Université de Sfax

République Tunisienne
Ministère de l'Enseignement Supérieur,
De la Recherche Scientifique
et de la Technologie

École Nationale d'Ingénieurs de Sfax

MEMOIRE

Présenté à

L'École Nationale d'Ingénieurs de Sfax

en vue de l'obtention du

MASTERE

Dans la discipline informatique
Mastère NTSID

Par

Maïssa ELLEUCH SAHNOUN

(Ingénieur en informatique)

DEVELOPPEMENT D'UN MODELE FORMEL POUR DES RESEAUX MULTI-ETAGES DEDIES AUX SYSTEMES MULTIPROCESSEURS SUR PUCE

Soutenu le 27 Juin 2008, devant le jury composé de :

M.	**Abdoulaye GAMATIÉ**	*Président*
M.	**Adel MAHFOUDHI**	*Membre*
M.	**Mohamed ABID**	*Membre*

Remerciements

En premier lieu, je tiens à exprimer mes reconnaissances à Monsieur Mohamed ABID, professeur à l'ENIS et directeur du laboratoire CES, de bien vouloir m'accepter au sein de son laboratoire, pour son soutien, son assistance et ses conseils précieux.

Je tiens également à remercier profondément mon encadreur Monsieur Yassine AYDI, pour sa grande disponibilité ; le temps qu'il m'a accordée, sa volonté de m'aider dans mes recherches, ses conseils, ses idées, son esprit critique et ses encouragements pour l'aboutissement de mon mastère.

Je remercie beaucoup tous les membres du jury d'avoir accepté d'être présent lors de ma soutenance afin de juger mon travail.

J'adresse aussi mes remerciements à tous les membres du laboratoire CES-Computer and Embedded Systems, pour leur accueil chaleureux, leur amabilité, leur soutien et leur contribution à ma bonne intégration au sein de l'équipe.

Un grand merci à Monsieur Julien Schmaltz, post-doctorant à l'université Radboud aux Pays Bas, pour ses précieuses recommandations au tout début de ce mastère.

Un grand merci également à tous les membres de la liste d'aide ACL2 *(ACL2 help list)*, notamment aux membres du groupe ACL2 à l'université du Texas : Matt, Sandip, Peter, Jared, David et les autres. Leur grande disponibilité, leurs explications et leurs commentaires constructifs via cette liste m'ont beaucoup aidé lors de mon apprentissage d'ACL2. Sans leur assistance, il m'aurait été difficile d'achever correctement les démonstrations de certains de mes théorèmes.

Enfin, je ne manquerai pas de remercier vivement mon mari, ma famille et ma belle famille, qui m'ont beaucoup encouragée pour aller jusqu'au bout de cette étude.

Tables des matières

Table des figures

Liste des tableaux

Introduction générale

1. Contexte de l'étude

Les systèmes embarqués envahissent notre quotidien : il s'agit d'une réalité du monde moderne. En effet, ces systèmes servent des domaines très variés tels que l'automobile, les télécommunications, les multimédias et évoluent tous les jours comme par magie. Ainsi, on découvre chaque jour de nouveaux systèmes plus sophistiqués ayant, à la fois, des capacités plus élevées et des tailles plus réduites. À l'égard de cette révolution, plusieurs questions se posent. Néanmoins, la plus captivante reste la suivante : « Q*uel est le mystère qui se cache derrière tous ces systèmes* ? ».

La réponse est toute simple. C'est la puce électronique qui, conduite par l'évolution phénoménale de la technologie submicronique et notamment de la technologie de fabrication des circuits intégrés, se voit être à l'origine de cette révolution. Une puce est capable d'intégrer plusieurs ressources ayant différentes fonctions. Ces ressources peuvent être des processeurs, des composants hétérogènes (des mémoires, des périphériques, des unités numériques spécialisés), du logiciel et souvent des circuits mixtes. Tous ces composants matériels sont interconnectés à l'aide de mécanismes de communication très sophistiqués (Cesa *et al.*, 2002). On parle alors des systèmes multiprocesseurs sur puce dénotés *MPSoCs* (MultiProcessor Systems on Chip).

Une autre question s'impose par rapport à cette dépendance croissante et elle est exprimée explicitement par J.C Laprie (Laprie, 1990) : « Do you have enough confidence in computer systems that we let them handle our most valuable goods, namely our life and our money? ». En réalité, la dépendance des humains vis-à-vis des systèmes techniques n'est pas corrélée à une vraie conscience des conséquences que peut avoir un dysfonctionnement aléatoire de ces systèmes ; or la puce contrôle non seulement le système dans lequel elle est implantée mais aussi nos vies.

Ainsi, le niveau de confiance des humains par rapport à ces systèmes dépend essentiellement du degré de vérification auquel ils étaient soumis. Il faut noter que la vérification d'un système matériel ou d'une puce s'avère comme l'une des tâches les plus fastidieuses lors de sa conception. On montre même que la vérification d'un circuit fait appel à un nombre d'ingénieurs largement supérieur au nombre de concepteurs (au moins deux fois supérieur) et consomme 60 à 70% du temps de conception (Nobl, 2002).

De ce fait, les industriels ont de plus en plus recours à des méthodes de vérification dites « intelligentes » en opposition avec la méthode traditionnelle par simulation numérique qui a déjà montré ses limites face à des systèmes complexes. A cet effet, la nouvelle tendance dans la vérification s'oriente alors vers l'utilisation des méthodes formelles.

Dans la section suivante, nous présentons les contributions attendues de notre travail de mastère.

2. Contributions

Par l'effet de la complexité croissante des systèmes multiprocesseurs sur puce (MPSoCs) en général, et l'augmentation du nombre de composants sur la puce en particulier, les architectures de communication sur la puce (on-chip) ont dû évoluer. Comme alternative prometteuse aux bus classiques, les réseaux sur puce appelés aussi NoCs, ont témoigné d'un niveau de performance assez élevé.

L'exploration actuelle des NoCs se limite à des réseaux de type statiques. Une innovation dans ce domaine serait l'exploitation des réseaux dynamiques dans des MPSoCs. Comme l'idée est assez récente, il n'existe pas à notre connaissance de travaux qui touchent la vérification formelle de ce dernier type de réseau (sur puce).

Une spécification de type informel est souvent à l'origine de la mise en place des services dans les réseaux dynamiques. Toutefois, pour une multitude de raisons, l'aspect informel s'est avéré très insuffisant. Ainsi, afin de rendre les processus de configuration et de tests plus rapides, plus efficaces et plus sécuritaires, il faut apporter des solutions qui permettront d'élever le niveau d'abstraction de la représentation des configurations des services du réseau. Une solution possible consisterait en la vérification de certaines propriétés pertinentes au fonctionnement de ces réseaux après construction du modèle formel correspondant.

Les différentes étapes de la démarche de l'étude faite, ainsi que la structure du manuscrit, seront illustrées dans ce qui suit.

3. Structure du manuscrit et démarche de l'étude

Notre travail de mastère vise *le développement d'un modèle formel pour des réseaux multi-étages dédiés aux systèmes multiprocesseurs sur puce*. En effet, nous souhaitons à la fois construire et vérifier formellement un modèle décrivant les communications dans les réseaux

dynamiques multi-étages dédiés aux MPSOCs. Nous organisons notre travail selon les grandes étapes suivantes :

- Une exploration exhaustive des travaux de vérification formelle des réseaux sur puce, des réseaux multi-étages et des outils formels,
- L’identification et la traduction dans la logique de l’outil de vérification choisi la spécification des communications dans les réseaux multi-étages dédiés aux MPSoCs,
- La validation du modèle formel par la vérification d’une ou de plusieurs propriétés pertinentes pour le fonctionnement des réseaux multi-étages.

Chapitre 1 : Contexte de l'étude

1. Introduction

Afin de surmonter les problèmes de communication rencontrés dans les systèmes multiprocesseur sur puce (Ho *et al.*, 2001), les chercheurs se sont investis dans la conception de nouvelles plateformes d'interconnexion fiable, à énergie réduite et à rendement élevé, baptisés réseaux sur puce ou NoCs (Networks on Chip).

Il est évident que l'intégration d'un NoC dans un système multiprocesseur sur puce (MPSoC) ne peut se faire sans sa vérification. Cette dernière a pour principal objectif de montrer la conformité de la conception d'un niveau donné avec les spécifications du niveau précédent.

Nous insisterons dans ce chapitre sur l'importance de l'aspect vérification dans la conception des systèmes multiprocesseur sur puce (MPSoCs) en général, et dans celui des réseaux sur puce en particulier. Nous exposerons ainsi plusieurs travaux relatifs à la vérification formelle de réseaux sur puce existants.

2. La vérification des MPSoCs : utilité et techniques

Avant d'aborder l'aspect vérification dans les MPSoCs, nous présentons ici un aperçu de l'évolution de la technologie correspondante, ainsi que les deux architectures les plus répandues et quelques uns des réalisations dans ce domaine.

2.1. Evolution de la technologie submicronique

Les densités d'intégration actuelles permettent d'assembler sur une même puce un système numérique complet.

D'une part, les systèmes sur puce ou SoCs (Systems on Chip) deviennent de plus en plus complexes, l'ITRS (Zei *et al.*, 2005) prévoit en 2012 l'intégration de systèmes électroniques de 4 milliards de transistors pour des fréquences proches de 10 GHz, comme l'illustre le tableau 1. L'enjeu de la prochaine décennie dans le secteur des semi-conducteurs est alors d'intégrer sur une même puce un système multiprocesseur hétérogène (Multiprocessors System-on-Chip).

Tableau 1. *Prévisions de l'ITRS*

	2001	2006	2012
Technologie (nm)	150	90	50
Complexité (M transistors)	40	200	4000
Surface de la puce (cm²)	3,85	5,2	7,5
Fréquence Local (ghz)	1,25	3,5	10

De façon générale, les interconnexions des composants sur la puce se font par bus central (figure 1). Cependant, malgré la simplicité de mise en œuvre qui caractérise les bus, ils ont montré très vite leurs limites. En effet, un bus central est une ressource partagée ; son extensibilité est donc très mauvaise. De plus, si le nombre de composants (mémoire, processeur, DSP, IP...) sur le bus augmente, la bande passante disponible pour chacun décroît. Dans ce cas, les capacités parasites vont également augmenter et la fréquence d'utilisation du bus sera limitée. Puis, le bus perd aussi en performances s'il a beaucoup de blocs à satisfaire puisque le temps alloué à chacun diminue alors que le temps d'arbitrage augmente. Enfin, les interconnexions physiques actuelles à base de bus sont des facteurs limitant de performance des SoCs (longueur des interconnexions, bande passante, consommation d'énergie).

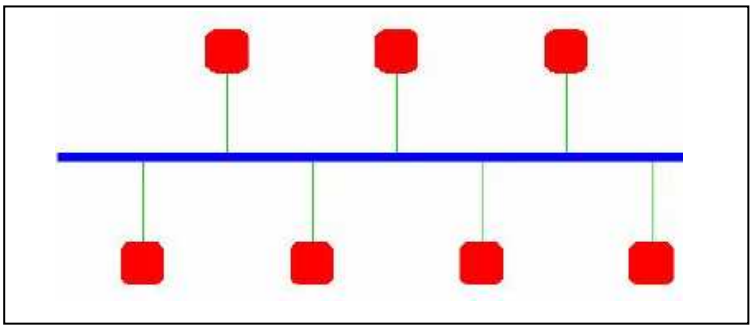


Figure 1. *Une architecture à bus central*

D'autre part, les réseaux sur puce ou NoC (Network on Chip) sont susceptibles de proposer des solutions efficaces aux problèmes d'intégration complexes des systèmes sur puce (Beni *et al.*, 2002). Ces architectures d'interconnexions devront tout de même, faire face à de nombreuses contraintes : consommation d'énergie, surface de silicium, performances, synchronisation... De plus, le coût et les caractéristiques de ces réseaux sur puce dépendent des applications considérées (Ateris, 2005).

Par ailleurs, l'industrie des semi-conducteurs qui est à l'origine de la fabrication des systèmes sur puce, a focalisé ses activités en deux domaines importants : les architectures à usage général (les processeurs) et les systèmes embarqués. Ces deux types d'architectures seront détaillés dans les deux sections suivantes.

2.1.1. Les architectures à usage général

Les architectures à usage général sont utilisées essentiellement dans les ordinateurs personnels où la haute performance et la fiabilité sont des critères considérés comme prioritaires par rapport aux autres facteurs tels que la consommation d'énergie et le coût de fabrication. Ainsi, dans le but d'augmenter la performance de ce type d'architecture à une fréquence d'horloge relativement stable, la tendance courante est d'implanter sur la même puce plusieurs processeurs identiques. Ceci permet sûrement d'exécuter des tâches en parallèle et donc une augmentation recherchée de la performance. Par exemple, le processeur « the Itanium 2 Montecito dual core » est une illustration du progrès réalisé pour les architectures à usage général (figure 2). En effet, il a pu intégrer en 2006, 1.7 milliards de transistors pour une puissance de 100W à une fréquences de 1.6 GHz, une taille de la puce est 580mm^2, pour une technologie de 90nm (Intel, 2006).

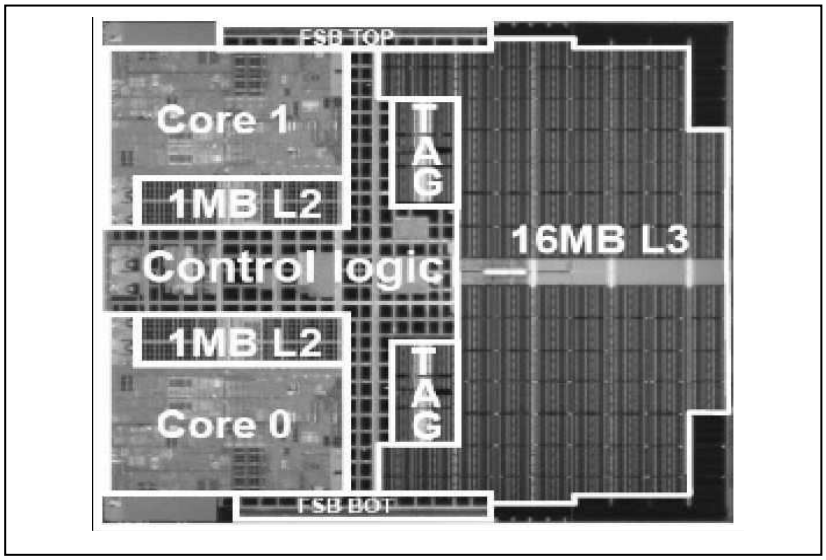


Figure 2. *Le processeur the Itanium 2 Montecito dual core*

Dans la suite, nous présentons les systèmes embarqués qui constituent le deuxième type d'architecture exposée.

2.1.2. Les systèmes embarqués

Les systèmes embarqués (embedded systems) sont des systèmes à la fois électroniques et informatiques (numériques) autonomes dans lesquels le matériel et le logiciel sont intimement liés. Le logiciel est généralement dédié à une fonctionnalité bien précise, alors que le matériel peut comporter plusieurs composants tels que des circuits numériques FPGA, ASIC ou des circuits analogiques, et ceci dans le but d'augmenter les performances de l'application ou sa fiabilité. Ces composants matériels sont interconnectés par des mécanismes de communication très sophistiqués.

La conception des systèmes embarqués requiert une approche à base de plateforme. De plus, le développement du matériel et du logiciel est fait simultanément dans les différents niveaux d'abstraction. En effet, ce développement doit satisfaire plusieurs contraintes à la fois comme la consommation d'énergie, la surface, les performances, la synchronisation, le coût et le temps de fabrication.

Ainsi, la plateforme conçue par Sony, Toshiba et IBM est une architecture embarquée de haute performance (Kahl, 2005). Par exemple, les jeux « playstation3 » et les téléviseurs hautes définitions utilisent ce système renfermant sur la même puce un processeur et 8 coprocesseurs graphiques (figure 3). La taille de la puce est de 221mm^2 pour 8 niveaux de métal.

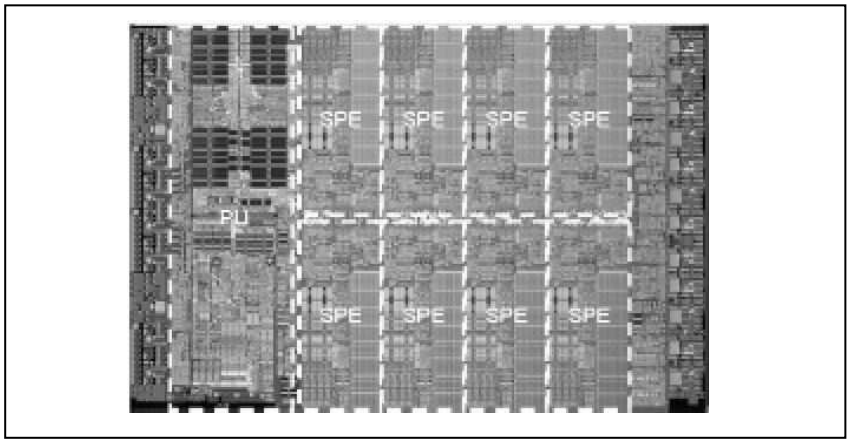


Figure 3. *La plateforme Cell*

Par ailleurs, l'IMEC a développé un système multiprocesseur sur puce baptisée 3MF (figure 4). Cette plateforme supporte les standards de compression vidéo et audio (MPEG4, AVC, SVC, 3D-graphics) (IMEC, 2006). La puce renferme plusieurs processeurs ADRES, un DSP, des mémoires, et un module d'entrée/sortie. L'interconnexion des composants est faite par un réseau sur puce. La puissance dissipée est de 700mW à une tension d'alimentation de 1.0 V, pour une technologie de 90nm.

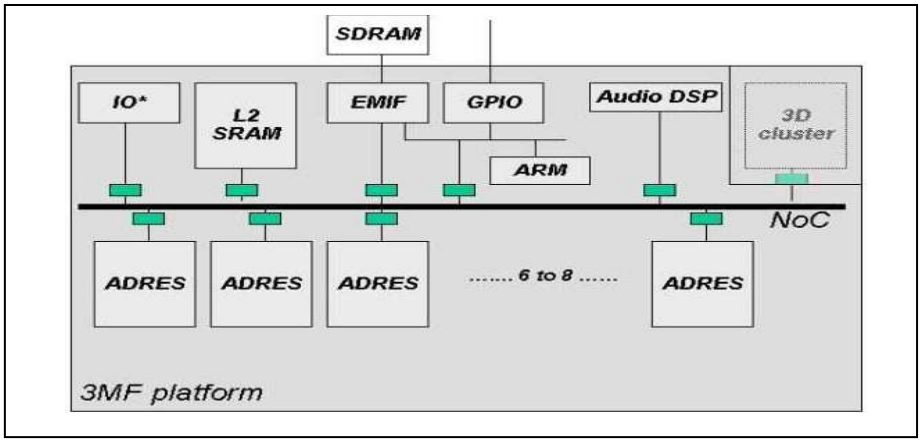


Figure 4. *La plateforme 3MF*

La diversité et le nombre de ressources qui composent un système multiprocesseur sur puce, n'ont fait qu'augmenter sa complexité de conception. Cette complexité sera illustrée dans le paragraphe suivant.

2.2. Complexité conceptuelle des MPSoCs

Vu l'hétérogénéité des composants formant la puce, il devient crucial de maîtriser la conception de tels systèmes tout en respectant les contraintes de mise sur le marché et les objectifs de qualité. La conception des systèmes multiprocesseurs sur puce doit se faire conformément à une démarche spécifique illustrée à la figure 5. Chacune des étapes du flot décrit un niveau d'abstraction donné. En réalité, l'abstraction est le concept clé de toute la conception des MPSoCs. A chaque étape, on s'occupe d'un aspect du système à concevoir : on fait apparaître les détails liés à cet aspect et on camoufle d'autres inutiles à ce niveau. Ce flot part alors d'une spécification de niveau système réalisée grâce à des langages dits aussi système (VHDL, Verilog, System C) pour obtenir une spécification fonctionnelle. L'étape suivante est l'exploration architecturale. Il s'agit ici d'explorer l'espace de solutions en terme d'architectures pour déterminer les parties qui seront implémentées en logiciel et celles en matériel. L'architecture retenue devrait être celle qui répond le mieux aux exigences du système. Ensuite, chacune des parties logicielles et matérielles seront raffinées séparément pour arriver enfin à les intégrer dans un système unique.

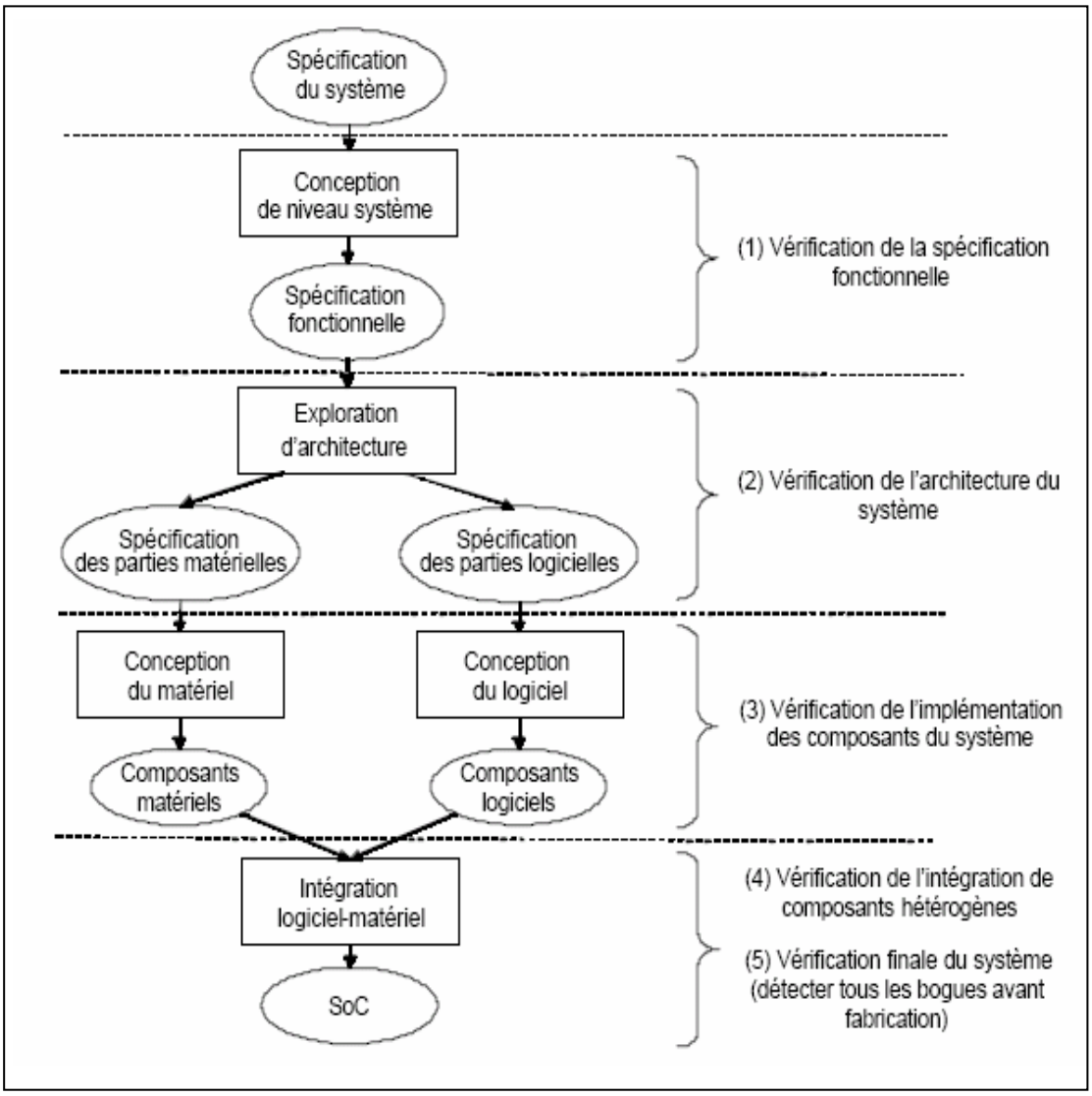


Figure 5. *Flot de conception des systèmes monopuces*

Dans son livre *« Introduction to Formal Hardware Verification »* (Kropf, 1999), Thomas Kropf partitionne le processus de conception en montrant clairement la relation qui doit exister entre les différentes étapes de la conception. En effet, l'implémentation de la spécification résultante d'une certaine étape joue le rôle de spécification pour l'étape suivante. Ainsi, une implémentation de la spécification de niveau architectural constitue la spécification du niveau transfert de registres (RTL) suivant.

Les multiples contraintes sur les MPSoCs ont dû renforcer la place de la vérification dans leur flot de conception. D'ailleurs, nous pouvons remarquer d'après la figure 5 que la vérification accompagne chacune des étapes de conception. En visant une meilleure crédibilité des systèmes sur puce, notamment ceux qui sont critiques, l'une des méthodes robustes est la vérification formelle. Cette méthode sera illustrée dans la suite.

2.3. *Les techniques de vérification des circuits numériques*

La correction absolue du flot de conception de la figure 5 pour un système donné est garantie si et seulement si la spécification initiale est valide et le passage d'un niveau à un autre est vérifié. Plusieurs techniques de vérification existent. Dans ce qui suit, nous établirons une comparaison entre la technique classique par simulation et les nouvelles techniques à base de formalisme, et ceci après les avoir présenté.

2.3.1. *La vérification par simulation*

La simulation soumet le système à des vecteurs de tests, compare les résultats obtenus aux résultats attendus et corrige les éventuelles erreurs. Les valeurs de tests sont générées manuellement ou par un logiciel dédié. Même si la simulation est une technique qui a l'avantage d'être naturelle et simple, elle présente des performances limitées. En réalité, dès qu'il s'agit de l'appliquer à un bas niveau d'abstraction pour des systèmes assez complexes, les temps de réponse deviennent énormes. En plus, la simulation ne peut être que partielle : il est quasiment impossible de couvrir toutes les valeurs possibles pour les entrées. Autrement dit, les tests ne peuvent jamais être exhaustifs. Par conséquent, la garantie d'un système fiable par simulation est impossible.

2.3.2. La vérification formelle

Si l'intérêt porté par les industriels au « formel » est assez récent, l'idée en revanche ne date pas d'aujourd'hui. En 1962, J. McCarthy formulait déjà les principes de la vérification formelle automatique donnant ainsi naissance aux « machines raisonnant sur des machines » (McCa, 1962). D'ailleurs, auparavant, l'utilisation de telles méthodes se limitait à quelques domaines et nécessitait un apprentissage effectif. Récemment, ce n'est plus le cas. Les méthodes formelles sont devenues d'usage courant (Clar, 1996).

Vérifier formellement un circuit consiste en la preuve mathématique qu'un modèle de ce circuit se comporte conformément aux propriétés exigées. Une telle vérification est possible à tous les stades de la conception du futur système via deux approches principales : la vérification de modèles (model checking) et la démonstration de théorèmes (theorem proving).

Il a été démontré que l'utilisation des formalismes mène à une réduction des coûts de maintenance. Leurs techniques ont l'avantage d'être applicables à tout type de système en assurant un développement fiable et en minimisant le risque de pertes économiques et

humaines. Les méthodes formelles ont en outre la possibilité de couvrir implicitement tous les cas de tests. Nous illustrerons dans la suite comment cela est possible à travers un exemple concret.

2.3.3. La simulation vs la vérification formelle

Une différence fondamentale existe entre la simulation classique et la vérification formelle. Nous reprenons ici un exemple très répandu dans la littérature mettant en évidence cette différence (Gord, 1989). Supposons que l'on souhaite démontrer que la formule (1-1) est correcte c'est-à-dire que les deux termes de l'égalité donnent la *même* valeur et ceci pour *toutes* les entrées possibles.

$$\mathbf{(x + 1)^2 = x^2 + 2x + 1} \qquad \mathbf{(1\text{-}1)}$$

Une approche par simulation consisterait à prendre plusieurs valeurs de *x* et de tester la validité de l'égalité pour ce *x*. Cette approche est illustrée dans le tableau 2.

Tableau 2. *Les résultats de simulation de (1-1)*

x	$(x + 1)^2$	$x^2 + 2x + 1$
0	**1**	**1**
1	**4**	**4**
2	**9**	**9**
3	**16**	**16**
9	**100**	**100**
67	**4624**	**4624**
…	**…**	**…**

La formule (1-1) doit être démontrée pour tous les nombres sans exception. La simulation ne produit ici que des résultats pour les nombres entiers, elle ne peut pas couvrir toutes les valeurs des nombres : il y aura une infinité de cas à tester. La simulation est alors incapable de valider la formule en question. L'approche formelle, quant à elle, démontre l'égalité sans pour autant émettre une restriction sur les valeurs possibles de *x* et ceci en appliquant des transformations d'ordre mathématique rapporté dans le tableau 3.

Dans le reste de ce chapitre, nous illustrerons plusieurs travaux de vérification formelle de NoCs, et ceci après une présentation de quelques notions générales liées aux réseaux sur puce.

Tableau 3. *La preuve mathématique de (1-1)*

1.	$(x + 1)^2 = (x + 1)(x + 1)$	**définition de la puissance**
2.	$(x + 1)(x + 1) = (x + 1)x + (x + 1)1$	**distributivité**
3.	$(x + 1)^2 = (x + 1)x + (x + 1)1$	**substitution de 2 dans 1**
4.	$(x + 1)1 = (x + 1)$	**élément neutre 1 pour x**
5.	$(x + 1)x = xx + 1x$	**Distributivité**
6.	$(x + 1)^2 = xx + 1x + x + 1$	**substitution de 4 et 5 dans 3**
7.	$1x = x$	**élément neutre 1 pour x**
8.	$(x + 1)^2 = xx + x + x + 1$	**substitution de 7 dans 6**
9.	$xx = x^2$	**définition de la puissance**
10.	$x + x = 2x$	**définition de 2x**
11.	$(x + 1)^2 = x^2 + 2x + 1$	**substitution de 9 et 10 dans 8**

3. Les réseaux sur puce : un nouveau paradigme

3.1. Introduction

La façon avec laquelle sont organisées les interconnexions entre les différents unités d'un système sur puce, définit ce qu'on appelle sa topologie physique. Dans le cas des MPSoCs, les deux topologies physiques dominantes sont les bus et les réseaux. Les réseaux sur puce ont été inventés dans le principal but de remédier aux inconvénients connus des systèmes d'interconnexion classiques c'est-à-dire les bus. D'une conception assez délicate, les réseaux sur puce (NoC) sont soumis à plusieurs contraintes de performance comme la latence, le débit et doivent être en plus flexibles en offrant une bonne qualité de service. Dans la suite, nous détaillerons une classification des réseaux sur puce par topologie.

3.2. Classification des NoCs

3.2.1. Les critères de classification

Le critère incontournable de classification des réseaux sur puce (NoCs) est la topologie (Bjer *et al.*, 2006). Toutefois, en plus d'une classification suivant la forme des liens, on peut aussi ajouter d'autres critères de classification telles que le type des nœuds (direct, indirect), le nature des liens (unidirectionnel, bidirectionnel), le nombre d'étages (à zéro étage, à un étage, multi-étages), le type de réseau (statique, dynamique), la régularité du réseau (régulier,

irrégulier). Nous présentons ici une simple classification des réseaux sur puce par topologie sans pour autant se soucier des critères additionnels.

3.2.2. Classification par topologie

Les topologies dominantes pour implémenter les NoCs sont les bus et les réseaux.

