique de séquencement en fonction de la procédure de codage.

Exemples	Nb. Etats	Aléatoire (µm2)	Global (gain)	Hamming (gain)
cpt100	100	974 804	62,9%	29,3%
dk17	8	89 175	15,2%	18,1%
ex7	10	143 349	23,7%	16,1%
imec1	101	1 981 778	43,3%	36,3%
keyb	19	712 141	22,6%	11,3%
kirkman	17	112 065	16,4%	21,62%
s1a	20	774 752	25,3%	27,1%
sr8a_slave	16	211 433	-4,6%	-0,6%
tav	4	4 309	-41,6%	-49,1%
zeegers	120	2 052 919	9,36%	13,6%

Tableau IV.5 - Gains dus au codage de Hamming par rapport au codage Global.

Exemples	Toute la logique	Logique de séquencement	Logique de sortie	Logique de décodage
cpt100	-5,7%	-90,6%	30,54%	78,0%
dk17	10,0%	3,3%	13,9%	6,4%
ex7	25,5%	-10,1%	-25,1%	68,5%
imec1	31,0%	-12,3%	44,15%	82,7%
keyb	8,45%	-14,6%	22,1%	72,0%
kirkman	40,9%	6,3%	31,5%	71,4%
s1a	22,7%	2,4%	--	71,4%
sr8a_slave	27,0%	3,8%	2,1%	78,3%
tav	3,43%	-5,3%	11,0%	21,6%
zeegers	17,7%	4,7%	13,8%	77,9%

En ce qui concerne la logique de sortie, la procédure de codage classique utilisée lors d'une synthèse de Hamming permet d'optimiser efficacement les équations de calcul des sorties primaires (qui n'utilisent que les k bits d'information des codes d'états). Il en résulte un gain en surface supérieur en moyenne à 20% comparé au codage aléatoire distance 3. Par contre, le codage Global ne peut pas satisfaire aussi bien les contraintes d'adjacence du fait de la distance minimale égale à 3 entre deux codes d'états. Le gain moyen de ses résultats comparé à ceux du codage aléatoire n'est dès lors plus que de 5% pour le bloc de calcul des sorties.

En fait, même lorsque le codage Global donne de bons résultats pour la logique de séquencement, la machine d'états finis (dans sa totalité) est généralement plus petite en surface après une synthèse de Hamming, du fait des optimisations apportées sur la logique de décodage et la logique de calcul des sorties (annexe IV.3). Ceci est bien illustré par des exemples comme ex7,

imec1 et keyb dans le tableau IV.5. Ainsi, la synthèse de Hamming a donné un meilleur résultat que le codage global dans plus de 90% des exemples traités.

IV.3.1.2. Résultats en vitesse

Toujours du fait de sa structure régulière, le décodeur de Hamming comporte un nombre de couches de logique qui varie peu avec le nombre d'états de la machine d'états finis. Aussi, le chemin critique d'un tel bloc n'augmente presque pas lorsque k augmente. D'un autre coté, comme pour la surface, n'ayant pas de structure prédéfinie, un décodeur pour le codage Global peut avoir un chemin critique très petit lorsque k est petit et inversement assez important lorsque k est grand. Ainsi, comme le montre le tableau IV.6a, le codage Global donne le meilleur chemin critique lorsque k est petit (dk17, tav), et le codage de Hamming donne le meilleur résultat pour les machines d'états finis plus importantes.

Lorsque toute la logique est considérée, le codage Global donne de bons résultats. Du fait du nombre non minimal de sorties du décodeur, la sortance de chacune de ses sorties est plus faible, et le chemin critique est souvent plus petit que celui obtenu avec une synthèse de Hamming (Tab. IV.6b).

Tableau IV.6 - Différences de chemin critique (Global moins Hamming, nanosecondes).

Bloc de Décodage uniquement (a)			
Exemples	Glo - Ham	Exemples	Glo - Ham
cpt100	3,8	kirkman	-0,2
dk17	-3,8	s1a	0,7
ex7	0,9	sr8a_slave	0,9
imec1	4,8	tav	-3,1
keyb	0,7	zeegers	4,3

Toute la logique (b)			
Exemples	Glo - Ham	Exemples	Glo - Ham
cpt100	-9,4	kirkman	1,5
dk17	-3,5	s1a	1,6
ex7	0,3	sr8a_slave	2,0
imec1	7,9	tav	-3,4
keyb	-2,9	zeegers	-10,1

IV.3.1.3. Résultats en fiabilité

Les temps de mission calculés pour une fiabilité de 0,8 et les MTFF obtenus pour un peu moins d'une dizaine d'exemples avec le modèle de Markov générique sont donnés dans la figure IV.8.

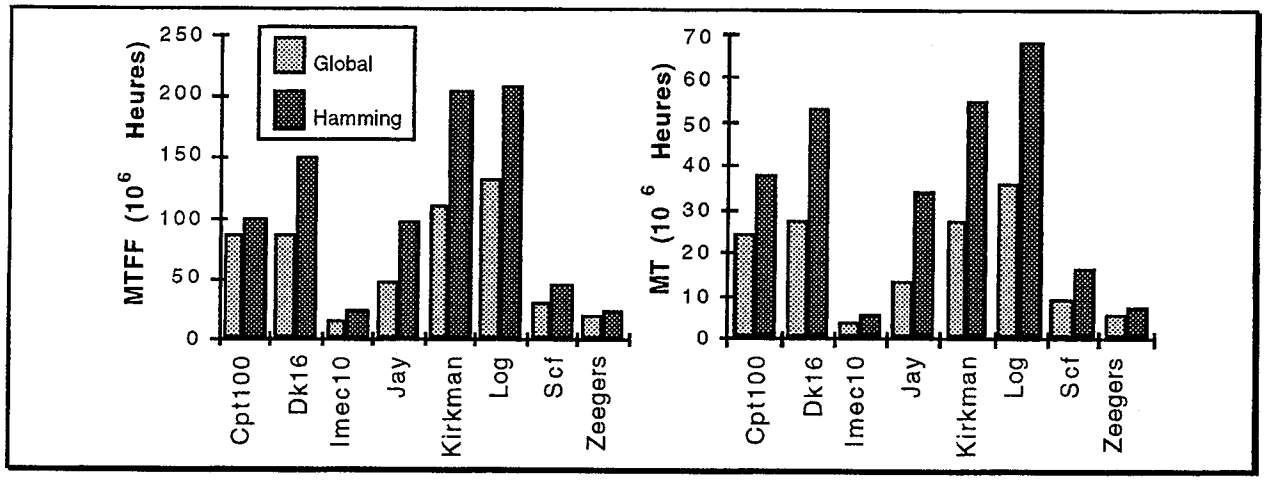

Figure IV.8 - MTFF et Temps de mission (MT) pour l'architecture SID.

Ces résultats montrent tout l'intérêt du flot de synthèse utilisant le code de Hamming. En effet, en donnant la priorité à l'optimisation du bloc de décodage et du bloc de calcul des sorties

plutôt qu'à l'optimisation de la logique de séquencement, ce flot de synthèse réduit la surface des blocs critiques, où les fautes ne sont pas nécessairement tolérées, et améliore donc la fiabilité de l'implantation. Ainsi, même un exemple comme Cpt100, dont la surface est plus faible après une synthèse dite Global, a un meilleur MTFF avec un codage de Hamming, alors que nous l'avons dit, cette mesure de la fiabilité est particulièrement sensible à la surface globale du circuit implanté.

Le choix du flot de synthèse pour l'architecture SID n'est donc pas innocent et a un impact direct, non seulement sur la surface et les performances de la machine implantée mais aussi sur ses caractéristiques de sûreté de fonctionnement. L'étude de l'impact de la prise en compte des coefficients C1 et C2 (définis au paragraphe IV.2.3) sur les résultats de fiabilité est présentée au paragraphe IV.3.3.

IV.3.2. Impact du choix de l'architecture

Dans cette section, l'impact du choix de l'architecture (simplex, SID ou TMR) est étudié en terme de surface, performances et fiabilité. Etant donné que le flot de synthèse dit de Hamming donne généralement de meilleurs résultats que le flot de synthèse Global, seuls seront évoqués ici les résultats obtenus avec le codage de Hamming pour l'architecture SID. Le flot de synthèse Global sera toutefois à nouveau considéré au paragraphe IV.3.3.

IV.3.2.1. Résultats en surface

Les résultats obtenus montrent que dans la majeure partie des cas, la synthèse de l'implantation SID donne un meilleur résultat en surface que les implantations TMR. Le gain moyen de l'architecture SID par rapport à l'implantation TMR Seq est de 17% avec des gains pouvant atteindre 42% (annexe IV.3). Plus précisément, une synthèse SID donne un meilleur résultat qu'une implantation TMR Seq pour 92% des exemples. Le gain en surface est supérieur à 10% dans 75% des cas, et supérieur à 20% pour 49% d'entre eux. Comparé à l'implantation TMR Tot, le gain moyen est de 32% avec des gains pouvant atteindre 62%.

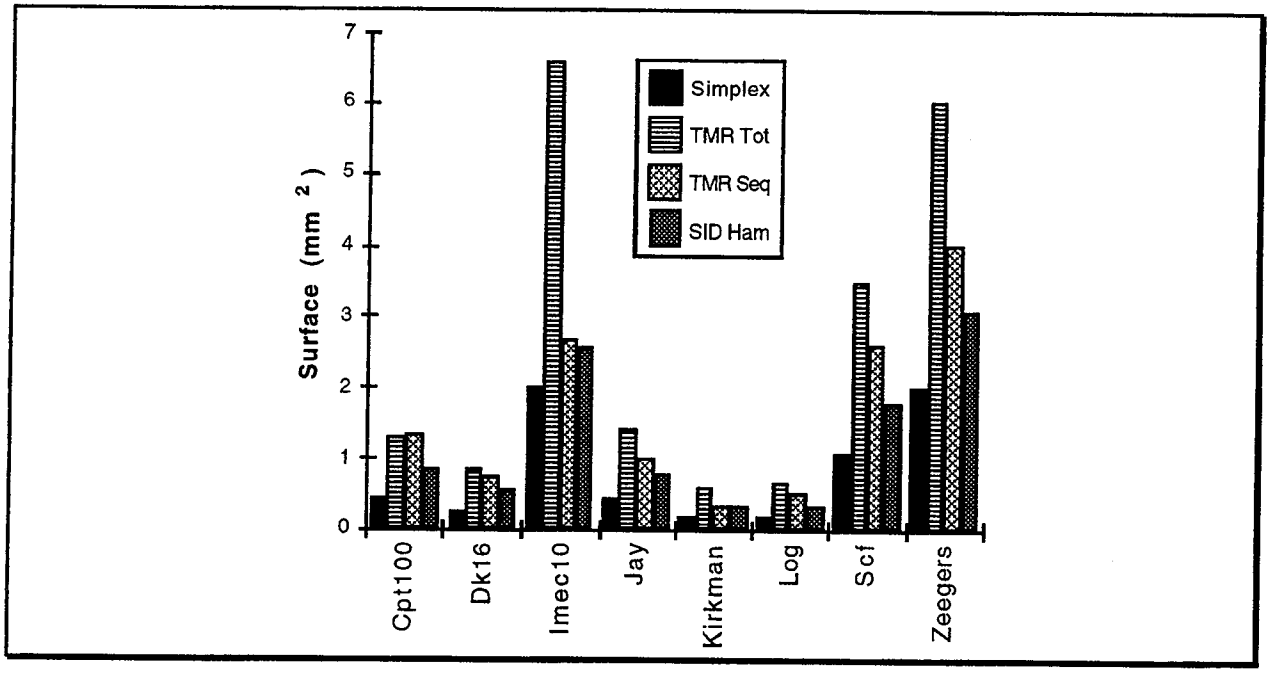

Figure IV.9 - Surface des différentes implantations pour quelques exemples.

Comparé à une implantation simplex, le surcoût de l'implantation SID est en moyenne de 119%, contre 162% pour l'implantation TMR Seq et 223% pour l'implantation TMR Tot.

Le gain en surface autorisé par le choix d'un codage à base de codes correcteurs d'erreurs simples plutôt qu'un triplement de la logique de séquencement est donc loin d'être négligeable. Ceci est illustré par la figure IV.9 qui donne les surfaces de quelques exemples représentatifs. Notons que pour des exemples comme Cpt100, pour lesquels le bloc de calcul des sorties est très petit, la surface de l'implantation TMR Seq peut être légèrement supérieure à la surface de l'implantation TMR Tot. Ceci est dû principalement à l'absence de partage de logique entre les blocs de calcul de l'état suivant et le bloc de calcul des sorties dans l'implantation TMR Seq, combiné à un placement / routage moins efficace. Le nombre portes majorité implantées intervient aussi lorsque le nombre de bits du registre d'états est important par rapport au nombre de sorties primaires.

IV.3.2.2. Résultats en vitesse

Du point de vue des chemins critiques, le temps de traversée d'un bloc de décodage est plus important que celui d'une porte majorité. Les implantations TMR sont donc mieux adaptées aux applications nécessitant des performances en vitesse élevées. Toutefois, pour ce qui est de l'architecture TMR Seq, les sorties des portes majorités étant connectées aux entrées de trois blocs de séquencement, leur sortance est assez importante et tend à réduire l'écart entre les chemins critiques des deux implantations TMR Seq et SID. La figure IV.10 montre les résultats en vitesse pour quelques exemples.

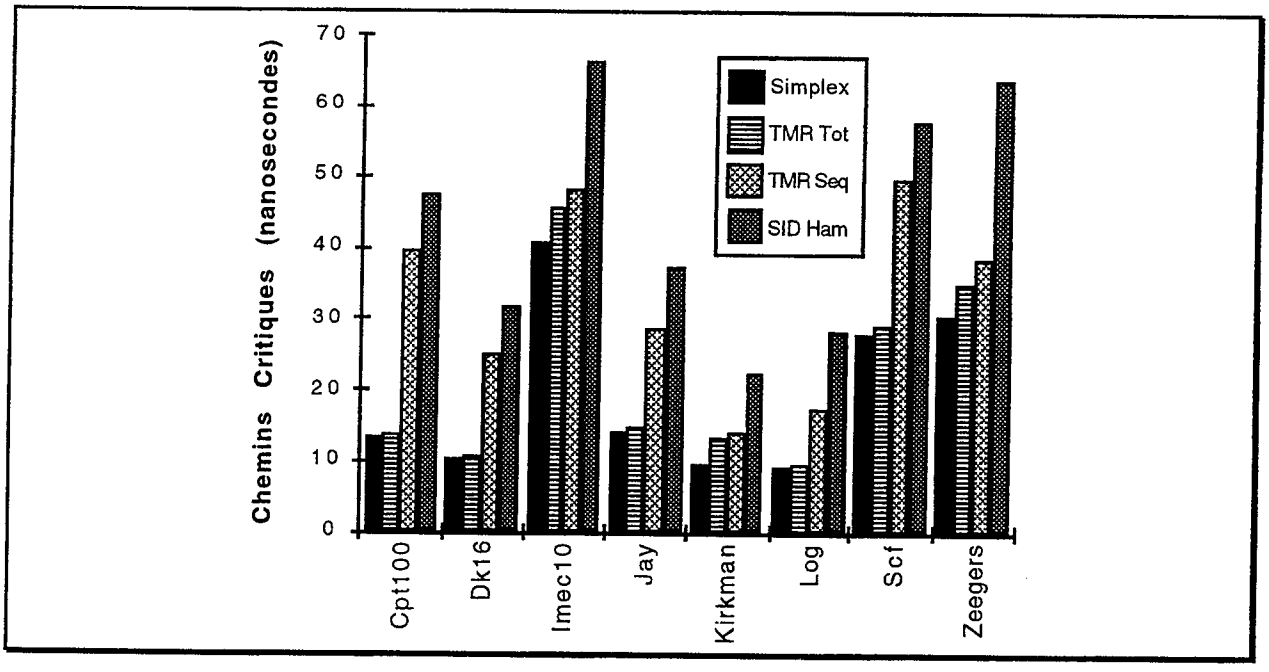

Figure IV.10 - Chemins critiques des différentes implantations pour quelques exemples.

Le temps de traversée d'un bloc de décodage (de Hamming) varie de 7,2 à 12,5 ns, contre 3 ns pour une porte majorité. Toutefois, les performances des implantations SID restent acceptables puisque tous les exemples synthétisés peuvent fonctionner avec une fréquence d'horloge de 10Mhz, 60% d'entre eux avec une fréquence de 30Mhz et 30% avec une fréquence d'horloge supérieure à 40Mhz (annexe IV.4).

Le choix de l'architecture cible doit donc se faire en fonction de la priorité des contraintes performances/surface. L'architecture SID est résolument dédiée à la réduction du coût en surface de la tolérance aux fautes. A l'inverse, l'architecture TMR Tot n'induit qu'un faible accroissement du chemin critique par rapport au simplex, mais au détriment du coût en surface.

IV.3.2.3. Résultats en fiabilité

Du point de vue de la fiabilité, l'architecture TMR Tot conduisant à des coûts importants en surface, les MTFF obtenus sont souvent les plus faibles des quatre implantations discutées dans cette section. A l'opposé, l'architecture SID conduit très souvent au meilleur résultat des implantations masquant les erreurs, et donne même parfois un meilleur résultat que les implantations simplex. Ceci est illustré par la figure IV.11 où sont données les valeurs des MTFF pour quelques exemples (modèle de Markov générique).

L'analyse combinée des figures IV.9 et IV.11 ou des figures IV.9 et IV.12 permet d'évaluer l'impact de la surface sur le MTFF ou le temps de mission, et donc d'évaluer le rôle de l'architecture dans l'amélioration des caractéristiques de fiabilité. Ainsi, pour un exemple comme Zeegers, lorsqu'on considère le MTFF, on s'aperçoit que le bon résultat de l'implantation SID est plus une conséquence de la structure d'implantation utilisée que de la faible surface du circuit. En effet, les implantations TMR Seq et SID ont un MTFF légèrement meilleur que l'implantation simplex alors que leur surface est plus grande.

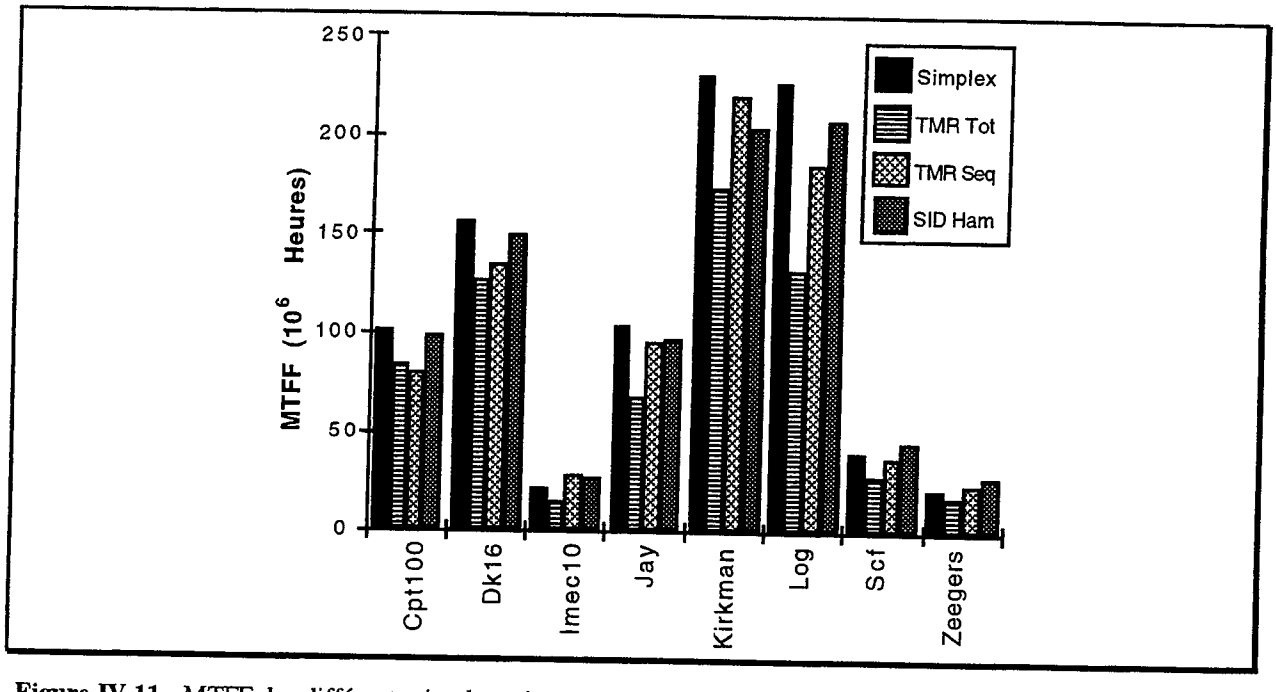

Figure IV.11 - MTFF des différentes implantations pour quelques exemples (modèle de Markov générique).

La figure IV.12 montre les temps de mission obtenus pour une fiabilité de 0,8. Ces résultats sont moins sensibles à la surface de l'implantation qu'à sa structure interne. On peut constater que l'utilisation des codes correcteurs d'erreurs simples de Hamming donne le meilleur résultat dans 50% des cas, les implantations TMR Tot et Seq se partageant les 50% restants. L'implantation sans dispositifs de masquage est, elle, très nettement en retrait.

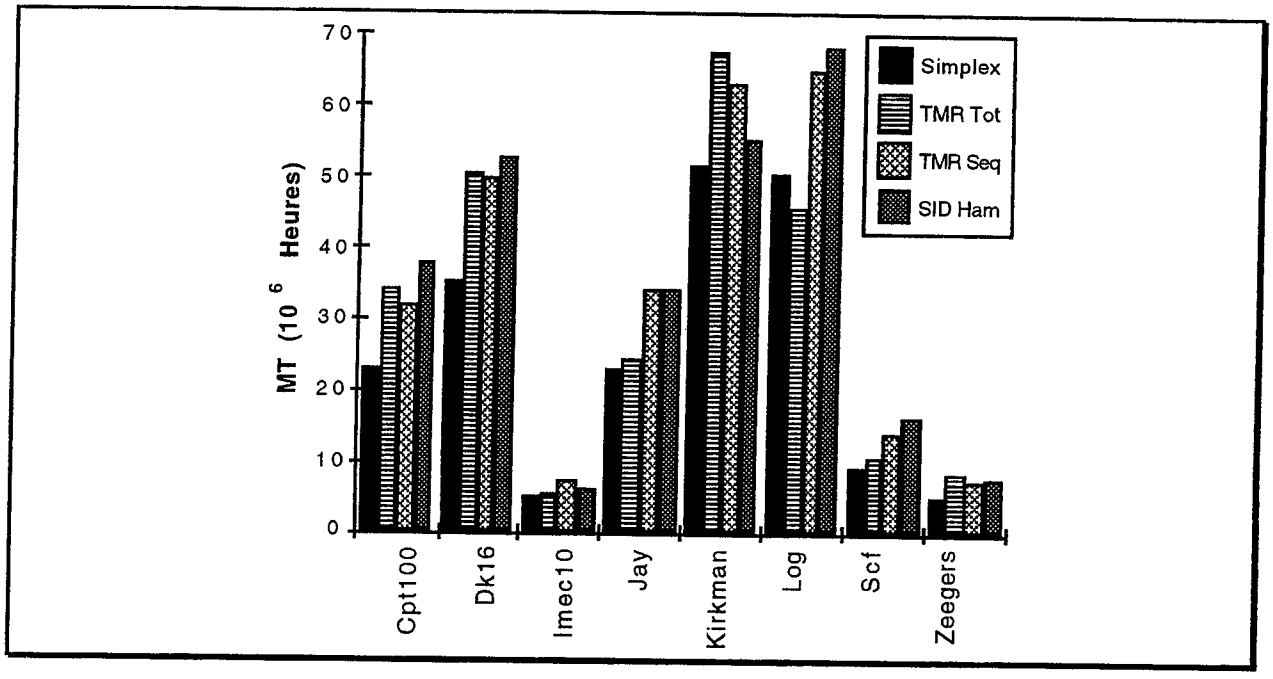

Figure IV.12 - Temps de mission (MT) pour quelques exemples (modèle de Markov générique).

