



**HAL**  
open science

# Validation de Spécifications de Circuits Asynchrones : Méthodes et outils

M. Boubekur

► **To cite this version:**

M. Boubekur. Validation de Spécifications de Circuits Asynchrones : Méthodes et outils. Autre [cs.OH]. Université Joseph-Fourier - Grenoble I, 2004. Français. NNT : . tel-00007774

**HAL Id: tel-00007774**

**<https://theses.hal.science/tel-00007774>**

Submitted on 16 Dec 2004

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITE JOSEPH FOURIER GRENOBLE 1

# THESE

Pour obtenir le grade de

**DOCTEUR DE L'UNIVERSITE JOSEPH FOURIER**

Discipline : Informatique

ÉCOLE DOCTORALE MATHÉMATIQUES, SCIENCES ET TECHNOLOGIES DE  
L'INFORMATION, INFORMATIQUE

**Présentée et soutenue publiquement  
par**

**Menouer BOUBEKEUR**

Le 22 Octobre 2004

---

## **Validation de Spécifications de Circuits Asynchrones : Méthodes et Outils**

---

### **JURY**

<i>Rapporteurs</i>	Laurence PIERRE	Professeure, Université de Nice
	Bruno ROUZEYRE	Professeur, Université de Montpellier
<i>Examineurs</i>	Eric MARTIN	Professeur, Université de Bretagne Sud
	Laurent MOUNIER	Maître de Conférences, Université Joseph Fourier
	Marc RENAUDIN	Professeur, INP Grenoble
	Pascal URARD	STMicroelectronics, invité
<i>Directeur de thèse</i>	Dominique BORRIONE	Professeure, Université Joseph Fourier

**Thèse préparée au Laboratoire TIMA "Techniques de l'Informatique et de la  
Microélectronique pour l'Architecture des Ordinateurs"**



*A celle qui m 'a soutenue ces deux dernières longues années*  
*Celle qui est restée toujours avec moi en étant loin*  
*A ma femme*

*Ils sont loin, de l 'autre côté de la Méditerranée*  
*Ils m 'ont éduqué et contribué à ma réussite*  
*Ils sont fiers de moi, comme je suis fière d 'eux*  
*A mes parents*

*A mes frères et sœurs*



# Remerciements

Je tiens tout d'abord à remercier :

Madame Dominique BORIOME, qui a assuré la direction de cette thèse, pour m'avoir accueilli au sein de son équipe, pour son amitié et son soutien constant qu'elle m'a offert pendant toute la durée de cette thèse, et dont j'ai pu mesurer la valeur en de multiples occasions. Je lui suis particulièrement reconnaissant pour les motivations et les doses d'encouragement qu'elle a toujours su me donner.

Monsieur Eric MARTIN pour m'avoir fait l'honneur de présider le jury de cette thèse.

Madame Laurence PIERRE et Monsieur Bruno ROUSSEUR pour avoir examiné soigneusement ce travail et pour leurs précieux commentaires et suggestions qui ont contribué à enrichir ce document.

Monsieur Marc RENAUDIN, qui a suivi de près ce travail, pour son aide, sa disponibilité et ses encouragements.

Monsieur Laurent MOUNIER, pour son aide, sa sympathie et les nombreux éclaircissements qu'il m'a apportés.

Monsieur Pascal URARD, de ST Microelectronics, pour s'être intéressé à ce travail et faire partie du jury.

Je remercie également Saddek BENSALAM, avec qui j'ai fait mes premiers pas dans le domaine de la recherche, pour l'aide et le soutien qu'il m'a apportés durant mes premiers temps en France.

J'exprime mes remerciements à Bénédicte FLUXA, Claude LE FAOU, Eric GASCARD et Pierre OSTIER pour leur amitié et pour l'aide qu'ils m'ont apportée durant ma thèse.

Je souhaite aussi adresser mes amitiés à mes ami(e)s et collègues membres du laboratoire TMA et en particulier ceux de l'équipe VDS avec qui j'ai passé des moments inoubliables. Je ne peux pas les nommer tous, cependant je cite : Amel, Amer, Amine, Cyrille, Diana, Emil, Ghiath, Julien, Kamel, Karim, Nacer, Nadir, Philippe.

Ces années de thèses m'ont permis de vivre des moments extrêmement enrichissants sur le plan humain et culturel. C'est avec joie que j'exprime un grand merci à tous mes amis des associations grenobloises avec qui j'ai chaleureusement œuvré.

Enfin, je ne remercierai jamais assez mes parents, ma femme, ma famille, mes ami(e)s et mes proches pour leur soutien moral et affectif permanent.



*Le doute est la clef de toute connaissance.*

*Proverbe arabe*



## Résumé

La conception asynchrone vise à répondre aux problèmes de plus en plus complexes rencontrés par les concepteurs de circuits synchrones. Les circuits asynchrones, contrairement aux circuits synchrones, ne sont pas commandés par une horloge globale. Même de taille moyenne, ils peuvent montrer un comportement complexe, dû à l'explosion combinatoire dans la chronologie des événements qui peuvent se produire. Il est ainsi essentiel d'appliquer des méthodes rigoureuses de conception et de validation.

Ce travail de thèse traite de l'analyse et de la validation automatique des spécifications de circuits asynchrones écrites en CHP, avant leur synthèse avec le flot de conception asynchrone TAST, développé par le groupe CIS de TIMA. Deux approches sont proposées. La première consiste à adapter la vérification symbolique de modèles, initialement dédiée aux circuits synchrones, pour la vérification des circuits asynchrones. Les spécifications de circuits sont alors traduites dans un modèle en VHDL *peuso-synchrone* et ensuite vérifiées par des outils industriels de vérification symbolique de modèles.

Dans la deuxième approche, la sémantique de CHP, initialement donnée en termes de réseaux de Pétri, est reformulée en termes de Systèmes de Transitions Etiquetées Etendus (STEE). Les spécifications de circuits sont alors validées par des méthodes énumératives de vérification de modèles. Pour augmenter les performances de l'approche énumérative et faire face au problème d'explosion d'états, nous avons développé et implémenté un certain nombre de techniques automatiques de réduction et d'abstraction.

**Mots-clés :** *Vérification Formelle, Validation de Spécifications, Circuits Asynchrones, Processus Séquentiels Communicants, CHP, Modélisation Pseudo-synchrone, Vérification Symbolique de Modèles, Vérification énumérative de Modèles, Outils Symboliques de Décision.*

---

**Title:** Validation of Asynchronous Circuits Specifications: Methods and Tools.

---

## Abstract

Asynchronous designs aim at answering the increasingly complex problems (clock distribution, energy, modularity) encountered by the synchronous circuits designers. Asynchronous circuits, contrary to the synchronous circuits, are not ordered by a global clock. Even medium size asynchronous circuits may display a complex behavior, due to the combinational explosion in the chronology of events that may happen. It is thus essential to apply rigorous design and validation methods.

This thesis work addresses the analysis and the automatic validation of asynchronous specifications written in the CHP, prior to their synthesis with the TAST asynchronous design flow developed by the CIS group of TIMA. Two approaches are proposed. In the first approach we use symbolic model checking and pseudo-synchronous modeling, to perform property checking on RTL designs. The approach consisted in translating the Petri Net, interpreted as a finite state machine, as a pseudo-synchronous VHDL description, which can then be input to industrial symbolic model checking software.

In the second approach, CHP semantics, initially given in terms of Petri Nets, are reformulated as Extended Labeled Transition Systems (ELTS). Circuit specifications are then validated using enumerative model checking tools. To increase the performances of the enumerative approach and avoid the state explosion problem, we have developed and implemented several automatic reduction and abstraction techniques.

**Keywords :** *Formal Verification, Validation of Specifications, Asynchronous Circuits, Communicating Sequential Processes, CHP, Pseudo-synchronous Modeling, Symbolic Model Checking, Enumerative Model Checking, Symbolic Decision Tools.*



# Table des matières

<b>1. INTRODUCTION ET ÉTAT DE L'ART SUR LA SPÉCIFICATION ET LA VÉRIFICATION DES CIRCUITS ASYNCHRONES .....</b>	<b>1</b>
1.1. INTRODUCTION : LES CIRCUITS ASYNCHRONES .....	2
1.1.1. <i>Circuits asynchrones : avantages et contraintes</i> .....	2
1.1.2. <i>Principes de base des circuits asynchrones</i> .....	3
1.1.2.1. Mode de fonctionnement .....	3
1.1.2.2. Communication et codage des données .....	3
1.1.3. <i>Classification des circuits asynchrones</i> .....	6
1.1.3.1. Les circuits insensibles aux délais .....	7
1.1.3.2. Les circuits quasi insensibles aux délais .....	7
1.1.3.3. Les circuits indépendants de la vitesse .....	8
1.1.3.4. Circuits Micropipelines .....	8
1.1.3.5. Circuits de Huffman .....	9
1.2. SPÉCIFICATION DES CIRCUITS ASYNCHRONES .....	10
1.2.1. <i>Spécification basée sur les graphes</i> .....	10
1.2.1.1. Le graphe de transitions de signaux "STG" .....	11
1.2.1.2. Les espaces de processus "Process spaces" .....	12
1.2.1.3. Machines à états asynchrones .....	13
1.2.2. <i>Spécification basée sur les langages de haut niveau</i> .....	14
1.2.2.1. Les langages Tangram et Balsa .....	15
1.2.2.2. Le langage CHP .....	16
1.2.2.3. Le langage VHDL dans la méthode NCL .....	17
1.3. VÉRIFICATION DE CIRCUITS ASYNCHRONES .....	19
1.3.1. <i>Vérification formelle de circuits asynchrones temporisés</i> .....	19
1.3.2. <i>Vérification formelle de circuits insensibles aux délais</i> .....	20
1.3.2.1. Vérification formelle à bas niveau .....	20
1.3.2.2. Vérification formelle au niveau langage .....	21
1.3.3. <i>Discussion</i> .....	22
1.4. OBJECTIF ET PLAN DU MANUSCRIT .....	23
<b>2. SÉMANTIQUES POUR LE LANGAGE CHP .....</b>	<b>25</b>
2.1. INTRODUCTION .....	26
2.2. LA SÉMANTIQUE DU LANGAGE CHP EN RÉSEAUX DE PETRI .....	27
2.2.1. <i>Réseau de Petri</i> .....	27
2.2.2. <i>Combinaison de réseaux de Petri et des graphes de flot de données</i> .....	28
2.2.3. <i>Construction du réseau PN-DFG d'une spécification CHP</i> .....	29
2.2.3.1. Exemple type d'un arbitre asynchrone .....	29
2.3. SÉMANTIQUE DU LANGAGE CHP EN SYSTÈMES DE TRANSITIONS ÉTIQUETÉES ÉTENDUS .....	30
2.3.1. <i>Systèmes de transitions étiquetées étendus</i> .....	30
2.3.1.1. Sémantique opérationnelle d'un système de transitions étiquetées étendu représenté en IF .....	31
2.3.2. <i>Construction du STEE correspondant à une spécification CHP</i> .....	33
2.3.2.1. Structure du programme CHP .....	33
2.3.2.2. Contrôle et structures .....	36
2.3.2.3. Types et variables .....	40
2.3.2.4. Communication .....	42
<b>3. APPROCHES DE VÉRIFICATION .....</b>	<b>47</b>
3.1. INTRODUCTION .....	48
3.2. PREMIÈRE APPROCHE : MODÉLISATION PSEUDO-SYNCHRONE ET VÉRIFICATION SYMBOLIQUE .....	49
3.2.1. <i>Description de la méthode</i> .....	50
3.2.2. <i>Techniques de vérification symbolique de modèles</i> .....	50
3.2.2.1. Modèle de machine d'états finis .....	51
3.2.2.2. Représentation symbolique d'une machine d'états finis .....	51
3.2.2.3. Application à la vérification .....	52
3.2.2.4. Expression de propriétés .....	52
3.2.3. <i>Traduction du Petri net vers VHDL</i> .....	53

3.2.3.1.	Représentation d'un réseau de Petri dans le flot TAST .....	53
3.2.3.2.	Sous-VHDL pour la vérification formelle .....	54
3.2.3.3.	Passage du réseau de Petri vers un modèle vérifiable en VHDL.....	55
3.2.3.4.	Algorithme de traduction.....	57
3.2.3.5.	Traduction des communications .....	59
3.2.3.6.	Exemple : Traduction de l'arbitre asynchrone .....	59
3.2.4.	<i>Vérification de propriétés</i> .....	60
3.2.4.1.	Modélisation de l'environnement .....	60
3.2.4.2.	Ecriture de vérification de propriétés.....	63
3.2.5.	<i>Vérification du circuit asynchrone après synthèse</i> .....	64
3.2.5.1.	Application à l'exemple type d'arbitre asynchrone .....	64
3.3.	<b>DEUXIÈME APPROCHE : MODÉLISATION PUREMENT ASYNCHRONE ET VÉRIFICATION ÉNUMÉRATIVE ...</b>	<b>66</b>
3.3.1.	<i>Description de la méthode</i> .....	66
3.3.2.	<i>Techniques de vérification énumérative de modèles</i> .....	67
3.3.2.1.	Expression de spécifications attendues.....	69
3.3.2.2.	Le langage temporel $\mu$ -calcul.....	69
3.3.2.3.	Sémantique des actions parallèles.....	70
3.3.3.	<i>Traduction des spécifications CHP vers des STEE</i> .....	71
3.3.4.	<i>Vérification de propriétés</i> .....	73
3.3.4.1.	Modélisation de l'environnement .....	73
3.3.4.2.	Écriture et vérification de propriétés temporelles .....	74
3.3.5.	<i>Vérification par réduction</i> .....	75
3.3.5.1.	Relations de bisimulation .....	75
3.3.5.2.	Application à la vérification.....	75
3.3.6.	<i>Vérification de propriétés après expansion de la communication</i> .....	77
3.3.6.1.	Modélisation de l'environnement .....	77
3.3.6.2.	Écriture et vérification de propriétés.....	78
3.3.7.	<i>Discussion</i> .....	79
<b>4.</b>	<b>ÉVALUATION DE LA PERFORMANCE DES DEUX MÉTHODES DE VÉRIFICATION .....</b>	<b>81</b>
4.1.	INTRODUCTION .....	82
4.2.	LA GÉNÉRATION DE JEUX DE TEST.....	83
4.2.1.	<i>Génération automatique d'exemples d'arbitre en IF</i> .....	83
4.2.2.	<i>Génération d'exemples d'arbitres en VHDL pseudo-synchrone</i> .....	83
4.2.3.	<i>Traitement du paramètre "nombre de concurrences"</i> .....	84
4.3.	DÉROULEMENT DES EXPÉRIMENTATIONS .....	84
4.3.1.	<i>Les propriétés exprimées pour l'évaluation</i> .....	85
4.3.2.	<i>Résultats de l'étude d'évaluation</i> .....	86
4.3.2.1.	Légendes .....	86
4.3.2.2.	Variation de la taille des données.....	86
4.3.2.3.	Variation de la taille du contrôle.....	88
4.3.2.4.	Variation de la concurrence : nombre d'écritures concurrentes .....	89
4.4.	CONCLUSIONS.....	91
<b>5.</b>	<b>ENVIRONNEMENT DE VALIDATION DE SPÉCIFICATIONS ASYNCHRONES.....</b>	<b>93</b>
5.1.	INTRODUCTION .....	94
5.2.	TECHNIQUES D'OPTIMISATION .....	95
5.2.1.	<i>Réduction des automates</i> .....	95
5.2.2.	<i>Abstraction automatique</i> .....	96
5.2.2.1.	Application à l'exemple de l'arbitre asynchrone.....	97
5.2.2.2.	Abstraction automatique paramétrable .....	97
5.2.3.	<i>Génération du parallélisme</i> .....	98
5.2.3.1.	Exemple d'entrelacement .....	98
5.2.3.2.	Cas général.....	99
5.2.3.3.	Génération de l'entrelacement en présence de l'opérateur "probe" .....	99
5.2.3.4.	Nécessité de création de processus pour modéliser le parallélisme .....	100
5.2.4.	<i>Réduction de pré-ordre</i> .....	101
5.3.	VÉRIFICATION DE L'EXCLUSION MUTUELLE ENTRE LES GARDES .....	102
5.3.1.	<i>Vérification statique par utilisation d'un outil de décision</i> .....	102
5.3.1.1.	Cas simples .....	103
5.3.1.2.	Algorithme .....	104
5.3.2.	<i>Vérification dynamique</i> .....	105
5.3.2.1.	Format intermédiaire pour la vérification dynamique.....	105

5.3.2.2.	Analyse du tableau de marquage : .....	106
5.3.2.3.	Vérification par un vérificateur énumératif de modèles.....	106
5.3.3.	<i>Exemple d'application</i> .....	107
<b>6.</b>	<b>MISE EN ŒUVRE ET ETUDES DE CAS.....</b>	<b>111</b>
6.1.	INTRODUCTION .....	112
6.2.	PRÉSENTATION DES OUTILS .....	112
6.2.1.	<i>Pnet2VHDL</i> .....	112
6.2.2.	<i>L'environnement de validation CHP2IF</i> .....	112
6.2.3.	<i>Outils utilisés</i> .....	112
6.2.3.1.	FormalCheck .....	112
6.2.3.2.	CADP .....	113
6.2.3.3.	IF.....	113
6.2.3.4.	Omega .....	114
6.3.	ETUDES DE CAS .....	114
6.3.1.	<i>Filtre à Réponse Impulsionnelle Finie ou filtre RIF</i> .....	114
6.3.1.1.	Description en CHP du Filtre RIF quatre étapes .....	114
6.3.1.2.	Modélisation du Filtre en STEE.....	115
6.3.1.3.	Mise en évidence des réductions.....	118
6.3.1.4.	Vérification de quelques propriétés .....	118
6.3.1.5.	Vérification par réduction.....	119
6.3.2.	<i>Circuit de DES</i> .....	119
6.3.2.1.	Algorithme du DES .....	120
6.3.2.2.	Description en CHP du circuit DES.....	120
6.3.2.3.	Modélisation du DES en STEE.....	121
6.3.2.4.	Vérification de quelques propriétés .....	124
6.3.2.5.	Vérification par réduction.....	124
<b>7.</b>	<b>CONCLUSION ET PERSPECTIVES .....</b>	<b>127</b>

# Liste des figures

Figure 1 : Communication de type poignée de main (requête/acquittement)	3
Figure 2 : Les diagrammes de transitions pour les codages trois et quatre états.	5
Figure 3 : Principe du protocole 2 phases	5
Figure 4 : Principe du protocole 4 phases	6
Figure 5 : Symbole, spécification et implémentation de la porte de Muller à deux entrées	6
Figure 6 : Les différentes classes de circuits asynchrones	7
Figure 7 : Structure de base pour circuits micropipelines	8
Figure 8 : Demi-buffer en STG et RdP équivalent	11
Figure 9 : Spécification en mode rafale et son implémentation à l'aide d'une porte de Muller	13
Figure 10 : Spécification d'un exemple d'affectation en Tangram	15
Figure 11 : Circuit asynchrone d'un buffer obtenu par la méthode Caltech	17
Figure 12 : Fonctionnement d'une porte à seuil ( $M \leq N$ )	18
Figure 13 : TAST : Flot de synthèse asynchrone	26
Figure 14 : Structure générale d'une spécification en CHP	27
Figure 15 : exemple du format PN-DFG	28
Figure 16 : exemple type d'arbitre asynchrone.	29
Figure 17 : Spécification en CHP du sélecteur et sa représentation en réseaux de Petri	30
Figure 18 : Une partie d'un STEE en IF	32
Figure 19 : Exemple de synchronisation	35
Figure 20 : Généralisation de la synchronisation à $n$ processus concurrents	36
Figure 21 : Opérateur de commande gardée	37
Figure 22 : Opérateur de séquentialité	37
Figure 23 : Opérateur de parallélisme.	38
Figure 24 : Opérateur de répétition	39
Figure 25 : Structure de répétition avec commandes gardées.	40
Figure 26 : Communications entre processus	42
Figure 27 : Actions de Communication	42
Figure 28 : Protocole 4 phases dans un canal de communication asynchrone	43
Figure 29 : Exemple de codage en DI	43
Figure 30 : L'expansion d'une action de lecture "E ?x"	44
Figure 31 : L'expansion d'une action d'écriture "S!x"	44
Figure 32 : Etendre une action d'écriture "A!x"	45
Figure 33 : Etendre une action de lecture "A?x"	45
Figure 34 : Traduction de l'opérateur probe (#)	46
Figure 35 : Approches de vérification de circuits asynchrones	48
Figure 36 : Approche de vérification pseudo-synchrone	49
Figure 37 : Exemple de réseau de Petri manipulé dans le flot TAST	54
Figure 38 : Algorithme de traduction de réseau de Petri en VHDL	58
Figure 39 : Traduction en VHDL d'une action d'écriture et modélisation de son environnement.	61
Figure 40 : Traduction en VHDL d'une action de lecture et modélisation de son environnement.	61
Figure 41 : Modélisation de l'environnement de l'arbitre par processus VHDL concurrents	62
Figure 42 : Le modèle VHDL pour les portes Muller-C pour la vérification formelle.	65
Figure 43 : Modélisation purement asynchrone et vérification formelle énumérative	66
Figure 44 : Environnement de vérification comportementale énumérative	67
Figure 45 : Exemple de deux systèmes concurrents "Sender    Receiver"	68
Figure 46 : Le STE modélisant toutes les exécutions possibles	68
Figure 47 : Comportement souhaité de la communication	69
Figure 48 : Entrelacement de deux actions ( $a1$ et $a2$ )	71
Figure 49 : un morceau du réseau de Petri de l'arbitre et de sa représentation en IF.	72
Figure 50 : Le réseau de Petri et le format IF de l'arbitre asynchrone	72
Figure 51 : Traduction de l'arbitre asynchrone dans le format IF	73
Figure 52 : STE : Les exécutions possibles de l'arbitre.	76
Figure 53 : STE de l'arbitre après réduction	76
Figure 54 : Modélisation de l'arbitre asynchrone après expansion des communications.	77
Figure 55 : Les processus modélisant l'environnement de l'arbitre	78

Figure 56 : Quelques macros de substitution d'étiquettes pour l'arbitre asynchrone	79
Figure 57 : Variations de la taille du contrôle	85
Figure 58 : Variations du nombre de concurrences	85
Figure 59 : Les résultats en temps d'exécution quand on fait varier la taille des données	87
Figure 60 : Les résultats en taille mémoire utilisée quand on fait varier la taille des données	88
Figure 61 : Les résultats en temps d'exécution quand on fait varier la taille du contrôle	89
Figure 62 : Les résultats en taille mémoire utilisée quand on fait varier la taille du contrôle	89
Figure 63 : Les résultats en temps d'exécution quand on fait varier le nombre de concurrences	90
Figure 64 : Les résultats en taille mémoire utilisée quand on fait varier le nombre de concurrences	90
Figure 65 : Environnement de validation de spécifications asynchrones	94
Figure 66 : Règles de réduction des STEE lors de la compilation	95
Figure 67 : STE du modèle d'exécution de l'arbitre asynchrone généré par abstraction	97
Figure 68 : Génération explicite de l'entrelacement des actions concurrentes	98
Figure 69 : Méthode de construction des entrelacements	99
Figure 70 : Génération d'entrelacement entre actions concurrentes en présence d'un probe	100
Figure 71 : Exemple d'une concurrence nécessitant la création de nouveaux processus concurrents	101
Figure 72 : Analyse statique pour la vérification de l'exclusion mutuelle.	102
Figure 73 : Structures de choix en réseau de Petri	103
Figure 74 : Algorithme de vérification de l'exclusion mutuelle entre les gardes d'une structure déterministe.	104
Figure 75 : Analyse dynamique pour la vérification de l'exclusion mutuelle.	105
Figure 76 : Spécification IF dédiée à la vérification de l'indéterminisme.	106
Figure 77 : Programme CHP et sa représentation en IF de l'exemple	107
Figure 78 : Modèle d'exécution de l'exemple pour la vérification dynamique	108
Figure 79 : Modèle d'exécution de l'exemple en contraignant l'environnement	109
Figure 80 : Filtrage d'un signal	114
Figure 81 : Schéma de l'implémentation en CHP du Filtre RIF quatre phases	115
Figure 82 : Code CHP, code IF et représentation graphique du IF du processus "accumulator_r"	115
Figure 83 : Code CHP du Processus "Multiplexer" et représentation en IF	116
Figure 84 : Le code CHP du processus fsm_s et sa représentation en IF.	117
Figure 85 : Processus fsm_r et sa représentation en IF	117
Figure 86 : Le STE réduit pour la propriété P3	119
Figure 87 : L'architecture globale du circuit DES	119
Figure 88 : Architecture en CHP du DES	121
Figure 89 : Le processus xor32_dtl et sa représentation en IF	121
Figure 90 : Le processus Mux_3L et sa représentation en IF	122
Figure 91 : Le processus loop_compteur et sa représentation en IF	122
Figure 92 : Processus Compteur en CHP	123
Figure 93 : Représentation en IF du processus Compteur	124
Figure 94 : Le STE réduit pour les propriétés P3 et P4	125



# **Chapitre 1**

---

## **1. Introduction et État de l'Art sur la Spécification et la Vérification des Circuits Asynchrones**

## 1.1. Introduction : les circuits asynchrones

Avant l'apparition des technologies submicroniques, les transistors, et non les interconnexions, déterminaient le coût et les performances. Les conceptions synchrones produisaient alors des circuits plus rapides et meilleur marché que les circuits asynchrones. Les conceptions asynchrones ont donc été longtemps négligées, en raison de leur coût un peu plus élevé, et de la difficulté de leur mise au point. Par ailleurs, la croissance persistante et exponentielle ces dernières années, de l'intégration des semi-conducteurs, a eu comme conséquence une industrie de la microélectronique où la conception asynchrone est *presque absente*. Cette absence reste non justifiée pour plusieurs raisons, mais en particulier par le fait que le rapport fondamental entre les transistors et les interconnexions est maintenant inversé : l'interconnexion est à présent lente et prend essentiellement plus d'espace que des transistors. Avec l'arrivée des systèmes complexes sur la puce impliquant l'interaction entre des parties analogiques et numériques, et la communication entre modules de circuit physiquement éloignés en utilisant de longues intercommunications, l'hypothèse d'horloge globale n'est plus réaliste.

Les circuits asynchrones offrent un potentiel intéressant dans plusieurs domaines tels que la conception des opérateurs numériques, des cartes à puce et des circuits à faible consommation de courant. Malgré ces avantages, il est malheureusement incertain que la conception asynchrone aura dans un futur proche le même succès commercial que celui de la conception synchrone. En revanche, les travaux de recherches, que ce soit dans le domaine académique ou industriel, ne cessent de grandir en qualité et de en quantité.

### 1.1.1. Circuits asynchrones : avantages et contraintes

La conception asynchrone est un terme employé pour classifier tous les circuits numériques qui ne sont pas synchrones, et donc ne sont pas commandés par un signal d'horloge périodique. Ce qui rend ces circuits potentiellement sensibles aux événements sur n'importe quel fil. Les problèmes qui en résultent sont principalement la difficulté de gérer le temps, la validité et le transfert des données. Ces problèmes sont résolus localement par le biais de protocoles spécifiques développés à cet effet.

Les avantages des circuits asynchrones sont essentiellement dûs à l'absence d'une l'horloge globale de commande. Nous présentons ci-dessous quelques-uns de ces avantages :

- ✓ Ignorer le temps de propagation rend possible la conception et l'utilisation des opérateurs dont le temps de calcul est variable. C'est évidemment plus avantageux que l'approche "pire cas" des circuits synchrones, qui échantillonne le circuit par rapport au temps de propagation le plus long.
- ✓ Le mode asynchrone se prête bien au traitement d'événements non-déterministes comme l'apparition d'un signal d'interruption. En synchrone, de tels événements peuvent faire apparaître, à cause de l'échantillonnage de ce signal, un état métastable pendant une durée indéterminée, non bornée [KCa87].
- ✓ La localité du contrôle offre une propriété de modularité aux circuits asynchrones. Cette propriété est particulièrement intéressante puisqu'elle permet la ré-utilisabilité des blocs asynchrones préexistants.
- ✓ La conception asynchrone offre des avantages en matière de consommation. En effet, le mode de fonctionnement asynchrone implique de façon naturelle et implicite une mise en veille de tout opérateur non sollicité, à tous les niveaux de granularité.

Cependant, malgré ces avantages, la conception asynchrone engendre un certain nombre de contraintes qui sont principalement liées aux phénomènes d'aléas :

- ✓ Les méthodes de conception de circuits asynchrones doivent garantir une implémentation sans aléas et assurer la correction du fonctionnement global.
- ✓ L'environnement doit se restreindre à un mode de fonctionnement fondamental (un seul changement des entrées du circuit à la fois). Les techniques éliminant les aléas pour ce mode sont bien connues et plus simples que celles pour le mode où plusieurs changements sont permis [Ung69].
- ✓ Les performances des circuits asynchrones peuvent être affaiblies en pratique lorsque l'on rajoute des éléments de retard souvent utilisés pour éviter les aléas. Cette technique garantit la correction fonctionnelle aux dépens de la performance du circuit.
- ✓ Dans la perspective de concevoir des systèmes mixtes (synchrones / asynchrones), la communication entre blocs hétérogènes nécessite l'implémentation d'interfaces spécifiques.

### 1.1.2. Principes de base des circuits asynchrones

#### 1.1.2.1. Mode de fonctionnement

Le principe de fonctionnement des circuits asynchrones est basé sur l'occurrence des événements, sans connaissance des instants des occurrences. Le fonctionnement est similaire à celui des systèmes flot de données. Il suffit de décrire l'enchaînement des événements et des opérations sous la forme d'un graphe de dépendance (réseau de Pétri par exemple). L'évolution globale du système est garantie par l'évolution conjointe et éventuellement concurrente des différents blocs qui le composent. Chaque bloc ou processus évolue avec les seules "informations" reçues des autres processus auxquels il est *connecté*.

La sémantique d'exécution est très similaire aux modèles de processus séquentiels communicants [Hoa76]. Elle est en effet événementielle et donc à l'opposé de la sémantique d'exécution des circuits synchrones qui évoluent de façon synchronisée. Dans le mode asynchrone, la correction fonctionnelle est indépendante de la durée d'exécution des éléments du circuit, on parle de circuits et systèmes insensibles aux délais.

#### 1.1.2.2. Communication et codage des données

Les circuits asynchrones sont décomposés en blocs fonctionnels communicant avec ou sans données via des canaux de communication. Pour permettre un fonctionnement correct et indépendamment du temps, toute communication doit être acquittée par le récepteur afin que l'émetteur puisse émettre à nouveau. Les communications sont dites à poignée de main ou de type requête-acquittement (Figure 1).

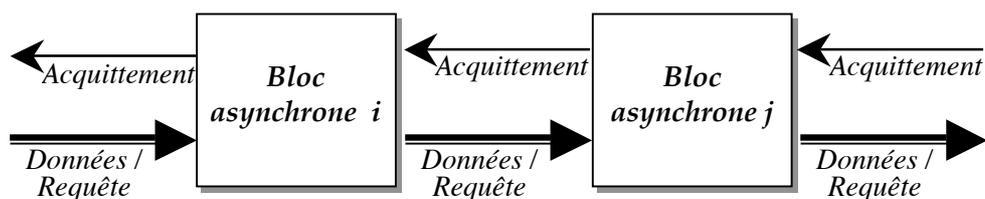


Figure 1 : Communication de type poignée de main (requête/acquittement)

### a) Canaux de communication

Un canal de communication est le moyen d'échange de données point à point entre modules asynchrones. Il se compose d'un paquet de fils : des signaux de données et des signaux de contrôle nécessaires pour accomplir la communication. Un protocole de communication est nécessaire pour gouverner la communication et maintenir un fonctionnement correct.

Dans une communication, l'émetteur (ou le récepteur) est dit *actif* s'il est l'initiateur du transfert des données. Il est dit *passif* s'il répond à une demande de transfert. Un émetteur actif initialise le transfert en indiquant que la donnée sur le canal est valide, ce qui est détecté par le récepteur grâce au codage de données adopté. Par défaut les émetteurs sont *actifs* et les récepteurs sont *passifs*.

Il est possible de définir des canaux de communication où les données sont transférées de manière bidirectionnelle [BBi96]. Ce type de canal est souvent utilisé pour accomplir l'interface avec des mémoires de type RAM ou ROM.

### b) Codage des données

Un point clé dans les circuits asynchrones est comment détecter la présence des données et indiquer la fin d'un traitement : la solution réside dans le type de codage adopté pour les données. L'utilisation d'un seul fil par bit de donnée ne permet pas de détecter un changement de valeurs de données lorsque la nouvelle donnée est identique à la précédente. À ce jour, nous distinguons principalement deux types de codage pour les données [RVG00] : codages insensibles aux délais et codage "données groupées". Ces codages associent à chaque canal une information sur la validité des données.

**Codage "données groupées" :** Dans ce codage, les données sont représentées avec le schéma traditionnel de la logique synchrone : un fil par bit de donnée. La validité des données est spécifiée séparément, un signal dit *signal de requête* est explicitement créé pour déclencher le traitement des données qui lui sont associées (Figure 3). Ce signal est implémenté avec un retard (entre le transfert de la donnée et le changement de la valeur du signal) égal ou supérieur au temps du calcul dans le "pire cas" [KMa99]. Ce codage est moins utilisé à cause de la difficulté de sa manipulation par les algorithmes de synthèse.

**Codage insensible aux délais :** Contrairement au codage "données groupées", où l'information de validité est séparée des données, le codage insensible aux délais intègre cette information de validité dans les données. Dans ce codage, les données sont détectées à l'arrivée sans aucune hypothèse temporelle. Chaque bit de donnée est codé par deux fils, quatre états sont alors disponibles pour exprimer deux valeurs logiques ("0", "1"). Cela double donc le nombre de fils par rapport aux réalisations synchrones.

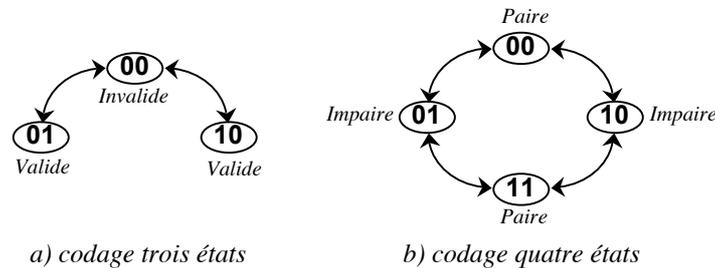
Deux codages sont communément adoptés : l'un utilisant seulement trois états et l'autre utilisant les quatre états (Figure 2).

Pour le codage trois états, un bit de donnée peut prendre les trois états suivants :

"01" signifie que le bit de donnée vaut 0, "10" que le bit de donnée vaut 1, "00" que la donnée est invalide. L'état "11" est interdit.

La convention adoptée dans le cas du codage quatre états, est de coder les valeurs 0 et 1 d'un bit avec deux combinaisons. L'une des combinaisons est considérée comme étant de parité impaire et l'autre de parité paire. Chaque fois qu'une donnée est émise, on change sa parité.

Ceci permet de passer d'une valeur logique à une autre sans passer par l'état d'invalidité. L'analyse de la parité permet de détecter la présence d'une nouvelle donnée et de générer un signal de fin de calcul. Cette méthode très élégante est cependant d'une implémentation très coûteuse [Mca92].



**Figure 2 :** Les diagrammes de transitions pour les codages trois et quatre états.

Le codage trois états est bien adapté au protocole 4 phases alors que le codage quatre états est plutôt bien adapté au protocole 2 phases, ces codages sont traités dans le paragraphe suivant.

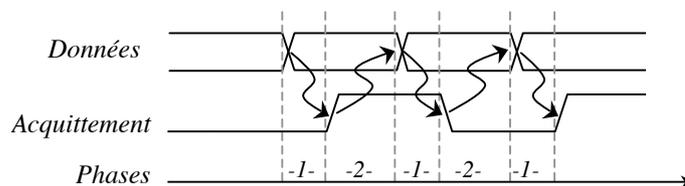
### c) Les protocoles de communication

Les deux principaux protocoles utilisés pour gérer les actions de communication sont : le protocole 2 phases (ou NRZ pour Non Retour à Zéro ou encore "Half-handshake"), et le protocole 4 phases (ou RZ pour Retour à Zéro ou encore "Full-handshake").

Le protocole deux phases fonctionne comme décrit dans la Figure 3 :

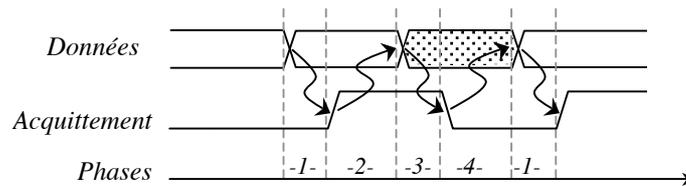
**Phase 1 :** c'est la phase active du récepteur qui détecte la présence de nouvelles données, effectue le traitement et génère le signal d'acquiescement.

**Phase 2 :** c'est la phase active de l'émetteur qui détecte le signal d'acquiescement et émet les nouvelles données si elles sont disponibles.



**Figure 3 :** Principe du protocole 2 phases

Le fonctionnement d'un protocole 4 phases est décrit dans la Figure 4, il se déroule comme suit : dans la **phase 1**, le récepteur détecte la présence de nouvelles données, effectue le traitement et génère le signal d'acquiescement. Dans la **phase 2**, l'émetteur détecte le signal d'acquiescement et émet des données invalides (retour à zéro). Dans la **phase 3** le récepteur détecte l'état d'invalidité des données, place ensuite le signal d'acquiescement dans l'état initial (retour à zéro). Enfin, dans la **phase 4** l'émetteur détecte le retour à zéro de l'acquiescement, il est alors prêt à émettre de nouvelles données.

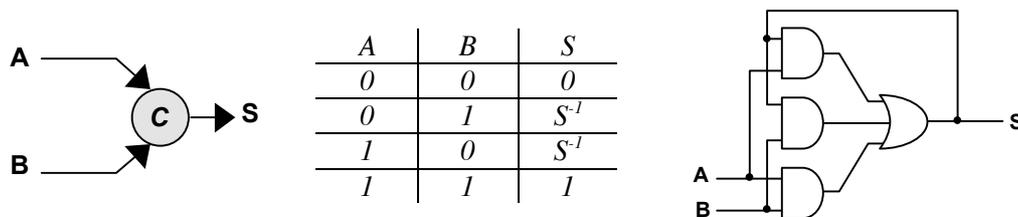


**Figure 4 :** Principe du protocole 4 phases

#### d) La porte de Muller : implémentation des protocoles de communication

Les portes de Muller ou encore les C-éléments, proposées dans [Mil65], sont les plus petits circuits qui répondent aux exigences de la logique asynchrone. Elles sont essentielles pour l'implémentation des protocoles de communication.

Une porte de Muller réalise le rendez-vous entre plusieurs signaux. La sortie copie la valeur de ses entrées lorsque celles-ci sont identiques, sinon elle mémorise l'état précédent. La Figure 5 représente le symbole d'une porte de Muller, sa spécification ainsi que sa réalisation au niveau portes.

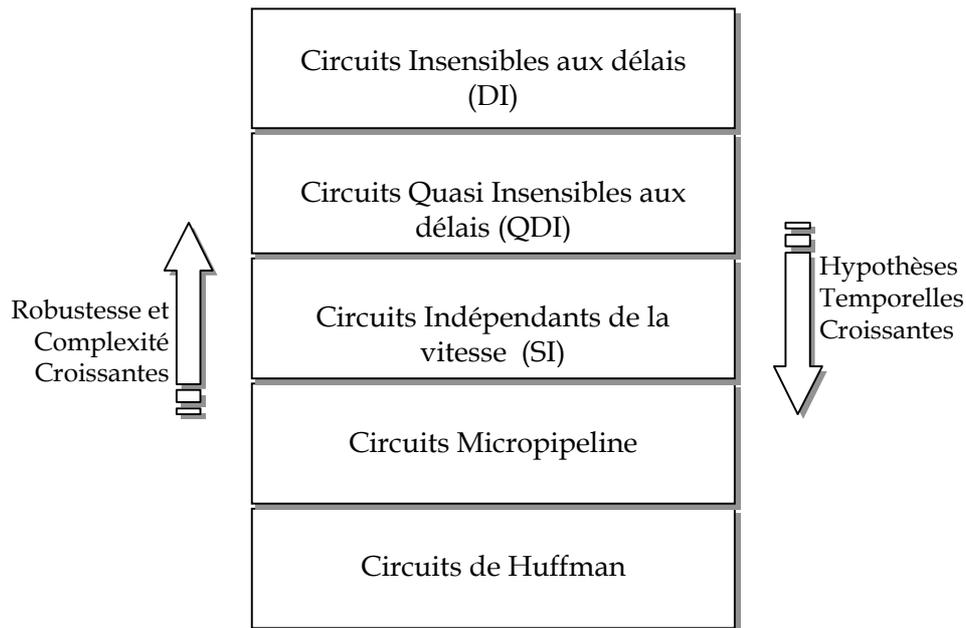


**Figure 5 :** Symbole, spécification et implémentation de la porte de Muller à deux entrées

### 1.1.3. Classification des circuits asynchrones

Une façon de distinguer les différents types de circuits asynchrones est de considérer les modèles qui régissent le comportement des délais des circuits et de l'environnement. Un délai est dit "*fixe*" lorsqu'il possède une valeur connue déterminée, il est dit "*borné*" lorsqu'il possède une valeur située dans un intervalle connu. Enfin, un délai peut être "*non borné*", et sa valeur est finie mais inconnue.

Certaines réalisations asynchrones parviennent à relâcher la contrainte de correction fonctionnelle indépendamment des délais, en introduisant des hypothèses temporelles qui conduisent à des réalisations et une conception plus simples.



**Figure 6 :** Les différentes classes de circuits asynchrones

Nous présentons brièvement certaines classes de circuits asynchrones qui se caractérisent par des hypothèses temporelles plus ou moins fortes (Figure 6). Nous commençons par la classe des circuits asynchrones dont le fonctionnement est insensible aux délais. Puis viennent les classes dérivées qui introduisent des hypothèses temporelles de plus en plus fortes, et qui se situent entre le mode de fonctionnement asynchrone pur et le mode de fonctionnement synchrone.

### 1.1.3.1. Les circuits insensibles aux délais

Cette classe de circuits est basée sur les travaux de [Cla67]. Elle utilise un mode de fonctionnement purement asynchrone. Dans cette classe, aucune hypothèse temporelle n'est introduite, les circuits sont fonctionnellement corrects indépendamment des délais introduits par les fils ou les éléments logiques composant le circuit, quelle que soit leur complexité. Cela signifie que ce type de circuits est basé sur un modèle de délai *non-borné* pour les fils et les éléments du circuit.

Ainsi, on suppose qu'un circuit répondra toujours correctement à une sollicitation externe après un temps de calcul inconnu. Ceci impose donc au récepteur d'un signal de toujours informer l'expéditeur que l'information a été reçue. Les circuits récepteurs doivent donc être capables de détecter la réception d'une donnée et/ou la fin de son traitement. Les circuits émetteurs quant à eux doivent attendre un compte-rendu avant d'émettre une nouvelle donnée.

### 1.1.3.2. Les circuits quasi insensibles aux délais

Appelés aussi "*QDI*" pour *Quasi-Delay Insensitive* en anglais, cette classe de circuits est un sur-ensemble de la classe précédente. En plus du modèle de délais "non borné" adopté pour les connexions, la notion de fourche isochrone "*isochronic fork*" [Mar90b, Ber92] est ajoutée.

Une fourche consiste en un fil connectant un expéditeur unique à deux récepteurs. Elle est qualifiée d'isochrone lorsque les délais entre l'expéditeur et les deux récepteurs sont identiques. Avec l'hypothèse d'une fourche isochrone, il est permis de ne tester qu'une seule branche de la fourche en supposant que le signal s'est propagé de la même façon dans l'autre branche. Par conséquent, on peut autoriser l'acquittement d'une seule des branches de la fourche isochrone sans avoir vérifié la propagation du signal dans l'autre branche.

L'hypothèse temporelle de "fourche isochrone" est la plus faible à ajouter aux circuits insensibles aux délais pour les rendre réalisables avec des portes à plusieurs entrées et une seule sortie. Cette affirmation a été démontrée par A. Martin [Mar93]. Il est donc possible d'implémenter les circuits quasi insensibles aux délais avec des cellules standards comme celles utilisées dans la conception de circuits synchrones.

### 1.1.3.3. Les circuits indépendants de la vitesse

Les circuits indépendants de la vitesse appelés "SI" [Mil65] pour *Speed Independence* adoptent le modèle de délais non borné pour les portes, mais font l'hypothèse que les délais dans les fils sont négligeables. Les différences entre les circuits "QDI" et "SI" sont très minimes. Aujourd'hui, il y a un consensus pour considérer que ces deux classes sont équivalentes. Hauck dans [Hau95] montre comment la propriété de fourche isochrone des circuits QDI peut être représentée par un circuit indépendant de la vitesse.

Enfin, le modèle QDI permet d'identifier les fourches isochrones, c'est-à-dire les connexions du circuit qui ne sont pas insensibles aux délais, alors que le modèle "SI" ne les identifie pas. Il considère toutes les connexions comme potentiellement "sensibles" et toutes les fourches comme isochrones. Le modèle QDI est alors plus précis, donc plus pertinent à utiliser.

### 1.1.3.4. Circuits Micropipelines

La conception de circuits asynchrones *Micropipeline* a été initialement proposée par Ivan Sutherland[Suth89], elle représente une alternative aux pipelines synchrones. Un circuit asynchrone micropipeline est composé de parties contrôles insensibles aux délais qui commandent des chemins de données conçus en utilisant le modèle de délai borné. La structure de base de cette classe de circuits est le contrôle d'une queue de type FIFO. Cette queue se compose d'éléments identiques connectés tête-bêche. Les opérateurs sont des C-éléments (§1.1.2.2.d).

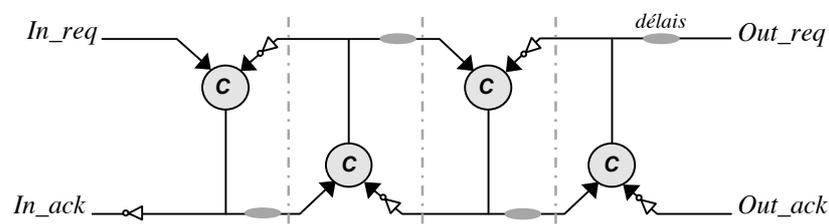


Figure 7 : Structure de base pour circuits micropipelines

Les signaux de requête sont initialement à 0 et les acquittements à 1, on note la présence d'inverseur sur les chemins d'acquittements. Une transition positive sur *In\_req* provoque une transition négative sur *In\_ack* qui se propage à l'étage suivant. Le deuxième étage produit une transition positive qui, d'une part, se propage à l'étage suivant, mais qui, d'autre part, revient au premier étage l'autorisant cette fois à traiter une transition négative. Les transitions de

signaux se propagent donc dans la structure tant qu'elles ne rencontrent pas de cellule "occupée", le fonctionnement est alors de type FIFO.

Les circuits micropipelines sont parfois qualifiés de circuits pseudo-synchrones. Au lieu d'avoir une horloge de fréquence fixe qui commande l'ensemble des *latches* du circuit, les C-éléments sont commandés par des contrôleurs locaux (partie contrôle) à des fréquences locales correspondant aux délais des chemins de données.

### 1.1.3.5. Circuits de Huffman

Dans cette classe de circuits, le modèle de délai est identique aux circuits synchrones. En effet, les délais associés à tous les éléments du circuit sont bornés. Leur conception repose sur l'analyse des délais dans tous les chemins et boucles mais aussi sur des hypothèses de délais relatives aux signaux issus de l'environnement. La conception est à base de machines à états composées de boucles combinatoires. Les boucles réalisent la mémorisation des états. Pour être correct, le circuit comporte des délais dans les boucles afin de respecter les hypothèses temporelles.

Cependant, la conception de circuits de Huffman[Huf54] présente quelques limitations. Il est supposé qu'une seule entrée peut changer à la fois, cela rend plus difficile la conception comparée à la logique synchrone où plusieurs changements sont permis. Plus récemment, un autre modèle très similaire aux circuits de Huffman appelé *mode rafale* ou encore "*burst-mode*" a été proposé [DSC93, NDi95]. Contrairement au modèle de circuits de Huffman, dans ce nouveau modèle plusieurs signaux d'entrée ou de sortie peuvent changer simultanément.

## 1.2. Spécification des circuits asynchrones

Il est nécessaire de définir des formalismes ou d'en adapter d'autres pour la description des circuits asynchrones. En effet, on ne peut se contenter des instruments de conception largement connus et utilisés pour les circuits synchrones pour deux raisons principales :

- La première est liée au problème majeur inhérent à la conception asynchrone : la façon de traiter les aléas. Ce problème nécessite d'adopter une synthèse exempte d'aléas puisque les outils de CAO standards n'adressent pas ce genre de contrainte.
- Les langages de description matérielle tels que VHDL et Verilog, utilisés dans la conception synchrone, ne sont pas totalement adaptés aux exigences de l'asynchrone. En particulier, ces langages ne sont pas suffisamment flexibles pour modéliser la concurrence et la séquentialité des blocs fonctionnels asynchrones. Par ailleurs, ils ne permettent pas de décrire les communications par canaux interposés nécessaires dans la conception asynchrone.

On peut distinguer deux orientations majeures pour les méthodologies de spécification et de représentation des circuits asynchrones. La première concerne les méthodes basées sur les langages de description de haut niveau tels que CSP, CHP, Occam, VHDL, Tangram, Balsa. La seconde regroupe les méthodes basées sur une description du comportement des circuits en graphes tels que les réseaux de Petri, les graphes d'états et les STG "*Signal Transition Graph*".

Les méthodes basées sur les langages offrent une bonne expressivité, elles sont adéquates pour décrire des systèmes complexes, hiérarchiques et modulaires. En ce sens, elles rendent plus facile la tâche du concepteur, toutefois le circuit synthétisé obtenu à partir de ces spécifications est moins optimal.

Les méthodes basées sur les graphes sont meilleures pour l'analyse temporelle et la synthèse. L'inconvénient de ce type de représentation reste à l'évidence le caractère ardu et pénible de la spécification que doit décrire le concepteur.

### 1.2.1. Spécification basée sur les graphes

Les méthodes de spécification basées sur les graphes spécifient le comportement des circuits asynchrones avec un faible niveau d'abstraction, souvent au niveau signal. Les graphes utilisés pour ces approches incluent les réseaux de Petri [Pet62], les graphes de transitions de signaux "STG" [Chu87], les diagrammes de changements [Var90], les machines à états asynchrones [YDi92, DCS93b, Huf54], et les graphes d'états [MBa59].

Les premiers modèles utilisés dans de nombreuses méthodes de spécification des circuits asynchrones utilisent les réseaux de Petri. Ce choix est motivé par le fait que les réseaux de Petri permettent la description des événements concurrents autant que séquentiels. A titre d'exemple, les réseaux M "*M-Nets*" proposés par Seitz [Sei70], les réseaux I "*I-Nets*" définis par Molnar [MFR85] et les graphes de signaux développés par Yakovlev [RYa85] sont une classe restreinte de réseaux de Petri (appelée les graphes marqués).

Tous ces modèles représentent la concurrence entre des événements, mais avec une importante restriction : ils ne peuvent pas décrire le comportement conditionnel, tel qu'un choix entre des entrées.

A l'heure actuelle nous distinguons principalement trois formalismes utilisés dans la communauté asynchrone pour spécifier les circuits : les STG qui sont une classe restreinte des réseaux de Pétri, les espaces de processus et les machines d'états asynchrones.

### 1.2.1.1. Le graphe de transitions de signaux "STG"

Le graphe de transitions de signaux (STG) est largement étudié et utilisé par la communauté asynchrone. Les STG sont basés sur l'idée que les circuits asynchrones sont entraînés par les événements. Il est donc opportun de modéliser le comportement exigé des entrées/sorties du point de vue de la causalité. Les STG sont alors des réseaux de Pétri interprétés basés sur la causalité, le franchissement d'une transition représente l'occurrence d'une montée (+) ou d'une descente (-) du signal binaire avec lequel la transition est marquée.