- ***La topologie en bus* :** elle est réalisée physiquement grâce à un médium unique partagé par plusieurs unités *(cf. figure 1)*. Sur ce médium, les données ne circulent pas de façon arbitraire mais elles suivent plutôt une logique bien déterminée, c'est ce qu'on appelle la topologie logique. Par exemple, dans le cas des réseaux d'ordinateurs ayant une topologie physique en bus, la topologie logique courante est l'Ethernet. Bien que d'extensibilité très limitée avec un faible degré de parallélisme, l'architecture en bus reste fréquemment utilisé vu sa simplicité de mise en œuvre et de fonctionnement. Plusieurs extensions du bus partagé ont été implémentées afin d'améliorer ses performances (bus matriciel).

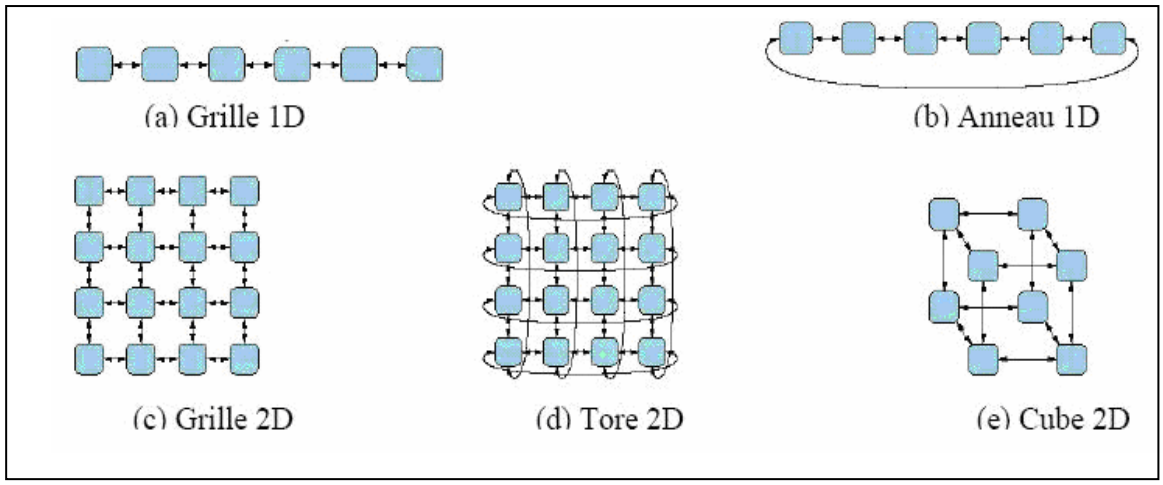


Figure 6. Quelques topologies

- ***La topologie en réseau :*** dans sa définition la plus simple, un réseau est un ensemble de nœuds connectés par des liens de communication. Dans le cas des systèmes sur puce, un noeud peut contenir un ou plusieurs composants tels que des processeurs, des mémoires ou encore des périphériques d'entrée/sortie. Les principales topologies utilisées pour les réseaux sur puce sont les grilles et les cubes. Dans une grille, les nœuds sont identifiés par des coordonnées. Deux nœuds *x* et *y* de la grille sont connectés si et seulement leurs coordonnées respectives sont identiques sauf sur une dimension. Suivant cette dernière, les identificateurs de *x* et *y* ne doivent différer que de 1. La figure 6 illustre quelques types de grilles telles que la grille 1D (graphe linéaire) et la grille 2D. Pour les réseaux en cube, ils ont la même structure

qu'un cube où un nœud est aussi identifié par ses coordonnées. La figure 6 présente un exemple de cube 2D. La topologie tore 2D illustré dans cette même figure n'est autre qu'une grille dans laquelle on connecte les nœuds des deux extrémités.

3.3. Les travaux de formalisation des réseaux sur puce (NoCs)

Un système sur puce typique est composé d'un ensemble d'unités interconnectées par un système de communication. Sachant que chaque unité constitue souvent une sorte de « boîte noire » pré-validée (IP), un aspect essentiel de la validation du système en entier est celui de la validation des interactions entre ces IPs via le système de communication (Spir, 2004).

Cette partie s'intéresse aux travaux relatifs à la validation des réseaux sur puce par vérification formelle. Pour chacun de ces travaux, nous commencerons par une présentation brève du réseau sur puce vérifié, ensuite, nous exposerons un exemple de formalisation dont il a fait objet.

3.3.1. Le bus AMBA, ARM

3.3.1.1. Présentation

La figure 7 montre la façon avec laquelle on peut utiliser un bus AMBA AHB/ASB en conjonction avec un bus APB. Cette conjonction est réalisée grâce à un pont (bridge). En réalité, le système AMBA (The Advanced Microcontroller Bus Architecture) est défini en fonction de trois types de bus (ARM, 1999) :

- *Advanced High-performance Bus (AHB)* : c'est un bus utilisé pour établir les communications entre les modules du système de fréquence élevée (les processeurs) et les modules nécessitant une grande bande passante (les mémoires on-chip et les mémoires off-chip). Il a l'avantage d'avoir un rendement élevé. De plus, il est spécifié de façon à bien s'intégrer dans un flot de conception qui utilise les techniques de synthèse et de tests automatisés.
- *Advanced System Bus (ASB)* : quand les rendements élevés du bus AHB ne sont pas exigés, le bus ASB peut le remplacer. Pour accélérer les échanges, ASB intègre juste le *« pipeline »* des opérations.
- *Advanced Peripheral Bus (APB)* : c'est un bus périphérique utilisé pour interconnecter des dispositifs à consommation réduite et nécessitant une faible bande passante.

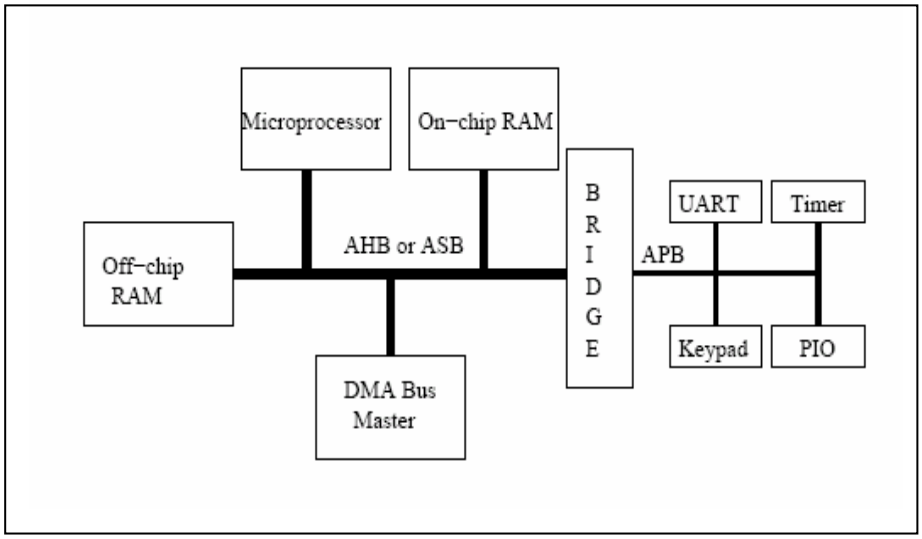


Figure 7. *Un microcontrôleur typique basé sur AMBA*

3.3.1.2. Formalisation

Le système AMBA d'ARM a fait l'objet de plusieurs travaux de formalisation. Le but principal de tous ces travaux est de vérifier la validité de certaines propriétés sur le bus. *Roychoudhury et al.* se sont intéressés en particulier au protocole AMBA AHB (Royc *et al.*, 2003). Leur travail de vérification a été accompli en appliquant la technique de vérification de modèles via le model-checker SMV (Symbolic Model Verifier). Les modules qui composent le système ont été alors décrits sous forme de machine à états finis. La deuxième phase de la formalisation a concerné la spécification manuelle des propriétés à vérifier dans la logique CTL (Computation Tree Logic) ; une logique qui est supportée par le vérificateur de modèles SMV. On a dû alors définir formellement la propriété de non famine (non starvation) : *« peu importe le maître m souhaitant utiliser le bus, il sera toujours autorisé à y accéder »*. En définitive, au bout de 0.17 secondes, SMV a pu automatiquement détecter un scénario de famine. Un master, ayant été momentanément suspendu par un signal *« split »*, ne sera plus jamais autorisé à accéder au bus. La configuration de vérification utilisée comportait 2 masters et 1 slave.

La conclusion tirée par Roychoudhury *et al.* est que le scénario détecté provient de la façon dont l'arbitre se rend compte d'un transfert de type « split ». Ainsi, on peut dire que le protocole de communication utilisé par AMBA n'entraîne pas des situations de famine réelles. Cette famine aurait pu être évité si l'arbitre utilisé sur le bus sait se rendre compte d'un transfert de type « split ».

3.3.2. Æthereal

3.3.2.1. Présentation

Le réseau sur puce Æthereal a été développé au laboratoire de recherches de Philips aux Pays Bas (Rijp, 2003). Il est basé sur une topologie irrégulière (figure 8). Les ressources (processeur, mémoire, IP,…) sont connectées au routeurs par des interfaces-réseaux. Le routeur Æthereal utilise un routage de source déterministe (source routing), une commutation de type *Wormhole* et une mémorisation de paquets en entrée. Chaque paquet est découpé en « flits » de 32 bits, le premier « flit » renferme l'entête (identification de paquet, taille, chemin, fenêtre d'anticipation, indicateur de fin de paquet). Æthereal fournit un transfert fiable de données via des routeurs opérant en deux catégories de trafic (établissement de connexion de bout en bout puis échange de données). Les interfaces-réseaux assurent plusieurs fonctions telles que le contrôle de flux, le paquetage de données, la connexion avec les protocoles standard d'interface, ainsi que l'ordonnancement des transactions.

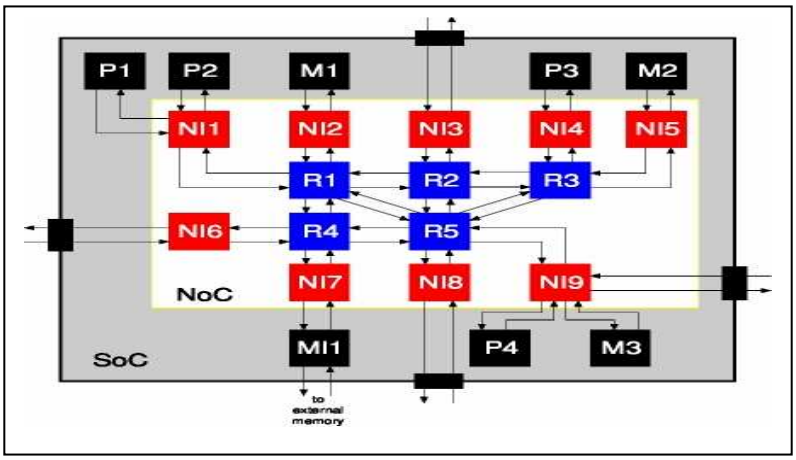


Figure 8. *Synoptique d'Æthereal*

3.3.2.2. Formalisation

Le but principal du réseau sur puce Æthereal est fournir un service « garanti » dans un micro réseau. Ce type de service se base sur la réservation des ressources pour la durée entière de la communication. Une fois la connexion établie, les données peuvent circuler facilement sur le chemin préétabli. Comme chaque nœud du réseau possède un nombre fini de « buffers », des scénarios d'interblocage peuvent alors facilement avoir lieu. Dans ce cadre, des travaux visant la vérification formelle de la propriété d'interblocage dans le réseau

Æthereal, ont été développé. Le travail de (Gebr *et al.*, 2005) a utilisé l'outil PVS pour démontrer que l'interblocage ne peut pas avoir lieu pour une version abstraite du NoC Æthereal.

3.3.3. Octagon, ST-Microelectonics

3.3.3.1. Présentation

Le modèle du réseau direct d'Octagon, proposé par Karim (Kari *et al.*, 2001), est basé sur une topologie en anneaux raccordés (figure 9). Chaque anneau renferme huit nœuds. Les fonctionnalités de routage et de commutation sont co-implantées avec le processeur. Le paquet circulant à travers le réseau est de taille variable, l'entête du paquet renferme trois bits dédiés pour le contrôle (bits d'adresses). Ce réseau utilise la commutation de paquets et de circuits. La technique de routage adoptée est de type distribuée et adaptative. La communication entre deux nœuds quelconques d'un anneau exige au plus deux liens intermédiaires. La bande passante de ce réseau peut atteindre 40Gbits/s, ce qui permet d'obtenir des circuits à rendement élevé.

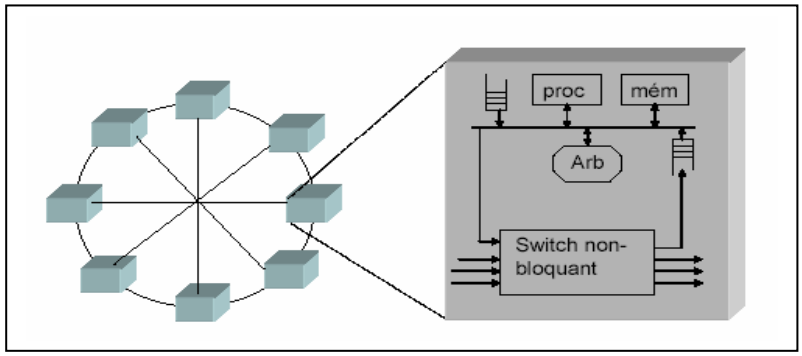


Figure 9. *Le réseau Octagon*

3.3.3.2. Formalisation

Le réseau Octagon a été formellement vérifié dans le cadre d'un travail de modélisation générique basé sur la technique de démonstration de théorèmes (Schm *et al.*, 2006). Un modèle générique dénoté GeNoC (Generic Network on Chip) décrit dans une notation complètement formelle les communications sur puce. Ce modèle représente les principaux composants de toute architecture de communication sur puce : la topologie (les nœuds), le routage et l'ordonnancement. Son critère de correction est la fiabilité du réseau : *« Tout message émis depuis une source du réseau atteint sa destination sans modification de son*

contenu ». Pour montrer l'adéquation du modèle générique avec la réalité, il a fallu le valider sur des réseaux concrets tels que l'Octagon et le Mesh 2D.

Le tableau 4 montre le nombre de fonctions et théorèmes et le temps de preuve pour la définition et la validation de l'Octagon dans la logique du démonstrateur ACL2. La modélisation des nœuds et de l'algorithme de routage a nécessité 21 fonctions. La preuve de conformité avec GeNoC a nécessité quant à elle 13 théorèmes.

Tableau 4. *Fonctions, théorèmes et temps de preuve pour la définition et la validation de l'Octagon*

	Nbre de fonctions	Nbre de théorèmes	Temps de preuve (en secondes)	Taille
OctagonNodeSet	5	4	< 1	70 lignes
Lemmes sur mod	0	10	< 3	150 lignes
Routage (def. et terminaison)	4	2	~ 120	140 lignes
Strategie	11	10	~ 300	400 lignes
Validation de ρ_{Oct}	4	29	~ 300	415 lignes
Conformité avec $GeNoC$	2	13	< 10	210 lignes
Total	21	64	< 740	1325 lignes

3.3.4. Le réseau Hermes

3.3.4.1. Présentation

L'architecture du réseau Hermes adopte la topologie en grille 2D. Chaque ressource (processeur, IP) est connectée à un routeur (figure 10). Ce dernier est composé de cinq ports (est, ouest, nord, sud et local). Le port local est relié à la ressource alors que les autres ports sont reliés aux routeurs voisins. Chaque port possède une file d'attente en entrée pour stocker provisoirement les données. La technique de commutation utilisée est de type *Wormhole* afin de diminuer la latence et l'utilisation de mémoires tampons. Les paquets circulant dans le réseau contiennent des données, un en-tête qui renferme l'adresse destination et un compteur indiquant le nombre de mots dans le paquet. L'acheminement des paquets dans le réseau se fait suivant une stratégie de routage arithmétique basée sur l'adresse du routeur exprimé en XY, où X représente sa position horizontale et Y sa position verticale. Les avantages primordiaux de cette plate-forme sont sa performance, notamment en terme de latence et débit, ainsi que sa flexibilité du fait que les files d'attente et la taille des paquets sont paramétrables.

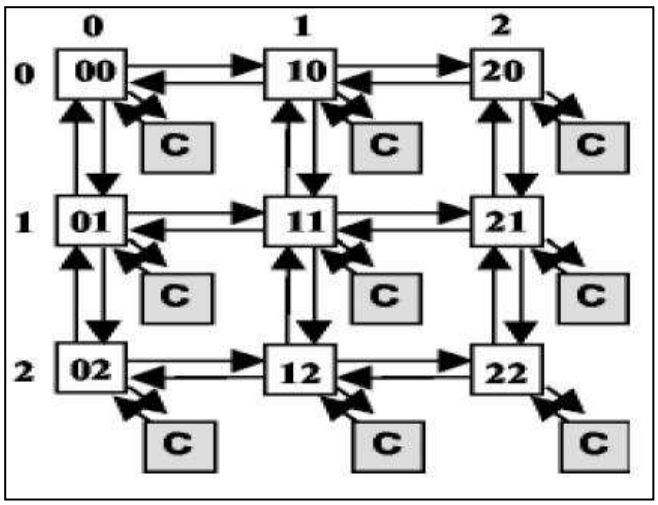


Figure 10. *Le réseau Hermes*

3.3.4.2. Formalisation

Le réseau Hermes a été validé avec approximativement la même approche que l'Octagon. En effet, par l'application du modèle générique GeNoC étendu *(cf. la section 3.3.3.2.)*, on a pu démontrer que le réseau Hermes est fiable (Borr *et al.*, 2006). Cette validation a nécessité l'extension du modèle GeNoC afin de décrire des aspects plus concrets des communications sur puce. Ainsi, il a fallu par exemple redéfinir certains types de données relatifs à GeNoC.

4. Conclusion

Tout au long de ce premier chapitre, nous avons insisté sur l'utilité de la vérification formelle en montrant les limites de la simulation classique à travers un exemple concret. Nous avons aussi passé en revue plusieurs travaux de vérification formelle de réseaux sur puce (NoCs). En définitive, ces travaux serviront à guider nos choix dans la conception et l'implémentation de notre modèle formel des réseaux multi-étages dédiés aux systèmes multiprocesseur sur puce (MPSoCs).

Chapitre 2 : Concepts et outils de la vérification formelle des circuits numériques

1. Introduction

Certes, le célèbre « bug » du Pentuim II survenu en 1994 au niveau de son unité de division de virgule flottante, a fait perdre à Intel 475 millions de Dollars (Mark, 1994). Cependant, c'est en grande partie « grâce » à cet incident que le monde de l'ingénierie des systèmes a vécu un bouleversement de ses techniques de vérification. Si l'erreur du Pentuim II n'a causé que des pertes économiques, d'autres incidents aussi célèbres comme : l'explosion de la fusée ARIANE5 en juin 1996 (Aria, 1996), le « bug » de la machine de radiothérapie THERAC-25 causant entre 1985 et 1987 le décès de 6 patients (Leve *et al.*, 1993), le crash répété d'avions tels que Airbus, sont des incidents catastrophiques dont les conséquences sont plutôt humaines. Devant l'insuffisance des méthodes traditionnelles de vérification et la complexité croissante des systèmes numériques développés, la solution était alors d'utiliser une vérification à caractère formel comme alternative complémentaire à la vérification classique par simulation.

Nous présentons dans ce chapitre les concepts de la vérification formelle : ses fondements et quelques uns de ses outils, en détaillant dans sa dernière section l'outil formel choisi pour notre travail. Dans notre cas d'étude, nous nous intéressons à l'application des techniques de vérification formelle à des systèmes multiprocesseurs sur puce (MPSoCs).

2. La vérification formelle des systèmes sur puce

2.1. *Définition*

La vérification formelle d'un système ou d'un circuit consiste en la preuve mathématique qu'il se comporte conformément à un ensemble de besoins formulés (Kropf, 1999).

2.2. *Démarche globale*

Indépendamment de la méthode de vérification, le travail formalisation se fait en général en deux grandes phases. La première consiste à spécifier un modèle formel du système à partir d'un ensemble de besoins. La deuxième phase se préoccupe de vérifier les propriétés

modélisées. Ces propriétés modélisant par exemple l'absence d'un type d'erreur, sont concrétisées par des formules logiques et doivent exprimer une compréhension de la correction de ce système. Pour réussir cette dernière phase, il faut essayer de préciser au maximum les notations informelles et d'identifier les hypothèses implicites. Cette phase inclut aussi l'exploration des documents de spécification, l'expérimentation du système réel si possible et enfin, les discussions avec les concepteurs du système.

Les équipes de conception et de vérification formelle ont des approches très différentes pour aborder le système. Par conséquent, un ingénieur de vérification formelle doit être prudent lors de son contact avec des concepteurs. Il ne doit pas être très influencé par « ce que disent les concepteurs ». En fait, la tâche de conception n'a pas vraiment de critère de correction. Le plus important pour les concepteurs est le fait que « le système tourne » en ayant le moins de « bugs » ou d'erreurs dans son implémentation. Les ingénieurs de vérification formelle, quant à eux, doivent rassembler le plus d'informations afin de pouvoir déceler des propriétés subtiles du système.

2.3. *Objectifs*

La vérification formelle a pour principal but de montrer qu'un système ou plutôt un modèle du système est correct. Un système correct signifie un system exempt d'erreurs, appelées aussi « bugs ». Pour des systèmes critiques tels que les voitures, les appareils médicaux, les banques ou les avions, une propriété telle que celle de l'absence de bugs est indispensable et sa non-satisfiabilité peut être fatale. Plusieurs types d'erreurs peuvent exister. On peut vouloir prouver qu'un système ne se « plante » jamais, autrement qu'il n'est jamais induit en un type donné d'erreur (blocage, famine…). La vérification formelle n'est donc réellement utile que pour vérifier la non-existence d'erreurs dites « critiques » ou encore l'existence inévitable de situations souhaitables ou vitales. Par exemple, il serait très intéressant de détecter de façon formelle qu'un système risque d'être bloqué alors que conformément à ses spécifications, ce même système ne doit jamais être induit dans une telle situation.

Des normes telles que la norme ISO/CEI 9126 (ISO) ont été spécialement développées dans le but de décrire les exigences qualité d'un système telles que sa capacité fonctionnelle, sa fiabilité et sa facilité d'utilisation. D'ailleurs, depuis 1995, l'usage des méthodes formelles est devenu une exigence imposée par les ITSEC (Information Technology Security Evaluation Criteria) et ceci à partir d'un certain niveau de sécurité.

2.4. *Les types de formalisation*

Il existe deux grands types de formalisation (Kropf, 1999) : la formalisation des spécifications et la formalisation de l'implémentation.

2.4.1. *Formalisation des spécifications*

La spécification formelle d'un système est une description concise et abstraite de son comportement et de ses propriétés. Elle doit décrire dans une notation mathématique « ce que le système est supposé faire » et non pas « comment le système doit être implémenté ». Donc, la spécification formelle d'un système se doit d'être assez générale en éliminant tous les détails relatifs à l'implémentation et qui sont inutiles à ce stade.

La figure 11 illustre les deux grandes phases de la formalisation des spécifications : la génération et la validation. La génération se fait à partir des descriptions informelles du système. Ces dernières sont constituées souvent de texte et de diagrammes. Il est assez courant d'avoir des descriptions informelles qui semblent être assez précise alors qu'une fois considérées le point de départ de la formalisation, se dévoilent plutôt vagues, incomplètes et mal organisés. La deuxième phase présentée à la consiste en la preuve de correction ou la validation de la spécification générée. Pour y arriver deux alternatives sont possibles : procéder par simulation du modèle fonctionnel construit ou vérifier formellement certaines propriétés par rapport à la fiabilité ou la consistance du système.

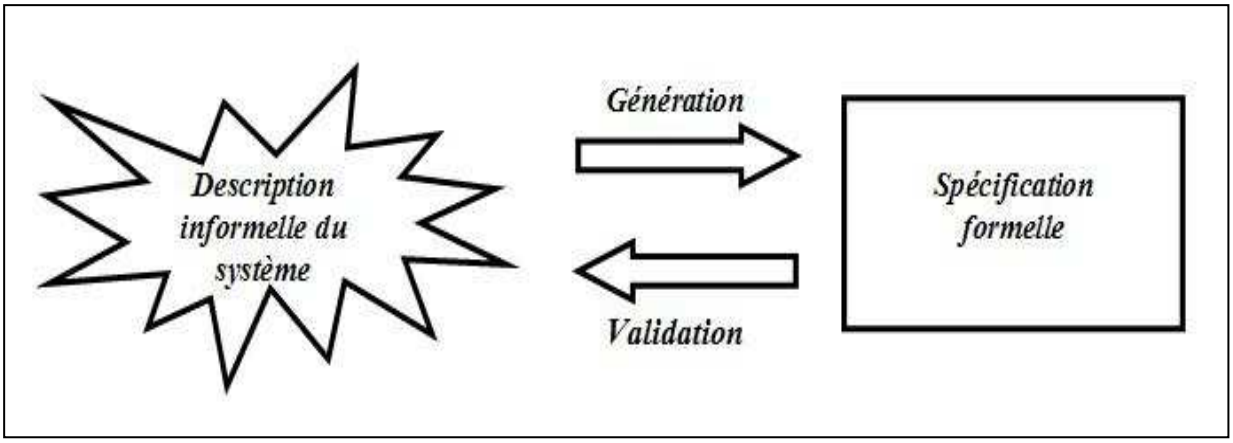


Figure 11. *Etapes de la spécification formelle*

La formalisation des spécifications est une étape très importante dans la conception d'un circuit. En effet, elle est le point de départ de toute la conception et permet d'avoir une vue assez abstraite et précise du système. Il est vrai que sa démarche semble être simple. Cependant, compte tenu de la qualité que peut avoir les spécifications informelles, le

processus de génération des descriptions formelles devient très difficile. En plus, l'étape de validation est en réalité une étape délicate : d'une part la capture de propriétés utiles au système n'est pas immédiate, et d'autre part, on n'est pas toujours sûr que la spécification formelle décrit la totalité du comportement attendu vu qu'elle s'appuie sur des descriptions informelles.

2.4.2. *Formalisation de l'implémentation*

La formalisation de l'implémentation d'un système consiste en la construction d'un modèle formel de l'implémentation du système à partir de son programme source codé en VHDL ou Verilog. Elle se fait en deux étapes : la production de sa spécification formelle et la preuve de sa correction.

Dans ce contexte, deux aspects sont très importants : le niveau d'abstraction et la solidité (*soundness*) du modèle développé. Le niveau d'abstraction signifie le niveau de détails dans lequel on décrit le modèle formel de l'implémentation. Le programme en code source contenant tout ce qui concerne le système dans ses moindres détails, il faut alors savoir ne modéliser que les parties dont on a vraiment besoin. En réalité, se lancer dans une formalisation complète de l'implémentation peut rendre complexe le modèle et par la suite fastidieux le processus de vérification. Toutefois, il faut aussi savoir ne pas négliger des détails de taille dont l'absence peut entraîner facilement une modélisation erronée. Par exemple, en modélisant un circuit au niveau des portes logiques (gate level), il serait complètement absurde de ne pas prendre en considération le délai des portes. Enfin, il s'agit ici de trouver un compromis entre ce qu'on souhaite vérifier et le niveau de détail établi dans le modèle.

La vérification du modèle formel de l'implémentation commence par établir le lien entre l'implémentation et la spécification donnant ainsi la nature du théorème de correction à établir. Il s'agit d'un théorème d'équivalence (↔) si l'implémentation et la spécification décrivent exactement le même comportement. C'est un théorème d'implication (→) si la spécification est partielle c'est à dire ne décrit que quelques propriétés.

2.5. *Avantages et inconvénients de la vérification formelle*

Quand les aspects de la conception à vérifier sont subtiles ou compliqués, seule la vérification formelle est capable de donner une réponse complète (Kauf *et al.*, 2000). Grâce à ses notations mathématiques, ce type de vérification ne peut être que concis et l'ambiguïté n'a

pas de place. Son principal atout est celui de pousser le concepteur à poser « la bonne question » à un stade plus au moins avancé de la conception du système vérifié. Par conséquent, la vérification formelle assure un examen très rigoureux du système permettant ainsi une livraison plus rapide et moins coûteuse (Hall, 1990). Enfin, les systèmes soumis à des tests formels ne peuvent avoir qu’un degré de fiabilité plus grand que ceux testés par simple simulation (Kropf, 1997).

Malgré les multiples avantages que peut avoir la vérification formelle du matériel, il faut aussi être conscient de ses limitations (Kropf, 1999). En réalité, il est impossible de garantir par vérification formelle qu’un système est correct dans l’absolu c'est-à-dire « le zéro erreur ». En outre, le processus de vérification peut être induit en erreur à cause de spécifications formelles ou informelles incomplètes voire erronées, ou encore par inconsistance dans le modèle d’implémentation. De plus, la portée de la vérification formelle est toujours limitée à la conception : elle ne permet de détecter que les fautes de conception (design faults) mais jamais des fautes reliées à la fabrication du produit ou à son utilisation.

Une autre limitation de la vérification formelle consiste en l’impossibilité de l’accomplissement de cette tâche à l’intégralité du système. Quand on vise à appliquer ce type de vérification à un système cela ne veut pas dire qu’on cherche à démontrer formellement la correction du système en entier. Souvent, la vérification formelle ne se fait que pour certains composants. Il s’agit là de vérifier une propriété d’un modèle de ce composant. La propriété à vérifier ici est souvent critique. Malheureusement, même les composants vérifiés formellement peuvent être facilement induit en erreur. Par exemple, il suffit que le composant vérifié reçoive en entrée des données invalides en provenance d’un autre composant non vérifié.

3. Un peu de logique

Les outils de vérification formelle sont toujours fondés sur l’utilisation de logiques. Il existe plusieurs logiques différentes. Parmi elles, on trouve la logique classique dont la logique propositionnelle est une composante. Nous présentons dans la suite quelques notions sur la logique propositionnelle qui est souvent la base des démonstrateurs de théorèmes. Par la suite, nous aborderons brièvement les logiques temporelles qui constituent quant à elles la base des vérificateurs de modèles.

3.1. *La logique propositionnelle*

La logique propositionnelle est l'une des logiques les plus simples. Elle permet de représenter de façon abrégée et non ambiguë des « réalités » qui sont souvent des affirmations. La logique propositionnelle offre aussi des opérateurs de coordination permettant de réaliser différents arrangements plus au moins complexes entre les variables (tableau 5).