Il est à noter que même lorsque la synthèse SID ne donne pas le meilleur résultat, elle conduit souvent à un intéressant compromis surface/fiabilité. Ceci est illustré par les figures IV.13 et IV.14 pour les exemples Zeegers et Jay.

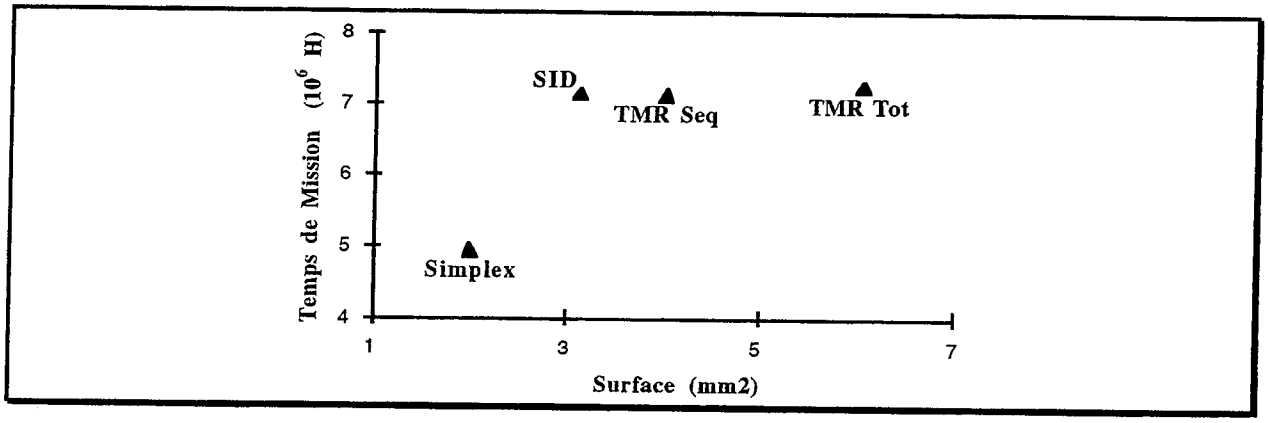

Figure IV.13 - Compromis Surface-Fiabilité pour l'exemple Zeegers.

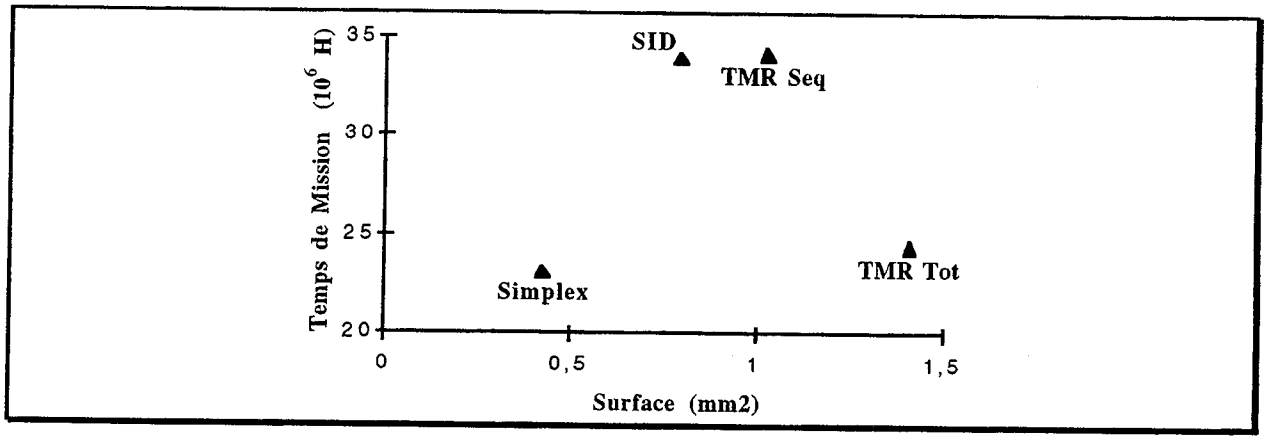

Figure IV.14 - Compromis Surface-Fiabilité pour l'exemple Jay.

Le fait que l'implantation TMR Seq donne parfois un meilleur temps de mission que l'implantation TMR Tot dépend principalement du rapport « surface de la logique de sortie sur le nombre de sorties ». Plus le nombre de sorties est grand et la surface petite, et plus l'implantation TMR Tot a tendance à donner un mauvais résultat par rapport au TMR Seq, comme c'est le cas pour l'exemple Jay de la figure IV.14.

IV.3.3. Impact des coefficients C1 et C2 sur l'étude

Nous l'avons mentionné au paragraphe IV.2.3, une caractéristique importante des décodeurs de l'architecture SID est qu'un grand nombre de fautes dans ce bloc conduisent à des erreurs équivalentes à un code d'état erroné (à distance 1 d'un code valide) en entrée du bloc, et sont donc tolérées. Ceci a été observé que le codage soit un codage Global ou de Hamming. Afin d'en étudier l'impact sur les caractéristiques de fiabilité, nous avons défini le modèle de Markov spécifique.

Tableau IV.7 - Taux de couverture C1 & C2 pour les fautes de collages.

k / n	Ham. C1	Ham. C2	k / n	Glob. C1	Glob. C2
2 / 5	20,0 %	90,0 %	2 / 5	19,8%	100 %
3 / 6	22,2 %	92,2 %	3 / 6	11,6%	100 %
4 / 7	28,1 %	100 %	4 / 7	15,7%	100 %
5 / 9	23,7 %	97,4 %	5 / 9	19,5%	100 %
6 / 10	24,1 %	96,5 %	6 / 10	11,62%	100 %

Les coefficients C1 et C2 utilisés par ce modèle ont été obtenus, pour des fautes de type collage simple, à partir de simulations de fautes avec l'outil SUNRISE. Les résultats sont résumés dans le tableau IV.7 pour des blocs de décodage de différentes tailles. Nous rappelons que pour C1, le jeu de vecteurs de test appliqué en entrée du bloc est la liste des codes d'états non erronés pour la valeur de k donnée, et que pour C2, cette liste se complète de la liste des codes d'états erronés à masquer. Ceci permet d'étudier le comportement du bloc de décodage face à une faute lorsque:

C1- le bloc de séquencement ne comporte pas de faute,

C2- le bloc de séquencement comporte une faute simple.

Dans le cas d'une synthèse de type Hamming, C1 est souvent inférieur à 30% et C2 est rarement égal à 100%. Pour C1, par exemple, ceci signifie que plus de 70% des fautes de collage dans le bloc de décodage sont tolérées lorsque le bloc de séquencement ne comporte pas lui-même de faute. Lors d'une synthèse de type Global, C1 est entre 10 et 20%, et C2 est égal à 100% pour tous les cas étudiés. Notons que ces valeurs ont été calculées dans un pire cas. En effet, le jeu de vecteurs de test a été généré pour tous les codes d'états possibles sur k bits (dans le cas du codage Global, 2^k mots du code ont été générés sur n bits). En fait, la plupart des machines d'états finis n'utilisent pas tous les codes d'états. Par exemple, Kirkman comporte 17 états (k=5). Afin de calculer C1 pour k=5, nous avons généré 32 vecteurs de test. En fait, dans le cas de l'exemple Kirkman, seuls les 17 réellement utilisés par la machine sont significatifs.

Les figures IV.15 et IV.16 montrent les temps de mission et les MTFF obtenus en prenant en compte C1 et C2 pour les deux types de synthèse de l'architecture SID. L'augmentation des valeurs absolues (représentée par des hachures diagonales dans les figures) n'est pas négligeable mais ne change pas le rapport de force entre la synthèse de Hamming et la synthèse de type Global, sauf en ce qui concerne l'exemple Cpt100 pour le MTFF. Lorsqu'on recompare les résultats de la synthèse SID Hamming et des synthèses TMR, par contre, l'évolution est importante. Ainsi, l'implantation SID donne le meilleur MTFF dans 50% des cas, et donne le

deuxième meilleur MTFF (généralement derrière l'implantation simplex) pour le reste des exemples. Pour les temps de mission, la synthèse SID conduit au meilleur résultat dans 75% des cas. Ainsi, la synthèse SID donne désormais le meilleur résultat pour Zeegers et Jay. De plus, les résultats prenant en compte les coefficients C1 et C2 montrent que ce flot de synthèse conduit à un intéressant compromis surface/fiabilité pour les exemples Kirkman et Imec10.

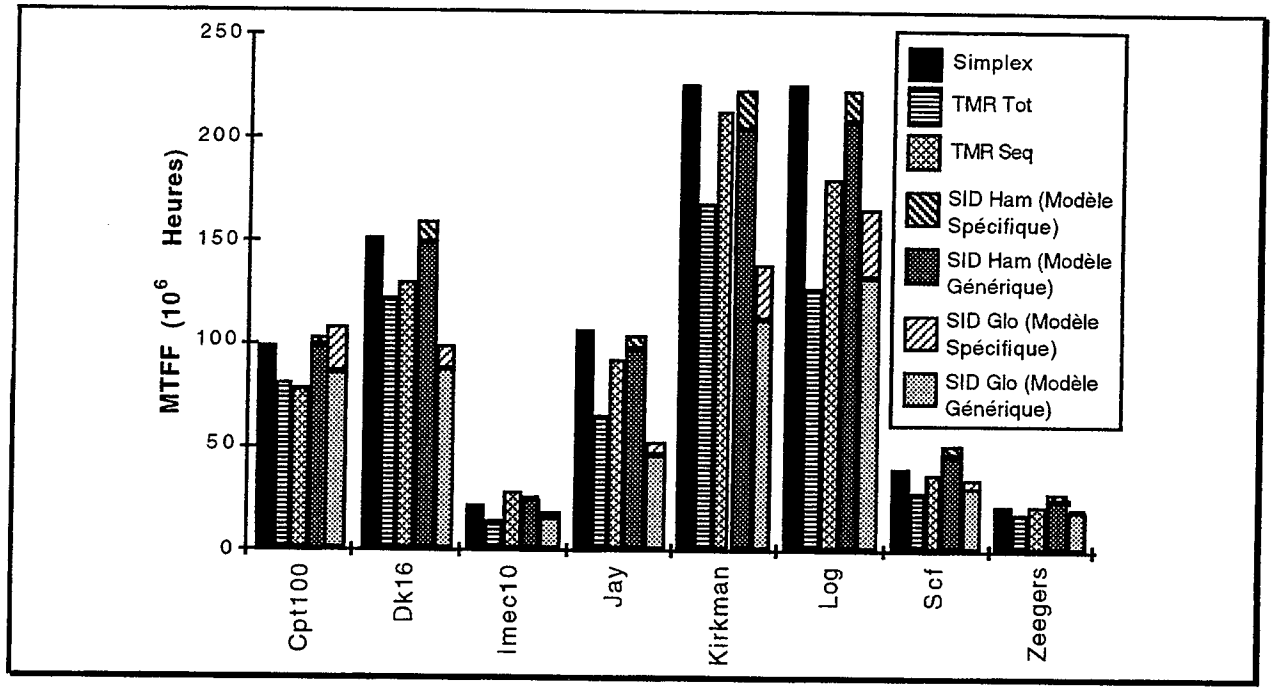

Figure IV.15 - Impact du modèle de Markov Spécifique sur le MTFF des implantations SID.

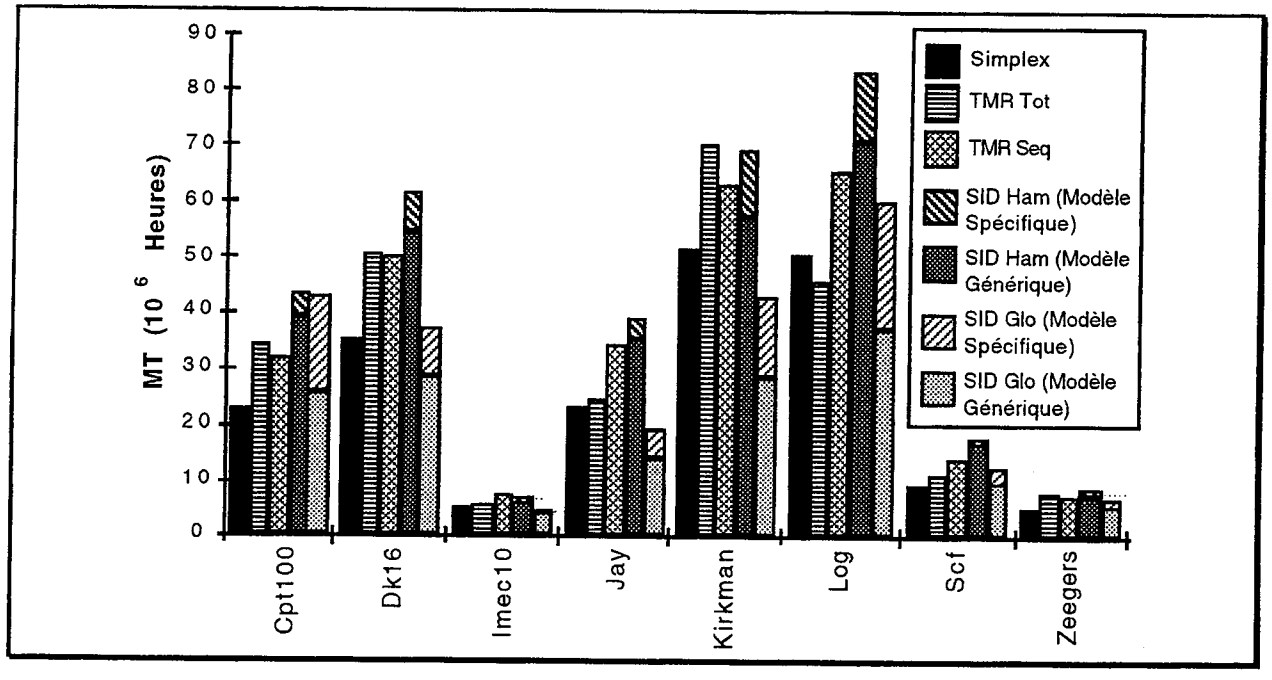

Figure IV.16 - Impact du modèle de Markov Spécifique sur le MT des implantations SID.

La prise en compte des coefficients C1 et C2 conduit donc à une amélioration non négligeable des résultats chiffrés de l'évaluation de la fiabilité de l'architecture SID, ce qui confirme l'intérêt de ce type d'implantations. Toutefois, les simulations de fautes se sont limitées

aux collages à 1 et à 0, et les résultats présentés doivent donc être considérés avant tout comme illustratifs.

IV.4. CONCLUSION

L'étude de l'impact du flot de synthèse sur la sûreté de fonctionnement et son coût matériel pour les machines d'états finis a montré l'intérêt de l'implantation SID en matière de réduction des coûts en surface sans réduction notable de fiabilité. Ainsi le gain en surface dû à l'implantation SID comparée à l'implantation TMR Seq est supérieur à 20% pour près d'un exemple sur deux. Du point de vue fiabilité, l'étude a porté essentiellement sur les fautes permanentes. Toutefois, différentes remarques laissent prévoir un bon comportement de l'architecture SID vis-à-vis des fautes transitoires (comme indiqué dans l'introduction du paragraphe IV.2). Les implantations de type TMR restent mieux adaptées aux applications exigeant des performances en vitesse élevées.

L'étude a aussi montré l'importance de la procédure de codage, du code correcteur d'erreurs utilisé et du flot de synthèse qui en découle, lors de la synthèse de l'architecture SID. Ainsi le flot de synthèse de type Global est mieux adapté aux petites machines d'états finis ayant peu d'éclatements dans leur graphe d'états, tandis que le flot de synthèse de type Hamming donne de meilleurs résultats pour les machines ayant un plus grand nombre d'états.

Notons qu'une évaluation du flot de synthèse de Hamming de l'architecture SID a, aussi, été effectuée par Thomson-CSF ([CSF 95]). Les résultats de cette évaluation ont été publiés dans [Roch 96a]. Les conclusions de cette évaluation confirment les remarques citées ci-dessus, et ce pour une bibliothèque de cellules standard, un outil de placement routage et un protocole d'évaluation de la fiabilité différents. Des extraits de l'article [Roch 96a] sont fournis en annexe IV.5. Le résultat le plus marquant est lié à la fiabilité. Les temps de mission ont, en effet, été calculés avec la procédure du CNET ([CNET 93]) pour un niveau de fiabilité égal à 0,999, et l'étude a montré que l'architecture SID reste une bonne alternative à l'implantation TMR Seq, même lorsqu'un tel niveau de fiabilité très élevé est exigé.

Le développement d'outils de synthèse prenant en compte des contraintes de sûreté de fonctionnement, comme c'est le cas pour ASYL-SdF, permet donc, outre une amélioration de la productivité des concepteurs, un meilleur ciblage des implantations en surface, performances et fiabilité, c'est-à-dire une amélioration de la conception du point de vue qualitatif. De plus, en automatisant certains mécanismes, il permet de réaliser des implantations avec un grain de redondance plus fin et mieux ciblé, comme c'est le cas avec l'architecture SID par rapport aux implantations TMR classiques, et ainsi de réduire très sensiblement le coût de la sûreté de fonctionnement pour un niveau de fiabilité comparable.

CONCLUSION

Outre l'étude bibliographique, une des deux principales contributions de cette thèse est de proposer des flots de synthèse originaux, automatisés et intégrés dans un outil de synthèse commercialisé. La seconde est d'exposer une étude détaillée des avantages et des limites des approches proposées. Les flots de synthèse implantés concernent l'amélioration de la sûreté de fonctionnement des contrôleurs câblés par des techniques de détection et de masquage d'erreurs.

Deux des motivations à l'origine de ces travaux sont, pour la première, la nécessité d'intégrer les préoccupations de sûreté de fonctionnement dans le flot classique de conception afin d'en faciliter l'accès aux concepteurs du monde industriel, et pour la seconde, de profiter de l'automatisation qui en découle pour proposer de nouvelles implantations, utilisant une redondance plus judicieuse que la redondance massive, et mieux adaptées à certains types d'applications.

En ce qui concerne l'automatisation de la synthèse pour faciliter l'accès des concepteurs aux préoccupations de sûreté de fonctionnement, les résultats des travaux effectués sont les suivants.

Outre la redondance massive, les techniques de détection utilisées sont basées sur la vérification d'un flot de contrôle par analyse de signature. L'efficacité des algorithmes utilisés lors de la synthèse est démontrée, et les résultats obtenus pour un grand nombre d'exemples analysés. En particulier, un nouvel algorithme de réparation des graphes pour la vérification d'un flot de contrôle à base d'ajustements implicites a été proposé, ainsi qu'un algorithme de placement des ajustements pour la vérification d'un flot de contrôle à base d'ajustements explicites.

Le masquage d'erreurs est basé sur l'utilisation de codes correcteurs d'erreurs simples lors du codage des états de la machine. Un algorithme de codage défini par R. Leveugle a été implanté et l'analyse des résultats obtenus pour une centaine d'exemples a permis de définir un nouvel algorithme plus efficace, permettant en particulier un meilleur contrôle de la surface des blocs critiques de l'architecture SID synthétisée. Cette architecture est dérivée des travaux de D. B. Armstrong publiés dans [Arms 61]. Le nouvel algorithme est basé sur l'utilisation du code de Hamming, dont l'une des propriétés est de permettre la synthèse d'un bloc de décodage de faible surface.

En ce qui concerne la proposition d'alternatives à la redondance massive, l'analyse des résultats obtenus montre l'intérêt des choix effectués au niveau architectural et au niveau des flots de synthèse.

Ainsi, pour ce qui est de la détection d'erreurs, l'efficacité (déjà mentionnée, en particulier du point de vue du coût en surface de silicium, dans [Leve 90b]) des techniques de vérification

d'un flot de contrôle est confirmée. Toutefois, une comparaison faite avec les résultats obtenus pour la méthode définie par S. Robinson dans [Robi 92a] et pour la duplication, montre que la vérification d'un flot de contrôle n'est pas systématiquement intéressante en tant qu'alternative à la duplication. Le coût matériel du contrôleur implanté dépend, en effet, étroitement de certaines caractéristiques structurelles, identifiables, de la spécification de départ. Il n'en est pas moins vrai que pour le nombre non négligeable de contrôleurs bien adaptés à ce type de techniques, les nouveaux flots de synthèse implantés permettent d'obtenir un intéressant compromis entre coût matériel et niveau de sûreté de fonctionnement.

Pour ce qui est du masquage d'erreurs, l'analyse des résultats obtenus avec les deux flots de synthèse implantés, ainsi qu'avec deux implantations utilisant la redondance massive (TMR), montre l'intérêt, en termes de surface et de fiabilité, de l'utilisation de codes correcteurs d'erreurs. Plus encore que le choix architectural, l'intérêt du choix, au niveau du flot de synthèse, de l'utilisation du code de Hamming est démontré.

Que ce soit pour la détection ou pour le masquage d'erreurs, les techniques mises en oeuvre visent principalement la réduction du coût, en terme de surface de silicium, de la redondance implantée. L'automatisation des traitements permet de plus de réduire le coût de conception lié à l'amélioration de la sûreté de fonctionnement des contrôleurs, en particulier lorsque des techniques plus pointues sont préférées à la redondance massive. Le coût en vitesse des alternatives proposées est, par contre, généralement supérieur à celui de la duplication et du triplement, en particulier lorsqu'on considère le masquage d'erreurs basé sur l'utilisation de codes correcteurs d'erreurs. Toutefois, l'accroissement du coût en vitesse reste raisonnable et les alternatives à la redondance massive proposées sont particulièrement bien adaptées aux applications liées au contrôle des processus industriels ou aux applications critiques civiles.

Les travaux exposés dans ce document ont en partie été réalisés dans le cadre du projet européen JESSI AC8 « New Challenges in Synthesis, Optimization & Analysis of Digital Systems ». Ils ont de plus fait l'objet d'une collaboration avec Sextant Avionique [Leve 93a] et Thomson-CSF [Roch 96a], cette dernière ayant confirmé les conclusions obtenues lors de notre étude (annexe IV.4). Enfin, l'outil programmé au cours de cette thèse (environ 26 000 lignes de C) a fait l'objet d'un transfert technologique vers la société Innovative Synthesis Technologies (IST).

Les perspectives du travail décrit dans ce document sont nombreuses. Nous n'en mentionnerons ici que quelques-unes.