Un STG permet de modéliser la concurrence et adopte une sémantique limitée pour les choix sur les entrées. Plusieurs méthodologies de conception de circuits asynchrones existantes sont basées sur ce type de graphes [Chu87, CKK02, MMe92, SSL92].

#### STG dans l'outil de synthèse Petrify [CKK97]

Les STG sont utilisés dans la méthode de synthèse proposée par Cortadella [CKK02]. Ils servent à décrire des circuits de type *indépendant de la vitesse* "SI".

Les STG considérés dans l'outil Petrify sont en fait des réseaux de Petri limités par les caractéristiques suivantes :

- ✓ Liberté de choix : la sélection entre des alternatives doit être seulement contrôlée par les entrées exclusives mutuellement.
- ✓ 1-borné : il n'existe jamais plus d'un jeton dans une place
- ✓ Vivacité : STG sans blocage

Dans cette méthode, un circuit asynchrone peut être décrit par un STG si ce dernier possède les caractéristiques suivantes :

- ✓ Cohérence : les transitions d'un signal doivent strictement alterner entre la montée et la descente dans n'importe quelle exécution du STG
- ✓ Persistance : si une transition de signal est autorisée, elle doit avoir lieu, c'est-à-dire qu'elle ne peut pas être désactivée par une autre transition. La persistance des signaux internes et des signaux de sortie doit être garantie par le STG, tandis que celle des signaux d'entrée est assurée par l'environnement.

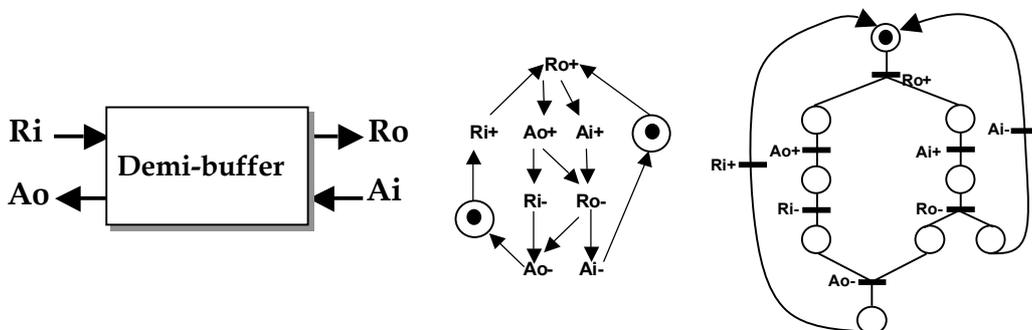


Figure 8 : Demi-buffer en STG et RdP équivalent

La Figure 8 présente la conception en STG d'un demi-buffer, et la représentation de son comportement en réseaux de Petri. Les signaux d'entrée sont Ri et Ai alors que Ao et Ro sont des sorties. La sortie Ro peut être vue comme un *latch* permettant la réception des données sur Ri. La sortie Ao est un signal d'acquiescement envoyé à l'étage précédent.

Lors de la tâche de synthèse asynchrone, la fonction de chaque sortie du circuit est déduite de l'espace d'états atteignables qui doit être produit [MBa59]. Le nombre d'états de cet espace augmente exponentiellement avec le nombre de signaux du STG. C'est le principal point faible de cette méthode qui en pratique limite le nombre de signaux à une vingtaine. En utilisant la théorie des régions [KKT94] sur cet espace d'états, les équations logiques booléennes des signaux internes et des signaux de sortie sont calculées.

Un formalisme basé sur les STG appelé *gSTG* a été proposé dans [WBe00], il permet de décrire tous les comportements possibles disponibles dans les STG, de plus il exprime d'autres comportements jugés intéressants dans l'environnement de synthèse CASCADE [WBe00] comme : la pseudo-causalité, la causalité triangulaire et la simultanéité.

Les méthodes basées sur les STG [Chu87, MBM89, MenG91, MMe92, CKK02] possèdent un certain nombre d'handicaps tels que la limitation dans la description de l'opérateur de choix entre des entrées. Une autre restriction est qu'aucun signal ne peut avoir plus d'une transition montante et d'une transition descendante dans un réseau. Ces contraintes rendent le STG peu utilisable lors de descriptions de systèmes complexes.

### 1.2.1.2. Les espaces de processus "*Process spaces*"

Les espaces de processus sont un modèle de concurrence basé sur une notion abstraite de l'exécution qui ne fait aucune référence explicite à l'ordre entre les événements [Neg98].

Un processus  $P$  sur l'ensemble  $E$  est défini par le couple  $(X, Y)$ , où  $E$  est un ensemble abstrait d'exécutions et  $(X \cup Y) = E$ .  $P$  est modélisé en deux ensembles  $X$  et  $Y$  d'exécutions abstraites, correspondant à la spécification supposé/garanti (*assumption/guarantee*), un ensemble représente ce que l'on peut garantir concernant le comportement du système à condition que les suppositions sur le comportement de l'environnement soient valides.

Pour chaque processus  $P = (X, Y)$  les ensembles  $gP = (X \cap Y)$ ,  $rP = \bar{Y}$ , et  $eP = \bar{X}$  dénotent un partitionnement de  $E$  en trois sous-ensembles :

- ✓ l'ensemble  $gP$  dénote les états valides, contenant les exécutions valides ;
- ✓ l'ensemble  $rP$  dénote les états de rejets, contenant les exécutions qui doivent être évitées par l'environnement ;
- ✓ et l'ensemble  $eP$  dénote les états d'erreurs, qui sont des exécutions qui doivent être évitées par le système.

Le système est présenté sous forme d'une machine d'états dont les états sont: e: évite (*escape*), r: rejet (*reject*) g: valide (*goal*) et les transitions représentent les actions, c'est-à-dire les événements.

Généralement, dans la conception asynchrone, les limites sur les délais des composants ne sont pas connues jusqu'après l'implémentation matérielle (disposition des composants, taille des transistors, et la longueur des fils). De ce fait, ce travail propose une technique qui

consiste à inclure dans le modèle de base des contraintes relatives aux retards (délais), qui simplement assurent l'ordre des événements.

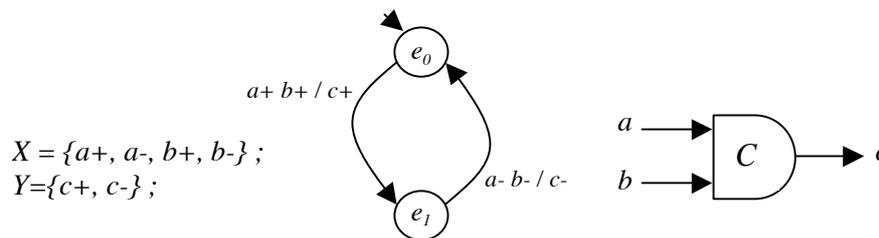
Le modèle d'espaces de processus est discuté en détail par Negulescu [Neg98] où il est utilisé comme base pour la synthèse et la vérification automatique de circuits asynchrones dans l'outil Firemaps.

### 1.2.1.3. Machines à états asynchrones

Les machines à états asynchrones de type Mealy (ASM pour *Asynchronous State Machine*) ont été proposées dans [NDi95] pour la spécification des circuits asynchrones. L'ASM est en fait une machine de Huffman dans laquelle, à chaque état, elle peut recevoir des entrées, générer des sorties et changer son état. Sa structure est alors similaire à la machine à états synchrones mais sans l'élément de stockage piloté par une horloge.

#### a) Les graphes ASM dans l'outil de synthèse asynchrone Minimalist

La méthode utilisant les ASMs, proposée par Nowick [Now93], traite la conception de contrôleurs asynchrones fonctionnant en mode rafale. Les circuits sont initialement spécifiés en une machine à états asynchrone de type Mealy. Une ASM est définie par un n-uplet  $M = \langle X, Y, S, s, F \rangle$ , tel que :  $X$  et  $Y$  sont respectivement l'ensemble des actions d'entrée et l'ensemble des actions de sortie,  $S$  est un ensemble fini d'états,  $s$  est l'état initial et  $F \subseteq S \times 2^X \times 2^Y \rightarrow S$  est la relation de transition entre les états. Du point de vue sémantique d'exécution : à chaque état, la machine peut recevoir des entrées, génère des sorties et avance jusqu'à l'état suivant.



**Figure 9 :** Spécification en mode rafale et son implémentation à l'aide d'une porte de Muller

A titre d'exemple, la Figure 9 représente un exemple d'une spécification en mode rafale. Cette spécification est plus concise que le STG, notamment quand la concurrence du système est importante.

Des contraintes sur la spécification sont définies pour que le contrôleur fonctionne correctement. D'abord, à chaque état donné, aucun changement des entrées ne peut être recouvert par d'autres changements puisque dans ce cas le comportement du contrôleur pourrait être ambigu. La deuxième restriction est que chaque état a un point d'entrée unique, ce qui simplifie la minimisation et garantit une implémentation sans aléa. Une dernière restriction sur la spécification est qu'elle ne permet pas un changement d'état sans un changement sur des entrées. Cela signifie que le système reste dans un état stable si aucune entrée ne change. Ces restrictions distinguent cette méthode de spécification de la spécification pilotée par événements, telle que celle développée par Davis [DCS93b].

Un outil de synthèse appelé Minimalist [FNT99] a été développé pour prototyper cette méthode. Minimalist prend en entrée une machine à états de type ASM, il génère et manipule

efficacement les contrôleurs asynchrones [Dea92]. La spécification en mode rafale est pratique pour décrire des systèmes dont la taille est petite ou moyenne. Cependant, il est difficile de décrire un grand système concurrent par cette méthode.

En conclusion, même si elles produisent généralement des circuits efficaces et rapides, les méthodes de synthèse basées sur les graphes exigent souvent l'exploration complète de l'espace d'états pour trouver tous les états accessibles. Par conséquent, le nombre d'états accessibles de l'espace d'états explose rapidement quand la complexité de la spécification augmente. De plus, comme les spécifications avec ces approches sont en général au niveau des signaux, l'écriture est ardue et est sujette à des erreurs surtout si la taille du circuit à concevoir est conséquente. Même si les problèmes d'affectation des états et de description des choix des entrées sont maîtrisés, le problème de la production des circuits sans aléa reste toujours un obstacle majeur à l'utilisation pratique de ces méthodes.

### 1.2.2. Spécification basée sur les langages de haut niveau

Les méthodes de spécification de circuits asynchrones basées sur les langages présentent des avantages très importants. Depuis une première spécification de haut niveau jusqu'à un niveau de description structurelle à grain fin, le circuit peut être décrit en utilisant un même langage. Ceci offre une continuité sémantique entre tous les niveaux de description, y compris les environnements de test et de vérification, et constitue donc un outil puissant pour faire de l'exploration architecturale. Ainsi, les approches langages, en cachant les aspects liés à l'implémentation, permettent d'étudier des architectures tout en programmant.

Les langages qui sont employés pour spécifier des circuits asynchrones incluent CSP [Hoa78], Occam [May90], Tangram [BKR91], Balsa [BE97] VHDL [Zhe98] et Verilog [BLa00].

Les langages de description de matériel comme VHDL et Verilog, actuellement supportés par les outils commerciaux et largement adoptés par l'industrie, fournissent un niveau d'abstraction élevé et évitent de spécifier le séquençage des transitions de signaux. Néanmoins, ils n'utilisent pas le concept de canal de communication. Il faut donc spécifier des paquetages qui permettent au concepteur de définir la communication entre processus concurrents, tel que présenté dans [RVR99].

Dans les langages dérivés du langage CSP, les processus concurrents communiquent par passage de message via des canaux. Ceci offre au concepteur la facilité de décrire des blocs fonctionnels asynchrones communiquant concurrentement et séquentiellement entre eux. Cependant, même si de nombreux langages basés sur CSP sont largement utilisés pour modéliser des circuits asynchrones, il n'existe pas aujourd'hui de réel consensus sur un unique langage de spécification dédié à leur modélisation.

Nous présentons, par la suite, les plus importants langages de spécification de circuits asynchrones utilisés à l'heure actuelle. Nous abordons brièvement contexte et outils de synthèse associés à ces langages.

Nous distinguons trois familles de langage :

- ✓ **Tangram et Balsa** deux langages de spécification utilisant une représentation intermédiaire à base de composants qui implémentent le protocole *poignée-de-main* (§1.1.2.2) ;
- ✓ **CHP** qui englobe le langage CHP proposé par Alain Martin et utilisé dans l'outil de synthèse de l'université de Caltech, et le CHP étendu utilisé dans l'outil TAST du laboratoire TIMA ;

- ✓ Et enfin le langage *VHDL*, qui a été adapté aux circuits asynchrones et utilisé dans l'outil de synthèse NCL[FBr96].

### 1.2.2.1. Les langages Tangram et Balsa

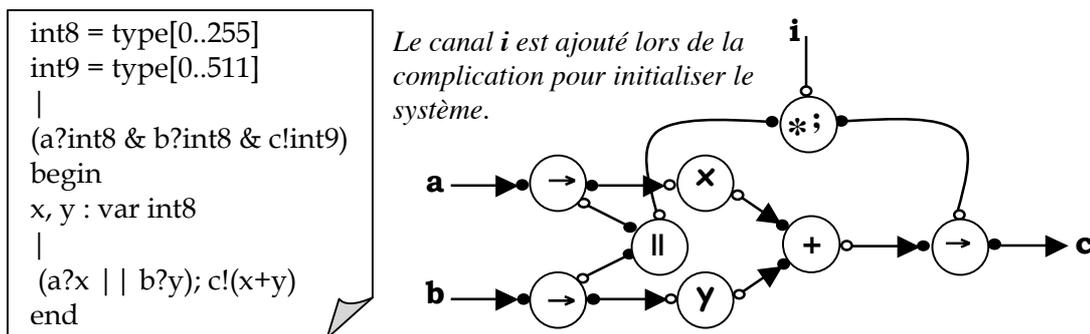
Tangram et Balsa sont des langages de description de haut niveau dédiés à la spécification de circuits asynchrones. A chaque langage est associé un compilateur qui permet de traduire les structures de spécification du circuit en des structures de composants "poignée de main".

#### a) Le langage Tangram

Tangram est la seule méthode de conception de circuits asynchrones utilisée et développée aujourd'hui par une société privée. En effet, Tangram est un système propriétaire de Philips [BKR91] comprenant un langage de description de haut niveau (Tangram) et un compilateur associé qui permet de traduire les structures de langage en des structures de composants *poignée-de-main*.

La syntaxe de *Tangram* ressemble à celle des langages de programmation traditionnels tels que C. Ce langage utilise les concepts de CSP et modélise des processus concurrents communiquant par passage de messages synchrones via des canaux de communication point à point.

L'exemple de la Figure 10 décrit une affectation  $c := a + b$ , en Tangram et sa représentation en format *poignée-de-main*.



**Figure 10 :** Spécification d'un exemple d'affectation en Tangram

Le comportement est initialisé par un signal de requête sur le canal (i) connecté à composant "\*,;" qui réalise le séquençage entre le composant de concurrence "||" et celui du transfert "→". La requête est propagée aux variables x et y, qui récupèrent de manière concurrente les valeurs sur les canaux (a) et (b). Les variables x et y affirment la réception des données, qui sont traitées par l'additionneur. Le résultat est transmis via le composant de transfert "→" sur le canal (c).

Le flot de synthèse associé à Tangram dispose d'une bibliothèque de circuits de type poignée de main que le compilateur cible. Actuellement, de nombreuses bibliothèques de ce type existent, ce qui permet des implémentations avec différentes technologies cibles (cellules standard CMOS, FPGA). Ces éléments de bibliothèque sont spécifiés et conçus manuellement à l'aide des méthodes STG ou en utilisant les étapes de la méthode de Caltech présentée ultérieurement.

Puisqu'il existe une correspondance entre les structures du langage et un circuit de type poignée de main généré, le processus de compilation reste simple et entièrement transparent pour le concepteur : un changement progressif de la spécification a pour résultat un changement prévisible de l'implémentation du circuit.

### b) Méthode Balsa

Balsa est un outil de synthèse de circuits asynchrones développé par l'université de Manchester. Cet outil emprunte la fonction de compilation de Tangram, il traite des circuits asynchrones spécifiés par le langage de haut niveau Balsa. Les structures du langage sont mises en correspondance directe avec des composants communicants de type poignée de main.

A l'instar du langage Tangram, le langage Balsa [BEd97, BEd00a] fait partie de la famille des langages de programmation concurrents. Il se base sur des communications synchronisées sur des canaux et un style de description parallèle.

Breeze est le format intermédiaire du flot de synthèse Balsa. Il intègre une bibliothèque de *netlists* qui définit le réseau de circuits "poignée de main", il renforce ainsi l'indépendance des outils finaux par rapport aux outils frontaux.

Cette méthode de conception a été illustrée par la conception du microprocesseur Amulet3i [BEd00b].

### 1.2.2.2. Le langage CHP

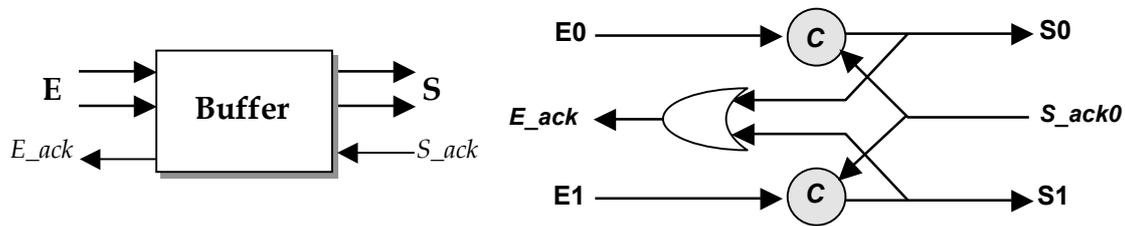
Le langage CHP a été développé pour décrire des circuits asynchrones corrects en termes d'hypothèses temporelles et de fonctionnalité. L'objectif est de préserver cette correction tout au long des transformations appliquées. La nécessité de modéliser l'ensemble des contraintes depuis le plus haut niveau à toutes les étapes du développement s'impose comme une certitude, CHP inspire à devenir un langage idéal, rigoureux et complet.

#### a) CHP dans la méthode Caltech

La méthode de Caltech est universellement reconnue dans la communauté asynchrone comme étant l'une des voies de développement asynchrone les plus probantes. Cette méthode repose sur une description de haut niveau sous forme de processus concurrents communicants. Basée sur le langage CSP [Hoa78], cette modélisation CHP [Mar90] garantit depuis le début du développement jusqu'à la fin la préservation des clauses d'insensibilité aux délais imposées par le modèle : les processus dialoguent entre eux sans jamais faire d'hypothèse concernant la propagation des signaux le long des canaux.

Le principe de la méthode se présente comme suit : lors d'une première étape de *décomposition des processus*, le but est de convertir chaque processus en processus élémentaires de faible complexité. Cela permet de faire apparaître la concurrence et d'identifier les problèmes d'arbitrage et de synchronisation. La deuxième étape est l'*expansion des communications*, elle concerne les actions de communication. Les protocoles deux phases ou quatre phases peuvent être indifféremment utilisés. Le code obtenu est composé d'instructions d'affectation de signaux et d'instructions d'attente ou de test sur des signaux. Enfin, la dernière étape est la construction des *règles de production*. A partir de ces règles, une cellule CMOS à une sortie et plusieurs entrées est construite pour chaque signal. La connexion de toutes ces cellules entre elles implémente le circuit.

Si on considère l'exemple d'un élément de mémorisation, sa fonction est simplement de copier une donnée d'un canal E sur un canal S. Sa spécification en CHP est :  $*[E?x ; S!x]$



**Figure 11 :** Circuit asynchrone d'un buffer obtenu par la méthode Caltech

La Figure 11 présente le schéma obtenu après application des différentes étapes de la méthode.

La méthode de synthèse de circuits asynchrones développée par Alain Martin à Caltech est certainement une des plus performantes à l'heure actuelle. Un microprocesseur CAP [MAR 89] et un microprocesseur MiniMIPS [MLM97] ont été réalisés en utilisant cette méthode de conception.

#### b) CHP dans le flot de synthèse TAST

L'outil de synthèse asynchrone TAST développé à TIMA est une généralisation de la synthèse de haut niveau. L'approche est multi-langage, elle permet d'accepter plusieurs langages en entrée, tels que des HDL (en particulier VHDL), le langage CHP, et à terme des langages comme system C. TAST repose sur la définition d'une forme synthétisable du langage d'entrée CHP qui permet la modélisation et la synthèse de machine à état de taille importante, l'outil permet par ailleurs de cibler différents styles de circuits. Le concepteur peut ainsi évaluer les méthodes des implémentations asynchrones, micropipeline ou QDI, et aussi synchrones à partir d'une spécification unique.

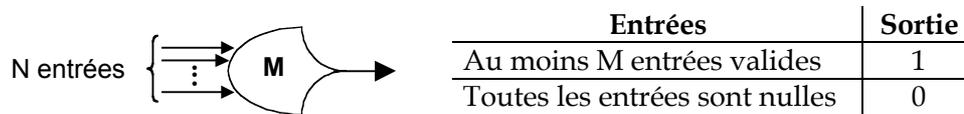
L'objectif du flot de conception TAST est d'allier l'avantage offert par une spécification en langage de haut niveau (conception hiérarchique de circuits complexes) à l'avantage d'une spécification en graphes (facilité de la synthèse). Les circuits dans TAST sont spécifiés en langage de haut niveau dit CHP étendu. Cette spécification est compilée en graphes (réseaux de Petri et graphes de flots de données). Celle-ci est utilisée par l'ensemble des outils intégrés dans TAST.

Les performances du flot de synthèse TAST ont été illustrées par la conception et la fabrication de deux microprocesseurs asynchrones : ASPRO-216 [RVR98] et MICA [ABR01]. Le langage CHP fera l'objet d'une plus grande attention dans le chapitre 2.

#### 1.2.2.3. Le langage VHDL dans la méthode NCL

Dans la méthode NCL "Null Convention Logic" [FBr96, LFS00], le langage utilisé est VHDL, en fait le langage est adapté afin de modéliser les communications. Ces aspects sont naturellement essentiels pour concevoir des circuits asynchrones. La méthode est une logique propriétaire proposée et brevetée par Theseus Logic. Elle est basée sur l'utilisation des portes de Muller généralisées appelées aussi portes à seuil "threshold gates" (voir Figure 12). Les données sont codées en 3 états (§1.1.2.2.b).

La *valeur ajoutée* de cette méthode se situe au niveau de l'optimisation logique et des possibilités de projection technologique supérieures qu'offrent ces portes généralisées.



**Figure 12** : Fonctionnement d'une porte à seuil ( $M \leq N$ )

Toutefois, le code VHDL tel qu'il est à ce stade ne peut être simulé sans introduire la notion de "rendez-vous". Pour pallier ce problème, une nouvelle procédure, appelée "*hysteresis*", dont la propriété est d'informer le simulateur sur les conditions à remplir pour produire chaque sortie, est proposée. Les registres sont explicitement spécifiés dans le code afin de réaliser des étages de pipeline.

Enfin, la génération des signaux d'acquiescement, élément crucial des circuits asynchrones, doit se faire *à la main*, c'est-à-dire en instanciant les fonctions logiques dans le code VHDL sans garantie que les conditions d'acquiescement sont correctement gérées. Ce point faible est crucial dans la mesure où la grande majorité des méthodologies de circuits asynchrones gèrent d'une manière ou d'une autre les signaux d'acquiescement selon des règles bien définies. Cela représente souvent plus de travail que la partie combinatoire elle-même.

La méthode NCL n'est pas basée, contrairement à Tangram, Balsa et Caltech, sur la description des communications, mais sur une forme de communication simplifiée dans laquelle le concepteur a la charge d'implémenter *manuellement* les acquiescements selon le modèle souhaité. Il n'est nulle part possible de vérifier si la génération des signaux d'acquiescement est correcte ni qu'elle réalise le schéma de communication souhaité, ce qui constitue l'inconvénient majeur de cette méthode.

### 1.3. Vérification de circuits asynchrones

Un circuit est considéré comme correct si les séquences d'opérations qu'il effectue sont conformes au comportement prévu par le concepteur. Pour vérifier qu'un circuit est correct, son comportement prévu doit d'abord être exprimé d'une manière non-ambiguë, désignée sous le nom de spécification. Puis un procédé bien défini peut alors être exécuté pour déterminer si ce circuit est conforme à ses spécifications. La simulation numérique est aujourd'hui la méthode la plus couramment utilisée.

Quand le circuit et ses spécifications ont une base formelle, la vérification est apparentée à une preuve mathématique d'un comportement correct du circuit. Une telle preuve est aussi fiable qu'une simulation exhaustive, alors que les résultats obtenus par simulation de jeux de tests démontrent simplement qu'un circuit répond d'une certaine manière à un ensemble spécifique de stimuli d'entrée. Malheureusement, la vérification formelle est complexe et ses bases formelles sont méconnues des ingénieurs de matériel. En conséquence, le coût commercial de la vérification formelle est souvent élevé, rendant son utilisation peu répandue comparée à la simulation.

L'introduction des méthodes formelles dans le flot de conception est motivée par un besoin de correction et d'assurance plus élevée pour les concepteurs. Jusqu'ici, la simulation était le seul moyen de vérification, et elle intervenait tard dans le processus de conception. Il est donc d'un grand intérêt de concevoir et de développer des méthodes formelles dédiées à la conception asynchrone.

Les principaux travaux de recherche qui ont traité de la vérification formelle de circuits asynchrones sont fortement liés à deux caractéristiques :

- ✓ en premier lieu les formalismes de spécification : orientés langages ou orientés graphes,
- ✓ et en second lieu la modélisation du temps : circuits temporels ou insensibles aux délais.

Dans les paragraphes suivants, nous aborderons les outils les plus aboutis de vérification de circuits asynchrones. Nous commençons par les outils dédiés à la vérification des circuits asynchrones temporisés, qui peuvent dans certains cas concerner les circuits insensibles aux délais. La vérification des circuits insensibles aux délais est présentée en deux sections selon les niveaux de spécification.

#### 1.3.1. Vérification formelle de circuits asynchrones temporisés

Le choix de vérifier des modèles temporisés est généralement motivé par le fait que le flot de synthèse associé intègre les aspects de temporisation.

Un des principaux outils de vérification de circuits temporisés est l'outil ATACS. Il supporte la synthèse, l'analyse, et la vérification formelle [ZMM03]. Les spécifications en entrée sont principalement des conceptions décrites en langage VHDL, mais l'outil admet aussi les spécifications en STG de Petrify et les machines à état asynchrones. Les informations concernant les contraintes sur les délais de n'importe quelle transition de signal sont données explicitement. Elles sont utilisées le long du processus de conception pour les optimisations.

Les spécifications d'entrée sont compilées en une représentation interne en réseaux de Petri. La vérification est alors effectuée sur cette représentation grâce à des techniques de vérification à base de BDD. Plusieurs propriétés sont vérifiées pendant l'exploration de

l'espace d'états comprenant des propriétés de sûreté et l'absence d'interblocage. Quand des erreurs sont trouvées, une analyse sur le chemin conduisant à l'erreur est fournie.

L'outil OpenKronos dédié initialement à la vérification d'automates temporisés a été proposé par Bozga et al [BJM02] pour la vérification des circuits asynchrones temporisés. Dans ce travail, la vérification consiste à affirmer que les circuits synthétisés, exprimés en STG, se comportent correctement sachant que les délais sont compris entre deux bornes.

Cependant, dans ces approches de vérification, seuls des exemples petits ou de taille modérée peuvent être traités. La vérification des systèmes temporisés fait face à des problèmes sérieux de complexité, et reste très coûteuse. En revanche, notre travail reste à un niveau plus élevé d'abstraction, et l'on se restreint à des spécifications de circuits asynchrones insensibles aux délais.

### **1.3.2. Vérification formelle de circuits insensibles aux délais**

#### **1.3.2.1. Vérification formelle à bas niveau**

A ce niveau, les différents travaux de vérification utilisent l'approche symbolique de vérification de modèles. En effet, les formalismes de spécification sont basés sur les changements de valeurs d'un signal, c'est le cas des réseaux de Petri et des STG, il est alors naturel de choisir une représentation symbolique qui est très proche et par conséquent très adéquate. Un codage d'état symbolique est associé, permettant l'application des techniques de vérification symbolique de modèles en utilisant les BDD [KCK95, RCP96, KKo01, BNe01].

Parmi les principaux outils de vérification développés à l'heure actuelle pour la vérification de circuits asynchrones exprimés à bas niveau, nous présentons brièvement l'outil *Firemaps*[Neg98] dédié à la vérification des circuits asynchrones décrits en format *espaces de processus*, et l'outil *Versify*[Roi97] dédié, lui, à la vérification de circuits décrits en STG.

##### **a) FIREMAPS**

Firemaps est un outil de vérification, de conception et de test de circuits asynchrones ou mixtes (synchrones-asynchrones). Il fournit des constructeurs directs pour des modèles de circuit à temporisation relative au niveau porte, au niveau commutateur et au niveau cycle d'horloge. Ces constructeurs servent pour la description du circuit asynchrone.

FIREMAPS permet la vérification formelle de diverses classes de circuits asynchrones (insensibles aux délais, à délais relatifs, et même des circuits synchrones à horloges multiples) aux différents niveaux d'abstraction considérés et pour des combinaisons de ces niveaux.

Les systèmes et les propriétés à vérifier sont définis dans le formalisme d'espaces de processus. Cette approche permet la modélisation des circuits asynchrones insensibles aux délais ou à délais relatifs du genre "un chemin dans un circuit est toujours plus long en temps qu'un autre chemin"[BNe01]. Ce type de vérification présente un intérêt particulier par rapport aux approches basées sur des valeurs numériques pour les délais. En outre, il permet de procéder à la vérification quand les délais sur les fils ne sont pas encore connus.

L'outil de vérification Firemaps est basé sur une approche de vérification symbolique, il se sert en effet d'une bibliothèque riche de fonctionnalités pour la manipulation des BDD qui servent à structurer et manipuler les fonctions booléennes.

**b) Versify**

Versify est un outil pour la vérification de systèmes insensibles aux délais. Il s'agit d'une vérification de circuits décrits en STG dédiés à l'outil Petrify[CKK97]. La vérification est effectuée grâce à des techniques symboliques de vérification de modèles[RCP96].

La méthode de vérification est appelée *vérification plate*, elle consiste à calculer l'ensemble des états accessibles en exploitant les techniques de BDD et en vérifiant la non-présence d'un état d'erreur. Pour remédier au problème d'explosion d'états en réduisant le nombre de variables, Versify intègre une méthode de vérification hiérarchique, dans laquelle on calcule un circuit complexe, par abstraction de certaines portes internes. Alors pour un circuit  $C$  on obtient un circuit  $C_x$  formé par plusieurs portes complexes. Après avoir prouvé que  $C_x$  est correct, le graphe d'état obtenu est projeté sur les signaux d'interface de chaque porte complexe et pris comme environnement de cette porte. Vérifier que  $C$  est correct revient à vérifier la correction de toutes les portes complexes séparément.

La continuité de ce travail est un outil appelé Transyt[Transyt]. Dans ce nouvel outil, la vérification des propriétés temporelles est réalisée à travers une analyse symbolique d'atteignabilité basée sur les BDD. En cas de systèmes temporisés, la vérification est effectuée itérativement. À partir du système fondamental non-temporisé, l'outil insère automatiquement des contraintes de temps jusqu'à ce que la vérification soit satisfaite, ou qu'un contre-exemple temporisé soit trouvé. Transyt fournit une analyse en arrière, indiquant entre autres les contraintes de temporisation que l'outil a considérées pendant la vérification.

**1.3.2.2. Vérification formelle au niveau langage**

Les principaux langages de spécification ayant fait l'objet de travaux de recherches pour la vérification de circuits asynchrones sont : *Circal*, *Lotos* et *Promela*.

Les premiers travaux de vérification incluent des expériences avec l'environnement Circal. Les spécifications de circuits asynchrones sont modélisées avec le langage Circal. La preuve est exécutée sur la composition parallèle du système et des propriétés à vérifier, le tout modélisé par des processus concurrents [CMi00]. A notre connaissance, cette voie, qui n'a été appliquée qu'à des exemples très réduits, a été abandonnée.

Les travaux de vérification de circuits asynchrones utilisant le langage *LOTOS* ont été initiés par Yeol [YGi01]. Il propose une modélisation de la spécification et de son implémentation dans le langage *LOTOS*. La vérification consiste alors à vérifier par le biais de la boîte à outil CADP que l'implémentation est une réalisation correcte de la spécification.

*LOTOS* a été aussi choisi par He et Turner dans [HTu00]. Ils proposent l'utilisation du formalisme DIL "*Digital Logic in Lotos*" pour la spécification, la vérification et le test des circuits asynchrones. Les circuits asynchrones sont spécifiés en DIL, la vérification consiste à évaluer des propriétés temporelles par l'environnement CADP[CADP].

Dans de récents travaux, Marc Joseph propose dans une première approche d'utiliser l'outil SPIN[Ho197]. Il suggère de modéliser les circuits asynchrones en *Promela*[Ho197], le formalisme d'entrée de SPIN. On peut alors accéder aux performances de vérification de cet outil. Dans une deuxième approche, Joseph et Furey [JFu02] proposent un nouveau formalisme appelé DISP pour *Delay-Insensitive Sequential Processes* (Processus Séquentiels insensibles aux délais). DISP est un langage de programmation structurée et parallèle, mais

sans variables de programme ni affectation. Les instructions de base sont semblables au modèle *burst-mode* utilisé par Nowick, où les entrées et les sorties sont interprétées comme des transitions de signal.

Les spécifications en DISP se composent d'un couple de blocs, l'un décrivant le comportement du circuit et l'autre le comportement de l'environnement dans lequel il fonctionnera. Les outils Petrify et di2pn sont employés pour valider et synthétiser automatiquement les spécifications en DISP. L'outil di2pn sert à traduire les spécifications DISP au format d'entrée de Petrify. A ce niveau l'outil Versify se charge de la validation, alors que l'outil Petrify, lui, traite de la synthèse logique.

### **1.3.3. Discussion**

L'absence de standardisation dans le domaine des circuits asynchrones et plus particulièrement l'absence d'un langage ou d'un formalisme de spécification, justifie la grande diversité des méthodes et des orientations de recherche. Cette absence pose, entre autres, un problème concernant la convergence de ces travaux.

Les travaux de vérification de circuits à bas niveau présentent l'avantage d'une intégration dans de réels flots de synthèse asynchrone. Ces travaux profitent aussi des avantages des représentations en BDD, mais malgré cela ils souffrent, comme pour la spécification et la synthèse, de problèmes d'explosion d'états ; en effet les exemples traités restent de tailles modérées.

L'inconvénient principal des travaux basés sur les langages est l'absence de connexion aux flots de conception de circuits asynchrones. En effet, les langages LOTOS, Circal et Promela ne sont les entrées d'aucun flot de synthèse produisant les primitives de synchronisation spécifiques des circuits asynchrones. Concernant les travaux de Marc Josephs, on ne dispose pas de références nous permettant d'évaluer leur aboutissement. Néanmoins, dans l'approche où il propose d'utiliser DISP, di2pn et Petrify, le niveau d'abstraction est faible et les exemples qui peuvent être traités restent de tailles modérées.

Contrairement aux autres travaux, nos deux approches de vérification, que nous allons développer dans la suite du manuscrit, présentent l'avantage d'être pleinement intégrées dans le flot de conception asynchrone TAST. Elles présentent aussi une diversité dans les techniques adoptées, en effet nos travaux de vérification intègrent des techniques de vérification symboliques et aussi des techniques énumératives. Enfin, la possibilité d'une vérification à un niveau élevé d'abstraction est un net intérêt de nos travaux, ce qui nous a permis de vérifier des circuits asynchrones présentant un réel intérêt.

## 1.4. Objectif et plan du manuscrit

### *Objectifs*

L'objectif premier de notre travail est d'intégrer les méthodes formelles dans le flot de synthèse asynchrone TAST. Dans ce flot, les spécifications initiales sont écrites dans le langage CHP, elles sont interprétées en termes de réseaux de Pétri. L'étape de synthèse, effectuée sur ce format intermédiaire, exécute différentes étapes de décomposition et de raffinement, selon le choix de l'architecture cible : circuits micro-pipelines, quasi insensibles aux délais, ou même des circuits synchrones.

Nous avons, d'une part, proposé des approches pour la vérification formelle des circuits asynchrones, et d'autre part, développé des prototypes ou environnements permettant l'implémentation de ces approches.

### *Plan du manuscrit*

**Le chapitre 2** est consacré à la définition d'une sémantique en termes de systèmes de transitions étiquetées (STEE) pour le langage CHP. Il met en évidence tous les constructeurs du langage CHP et leurs correspondants dans un format intermédiaire décrit en STEE.

**Le chapitre 3** présente en détail les approches de vérification que nous avons proposées : une première approche basée sur un modèle dit *pseudo-synchrone* et utilisant des méthodes symboliques de parcours de l'espace d'états, et une deuxième approche basée sur un modèle purement asynchrone et utilisant des méthodes énumératives.

**Le chapitre 4** est une évaluation de performances de ces deux méthodes de vérification. Il met en évidence les difficultés rencontrées lors de la vérification d'un circuit de type arbitre asynchrone, la difficulté principale est bien entendu le problème de l'explosion du nombre d'états.

**Le chapitre 5** présente l'environnement de vérification CHP2IF, il met en évidence les différentes techniques et algorithmes de réduction et d'abstraction que nous avons réalisées.

**Le chapitre 6** présente quelques études de cas que nous avons traitées grâce aux prototypes que nous avons développés. On y trouve aussi de brèves descriptions des outils que nous avons utilisés dans nos travaux.



## **Chapitre 2**

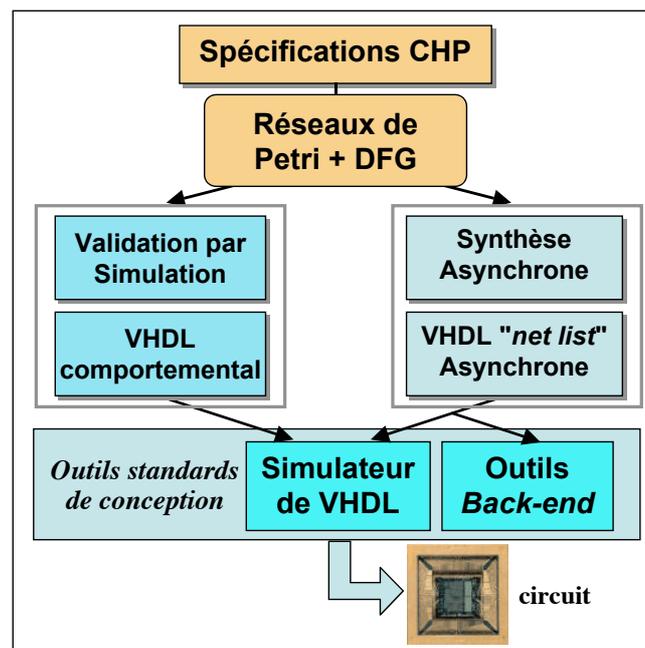
---

### **2. Sémantiques pour le Langage CHP**

## 2.1. Introduction

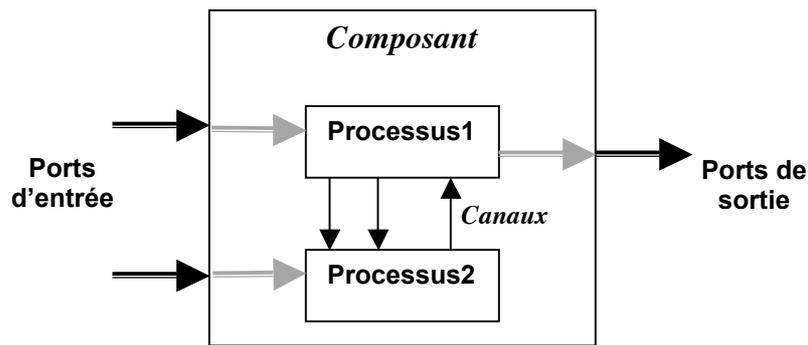
Les spécifications de circuits asynchrones que nous considérons sont décrites dans un langage de spécification de haut niveau. Ceci permet une description expressive et adéquate du fonctionnement global de systèmes asynchrones complexes, modulaires et éventuellement hiérarchiques. Ces circuits asynchrones sont constitués d'un ensemble de blocs fonctionnels concurrents connectés par des canaux. Les canaux assurent une communication point-à-point entre deux blocs ou entre un bloc et l'environnement du système. En conséquence, le langage choisi pour décrire les circuits asynchrones devrait être basé sur le concept de processus concurrents, qui communiquent par diverses formes : passage de messages, affectations de variables globales, files d'attente etc....

Comme il a été mentionné dans le chapitre précédent, plusieurs travaux de recherches ont été menés dans le domaine de la spécification et de la synthèse des circuits asynchrones. Dans notre travail, nous nous intéressons à l'environnement TAST[DFR02,RDR02], acronyme de "TIMA Asynchronous Synthesis Tools"(Figure 13). TAST est un environnement dédié à la synthèse asynchrone, et développé au sein du laboratoire TIMA par le groupe CIS du Professeur Marc Renaudin.



**Figure 13 :** TAST : Flot de synthèse asynchrone

Le langage CHP est le langage de spécification actuellement utilisé dans l'outil TAST : c'est une version enrichie du langage développé par Alain Martin à Caltech [Mar90]. Le Langage CHP s'inspire du langage CSP[Hoa78, Ren00], introduit par Hoare pour décrire des systèmes parallèles sous forme de processus séquentiels communicants, et des commandes gardées de Dijkstra [Dij76]. En effet, le circuit asynchrone est vu comme un ensemble de processus communicants qui lisent à partir de ports d'entrée, exécutent un traitement, et enfin écrivent sur des ports de sortie (Figure 14). Les extensions apportées au langage CHP, qui sont en majorité empruntées au langage VHDL, répondent aux besoins de synthèse et de simulation asynchrone.



**Figure 14 :** Structure générale d'une spécification en CHP

Comme il sera décrit plus précisément dans la suite de ce chapitre, syntaxiquement un programme CHP est composé d'un ou de plusieurs composants interconnectés par des canaux. Un composant noté "component" est composé :

- ✓ d'une partie déclarative contenant la description des ports, des canaux ;
- ✓ d'un ensemble de processus communicants via des canaux déclarés dans le composant.

Un processus est lui aussi défini par une partie déclarative et une partie fonctionnelle qui décrit le comportement souhaité.

## 2.2. La sémantique du langage CHP en réseaux de Petri

Le modèle utilisé comme base, pour les méthodes de représentation, de raffinement et de génération de circuits asynchrones, est dérivé du modèle de Réseau de Petri [Pet81]. En effet, les réseaux de Petri sont particulièrement adaptés à la mise en œuvre des méthodes de représentation des systèmes ayant des comportements mixtes concurrents et séquentiels. Un format intermédiaire n'est généralement pas lié à un langage en particulier, ce qui constitue une force pour le flot de synthèse asynchrone. Pour TAST, l'adoption du format intermédiaire permettrait facilement l'extension de l'outil en intégrant, en plus du langage CHP, d'autres langages de description de circuits asynchrones.

Les concepts présentés dans ce travail sont traités de manière plus élaborée dans [Din03], nous nous contentons alors de présenter brièvement les réseaux de Petri et le modèle particulier dédié à la synthèse asynchrone. La modélisation en un réseau de Petri d'une spécification CHP est illustrée à travers un exemple type : un arbitre asynchrone.

### 2.2.1. Réseau de Petri

#### Définition 1 (Réseau de Petri)

Un réseau de Petri, est un quadruplet  $(P, T, F, M_0)$ , où :

- $P$  est un ensemble fini de places
- $T$  est un ensemble fini de transitions
- $F$  est la relation de transitions du réseau de Petri,  $F \subseteq (P \times T) \cup (T \times P)$
- $M_0$  est le marquage initial

Le symbole  $\bullet t$  (respectivement  $t \bullet$ ) définit l'ensemble des places prédécesseurs (successeurs) de la transition  $t$ . Tandis que le symbole  $\bullet p$  (respectivement  $p \bullet$ ) définit l'ensemble des transitions prédécesseurs (successeurs) de la place  $p$ .

Un marquage du réseau de Petri associe un nombre entier positif, représentant le nombre de jetons, pour chaque place. Si un nombre  $k$  est associé à une place  $p$ , alors la place  $p$  est marquée par  $k$  jetons.

La relation de transition  $F$  définit un ensemble de règles concernant le comportement du réseau. Une transition est activée quand chaque place  $p \in \bullet t$  a au moins un jeton. Le réseau de Petri passe d'un marquage à l'autre par le franchissement d'une des transitions actives. Quand une transition active  $t$  est franchie, un jeton est supprimé de chaque place  $p \in \bullet t$  et un autre est ajouté pour chaque place  $p \in t \bullet$ .

Le réseau de Petri que nous considérons vérifie la propriété : à tout moment, chaque place contient au plus une marque. Ce type de Réseau de Petri est appelé “*sauf*” ou ordinaire “*safe Petri nets*” en anglais.

### 2.2.2. Combinaison de réseaux de Petri et des graphes de flot de données

La forme intermédiaire intégrée dans le flot de synthèse est un réseau de Petri surchargé par les graphes de flot de données DFG pour “*Data Flow Graph*” en anglais [Din03]. Les DFG sont des représentations graphiques qui décrivent les dépendances entre des opérateurs et des données.

#### Définition 2 (DFG : graphes de flot de données)

Un graphe de flot de données  $G$  est un graphe orienté défini par le tuple  $G = (V, E, T)$ , où :

- ✓  $V$  est l'ensemble des nœuds de  $G$ ,  $V = \{v_i, \dots, v_n\}_{i=1..n}$ , un nœud correspond à une opération.
- ✓  $T$  est l'ensemble des états terminaux, ce sont généralement des variables, ports ou des constantes du programme.
- ✓  $E$  est l'ensemble des arcs orientés de  $G$ ,  $E = \{(v_i, v_j) \mid v_i \in (V \cup T), v_j \in V\}$

Cette représentation qui combine les réseaux de Petri et les DFG, notée PN-DFG, permet la modélisation du parallélisme de manière concise. Les variables et les ports de la spécification ainsi que les opérations sont représentés en compréhension, et non pas en extension, ce qui se prête mieux à une représentation intermédiaire compréhensible.

La Figure 15 est une partie d'un PN-DFG, l'action de communication est “ $E2 !x+y$ ” exécutée si la garde “ $x = y$ ” est vraie.

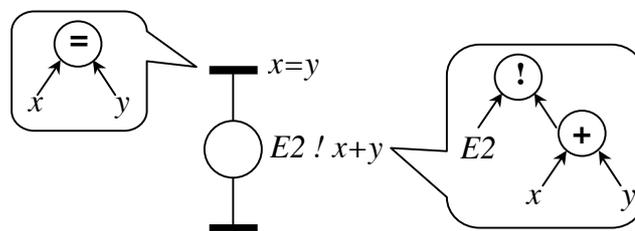


Figure 15 : exemple du format PN-DFG

Dans la sémantique d'exécution d'un réseau de Petri, les instructions sont associées aux places et les conditions de propagation aux transitions. Une instruction associée à une place est représentée par un DFG, elle peut être une affectation, une opération arithmétique, une

action de communication ou une expression plus compliquée. Une garde est une expression booléenne, représentée elle aussi par un DFG. En plus des opérateurs arithmétiques, les nœuds de ce DFG peuvent inclure des opérateurs de comparaison et l'opérateur “*probe*” chargé de tester la présence d'une communication sur un canal.

Tous les réseaux de Petri qu'on manipulera dans cette thèse sont des PN-DFG, aussi dans la suite du manuscrit nous ne faisons pas de distinction entre un PN-DFG et un réseau de Petri.

### 2.2.3. Construction du réseau PN-DFG d'une spécification CHP

Une spécification en CHP est composée d'une partie structure de contrôle et d'une autre partie flot de données. Le format PN-DFG est construit à partir de la spécification CHP de manière à conserver la séquentialité et le parallélisme des actions.

Par correspondance, chaque processus CHP est représenté par un PN-DFG. La modélisation en réseau de Petri ne traite que les structures de contrôle, les dépendances de données sont représentées par des DFG qui sont associés aux places et aux transitions du réseau de Petri.

La construction d'une représentation en format PN-DFG d'une spécification CHP prend en considération toutes les structures du langage CHP. Cette traduction est entièrement définie dans la thèse d'Anv hu[Din03] ; nous n'en donnons ici qu'une illustration.

#### 2.2.3.1. Exemple type d'un arbitre asynchrone

Nous prenons pour exemple un arbitre asynchrone typique (Figure 16). Cet exemple nous accompagne tout au long de la thèse.



Figure 16 : exemple type d'arbitre asynchrone.

Le circuit est caractérisé par quatre ports : un port de contrôle “C”, une entrée “E” et deux sorties (S1, S2). Le port de contrôle est de type Multi Rail “MR[3][1]”, c'est-à-dire un seul digit à (3) fils codé en 1 parmi 3. Notons que les aspects du langage CHP comme la structure du langage, les types MR et les constructeurs du langage sont traités dans la section 2.3.2.

Le circuit se comporte de la manière suivante : le signal de contrôle est lu, puis sa valeur est affectée à la variable locale *ctrl*. Cette dernière variable est testée par une structure de choix :

Si *ctrl* = “001”, (= “0”[3]), Alors on lit E et on l'écrit dans S1.

Si *ctrl* = “010”, (= “1”[3]), Alors on lit E et on l'écrit dans S2.

Si *ctrl* = “100”, (= “2”[3]), Alors on lit E et on l'écrit en parallèle dans S1 et S2.

Dans la Figure 17, on présente respectivement la spécification en CHP du sélecteur et sa représentation en termes de réseaux de Petri

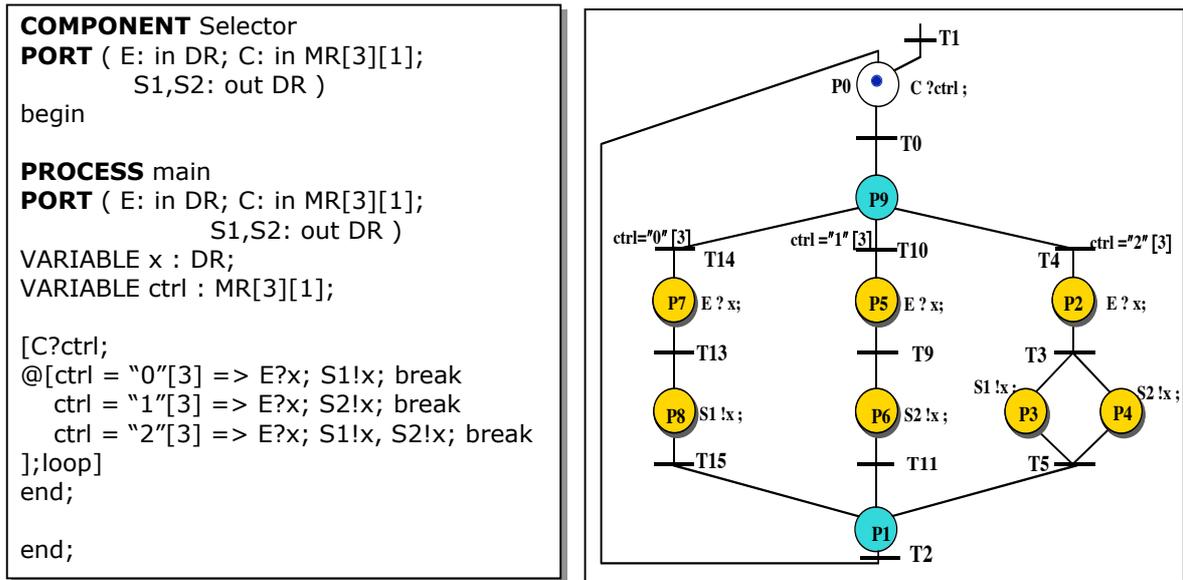


Figure 17 : Spécification en CHP du sélecteur et sa représentation en réseaux de Petri

## 2.3. Sémantique du langage CHP en systèmes de transitions étiquetées étendus

À partir de la sémantique opérationnelle du langage CHP, il est possible d'associer à tout programme donné un système de transitions étiquetées, qui est en fait un ensemble d'états muni d'une relation de transition. Ce STE modélise le comportement du programme, c'est-à-dire, l'ensemble de toutes ses exécutions possibles. Les STE sont le modèle de base pour plusieurs outils de vérification de modèles par énumération de l'espace d'états [CADP, Hol97].