- **Un petit exemple :** une affirmation du type « Je suis étudiant » est représentée par une variable *p* dont la valeur ne peut prendre que les deux valeurs : vrai ou faux. Elle ne prendra la valeur « vrai » que si je suis effectivement étudiant en ce moment. Dans le cas contraire, elle prendra la valeur « faux ».
- **Axiome :** les axiomes constituent la base de la logique propositionnelle. Un axiome est une expression qu'on assume être un théorème sans pour autant donner sa démonstration. Une liste exhaustive d'axiomes est disponible dans (Gries *et al.*, 1994).
- **Théorème :** un théorème est soit un axiome, soit la conclusion d'une règle dont les prémisses sont des théorèmes, soit une expression dont on démontre qu'elle est égale à un axiome ou à un théorème précédemment démontré (Desh, 2000). La démonstration d'un théorème est possible par un processus de dérivation en appliquant des règles appelées règles d'inférence à certains axiomes. Puisque les axiomes ont la propriété d'être vrai et on sait déjà que les règles d'inférence préservent cette propriété, alors un théorème est toujours évalué à vrai. Il s'agit là d'une propriété fondamentale pour les théorèmes (Kauf *et al.*, 2000).
- **La syntaxe de la logique propositionnelle :** la logique propositionnelle possède une syntaxe bien déterminée. En utilisant cette syntaxe, il est possible de traduire les phrases ou les descriptions informelles telles que « Je suis étudiant et je travaille » en des axiomes du type *« $p \wedge q$ »* en représentant le fait que « je suis étudiant » par la variable *p* et le fait que « je travaille » par *q*. La connexion entre les deux variables *p* et *q* est possible grâce au connecteur *« $\wedge$ »* qui dénote *« et »*. La liste de tous les autres connecteurs utilisée dans la logique propositionnelle est présentée dans le tableau 5. Tableau 5
- **Avantage et limite :** il est clair que la logique propositionnelle permet de traduire des phrases informelles en fournissant un bon degré d'abstraction. Cependant, c'est une logique qui reste très peu expressive puisqu'elle ne permet pas d'exprimer des notions telles que le temps.

Tableau 5. *Syntaxe de la logique propositionnelle*

Opérateurs de coordination	Équivalent en français	Utilisation	Nom de l'opérateur
$\neg$	Non	$\neg p$	Négation
$\wedge$	Et	$p \wedge q$	Conjonction
$\vee$	Ou	$p \vee q$	Disjonction
$\Rightarrow$	Implique	$p \Rightarrow q$	Implication
$\Leftarrow$	Conséquence	$p \Leftarrow q$	Conséquence
$\not\equiv$	Inéquivalent	$p \not\equiv q$	Inéquivalence
$\equiv$	Équivalent	$p \equiv q$	Équivalence
$=$	Égal	$p = q$	Égalité

3.2. *Les logiques temporelles*

Il existe deux grandes catégories de logiques temporelles : qualitatives et quantitatives.

3.2.1. *Les logiques temporelles qualitatives*

Les logiques temporelles qualitatives ont été introduites par (Man, 1982) afin d'exprimer des propriétés dynamiques des programmes séquentiels et concurrents, telles que *« toute exécution d'un programme commençant à un état initial doit atteindre un état terminal »*. L'inconvénient principal de ce type de logique est leur expressivité limitée de l'aspect temporel. En réalité, elle exprime l'aspect « temps » de façon très abstraite : il n'est pas possible par exemple de modéliser à quel instant se produit un état ou un évènement, ni sa durée, ni la durée qui le sépare d'un autre évènement…

Il existe dans la littérature plusieurs logiques temporelles qualitatives. Parmi elles, on peut citer CTL (Computational Tree Logic) (Clar *et al.*, 1986), FIL (Future Interval Logic) (Kut, 1994).

3.2.2. *Les logiques temporelles quantitatives*

La seconde classe de logique est constituée des logiques temporelles quantitatives (à temps discret ou à temps continu). C'est une classe qui garde les propriétés de la classe précédente en permettant en plus d'exprimer le temps de façon explicite. Ainsi, le temps est simulé par le nombre entier de fois où un état apparaît dans une séquence (temps discret) ou par une variable quelconque réelle (temps continu). MTL (Koy, 1990) et CTL*[DC] (Pandya, 2001) sont des exemples de logiques temporelles quantitatives.

4. Les méthodes de vérification formelle

Il existe plusieurs catégories de méthodes permettant d'effectuer la vérification formelle. Dans ce qui suit, nous allons nous intéresser de près aux 2 techniques de base : la vérification de modèles et la démonstration de théorèmes. Nous étalerons ainsi les principes, les avantages, les inconvénients et quelques uns des outils relatifs à chacune de ces techniques. Les autres techniques existantes sont en général de type hybrides combinant la vérification de modèles ou la démonstration de théorèmes avec la simulation numérique.

4.1. *Les méthodes basées sur la vérification de modèles*

4.1.1. *Principe*

La vérification de modèles (model checking) modélise d'une part le système à vérifier par un automate à états finis (AFS), et d'autre part la propriété à vérifier par une formule logique. La vérification se base ensuite sur l'exploration de l'espace d'états de l'automate afin de vérifier s'il satisfait ou non la propriété modélisée.

En pratique, la technique de vérification de modèles est réalisable de deux façons : l'équivalence de modèles et la vérification temporelle de modèles. La première technique appelée aussi « equivalence checking » consiste à vérifier si deux descriptions matérielles possèdent des spécifications fonctionnelles équivalentes. Par opposition, la vérification temporelle de modèles « temporel model checking » vérifie si une contrainte temporelle, exprimée dans une logique temporelle donnée, est respectée par une description matérielle.

4.1.2. *Quelques vérificateurs de modèles*

La diversité des algorithmes d'exploration de l'espace d'états du système à vérifier, a donné naissance à une multitude de vérificateurs de modèles ou « model checkers ». Suivant le type de la logique utilisée, on peut classer ces model-checkers en deux grandes catégories : les model-checkers qualitatifs et les model-checkers quantitatifs. Nous présentons brièvement ici un exemple de vérificateur de chaque classe mais nous ne donnerons pas plus de détails dans la mesure où nous ne nous intéresserons pas aux techniques basées sur la vérification de modèles dans la suite.

- **Les model-checkers qualitatifs :** un model-checker qualitatif se base sur l'utilisation des logiques temporelles qualitatives lors de la description du comportement du système et des propriétés à vérifier. Un exemple de vérificateur de modèles qualitatif est SMV (Symbolic

Model Verifier). Développé par K. L. McMillan à l'université Carnegie-Mellon (McMill, 1992), SMV est basé sur les automates non temporisés. Il permet d'exprimer les propriétés à vérifier telles que « ordre d'événement » et « fin de tâche » dans la logique CTL. A la fin de la vérification, SMV indique en sortie si la propriété considérée est satisfaite par le système modélisé. En cas d'échec, il fournit un contre exemple. Bien qu'il est de type qualitatif et ne donne pas une notion explicite du temps, SMV est considéré comme étant le langage complet pour la description matérielle.

- **Les model-checkers quantitatifs :** pour vérifier une propriété de vivacité bornée du type « *Toute requête finira par être satisfaite en au moins de 5 mn.* » pour un système, on a besoin de model-checkers quantitatifs. Un vérificateur de modèles quantitatif est basé sur une logique temporelle quantitative. Il modélise le système par des automates temporisés ou hybrides rajoutant des structures permettant de calculer le temps.

Parmi les model-checkers quantitatifs, on peut citer DCVALID qui permet la vérification des formules exprimées en CTL [DC] (Pandya, 2001), UPPAAL (UPPAAL) qui est un ensemble d'outils pour la vérification automatique des propriétés de sûreté et de vivacité bornée, des systèmes temps réel.

4.1.3. *Avantages et inconvénients du model-checking*

Grâce à la méthode basée sur la vérification de modèles, il est possible de couvrir entièrement l'espace d'états du système. Cette couverture complète de l'espace d'états n'était pas possible par simulation classique. Ainsi, cette technique s'avère beaucoup plus efficace que la simulation classique. De plus, la vérification de modèles a l'avantage d'être automatique et rapide. Une fois, le model-checker est lancé sur la vérification de la propriété modélisée, il n'est plus possible de l'arrêter jusqu'à ce que soit il donne un contre exemple montrant la non-satisfiabilité de la propriété, soit qu'il admet que la propriété est vérifiée.

Toutefois, la technique de vérification de modèles présente aussi beaucoup d'inconvénients. En réalité, la vérification de modèles n'est autre qu'une simulation exhaustive de l'espace d'états du système à l'aide d'algorithmes astucieux. Donc, une fois que cet espace d'états devient très grand, les algorithmes d'exploration montrent leurs limites entraînant ainsi une limitation majeure pour ce type de technique. Ce genre de problème est plus connu sous le nom du problème d'explosion d'états. Des solutions potentielles à ce problème consistent en l'utilisation des diagrammes de décision binaire (BDD) (Bry, 1986),

ou encore en la réduction modèle à vérifier de façon à le restreindre aux parties concernées par la propriété à vérifier (Ber *et al.*, 1998).

4.2. *Les méthodes basées sur la démonstration de théorèmes*

4.2.1. *Principe*

Les méthodes basées sur la démonstration de théorèmes ont pour but de démontrer qu'un énoncé est la conséquence logique d'un ensemble d'énoncés (les axiomes et les hypothèses). Tous les énoncés doivent être formulées dans le langage logique du démonstrateur. Le moteur de preuve tente ensuite de démontrer la conjecture à partir des axiomes, des règles, des définitions dérivées à partir des lemmes intermédiaires et des hypothèses. Le processus de démonstration est basé sur l'utilisation de techniques telles que de la déduction logique, la réécriture et encore la récurrence…

4.2.2. *Etude de quelques démonstrateurs de théorèmes*

Deux cent dix-neuf outils différents de démonstration de théorèmes sont énumérés par Freek Wiedijk (Freek) dans une liste intitulée *« overview of systems implementing mathematics in the computer »*. Parmi ces outils, quatre-vingt ont été vérifiés dans la recherche de (Bol, 2001). Nous nous sommes principalement inspiré de cette recherche pour présenter quelques outils de démonstration de théorèmes. Un récapitulatif de ces démonstrateurs est illustré au tableau 6.

- **Isabelle :** c'est un démontreur de théorème générique populaire développé à l'université de Cambridge et au TU Munich (Isabel). Les logiques existantes comme Isabelle/HOL fournissent un environnement de démonstrateur de théorème prêt à être utilisé pour des applications de taille. Isabelle peut également servir comme châssis pour un prototypage rapide de systèmes déductifs. Il est présenté avec une grande bibliothèque comprenant Isabelle/HOL (logique classique d'ordre supérieur), Isabelle/HOLFS (Logique de Scott pour des fonctions calculatoire avec HOL), Isabelle/FOL (logique du premier ordre classique et intuitive), et Isabelle/ZF (ensemble de théories de Zermelo-Fraenkel au dessus de FOL).

- **PVS :** l'outil PVS (PVS) est un prouveur pouvant être utilisé pour réaliser de la vérification du matériel (Owre *et al.*, 1994). Il a montré son efficacité lors de son application à des systèmes industriels assez complexes tels que la vérification d'un processeur destiné à être embarqué dans des avions (Miller *et al.*, 1995) et l'analyse d'un switch ATM (Rajan, 1997).

- ACL2 : ACL2 (A Computational Logic for Applicative Common LISP) est à la fois un langage de programmation, un langage de spécification basé sur GCL (GNU Common Lisp) et une logique mathématique formelle permettant la démonstration de théorèmes semi-automatique (ACL2). Le système ACL2 est considéré comme l'un des démonstrateurs les plus fiables. En 2005, on lui a décerné « l'ACM Software Award » (ACM, 2006) le nommant comme le plus efficace des démonstrateurs pour la vérification des systèmes logiciels et matériels critiques (« for pioneering and engineering a most effective theorem prover (...) as a formal methods tool for verifying safety-critical hardware and software »). L'outil ACL2 sera abordé avec plus de détails dans la section 4 de ce chapitre.

- Z/EVES : créé par ORA Canada lors d'un projet débuté en 1996 (Zeves), Z-eves est un logiciel gratuit basé sur un système automatique de preuve appelé NEVER. Il réunit la puissance du démonstrateur Eves (ORA, 1999) avec le langage de spécification Z (Spiv, 1992). D'un apprentissage relativement facile, Z-eves a l'avantage de tourner sur les deux plateformes Unix et Windows. En plus, ses preuves avec la théorie des ensembles présentent un bon degré d'automatisme.

- Coq : Coq est un démonstrateur de théorèmes (Coq). Il permet ainsi d'énoncer des spécifications de programmes sous forme de théorèmes mathématiques et de développer interactivement les preuves formelles de ces théorèmes ou spécifications au moyen de « tactiques ». Coq intègre en plus un langage de programmation fonctionnelle. Il permet aussi la communication avec des logiciels externes tels que les systèmes de calcul formel ou les prouveurs automatiques.

Tableau 6. *Récapitulatif de quelques démonstrateurs*

Outil	***Type de logique***	***Plate forme***	***Langage de base***	***Site web***
PVS	classique	Unix	-	http://pvs.csl.sri.com/
Isabelle	classique et intuitionniste	Linux, Solaris	Standard ML	http://www.cl.cam.ac.uk/Research/HVG/Isabelle/
ACL2	-	Unix, Windows, Macintosh	Common Lisp	http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html

Z-eves	classique	Unix, Windows	Common Lisp	http://www.ora.on.ca/z-eves/welcome.html
Coq	constructive	Unix, Windows	Objective CAML	http://pauillac.inria.fr/coq/

4.2.3. *Avantages et inconvénients du theorem-proving*

Capable de démontrer des propriétés algorithmiques complexes, la technique de vérification par démonstration de théorèmes a prouvé son efficacité dans le domaine de la vérification des composants matériels. Elle s'applique à tous les niveaux d'abstraction et elle est particulièrement efficace à un haut niveau d'abstraction pour des systèmes très complexes (la taille des données n'a plus d'importance) et ceci indépendamment du type du système ou du circuit considéré (séquentiel ou combinatoire).

Cependant, le principal inconvénient des démonstrateurs de théorèmes est leur faible degré d'automatisme. En réalité, l'utilisateur doit nécessairement assister l'outil de preuves lors des démonstrations non triviales, et les démonstrations reflétant souvent des propriétés complexes, risquent d'être lentes et difficiles. On évoque dans ce contexte le problème du « chapeau mexicain » repris à la figure 12. Pour arriver à démontrer le théorème final, cela nécessite souvent une énorme quantité de lemmes intermédiaires qui sont en général non réutilisables. De ce fait, pour une grande majorité des démonstrateurs, l'interactivité entre l'utilisateur et l'outil peut nécessiter un temps d'apprentissage assez long.

De plus, la majorité des démonstrateurs souffrent d'un problème d'incomplétude. Ainsi, si une preuve de la formule proposée existe alors la formule est un théorème. Dans le cas où la preuve échoue, le démonstrateur est incapable de décider que la formule non démontrée n'est pas un théorème. Néanmoins, l'idéal serait d'avoir des démonstrateurs complets où on peut affirmer qu'une formule à démontrer est un théorème en retournant la preuve correspondante ou décider de la non-existence de la preuve.

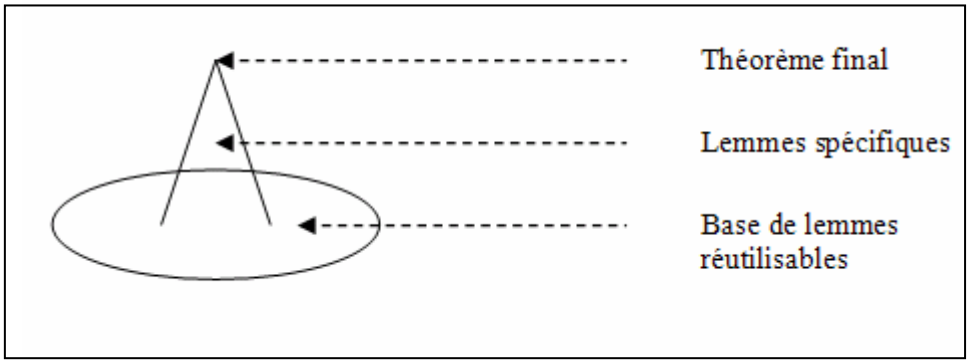


Figure 12. *Chapeau mexicain*

4.3. *Le choix de la méthode de vérification formelle*

Le choix de la méthode de vérification formelle doit se faire principalement en fonction de la nature du problème de vérification. Dans (Kropf, 1997), T. Kropf donne tout un diagramme de décision permettant de sélectionner l'approche de vérification la plus appropriée suivant le type de circuit en question. Un chapitre est consacré pour chacune des techniques identifiées.

De façon similaire, le choix de l'outil (model-checker ou theorem-prover) est principalement guidé par le type de systèmes qu'il supporte (temps réel ou non pour les model-checkers) et par le type de propriétés que peut modéliser l'outil. En réalité, il n'est pas toujours possible de tout représenter dans un model-checker et les propriétés à vérifier varient entre l'atteignabilité, la sûreté, le non-bFlocage et la vivacité. La même limitation peut survenir aussi pour les démonstrateurs. Le fait que sa logique se limite à une logique de premier ordre peut amener à convertir tout ce qui doit être exprimer dans une logique d'ordre supérieur à une logique de premier ordre. Dans le cas où cette conversion induit à une perte dans les concepts qu'on souhaite formuler alors il est obligatoire de choisir un autre outil plus expressif tel que HOL (HOL) ou PVS (PVS). Les critères de performance tels que le temps consommé ou le blocage lors de la vérification, peuvent aussi être des critères déterministes.

Dans notre cas d'étude, le but étant le développement d'un modèle formel des réseaux multi-étages dédiés aux MPSoCs, nous avons alors opté pour une méthode basée sur la démonstration de théorèmes. En effet, nous souhaitons traduire dans une notation formelle les spécifications informelles de ce type de réseaux tout en tenant compte des contraintes des systèmes sur puce. Autrement dit, il s'agit de formaliser des spécifications afin de vérifier ensuite une ou quelques propriétés sur le modèle *(cf. la section 2.4.1.)*. D'autres considérations ont dû aussi guider ce choix. Elles seront illustrées dans le chapitre 4. Dans ce qui suit, nous détaillons l'outil choisi pour notre travail de formalisation.

5. Le système ACL2

Dans cette partie, nous exposons l'outil de démonstration de théorèmes ACL2 que nous avons choisi dans le cadre de notre travail.

5.1. *Présentation d'ACL2*

ACL2 ou « A Computational Logic for Applicative Common LISP » est la réunion d'un langage de programmation fonctionnelle, une logique mathématique et un démonstrateur de théorèmes au sein d'un même outil. La logique ACL2 n'est autre qu'un sous ensemble du langage de programmation Common LISP, alors que le démonstrateur de théorèmes est une version industrielle puissante du démonstrateur de théorèmes de Boyer et Moore Nqthm (Kauf *et al.*, 1996). ACL2 a été développé dans le principal but de répondre aux difficultés trouvées par les utilisateurs de Nqthm en appliquant ce dernier à des projets formels de grande échelle. Comme son prédécesseur, ACL2 a été aussi conçu à Austin au Texas. Sa première version a vu le jour en 1989 et elle a été réalisée par R. Boyer et J. Moore. En 1995, Moore et Kaufmann ont pu donner naissance à la première version publique d'ACL2.

Le démonstrateur d'ACL2 est un programme qui prend en entrée un théorème potentiel et essaie de trouver la preuve mathématique correspondante. Une preuve est définie comme une structure finie montrant la dérivation du théorème à partir des axiomes. ACL2 ne crée pas une preuve réellement formelle. Il vérifie plutôt que la preuve en sortie peut en principe être transformée en une preuve formelle. Lors de la preuve, le démonstrateur se réfère au monde logique qui est constitué essentiellement d'axiomes, de définitions et de théorèmes préalablement démontrés.

Selon (Kauf *et al.*, 2000), les théorèmes à démontrer par ACL2 doivent être à la fois intéressants et difficiles à prouver. Autrement, des petits théorèmes de valeur et non trivial traitant des parties de systèmes très compliqués.

L'une des difficultés majeures que rencontre l'utilisateur ACL2, notamment le novice, est de procurer de l'aide à un démonstrateur ayant échoué lors d'une preuve. Les développeurs d'ACL2 sont conscients de cette difficulté. Pour cela, ils ont essayé de traiter cet aspect en fournissant toute une méthodologie pour aborder de façon efficace les démonstrations (Kauf *et al.a*, 2000).

5.2. *La démarche dans ACL2*

Afin d'accomplir la vérification formelle d'un système dans ACL2, l'utilisateur doit suivre la démarche illustrée par la figure 13. Il doit alors commencer par spécifier le modèle fonctionnel du système qui est représenté par des fonctions définies dans le langage de programmation fonctionnelle d'ACL2 (sous ensemble du langage Common Lisp).

L'utilisateur a alors la possibilité de simuler le model fonctionnel ainsi construit à travers un environnement d'exécution (execution environment). Dans une deuxième phase, la vérification requiert de traduire les théorèmes (proposed definitions conjectures) concernant le modèle dans la logique du démonstrateur. Enfin, l'utilisateur peut procèder à la preuve de ces théorèmes par le biais du démonstrateur fournit (theorem prover). Au cours des démonstrations, ACL2 fait appel à des règles (rules) stockées préalablement dans sa base de données (database). En plus, il doit être guidé par une stratégie spécifique inspirée par l'utilisateur. Une fois démontré, un théorème peut être converti sous le contrôle de l'utilisateur en une règle (rule generator) qui va mettre à jour le monde logique. Si la preuve échoue, l'utilisateur doit avoir recours à deux alternatives : reformuler le théorème ou donner des guides additionnelles (proposed advices) au démonstrateur et ceci en examinant les tentatives de preuves ayant échoué. Les guides additionnelles sont des conseils formulés sous la forme de lemmes intermédiaires.

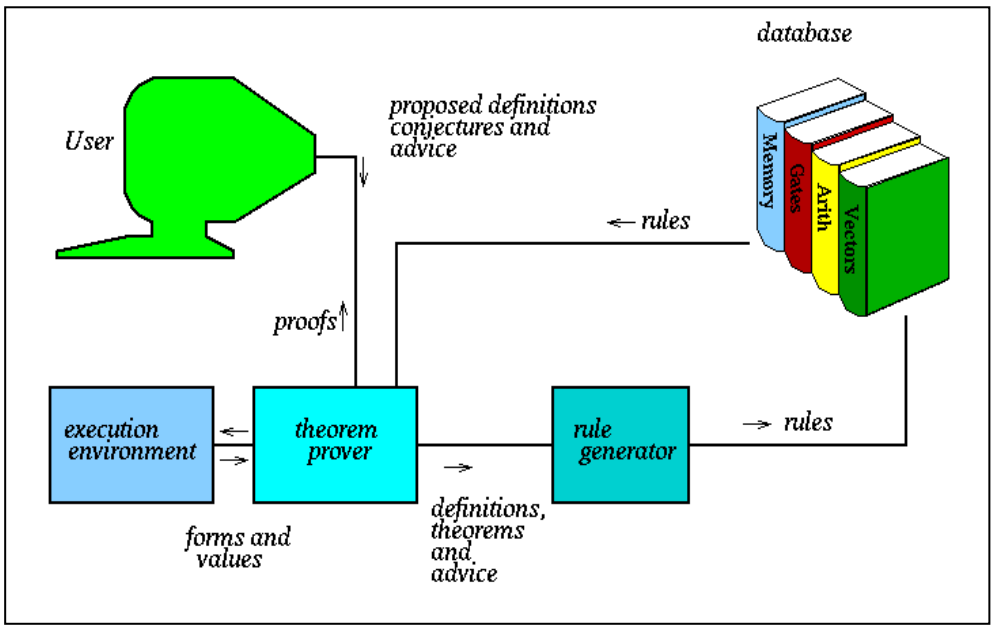


Figure 13. *Démarche générale (ACL2)*

- Les mécanismes de preuve : ACL2 utilise des techniques variées pour démontrer des théorèmes de premier ordre. Parmi ces techniques, on trouve la réécriture, la simplification par la substitution répétée de « equals for equals », les procédures de décision, l'induction mathématique et d'autres techniques de preuves exprimées pour des fonctions définies récursivement et pour des objets construits de façon inductive.

- Les propriétés du démonstrateur : comme la majorité des démonstrateurs de théorèmes, ACL2 est interactif : il doit être guidé par l'utilisateur. Ce dernier est le seul responsable de la stratégie utilisée dans une preuve. L'aspect interactivité dans ACL2 impose que l'utilisateur soit capable de comprendre l'intégralité des tentatives de preuves pour pouvoir fournir l'aide nécessaire au démonstrateur. ACL2 est aussi automatique. Une fois il est lancé sur une preuve alors il n'est pas possible de l'interrompre. Cependant, des preuves directes sans interactivité avec l'utilisateur ne sont valables que pour des théorèmes simples. Pour des problèmes compliqués, les preuves directes sont impossibles. En fait, le démonstrateur de théorèmes se trouve limité par des considérations pratiques telles que le temps de calcul.

5.3. *Quelques principes dans ACL2*

5.3.1. *Le principe de définition*

Le langage de programmation fonctionnelle d'ACL2 ne contient pas les structures de contrôle itératives telles que for, foreach, while… La programmation des fonctions se base alors sur les définitions récursives pour remédier à ce manque. Dans ACL2, une fonction récursive ne peut être admise que si pour chacun des appels récursifs qu'elle contient, on peut trouver un argument ou une combinaison d'arguments dont la mesure décroît selon une relation bien fondée.

Un exemple de relation bien fondée est « < » sur le domaine des entiers naturels. ACL2 utilise EO-ORD pour admettre une fonction récursive ayant des arguments de type entier. De plus, ACL2 utilise une mesure appelée ACL2-COUNT pour ordonner les termes de même type entre eux. ACL2-COUNT peut être une mesure assez évidente comme dans le cas des entiers, ACL2-COUNT est la valeur d'un entier, ou dans le cas des listes, la mesure vaut la longueur d'une liste. Cependant, dans le cas où la mesure n'est pas immédiate, il faut fournir de l'aide à ACL2 en lui indiquant la mesure qu'il doit utiliser.

5.3.2. *Le principe d'induction*

Le principe d'induction généralisant le principe de récurrence, se base sur la notion de relation bien fondée. Ainsi, la propriété P à démontrer est vérifiée pour le cas de base de x sachant que x est la variable d'induction. Ensuite, on suppose que P est vrai pour un élément y dont la mesure est inférieure à celle de x et on essaie de montrer que P est vrai pour x.

5.3.3. *Le principe d'encapsulation*

Le principe d'encapsulation est un principe avancé dans ACL2. Il permet d'introduire des fonctions ne possédant pas de définitions explicites. Ce sont des fonctions définies par un ensemble de contraintes qui sont exprimées par des théorèmes. Il est obligatoire de fournir à la commande « encapsulate » des fonctions locales, appelées aussi fonctions témoins, qui vérifient les contraintes. Les fonctions témoins conservent la cohérence de la logique lors de l'introduction de symboles de fonctions encapsulées.

5.4. *Exemples de travaux réalisés avec ACL2*

ACL2 a prouvé réellement son efficacité quand il a été déployé pour produire des preuves pour des composants de systèmes très compliqués tels que :

- le circuit arithmétique élémentaire de calcul flottant du processeur AMD Athlon : vérification que la description RTL de ce circuit respecte le standard IEEE correspondant (Russ, 1998). Le même travail a été effectué sur le processeur AMD K5 (Moore *et al.*, 1998),
- le DSP de Motorola : vérification qu'un modèle micro architectural de ce DSP implémente un microcode donné et que le microcode spécifique extrait à partir de la ROM implémente certains algorithmes des DSP (Broc *et al.*, 1999),
- le compilateur javac de Sun : vérification que le « JVM bytecode » produit par ce compilateur pour de simples classes, implémente les fonctionnalités désirées (Moore, 2003),
- le démonstrateur Ivy du National Argonne Labs ; vérification de la propriété de « soundness » d'un programme Lisp qui vérifie les preuves produites par ce démonstrateur (Kauf *et al.*, 2000)

6. Conclusion

Les industriels des systèmes sur puce sont de plus en plus conscients du danger et du risque qu'il peut y avoir en négligeant l'étape de vérification lors de la conception de leurs produits. L'intégration des méthodes formelles dans leur cycle de développement se fait donc progressivement. Les techniques hybrides combinant model-checking et simulation numérique et la technique de vérification de modèles se trouvent largement utilisée par ces industriels. Actuellement, on cite même des produits de vérification formelle produit par des entreprises géantes de la microélectronique tels Cadence, Synopsis, IBM et Intel. En revanche, vu les difficultés d'apprentissage que présente la technique de démonstration de théorèmes, il est moins fréquent d'entendre parler d'industriels utilisant cette dernière.

Dans le cadre de notre travail, nous avons choisi de travailler avec la technique de démonstration de théorèmes et ceci afin de vérifier formellement les réseaux multi-étages dédiés aux MPSoCs. Dans le chapitre suivant, nous présenterons une étude détaillée des réseaux multi-étages (MINs).

Chapitre 3 : Etude des réseaux multi-étages dédiés aux MPSoCs

1. Introduction

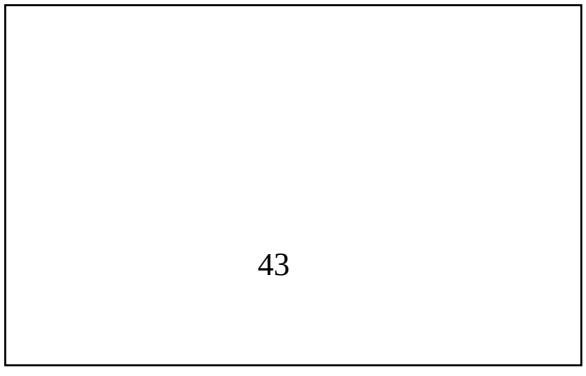
Les réseaux multi-étages dénotés MINs pour Multistage Interconnection Networks, ont été fréquemment utilisés au sein des systèmes multiprocesseurs classiques, ainsi que dans les commutateurs du réseau ATM.

Dans ce chapitre, nous présenterons les réseaux multi-étages : leur architecture, leurs propriétés et leur technique de routage. Nous allons en particulier insister sur une sous-classe très intéressante de ces réseaux formés par la famille Delta. Nous dresserons notamment un résumé des différentes permutations possibles dans les Delta MINs.

2. Architecture des réseaux multi-étages (MINs)

2.1. *Les réseaux statiques vs les réseaux dynamiques*

Les réseaux statiques sur puce possèdent des liens fixes entre les processeurs et un algorithme de routage prédéterminé. Le réseau point à point de la figure 14 est un exemple de réseau statique. Ce type de réseaux est approprié pour des applications structurées dans lesquelles les patrons de communication sont prévisibles. Dans tous les autres cas, le fait d'avoir un réseau fixe peut être désavantageux. Par exemple, pour des applications non structurées demandant de la flexibilité (e.g. contraintes temporelles variables sur différents processus), il serait plus intéressant d'avoir un réseau adaptable aux besoins dynamiques en communications. Pour ce faire, les noeuds de calcul ne doivent plus être reliés de façon permanente.



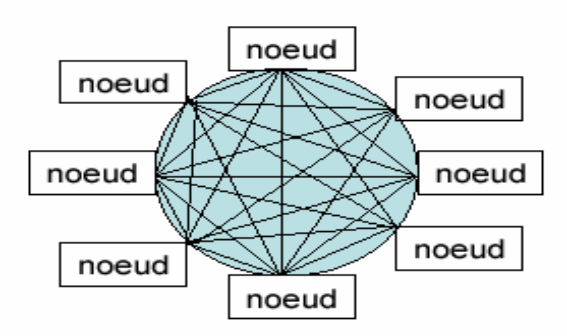


Figure 14. *Réseau point à point*

Dans un réseau dynamique, il y a plutôt un « commutateur global » (réseau de commutation formé de plusieurs commutateurs) liant les différents processeurs, ce qui permet de configurer de façon dynamique les liens directs entre les processeurs c'est-à-dire la topologie du réseau. Après que les commutateurs du réseau de commutation soient activés, un ou plusieurs liens directs existent entre les noeuds. L'activation des commutateurs correspond donc, à la fois, à la formation d'un réseau adapté à la communication désirée et au routage du ou des messages.