Ainsi, une extension, à d'autres types d'implantation des contrôleurs, des techniques utilisées ici pour les contrôleurs câblés est possible. En particulier, une adaptation aux contrôleurs à base de ROM des flots de synthèse implantés a déjà commencé ([Wend 95], [Wend 96a]). Des recherches sont aussi en cours pour comparer la consommation électrique des différentes architectures synthétisées ([Wend 96b]).

Un complément possible des travaux sur le masquage d'erreurs décrits au chapitre IV, est l'extension aux erreurs Byzantines du modèle d'erreurs simples utilisé. Une erreur Byzantine est une erreur conduisant à interpréter, de manière non déterministe, un même signal comme un 0 en certains points du circuit, et comme un 1 en d'autres points. Cette valeur non définie est notée X. T. Nanya a publié dans [Nany 87] une étude indiquant les modifications à apporter à la structure des blocs de décodage pour que ceux-ci puissent aussi corriger les erreurs Byzantines simples. Toutefois, il est à noter que la prise en compte de ce type d'erreurs risque d'augmenter sensiblement le coût matériel de l'architecture SID, et ce pour deux raisons : la modification de la structure du décodeur paraît a priori assez coûteuse, et la prise en compte des erreurs Byzantines nécessite l'utilisation d'un code de type Reed-Müller modifié ([Nany 87]).

Enfin, bien sûr, le but ultime de ces travaux est d'étendre les dispositifs de détection et de masquage à l'ensemble du circuit, sans plus se limiter à la seule partie contrôle, en définissant un ensemble cohérent de modifications à apporter au circuit par rapport à son implantation simplex, de manière à éviter les redondances inutiles.

BIBLIOGRAPHIE

[Abad 85] M. Abadir, M.A. Breuer,
« A Knowledge-Based System for Designing Testable VLSI Chips »,
IEEE Design & Test, Août 1985, pp. 56-68.

[Abou 92] P. Abouzeid,
« Méthodes de Factorisation Algébrique Dédiées aux Circuits Intégrés Complexes »,
Thèse de Doctorat, INPG, Grenoble, France, 1993.

[Aho 88] A. Aho, R. Sethi, J. Ullman,
« Compilers: Principles, Techniques and Tools »,
Addison-Wesley Publishing Company, Mars 1988 (Edition originale 1986).

[ALLI] Manuel utilisateur et de référence de l'outil ASIMUT de l'ensemble ALLIANCE
Anonymous ftp.ibp.fr. /ibp/softs/masi/alliance.

[Anto 90] A. Antola, R. Negrini, M. Sami, N. Scarabottolo,
« Tolerance to transient faults in microprogrammed control units »,
IEEE transactions on Reliability, vol. 39, no. 5, Décembre 1990, pp. 535-546.

[Arms 61] D. B. Armstrong,
« A general method of applying error correction to synchronous digital systems »,
The Bell System Technical Journal, vol. 40, no. 2, Mars 1961, pp. 577-593.

[ASTE 91] « La fiabilité des composants électroniques »,
ASTE, Février 1991.

[ASYL] Manuel Utilisateur et de référence du logiciel ASYL
Disponible par ftp anonymous ftp.imag.fr. /pub/asyl.

[Bail 88] A. Bailas, L. L. Kinney,
« Evaluation of a concurrent error detection method for microprogrammed control units »,
21st Annual Microprogramming Workshop (Micro 21), 1988, pp. 1-10.

[Bela 92] N. Belabbes, A. Guterman, Y. Savaria, M. Dagenais,
« Ratioed voter circuit for testing and fault-tolerance in VLSI processing arrays »,
Int. Symposium on Circuits and Systems, 1992, pp. 1125-1128.

[Belh 93] H. Belhadj, A. Fortas, G. Saucier,
« State Assignment Selection for FPGAs and CPLDs »,
User Forum, ED&TC'93, 1993, pp. 249-259.

[Béou 93] C. Béounes et *al.*,
« SURF-2: a program for dependability evaluation of complex hardware and software systems »,
23rd FTCS, 1993, pp. 668-673.

[Berr 90] C. Berrou, C. Douillard,
« Générateurs de machines séquentielles auto-testables pour circuits intégrés spécifiques »,
Septième Colloque International de Fiabilité et de Maintenabilité, Brest, France, 18-22 Juin 1990,
pp. 483-488.

[Bess 92] T. Besson, H. Bouzouzou, M. Crastes, I. Floricica, G. Saucier,
« Synthesis on Multiplexer-based FPGA using Binary Decision Diagrams »,
IEEE Proc. of ICCD 1992, pp. 163-167.

[Beus 70] H. J. Beuscher, W. N. Toy,
« Check schemes for integrated microprogrammed control and data transfer circuitry »,
IEEE transactions on Computers, vol. C-19, no. 12, Décembre 1970, pp. 1153-1159.

[Birm 88] W.P. Birmingham, A. Brennan, A.P. Gupta, and D.P. Siewiorek,
« MICON: a single board computer synthesis tool »,
IEEE Circuits and Devices Magazine, Janvier 1988.

[Blou 95] D. M. Blough, F. J. Kurdahi, S. Y. Ohm,
« Optimal recovery point insertion for high-level synthesis of recoverable microarchitectures »
25th Symposium on Fault-Tolerant Computing (FTCS), 1995, pp. 50-59.

[Bogl 95] A. Bogliolo, M. Damiani,
« Synthesis of multi-level fault-tolerant combinational circuits »,
ED&TC'95, 1995, pp. 80-85.

[Bolc 95a] C. Bolchini, R. Montandon, F. Salice, D. Sciuto,
« A state encoding for self-checking finite state machines »,
VLSI'95, Chiba, Japon, 1995, pp. 711-716.

[Bolc 95b] C. Bolchini, R. Montandon, F. Salice, D. Sciuto,
« Self-Checking FSMs Based on a constant distance state encoding »,
The IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems, IEEE Computer Society Press, Lafayette, Louisiana, 1995, pp. 269-277.

[Bour 91] J. Bourrieau,
« L'environnement spatial (flux, dose, blindage, effets des ions lourds) »,
Cours2-RADECS91, 1991.

[Bray 82] R.K. Brayton and C.T. McMullen,
« The Decomposition and Factorization of Boolean Functions »,
ISCAS Proceedings, 1982.

[Bray 85] R.K. Brayton, G.D. Hachtel, C.T. McMullen, A. Sangiovanni-Vincentelli,
« Logic Minimization Algorithms for VLSI Synthesis »,
Kluwer Academic Publishers, 1985.

[Camp 90] R. Camposano, R. Bergamaschi,
« Redesigning Using State Splitting »,
Proc. EDAC 90, Glasgow, Scotland, Mars 12-15, 1990, pp. 157-163.

[Carl] S. Carlson,
« Introduction to HDL-Based Design Using VHDL »,
Edité par la société SYNOPSYS.

[Clun 87] E. Clune and E. Czeck,
« STARS Simulator Manual »,
ECE Dept., Carnegie Mellon University, 1987.

[CNET 93] CNET, France Télécom,
« Recueil de données de fiabilité des composants électroniques »,
Juin 1993.

[COMP] Manuel utilisateur et de référence de l'outil ASIC Synthesizer de la société Compass.

[Cook 73] R. W. Cook, W. H. Sisson, T. F. Storey, W. N. Toy,
« Design of a self-checking microprogram control »,
IEEE transactions on Computers, vol. C-22, no. 3, Mars 1973, pp. 255-262.

[Cour 91] B. Courtois, M.C. Gaudel, J-C. Laprie, D. Powell,
« Sûreté de Fonctionnement Informatique », « Evolutions 1987-1992, Tendances et Perspectives »,
Rapport TIMA n. RR 905-I, Rapport LRI n. 786, Rapport LAAS n. 92.382, Décembre 1992.

[CSF 95] Thomson-CSF,
« Rapport sur la synthèse d'automates tolérants aux fautes »,
Référence R.TCS-SP6-T6.2-Q4, Septembre 1995.

[De 92] K. De, C. Wu, P. Banerjee,
« Reliability driven logic synthesis of multilevel circuits »,
Int. Symposium on Circuits and Systems, 1992, pp. 1105-1108.

[De 94] K. De, C. Natarajan, D. Nair, P. Banerjee,
« RSYN: a system for automated synthesis of reliable multilevel circuits »,
IEEE transactions on Very Large Scale Integration (VLSI) Systems, vol. 2, no. 2, Juin 1994,
pp. 186-195.

[DeMi 84] G. de Micheli,
« KISS : a Program for Optimal State Assignment of Finite State Machines »,
ICCAD 84, Santa Clara, Novembre 1984.

[Delo 89] X. Delord, R. Leveugle, G. Saucier,
« Built-in concurrent checking of ASICs »,
Colloque Euro ASIC 89, Grenoble, France, 25-27 Janvier 1989.

[Dev 88] S. Devadas, H.-K. Ma, A. R. Newton, A. Sangiovanni-Vincentelli,
« MUSTANG: state assignment of finite state machines targeting multilevel logic implementations »,
IEEE trans. on CAD, vol. 7, no. 12, Décembre 1988, pp. 1290-1300.

[Diaz 74] M. Diaz,
« Design of totally self checking and fail safe sequential machines »,
4th Symposium on Fault-Tolerant Computing (FTCS), 1974, pp. 3.19-3.24.

[Diaz 75] M. Diaz, J. Moreira de Souza,
« Design of self-checking microprogrammed controls »,
5th Symposium on Fault-Tolerant Computing (FTCS), 1975, pp. 137-142.

[Duba 88] P. Duba, R. K. Iyer,
« Transient fault behavior in a microprocessor - A case study »,
International Conference on Computer Design (ICCD), 1988, pp. 272-276.

[Duff 91] C. Duff,
« Codages d'automates et théorie des cubes intersectants »,
Thèse de Doctorat, INPG, Grenoble, France, 1991.

[Dura 83] J. Duran, T.E. Mangir,
« A design approach for a microprogrammed control unit with Built In Self Test »,
16th Annual Microprogramming Workshop (Micro 16), 1983, pp. 55-60.

[EDIF 88] Edif Steering Committee,
« Introduction to EDIF »,
Edif Monograph Series, Volume 1, Electronic Industries Association, Sept. 88 (EIA / EDIF-1).

[EDIF 89] Edif Steering Committee,
« EDIF Connectivity »,
Edif Monograph Series, Volume 2, Electronic Industries Association, Juin 89 (EIA / EDIF-2).

[Edmo 90] P. Edmond, A. P. Gupta, D. P. Siewiorek, A. A. Brennan,
« ASSURE: automated design for dependability »,
27th Design Automation Conference (DAC), 1990, pp. 555-560.

[Eife 84] J. B. Eifert, J. P. Shen,
« Processor monitoring using asynchronous signatured instruction streams »,
14th Symposium on Fault-Tolerant Computing (FTCS), 1984, pp. 394-399.

[Elki 83a] S.A. Elkind,
« SEC Version 3.3 User's Manual »,
ECE Dept., Carnegie Mellon University, 1983.

[Elki 83b] S.A. Elkind,
« Lambda; a MIL-217D System Failure Rate Analysis Program User's Manual »,
ECE Dept., Carnegie Mellon University, 1983.

[Elli 90] I. D. Elliott, I. L. Sayers,
« Implementation of 32-bit RISC processor incorporating hardware concurrent error detection and correction »,
IEE proceedings, vol. 137, no. 1, Janvier 1990, pp. 88-102.

[Esch 92] B. Eschermann,
« On combining off-line BIST and on-line control flow checking »,
22nd Symposium on Fault-Tolerant Computing (FTCS), 1992, pp. 298-305.

[Fran 66] H. Frank, S. S. Yau,
« Improving reliability of a sequential machine by error-correcting state assignments »,
IEEE transactions on Electronic Computers, vol. EC-15, Février 1966, pp. 111-113.

[Fuji 84] E. Fujiwara, N. Mutoh, K. Matsuoka,
« A self-testing group-parity prediction checker and its use for built-in testing »,
IEEE transactions on Computers, vol. C-33, no. 6, Juin 84, pp. 578-583.

[Gabo 88] H. Gabow, R. Tarjan,
« A linear-time algorithm for finding a minimum spanning pseudoforest »,
Info. Proc. Letts., Vol. 27, 1988, pp. 259-263.

[Gajs 88] D. D. Gajski,
« Silicon Compilation »,
Addison-Wesley, 1988.

[Gajs 92] D.D. Gajski, N.D. Dutt, A.C-H. Wu, S.Y-L. Lin,
« High Level Synthesis, Introduction to Chip and System Design »,
Kluwer Academic Publishers, 1992.

[Gerb 92] L. Gerbaux, G. Saucier,
« Automatic synthesis of large Moore sequencers »,
Proc. EDAC 92, Brussels Belgium, Mars 16-19, 1992, pp. 237-244.

[Gerb 94] L. Gerbaux,
« Synthèse de contrôleurs complexes avec partitionnement »,
Thèse de Doctorat, INPG, Grenoble, France, 1994.

[Gies 83] R. Giest and K. Trivedi,
« Ultra high reliability prediction for fault tolerant computer systems »,
IEEE Transactions on Computers, C-32, pp. 1118-1127, Dec. 1983.

[Göss 93] M. Gössel, E-S. Sogomonyan,
« Self-parity combinational circuits for self-testing, concurrent fault detection and parity scan design »,
Elsevier Science Publishers, IFIP 94, VLSI'93 (A-42),
T. Yanagawa and P. A. Ivey Editors, B.V. North Holland, 1993, pp. 103-111.

[Grim 94] B. Grimal,
« Synthèse d'architectures auto-testables dédiées à des applications de traitement du signal »,
Thèse de Doctorat, Université de Rennes 1, France, 1994.

[Guha 84] A. Guha, L. L. Kinney,
« Monitors of varying complexity for concurrently testing microprogrammed control units »,
International Conference on Computer-Aided Design (ICCAD), 1984, pp. 137-139.

[Guha 87] A. Guha,
« Optimizing codes for concurrent fault detection in microprogrammed controllers »,
International Conference on Computer Design (ICCD), 1987, pp. 486-489.

[Hala 79] C. Halatsis, N. Gaitanis,
« Positive fail-safe realization of synchronous sequential machines »,
IEEE transactions on Computers, vol. C-28, no. 2, Février 1979, pp. 167-172.

[Hamd 94] B. Hamdi,
« Outils CAO pour la Génération Automatique de Parties Opératives Auto-Contrôlables »,
Thèse de Doctorat, INPG, Grenoble, France, 1994.

[Hamm 50] R. W. Hamming
« Error detecting and error correcting codes »,
The Bell System Technical Journal, vol. 26, no. 2, Avril 1950, pp. 147-160.

[Holm 88] L. P. Holmquist, L. L. Kinney,
« Error detection with latency in sequential circuits »,
International Test Conference (ITC), 1988, pp. 926-933.

[Holm 91] L. P. Holmquist, L. L. Kinney,
« Concurrent error detection for restricted fault sets in sequential circuits and microprogrammed control units using convolutional codes »,
International Test Conference (ITC), 1991, pp. 926-935.

[Hsu 93] Y.-M. Hsu, E. E. Swartzlander,
« VLSI concurrent error correcting adders and multipliers »,
The IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems, F. Lombardi, M. Sami, Y. Savaria, R. Stefanelli editors, IEEE Computer Society Press, Los Alamitos, California, 1993, pp. 287-294.

[Hsu 94] Y.-M. Hsu, E. E. Swartzlander,
« Reliability estimation for time redundant error correcting adders and multipliers »,
The IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems, IEEE Computer Society Press, Los Alamitos, California, 1994, pp. 159-167.

[Hwan 91] C.T. Hwang, J.H. Lee, and Y.C. Hsu,
« A Formal Approach to the Scheduling Problem in High Level Synthesis »,
IEEE Transactions on CAD, 10(4):464-475, 1991.

[Iyen 82] S. V. Iyengar, L. L. Kinney,
« Concurrent testing of flow of control in simple microprogrammed control units »,
International Test Conference (ITC), 1982, pp. 469-479.

[Iyen 85] V. S. Iyengar, L. L. Kinney,
« Concurrent fault detection in microprogrammed control units »,
IEEE Transactions on Computers, vol. C-34, no. 9, Septembre 1985, pp. 810-821.

[Iyer 86] R. K. Iyer, D. J. Rossetti,
« A Measurement-Based Model for Workload Dependence of CPU Errors »,
IEEE Transactions on Computers, vol. C-35, Juin 1986, pp. 511-519.

[Jay 86] C. Jay,
« HSURF : un microprocesseur facilement testable pour des applications à haute sûreté de fonctionnement »,
Thèse de Doctorat, INPG, Grenoble, France, Juin 1986.

[Jha 91] N. K. Jha, S.-J. Wang,
« Design and synthesis of self-checking VLSI circuits and systems »,
International Conference on Computer Design (ICCD), 1991, pp. 578-581.

[Jha 93] N. K. Jha, S.-J. Wang,
« Design and synthesis of self-checking VLSI circuits »,
IEEE transactions on Computer-Aided Design, vol. 12, no. 6, Juin 1993, pp. 878-887.

[Kara 91] M. Karam,
« Génération de test de circuits intégrés fondée sur des modèles fonctionnels »,
Thèse de Doctorat, INPG, Grenoble, France, 1991.

[Karr 92a] R. Karri, A. Orailoglu,
« Transformation-based high-level synthesis of fault-tolerant ASICs »,
29th Design Automation Conference (DAC), 1992, pp. 662-665.

[Karr 92b] R. Karri, A. Orailoglu,
« Scheduling with rollback constraints in high-level synthesis of self-recovering ASICs »,
22nd Symposium on Fault-Tolerant Computing (FTCS), 1992, pp. 519-526.

[Karr 93] R. Karri, A. Orailoglu,
« Optimal self-recovering microarchitecture synthesis »,
23rd Symposium on Fault-Tolerant Computing (FTCS), 1993, pp. 512-521.

[Khan 89a] M. Z. Khan, J. G. Tront,
« Detection of upset induced execution errors in microprocessors »,
International Phoenix Conference on Computers & Communications, Phoenix, USA, 1989, pp. 82-86.

[Khan 89b] M. Z. Khan, J. G. Tront,
« Detection of transient faults in microprocessors by concurrent monitoring »,
International Test Conference (ITC), 1989, pp. 948.

[Kunt 69] J. Kuntzmam,
« Algèbre de Boole »,
Eddition Dunod, Paris, 1969

[Lapr 88] J-C. Laprie,
« Sûreté de Fonctionnement et Tolérance aux Fautes : Concepts de Base »,
Rapport LAAS n. 88.287, Novembre 1988.

[Lars 72] R. W. Larsen, I. S. Reed,
« Redundancy by coding versus redundancy by replication for failure-tolerant sequential circuits »,
IEEE transactions on Computers, vol. C-21, no. 2, Février 1972, pp. 130-137.

[Leve 89a] R. Leveugle, G. Saucier,
« Synthesis of dedicated controllers for concurrent checking »,
International Conference on Very Large Scale Integration (VLSI 89), Munich, F.R.G., Août 16-18, 1989 (also in "VLSI 89", G. Musgrave and U. Lauther editors, Elsevier Science Publishers, North-Holland, Amsterdam, 1989, pp. 123-132.

[Leve 89b] R. Leveugle, G. Saucier,
« Optimized synthesis of dedicated controllers with concurrent checking capabilities »,
International Test Conference (ITC), 1989, pp. 355-363.

[Leve 89c] R. Leveugle, G. Saucier,
« Concurrent checking in dedicated controllers »,
International Conference on Computer Design (ICCD), 1989, pp. 124-127.

[Leve 90a] R. Leveugle, G. Saucier,
« Optimized synthesis of concurrently checked controllers »,
IEEE transactions on Computers, vol. 39, no. 4, Avril 1990, pp. 419-425.

[Leve 90b] R. Leveugle,
« Analyse de Signature et Test en Ligne Intégré sur Silicium »,
Thèse de Doctorat, INPG, Grenoble, France, 1990.

[Leve 90c] R. Leveugle, G. Saucier,
« Une méthode de synthèse pour séquenceurs avec test en ligne continu »,
Septième Colloque International de Fiabilité et de Maintenabilité, Brest, France, 18-22 Juin 1990, pp. 496-501.

[Leve 90d] R. Leveugle, T. Michel, G. Saucier,
« Design of microprocessors with built-in on-line test »,
20th Symposium on Fault-Tolerant Computing (FTCS), 1990, pp. 450-456.

[Leve 93a] R. Leveugle, R. Rochet, G. Saucier, L. Martinez, C. Pitot,
« A synthesis tool for fault-tolerant finite state machines »,
23rd FTCS, 1993, pp. 502-511.

[Leve 93b] R. Leveugle,
« Optimized state assignment of singlr fault-tolerant FSMs based on SEC codes »,
30th DAC, 1993, pp. 14-18.

[Leve 93c] R. Leveugle,
« Test of single fault tolerant controllers in VLSI circuits »,
Proc. International Conference on Very Large Scale Integration, Grenoble, France,
T. Yanagawa and P.A. Ivey editors,
Elsevier Science Publishers, North-Holland, September 6-10, 1993, pp. 115-124.

[Leve 94] R. Leveugle, R. Rochet, G. Saucier,
« Alternative Approaches to Fault Detection in FSMs »,
The IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems, Montreal, Quebec, Canada, 17-19 Octobre, 1994,
IEEE Computer Society Press, Los Alamitos, California, 1994, pp. 271-279.

[Lin 89] B. Lin, A. R. Newton,
« Synthesis of multiple level logic from symbolic high-level description languages »,
VLSI 89, G. Musgrave and U. Lauther editors, Elsevier Science Publishers, North-Holland, Amsterdam, 1989, pp. 187-196.

[Marw 86] P. Marwedel,
« A new synthesis algorithm for the MIMOLA software system. »,
IEEE Proc. 23rd DAC, 1986, pp. 271-277.

[Mahm 85] A. Mahmood, E. J. McCluskey,
« Watchdog processors : error coverage and overhead »,
15th Symposium on Fault-Tolerant Computing (FTCS), 1985, pp. 214-219.

[Mahm 92] M. Mahmood, B. Sharma, C. Kingsley,
« A datapath synthesizer for high performance ASICs »,
Proc. IEEE Custom Integrated Circuits Conference, Boston, MA, USA, Mai 3-8, 1992.

[Mand 72] D. Mandelbaum,
« On error control in sequential machines »,
IEEE transactions on Computers, vol. C-21, no. 5, Mai 1972, pp. 492-495.

[MENT] Manuel de référence de l'outil AUTOLOGIC de MENTOR.

[Meye 71] J. F. Meyer,
« Fault-tolerant sequential machines »,
IEEE transactions on Computers, vol. C-20, no. 10, Octobre 1971, pp. 1167-1177.

[Mich 91] T. Michel, R. Leveugle, G. Saucier,
« A new approach to control flow checking without program modification »,
21st Symposium on Fault-Tolerant Computing (FTCS), 1991, pp. 334-341.

[Mich 93] T. Michel,
« Test en ligne des systèmes à base de microprocesseur »,
Thèse de Doctorat, INPG, Grenoble, France, 1993.

[Micz 83] A. Miczo,
« A self-test hardwired control section »,
IEEE transactions on Computers, vol. C-32, no. 7, Juillet 1983, pp. 695-696.

[Mign 92] A. Mignotte, M.-C. Bertrand, M. Crastes, J. Fron, J. Rampon,
« Interactive register transfer level synthesis using library blocks »,
IEEE Proc. EuroASIC 92, Paris, France, Juin 1-5, 1992, pp. 53-58.