Nous avons défini une sémantique des spécifications écrites en CHP en termes de systèmes de transitions étiquetées étendus «STEE». Un STEE est un automate en forme de système de transitions étiquetées sur lequel nous ajoutons la possibilité de manipuler des données (variables ou signaux). Les STEE servent de modèles intermédiaires entre des langages de spécification comme CHP et les modèles de STE utilisés dans la vérification formelle. Le modèle de STEE que nous obtenons modélise exactement le même comportement obtenu par les réseaux de Pétri.

Le besoin d'une sémantique est motivé par la méthode de vérification formelle des circuits asynchrones que nous avons développée et qui sera présentée dans la suite du manuscrit.

### 2.3.1. Systèmes de transitions étiquetées étendus

Un système de transitions étiquetées étendu est généralement exprimé en deux parties : une partie déclarative et une partie contrôle. La partie déclarative définit la structure et les paramètres caractérisant le système à modéliser. La partie contrôle est représentée par un automate dont les transitions sont étiquetées par des actions, c'est-à-dire des instructions qui décrivent le comportement du système. Nous commençons par présenter la notion de système de transitions étiquetées, utile pour la compréhension de la sémantique des systèmes de transitions étiquetées étendus présentés par la suite.

**Définition 3 (STE : Système de Transitions Etiquetées)**

Un système de transitions étiquetées  $S$  est un tuple  $S = (Q, A, T)$ , où :

- ✓  $Q$  est l'ensemble des états de  $S$ ,
- ✓  $A$  est l'ensemble des actions de  $S$ . Nous notons  $A \tau$  l'ensemble  $A \cup \{\tau\}$  où  $\tau$  est l'étiquette représentant une action non observable,
- ✓  $T$  est la relation de transition,  $T \subseteq Q \times A \tau \times Q$ , elle constitue un ensemble de transitions étiquetées. Une transition  $t = (q, a, q') \in T$  sera notée par  $q \xrightarrow{a} q'$ ;  $q, q'$  sont respectivement les états source et destination et  $a$  est l'action associée à  $t$ .

**Définition 4 (STEE : Système de Transitions Etiquetées Etendu)**

Un système de transitions étiquetées étendu  $E$  est un n-uplet  $(D, X, S, G, Q, T)$  où :

- ✓  $D$  est un ensemble de types de données. Pour chaque type de  $D$ , nous supposons que l'on dispose d'un ensemble d'opérations ;
- ✓  $X$  est un ensemble de variables discrètes typées.  $Exp[X]$  représente l'ensemble des expressions construites à partir de variables de  $X$  et d'opérations définies sur les types. Nous notons par  $Gard[X]$  l'ensemble des expressions booléennes. Finalement,  $Rst[X]$  l'ensemble des affectations aux variables de  $X$  ;
- ✓  $S$  est un ensemble de signaux typés. Les signaux permettent de rendre visibles à l'extérieur du modèle les valeurs des variables de ce modèle ;
- ✓  $G$  est un ensemble de portes de synchronisation typées. Deux processus échangent une valeur de manière synchrone, l'un écrivant et l'autre lisant sur une même porte. L'un des processus exécute  $g!e$  (écriture de  $e \in Exp[X]$  sur  $g$ ), et l'autre exécute  $g?x$  (lecture sur  $g$  et affectation de la variable  $x$ ).  
Nous notons par  $Sync[G, X]$ , l'ensemble d'actions synchrones aux portes de  $G$  ;
- ✓  $Q$  est l'ensemble des états de contrôle ;
- ✓  $T \subseteq Q \times Gard[X] \times Sync[G, X] \times Rst[X] \times Q$  est l'ensemble des transitions tel que : chaque transition contient de façon optionnelle : une garde, une ou plusieurs affectations de variables et une action de communication.

Ces STEE permettent de décrire le fonctionnement de systèmes parallèles et asynchrones. Pour les représenter, nous avons adopté le format intermédiaire IF [BFG00](§6.2.3.3), afin de disposer de l'ensemble des outils qui prennent ce format en entrée [BFG99, BGM01].

**2.3.1.1. Sémantique opérationnelle d'un système de transitions étiquetées étendu représenté en IF**

Dans la forme intermédiaire IF, la sémantique opérationnelle est complètement définie en termes de systèmes de transitions étiquetées. Concrètement, la sémantique consiste en un ensemble de règles permettant de construire, à partir d'un programme IF, l'ensemble de tous ses comportements, sous la forme d'un système de transitions étiquetées.

La sémantique d'exécution est indéterministe, à tout moment un processus IF est dans un certain état et, suivant les événements internes ou externes, il peut passer dans un autre. Le processus se déroule d'une manière totalement séquentielle en exécutant une transition à la fois.

Les éléments permettant de construire une sémantique opérationnelle dans un STEE décrit en IF sont les états, les transitions et les actions.

**a) Etat**

Les états correspondent aux modes de fonctionnement des processus : à tout moment un processus est dans un état et suivant les évènements internes ou externes qui se produisent, il peut passer dans un nouvel état. Un état peut être “*initial*”, c’est-à-dire que le processus peut démarrer son exécution à partir de cet état.

**b) Transition**

Les processus évoluent d’un état à l’autre en exécutant des transitions. Elles comportent une garde qui conditionne leur exécution et des affectations aux variables. Une transition synchrone comporte un rendez-vous synchrone avec échanges de valeurs sur une porte spécifiée.

**c) Action**

Les actions élémentaires associées aux transitions sont des synchronisations et des affectations. Les réceptions sur les ports d’entrée sont asynchrones, elles indiquent le signal attendu et les paramètres en réception. Les synchronisations comportent le nom de la porte utilisée et une liste d’offres, respectivement émissions de valeurs d’expressions ou réceptions de valeurs dans des variables. Là aussi, une garde supplémentaire portant sur les valeurs reçues lors de la synchronisation peut encore contraindre sa réalisation. Finalement, les affectations consistent soit à attribuer la valeur d’une expression à une variable, soit à remettre la variable à une valeur initiale fixée.

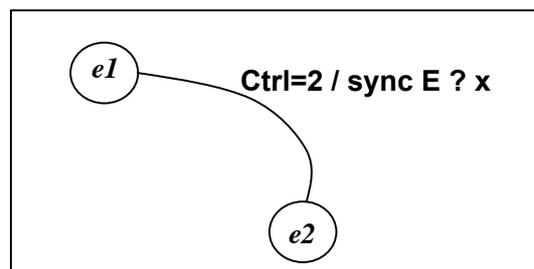
Les programmes IF manipulés au sein de notre approche sont des systèmes de transitions étiquetées étendus avec commande gardée et ayant la forme :

```
from EtatDépart {if Condition} {sync ActionCom} {do Actions} to EtatArrivée
```

A partir de l’état “EtatDépart”, si la condition “Condition” est vraie on exécute l’action de communication “ActionCom” et les actions “Actions” pour enfin passer à l’état “EtatArrivée”.

**d) Exemple**

Dans l’exemple de Figure 18, le processus se trouve dans l’état “*e1*”, une fois la condition “*Ctrl=2*” vraie, il exécute l’action de synchronisation “*E ?x*” et passe à l’état “*e2*”.



**Figure 18** : Une partie d’un STEE en IF

### 2.3.2. Construction du STEE correspondant à une spécification CHP

Dans ce paragraphe, nous décrivons en détail notre modèle de représentation de la sémantique du langage CHP en termes de systèmes de transitions étiquetées étendus. La construction du STEE est effectuée de manière compositionnelle, c'est-à-dire qu'un composant CHP est représenté par un système IF, et que chaque processus CHP est traduit en un processus IF.

La construction du système IF à partir d'une spécification CHP est présentée en quatre parties principales :

- ✓ Nous commençons par présenter la structure du programme CHP,
- ✓ nous continuons par la définition d'une sémantique opérationnelle des différentes instructions et structures de composition,
- ✓ ensuite on aborde la traduction des types et des variables,
- ✓ finalement on traite les aspects de communication.

#### 2.3.2.1. Structure du programme CHP

La structure globale d'une spécification CHP consiste en un ou plusieurs composants, chaque composant est un ensemble de processus concurrents qui communiquent à travers des canaux.

##### a) Composant

Chaque composant "component" CHP est traduit en un système "system" IF et de même, chaque processus "process" CHP est traduit en un processus "process" IF. Le tableau ci-dessous nous montre les correspondances entre les différents éléments des composants des deux formalismes.

Spécification en CHP	Programme IF
COMPONENT système_id	SYSTEM système_id ;
/	TYPE ...
PORT ( I1, I2, ..., : IN Type ; ... .. O1, O2, ..., : OUT Type ; ... .. )	GATE ( I1, I2, ..., : Type ; /* Input */ ... .. O1, O2 ..., : Type ; /* Output */ ... .. )
/	VAR ... ..
channel c1, c2, ... BD/DI : Type ; ... ..	SYNC ... .. END ;
-- Fin Partie déclaration	/* Fin Partie déclaration */

<b>Begin</b> -- Le corps d'un ou de plusieurs processus concurrents, <i>Process_1</i> ... <i>Process_i</i> end système_id	/* Le corps d'un ou de plusieurs processus concurrents */ <i>Process_1</i> ... <i>Process_i</i>
---	---

### b) Processus

C'est dans le corps du processus qu'est décrit le comportement de la spécification ainsi que les différentes actions de communications avec les autres processus ou l'environnement. Le bloc d'instructions décrivant le comportement du processus est traduit en un ensemble d'états et de transitions suivant les règles de construction définies par la suite.

Spécification en CHP	Programme IF
<b>PROCESS</b> Process_id <b>PORT</b> ( I1, I2, ..., :IN Type ; ... .. O1, O2, ..., :OUT Type ; ... .. )	<b>PROCESS</b> Process_id ;  /
<b>VARIABLE</b> v1, v2 :TYPE ; ... ..	<b>VAR</b> v1, v2, ... :TYPE ; ... ..
<b>begin</b> [ S1, ..., Si ; Sj, ... --- Si est une suite d'instructions. ]	<b>State</b> state +
<b>end</b>	<b>transition</b> transition +

Les ports d'un processus CHP sont déclarés en IF dans la partie déclarative du composant. En effet, en IF tous les ports sont déclarés dans un seul endroit au début du système. Par conséquent, pour chaque *port* déclaré au niveau d'un processus CHP : on vérifie la présence de ce *port* dans la partie déclarative du système IF, s'il n'est pas présent, on l'ajoute. Concernant les variables locales, elles sont traduites au même niveau dans le système IF.

### c) Synchronisation

#### Sémantique des rendez-vous en CHP

La communication par rendez-vous en CHP est totalement point à point, elle ne peut se faire qu'entre deux processus distincts (ou un processus et l'environnement). Par exemple on ne peut pas écrire à partir d'un port une donnée sur un canal, et cette donnée est récupérée par plus d'un processus concurrent. La synchronisation est alors implicite, il suffit de déclarer les canaux, qui doivent avoir le même nom que les ports reliés à ces canaux.

#### La sémantique de synchronisation dans IF

Comme en CHP, la communication en IF est par rendez-vous. Mais contrairement à CHP, une expression de synchronisation explicite décrit comment les processus communiquent de

manière synchrone les uns avec les autres par l'intermédiaire des portes de synchronisation. Une telle expression est construite à partir des identificateurs de processus en utilisant l'opérateur binaire de composition parallèle avec synchronisation sur un ensemble de portes.

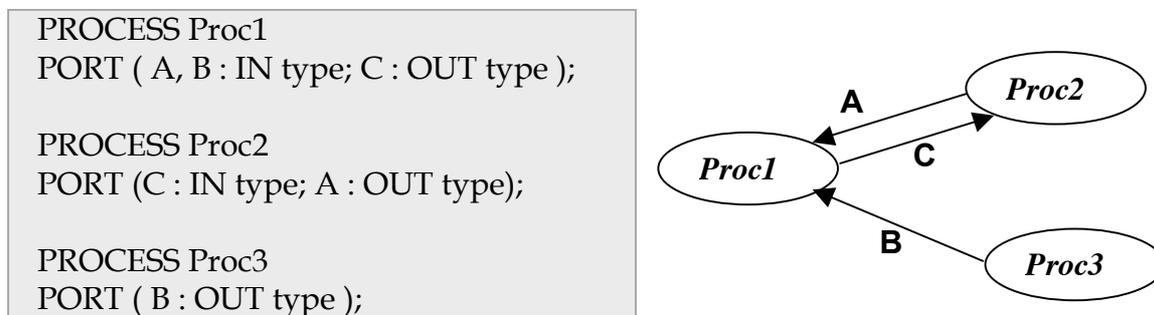
Pour établir une synchronisation entre deux processus P1 et P2 sur un ensemble de portes G0, ... Gn, on dispose de trois opérateurs de synchronisation :

- ✓ La construction  $P1 \mid [G0, \dots Gn] \mid P2$  dénote le comportement qui exécute P1 et P2 en parallèle. La synchronisation et la communication entre P1 et P2 s'effectuent uniquement par rendez-vous sur les portes  $\{G0, \dots Gn\}$ . Lorsqu'un des processus veut exécuter une transition étiquetée par une porte  $G_i$ , il doit attendre que l'autre puisse faire autant. Lorsque le rendez-vous est possible, les deux processus exécutent simultanément une même transition étiquetée par la porte  $G$  ; puis ils reprennent chacun leur exécution.
- ✓ Le deuxième opérateur " $P1 \mid \mid P2$ " exprime l'absence de synchronisation, les deux processus sont exécutés de manière totalement indépendante.
- ✓ Le dernier opérateur " $P1 \mid \mid P2$ " exprime la synchronisation sur toutes les portes, les deux processus sont exécutés de manière synchronisée.

### Construction de l'expression de synchronisation

La déclaration des canaux en CHP est remplacée en IF par une expression de synchronisation. Il faut parcourir l'ensemble des processus présents et identifier les ports de synchronisation avec les processus restants.

**Exemple :** Prenons comme exemple les en-têtes des processus et la représentation schématique de la Figure 19.



**Figure 19 :** Exemple de synchronisation

Pour l'exemple de la Figure 19, nous pouvons utiliser les formes de synchronisation suivantes :

```

Proc1 |[ A,B,C ]| (Proc2 | | Proc3)
ou
Proc2 |[ A,C ]| ( Proc1 |[B]| Proc3)
ou
Proc3 |[B]| (Proc1 |[ A,C ]| Proc2)

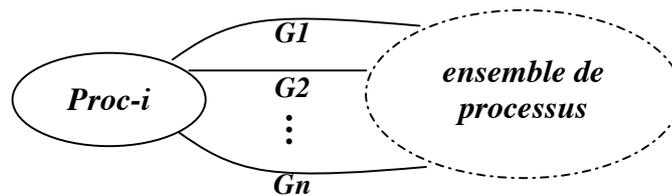
```

La formulation de synchronisation "**Proc1** |[X,Y]| **Proc2**" signifie que le processus **Proc1** est synchronisé avec le processus **Proc2** sur les ports de communication X et Y. L'expression

**Proc1** || **Proc2** exprime l'absence de synchronisation, alors que **Proc1** || **Proc2** synchronise les deux processus sur l'ensemble des ports existants dans le système.

On remarque que IF ne se préoccupe pas du sens de la communication, mais cela ne change en rien la sémantique d'exécution du système.

Dans le cas général, pour «*n*» processus concurrents, on sélectionne un processus qui est en communication avec les *n-1* autres processus par le biais de la totalité de ses ports. L'opération est répétée pour les processus restants, il faut prendre soin de ne pas répéter un canal déjà utilisé comme synchronisation.



**Figure 20** : Généralisation de la synchronisation à *n* processus concurrents

D'où l'expression de synchronisation :

$\text{Proc-}i \text{ } || \text{ } [ p_1, p_2 \dots p_n ]$  ( *synchronisation de l'ensemble restant* )

La génération automatique de l'expression de synchronisation est implémentée grâce à un algorithme récursif qui effectue le parcours des en-têtes de tous les processus, détermine les ports et les opérateurs de synchronisation entre les différents processus et construit comme expliqué auparavant l'expression de synchronisation. Notons que ce travail doit être réalisé à la fin de l'étape d'analyse du programme CHP pour permettre la synchronisation des nouveaux processus générés au moment de la compilation avec l'ensemble des processus existant à l'origine. Ces nouveaux processus sont généralement générés pour la traduction du parallélisme interne à un processus CHP.

### 2.3.2.2. Contrôle et structures

Dans ce paragraphe, on traite de la modélisation des structures de contrôle du langage CHP en termes de STEE. Dans ce qui suit, les symboles  $G_i$  expriment des gardes et les  $S_i$  symbolisent des ensembles d'instructions.

#### a) Structure de commande gardée

Une garde dans le langage CHP est une expression booléenne, elle est associée à la transition dans le cas de la modélisation en réseaux de Petri.

Généralement, une garde est suivie par une instruction ou un bloc d'instructions. Tant que la garde est fautive, on attend. Une fois qu'elle devient vraie, on exécute l'instruction ou les instructions correspondantes, elle est aussi utilisée dans les structures de choix qui seront présentées par la suite.

Dans la modélisation en STEE, la garde est reproduite sur la transition du STEE, la suite des instructions ne serait exécutée qu'après le franchissement de cette garde.

Spécification en CHP	Programme IF
G->S	from ei if G do S to ej ; /* Traduction de S en IF */



Figure 21 : Opérateur de commande gardée

**b) Opérateur de séquentialité**

Deux instructions ou blocs d'instructions S1 et S2 sont en séquence si S2 ne peut commencer qu'après la fin de S1, la séquentialité s'écrit : S1 ; S2. Pour décrire cet opérateur en STEE on distingue trois cas :

- ✓ Le premier cas est le cas général, la séquentialité entre deux blocs d'instructions (S1 et S2) est traduite en deux blocs de STEE successifs séparés par un état ;
- ✓ Dans le deuxième cas, S1 et S2 sont de simples affectations (ou S1 est une simple affectation et S2 une liste de simples affectations), la séquentialité est traduite en STEE par une séquence de simples affectations sur une même transition ;
- ✓ Enfin, S1 peut être une action de communication alors que S2 est une liste de simples affectations. Dans ce cas aussi la séquentialité est traduite par une action de communication suivie par une séquence de simples affectations sur la même transition.

Dans les deux derniers cas, S1 et S2 sont séparées par une virgule “,”, qui signifie l'opérateur de séquentialité en STEE, à la place du point-virgule “;”.

Spécification en CHP	Programme IF
S1 ; S2	<p><b>Cas1</b> : cas général from ei do S1 to ej ; from ej do S2 to ek ;</p> <p><b>Cas2</b> : S1 et S2 sont de simples affectations from ei do S1, S2 to ej ;</p> <p><b>Cas3</b> : S1 est une action de communication et S2 une simple affectation from ei sync S1 do S2 to ej ;</p>

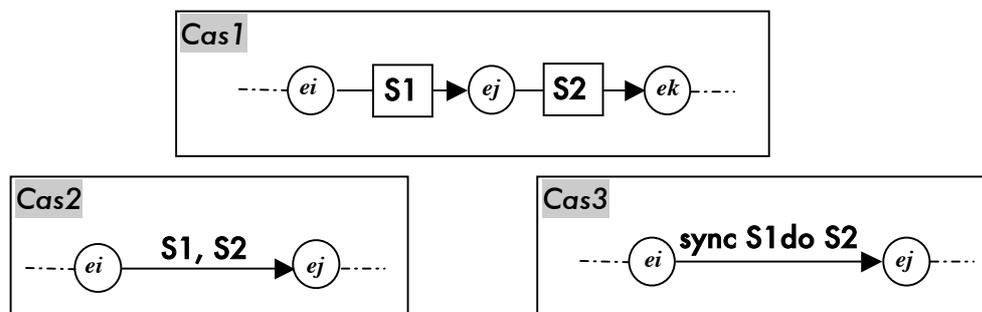


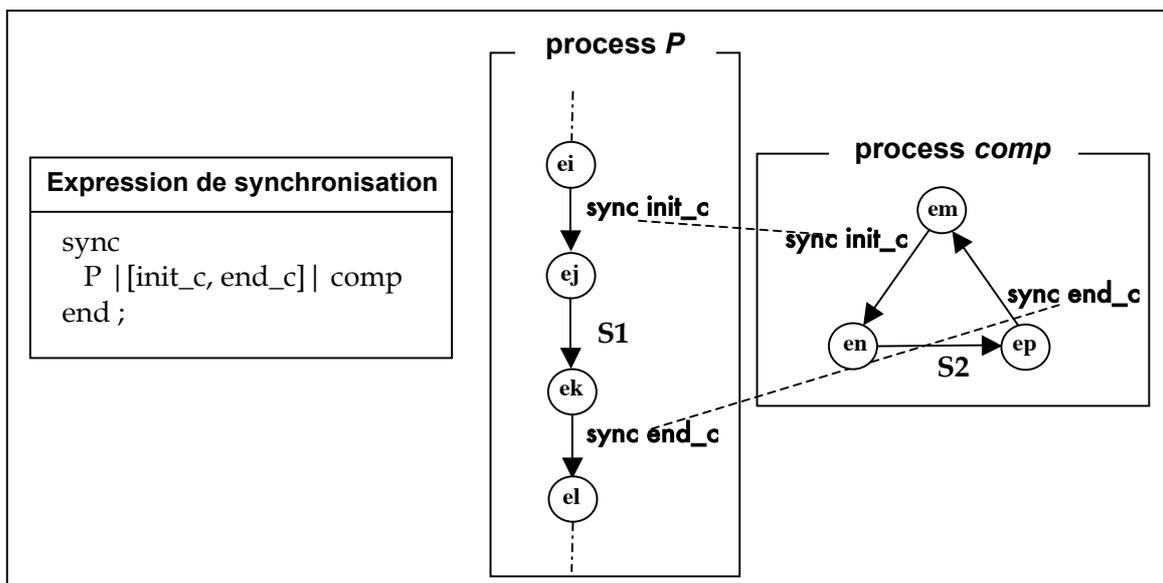
Figure 22 : Opérateur de séquentialité

**c) Opérateur de parallélisme**

A l'intérieur d'un processus CHP, la concurrence entre deux instructions ou blocs d'instructions est exprimée par l'opérateur “,”. Dans un système STEE, on ne peut pas exprimer de la concurrence à l'intérieur d'un processus. Les processus sont en effet séquentiels et ne peuvent contenir des opérateurs de mise en parallèle. On peut toutefois obtenir le même résultat en utilisant de nouveaux processus concurrents.

Par exemple, pour exprimer en IF la concurrence entre [S1 , S2], on ajoute un nouveau processus *comp* décrivant le comportement de S2 et une expression de synchronisation pour synchroniser le nouveau processus sur le processus original. Comme décrits dans la Figure 23 : les comportements S1 et S2 s'exécutent en parallèle, ils sont synchronisés sur les ports (init\_c et end\_c).

Spécification en CHP	Programme IF
<p style="text-align: center;">S1 , S2</p>	<pre> ... .. sync /* synchronisation sur les portes : init_c et end_c. */   P  [init_c, end_c]  (comp) end ; <b>Process P</b> ... .. from ei sync init_c to ej ; /* démarrer le processus Comp <b>S1</b> /* traduction de S1 en IF */ from ek sync end_c to el ; /*synchro. avec Comp ... .. <b>process Comp</b> ... .. from em sync init_c to en ; /*synchro. avec P <b>S2</b> /* traduction de S2 en IF */ from en sync end_c to ep ; /*synchro. avec P                     </pre>



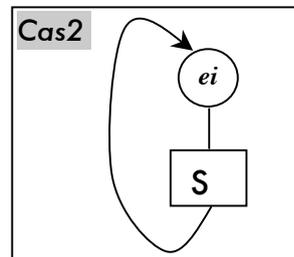
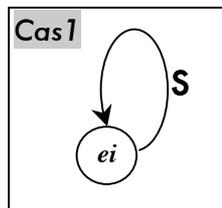
**Figure 23 :** Opérateur de parallélisme.

**d) Opérateur de répétition**

En CHP, pour exprimer une boucle d'une ou de plusieurs instructions, il suffit de terminer par l'opérateur "loop". Pour reproduire ce même schéma en IF, on distingue deux cas :

- ✓ le cas d'une simple instruction et là on revient au même état,
- ✓ le second cas, s'il y a une boucle sur tout un bloc d'instructions, alors tous les états finaux sont reliés à l'état initial.

Spécification en CHP	Programme IF
[ S ; loop ]	<p><b>Cas1</b> : S est atomique : from ei do S to ei ;</p> <p><b>Cas2</b> : S est une suite d'instructions : from ei ...     - -traduction de S en IF to ei ;</p>



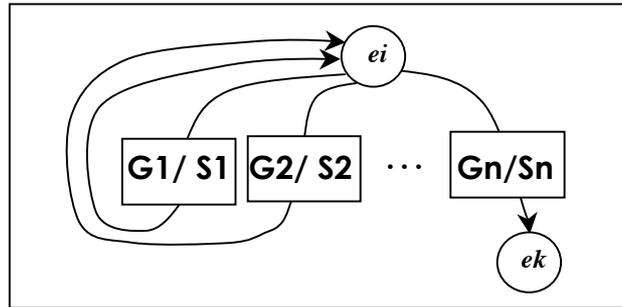
**Figure 24** : Opérateur de répétition

**e) Structure de sélection**

Le langage CHP offre deux types de structure de choix gardée : la structure déterministe (@) et la structure non déterministe (@@). Dans le cas d'une structure déterministe, tant qu'aucune garde n'est vraie, on attend. Dès qu'une garde et une seule est vraie, on exécute l'instruction correspondante. Dans le cas d'une structure de choix indéterministe, tant qu'aucune garde n'est vraie, on attend. Dès qu'une ou plusieurs gardes deviennent vraies, un tirage aléatoire d'une garde parmi l'ensemble des gardes vraies est effectué et l'instruction correspondante à cette garde est ensuite exécutée.

Dans les deux cas, la structure de choix résultante en IF est une structure indéterministe puisque IF n'offre qu'une seule forme de choix.

Spécification en CHP	Programme IF
<p>@[  G1 =&gt; S1 ; loop  G2 =&gt; S2 ; loop  ... ..  Gn =&gt; Sn ; break  ]</p>	<p>from ei if (G1) do S1 to ej ;  from ei if (G2) do S2 to ej ;  ... ..  from ei if (Gn) do Sn to ek ;  from ej to ei ;</p>



**Figure 25** : Structure de répétition avec commandes gardées.

Comme on le voit dans la Figure 25, à l'intérieur d'une structure de choix on peut avoir des opérateurs de répétition. Dans ce cas, on boucle sur le bloc d'instruction en testant à chaque fois l'ensemble des gardes. On dispose de la commande *break* pour sortir de la boucle.

En CHP, c'est le concepteur qui est en charge d'assurer l'exclusion mutuelle entre gardes dans le cas d'utilisation d'une structure déterministe, car ces contraintes ne sont pas vérifiées par le compilateur CHP. Cette tâche de vérification sera traitée dans le chapitre 5.

### 2.3.2.3. Types et variables

#### a) Les types

Le CHP possède un grand nombre de types différents. Tous ces types sont basés sur le type principal appelé MR "Multi Rails".

MR[B] : type Multi-Rails avec une base B. Il est conforme à un codage "1 parmi B"

MR[B][L] : un type vecteur de L éléments de type Multi-Rails dans une base B. Une variable de ce type peut représenter un nombre entre 0 et  $(B^L - 1)$ .

A la base de ces deux types, d'autres types génériques sont définis :

- ✓ BIT : représente un nombre binaire. → équivalent à MR[2]
- ✓ BOOLEAN : représente une valeur booléenne. → équivalent à MR[2]
- ✓ DR : type de double rail. → équivalent à MR[2]
- ✓ DR[L] : type de l'élément de type double-rails. → équivalent à MR[2][L]
- ✓ SMR[b][m] : vecteur multi-rails signé dans la base b.
- ✓ INTEGER [max] : représente un entier allant de 0 à max. → équivalent à MR[2][L] (L est le plus petit nombre entier supérieur ou égal à  $\log_2[\text{Max}+1]$ )
- ✓ SR : représente un seul rail, sert uniquement pour la synchronisation. → équivalent à MR[1]

Dans un premier temps, nous ne nous intéressons pas à la manière dont les communications sont réalisées, mais seulement à la valeur des variables. C'est-à-dire, qu'on se situe à un niveau d'abstraction dans lequel les actions de communications sont considérées comme atomiques, et s'exécutent d'une manière synchronisée par un mécanisme de poignée de main. Nous avons donc converti chaque type CHP en un intervalle de valeurs entières.

**Exemple :**

Revenons à notre exemple d'arbitre, la figure suivante présente la traduction d'une des parties déclaratives de l'exemple.

Spécification en CHP	Programme IF
<b>PORT</b> ( E : IN DR ; C : IN MR[3][1] ; S1, S2 : OUT DR )	<b>TYPE</b> MR_3_1 = range 0..2 ; MR_2_1 = range 0..1 ; <b>GATE</b> E(MR_2_1) ; /* IN PASSIVE */ C(MR_3_1) ; /* IN PASSIVE */ S1(MR_2_1), S2(MR_2_1) ; /* OUT ACTIVE */

**b) Les variables :**

En termes de variables, le langage CHP ne dispose que des variables locales, en effet il n'y a pas de possibilité de déclaration de variables globales. Les variables locales à un processus CHP sont traduites en variables IF locales elles aussi au processus.

Certaines variables locales à un processus CHP sont amenées à être globalisées en IF. Cette tâche concerne les variables d'un processus qui a été éclaté, afin de traduire les actions concurrentes internes à ce processus. Ces variables sont évidemment indexées par le nom du processus pour permettre leur identification.

**c) Les opérations associées aux différents types**

La majeure partie des opérateurs en CHP a leur correspondant en IF, et dans certains cas elle a les mêmes identificateurs. Alors, la traduction des expressions se fait en substituant les opérateurs CHP par les opérateurs de IF.

Certains opérateurs binaires à connotation logique (xor, nand...) sont présents dans le CHP, mais n'existent pas en IF. Pour les constantes, on peut calculer selon l'opérateur la nouvelle valeur à utiliser. Par contre pour les variables, on ne peut pas le faire directement. Toutefois, IF permet de créer son propre type permettant d'utiliser des fonctions personnelles créées en C. Il est donc possible de réaliser toutes les opérations.

Spécification en CHP	Programme IF
Opérateurs unaires not   +   neg	Opérateurs unaires not   +   -
Opérateurs binaires or   and   =   /=   <   >   <=   >=   +   -   *   /   mod	Opérateurs binaires or   and   =   <>   <   >   <=   >=   +   -   *   /   %
Affectation :=	Affectation :=
Opérations logiques Xor, Nand	<i>N'existe pas</i>

### 2.3.2.4. Communication

La synchronisation en CHP est implicite, il suffit de déclarer les canaux, qui ont les mêmes noms que les ports auxquels ils sont connectés. En IF c'est différent, la synchronisation est explicite : une expression de synchronisation décrit les communications entre les différents processus et les ports de synchronisation.

Les lignes de déclaration des canaux sont remplacées par une expression de synchronisation. La construction de cette expression a été développée dans la Section 2.3.2.1.

On ajoute dans la traduction des commentaires, par exemple : (*Input, et Output*), afin de distinguer les ports d'entrées et de sorties dans le système IF.

#### a) Actions de communication

La communication est établie via des canaux, ils portent le même nom (identificateur) que les ports communicants des deux processus, ce qui garantit la correspondance entre les ports et les canaux. Les échanges de données à travers un canal sont directionnelles : de l'émetteur au récepteur. L'émetteur (ou le récepteur) est appelé *actif* s'il est l'initiateur de la communication, sinon il est *passif*. Cette notion *d'initiateur de communication* est utilisée, plus loin (§c), pour expliquer la traduction de l'opérateur *probe*.

Par défaut, l'émetteur est considéré comme *actif*, et le récepteur *passif*. Les mots clés "passive" et "active" associés à la déclaration des ports permettent d'imposer l'initiateur de la communication.

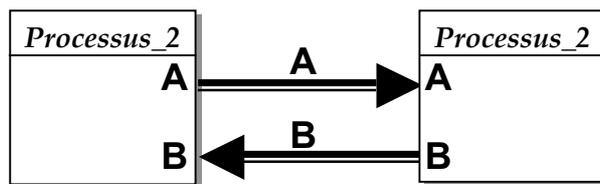


Figure 26 : Communications entre processus

Spécification en CHP	Programme IF
E ?x	from e0 sync E ?x to e1 ;
S !exp	from e0 sync S !exp to e1 ;

**Lecture d'un canal** : E ?x, où E est un port d'entrée et x une variable interne.

**Ecriture sur un canal** : S !exp, où S est un port de sortie et exp peut être une expression au format *Data Flow Graph* (§2.2.2) ou une variable interne.

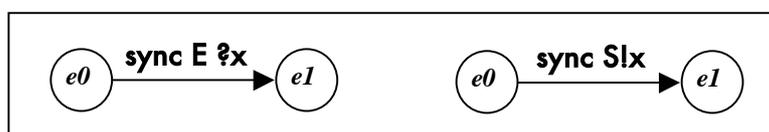


Figure 27 : Actions de Communication

#### b) Expansion des communications

Le long du processus de synthèse asynchrone, les actions de communications sont amenées à être raffinées. En effet, une des étapes de la synthèse asynchrone est l'expansion de la communication. Cette étape d'expansion explicite la manière dont les processus communiquent entre eux, elle consiste en premier lieu à choisir un protocole pour décrire les différentes phases de communication. Le protocole le plus souvent utilisé est le protocole

quatre phases, il est également nommé RZ comme Retour à Zéro. Ce protocole de communication entre un émetteur et un récepteur est implémenté grâce au mécanisme de *hand shake* en quatre phases :

Le récepteur détecte une nouvelle donnée, effectue le traitement et génère le signal d’acquiescement. (2) l’émetteur détecte le signal d’acquiescement et invalide la donnée. (3) le récepteur détecte l’état d’invalidité des données et désactive le signal d’acquiescement. Enfin (4) l’émetteur détecte l’état d’invalidité de l’acquiescement, il émet une nouvelle donnée si elle est disponible. (voir l’exemple de la Figure 28).

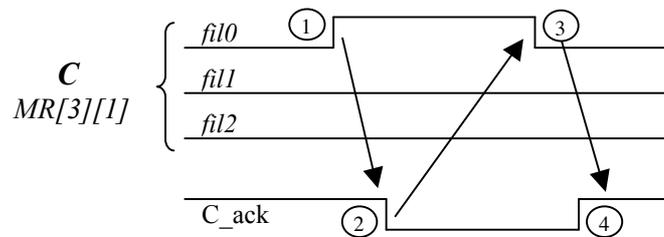


Figure 28 : Protocole 4 phases dans un canal de communication asynchrone

Lors des différentes étapes du protocole, il est important de choisir un bon codage pour les données afin d’exprimer la validité des données, le codage des requêtes et des acquiescements. Pour ce faire, le codage DI (*Delay insensitive ou insensible au délai*) est largement adopté, il s’agit d’un codage de type MR (multi rail). C’est un codage (1) parmi (n). Pour (n) bits de données, on prévoit (2n) fils plus (1) fil d’acquiescement, une donnée invalide est codée par la valeur (0) partout (voir chapitre1).

**Exemple du codage DI :**

Prenons un port A d’un circuit asynchrone en CHP, on suppose que A est de type MR[2][2], A : IN MR[ 2 ][ 2 ] (A a 2 digits de 2 fils)

		valeurs			
		0.0	1.0	0.1	1.1
A	0	1	0	1	0
	1	0	1	0	1
	1	1	1	0	0
	0	0	0	1	1

Figure 29 : Exemple de codage en DI

Si A = “0.0”[ 2 ] alors, A(0)=1, A(1)=0, A(2)=1 et A(3) =0. En d’autres termes A=“0101”.

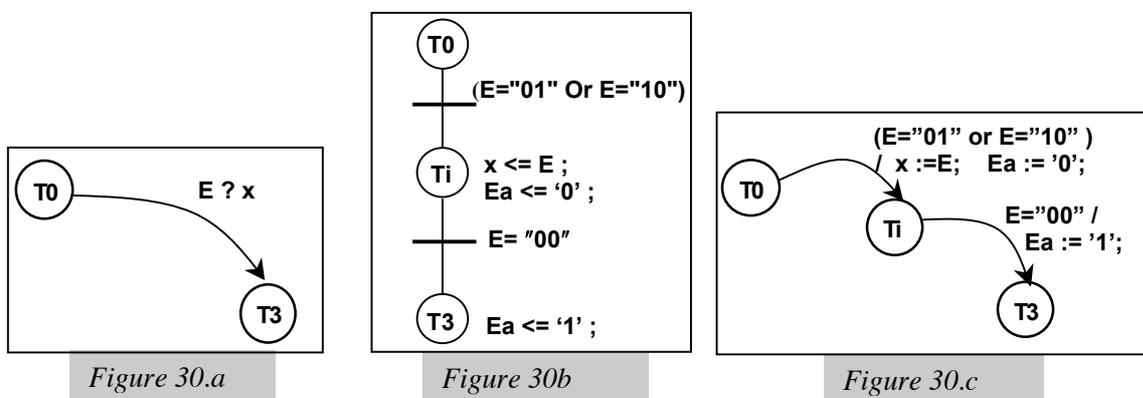
En IF comme on l’a vu, les actions de communication sont instantanées, elles s’exécutent de façon atomique. Afin d’avoir une représentation de la spécification CHP après expansion de la communication, nous proposons une présentation de ce mécanisme en réalisant explicitement l’action de communication par une partie d’un système de transitions étiquetées en s’inspirant du protocole quatre phases décrit ci-dessus. Il est important de noter que les ports sont remplacés par des variables et qu’on crée de nouvelles variables pour décrire le comportement des acquiescements nécessaires au déroulement de la synchronisation. Pour chaque port, on dispose de : une variable d’acquiescement et une variable de donnée qui sert aussi pour la synchronisation (voir Figure 28).

Cette traduction permet de construire une deuxième représentation des spécifications écrites en CHP, à un niveau d'abstraction moins élevé que le précédent puisqu'il s'agit de la spécification après expansion des communications. En construisant ce nouveau STEE décrit en IF, on peut alors valider le protocole de communication et traduire directement l'opérateur "Probe" décrit ci-après.

La traduction est illustrée sur l'exemple de l'arbitre asynchrone, on choisit une communication entre les deux ports E et S1, l'un émettant un  $x$  sur un port, et l'autre le recevant.

**Action de lecture (E ? x)**

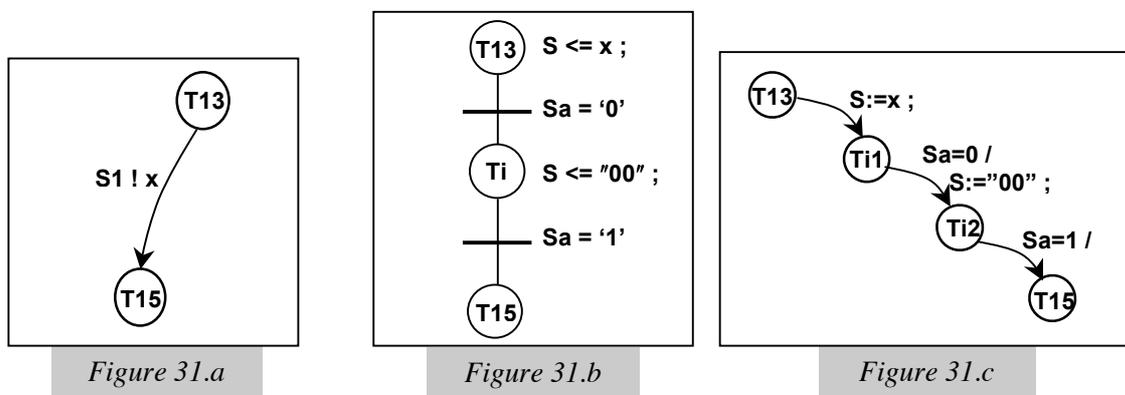
Les figures : *Figure 30.a*, *.b* et *.c* représentent respectivement : l'action de lecture avant l'expansion des communications, en réseau de Petri et après expansion des communications en STEE. L'implémentation d'une action de lecture nécessite l'introduction d'une nouvelle place  $T_i$  et d'un nouveau signal d'acquiescement  $Ea$ .



**Figure 30** : L'expansion d'une action de lecture "E ?x"

**Action d'écriture (S1 ! x)**

Les figures b.4, b.5 et b.6 représentent respectivement : l'action d'écriture avant l'expansion des communications, en réseau de Petri et après expansion des communications en STEE. L'implémentation d'une action d'écriture nécessite l'introduction de deux nouvelles places  $T_{i1}$  et  $T_{i2}$  et d'un nouveau signal d'acquiescement  $Sa$ .



**Figure 31** : L'expansion d'une action d'écriture "S!x"

### c) Opérateur probe “#”

Dans certains cas, les concepteurs de circuits asynchrones désirent savoir si une communication est prête à être exécutée. Cette fonctionnalité est assurée par le biais d'un opérateur appelé "probe". Cet opérateur s'applique à un port de communication. Il teste la présence d'une donnée sur le canal concerné si ce port est en réception, ou la possibilité d'envoyer une donnée si le port est en émission. Comme cet opérateur n'existe pas en IF, il faut le simuler en générant de nouvelles actions.

```
@[A# => A?x; break]
```

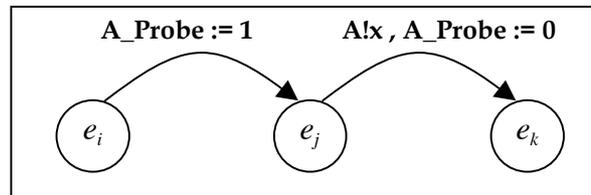
Il est intéressant de remarquer qu'une communication en CHP ne peut se faire qu'entre deux processus. Cela permet d'utiliser une seule variable globale booléenne quel que soit le type de communication pour exprimer l'attente dans le protocole *poignée de main*.

```
A_Probe = 0
```

La valeur de cette variable exprime qu'une communication peut ou non être établie. Les processus communicant avec un processus utilisant l'opérateur probe doivent changer les actions de communication mettant en jeu ce port. La variable globale doit être affectée à un (1) juste avant la communication, puis remise à zéro (0) lorsque la communication se termine.

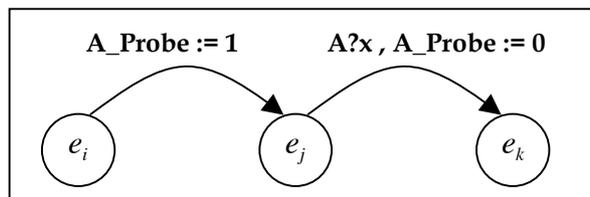
Dans d'une action de communication, seul le processus connecté au canal *actif* est l'initiateur de la communication. Cette règle garantit l'absence de conflit sur la variable  $A\_Probe$ .

Si un port  $A$  est interrogé par un opérateur *probe* alors l'action d'écriture  $A!x$  est étendue de la manière suivante :



**Figure 32 :** Etendre une action d'écriture “ $A!x$ ”

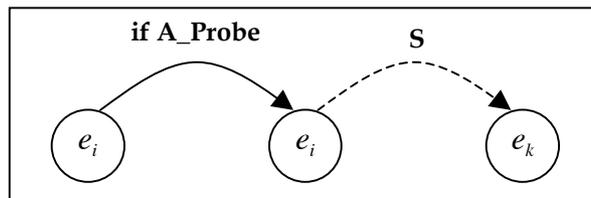
L'action de lecture  $A?x$  est elle aussi étendue :



**Figure 33 :** Etendre une action de lecture “ $A?x$ ”

Après transformations des actions de communication concernées par un opérateur *probe*, nous pouvons alors utiliser une garde qui teste la variable **A\_Probe** à la place de l'opérateur *probe* (#) quel que soit le sens de la communication.

Spécification en CHP	Programme IF
[ A# => S]	from ei if A_Probe to e1 ; /* Traduction de S en IF. */



**Figure 34 :** Traduction de l'opérateur probe (#)

## **Chapitre 3**

---

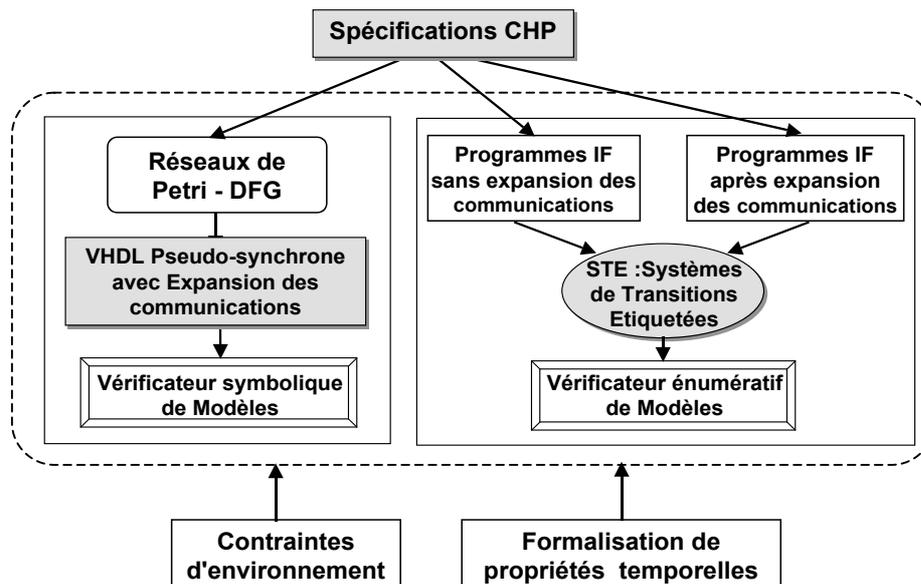
### **3. Approches de Vérification**

### 3.1. Introduction

L'objectif de notre travail étant d'intégrer les méthodes formelles dans le flot de synthèse de circuits asynchrones, nous allons présenter dans ce chapitre les approches proposées pour l'introduction des méthodes de vérification formelle dans le flot de synthèse TAST. Les circuits asynchrones à vérifier sont exprimés initialement dans le langage de spécification CHP, ensuite traduits en forme intermédiaire appelée PN-DFG[Din03] basée sur les réseaux de Petri et les graphes de dépendance des données.

Une première approche a été de proposer une méthodologie de vérification utilisant les outils industriels existants de vérification de circuits. Cette approche est basée sur un modèle de représentation dit pseudo-synchrone en utilisant les techniques de vérification symbolique de modèles "*symbolic model checking*" en anglais.

Constatant que cette première approche nécessite une modélisation à un niveau d'abstraction inférieur (après expansion des communications), et que nous introduisons une horloge fictive pour rendre visibles les cycles d'exécution, nous avons développé une deuxième approche qui se veut purement asynchrone. Dans cette approche, les spécifications initiales sont modélisées en termes de systèmes de transitions étendus pour ensuite être vérifiées par des méthodes de *model checking énumératives*. Cette alternative consiste alors à utiliser des formalismes et outils de vérification issus du domaine de la validation de logiciel, et notamment des systèmes distribués, dont le modèle d'exécution est similaire à celui des circuits asynchrones.



**Figure 35** : Approches de vérification de circuits asynchrones

La Figure 35 présente les deux approches de vérification qui seront développées en détail dans la suite de ce chapitre.

## 3.2. Première approche : Modélisation pseudo-synchrone et vérification symbolique

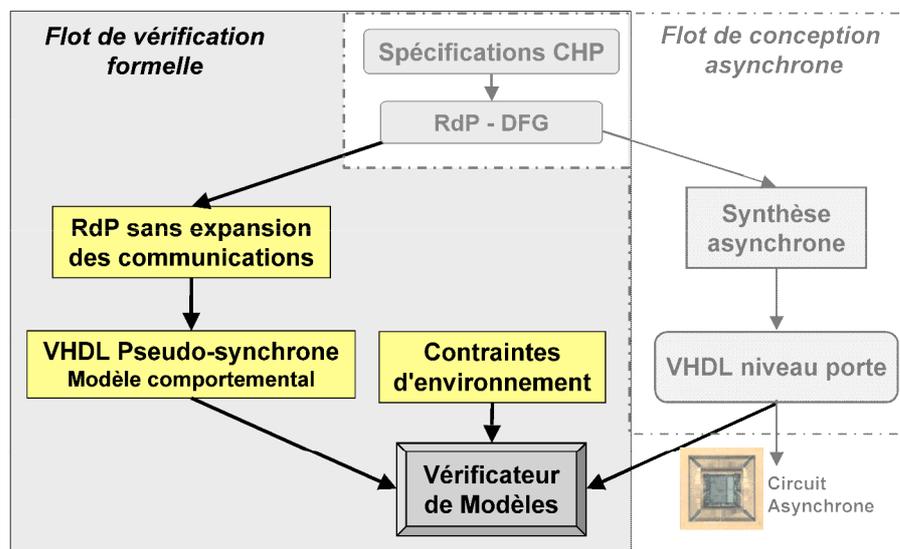
L'approche de vérification consiste à considérer l'utilisation d'outils industriels existants pour vérifier les descriptions de circuits asynchrones. Pour ce faire, nous avons développé un flot de vérification formelle permettant l'accès à des outils de vérification, tels que FormalCheck[Bell], VIS[Bra95], RuleBase[BBE97], etc...

Dans la suite de cette partie, nous allons présenter cette méthode de vérification basée sur un modèle que nous avons appelé *pseudo-synchrone* en utilisant des techniques de vérification symbolique de modèles. Notons que dans le reste du manuscrit nous ne considérons que le langage VHDL[IE00a], un langage de description de circuits standardisé IEEE. Ce langage est supposé connu du lecteur.

Dans cette approche, nous intervenons à deux niveaux différents dans le processus de synthèse asynchrone :

- ✓ Une première étape de vérification consiste en une validation de la spécification initiale du circuit asynchrone écrite en CHP, et ce par la construction d'un programme VHDL correspondant au programme CHP initial et (ou) au réseau de Petri intermédiaire, ensuite l'application d'une méthodologie de vérification utilisant des outils de vérification symboliques sur la description résultante qui est en VHDL ;
- ✓ Dans une autre étape, nous vérifions la préservation de certaines propriétés logiques du circuit asynchrone après synthèse par rapport à sa spécification initiale traduite en VHDL.

La figure suivante présente le flot de vérification qui constitue cette première approche de vérification.



**Figure 36 :** Approche de vérification pseudo-synchrone

Nous commencerons par une description générale de la méthode, suivie par un bref rappel des techniques de vérification symbolique pour aider à la compréhension de la méthode de vérification. Enfin, nous présenterons les différentes étapes de vérification : traduction, modélisation et vérification de propriétés temporelles [BBD02, BBD03].

### 3.2.1. Description de la méthode

Le principe fondamental dans cette approche est de considérer le réseau de Petri comme une machine d'états finis (voir paragraphe 3.2.2.1), où le marquage représente l'état courant, et où l'on passe d'un état à un autre en changeant de marquage. La construction à partir du réseau de Petri d'une représentation en forme de machine d'états finis décrite en un *langage de description de matériel HDL* nous permet de procéder à un processus de vérification formelle en utilisant des outils industriels efficaces de vérification de modèles.

Ces outils de vérification sont dédiés à la vérification de circuits numériques *clockés*, ils prennent en entrée des descriptions écrites en langages *HDL*. (VHDL[IE00a], Verilog [IEE01], ...). Ces descriptions doivent être conformes à certaines normes d'écriture. Par conséquent dans un flot de conception, un passage du Petri net vers une description HDL (VHDL dans notre cas) est nécessaire pour pouvoir appliquer certains outils de vérification de circuits numériques.