Les réseaux dynamiques peuvent être à un étage ou à plusieurs. On appelle les réseaux à un étage réseaux « Crossbar » (figure 15). Le réseau « Crossbar » est caractérisé par une bonne performance. Toutefois, son nombre de liens est aussi élevé que celui d'un réseau point à point. C'est un nombre qui devient excessif pour des réseaux de grande taille. Pour les réseaux à plusieurs étages ou multi-étages dénotés aussi MINs (Multi-stage Interconnection Networks), on compte plusieurs colonnes de commutateurs qui sont reliés entre eux et avec les noeuds de calcul. Les MINs se situe entre la connexité minimale des réseaux en bus et la connexité maximale des réseaux de commutation matricielle. Ces réseaux seront abordés avec plus de détails dans le reste de ce chapitre.

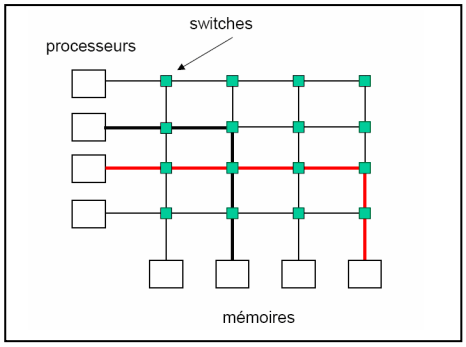


Figure 15. *Réseau crossbar*

Tableau 7. *Comparaison de quelques topologies*

Propriété	**Bus**	**Crossbar**	**Multi-étages**
Vitesse	basse	élevée	élevée
Coût	bas	élevé	moyen
Fiabilité	basse	élevée	élevée
Configurabilité	élevée	basse	moyenne
Complexité	basse	élevée	moyenne

Le tableau 7 compare trois topologies parmi les plus populaires au moyen de quelques propriétés intéressantes. On remarque que sur les trois citées, les réseaux multi-étages présentent les critères les plus intéressants : une vitesse et une fiabilité élevées avec un coût moyen. Dans ce qui suit, nous allons nous intéresser de près à ce type de réseaux.

2.2. *L'historique des MINs*

Les réseaux multi-étages sont utilisés depuis longtemps dans les systèmes multiprocesseurs pour connecter des processeurs aux modules mémoires (Gott *et al.*, 1983) (Pfis *et al.*, 1985). Ils sont par exemple fréquemment utilisé dans les machines parallèles tels IBMSP (Stun *et al.*, 1995) et CRAY Y-MP (Cheu *et al.*, 1986).

Les MINs de type banyan ont en plus servi pour réaliser les interconnexions à l'intérieur des commutateurs ATM (Zegu, 1993) (Giac *et al.*, 1991). Ils étaient par exemple proposés pour le réseau Broadband ISDN (B-ISDN). Dans une architecture multi-étages, le switch n'est plus formé d'un seul étage mais de plusieurs. Les paquets en provenance du réseau passent à travers les différents étages composés de commutateurs (switches) élémentaires. De cette manière, le switch peut profiter d'un certain degré de parallélisme.

2.3. *Formalisme de description des réseaux multi-étages*

Un réseau multi-étages *NxM* est un réseau d'interconnexion dynamique reliant *N* entrées à *M* sorties et construit par des commutateurs de taille *a x b*. Ces commutateurs sont ordonnés en étages $(Stg_d,Stg_{(d-1)},...,Stg_0)$ interconnectés par des étages de connexions $(C_d,C_{(d-1)},...,C_1,C_0)$. Ces derniers sont réalisés grâce à des permutations. Une permutation est en général une permutation sur les bits formant l'adresse d'un nœud.

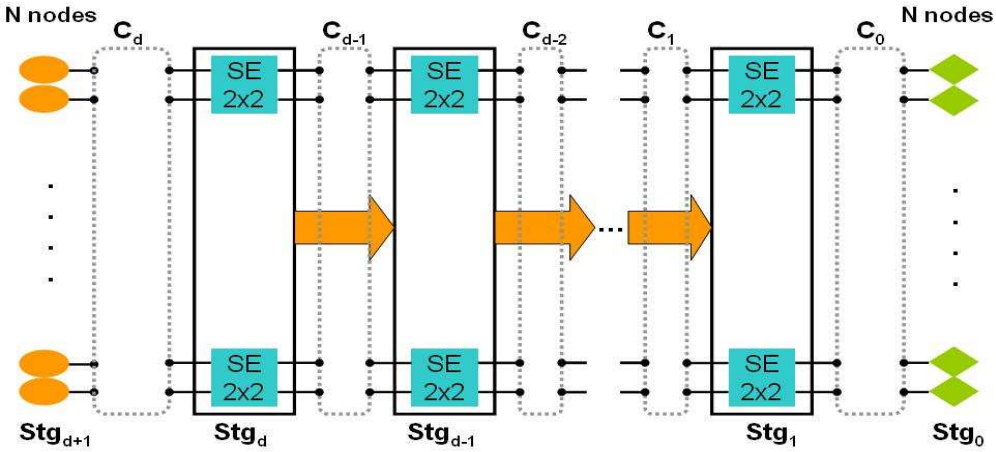


Figure 16. *Architecture de MIN générique*

La majorité de MINs utilisés sont de taille *N* et de degré *r*. On dit que ce sont des MINs (N,r) où *N* et *M* sont égaux et les commutateurs sont de degré *r*. La figure 16 illustre l'architecture d'un MIN générique *NxN* utilisant des crossbars de degré 2x2. Toutefois, il reste toujours possible de travailler avec des MINs tels que *N* et *M* sont différents. Ainsi, si *N* strictement supérieur à *M*, des commutateurs *axb* avec *a* supérieur à *b (a > b)* seront utilisés. Ces commutateurs sont appelés des commutateurs de concentration (concentration switches). Dans le cas où on a *N* est strictement inférieur à *M*, on utilisera des commutateurs avec *a* inférieur à *b (a < b)*. De tels commutateurs sont appelés des commutateurs de distribution (distribution switches). Le Delta MIN de la figure 17 est un MIN de taille *N* égal à 6 utilisant des commutateurs *3x3* sur son premier étage.

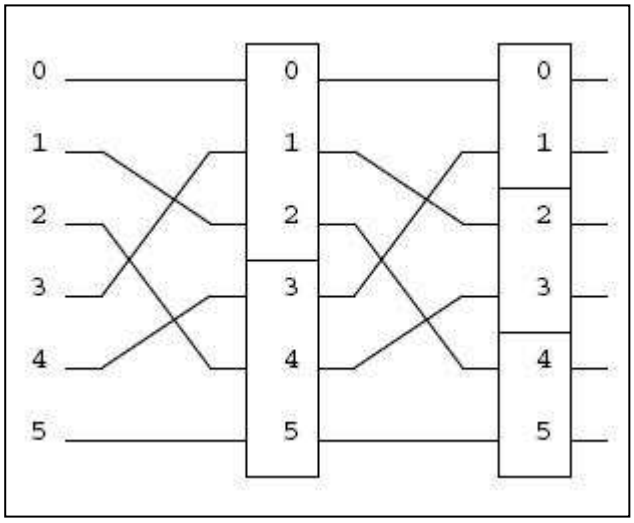


Figure 17. *Un Delta MIN de taille 6*

2.4. *Classification des MINs*

Il existe divers critères de classification des réseaux multi-étages (Szym *et al.*, 1994). En effet, on peut distinguer plusieurs classes de MINs suivant les types de commutateurs utilisés et / ou les types de permutation. Avant de présenter la classification choisie pour les MINs, il est important de commencer par explorer quelques définitions. Un réseau multi-étages est dit :

- *banyan :* s'il offre un chemin unique entre n'importe quelle entrée et n'importe quelle sortie. Il est appelé aussi « unipath network ».
- *uniforme :* si tous les commutateurs d'un même étage sont du même degré r.
- *carré ou « square »* : si les étages $(Stg_d, Stg_{(d-1)}, \ldots, Stg_0)$ sont constitués de commutateurs du même degré *r*. On l'appelle aussi MIN de degré *r*.
- *rectangulaire :* c'est un MIN ayant le même nombre d'entrées et de sorties c'est-à-dire *N* est égale à *M*.
- *bloquant :* s'il n'est pas toujours possible d'établir une connexion entre un couple d'entrée-sortie même lorsque la sortie en question n'est pas en cours d'utilisation.
- *non bloquant :* Si de toute entrée inactive il existe toujours un chemin vers toute sortie inactive. On peut donc effectuer n'importe quelle permutation en cours d'exécution. Un exemple populaire de réseau sans blocage est le réseau de Clos.

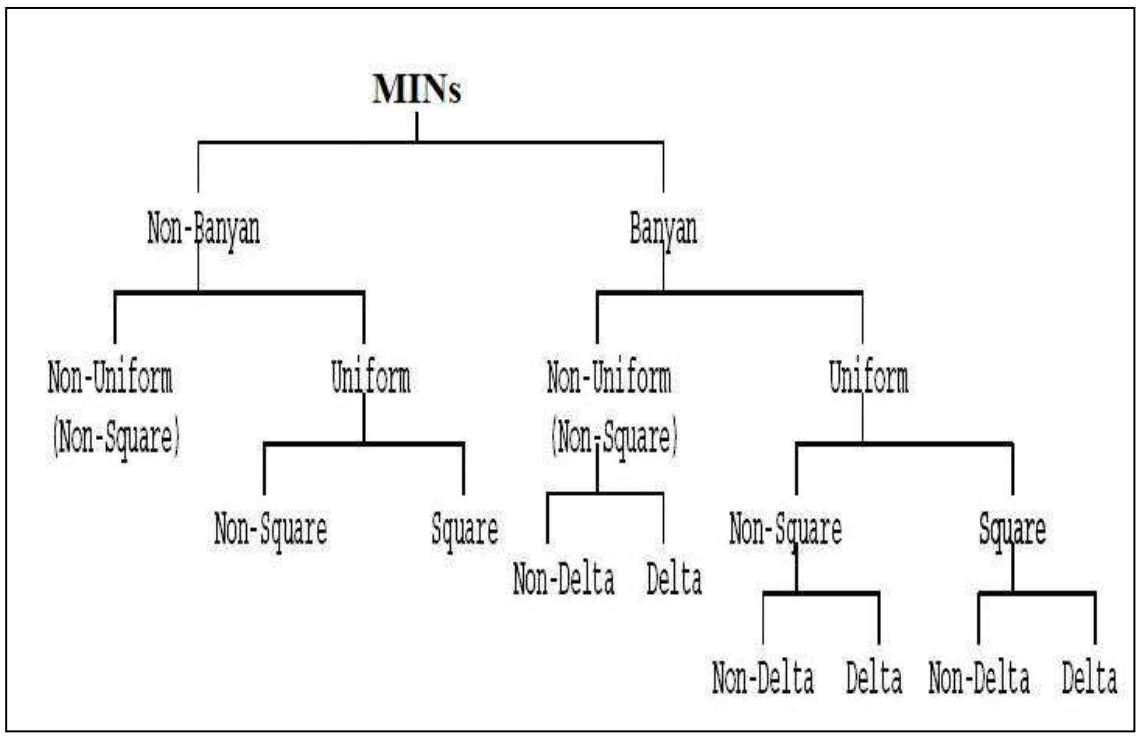


Figure 18. *Classification topologique des MINs*

La figure 18 représente une classification topologique des MINs. En réalité, un MIN peut posséder ou non la propriété banyan. Dans un MIN banyan non uniforme (Banyan Non-Uniform), il existe au moins un étage renfermant des commutateurs de degré différent. Donc, il est clair que ce type de MIN est non carré (Non-Square). Les MINs banyan peuvent avoir ou non la propriété Delta. C'est une propriété qui décrit une sous-classe importante de réseaux banyans appelé les réseaux Delta.

Les MINs qui présentent de multiples chemins entre toute paire de source-destination forment la classe des MINs non banyan. Ils ont l'avantage d'être tolérants aux fautes mais ils sont généralement plus coûteux et plus complexe à contrôler que les MINs banyan.

2.5. *Construction d'un réseau MIN*

Une permutation est une application (mapping) bijective *N:N* où chacun des *N* éléments d'entrée (source) est relié à un et un seul des *N* éléments de sortie (destination) et réciproquement. En assignant à chaque élément d'entrée et de sortie une étiquette ou adresse entre 0 et N-1, la permutation peut être définie par une modification de l'adresse de chaque élément d'entrée pour produire l'adresse de l'élément de sortie correspondant. En général, on utilise une adresse binaire et la permutation correspond à une opération sur les bits formant l'adresse. Dans un réseau MIN, les liens à chaque étage et à la sortie sont agencés en fonction d'une permutation parmi plusieurs types standard. Les différents types de permutations utilisés dans les MINs seront détaillés plutard au cours de ce chapitre.

2.6. *Propriétés d'un réseau MIN*

Le réseau d'interconnexion multi étages possède les propriétés d'être :

– *Dynamique et réarrangeable* : il permet plus d'un chemin pour établir une liaison entre une entrée libre et une sortie libre.

– *Tolérant aux pannes* : on peut établir tous les chemins même avec le blocage d'un commutateur au niveau de chaque sous-réseau qui le constitue.

– *Efficace et plus pratique et moins coûteux* pour augmenter le nombre de processeurs.

2.7. *Les réseaux Delta*

2.7.1. *Définition*

Les réseaux Delta proposés par Patel (Patel, 1981) sont construits à l'aide de crossbars *axb*. La propriété Delta est définie comme suit : on désigne par O_i ($i= 0,1,\ldots ,b-1$) l'output d'index *i* d'un crossbar dans un MIN, si un input d'un crossbar de l'étage *j* est connecté à l'output O_i d'un crossbar de l'étage *j-1*, alors tous les autres inputs de ce crossbar doivent être connectés à l'étage précédent en des outputs de crossbars ayant le même indice *i*. De façon générale, on peut dire qu'un réseau Banyan est un réseau Delta si tous ses étages ont la propriété Delta.

La figure 19 illustre la propriété Delta sur une portion d'un réseau MIN. On note que l'input *O0* du crossbar 3 est connecté à un output d'indice *0* de l'étage *j-1*. La propriété Delta exige que tous les inputs du crossbar 3 soient connectés à l'étage *j-1* en des outputs ayant le même index *0*. Ce qui est le cas puisque l'input I_1 du crossbar 3 est connecté à un output d'index *0* de l'étage *j-1*.

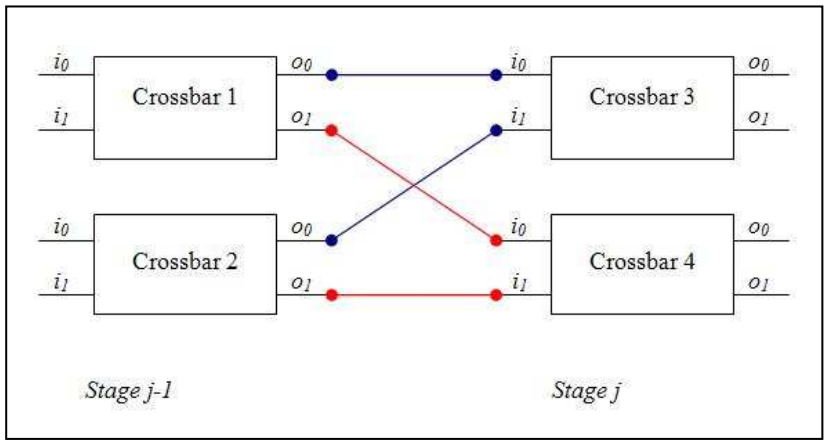


Figure 19. *Illustration de la* propriété *Delta*

2.7.2. *Propriétés des réseaux Delta*

Les réseaux multi étages de type Delta ont des propriétés très intéressantes :

- *une connexité moyenne* : un nombre de commutateurs de l'ordre de $Nlog_kN$ qui reste plus faible que N^2 pour les réseaux matriciels,

– *un accès total* : les types de permutations utilisées pour construire les étages de connexion, doivent garantir l'accès total au réseau. Ainsi, par une configuration correcte des commutateurs à chaque étage, *n'importe quelle entrée doit être capable d'atteindre n'importe quelle sortie*,

– *un routage simple* : un routage distribué qui se fait en fonction de l'adresse de destination et ceci indépendamment de la source,

– *une équivalence topologique* : il a été prouvé dans (Arga, 1983), (Wu et *al.*, 1980) et (Coll, 2002) que tous les MINs Delta sont équivalents du point de vue topologique. Il suffit de réordonner les positions des commutateurs sans rompre les connexions pour passer d'un réseau à un autre.

2.7.3. *Panorama des réseaux Delta*

Plusieurs types de réseaux Delta sont définis en fonction des types de permutations utilisées. Il existe plusieurs formes (types) de réseaux Delta, dépendamment de leurs connexions. Dans ce qui suit, nous détaillerons les Delta MINs les plus utilisés ainsi que les réseaux inverses correspondants (reverse). Pour obtenir d'un réseau inverse, il suffit de regarder le schéma du réseau original de gauche à droite ou juste inverser la numérotation adoptée.

Les réseaux présentés sont de taille *N* et constitués de *d* étages de commutateurs de degré *2* chacun. Nous adopterons la numérotation utilisée dans la figure 16.

- *Le réseau Oméga :* sa connexion C_0 utilise la permutation identité I qui ne produit aucune permutation, alors que toutes les autres connexions utilisent la permutation « perfect shuffle » σ^k. Cette dernière consiste en un *décalage cyclique de tous les bits de l'index d'une position vers la gauche* (formule 3-1). Ainsi, le réseau Oméga est décrit mathématiquement par la formule (3-2). La figure 20 illustre un Oméga (16,2).

$$\sigma^k (x_{n-1}\ x_{n-2} \ldots x_1\ x_0) = x_{n-2}\ \ldots x_1\ x_0\ x_{n-1} \qquad \textbf{(3-1)}$$

Définition 1. *Modèle de connexion du réseau Oméga*

$$\begin{cases} \text{Pour } 1 \le i \le d, & C_i = \sigma^k \\ \text{Pour } i = 0, & C_0 = I \end{cases} \qquad \textbf{(3-2)}$$

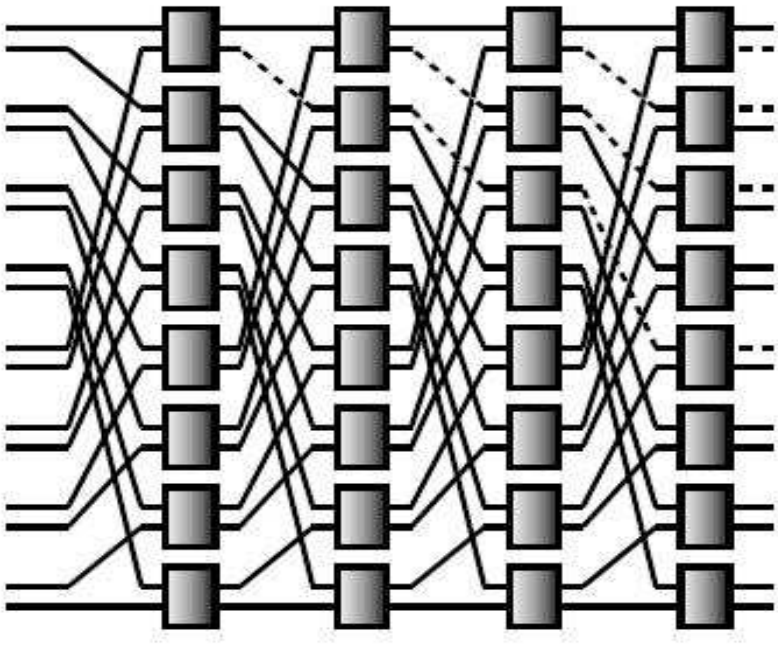


Figure 20. *Un réseau Oméga (16,2)*

- *Le réseau Flip :* il est considéré comme l'image miroir du réseau Oméga, le réseau Flip utilise la permutation inverse du « perfect-shuffle ». Dans ce cas, *le décalage cyclique est effectué vers la droite* et appelé $\sigma^{k\ -1}$ (formule 3-3). Le modèle de connexion du réseau Flip est décrit par la formule (3-4).

$$\sigma^{k\ -1}\ (x_{n-1}\ x_{n-2}\ldots x_1\ x_0) = x_0\ x_{n-1}\ x_{n-2}\ldots x_2\ x_1 \qquad \textbf{(3-3)}$$

Définition 2. *Modèle de connexion du réseau Flip*

$$\begin{cases} \text{Pour } 0 \le i \le d\text{-}1, & C_i = \sigma^{k\ -1} \\ \text{Pour } i = d, & C_d = I \end{cases} \qquad \textbf{(3-4)}$$

- *Le réseau Butterfly :* c'est un réseau qui utilise la permutation identité *I* au premier étage et en sortie et la permutation « butterfly » dans tous les autres étages. La permutation « butterfly » est *un échange entre le $i^{ème}$ bit et le bit 0*. Elle est par la formule (3-5). Un exemple de réseau Butterfly est illustré par la figure 21. De façon générale, le réseau Butterfly est défini par les permutations présentées dans la définition 3.

$$ß_i^k\ (x_{n-1}\ x_{n-2}\ldots x_{i+1}\ x_i\ x_{i-1}\ \ldots\ x_1\ x_0) = x_{n-1}\ x_{n-2}\ldots x_{i+1}\ x_0\ x_{i-1}\ \ldots\ x_1\ x_i. \qquad \textbf{(3-5)}$$

Définition 3. *Modèle de connexion du réseau Butterfly*

$$\begin{cases} Pour\ i = d,\ C_d = \sigma^k \\ Pour\ (1 \le i \le d\text{-}1),\ C_i = \beta_i^k \\ Pour\ i=0,\ C_0,\ \beta_0^k \end{cases} \quad \textbf{(3-6)}$$

- *Le réseau Reverse Butterfly :* le réseau Reverse Butterfly qui n'est autre que l'image miroir du Butterfy, est décrit par la définition 4.

Définition 4. *Modèle de connexion du réseau reverse Butterfly*

$$\begin{cases} Pour\ i=d,\ C_d = \beta_0^k \\ Pour\ (1 \le i \le n\text{-}1),\ C_i = \beta_{(d-i)}^k \\ Pour\ i = 0,\quad C_0 = \sigma^{k^{-1}} \end{cases} \quad \textbf{(3-7)}$$

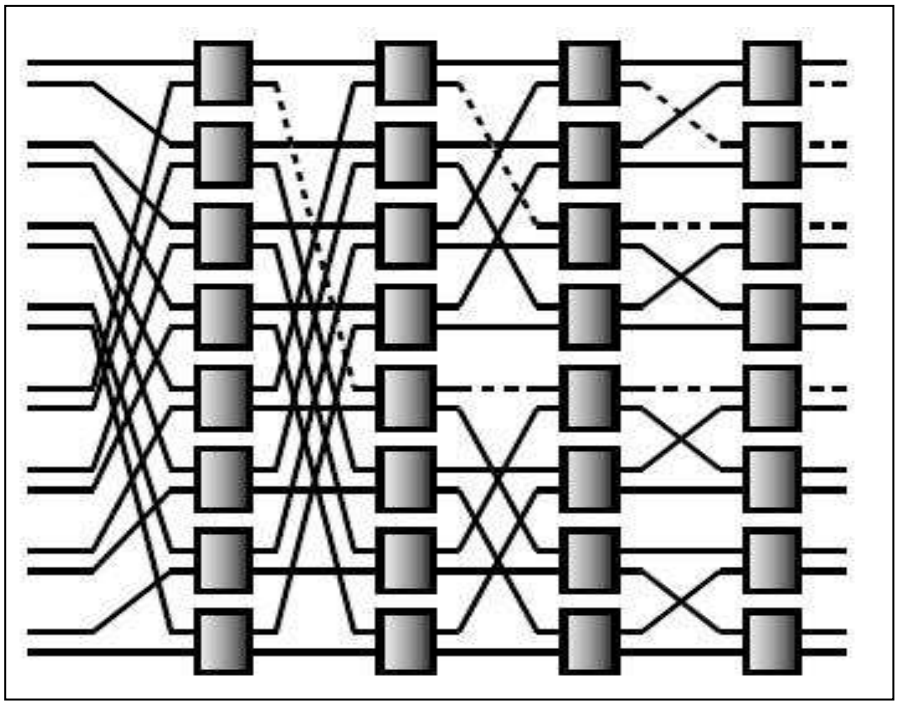


Figure 21. *Un réseau Butterfly (16, 2)*

- *Le réseau Baseline :* il est définie à l'aide de la permutation $\boldsymbol{\delta_i^k}$. Elle consiste en *un décalage cyclique d'une position vers la droite, des (i+1) bits les moins significatifs* de

l'index (formule 3-8). La figure 22 illustre un exemple de Baseline (8,2). Ainsi, le réseau Baseline est défini par les permutations suivantes de la formule (3-9).

$$\delta_i^k (x_{n-1}\, x_{n-2} \ldots x_{i+1}\, x_i\, x_{i-1} \ldots\ x_1\, x_0) = x_{n-1}\, x_{n-2} \ldots x_{i+1}\, x_0\, x_i\, x_{i-1} \ldots\ x_1 \quad \textbf{(3-8)}$$

Définition 5. *Modèle de connexion du réseau Baseline*

$$\begin{cases} \text{Pour } i = d,\ C_d = I \\ \text{Pour } (1 \le i \le d\text{-}1),\ C_i = \delta_i^k \\ \text{Pour } i = 0,\ C_0 = I \end{cases} \quad \textbf{(3-9)}$$

- *Le réseau Reverse Baseline :* il est l'image miroir du Baseline. Il utilise alors la permutation inverse de $\boldsymbol{\delta_i^k}$ appelée $\boldsymbol{\delta_i^{k\,-1}}$. $\boldsymbol{\delta_i^{k\,-1}}$ est *un décalage cyclique d'une position vers la gauche, des (i+1) bits les moins significatifs de l'index* (formule 3-10). La formule (3-11) décrit le mode de connexion du Reverse Baseline.

$$\delta_i^{k\,-1} (x_{n-1}\, x_{n-2} \ldots x_{i+1}\, x_i\, x_{i-1} \ldots\ x_1\, x_0) = x_{n-1}\, x_{n-2} \ldots x_{i+1}\, x_{i-1} \ldots\ x_1\, x_0\, x_i \quad \textbf{(3-10)}$$

Définition 6. *Modèle de connexion du réseau reverse Baseline*

$$\begin{cases} \text{Pour } i = d,\ C_d = I \\ \text{Pour } (0 \le i \le d\text{-}1),\ C_i = \delta_{(d-i)}^{k\,-1} \\ \text{Pour } i = 0,\ C_0 = I \end{cases} \quad \textbf{(3-11)}$$

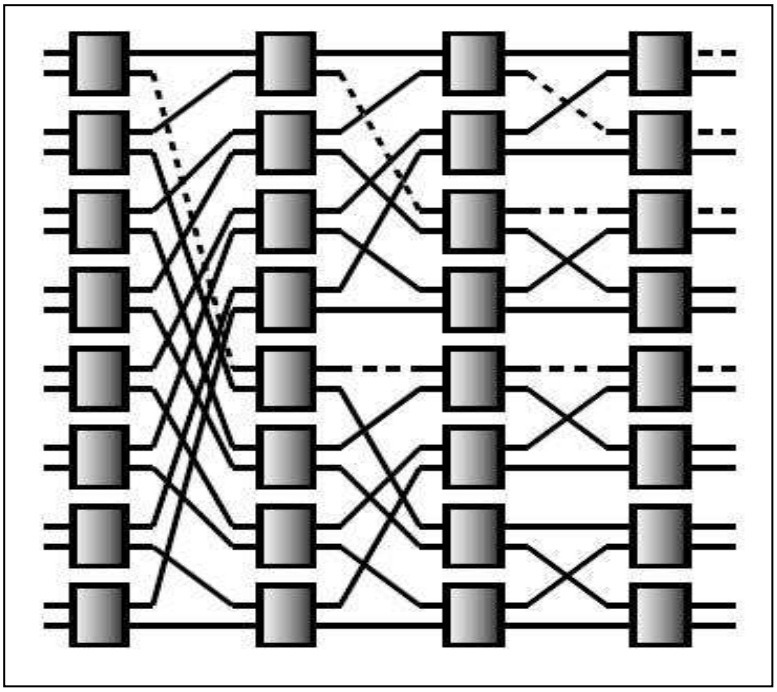


Figure 22. *Un réseau Baseline (16,2)*

2.7.4. *Synthèse des formalismes de description des Delta MINs*

Le tableau 8 résume les différents types de réseaux Delta, alors que le tableau 9 représente les définitions formelles des permutations qui sont utilisées pour construire ces réseaux. Il est à noter que les permutations $ß_0^k$, δ_0^k et $\delta_0^{k^{-1}}$ représentent la fonction identité *I*.

Tableau 8. *Panorama des réseaux Delta*

permutation C_k ***réseau Delta***	C_d	$C_{i\ [1..d-1]}$	C_0
Oméga	σ^k	σ^k	I
Flip	I	$\sigma^{k^{-1}}$	$\sigma^{k^{-1}}$
Butterfly	σ^k	$ß_i^k$	I
Reverse Butterfly	I	$ß_{(d-i)}^k$	$\sigma^{k^{-1}}$

Baseline	I	δ_i^k	I
Reverse Baseline	I	$\delta_{(d-i)}^{k\ -1}$	I

Tableau 9. *Formalisme de description des permutations*

Fct	***Définition formelle***	***Explication***
σ^k	$\sigma^k (x_{n-1}\ x_{n-2} \ldots x_1\ x_0) = x_{n-2}\ \ldots x_1\ x_0\ x_{n-1}$	décalage cyclique de tous les bits de l'index d'une position vers la gauche
$\sigma^{k\ -1}$	$\sigma^{k\ -1} (x_{n-1}\ x_{n-2} \ldots x_1\ x_0) = x_0\ x_{n-1}\ x_{n-2} \ldots x_2\ x_1$	décalage cyclique de tous les bits de l'index d'une position vers la droite
$ß_i^k$	$ß_i^k (x_{n-1} \ldots x_{i+1}\ x_i\ x_{i-1}\ \ldots\ x_1\ x_0) = x_{n-1} \ldots x_{i+1}\ x_0\ x_{i-1}\ \ldots\ x_1\ x_i$	échange entre le i^{ème} bit et le bit 0
δ_i^k	$\delta_i^k (x_{n-1} \ldots x_{i+1}\ x_i\ x_{i-1}\ \ldots\ x_1\ x_0) = x_{n-1} \ldots x_{i+1}\ x_0\ x_i\ x_{i-1}\ \ldots\ x_1$	décalage cyclique d'une position vers la droite, des (i+1) bits les moins significatifs de l'index
$\boldsymbol{\delta_i^{k\ -1}}$	$\boldsymbol{\delta_i^{k\ -1} (x_{n-1} \ldots x_{i+1}\ x_i\ x_{i-1}\ \ldots\ x_1\ x_0) = x_{n-1} \ldots x_{i+1}\ x_{i-1}\ \ldots\ x_1\ x_0\ x_i}$	décalage cyclique d'une position vers la gauche, des (i+1) bits les moins significatifs de l'index

2.7.5. *Le routage dans les réseaux Delta*

Les réseaux Delta sont caractérisés par un mode de routage qui est très simple. L'adresse destination présentée dans la base r où r est le degré du réseau, va servir comme étant une séquence de contrôle pour router le message à travers les différents commutateurs. Le message à livrer va être alors commuté à l'output d'indice *i* du commutateur courant, si le digit correspondant de la séquence de contrôle est égal à *i*.