[Mili 91] United States Department of Defense,
« Military Standardization Handbook: Reliability Prediction of Electronic Equipment »,
MIL-HDBK-217F, 1991.

[Namj 82a] M. Namjoo,
« Techniques for concurrent testing of VLSI processor operation »,
International Test Conference (ITC), 1982, pp. 461-468.

[Namj 82b] M. Namjoo,
« Design of concurrently testable microprogrammed control units »,
15th Annual Microprogramming Workshop (Micro 15), 1982, pp.173-180.

[Namj 83] M. Namjoo,
« CERBERUS-16: an architecture for a general purpose watchdog processor »,
13th FTCS, 1983, pp. 216-219.

[Nany 87] T. Nanya, H. A. Goosen,
« Effect of byzantine hardware faults on concurrent error checking »,
International Conference on Computer-Aided Design (ICCAD), 1987, pp. 242-245.

[Nara 95] R. Narasimhan, D. J. Rosenkrantz, S. S. Ravi,
« Efficient Algorithms for Analysing and Synthesizing Fault-Tolerant Datapaths »,
The IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems,
IEEE Computer Society Press, Los Alamitos, California, 1995, pp.81-89.

[Nico 89] M. Nicolaidis, S. Noraz, B. Courtois,
« A generalized theory of fail-safe systems »,
19th Symposium on Fault-Tolerant Computing (FTCS), 1989, pp. 398-406.

[Nico 90] M. Nicolaidis,
« Efficient UBIST implementation for microprocessor sequencing parts »,
International Test Conference (ITC), 1990, pp. 316-326.

[Note 89] S. Note, J. Van Meerbergen, F. Catthoor, H. De Man,
« Hardwired data path synthesis for high speed DSP systems with the Cathedral-II compilation environment »,
Logic and Architecture Synthesis for Silicon Compilers, G. Saucier, P. M. McLellan (eds), North Holland, 1989, pp. 243-254.

[Ohls 95] J. Ohlsson, M. Rimen,
« Implicit signature checking »,
25th Symposium on Fault-Tolerant Computing (FTCS), 1995, pp. 218-227.

[Ohm 96] S. Y. Ohm, D. M. Blough, F. J. Kurdahi,
« High-Level Synthesis of Recoverable Microarchitectures »,
ED&TC'96, 1996, pp. 55-62.

[Orto 92] G. A. Orton, L. E. Peppard,
« New fault tolerant techiniques for residue number systems »,
IEEE transactions on Computers, vol. 41, no. 11, Novembre 92, pp. 1453-1464.

[Osma 73] M. Y. Osman, C. D. Weiss,
« Shared logic realizations of dynamically self-checked and fault-tolerant logic »,
IEEE transactions on Computers, vol. C-22, no. 3, Mars 1973, pp. 298-306.

[Özgü 77] F. Özgüner,
« Design of totally self-checking asynchronous and synchronous sequential machines »,
7th Symposium on Fault-Tolerant Computing (FTCS), 1977, pp. 124-129.

[Papa 91] C. A. Papachristou, S. Chiu, H. Harmanani,
« A data path synthesis method for self-testable designs »,
28th Design Automation Conference (DAC), 1991, pp. 378-384.

[Pare 91] R. A. Parekhji, G. Venkatesh, S. D. Sherlekar,
« A methodology for designing optimal self-checking sequential circuits »,
International Test Conference (ITC), 1991, pp. 283-291.

[Pare 93] R. B. Parekhji,
« Design of monitored self-checking sequential circuits »,
Thèse de Doctorat, Indian Institute of Technology, Bombay, Inde, 1993.

[Park 92] I. Park, K. O'Brien, A. Jerraya,
« Tutorial: AMICAL: Interactive Architectural Synthesis Based on VHDL »,
Rapport interne, TIM3 INPG, Grenoble, France, Mars 92.

[Park 93] I. Park, K. O'Brien, A. Jerraya,
« AMICAL: Architectural Synthesis Based on VHDL »,
Elsevier Science Publishers, IFIP Trans., Synthesis for Control-Dominated Circuits (A-22),
G. Saucier & J. Trilhe Editors, B.V. North Holland, 1993, pp. 219-235.

[Paul 86] P. Paulin, J. Knight,
« HAL a Multi-Paradigm Approach to Automatic Datapath Synthesis »,
IEEE Proc. 23rd DAC, 1986, pp. 263-270.

[Peer 93] M. Peercy, P. Banerjee,
« Fault tolerant VLSI systems »,
Proceedings of the IEEE, vol. 81, no. 5, Mai 1993, pp. 745-758.

[Peng 94] Zebo Peng and Krzysztof Kuchcinski,
« Automated Transformation of Algorithms into Register-Transfer Level Implmentations »
IEEE Trans. on CAD of ICs and Systems, Vol. 13, No. 2, Février 1994, pp. 150-166.

[Pete 72] W.W. Peterson, E.J. Weldon,
« Error-correcting codes »,
Second edition, MIT Press, 1972.

[Ragh 91] V. Raghavendra, C. Lursinsap,
« Automated micro-roll-back self-recovery synthesis »
28th Design Automation Conference (DAC), 1991, pp. 385-390.

[Rama 92] C. Ramachandran, F. J. Kurdahi, D. D. Gajski, A. C.-H. Wu, V. Chaiyakul,
« Accurate layout area and delay modeling for system level design »,
IEEE Proc. ICCAD, 1992, pp. 355-361.

[Ramp 91] J. Rampon, M. Baatour,
« Synthèse automatique de circuits à partir d'une description RTL »,
Rapport de DEA, INPG, CSI, 1991.

[RayC 61] D. K. Ray-Chaudhuri,
« On the construction of minimally redundant reliable system designs »,
The Bell System Technical Journal, vol. 40, no. 2, Mars 1961, pp.595-611.

[Reed 70] I. S. Reed, A. C. L. Chiang,
« Coding techniques for failure-tolerant counters »,
IEEE transactions on Computers, vol. C-19, no. 11, Novembre 1970, pp. 1035-1038.

[Robi 90] S. H. Robinson, J. P. Shen,
« Evaluation and synthesis of self-monitoring state machines »,
International Conference on Computer-Aided Design (ICCAD), 1990, pp. 276-279.

[Robi 92a] S. H. Robinson,
« Finite-state machine synthesis for continuous, concurrent error detection using signature-invariant monitoring »,
Thèse de Doctorat, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, Mai 1992.

[Robi 92b] S. H. Robinson, J. P. Shen,
« Direct methods for synthesis of self-monitoring state machines »,
22nd Symposium on Fault-Tolerant Computing (FTCS), 1992, pp. 306-315.

[Roch 91] R. Rochet,
« Conception Modulaire de Processeurs 'Watchdog' »,
Rapport de DEA, INPG, Grenoble, France, 1991.

[Roch 93a] R. Rochet, R. Leveugle, G. Saucier,
« Analysis and comparison of fault-tolerant FSM architectures based on SEC codes »,
The IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems, Venice, Italy,
F. Lombardi, M. Sami, Y.Savaria, R. Stefanelli editors,
IEEE Computer Society Press, Los Alamitos, California, 1993, pp. 9-16.

[Roch 93b] R. Rochet, R. Leveugle, G. Saucier,
« SID : une Architecture de Machines d'Etats Finis Tolerante aux Fautes »,
Journées des Jeunes Chercheurs en Architecture de Machines, Rennes, 7-8 Décembre, 1993.

[Roch 94a] R. Rochet, R. Leveugle, G. Saucier,
« Alternative Implementations of Fault-Tolerant Controllers »,
Rapport Interne, Highly Dependable Architectures, CSI-SdF-94.2, Octobre 1994.

[Roch 94b] R. Rochet, R. Leveugle, G. Saucier,
« Alternative Approaches to Fault Detection in FSMs »,
Rapport Interne, Highly Dependable Architectures, CSI-SdF-94.3, Novembre 1994.

[Roch 94c] R. Rochet, R. Leveugle, G. Saucier,
« Impact of the State Assignment on the Efficiency of Fault-Tolerant Controller Synthesis »,
IFIP Workshop on Logic and Architecture Synthesis, Institut Polytechnique de Grenoble, France,
19-20 Décembre, 1994, pp. 147-156.

[Roch 95a] R. Rochet, R. Leveugle, G. Saucier,
« Efficient Synthesis of Fault-Tolerant Controllers »,
EDTC'95, Paris la Défence, France, 6-9 Mars, 1995, p. 593 (poster).

[Roch 95b] R. Rochet, R. Leveugle, G. Saucier,
« Control Flow Checking in FSMs: Sequencing Error Detection, Efficiency Study »,
1st IEEE Internationnal On-Line Testing Workshop, Nice, France, 4-6 Juillet, 1995, pp. 125-129.

[Roch 95c] R. Rochet, R. Leveugle, G. Saucier,
« Synthesis for Dependability: the ASYL-SdF System »,
SASIMI'95, Nara, Japan, 25-26 Août, 1995, pp. 57-64.

[Roch 95d] R. Rochet, R. Leveugle, G. Saucier,
« Efficiency Comparison of Signature Monitoring Schemes for FSMs »,
VLSI'95, Chiba, Japan, 30 Août - 1 Septembre, 1995, pp. 705-710.

[Roch 95e] R. Rochet, R. Leveugle,
« SEC state assignment selection: consequences on the area and reliability of the fault-tolerant controllers »,
Novel approaches in logic and architecture synthesis, A. Mignotte & G. Saucier editors, Chapman & Hall, 1995, pp. 190-196.

[Roch 96a] R. Rochet (INPG /CSI), J.F. Agaesse, N. Marty (Thonson-CSF),
« Evaluation of a Synthesis Tool for Single Fault Tolerant Controllers »,
User Forum, ED&TC'96, 1996, pp. 59-63.

[Roch 96b] R. Rochet, R. Leveugle, G. Saucier,
« Synthèse pour la sûreté de fonctionnement : cas des machines d'états finis »,
Technique et Science Informatiques, AFCET, publication prévue en Avril 1996.

[Roch 96c] R. Rochet, R. Leveugle, G. Saucier,
« ASYL-SdF: a Synthesis Tool for Dependability of Controllers »,
IEICE Trans. on Information and Systems, publication prévue en Octobre 1996.

[Russ 65] R. L. Russo,
« Synthesis of error-tolerant counters using minimum distance three state assignments »,
IEEE transactions on Electronic Computers, vol. EC-14, Juin 1965, pp. 359-366.

[Russ 91] G. Russel, I. D. Elliott,
« Design of highly reliable VLSI processors incorporating concurrent error detection/correction »,
IEEE Proc. EuroASIC'91, Paris, France, 27-31 Mai 1991, pp. 316-321.

[Safi 95] C. Safinia,
« Optimisations et analyse de performances en synthèse RTL orientée par le contrôle »,
Thèse de Doctorat, INPG, Grenoble, France, 1995.

[Sako 93] K. Sakouti,
« Synthèse et Optimisation Temporelle de Réseaux Booléens »,
Thèse de Doctorat, INPG, Grenoble, France, 1993.

[Sauc 90] G. Saucier, C. Duff, F. Poirot,
« State assignment of controllers for optimal area implementation »,
EDAC, 1990, pp. 547-551

[Saxe 95] N. Saxena, C. Chen, R. Swami, H. Osone, S. Thusoo, D. Lyon, D. Chang, A. Dharmaraj, N. Patkar, Y. Lu, B. Chia,
« Error detection and handling in a superscalar, speculative out-of-order execution processor system »,
25th Symposium on Fault-Tolerant Computing (FTCS), 1995, pp. 464-471.

[Seng 77] A. Sen Gupta, D. K. Chattopadhyay, A. Palit, A. K. Bandyopadhyay, M. S. Basu, A. K. Choudhury,
« Realization of fault-tolerant and fail-safe sequential machines »,
IEEE transactions on Computers, vol. C-26, no. 1, Janvier 1977, pp. 91-96.

[Seng 81] A. Sengupta, D. K. Chattopadhyay, A. Palit, A. K. Bandyopadhyay, A. K. Bandyopadhyay, A. K. Choudhury,
« Realization of fault-tolerant machines - linear code application »,
IEEE transactions on Computers, vol. C-30, no. 3, Mars 1981, pp. 237-240.

[Shen 83] J. P. Shen, M. A. Schuette,
« On-line self-monitoring using signatured instruction streams »,
International Test Conference (ITC), 1983, pp. 275-282.

[Sica 88] P. Sicard,
« Nouvelles Méthodes de Synthèse Logique »,
Thèse de Doctorat, INPG, Grenoble, France, 1988.

[Sogo 93a] E. S. Sogomonyan, M. Goessel,
« Design of self-testing and on-line fault detection combinational circuits with weakly independent outputs »,
Journal of Electronic Testing: Theory and Applications (JETTA), vol. 4, no. 3, Août 1993, pp. 267-281.

[Sogo 93b] E. S. Sogomonjan, M. Goessel,
« Design of self-parity combinational circuits for self-testing and on-line detection »,
The IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems, F. Lombardi, M. Sami, Y. Savaria, R. Stefanelli editors, IEEE Computer Society Press, Los Alamitos, California, 1993, pp. 239-246.

[Srid 82] T. Sridhar, S. M. Thatte,
« Concurrent checking of program flow in VLSI processors »,
International Test Conference (ITC), 1982, pp. 191-199.

[Stae 85] C. H. Staelin, A. Albicki
« Evaluation of monitor complexity for concurrently testing microprogrammed control units »,
International Test Conference (ITC), 1985, pp. 733-736.

[Stif 79] J.J. Stiffler, L.A. Bryant, and L.J. Guccione,
« CARE III Final Report »,
Technical report, Raytheon Co., Nov. 1979.

[Stro 93] C. E. Stroud, A. E. Barbour,
« Testability and test generation for majority voting fault-tolerant circuits »,
Journal of Electronic Testing: Theory and Applications (JETTA), vol. 4, no. 3, Août 1993, pp. 201-214

[SYNO] Manuels de Référence des outils Design Compiler et Behavioral Compiler de la société SYNOPSYS.

[Tahi 95] J. M. Tahir, S. S. Dlay, R. N. G. Naguib, O. R. Hinton,
« Fault tolerant arithmetic unit using duplication and residue codes »,
Integration, The VLSI Journal, Elsevier Science Publishers, vol. 18, no. 2&3, Juin 1995, pp. 187-200.

[Tohm 71] Y. Tohma, S. Aoyagi,
« Failure-tolerant sequential machines with past information »,
IEEE transactions on Computers, vol. C-20, no. 4, Avril 1971, pp. 392-396.

[Thom 91] D.E. Thomas, P. Moorby,
« The Verilog Hardware Description Language »,
Leluwer Academic Publishers, 1991.

[Tork 91] K. Torki, M. Nicolaidis, A. O. Fernandes,
« A self-checking PLA automatic generator tool based on unordered codes encoding »
The European Design Automation Conference (EDAC), Amsterdam, The Netherlands, 25-28 Février 1991, pp. 510-515.

[Toub 93] N. A. Touba,
« Logic synthesis for concurrent error detection »,
Center for Reliable Computing Technical Report No. 93-x, Stanford University, California, USA, Février 1993.

[Tung 86] C.-H. Tung, J. P. Robinson
« On concurrently testable microprogrammed control units »,
International Test Conference (ITC), 1986, pp. 895-900.

[VHDL 87] IEEE Standard VHDL Language Reference Manual
publié par l'IEEE, Inc 345 East 47th Street, NY 10017, USA.

[Vill 89] T. Villa, A. Sangiovanni-Vincentelli,
« NOVA : State Assignment of Finite State Machines for Optimal Two-Level Logic Implementations »,
26th Desgin Automation Conference, Las Vegas, pp. 327-332, Juin 1989.

[Vosk 96] E. Voskamp, W. Rosenstiel,
« Error Detection in Fault Secure Controllers Using State Encoding »,
ED&TC'96, 1996, pp. 200-204.

[Wang 84] G. X. Wang, G. R. Redinbo,
« Probability of state transition errors in a finite state machine containing soft failures »,
IEEE transactions on Computers, vol. C-33, no. 3, Mars 1984, pp. 269-277.

[Wart 90] N. J. Warter, W. W. Hwu,
« A software based approach to achieving optimal performance for signature control flow checking »,
20th Symposium on Fault-Tolerant Computing (FTCS), 1990, pp. 442-449.

[Wend 95] X. Wendling, V. Sornette, R. Leveugle,
« ROM-Based Synthesis of Large Controllers with Control-Flow Checking Capabilities »,
1st IEEE Internationnal On-Line Testing Workshop, Nice, France, 4-6 Juillet, 1995, pp. 129-133.

[Wend 96a] X. Wendling, R. Rochet, R. Leveugle,
« Standard and ROM-Based Synthesis of FSMs with Control-Flow Checking Capabilities »,
14th IEEE VLSI Test Symposium, Princeton, 1996, pp. 81-86.

[Wend 96b] X. Wendling, R. Rochet, R. Leveugle,
« Power Dissipation Considerations when Implementing Fault Detection in FSMs »,
2nd IEEE Internationnal On-Line Testing Workshop, Biarritz, France, 8-10 Juillet, 1996, pp. 72-75.

[Wend 96c] X. Wendling, R. Rochet, R. Leveugle, "ROM-Based Synthesis of Fault-Tolerant Controllers", "The IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems, Boston, USA, November 6-8, 1996", IEEE Computer Society Press, Los Alamitos, California, 1996.

[West 93] N.H.E. Weste, K. Eshraghian,
« Principles of CMOS VLSI Design : A Systems Perspective »,
Addison-Wesley Publishing Company, 2ème édition, 1993.

[Wilk 87] K. D. Wilken, J. P. Shen,
« Embedded Signature Monitoring : analysis and technique »,
International Test Conference (ITC), 1987, pp. 324-333.

[Wilk 88] K. Wilken, J. P. Shen,
« Continuous signature monitoring: efficient concurrent-detection of processor control errors »,
International Test Conference (ITC), 1988, pp. 914-925.

[Wilk 90] K. Wilken, J. P. Shen,
« Continuous signature monitoring: low-cost concurrent detection of processor control errors »,
IEEE transactions on Computer-Aided Design, vol. 9, no. 6, Juin 1990, pp. 629-641.

[Wilk 91] K. D. Wilken,
« Optimal signature placement for processor-error detection using signature monitoring »,
21st Symposium on Fault-Tolerant Computing (FTCS), 1991, pp. 326-333.

[Wilk 93] K. D. Wilken,
« An optimal graph-construction approach to placing program signatures for signature monitoring »,
IEEE transactions on Computers, vol. 42, no. 11, November 1993, pp. 1372-1381.

[Wong 83] C. Y. Wong, W. K. Fuchs, J. A. Abraham, E. S. Davidson,
« The design of a microprogram control unit with concurrent error detection »,
13th Symposium on Fault-Tolerant Computing (FTCS), 1983, pp. 476-483.

[Youn 91] C. R. Yount, D. P. Siewiorek,
« SIDECAR: design support for reliability »,
28th Design Automation Conference (DAC), 1991, pp. 199-204.

[Zege 90] I. Zegers, P Six, J. Rabaey, H. DeMan,
«.Automatic Generation of Controllers in the Cathedral II Silicon Compiler »,
EDAC 90, Glasgow, Scotland, Mars 12-15, 1990, pp. 617-621.

ANNEXES

ANNEXE II.1 - RÉPARATION D'UN NSC-GRAPHE

L'annexe II.1 présente un exemple de réparation d'un NSC-Graphe avec l'algorithme de [Robi 92a] succinctement décrit au paragraphe II.1.4.3. La figure Ax.II.1A montre la phase d'expansion maximale qui permet de rendre le graphe S-codable (algorithme de la figure II.16).

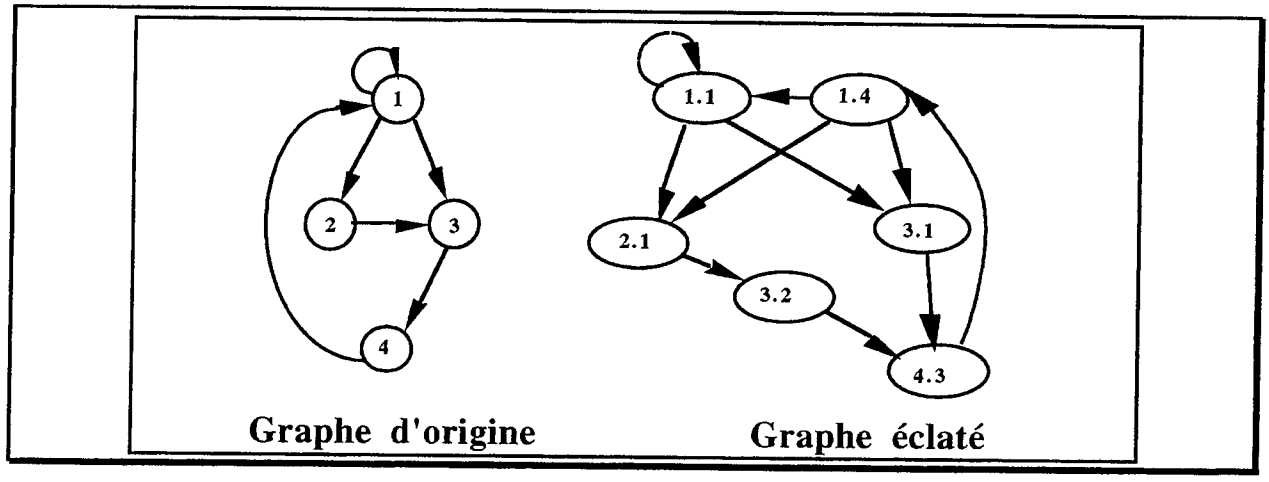

Figure Ax.II.1A - Phase d'expansion maximale du NSC-graphe.

La figure Ax.II.1B montre la phase de fusion des états. Dans le présent exemple, la liste des fusions possibles est Lf = {(1.1; 1.4), (3.1; 3.2)}. Seule la première est réalisable, le cumule des deux fusions conduisant à un NSC-graphe.

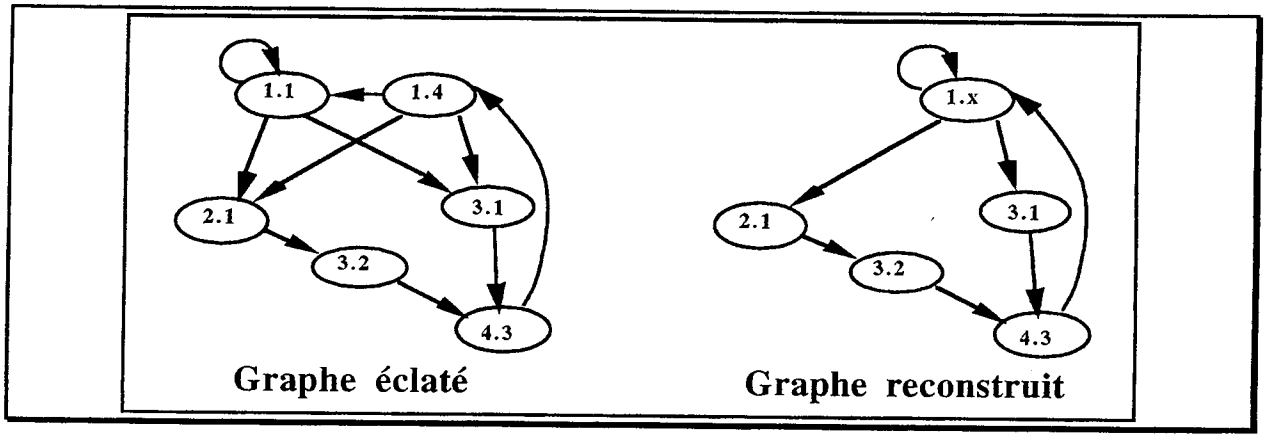

Figure Ax.II.1B - Phase de reconstruction du graphe.