L'objectif est de proposer une sémantique en VHDL équivalente à ce que serait la simulation en réseaux de Petri en utilisant la sémantique de simulation de VHDL. La méthode proposée pour réaliser ce passage est décrite par un algorithme de traduction, qui parcourt le réseau de Petri (places et transitions) pour produire un VHDL respectant la sémantique des constructions en termes de concurrence et de séquentialité.

Tous les concepts du réseau de Petri peuvent trouver rapidement leurs correspondants en VHDL sauf le canal. L'absence de cette notion de canal de communication en VHDL rend impossible l'expression directe des actions de communication, il est alors nécessaire d'explicitier le protocole de déroulement de ces actions. L'expansion de la communication est réalisée grâce au mécanisme de poignée de main. Une fois l'expansion de la communication faite, on obtient un réseau de Petri dont les instructions associées aux places ne sont que de simples affectations. Le réseau de Petri obtenu après expansion des communications est automatiquement traduit en VHDL pour les besoins de vérification formelle.

La description en VHDL obtenue par traduction est destinée à être soumise à un outil de vérification symbolique. Généralement ces outils sont dédiés à la vérification de circuits synchrones, une étape de *pseudo-synchronisation* est alors nécessaire pour pouvoir utiliser de tels outils. Cette étape consiste à ajouter une horloge fictive qui rend visibles les différents *cycles* de simulation du circuit (*cycles delta* en VHDL). L'ajout de cette horloge nous permettra d'accéder à des outils de vérification puissants (FormalCheck, VIS ...).

### 3.2.2. Techniques de vérification symbolique de modèles

Les méthodes symboliques peuvent être appliquées pour modéliser et vérifier toutes sortes de systèmes. Elles consistent à ne pas énumérer des ensembles d'états, comme c'est le cas des méthodes énumératives qui seront présentées plus loin, mais à représenter ces ensembles à l'aide de formules. Dans le cas d'un système d'états finis, il est extrêmement fréquent qu'un état du système soit codé au moyen d'un ensemble fini de variables booléennes, ou variables d'états. C'est le cas, plus particulièrement des circuits, où le système peut être codé par des variables booléennes. Tout ensemble d'états est alors représentable par une formule.

Comme toute technique de vérification formelle, les méthodes symboliques dépendent nécessairement du modèle de description, nous allons alors présenter brièvement un des modèles de représentation utilisés dans les méthodes symboliques ainsi que les principes de la vérification symbolique [Deh93, Mad90].

### 3.2.2.1. Modèle de machine d'états finis

Le modèle le plus largement adopté pour la vérification formelle est celui des Machines d'Etats Finis, (FSM pour *Finite State Machine*). Dans une FSM, il y a un nombre fini d'états, et chaque état a des transitions vers zéro, un ou plusieurs états. Les états qui ne possèdent pas de transitions sont appelés *états finaux*.

#### Définition 5 (*Machine d'états finis*)

Une machine d'états finis est un n-uplet  $M = \langle X, O, S, s_0, F_t, F_o \rangle$ , où :

- ✓  $X$  est un ensemble fini de symboles d'entrée,
- ✓  $O$  est un ensemble fini de symboles de sortie,
- ✓  $S$  est un ensemble fini d'états,
- ✓  $s_0$  est l'état initial de la machine d'états finis  $M$ ,  $s_0 \in S$ ,
- ✓  $F_t : X \times S \rightarrow S$  représente la fonction de transition de  $M$ ,
- ✓  $F_o : X \times S \rightarrow O$  représente la fonction de sortie de  $M$ ,

Les machines d'états finis que nous adopterons sont déterministes, c'est-à-dire que la fonction de transition  $F_t$  associe pour chaque couple appartenant à  $(X \times S)$  une valeur unique dans  $S$ . Elles sont conformes à la machine de Mealy [Mea55], le changement d'état s'opère suite au changement de valeurs de symboles d'entrée, la fonction de transition est alors exécutée et l'état successeur est atteint.

Une échelle discrète de temps est généralement associée à une machine d'états finis, à tout instant, une machine d'états finis  $M$  se trouve dans une configuration, par exemple à l'état initial  $s_0$   $M$  est dans la configuration initiale, et on dit alors qu'à l'instant "0"  $M$  est à l'état initial.

### 3.2.2.2. Représentation symbolique d'une machine d'états finis

La représentation symbolique d'une machine d'états finis est basée sur la logique propositionnelle, elle permet de manipuler des ensembles arbitraires d'états d'une machine sous forme de formule logique. L'avantage d'une telle représentation est que la taille d'une formule n'est pas nécessairement liée au nombre d'états qui la satisfont, et que l'on peut ainsi représenter et calculer des ensembles gigantesques d'états.

Les machines d'états finis ont tout d'abord été représentées par des Diagrammes de Décision Binaire (BDD "*Binary Decision Diagrams*" en anglais) [Bry86, Bry92]. Dans le but de remédier au problème de l'explosion d'états en nombre de variables, d'autres approches de codage et de parcours des *FSMs* ont été proposées : les techniques de SAT [BCC99], le parcours logarithmique [Kuk96].

#### a) Représentation symbolique d'une machine d'états finis

La représentation symbolique d'une machine d'états finis  $M$  consiste à coder les fonctions de transition et de sortie en une représentation symbolique à bases des BDD par exemple.

La machine  $M$  est alors définie par le  $n$ -uplet  $M = \langle X, O, S, F_t, F_o, \chi_0 \rangle$ , où :

- ✓  $X, O$  et  $S$  sont respectivement l'ensemble des variables d'entrée, de sortie et d'état.
- ✓  $F_t : \mathbb{B}^m \times \mathbb{B}^n \rightarrow \mathbb{B}^n$  est la fonction de transition, telle que,  $m=|X|$ ,  $n=|S|$  ;
- ✓  $F_o : \mathbb{B}^m \times \mathbb{B}^n \rightarrow \mathbb{B}^k$  est la fonction de sortie, telle que,  $k=|O|$  ;
- ✓  $\chi_0$  est la fonction caractéristique de l'état initial de la machine d'états finis  $M$ ,

### b) Parcours symbolique d'une machine d'états finis

Le modèle symbolique d'une FSM sert de point de départ dans les outils actuels de vérification symbolique de modèles. Le parcours symbolique d'une machine d'états  $M$  est effectué à l'aide de la fonction de transition. Le calcul des successeurs d'un ensemble d'états  $E$  est appelé *image* de  $E$  par la fonction de transition  $F_t$  de  $M$ .

Sachant que  $M$  contient un nombre fini d'états, lorsque l'ensemble de départ du parcours est la fonction caractéristique de l'état initial, l'application itérative du calcul d'image converge vers un ensemble d'états à partir duquel aucun nouvel état n'est atteint, ce qu'on appelle point fixe. Cet ensemble est l'*ensemble des états atteignables* de  $M$ .

### 3.2.2.3. Application à la vérification

La description du système à vérifier étant modélisée par des machines d'états finis symboliques, les spécifications attendues du système sont elles aussi exprimées en formules logiques. Ce type de vérification est appelé vérification symbolique de modèles. Le processus de vérification consiste à déterminer si le système symbolique décrit en machine d'états finis satisfait un ensemble de spécifications de bon comportement.

L'évaluation de la *satisfaisabilité* d'une formule logique  $P$  sur une machine d'états finis symbolique  $M$ , revient à évaluer la formule  $M \models P$ , qui signifie que la formule logique  $P$  est satisfaite par  $M$ . Cette évaluation est réalisée grâce à deux types d'algorithmes :

- ✓ La *vérification en avant* consiste à appliquer de manière itérative la fonction de transition  $F_t$ , en commençant par la fonction caractéristique de l'état initial  $\chi_0$ . Le calcul s'arrête si au moins un état atteint ne satisfait pas  $P$ , ou si l'ensemble des états atteignables a été parcouru. Dans le cas où l'ensemble des états atteignables aurait été parcouru sans que  $P$  n'ait été faux, alors  $M \models P$ .
- ✓ La *vérification en arrière* concerne généralement la vérification des propriétés de sûretés. Elle part de l'ensemble des états qui vérifient  $\neg P$  et applique de façon itérative la fonction *pré-image*, duale de la fonction de transition, jusqu'à calculer le plus petit point fixe. Si l'état initial ne se trouve pas parmi l'ensemble des états résultant, alors  $M \models P$ .

### 3.2.2.4. Expression de propriétés

Le comportement souhaité pour un système est exprimé par un ensemble de propriétés. Ces propriétés sont d'une manière classique [Lam77] distinguées en deux catégories qui font chacune appel à des techniques différentes de vérification.

- ✓ Les propriétés de sûreté (*safety en anglais*) qui expriment que quelque chose de mauvais ne se produit jamais. Un exemple type d'une propriété de sûreté est l'exclusion mutuelle.
- ✓ Les propriétés de vivacité (*liveness en anglais*) qui expriment que quelque chose de bon finira par arriver, par exemple l'absence de famine.

Ces propriétés sont exprimées par des formules logiques écrites dans des langages dits “*langages temporels*”. Les langages temporels sont des extensions de la logique classique par des opérateurs modaux permettant de construire des formules dont l’interprétation en un certain nœud de l’arbre dépend des interprétations dans les autres nœuds.

Le comportement d’un système peut être modélisé soit par un ensemble de séquences soit par un ensemble d’arbres, par conséquent on distingue deux sémantiques pour les langages temporels, linéaire et arborescente :

- ✓ Logiques linéaires comme LTL[Lam80] et PTL[MPn92], qui permettent d’exprimer des propriétés portant sur les états d’un chemin d’exécution du système.
- ✓ Arborescente comme CTL[CEm81] et CTL\*[CES83], permettant d’exprimer des propriétés portant sur les arbres d’exécution du système.

### La logique linéaire et arborescente CTL\*

La logique temporelle CTL\* est une extension de la logique CTL (*Computation Tree Logic en anglais*). CTL\* exprime aussi bien des comportements arborescents que linéaires. Cette logique est particulièrement adaptée à la spécification de circuits numériques et elle nous permettra de comprendre les propriétés exprimées dans le paragraphe 3.2.4.2.

#### Définition 6 (La logique temporelle CTL\*)

La logique temporelle CTL\* contient des formules sur les états  $\varphi$  et des formules sur chemins  $\vartheta$  ayant la syntaxe suivante :

$$\begin{aligned}\varphi &::= p \mid \neg \varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid E\vartheta \\ \vartheta &::= \varphi \mid \neg \vartheta_1 \mid \vartheta_1 \wedge \vartheta_2 \mid X\vartheta_2 \mid \vartheta_1 U \vartheta_2\end{aligned}$$

où  $p$  est une proposition atomique,  $E$  dénote le quantificateur existentiel sur les chemins,  $X$  est l’opérateur “neXt” et  $U$  est l’opérateur “Until”.

Soit  $M$  un modèle d’exécution,  $e = (e_0, e_1, \dots, e_i, \dots) \in \mathbb{B}^n$  un chemin d’états,  $s$  un état,  $P$  un prédicat exprimé sur  $X \cup S \cup O$  et  $f$  un prédicat, une formule de chemin ou d’état.

- ✓ Le chemin  $e$  satisfait  $p$ ,  $e \models p$ , si et seulement si  $P$  est vrai à l’état  $e_0$  ;
- ✓  $e_0 \models Xf$  si et seulement si le successeur direct de  $e_0$  satisfait  $f$ . ( $\Leftrightarrow e_1 \models f$ ) ;
- ✓  $e \models f_1 U f_2$  si et seulement si  $\exists k \geq 0$  tel que  $e_k \models f_2$  et  $\forall i : 0 \leq i < k : e_i \models f_1$  ;
- ✓  $s \models Ef$  si et seulement si il existe un chemin  $e = (s, \dots)$  commençant par  $s$  tel que  $e \models f$  ;
- ✓  $e \models Ff$  si et seulement si  $\exists i \geq 0$  tel que  $e_i \models f$  ; où  $Ff = true U f$  ;
- ✓  $e \models Gf$  si et seulement si  $\forall i \geq 0 : e_i \models f$  ; où  $Gf = \neg F\neg f$  ;

L’opérateur  $A$  est déduit des opérateurs primitifs définis ci-dessus :

- ✓  $s \models Af$  si et seulement si pour tout chemin  $e = (s, \dots)$  commençant par  $s$ ,  $e \models f$  ;

### 3.2.3. Traduction du Petri net vers VHDL

#### 3.2.3.1. Représentation d’un réseau de Petri dans le flot TAST

Les réseaux de Petri que nous considérons sont constitués de nœuds hétérogènes : des places où sont rattachées les actions et des transitions dont le franchissement est conditionné par des gardes (Figure 37). Afin de traduire fidèlement le réseau de Petri en VHDL, nous commençons par présenter les paramètres caractéristiques associés aux places et aux transitions, pour les prendre en compte lors du processus de traduction.

Chaque place est caractérisée par les éléments suivants :

- id* : identificateur
- det* : le choix est déterministe ou pas,
- action* : action à exécuter,
- nbr\_pt* : nombre de transitions d'entrée,
- trans* : les transitions de destination.

Une place P est alors structurée comme suit :  $P(id, det, action, nbr\_pt) \rightarrow trans$

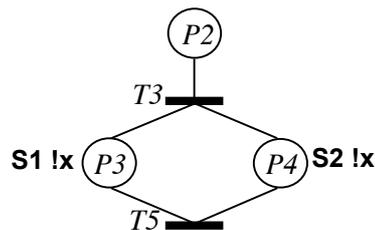
La place P3 de la Figure 37 est caractérisée par : un identificateur  $id = 3$ , une action d'écriture  $action = S1 !x$ , une transition d'entrée T3 et une transition de sortie T5.

Chaque transition est caractérisée par les éléments suivants :

- id* : identificateur
- cond* : condition à satisfaire,
- boucle* : elle forme une boucle vers une place qui la précède ou pas,
- nbr\_pt* : nombre de places d'entrées,
- places* : les places de destination.

Une transition T est structurée comme suit :  $T(id, cond, boucle, nbr\_pt) \rightarrow places$

La transition T3 de la Figure 37 est caractérisée par : un identificateur  $id = 3$ , une condition  $cond = true$ , une place d'entrée P2 et deux places de sortie P3, P4.



**Figure 37 :** Exemple de réseau de Petri manipulé dans le flot TAST

### 3.2.3.2. Sous-VHDL pour la vérification formelle

Il existe un certain nombre de propositions de sous-ensembles VHDL pour la vérification formelle [DOD93, Deh93, Deh96], qui s'appuient sur la sémantique de simulation. Pour ces restrictions du langage, la sémantique de simulation est modélisable par une machine d'états fini FSM[Dum03].

Du point de vue de la synthèse, des sous-ensembles ont été proposés, pour lesquels il est possible d'associer à chaque primitive retenue une réalisation matérielle assurant la même fonctionnalité. Le sous-ensemble de VHDL pour la synthèse a été standardisé IEEE[IE00b] et les outils de vérification prennent en entrée une restriction de ce sous-ensemble. Ces restrictions permettent de garantir que toutes les mémoires qui sont écrites sont synchronisées sur le même front d'horloge.

Les principales restrictions pour l'écriture de descriptions VHDL dédiées à la vérification formelle sont :

- ✓ Il existe une seule horloge globale
- ✓ Absence d'éléments mémorisants (les *latches*) autres que les bascules synchronisées sur front d'horloge.
- ✓ Les fonctions de résolution ne sont pas permises dans certains outils de vérification

- ✓ Les STD-ULOGIC sont ramenés à des bits. Les outils de vérification formelle ne peuvent prendre en charge, dans le meilleur des cas, plus de 4 valeurs pour un signal
- ✓ Les objets de type *entier* ou *naturel* sont convertis en leur représentation *bit-vector*.
- ✓ Les instructions *after* sont ignorées dans la synthèse, par contre refusées pour la vérification

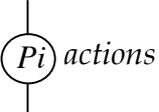
### 3.2.3.3. Passage du réseau de Petri vers un modèle vérifiable en VHDL

Le comportement que l'on désire observer dans l'exécution d'un réseau de Petri est l'évolution du marquage des places. La traduction du réseau de Pétri est réalisée autour des deux concepts : place et de transition. Lorsqu'une place est marquée par jeton, l'action qui lui est associée est alors exécutée. Les transitions décrivent l'évolution du marquage.

Un réseau de Petri est traduit en VHDL par un processus synchronisé par le front montant de l'horloge fictive. Un signal de «reset» permet de le réinitialiser. Les places sont traduites par des signaux affectés lors du franchissement des transitions, lesquelles sont exprimées par des instructions conditionnelles d'affectation.

#### a) Traduction d'une place

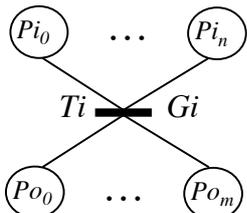
La place est représentée par un signal booléen. Le jeton est dans la place, donc elle est active et le signal vaut *true*, sinon la place n'est pas active et le signal vaut *false*.

Réseau de Petri	Programme en VHDL
	<pre> ... .. Signal Pi : boolean ; ... .. if Pi then -- traduction en VHDL des actions                     </pre>

Un signal VHDL est calculé une fois par cycle de simulation. Aussi, l'introduction d'une horloge fictive et la représentation d'une place par un signal chargé sur condition de front de cette horloge permet d'observer le changement de marquage, et donc le comportement du réseau de Petri.

#### b) Traduction d'une transition

Pour chaque transition  $T_i$  correspond un code VHDL qui exprime : si les places entrantes de la transition  $T_i$  sont activées et les places sortantes sont désactivées, et que la condition  $G_i$  associée à  $T_i$  est vraie, alors on désactive les places entrantes et on active les places sortantes.

Réseau de Petri	Programme en VHDL
	<pre> ... .. if (Gi and (Pi0 and ... PIn) ) then   Pi0 &lt;= false ; ... PIn &lt;= false ;   Po0 &lt;= true ; ... Pom &lt;= true ;                     </pre>

### c) Traduction des gardes et des actions

Les gardes et les actions sont issues du langage CHP. Sachant que les instructions de CHP sont inspirées de VHDL, les gardes et les actions sont généralement en correspondance directe avec les expressions et les instructions de VHDL.

Notons quand même qu'il existe des aspects propres à CHP qui doivent être pris en compte. Les opérations qui manipulent les types MR sont implémentées dans une bibliothèque appelée TAL "*TAST Asynchronous Library*" [Rig02], elles prennent généralement comme paramètres la *base* et le *digit* caractérisant le type MR. Ces informations sont conservées lors de la compilation pour paramétrer les appels d'opérations de la bibliothèque.

Réseau de Petri	Programme en VHDL
<p><math>P_0</math> CTRL[0..0] = "1"[3]</p>	if (CTRL = "010" and P9) then
<p><math>P_{8_1}</math> S1 &lt;= "0"[2]</p>	if (P8_1) then S1 <= "00" ;

La traduction des actions de communications est expliquée dans le paragraphe 3.2.3.5.

### d) Séquentialité

Deux places qui se succèdent expriment une séquentialité entre les actions associées à ces places. En VHDL, le signal associé à la place suivante n'est activé qu'après la fin de l'action sur la place précédente et la désactivation du signal qui lui est associé. Les actions s'exécutent alors de manière séquentielle, à des fronts d'horloge fictive successifs.

Réseau de Petri	Programme en VHDL
<p><math>P_1</math> <math>a1</math> <math>T_i</math> <math>G_i</math> <math>P_2</math> <math>a2</math></p>	if $P_1$ then - traduire en VHDL $a1$ if ( $G_i$ and ( $P_1$ and not $P_2$ )) then $P_1$ <= false ; $P_2$ <= true ; if $P_2$ then - traduire en VHDL $a2$

### e) Concurrence

Une concurrence est modélisée en réseaux de Petri par plusieurs places succédant à une même transition. En VHDL, il suffit d'appliquer la procédure de traduction d'une transition expliquée plus haut.

Toutefois, la concurrence de CHP ne se traduit pas fidèlement en VHDL. Les branches concurrentes évoluent de manière asynchrone, et il est habituel de considérer comme des exécutions distinctes tous les entrelacements des instructions des branches concurrentes. En VHDL, les affectations concurrentes s'exécutent simultanément. Notre traduction réalise une réduction de pré-ordre, pour laquelle il est nécessaire de s'assurer de l'indépendance entre ces affectations concurrentes (§3.3.7).

Notons en outre que nous affectons des *signaux* et non des *variables* dans les processus. Dans la mesure où les signaux affectés sont distincts, l'ordre d'écriture des affectations n'influe pas sur le résultat.

Réseau de Petri	Programme en VHDL
	<pre> ... .. if (Gi and P0) then   P0 &lt;= false ; P1 &lt;= true ; ... Pn &lt;= true ;   traduction en VHDL des places et leurs   actions                     </pre>

**f) Structure de choix**

En réseaux de Petri, les choix peuvent être déterministes ou indéterministes, en VHDL la branche alternative correspondant à la première transition valide est exécutée.

Réseau de Petri	Programme en VHDL
	<pre> ... .. if (G1 and P0) then   P0 &lt;= false ; P1 &lt;= true ;   ... .. elsif (Gn and P0) then   P0 &lt;= false ; Pn &lt;= true ;   traduction en VHDL des places et leurs   actions                     </pre>

**g) Structure de répétition**

Une boucle est exprimée en réseaux de Petri par un retour de transition vers une précédente place. En VHDL, il suffit de réactiver le signal associé à cette place.

Réseau de Petri	Programme en VHDL
	<pre> ... .. if (Gf and Pf) then   Pf &lt;= false ; Pi &lt;= true ;   ... ..                     </pre>

**3.2.3.4. Algorithme de traduction**

Un composant CHP est exprimé en plusieurs réseaux de Petri communicants, la traduction en VHDL de ce composant consiste à associer pour chaque réseau de Petri un processus VHDL. La traduction automatique d'un réseau de Petri en VHDL est développée par l'algorithme de la Figure 38. Concrètement, en première étape on effectue une analyse de la partie déclarative, pour extraire les types, les variables et les ports. Chacun de ces éléments est traduit selon les paramètres du type de base MR "Multi Rail". Généralement un MR est traduit en un Bit-

Vector, les aspects d'interface sont aussi pris en compte (Input et Output). Vient ensuite la traduction de la partie contrôle. Le réseau de Petri est alors analysé place par place et transition par transition, et là on distingue deux types de places : les places avec communication, c'est-à-dire des places qui implémentent le mécanisme de communication poignée de main soit en lecture ou en écriture, et des places atomiques avec de simples affectations. Ces actions de communication, comme ce sera expliqué par la suite, doivent être expansées, nous obtenons alors un réseau de Petri plus gros, qui sera traduit par application de l'algorithme, conformément à la sémantique de traduction défini auparavant (§3.2.3.3).

**Début****Pour chaque Réseau de Petri****Traduction de la partie déclaration**

*Parcourir la partie déclarative pour extraire les types, variables et ports.  
Traduire en premier temps les types. La variable ou le port est déclaré ensuite selon son type et (ou) ses paramètres (Base et Digit).*

**Parcourir le réseau de Petri global (places et transitions)**

**Si** (place Pi) **Alors**

**Si** (place est atomique) **Alors**

associer un signal Pi à la place ;

**Ajouter** : if Pi then action(Pi) , ... end if ;

-- action(Pi) : actions associées à la place.

**Sinon** (la place exprime une attente ou de la communication)

Put\_Com () ;

**Fin Si**

**Si** (transition Ti) **Alors**

**Ajouter** :

if (And[cond(Ti), places\_in(Ti) , not(places\_out(Ti))])

-- cond(Ti) : condition associée à la transition Ti.

Places\_in(Ti) = false ; -- places\_in(Ti) : places entrantes en Ti.

Places\_out(Ti) = true ; -- places\_out(Ti) : places sortantes.

End if ;

**Fin SI**

**Fin du parcours.**

**Fin**

**Put\_com ()**

Implémente l'expansion des actions de communications à effectuer, dans notre cas la lecture ou l'écriture, Read ? Ou Write !. L'expansion est implémentée soit par insertion du Petri net, correspondant à cette action, dans le Petri net global, soit par la traduction directe en VHDL de l'action de communication.

**Figure 38** : Algorithme de traduction de réseau de Petri en VHDL

### 3.2.3.5. Traduction des communications

Dans cette approche de vérification, l'expansion de la communication est nécessaire. En effet, toute place du réseau de Petri contenant une action de communication doit être expansée en accord avec le protocole de communication à quatre phases. Cela signifie que pour chaque action de communication nous obtenons une partie d'un réseau de Petri comme présentée en détail dans le chapitre précédent. Cette partie remplace la place correspondante (voir Figure 39 et Figure 40).

L'expansion des communications introduit l'expansion des ports d'interfaçage des différents processus en requêtes, données et acquittement. Le comportement de certains de ces éléments est indépendant du temps, c'est-à-dire qu'ils s'exécutent à des instants indéterminés. Alors pour chaque partie du réseau de Petri générée pour les besoins de l'expansion d'une action de lecture ou d'écriture, nous avons besoin de modéliser le comportement des signaux ajoutés (les acquittements).

Dans le cas des communications internes à un composant, les actions de communications entre processus se synchronisent automatiquement, une lecture (écriture) sur un port est synchronisée avec une écriture (lecture) sur le même port. Il n'y a alors pas besoin de modéliser le comportement des signaux (données ou acquittements).

### 3.2.3.6. Exemple : Traduction de l'arbitre asynchrone

La traduction automatique de la représentation intermédiaire en réseau de Petri de l'arbitre asynchrone est un programme VHDL fidèle à la spécification initiale. La source de ce programme est présentée ci-dessous. Notons que certains commentaires ont été ajoutés manuellement pour les besoins de la compréhension.

Réseau de Petri de l'Arbitre	Programme VHDL de l'Arbitre
<pre> <b>COMPONENT</b> Arbitrer <b>port</b> E : in BD passive MR[2][1] ; <b>port</b> C : in BD passive MR[3][1] ; <b>port</b> S1, S2 : out BD active MR[2][1] ;  <b>process</b> MAIN <b>port</b> C : in BD passive MR[3][1] ; <b>port</b> E : in BD passive MR[2][1] ; <b>port</b> S1, S2 : out BD active MR[2][1] ;  <b>variable</b> X : MR[2][1] ; <b>variable</b> CTRL : MR[3][1] ;  T1 (1, TRUE, 0, 0) =&gt; P0 P0 (0, 0, C[0..0] ? CTRL[0..0], 2) =&gt; T0 T0 (0, TRUE, 0, 1) =&gt; P9 P9 (9, 0, skip, 1) =&gt; T14, T10, T4 T14 (14, CTRL[0..0] = "0" [3], 0, 1) =&gt; P7 P7 (7, 0, E[0..0] ? X[0..0], 1) =&gt; T13 T13 (13, TRUE, 0, 1) =&gt; P8 </pre>	<pre> <b>entity</b> Arbitrer_Ent is <b>port</b>( C : in bit_vector(2 downto 0) ;       E : in bit_vector(4 downto 0) ;       S1 : out bit_vector(1 downto 0) ;       S2 : out bit_vector(1 downto 0) ;       C_a : out bit ; E_a : out bit ;       S1_a : in bit ; S2_a : in bit ;       clk, rst : in bit) ; <b>end</b> Arbitrer_Ent ;  <b>architecture</b> Arbitrer_a of Arbitrer_Ent is <b>signal</b> X : bit_vector(1 downto 0) ; <b>signal</b> CTRL : bit_vector(2 downto 0) ; <b>signal</b> P0, P9, P7, P8, P1, P5, P6, P2, P3, P4, P0_1, P0_2, P7_1, P7_2, P8_1, P8_2, P5_1, P5_2, P6_1, P6_2, P2_1, P2_2, P3_1, P3_2, P4_1, P4_2 : boolean ;  <b>begin</b> <b>process</b>(clk, rst) <b>begin</b> if (rst='0') then -- bloc d'initialisation   S1 &lt;="00" ; S2 &lt;="00" ; C_a &lt;= '1' ;   E_a &lt;= '1' ; P0 &lt;= true ; <b>elsif</b> clk'event and clk='1' then   -- un wait sur l'horloge fictive   <b>if</b> P8 then S1 &lt;= X ; <b>end if</b> ; </pre>

<pre> P8 (8, 0, S1[0..0] !X[0..0], 1) =&gt; T15 T15 (15, TRUE, 0, 1) =&gt; P1 P1 (1, 0, skip, 3) =&gt; T2 T2 (2, TRUE, 1, 1) =&gt; P0 T10 (10, CTRL[0..0] = "1" [3], 0, 1) =&gt; P5 P5 (5, 0, E[0..0] ?X[0..0], 1) =&gt; T9 T9 (9, TRUE, 0, 1) =&gt; P6 P6 (6, 0, S2[0..0] !X[0..0], 1) =&gt; T11 T11 (11, TRUE, 0, 1) =&gt; P1 T4 (4, CTRL[0..0] = "2" [3], 0, 1) =&gt; P2 P2 (2, 0, E[0..0] ?X[0..0], 1) =&gt; T3 T3 (3, TRUE, 0, 1) =&gt; P3, P4 P3 (3, 0, S1[0..0] !X[0..0], 1) =&gt; T5 T5 (5, TRUE, 0, 2) =&gt; P1 P4 (4, 0, S2[0..0] !X[0..0], 1) =&gt; T5  end ; // process MAIN END ; // COMPONENT Arbiter </pre>	<pre> ... -- débiter l'action d'écriture sur la place P8 ... -- les mêmes instructions pour P6, P3 et P4 if P7_1 then X &lt;= E ; E_a &lt;= '0' ; end if ; -- débiter l'action d'écriture E ?x sur la place P7 -- la place P7 est expansée if ((E="00") and P7_1) then P7_2 &lt;= true ; P7_1 &lt;= false ; -- La requête est désactivée if P7_2 then E_a &lt;= '1' ; end if ; -- acquitter la poignée-de-mains, if (P7_2) then P8 &lt;= true ; P7_2 &lt;= false ; end if ; -- Franchissement de T13, on passe à la place P8 if P8_1 then S1 &lt;="00" ; end if ; ... .. -- La structure de choix au niveau de P9 if ((CTRL="001")andP9) then P7&lt;=true ; P9&lt;=false ; end if ; if ((CTRL="010")andP9) then P5&lt;=true ; P9&lt;=false ; end if ; if ((CTRL="100")andP9) then P2&lt;=true ; P9&lt;=false ; end if ; -- La même chose pour le restes des places et transitions ... .. end if ; end process ; end Arbiter ; </pre>
--	--

### 3.2.4. Vérification de propriétés

La description obtenue après la traduction est conforme au sous-VHDL pour la vérification formelle. Nous pouvons à présent écrire et prouver des propriétés de sûreté ou de vivacité caractérisant le bon comportement du système.

Dans cette étape de vérification nous procédons aux étapes suivantes :

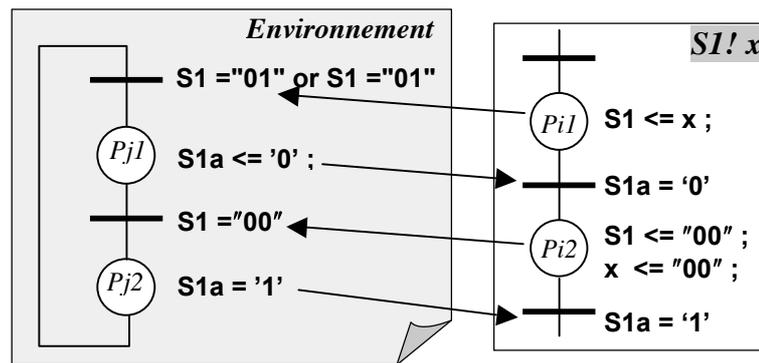
- ✓ Modélisation de l'environnement pour la prise en compte de certains aspects de la méthode de vérification :
  - l'environnement répondra aux requêtes d'entrée du système ;
  - les requêtes d'entrée sont stables ;
  - une contrainte (*reset*), pour effectuer l'initialisation ;
  - une horloge (*clock*) artificielle.
- ✓ Enfin l'évaluation des spécifications attendues du système, ces spécifications sont exprimées dans un langage temporel. La correction du modèle est évaluée à l'aide d'un outil de vérification symbolique de modèles.

#### 3.2.4.1. Modélisation de l'environnement

Une étape de modélisation de l'environnement du circuit est nécessaire, elle nous permettra de s'assurer du bon fonctionnement des canaux d'entrées et de sorties. Cette étape consiste en une définition de quelques propositions, généralement des contraintes d'équité, telles que : les requêtes d'entrée sont stables jusqu'à leur acquittement, l'environnement garantira la présence des requêtes, il répondra aux requêtes du système.

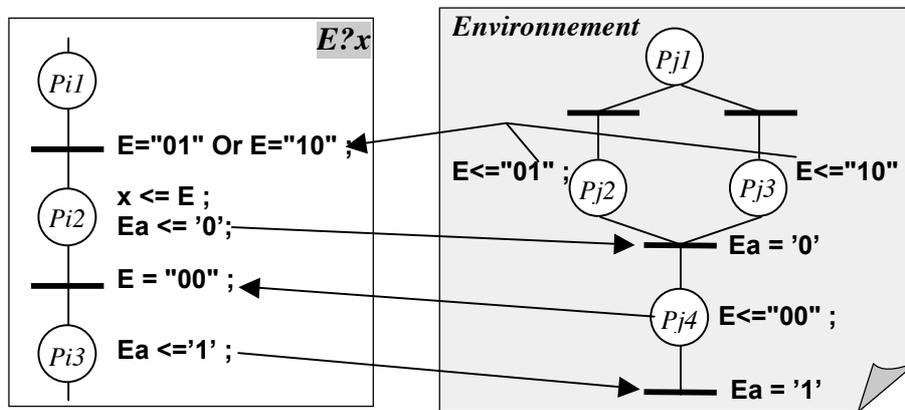
Nous présentons une modélisation de l'environnement pour les ports extérieurs du composant. Cette modélisation, exprimée en réseau de Petri, complètera le déroulement des protocoles de communication. La Figure 39 présente une action d'écriture "S !x" expansée et une modélisation en réseau de Petri de son environnement. Le réseau de Petri proposé constitue une sorte de "bouchon" qui sert à fermer le système. Ce composant ressemble à l'expansion d'une action de lecture, cela s'explique par la dualité entre les deux actions de lecture et d'écriture, l'une est l'environnement de l'autre.

**Action d'écriture : S1 ! X**



**Figure 39 :** Traduction en VHDL d'une action d'écriture et modélisation de son environnement.

**Action de lecture : E ? x**



**Figure 40 :** Traduction en VHDL d'une action de lecture et modélisation de son environnement.

Concrètement, le comportement peut être modélisé comme un système par deux façons : des processus VHDL concurrents ou par des propriétés temporelles.

**a) Par des processus concurrents**

Dans la première méthode, le comportement de l'environnement est modélisé grâce à des processus VHDL qui opèrent en concurrence avec les processus VHDL qui décrivent le comportement du circuit. Après l'écriture des processus d'environnement, on obtient une description VHDL fermée, c'est-à-dire qu'elle modélise le circuit et son environnement.

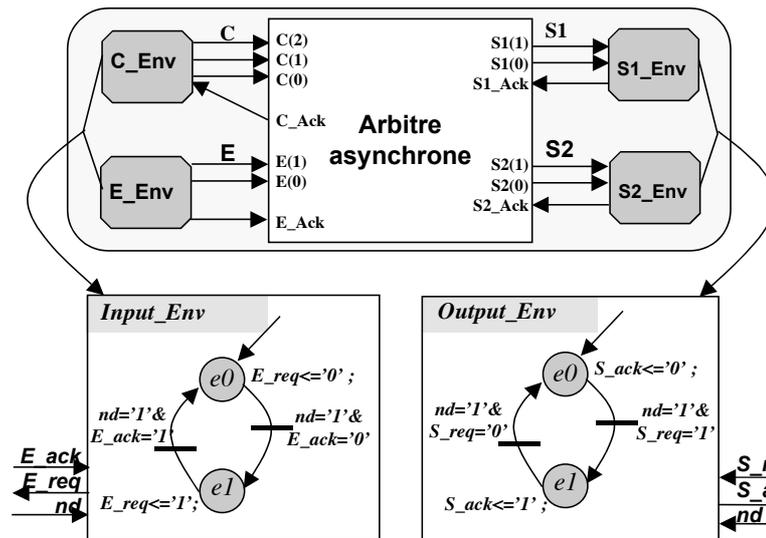


Figure 41 : Modélisation de l'environnement de l'arbitre par processus VHDL concurrents

**b) Contraintes d'équité pour modéliser l'environnement**

L'environnement d'un système peut être aussi modélisé par des propriétés temporelles, ces propriétés expriment généralement le même comportement que celui décrit par les processus VHDL. Cette technique est disponible dans certains outils de vérification comme FormalCheck, il suffit de signifier à l'outil que ces propriétés temporelles sont des contraintes et concernent le comportement de l'environnement.

Examinons les contraintes exprimées pour l'exemple de l'arbitre asynchrone. On distingue des contraintes sur les entrées, des contraintes sur les sorties et des contraintes de stabilité.

Rappelons que C est déclaré en CHP "C : IN MR[3][1] ;" et traduit en VHDL en :

```
C : IN bit_vector(3 downto 0) ;
C_a : OUT bit ;
```

Notons que l'expression «C = x"2"» est équivalente à «C = "010"» et que l'acquiescement est actif bas.

**Contraintes sur les canaux d'entrées**

✓ **C\_Env1**

A chaque fois que la place P0 est active, une requête de contrôle sera certainement (Eventually) envoyée.

```
After P0 = True
Eventually (C = x "1" or C = x "2" or C = x "4") and P0 = True
```

✓ **C\_Env2**

Après (After) l'acquiescement de la requête (C\_a = 0), il est certain que (Eventually) celle-ci sera remise à zéro (C = x"0").

```
After C_a = 0
Eventually C = x "0"
```

**Contraintes sur les canaux de sorties**

✓ **S1\_Env1**

Après l'écriture sur S1 (S1(1) = 1 or S1(0) = 1), il est certain qu'un acquiescement sera envoyé (S1\_a = 0).

After  $(S1(1) = 1 \text{ or } S1(0) = 1)$   
 Eventually  $S1\_a = 0$

✓ **S1\_Env2**

Après la terminaison de l'écriture sur S1 ( $S1 = x''0''$ ), il est certain que l'acquiescement sera désactivé ( $S1\_a = 1$ ).

After  $S1 = x''0''$   
 Eventually  $S1\_a = 1$

**Remarque :** pour les contraintes concernant les deux canaux E et S2, on définit des contraintes similaires aux canaux C et S1.

**Contraintes de stabilité**

En plus des contraintes précédentes, on ajoute des contraintes qui expriment la stabilité des requêtes, une requête est stable (ne change pas de valeur) jusqu'à son acquiescement.

✓ **Stable\_C**

After :  $C = x''1''$  or  $C = x''2''$  or  $C = x''4''$   
 Eventually :  $C = \text{stable}$   
 Unless :  $C\_a = 0$

✓ **Stable\_E**

After :  $E = x''1''$  or  $E = x''2''$   
 Eventually :  $E = \text{stable}$   
 Unless :  $E\_a = 0$

### 3.2.4.2. Ecriture et vérification de propriétés

Dans cette approche, comme dans toute approche de vérification formelle, nous souhaitons naturellement évaluer des propriétés comportementales du système. Nous considérons dans cette partie que ces propriétés sont exprimées dans le langage temporel CTL\* introduit dans le paragraphe 0. Nous présentons un certain nombre de propriétés vérifiées sur notre exemple type d'arbitre asynchrone.

#### a) Propriétés vérifiées sur l'exemple de l'arbitre asynchrone

Concernant l'arbitre asynchrone, les propriétés exprimées sont des propriétés comportementales. Elles correspondent généralement aux branches du réseau de Petri, c'est-à-dire, l'atteignabilité de certaines places ou transitions en suivant un chemin donné.

P1) After  $P0 = \text{True}$  and  $C = x''1''$

Eventually :  $S1(1) = 1$  or  $S1(0) = 1$

Cette première propriété P1 veut dire que : si le système se trouve à la place P0 et qu'il reçoit une entrée  $C = x''1''$  alors, il est certain qu'il y aura une écriture sur S1 ( $S1(0)=1$  Or  $S1(1)=1$ ).

P2) After  $P0 = \text{True}$  and  $C = x''2''$

Eventually :  $S2(1) = 1$  or  $S2(0) = 1$

P3) After :  $P0 = \text{True}$  and  $C = x''4''$

Eventually :  $(S1(1) = 1 \text{ or } S1(0) = 1)$  and  $(S2(1) = 1 \text{ or } S2(0) = 1)$

### 3.2.5. Vérification du circuit asynchrone après synthèse

Le long des différentes étapes de synthèse, des erreurs peuvent intervenir produisant une description erronée du circuit. Il est alors important de procéder à une vérification du circuit obtenu après la synthèse. Lors des étapes du processus de synthèse, des techniques d'optimisation et de pipeline sont exécutées. Le circuit obtenu après synthèse ne peut donc être prouvé équivalent à sa spécification initiale. Dans certains cas, l'ordre des actions n'est pas nécessairement préservé. Par exemple, nous considérons un seul processus VHDL pour chaque processus CHP, alors que dans le circuit après synthèse, le comportement souhaité par ce processus lors de la spécification est exprimé en plusieurs processus concurrents. L'approche de vérification formelle que nous introduisons à ce niveau consiste essentiellement à prouver la préservation de certaines propriétés essentielles de sûreté prouvées correctes pour la spécification.

Le circuit synthétisable est décrit dans un standard VHDL pour la synthèse qui n'est pas toujours conforme au sous-VHDL pour la vérification formelle. L'outil de synthèse est basé sur une bibliothèque dédiée à la synthèse asynchrone, alors nous avons procédé à une adaptation (transformation) des éléments de cette bibliothèque pour les rendre conformes au sous-VHDL pour la vérification formelle.

Les portes Muller-C[Rig02] qui sont des composants fréquemment utilisés dans la synthèse contiennent des délais et des *latches*, on les change par des flip-flops, en introduisant une horloge fictive, comme décrit dans la Section 3.2.1. Tous les types sont transformés en types de base : Boolean, Bit et Bit-Vector.

Les transformations préservent la sémantique comportementale du code VHDL : les flip-flops modélisent exactement le comportement des latches, et remplacer les types STD\_ULOGIC en Bit ne change rien dans la sémantique d'exécution du programme VHDL.

#### 3.2.5.1. Application à l'exemple type d'arbitre asynchrone

Prenons quelques composants de l'arbitre synthétisé. Nous avons effectué certaines transformations pour le rendre conforme au sous-ensemble VHDL pour la vérification formelle. Ci-dessous les modifications sur une partie du code VHDL résultat de la synthèse asynchrone :

- ✓ Le type "STD\_ULOGIC" est remplacé par le type "bit".
- ✓ Le signal "clk" est déclaré et la structure "wait until clk = '1'" est ajoutée dans le modèle VHDL transformé pour modéliser en *pseudosynchrone*.
- ✓ Les opérateurs de délais sont simplement éliminés
- ✓ Les structures

```
signal <= val1 when cond1 else
      val2 when cond2 else
```

... ..

sont transformées dans la structure suivante :

```
if cond1 then signal <= val1 ;
elsif cond2 then signal <= val2 ;
```

... ..

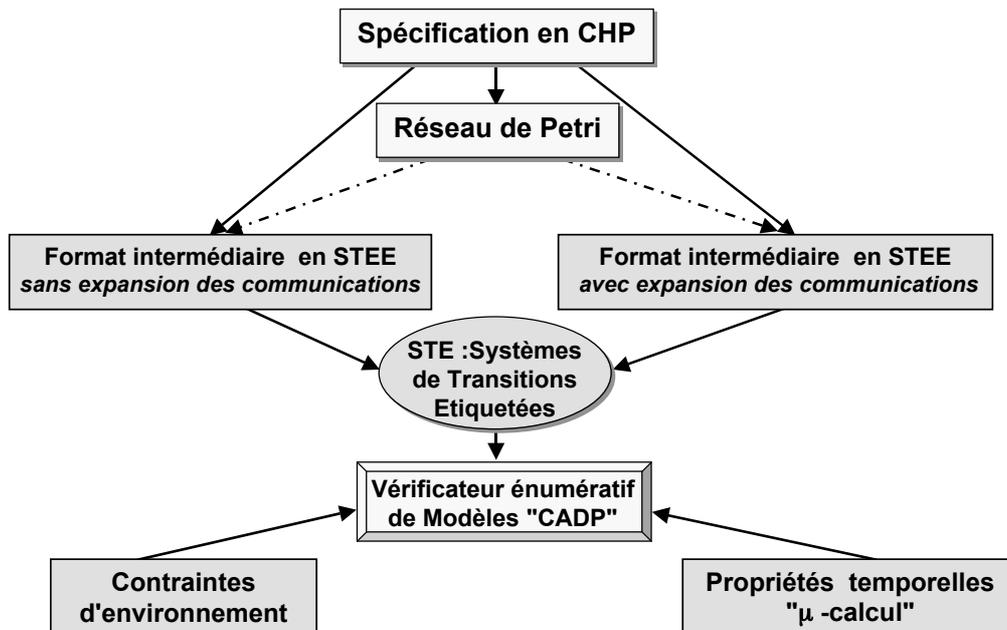
```
end if ;
```

<pre> entity MULLER2_R is port ( resetb : in STD_ULOGIC ;       S      : out STD_ULOGIC ;       A, B : in STD_ULOGIC) ; end MULLER2_R ;  architecture behaviour of MULLER2_R is signal s_s : STD_ULOGIC ; begin  S&lt;= s_s ; s_s &lt;= '0' after 1 ns when resetb = '0' else '1' after 1 ns when a='1' and B='1' else '0' after 1 ns when a='0' and b='0' else s_s ;  end behaviour ; </pre> <p style="text-align: center;"><b>VHDL après synthèse</b></p>	<pre> entity MULLER2_R is port ( resetb, clk : in bit ;       S      : out bit ;       A, B      : in bit) ; end MULLER2_R ;  architecture behaviour of MULLER2_R is signal s_s : bit ; begin S&lt;= s_s ; process begin <b>wait until clk = '1' ;</b> if resetb = '0' then s_s &lt;= '0' ; elsif a='1' and B='1' then s_s &lt;= '1' ; elsif a='0' and b='0' then s_s &lt;= '0' ; end if ; end process ;  end behaviour ; </pre> <p style="text-align: center;"><b>VHDL transformé pour la vérification formelle</b></p>
---	--

**Figure 42 :** Le modèle VHDL pour les portes Muller-C pour la vérification formelle.

### 3.3. Deuxième approche : Modélisation purement asynchrone et vérification énumérative

Dans cette deuxième approche, nous nous focalisons sur la validation des spécifications décrites en CHP. La figure suivante présente le flot de vérification formelle, les spécifications asynchrones sont alors modélisées par un format intermédiaire de systèmes de transitions étiquetées étendu "STEE". La vérification est enfin effectuée en utilisant des techniques énumératives de vérification de modèles.



**Figure 43 :** Modélisation purement asynchrone et vérification formelle énumérative

Nous intervenons aussi après l'expansion des communications, ce qui permet de comparer les spécifications des circuits asynchrones avant et après expansion des communications. C'est une façon aussi de valider la correction du protocole de communication.

Dans la suite de cette section, nous présenterons brièvement la description générale de cette méthode. Nous introduisons par la suite les méthodes de vérification énumératives utilisées dans la présente approche. Enfin, nous présenterons plus en détail le processus de vérification avant et après expansion des communications.

#### 3.3.1. Description de la méthode

Dans cette méthode, le processus de vérification est composé des étapes suivantes :

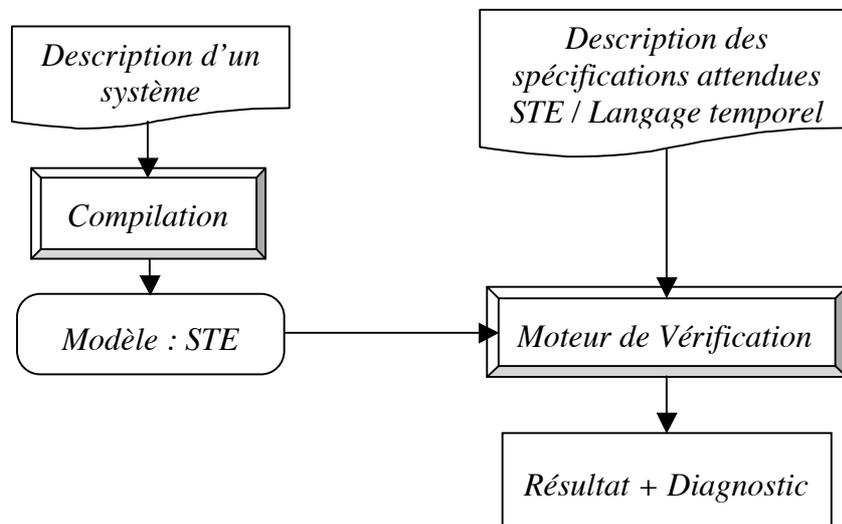
- ✓ La première étape constitue le noyau de cette deuxième approche, elle consiste en la construction d'un système de transitions étiquetées étendu (STEE) correspondant à notre spécification du circuit. Cette représentation est entièrement construite sur la base de la sémantique du langage CHP en termes de STEE que nous avons définie dans le chapitre 2.
- ✓ Une modélisation de l'environnement, est réalisée par des STEE concurrents.
- ✓ Génération à partir du système et de son environnement d'un STE représentant toutes les exécutions possibles.

- ✓ L'expression de propriétés temporelles de comportement exprimées dans le langage temporel  $\mu$ -calcul. Ces propriétés seront vérifiées, sur le STE de base, par un outil de vérification énumérative de modèles.

Ce même processus de vérification est effectué après l'expansion des communications. A ce niveau, nous effectuons les mêmes étapes tout en prenant en considération à chaque étape que les communications sont expansées.

### 3.3.2. Techniques de vérification énumérative de modèles

Dans les méthodes de vérification à base de techniques d'énumération exhaustive, les modèles considérés sont des programmes concurrents. Ces programmes sont exprimés dans un langage possédant une sémantique opérationnelle bien définie, tels que les langages basés sur les algèbres de processus (CSP [Hoa78], CCS [Mil80], CHP[Mar90], Meije [Bou85]) et aussi certaines techniques de description formelle normalisées (Estelle [ISO88], Lotos [ISO87, EVi89]).



**Figure 44 :** Environnement de vérification comportementale énumérative

Concrètement, comme décrit dans la Figure 44, la vérification de spécifications comportementales nécessite deux types d'outils :

- ✓ Un compilateur, dont le rôle est de traduire le programme à vérifier en un système de transitions étiquetées (STE) qui modélise l'ensemble de ses exécutions possibles,
- ✓ Un moteur de vérification, qui permet de comparer le système de transitions étiquetées représentant le programme avec ses spécifications. Ce moteur de vérification doit également fournir des éléments de diagnostic pour examiner le résultat en cas d'échec de la vérification.

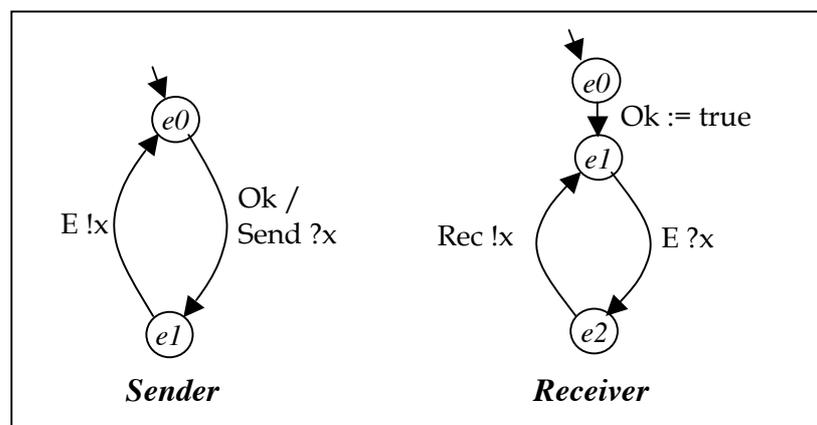
L'objectif de la vérification formelle basée sur les modèles est de déterminer si certaines formules, spécifiant le bon fonctionnement du système, sont correctes. La tâche de vérification consiste alors à évaluer la correction de ses spécifications, exprimées en logiques temporelles ou en système, sur le modèle en question.