Pour aller de gauche à droite, la séquence de contrôle est considérée dans le même sens, c'est-à-dire de gauche à droite. Par contre, pour aller de droite à gauche, il faudra considérer l'adresse destination *k bits par k bits* dans le sens contraire. D'ailleurs, pour que l'algorithme de self-routing marche dans un réseau de type « reverse Delta MIN », il faudrait considérer l'adresse destination de droite à gauche.

On associe donc aux réseaux MINs de type Delta la propriété d'auto routage (self-routing) dans le sens que le canal de sortie choisi à chaque commutateur ne dépend pas de la source mais seulement de la destination. La figure 23 illustre le routage d'un message de l'entrée d'adresse 001 vers la sortie d'adresse 110 et ceci dans un Delta MIN (8,2).

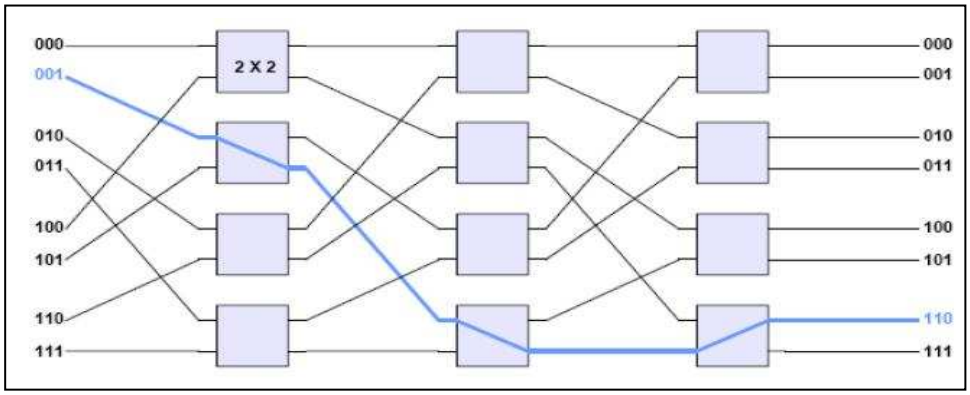


Figure 23. *Routage dans un Delta MIN (8,2)*

3. Conclusion

Dans ce chapitre, nous avons passé en revue les différentes notions relatives aux réseaux multi-étages (MINs). Nous avons insisté en particulier sur les réseaux de la famille Delta. Vu les propriétés intéressantes que possède cette dernière classe de réseaux, notre travail de formalisation se focalisera sur cette famille de réseaux.

Chapitre 4 : Formalisation générique des réseaux sur puce

1. Introduction

Dans le premier chapitre de ce manuscrit, nous avons étudié différents travaux de formalisation relatifs aux réseaux sur puce. Nous avons alors constaté que ces travaux pourraient être classé en deux catégories : spécifique et générique. La première catégorie est taillée pour le système à vérifier, elle ne permet pas donc la réutilisation du modèle formel construit *(Æthereal, AMBA).* Par opposition à cette première catégorie, la seconde est caractérisée par une approche commune valable pour tous les réseaux sur puce *(Octagon, Hermes).* C'est une approche illustrée à travers le modèle générique GeNoC (Generic Networks on Chip).

Nous présenterons dans ce chapitre la conception de l'extension du modèle GeNoC dans le cadre de la formalisation des réseaux multi-étages dédiés aux MPSoCs. Une étude détaillée de GeNoC sera aussi exposée.

2. Formalisation générique : GeNoC

GeNoC représente un modèle formel générique décrivant les communications dans les réseaux sur puce (Schm, 2006). Les architectures de communication ont plusieurs concepts en commun tels que les interfaces, la topologie, le routage et l'ordonnancement. GeNoC modélise dans un style fonctionnel ces notions clés en ne faisant aucune hypothèse sur la topologie, le type de routage ou de l'ordonnancement. Son critère de correction est *la non modification d'un message transmis depuis une source vers une destination*.

2.1. *Les fonctions de GeNoC*

En réalité, GeNoC est un modèle défini en fonction de quatre fonctions clés (figure 24). Ces dernières n'ont pas une définition explicite mais elles sont plutôt exprimées en fonction de contraintes à satisfaire.

2.1.1. *Les fonctions « Send » et « Recv »*

Avant l'envoi d'un message sur le réseau, la couche interface du nœud source applique la fonction *« Send »* pour encapsuler ce message dans une trame (ou frame). Pour récupérer le message au niveau du nœud distant, la couche interface effectue l'opération inverse en appliquant la fonction *« Recv »*. Le codage et le décodage des trames se fait selon le modèle OSI (Open System Interconnection). *La contrainte principale* à satisfaire par ces deux

fonctions est la suivante : le récepteur doit être capable d'extraire l'information codée c'est à dire *(Recv o Send)* est égale au message initiale.

2.1.2. *La fonction « Routing »*

C'est la fonction responsable du routage des trames. Pour chacune de ces trames, elle va calculer la liste des routes autorisées. Ce calcul se base sur le principe des déplacements unitaires. *La contrainte principale* de la fonction *« Routing »* est que toute route entre un couple source-destination débute réellement à la source et emprunte uniquement les nœuds existants du réseau pour aboutir à la destination.

2.1.3. *La fonction « Scheduling »*

La fonction *« Scheduling »* modélise l'ordonnancement des trames. En ayant une liste de trames à ordonnancer, elle extrait la sous-liste de trames pouvant voyager simultanément sur le réseau. La définition de cette fonction exige *d'assurer* l'exclusion mutuelle entre les trames ordonnancées (*Scheduled*) et celles retardées (*Delayed*).

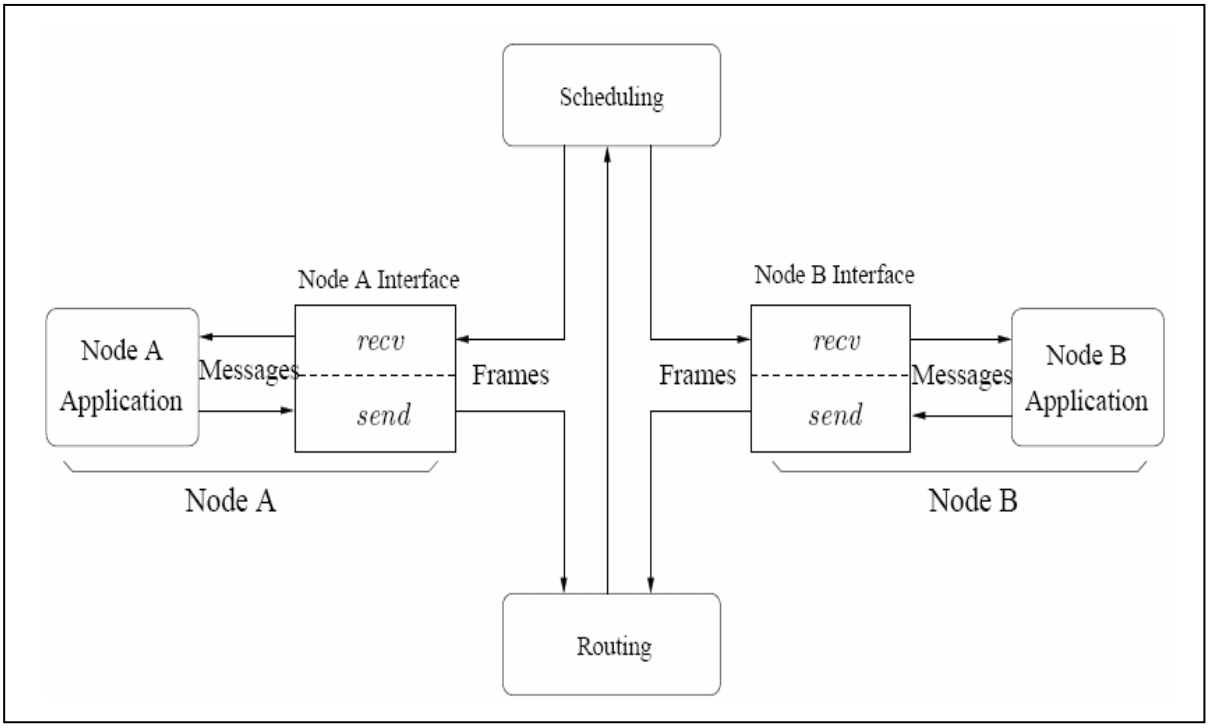


Figure 24. *GeNoC : un réseau générique*

2.2. *Déroulement de la fonction GeNoC*

La figure 25 illustre la logique du déroulement de la fonction GeNoC, ainsi que les différents objets manipulés. Cette figure est commentée comme suit :

- Un message à envoyer (*msg*) est une opération de communication modélisée par une transaction *t*. Cette transaction doit avoir une forme particulière : *« id s msg d »*, où *id* représente l'identificateur de *t* du message, *s*, le nœud source, *d*, le nœud destination et enfin *msg*, le message à envoyer. L'identificateur *id* doit être unique. La liste de toutes les transactions de toutes les applications actives forme l'ensemble *T*. La validité de *T* est reconnue par un prédicat dénoté T_{lstp},

- La fonction *« ComputeMissives »* applique la fonction *« Send »* de l'interface pour former la trame à partir du message *msg*. La trame construite est mise dans une missive *m* ayant la même forme que la transaction initiale *t*, la seule différence est que le message (*msg*) est remplacé par la trame (*frm*) correspondante. L'ensemble de toutes les missives *M* est reconnu par le prédicat M_{lstp},

- L'application de la fonction *« Routing »* donne pour chaque missive *m*, un voyage *v*. Un voyage *v* est un triplet constitué de l'identificateur (*id*) et de la trame (*frm*) de la missive initiale et de la liste des routes possibles dans le réseau (*Routes*). L'ensemble de tous les voyages forme *V*,

- L'ordonnancement de l'ensemble *V* dépend de la fonction *« Scheduling »*. Cette dernière divise *V* en deux sous-ensembles : une liste de voyages à effectuer (*Scheduled*) et une liste de voyages à retarder (*Delayed*). Les voyages retardés seront reconvertis en missives (par la fonction *« ToMissives »*) et repassés à la fonction de routage. Un nombre de tentatives est associé à chaque nœud pour qu'il puisse renvoyer ses messages retardés. Une fois ce nombre expire c'est-à-dire atteint 0, ces messages retardés seront considérés comme avortés et feront partie de l'ensemble résultat *A*. Les voyages valides sont reconnus par le prédicat V_{lstp},

- La fonction *« ComputeResults »* fait appel à la fonction *« Recv »* au niveau de l'interface du nœud récepteur. Ainsi, elle forme les messages résultats à partir des trames de l'ensemble *« Scheduled »*. Un message résultat noté *r* doit être égale à la transaction initiale moins le nœud source. La validité de l'ensemble résultat *R* est reconnue par le prédicat R_{lstp}.

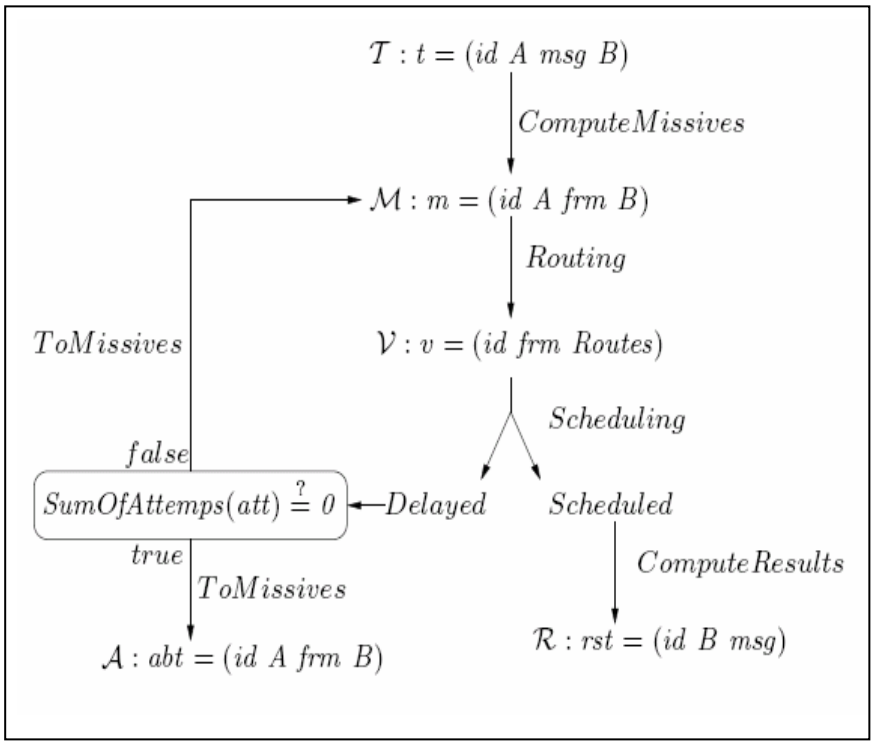


Figure 25. *Déroulement de GeNoC*

2.3. *Formalisation de GeNoC*

La formalisation concerne les nœuds, les interfaces, le routage et enfin l'ordonnancement des communications d'un réseau sur puce générique. Dans la suite, nous nous limiterons à la présentation des deux composantes qui intéressent notre travail : les nœuds et le routage.

2.3.1. *Formalisation des nœuds*

Le domaine de définition de tous les nœuds est *GenNodeSet*. Les éléments de ce domaine sont reconnus par le prédicat *ValidNodep*. *NodeSet* est un sous-ensemble de *GenNodeSet*, il forme les nœuds d'un réseau particulier. La fonction *NodeSetGen* prend en argument un paramètre *pms* valide c'est-à-dire reconnu par le prédicat *ValidParamsp* et génère un nœud de *NodeSet*. La définition de l'ensemble *NodeSet* doit vérifier le théorème 4-1. Cette obligation exprime le fait que pour tout paramètre *pms* valide, tout élément produit par la fonction *NodeSetGen* doit être un élément du domaine GenNodeSet ; c'est à dire vérifie le prédicat *ValidNodep.*

Théorème 4-1. *Définition de NodeSet*

$$\forall pms,\ ValidParamsp(pms) \Rightarrow \forall x \in NodeSetGen(pms),\ ValidNodep(x) \quad \textbf{(4-1)}$$

2.3.2. *Formalisation du routage*

Le routage désigne le mécanisme avec lequel les données sont acheminées depuis un nœud source jusqu'à la destination. Deux types d'algorithmes de routage : déterministe ou adaptatif. Le routage déterministe fournit un unique nœud constituant la prochaine étape dans la route de *s* vers *d*. Par opposition au routage déterministe, le routage adaptatif donne au niveau de chaque nœud intermédiaire une liste de prochains nœuds.

Que le routage soit déterministe ou adaptatif, il applique généralement une fonction ρ qui effectue le calcul des routes possibles entre un nœud source et un noeud destination. Pour chaque missive *m* de la liste *M*, la fonction *« Routing »* (définition 4-1) applique la fonction ρ pour calculer la liste des routes réalisables. Elle forme alors la liste des voyages *V*. Dans la définition 4-1, on utilise les symboles Id_M, Frm_M, Org_M et $Dest_M$ qui sont des fonctions GeNoC permettant l'accès respectif aux éléments d'une missive : l'identificateur *(id),* le contenu du message *(frm),* l'origine *(Org)* et la destination (*Dest).*

Définition 4-1. *Définition de « Routing »*

$$Routing\ (M, NodeSet) = \bigwedge_{m \in M} (List\ (Id_M(m), Frm_M(m), \rho(Org_M\ (m), Dest_M\ (m))))$$

(4-2)

La correction de la fonction *« Routing »* est exprimée par quatre obligations de preuve. Nous nous limitons ici aux trois obligations qui nous intéressent dans le cadre de ce travail :

- Le prédicat *ValidRoutep* reconnaît une route valide (définition 4-2). Une route *r* est valide que si elle débute au nœud source, se termine au nœud destination, tous les nœuds de la route sont inclus dans *NodeSet* et que la route comprend au moins deux nœuds. L'obligation de preuve de la validité des routes produites par ρ est donnée par le théorème 4-2. Elle exprime le fait que pour chaque missive m appartenant à un ensemble *M* valide, toute route *r* produite par la fonction ρ doit être valide (vérifie *ValidRoutep*). La longueur de la route serait égale exactement à deux si les deux nœuds la source et la destination du message sont directement reliés dans le réseau.

Définition 4-2. *Définition du prédicat « Ext-ValidRoutep »*

$$ValidRoutep\ (r, m, NodeSet) = \bigwedge \begin{cases} r[0] = Org_M\ (m) \\ r[l\text{-}1] = Dest_M\ (m) \\ r \subseteq NodeSet \ \wedge\ (len(r) \geq 2) \end{cases}$$

(4-3)

Théorème 4-2. *Validité des routes produites par la fonction* ρ

$$\forall M, M_{lstp}(M,NodeSet)$$
$$\Rightarrow \forall m \in M, \forall r \in \rho(Org_M(m), Dest_M(m)), ValidRoutep(r, m, NodeSet) \quad \textbf{(4-4)}$$

- Il faut montrer aussi que la liste des voyages générés à partir d'une liste de missives valides, est valide (reconnu par V_{lstp}). D'où l'obligation de preuve de la formule (4-5),

Théorème 4-3. *Validité des voyages produits par la fonction « Routing »*

$$\forall M, M_{lstp}(M,NodeSet) \Rightarrow V_{lstp}(Routing(M, NodeSet)) \quad \textbf{(4-5)}$$

- La troisième contrainte concerne la correspondance entre une missive et un voyage calculé par la fonction *« Routing »*. Ainsi, à chaque voyage calculé par la fonction de routage lui correspond une unique missive (théorème 4-4),

Théorème 4-4. *Correspondance entre missives et voyages*

$$\forall M, M_{lstp}(M,NodeSet)$$
$$\Rightarrow \forall v \in Routing(M, NodeSet), \forall m \in M, Id_V(v) = Id_M(m), Frm_V(v) = Frm_M(m) \quad \textbf{(4-6)}$$

2.4. *Analyse critique de la fonction GeNoC*

En plus des messages à router, la fonction *« Routing »* de GeNoC ne prend en considération que l'ensemble de nœuds *(NodeSet)*, et ceci en faisant abstraction des connexions qui peuvent être entre les différents nœuds du réseau. Une telle définition suppose en fait l'existence implicite des connexions entre deux nœuds successifs d'une route calculée. Ainsi, si l'algorithme de routage du réseau à vérifier permet d'aller d'un nœud *A* vers un nœud *B*, il existe alors une connexion implicite dans la topologie qui permet de joindre *B* à partir de *A*.

Le modèle GeNoC s'est montré très efficace lors de la validation formelle de réseaux sur puce tels que l'Octogon et l'Hermes. En réalité, les algorithmes de routage de ces NoCs donne explicitement le prochain nœud d'une route. Toutefois, l'algorithme de routage dans les réseaux multi-étages de type Delta est différent. En donnant seulement le port à travers lequel le message doit être commuté, le « self routing » ne donne aucune indication sur la

position du prochain commutateur. Pour connaître cette position, il faudra faire intervenir concrètement la connexion qui relie ce commutateur à l'étage suivant (Elle *et al.*, 2008).

En conséquence, et compte tenu de l'importance de l'aspect topologie dans les Delta MINs (soulignée dans le chapitre précédent), nous remarquons que l'application du modèle actuel de GeNoC pour valider ce type de réseau est impossible. Il faudra prendre en compte la composante topologie de façon plus explicite en faisant intervenir en plus des nœuds, les connexions.

3. Formalisation générique par extension du modèle GeNoC

Comme il a été mentionné plus haut, appliquer la version actuelle du modèle GeNoC pour valider de façon formelle les réseaux sur puce de type Delta MINs est impossible sans tenir compte réellement de l'aspect topologique des réseaux. L'idée est alors d'étendre le modèle générique GeNoC en lui ajoutant une composante topologie et en développant la composante routage étendu résultante de cette extension. Comme l'ensemble de nœuds a été déjà pris en considération dans le modèle GeNoC, nous nous focaliserons sur la généralisation des connexions. Par ailleurs, nous gardons la même notation GeNoC concernant la formalisation de l'ensemble de nœuds *NodeSet* exposée dans la section 2.3.1 de ce chapitre.

3.1. *L'aspect générique dans ACL2*

La traduction des définitions génériques dans ACL2 est possible grâce au principe d'encapsulation (Kauf *et al.*, 2001). Ce principe introduit sous certaines contraintes, des symboles de fonctions n'ayant pas une définition explicite. Les contraintes sont des théorèmes. Dans l'exemple ci-dessous, lorsque la fonction *f* encapsulée est admise, la théorie d'ACL2 est étendue par l'événement suivant *« la fonction f est contrainte par l'axiome ϕ »*. Ainsi, la fonction *f* ne possède pas une définition explicite mais on sait qu'elle possède la propriété ϕ à satisfaire. Par conséquence, une fonction *g* est une instance de la fonction encapsulée *f*, si et seulement, *g* satisfait les mêmes contraintes que *f*. Autrement dit, tous les théorèmes exprimés au niveau générique pour *f*, doivent être prouvés pour *g*.

```
(encapsulate (((f x1...xn) => *))
    (local (defun f (x1 ...xn)    β))
    (defthm thm-1 ϕ))
```

3.2. *La composante topologie générique*

En prenant en considération les aspects communs à toutes les architectures de communication des réseaux sur puce, nous détaillons ici la formalisation d'une composante générique décrivant leurs topologies (Elle *et al.a*, 2008).

3.2.1. *Principe*

La disposition des éléments (nœuds et liens) d'un réseau sur puce, en particulier les interconnexions physiques (réelles) et logiques (virtuelles) entre les noeuds, définit sa topologie. L'étude de la topologie physique du réseau est souvent assimilée à l'étude d'un graphe *(G)* dont les sommets sont les nœuds du réseau et les arcs sont les liens entre les paires de sommets. Ainsi, un graphe est d'habitude défini de façon statique par une collection de sommets *(V)* et une collection d'arêtes *(E)* (Gro *et al.*, 2005). Notre approche est différente. Elle est plutôt basée sur l'identification des fonctions d'interconnexions qui doivent être appliquées pour relier un nœud à un autre dans un graphe direct de la topologie.

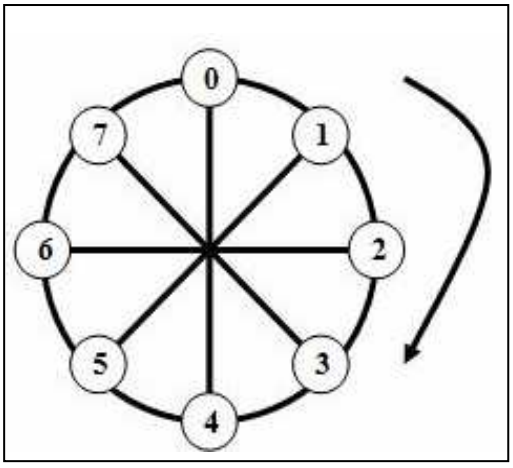
En réalité, un graphe de type direct est caractérisé par des arcs qui sont orientés d'un sommet vers l'autre. Un sommet *x* du graphe peut être connecté à un ou plusieurs autres sommets. Pour générer un lien à partir du sommet *x*, on doit appliquer une fonction mathématique dénotée *fp*. Cette dernière exprime la relation entre le sommet *x* et l'un de ses arcs sortants. Tous les arcs sortants de *x* sont alors le résultat de l'application d'une liste de fonctions désignée par *lfpx*. Nous supposons que, pour chaque sommet du graphe, une telle liste de fonctions mathématiques existe.

La figure 26 illustre un graphe simple de la topologie d'un Octagon à huit nœuds où chaque nœud est représenté par un nombre naturel (NodeSet = (0,1,2,..,7)). On pose N le nombre total de nœuds dans ce réseau, soit ici 8. En orientant le graphe dans le sens indiqué sur la figure, on peut identifier pour chaque nœud les trois fonctions de connexions à considérer. Ainsi, pour le nœud « 1 », on devra appliquer une fonction incrémentale « +1 » (connexion (1,2)), une fonction de décrémentation « -1 » (connexion (1,0)) et une fonction du type « + N/2 = + 4 » (connexion (1,5)).

Pour générer toutes les connexions d'un graphe donné, il existe deux alternatives. La première consiste en une procédure d'itération par nœud dans laquelle on identifiera pour chaque nœud la liste de fonctions de connexions qui doit lui être appliquée. L'inconvénient de cette alternative est le fait de devoir éliminer les connexions redondantes après avoir procéder par nœud ; ce qui peut être un peu lourd dans le cas où le réseau comporte beaucoup de

nœuds. Une deuxième alternative consisterait à identifier tous les nœuds auxquels on doit appliquer la même fonction de connexions *fp*. Cet ensemble formera un sous-ensemble de l'ensemble initial de nœuds. Les connexions obtenues par application de la fonction *fp* formeront quant à eux une sorte de sous-topologie de la topologie finale.

Dans le cadre de notre spécification générique, nous avons opté pour la deuxième alternative. Une façon de spécifier les connexions du réseau de la figure 26 selon cette dernière alternative, aboutit à l'identification de trois sous-ensembles de noeuds appelé ici X, Y et Z. X est l'ensemble formé par les nœuds (0,1,2,3,4,5,6) pour lesquels il faut appliquer la fonction de connexion incrémentale « +1 ». Le second ensemble Y est constitué des nœuds (0,1,2,3). Pour obtenir les connexions partant de ce sous-ensemble, il faut appliquer la fonction de connexion « + N/2 = + 4 ». Le troisième ensemble Z contient le singleton 7 à partir du quel on décrit la dernière connexion (7,0) par une fonction du type « -(N-1) = -7 ».



X = (0,1,2,3,4,5,6) : fp = « + 1 »

Y = (0,1,2,3) : fp = « +N/2 » = « + 4 »

Z = (7) : fp = « - (N - 1) » = « - 7 »

Figure 26. *Un graphe topologique du réseau Octagon*

3.2.2. *Spécification*

On définit la fonction notée *Gen-Cnx* qui génère tous les arcs sortants d'un sommet *x* à partir de la liste de fonctions de connexions correspondante notée *lfpx*. *Gen-Cnx* itère sur la liste *lfpx* pour appliquer chacune de ses fonctions et obtenir ainsi toutes les connexions du nœud *x*. Le prédicat *Validlfp* reconnaît la validité d'une liste *lfpx*.

On désigne aussi par g*en-top* la fonction qui sert pour générer tous les arcs (liens) d'un réseau. Elle prend en entrée un seul paramètre qu'on nomme *pms-top*. Ce dernier est une liste constituée de couples ayant la forme *(X, fp)* où *fp* est la fonction de connexion qui doit être appliquée à tous les nœuds contenus dans *X*. La fonction *gen-top* fait appel à une fonction génératrice des différentes sous-topologies désignée par *gen-top-1* (définition 4-3).

Définition 4-3. *Définition de la fonction gen-top-1*

$$gen\text{-}top\ (pms\text{-}top) = \bigwedge_{\substack{X \subseteq NodeSet \\ fp \in Listfp}} (List\ (gen\text{-}top\text{-}1\ (X, fp)))$$

(4-7)

3.2.3. *Critères de correction*

La principale contrainte à vérifier sur le graphe d'une topologie quelconque est qu'un sommet *v* produit par une fonction de connexion donnée *fp*, fait réellement partie de l'ensemble des noeuds *(NodeSet)*. Le théorème 4-5 exprime la correction de n'importe quelle liste de fonctions de connexions. Cette dernière n'est valable que si, pour toute connexion *cnx* produite à partir d'un nœud *x* de *NodeSet* et de sa liste de connexions *lfpx*, la seconde extrémité de *cnx* appartient à *NodeSet*. L'accès à la deuxième extrémité d'une connexion *cnx* est possible via la fonction *ext2*.

Théorème 4-5. *Validité des fonctions de connexions*

$$\forall x \in NodeSet,\ \forall lfpx, Validlfp\ (lfpx) \Rightarrow \forall cnx \in Gen\text{-}Cnx\ (x, lfpx), ext2\ (cnx) \in NodeSet$$

(4-8)

La contrainte 4-6 servira comme un lemme intermédiaire. En effet, pour ne pas avoir à vérifier pour chaque nœud du réseau, on préfère plutôt vérifier le théorème 4-6. Ce dernier est défini pour l'ensemble des nœuds formant les extrémités des connexions de la topologie. Le prédicat *ValidParams-top* sert pour vérifier la validité des paramètres de génération de la topologie. La fonction *ext2s-top* donne accès aux nœuds extrémités des connexions.

Théorème 4-6. *Validité des extrémités générées par les fonctions de connexion*

$$\forall pms,\ \forall pms\text{-}top,\ (ValidParamsp\ pms) \wedge (ValidParamsp\text{-}top\ pms\text{-}top)$$

$$\Rightarrow (ext2s\text{-}top\ (gen\text{-}top\ pms\text{-}top)) \subseteq (NodesetGenerator\ pms)$$

(4-9)

La validité de toute la topologie du point de vue représentation, est définie par le théorème 4-7. Ainsi, pour tout paramètre *pms-top* (reconnu par le prédicat *ValidParams-top*), toute connexion produite par la fonction g*en-top* doit être correcte, c'est-à-dire reconnue par le prédicat *ValidCnxp*. *ValidTop* est le prédicat associé à la validité de toute la topologie.

Théorème 4-7. *Validité de la topologie*

$\forall x \in NodeSet, \forall pms\text{-}top, (ValidParamsp\text{-}top\ pms\text{-}top)$

$\Rightarrow \forall cnx \in gen\text{-}top\ (pms\text{-}top),\ ValidCnxp\ (cnx)$

(4-10)

3.2.4. *Traduction dans la logique ACL2*

Dans cette partie, nous retraçons l'expression de tous les théorèmes décrivant la correction de la topologie générique. Des précisions sur la syntaxe ACL2 sont disponibles en annexe de ce manuscrit. La fonction *params-t* servira pour extraire les paramètres topologie à partir des paramètres des nœuds.

Expression du théorème 4-5. *Validité des fonctions de connexion (en locale)*

```
(local(defthm ext2-lfpx-in-nodeset
 (let* ((nodeset (NodeSetGenerator pms))
        (cnx (Gen-Cnx x lfpx))
        (ext2 (ext2 cnx)))
   (implies (and (ValidParamsp pms)(member-equal x nodes)
                 (Validlfp lfpx))
            (member-equal ext2 nodeset)))))
```

Expression du théorème 4-6. *Validité des noeuds extrémités des connexions*

```
(defthm gen-top-generates-nodes-in-nodeset
 (let* ((nodeset (NodesetGenerator pms))
        (pms-top (params-top pms))
        (top (gen-top pms-top))
        (ext2s (ext2s-top top)))
   (implies (and (ValidParamsp pms) (ValidParamsp-top pms-top)
                 (valid-ext2s ext2s))
            (subsetp-equal ext2s nodeset))))
```

Expression du théorème 4-7. *Validité de la topologie*

```
(defthm gen-top-generates-valid-top-1
 (let* ((pms-t (params-t pms)))
    (implies (ValidParamsp-top pms)
             (valid-top (gen-top pms-t)))))
```

Preuves : Dans ACL2, les preuves de trois théorèmes précédents sont assez instantanées. ACL2 ne pourra admettre le fichier générique que si on lui fournit en locale des définitions qui vérifient ces théorèmes. Il suffit donc de considérer les plus simples définitions possibles des fonctions locales. ACL2 s'assure ainsi qu'il existe au moins une instance de chaque fonction générique vérifiant les contraintes qui découlent des théorèmes ou des contraintes imposées sur ces fonctions.