ANNEXE III.1 - GRAMMAIRE DES FICHIERS CFC

L'annexe III.1 décrit la grammaire du fichier des contraintes utilisateurs pouvant être spécifiées lors de la synthèse d'un contrôleur câblé avec l'architecture CFC NoAjs (paragraphe III.1.3). Les principaux paramètres sont :

POL_SPEC : définition du polynôme diviseur du MISR (des polynômes diviseurs par défaut existent jusqu'au degré 16). Un polynôme de degré d est défini par l'entier dérivé de ses d+1 coefficients. Cet entier peut être donné en binaire.

MISR_SIZE : permet de définir la taille du MISR si l'utilisateur veut une signature comportant un nombre de bits supérieur au nombre de bits des codes des états.

ST_CODE : liste de codes d'états. Permet de forcer le code de toute ou partie des états du contrôleur.

BEFORE : spécifie la valeur de la signature de référence voulue avant l'état indiqué.

AFTER : spécifie la valeur de la signature de référence voulue après l'état indiqué. Permet par exemple de forcer la valeur de la signature après les points de comparaison ou l'état d'initialisation.

CMP_POINT : spécifie la liste des états après lesquels la signature doit être vérifiée.

```
CFCFILE:
        ^
        [POL_SPEC]
        [BIT_SUP]
        [MISR_SIZE]
        [ST_CODE]
        [BEFORE]
        [AFTER]
        [CMP_POINT]

POL_SPEC:
        PolSpec ^ = ^ [INT]; ^

BIT_SUP:
        BitSup ^ = ^ [INT ^ [STATE_LIST]]; ^

MISR_SIZE:
        MisrSize ^ = ^ [INT ^]; ^

ST_CODE:
        StateCode ^ = ^ [STATE_LIST]; ^

BEFORE:
        Before ^ = ^ [STATE_LIST]; ^

AFTER:
        After ^ = ^ [STATE_LIST]; ^

CMP_POINT:
        CmpPoint ^ = ^ STR_LIST;^
```

```
STATE_LIST:
        STRING ^ INT ^ [STATE_LIST]

STR_LIST:
        STRING ^ [STRING]

STRING:
        ALPHA [ALPHA | NUM]

ALPHA:
        {n'importe quel caractère autre que ` ` `;` `#` `=` `\n` `\t` End_of_File}

NUM:
        0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

INT:
        [%B | %H | %O | %D] NUM_LIST

NUM_LIST:
        META_NUM [NUM_LIST]

META_NUM:
        A | B | C | D | E | F | NUM

^:
        [COMMENT] [SPACE] [TAB] [RC] [^]

COMMENT:
        # [{n'importe quel caractère autre que '\n' End_of_File}] RC

RC:
        '\n'

TAB:
        '\t'

SPACE:
        ' '
```

Figure Ax.III.1A - Grammaire du fichier de contraintes utilisateur.

```
# Définition du polynôme diviseur x^5 + x^2 + 1
PolSpec = %b100101;
# Signature forcée à 0 après l'état "init"
After = init %d0
        st4 %d10;
# Définition d'un ensemble de trois points de comparaison (états "st4", "st9" et "st17")
# La comparaison est effectuée après la compaction du code de l'état
CmpPoint = st4 st9 st17;
# remarque : le nom des états doit être celui utilisé dans la spécification VHDL du contrôleur
```

Figure Ax.III.1B - Exemple de fichier de contraintes utilisateur.

ANNEXE III.2 - GRAPHE DE CONTRÔLE DE L'EXEMPLE EX4

La figure Ax.III.2A présente le graphe de contrôle de l'exemple ex4. Seules les sorties à 1 sont notées. Les sorties sont appelées o_1, o_2, ... , o_9 et les entrées i_1, i_2, ... , i_5. L'annexe III.2 est citée au paragraphe III.3.2.

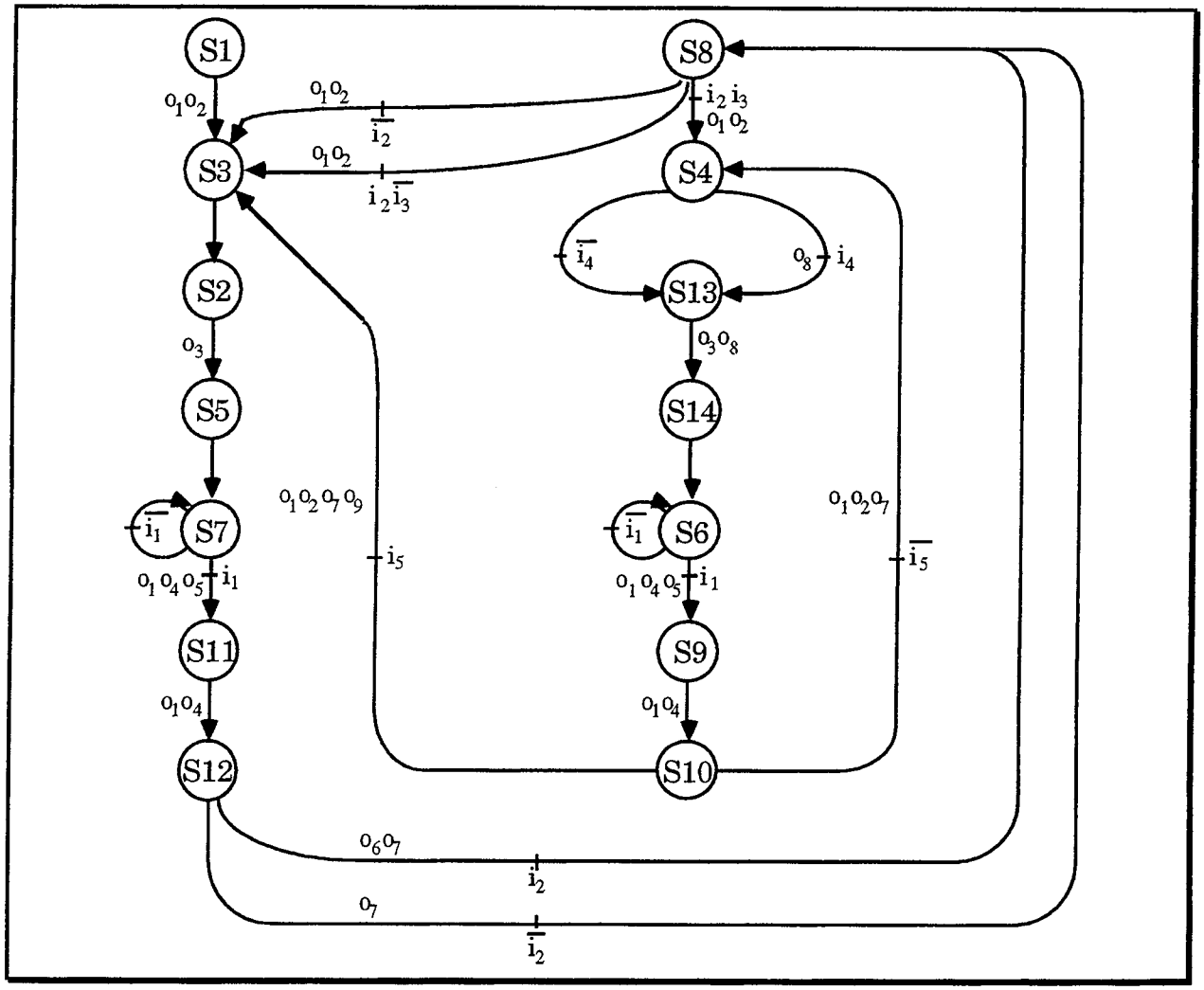

Figure Ax.III.2A - Graphe de contrôle de l'exemple ex4.

ANNEXE III.3 - RÉSULTATS DE LA RÉPARATION DES NSC-GRAPHES

Dans les figures Ax.III.3A et B, les noms des colonnes ont la même signification que dans le tableau III.1 du paragraphe III.4.1. La colonne « Nb d'états Comparaison » correspond à la différence entre le nombre d'états obtenus avec ASYL et celui obtenu avec I-Tool.

Nom	Graphe original				Gmax		Résultats ASYL				Résultats I-Tool				Nb d'états
	Type	Nb d'états	Nb Trans	Nb Classes	Card max	Nb d'états	Nom	nb d'états	delta	nb trans	Max mem (Ko)	CPU (s)	Nom	Nb d'états	Nb Trans	CPU (s)	Comparaison
bbara	Mealy	10	60	1	10	37	bbara	37	27	222	57	0,52	bbara	37	222	14,1	0
bbl13	Moore	504	7655	48,1	3	774	bbl13	527	23	8622	25 193	26,27	bbl13				
bbsse	Mealy	16	56	4	13	42	bbsse	34	18	141	56	1,29	bbsse	37	149	0,1	-3
bbtas	Mealy	6	24	1	6	12	bbtas	10	4	40	38	0	bbtas	10	40	0	0
beecount	Mealy	7	28	1	7	23	beecount	23	16	92	47	0,09	beecount	23	92	0,1	0
cse	Mealy	16	91	1	16	55	cse	55	39	354	148	1,15	cse	55	354	0,1	0
dk14	Mealy	7	56	1	7	27	dk14	23	16	184	46	0,12	dk14	19	152	0	4
dk15	Mealy	4	32	1	4	12	dk15	12	8	96	40	0,01	dk15	12	96	0	0
dk16	Mealy	27	108	1	27	102	dk16	75	48	300	107	64,93	dk16	67	268	0,2	8
dk17	Mealy	8	32	1	8	23	dk17	18	10	72	40	0,04	dk17	17	68	0	1
dk27	Mealy	7	14	3	3	13	dk27	10	3	20	37	0,02	dk27	10	20	0	0
dk512	Mealy	15	30	9	3	30	dk512	19	4	38	41	0,06	dk512	19	38	0	0
dmaread3	Mealy	10	874	1	10	31	dmaread3	31	21	3243	342	0,28	dmaread3	31	3243	1,9	0
dmawr	Mealy	10	457	1	10	33	dmawr	33	23	1773	170	0,29	dmawr	33	1773	0,8	0
donfile	Mealy	24	96	1	24	96	donfile	96	72	384	106	16,96	donfile	96	384	0,1	0
dstate	Mealy	12	63	1	12	35	dstate	32	20	200	62	0,1	dstate	31	192	0,1	1
dvram	Mealy	35	477	29	7	47	dvram	41	6	59	103	0,09	dvram	41	59	0,1	0
esd	Mealy	21	154	1	21	66	esd	66	45	600	246	3,37	esd	66	600	0,2	0
essai1	Mealy	8	32	1	8	23	essai1	18	10	72	42	0,06	essai1	18	72	0	0
essai2	Mealy	8	45	1	8	35	essai2	35	27	204	72	0,04	essai2	35	204	0,1	0
essai3	Mealy	12	384	1	12	144	essai3	144	132	4608	873	2,99	essai3	144	4608	1,4	0
essai4	Mealy	9	306	1	9	29	essai4	29	20	1214	109	0,12	essai4	29	1214	0,4	0
ex1	Mealy	20	138	1	20	73	ex1	69	49	455	119	8,58	ex1	69	455	0,2	0
ex2	Mealy	19	73	1	19	57	ex2	45	26	147	156	6,78	ex2	42	136	0,1	3
ex3	Mealy	10	37	1	10	29	ex3	19	9	66	45	0,25	ex3	23	72	0	-4
ex4	Mealy	14	21	14	1	18	ex4	14	0	21	45	0,01	ex4	14	21	0	0
ex5	Mealy	9	33	1	9	26	ex5	23	14	72	41	0,15	ex5	19	56	0	4
ex6	Mealy	8	34	2	7	31	ex6	32	24	139	51	0,31	ex6	29	126	0	3
ex7	Mealy	10	37	1	10	27	ex7	22	12	67	42	0,23	ex7	18	56	0	4
ex_moore	Moore	8	16	2	4	16	ex_moore	16	8	32	39	0,02	ex_moore				
exem1	Moore	19	19	19	1	2,5	exem1	19	0	19	48	0,02	exem1				
exem2	Moore	20	30	20	1	30	exem2	20	0	30	51	0,02	exem2				
fetch	Mealy	26	35	25	2	35	fetch	27	1	37	65	0,03	fetch	27	37	0	0
fsm	Moore	4	10	1	4	7	fsm	6	2	14	36	0,01	fsm				
gestionfifo	Mealy	7	13	4	4	13	gestionfifo	12	5	29	37	0,01	gestionfifo	12	29	0	0
hsurf	Mealy	46	249	46	1	77	hsurf	46	0	249	175	0,1	hsurf	46	249	6,7	0
hsurf32	Mealy	32	1507	14	19	64	hsurf32	49	17	3582	414	0,48	hsurf32	49	3398	6,7	0
hsurf32mod	Mealy	55	3598	55	1	136	hsurf32mod	55	0	3598	655	0,15	hsurf32mod	55	3417	6,7	0
hyeti1	Moore	175	625	163	9	295	hyeti1	194	19	782	1 894	7,11	hyeti1				
hyeti2	Moore	107	665	89	17	296	hyeti2_kis	153	46	837	723	571,31	hyeti2				
imec1	Moore	101	226	68	15	158	imec1	148	47	346	880	0,71	imec1	148	346	0,2	0
imec10	Moore	96	130	84	4	122	imec10	109	13	150	542	0,45	imec10	109	150	0,1	0
imec2	Moore	409	974	247	6	859	imec2	795	386	2152	74 141	37,56	imec2	795	2152	2,5	0
imec3	Moore	318	918	264	3	411	imec3	386	68	1023	10 492	8,02	imec3	386	1021	0,8	0
imec4	Moore	318	619	264	3	411	imec4	386	68	724	10 459	8,8	imec4	386	722	0,6	0
imec5	Moore	213	350	196	3	251	imec5	232	19	384	2 975	2,78	imec5	233	385	0,2	-1
imec6	Moore	75	152	61	3	110	imec6	98	23	199	466	0,28	imec6	98	199	0,1	0
imec7	Moore	178	409	91	6	364	imec7	348	170	822	7 561	4,56	imec7	348	822	0,5	0
imec8	Moore	227	366	221	2	257	imec8	234	7	391	3 063	2,82	imec8	234	391	0,2	0
imec9	Moore	190	1340	139	6	298	imec9	278	88	2358	4 334	3,24	imec9	279	2358	0,9	-1
jav	Moore	58	123	1	5,8	144	jav	144	86	305	2 019	10,18	jav				
keyb	Mealy	19	170	1	19	46	keyb	37	18	326	100	0,47	keyb	36	314	0,2	1
kirkman	Mealy	17	370	16	2	19	kirkman	18	1	376	86	0,02	kirkman	18	376	0,2	0

Figure Ax.III.3A - Résultats de la réparation des NSC-Graphes en terme de nombre d'états.

Nom	Type	Graphe original: Nb d'états	Graphe original: Nb Trans	Graphe original: Nb Classes	Graphe original: Card max	Gmax: Nb d'états	Résultats ASYL: Nom	Résultats ASYL: nb d'états	Résultats ASYL: delta	Résultats ASYL: nb trans	Résultats ASYL: Max mem (Ko)	Résultats ASYL: CPU (s)	Résultats I-Tool: Nom	Résultats I-Tool: Nb d'états	Résultats I-Tool: Nb Trans	Résultats I-Tool: CPU (s)	Nb d'états: Comparaison
lion	Mealy	4	11	1	4	10	lion	10	6	28	36	0,01	lion	10	28	0	0
lion9	Mealy	9	25	1	9	25	lion9	25	16	71	46	0,02	lion9	25	71	0	0
log	Mealy	17	29	10	5	29	log	23	6	39	61	0,06	log	22	38	0	1
mark1	Mealy	15	23	10	6	22	mark1	20	5	28	48	0,02	mark1	21	71	0,1	-1
mc	Mealy	4	10	1	4	8	mc	8	4	20	37	0,01	mc	8	20	0	0
modulo12	Mealy	12	24	1	12	24	modulo12	24	12	48	46	0,02	modulo12	24	48	0	0
nucpwr	Mealy	29	43	26	2	43	nucpwr	31	2	47	78	0,05	nucpwr	31	47	0,1	0
opus	Mealy	10	22	3	8	21	opus	19	9	47	44	0,03	opus	20	73	0	1
pair	Mealy	16	48	1	16	31	pair	31	15	160	59	0,04	pair	31	160	0,1	0
planet	Mealy	48	115	41	2	71	planet	55	7	128	158	0,25	planet	54	127	0,1	1
planet1	Mealy	48	115	41	2	71	planet1	55	7	128	158	0,25	planet1	54	127	0,1	1
pma	Mealy	24	73	10	11	49	pma	36	12	100	85	0,43	pma	35	97	0,1	1
protocole	Moore	20	26	13	5	32	protocole	27	7	38	61	0,06	protocole				
ram_test	Mealy	72	117	57	3	117	ram_test	87	15	147	398	0,89	ram_test	87	147	0,2	0
rie	Mealy	29	49	25	2	49	rie	33	4	56	92	0,08	rie	33	56	0	0
s1	Mealy	20	107	1	20	80	s1	64	44	363	138	1,31	s1	68	391	0,2	-4
s1488	Mealy	48	251	1	48	117	s1488	97	49	587	1 882	1538,5	s1488	83	450	1,2	14
s1494	Mealy	48	250	1	48	117	s1494	95	47	626	1 515	2149,4	s1494	83	478	1,1	12
s1a	Mealy	20	107	1	20	80	s1a	64	44	363	138	1,31	s1a	68	391	0,2	-4
s208	Mealy	18	153	2	17	35	s208	28	10	209	80	0,62	s208	27	204	0,2	1
s27	Mealy	6	34	1	6	25	s27	25	19	139	44	0,08	s27	25	139	0	0
s386	Mealy	13	64	1	13	39	s386	30	17	173	56	0,46	s386	32	184	0	-2
s420	Mealy	18	137	2	17	35	s420	28	10	198	82	0,65	s420	27	192	0,2	1
s510	Mealy	47	77	42	4	75	s510	52	5	82	196	0,47	s510	52	82	0,1	0
s8	Mealy	5	20	1	5	13	s8	13	8	52	39	0,01	s8	13	52	0,6	0
s820	Mealy	25	232	1	25	107	s820	104	79	886	185	138,47	s820	101	865	0,5	3
s832	Mealy	25	245	1	25	107	s832	106	81	1224	199	367,89	s832	101	1166	0,6	5
sand	Mealy	32	184	1	32	90	sand	88	56	632	448	17,1	sand	88	632	0,3	0
scf	Mealy	121	166	120	2	151	scf	122	1	167	715	0,67	scf	122	448	16,7	0
segbis	Mealy	141	489	74	60	265	segbis	229	88	941	4 004	21,26	segbis	224	945	2,5	5
shiftreg	Mealy	8	16	1	1	16	shiftreg	8	0	16	37	0	shiftreg	8	16	0	0
sr8a_master	Mealy	11	24	5	7	22	sr8a_master	20	9	47	44	0,03	sr8a_master	20	47	0	0
sr8a_slave	Mealy	16	44	5	12	35	sr8a_slave	31	15	92	63	0,26	sr8a_slave	31	94	0,1	0
sse	Mealy	16	56	4	13	42	sse	34	18	141	56	1,26	sse	34	137	0,1	0
stef	Moore	667	5266	475	4	1000	stef	995	328	5962	151 946	82,23	stef				
styr	Mealy	30	166	1	30	92	styr	88	58	563	628	68,72	styr	86	562	0,3	2
sync	Mealy	52	80	52	1	80	sync	52	0	80	161	0,13	sync	52	80	0,1	0
tav	Mealy	4	49	4	1	4	tav	4	0	49	42	0,01	tav	4	49	0	0
tbk	Mealy	32	1569	1	32	216	tbk	216	184	10587	3 079	74,39	tbk	216	10584	4,7	0
tma	Mealy	20	44	12	4	38	tma	28	8	59	63	0,11	tma	26	55	0	2
train11	Mealy	11	25	1	11	25	train11	25	14	59	51	0,07	train11	25	59	0	0
train4	Mealy	4	14	1	4	8	train4	8	4	28	36	0	train4	8	28	0	0
vtiether	Mealy	22	411	5	18	41	vtiether	32	10	59	94	1,15	vtiether	31	57	0,1	1
vtlidec	Mealy	5	37	2	4	9	vtlidec	8	3	72	57	0,01	vtlidec	8	72	0	0
vtiuar	Mealy	24	108	1	24	89	vtiuar	89	65	403	215	3,92	vtiuar	89	403	0,2	0
zeegers	Mealy	120	245	95	16	181	zeegers	143	23	282	892	50,78	zeegers	139	495	13,7	4

Figure Ax.III.3B - Résultats de la réparation des NSC-Graphes en terme de nombre d'états.

ANNEXE III.4 - RÉSULTATS EN SURFACE POUR LA DÉTECTION

Les figures Ax.III.4A, B, C, D et E montrent les résultats obtenus pour la détection des erreurs dans les contrôleurs. Pour ces cinq figures, les colonnes « surface » fournissent un résultat en surface après placement / routage. Les colonnes « MOS » fournissent un résultat sur le nombre de transistors.

Dans la figure Ax.III.4A, une partie indique les caractéristiques des exemples (nombre d'entrées primaires, nombre de sorties primaires, nombre de transitions dans le graphe d'états, nombre d'états et nombre de bits minimum k pour le codage des états), et une partie fournit les résultats obtenus pour l'implantation Simplex (résultats en micronmètres carrés pour la surface).

Dans la figure Ax.III.4B, une partie donne la surface et le nombre de transistors pour l'implantation DPX Seq, et une partie fournit le surcoût de l'implantation DPX Seq comparée à l'implantation Simplex. Le surcoût est calculé pour les résultats en surface et le nombre de transistors. Il est calculé selon la formule suivante : (résultat DPX Seq - résultat Simplex) / résultat Simplex.

Dans la figure Ax.III.4C, une partie donne la surface et le nombre de transistors pour l'implantation CFC Ajs, une partie fournit le surcoût de l'implantation CFC Ajs comparée à l'implantation Simplex, et une partie fournit le surcoût de l'implantation CFC Ajs comparée à l'implantation DPX Seq. Les surcoût sont calculés pour les résultats en surface et le nombre de transistors.

Dans la figure Ax.III.4D, une partie donne la surface et le nombre de transistors pour l'implantation CFC NoAjs, une partie fournit le surcoût de l'implantation CFC NoAjs comparée à l'implantation Simplex, une partie fournit le surcoût de l'implantation CFC NoAjs comparée à l'implantation DPX Seq, et une partie fournit le surcoût de l'implantation CFC NoAjs comparée à l'implantation CFC Ajs. Les surcoût sont calculés pour les résultats en surface et le nombre de transistors.