Généralement, la méthode d'évaluation opère de la manière suivante : le compilateur génère à partir de la description du système une représentation explicite du modèle décrite en STE. Les spécifications décrivant des propriétés comportementales et exprimées en logiques

temporelles sont vérifiées sur le modèle par un évaluateur. Cet évaluateur devrait être capable, selon la complexité du modèle à vérifier, de fournir une réponse sur la véracité des propriétés accompagnées, le cas échéant, d'un diagnostic.

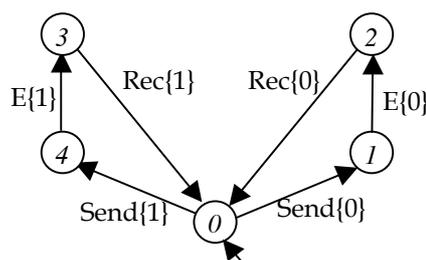
La construction du modèle global d'exécution est la tâche la plus importante, le modèle est alors généré une seule fois et il ne dépend pas des propriétés à vérifier. Ce qui explique le fait que les efforts, en termes de performance et d'optimisation, se focalisent sur la phase de compilation. L'évaluation dépend du type de logique temporelle utilisée, les propriétés, comme nous allons le décrire par la suite, peuvent être exprimées en différents types de langage temporel.

Prenons un simple exemple de communication, on dispose de deux sous-STEES : un système "Sender" qui envoie une donnée, et un autre système "Receiver" qui la reçoit. Les deux systèmes se synchronisent sur la porte E. La variable Ok sert à l'initialisation.



**Figure 45 :** Exemple de deux systèmes concurrents "Sender || Receiver"

La Figure 46 est un STE qui modélise l'ensemble des exécutions possibles. Ce STE est construit par le composant de l'environnement de vérification chargé de la compilation.



**Figure 46 :** Le STE modélisant toutes les exécutions possibles

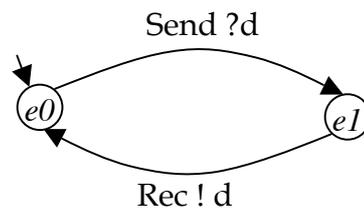
Notons que les techniques et modèles de vérification énumérative ont fait l'objet de nombreux travaux de recherches. Ces travaux s'intéressent plus particulièrement aux techniques et algorithmes pour éviter le problème de l'explosion d'états, plusieurs techniques ont été proposées : vérification à la volée [FJJ92, Mat03], techniques d'abstraction [AAB03, GLo93], techniques de réduction [GWo94, HPu99], vérification modulaire ou hiérarchique [Pnu85, TLG03] ...etc

### 3.3.2.1. Expression de spécifications attendues

Les propriétés attendues d'un système en cours de vérification peuvent être exprimées sous deux formes :

- ✓ Dans la première approche, les propriétés sont modélisées sous la forme d'un programme tel qu'un système de transitions étiquetées par exemple. Cette approche consiste à caractériser le comportement attendu d'un système  $S1$  par un autre système plus abstrait  $S2$ . Le comportement de  $S1$  est vérifié comme étant le même comportement que celui de  $S2$  grâce à des relations d'équivalence ou de préordre, comme la bisimulation forte [Par81], l'équivalence observationnelle [Mil89], l'équivalence de branchement [GWe89], etc.

Le comportement souhaité pour l'exemple de la Figure 45 peut être exprimé par le STE de la Figure 47.



**Figure 47 :** Comportement souhaité de la communication

- ✓ La deuxième approche, évoquée dans la Section 3.2.2.4, consiste à écrire les propriétés attendues en tant que formules de logique temporelle. La spécification attendue est alors exprimée en formule temporelle écrite dans un langage spécifique. Les spécifications logiques permettent généralement une meilleure caractérisation des propriétés caractéristiques d'un système. En effet, il est souvent plus aisé de traduire les propriétés attendues, exprimées informellement en langage naturel, sous forme de formules de logique temporelle que sous forme de programmes. Nous introduisons brièvement le langage temporel utilisé dans notre méthode dans la Section 3.3.2.2.

En pratique, parmi l'ensemble de propriétés que l'on désire vérifier, certaines peuvent facilement être représentées comme une abstraction du comportement attendu, alors que d'autres s'expriment mieux dans un formalisme plus déclaratif, comme les logiques temporelles. Ce qui nous permet de dire que ces deux approches peuvent être complémentaires.

L'évaluation des formules temporelles pour chaque état est efficace. Par contre, la construction et la mémorisation du STE qui modélise le système à vérifier constitue le goulot d'étranglement : dès qu'on s'intéresse à des systèmes relativement complexes, la taille limite est rapidement atteinte. Néanmoins, ce problème dit "*problème de l'explosion d'états*" est commun à toutes les méthodes de vérifications énumératives basées sur les modèles.

### 3.3.2.2. Le langage temporel $\mu$ -calcul

Dans cette deuxième approche de vérification, la logique temporelle que nous considérons est le langage  $\mu$ -calcul [Koz83]. Le  $\mu$ -calcul permet d'exprimer : des logiques temporelles arborescentes comme CTL [CEm89] ou ACTL [NVa90] et des logiques régulières comme PDL [FLa79] ou PDL-delta [Str82].

On distingue deux types de formules : **A** : formule d'action et **F** : formule d'état. Les formules sont interprétées pour un STE  $\langle S, A, T, s_0 \rangle$ , où :  $S$  est l'ensemble d'états,  $A$  l'ensemble d'actions (étiquettes de transition),  $T$  est la relation de transition, et  $s_0$  est l'état initial. Une transition  $(s_1, a, s_2)$  de  $T$  indique que le programme dont le STE a été produit peut se déplacer de l'état  $s_1$  à l'état  $s_2$  en effectuant l'action  $a$ .

**Formule d'action** : c'est une formule logique d'action et d'opérateurs booléens, selon la grammaire :

$$A ::= a \mid true \mid \neg A \mid A_1 \wedge A_2$$

- ✓  $a$  est une chaîne de caractères qui dénote une étiquette du STE.
- ✓ Une étiquette de transition du STE satisfait  $a$  si elle est identique à la chaîne de caractères correspondante.

**Formule d'état** : une formule logique construite selon la grammaire ci-dessous :

$$F ::= p \mid true \mid Y \mid \neg F_1 \mid F_1 \wedge F_2 \mid \langle A \rangle F_1 \mid \mu Y.F_1$$

Telle que :  $A$  est une formule d'action,  $\langle A \rangle$  est l'opérateur de possibilité,  $\mu Y.F$  est un opérateur de point fixe, et  $Y$  une variable propositionnelle.

- ✓ Un état du STE satisfait toujours  $true$  ; il ne satisfait jamais  $false$  ;
- ✓ Un STE satisfait  $\langle A \rangle F$  s'il existe au moins une séquence de transition commençant à cet état et menant par l'étiquette satisfaisant  $A$  à un état satisfaisant  $F$  ;
- ✓ Un état satisfait  $\mu Y.F$  s'il appartient à la solution minimale de l'équation de point fixe  $Y = F(Y)$ , où la variable  $Y$  dénote un ensemble d'états de STE.
- ✓ Un STE satisfait une formule d'état  $F$  si son état initial  $s_0$  satisfait  $F$ .

D'autres opérateurs sont déduits des opérateurs primitifs définis ci-dessus :

- ✓  $[A] F = \neg \langle A \rangle \neg F$  est l'opérateur de nécessité, un état du STE satisfait  $[A] F$  si toutes les transitions étiquetées, par des étiquettes satisfaisants  $A$ , mènent à des états satisfaisant  $F$  ;
- ✓  $\eta Y.F = \neg \mu Y. \neg F \neg Y$  est le plus grand point fixe, un état satisfait  $\eta Y.F$  s'il appartient à la solution maximale de la même équation ;

Dans la suite de ce document, en raison de l'utilisation du  $\mu$ -calcul dans des outils particuliers, nous adopterons les extensions sur les expressions régulières et la syntaxe de l'outil « Evaluator » (§ 6.2.3.2), que nous rappelons ci-dessous :

	$\mu$ -calcul	Langage d'entrée d'Evaluator
Le plus petit point fixe	$\mu$	nu
Le plus grand point fixe	$\eta$	mu
Opérateurs sur les expressions régulières	Non disponible	Concaténation "." ; Choix " " ; Fermeture transitive "+"; Fermeture transitive et réflexive "*";

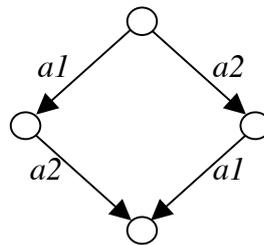
### 3.3.2.3. Sémantique des actions parallèles

Les systèmes asynchrones se comportent de manière concurrente et communiquent à travers des rendez-vous ou par échange de messages. Le comportement de chaque composant est

décrit par un ensemble d'actions. Dans le processus de vérification formelle, ces actions obéissent à certaines hypothèses :

- ✓ *Atomicité* : à un certain niveau d'abstraction, une action est considérée comme étant atomique. Elle ne peut par conséquent être décomposée en actions plus élémentaires ;
- ✓ *Non-simultanéité* : dans le cas d'actions parallèles on ne peut observer une exécution simultanée de ces actions.

La sémantique comportementale associée à un système asynchrone est principalement basée sur l'entrelacement des comportements de ses sous-systèmes. La sémantique d'entrelacement (*interleaving en anglais*) est basée sur les hypothèses d'atomicité et de *non-simultanéité*. La Figure 48 présente l'entrelacement entre deux actions parallèles ( $a1$  et  $a2$ ). Les deux actions  $a1$  et  $a2$  sont simultanément exécutables, le comportement du système considère deux séquences  $a1a2$  et  $a2a1$  produites par l'entrelacement entre  $a1$  et  $a2$ .



**Figure 48** : Entrelacement de deux actions ( $a1$  et  $a2$ )

Une autre sémantique du parallélisme, en plus de la sémantique d'entrelacement que nous considérons, est utilisée dans les langages synchrones. Cette sémantique est basée sur la simultanéité des actions, plusieurs actions peuvent alors se produire en même temps. Nous aborderons ce point dans la fin de ce chapitre.

### 3.3.3. Traduction des spécifications CHP vers des STEE

La traduction des spécifications CHP vers des STEE a fait l'objet du développement d'un compilateur. Ce compilateur est entièrement basé sur la sémantique du langage CHP en termes de systèmes de transitions étiquetées étendus développé en détails dans le chapitre 2.

Dans ce paragraphe, nous illustrons cette étape du processus de validation de spécifications asynchrone sur notre exemple type d'arbitre asynchrone. Rappelons que dans la sémantique d'exécution d'un réseau de Petri les actions sont associées aux places et les conditions de propagation aux transitions, alors que dans IF ce sont des commandes gardées, c'est-à-dire que les actions et les conditions de propagation sont associées aux transitions.

La Figure 49 est une partie du réseau de Petri de l'arbitre asynchrone ( $P9$  ;  $T14$  ;  $P7$ ). On attend sur  $P9$  la condition ( $T14$  :  $ctrl = "0"$ ) pour passer à  $P7$  et exécuter  $E ?x$ . Cette partie peut être traduite en IF de la façon suivante : de la place  $t14$ , on attend la condition  $ctrl = "0"$  pour exécuter  $E ?x$  et passer à la place suivante.

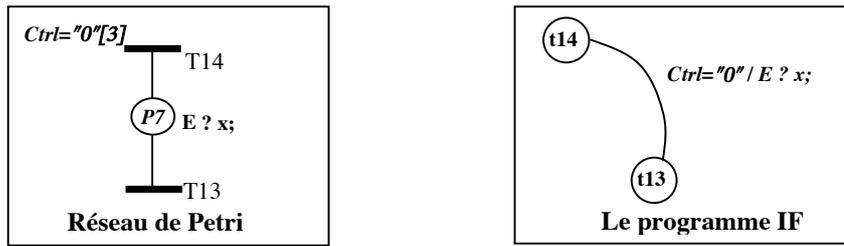


Figure 49 : un morceau du réseau de Petri de l'arbitre et de sa représentation en IF.

Nous remarquons que les places du programme IF correspondent aux transitions du réseau de Petri. Les actions et les conditions de franchissement sont associées aux transitions.

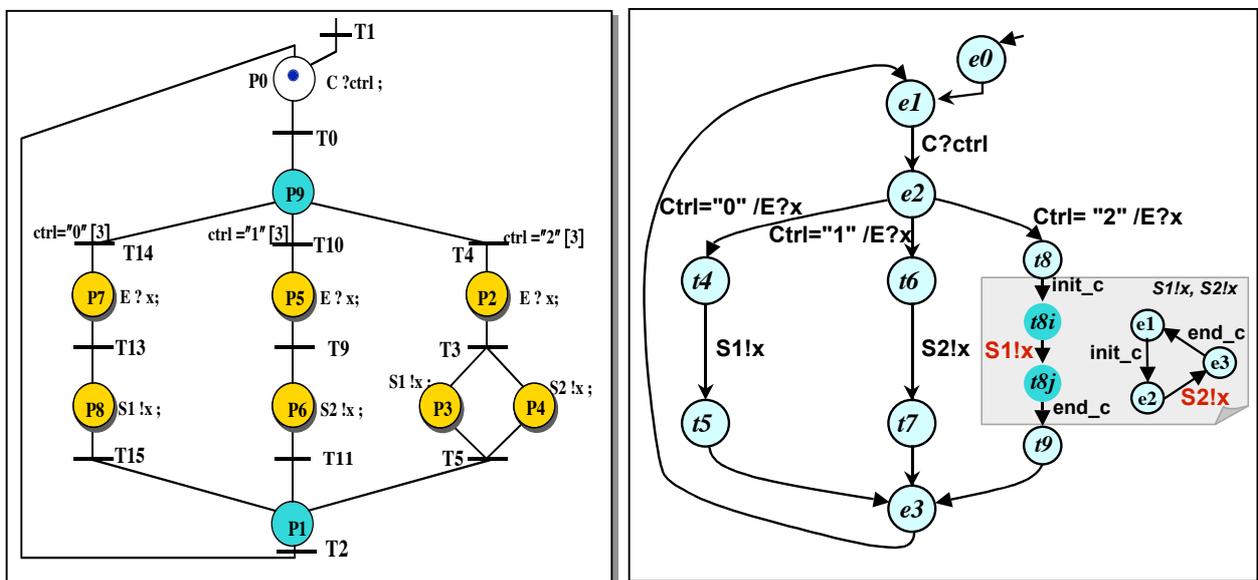
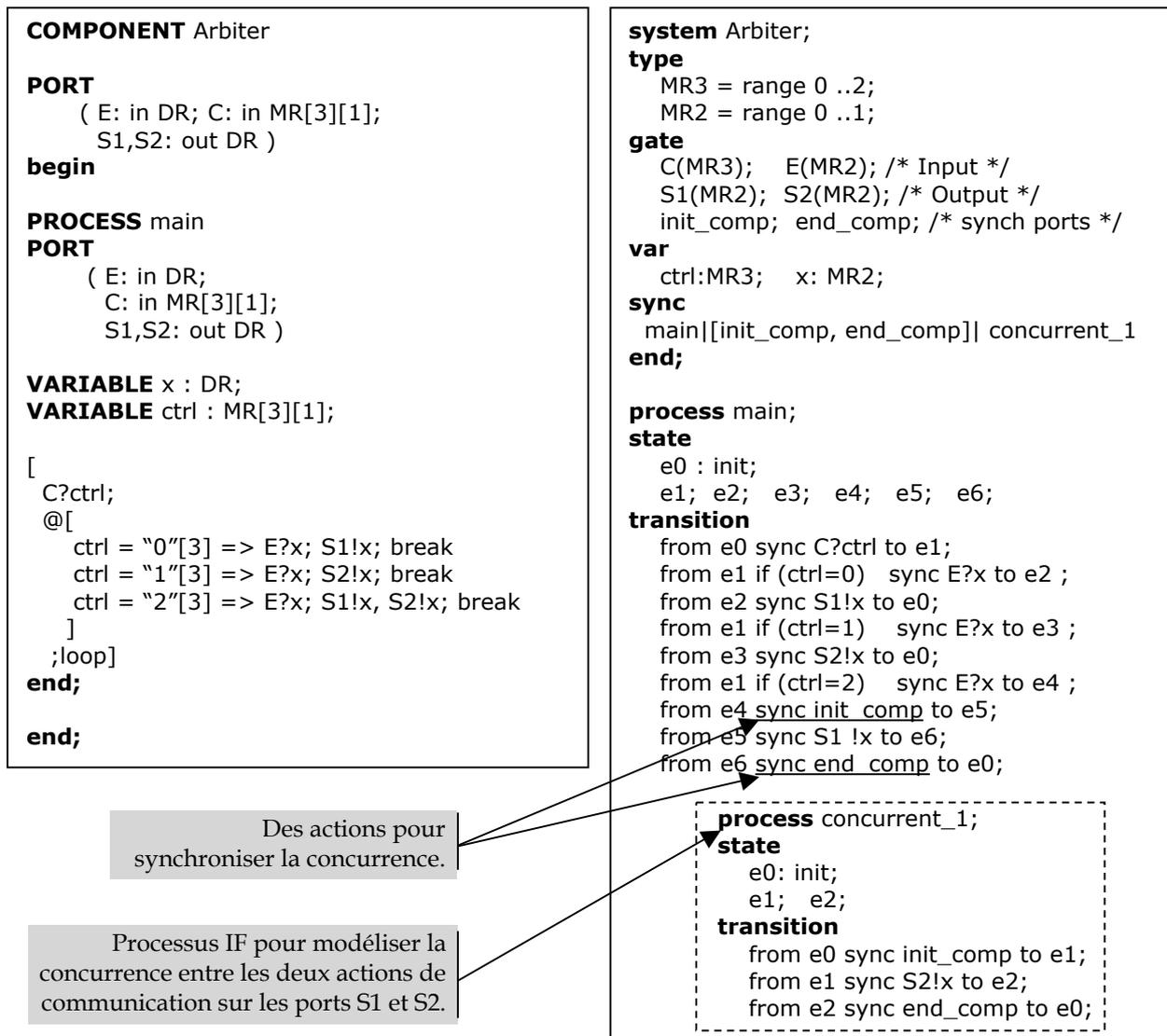


Figure 50 : Le réseau de Petri et le format IF de l'arbitre asynchrone

Le résultat du traitement par notre compilateur CHP2IF de la spécification en CHP est un système IF. Ci-dessous la représentation en réseaux de Petri de la spécification CHP et le graphe du système IF, résultat de la traduction.

Dans la Figure 51, on remarque que la traduction de la concurrence entre les deux actions d'écritures (S1 !x et S2 !x) nécessite la création d'un nouveau processus (concurrent\_1) et l'ajout de deux ports de synchronisation.



**Figure 51 :** Traduction de l'arbitre asynchrone dans le format IF

### 3.3.4. Vérification de propriétés

Une fois que le STEE est construit de manière automatique à partir de la spécification CHP, nous procédons à la phase de vérification.

Dans cette phase, nous distinguons principalement trois étapes :

- ✓ Une étape nécessaire de modélisation de l'environnement ;
- ✓ Générer ensuite à partir du STEE correspondant à la spécification à vérifier et à l'aide d'un compilateur spécifique, un STE contenant explicitement toutes les exécutions possibles.
- ✓ Enfin exprimer dans le langage temporel  $\mu$ -calcul des propriétés spécifiques au bon comportement de la spécification. La valeur de vérité de ces propriétés est évaluée à l'aide d'un outil énumératif de vérification de modèles.

#### 3.3.4.1. Modélisation de l'environnement

Dans cette étape, il s'agit de prendre en compte les conditions spécifiques à l'environnement où le circuit est amené à fonctionner. Généralement cela concerne le comportement souhaité des entrées qui peuvent être, dans certains cas, influencées par les valeurs de sorties. Les

contraintes de l'environnement peuvent, à titre d'exemple, concerner les séquences des entrées, l'indéterminisme d'une entrée ou une réaction aux valeurs de sortie. Ces contraintes peuvent être exprimées ou spécifiées soit par des systèmes, soit par des formules de logiques temporelles.

La modélisation de l'environnement peut ne pas être nécessaire, comme c'est le cas dans notre exemple de l'arbitre asynchrone. Un exemple où la modélisation de l'environnement est indispensable est donné dans la Section 3.3.6. Cet exemple traite de la vérification des spécifications asynchrones après expansion des communications.

### 3.3.4.2. Écriture et vérification de propriétés temporelles

Après la modélisation de l'environnement, nous générons à partir du nouveau système (STEE + environnement) son modèle d'exécution. La vérification est alors effectuée sur ce modèle. Le système obtenu n'est que le STE qui décrit explicitement toutes les exécutions possibles de la spécification.

Les propriétés que nous souhaitons vérifier sont exprimées dans le langage temporel  $\mu$ -calcul (voir Section 3.3.2.2). Les opérateurs des propriétés sont les étiquettes du STE de base. Une bonne compréhension de la signification de ces étiquettes est donc essentielle.

En raison du manque de convivialité du  $\mu$ -calcul, il arrive parfois qu'une propriété s'avère fausse non pas à cause d'une erreur de spécification mais plutôt parce que la propriété elle-même est mal écrite.

#### a) Etape de pré-vérification : Analyse des étiquettes du STE résultant

Une première étape de la vérification consiste à vérifier la présence, dans le STE résultat de la génération, de toutes les étiquettes qui sont supposées être exécutées, c'est-à-dire, faisant partie d'un chemin d'exécution possible de la spécification. Il suffit de constater l'absence, de l'ensemble des étiquettes, d'au moins une étiquette pour en déduire la présence d'erreurs dans la spécification.

L'ensemble des étiquettes qui figurent dans le STE de l'arbitre asynchrone :

"i", "C{0}", "C{1}", "C{2}", "E{0}", "E{1}", "S1{0}", "S2{0}", "S1{1}", "S2{1}"

Lors de la *pré-vérification*, si on constate que l'une des étiquettes citées ci-dessus n'est pas présente dans STE, il y a alors forcément une erreur dans la spécification de l'arbitre.

#### b) Vérification de quelques propriétés sur l'arbitre asynchrone

Nous exprimons les mêmes propriétés que celles qui ont été vérifiées avec la première approche. Ces propriétés correspondent à l'atteignabilité de certains états ou transitions en suivant un chemin donné dans le STEE.

Prop.	Écriture en $\mu$ -calcul	Temps de vérif.
P1	$\text{nu } X . (<\text{true}> \text{true and } [\text{true}]X)$	< 1 sec
P2	$[\text{true}^*]([C\{0\}] (\text{mu } X . (<\text{true}> \text{true and } [\text{not } (S1\{0\} \text{ or } S1\{1})]X)))$	< 1 sec
P3	$[\text{true}^*]([C\{1\}] (\text{mu } X . (<\text{true}> \text{true and } [\text{not } (S2\{0\} \text{ or } S2\{1})]X)))$	< 1 sec
P4	$[\text{true}^*]([C\{2\}] (\text{mu } X . (<\text{true}> \text{true and } [\text{not } (S1\{0\} \text{ or } S1\{1}) \text{ and } (S1\{0\} \text{ or } S1\{1})]X)))$	< 1 sec

P1 exprime l'absence d'inter-blocage.

P2 exprime : si on reçoit une entrée  $C\{0\}$  alors, il est certain qu'il y aura une écriture sur  $S1$  ( $S1\{0\}$  or  $S1\{0\}$ ).

P3 exprime : si on reçoit une entrée  $C\{1\}$  alors, il est certain qu'il y aura une écriture sur  $S2$  ( $S2\{0\}$  or  $S2\{0\}$ ).

P4 exprime : si on reçoit une entrée  $C\{2\}$  alors, il est certain qu'il y aura deux écritures parallèles sur  $S1$  et sur  $S2$ .

### 3.3.5. Vérification par réduction

Certaines spécifications peuvent être vérifiées grâce à une technique de vérification différente de l'évaluation de propriétés temporelles. Cette technique est basée sur la théorie des relations d'équivalence et de préordre entre systèmes de transitions étiquetées. Ces relations sont utilisées dans le cadre de la vérification des systèmes parallèles, pour lequel elles offrent un intérêt particulier[Mou92, Fer89]. Dans notre travail, nous avons étudié l'adéquation et l'application de ces techniques de vérification pour la validation des spécifications de circuits asynchrones.

#### 3.3.5.1. Relations de bisimulation

La bisimulation est une relation d'équivalence entre systèmes de transitions étiquetées. Deux STE  $S1$  et  $S2$  se bisimulent s'ils représentent des comportements identiques : à toute évolution par une séquence d'actions du système  $S1$  correspond une évolution par la même séquence d'actions pour  $S2$ . Un certain nombre de relations d'équivalence sont issues de la bisimulation par des critères d'abstraction et qui sont généralement dédiées à la vérification formelle. Des relations comme la bisimulation forte, l'équivalence observationnelle, la bisimulation de branchement et la  $\tau^*$ -bisimulation sont présentées dans [Gla90]. Elles correspondent chacune à des critères d'abstraction différents.

#### 3.3.5.2. Application à la vérification

Plusieurs travaux de recherches se sont intéressés à l'utilisation des techniques de bisimulation pour la vérification des STE[Fer88, FM91, FKM93, PLM03]. Ces travaux ont proposé de nouvelles relations de bisimulation, et surtout ils ont proposé des algorithmes efficaces de réduction et de comparaison des STE. Car en pratique une relation de bisimulation n'est utile que s'il lui est associé un algorithme efficace de comparaison ou de réduction[Mou92].

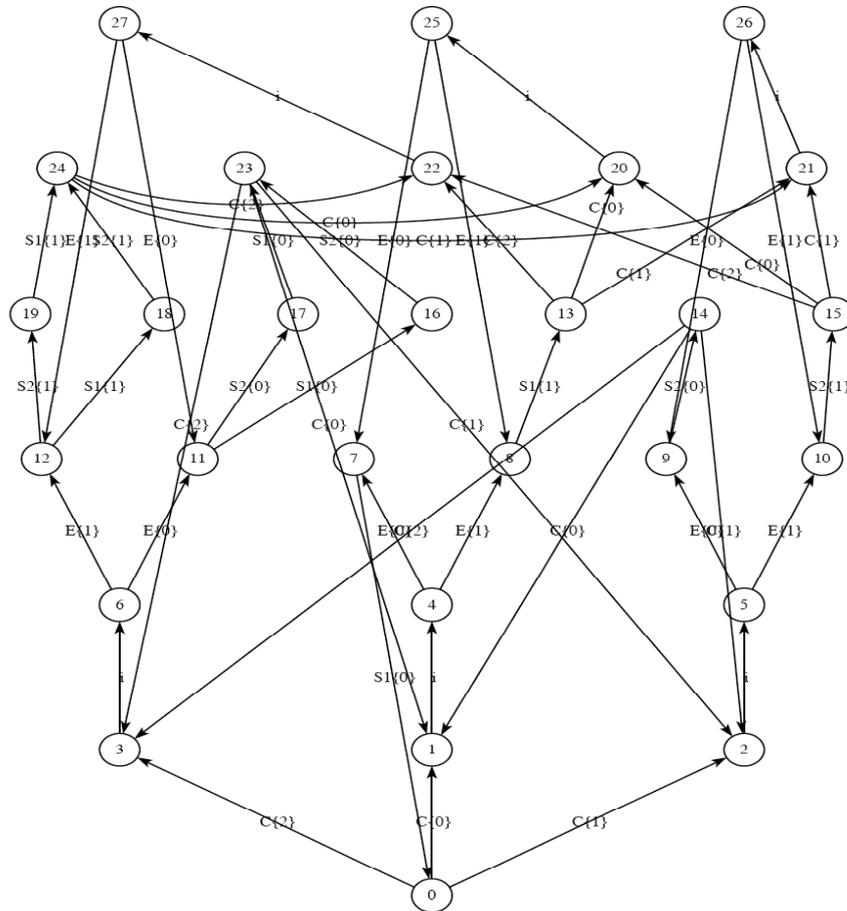
La stratégie de vérification basée sur les techniques de bisimulation que nous proposons consiste en trois étapes :

- ✓ Une étape d'abstraction : généralement consiste à cacher (remplacer par une  $\tau$  action) un certain nombre d'étiquettes relatives à des variables ou ports qui n'influence pas la propriété à vérifier.
- ✓ Une étape de réduction, qui utilise une relation de bisimulation préservant la propriété en question [Mou92].
- ✓ Comparer le STE obtenu avec la spécification de la propriété à vérifier.

Finalement, notons que le choix de relation d'équivalence dépend de la nature des propriétés que l'on souhaite vérifier. Il est très intéressant, d'obtenir une description intuitive du comportement attendu en effectuant des abstractions sur les aspects du programme qui ne sont pas considérés par la vérification.

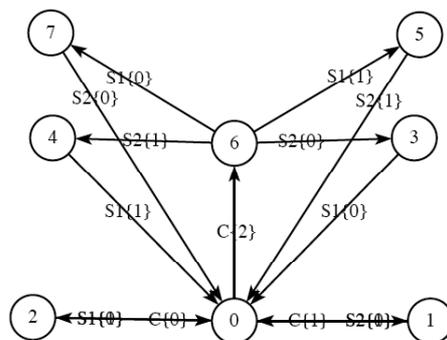
**a) Application à l'arbitre asynchrone**

Nous avons appliqué cette stratégie de vérification sur notre exemple d'arbitre asynchrone. La Figure 52 présente le STE qui modélise toutes les exécutions possibles de l'arbitre.



**Figure 52 :** STE : Les exécutions possibles de l'arbitre.

L'objectif est d'obtenir, par application des techniques de réduction, le STE décrivant le comportement attendu du système à un certain niveau d'abstraction. Nous avons donc caché les actions sur le port d'entrée E, puis le STE est réduit par application de la technique de bisimulation de branchement. Nous obtenons le STE de la Figure 53 dans lequel nous observons un comportement satisfaisant les propriétés exprimées et vérifiées dans la Section 3.3.4.2.



**Figure 53 :** STE de l'arbitre après réduction

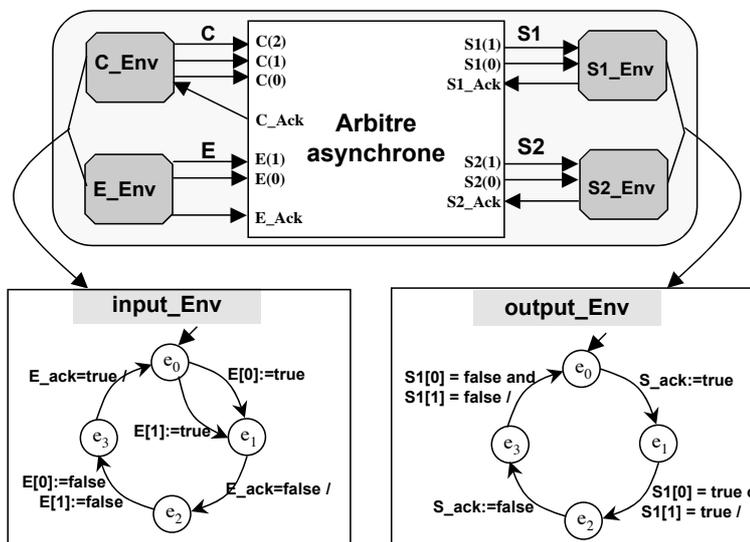
### 3.3.6. Vérification de propriétés après expansion de la communication

La vérification peut intervenir à un niveau plus bas, après la phase d'expansion des communications suivant un protocole de communication. Cela nous permettra de vérifier la correspondance des deux niveaux : avant et après expansion de la communication.

A ce niveau de représentation, la traduction des programmes CHP se base évidemment sur la sémantique de spécification définie dans la chapitre 2, mais elle se différencie légèrement de la méthode de traduction appliquée avant l'expansion des communications. En effet, l'expansion des communications, autrement dit l'explicitation du protocole de communication à quatre phases pour chaque action de communication induit la prise en compte de nouveaux paramètres : nouvelles correspondances pour les types, ajouts de signaux d'acquittements, et un codage quelquefois différent. Les canaux d'entrée et de sortie de la spécification CHP ne sont plus représentés par des ports de type *intervalles d'entier*, ils sont à ce niveau de représentation déclarés comme variables. Et comme dans notre modèle de STEE, les actions qui manipulent une variable sont, par défaut, considérées comme actions internes ou des  $\tau$ -actions. Alors, nous faisons apparaître les actions sur les variables par envoi de leurs valeurs sur des signaux. La valeur du signal est alors envoyée dans une file d'attente que nous avons appelée *env*.

#### 3.3.6.1. Modélisation de l'environnement

Une étape de modélisation de l'environnement du circuit est nécessaire, elle nous permettra de s'assurer du bon fonctionnement des canaux d'entrées et de sorties. Cette étape consiste en une définition de quelques propositions, généralement des contraintes d'équité, telles que : les requêtes d'entrée sont stables jusqu'à leur acquittement, l'environnement garantira la présence des requêtes et répondra aux requêtes du système.



**Remarque :**  
 Le comportement de C\_Env (S1\_Env) est similaire à celui de E\_Env (S2\_Env). Il suffit de prendre en compte les changements d'identificateur et de type. Noter que C\_Env nécessite l'ajout d'une transition (e0,e1), étiquetée C[2]:=true.

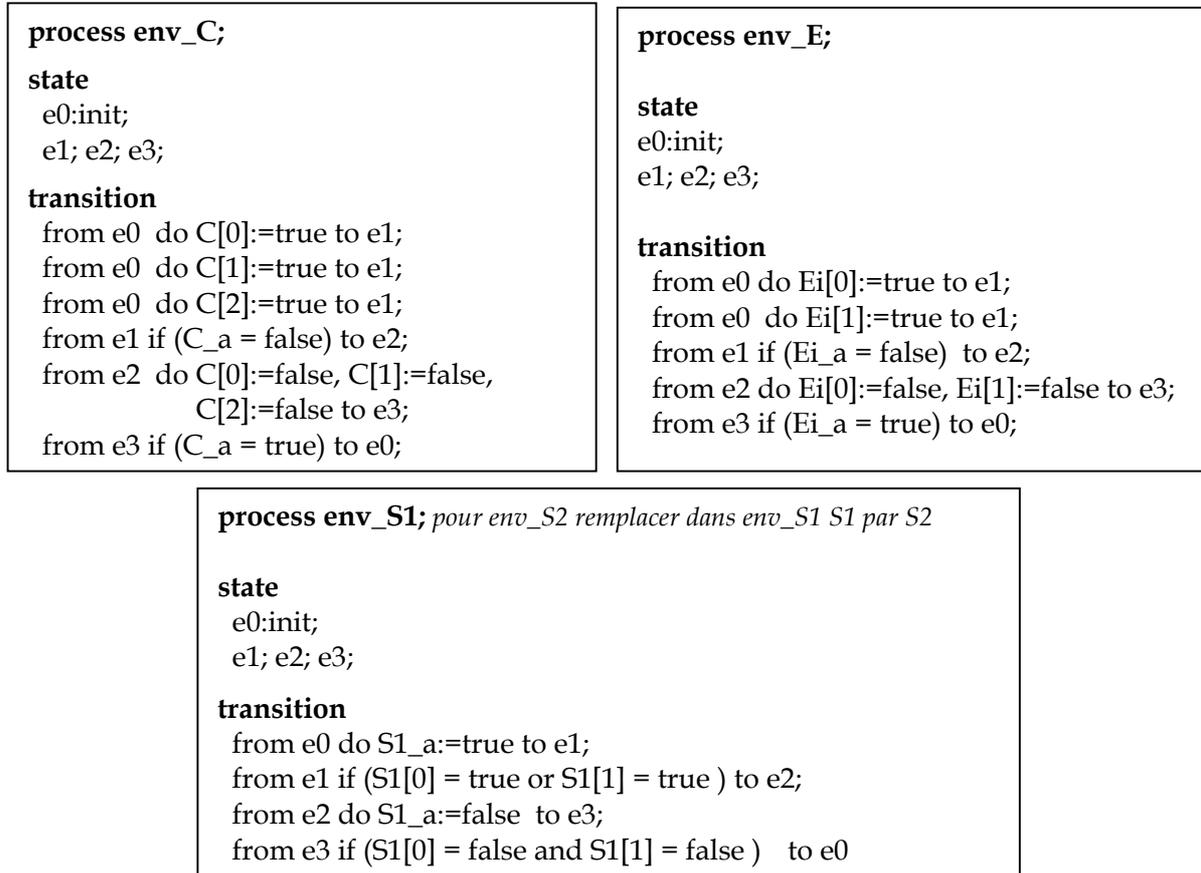
Figure 54 : Modélisation de l'arbitre asynchrone après expansion des communications.

Le comportement de l'environnement est modélisé, comme dans la première approche de vérification, grâce à des processus IF qui opèrent en concurrence avec le système global. La seule différence avec les processus VHDL concerne l'indéterminisme, qui est naturel en IF et qui doit être modélisé par des artifices en VHDL. Après l'écriture des processus

d'environnement, on obtient un grand *système fermé* IF (le système + les processus de l'environnement).

### a) Les processus modélisant l'environnement de l'arbitre

On compte quatre processus : deux pour modéliser les canaux d'entrée (env\_C et env\_E) et les deux autres pour modéliser les canaux de sorties (env\_S1 et env\_S2).



**Figure 55 :** Les processus modélisant l'environnement de l'arbitre

Le *système fermé*, composé du programme IF de l'arbitre et de son environnement, est un programme IF contenant l'ensemble de ces processus communicants.

### 3.3.6.2. Écriture et vérification de propriétés

Le STE généré pour une vérification après expansion des communications est différent, il est plus important en taille que celui généré pour la phase de vérification avant l'expansion de la communication. Les étiquettes générées sont elles aussi différentes et naturellement leur nombre est plus important (à ce niveau on manipule plus de valeurs à cause de l'implémentation du protocole de communication). Pour aider la lecture, les étiquettes du programme STE sont substituées par d'autres plus lisibles. Cette substitution est effectuée par la génération de macros en  $\mu$ -calcul. Les propriétés sont exprimées sur ces macros à la place des étiquettes.

Rappelons qu'il est toutefois intéressant de procéder à une phase *pré-vérification*, comme cela a été montré dans la section précédente.

### a) Application à l'arbitre asynchrone

Le STE généré pour l'arbitre asynchrone après expansion des communications contient 5660 états et 18600 transitions.

Par exemple l'étiquette " $+(Select,env,data\_S2,\{Select,\{false,true\}\})$ " veut dire : dans le processus *Select* le signal *data\_S2* est envoyée avec la valeur  $\{false,true\}$  dans la file d'attente *env*. Cette étiquette est remplacée par une plus lisible *S2\_01*.

Quelques macros de substitutions sont présentées dans le tableau suivant :

Port	Nouvelles étiquette	Étiquette originale
C	C_001	" $+(Select,env,data\_C,\{Select,\{false,false,true\}\})$ "
	C_010	" $+(Select,env,data\_C,\{Select,\{false,true,false\}\})$ "
E	E_01	" $+(Select,env,data\_E,\{Select,\{false,true\}\})$ "
S1	S1_01	" $+(Select,env,data\_S1,\{Select,\{false,true\}\})$ "
S2	S2_10	" $+(Select,env,data\_S2,\{Select,\{true,false\}\})$ "

**Figure 56 :** Quelques macros de substitution d'étiquettes pour l'arbitre asynchrone

### b) Vérification de quelques propriétés sur l'arbitre asynchrone après expansion des communications

On exprime les mêmes propriétés que celles qui étaient exprimées pour la vérification avant l'expansion des communications.

Prop.	Ecriture en $\mu$ -calcul	Temps de vér.
P1	$[true^*] \langle true \rangle true$	< 1 sec
P2	$[true^*]([C\_001] (\mu X . (\langle true \rangle true \text{ and } [not (S1\_01 \text{ or } S1\_10)]X)))$	< 1 sec
P3	$[true^*]([C\_010] (\mu X . (\langle true \rangle true \text{ and } [not (S2\_10 \text{ or } S2\_10)]X)))$	< 1 sec
P4	$[true^*]([C\_100] (\mu X . (\langle true \rangle true \text{ and } [not ((S1\_01 \text{ or } S1\_10) \text{ and } (S1\_01 \text{ or } S1\_10)]X)))$	< 1 sec

Notons que les propriétés ont la même signification une à une avec les propriétés vérifiées dans la Section 3.3.4.2.

### 3.3.7. Discussion

Nous avons présenté dans ce chapitre les deux approches de vérification que nous avons implémentées pour la vérification de circuits asynchrones: dans la première, on utilise des méthodes symboliques alors que dans la deuxième on utilise des méthodes énumératives.

Au-delà de la différence concernant le modèle de vérification adopté, les deux approches présentent une différence dans la sémantique d'exécution des actions parallèles :

- ✓ dans la première, les affectations concurrentes sont exécutées de manière simultanée. Cette simultanéité d'exécution est due à la sémantique d'exécution des affectations concurrentes du langage VHDL. L'approche *pseudo-synchrone* bénéficie alors d'une réduction de pré-ordre. Dans le cas d'une concurrence d'affectations, on n'exécute pas tous les entrelacements possibles, les affectations sont exécutées simultanément. Concernant la concurrence entre les actions de communication, l'entrelacement est assuré par l'indéterminisme de la réponse des acquittements.

- ✓ la deuxième approche conserve la sémantique du langage CHP en termes de sémantique d'exécution de la concurrence. En effet, tous les entrelacements possibles entre les actions concurrentes sont considérés. Par contre, la possibilité d'exécution simultanée de plusieurs actions concurrentes n'est pas prise en compte. Ce facteur est très coûteux et constitue, comme on va le voir par la suite, une limite de cette deuxième approche.

Dans [SBr03], Silver et Brzozowski traitent de l'influence des deux sémantique d'exécution. Ils démontrent que, dans le contexte des circuits asynchrones insensibles aux délais, la sémantique de l'entrelacement qui ne prend pas en considération la possible simultanéité des exécutions est valide.

Enfin, les deux modèles sous-jacents aux approches de vérification présentées dans ce chapitre : les réseaux de Petri et les STEE forment deux sémantiques complètes et bien définies du langage CHP.

## **Chapitre 4**

---

### **4. Évaluation de la Performance des deux Méthodes de Vérification**

## 4.1. Introduction

Dans ce chapitre, nous présentons les résultats de l'évaluation des performances des deux méthodes que nous avons développées :

- (1) Modélisation *pseudo-synchrone* avec l'utilisation d'un outil symbolique de vérification de modèles pour l'évaluation des propriétés temporelles ;
- (2) Modélisation en STEE avec utilisation d'un outil énumératif de vérification de modèles.

Nous évaluons le temps de vérification, le pic de mémoire vive consommée et la taille des BDD ou du STE par rapport à des paramètres significatifs des circuits asynchrones :

- ✓ La taille des données, ce qui demande de distinguer les ports de données des ports de contrôle ;
- ✓ La taille des structures de choix, ce qui correspond au nombre de chemins alternatifs ;
- ✓ Le nombre d'actions concurrentes.

L'exemple d'arbitre asynchrone décrit dans le chapitre 2, paragraphe 2.3.1 se prête bien à cette étude d'évaluation ; il contient : trois ports de données, un port de contrôle, une structure de choix et un bloc d'instructions concurrentes (des actions d'écritures). Pour cet exemple nous distinguons les paramètres caractéristiques suivants : la taille des ports de données (E1, S1, S2), la taille du port de contrôle qui correspond au nombre de chemins alternatifs, le nombre d'écritures concurrentes. Ces deux derniers paramètres font varier le nombre de sorties.

Pour chacun de ces paramètres caractéristiques, on réalise un jeu d'essai. Nous évaluons ensuite pour le paramètre en question les mesures de performance. Les résultats sont reportés dans des tableaux d'évaluations. Concrètement, on fait "*grossir*" un des paramètres caractéristiques et on fixe les autres, ce qui nous permettra d'observer l'évolution du temps de vérification, de la mémoire et de la taille du système.

Dans l'approche *pseudo-synchrone*[BBD03] nous ne considérons que les descriptions après expansion des communications. Les actions de synchronisations de ports ne sont pas disponibles en VHDL, alors les communications doivent être expansées. Dans la deuxième méthode[BBM03a, BBM03b], les deux niveaux de spécification sont considérés.

Pour mener à bien cette étude d'évaluation, nous avons choisi d'utiliser l'outil industriel de vérification symbolique FormalCheck[Bell](§6.2.3.1) dans la première approche. Dans la deuxième, nous avons utilisé l'environnement IF/CADP[BFG99, GJM97, IF, CADP](§6.2.3.2). Les résultats de l'étude sont obtenus en utilisant une machine SUN-Blade-100 Sparc avec 640 Méga-octets de mémoire vive.

## 4.2. La génération de jeux de test

Cette étape de génération de jeux de test consiste à générer des programmes paramétrés conformes aux formalismes d'entrée des deux outils de vérification : FormalCheck et IF/CADP.

### 4.2.1. Génération automatique d'exemples d'arbitre en IF

Pour permettre une génération sûre et correcte des exemples, on a réalisé un générateur automatique d'exemples. Le générateur prend comme paramètres de ligne de commande : la taille du contrôle, la taille des données et le nombre d'écritures concurrentes. Il génère en sortie le programme IF correspondant.

### 4.2.2. Génération d'exemples d'arbitres en VHDL pseudo-synchrone

Une manière plus aisée pour générer des exemples d'arbitres en VHDL est de les écrire en CHP. Ces exemples sont alors traduits par l'outil TAST pour produire le format Petri net correspondant. Finalement nous appliquons notre prototype de traduction pour enfin obtenir le code VHDL.

Par exemple un arbitre avec un signal de contrôle de quatre bits est similaire à l'arbitre de base (un signal de contrôle avec trois bits et des données à 2 bits), avec quelques modifications et adjonctions. Ces modifications sont notées en caractères gras.

```

COMPONENT Selector

PORT (   E : IN DR ;
        C : IN MR[4][1] ;
        S1, S2, S3 : OUT DR )

BEGIN

PROCESS main
PORT (   C : IN MR[4][1] ;
        E : IN DR ;
        S1, S2, S3 : OUT DR )

Variable x : DR ;
Variable ctrl : MR[4][1] ;

[C ?ctrl ;
@ [ ctrl = "0"[4] => E ?x ; S1 !x ; break
  ctrl = "1"[4] => E ?x ; S2 !x ; break
  ctrl = "2"[4] => E ?x ; S3 !x ; break
  ctrl = "3"[4] => E ?x ; S1 !x, S2 !x ; break
]
]
END ;
END ;

```

### 4.2.3. Traitement du paramètre “nombre de concurrences”

Le jeu de test a été réalisé sur le même exemple utilisé dans les précédentes étapes d'évaluation.

On commence par un arbitre sans concurrence, et on continue d'augmenter le nombre de concurrences et en même temps d'observer la complexité du système, la taille mémoire nécessaire et le temps que prend la vérification d'une même propriété.

La partie du programme CHP sujet de modification est en caractères gras.

COMPONENT Selector

```
PORT ( E : IN DR ;
      C : IN MR[3][1] ;
      S1, S2, S3, ... : OUT DR )
```

BEGIN

```
PROCESS main
PORT ( C : IN MR[3][1] ;
      E : IN DR ;
      S1, S2, S3, ... : OUT DR )
```

```
Variable x : DR ;
Variable ctrl : MR[4][1] ;
```

```
[C ?ctrl ;
@ [ctrl = "0"[3] => E ?x ; S1 !x ; break
  ctrl = "1"[3] => E ?x ; S2 !x ; break
  ctrl = "2"[3] => E ?x ; S3 !x ; break
  ctrl = "2"[3] => E ?x ; S1 !x , S3 !x ; break
  ...
  ctrl = "2"[3] => E ?x ; S1 !x , S2 !x, S3 !x, S4 !x, S5 !x, ... S32 !x ; break
]
END ;
END ;
```

Chaque instruction correspond à une étape d'évaluation. Il faut comprendre qu'une seule transition, correspondant au nombre de concurrences, est utilisée.

### 4.3. Déroulement des expérimentations

L'étude est effectuée en trois étapes :

- ✓ Dans la première étape, nous faisons varier la taille des données pour une taille fixe du contrôle (3bits) et un nombre fixe d'écritures concurrentes sur les ports S1 et S2.
- ✓ Dans la deuxième étape (voir Figure 57), nous faisons varier le nombre de chemins alternatifs, qui induit la variation du nombre de sorties. C'est-à-dire, varier la taille du contrôle pour une taille fixe des données (2bits) et un nombre fixe d'écritures concurrentes.
- ✓ Enfin, dans la dernière étape (voir Figure 58), pour une taille de donnée fixe (2bits) et une structure de choix aussi fixe (3 chemins), nous faisons varier le nombre d'écritures concurrentes et donc varier le nombre de sorties.

Pour chaque variation, nous évaluerons les paramètres de performance : temps de vérification, mémoire maximale utilisée et taille du BDD ou du STE.

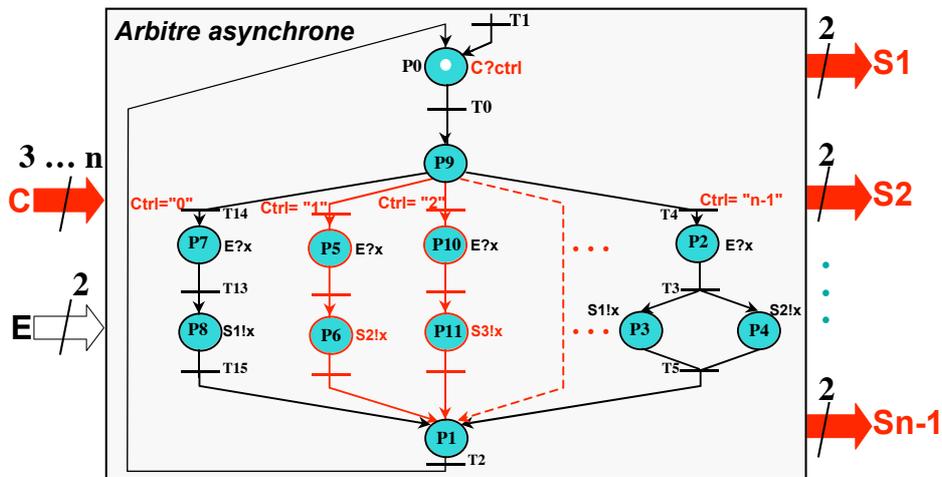


Figure 57 : Variations de la taille du contrôle

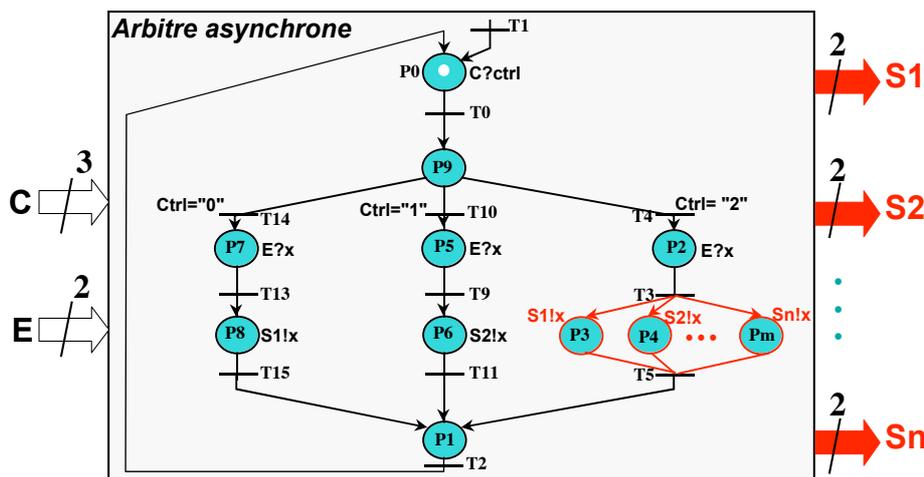


Figure 58 : Variations du nombre de concurrences

Dans les tableaux et les graphes illustrant les résultats, pour IF/CADP, nous mesurons séparément le temps de génération du modèle et les temps de vérification de propriété : le modèle peut être stocké et directement réutilisé pour une autre propriété. Pour FormalCheck, le temps de vérification inclut la construction du modèle.