3.3. *La composante routage générique étendue*

La fonction de routage défini dans GeNoC considère la seule information des nœuds *(NodeSet)* pour effectuer le routage. Ainsi, pour chaque missive *m* de l'ensemble *M*, la fonction *« Routing »* calcule toutes les routes possibles entre l'origine de la missive (Org_M) et sa destination ($Dest_M$).

3.3.1. *Principe*

La conception de la nouvelle composante topologie au niveau générique entraîne nécessairement des modifications dans la fonction *« Routing »*. Nous avons alors redéfini cette fonction de façon à ce qu'elle prenne en compte toute la topologie *(Top)* composée des nœuds et des connexions. En particulier, nous allons redéfinir la fonction ρ qui calcule les routes entre toute paire source-destination du réseau générique. La nouvelle route calculée *r* n'est plus composée de nœuds mais de connexions faisant intervenir des paires de nœuds.

3.3.2. *Spécification*

Nous avons redéfini le prédicat *ValidRoutep* de façon à y inclure explicitement la composante connexion (définition 4-4). On note alors par *r[i]* le $i^{ème}$ élément d'une route et par *l* la longueur de *r*. Le nouveau prédicat *Ext-ValidRoutep* exige que la longueur *(len)* de *l* soit supérieur ou égale à 1, que le premier élément *(First)* de la première connexion composant *r (r[0])* soit égale à l'origine de la missive et que la route *r* ne soit composée que de connexions faisant partie de la topologie *(r[i] ⊆ Top)*. La fonction étendue de routage est désignée par *Ext-Routing* (définition 4-5). Elle tient compte de la nouvelle fonction de calcul des routes ρ*-ext* (Elle *et al.a*, 2008).

Définition 4-4. *Définition du prédicat « Ext-ValidRoutep »*

$$Ext\text{-}ValidRoutep\ (r, m, Top) = \wedge \begin{cases} (First\ (r[0]) = Org_M\ (m) \\ r \subseteq Top \ \wedge \ (len(r) \geq 1) \end{cases} \quad \textbf{(4-11)}$$

Définition 4-5. *Définition de la fonction « Routing » étendu*

$$Ext\text{-}Routing\ (Top,M) = \bigwedge_{m \in M} (List\ (Id_M(m), Frm_M(m), \rho\text{-}ext(Org_M\ (m), Dest_M\ (m), Top)) \quad \textbf{(4-12)}$$

3.3.3. *Critères de correction*

GeNoC décrit trois contraintes sur la fonction de routage *(cf. les théorèmes 4-2, 4-3 et 4-4)*. Dans le cadre de notre extension, nous nous intéressons uniquement aux deux premières contraintes. Ainsi, le théorème 4-2, décrit en fonction du prédicat *ValidRoutep*, est redéfinit à travers le théorème 4-8 en fonction du nouveau prédicat *« Ext-ValidRoutep »*.

La deuxième contrainte sur la fonction *« Routing »* ne fera pas l'objet de grandes modifications. En réalité, il suffit de remplacer la composante nœuds *(NodeSet)* par la composante topologie *(Top)* pour avoir le théorème 4-9. On redéfinit en plus le prédicat de validité des missives M_{lstp} en fonction de la topologie *(Top)*.

Théorème 4-8. *Validité des routes produites par la fonction ρ-ext*

$$\forall M, M_{lstp}\ (M,Top)$$

$$\Rightarrow \forall m \in M,\ \forall r \in \rho\text{-}ext\ (Org_M\ (m), Dest_M\ (m), Top),\ Ext\text{-}ValidRoutep\ (r,m,Top) \quad \textbf{(4-13)}$$

Théorème 4-9. *Validité des voyages produits par la fonction « Routing »*

$$\forall M, M_{lstp}\ (M,Top) \Rightarrow V_{lstp}\ (Ext\text{-}Routing\ (M,Top)) \quad \textbf{(4-14)}$$

4. Conclusion

Nous avons exposé tout au long de ce chapitre notre conception d'une composante topologie générique et d'une composante routage étendue afin de les intégrer dans le modèle générique GeNoC. Au début de ce travail, nous avons cru que l'approche du modèle générique GeNoC pouvait être appliquée telle qu'elle est pour valider les MINs de la famille Delta. Cependant, après une étude exhaustive du modèle et une exploration des réseaux déjà validés à travers cette approche et des réseaux multi-étages, nous avons constaté qu'une extension au niveau générique du modèle est indispensable. Nous détaillerons dans le chapitre suivant notre modèle formel des Delta MINs, ainsi que sa validation à travers le modèle générique étendu.

Chapitre 5 : Vérification formelle des réseaux multi-étages de la famille Delta : Etude de cas

1. Introduction

Nous avons défini dans le chapitre précèdent l'extension au niveau générique du modèle GeNoC. Cette extension a été nécessaire pour que des réseaux sur puce ne faisant pas intervenir explicitement la notion de « connexions » dans leurs algorithmes de routage, puissent être validés en appliquant l'approche générique étendue. Ainsi, il nous est possible maintenant de valider formellement les communications dans des réseaux multi-étages dédiés aux systèmes multiprocesseurs sur puce (MPSoCs).

Nous exposons dans ce chapitre la spécification formelle développée pour décrire les MINs de la famille Delta. Au fur et à mesure de cette spécification formelle, nous dévoilerons les différents théorèmes développés pour la validation formelle.

2. Formalisation des réseaux Delta MINs dédiés aux MPSoCs

Dans cette partie, nous détaillons dans la logique du démonstrateur de théorèmes ACL2 les différentes étapes suivies pour la formalisation des réseaux multi-étages de la famille Delta dédiés aux MPSoCs. Nous formalisons dans ce contexte les deux composantes topologie et routage en effectuant une validation suivant le modèle GeNoC étendu. Cette formalisation est réalisée à partir des descriptions informelles disponibles dans la littérature sous forme de textes et schémas *(voir chapitre 3).*

Nous appuierons notre travail par les fonctions et théorèmes principaux définis dans la logique ACL2. Nous essayerons à chaque fois de simuler ces définitions en les commentant par des descriptions textuelles détaillées. Néanmoins, il se peut que la logique ACL2 semble un peu ambiguë ou que certaines fonctions dépendent d'autres fonctions qui ne sont pas présentées ici. Dans ce cas, une revue brève de la syntaxe ACL2 disponible en annexe peut sembler nécessaire.

2.1. *La logique ACL2 : des précisions*

Il est à noter que nous nous intéressons uniquement aux preuves vérifiées par ordinateur, c'est-à-dire par l'assistant de preuves d'ACL2. Ainsi, un théorème soumis au prouveur est soit

automatiquement admis, soit doit être enrichi par des lemmes intermédiaires. Cet enrichissement dénote les tactiques à suivre lors de la démonstration (application de théorème, réécriture, simplification…) pour faire comprendre et accepter à l'assistant toutes les étapes de raisonnement.

Vu que le démonstrateur ne formalise qu'un sous ensemble du langage Common Lisp, il est courant qu'une preuve soit très difficile à établir dans ACL2. C'est pour cette raison qu'il est souhaitable d'enrichir le monde logique du démonstrateur par les différentes bibliothèques disponibles sous forme de « books ». Deux bibliothèques que les experts ACL2 conseillent toujours d'inclure sont le « book » d'arithmétique *(arithmetic 3)* et le « book » manipulant les structures de données *(data-structures).* Il s'avère très bénéfique d'inclure dans ACL2 ces deux « books » surtout pour les novices. Toutefois, l'ajout de ces « books » peut aussi ralentir l'admission des théorèmes à démontrer. D'ailleurs, on dit toujours que *« plus grand le monde logique, plus lentement ACL2 fonctionnera »*. En effet, pour accomplir une démonstration, ACL2 doit mettre en pratique toutes les règles *(rules)* figurant dans sa base de données et qui lui semble applicables. Ainsi, plus la base de données est grande, plus l'espace de recherche sera grand et plus ACL2 ira moins vite dans ses démonstrations. Une méthode se basant sur un mécanisme d'activation *(enable)* et de désactivation *(disable)* locale des règles inutiles peut néanmoins accélérer considérablement le processus d'admission. Cette méthode sera illustrée dans ce qui suit.

```
(local
   (in-theory (disable elim-dieze-l assoc OUR-DIGIT-CHAR-P NONNEGATIVE-
INTEGER-QUOTIENT DEFAULT-+-2 DEFAULT-+-1 DEFAULT-*-2 DEFAULT-UNARY-MINUS
NFIX DEFAULT-<-2 DEFAULT-<-1 ASSOCIATIVITY-OF-+ ZP-OPEN)))
```

2.2. *La composante topologie d'un Delta MIN*

L'approche du modèle générique étendue détaillée dans le chapitre précédent, nécessite de spécifier et de valider dans un premier temps la contrainte sur l'ensemble des nœuds du réseau (théorème 4-1), ensuite de décrire et de vérifier formellement l'ensemble des connexions par les théorèmes 4-5, 4-6 et 4-7 (Elle *et al.a*, 2008).

2.2.1. *Formalisation de l'ensemble de nœuds*

Avant de présenter la formalisation de l'ensemble des nœuds d'un Delta MIN, nous commencerons par la spécification d'un nœud élémentaire du réseau.

2.2.1.1. Spécification d'un nœud

- *Principe :* un nœud du réseau est représenté par une paire de coordonnées *(X Y)*. La coordonnée *X* est décimale et représente l'étage de commutateurs auquel appartienne le noeud. La coordonnée *Y* est binaire et elle décrit la position du nœud à l'intérieur de cet étage.

Pour générer la coordonnée binaire *Y*, on a utilisé la fonction ACL2 *explode-nonnegative-integer*. Cette fonction permet de faire la conversion d'un nombre *n* non négatif dans une base *r* donnée. Les valeurs possibles de la base sont 2, 8, 10 et 16. On peut remarquer d'après l'exemple donné ci-dessous que cette conversion n'est pas aussi directe puisqu'elle ne donne pas exactement le nombre binaire (1 1 1) mais plutôt une liste de caractères associés à des « # ». Il a fallu alors plusieurs autres fonctions pour arriver à la représentation voulue. Il est à noter que nous avons opté pour une représentation binaire de la coordonnée *Y* du nœud pour faciliter par la suite l'application des fonctions de permutations lors de la génération des connexions.

Exemple : *explode-nonnegative-integer*

```
ACL2 !>(explode-nonnegative-integer 7 2 nil) > '(#1 #1 #1)
```

- *Spécification :* on définit la fonction *y-gen-node* qui fait appel à toute une série de fonctions élémentaires pour donner la représentation définitive de la coordonnée *Y* du nœud. Cette fonction prend en entrée *i*, la position décimale du nœud dans l'étage, et *d*, le nombre de bits sur lequel doit être représenté le nombre binaire en sortie. On définit aussi la fonction *gen-node* qui produit le format souhaité du nœud et ceci en faisant appel à la fonction secondaire *y-gen-node*. Finalement, on pose le prédicat *valid-node-dmin* qui reconnaît la validité d'un nœud. Ce prédicat a pour rôle de vérifier que la représentation d'un nœud est correcte c'est-à-dire a exactement la forme ((X) (Y)).

- *Expression dans ACL2 :* la traduction des fonctions spécifiées précédemment dans ACL2 est donnée à travers les définitions 5-1. Les résultats de simulation correspondants à ces fonctions sont donnés à la suite.

Définitions 5-1. *Spécification d'un nœud dans ACL2*

```
;;fct de génération de la coordonnée y
(defun y-gen-node (d i)
  (let* ((l (elim-dieze-l (explode-nonnegative-integer i 2 nil)))
         (nb-z-0 (- d (len l)))
         (l-0 (list-nb-z-0 nb-z-0)))
  (append l-0 l)))
```

```
;;fct de génération d'un noeud
(defun gen-node (d i ind)
  (list (list ind) (y-gen-node d i)))

;;prédicat valid-node-dmin
(defun valid-node-dmin (l)
   (and (consp l)(consp (cdr l)) (null (cddr l)) (consp (car l))
        (natp (caar l))(consp (cadr l)) (valid-01 (cadr l))))
```

Simulation des définitions 5-1. *Spécification d'un nœud dans ACL2*

```
ACL2 !>(y-gen-node 3 4) >> '(1 0 0)

ACL2 !>(gen-node 3 4 3) >> '((3) (1 0 0))

ACL2 !>(valid-node-dmin '((3) (1 0 0))) >> T

ACL2 !>(valid-node-dmin '((r) (1 0 0))) >> NIL
```

2.2.1.2. Spécification de l'ensemble des nœuds

- *Principe :* un réseau du type Delta MIN a la caractéristique d'avoir deux types de nœuds : des terminaux et des commutateurs. Pour profiter de la notion d'étages, nous avons opté pour une génération des nœuds par étages. Cependant, vu que le nombre de nœuds sur chaque étage n'est pas le même, nous ne nous pouvons pas utiliser les mêmes paramètres pour générer tous les nœuds. Ainsi, la génération de chaque type de nœuds se fera de façon indépendente, ensuite on pourra les fusionner dans une seule liste.

- *Spécification :* on définit la fonction *gen-nodes-dmin* qui génère tous les noeuds du réseau. Elle prend en paramètres *d*, le nombre total des étages de commutateurs dans le réseau, *r*, le degré des commutateurs, et *network*, le nom du réseau Delta. A partir de l'argument *d*, on est capable de déduire les informations pertinentes relatives au réseau c'est-à-dire le nombre total de nœuds terminaux qu'on a qualifié de *N* dans le chapitre 3 et le nombre de commutateurs par étage *(N/r=N/2)*. La fonction *gen-nodes-dmin* fait principalement appel à la fonction récursive *gen-nodes-inv1-call* pour la génération des nœuds d'un étage *N* donné.

- *Expression dans ACL2 : l*a définition de toutes ces fonctions dans ACL2 est donnée dans les définitions 5-2. La figure 27 donne un aperçu de la façon dont les nœuds sont représentés dans un réseau Oméga (8, 2).

Définitions 5-2. *Spécification des nœuds d'un Delta MIN dans ACL2*

```
;;fct de génération secondaire de tous les nœuds d'un étage
(defun gen-nodes-inv1-call (d Nb ind)
   (declare (xargs :guard (and (natp d) (natp Nb)(natp ind))
                   :verify-guards nil))
```

```
  (if (zp Nb) nil
      (cons (gen-node d (1- Nb) ind)
            (gen-nodes-inv1-call d (1- Nb) ind))))
;;fct d'appel
 (defun gen-nodes-dmin-pms-1 (d r network)
   (declare (ignore r network))
     (let ((S (gen-nodes-dmin-s d))
           (Sw (gen-nodes-dmin-sw-1 (1- d) d))
           (D (gen-nodes-dmin-d d)))
     (append S (append Sw D))))

;;fct de génération de tous les noeuds
(defun gen-nodes-dmin (pms)
   (gen-nodes-dmin-pms-1 (car pms)(cadr pms)(caddr pms)))
```

Simulation des définitions 5-2. *Spécification des nœuds d'un Delta MIN dans ACL2*

```
ACL2 !>(gen-nodes-inv1-call 2 4 2)
       (((2)(0 0)) ((2)(0 1)) ((2)(1 0)) ((2)(1 1)))

ACL2 !>(gen-nodes-dmin '(3 2 omega))
       (((4)(0 0 0)) ((4)(0 0 1)) ((4)(0 1 0)) ((4)(0 1 1))
        ((4)(1 0 0)) ((4)(1 0 1)) ((4)(1 1 0)) ((4)(1 1 1))
        ((3)(0 0)) ((3)(0 1)) ((3)(1 0)) ((3)(1 1))
        ((2)(0 0)) ((2)(0 1)) ((2)(1 0)) ((2)(1 1))
        ((1)(0 0)) ((1)(0 1)) ((1)(1 0)) ((1)(1 1))
        ((0)(0 0 0)) ((0)(0 0 1)) ((0)(0 1 0)) ((0)(0 1 1))
        ((0)(1 0 0)) ((0)(1 0 1)) ((0)(1 1 0)) ((0)(1 1 1)))
```

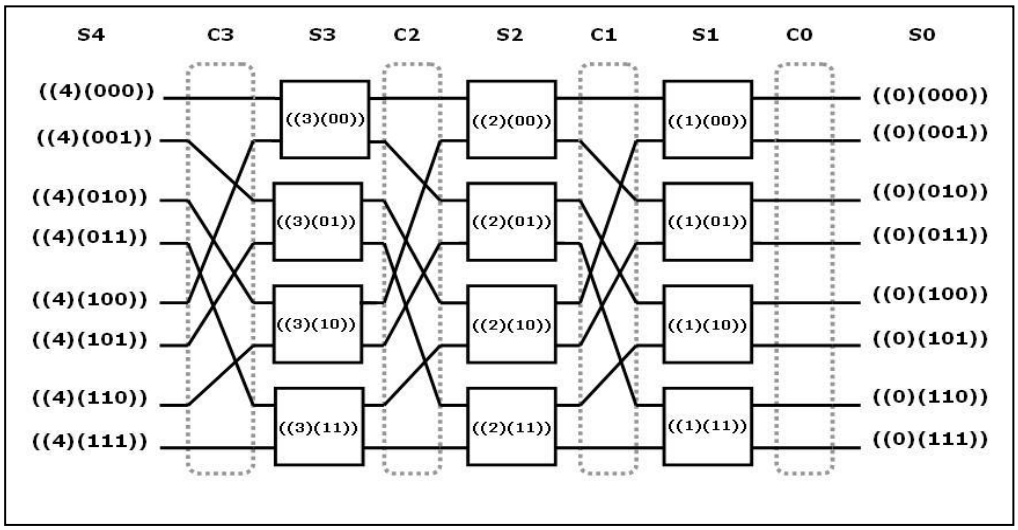


Figure 27. *Spécification formelle des noeuds*

2.2.1.3. Vérification du théorème 4-1

La génération de tous les nœuds est contrainte par le théorème 5-1 qui n'est qu'une instance du théorème générique 4-1. Pour la validité des paramètres de génération de tous les nœuds, on pose le prédicat *ValidParamsp-dmin*. On définit également un autre prédicat appelé *valid-nodes-dmin* pour vérifier la validité de l'ensemble de nœuds. Ce prédicat fait bien sûr

appel au prédicat *valid-node-dmin* (définitions 5-3). Le tableau 10 résume les fonctions utilisées dans cette partie.

Tableau 10. *Résumé des fonctions de spécification des noeuds*

Fonction	**Argument (s)**	**Rôle**
y-gen-node	*d i*	génère la coordonnée *y* en binaire d'un nœud
gen-node	*d i ind*	génère le nœud à la position *i* de l'étage *ind*
valid-node-dmin	*l*	teste la validité d'un nœud représenté par une liste *l*
ValidParamsp-dmin	*pms*	teste la validité des paramètres de génération *pms*
gen-nodes-inv1-call	*d Nb ind*	génère les *N* nœuds de l'étage *ind*
gen-nodes-dmin	*pms*	génère tous les nœuds du réseau en fonction de *pms*
valid-nodes-dmin	*X*	teste la validité de tous les nœuds *X*

Définitions 5-3. *Prédicats de validité des paramètres et des nœuds*

```
;;predicat qui valide tous les noeuds
(defun valid-nodes-dmin (X)
  (if (endp X) t
      (and (valid-node-dmin (car X))
           (valid-nodes-dmin (cdr X)))))

;;prédicat de validité des paramètres
(defun ValidParamsp-dmin (pms)
  (and (consp pms)(consp (cdr pms))(consp (cddr pms)) (null (cdddr pms))
       (natp (car pms))(>= (car pms) 1) (natp (cadr pms))
       (< 0 (cadr pms))(dmin-networkp (caddr pms))))
```

Théorème 5-1. *Validité de la définition des nœuds d'un Delta MIN*

$$\forall pms, ValidParamsp\text{-}dmin\ (pms) \Rightarrow \forall x \in gen\text{-}nodes\text{-}dmin\ (pms),\ valid\text{-}node\text{-}dmin\ (x) \quad \textbf{(5-1)}$$

Dans ACL2, ceci donne :

```
(defthm gen-nodes-dmin-correct
 (implies (ValideParams-dmin pms)
          (valid-nodes-dmin (gen-nodes-dmin pms))))
```

Preuve : La preuve du théorème 5-1 fait appel à plusieurs lemmes intermédiaires. A vrai dire, le fait d'avoir opter pour une représentation binaire de la coordonnée *Y* du nœud a rendu compliquée la procédure de génération des nœuds. Les deux lemmes intermédiaires de base qu'on a dû démontrer dans ce contexte, concernent le type de données retourné par la fonction *explode-nonnegative-integer* (théorèmes 5-2). Sans ces deux théorèmes, il aurait été en fait impossible de démontrer la validité d'un nœud (prédicat *valid-node-dmin*) et par la suite la validité de l'ensemble des nœuds. Le théorème 5-3 a été aussi indispensable pour la démonstration du théorème 5-1. Il exprime la validité d'un nœud via la fonction *gen-node*.

Théorèmes 5-2. *Les deux théorèmes concernant explode-nonnegative-integer*

```
(defthm character-listp-explode-n-generalized
  (implies (character-listp ans)
           (character-listp (explode-nonnegative-integer n 2 ans))))

 (defthm character-listp-explode-n
           (character-listp (explode-nonnegative-integer n 2 nil)))
```

Preuve : Le premier théorème figurant dans la liste des théorèmes 5-2 est une généralisation du deuxième théorème. Ainsi, une fois ce premier théorème admis dans ACL2, le second sera automatiquement démontré par une simple déduction *(rewrite rule).*

Théorème 5-3. *Validité d'un noeud*

```
(defthm valid-node-dmin-gen-node
  (implies (and (natp d)(natp i)(natp ind))
           (valid-node-dmin (gen-node d i ind)))
  :hints (("GOAL" :in-theory (disable explode-nonnegative-integer))))
```

Preuve : Dans la preuve du théorème 5-3, la seule astuce consiste en la désactivation *(disable)* de la fonction *explode-nonnegative-integer*. De cette façon, le démonstrateur ne « bouclera » pas. En effet, l'utilisation de la fonction de conversion peut amener à des démonstrations un peu étranges. On doit à chaque fois *généraliser* le troisième argument *(ans)*. Autrement, le démonstrateur ne trouvera pas l'argument sur lequel il doit effectuer son induction.

2.2.2. *Les connexions*

2.2.2.1. Spécification d'une connexion

- *Principe :* dans un réseau Delta MIN, on représente une connexion par une liste ayant la forme *((X px) (Y py))*. *x* représente l'origine de la connexion, *px* le port de *x* inclut dans cette connexion, *y* la deuxième extrémité et *py* le port de *y*. Par exemple, la connexion représentée par *(((3) (0 1)) O0) (((2) (1 0)) I0)* dénote que le port *« O0 »* du commutateur *((3) (0 1))* est connecté au port « *I0* » du commutateur *((2) (1 0))*.

- *Spécification :* on définit la fonction *gen-one-cnx* qui génère la connexion ayant pour origine le port *i* du nœud *ext1*. Cette fonction prend en argument le nœud *ext1*, le port *i* de connexion et la fonction de permutation *fp* à appliquer (définitions 5-5). Suivant le type de nœud *(cond équivalente à case)*, cette fonction fait appel à deux autres fonctions *gen-ext2* et *gen-ext2-p* dont les définitions sont aussi données ci-dessous. La première sert pour générer le nœud extrémité de la connexion sans le port de connexion et la seconde sert à générer ce port.

Ces deux fonctions font appel aux deux fonctions *next-y-sw* et *next-p-sw* (définitions 5-4) mettant en évidence la façon de « commuter » entre les différentes fonctions de permutation. On peut remarquer que l'application des fonctions de permutations n'est pas intégrée dans la fonction *gen-one-cnx*. Il est alors possible d'ajouter autant de permutations qu'on veut ; il suffit pour cela d'implémenter le code ACL2 correspondant (exemple des fonctions *sigmak-app-1* et *sigmak-app-2* pour la permutation *sigmak*). Le tableau 11 donne un résumé des différentes fonctions utilisées pour la spécification d'une connexion.

- *Expression dans ACL2 :* L'expression dans ACL2 de toutes les fonctions relatives à une connexion, est donnée aux définitions 5-4 et 5-5.

Définitions 5-4. *Spécifications des fonctions de connexions*

```
;;fct applique sigmak-fp retourne la liste  (y-ext2) après permutation
(defun sigmak-app-1 (y-s i)
  (let* ((l (Gen-input y-s i))
         (l-sig (sigmak-fp l)))
  (rem-last l-sig)))

;;fct applique sigmak-fp retourne une liste contenant (numport)
(defun sigmak-app-2 (y-s i)
  (let* ((l (Gen-input y-s i))
         (l-sig (sigmak-fp l)))
  (last l-sig)))

;;fct génère le prochain noeud suivant la permutation fp à appliquer
(defun next-y-sw (y-sw i fp)
   (cond
    ((equal fp 'sigmak) (sigmak-app-1 y-sw i))
    ((equal fp 'I) (I-app-1 y-sw i))))

;;fct génère le prochain port suivant la permutation fp à appliquer
(defun next-p-sw (y-sw i fp)
   (cond
    ((equal fp 'sigmak) (sigmak-app-2 y-sw i))
    ((equal fp 'I) (I-app-2 y-sw i))))
```

Définitions 5-5. *Spécification d'une connexion dans la logique ACL2*

```
;; fct génère une connexion élémentaire d'un noeud donné
(defun gen-one-cnx (ext1 i fp)
   (let* ((l-ext2 (gen-ext2 ext1 i fp))
          (x-ext2 (car (x-node l-ext2)));;=(caar l-ext2)
          (p-ext2 (car (gen-ext2-p ext1 i fp)));;numport-ext2 sans les ()
          (p-ext1 (car i));;port ext1=i=0, 1

(cond

  ;;ext2=d-node ==> ajouter port local L
  ((equal x-ext2 0)
   (list (gen-ext1-cnx ext1 'O p-ext1) (gen-ext2-cnx l-ext2 'L 'nil)))
```

```
  ;;ext1=s-node ==> ajouter port local L
  ((equal i nil)
   (list (gen-ext1-cnx ext1 'L 'nil) (gen-ext2-cnx l-ext2 'I p-ext2)))

  ;;ext1=switch et ext2=switch ==> ajouter les ports du switch I et O
  (t
  (list (gen-ext1-cnx ext1 'O p-ext1) (gen-ext2-cnx l-ext2 'I p-ext2))))))
```

;;fct gen-ext2 génère le noeud ext2 extrémité de la connexion

```
(defun gen-ext2 (ext1 i fp)
  (let* ((y-ext1 (y-node ext1))
         (y-ext2 (next-y-sw y-ext1 i fp))
         (p-ext2 (next-p-sw y-ext1 i fp)) ;; génère le port ext2
         (ext2-2 (append y-ext2 p-ext2 ))
         (x-ext2 (- (car (x-node ext1)) 1)))
   (list (list x-ext2) (if (equal x-ext2 0)
                           ext2-2
                           y-ext2))))
```

;;fct gen-ext2-p génère le port de cnx sous forme '(0), '(1)

```
(defun gen-ext2-p (ext1 i fp)
   (let* ((y-ext1 (y-node ext1)))
     (next-p-sw y-ext1 i fp)))
```

Simulation des définitions 5-4 et 5-5. *Spécification d'une connexion*

```
ACL2 !>(sigmak-app-1 '(0 1) '(1)) >> '(1 1)

ACL2 !>(sigmak-app-2 '(0 1) '(1)) >> '(0)

ACL2 !>(next-y-sw '(0 1) '(1) 'sigmak) >> '(1 1)

ACL2 !>( next-p-sw '(0 1) '(1) 'sigmak) >> '(0)

ACL2 !>(gen-ext2 '((2) (0 1)) '(1) 'sigmak) >> '(1 1)

ACL2 !>(gen-ext2-p '((2) (0 1)) '(1) 'sigmak) >> '(0)

ACL2 !>(gen-one-cnx '((2) (0 1)) '(1) 'sigmak)
     >> '((((2) (0 1)) O1) (((1) (1 1)) I0))
```

Tableau 11. *Résumé des fonctions de spécification d'une connexion*

Fonction	**Argument (s)**	**Rôle**
sigmak-app-1	*y-s i*	applique la perm. sigmak-fp et retourne la coordonnée y
sigmak-app-2	*y-s i*	applique la perm. sigmak-fp et retourne le prochain port
next-y-sw	*y-sw i fp*	génère la coordonnée y suivant la perm. fp à appliquer
next-p-sw	*y-sw i fp*	génère le prochain port suivant la perm. fp à appliquer
gen-ext2	*ext1 i fp*	génère la coordonnée y après application de la perm. fp
gen-ext2-p	*ext1 i fp*	génère le prochain port après application de la perm. fp
gen-one-cnx	*ext1 i fp*	génère toute la connexion résultante de l'app. de fp au port i de ext1

2.2.2.2. Spécification des connexions

- *Principe :* pour valider les connexions en suivant l'approche de la topologie générique présentée au chapitre 4, il faut commencer par identifier la liste des fonctions de connexions à appliquer sur un graphe directe de la topologie. Dans le cas des MINs Delta, en orientant le graphe de la topologie de gauche à droite, cette liste sera toujours constituée de trois fonctions à appliquer respectivement au premier étage de connexions *($C_{(d+1)}$)*, aux étages du milieu *(C_i avec $(0 < i < d+1)$)* et au dernier étage C_0. Nous effectuerons alors une génération par « étage de connexions ».

- *Spécification :* la fonction principale de génération de toutes les connexions d'un MIN Delta est *gen-top-dmin*. C'est une fonction qui fait appel à trois autres fonctions qu'on désigne par *gen-top-dmin-src_sw*, *gen-top-dmin-sw_sw* et *gen-top-dmin-sw_dest*. Le rôle de chacune de ces fonctions est de générer les connexions d'un type d'étage donné *(sous-topologie)*. Ainsi, *gen-top-dmin-src_sw* permet la génération du premier étage de connexions entre les nœuds sources et le premier étage de switches. Ensuite, la fonction *gen-top-dmin-sw_sw* produit les étages de connexions du « milieu » du réseau c'est-à-dire entre étages de switches. Finalement, la fonction *gen-top-dmin-sw_dest* permet la génération du dernier étage de connexions. Chacune de ces fonctions prendra en paramètres l'ensemble des nœuds de l'étage origine des connexions, la permutation *fp* à appliquer et un dernier argument dénoté *pms* relatif aux caractéristiques du réseau.

La fonction *gen-top-dmin* prend en entrée un seul paramètre dénoté *pms-t*. Ce dernier est en effet produit par la fonction *params-top-t* à partir des paramètres *pms* de génération des noeuds. C'est notamment à partir du second paramètre de *pms*, c'est à dire le nom du réseau *network* qu'on pourra sélectionner les permutations adéquates au réseau en question. Enfin, on pose le prédicat *valid-top-dmin* qui reconnaît la validité d'une topologie. Ce prédicat est une fonction récursive définie en fonction du prédicat *validp-cnx* qui admet une connexion valide. Ce dernier prédicat vérifie que la connexion produite possède exactement la représentation souhaitée. Le tableau 12 donne un résumé des fonctions utilisées dans la spécification de la topologie.

- *Expression dans ACL2 :* l'illustration de toutes les fonctions spécifiées précédemment dans ACL2 est donnée aux définitions 5-6 et 5-7. Les trois premières fonctions des définitions 5-6 ; à savoir *gen-top-dmin-src_sw*, *gen-top-dmin-sw_sw* et *gen-top-dmin-sw_dest*, sont des

fonctions récursives définies en fonction de la fonction de génération d'une connexion élémentaire *gen-one-cnx*.