Dans la figure Ax.III.4E, une partie donne la surface et le nombre de transistors pour l'implantation I-Tool, une partie fournit le surcoût de l'implantation I-Tool comparée à l'implantation Simplex, une partie fournit le surcoût de l'implantation I-Tool comparée à l'implantation DPX Seq, une partie fournit le surcoût de l'implantation I-Tool comparée à l'implantation CFC Ajs, et une partie fournit le surcoût de l'implantation I-Tool comparée à l'implantation CFC NoAjs. Les surcoût sont calculés pour les résultats en surface et le nombre de transistors.

Pour les implantations I-Tool et CFC NoAjs, tous les exemples n'ont pas été synthétisés en raison de l'intervention humaine parfois nécessaire dans le flot de synthèse, et de l'investissement en temps que cela représente.

Caractéristiques des Exemples						Architecture Simplex							
Nom	Nb. Entrées	Nb. Sorties	Nb. Transitions	Nb. Etats	k	Bloc Total		Bloc Séquencement		Bloc de Sorties	
						Surface	MOS	Surface	MOS	Surface	MOS
bbara	4	2	60	10	4	75 202	336	43 854	182	12 240	52
bbsse	7	7	56	16	4	155 783	618	74 760	308	63 508	262
bbtas	2	2	24	6	3	38 001	188	18 034	80	5 493	24
beecount	3	4	28	7	3	55 070	255	29 770	134	12 131	52
cpl8	4	1	37	8	3	108 702	472	58 644	256	28 865	143
cse	7	7	91	16	4	233 337	911	154 691	590	81 431	320
dk14	3	5	56	7	3	110 240	501	44 094	190	47 658	225
dk16	2	3	108	27	5	280 145	1 070	189 226	831	51 204	222
dk17	2	3	32	8	3	72 230	333	37 711	177	24 889	112
dk27	1	2	14	7	3	39 379	182	16 795	72	5 710	30
dk512	1	3	30	15	4	107 013	472	67 946	289	17 248	82
dmaread3	55	44	874	10	4	1 029 501	3 458	228 334	962	753 927	2 697
donfile	2	1	96	24	5	134 996	616	103 010	466	0	
dvram	8	15	47	35	6	253 868	1 074	132 149	536	97 717	410
esd	14	7	154	21	5	306 151	1 239	199 892	815	97 932	383
essai1	2	2	32	8	3	68 369	313	35 965	177	19 166	88
essai2	5	2	45	8	3	140 729	610	91 217	376	39 939	174
essai3	5	2	384	12	4	104 134	456	66 856	278	13 036	58
essai4	23	3	306	9	4	314 988	1 304	186 533	800	172 664	662
ex1	9	19	138	20	5	249 258	978	99 358	416	127 842	518
ex2	2	2	73	19	5	178 998	730	139 837	552	7 177	32
ex3	2	2	37	10	4	91 088	408	59 454	250	9 583	44
ex4	6	9	21	14	4	99 112	436	46 665	210	39 759	164
ex5	2	2	33	9	4	78 313	359	48 308	217	5 465	24
ex6	5	8	34	8	3	153 925	614	40 766	171	103 010	424
ex7	2	2	37	10	4	102 167	429	59 357	266	13 097	60
ex_moore	2	3	16	8	3	51 937	220	23 964	114	4 692	16
exem1	3	7	19	19	5	124 384	532	71 155	312	30 365	133
exem2	4	10	30	20	5	142 758	599	68 563	323	44 700	193
fetch	9	15	35	26	5	148 716	610	78 477	334	44 700	174
fsm	2	3	10	4	2	18 000	85	5 766	25	0	
gestionfifo	2	4	13	7	3	57 048	232	20 991	92	22 391	90
hsurf	23	78	249	46	6	3 064 427	8 862	417 753	1 677	2 681 845	7 522
hsurf32	23	124	1507	32	5	3 593 427	10 287	437 147	1 727	3 265 995	8 712
hyeti1	27	72	625	175	8	5 425 370	13 985	1 429 296	4 999	3 361 565	9 373
hyeti2	23	62	665	107	7	2 819 003	8 589	1 154 814	4 009	1 471 860	4 881
imec1	14	67	226	101	7	1 744 546	5 540	597 974	2 202	1 017 927	3 482
imec10	5	120	130	96	7	2 025 804	6 351	342 119	1 404	1 477 053	4 864
imec4	17	126	619	318	9	17 239 497	27 674	3 012 544	8 948	10 245 406	19 739
imec5	6	76	350	213	8	5 508 806	14 250	1 198 917	4 265	4 220 617	10 434
imec6	6	150	152	75	7	1 472 497	4 871	545 665	2 085	783 959	2 706
imec8	8	35	366	227	8	4 370 411	11 748	1 278 124	4 265	2 593 654	7 731
lav	4	33	123	58	6	424 959	1 638	255 265	954	107 824	498

Figure Ax.III.4A[1] - Caractéristiques des exemples et surface de l'implantation Simplex.

Caractéristiques des Exemples						Architecture Simplex					
						Bloc Total		Bloc Séquencement		Bloc de Sorties	
Nom	Nb. Entrées	Nb. Sorties	Nb. Transitions	Nb. Etats	k	Surface	MOS	Surface	MOS	Surface	MOS
kexb	7	2	170	19	5	314 988	1 328	283 591	1 078	21 109	108
kirkman	12	6	370	17	5	190 706	779	39 927	200	110 020	474
lion	2	1	11	4	2	24 691	126	10 106	48	3 740	18
lion9	2	1	25	9	4	51 113	250	26 879	124	3 070	14
log	9	24	29	17	5	194 443	716	111 337	470	56 471	196
mc	3	5	10	4	2	36 960	193	14 387	58	12 111	66
modulo12	1	1	24	12	4	51 937	242	28 393	122	0	
opus	5	6	22	10	4	110 244	480	48 497	236	52 250	222
pair	4	1	48	16	4	57 859	284	23 490	102	12 560	71
planet	7	19	115	48	6	613 815	2 414	310 078	1 288	308 972	1 074
planet1	7	19	115	48	6	653 068	2 435	312 384	1 211	306 979	1 096
protocole	9	9	26	20	5	121 219	556	58 532	282	28 275	116
ram_test	16	24	117	72	7	728 109	2 739	477 868	1 860	197 122	837
rie	9	26	49	29	5	283 689	1 000	174 573	646	59 633	234
s1	8	6	107	20	5	414 250	1 386	234 183	872	133 932	568
s1a	8	6	107	20	5	255 974	962	210 528	812	0	
s208	11	2	153	18	5	112 787	504	63 682	266	23 699	112
s27	4	1	34	6	3	32 052	156	13 066	54	5 433	26
s386	7	7	64	13	4	165 732	642	76 984	318	63 574	294
s420	19	2	137	18	5	111 169	485	53 369	237	25 557	114
s510	19	7	77	47	6	419 629	1 610	334 596	1 212	49 494	216
s8	4	1	20	5	3	40 108	190	21 546	100	0	
s820	18	19	232	25	5	349 018	1 421	272 295	1 007	90 869	414
s832	18	19	245	25	5	331 970	1 272	224 734	950	76 782	356
scf	27	56	166	121	7	1 081 292	3 768	728 194	2 789	254 560	874
shiftreg	1	1	16	8	3	21 025	90	0		0	
sr8a_master	7	7	24	11	4	107 947	510	62 454	284	39 241	186
sse	7	7	56	16	4	155 783	618	74 760	308	63 508	262
styr	9	10	166	30	5	577 238	2 230	339 442	1 254	190 166	778
sync	19	7	80	52	6	394 279	1 609	291 161	1 100	72 693	324
tav	4	4	49	4	2	33 858	167	3 457	15	19 170	92
tbk	6	3	1569	32	5	351 842	1 276	289 926	1 104	82 137	355
tma	7	6	44	20	5	163 298	730	86 767	362	41 493	188
train11	2	1	25	11	4	73 685	306	39 997	176	2 611	10
train4	2	1	14	4	2	23 880	125	13 713	55	2 162	10
vtiether	1	5	41	22	5	115 804	516	75 189	318	15 526	66
vtijdec	11	32	37	5	3	212 962	807	24 896	123	163 760	641
vtiuar	8	26	108	24	5	278 771	1 010	164 832	692	74 906	267

Figure Ax.III.4A² - Caractéristiques des exemples et surface de l'implantation Simplex.

Architecture DPX Seq				
Nom	Total		DPX Seq / Simplex	
	Surface	MOS	Surface	MOS
bbara	168 385	714	124%	113%
bbsse	281 465	1 176	81%	90%
bbtas	92 887	408	144%	117%
beecount	122 997	544	123%	113%
cpt8	197 480	879	82%	86%
cse	459 247	1 798	97%	97%
dk14	187 173	829	70%	65%
dk16	515 201	2 256	84%	111%
dk17	151 639	690	110%	107%
dk27	90 626	398	130%	119%
dk512	221 575	958	107%	103%
dmaread3	1 279 030	4 919	24%	42%
donfile	291 565	1 304	116%	112%
dvram	464 668	1 928	83%	80%
esd	583 260	2 385	91%	92%
essai1	142 423	666	108%	113%
essai2	273 699	1 150	94%	89%
essai3	215 182	912	107%	100%
essai4	614 166	2 560	95%	96%
ex1	412 103	1 722	65%	76%
ex2	372 395	1 508	108%	107%
ex3	196 927	842	116%	106%
ex4	201 525	882	103%	102%
ex5	170 517	756	118%	111%
ex6	235 868	990	53%	61%
ex7	200 246	890	96%	107%
ex_moore	103 948	468	100%	113%
exem1	258 219	1 129	108%	112%
exem2	267 370	1 211	87%	102%
fetch	287 199	1 214	93%	99%
fsm	45 749	200	154%	135%
gestionfifo	115 700	498	103%	115%
hsurf	3 620 004	11 322	18%	28%
hsurf32	4 225 834	12 538	18%	22%
hyeti1	6 357 028	19 965	17%	43%
hyeti2	3 901 251	13 419	38%	56%
imec1	2 333 638	8 406	34%	52%
imec10	2 281 053	8 192	13%	29%
imec4	16 424 474	38 303	-5%	38%
imec5	6 755 322	19 558	23%	37%
imec6	1 995 051	7 396	35%	52%
imec8	5 286 773	16 855	21%	43%
jay	721 007	2 852	70%	74%
keyb	673 835	2 636	114%	98%
kirkman	275 418	1 246	44%	60%
lion	58 169	264	136%	110%
lion9	125 264	560	145%	124%
log	364 690	1 508	88%	111%
mc	75 103	332	103%	72%
modulo12	125 222	542	141%	124%
opus	217 680	992	97%	107%
pair	127 975	573	121%	102%
planet	1 031 782	4 096	68%	70%
planet1	1 034 400	3 964	58%	63%
protocole	230 884	1 052	90%	89%
ram_test	1 272 619	5 077	75%	85%
rie	494 324	1 898	74%	90%
s1	687 843	2 684	66%	94%
s1a	506 601	1 996	98%	107%
s208	236 607	1 016	110%	102%
s27	82 892	358	159%	129%
s386	285 977	1 228	73%	91%
s420	217 840	960	96%	98%
s510	821 340	3 086	96%	92%
s8	94 419	424	135%	123%
s820	721 003	2 800	107%	97%
s832	611 795	2 628	84%	107%
scf	1 830 710	6 972	69%	85%
shiftreg	51 327	224	144%	149%
sr8a_master	232 586	1 052	115%	106%
sse	281 465	1 176	81%	90%
styr	954 595	3 658	65%	64%
sync	757 608	2 970	92%	85%
tav	60 303	272	78%	63%
tbk	747 533	2 935	112%	130%
tma	300 572	1 284	84%	76%
train11	151 041	660	105%	116%
train4	63 806	270	167%	116%
vtiether	251 448	1 074	117%	108%
vtiidec	264 879	1 111	24%	38%
vtiuar	490 114	2 023	76%	100%

Figure Ax.III.4B - Résultats en surface et nombre de transistors pour l'implantation DPX Seq.

Architecture CFC Ajs						
	Bloc Total		CFC Ajs / Simlex		CFC Ajs / DPX Seq	
Nom	Surface	MOS	Surface	MOS	Surface	MOS
bbara	246 051	1 027	227%	206%	46%	44%
bbsse	350 436	1 389	125%	125%	25%	18%
bbtas	108 945	529	187%	181%	17%	30%
beecount	158 276	708	187%	178%	29%	30%
cpt8	193 698	829	78%	76%	-2%	-6%
cse	489 830	1 935	110%	112%	7%	8%
dk14	245 328	1 060	123%	112%	31%	28%
dk16	603 217	2 362	115%	121%	17%	5%
dk17	174 505	790	142%	137%	15%	14%
dk27	106 938	503	172%	176%	18%	26%
dk512	236 745	1 035	121%	119%	7%	8%
dmaread3	1 215 888	4 223	18%	22%	-5%	-14%
donfile	480 702	2 020	256%	228%	65%	55%
dvram	398 726	1 641	57%	53%	-14%	-15%
esd	635 954	2 479	108%	100%	9%	4%
essai1	168 644	767	147%	145%	18%	15%
essai2	291 067	1 184	107%	94%	6%	3%
essai3	926 836	3 386	790%	643%	331%	271%
essai4	751 425	2 836	139%	117%	22%	11%
ex1	557 828	2 273	124%	132%	35%	32%
ex2	398 753	1 543	123%	111%	7%	2%
ex3	248 676	1 065	173%	161%	26%	26%
ex4	181 945	808	84%	85%	-10%	-8%
ex5	214 089	919	173%	156%	26%	22%
ex6	282 193	1 149	83%	87%	20%	16%
ex7	244 003	1 027	139%	139%	22%	15%
ex_moore	130 291	586	151%	166%	25%	25%
exem1	249 476	1 069	101%	101%	-3%	-5%
exem2	293 325	1 217	105%	103%	10%	0%
fetch	279 063	1 192	88%	95%	-3%	-2%
fsm						
gestionfifo	124 604	571	118%	146%	8%	15%
hsurf						
hsurf32	4 347 486	11 484	21%	12%	3%	-8%
hyeti1	8 272 413	17 801	52%	27%	30%	-11%
hyeti2	4 637 104	12 814	64%	49%	19%	-5%
imec1	2 291 179	7 270	31%	31%	-2%	-14%
imec10	2 545 278	7 649	26%	20%	12%	-7%
imec4	17 905 703	31 161	4%	13%	9%	-19%
imec5	6 274 652	16 041	14%	13%	-7%	-18%
imec6	1 839 392	5 936	25%	22%	-8%	-20%
imec8	4 737 395	13 159	8%	12%	-10%	-22%
jay	985 822	3 648	132%	123%	37%	28%
keyb	690 903	2 635	119%	98%	3%	0%
kirkman	268 894	1 289	41%	65%	-2%	3%
lion	78 425	365	218%	190%	35%	38%
lion9	184 752	841	261%	236%	47%	50%
log	321 315	1 307	65%	83%	-12%	-13%
mc	83 968	399	127%	107%	12%	20%
modulo12	164 150	751	216%	210%	31%	39%
opus	244 339	1 006	122%	110%	12%	1%
pair	194 538	861	236%	203%	52%	50%
planet	1 007 450	3 617	64%	50%	-2%	-12%
planet1	1 007 450	3 617	54%	49%	-3%	-9%
protocole	288 731	1 146	138%	106%	25%	9%
ram_test	1 283 730	4 323	76%	58%	1%	-15%
rie						
s1	943 888	3 356	128%	142%	37%	25%
s1a	668 062	2 596	161%	170%	32%	30%
s208	263 652	1 098	134%	118%	11%	8%
s27	154 836	669	383%	329%	87%	87%
s386	329 079	1 301	99%	103%	15%	6%
s420	261 572	1 166	135%	140%	20%	21%
s510	790 128	2 925	88%	82%	-4%	-5%
s8	107 783	531	169%	179%	14%	25%
s820	925 489	3 310	165%	133%	28%	18%
s832	761 069	2 985	129%	135%	24%	14%
scf	1 523 079	5 221	41%	39%	-17%	-25%
shiftreg	80 574	431	283%	379%	57%	92%
sr8a_master	224 627	949	108%	86%	-3%	-10%
sse	350 134	1 387	125%	124%	24%	18%
styr	1 103 911	3 879	91%	74%	16%	6%
sync	701 473	2 784	78%	73%	-7%	-6%
tav	80 080	368	137%	120%	33%	35%
tbk	1 484 600	5 090	322%	299%	99%	73%
tma	345 625	1 421	112%	95%	15%	11%
train11	174 085	821	136%	168%	15%	24%
train4	84 803	377	255%	202%	33%	40%
vtiether	276 692	1 216	139%	136%	10%	13%
vtiidec	285 330	1 148	34%	42%	8%	3%
vtiuar	517 822	2 018	86%	100%	6%	0%

Figure Ax.III.4C - Résultats en surface et nombre de transistors pour l'implantation CFC Ajs.

Architecture CFC NoAjs								
Bloc Total			CFC NoAjs / Simplex		CFC NoAjs / DPX Seq		CFC NoAjs / CFC Ajs	
Nom	Surface	MOS	Surface	MOS	Surface	MOS	Surface	MOS
bbara								
bbsse								
bbtas								
beecount	288486	1238	424%	385%	135%	128%	82%	75%
cpt8	298722,6	1232	175%	161%	51%	40%		49%
cse								
dk14								
dk16								
dk17								
dk27	97446,24	457	147%	151%	8%	15%	-9%	-9%
dk512	186563,52	842	74%	78%	-16%	-12%	-21%	-19%
dmaread3								
donfile								
dvram	441607	1687	74%	57%	-5%	-13%	11%	3%
esd								
essai1								
essai2								
essai3								
essai4								
ex1								
ex2								
ex3	189997	876	109%	115%	-4%	4%	-24%	-18%
ex4	140512,32	648	42%	49%	-30%	-27%	-23%	-20%
ex5								
ex6								
ex7								
ex_moore								
exem1	203212	862	63%	62%	-21%	-24%	-19%	-19%
exem2	221140	962	55%	61%	-17%	-21%	-25%	-21%
fetch	277578,72	1146	87%	88%	-3%	-6%	-1%	-4%
fsm								
gestionfifo	143175,6	622	151%	168%	24%	25%	15%	9%
hsurf	3433201,2	9394	12%	6%	-5%	-17%		
hsurf32	4759105,32	11993	32%	17%	13%	-4%	9%	4%
hyeti1								
hyeti2								
imec1								
imec10								
imec4								
imec5								
imec6								
imec8	4581400		5%		-13%		-3%	
lav	3072867	9119	623%	457%	326%	220%	212%	150%
keyb								
kirkman	228690	1102	20%	41%	-17%	-12%	-15%	-15%
lion								
lion9								
log	308070	1216	58%	70%	-16%	-19%	-4%	-7%
mc								
modulo12								
opus								
pair								
planet	1064503	3739	73%	55%	3%	-9%	6%	3%
planet1	1064503	3739	63%	54%	3%	-6%	6%	3%
protocole								
ram_test								
rie	472329,36	1913	66%	91%	-4%	1%		
s1								
s1a								
s208								
s27								
s386								
s420								
s510	700219,8	2747	67%	71%	-15%	-11%	-11%	-6%
s8								
s820								
s832								
scf								
shiftreg								
sr8a_master	274485,6	1106	154%	117%	18%	5%	22%	17%
sse								
styr	3675234,6	10180	537%	357%	285%	178%	233%	162%
sync	671972,4	2566	70%	59%	-11%	-14%	-4%	-8%
tav								
tbk								
tma	377804,16	1465	131%	101%	26%	14%	9%	3%
train11								
train4								
vtiether								
vtiidec								
vtiuar								

Figure Ax.III.4D - Résultats en surface et nombre de transistors pour l'implantation CFC NoAjs.

Architecture I-Tool										
Bloc Total			I-Tool / Simplex		I-Tool / DPX Seq		I-Tool / CFC Ajs		I-Tool / CFC NoAjs	
Nom	Surface	MOS	Surface	MOS	Surface	MOS	Surface	MOS	Surface	MOS
bbara										
bbsse										
bbtas										
beecount	135108	649	145%	155%	10%	19%	-15%	-8%	-53%	-48%
cpt8	161971	823	49%	74%	-18%	-6%	-16%	-1%	-46%	-33%
cse										
dk14										
dk16										
dk17										
dk27	89672	443	128%	143%	-1%	11%	-16%	-12%	-8%	-3%
dk512	168323	774	57%	64%	-24%	-19%	-29%	-25%	-10%	-8%
dmaread3										
donfile										
dvram	426641	1701	68%	58%	-8%	-12%	7%	4%	-3%	1%
esd										
essai1										
essai2										
essai3										
essai4										
ex1										
ex2										
ex3	203675	902	124%	121%	3%	7%	-18%	-15%	7%	3%
ex4	143327	666	45%	53%	-29%	-24%	-21%	-18%	2%	3%
ex5										
ex6										
ex7										
ex_moore										
exem1										
exem2										
fetch	304990	1305	105%	114%	6%	7%	9%	9%	10%	14%
fsm										
gestionfifo	125364	617	120%	166%	8%	24%	1%	8%	-12%	-1%
hsurf	3290883	9470	7%	7%	-9%	-16%			-4%	1%
hsurf32	4395387	11274	22%	10%	4%	-10%	1%	-2%	-8%	-6%
hyeti1										
hyeti2										
imec1	2493702	7354	43%	33%	7%	-13%	9%	1%		
imec10	2727159	8110	35%	28%	20%	-1%	7%	6%		
imec4	16317644	28826	-5%	4%	-1%	-25%	-9%	-7%		
imec5										
imec6										
imec8										
jay										
keyb										
kirkman	284019	1222	49%	57%	3%	-2%	6%	-5%	24%	11%
lion										
lion9										
log										
mc										
modulo12										
opus										
pair										
planet										
planet1										
protocole										
ram_test										
rie										
s1										
s1a										
s208										
s27										
s386										
s420										
s510	473175	2025	13%	26%	-42%	-34%	-40%	-31%	-32%	-26%
s8										
s820										
s832										
scf	1278221	4473	18%	19%	-30%	-36%	-16%	-14%		
shiftreg										
sr8a_master	233741	966	117%	89%	0%	-8%	4%	2%	-15%	-13%
sse										
styr	1594529	5404	176%	142%	67%	48%	44%	39%	-57%	-47%
sync	566915	2315	44%	44%	-25%	-22%	-19%	-17%	-16%	-10%
tav										
tbk	1559509	5283	343%	314%	109%	80%	5%	4%		
tma	250211	1086	53%	49%	-17%	-15%	-28%	-24%	-34%	-26%
train11										
train4										
vtiether										
vtiidec										
vtiuar										

Figure Ax.III.4E - Résultats en surface et nombre de transistors pour l'implantation I-Tool.

ANNEXE IV.1 - RÉSULTATS POUR L'IMPLANTATION DE TYPE [MEYE 71]

La figure Ax.IV.1A montre les résultats obtenus pour l'implantation de type [Meye 71]. Ces résultats sont indiqués en nombre de transistors. Le détail de la complexité de chacun des blocs constituant l'architecture est indiqué (bloc de séquencement, bloc de calcul des sorties, registre d'états, et signal de correction). Les colonnes « Surcoût » indiquent le surcoût de l'implantation masquant les erreurs comparée à l'implantation Simplex. Un surcoût est calculé pour l'implantation masquant les erreurs sans signal de correction, et un surcoût est calculé pour l'implantation avec le signal de correction.