### 4.3.1. Les propriétés exprimées pour l'évaluation

Les propriétés exprimées sont des propriétés fonctionnelles correspondant au comportement de l'arbitre. Ces propriétés ont été étudiées dans le chapitre précédent. Pour avoir une évaluation objective de la performance des deux approches de vérification, les expérimentations sont réalisées pour la même propriété. Nous donnons la syntaxe des propriétés dans le format de l'outil FormalCheck.

Dans les deux premières étapes d'évaluation concernant les paramètres : taille des données et taille du contrôle, la propriété suivante est exprimée pour tous les exemples :

“Si on se trouve à la place P0 et si on reçoit une entrée  $C = x''1$ ” alors, il est certain qu’il y aura une écriture sur S1”

After : (P0 = True and C = x''1")  
 Eventually : S1\_a = 1

Dans l’étape d’évaluation de la concurrence, la propriété choisie pour effectuer l’évaluation est la suivante :

“Si on se trouve à la place P0 et on reçoit une entrée  $C = x''4$ ” alors, il est certain qu’il y aura une écriture sur S3”

After : (P0 = True and C = x''4")  
 Eventually : S3\_a = 1

C’est une propriété qui concerne une des écritures concurrentes communes à tous les exemples.

**Remarque :** l’expression des propriétés que nous considérons dans cette étude d’évaluation, dépend des paramètres : taille des données, taille du contrôle et du nombre de concurrences.

### 4.3.2. Résultats de l’étude d’évaluation

Les résultats de cette étude d’évaluation sont reportés sur des tableaux et illustrés par des graphes.

#### 4.3.2.1. Légendes

Les légendes utilisées dans tous les graphes sont les suivantes :

a) pour les graphes de mesures du temps de vérification :

- ✓ CADP-Mgtime : le temps de génération par IF/CADP du modèle d’exécution ;
- ✓ CADP-Veriftime : le temps de vérification de la propriété par CADP ;
- ✓ CADP-Mgtime-EC : le temps de génération par IF/CADP du modèle d’exécution après l’expansion des communications ;
- ✓ CADP-Veriftime-EC : le temps de vérification de la propriété par CADP après l’expansion des communications ;
- ✓ FcheckVeriftime : le temps de vérification de la propriété par FormalCheck.

b) pour les graphes de mesures de la mémoire consommée :

- ✓ CADP-Memory size : la taille mémoire maximale utilisée par IF/CADP sans expansion des communications ;
- ✓ CADP-Memory size-EC : la taille mémoire maximale utilisée par IF/CADP après l’expansion des communications ;
- ✓ Fcheck-Memory size : la taille mémoire maximale utilisée par FormalCheck.

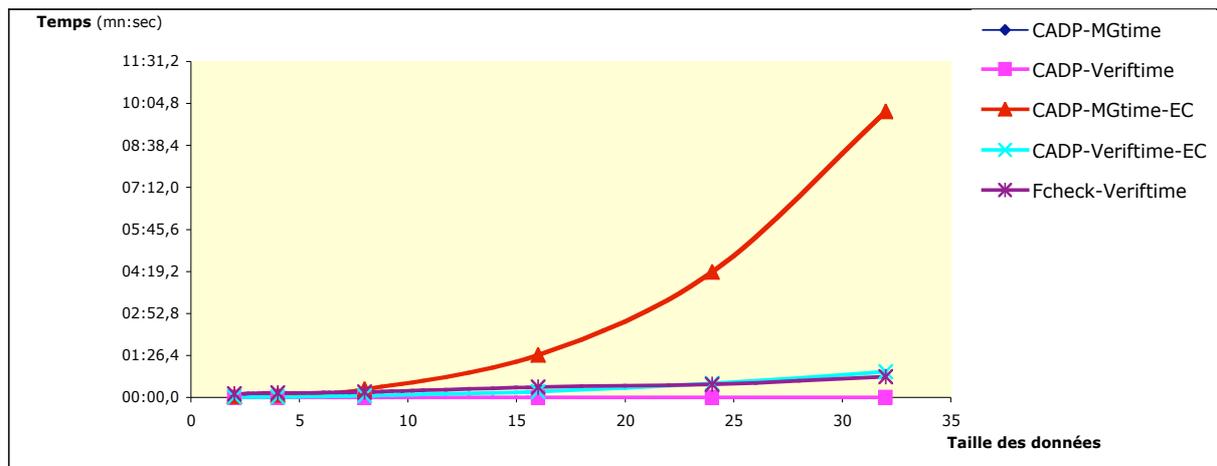
#### 4.3.2.2. Variation de la taille des données

Le Tableau 1 présente les résultats de l’évaluation en termes d’états atteignables, de taille mémoire et de temps d’exécution par rapport à la taille des données.

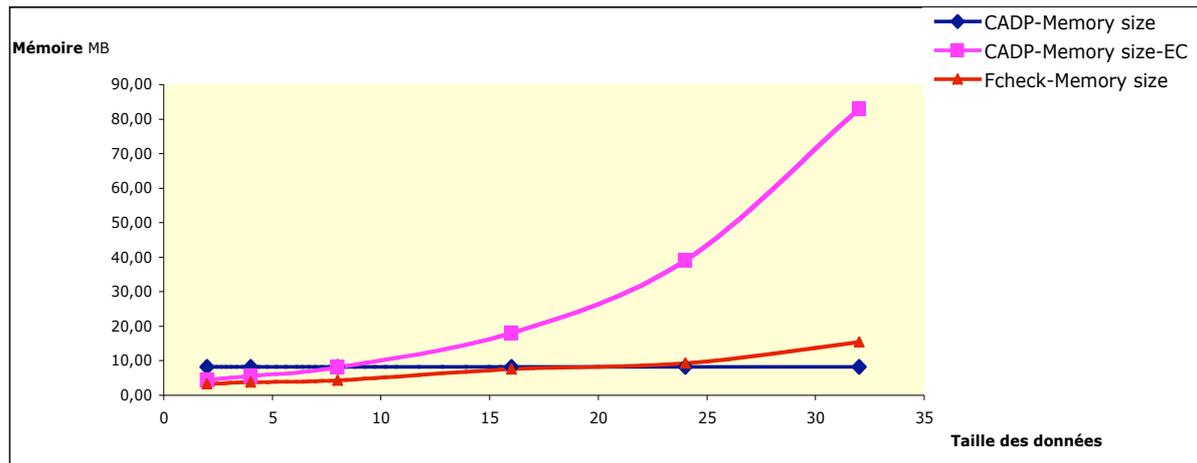
Taille des données (E, S1, S2)		2	4	8	16	24	32
a) Mesures sans expansion des communications							
CADP/ IF	Taille du STE: trans / states	0.42+e2 / 0.2+e2	1.08+e2 / 0.44+e2	3.12+e2 / 0.88+e2	1.01+e3 / 1.77+e2	2.09+e3 / 2.64+e2	3.55+e3 / 3.52+e2
	Taille mémoire	8168 KB	8168 KB	8176 KB	8184 KB	8224 KB	8232 KB
	Temps de génération	00 :00,5	00 :00,5	00 :00,5	00 :00,6	00 :00,6	00 :00,6
	Temps de vérification	00 :00,1	00 :00,1	00 :00,1	00 :00,1	00 :00,1	00 :00,2
b) Mesures après expansion des communications							
CADP/ IF	Taille du STE: trans / states	1.86+e4 / 5.66+e3	5.18+e4 / 1.53+e4	1.65+e5 / 4.77+e4	5.59+e6 / 1.64+e5	1.24+e6 / 3.51+e5	2.15+e6 / 6.06+e5
	Taille mémoire	4400 KB	5624 KB	8160 KB	18 MB	39 MB	83 MB
	Temps de génération	00 :02	00 :05	00 :17	01 :27	04 :18	09 :48
	Temps de vérification	00 :00,5	00 :01	00 :04	00 :12	00 :29	00 :53
Formal Check	Etats atteignables	2.71 e+03	1.07 e+04	1.71 e+05	4.38 e+07	1.12 e+10	2.87 e+12
	Taille mémoire	3.26 MB	3.84 MB	4.32 MB	7.60 MB	9.21 MB	15.37 MB
	Temps de vérification	00 :07	00 :09	00 :11	00 :21	00 :27	00 :43

**Tableau 1 :** Résultats de la variation de la taille des données

Sans expansion des communications, CADP donne des résultats meilleurs en temps d'exécution, mais consomme plus de mémoire à partir d'une taille des données supérieure à 16 bits. FormalCheck donne de meilleurs résultats que CADP après expansion des communications. En effet, CADP montre quelques sensibilités à l'évolution de la taille des données après l'expansion des communications.



**Figure 59 :** Les résultats en temps d'exécution quand on fait varier la taille des données



**Figure 60 :** Les résultats en taille mémoire utilisée quand on fait varier la taille des données

#### 4.3.2.3. Variation de la taille du contrôle

Le Tableau 2 présente les résultats de l'évaluation en termes d'états atteignables, de taille mémoire et de temps d'exécution par rapport à la variation de la taille du contrôle. Varier la taille de contrôle induit la variation du nombre de chemins alternatifs et augmente le nombre de ports de sorties de l'arbitre asynchrone.

Taille du contrôle (C)		2	4	8	16	24	32
a) Mesures sans expansion des communications							
CADP/ IF	Taille du STE: trans / states	0.42+e2 / 0.2+e2	0.62+e2 / 0.32+e2	1.82+e2 / 0.52+e2	5.84+e2 / 0.98+e2	1.26+e3 / 1.46+e2	2.25+e3 / 1.96+e2
	Taille mémoire	8176 KB	8184 KB	8184 KB	8192 KB	8224 KB	8242 KB
	Temps de génération	00 :00,5	00 :00,6	00 :00,8	00 :00,9	00 :01	00 :01
	Temps de vérification	00 :00,1	00 :00,1	00 :00,1	00 :00,1	00 :00,1	00 :00,1
b) Mesures après expansion des communications							
CADP/ IF	Taille du STE: trans / states	1.86+e4 / 5.66+e3	6.37+e4 / 1.68+e4	6.65+e5 / 2.01+e4	2.01+e5 / 6.08+e4	0.07+e5 / 1.23+e5	
	Taille mémoire	4400 KB	5352 KB	5872 KB	9808 KB	19 MB	
	Temps de génération	00 :02	00 :06	00 :08	00 :35	01 :35	
	Temps de vérification	00 :00,6	00 :01	00 :01	00 :04	00 :08	
Formal Check	Etats atteignables	2.71 e+03	8.62 e+03	1.33 e+06	4.30 e+10	9.92 e+23	4.75 e+30
	Taille mémoire	3.26 MB	4.45 MB	7.54 MB	14.04 MB	24.85 MB	38.89 MB
	Temps de vérification	00 :07	00 :10	00 :10	00 :41	10 :31	21 :36

**Tableau 2 :** Variation de la taille du contrôle

Nous remarquons que Formalcheck arrive toujours à vérifier la propriété, par contre il consomme plus de mémoire que CADP avant l'expansion des communications (voir Figure 62). CADP donne de bons résultats en temps d'exécution comparé à FormalCkeck. Concernant l'usage de mémoire, CADP après expansion des communications, n'arrive pas dans tous les cas à construire le modèle d'exécution (voir Figure 61).

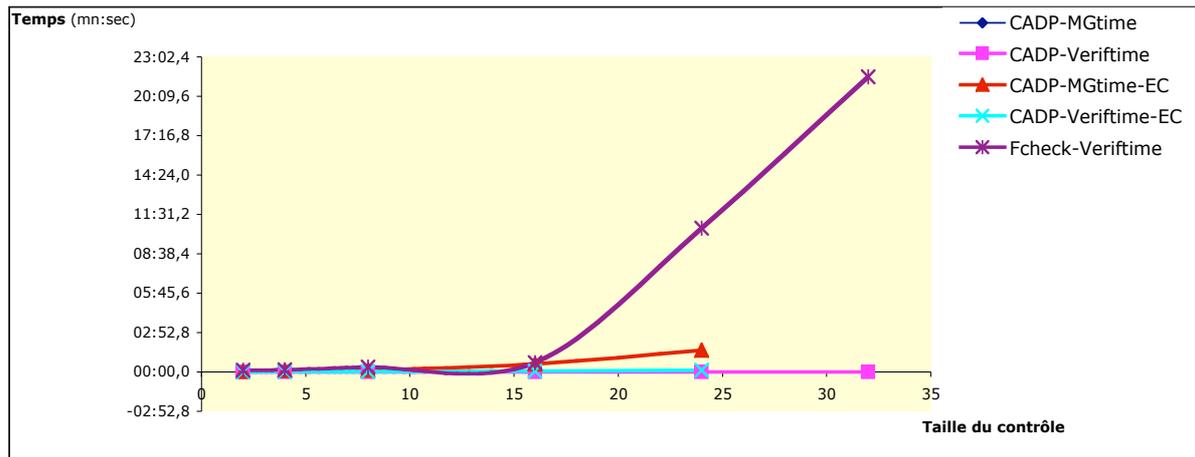


Figure 61 : Les résultats en temps d'exécution quand on fait varier la taille du contrôle

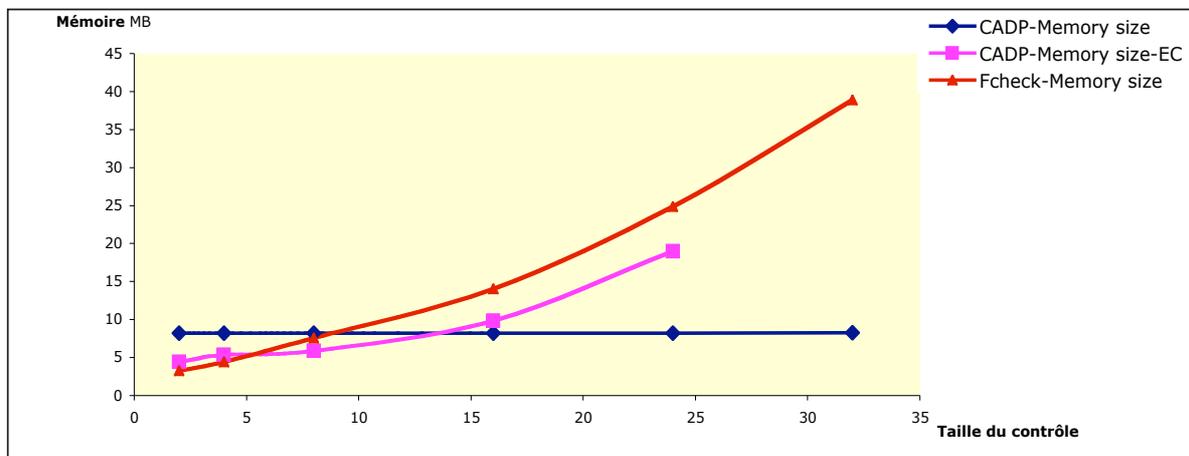


Figure 62 : Les résultats en taille mémoire utilisée quand on fait varier la taille du contrôle

#### 4.3.2.4. Variation de la concurrence : nombre d'écritures concurrentes

Le tableau suivant présente les résultats de l'évaluation en termes d'états atteignables, de temps d'exécution et de mémoire maximale utilisée par rapport au paramètre de concurrence. Les résultats sont obtenus en fixant la taille de données à un minimum, et en augmentant le nombre de sorties qui sont écrites de manière concurrente (Voir Figure 58).

Clairement, le nombre de chemins concurrents est le paramètre le plus dur pour un système qui énumère tous les entrelacements possibles d'événements. La mémoire requise pour la génération du modèle d'exécution est le facteur limitatif : par exemple, pour 8 écritures concurrentes, la construction a été manuellement stoppée après deux jours. Une large partie de ce temps a été passée sur l'échange avec le disque "swaping".

Dans le cas d'une variation de la concurrence, FormalCheck est clairement avantageux en mémoire et en temps d'exécution (voir Figure 63 et Figure 64). Par contre les performances obtenues par FormalCheck s'expliquent en partie par les avantages des modèles *pseudo-synchrones*. Ces derniers bénéficient d'une réduction de *pré-ordre* des entrelacements d'événements. Cette optimisation n'a pas été, à ce niveau, implémentée sur les modèles de STEE traités par CADP.

Nombre d'écritures concurrentes		1	2	4	8	16	24	32
a) Mesures sans expansion des communications								
CADP/ IF	Taille du STE: trans / states	0.36+e2 / 0.18+e2	0.42+e2 / 0.2+e2	0.98+e2 / 0.46+e2	2.08+e3 / 5.26+e2	1.05+e6 / 1.31+e5		
	Taille mémoire	8168 KB	8168 KB	8192 KB	8768 KB	17 MB		
	Temps de génération	00 :00,5	00 :00,5	00 :00,9	00 :15	01 :19		
	Temps de vérification	00 :00,1	00 :00,1	00 :00,2	00 :00,2	00 :15		
b) Mesures après expansion des communications								
CADP/ IF	Taille du STE: trans / states	3.2+e4 / 8.78+e3	4.28+e4 / 1.13+e4	5.94+e5 / 1.11+e5	?			
	Taille mémoire	4207 KB	5024 KB	13 MB	>1235 MB			
	Temps de génération	00 :03	00 :04	00 :48	> 156 :00			
	Temps de vérification	00 :01	00 :01	00 :11				
Formal Check	Etats atteignables	5.38 e+03	5.38 e+03	1.07 e+04	1.71 e+05	4.38 e+07	1.12 e+10	2.87 e+12
	Taille mémoire	4.23 MB	3.76 MB	5.56 MB	6.52 MB	11.21 MB	12.94 MB	20.51 MB
	Temps de vérification	00 :05	00 :05	00 :07	00 :09	00 :13	00 :22	00 :32

Tableau 3 : Variation du nombre de concurrences

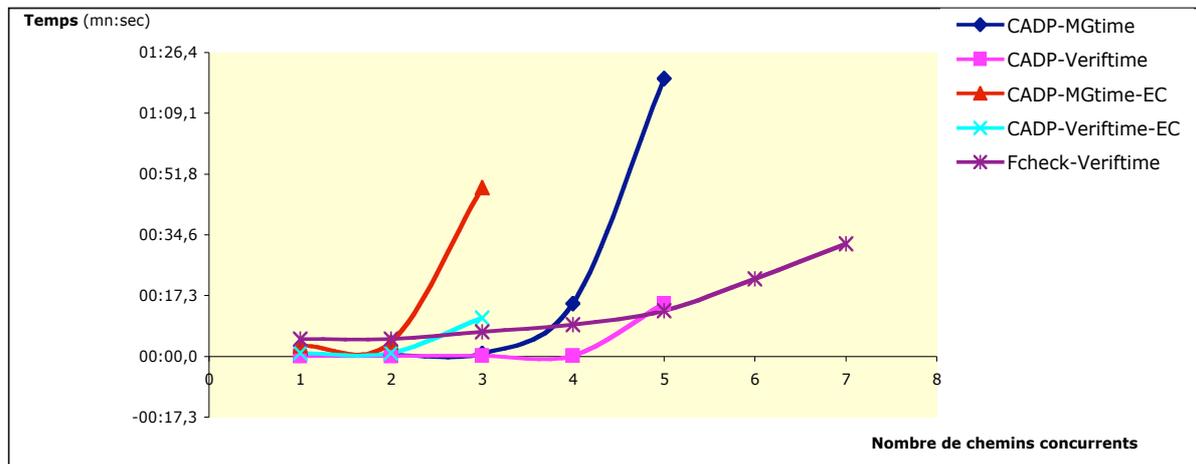


Figure 63 : Les résultats en temps d'exécution quand on fait varier le nombre de concurrences

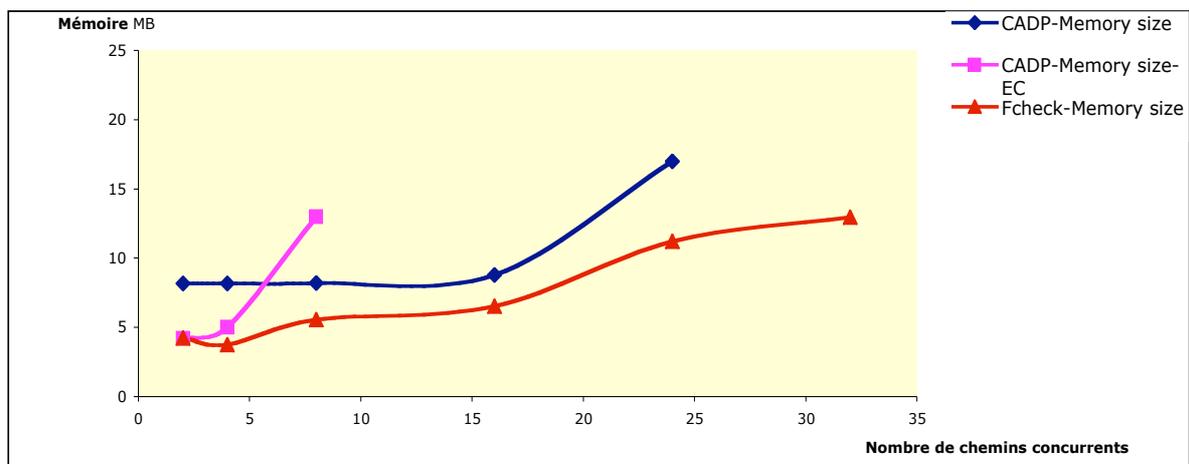


Figure 64 : Les résultats en taille mémoire utilisée quand on fait varier le nombre de concurrences

## 4.4. Conclusions

Quoique certaines exécutions dans le cas de CADP n'ont pas abouti, à cause de la capacité en mémoire vive requise, cette étude d'évaluation nous a permis de tirer quelques conclusions concernant les deux méthodes de vérification :

- ✓ Les méthodes énumératives de vérification de modèle offrent un niveau d'abstraction intéressant pour la vérification des spécifications asynchrones, avant l'expansion des communications.
- ✓ La vérification énumérative de modèles donne de meilleurs résultats en présence de nombreux chemins alternatifs.
- ✓ Le nombre de chemins concurrents augmente le temps de génération et la taille de STE.
- ✓ La vérification symbolique de modèles est plus efficace que la vérification énumérative de modèles :
  - en présence de nombreux chemins concurrents
  - après l'expansion des communications.

Ces expérimentations sont une preuve de plus que l'application directe et brutale d'outils de vérification de modèles ne fournira pas des résultats satisfaisants au problème de la vérification asynchrone. Comme cela a pu être prévu, des propriétés essentielles devraient être vérifiées au niveau le plus élevé possible de spécification. L'ingénieur de vérification devrait être disposé à appliquer une variété de techniques de réduction pour éviter le problème épineux, inhérent à toutes les méthodes de vérifications : l'explosion d'états.

Pour améliorer les performances de notre deuxième méthode de vérification, un certain nombre de techniques de réduction, et des stratégies de vérification, ont été implémentées dans une plate-forme globale pour la vérification de circuits asynchrones.



## **Chapitre 5**

---

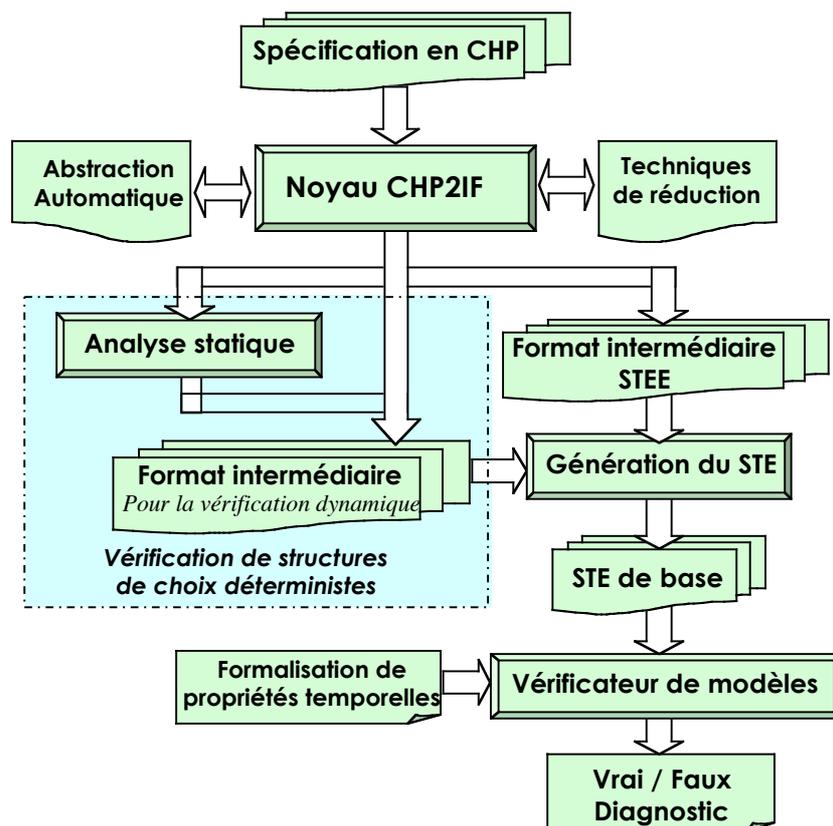
### **5. Environnement de Validation de Spécifications Asynchrones**

## 5.1. Introduction

Pour augmenter les performances de l'approche énumérative, nous avons développé et implémenté un certain nombre de techniques automatiques de réduction et d'abstraction. Ce travail a donné naissance à une plate-forme globale dédiée à la vérification formelle de circuits asynchrones.

Cette plate-forme consiste en un environnement de validation de spécifications de circuits asynchrones. Cet environnement est développé autour de notre approche de vérification énumérative. L'environnement de vérification a fait l'objet d'une implémentation et d'une démonstration à travers des études de cas qui seront présentées dans le chapitre suivant.

La Figure 65 présente l'environnement de validation. Nous partons d'une spécification en CHP, pour accomplir un certain nombre de tâches : vérification de propriétés temporelles, vérification de l'exclusion mutuelle entre les gardes d'une structure de choix déterministe. Cet environnement intègre un certain nombre de techniques de réduction et d'abstraction automatique[BBM03a, BBM03b, BBM04].



**Figure 65** : Environnement de validation de spécifications asynchrones

Dans la première partie de ce chapitre, nous allons présenter les différentes techniques d'optimisation que nous avons implémentées dans notre environnement de vérification. La deuxième partie présente une méthode de vérification de structures déterministes.

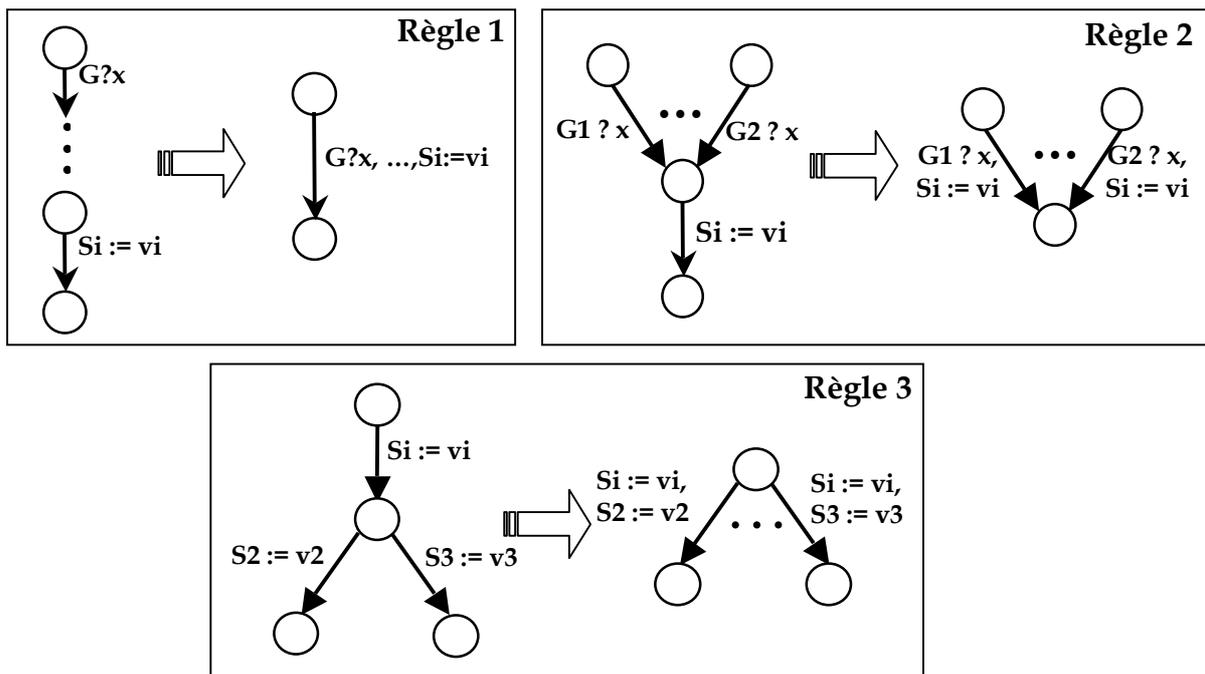
## 5.2. Techniques d'optimisation

Le but de ces techniques est de réduire lors de la phase de compilation le nombre d'états générés dans le STEE. Dans la majorité des cas, nous sommes amenés à générer, à partir du produit des STEE, le STE modélisant toutes les exécutions possibles. Il est alors évident que tout état supprimé du STEE réduira forcément la taille du système engendré pour la vérification et par conséquent le temps de calcul.

### 5.2.1. Réduction des automates

Dans un STEE, une transition peut être composée d'une garde, d'une expression de synchronisation et éventuellement de plusieurs affectations. En tirant avantage de l'expressivité du STEE, nous avons mis en place un ensemble de règles de transformations permettant de réduire la taille du STEE généré lors de la compilation de la spécification en CHP, tout en préservant la sémantique des automates.

Les règles de réductions sont présentées dans la Figure 66 .



**Figure 66** : Règles de réduction des STEE lors de la compilation

**Règle 1** : L'idée est de compacter un ensemble de transitions séquentielles en une seule transition. Cela est possible dans le cas d'une action de communication suivie d'une ou plusieurs affectations séquentielles.

**Règle 2** : Dans le cas d'une conjonction d'un ensemble de transitions suivie d'une affectation, cette dernière est rajoutée sur chacune des transitions précédentes.

**Règle 3** : Si une affectation est suivie d'une disjonction de plusieurs affectations, alors la première affectation est rajoutée au début de toutes les transitions qui lui succèdent.

Notons que chaque règle de transformation supprime au moins un état, et conserve strictement la même séquence dans les actions.

### 5.2.2. Abstraction automatique

L'abstraction est une technique connue pour remédier au problème de l'explosion d'états. Elle consiste à construire un modèle abstrait à partir du modèle d'origine dit modèle concret, tel que chaque action du modèle concret peut être simulée par une action dans le modèle abstrait.

Dans le processus de vérification, nous nous intéressons souvent à la vérification de l'aspect contrôle, puisque la vérification des propriétés fonctionnelles, de calcul par exemple, est difficile et très coûteuse. Une manière intuitive de vérifier plus aisément ces aspects contrôles consiste à faire abstraction des données. Dans cette optique, certains ports ou variables ne sont pas utilisés dans le contrôle des processus (la manière dont évoluent les processus), elles sont alors non-pertinentes pour la vérification des spécifications de contrôle, mais leur simple présence engendre un énorme surcoût de calcul et de mémoire lors de la phase de vérification.

L'idée est de modifier les variables qui ne sont pas indispensables pour qu'elles aient toujours une valeur constante. Au mieux, on peut complètement supprimer ces variables. Pour cela, nous avons mis au point une fonction d'abstraction appelée *Nec* diminutif de *Nécessaire*.

#### Définition 7 (la fonction d'abstraction)

La fonction d'abstraction *Nec* exprime la nécessité d'une variable ou d'un port pour la vérification d'une propriété de contrôle.

$Nec(x) = true$  si  $x$  est utilisée dans une garde ;

Ou si  $x$  intervient par dépendance dans le calcul de la valeur d'une variable  $y$ ,  
tel que  $Nec(y) = true$ ;

On applique cette fonction à l'ensemble des variables et des ports du programme CHP en utilisant les règles de dépendance suivantes.

Opérations CHP	Règles
<b>Garde</b> [ $f(v_1, \dots, v_n)$ ]	$Nec(v_1) \wedge \dots \wedge Nec(v_n)$
<b>Actions de communication</b> $P!(f(v_1, \dots, v_n))$ $P?v$	$Nec(P) \Rightarrow Nec(v_1) \wedge \dots \wedge Nec(v_n)$ $Nec(v) \Rightarrow Nec(P)$
<b>Action d'affectation</b> $v := f(v_1, \dots, v_n)$	$Nec(v) \Rightarrow Nec(v_1) \wedge \dots \wedge Nec(v_n)$

**Tableau 4 :** Règles de dépendance pour le calcul de l'abstraction automatique

La résolution de ce système d'équations se fait en pré-traitement de la compilation. Il donne ainsi lors de la génération du code IF toutes les informations concernant les variables qui ne sont pas nécessaires.

Après application de la fonction d'abstraction, nous pouvons construire l'ensemble *Abstract*. Soit  $x$  une variable ou un port d'un programme CHP,  $x \in Abstract$  si  $Nec(x) = false$ .

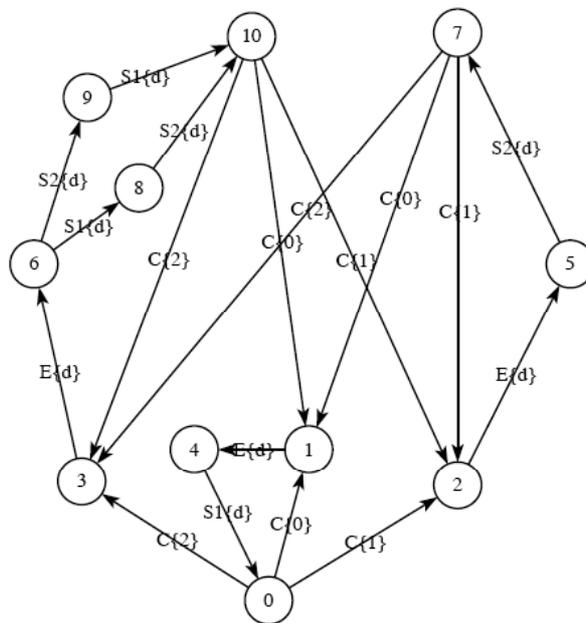
Finalemment la méthode d'abstraction consiste à construire l'ensemble *Abstract*, ensuite faire abstraction de tous les ports et les variables de cet ensemble.

### 5.2.2.1. Application à l'exemple de l'arbitre asynchrone

Le modèle d'exécution généré pour l'arbitre asynchrone sans abstraction des données est dans la Figure 52 du Chapitre 3. Avec l'application de l'abstraction automatique, nous obtenons le STE de la Figure 67.

Par application de la fonction d'abstraction nous obtenons l'ensemble des ports à abstraire.  $Abstract = \{E, S1, S2\}$ , le seul port nécessaire, et donc à ne pas abstraire est le port de contrôle *C*.

Les types des ports de données de l'ensemble *Abstract* sont alors abstraits par un type énuméré d'une seule valeur  $\{d\}$ .



**Figure 67** : STE du modèle d'exécution de l'arbitre asynchrone généré par abstraction

Nous observons que nous gagnons un facteur considérable en taille. Cette abstraction préserve les propriétés que nous avons vérifiées sur l'arbitre initial (§3.3.4.2).

### 5.2.2.2. Abstraction automatique paramétrable

Dans le cas où on désire vérifier une propriété qui dépend d'une variable de données, cette variable ne doit pas être abstraite, et bien entendu toutes les variables intervenant lors de son calcul. Nous avons prévu, pour répondre à ce besoin, l'implémentation d'une abstraction automatique paramétrée. Il est alors possible de signifier que telle ou telle variable est nécessaire pour la vérification d'une propriété. En permettant cette possibilité, nous avons amélioré notre technique d'abstraction, puisqu'elle ne devient plus réservée qu'aux aspects contrôle.

Concrètement nous éliminons le paramètre (variable ou port) de l'ensemble *Abstract*, il devient alors nécessaire et par conséquent ne peut pas être abstrait.

### 5.2.3. Génération du parallélisme

Comme nous l'avons vu précédemment, le langage CHP permet de créer des portions d'instructions parallèles à l'intérieur d'un processus alors que IF ne permet pas cette fonctionnalité. On pourrait résoudre dans tous les cas ce problème en créant de nouveaux processus qui ne contiendraient que les instructions séquentielles. Mais cette méthode engendre à chaque fois de nouvelles actions de communication pour permettre la synchronisation de ces nouveaux processus.

Une manière d'éviter la création de ces nouveaux processus est de générer explicitement l'entrelacement entre les actions concurrentes. La méthode consiste alors à isoler parmi l'ensemble des actions concurrentes, celles qui peuvent être représentées par une seule transition IF (expressions simples ou atomiques). On peut avec cet ensemble expliciter tous les entrelacements possibles (*interleaving*) des transitions (voir Figure 68).

#### 5.2.3.1. Exemple d'entrelacement

Voici un exemple de concurrence en CHP :

```
S1, S2, S3
```

où S1, S2, S3 sont des instructions atomiques (actions de communications ou de simples affectations)

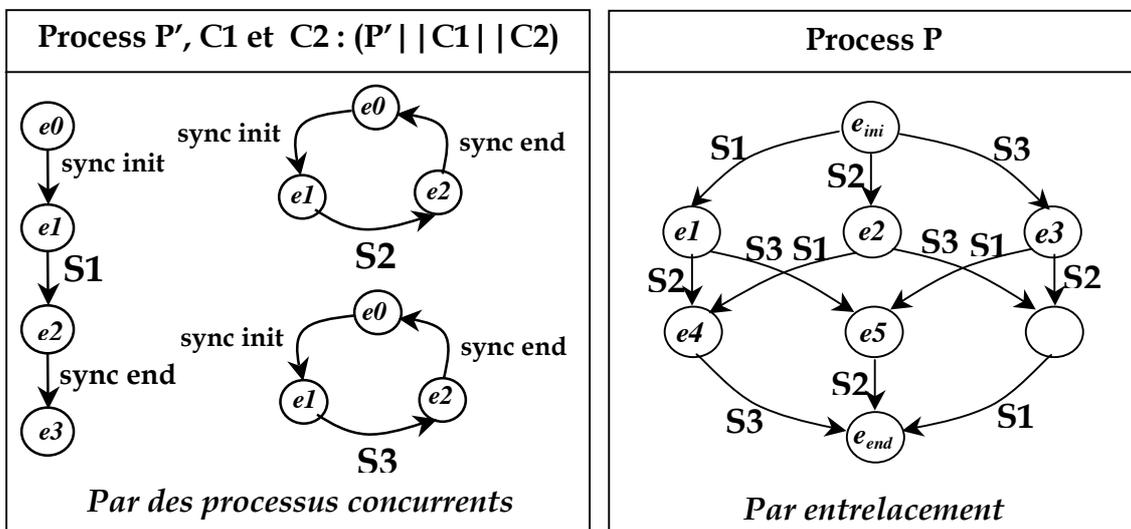


Figure 68 : Génération explicite de l'entrelacement des actions concurrentes

Notons que le nombre d'états nécessaires pour la génération des entrelacements possibles est :  $1 + 3 + 3 + 1 = 8$ .

### 5.2.3.2. Cas général

Soit  $k$  le nombre de transitions. Le nombre d'états nécessaires pour la génération de tous les entrelacements possibles est  $n_e = 2^k$ .

La construction du STE modélisant l'entrelacement de  $k$  transitions d'une manière optimale est basée sur l'utilisation du *Triangle de Pascal* [GKP94]. Pour construire tous les entrelacements possibles de  $k$  instructions concurrentes, il suffit de construire la représentation du Triangle de Pascal et d'ajouter les transitions entre les états.

La Figure 69 illustre les entrelacements entre 4 instructions : le nombre d'états nécessaires est  $2^4$  ; les 5 niveaux sont disposés de la manière suivante : «1. 4. 6. 4.1», selon les coefficients du

*Triangle de Pascal* :  $C_i = \frac{n!}{(n-i)!i!}$ .

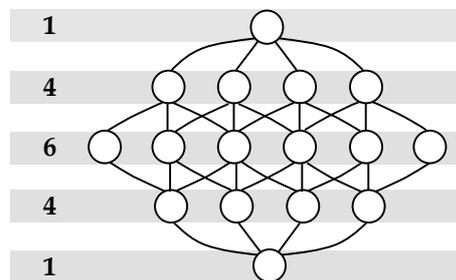


Figure 69 : Méthode de construction des entrelacements

### 5.2.3.3. Génération de l'entrelacement en présence de l'opérateur "probe"

Comme nous l'avons vu dans le Chapitre 2 pour la traduction de l'opérateur *probe*, le processus en communication avec celui qui utilise un opérateur *probe* doit faire précéder sa communication de l'affectation d'un indicateur. Cette action n'est plus atomique et ne devrait pas pouvoir être placée dans un entrelacement. On résout ce problème en réalisant la première partie de l'expansion avant l'entrelacement, la deuxième partie se trouvant dans l'entrelacement comme une communication normale.

Cet exemple est identique à celui de la Figure 68, S1 est remplacé par une action de communication. Le port P doit être étendu car un autre processus réalise un probe dessus.

Nous avons omis de préciser les actions S2 et S3 sur le schéma pour plus de clarté. (voir Figure 70 )

P!x , S2 , S3

En gras, nous avons placé les états dans lesquels P\_Probe reste à 1. Rappelons que *P\_Probe* doit être à 1 lorsqu'une communication peut être établie.

Une action **S1** (qui est l'expansion de la communication sur P) part bien de chacun de ces états. La sémantique de l'opérateur probe est donc respectée dans cet entrelacement.

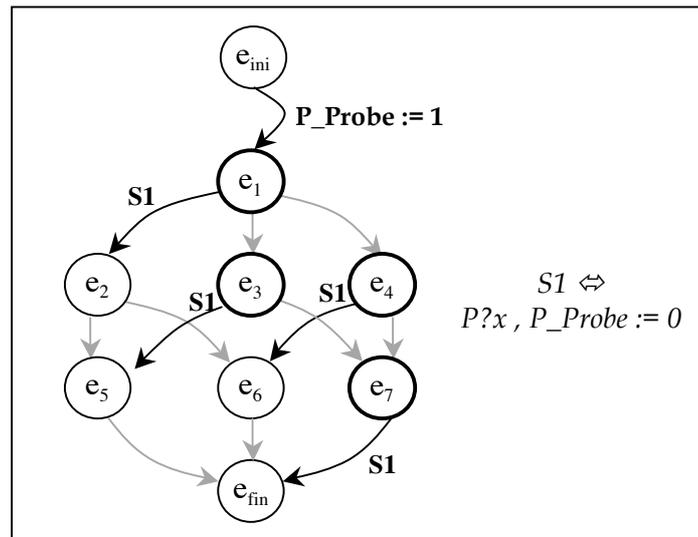


Figure 70 : Génération d'entrelacement entre actions concurrentes en présence d'un probe

#### 5.2.3.4. Nécessité de création de processus pour modéliser le parallélisme

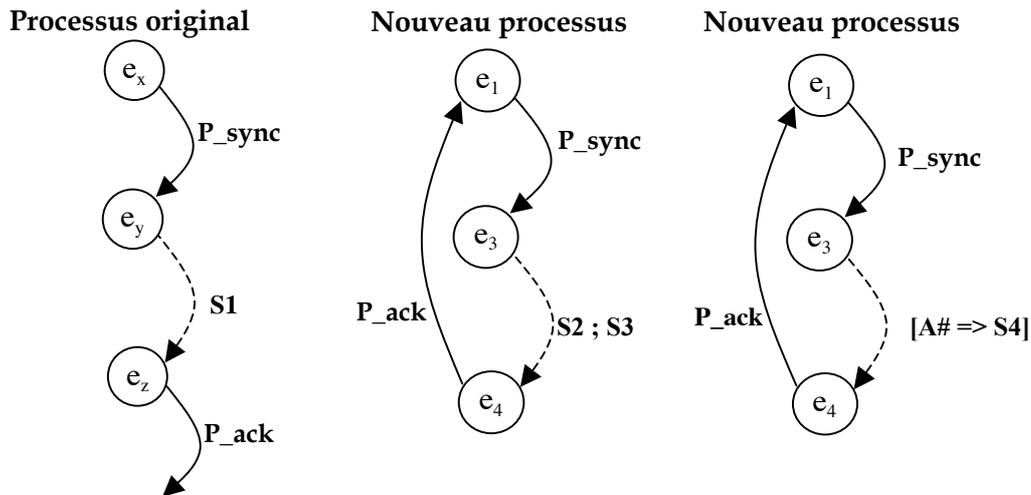
Dans certains cas, le traitement de la concurrence par génération explicite des entrelacements est plutôt compliqué. C'est typiquement le cas d'une concurrence entre des comportements séquentiels. Dans ce cas, nous sommes obligés de modéliser la concurrence par création de nouveaux processus concurrents synchronisés par des "extra-ports".

La partie de programme CHP suivante met plusieurs comportements en parallèle qui ne peuvent pas être générés en utilisant la méthode de génération explicite.

```
S1 , (S2 ; S3) , [A# => S4]
```

On doit extraire les portions séquentielles que l'on place dans de nouveaux processus. Ces nouveaux processus sont synchronisés sur le processus originel par les *extra-ports* : P\_sync et P\_ack. On découpe tout cela, puis on synchronise dans le processus qui contenait le parallélisme.

Le processus qui contenait le parallélisme récupère une des expressions (S1 dans l'exemple), et synchronise les nouveaux processus pour que ceux-ci soient exécutés au bon moment. Il faut prendre soin de globaliser les variables partagées entre les processus et de bien spécifier les ports synchronisés.



**Figure 71** : Exemple d'une concurrence nécessitant la création de nouveaux processus concurrents

#### 5.2.4. Réduction de pré-ordre

Toujours dans l'optique de réduire le STEE généré lors de la compilation, nous avons implémenté une technique de réduction de pré-ordre. Dans cette technique, nous considérons l'ensemble des affectations concurrentes. Nous procédons à l'extraction des affectations qui ne contiennent pas de dépendance de variable. Cet ensemble d'affectations concurrentes est alors remplacé par une seule exécution séquentielle.

Cet exemple CHP présente un ensemble de séquences (une par ligne) à réaliser en parallèle. On veut extraire les séquences qui ne changeront pas la sémantique indépendamment de l'ordre d'exécution.

- |     |            |
|-----|------------|
| (1) | x:=10,     |
| (2) | y:=x+1,    |
| (3) | z:=w,      |
| (4) | A!x,       |
| (5) | (C?t; B!t) |

Les deux premières lignes ne peuvent pas être retirés, car elles sont en inter-dépendance sur la variable x : selon la séquence d'exécution (1) ;(2) ou (2) ;(1), le résultat à la fin du parallélisme n'est pas le même.

L'instruction (3) peut être extraite, car ni la variable z, ni la variable w ne sont dépendantes d'une autre affectation.

La ligne (4) ne doit pas être extraite car c'est une action de communication, et nous pensons qu'il faudrait réaliser des vérifications très complexes pour savoir si la ligne (5) (les deux communications sur les ports B et C) influe sur les autres actions de communications. En effet, contrairement aux variables qui sont locales, le caractère global des ports rends plus difficile l'analyse de dépendance.

L'expression de la ligne (5) ne peut pas être extraite car ce n'est pas une action atomique.

En conclusion, sur cet exemple, nous pouvons exécuter séquentiellement l'instruction (3), suivie de l'entrelacement des instruction (1), (2), (4), (5).

### 5.3. Vérification de l'exclusion mutuelle entre les gardes

Il existe en CHP deux types de structure gardée : la structure déterministe (@) et la structure non déterministe (@@). En CHP, c'est le concepteur qui est chargé d'assurer l'exclusion mutuelle entre gardes dans le cas d'utilisation d'une structure déterministe, car ces contraintes ne sont pas vérifiées par le compilateur CHP.

La vérification formelle de l'exclusion mutuelle entre les gardes d'une structure de choix déterministe est d'un grand intérêt pour le concepteur de circuits asynchrones. Pour parfaire cette tâche de vérification, nous avons développé le schéma de vérification présenté dans la Figure 65. Nous proposons un traitement en deux phases :

- La première est statique, Pendant la compilation, nous faisons vérifier l'exclusion mutuelle à l'aide d'un outil symbolique de décision logique.
- Dans le cas où l'analyse statique rend comme résultat «*le modèle ne satisfait pas l'exclusion mutuelle*», la vérification dynamique permet de prendre en compte les contraintes de l'environnement pour affiner l'analyse. Si l'exclusion mutuelle n'est toujours pas satisfaite, elle donne un contre exemple. Le principe est de tester l'indéterminisme lors de la construction du modèle global d'exécution, à l'aide de CADP.

#### 5.3.1. Vérification statique par utilisation d'un outil de décision

Le principe de cette première méthode est simple. Il consiste à extraire les gardes d'une même structure de choix déterministe, les stocker dans une structure quelconque pour soumettre le résultat à un outil de décision qui répond par vrai ou faux.

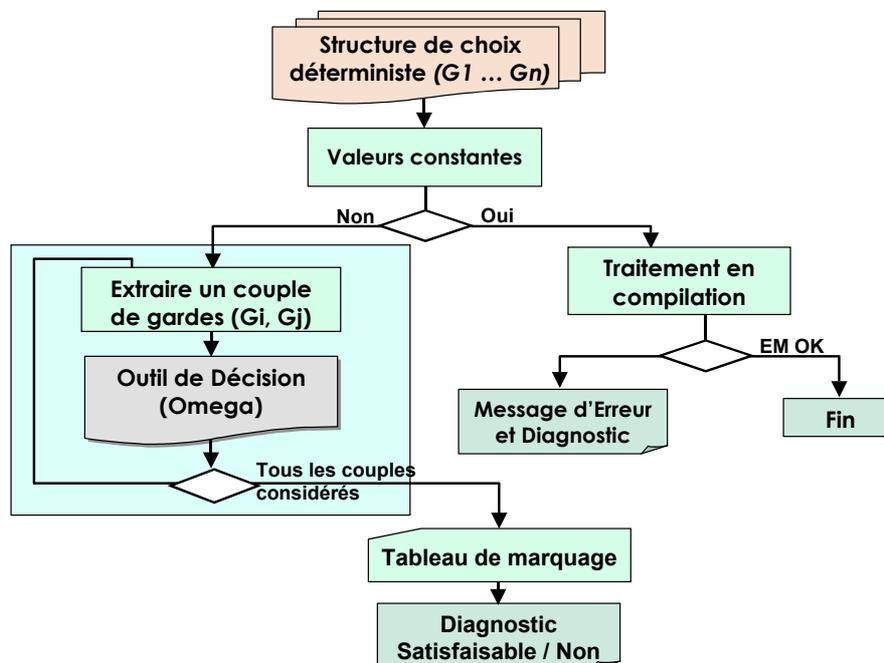


Figure 72 : Analyse statique pour la vérification de l'exclusion mutuelle.

Pour réaliser cette première méthode, nous avons spécifié un algorithme pour détecter les différentes structures de choix : à un ou à plusieurs niveaux d'imbrication de gardes.

Cas d'un seul niveau d'imbrication de gardes :

```
@[ G1 => ...
    ...
  Gn => ...
]
```

Cas de plus d'un niveau d'imbrication de gardes (voir Figure 73) :

```
@[ G1 => ... ;
  @[ G1_1 => ...
    G1_2 => ...
    ...
  ]
  G2 => ...
  ...
]
```

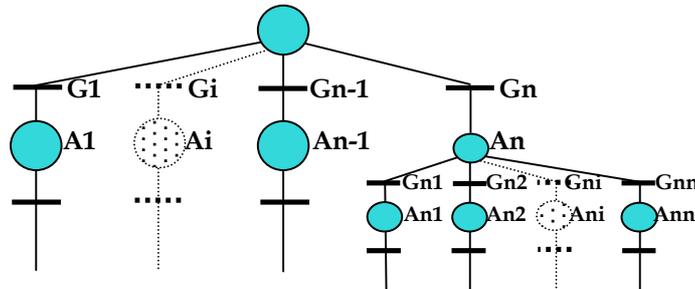


Figure 73 : Structures de choix en réseau de Petri

### 5.3.1.1. Cas simples

Souvent les structures de choix utilisées en CHP, les exemples en témoignent, font appel à un signal de contrôle dont on teste la valeur. Dans ce cas de figure, il est plus pratique de tester l'exclusion mutuelle entre les valeurs des gardes lors de la phase de compilation.