Définitions 5-6. *Spécifications de la topologie d'un Delta MIN*

```
;;fct de génération du premier étage de connexions (sources-switches)
(defun gen-top-dmin-src_sw (S pms-s fp-s)
 (if (endp S) nil
     (cons (gen-one-cnx (car S) pms-s fp-s)
           (gen-top-dmin-src_sw (cdr S) pms-s fp-s))))


;;fct de génération des connexions des étages du milieu (switches-switches)
(defun gen-top-dmin-sw_sw (Sw pms-sw fp-sw)
 (if (endp Sw) nil
     (append (gen-cnx-node (car Sw) pms-sw fp-sw)
             (gen-top-dmin-sw_sw (cdr Sw) pms-sw fp-sw))))


;;fct de génération des connexions du dernier étage de connexions (switches-destinations)
(defun gen-top-dmin-sw_dest (D pms-d fp-d)
 (if (endp D)   nil
     (append (gen-cnx-node (car D) pms-d fp-d)
             (gen-top-dmin-sw_dest (cdr D) pms-d fp-d))))


;;fct gen-top-dmin pour la génération de toute la topologie par étages
(defun gen-top-dmin (pms-t)
 (let* ((x1 (car pms-t))(x2 (cadr pms-t))(x3 (caddr pms-t))
        (S (car x1))(pms-s (cadr x1))(fp-s (caddr x1))
        (Sw (car x2))(pms-sw (cadr x2))(fp-sw (caddr x2))
        (D (car x3))(pms-d (cadr x3))(fp-d (caddr x3))

         ;;(S pms-s fps)
        (top-S (gen-top-dmin-src_sw S pms-s fp-s))

         ;; (Sw pms-sw fpsw)
        (top-Sw (gen-top-dmin-sw_sw Sw pms-sw fp-sw))

         ;;(D pms-d fpd)
        (top-D (gen-top-dmin-sw_dest D pms-d fp-d)))

   (append top-S (append top-Sw top-D))))
```

Définitions 5-7. *Prédicats de validité d'une connexion et d'une topologie*

```
;;fct qui reconnaît une connexion valide
(defun validp-cnx (c)
  (and (consp c) (consp (car c)) (consp (cadr c))
       (consp (cdr c))(null (cddr c))))

;;fct valid-top-dmin
(defun valid-top-dmin (top)
   (if (endp top)
        t
       (and (validp-cnx (car top))
            (valid-top-dmin (cdr top)))))
```

Simulation des définitions 5-6 et 5-7. *Spécification des connexions*

```
ACL2 !> (gen-top-dmin (params-top-t '(3 2 omega))))

 (((((4) (0 0 0)) L) (((3) (0 0)) I0))
  ((((4) (0 0 1)) L) (((3) (0 1)) I0))
  ((((4) (0 1 0)) L) (((3) (1 0)) I0))
  ((((4) (0 1 1)) L) (((3) (1 1)) I0))
  ((((4) (1 0 0)) L) (((3) (0 0)) I1))
  ((((4) (1 0 1)) L) (((3) (0 1)) I1))
  ((((4) (1 1 0)) L) (((3) (1 0)) I1))
  ((((4) (1 1 1)) L) (((3) (1 1)) I1))
  ((((3) (0 0)) O0) (((2) (0 0)) I0))
  ((((3) (0 0)) O1) (((2) (0 1)) I0))
  ((((3) (0 1)) O0) (((2) (1 0)) I0))
  ((((3) (0 1)) O1) (((2) (1 1)) I0))
  ((((3) (1 0)) O0) (((2) (0 0)) I1))
  ((((3) (1 0)) O1) (((2) (0 1)) I1))
  ((((3) (1 1)) O0) (((2) (1 0)) I1))
  ((((3) (1 1)) O1) (((2) (1 1)) I1))
  ((((2) (0 0)) O0) (((1) (0 0)) I0))
  ((((2) (0 0)) O1) (((1) (0 1)) I0))
  ((((2) (0 1)) O0) (((1) (1 0)) I0))
  ((((2) (0 1)) O1) (((1) (1 1)) I0))
  ((((2) (1 0)) O0) (((1) (0 0)) I1))
  ((((2) (1 0)) O1) (((1) (0 1)) I1))
  ((((2) (1 1)) O0) (((1) (1 0)) I1))
  ((((2) (1 1)) O1) (((1) (1 1)) I1))
  ((((1) (0 0)) O0) (((0) (0 0 0)) L))
  ((((1) (0 0)) O1) (((0) (0 0 1)) L))
  ((((1) (0 1)) O0) (((0) (0 1 0)) L))
  ((((1) (0 1)) O1) (((0) (0 1 1)) L))
  ((((1) (1 0)) O0) (((0) (1 0 0)) L))
  ((((1) (1 0)) O1) (((0) (1 0 1)) L))
  ((((1) (1 1)) O0) (((0) (1 1 0)) L))
  ((((1) (1 1)) O1) (((0) (1 1 1)) L)))

ACL2 !> (valid-top-dmin (gen-top-dmin (params-top-t '(3 2 omega)))) >> T
```

Tableau 12. *Résumé des fonctions de spécification de la topologie*

Fonction	**Argument (s)**	**Rôle**
gen-top-dmin-s_sw	*S pms-s fp-s*	génère le premier étage de connexion *(C_{d+1})*
gen-top-dmin-sw_sw	*Sw pms-sw fp-sw*	génère les étages de connexion du milieu *(C_i, $0 < i < d+1$)*
gen-top-dmin-sw_dest	*D pms-d fp-d*	génère le dernier étage de connexion *(C_0)*
gen-top-dmin	*pms-t*	génère toute la topologie
validp-cnx	*C*	teste la validité d'une connexion *c*
valid-top-dmin	*top*	teste la validité de toute la topologie *top*
params-top-t	*pms*	permet de construire la liste des *pms-t* à partir de *pms*

2.2.2.3. Vérification des théorèmes 4-5 et 4-6

Le théorème 5-4 est une instance valide du théorème 4-5. A travers ce théorème, nous démontrons que peu importe la fonction la permutation appliquée pour avoir une connexion,

l'extrémité de cette connexion sera dans l'ensemble des nœuds du réseau. La prise en compte de toutes les fonctions de permutations est possible via le prédicat *Validfp-dmin* (définition 5-8). Ce premier théorème servira comme lemme intermédiaire pour pouvoir démontrer le théorème 5-5 suivant. Ce dernier établit le fait que les extrémités de toutes les connexions forment un sous-ensemble de l'ensemble des nœuds. Le théorème 5-5 n'est qu'une instance du théorème 4-6.

Définition 5-8. *Définition du prédicat Validfp-dmin*

```
(defun Validfp-dmin (fp)
  (or (equal fp 'sigmak)(equal fp 'I)(equal fp ‘tetak))
```

Théorème 5-4. *Validité d'une extrémité de connexion*

```
(defthm ext2-gen-one-cnx-in-nodes
  (let* ((nodes (gen-nodes-dmin pms))
         (c (gen-one-cnx x i fp))
         (ext2 (y-node c)))
  (implies (and (ValidParamspD pms) (Validfp-dmin fp)
                (member-equal x nodes)(valid-node-dmin ext2))
           (member-equal ext2 nodes)))
:hints (("GOAL" :in-theory (disable gen-nodes-inv1))))
```

Preuve : La preuve de ce théorème est directe mais assez longue car le démonstrateur doit faire appel à toutes les définitions des fonctions secondaires, ce qui fait un nombre de cas extrêmement grand.

Théorème 5-5. *Validité des fonctions de connexion*

```
(defthm gen-top-dmin-generates-nodes-in-nodeset
   (let* ((pms-t (params-top-t pms))
          (nodes (gen-nodes-dmin pms))
          (top-dmin (gen-top-dmin pms-t))
          (ext2s (ext2s-top top-dmin)))
   (implies (ValidParamspD pms)
            (subsetp-equal ext2s nodes)))
:hints (("GOAL" :induct  (ext2s-top top-dmin))))
```

Preuve : Le théorème 5-5 a pu être démontré grâce à trois principaux lemmes intermédiaires. Chacun de ces lemmes est utilisé pour démontrer que les extrémités d'une sous-topologie forment un ensemble inclut dans l'ensemble de nœuds *(nodes)*. Par exemple, le théorème 5-6 est établi dans le but de vérifier que toutes les extrémités de la sous-topologie générée par la fonction *gen-top-dmin-src_sw*, sont effectivement dans l'ensemble des nœuds du réseau. Il en est de même pour les deux autres fonctions *gen-top-dmin-sw_sw* et *gen-top-dmin-sw_dest*, des théorèmes similaires ont dû être démontrés.

Théorème 5-6. *Validité des extrémités générées par la fonction gen-top-dmin-src_sw*

```
(defthm gen-top-dmin-src_sw-generates-nodes-in-nodeset
    (let* ((pms-t (params-top-t pms))
           (x1 (car pms-t))
           (S (car x1))
           (pms-s (cadr x1))
           (fp-s (caddr x1))
           (nodes (gen-nodes-dmin pms))
           (top-dmin-s_sw (gen-top-dmin-src_sw S pms-s fp-s))
           (ext2s (ext2s-top top-dmin-s_sw)))
   (implies (and (ValidParamspD pms)(valid-params-t pms-t)
                 (valid-ext2s-dmin ext2s))
   (subsetp-equal ext2s nodes)))
:hints (("GOAL" :induct (ext2s-top top-dmin-s))))
```

Preuve : Le théorème 5-6 n'est automatiquement admis dans ACL2 que si on lui fournit le conseil *(hint)* d'effectuer une induction sur la structure de la fonction *ext2s-top* qui est une fonction définie récursivement.

2.2.2.4. Vérification du théorème 4-7

La concrétisation du théorème 4-7, qui concerne la validité de la représentation de toute la topologie générée, est illustrée à travers le théorème 5-7.

Théorème 5-7. *Validité de la topologie d'un Delta MIN*

```
;;thm de validité de toute la topologie
(defthm valid-gen-top-dmin
   (let* ((pms-t (params-top-t pms))
          (top (gen-top-dmin pms-t)))
      (implies (and (ValidParamspD pms)(valid-params-t pms-t))
                    (valid-top-dmin top)))
 :hints (("GOAL" :in-theory (disable Validfp-dmin GEN-NODES-DMIN-S-pms
                          gen-nodes-dmin-sw-1-pms GEN-NODES-DMIN-d-pms ))))
```

Preuve : L'admission du théorème 5-7 dans ACL2 n'est pas directe. En effet, de façon similaire à la stratégie utilisée dans la démonstration du théorème 5-5, il a fallu aussi raisonner par étages. Ainsi, on a décomposé la contrainte à démontrer sur toute la topologie en trois contraintes élémentaires à vérifier sur chacune des sous-topologies générées respectivement par les fonctions *gen-nodes-dmin-s*, *gen-nodes-dmin-sw* et *gen-nodes-dmin-d*. L'un de ces lemmes intermédiaires est illustré à travers le théorème 5-8.

Théorème 5-8. *Validité du premier étage de connexion d'un Delta MIN*

```
;;thm de validté du premier étage de connexion
(defthm valid-gen-top-dmin-src_sw
  (let* ((S (gen-nodes-dmin-s-pms  pms))
         (top-s_sw (gen-top-dmin-src_sw S pms-s fp)))
```

```
(implies (and (valid-pms-SubS pms)(valid-S S)
              (valid-SubS pms-s)(Validfp-dmin fp))
         (valid-top-dmin top-s_sw)))
:hints (("GOAL" :induct (gen-top-dmin-src_sw S pms-s fp))))
```

Preuve : En conseillant le prouveur d'ACL2 d'essayer de démontrer le théorème en procédant par induction sur la structure de la fonction *gen-top-dmin-src_sw*, la preuve du théorème 5-8 est alors automatique.

2.3. *La composante routage*

2.3.1. *Spécification de la fonction de routage*

Dans ce qui suit, nous spécifions l'algorithme de routage utilisé dans les réseaux d'interconnexions multi-étages, ainsi que sa traduction dans la logique ACL2 (Elle *et al.a*, 2008). Enfin, nous développons quelques théorèmes dans le but de vérifier la conformité des fonctions définies.

- *Principe :* l'algorithme de routage des Delta MINs, dénoté « self routing », ne dépend que de l'adresse destination. Cette dernière est appelée aussi séquence de contrôle (control sequence). Puisque les Delta MINs qu'on modélise utilisent des crossbars 2x2, l'algorithme de calcul d'une route a le principe suivant : si le $i^{ème}$ *bit* courant de la séquence de contrôle est à 1 alors le message sera commuté à travers le port haut du commutateur, sinon par le port bas.

Dans ce contexte, on doit aussi définir une fonction qui calcule toutes les routes valables des missives ou messages d'un ensemble donné M. Cette nouvelle fonction fera appel à la fonction de routage élémentaire définie précédemment.

- *Spécification* : On désigne par *routing-dmin*, la fonction principale de routage. Cette dernière est une instance de la fonction *« Ext-Routing »* décrite au niveau générique (définition 4-4). Elle prend en arguments l'ensemble *M* des messages à router et la topologie *Top* du réseau en question (définition 5-10). Cette fonction doit nécessairement faire appel aux accesseurs Id_M, Org_M, $Dest_M$ et Frm_M pour accéder aux différents éléments d'un message et mettre ainsi le message résultat (appelé voyage dans la notation GeNoC) dans le format souhaité *(id frm routes)*. La même fonction *routing-dmin* doit en plus faire appel à une fonction récursive appelée *compute-routes-dmin* pour calculer une route élémentaire entre l'origine et la destination du message à router.

En réalité, la fonction *compute-routes-dmin* n'est autre qu'une instance de la fonction générique *ρ-ext*. Ainsi, pour chaque missive ou message *m* de l'ensemble *M*, cette fonction teste le $i^{ème}$ *bit* courant de la séquence de contrôle et appelle la fonction secondaire désignée par *rech-top* pour chercher la connexion correspondante au port *i* dans la topologie. Sachant que chacun des ports au niveau d'un nœud possède une connexion unique avec l'un des nœuds de l'étage suivant, la valeur retournée par la fonction *rech_top* est alors unique.

- *Traduction dans ACL2 :* le pseudo code, ainsi que le code ACL2 relatif au mécanisme de routage élémentaire sont illustrés dans la définition 5-9. La fonction *rech_top* utilisée au niveau du pseudo code de la fonction *compute-routes-dmin*, est substituée par deux fonctions ACL2 : *switch* et *assoc-equal*. La première sert à commuter le message au niveau d'un switch en fonction du bit courant lu *(bit_rtg)*, alors que la seconde permet de retourner la connexion correspondante à partir de la topologie.

Définition 5-9. *Spécification de la fonction de calcul d'une route élémentaire dans ACL2*

- *Pseudo code de la fonction de calcul d'une route :*

```
compute-routes-dmin (from dest top)
  if (from = dest)   /* destination reached */
     then take the local port of the destination
     else
        if (dest [i] = 0) /* ith bit equals 0 */
           then take the upper output of the switch at Si
           from = rech_top (from top 0)
           compute-routes-dmin (from dest top)
        else /* ith bit equals 1 */
           take the lower output of the switch at Si
          from = rech_top (from top 1)
          compute-routes-dmin (from dest top)
```

- *Traduction dans la logique ACL2 :*

```
(defun compute-routes-dmin (from to top)
 (if (endp to)
      nil
 (let* ((bit_rtg (car to))
        (from-a (switch from bit_rtg))
        (cnx (assoc-equal from-a top))
        (next-node (cadr cnx)))
   (cons cnx (compute-routes-dmin next-node (cdr to) top)))))
```

Définition 5-10. *Spécification de la fonction de routage*

```
(defun routing-dmin (Missives topology)
  (if (endp Missives)
      nil
  (let* ((miss (car Missives))
         (from (OrgM miss))
         (to (DestM miss))
         (id (IdM miss))
         (frm (FrmM miss))
         (cdrto (cadr to)))
  (cons (list id frm
              (compute-routes-dmin from (append '(s) cdrto) topology))
              (routing-dmin (cdr Missives) topology)))))
```

Simulation de la définition 5-10. *Spécification de la fonction de routage*

Nous simulons l'exécution de la fonction de routage donnée à la définition 5-10 pour la liste de missives du tableau 13 et ceci dans un réseau Oméga de taille 8 utilisant des crossbars 2x2. La figure 28 présente les chemins empruntés par les deux missives sur le schéma de l'Oméga (8, 2). Les résultats de cette simulation sont illustrés au tableau 14.

Tableau 13. *Une liste de missives*

id	*origine*	*contenu*	*destination*
1	((4) (001))	frm1	((0) (100))
2	((4) (110))	frm2	((0) (111))

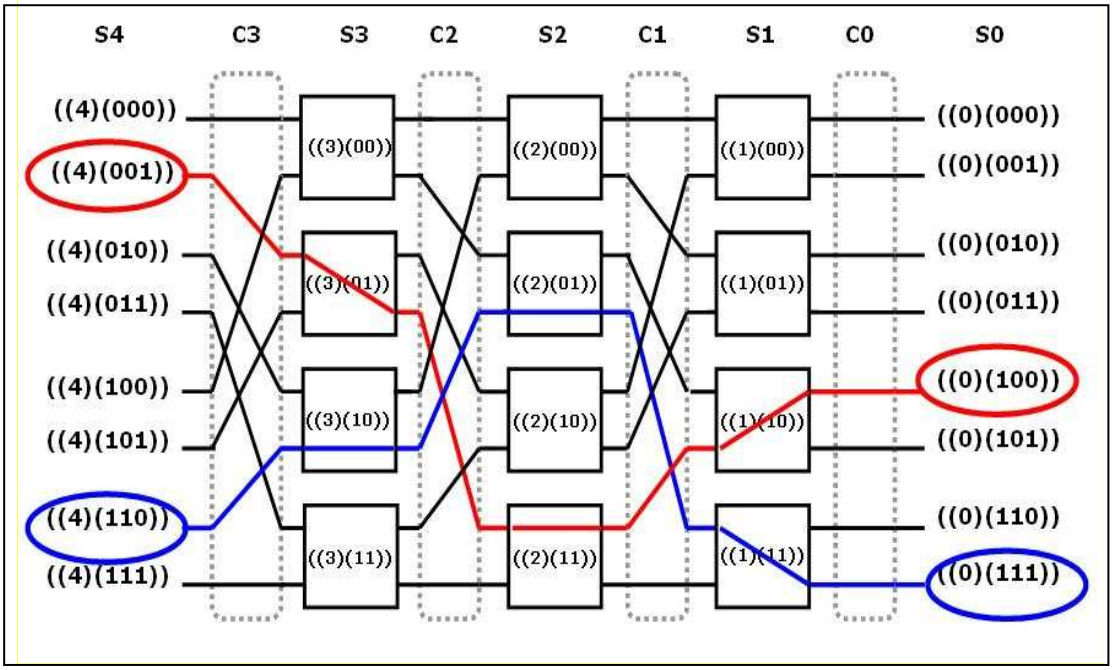


Figure 28. *Illustration des résultats de simulation*

Tableau 14. *Les résultats de simulation de la liste de missives*

id / *étage*	1	2
C3	((((4) (001)) L) (((3) (01)) I0))	((((4) (110)) O) (((3) (10)) I1))
C2	((((3) (01)) O1) (((2) (11)) I0))	((((3) (10)) O1) (((2) (01)) I1))
C1	((((2) (11)) O0) (((1) (10)) I1))	((((2) (01)) O1) (((1) (11)) I0))
C0	((((1) (10)) O0) (((0) (100)) I))	((((1) (11)) O0) (((0) (111)) L))

2.3.2. *Validation de la fonction de routage « routing-dmin »*

Nous détaillons dans cette dernière partie, les principaux théorèmes validés dans le contexte de la fonction de routage *routing-dmin* définie précédemment.

2.3.2.1. Vérification de théorèmes intermédiaires

Avant de vérifier les deux principales contraintes de la fonction de routage, il existe deux autres théorèmes à démontrer (théorèmes 5-9 et 5-10). Ces derniers assurent que pour un ensemble donné de missives valides (reconnu par le prédicat *Missivep*), tous les nœuds origines et destinations de ces missives sont inclus *(subsetp)* dans l'ensemble des nœuds du réseau *(NodeSet)*. L'accès à ces deux ensembles se fait via les deux fonctions *M-orgs* et *M-Dests*.

Théorème 5-9. *L'ensemble origine des missives est inclus dans NodeSet*

```
(defthm M-orgs-subsetp-Nodeset
    (let ((NodeSet (gen-nodes-dmin pms))
          (pms-t (params-top-t pms))
          (top (gen-top-dmin pms-t)))
      (implies (and (ValidParamspD pms) (Missivesp M top)
                    (valid-params-t pms-t))
               (subsetp (M-orgs M) NodeSet)))
:hints (("GOAL" :in-theory (disable EXPLODE-NONNEGATIVE-INTEGER MOD FLOOR
DIGIT-TO-CHAR DEFAULT-<-2 Y-GEN-NODE GEN-NODE))))
```

Théorème 5-10. *L'ensemble destination des missives est inclus dans NodeSet*

```
(defthm M-dests-subsetp-Nodeset
    (let (NodeSet (gen-nodes-dmin pms))
          (pms-t (params-top-t pms))
          (top (gen-top-dmin pms-t)))
      (implies (and (ValidParamspD pms) (Missivesp M top)
                    (valid-params-t pms-t))
               (subsetp (M-dests M) NodeSet)))
:hints (("GOAL" :in-theory (disable EXPLODE-NONNEGATIVE-INTEGER MOD FLOOR
DIGIT-TO-CHAR DEFAULT-<-2 Y-GEN-NODE GEN-NODE))))
```

Preuves : Les preuves des deux théorèmes sont automatiques et rapides. ACL2 exécute les deux démonstrations par une simple induction sur la structure des fonctions accesseurs M-

Orgs et M-Dests. Finalement, ces deux théorèmes seront utiles dans les futures démonstrations.

2.3.2.2. Vérification du théorème 4-8

Le théorème 5-11 suivant est une instance valide du théorème générique 4-8. Ce dernier doit assurer la validité des routes produites par la fonction *compute-rtes-dmin*. Pour cela, on doit vérifier le prédicat *Ext-ValidRoutep* défini déjà de façon statique. On rappelle que ce prédicat exige que la longueur d'une route calculée soit supérieure ou égale à 1 et que le premier élément de la première connexion de cette route soit égale à l'origine du message à router.

Théorème 5-11. *Validité des routes produites par la fonction compute-rtes-dmin*

$\forall M, M_{lstp}\ (M,Top) \Rightarrow \forall m \in M,\ \forall r \in$ *compute-routes-dmin* $(Org_M\ (m), Dest_M\ (m), Top),$

Ext-ValidRoutep (r,m,Top)

Dans ACL2, ceci donne :

```
(defthm rte-ext-validroutep
   (let ((rt (compute-routes-dmin-call from to top)))
          (implies (and (valid-node-dmin from) (valid-node-dmin to)
                        (not (endp (ASSOC-EQUAL (CONS FROM '(L)) TOP)))
                        (valid-top-dmin top)(alistp top)(not (endp rt)))
          (ext-validroutep rt)))
:hints (("GOAL" :in-theory (disable compute-routes-dmin MOD FLOOR DIGIT-TO-
CHAR DEFAULT-<-2 Y-GEN-NODE GEN-NODE EXPLODE-NONNEGATIVE-INTEGER))))
```

Preuve : La preuve de ce théorème n'est pas directe. Comme le prédicat *Ext-ValidRoutep* n'est autre que la conjonction *(et)* de deux autres prédicats *(len et first)*, on doit procéder par décomposition du but final en deux lemmes intermédiaires (théorèmes 5-13 et 5-14). Chacun de ces deux lemmes sera relatif à chacun de ces prédicats. Un autre théorème est aussi utile (théorème 5-12) pour achever cette preuve. Enfin, on doit effectuer la désactivation *(disable)* de quelques définitions pour que le démonstrateur ne les prenne pas en considération.

Théorème 5-12. *Type de valeur retourné par compute-rtes-dmin*

```
(defthm true-listp-compute-routes-dmin
(let* ((pms-t (params-top-t pms))
       (top (gen-top-dmin pms-t))
       (rt (compute-routes-dmin from to top)))
 (implies (and (valid-node-dmin from)
               (valid-node-dmin to)
               (ValidParamspD pms) (valid-params-t pms-t))
          (true-listp rt)))
:hints (("GOAL" :in-theory (disable compute-routes-dmin))))
```

Preuve : Il s'agit d'une preuve triviale mais nécessaire dans ACL2. En réalité, le démonstrateur n'a même pas besoin de connaître les hypothèses fournies, il lui suffit de faire appel à la règle de réécriture relative au type de valeur stocké correspondant à la fonction *compute-routes-dmin (:type-prescription-rule)*. D'ailleurs, on désactive même la fonction *compute-routes-dmin* afin de guider au maximum la démonstration.

Théorème 5-13. *Longueur d'une route produite par la fonction compute-rtes-dmin*

```
(defthm len-routing-dmin->=-1
   (let ((rt (compute-routes-dmin from to top)))
      (implies (and (valid-node-dmin from) (valid-node-dmin to)
                    (not (endp (ASSOC-EQUAL (CONS FROM '(L)) TOP)))
                    (valid-top-dmin top)(alistp top)(not (endp rt)))
               (>= (len rt) 1)))
:hints (("GOAL" :in-theory (disable MOD FLOOR DIGIT-TO-CHAR DEFAULT-<-2 Y-
GEN-NODE GEN-NODE compute-routes-dmin EXPLODE-NONNEGATIVE-INTEGER))))
```

Preuve : La seule condition *(not (endp rt))* permet au démonstrateur de procéder par déduction. En effet, il existe une règle stockée dans a base de données qui associe à chaque liste *non nil*, une longueur supérieure ou égale à 1.

Théorème 5-14. *Le premier élément d'une route produite par la fonction compute-rtes-dmin*

```
(defthm first-routing-dmin
  (let ((rt (compute-routes-dmin from to top)))
    (implies (and (valid-node-dmin from) (valid-node-dmin to)
                  (not (endp (ASSOC-EQUAL (CONS FROM '(L)) TOP)))
                  (valid-top-dmin top)(alistp top)(not (endp rt)))
             (equal (caaar rt) from)))
:hints (("GOAL" :in-theory (disable compute-routes-dmin MOD FLOOR DIGIT-TO-
CHAR DEFAULT-<-2 Y-GEN-NODE GEN-NODE EXPLODE-NONNEGATIVE-INTEGER))))
```

Preuve : Le prouveur d'ACL2 opère automatiquement par induction sur la structure de la fonction *compute-routes-dmin* pour compléter la preuve de ce théorème. Dans ce cas, il n'est pas nécessaire de fournir à ACL2 un *hint* (conseil) lui montrant le schéma d'induction à utiliser *(:induct)*. Comme il existe plusieurs manières pour effectuer une démonstration donnée, il faudra encore avoir recours au mécanisme de désactivation. Autrement, les démonstrations risquent d'être inutilement allongées. D'autres lemmes plus élémentaires, non cités ici, ont dû aussi être fourni.

2.3.2.3. Vérification du théorème 4-9

Le théorème 4-9, concrétisé à travers le théorème 5-15, doit vérifier la validité des voyages produits par la fonction *routing-dmin*. Cette validité est reconnue par le prédicat récursif *Validfields-TrLst*. Ce prédicat assure que chacun des voyages calculés par la fonction *routing-*

dmin possède le format souhaité (prédicat *validfield-travelp*), que son identificateur est un naturel *(natp (IdV tr))*, que le contenu de son message est correctement représenté *(FrmV tr)* et enfin que l'ensemble des routes calculées forment une liste *(true-listp (RoutesV tr))*.

Définition 5-11. *Définition du prédicat Validfields-TrLst*

```
(defun Validfields-TrLst (TrLst)
  (if (endp TrLst)
      t
    (let ((tr (car TrLst)))
      (and (validfield-travelp tr)
           (natp (IdV tr))                    ;; id is a natural
           (FrmV tr)                          ;; frm /= nil
           (true-listp (RoutesV tr))
           (Validfields-TrLst (cdr TrLst))))))
```

Théorème 5-15. *Validité des voyages produits par la fonction routing-dmin*

$\forall M, M_{lstp}(M,Top) \Rightarrow V_{lstp}(routing\text{-}dmin\ (M,Top))$

Ce qui donne en ACL2 :

```
(defthm Valid-voyg-routing-dmin
    (let* ((pms-t (params-top-t pms))
           (NodeSet (gen-nodes-dmin pms))
           (top (gen-top-dmin pms-t))
           (voyg (routing-dmin M top)))
     (implies (and  (ValidParamspD pms)(valid-params-t pms-t)
                    (Missivesp M NodeSet))
              (Validfields-TrLst  voyg))))
:hints (("GOAL" :in-theory (disable MOD FLOOR DEFAULT-<-2 Y-GEN-NODE GEN-
NODE gen-nodes-dmin compute-routes-dmin EXPLODE-NONNEGATIVE-INTEGER))))
```

Preuve : En procédant par décomposition du but final en sous-buts élémentaires, deux théorèmes sont nécessaires pour parvenir à accomplir la démonstration. Ce sont les théorèmes 5-16 et 5-17.

Théorème 5-16. *Type de valeur retourné par routing-dmin*

```
(defthm True-listp-routing-dmin
    (let* ((pms-t (params-top-t pms))
           (NodeSet (gen-nodes-dmin pms))
           (top (gen-top-dmin pms-t))
           (voyg (routing-dmin M top)))
     (implies (and  (ValidParamspD pms)(valid-params-t pms-t)
                    (Missivesp M NodeSet))
              (true-listp voyg))))
:hints (("GOAL" :in-theory (disable MOD FLOOR DIGIT-TO-CHAR DEFAULT-<-2 Y-
GEN-NODE  GEN-NODE  gen-nodes-dmin  compute-routes-dmin  EXPLODE-NONNEGATIVE-
INTEGER))))
```

Preuve : La preuve de ce théorème est directe. Le démonstrateur fait juste appel à la règle qui donne le type de la valeur retournée par la fonction *routing-dmin (:type-prescription-rule)*.

Cette règle a dû être stockée dans la base de données depuis l'admission de la fonction *routing-dmin*.

Théorème 5-17. Validité du format d'un voyage donné

```
(defthm valid-field-travelp-rt
    (let* ((pms-t (params-top-t pms))
           (NodeSet (gen-nodes-dmin pms))
           (top (gen-top-dmin pms-t))
           (rt (compute-routes-dmin from to top)))
     (implies (and  (ValidParamspD pms)(valid-params-t pms-t)
                    (Missivesp M NodeSet))
              (validfield-travelp tr))))
:hints (("GOAL" :in-theory (disable MOD FLOOR DIGIT-TO-CHAR DEFAULT-<-2 Y-
GEN-NODE GEN-NODE gen-nodes-dmin compute-routes-dmin EXPLODE-NONNEGATIVE-
INTEGER))))
```

Preuve : La preuve de ce théorème est assez longue. Elle doit faire appel à toutes les définitions des fonctions élémentaires définies pour vérifier que la route produite par la fonction *compute-routes-dmin* est au format du prédicat *validfield-travelp*.