Nom	Complexite sans Signal decorrection					Avec Signal de Correction		
	Sequencement	Sorties	Registre	Total	Surcoût	Signal	Total	Surcoût
bbara	3023	144	210	3377	905%	206	3583	966%
bbsse	3200	1018	210	4428	617%	126	4554	637%
bbtas	759	56	180	995	429%	127	1122	497%
beecount	1080	206	180	1466	475%	108	1574	517%
cpt100	19136	130	330	19596	980%	387	19983	1002%
cpt8	1995	410	180	2585	448%	108	2693	471%
cse	6426	1213	210	7849	762%	134	7983	776%
dk14	1997	879	180	3056	510%	116	3172	533%
dk15	708	505	150	1363	287%	72	1435	308%
dk16	12202	1646	270	14118	1219%	217	14335	1240%
dk17	1436	528	180	2144	544%	116	2260	579%
dk27	844	154	180	1178	547%	108	1286	607%
dk512	2640	398	210	3248	588%	126	3374	615%
dmawr	5396	4258	210	9864	190%	223	10087	197%
donfile	9463	0	270	9733	1480%	207	9940	1514%
dstate	4478	1200	210	5888	453%	191	6079	471%
dvram	8623	2220	300	11143	938%	496	11639	984%
esd	11191	1716	270	13177	964%	347	13524	992%
essat1	1667	386	180	2233	613%	102	2335	646%
essai2	2610	523	180	3313	443%	108	3421	461%
essai4	4464	1414	210	6088	367%	202	6290	382%
ex1	8574	3223	270	12067	1134%	310	12377	1166%
ex2	6861	246	270	7377	911%	304	7681	952%
ex3	2434	182	210	2826	593%	195	3021	640%
ex4	1924	844	210	2978	583%	200	3178	629%
ex5	2109	158	210	2477	590%	187	2664	642%
ex6	2242	1532	180	3954	544%	116	4070	563%
ex7	2220	266	210	2696	528%	191	2887	573%
ex_moore	1336	320	180	1836	735%	108	1944	784%
exem1	3546	794	270	4610	767%	302	4912	823%
exem2	3804	814	270	4888	716%	312	5200	768%
fetch	5070	1178	270	6518	969%	258	6776	1011%
fsm	271	71	150	492	479%	72	564	564%
gestionfifo	929	524	180	1633	604%	116	1749	654%
jay	17639	2805	300	20744	1166%	202	20946	1179%
keyb	9105	511	270	9886	644%	304	10190	667%
kirkman	2025	2238	270	4533	482%	295	4828	520%
lion	456	65	150	671	433%	72	743	490%
lion9	1604	114	210	1928	671%	182	2110	744%
log	4004	1026	270	5300	640%	259	5559	676%
mark1	1692	753	210	2655	495%	120	2775	522%
mc	608	297	150	1055	447%	72	1127	484%
modulo12	2267	0	210	2477	924%	253	2730	1028%
nucpwr	7001	1527	270	8798	836%	235	9033	861%
opus	1758	755	210	2723	467%	212	2935	511%
pair	2092	502	210	2804	887%	134	2938	935%
planet	14779	7911	300	22990	852%	341	23331	866%
planet1	14779	7911	300	22990	844%	341	23331	858%
pma	7878	2249	270	10397	1098%	276	10673	1130%
protocole	4163	916	270	5349	862%	204	5553	899%
ram_test	25127	4695	330	30152	1001%	998	31150	1037%
rie	8098	1655	270	10023	902%	243	10266	927%
s1	10873	2984	270	14127	919%	235	14362	936%
s1488	15899	9760	300	25959	830%	572	26531	851%
s1494	15972	9509	300	25781	817%	659	26440	840%
s1a	10595	0	270	10865	1029%	289	11154	1059%
s208	2750	675	270	3695	633%	244	3939	682%
s27	1161	164	180	1505	865%	133	1638	960%
s386	3098	1136	210	4444	592%	223	4667	627%
s420	2772	822	270	3864	697%	247	4111	748%
s510	14460	3120	300	17880	1011%	513	18393	1042%
s8	1076	0	180	1256	561%	106	1362	617%
s820	12061	1734	270	14065	890%	172	14237	902%
s832	13712	1867	270	15849	1146%	285	16134	1168%
sand	15470	4155	270	19895	683%	151	20046	689%
scf	36796	8464	330	45590	1110%	197	45787	1115%
shiftreg	336	84	180	600	567%	108	708	687%
sr8a_master	2289	606	210	3105	509%	244	3349	557%
sr8a_slave	4451	1830	210	6491	639%	134	6625	655%
sse	3200	1018	210	4428	617%	126	4554	637%
styr	16537	4493	270	21300	855%	151	21451	862%
sync	14846	3337	300	18483	1049%	202	18685	1061%
tav	190	114	150	454	172%	72	526	215%
tbk	32054	2068	270	34392	2595%	151	34543	2607%
tma	4811	1482	270	6563	799%	194	6757	826%
train11	1713	56	210	1979	547%	191	2170	609%
train4	431	34	150	615	392%	72	687	450%
vtiether	5397	864	270	6531	1166%	229	6760	1210%
vtiidec	854	1404	180	2438	202%	106	2544	215%
vtiuar	8732	1743	270	10745	964%	204	10949	984%
Moyenne :					743%			779%
Min :					172%			197%
Max :					2595%			2607%

Figure Ax.IV.1A - Résultats en nombre de transistors pour l'implantation de type [Meye 71] et surcoût par rapport à l'implantation Simplex.

ANNEXE IV.2 - CARACTÉRISTIQUES DES EXEMPLES SYNTHÉTISÉS

La figure Ax.IV.2A donne les caractéristiques des différents exemples synthétisés : nombre d'entrées primaires, nombre de sorties primaires, nombre de transitions, nombre d'états, nombre minimum de bits nécessaires k pour coder les états, et nombre minimum de bits nécessaires n pour coder les états avec un code correcteur d'erreurs simples.

Nom	Nb. Entrées	Nb. Sorties	Nb. Transitions	Nb. Etats	k	n
bbara	4	2	60	10	4	7
bbsse	7	7	56	16	4	7
bbtas	2	2	24	6	3	6
beecount	3	4	28	7	3	6
cpt100	1	1	200	100	7	11
cpt257	1	1	514	257	9	13
cpt8	4	1	37	8	3	6
cse	7	7	91	16	4	7
dk14	3	5	56	7	3	6
dk15	3	5	32	4	2	5
dk16	2	3	108	27	5	9
dk17	2	3	32	8	3	6
dk27	1	2	14	7	3	6
dk512	1	3	30	15	4	7
dmaread3	55	44	874	10	4	7
dmawr	32	34	457	10	4	7
donfile	2	1	96	24	5	9
dstate	20	12	63	12	4	7
dvram	8	15	47	35	6	10
esd	14	7	154	21	5	9
essai1	2	2	32	8	3	6
essai2	5	2	45	8	3	6
essai3	5	2	384	12	4	7
essai4	23	3	306	9	4	7
ex1	9	19	138	20	5	9
ex2	2	2	73	19	5	9
ex3	2	2	37	10	4	7
ex4	6	9	21	14	4	7
ex5	2	2	33	9	4	7
ex6	5	8	34	8	3	6
ex7	2	2	37	10	4	7
ex_moore	2	3	16	8	3	6
exem1	3	7	19	19	5	9
exem2	4	10	30	20	5	9
fetch	9	15	35	26	5	9
fsm	2	3	10	4	2	5
gestionfifo	2	4	13	7	3	6
hsurf	23	78	249	46	6	10
hsurf32	23	124	1307	32	5	9
hsurf32mod	23	124	3598	55	6	10
hyeti1	27	72	625	175	8	12
hyeti2	23	62	665	107	7	11
inecl	14	67	226	101	7	11
inecl0	5	120	130	96	7	11
jay	4	33	123	58	6	10
keyb	7	2	170	19	5	9
kirkman	12	6	370	17	5	9
lion	2	1	11	4	2	5
lion9	2	1	25	9	4	7
log	9	24	29	17	5	9
mark1	5	16	23	15	4	7
mc	3	5	10	4	2	5
modul612	1	1	24	12	4	7
nucpwr	13	27	43	29	5	9
opus	5	6	22	10	4	7
pair	4	1	48	16	4	7
planet	7	19	115	48	6	10
planet1	7	19	115	48	6	10
pma	8	8	73	24	5	9
protocole	9	9	26	20	5	9
ram_test	16	24	117	72	7	11
rie	9	26	49	29	5	9
s1	8	6	107	20	5	9
s1488	8	19	251	48	6	10
s1494	8	19	250	48	6	10
s1a	8	6	107	20	5	9
s208	11	2	153	18	5	9
s27	4	1	34	6	3	6
s386	7	7	64	13	4	7
s420	19	2	137	18	5	9
s510	19	7	77	47	6	10
s8	4	1	20	5	3	6
s820	18	19	232	25	5	9
s832	18	19	245	25	5	9
sand	11	9	184	32	5	9
scf	27	56	166	121	7	11
shiftreg	1	1	16	8	3	6
sr8a_master	7	7	24	11	4	7
sr8a_slave	5	7	44	16	4	7
sse	7	7	56	16	4	7
styr	9	10	166	30	5	9
sync	19	7	80	32	6	10
tav	4	4	49	4	2	5
tbk	6	3	1569	32	5	9
tma	7	6	44	20	5	9
train11	2	1	25	11	4	7
train4	2	1	14	4	2	5
vtiether	1	5	41	22	5	9
vtuidec	11	32	37	5	3	6
vtuar	8	26	108	24	5	9
zeegers	11	11	245	120	7	11

Figure Ax.IV.2A - Caractéristiques des exemples synthétisés pour les techniques de masquage d'erreurs.

ANNEXE IV.3 - RÉSULTATS EN SURFACE POUR LE MASQUAGE

Les figures Ax.IV.3A et Ax.IV.3B montrent les résultats obtenus pour l'implantation SID (pour les deux flots de synthèse Global et Hamming) et pour l'implantation TMR Seq.

Dans la figure Ax.IV.3A, une partie indique le surcoût de l'implantation SID par rapport à l'implantation Simplex en terme de nombre de transistors, de surface et de chemin critique. Le détail est donné pour les deux flots de synthèse. Une seconde partie indique le flot de synthèse permettant d'obtenir le meilleur résultat pour l'implantation SID. Une troisième partie indique le surcoût de l'implantation TMR Seq comparée à l'implantation Simplex. Enfin, une quatrième partie indique l'implantation ayant conduit au plus petit résultat en terme de nombre de transistors et de surface.

Dans la figure Ax.IV.3B, une partie indique le gain en surface (réduction de la surface du circuit) obtenu, par rapport à une implantation TMR Seq, grâce à l'utilisation d'une implantation SID (le flot de synthèse de Hamming). De même, une seconde partie indique le gain en surface obtenu, par rapport à une implantation TMR Tot, grâce à l'utilisation d'une implantation SID.

Exemples	Surcoût de l'architecture SID comparée au Simplex						Meilleur Codage SID			TMR Seq comparé au Simplex		Surcoût le plus faible	
	MOS		Surface		Chemin Critique (ns)								
Nom	Global	Hamming	Global	Hamming	Global	Hamming	MOS	Surface	Chemin Crit.	MOS	Surface	MOS	Surface
bbara	327,08%	208,63%	346,64%	191,94%	14,60	12,90	Hamming	Hamming	Hamming	206,55%	225,80%	TMR	SID
bbsse	201,13%	123,30%	188,87%	101,72%	18,00	15,30	Hamming	Hamming	Hamming	161,81%	158,67%	SID	SID
bbtas	179,26%	194,15%	194,99%	209,33%	7,10	10,50	Global	Global	Global	212,77%	245,06%	SID	SID
beecount	148,24%	150,98%	150,61%	146,03%	8,70	11,80	Global	Hamming	Global	205,10%	198,02%	SID	SID
cpt100	96,64%	105,68%	88,27%	99,06%	25,90	35,30	Global	Global	Global	204,74%	209,43%	SID	SID
cpt257	154,88%	195,78%	140,10%	192,17%	20,10	41,90	Global	Global	Global	206,67%	200,73%	SID	SID
cpt8	119,49%	125,21%	116,45%	122,24%	8,10	14,10	Global	Global	Global	161,65%	145,33%	SID	SID
cse	228,87%	134,69%	200,49%	112,19%	19,80	15,40	Hamming	Hamming	Hamming	176,84%	166,67%	SID	SID
dk14	140,92%	92,02%	124,17%	88,18%	8,80	9,40	Hamming	Hamming	Global	123,35%	145,36%	SID	SID
dk15	115,34%	91,19%	92,84%	88,15%	5,50	7,80	Hamming	Hamming	Global	105,40%	92,96%	SID	SID
dk16	249,81%	121,59%	212,13%	92,75%	29,10	22,20	Hamming	Hamming	Hamming	204,21%	174,76%	SID	SID
dk17	146,85%	127,93%	146,23%	121,60%	8,80	12,30	Hamming	Hamming	Global	190,39%	184,14%	SID	SID
dk27	201,65%	177,47%	192,06%	168,00%	7,10	11,00	Hamming	Hamming	Global	213,19%	226,73%	SID	SID
dk512	242,58%	118,22%	250,21%	96,75%	16,60	13,50	Hamming	Hamming	Hamming	181,36%	181,00%	SID	SID
dmaread3	77,99%	69,81%	70,70%	74,63%	16,20	12,90	Hamming	Global	Hamming	73,94%	85,02%	SID	SID
dmawr	65,23%	50,93%	65,43%	48,57%	10,10	13,80	Hamming	Hamming	Global	69,71%	69,46%	SID	SID
donfile	402,27%	188,80%	439,62%	183,31%	30,80	19,70	Hamming	Hamming	Hamming	224,35%	225,05%	SID	SID
dstate	193,05%	138,35%	199,32%	145,34%	19,30	15,90	Hamming	Hamming	Hamming	163,91%	189,68%	SID	SID
dvram	203,45%	94,69%	204,79%	92,09%	31,80	24,80	Hamming	Hamming	Hamming	141,90%	152,39%	SID	SID
esd	248,99%	158,51%	253,87%	172,63%	27,10	26,20	Hamming	Hamming	Hamming	171,83%	177,38%	SID	SID
essai1	172,52%	134,82%	159,54%	131,97%	10,00	12,50	Hamming	Hamming	Global	201,28%	177,56%	SID	SID
essai2	123,77%	113,28%	114,46%	107,25%	12,00	16,90	Hamming	Hamming	Global	166,56%	166,22%	SID	SID
essai3	246,93%	115,35%	263,58%	107,19%	5,90	2,70	Hamming	Hamming	Hamming	190,35%	196,34%	SID	SID
essai4	211,73%	171,78%	241,79%	193,99%	7,70	15,80	Hamming	Hamming	Global	167,94%	198,59%	TMR	SID
ex1	276,28%	123,11%	262,93%	113,81%	26,60	20,00	Hamming	Hamming	Hamming	149,69%	138,86%	SID	SID
ex2	257,81%	128,63%	243,72%	102,90%	23,90	19,60	Hamming	Hamming	Hamming	166,58%	218,97%	SID	SID
ex3	240,69%	157,60%	280,47%	136,30%	15,90	13,50	Hamming	Hamming	Hamming	200,49%	202,94%	SID	SID
ex4	252,06%	118,58%	251,41%	109,88%	16,30	11,40	Hamming	Hamming	Hamming	181,19%	185,85%	SID	SID
ex5	242,62%	167,41%	244,02%	157,44%	12,20	13,10	Hamming	Hamming	Global	208,36%	212,05%	SID	SID
ex6	110,26%	103,09%	99,16%	88,39%	7,00	11,80	Hamming	Hamming	Global	105,37%	97,63%	SID	SID
ex7	196,27%	144,29%	185,60%	112,82%	14,30	14,00	Hamming	Hamming	Hamming	200,70%	179,08%	SID	SID
ex_moore	319,09%	167,27%	283,92%	130,70%	8,50	9,90	Hamming	Hamming	Global	210,00%	185,13%	SID	SID
exen1	313,91%	115,23%	346,30%	103,88%	17,70	15,00	Hamming	Hamming	Hamming	187,97%	197,16%	SID	SID
exen2	301,00%	137,23%	310,31%	129,54%	20,20	18,10	Hamming	Hamming	Hamming	189,98%	195,31%	SID	SID
fetch	277,87%	143,61%	276,56%	134,70%	19,90	15,20	Hamming	Hamming	Hamming	181,31%	194,41%	SID	SID
fsm	376,47%	255,29%	399,22%	240,26%	7,20	8,20	Hamming	Hamming	Global	242,35%	272,30%	TMR	SID
gestionfifo	197,84%	173,28%	156,39%	148,80%	7,10	11,70	Hamming	Hamming	Global	197,41%	172,57%	SID	SID
hsurf	49,42%	34,08%	48,69%	37,05%	15,20	24,40	Hamming	Hamming	Global	48,96%	33,50%	SID	TMR
hsurf32	30,67%	30,92%	14,93%	23,90%	36,90	28,90	Global	Global	Hamming	33,62%	31,00%	SID	SID
hsurf32mod	71,24%	38,15%	69,78%	32,49%	32,70	19,60	Hamming	Hamming	Hamming	55,45%	30,62%	SID	TMR
hyeti1	73,80%	39,89%	58,91%	23,90%	49,10	37,70	Hamming	Hamming	Hamming	80,44%	44,83%	SID	SID
hyeti2	142,10%	62,25%	159,97%	50,60%	48,70	32,00	Hamming	Hamming	Hamming	105,66%	81,57%	SID	SID
imec1	117,89%	60,52%	113,93%	47,58%	31,20	23,30	Hamming	Hamming	Hamming	95,74%	71,61%	SID	SID
imec10	89,62%	34,20%	86,31%	28,33%	25,60	27,90	Hamming	Hamming	Global	54,81%	32,56%	SID	SID
jay	287,55%	112,70%	265,13%	94,98%	35,10	22,90	Hamming	Hamming	Hamming	147,50%	159,93%	SID	SID

Figure Ax.IV.3A[1] - Comparaison du codage Global, du codage de Hamming et de l'implantation TMR Seq.

Exemples	Surcoût de l'architecture SID comparée au Simplex				Chemin Critique (ns)		Meilleur Codage SID			TMR Seq		Surcoût
	MOS		Surface							comparé au Simplex		le plus faible
Nom	Global	Hamming	Global	Hamming	Global	Hamming	MOS	Surface	Chemin Crit	MOS	Surface	MOS	Surface
keyb	178,24%	140,81%	174,10%	150,94%	19,40	22,30	Hamming	Hamming	Global	192,32%	213,00%	SID	SID
kirkman	188,45%	79,08%	186,88%	69,60%	14,90	13,30	Hamming	Hamming	Hamming	107,19%	92,24%	SID	SID
lion	191,27%	219,05%	233,60%	220,43%	3,90	8,30	Global	Hamming	Global	200,00%	210,33%	SID	TMR
lion9	276,40%	180,40%	292,77%	191,88%	12,10	12,70	Hamming	Hamming	Global	227,20%	243,83%	SID	SID
log	187,99%	115,50%	167,78%	87,75%	17,30	19,30	Hamming	Hamming	Global	178,49%	172,90%	SID	SID
mark1	224,89%	126,23%	282,17%	132,43%	12,70	14,00	Hamming	Hamming	Global	184,30%	204,10%	SID	SID
mc	120,73%	141,45%	137,15%	161,52%	3,60	8,10	Global	Global	Global	136,27%	175,69%	SID	SID
modulo12	412,81%	221,90%	469,16%	225,20%	22,10	14,40	Hamming	Hamming	Hamming	229,75%	236,23%	SID	SID
mucpwr	180,74%	115,85%	179,52%	105,53%	28,10	22,80	Hamming	Hamming	Hamming	181,91%	184,20%	SID	SID
opus	197,29%	120,83%	223,13%	111,03%	15,00	11,90	Hamming	Hamming	Hamming	183,75%	167,37%	SID	SID
pair	468,31%	210,92%	565,06%	201,49%	15,50	12,40	Hamming	Hamming	Hamming	184,86%	192,43%	TMR	TMR
planet	134,30%	78,83%	130,68%	78,18%	30,20	28,60	Hamming	Hamming	Hamming	131,40%	150,64%	SID	SID
planet1	132,28%	77,29%	116,81%	67,47%	30,50	28,90	Hamming	Hamming	Hamming	120,82%	114,40%	SID	SID
pma	273,85%	122,00%	251,08%	103,01%	27,30	19,40	Hamming	Hamming	Hamming	163,82%	146,75%	SID	SID
protocole	351,08%	130,04%	421,95%	130,76%	21,40	19,40	Hamming	Hamming	Hamming	170,14%	189,19%	SID	SID
ram_test	140,71%	92,19%	134,41%	96,50%	34,80	35,20	Hamming	Hamming	Global	161,88%	169,50%	SID	SID
rie	204,80%	128,10%	161,01%	87,17%	24,80	20,60	Hamming	Hamming	Hamming	179,80%	160,87%	SID	SID
s1	218,98%	136,00%	170,44%	95,29%	25,30	24,40	Hamming	Hamming	Hamming	168,69%	148,51%	SID	SID
s1488	121,86%	69,93%	122,68%	69,06%	30,70	24,00	Hamming	Hamming	Hamming	126,95%	121,36%	SID	SID
s1494	116,22%	64,94%	89,71%	50,21%	31,70	26,00	Hamming	Hamming	Hamming	114,79%	88,82%	SID	SID
s1a	274,53%	192,62%	249,95%	170,57%	26,10	24,30	Hamming	Hamming	Hamming	226,82%	221,59%	SID	SID
s208	249,40%	122,22%	252,14%	126,58%	12,90	14,10	Hamming	Hamming	Global	187,70%	164,90%	SID	SID
s27	372,44%	248,08%	424,88%	272,77%	11,70	10,80	Hamming	Hamming	Hamming	228,21%	268,89%	TMR	TMR
s386	193,93%	128,04%	174,81%	98,75%	16,90	16,60	Hamming	Hamming	Hamming	161,68%	152,52%	SID	SID
s420	287,84%	122,68%	299,57%	112,16%	16,40	15,10	Hamming	Hamming	Hamming	178,76%	170,22%	SID	SID
s510	182,11%	94,91%	169,07%	83,48%	29,50	28,80	Hamming	Hamming	Hamming	171,18%	214,28%	SID	SID
s8	201,58%	214,21%	212,03%	208,68%	8,50	11,60	Global	Hamming	Global	228,42%	245,08%	SID	SID
s820	192,54%	116,68%	197,71%	114,90%	27,80	24,10	Hamming	Hamming	Hamming	161,51%	215,16%	SID	SID
s832	244,81%	136,16%	234,32%	115,34%	50,40	24,10	Hamming	Hamming	Hamming	194,50%	190,94%	SID	SID
sand	132,94%	89,89%	123,75%	82,99%	32,70	23,10	Hamming	Hamming	Hamming	110,23%	126,83%	SID	SID
scf	140,63%	74,02%	122,14%	68,10%	41,60	32,90	Hamming	Hamming	Hamming	148,09%	170,19%	SID	SID
shiftreg	316,67%	306,67%	262,57%	241,95%	6,80	10,40	Hamming	Hamming	Global	260,00%	222,41%	TMR	TMR
sr8a_master	222,16%	120,39%	275,22%	130,66%	14,60	12,60	Hamming	Hamming	Hamming	188,24%	214,35%	SID	SID
sr8a_slave	173,92%	101,82%	150,50%	82,87%	19,20	17,20	Hamming	Hamming	Hamming	159,91%	153,04%	SID	SID
sse	201,13%	123,30%	188,87%	101,72%	18,00	15,30	Hamming	Hamming	Hamming	161,81%	160,41%	SID	SID
styr	175,20%	101,12%	169,19%	90,91%	33,70	27,50	Hamming	Hamming	Hamming	127,80%	131,88%	SID	SID
sync	180,36%	106,96%	178,81%	111,21%	31,10	27,60	Hamming	Hamming	Hamming	164,01%	231,54%	SID	SID
tav	118,56%	121,56%	128,95%	121,11%	3,70	7,10	Global	Hamming	Global	111,38%	122,46%	TMR	SID
tbk	622,02%	262,15%	597,17%	215,31%	47,80	29,70	Hamming	Hamming	Hamming	229,70%	211,75%	TMR	TMR
tma	224,66%	111,64%	222,24%	117,13%	21,00	17,00	Hamming	Hamming	Hamming	168,36%	171,50%	SID	SID
train11	317,32%	147,39%	308,80%	111,72%	15,30	11,30	Hamming	Hamming	Hamming	216,99%	184,70%	SID	SID
train4	236,00%	232,00%	286,97%	232,31%	6,00	8,20	Hamming	Hamming	Global	212,80%	252,51%	TMR	SID
vtether	322,29%	156,78%	343,89%	133,39%	20,30	15,50	Hamming	Hamming	Hamming	218,60%	228,18%	SID	SID
vtidec	50,43%	58,49%	41,06%	43,22%	3,80	9,20	Global	Global	Global	65,30%	52,95%	SID	SID
vtuar	251,68%	145,15%	217,66%	123,67%	26,40	20,90	Hamming	Hamming	Hamming	162,87%	178,66%	SID	SID
zeegers	107,92%	69,20%	88,67%	55,32%	25,30	23,60	Hamming	Hamming	Hamming	129,95%	109,21%	SID	SID
Moyenne	209,20%	129,96%	209,29%	120,19%	20,10	18,21	Hamming	Hamming	Hamming	165,81%	167,91%	SID	SID

Figure Ax.IV.3A[2] - Comparaison du codage Global, du codage de Hamming et de l'implantation TMR Seq.