#### Exemple

```
...
C : IN MR[2];
...
variable Ctrl: MR[2];
[C?Ctrl;
  @ [ Ctrl = "0"[2] => ... break
    Ctrl = "1"[2] => ... break ];
```

Vérifier que les deux gardes ( $Ctrl = "0"[2]$  et  $Ctrl = "1"[2]$ ) sont exclusives peut se faire simplement par analyse syntaxique lors de la phase de compilation. De cette manière, on vérifie rapidement l'exclusion mutuelle sans faire appel à un outil de décision.

### 5.3.1.2. Algorithme

L'algorithme de la Figure 74 présente la méthode de vérification de l'exclusion mutuelle entre les gardes d'une structure déterministe dans un niveau statique.

```

Parcourir la spécification CHP ;
Extraire les gardes  $G_i$  d'une structure de choix  $S_i$ ;
Vérifier l'exclusion mutuelle deux à deux par un outil de décision ;
  Pour la structure  $S_i$  composée de  $n$  gardes, l'exclusion mutuelle
  entre chaque couple de gardes  $(G_i, G_j)_{(i \neq j)}$  est évaluée par Omega:
     $exp\_EM(G_i, G_j) = (G_i \text{ and } G_j)_{i \neq j}$ 

  Construction d'un tableau de marquage (TM) ;
    for( $i=1$ ;  $i < n$  ;  $i++$ ) /* Remplir un triangle de valeurs */
      for( $j=i$  ;  $j < n$  ;  $j++$ )
        if ( $exp\_EM(G_i, G_{j+1}) = false$ )  $TM[i, j+1] := t$  ;
        else  $TM[i, j] := f$  ;
  Si quel que soit  $(i, j)$  tel que  $(i < j)$ ,  $TM[i, j] = t$  : la structure est
  déterministe.
  Sinon les couples  $(G_i, G_j)$  ne satisfaisant pas l'EM sont imprimés.

```

**Figure 74 :** Algorithme de vérification de l'exclusion mutuelle entre les gardes d'une structure déterministe.

L'exclusion mutuelle est garantie si toutes les valeurs du tableau sont soit  $t$  soit  $d$ . Ce qui signifie que toutes les expressions logiques fournies au démonstrateur sont fausses. Dans le cas contraire, les informations (numéro de ligne dans le code CHP, l'expression de la garde etc ...) concernant les couples de gardes qui ne vérifient pas l'exclusion mutuelle sont imprimées.

#### Exemple :

Pour vérification à un niveau statique, l'exclusion mutuelle entre les gardes de la structure de choix :

```

@@[ ctrl=0 => ...
   ctrl=1 => ...
   ctrl=2 => ...
]

```

<i>Expression logique fournie à l'outil de décision</i>	<i>Résultat</i>
$(ctrl=0 \ \& \ ctrl=1)$	False
$(ctrl=0 \ \& \ ctrl=2)$	False
$(ctrl=1 \ \& \ ctrl=2)$	False

Pour chaque structure de choix déterministe, on construit un tableau de marquage, de la façon suivante :

```

TM[1,2] = TM_Eval( exp_EM(C1, C2)) = false
TM[1,3] = TM_Eval( exp_EM(C1, C3)) = false
TM[2,3] = TM_Eval( exp_EM(C2, C3)) = false

```

L'exclusion mutuelle entre les gardes de la structure du choix est vérifiée, toutes les réponses fournies par *Omega* sont de valeur *False* ;

### 5.3.2. Vérification dynamique

Dans la phase de vérification statique, si l'exclusion mutuelle est vérifiée, alors la structure de choix est déterministe, sinon on ne peut rien dire. En effet, dans certains cas, l'outil de décision peut trouver un recouvrement entre les gardes ou constater que les gardes sont indépendantes. Dans ces cas, on ne peut pas affirmer que l'exclusion mutuelle est violée. Alors, il est important de vérifier que, lors de l'exécution, l'exclusion mutuelle entre les gardes est bien vérifiée.

L'idée de cette deuxième phase de vérification est de faire vérifier par CADP le déterminisme. Il suffit de construire des expressions booléennes rattachées aux gardes de la structure de choix à vérifier. On peut alors communiquer à l'environnement un message sur la présence ou non du *non-déterminisme*.

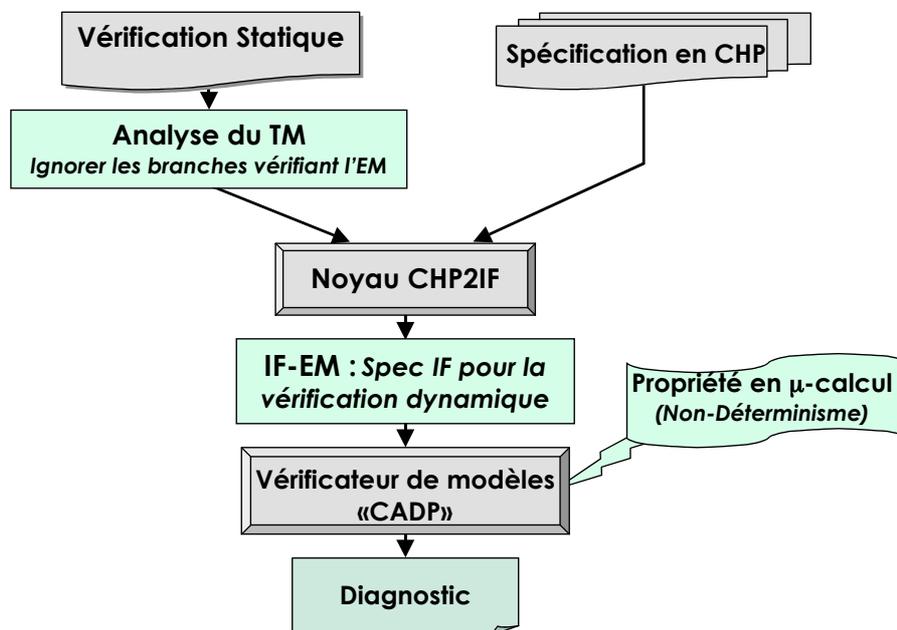


Figure 75 : Analyse dynamique pour la vérification de l'exclusion mutuelle.

#### 5.3.2.1. Format intermédiaire pour la vérification dynamique

La vérification dynamique est basée sur la construction d'un format intermédiaire qui permet, lors de la construction du modèle d'exécution, de mettre en évidence la violation de l'indéterminisme.

Nous ajoutons au niveau de chaque garde de la structure de choix une action qui consiste en l'envoi d'une expression booléenne spécifique. Cette expression booléenne est construite de la manière suivante :

Prenons l'exemple d'une structure de choix à  $n$  gardes  $G_1, \dots, G_n$ . Pour chaque chemin  $i$  de la structure de choix, on communique sur un *buffer* la valeur de l'expression booléenne composée du ou de toutes les gardes à l'exception de la garde correspondant au chemin actuel :

$$Exp_i := \bigvee_{j=1 \dots n (j \neq i)} G_j$$

Si la valeur de cette expression est “true”, alors il y a un indéterminisme entre la garde  $G_i$  et le reste des gardes.

D’une manière plus concrète, on définit un ensemble de paramètres :

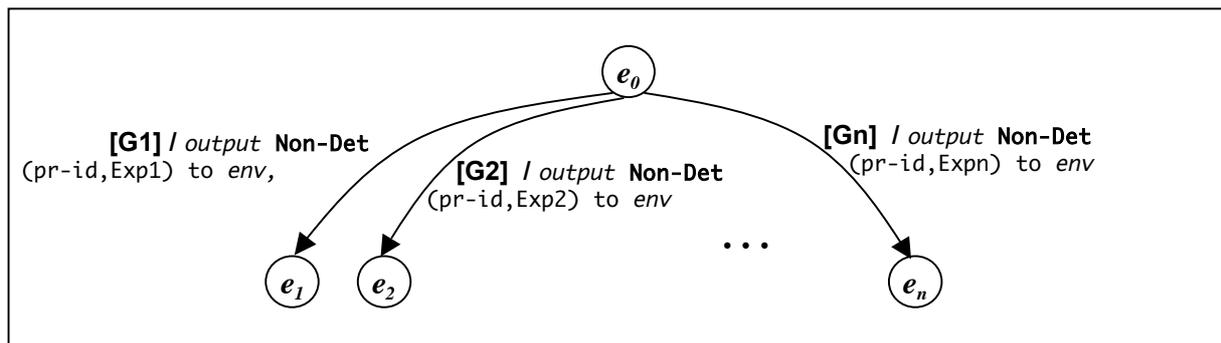
- **TG** : un type énumératif qui contient les identificateurs des gardes,
- **Non-Det** : un signal contenant la valeur de l’expression booléenne
- **env** : un buffer sur lequel on écrit la valeur du signal Non-Det

Dans IF, on retrouve ces paramètres décrits de la façon suivante :

```
Type TG enum g1, g2, ..., gn ; /* correspond aux gardes d’une structure */
Buffer env : queue :toenv of * ;
Signal Non-Det(pid, bool) ;
```

Pour la structure CHP : @[ G1 => ...  
 G2 => ...  
 ...  
 Gn => ... ]

on construit le système IF suivant :



**Figure 76** : Spécification IF dédiée à la vérification de l’indéterminisme.

Dans Figure 76, l’étiquette “ $[G1] / output \text{ Non-Det } (pr-id, Exp1) \text{ to env}$ ” veut dire que si la garde  $G1$  est vraie, alors on envoie la valeur  $Exp1$  du signal  $Non-Det$  dans le buffer  $env$ .

### 5.3.2.2. Analyse du tableau de marquage :

Une manière de rendre la vérification de l’exclusion mutuelle plus efficace et plus rapide est d’exploiter les résultats de la phase de vérification statique. En effet en analysant le tableau de marquage, on élimine les chemins correspondant aux gardes vérifiant l’exclusion mutuelle. Cela permet de réduire le nombre de choix à traiter dans la structure gardée, et par conséquent de rendre la phase de vérification plus performante.

### 5.3.2.3. Vérification par un vérificateur énumératif de modèles

Après la construction d’un système IF dédié à la vérification de l’exclusion mutuelle, il suffit de vérifier la propriété temporelle qui annonce qu’à partir d’un état initial, on n’arrive jamais à une transition avec un label vérifiant la condition de l’exclusion mutuelle citée auparavant.

**Propriété temporelle en  $\mu$ -calcul** : la propriété peut être exprimée de différentes façons. La propriété ci-dessous exprime qu'à partir d'un état initial, on n'arrive jamais à une action avec l'étiquette "Determin" :  $[\text{true}^*.\text{not}(\text{Determin}).\text{true}^*]$

L'étiquette Determin = " +(pr-id,env,Non-Det,{glob,{true}})" veut dire que le signal Non-Det est communiqué sur le buffer *env* avec la valeur *true*.

Si cette propriété est vérifiée, alors l'exclusion mutuelle entre les gardes est vérifiée.

### 5.3.3. Exemple d'application

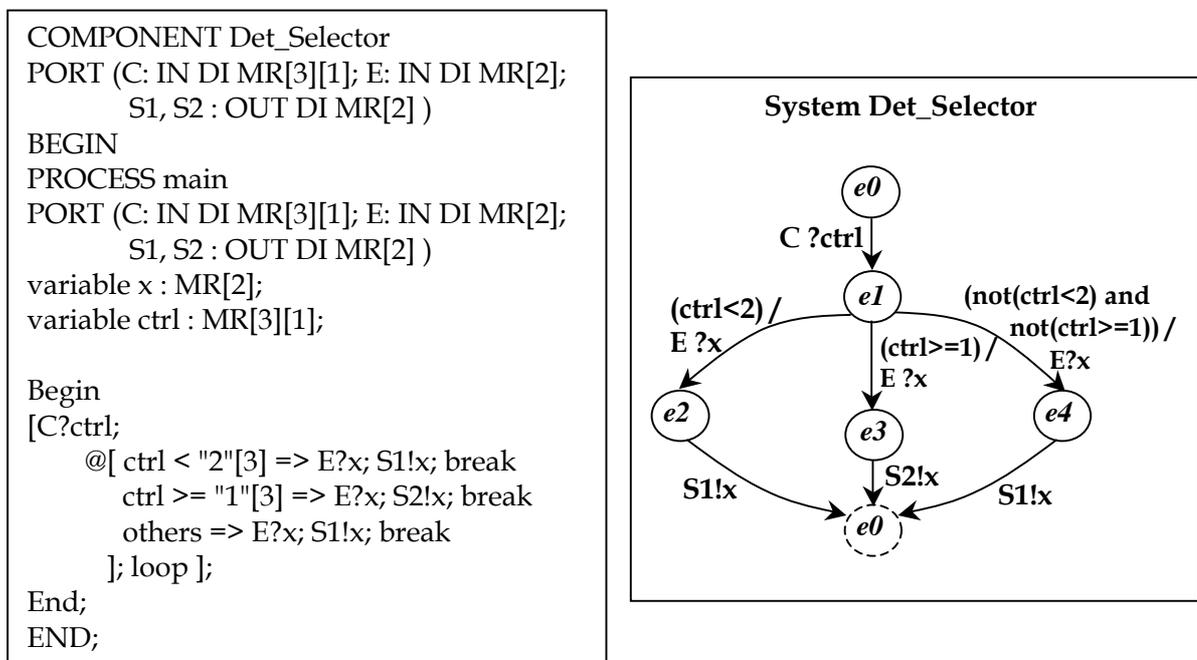
Pour mettre en évidence l'approche de vérification de l'exclusion mutuelle entre les gardes d'une structure de choix déterministe, nous présentons son application sur un petit exemple d'arbitre. Cet arbitre est semblable à l'exemple de l'arbitre qui nous a accompagnés le long du manuscrit.

Le circuit se comporte de la manière suivante : le signal de contrôle est lu, puis sa valeur est affectée à la variable locale *ctrl*. Cette dernière variable est testée par une structure de choix :

- Si  $\text{ctrl} < "2"[3]$ , alors on lit E et on l'écrit dans S1.
- Si  $\text{ctrl} \geq "1"[3]$ , alors on lit E et on l'écrit dans S2.
- Dans les autres cas, E est lu et ensuite écrit dans S1.

Le programme CHP de l'exemple et sa représentation en STEE sont donnés dans la Figure 77.

Dans cet exemple, nous utilisons les deux méthodes de vérification. En utilisant un outil de décision symbolique, on vérifie à un niveau statique l'exclusion mutuelle entre les gardes de la structure de choix.



**Figure 77** : Programme CHP et sa représentation en IF de l'exemple

Dans l'exemple, on distingue deux gardes :  $(\text{ctrl} < 2)$  et  $(\text{ctrl} \geq 1)$ . L'expression soumise à l'outil Omega est donc :  $(\text{ctrl} < 2) \& (\text{ctrl} \geq 1)$ . Le résultat de la vérification est différent de la

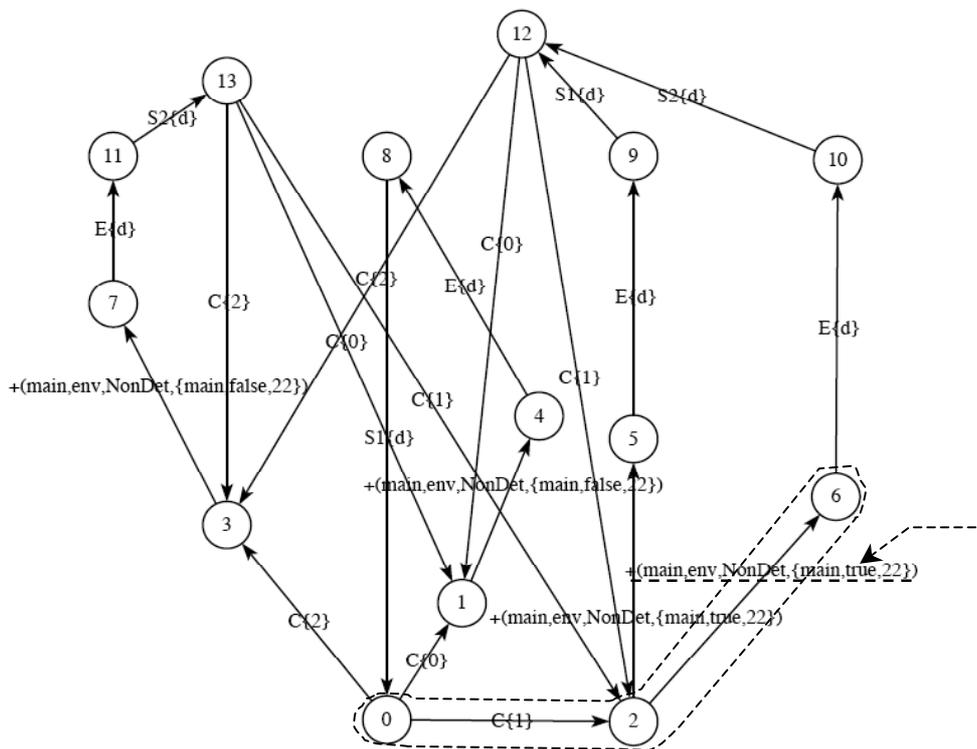
valeur «False». Alors à ce niveau, l'exclusion mutuelle n'est pas garantie. En effet, si l'environnement envoie sur le port de contrôle la valeur 1, on est devant un choix indéterministe, puisque les deux gardes :  $(ctrl < 2)$  et  $(ctrl \geq 1)$  seront vraies.

Expression logique fournie à l'outil de décision	Résultat
$(ctrl < 2) \ \& \ (ctrl \geq 1)$	{1 }

**Tableau 5 :** Résultats de l'analyse statique de l'exclusion mutuelle

L'application de la méthode dynamique sur notre exemple permet de construire le modèle d'exécution donné à la Figure 78. Nous pouvons facilement observer sur ce petit exemple que l'exclusion mutuelle n'est pas vérifiée. Un chemin d'exécution qui viole cette propriété est indiqué par des traits pointillés.

Si nous forçons l'environnement à ne communiquer que la valeur 2, il est évident que, dans ce cas, l'exclusion mutuelle est vérifiée. La méthode statique ne peut rien faire pour prouver l'exclusion mutuelle dans ce cas de figure, puisqu'elle n'exécute pas le modèle. Par contre la méthode dynamique est dédiée à la vérification lors de l'exécution de la spécification.



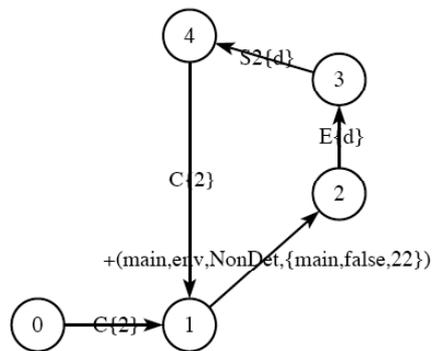
**Figure 78 :** Modèle d'exécution de l'exemple pour la vérification dynamique

```

PROCESS Environ
PORT ( C : OUT DI MR[3][1] )
Begin
[ C!"2"[3] ; loop] ;
End ;

```

En ajoutant le processus “Environ” à l’arbitre, par application de la méthode de vérification dynamique, nous obtenons le modèle d’exécution décrit dans la Figure 79. Le modèle est beaucoup plus petit que celui de la Figure 78 car l’environnement est contraint à ne communiquer que la valeur 2 sur le port de contrôle. Il est clair que l’exclusion mutuelle est vérifiée puisqu’il n’existe pas d’étiquette signalant la présence d’indéterminisme.



**Figure 79 :** Modèle d’exécution de l’exemple en contraignant l’environnement



## **Chapitre 6**

---

### **6. Mise en œuvre et Etudes de cas**

## 6.1. Introduction

Dans ce chapitre nous présenterons quelques études de cas concrets de l'application de notre méthode de vérification. Nous avons choisi deux exemples de circuits asynchrones avec des fonctionnalités différentes :

- ✓ Filtre FIR : une implémentation asynchrone d'un circuit de filtrage de signaux,
- ✓ Le circuit DES : une implémentation asynchrone d'un circuit de cryptage/décryptage de données.

Nous commençons par présenter brièvement les deux prototypes que nous avons développés, ensuite les outils que nous avons utilisés pour la vérification, et nous terminerons enfin par présenter les résultats des travaux de vérifications sur les deux circuits : le Filtre [Bou03, BBM03b] et le circuit DES[BBM03a].

## 6.2. Présentation des outils

### 6.2.1. Pnet2VHDL

Pnet2VHDL est un traducteur prototype qui a été développé pour les besoins de la vérification basée sur le modèle *pseudo-synchrone*. La méthode implémentée dans ce prototype a été expliquée dans le chapitre 3.

Pnet2VHDL prend en entrée le format intermédiaire PN-DFG de l'outil de synthèse de circuit asynchrone TAST et construit le VHDL pour la vérification formelle correspondant. Le prototype Pnet2VHDL est développé en C++, il a été testé sur de nombreux arbitres asynchrones, il a été également utilisé dans l'étude d'évaluation présentée dans le chapitre 4.

### 6.2.2. L'environnement de validation CHP2IF

C'est un environnement ouvert de vérification de circuits asynchrones, dont les principes et les méthodes ont été expliqués dans les chapitres 3 et 6. Nous avons implémenté une partie de cet environnement dans le cadre d'un stage de *maîtrise d'informatique*. La version actuelle de l'environnement est presque aboutie, elle permet de traiter une grande palette d'exemples, mais ne supporte pas tous les opérateurs et tous les types de variables. Certains opérateurs binaires, non disponibles en IF, ne fonctionnent pas, quant aux types de variables, ils doivent être des entiers non signés. Quel que soit le type de variable, il est transformé en un intervalle de valeurs. C'est le cas des types tableaux, qui se retrouvent transformés en un intervalle.

L'environnement CHP2IF permet de traiter également, comme cela a été montré dans le chapitre précédent, la vérification de l'exclusion mutuelle entre les gardes d'une structure déterministe.

### 6.2.3. Outils utilisés

#### 6.2.3.1. FormalCheck

FormalCheck est un outil industriel de vérification formelle développé par les laboratoires de Bells. Il considère une grande partie du sous-ensemble pour la synthèse des deux langages de conception de matériel Verilog et VHDL. Le modèle de vérification sous-jacent est le modèle symbolique.

L'utilisateur fournit un ensemble de propriétés, exprimées en CTL\*, à vérifier sur le modèle de conception. FormalCheck dispose des opérateurs simples de logique temporelle pour

décrire les propriétés comportementales à vérifier. Il est également possible de spécifier l'environnement grâce à des contraintes d'équité à indiquer à l'outil avant de procéder à la vérification.

FormalCheck assiste l'utilisateur lors de la vérification, par exemple pour chaque erreur, il indique la ligne susceptible d'être à l'origine de l'échec de la vérification.

### 6.2.3.2. CADP

CADP est un environnement de développement et de vérification de descriptions LOTOS. Il prend en entrée plusieurs formalismes : de haut niveau écrits en LOTOS, ou de bas niveau i.e, des systèmes de transitions étiquetées. Il est composé de plusieurs logiciels (ALDEBARAN, BCC, CAESAR, CAESAR.ADT, EVALUATOR, OPEN/CAESAR, XTL, etc) et offre plusieurs fonctionnalités : la simulation interactive ou aléatoire, la vérification comportementale par rapport à des relations de simulation ou de bisimulation, et la vérification logique des propriétés temporelles.

- ✓ CAESAR et CAESAR.ADT sont les compilateurs permettant la traduction des spécifications LOTOS vers des systèmes de transitions étiquetées.
- ✓ OPEN/CAESAR est une plate-forme qui permet l'implémentation des algorithmes de vérification à la volée.
- ✓ ALDEBARAN permet de minimiser ou de comparer des systèmes de transitions étiquetées modulo des relations d'équivalence et de préordre : la bisimulation forte, la bisimulation observationnelle, l'équivalence et le préordre de sûreté.
- ✓ XTL [MGa98, Mat98] est un langage fonctionnel conçu pour faciliter l'implémentation des divers opérateurs de logique temporelle, évalués sur des modèles construits explicitement.
- ✓ Et enfin, EVALUATOR[MSi00] est le composant de CADP chargé de la vérification. EVALUATOR est un model-checker à la volée de formules de  $\mu$ -calcul arborescent sans alternation. EVALUATOR effectue une vérification à la volée de la propriété temporelle sur le système de transitions étiquetées (STE). Le résultat de cette vérification (VRAIE ou FAUSSE) peut être accompagné d'un diagnostic selon les options d'exécution.

### 6.2.3.3. IF

IF est un format intermédiaire qui assure le lien entre des formalismes de description de haut niveau : SDL et LOTOS, et des modèles sémantiques de bas niveaux, notamment les systèmes de transitions étiquetées STE. IF dispose d'un environnement composé de plusieurs outils :

- ✓ IF/API est une bibliothèque pour la représentation et la manipulation syntaxique de programmes IF. IF/API offre quelques primitives générales de manipulation, en donnant accès aux objets contenus dans l'arbre syntaxique construit par la compilation d'un programme IF.
- ✓ IF2C est un compilateur qui produit les primitives de simulation en C, pour des programmes IF, selon la sémantique. Il a été réalisé à l'aide de l'interface IF/API.
- ✓ LIVE : l'outil LIVE implémente une analyse des variables actives et certaines optimisations syntaxiques induites sur les programmes IF. LIVE prend en entrée un programme IF et fournit en sortie un autre programme, où toute redondance produite par les variables inactives ou inutiles est éliminée explicitement par l'introduction systématique de re-initialisations.

- ✓ IF.OPEN est le générateur du modèle d'exécution en format STE. À partir d'une spécification en format IF, il nous fournit une représentation de toutes les exécutions possibles en format STE.

#### 6.2.3.4. Omega

*Omega* est un outil de décision symbolique [Omega, Pug92, KMP95], basé sur un algorithme de programmation en nombres entiers. Il peut déterminer si une dépendance existe entre deux tableaux de références, et si oui, dans quelles conditions. *Omega* a été initialement conçu comme un outil de décision sur des contraintes, ensuite étendu afin de fournir des réponses symboliques, plutôt que des réponses vrai/faux. Cette évolution permettrait une analyse standard plus rapide de la dépendance de données. L'outil *Omega* est actuellement étendu au point d'être un système complet pour simplifier et vérifier des formules de Presburger[KKr76].

### 6.3. Etudes de cas

#### 6.3.1. Filtre à Réponse Impulsionnelle Finie ou filtre RIF

Les filtres sont des dispositifs électroniques qui servent au filtrage linéaire des signaux à temps continu. Ils sont utilisés dans des applications dont l'importance économique est considérable : transmissions numériques, codage des sons MP3, synthèse de parole ...etc.

Le filtre que nous considérons ici est un filtre non-récurif ou à *Réponse Impulsionnelle Finie* «RIF». Ces filtres RIF sont des systèmes pour lesquels une valeur *Output* de sortie est obtenue par une somme pondérée d'un ensemble fini de valeurs d'entrée représentant les échantillons du signal à filtrer. Un filtre RIF possède une fonction de transfert polynomiale.



Figure 80 : Filtrage d'un signal

L'équation de la fonction de transfert s'écrit :

$$Output(n) = \sum_{i=0}^{N-1} Coeff(i) Input(n-i)$$

N est l'ordre du filtre.

##### 6.3.1.1. Description en CHP du Filtre RIF quatre étapes

Le filtre que nous avons vérifié est décrit dans Figure 81. C'est un filtre RIF d'ordre 4, avec des données d'entrée-sortie de 8 bits et 4 coefficients. Le comportement du filtre est modélisé par 13 processus en CHP. Les deux blocs : FSM et l'accumulateur contiennent des éléments mémorisants et sont chacun modélisés par deux processus pour les besoins de la synthèse[RDR02].

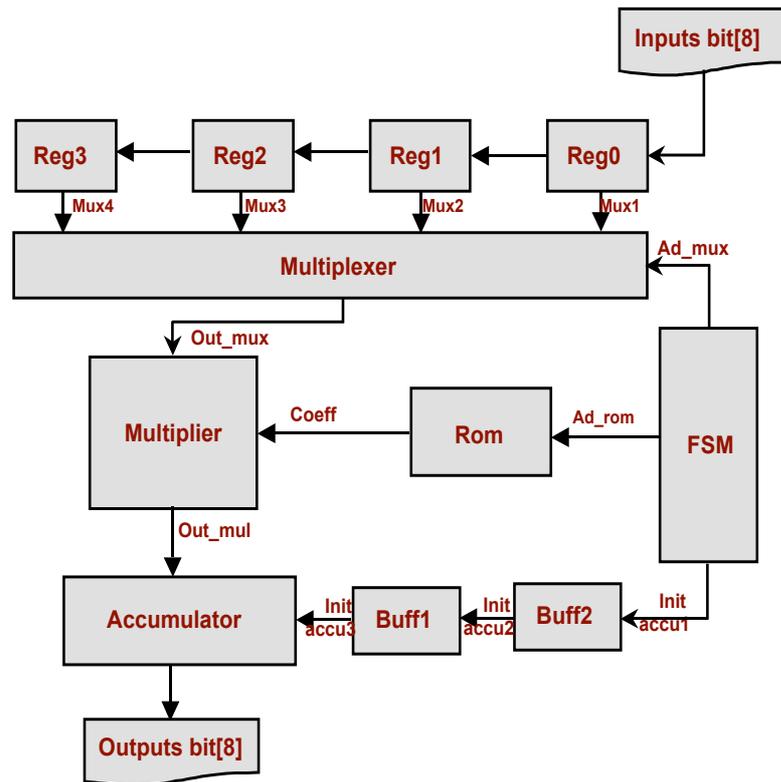


Figure 81 : Schéma de l'implémentation en CHP du Filtre RIF quatre phases

### 6.3.1.2. Modélisation du Filtre en STEE

Le composant CHP est décrit par un système IF, et chaque processus CHP est représenté par un processus IF. Pour illustrer l'étape de modélisation en STEE du Filtre, nous présenterons un certain nombre de processus CHP du filtre et leurs correspondants en STEE.

#### a) Le processus "accumulator\_r"

CHP	IF sans optimisation
<pre> PROCESS accumulator_r PORT(N_Acc : IN DI bit[18];      Acc : OUT DI bit[18] ) VARIABLE accu : bit[18]; BEGIN [Acc !"0000000000000000";  [N_Acc ?accu;   Acc !accu; loop ]; Break ]; END;                     </pre>	<pre> PROCESS accumulator_r; VAR   accu : MR_2_18; state   e1 :init;   e4; e3; e2; transition   from e4 sync acc !accu to e3;   from e3 sync n_acc ?accu to   e4;   from e1 sync acc !0 to e3;                     </pre>

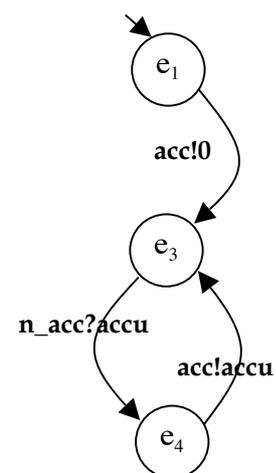


Figure 82 : Code CHP, code IF et représentation graphique du IF du processus "accumulator\_r"

**IF avec abstraction**

```

process accumulator_r ;

state
  e1 :init ;
  e4 ; e3 ; e2 ;

transition
  from e4 sync acc !d to e3 ;
  from e3 sync n_acc ?Abs-Data to e4 ;
  from e1 sync acc !d to e3 ;
    
```

Dans ce processus, l'abstraction ne fait que supprimer la variable locale abstraite "accu". **Abs-Data** est une variable globale introduite pour récupérer les informations abstraites envoyées par un autre processus. **D** représente l'envoi d'une donnée abstraite.

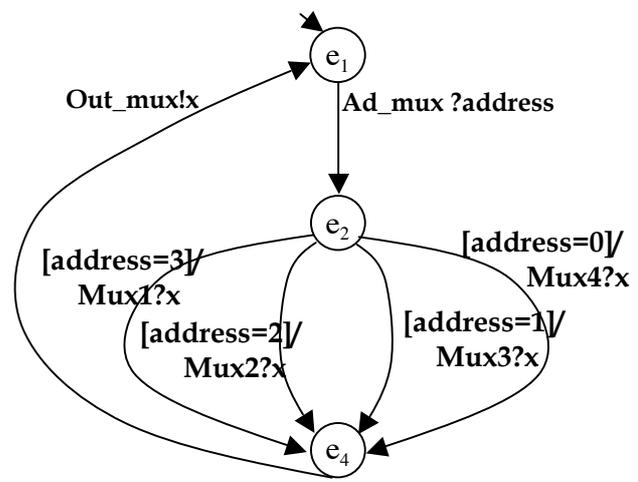
**b) Le processus "Multiplexer"**

Dans ce processus, le canal de commande (Ad\_mux) est de type MR[4][1 ], c'est-à-dire un codage de données de un parmi quatre. Le canal de contrôle Ad-mux est lu dans la variable locale "adresse". Selon la valeur de "adresse", un des quatre canaux (Mux1, Mux2, Mux3 ou Mux4) est lu et sa valeur est écrite sur le canal out\_mux.

```

PROCESS multiplexer
  Port (Ad_mux : in DI mr[4][1] ;
        Mux1, Mux2, Mux3,
        Mux4 : in DI mr[2] ;
        out_mux : out DI mr[2][8])

  Variable address : mr[4][1] ;
  Variable x : mr[2][8] ;
  Begin
    [ @[Ad_mux ?address ;
      [ address = 3 => Mux1 ?x ; break
        address = 2 => Mux2 ?x ; break
        address = 1 => Mux3 ?x ; break
        address = 0 => Mux4 ?x ; break ] ;
      out_mux !x ] ; loop ] ;
  End ;
    
```



(a) CHP code du "Multiplexer"

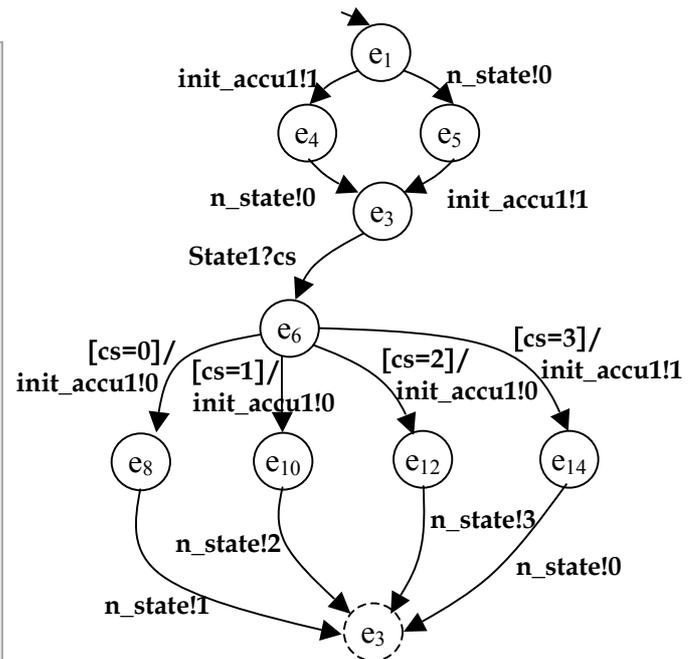
(b) représentation en IF du "Multiplexer"

**Figure 83 :** Code CHP du Processus "Multiplexer" et représentation en IF

c) Le processus "fsm\_s"

```

PROCESS fsm_c
PORT( State1 : IN DI MR[4];
      N_State : OUT DI MR[4];
      Init_accu1 : OUT DI bit )
VARIABLE cs : MR[4];
BEGIN
[N_State !"0000"[4], Init_Accu1 !'1';
 [State1 ?cs;
  @[cs="0"[4] => Init_Accu1 !'0';
    N_State !"1"[4]; break
  cs="1"[4] => Init_Accu1 !'0';
    N_State !"2"[4]; break
  cs="2"[4] => Init_Accu1 !'0';
    N_State !"3"[4]; break
  cs="3"[4] => Init_Accu1 !'1';
    N_State !"0"[4]; break
  ]; loop
]; break
];
END;
    
```



**Remarque :** l'état en bas en pointillés existe déjà. Il a été ajouté pour des raisons de lisibilité. L'automate possède donc bien une boucle.

Figure 84 : Le code CHP du processus fsm\_s et sa représentation en IF.

Le processus fsm\_s est une machine d'état qui commande le Filtre. Après une initialisation concurrente des ports : N\_State et Init\_Accu1, la machine d'état attend une information sur le port State1. Cette information indique l'état courant du Filtre, selon la valeur transmise sur le port State1 on écrit une des valeurs "0" ou "1" sur le port Init\_Accu1, puis on indique l'état suivant sur le port N\_State.

d) processus fsm\_r

```

PROCESSUS fsm_r
Port ( State : OUT DI mr[4];
      Ad_mux : OUT DI mr[4];
      Ad_rom : OUT DI mr[4];
      N_State : IN DI mr[4] )
Variable cs : mr[4];
Begin
[N_State ?cs;
 State1 !cs, Ad_mux !cs, Ad_rom !cs;
 Loop ];
End;
    
```

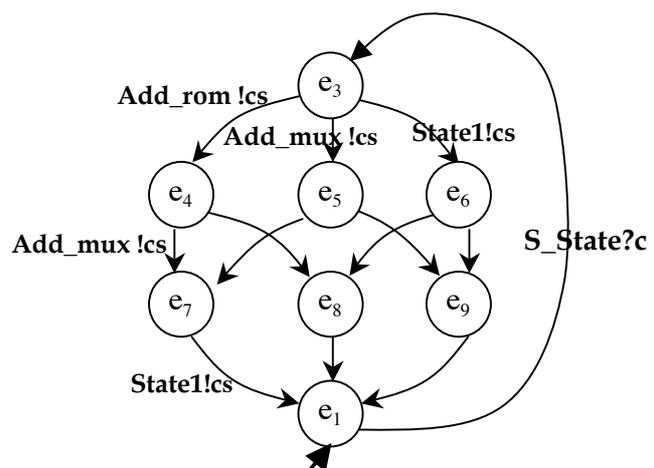


Figure 85 : Processus fsm\_r et sa représentation en IF

Le processus fsm\_r forme avec le processus fsm\_s présenté précédemment la machine d'états qui commande le Filtre. Il offre un exemple du traitement de la concurrence par génération

explicite des entrelacements de la structure de répétition. Le processus commence par récupérer une information du port  $N\_State$ , puis réalise les actions en parallèle.

### 6.3.1.3. Mise en évidence des réductions

Avant de procéder à la phase de vérification, les fichiers IF sont compilés. Cette phase réalise le produit des automates contenus dans le fichier IF. Lors de cette phase de génération, toutes les valeurs possibles des variables et ports sont considérées, c'est-à-dire que le résultat, exprimé en STE, est beaucoup plus grand que le programme IF. C'est sur ce format que nous pouvons constater l'utilité et les gains de l'application des techniques de réduction et notamment l'abstraction automatique.

Dans le cas du Filtre et sans l'application de la technique d'abstraction, aucune vérification ne peut être faite, car les ressources nécessaires en termes de mémoire et en temps de calcul sont trop importantes.

	Nombre d'états de l'automate généré par compilation du fichier IF du filtre	
	Sans abstraction	Avec abstraction
Sans réduction	> 1 000 000*	130 648
Avec réduction	> 1 000 000*	33 676

*Le compilateur a été arrêté à la main (CTRL-C)*

### 6.3.1.4. Vérification de quelques propriétés

Des propriétés caractéristiques du comportement de filtre ont été exprimées en  $\mu$ -calcul, et automatiquement vérifiées en utilisant CADP, sur un serveur SUN ultra 250 avec de la mémoire de 1.6 GB. Pour certaines d'entre elles, leur signification et le temps global de vérification (le temps de génération du modèle d'exécution + le temps de vérification) sont énumérés dans le tableau ci-dessous.

	Propriété	Temps de vérification.(sec)	Mémoire
<b>P1</b>	Absence de l'inter-blocage	$1.77 + 7.27 = 9.04$	884 MB
<b>P2</b>	Chaque entrée déclenche une sortie	$1.77 + 9.68 = 11.45$	865 MB
<b>P3</b>	Le multiplexeur doit recevoir les entrées (Muxi) dans l'ordre suivant : Mux4, Mux3, Mux2 et Mux1	$1.77 + 30.55 = 32.32$	885 MB
<b>P4</b>	Aucune nouvelle entrée n'est lue avant que Mux1 soit lu (règle de convolution).	$1.77 + 8.67 = 10.44$	879 MB

**Tableau 6** : Résultats de vérification de quelques propriétés temporelles

### 6.3.1.5. Vérification par réduction

Pour vérifier la propriété P3 qui dépend seulement des canaux de Mux(i), une technique alternative est disponible. Le comportement du filtre est réduit en cachant toutes les étiquettes qui ne dépendent pas des canaux de Mux(i). Ceci peut être obtenu en appliquant des techniques de réduction et de préservation d'équivalence. Le STE résultant est donné dans la Figure 86, il montre que le multiplexeur reçoit les entrées Mux(i) dans l'ordre correct.

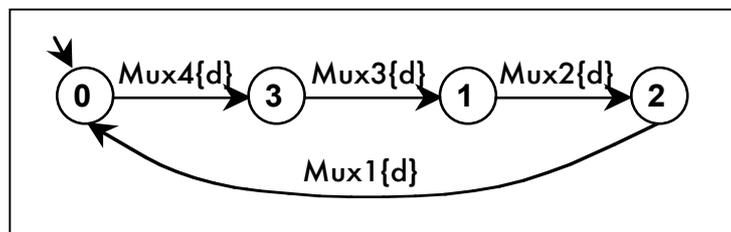


Figure 86 : Le STE réduit pour la propriété P3

### 6.3.2. Circuit de DES

Le DES “Data Encryption Standard” est un système de chiffrement par blocs de 64 bits, dont 8 bits (un octet) servent de test de parité (pour vérifier l’intégrité de la clé). Chaque bit de parité de la clé (1 tous les 8 bits) sert à tester un des octets de la clé par parité impaire, c’est-à-dire que chacun des bits de parité est ajusté de façon à avoir un nombre impair de ‘1’ dans l’octet à qui il appartient. La clé a donc une longueur “utile” de 56 bits, ce qui signifie que seuls 56 bits servent dans l’algorithme.

Le DES fut normalisé par l’ANSI (American National Standard Institute) sous le nom de ANSI X3.92, plus connu sous la dénomination DEA (Data Encryption Algorithm).

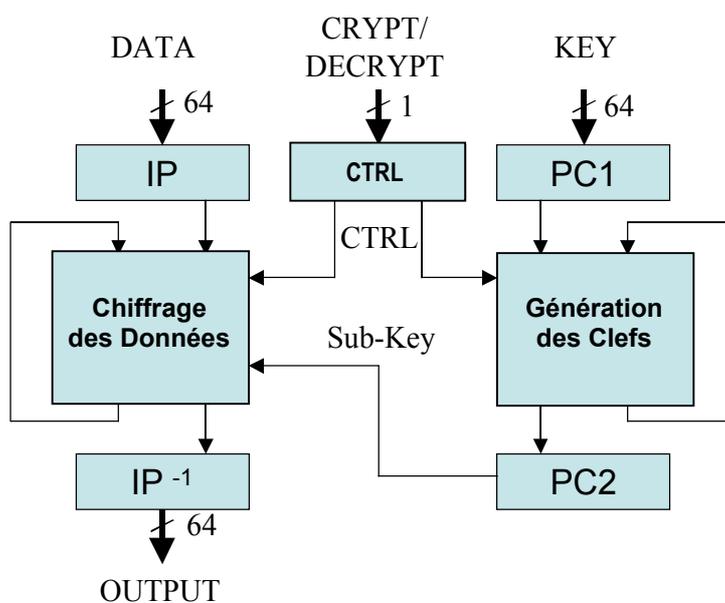


Figure 87 : L’architecture globale du circuit DES

L'architecture globale du circuit DES asynchrone est décrite dans la Figure 87. C'est fondamentalement une structure itérative, basée sur trois blocs en boucles synchronisées par des canaux de communication. Le canal "sub-key" synchronise le bloc de chiffage du chemin de données avec le bloc de génération des clefs. CTRL est un ensemble de canaux produits par le bloc contrôleur (une machine d'état fini) qui commande les chemins de données le long de seize itérations comme indiqué par l'algorithme du DES [DES].

Le canal d'entrée CRYPT/DECRYPT de 1-bit est employé par le contrôleur pour configurer le circuit et déclencher le chiffage. Les canaux 64-bit DONNÉES et CLEF sont utilisés pour écrire respectivement le texte plat et la clef. Le texte chiffré est produit sur le canal 64-bit de sortie Output.

### 6.3.2.1. Algorithme du DES

L'algorithme consiste à faire des combinaisons, des substitutions et des permutations entre la donnée à chiffrer et la clé, en faisant en sorte que les opérations puissent se faire dans les deux sens (pour le déchiffrement). La clé est codée sur 64 bits et formée de 16 blocs de 4 bits, généralement notés  $k_1$  à  $k_{16}$ . Etant donné que "seulement" 56 bits servent réellement à chiffrer, il y a  $2^{56}$  (soit  $7.2 \cdot 10^{16}$ ) possibilités de clés différentes !

L'algorithme du DES présenté ci-dessous étant public, toute la sécurité repose alors sur la complexité des clés de chiffrement.

Les grandes lignes de l'algorithme sont les suivantes :

- ✓ Fractionnement du texte en blocs de 64 bits (8 octets)
- ✓ Permutation initiale des blocs
- ✓ Découpage des blocs en deux parties : gauche et droite, nommées G et D
- ✓ Etapes de permutation et de substitution répétées 16 fois (appelées rondes)
- ✓ Recollement des parties gauche et droite puis permutation initiale inverse
- ✓ L'ensemble des étapes précédentes (rondes) est réitéré 16 fois.

A la fin des itérations, les deux blocs G16 et D16 sont recollés, puis soumis à la permutation initiale inverse, le résultat en sortie est un texte codé de 64 bits !

### 6.3.2.2. Description en CHP du circuit DES

La description en CHP du DES est décrite par un composant CHP. Ce composant est organisé en trois blocs : bloc de chiffage des données, bloc de génération des clefs et bloc de contrôle. L'architecture en CHP du DES est montrée dans la Figure 88. Elle est composée de 26 processus :

- 19 dans le bloc de chiffage des données,
- 5 dans le bloc de génération des clefs,
- et 2 dans le bloc de contrôle.

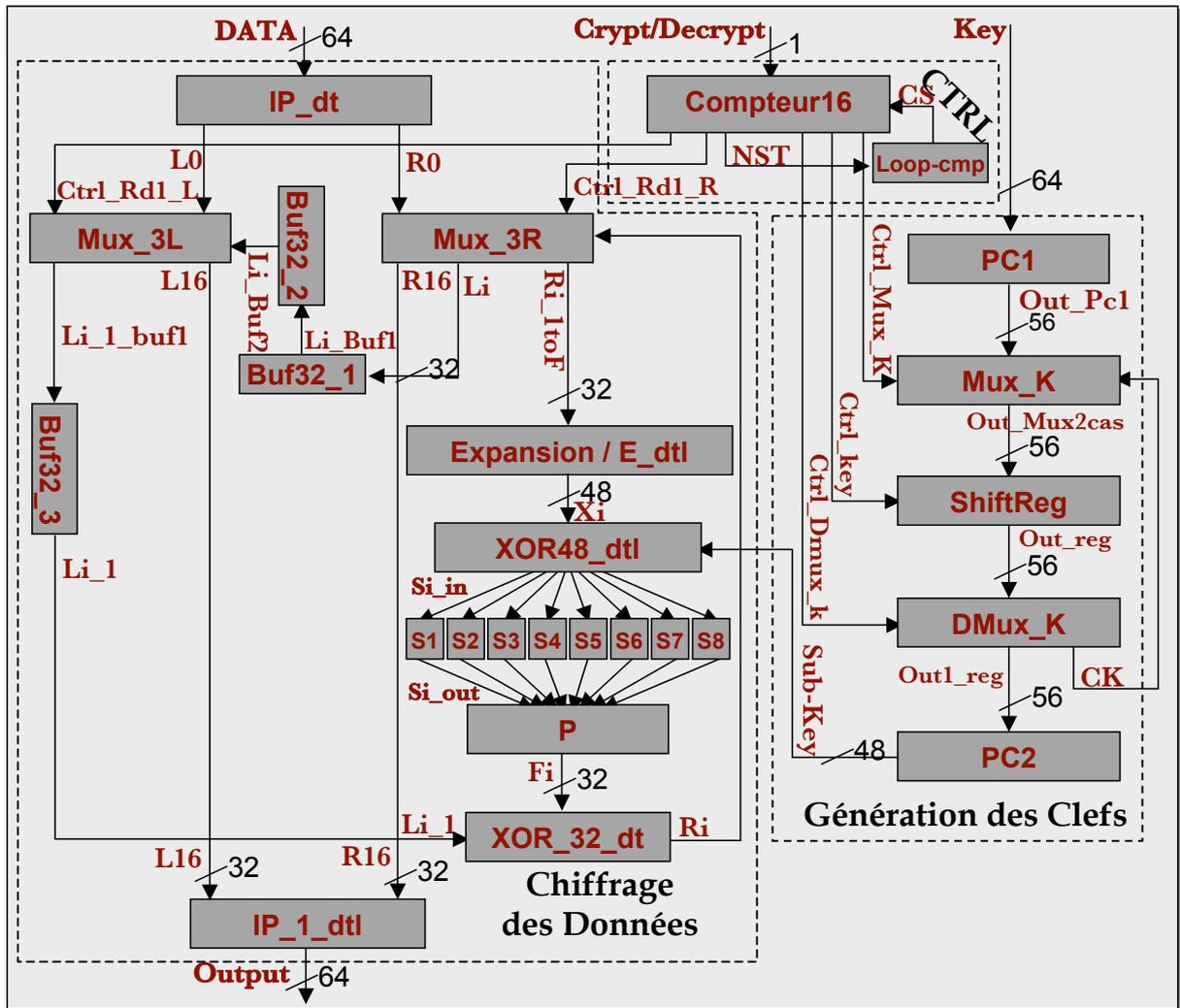


Figure 88 : Architecture en CHP du DES

### 6.3.2.3. Modélisation du DES en STEE

La spécification du DES en STEE a été générée automatiquement par notre environnement de validation. Le composant CHP a été traduit en système IF et chaque processus CHP en un processus IF. Comme pour le cas du Filtre, nous allons illustrer cette modélisation à travers le traitement de quelques processus CHP.