2.4. *Vérification de la conformité des définitions concrètes*

Après avoir démontré séparément les contraintes obligatoires sur la topologie et le routage dans un réseau Delta MIN quelconque, il faut maintenant vérifier la conformité des définitions du niveau concret avec les définitions du niveau générique. Cela revient à démontrer que toutes les définitions du niveau concret *(au niveau du fichier top-dmin.lisp et rt-dmin.lisp)* sont des instances valides des fonctions génériques *(au niveau du fichier gen-top.lisp et gen-rt.lisp)*. Cette contrainte est illustrée à travers le théorème 5-18 suivant.

Théorème 5-18. *Conformité avec les définitions génériques*

```
(defthm check-DMIN-TOP t

   :rule-classes nil
   :otf-flg t
   :hints (("GOAL"
   :use

((:functional-instance nodeset-generates-valid-nodes
(ValidParamsp ValidParamspD)
(NodeSetp valid-nodes-dmin)
(NodesetGenerator gen-nodes-dmin)
(gen-top gen-top-dmin)
(params-top params-top-t)
(ValidParamsp-top valid-params-t)
(valid-top valid-top-dmin)
(valid-ext2s valid-ext2s-dmin)
(routing-dmin ext-routing))
```

```
:in-theory (disable  subsets-are-valid nodeset-generates-valid-nodes gen-
top-generates-valid-top-1 gen-top-generates-nodes-in-nodeset ext-routing))

("Subgoal 4" :in-theory (disable gen-top-generates-nodes-in-nodeset  GEN-
NODES-DMIN-S-pms  gen-nodes-dmin-sw-1-pms  GEN-NODES-DMIN-d-pms  ALL-Y-
EQLABLE-1 valid-nodes-dmin  GEN-NODES-INV1-CALL gen-top-dmin gen-nodes-dmin
NOT PARAMS-TOP-S PARAMS-TOP-SW  PARAMS-TOP-T))
;;:induct (ext2s-top top-dmin)

("Subgoal 3" :in-theory (disable gen-top-generates-valid-top-1 Validfp-dmin
GEN-NODES-DMIN-S-pms gen-nodes-dmin-sw-1-pms GEN-NODES-DMIN-d-pms NETWORK-
PERM Validfp-dmin gen-nodes-dmin NOT PARAMS-TOP-S PARAMS-TOP-SW  PARAMS-
TOP-T VALIDPARAMSPD VALID-PARAMS-T valid-S valid-SubS valid-D valid-SubD
valid-Sw valid-SubSw GEN-TOP-DMIN))

("Subgoal 1" :in-theory (disable nodeset-generates-valid-nodes rev GEN-
NODES-INV1-CALL  GEN-NODES-DMIN-SW-1  GEN-NODES-DMIN  EXPLODE-NONNEGATIVE-
INTEGER valid-nodes-dmin))))
```

Preuve : La preuve de conformité entre les niveaux générique et concret dans ACL2 est automatique. En effet, après avoir démontré les contraintes sur la topologie de façon indépendante (théorèmes 5-1, 5-5 et 5-7) et sur le routage (théorèmes 5-11 et 5-15), le prouveur va utiliser *les règles de réécriture* correspondantes pour générer automatiquement la démonstration du « check ». La seule astuce dans ce cas consiste en la désactivation de quelques définitions au niveau de chaque sous-but (Subgoal) pour que ACL2 ne refasse pas certaines démonstrations déjà produites auparavant.

2.5. *Vérification du théorème de correction global du modèle*

Après avoir démontré la conformité des deux composantes topologie et routage de notre modèle formel des réseaux multi-étages de la famille Delta dédiés aux MPSoCs, nous pouvons maintenant déduire que le théorème global de correction, assurant la fiabilité des messages transmis à travers le réseau, est correcte (théorème 5-19).

En réalité, l'approche générique n'exige pas de redémontrer son théorème global de correction. Ceci peut être vérifié depuis les réseaux qui ont été déjà validés à travers cette approche (Octagon, algorithmes de routage adaptatifs). De plus, même en étendant le modèle (cas de Hermes), il n'a pas été nécessaire de redémontrer ce théorème.

Les différents théorèmes précédemment développés ont servi pour valider le théorème final 5-19. Dans la présentation de ces théorèmes, nous avons essayé de fournir les principaux lemmes intermédiaires servant à aboutir à la démonstration du but final. Outre l'apprentissage d'ACL2, la principale difficulté rencontrée concerne le développement des tactiques pour parfaire les différentes démonstrations. Ces lemmes n'étaient pas toujours réutilisables dans le

cadre d'autres démonstrations. Dans certains cas, il fallait même désactiver quelques uns pour guider au plus le démonstrateur. Cette difficulté met en évidence le problème du chapeau mexicain évoqué au chapitre 2 (figure 12).

Théorème 5-19. *Fiabilité des réseaux Delta MINs dédiés aux MPSoCs*

$$\forall rst \in R, \exists! \ t \in T, \begin{cases} IdR(rst) = IdT(t) \\ \wedge \ MsgR(rst) = MsgT(t) \\ \wedge \ DestR(rst) = DestR(t) \end{cases}$$

Preuve : La preuve de ce théorème dans ACL2 n'est pas nécessaire. Par l'approche du modèle générique étendu, on peut déduire la correction de ce théorème.

3. Conclusion

Nous avons détaillé dans ce chapitre notre spécification fonctionnelle dans une notation formelle des réseaux Delta MINs dédiés aux MPSoCs. Nous avons utilisé le démonstrateur de théorèmes ACL2 pour vérifier les différentes contraintes sur le modèle défini. Le développement du modèle formel dans ACL2 s'est fait sur une machine Pentuim 4 à 2,4 GHz fonctionnant sous Linux avec une mémoire à 256 MB. Le temps de vérification des théorèmes n'est pas assez grand vu le mécanisme d'activation et de désactivation des règles auquel on a eu recours.

Conclusion et perspectives

1. Conclusion

Le propos de ce sujet de mastère a été le développement d'un modèle formel pour des réseaux multi-étages dédiés aux systèmes multiprocesseurs sur puce (MPSoCs). Notre travail décrit en effet une méthodologie pour l'intégration des méthodes formelles dans la vérification des architectures de communication sur puce.

Plusieurs étapes ont été essentielles pour le développement du modèle formel. Pour mettre le problème dans son cadre, il a été nécessaire d'étudier le domaine des systèmes sur puce (SoCs), des formalismes et des réseaux multi-étages (MINs) de la famille Delta. Au cours de cette phase, plusieurs travaux antérieurs ont dû être discuté.

Dans le cadre de la vérification formelle des spécifications, l'un de ces travaux nous a particulièrement intéressées. En effet, cette étude a été basée sur une formalisation complètement générique des communications sur puce. Elle est concrétisée à travers un modèle appelé GeNoC (Generic Networks on chip). Ce dernier s'est avéré très efficace lors de la vérification de certains réseaux sur puce (NoCs) tels que l'Octagon. C'est un modèle qui a aussi témoigné d'une grande flexibilité pour valider les communications sur puce dans le réseau Hermes. La correction globale du modèle générique est illustrée à travers un théorème de fiabilité qui assure que « tout message émis dans le réseau atteint sa destination sans modification de son contenu. ». L'approche GeNoC garantit de la sorte la correction de ce théorème pour n'importe quel réseau sur puce, instance de GeNoC et vérifiant ses contraintes génériques.

Nous avons alors voulu appliquer la même approche pour valider les réseaux multi-étages dédiés aux MPSoCs. Toutefois, nous nous sommes rapidement confrontées à une grande contrainte : le modèle générique ne tient pas compte concrètement de l'aspect topologie et plus précisément des connexions entre les noeuds. Par ailleurs, il considère implicitement l'existence des liens entre les nœuds d'une route calculée ; or une telle connaissance serait primordiale pour la validation des MINs. Pour résoudre ce problème, nous avons eu l'idée de concevoir et implémenter une composante topologie générique, ainsi que la composante routage résultante du même niveau. Pour développer cette extension, il a fallu identifier les propriétés pertinentes et communes à toutes les topologies et les traduire sous forme de théorèmes dans la logique ACL2. Durant cette phase, nous avons essayé de tirer profit des

bases de la théorie des graphes pour construire formellement les connexions de toute la topologie d'un réseau sur puce. Les deux composantes ainsi développées ont pu finalement être intégré au sein du modèle GeNoC décrivant alors un modèle plus réaliste des communications dans les réseaux sur puce.

Dans une seconde phase, nous avons formalisé en suivant l'approche GeNoC étendue et dans la logique du démonstrateur de théorèmes ACL2, les communications dans les réseaux multi-étages de la famille Delta. Nous nous sommes notamment focalisées sur les deux composantes topologie et routage. Durant chacune des étapes précédentes, il nous a fallu en plus prendre en considération les contraintes des systèmes multiprocesseurs sur puce pour lesquels est destiné le modèle formel. Les réseaux multi-étages ainsi spécifiés et vérifiés formellement forment une instance valide du modèle GeNoC étendu.

2. Perspectives

Nous estimons qu'il est possible d'appliquer l'extension du modèle générique pour donner une spécification formelle de toute architecture de communication sur puce. En effet, pour valider des réseaux ayant une topologie en grille 2D incomplète ou des réseaux indirectes, il faudra adopter le modèle générique étendu et non plus le modèle GeNoC initial.

Il est vrai que dans le cadre de ce travail de mastère, l'objectif initial était de valider formellement l'ensemble des communications dans les réseaux multi-étages englobant le routage et l'ordonnancement. Cependant, l'intérêt que nous avons dû porter au modèle GeNoC nous a obligé de prévoir l'extension du niveau générique de façon que la composante ordonnancement n'ait pas pu finalement être validée. Nous estimons qu'il suffirait de choisir l'un des algorithmes d'ordonnancement déjà vérifié pour l'intégrer comme une composante pré-validée dans l'ensemble du modèle final. Une méthode similaire a été utilisée pour valider les algorithmes de routage adaptatifs dans des réseaux ayant une topologie en grille 2D.

Une autre perspective possible comme extension à ce mastère consiste en la formalisation de l'implémentation des réseaux MINs dédiés aux MPSoCs. Ceci revient à descendre dans le niveau d'abstraction pour s'occuper des détails de niveau implémentation de ces réseaux (signaux, files d'attente…). Le modèle développé dans le cadre de ce mastère servira alors comme un modèle de niveau spécification pour le futur prototype.

Bibliographie

(ACL2) http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html.

(ACM, 2006) ACM: Press Release, March 15, 2006.

(Arga, 1983) Argawal D.P., « Graph theoretical analysis and design of multistage interconnection networks », *IEEE Transactions on Computer*, vol. 32, n° 7, July 1983.

(Aria, 1996) « ARIANE 5 - Flight 501 Failure », http://www.cnes.fr/, rapport n° 501, 1996.

(ARM, 1999) ARM Limited. AMBA specification, 2.0 editions, 1999.

(Arte) Arteris, http://www.arteris.net.

(Ateris, 2005) Ateris, A comparison of Network-on-Chip and Busses, www.ateris.com, 2005.

(Beni *et al.*, 2002) Benini L., Micheli G., « Networks on chips: a new SoC paradigm », *IEEE Computer*, vol. 35, n° 1, p. 70-78, 2002.

(Ber *et al.*, 1998) Berzin S., Campos S., Clarke E.M, « Compositonnal Reasonning in Model Checking », *in COMPOS'97: The significant Difference*, p. 81-102, Springer-Verlag, 1998.

(Bjer *et al.*, 2006) Bjerregaard T., Mahadevan S., « A survey of research and practices of Network-on-chip », *ACM Computing Surveys*, vol. 38, n° 1, 2006.

(Bol, 2001) Bolduc C., Les démonstrateurs automatiques de théorèmes, rapport de recherche, Faculté des sciences et de génie, Université de Laval, 2001.

(Borr *et al.*, 2006) Borrione D., Helmy A., Pierre L., « ACL2-based Verification of the Communications in the Hermes Network on Chip », *Proc. International Workshop on Symbolic Methods and Applications to Circuit Design (SMACD'06)*, 2006.

(Broc *et al.*, 1999) Brock B., Hunt W. A., « Formal analysis of the motorola CAP DSP », *in Industrial-Strength Formal Methods*, Springer-Verlag, 1999.

(Bry, 1986) Bryant R.E., « Graph-based Algorithms for Boolean Function

Manipulation », *IEEE Transactions in Computers*, vol. 8, n° 35, p. 677-691, 1986.

(Cesa *et al.*, 2002) Cesario W.O., Lyonnard D., Nicolescu G., Paviot Y., Yoo S., Gauthier L., Diaz-Nava M., Jerraya A.A.; « Multi-processor SoC platforms: a component-based design approach », *IEEE Design and Test of Computers*, vol. 19, n° 6, p. 52–63, 2002.

(Cheu et al., 1986) Cheung T., Smith J. E., « A simulation study of the CRAY X-MP memory system », *IEEE Transactions on Computers*, vol. 35, n° 7, IEEE Computer Society, Washington, p. 613–622, 1986.

(Clar *et al.*, 1996) Clarke E. M., Win J. M., « Formal Methods: State of the Art and Future Directions », *ACM 50th-anniversary issue: strategic directions in computing research, ACM Computing Surveys (CSUR)*, vol. 28 n° 4, ACM, New York, p. 626-643, 1996.

(Clar *et al.*, 1986) Clarke E.M., Emerson E.A., Sistla A.P., « Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications », *in Transactions on Programming Languages and Systems*, vol.8, n° 2, p. 244- 263, 1986.

(Coll, 2003) Collier M., « A Systematic Analysis of Equivalence in Multi-Stage Networks », *JLT*, 2002.

(Coq) http://pauillac.inria.fr/coq/.

(Core) http://www-3.ibm.com/chips/products/coreconnect.

(Desh, 2000) Desharnais J., Notes de cours du cours de logique et structures discrètes. Département d'informatique, université Laval, 2000.

(Elle *et al.*, 2008) Elleuch M., Aydi Y., Abid M., « Formal Specification of Delta MINs for MPSOC in the ACL2 Logic », *in Proceedings of Forum on Design and specification Languages - FDL '08* (à apparaître).

(Elle *et al.a*, 2008) Elleuch M., Aydi Y., Abid M., « Formal Specification and Verification of a Delta-MIN Based Interconnection Architecture for MPSoC », *in Proceedings of ReCoSoC '08* (à apparaître).

(Freek) http://www.cs.ru.nl/~freek/digimath/index.html

(Gebr *et al.*, 2005) Gebremichael B., Vaandrager F.W., Zhang M., Goossens K., Rijpkema E., Radulescu, A., « Deadlock Prevention in the Aethereal Protocol », in D. Borrione and W. Paul, editors,

*Proceedings 13th IFIP Advanced Research Working Conference on Correct Hardware Design and Verification Methods (*CHARME'05*)*, p. 345-348, 2005.

(Giac *et al.*, 1991) Giacopelli J., Hickey J., Marcus W., Sincoskie W., Littlewood M., « Sunshine: A high-performance self routing broadband packet switch Architecture », *IEEE Journal on Selected Areas in Communications*, vol. 9, n° 8, p.1289-1298, 1991.

(Gord,1989) Gordon M.J.C., Lectures on the Specification and Verification of Hardware, notes de cours, université de Cambridge, 1989.

(Gott *et al.*, 1983) Gottlieb A., Grishman R., Kruskal C.P., McAuliffe K.P., Rudolph L., Snir M., « The NYU ultracomputer-Designing and MIMD shared memory parallel computer », *IEEE Transactions on Computers*, vol. 32, n° 2, p. 175-189, 1983.

(Gries *et al.*, 1994) Gries D., Schneider F.B., A Logical Approach to Discrete Math. Springer, 1994.

(Gro *et al.*, 2005) Groth D., Skandier T., *Network + Study Guide*, Sybex, San Francisco, 2005.

(Hall, 1990) Hall A., « Seven Myths of Formal Methods », *IEEE Software*, p. 11-19, 1990.

(Ho *et al.*, 2001) Ho, Mai K., Horowitz M., « The future of wires », *Proceedings of the IEEE*, p. 490-504, 2001.

(HOL) http://www.cl.cam.ac.uk/research/hvg/HOL/.

(IMEC, 2006) IMEC, « MultiMedia Multi-Format (3MF) codec », 2006.

(Intel, 2006) Intel Corp, « New Dual-Core Intel Itanium 2 Processor Doubles Performance, Reduces Power Consumption », 2006 http://www.intel.com/pressroom/archive/releases/.

(Isabel) http://www.cl.cam.ac.uk/Research/HVG/Isabelle.

(ISO) http://www.iso.org/iso/.

(Jant, 2006) Jantsch, A., « Standards for NoC: What can we gain? », *Future Interconnects and Networks on Chip Workshop, DATE 2006*, 2006.

(Kahl, 2005) Kahle, J., « The Cell Processor Architecture », *in MICRO*, IEEE Computer Society, 2005.

(Kari *et al.*, 2001) Karim F., Nguyen A., Dey S., Rao R., « On-chip Communication Architecture for OC-768 Network Processors », *Proc. Design Automation Conf. (DAC 01)*, 2001.

(Kauf *et al.*, 1996) Kaufmann M., Moore J. S., « ACL2: An industrial strength version of nqthm », *in Compass'96: Eleventh Annual Conference on Computer Assurance*, p. 23, 1996.

(Kauf *et al.*, 2000) Kaufmann M., Manolios P., Moore J., *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers, 2000.

(Kauf *et al.a*, 2000) Kaufmann M., Manolios P., Moore J., *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Academic Publishers, 2000.

(Kauf *et al.*, 2001) Kaufmann M., Moore J S., « Structured Theory Development for a Mechanized Logic », *J. Autom. Reasoning, Kluwer Academic Publishers*, USA, p. 161-203, 2001.

(Koy, 1990) Koymans R., « Specifying Real-Time Properties with Metric Temporal Logic », *Real-Time Systems*, vol. 2, n° 4, p. 255-299, Kluwer, 1990.

(Kropf, 1997) Kropf T., *Formal Hardware Verification - Methods and Systems in Comparison*, Springer-Verlag, London, 1997.

(Kropf, 1999) Kropf T., *Introduction to Formal Hardware Verification*, Springer Verlag, London, 1999.

(Kut, 1994) Kutvonen L., Comparison of the DRYAD Trading System to ODP-Trading Function Draft, rapport technique n° 51, Université de Helsinki, 1994.

(Laprie, 1990) Laprie J.C., « Dependability: Basic Concepts and Associated Terminology », rapport technique n° 31, PDCS, May1990.

(Leve *et al.*, 1993) Leveson N. G., Turner C. S., « An Investigation of the Therac-25 Accidents », *Computer*, vol. 26, n° 7, p.18-41, July 1993.

(Man, 1982) Manna Z., « Verification of sequentiel programs : Temporal axiomatization », *in M. Broy and G. Scmidt, editores, Theorical Foundations of Programming Methodology*, p. 53-102, 1982.

(Mark, 1994) Markoff J., « Circuit Flaw Causses Pentuim to Miscalculate: Intel Admits », *New York Times*, 24th November 1994.

(McCa, 1962) McCarthy, J., « Towards a Mathematical Science of Computation », *in Proceedings of the Information Processing Cong 62*, p. 21-28, 1962.

(McMill, 1992) McMillian K., The smv system, rapport technique, université de Carnegie Mellon, 1992.

(Miller *et al.*, 1995) Miller S. P., Srivas, M. « Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods », *in Proc. Workshop on Industrial-Strength Formal Specification Techniques*, IEEE Computer Society, p. 2-6, 1995.

(Moore *et al.*, 1998) Moore J S., Lynch T., Kaufmann M., « A mechnically checked proof of the correctness of the kernel of the AMD5K86 floating point division algorithm », *IEEE Transactions on Computers*, vol. 47, n° 9, p. 913-926, 1998.

(Moore, 2003) Moore J S., « Proving theorems about Java and the JVM with ACL2 », *In M. Broy and M. Pizka, editors, Models, Algebrs and Logic of Engineering Software*, p. 227-290, IOS Press, Amsterdam, 2003.

(Nobl, 2002) Noblanc J.P., « EDA and Systems-on-Chip: A key Challenge for MEDEA+ », *in Proc. MEDIA+ Design Automation Conference, Stresa*, Italy, p. 23-25, 2002.

(ORA, 1999) ORA Canada, « ORA Canada : EVES Verification System. ORA Canada », 1999, http://www.ora.on.ca/eves.html.

(Owre *et al.*, 1994) Owre S., Rushby J. M., Shankar N., Srivas M. K, A tutorial on using PVS for hardware verification, *in Proc. 2nd International Conference on Theorem Provers in Circuit Design (TPCD'94), LNCS*, vol. 901, p. 258–279, Springer, 1994.

(Pandya, 2001) Pandya K., « Model checking ctl*[dc] », *in Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, p. 559–573, Springer, 2001.

(Pate, 1981) Patel J. H., « Performance of processor-memory interconnections for multiprocessors », *IEEE. Trans. Comput.*, vol. 30, n° 10, 1981.

(Pfis *et al.*, 1985) Pfister et al., « The IBM Research Parallel Processor prototype (RP3) : introduction and architecture », *1985 International*

Conference on Parallel Processing, 1985.

(PVS) http://pvs.csl.sri.com/.

(Rajan, 1997) Rajan S.P., Fujita M., Yuan K., Lee M. T-C., « High-level design and validation of ATM switch », *in Proc. IEEE International High-Level Design Validation and Test Workshop(HLDVT'97)*, IEEE Computer Society, p. 40–44, 1997.

(Rijp , 2003) Rijpkema, Goossens E., Radulescu K., « A. Trade Offs in the Design of a Router with Both Guaranteed and Best-Effort Services for Networks on Chip », *in Design, Automation and Test in Europe (DATE'03)*, 2003.

(Royc *et al.*, 2003) Roychoudhury A., Mitra T., Karri S.R., « Using Formal Techniques to Debug the AMBA System-On-Chip Protocol », *in Design, Automation and Test in Europe*, IEEE Computer Society, p. 828-10833, 2003.

(Schm, 2006) Schmaltz J., Une formalisation fonctionnelle des communications sur la puce, Thèse de doctorat, Université Joseph Fourier– Grenoble 1, 2006.

(Schm *et al.*, 2006) Schmaltz J., Borrione D., « Towards a Formal Theory of Communication Architecture in the ACL2 Logic », *Proc. of 6th international workshop on the ACL2 theorem prover and its applications*, ACM Press, New York, p. 47- 56, 2006.

(Soni) http://www.sonicsinc.com/sonics/index_html.

(Spir, 2004) Spirakis, G., « Beyond Verification: Formal Methods in Design », in A. Hu and A. Martin, editors, *Formal Methods in Computer-Aided Design (FMCAD '04), LCNS*, vol. 3312, Springer-Verlag, 2004.

(Spiv, 1992) Spivey J.M., The Z Notation: A Reference Manual, Second Edition, Prentice Hall, 1992.

(Stun *et al.*, 1995) Stunkel C. B., Shea D. G., Aball B., Atkins M. G., Bender C. A., Grice D. G., Hochschild P., Joseph D. J., Nathanson B. J., Swetz R. A., Stucke R. F., Tsao M., Varker P. R., « The SP2 High-Performance Switch », *IBM Systems Journal*, vol. 34, n° 2, IBM Corp., Riverton, USA, p. 185–204, 1995.

(Szym *et al.*, 1994) Szymanski T., Hamacher V., « On the universality of multistage interconnection networks », *IEEE Computer Society Press*, 1994, p. 73-101.

(UPPAAL) UPPAAL. URL: http://www.uppaal.com.

(Wu *et al.*, 1980) Wu C.L., Feng T.Y., « On a class of multistage interconnection networks », *IEEE Transactions on Computers*, August 1980.

(Zegu, 1993) Zegura E. W., « Architectures for ATM switching systems », *IEEE Communications Magazine*, p. 28-37, 1993.

(Zei et al., 2005) Zeitzoff P.M., Chung J. E., « A perspective from the 2003 ITRS », IEEE circuits & devices magazine, 2005.

(Zeves) http://www.ora.on.ca/z-eves/welcome.html.

Annexes

L'annexe suivante comporte les éléments essentiels de la syntaxe de l'outil de démonstration de théorèmes ACL2 (ACL2).

1. La syntaxe ACL2

La syntaxe d'ACL2 est constituée des éléments suivants :

- *Variables :* `X`, `x`, `alist`, `temp2`, `prev-temp`
- *Constantes :*
    - *Nombres :* `23`, `-7`, `22/7`, `#c(3 5)`
    - *Caractères :* `#\A`, `#\a`, `#\,` , `#\Newline`, `#\Space`
    - *Chaînes de caractères :* `"Test"`, `"He said \"Hi!\" as we passed."`
    - *Symboles :* `t`, `nil`, `'ABC`, `'abc`, `'true-list`, `:key`, `math::abs`, `|John|`
    - *Paires (Conses) :* `'(1 2 3)`, `'((A . 1) (B . 2))`, `'((1 . 2) . (3 . 4))`
- *Fonctions or les appels de macros :* `(`*`fn arg1 arg2 ... argn`*`)`

2. Les Commandes utiles d'ACL2

- *Sélection d'un package :* `(in-package` *`"pkg"`*`)`
- *Création d'un package :* `(defpkg` *`"pkg" '(imported symbols)`*`)`
- *Modes :*

`(program)/:program`	Prototypage des définitions et tests
`(logic) / :logic`	Définition de fonctions, preuve de terminaison et autres propriétés
`(redef)` / `:redef`	Permet les redéfinitions des fonctions

- *Constantes :* `(defconst` *`*var* term`*`)`
- *Définitions :*

```
(defun fn (var1 ... varn)
    (declare (xargs :measure term1   ; declarations are optional
                    :guard   term2
                    :hints   hints
                    ...others))
    body)
```

- *Vérification de guardes :* `:set-guard-checking t/nil`
- *Compilation :* `:comp t`
- *Fonctions :* une liste complète des fonctions Common Lisp supportée par ACL2 est donnée ci-dessous.

- *Procédures de contrôle*

`(if x y z)`	si *x* est non-nil alors *y* sinon *z*
`(equal x y)`	prédicat d'égalité
`(cond (x1 y1) (x2 y2) ... (t z))`	`(if x1 y1 (if x2 y2 ... z))`
`(case key (c1 y1) (c2 y2) ... (t z))`	`(cond ((equal key 'c1) y1) ((equal key 'c2) y2) ... (t z))`
`(let ((var1 val1) ...) body)`	lier les variables locales en parallèle
`(let* ((var1 val1) ...) body)`	lier les variables locales de façon séquentielle
`(mv-let (var1 ...) vector body)`	lier les variables à un vecteur à plusieurs valeurs
`(mv val1 ...)`	retourne un vecteur à plusieurs valeurs

- *Booléen*

`(and p q ...)`	opérateur logique de conjonction
`(or p q ...)`	opérateur logique de disjonction
`(implies p q)`	implication logique
`(not p)`	négation logique
`(iff p q)`	équivalence logique

- *Arithmetique*

`(acl2-numberp x)`	reconnaît n'importe quelle type de nombres ACL2
`(integerp x)`	reconnaît les entiers
`(rationalp x)`	reconnaît les rationnels
`(complex-rationalp x)`	reconnaît les nombres complexes
`(equal x 0), (zerop x), (zip x), (zp x)`	reconnaît « x = 0 »
`(< x y)`	relation strictement inférieure
`(<= x y)`	relation inférieure ou égale
`(>= x y)`	relation supérieure ou égale
`(+ x y ...)`	opérateur d'addition
`(* x y ...)`	opérateur de multiplication
`(- x y)`	opérateur de soustraction

`(- x)`	opérateur de négation
`(/ x y)`	division rationnelle
`(1- x)`	décrémente de 1
`(1+ x)`	incrémente de 1
`(numerator r)`	numérateur d'un nombre rationnel
`(denominator r)`	dénominateur d'un nombre rationnel

- *Caractères*

`(characterp x)`	reconnaît les objets caractères
`(char-code char)`	convertit le code caractère au code entier
`(code-char n)`	convertit le code entier au code caractère

- *Chaînes de caractères*

`(stringp x)`	reconnaît les chaînes de caractères
`(char str n)`	cherche le nième caractère de la chaîne str
`(coerce str 'LIST)`	convertit la chaîne de caractère en une liste
`(coerce charlist 'STRING)`	convertit une liste en une chaîne de caractère
`(length str)`	longueur d'une chaîne de caractères ou d'une liste

- *Les paires « cons » et les listes*

`(consp x)`	reconnaît les paires ordonnées
`(cons x y)`	construit les paires ordonnées
`(car pair)`	le premier composant d'une paire ordonnée
`(cdr pair)`	le deuxième composant d'une paire ordonnée
`(endp x)`	reconnaît les non-paires
`(atom x)`	reconnaît les non-paires
`(list x0 x1 ... xk)`	construit une liste d'éléments donnés
`(list* x0 x1 ... xk cdrk)`	construit une liste d'éléments avec cdrk comme dernier élement
`(caar pair)`	le car du car
`(cadr pair)`	le car du cdr
`(cdar pair)`	le cdr du car
`(cddr pair)`	le cdr du cdr
`...`	...
`(cddddr pair)`	le cdr du cdr du cdr du cdr
`(append x y ...)`	concatène les listes linéaires
`(assoc-equal x alist)`	retourne l'élément associé à x dans une liste d'association
`(nth n lst)`	retourne le nième élément d'une liste
`(length list)`	longueur d'une liste (ou d'une chaîne)
`(true-listp x)`	reconnaît les listes linéaires

République Tunisienne
Ministère de l'Enseignement Supérieur,
De la Recherche Scientifique
et de la Technologie

Université de Sfax
École Nationale d'Ingénieurs de Sfax

Cycle de Formation Doctorale
dans la Discipline informatique
Mastère NTSID
Mémoire de *MASTERE*
N° d'ordre : 2008 – 496

DEVELOPPEMENT D'UN MODELE FORMEL POUR DES RESEAUX MULTI-ETAGES DEDIES AUX SYSTEMES MULTIPROCESSEURS SUR PUCE

Maïssa ELLEUCH SAHNOUN

الخلاصة : يتنزل هذا العمل ضمن الإطار العام لتطبيق الطرق الرسمية للتحقق من الدوائر الرقمية. وهو يتألف من تصميم وتطوير نموذج رسمي للشبكات متعددة الطوابق المخصصة لهذا النوع من الأ نظمة. هذا النموذج هو محدد لها ، وتم التحقق من صحته في منطق اداة النظريات ACL2. وهو يقوم على اساس تمديد نموذجية GeNoC.

Résumé : Ce travail est inclus dans le cadre de l'application des méthodes formelles dans la vérification des circuits numériques. Il consiste en la conception et le développement d'un modèle formel pour des réseaux multi-étages dédiés aux systèmes multiprocesseurs sur puce. Ce modèle est spécifié et vérifié dans la logique du démonstrateur de théorèmes ACL2. Il est basé sur l'extension d'un modèle générique dénoté GeNoC (Generic Networks on Chip) décrivant les communications sur puce.

Abstract: This work is to be seen as within the general context of formal hardware verification. It consists on the design and the development of a formal model for multistage interconnection networks dedicated to multiprocessor systems-on-chip. This model is specified and verified in the ACL2 theorem proving environment. It is based on the extension of a generic model called GeNoC (Generic Networks on Chip) describing on-chip communications.

المفاتيح : شبكات على الرقاقة، التحقق من الدوائر الرقمية، الطرق الرسمية، الشبكات متعددة الطوابق، مظاهرة نظرية، ACL2.

Mots clés : réseaux sur puce, vérification formelle des circuits numériques, réseaux multi-étages, démonstration de théorèmes, ACL2.

Key-words: networks on chip, formal hardware verification, multistage interconnection networks, theorem proving, ACL2.