Nom	Gain en surface de l'Architecture SID comparée au TMR Seq		Gain en surface de l'Architecture SID comparée au TMR Tot	
	Gain	Meilleur	Gain	Meilleur
bbara	10,39%	SID Ham	7,05%	SID Ham
bbsse	22,02%	SID Ham	37,69%	SID Ham
bbtas	10,35%	SID Ham	5,64%	SID Ham
beecount	17,44%	SID Ham	27,30%	SID Ham
cpt100	35,67%	SID Ham	33,91%	SID Ham
cpt257	2,85%	SID Ham	2,95%	SID Ham
cpt8	9,41%	SID Ham	27,10%	SID Ham
cse	20,43%	SID Ham	32,82%	SID Ham
dk14	23,30%	SID Ham	41,92%	SID Ham
dk15	2,49%	SID Ham	43,31%	SID Ham
dk16	29,85%	SID Ham	36,94%	SID Ham
dk17	22,01%	SID Ham	31,17%	SID Ham
dk27	17,98%	SID Ham	18,01%	SID Ham
dk512	29,98%	SID Ham	37,51%	SID Ham
dmaread3	5,62%	SID Ham	45,87%	SID Ham
dmawr	12,33%	SID Ham	53,27%	SID Ham
donfile	12,84%	SID Ham	6,78%	SID Ham
dstate	15,30%	SID Ham	24,36%	SID Ham
dvram	23,89%	SID Ham	42,01%	SID Ham
esd	1,71%	SID Ham	12,64%	SID Ham
essai1	16,42%	SID Ham	26,47%	SID Ham
essai2	22,15%	SID Ham	32,61%	SID Ham
essai3	30,08%	SID Ham	33,20%	SID Ham
essai4	1,54%	SID Ham	3,62%	SID Ham
ex1	10,49%	SID Ham	37,17%	SID Ham
ex2	36,39%	SID Ham	33,67%	SID Ham
ex3	22,00%	SID Ham	24,17%	SID Ham
ex4	26,58%	SID Ham	39,70%	SID Ham
ex5	17,50%	SID Ham	17,88%	SID Ham
ex6	4,67%	SID Ham	42,46%	SID Ham
ex7	23,74%	SID Ham	31,43%	SID Ham
ex_moore	19,09%	SID Ham	30,21%	SID Ham
exem1	31,39%	SID Ham	38,17%	SID Ham
exem2	22,27%	SID Ham	31,90%	SID Ham
fetch	20,28%	SID Ham	33,58%	SID Ham
fsm	8,60%	SID Ham	12,34%	SID Ham
gestionfifo	8,72%	SID Ham	26,19%	SID Ham
hsurf	-2,65%	TMR Seq	56,28%	SID Ham
hsurf32	5,42%	SID Ham	61,07%	SID Ham
hsurf32mod	-1,43%	TMR Seq	58,35%	SID Ham
hyeti1	14,45%	SID Ham	59,64%	SID Ham
hyeti2	17,06%	SID Ham	51,67%	SID Ham
imec1	14,00%	SID Ham	53,93%	SID Ham
imec10	3,19%	SID Ham	61,27%	SID Ham
jay	24,99%	SID Ham	42,83%	SID Ham
keyb	19,83%	SID Ham	17,28%	SID Ham
kirkman	11,77%	SID Ham	46,44%	SID Ham
lion	-3,26%	TMR Seq	0,31%	SID Ham
lion9	15,11%	SID Ham	5,95%	SID Ham
log	31,20%	SID Ham	48,60%	SID Ham
mark1	23,57%	SID Ham	40,47%	SID Ham
mc	5,14%	SID Ham	29,62%	SID Ham
modulo12	3,28%	SID Ham	-4,84%	TMR Tot
nucpwr	27,68%	SID Ham	43,23%	SID Ham
opus	21,07%	SID Ham	35,81%	SID Ham
pair	-3,10%	TMR Seq	2,48%	SID Ham
planet	28,91%	SID Ham	43,68%	SID Ham
planet1	21,89%	SID Ham	46,90%	SID Ham
pma	17,73%	SID Ham	36,26%	SID Ham
protocole	20,21%	SID Ham	31,98%	SID Ham
ram_test	27,09%	SID Ham	38,10%	SID Ham
rie	28,25%	SID Ham	46,29%	SID Ham
s1	21,42%	SID Ham	36,52%	SID Ham
s1488	23,63%	SID Ham	46,11%	SID Ham
s1494	20,45%	SID Ham	51,99%	SID Ham
s1a	15,87%	SID Ham	13,39%	SID Ham
s208	14,47%	SID Ham	26,76%	SID Ham
s27	-1,05%	TMR Seq	-17,78%	TMR Tot
s386	21,29%	SID Ham	38,34%	SID Ham
s420	21,69%	SID Ham	31,46%	SID Ham
s510	41,62%	SID Ham	40,59%	SID Ham
s8	10,55%	SID Ham	1,44%	SID Ham
s820	31,81%	SID Ham	34,64%	SID Ham
s832	25,98%	SID Ham	34,80%	SID Ham
sand	19,33%	SID Ham	40,41%	SID Ham
scf	37,78%	SID Ham	48,65%	SID Ham
shiftreg	-6,06%	TMR Seq	-5,16%	TMR Tot
sr8a_master	26,62%	SID Ham	31,00%	SID Ham
sr8a_slave	27,73%	SID Ham	42,24%	SID Ham
sse	22,54%	SID Ham	37,69%	SID Ham
styr	17,67%	SID Ham	38,25%	SID Ham
sync	36,30%	SID Ham	31,73%	SID Ham
tav	0,61%	SID Ham	39,00%	SID Ham
tbk	-1,14%	TMR Seq	-3,55%	TMR Tot
tma	20,03%	SID Ham	32,03%	SID Ham
train11	25,64%	SID Ham	31,08%	SID Ham
train4	5,73%	SID Ham	-3,16%	TMR Tot
vtiether	28,88%	SID Ham	27,71%	SID Ham
vtiidec	6,36%	SID Ham	62,26%	SID Ham
vtiuar	19,74%	SID Ham	35,97%	SID Ham
zeegers	25,76%	SID Ham	48,73%	SID Ham

Max :	41,62%	62,26%
Moyenne :	17,39%	31,97%

Figure Ax.IV.3B - Gain en surface obtenu avec l'implantation SID Hamming par rapport au TMR Seq et Tot.

ANNEXE IV.4 - RÉSULTATS EN VITESSE POUR LE MASQUAGE

La figure Ax.IV.4A montre les chemins critiques des implantations SID (flot de synthèse de Hamming), TMR Seq et TMR Tot. Ces résultats sont fournis en nanosecondes.

Nom	Chemin Critique (nanosecondes)		
	SID Ham	TMR Seq	TMR Tot
bbara	19,4	10,9	6,5
bbsse	23,5	17,1	10,6
bbtas	14,4	7,9	6,2
beecount	17	6,9	6,8
cpt100	48,3	38,7	13
cpt257	60,4	44,2	18,5
cpt8	23,7	15,2	10,8
cse	29,7	20,4	14,5
dk14	17,1	12,8	9,8
dk15	15,2	8,9	10,4
dk16	32,5	24,3	10,3
dk17	17,8	12,2	6,5
dk27	15,2	7,9	5,1
dk512	20,8	14,5	10,7
dmaread3	32,4	24,5	22,5
dmawr	36,2	30,1	25,4
donfile	29,5	17,5	7
dstate	25,1	18	9,3
dvram	32,6	21,1	10,9
esd	39,1	23,4	15
essai1	18,1	12,2	6,3
essai2	25,5	16,2	8,6
essai3	13,6	11,1	10,9
essai4	35,9	29,7	23,1
ex1	30,1	17,2	14,4
ex2	27,2	19,1	12,7
ex3	21,7	12,8	8,2
ex4	18,4	14,4	8,3
ex5	20,8	14	7,7
ex6	20,8	12,2	12
ex7	20	15,4	7
ex_moore	14,8	9,5	4,9
exem1	22,6	14,2	9,2
exem2	25,1	17,7	8,9
fetch	24,6	17,4	9,4
fsm	10,9	5,3	3
gestionfifo	16,4	7,9	7,3
hsurf	76,5	59,1	55,1
hsurf32	64,4	43,5	39,5
hsurf32mod	58,9	47,1	43,1
hyeti1	94	63,3	59,3
hyeti2	64,8	39,6	35,6
imec1	53,5	37,2	33,2
imec10	67,9	47	43
jay	37,9	28,2	14
keyb	37,4	29,6	15,1
kirkman	22,7	12,6	12,4
lion	12,8	6,7	5,9
lion9	17	10,3	5,1
log	29,1	16,9	9,2
mark1	21,1	13,3	7,1
mc	13,1	6,8	5,7
modulo12	18,6	9,8	4,2
nucpwr	31,6	20,9	10,6
opus	18,9	14	10
pair	17,4	8	8
planet	43	30,5	17,2
planet1	43	30,2	17,1
pma	29,6	17,2	10,2
protocole	25,2	14,1	8,8
ram_test	55,2	32,6	20
rie	34,2	22,6	11,1
s1	38,6	25,3	12,2
s1488	42,3	25,3	18,3
s1494	41,9	22,9	16,6
s1a	37,6	23,9	13,1
s208	23,4	13,7	9,8
s27	14,5	6,4	6,2
s386	23,3	16,7	9,7
s420	22,9	12,8	11,5
s510	41,5	26,8	14,6
s8	15,7	9,1	4,1
s820	38,6	28	14,7
s832	35	21,9	10,9
sand	37,8	29	19,1
scf	59,1	48,5	27,3
shiftreg	10,4	1,3	3
sr8a_master	18,6	13,2	8,7
sr8a_slave	27,4	20,7	12,1
sse	23,5	17	10,6
styr	42,3	21,8	14,8
sync	41	26,9	17
tav	11,8	5,7	7,7
tbk	43,3	28,4	13,6
tma	25,5	16	11,1
train11	16,9	10,8	5,6
train4	12	5	3,8
vtiether	23,8	16,2	8
vtiidec	21,4	14,7	15,2
vtiuar	32,8	22,4	13,7
zeegers	65,2	34,7	32,8

Figure IV.4A - Résultats en terme de chemin critique pour les architectures SID, TMR Seq et TMR Tot.

ANNEXE IV.5 - EXTRAITS DE L'ARTICLE [ROCH 96A]

L'annexe IV.5 fournit des extraits de l'article [Roch 96a]. Plus particulièrement, elle montre les résultats obtenus lors de l'évaluation de l'outil ASYL-SdF effectuée par Thomson-CSF.

IV.2. Thomson-CSF Evaluation

For this evaluation, twenty industrial examples have been implemented using ASYL-SdF. This set of FSMs covers a wide range of complexity characteristics: between 7 and 409 states. Ten examples have less than 100 states, and ten have more than 100 states. These examples have been generated using a Thomson's standard cell library and with the area-oriented logic synthesis optimizations in ASYL. The placement and routing was performed with the tool Autocells from MENTOR. Comparisons are made with the Simplex implementation synthesized with ASYL, using the optimized compact state assignment [6]. The mission times have been obtained with a Markov model, using the CNET procedure to evaluate the expected failure rates [9]. Fault simulations have been performed with the ZICAD tool.

As in the INPG/CSI evaluation, the SID architecture generally leads to lower area penalties than the TMR approach. The average gain is 21.5%, with gains up to 38% [3]. More precisely, the SID implementation is smaller than the TMR Seq solution for 95% of the examples. The gain is over 10% for 90% of the examples, and over 20% for more than 65% of them. The figure 4 shows the area overheads of the TMR and SID implementations compared to the Simplex implementation for the twenty examples. The Simplex implementation area after placement and routing is given in Table II. In the Table II, the examples are ordered with respect of the number of states (TCS1 is the first example with more than 100 states).

Table II - Simplex Area (mm^2) and MT (10^3 hours) results, TMR Seq & SID Mission Times

Benchmark	Simplex		TMR	SID
	Area	MT	Seq	
TCS12	23 000	16 206	63 948	35 040
TCS11	95 000	4 529	9 636	6 018
TCS18	116 000	3 539	5 781	4 450
TCS10	115 000	3 644	21 900	13 140
TCS21	156 000	2 855	7 603	5 571
TCS19	438 000	1 112	2 470	2 094
TCS24	293 000	1 734	8 103	6 044
TCS17	289 000	1 682	6 202	4 862
TCS5	780 000	718	1 252	1 060
TCS6	1 452 000	451	587	587

Benchmark	Simplex		TMR	SID
	Area	MT	Seq	
TCS1	1 058 000	552	981	911
TCS23	717 000	736	2 934	2 646
TCS13	3 562 000	210	315	302
TCS15	7 783 000	136	195	194
TCS16	5 856 000	166	228	228
TCS22	4 451 000	201	280	280
TCS8	3 101 000	254	411	403
TCS20	365 000	1 296	17 520	14 016
TCS3	11 725 000	99	140	138
TCS14	24 477 000	57	83	83

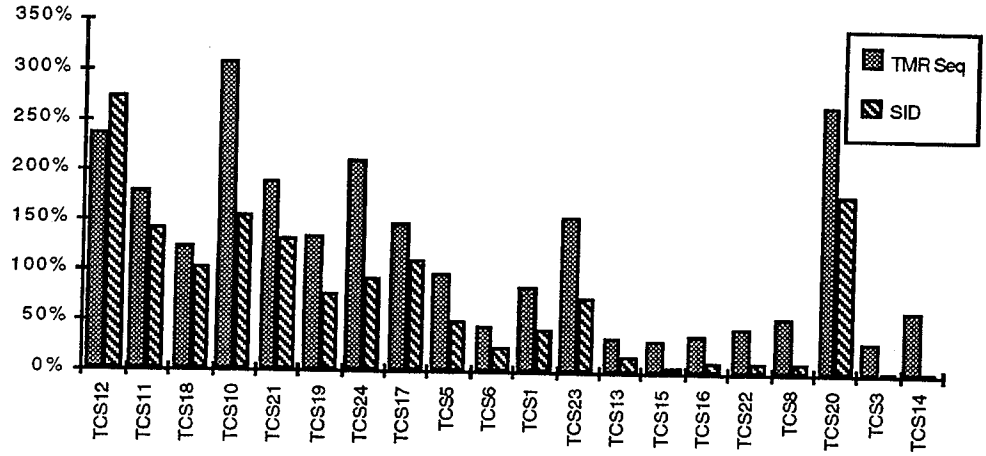

Figure 4 - Area overheads (compared to Simplex implementation).

From the critical path point of view, as in the INPG/CSI evaluation, the TMR gives the best results. But, all the examples, when synthesized with the SID architecture, can work with a 20Mhz clock, 80% of them can work with a 30Mhz clock, and 55% of them can work with a clock frequency of over 40Mhz [3].

Table II shows the area and the mission time results of the Simplex implementation, and the mission time results of the TMR Seq and SID implementations. These mission times were computed for a reliability equal to 0.999. When a very high reliability level is considered, the TMR architecture generally leads to the best result, but the SID architecture remains a good trade-off between area and reliability, with mission times noticeably greater than the mission times of the Simplex implementation. In fact, on average, the MTs obtained with the SID architecture are about 140% greater than thus obtained with the Simplex implementation and only 15% lower than thus obtained with the TMR Seq architecture.

Finally, Thomson-CSF studied the testability of each architecture. The test vector pattern generation was performed using the tool FastScan, after implementation of scan paths with the tool DFTadvisor. The figure 5a shows the difference of fault coverage rate (Simplex minus fault tolerant architecture). The figure 5b shows the increase of the number of test vector patterns due to the fault tolerant architecture compared to the Simplex implementation. These results show that the SID architecture gives a better fault coverage rate compared to the TMR architecture, but at the expense of a larger number of test vectors. Let us however notice that the specific test devices which can be generated by ASYL-SdF [2] have not been evaluated yet.

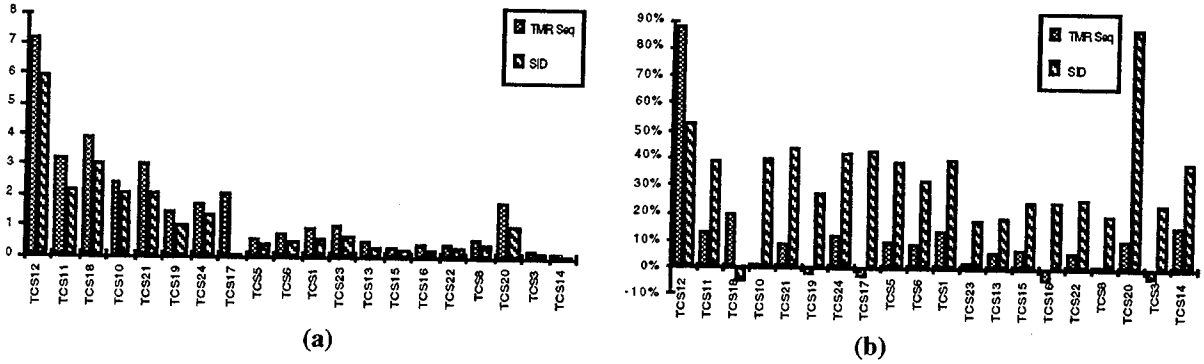

Figure 5 - Fault coverage rate differences (Simplex minus fault tolerant implementation) (a), Test vector number increase (compared to Simplex implementation) (b).

V. Conclusion

Both evaluations of the SID architecture, automatically synthesized by the ASYL-SdF tool, led to similar conclusions. Thus, it was shown that, even when different technologies and different place and route tools are used, the SID architecture leads to smaller areas than the TMR Seq implementation. In the same way, whatever the procedure chosen to evaluate the failure rates (CNET or MIL-HDBK-217F), the SID architecture gives a good trade-off between area and reliability. Both evaluations also showed that the TMR implementation is more suitable when high performances are required. However the SID implementations can meet the speed constraints of a lot of applications. Finally, the Thomson-CSF study showed that the TMR and the SID implementations give similar results from the testability point of view, with a little advantage for the SID architecture when only the fault coverage rate is considered (results obtained without the specific test devices which can be generated by ASYL-SdF).

On one hand, these results showed that, due to its small performance penalties and its high reliability level, the TMR implementation is better suited for the space, nuclear or military applications. On the other hand, the good testability and the small area for a good reliability level of the SID architecture, make it a good solution for the automotive and the industrial control process applications. Moreover, its synthesis flow is already automated, so the SID architecture meets very well the design cost and reliability constraints of this type of civil applications.

Acknowledgements

This work has been partially supported by the European JESSI AC8 project.

References

[1] H. Belhadj, L. Gerbaux, M. Crastes, "Innovative Synthesis Tools for Control Intensive Systems", User Forum, ED&TC 1995, pp. 143-146

[2] R. Rochet, R. Leveugle, G. Saucier, "Alternative implementations of fault-tolerant controllers", CSI Research Report, Highly Dependable Architectures, October 1994, CSI-SdF-94.2

[3] Thomson-CSF, "Rapport sur la synthèse d'automates tolérants aux fautes", référence R.TCS-SP6-T6.2-Q4, September 1995

[4] D. B. Armstrong, "A general method of applying error correction to synchronous digital systems", The Bell System Technical Journal, vol. 40, no. 2, March 1961, pp. 577-593

[5] R. Rochet, R. Leveugle, G. Saucier, "Analysis and comparison of fault-tolerant FSM architectures based on SEC codes", "The IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems, Venice, Italy, October 27-29, 1993", F. Lombardi, M. Sami, Y. Savaria, R. Stefanelli editors, IEEE Computer Society Press, Los Alamitos, California, 1993, pp. 9-16

[6] G. Saucier, C. Duff, F. Poirot, "State assignment using a new embedding method based on an intersecting cube theory", 26th DAC, 1989, pp. 321-326

[7] United States Department of Defense, Military Standardization Handbook: Reliability Prediction of Electronic Equipment, MIL-HDBK-217F, 1991.

[8] C. Beounes et al., "SURF-2: a program for dependability evaluation of complex hardware and software systems", 23rd FTCS, 1993, pp. 668-673

[9] CNET, France Telecom, "Recueil de données de fiabilité des composants électroniques", June 1993

AUTORISATION DE SOUTENANCE

Vu les dispositions de l'arrêté du 30 Mars 1992 relatif aux Etudes Doctorales

Vu les Rapports de présentation de :

Monsieur Jean-Dominique DECOTIGNIE

Monsieur Christian LANDRAULT

Monsieur Raphaël ROCHET

est autorisé à présenter une thèse en soutenance en vue de l'obtention du diplôme de **Docteur de l'Institut National Polytechnique de Grenoble,** spécialité "INFORMATIQUE".

Fait à Grenoble, le 4 septembre 1996.

Pierre GENTIL
Professeur INPG
Directeur du Collège Doctoral

INPG - 46, avenue Félix Viallet - 38031 GRENOBLE CEDEX 1 - FRANCE - Tél: (33) 76 57 45 00 - Télex 320 205 F - Télécopie (33) 76 57 45 01 - 36 16 code INPG