#### a) Le processus "xor32\_dtl"

```

PROCESS xor32_dtl
Port( Fi : IN DI passive DR[32] ;
      Li_1 : IN DI passive DR[32] ;
      Ri : OUT DI active DR[32])
Variable v_fi, v_li_1, v_ri : DR[32] ;
Begin [ Fi ? v_fi, Li_1 ? v_li_1 ;
        v_ri := v_fi XOR v_li_1 ;
        Ri ! v_ri ; loop ;
      ]
End ;
    
```

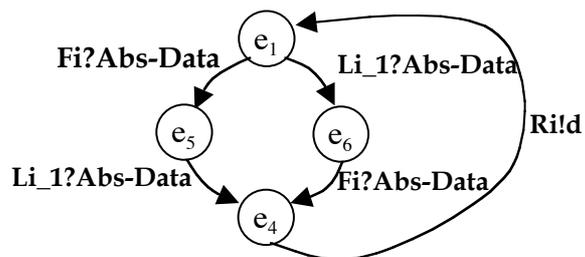


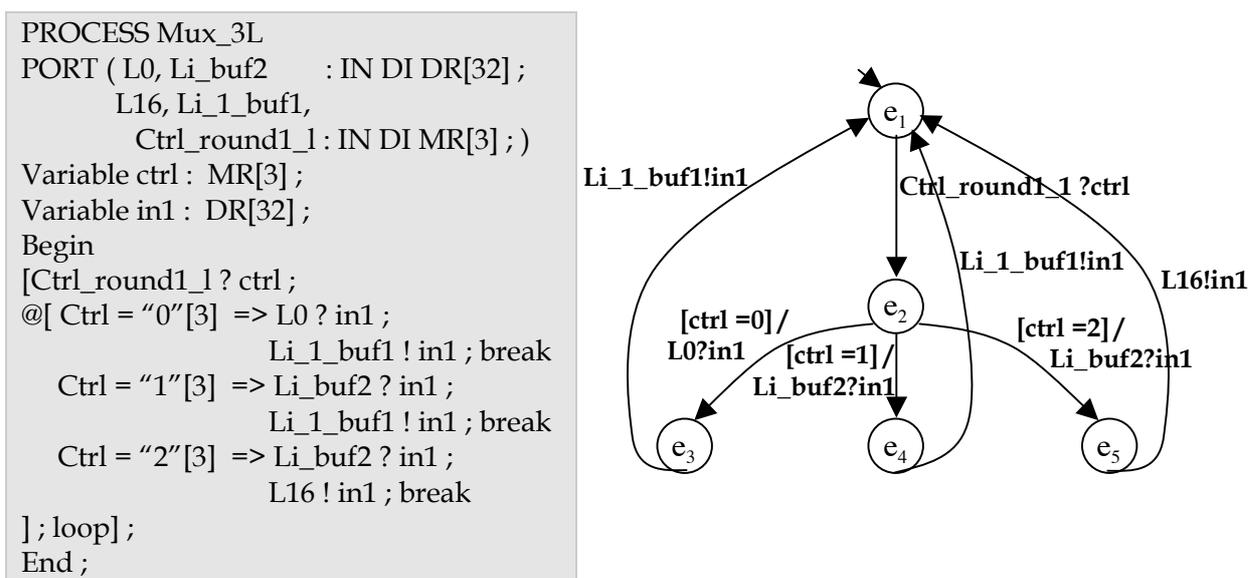
Figure 89 : Le processus xor32\_dtl et sa représentation en IF

Le processus *xor32\_dtl* réalise l'opération binaire XOR entre les deux ports : *Li\_1* et *Fi*. Les valeurs de ces deux ports sont lues d'une manière concurrente, ensuite le résultat du XOR entre *Li\_1* et *Fi* est écrit sur le port *Ri*.

**b) Le processus "Mux\_3L"**

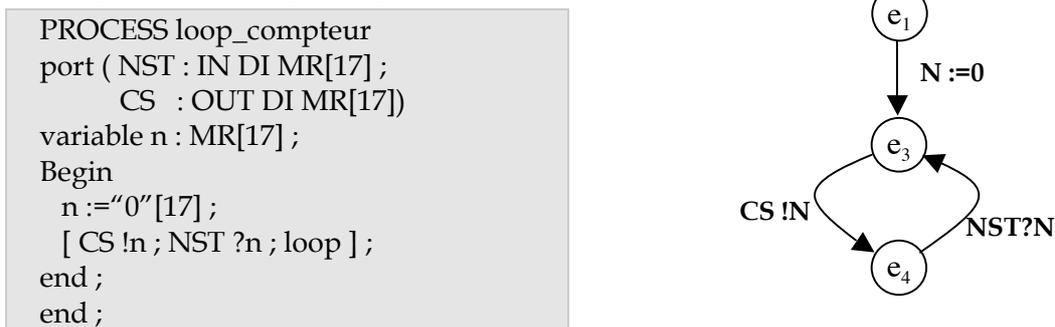
Dans ce processus, le canal de commande *Ctrl\_Round1\_L* est de type MR[3][1 ], c'est-à-dire un codage de données d'un parmi trois. Le canal de contrôle est lu dans la variable locale *Ctrl*. Selon la valeur de *Ctrl*, un des deux canaux (*L0*, *Li\_buf2*) est lu et sa valeur est écrite sur le canal *L16* ou *Li\_1\_buf1*.

Le programme CHP ainsi que sa représentation graphique en programme IF sont montrés sur la Figure 90.



**Figure 90 :** Le processus *Mux\_3L* et sa représentation en IF

**c) Le processus "loop\_compteur"**



**Figure 91 :** Le processus *loop\_compteur* et sa représentation en IF

Ce processus initialise le compteur avec la valeur "0", et à chaque fois il récupère la valeur suivante du port *NST* qui devient la valeur courante du compteur lors de la prochaine boucle. Ce processus permet d'éviter la mémorisation introduite par l'utilisation d'un seul processus pour modéliser le bloc contrôleur.

**d) Le processus "compteur"**

Le processus compteur est le processus chargé de la commande, il forme avec le processus loop\_compteur le bloc contrôle (voir la Figure 92). Dans ce processus, on teste la valeur du port *Decrypt* qui détermine si nous voulons crypter ou décrypter le texte récupéré sur le port *DATA*. Selon la valeur du port *Decrypt*, on déclenche une structure de choix sur la valeur courante *CS* du compteur. Suivant cette valeur, on exécute concurremment un certain nombre de commandes, qui servent généralement à déclencher les processus des deux blocs : Gestion des clefs et déchiffrement des données.

Vu que le processus est très grand, nous présentons dans les figures suivantes une partie du processus et sa représentation en IF.

```

PROCESS Compteur
Port ( Decrypt      : IN DI MR[2];
      CTRL_KEY      : OUT DI MR[5];
      CTRL_ROUND1_R, CTRL_ROUND1_L : OUT DI MR[3];
      CTRL_MUX_K, CTRL_DMUX_K : OUT DI DR[1];
      CS, NST : IN DI MR[17]; )

Variable n : MR[17];
Variable cd: MR[2];

Begin
[Decrypt ?cd ;
@[cd="1"=>[CS?n;
    @[n="0"[17]=>CTRL_KEY!"1"[5],CTRL_MUX_K!'0',CTRL_DMUX_K!'0',
        CTRL_ROUND1_R!"1"[3], CTRL_ROUND1_L!"1"[3],NST!"1"[17];break
        ... ..
    n="16"[17]=>CTRL_ROUND1_R!"2"[3],CTRL_ROUND1_L!"2"[3],
        NST!"0"[17];Decrypt?; break
    ]; break ]; break

cd="0"=> [CS?n;
    @[n="0"[17]=>CTRL_KEY!"0"[5],CTRL_MUX_K!'0',CTRL_DMUX_K!'0',
        CTRL_ROUND1_R!"1"[3], CTRL_ROUND1_L!"1"[3],NST!"1"[17];break
        ... ..
    n="16"[17]=>CTRL_ROUND1_R!"2"[3],CTRL_ROUND1_L!"2"[3],
        NST!"0"[17]; break
    ]; break ]; break

]; loop ];
End ;

```

**Figure 92** : Processus *Compteur* en CHP

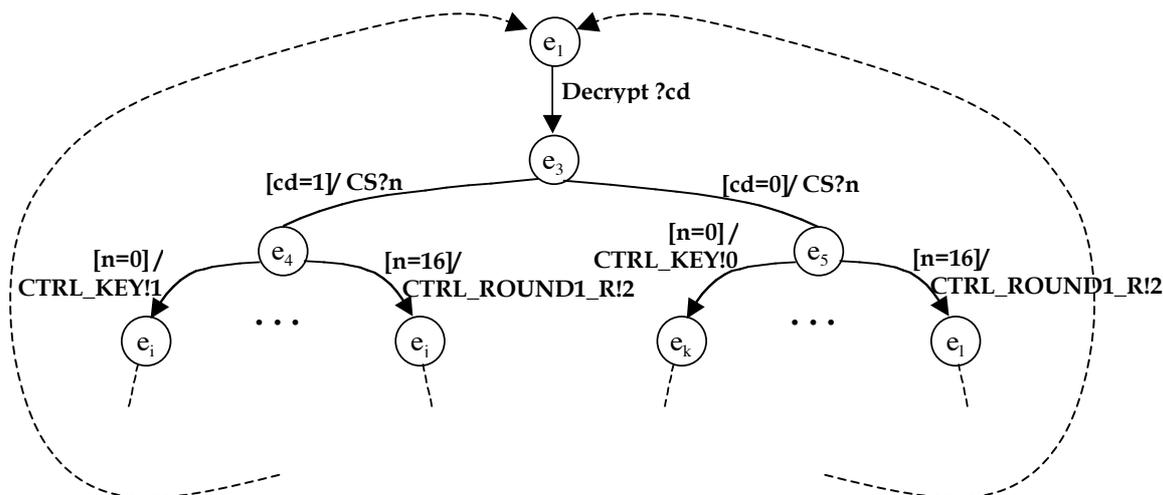


Figure 93 : Représentation en IF du processus Compteur

### 6.3.2.4. Vérification de quelques propriétés

Des propriétés caractéristiques du comportement du DES ont été exprimées en  $\mu$ -calcul et automatiquement vérifiées en utilisant CADP, sur un serveur SUN ultra 250 avec 1.6 GB de mémoire vive.

Le temps de génération du modèle STE est 1h :05mn :27.02 et sa taille est : (2.9 e+7 transitions, 5.2 e+6 états). La signification des propriétés et les performances d'exécution (temps de vérification exprimé en h :min :sec ; la mémoire en Mega-Bytes) sont énumérées dans le Tableau 7.

	Propriété	Temps de vérif.	Mémoire
P1	Absence de l'inter-blocage	27 :41.23	884 MB
P2	Après la réception des 3 entrées ( <i>Key</i> , <i>DATA</i> , <i>Decrypt</i> ), la sortie est toujours produite	27 :31.93	865 MB
P3	Le compteur du contrôleur compte correctement	26 :25.64	885 MB
P4	A chaque itération, les deux blocs (bloc de déchiffrement de données et le bloc des clés) se synchronisent correctement.	26 :25.65	879 MB

Tableau 7 : Résultats de la vérification de quelques propriétés temporelles

### 6.3.2.5. Vérification par réduction

Pour vérifier les propriétés P3 et P4 qui ne dépendent que du bloc de contrôle et de certains canaux de synchronisation, une technique alternative est disponible. Le modèle de comportement est réduit en cachant toutes les étiquettes qui ne se relient pas au bloc CTRL et au canal Sub-Key. Ceci peut être obtenu en appliquant des techniques de réduction et de préservation d'équivalence ce qui, pour ce modèle, prend un temps de calcul de 11 :46.31 et consomme 1.71 G. de mémoire.

Le modèle d'exécution en STE obtenu après réduction est présenté dans la Figure 94, une partie à l'intérieur du STE est omise pour le rendre plus lisible. Ce STE comporte 67 états et

102 transitions et montre le comportement cyclique ainsi que la synchronisation sur le canal Sub-Key.

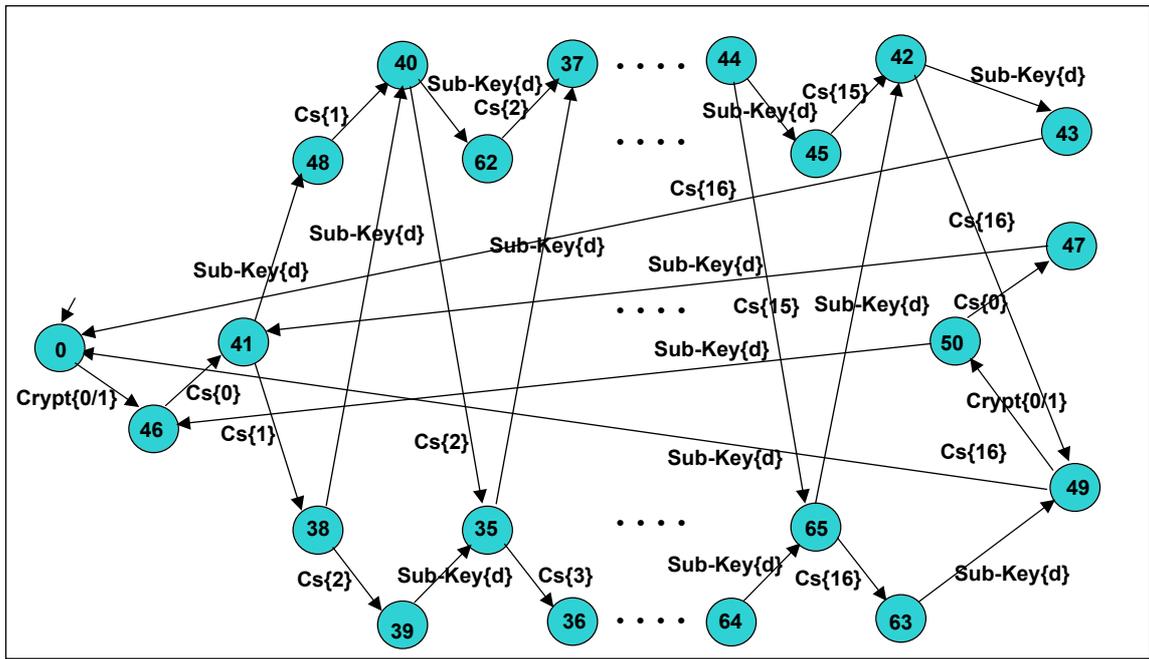


Figure 94 : Le STE réduit pour les propriétés P3 et P4



## 7. Conclusion et Perspectives

Dans le cadre de l'utilisation des méthodes formelles pour la validation et la vérification des systèmes concurrents asynchrones, le travail réalisé traite des spécifications exprimées en langage de haut niveau "CHP", et se restreint à une classe particulière de circuits asynchrones qui s'affranchit des problèmes de temporisations, appelés circuits asynchrones insensibles aux délais. L'objectif premier de ce travail était l'introduction des méthodes et techniques formelles dans le flot de conception asynchrone TAST. Nous avons proposé des méthodes cohérentes et pratiques et les avons implémentées dans des prototypes.

Les contributions de nos travaux de recherche sont à la fois théoriques et pratiques :

- ✓ Proposition d'une sémantique en langage VHDL pour les réseaux de Petri représentant les spécifications décrites initialement en CHP. Cette sémantique est la base d'un prototype de traducteur Pnet2VHDL, développé pour la vérification symbolique des spécifications de circuits asynchrones *pseudo-synchronisés*. Ce prototype a fait l'objet d'une démonstration dans la conférence DATE 2003.
- ✓ Proposition d'une sémantique en termes de systèmes de transitions étiquetées étendus (STEE) pour le langage CHP.
- ✓ Développement d'un environnement de validation de circuits asynchrones à base de méthodes énumératives de vérification. Cet environnement, basé sur la sémantique en STEE du langage CHP, intègre diverses techniques automatiques de réduction et d'abstraction. L'environnement CHP2IF a fait lui aussi l'objet d'une démonstration dans la conférence DATE 2004.
- ✓ Evaluation des deux approches de vérification : *pseudo-synchrone* avec des techniques symboliques de vérification formelle et *purement asynchrone* avec des techniques énumératives de vérification formelle. Cette étude nous a permis de bien identifier les problèmes des deux approches, et plus particulièrement le problème de l'explosion d'états concernant l'approche purement asynchrone qui s'aggrave lorsqu'on augmente le parallélisme.
- ✓ L'intégration dans l'environnement de validation CHP2IF d'une fonctionnalité de vérification formelle de l'exclusion mutuelle entre les gardes d'une structure de choix déterministe.
- ✓ Enfin, la vérification de quelques réels circuits asynchrones : des arbitres asynchrones, un filtre RIF quatre étapes et un circuit de cryptage/décryptage de données DES. La vérification formelle de ce dernier circuit (DES) a apporté une assurance et une qualité supplémentaires permettant ainsi au concepteur de procéder enfin à l'étape de fabrication.

L'ensemble de ces réalisations débouche sur la mise en place d'une plateforme pour la validation de spécifications de circuits asynchrones écrites dans le langage CHP. Cette plateforme, composée des deux environnements de validation, constitue une base solide pour une intégration efficace et performante des méthodes formelles dans le flot de conception asynchrone TAST.

D'importantes perspectives peuvent être envisagées : Etendre les deux environnements de vérification pour supporter toutes les structures et primitives du langage CHP, telles que les opérateurs signés, est une continuité naturelle pour ce travail.

Il est aussi important de consolider l'approche *pseudo-synchrone* par le développement et l'intégration de techniques automatiques de réduction et d'abstraction. Les techniques de simulation symbolique ont fait leurs preuves dans le domaine de la vérification formelle de circuits digitaux synchrones. Les introduire en liaison avec les méthodes formelles permet de s'affranchir du problème de l'explosion d'états. En effet, la représentation des signaux de données par des symboles permet une vérification formelle des aspects "*contrôle*" sans avoir besoin de traiter toutes les valeurs possibles des signaux de données.

Une première tentative de vérifications après synthèse a été mentionnée dans le 3<sup>ème</sup> chapitre du manuscrit. Malheureusement, la vérification des circuits à ce niveau reste une tâche difficile. Une façon de procéder serait d'analyser les transformations effectuées lors de la synthèse pour trouver des correspondances entre le circuit avant et après synthèse. Une relation de raffinement pourrait alors être construite, permettant de vérifier la correction du circuit synthétisé par rapport à sa spécification.

Nous pensons enfin à la vérification formelle de circuits asynchrones plus complexes, comme les circuits globalement asynchrones et localement synchrones.

# Bibliographie

- [AAB99] P. A. Abdulla, A. Annichini, S. Bensalem, A. Bouajjani and al. "Verification of Infinite-State Systems by Combining Abstraction and Reachability Analysis" In Nicolas Halbwachs, Doron Peled (Eds.) Proceedings of 11th Conference on Computer Aided Verification, CAV'99, Trento, Italy LNCS vol. 1633 Springer-Verlag July 1999
- [ABO98] R. Airiau, J-M Bergé, V. Olive, J. Rouillard. "VHDL – Langage, modélisation, synthèse". 2<sup>nd</sup> edition, Presse Polytechniques et Universitaires Romandes, 1998 (in French).
- [ABR01] A. Abrial, J. Bouvier, M. Renaudin, P. Senn and P. Vivet "A New Contactless Smart Card IC using On-Chip Antenna and Asynchronous Microcontroller". Journal of Solid-State Circuits, vol. 36, 2001, pp. 1101-1107.
- [BBD02] D. Borrione, M. Boubekur, E. Dumitrescu, M. Renaudin, et al. "Introducing formal validation in an asynchronous circuit design flow", The Fourth International Workshop on Designing Correct Circuits, Grenoble, France, April 6-7, 2002.
- [BBD03] D. Borrione, M. Boubekur, et al. "An approach to the Introduction of Formal Validation in an Asynchronous Circuit Design Flow". In Jr. R. H. Sprague, editor. Proceedings of the 36th Hawaii International Conference on Systems Science, Hawaiï, USA. IEEE Computer Society, January 6-9, 2003.
- [BBE97] I. Beer, S. Ben-David, C. Eisner, D. Geist, L. Gluhovsky, T. Heyman, A. Landver, P. Paanah, Y. Rodeh, G. Ronin, and Y. Wolfsthai. "RuleBase : Model checking at IBM". In Orna Grumberg, editor, Proc. 9 th Intl. Conference on Computer Aided Verification (CAV'97), volume 1254 of Lect. Notes in Comp. Sci., pages 480-483. Springer-Verlag, 1997.
- [BBi96] Kees van Berkel, Arjan Bink ; "Single-track handshaking signaling with application to micropipelines and handshake circuits" ; Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pp. 122-133 ; 1996.
- [BBM03a] M. Boubekur, D. Borrione, L. Mounier, M. Renaudin, A. Sirianni. "Modelling CHP descriptions in Labelled Transition Systems for an efficient formal validations of asynchronous circuit specifications". In Forum on Specification and Design Language (FDL'03), Frankfurt, Germany September 2003.
- [BBM03b] Dominique Borrione, Menouer Boubekur, Laurent Mounier, Marc Renaudin, Antoine Sirianni. "Validation of asynchronous circuit specifications using IF/CADP". In IFIP Intl. Conference on VLSI, Darmstadt, Germany December 2003.
- [BBM04] M. Boubekur, D. Borrione, et al. "Languages for System Specifications. Selected Contributions on UML, SystemC, System Verilog, Mixed-Signal Systems, and Property Specifications from FDL'03". Chapter : " Modelling CHP descriptions in Labelled Transition Systems for an efficient formal validations of asynchronous circuit specifications" Ed. Grimm, Christoph. Kluwer Academic Publishers, 2004. ISBN: 1-4020-7991-5
- [BCC99] A. Biere, A. Cimatti, E. Clarke, M. Fujita and Y. Zhu. "Symbolic Model Checking Without BDDs". Lecture Notes in Computer Science, 1579 :193-207, 1999.
- [BE97] A. Bardsley, D. Edwards. "Compiling the language Balsa to delay-insensitive hardware". In C.D. Kloos and E. Cerny, editors, Hardware description languages and their applications (CHDL), pp. 89-91, April, 1997.
- [BEd00a] A. Bardsley and D. A. Edwards. "The Balsa asynchronous circuit synthesis system". In Forum on Design Languages, September 2000.

- [BEd00b] A. Bardsley and D. A. Edwards. "Synthesising an asynchronous DMA controller with Balsa". *Journal of Systems Architecture*, 46 : 1309-1319, 2000.
- [BEd97] Bardsley A., Edwards D., "Compiling the language Balsa to delay-insensitive hardware", *Hardware description languages and their applications (CHDL)*, C.D. Kloos and E. Cerny, editors, , pp. 89-91, April, 1997.
- [Bel99] Wendy A. Belluomini. "Algorithms for Synthesis and Verification of Timed Circuits and Systems". PhD thesis, The university of Utah, Utah, 1999.
- [Bell] Bell Labs Design Automation. FormalCheck. [www.bell-labs.com/formalcheck](http://www.bell-labs.com/formalcheck).
- [Ber92] Kees van Berkel. "Beware the isochronic fork. *Integration, the VLSI journal*, 13(2) :103-128, June 1992.
- [Ber93] K. Van Berkel, "Handshake Circuits – An Asynchronous Architecture for VLSI Programming", Cambridge University Press, 1993, ISBN : 0-521-45254-6
- [BFG00] M. Bozga, J.Cl. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, L. Mounier. "IF : A Validation Environment for Timed Asynchronous Systems". *Proceedings of CAV'00 (Chicago, USA) July 2000*
- [BFG99] M. Bozga, J.Cl. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, L. Mounier. "IF : An Intermediate Representation and Validation Environment for Timed Asynchronous Systems" *Proceedings of FM'99 (Toulouse, France) September 1999*.
- [BGM01] Marius Bozga, Susanne Graf, and Laurent Mounier. "Automated validation of distributed software using the IF environment". In Scott D. Stoller and Willem Visser, editors, *Workshop on Software Model-Checking, associated with CAV'01 (Paris, France) July 2001*. Volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science
- [BJM02] M. Bozga, H. Jianmin , O. Maler, S.Yovine, "Verification of Asynchronous Circuits using Timed Automata", *Proc. TPTS'02 Workshop*, Elsevier Science Pub. April 2002.
- [BKR91] K. van Berkel, J. Keyssels, M. Ronken, R. Saeijs and F. Chaliq "The VLSI programming language Tangram and its translation into handshakes circuits". *Proceedings of the European Conference on Design Automation, Amsterdam*, pp. 384-389, February 1991.
- [BLa00] I. Blunno and L. Lavagno, "Automated synthesis of micro-pipelines from behavioral Verilog HDL", *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, Eilat, Israel, April 1-6, 2000*.
- [BNe01] Robert Berks and Radu Negulescu. "Partial-Order Correctness-Preserving of Delay-Insensitive Circuits". *Seventh International Symposium on Asynchronous Circuits and Systems, Salt Lake City, Utah, 2001*.
- [Bou03] M. Boubekeur, "Étude de cas : vérification formelle d'un filtre asynchrone à l'aide de techniques de model-checking énumératif ". *6th Journées Nationales du Réseau Doctoral de Microélectronique, JNRDM' 03, Toulouse, 14-16 Mai 2003*.
- [Bou85] G. Boudol. "Le calcul Meije". In J.P Verjus et G. Roucairo, editor, *Parallélisme, Communication, Synchronisation*. CNRS, 1985.
- [Bra95] R. K. Brayton et al. *VIS : A system for verification and synthesis*. Technical Report UCB/ERL M95/104, Electronics Research Lab, Univ. Of California, December 1995.
- [Bry86] E. Bryant. "Graph-based algorithms for boolean function manipulation". *IEEE Transactions on Computers*, C-35(8), 1986.
- [Bry92] R. E. Bryant. "Symbolic boolean manipulation with ordered binary decision diagrams". *ACM Computing Surveys*, 1992.
- [CADP] <http://www.inrialpes.fr/vasy/cadp/>
- [CEm81] E. M. Clarke and E. A. Emerson. "Design and Synthesis of Synchronisation Skeletons Using Branching Time Temporal Logic". In *Logic of Programs : Workshop*, number 131 in LNCS, Newyork, 1981. Springer.

- [CES83] E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications : A practical Approach". In Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages, pages 117-126. Austin, 1983.
- [CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications". ACM Transactions on Programming Languages and Systems, 8(2) :244-263, April 1986.
- [CGP99] Edmund M. Clarke, Orna Grumberg and Doron A. Peled. "Model Checking". The MIT Press, Cambridge, Massachusetts, 1999.
- [Chu87] Tam-Anh Chu. "Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications". PhD thesis, MIT Laboratory for Computer Science, June 1987
- [CKK02] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A Yakovlev. "Logic Synthesis of Asynchronous Controllers and Interfaces". Springer-Verlag, 2002
- [CKK97] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A Yakovlev. "Petrify : a tool for manipulating concurrent specifications and synthesis of asynchronous controllers". IEICE Transactions on information and Systems, E80-D(3) :315-325, March 1997.
- [Cla67] Wesley A. Clark. "Macromodular computer systems". In AFIPS Conference Proceedings ; 1967 Spring Joint Computer Conference, volume 30, pages 335-336, Atlantic City, NJ, 1967, Academic Press.
- [CMi00] Antonio Cerone, George Milne. "A Methodology for the Formal Analysis of Asynchronous Micropipelines". Proc. FMCAD 2000, LNCS N° 1954, Springer Verlag, pp.246-262
- [DCS93] A. Davis, B. Coates, K. Stevens, "The post office experiment : Designing a large asynchronous chip", Proc 26th annu. Hawaii Int. Conf. On Systems Sciences, vol. I, pp. 409-418, 1993.
- [DCS93b] A. Davis, B. Coates, K. Stevens, "Automatic Synthesis of fast compact asynchronous control circuits". In S. Furber and M. Edwards, editors Asynchronous Design Methodologies, volume A-28 of IFIP Transactions, pages 193-207. Elsevier Science Publishers, 1993.
- [Dea92] Mark E. Dean. "STRiP : A self Timed RISC Processor Architecture". PhD thesis, Stanford University, 1992.
- [Deh93] A. Déharbe. "Model Checking on Finite State Machines : Extensions and Applications to VHDL designs". In Proceedings of the First Asian-Pacific Conference on Hardware Description Languages : Standards and Applications, Brisbane, Australia, 1993.
- [Deh96] A. Déharbe. "Vérification Formelle de Propriétés Temporelles : Etude et Application au Langage VHDL". PhD thesis, Université Joseph Fourier. Grenoble 1, 1996.
- [DES] NIST, "Data Encryption Standard (DES)", FIPS PUB 46-3, National Institute of Standards and Technology, Reaffirmed 1999 October 25. <http://csrc.nist.gov/csrc/fedstandards.html>
- [DFR02] Anh Vu Dinh Duc, L. Fesquet, M. Renaudin, "Synthesis of QDI Asynchronous Circuits from DTL-style Petri-Net" IWLS-02, 11th IEEE/ACM Internat. Workshop on Logic & Synthesis, New Orleans, Louisiana, June 4-7, 2002.
- [Dij76] E. W. Dijkstra, "A Discipline of Programming, Prentice Hall", Englewood Cliffs, N.J. 1976
- [Din03] Anh Vu Dinh Duc, "Synthèse automatique de circuits asynchrone QDI", PhD thesis, INP of Grenoble, 2003
- [DOd93] A. Debreil and P.Oddo. "Synchronous designs in VHDL". In proceedings of the European Design Automation Conference (EURO-DAC), pages 486-491. IEE CS Press, 1993.

- [DPo99] G. Delzanno and A. Podelski, "Model Checking in CLP". Proc. 5th Int. Conf. TACAS'99. R. Cleaveland, ed., Springer Verlag LNCS N°1579, pp.223-239,1999.
- [Dum03] Emil. DUMITRESCU. "Construction de Modèles Réduits et Vérification Symbolique de Circuits Industriels Décrits au Niveau RTL ". PhD thesis, Université Joseph Fourier. Grenoble 1, 2003.
- [EVi89] P. H. J. van Eijk, C. A. Vissers, M. Diaz. "The formal description technique LOTOS", Elsevier Science Publishers B.V., 1989.
- [FBr96] Karl M. Fant and Scott A Brandt. "NULL Conventional Logic : A complete and consistent logic for asynchronous digital circuit synthesis". In International Conference on Application-specific Systems, Architectures, and Processors, pages 261-273, 1996.
- [Fer89] J.C. Fernandez. ALDEBARAN : A Tool for Verification of Communicating Processes. Rapport technique SPECTRE C14, Laboratoire de Génie Informatique. Institut IMAG, Grenoble, September 1989.
- [FJJ92] J.C. Fernandez, C. Jard, T. T. Jeron and L. Mounier. "On-the-fly Verification of Finite Transition Systems. In Formal Methods in System Design, 1992.
- [FKM93] J.C. Fernandez , A. Kerbrat and L. Mounier. "Symbolic Equivalence Checking". Proceedings of the 5th Workshop on Computer-Aided Verification (Heraklion, Greece), June 1993
- [FLa79] M. J. Fischer and R. E. Ladner. "Propositional Dynamic Logic of Regular Programs". Journal of Computer and System Sciences, (18) :194-211, 1979.
- [FMo91] J.C. Fernandez and L. Mounier. "On-the-fly Verification of Behavioural Equivalences and Preorders". In CAV'91, volume 575 of LNCS, pages 181—191. Springer—Verlag, 1991.
- [FNT99] R.M. Fuhrer, S.M. Nowick, M. Theobald, N.K. Jha, B. Lin, and L. Plana. "Minimalist : An environment for the synthesis, verification and testability of burst-mode asynchronous machines". Technical Report CUCS-020-99, Columbia University, July 1999.
- [Gar89b] Hubert Garavel. "Compilation of LOTOS Abstract Data Types". In Son T. Vuong, editor, Proceedings of the 2<sup>nd</sup> International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., France), pages 147{162. North-Holland, December 1989.
- [Gar94] Hubert Garavel. "Binary Coded Graphs. Definition of the BCG Format (version 1.0)". Rapport interne, INRIA Rhône-Alpes, Grenoble, 1994.
- [Gar98] H. Garavel. "OPEN/CAESAR : An Open Software Architecture for Verification, Simulation, and Testing". In Proceedings of 1st International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98, Lecture Notes in Computer Science, Berlin, March 1998. Springer Verlag.
- [GJM97] H. Garavel, M. Jorgensen, R. Mateescu, C. Pecheur, M. Sighireanu, and B. Vivien. "CADP'97 : Status, Applications and Perspectives". In Ignac Lovrek, editor, Proceedings of the 2<sup>nd</sup> COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia), June 1997.
- [GKP94] R. L. Graham, D. E. Knuth, and O. Patashnik, "Binomial Coefficients." Ch. 5 in Concrete Mathematics : A Foundation for Computer Science, 2<sup>nd</sup> ed. Reading, MA : Addison-Wesley, pp. 153-242, 1994.
- [Gla90] R.J. van Glabbeek. "The Linear Time – Branching Time Spectrum". CS R9029, Centrum voor Wiskunde en Informatica, Amsterdam, 1990.
- [GLo93] S. Graf and C. Loiseaux. " A Tool for Symbolic Program Verification and Abstraction In C. Courcoubetis (Eds.) Workshop on Computer-Aided Verification, CAV93, Heraklion, Crete LNCS vol. 697 Springer-Verlag June 1993
- [GWe89] R.J van Glabbeek and W. P. Weijland. "Branching-Time and Abstraction in Bisimulation Semantics". CS R8911, Centrum voor Wiskunde en Informatica,

- Amsterdam, 1989. also in proc. IFIP 11th World Computer Congress, San Francisco, 1989.
- [GWO94] P. Godefroid and P. Wolper. “A Partial Approach to Model Checking”. *Information and Computation*, 110 :305-326, 1994.
- [Hau95] Hauck S., “Asynchronous Design Methodologies : An Overview”, *Proceeding of the IEEE*, Vol. 83, N° 1, pp. 69-93, January, 1995.
- [Hoa78] C.A.R. Hoare. “Communicating Sequential Processes”. *Communications of the ACM*, vol. 8, pp. 666-677, Aug 1978.
- [Hol97] G.J. Holzmann. “The model checker SPIN”. *IEEE Trans. on Software Engineering*, 23(5):279--295, 1997.
- [Hor01] S. Höreth. “A Word-level Graph Manipulation Package”. *Software Tools for Technology Transfers*, 3(2) :182-192, 2001.
- [HPu99] G. J. Holzmann and A. Puri. A minimized automaton representation of Etats atteignables . *Software Tools for Technology Transfer*, 2(3):270--278, November 1999.
- [HTu00] Ji He and K. J. Turner. “Verifying and testing asynchronous circuits using LOTOS”. In T. Bolognesi and D. Latella, editors, *Proc. Formal Methods for Distributed System Development (FORTE XIII/PSTV XX)*, pages 267—283, London, UK, Oct. 2000. Kluwer Academic Publishers.
- [Huf54] HUFFMAN, D. A., “The Synthesis of Sequential Switching Circuits”, *Journal of the Franklin Institute*, 257, nos. 3 and 4, March and Apr. 1954, 294—295.
- [IE00a] IEEE Computer Society. “IEEE Standard VHDL Language Reference Manual”, 2000.
- [IE00b] IEEE Computer Society W.G.1076.6. “IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis”. IEEE, 2000.
- [IEE01] IEEE Computer Society. “IEEE 1364 Verilog Language Reference Manual”, 2001.
- [IF] <http://www-verimag.imag.fr/~async/IF/index.shtml>.
- [ISO87] ISO. “LOTOS : A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour”. Draft International Standard 8807, International Organization for Standardization. Information Processing Systems. Open Systems Interconnection, Genève, July 1987.
- [ISO88] ISO. “ESTELLE : A Formal Description Technique Based on an Extended State Transition Model”. International Standard 9074, International Organization for standardization. Information Processing Systems. Open Systems Interconnection, Genève, September 1988.
- [JFu2] Mark B. Josephs and Dennis P. Furey. “Asynchronous design using the DISP programming language and the tools di2pn and petrify”. *Second Acid-WG Workshop of the European Commission’s Fifth Framework Programme*, Munich, Germany, 2002.
- [KCa87] Kleeman L., Cantoni A., “Metastable behavior in digital systems”, *IEEE Design & Test of Computers*, December 1987, pp. 4-19.
- [KCK95] A. Kondratyev, J. Cortadella, M. Kishinevsky, et al., “Checking signal transition graph implementability by symbolic bdd traversal”. *Proc. EDTC’95*, pp 325-332, Paris, March 95.
- [KKr67] G. Kreisel and J. L. Krevine. “Elements of Mathematical Logic”. North-Holland Pub. Co., 1967.
- [KKT94] Kishinevsky M. A., Kondratyev A. K., Taubin A. R., Varshavsky V. I., “Concurrent Hardware, The Theory and Practice of Self-Timed Design”, *Wiley Series in Parallel Computing*, Chichester, 1994.
- [KKT94] A. Kishinevsky, A. K. Kondratyev, A. R. Taubin, V. I. Varshavsky, “Concurrent Hardware, The Theory and Practice of Self-Timed Design”, *Wiley Series in Parallel Computing*, Chichester, 1994.
- [KMa99] Joep Kessels and Paul Marston. “Designing asynchronous standby circuits for a low-

- power pager". Proceedings of the IEEE, 87(2) :257-267, February 1999.
- [KMP95] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman and David Wonnacott. "The Omega Library interface guide". Rapport No. UMIACS-TR-95-41. Univ. Of Maryland Institute for Advanced Computer Studies. 1995.
- [Koz83] D. Kozen. "Results on the Propositional  $\mu$ -calculus". Theoretical Computer Science, 27 :333-354, 1983.
- [Kuk96] Yuji Kukimoto. "Blif-mv". Technical report, The VIS Group, University of California, Berkeley, 1996.
- [Lam77] L. Lamport. "Proving the correctness of multiprocess programmes". IEE Transactions on Software Engineering, SE-3(2) :125-143, 1977.
- [Lam80] L. Lamport. "Sometime is Sometimes Not Never". On the Temporal Logic of Programs. In Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages POPL '80 (Las Vegas, Nevada), pages 163-173, January 1980.
- [LFS00] Michel Ligthart, Karl Fant, Ross Smith, Alexander Taubin, and Alex Kondratyev. "Asynchronous design using commercial HDL synthesis tools". In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), pages 114-125. IEEE Computer Society Press, April 2000.
- [Mad90] Jean-Christophe Madre "Un Outil de Vérification Formelle de Circuits Digitaux". PhD thesis, Telecom Paris. Paris, 1990.
- [Mar86] Alain J. Martin. "Compiling communicating processes into delay-insensitive VLSI circuits". Distributed Computing, 1(4) :226-234, 1986.
- [Mar90] A.J. Martin, "Programming in VLSI : from communicating processes to delay-insensitive circuits", in C.A.R. Hoare, editor, Developments in Concurrency and Communication, UT Year of Programming Series, 1990, Addison-Wesley, p. 1-64.
- [Mar90b] Alain J. Martin, "The limitations to delay-insensitivity in asynchronous circuits", Proceedings of the sixth MIT conference on Advanced research in VLSI, p.263-278, March 1990
- [Mar93] A.J. Martin. "Synthesis of Asynchronous VLSI Circuits". Internal Report, Caltech-CS-TR-93-28, California Institute of Technology, Pasadena, 1993.
- [Mat03] Radu Mateescu . "On-the-Fly Verification using CADP". Proceedings of the 8th International Workshop on Formal Methods for Industrial Critical Systems, FMICS2003 (Trondheim, Norway), June 2003
- [May90] May, D., "Compiling occam into silicon", in Developments in Concurrency and Communication, CAR Hoare, Ed., Addison-Wesley, pp. 87-129, 1990.
- [MBa59] D.E. Muller and W.S. Bartky. "A theory of asynchronous circuits". Annals of the Computation Laboratory of Harvard University. Volume XXIX : Proceedings of an International Symposium on the Theory of Switching, Part I, pages 204—243, 1959.
- [MBL89] Martin A. J., Burns S. M., Lee T. K., Borkovic D., and Hazewindus P. J., "The Design of an Asynchronous Microprocessor", Advanced Research in VLSI : Proceedings of the Decennial Caltech Conference in VLSI, Charles L. Seitz editor, , pp. 351-373, MIT Press, 1989.
- [MBM89] T.H.-Y. Meng, R.W. Brodersen, and D.G. Messerschmitt, "Automatic synthesis of asynchronous circuits from highlevel specifications", IEEE Transactions on Computer-Aided Design, Vol. 8, No. 11, pp. 1185-1205, Nov. 1989.
- [Mca92] McAulley A.J., "Four state asynchronous architectures", IEEE transactions on computers, Volume 41, N° 2, pp 129 –142, Feb. 1992.
- [McM92] Kenneh L. McMillan. "Using unfolding to avoid the state explosion problem in the verification of asynchronous circuits". In GG. V. Proc. International Workshop on computer Aided verification, volume 663, pages 164-177. Spriger-Verlag, 1992.

- [Mea55] George H. Mealy, "A method for synthesizing sequential circuits", Bell System Technical Journal, 34(5) :1045-1079, 1955.
- [Men91] Teresa H.-Y. Meng, "Synchronization Design for Digital Systems". Kluwer Academic Publishers, 1991
- [MFR85] Molnar C. E., Fang T. P., Rosenberg F. U., "Synthesis of delay-insensitive modules", Chapel Hill conference on VLSI, computer science press , pp 67 –85, 1985.
- [MGa98] R. Mateescu and H. Gavel. "XTL: A Meta-Language and Tool for Temporal Logic Model-Checking". Proceedings of the International Workshop on Software Tools for Technology Transfer STTT'98 (Aalborg, Denmark), July 1998.
- [Mil65] R.E. Miller. "Sequential Circuits and Machines". Volume 2 of Switching Theory . Wiley, 1965.
- [Mil80] R. Milner. "A Calculus of Communicating Systems". Volume 92 of LNCS, Berlin, 1980. Springer Verlag.
- [Mil89] Robin Milner. "Communication and Concurrency". Prentice-Hall, 1989. ISBN.
- [MLM97] Alain Martin, Andrew Lines, Rajit Manohar, Mika Nystroem, Paul Penzes, Robert Southworth, Uri Cummings , Tak Kwan Lee. "The Design of an Asynchronous MIPS R3000 Microprocessor". Proc. 17th Conference on Advanced Research in VLSI, 164-181, IEEE Computer Society Press, 1997.
- [MLM99] Rajit Manohar, Tak-Kwan Lee, and Alain J. Martin. "Projection : A Synthesis Technique for Concurrent Systems". Proc. 5th Int. Symposium on Advanced. Research in Asynchronous Circuits and Systems, April 1999.
- [MLM99] Rajit Manohar, Tak-Kwan Lee, and Alain J. Martin. Projection. "A Synthesis Technique for Concurrent Systems". Proc. 5th Int. Symposium on Advanced Research in Asynchronous Circuits and Systems, April 1999.
- [MMe92] Chris Myers and Teresa H.-Y. Meng. "Synthesis of Timed Asynchronous Circuits". In Proc. International Conf. Computer Design (ICCD), pages 279-282. IEEE Computer Society Press, October 1992
- [Mou92] L. Mounier. " Méthodes de Vérification de Spécifications Comportementales : étude et mise en oeuvre". L'Université de Joseph Fourier- GrenobleI, 1992.
- [MPn92] Z. Manna and A. Pnueli. "The Temporal Logic of Reactive and Concurrent Systems", volume I : Specification. Springer-Verlag, 1992.
- [MSi00] R. Mateescu and M. Sighireanu. "Efficient On-the-Fly Model-Checking for Regular Alternation-Free  $\mu$ -Calculus". Proceedings of the 5th International Workshop on Formal Methods for Industrial Critical Systems FMICS'2000 (Berlin, Germany), April 2000
- [NDi95] Nowick S.M., Dill D.L., "Exact two level minimization of hazard-free logic with multiple input changes", IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, vol. 14(8), pp. 986-997, August 1995.
- [Neg98] Radu Negulescu. "Process Spaces and Formal Verification of Asynchronous Circuits". University of Waterloo, Ontario, Canada, 1998.
- [Now93] S. M. Nowick, "Automatic Synthesis of Burst-Mode Asynchronous Controllers", PhD Thesis, Stanford University, Department of Computer Science, March 1993.
- [NVa90] R. De Nicola and F. W. Vaandrager. "Action versus State based Logics for Transition Systems". In Proceedings Ecole de Printemps on Semantics of Concurrency, volume 469 of Lecture Notes in Computer Science, pages 407{419. Springer Verlag, 1990.
- [Omega] <http://www.cs.umd.edu/projects/omega/>
- [Par81] David Park. "Concurrency and Automata on Infinite Sequences". In Peter Deussen, editor, Theoretical Computer Science, volume 104 of Lecture Notes in Computer Science, pages 167-183, Berlin, March 1981. Springer Verlag.
- [Pet62] C. A. Petri "Kommunikation mit Automaten". Phd Thesis, Bonn, Institut für

- Instrumentelle Mathematik, 1962
- [PLM03] G. Pace, F. Lang, and R. Mateescu . “Calculating Tau-Confluence Compositionally”. Proceedings of the 15th Computer-Aided Verification conference CAV’2003 (Boulder, Colorado, USA), July 2003
- [Pnu85] A. Pnueli. “In Transition from Global to Modular Temporal Reasoning about Programs”. In Logics and Models for concurrent Systems, 1985.
- [Pug92] William Pugh. The Omega test : a fast and practical integer programming algorithm for dependence analysis. Communications of the ACM, 8 :102-114, August 1992.
- [RAB95] R. K. Ranjan, A. Aziz, R. K. Brayton, B. F. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification. Presented at IWLS95, Lake Tahoe, CA., May 1995.
- [RCP96] Oriol Roig, Jordi Cortadella and Enric Pastor. “Verification of Asynchronous Circuits by BDD-based Model Checking of Petri Nets”. Polytechnic University of Catalunya, Barcelona, 1996.
- [RDR02] M. Renaudin, J.B. Rigaud, A. Dinhduc, A. Rezzag, A. Sirianni, J. Fragoso : “TAST CAD Tools”, ASYNC’02 TUTORIAL, ISRN : TIMA—RR-02/04/01—FR, 2002
- [Ren00] M. Renaudin, “Asynchronous Circuits and Systems : a promising design alternative”, in “MIGAS 2000”, special issue Microelectronics-Engineering Journal, Elsevier Science, Vol. 54, N° 1-2, December 2000, pp. 133-149.
- [Rig02] T.G. Rigau, “Spécification de Bibliothèques pour la Synthèse de Circuits Asynchrones”. PhD thesis, INP de Grenoble, 2002.
- [Roi97] Oriol Roig i Mansill. “Formal Verification and Testing of Asynchronous Circuits”. PhD thesis, Polytechnic University of Catalunya, Barcelona, 1997.
- [RVG00] Renaudin M., Vivet P., Geoffroy Ph., “ASPRO : a toy demo”, 4th AcID Workshop, Grenoble, France, 31st – 1st February, 2000.
- [RVR98] M. Renaudin, P. Vivet, F. Robin, “ASPRO-216 : a standard-cell Q.D.I. 16-bit RISC asynchronous microprocessor”, Proc. Of the Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC’98), San Diego – USA, 1998, p. 22-31.
- [RVR99] Renaudin M., Vivet P., Robin F., “A Design Frame Work for Asynchronous/ Synchronous Circuit Based on CHP to HDL Transaction”, International Symposium on Advanced Research in Asynchronous Circuits and Systems-ASYNC’98, Barcelona, Spain, April 19-21, pp 135-144, 1999.
- [RYa85] L. Y. Rosenblum and A.V. Yakovlev. “Signal graphs : from self-timed to timed ones”, Proc. Of the Int. Workshop on Timed Petri Nets, Torino, Italy, July 1985, IEEE Computer Society Press, NY, 1985, pp. 199-207.
- [SBr03] S.J. Silver and J.A. Brzozowski. “True Concurrency in Models of Asynchronous Circuit Behavior”. In International Journal of Formal Methods in System Design, Vol. 22, No. 3 (May 2003), pp. 183-203.
- [Sei70] Charles L. Seitz “Asynchronous machines exhibiting concurrency”, 1970. Record of the project MAC Concurrent Parallel Computation
- [SSL92] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton and A.L. Sangiovanni-Vincentelli, “SIS : A System for Sequential Circuit Synthesis”, Technical Report, U. C., Berkeley, May 1992.
- [Str82] R. Streett. “Propositional Dynamic Logic of Looping and Converse”. Information and Control, (54) :121-141, 1982.
- [Sut89] I.E. Sutherland. “Micro-pipelines”. Communication of the ACM, Volume 32, N°6, June 1989.
- [TLG03] F. Tronel, F. Lang and H. Garavel. “Compositional Verification using CADP of the ScalAgent Deployment Protocol for Software Components”. Proceedings of the 6th

- IFIP International Conference on Formal Methods for Open Object-based Distributed Systems FMOODS'2003 (Paris, France), November 2003
- [Transyt] "<http://research.ac.upc.es/VLSI/transyt/transyt.html>"
- [Ung69] Unger S.H., "Asynchronous sequential switching circuits", Wiley interscience, New York, NY, 1969.
- [Var90] Victor I. Varshavsky, editor. "Self-Timed Control of Concurrent Processes : The Design of Aperiodic Logical Circuits in Computers and Discrete Systems". Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990.
- [WBe00] R. Wollowski and J. Beister. "Comprehensive Causal Specification of Asynchronous Controller and Arbiter Behaviour". In : Yakovlev, A., Gomes, L., Lavagno, L. (eds.) : Hardware Design and Petri Nets. Kluwer Academic Publishers, Boston (2000) 3-32.
- [YDi92] Kenneth Y. Yun and David L. Dill. "Automatic synthesis of 3D asynchronous state machines". In Proc. International Conf. Computer-Aided Design (ICCAD), pages 576-580. IEEE Computer Society Press, November 1992.
- [YGi01] M. Yoeli and A. Ginzburg,, "LOTOS-based Verification of Asynchronous Circuits". Technical Report, Dept. Of Computer Science, Technion, Hifa, 2001.
- [Zhe98] Zheng H. "Specification and compilation of timed systems", Master thesis, University of Utah, 1998.
- [ZMM03] H. Zheng, E. Mercer, and C. Myers, "Modular verification of timed circuits using automatic abstraction", in IEEE Transactions on CAD, 22(9):1138-1153, September, 2003

---

## Résumé

La conception asynchrone vise à répondre aux problèmes de plus en plus complexes rencontrés par les concepteurs de circuits synchrones. Les circuits asynchrones, contrairement aux circuits synchrones, ne sont pas commandés par une horloge globale. Même de taille moyenne, ils peuvent montrer un comportement complexe, dû à l'explosion combinatoire dans la chronologie des événements qui peuvent se produire. Il est ainsi essentiel d'appliquer des méthodes rigoureuses de conception et de validation.

Ce travail de thèse traite de l'analyse et de la validation automatique des spécifications de circuits asynchrones écrites en CHP, avant leur synthèse avec le flot de conception asynchrone TAST, développé par le groupe CIS de TIMA. Deux approches sont proposées. La première consiste à adapter la vérification symbolique de modèles, initialement dédiée aux circuits synchrones, pour la vérification des circuits asynchrones. Les spécifications de circuits sont alors traduites dans un modèle en VHDL *peuso-synchrone* et ensuite vérifiées par des outils industriels de vérification symbolique de modèles.

Dans la deuxième approche, la sémantique de CHP, initialement donnée en termes de réseaux de Pétri, est reformulée en termes de Systèmes de Transitions Etiquetées Etendus (STEE). Les spécifications de circuits sont alors validées par des méthodes énumératives de vérification de modèles. Pour augmenter les performances de l'approche énumérative et faire face au problème d'explosion d'états, nous avons développé et implémenté un certain nombre de techniques automatiques de réduction et d'abstraction.

**Mots-clés :** *Vérification Formelle, Validation de Spécifications, Circuits Asynchrones, Processus Séquentiels Communicants, CHP, Modélisation Pseudo-synchrone, Vérification Symbolique de Modèles, Vérification énumérative de Modèles, Outils Symboliques de Décision.*

---

**Title:** Validation of Asynchronous Circuits Specifications: Methods and Tools.

---

## Abstract

Asynchronous designs aim at answering the increasingly complex problems (clock distribution, energy, modularity) encountered by the synchronous circuits designers. Asynchronous circuits, contrary to the synchronous circuits, are not ordered by a global clock. Even medium size asynchronous circuits may display a complex behavior, due to the combinational explosion in the chronology of events that may happen. It is thus essential to apply rigorous design and validation methods.

This thesis work addresses the analysis and the automatic validation of asynchronous specifications written in the CHP, prior to their synthesis with the TAST asynchronous design flow developed by the CIS group of TIMA. Two approaches are proposed. In the first approach we use symbolic model checking and pseudo-synchronous modeling, to perform property checking on RTL designs. The approach consisted in translating the Petri Net, interpreted as a finite state machine, as a pseudo-synchronous VHDL description, which can then be input to industrial symbolic model checking software.

In the second approach, CHP semantics, initially given in terms of Petri Nets, are reformulated as Extended Labeled Transition Systems (ELTS). Circuit specifications are then validated using enumerative model checking tools. To increase the performances of the enumerative approach and avoid the state explosion problem, we have developed and implemented several automatic reduction and abstraction techniques.

**Keywords :** *Formal Verification, Validation of Specifications, Asynchronous Circuits, Communicating Sequential Processes, CHP, Pseudo-synchronous Modeling, Symbolic Model Checking, Enumerative Model Checking, Symbolic Decision Tools.*

---

Laboratoire TIMA, 46 Avenue Félix Viallet, 38031 Grenoble Cedex, France.

ISBN : 2-84813-038-5

ISBNE : 2-84813-039-3