Une formalisation fonctionnelle des communications sur la puce

J. Schmaltz

UNIVERSITÉ JOSEPH FOURIER - GRENOBLE 1

THÈSE

présentée par

Julien SCHMALTZ

et préparée

au sein de l'équipe VDS du laboratoire TIMA

Présentée devant

DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER
Spécialité MICRO ET NANO ELECTRONIQUE

École Doctorale : E.E.A.T.S

Titre de la thèse :

Une formalisation fonctionnelle des communications sur la puce

soutenue le 31 janvier 2006 devant la commission d'examen

Président :	Marc	RENAUDIN	Professeur, INP Grenoble
Rapporteurs :	J Strother	MOORE	Professeur, Univ. du Texas à Austin, USA
	Pierre	FRAIGNIAUD	Directeur de Recherches, LRI, Orsay
Examinateurs :	Laurence	PIERRE	Professeure, Univ. de Nice-Sophia Antipolis
	Marcello	COPPOLA	Head Res. Lab., STMicroelectronics, Grenoble
Directrice de thèse :	Dominique	BORRIONE	Professeure, Univ. Joseph Fourier

Si les faits ne correspondent pas à la théorie, changez les faits.
Albert Einstein

Remerciements

En premier lieu, je tiens à remercier J Strother Moore et Pierre Fraigniaud pour avoir accepté la tâche de rapporteur. Leurs encouragements et leur soutien sont autant de sources de motivation pour mon avenir.

J'adresse tous mes remerciements aux membres du Jury. Un grand merci à Laurence Pierre, Marc Renaudin et Marcello Coppola, je sais leur temps précieux.

Si cette thèse est mienne, elle n'aurait jamais vu le jour sans Dominique Borrione. Entre Dominique et moi, c'est aujourd'hui une "longue" histoire. Professeure à Polytech'Grenoble, elle m'a donné goût à l'informatique et à la vérification formelle. Durant ma thèse, elle fut pour moi un réel mentor. La curiosité n'est pas toujours un vilain défaut, notamment en recherche, mais il faut savoir se fixer un cap. Dominique a su me montrer comment je devais tenir la barre pour arriver à mes fins. Je la remercie pour sa confiance et ses encouragements ; pour m'avoir permis de présenter mes travaux lors de divers congrès ; pour ses critiques constructives sur tous les aspects de mon travail aussi bien l'écriture (en particulier du présent manuscrit) que les réflexions scientifiques. Finalement, je la remercie d'avoir permis et encouragé ma visite à l'université du Texas à Austin, et pour l'école d'été en Italie.

Un grand merci à Warren Hunt pour son accueil chaleureux au sein du groupe ACL2 à Austin, pour ses encouragements et ses "free advices", ils me seront toujours très utiles dans ma vie professionnelle aussi bien que personnelle. Je le remercie, ainsi qu'Anna et leurs enfants, pour m'avoir hébergé lors de mes premiers jours à Austin. Je les remercie pour la table et les chaises, le réveil "américain" qui ne gagne pas une heure pendant la nuit ! Je tiens à remercier Jo O'Neil pour son aide précieuse auprès de la banque, des compagnies d'électricité, d'eau ... Merci à Jo et J pour m'avoir prêté lit, vaisselle, vélo, et autres ustensiles. Mon séjour à Austin fut des plus agréables. Merci aussi à EURODOC et la région Rhône-Alpes.

Un grand merci aux membres de l'équipe VDS. Merci à Katell pour sa motivation, sa disponibilité et son écoute, merci à Pierre pour ses critiques constructives et son humour, merci à Claude et Bénédicte pour leur bonne humeur et leur aide. Enfin, merci à tous les autres membres de l'équipe, passés ou présents, et aux membres du laboratoire TIMA. Merci à Nacer de m'avoir soutenu, surtout lors de mes débuts. Un grand merci aux personnels du laboratoire TIMA et de Polytech'Grenoble, leur bonne humeur et leur aide précieuse, facilitent la vie "administrative" du chercheur !

Un grand merci à tous les membres du groupe ACL2 à l'université du Texas : Erik ("Go Cal !"), Hanbing (souvenirs inoubliables dans les grottes du "rocher enchanté"), Sandip, Jared, David, John, Matt, Serita, et tous ceux que je n'ai pas cités.

Merci à tous mes proches pour leur amitié et leur soutien. Merci à Fred, Yannick, Ben, Séverine, Jock, Nelly, Christophe, Alex, Marie, Coralie, pour les soirées, concerts, et bien

plus encore ! Un merci spécial à Émilie. Merci aux membres du Hockey Club Grenoble, les tournois et les championnats m'ont apporté une oxygénation nécessaire ! Merci à toute l'équipe et longue vie au club ! Une pensée particulière à mes "field hockey girlfriends" du Texas FieldHockey ! Merci de m'avoir accepté dans votre équipe, et de m'avoir permis de pratiquer ce sport outre atlantique. Merci aux membres des diverses associations, notamment APo'G, avec qui j'ai eu plaisir à collaborer.

Enfin, merci à ma mère, mon père, et mes frères pour leur soutien, leurs encouragements. La thèse, et les études en générales, sont des parcours semés d'embuches. Leur présence dans les moments de bonheur ou de doute, m'a permis de prendre des décisions que je ne regrette pas aujourd'hui.

J.S.

Table des matières

Préface

La micro- et la nano-électronique forment un domaine très vaste. Quant il s'agit de fondre des composants sur des plaques de silicium, les acteurs sont les physiciens et les chimistes. Ils inventent des procédés pour graver des transistors de plus en plus petits, de plus en plus nombreux et répartis sur de plus en plus de niveaux. Quant il s'agit d'organiser ces transistors pour réaliser une fonctionnalité précise, les concepteurs "analogiques" et "numériques" mettent en jeu leur compétence en électronique pour concevoir les systèmes embarqués dans les fusées, les avions et les *pacemakers*. Quant il devient nécessaire de *prouver formellement* qu'une conception est conforme aux attentes de ces concepteurs, l'informatique et les mathématiques entrent dans la partie. La simulation numérique exécute un modèle d'une conception sur un nombre restreint des valeurs possibles des entrées. Cette technique permet d'identifier de nombreuses erreurs mais la complexité des systèmes induit un nombre quasi infini d'entrées possibles. La simulation numérique ne peut trouver *toutes* les erreurs, et "oublie" souvent les plus subtiles, les plus fatales. En prouvant formellement la correction d'un circuit, aucune erreur n'est oubliée et ceci pour toutes les entrées et tous les états possibles de ce circuit.

Cette thèse se situe à cette intersection où les mathématiques, l'informatique et la conception des systèmes digitaux se rencontrent. À l'heure actuelle, le premier réseau de communication sur silicium est toujours en développement dans les laboratoires des industriels. Des entreprises - comme STMicroelectronics - ont conçu des modèles d'architectures qu'ils ont décrit dans la littérature et qu'ils simulent numériquement. Ces descriptions informelles, publiées dans des actes de conférences, furent le point de départ des travaux présentés dans ce manuscrit. À partir de ces descriptions, nous avons formalisé un *concept* d'architecture de communication. On peut risquer ici une analogie avec le concept "Arbre". Tout arbre possède des racines, un tronc, des branches et des feuilles. Les racines se situent à la base du tronc qui se ramifie en branches sur lesquelles sont accrochées les feuilles. On peut associer certaines propriétés aux arbres *en général*. Ils grandissent, ils meurent, leurs feuilles changent de couleur en fonction des saisons, *etc.* Dans ces travaux de thèse nous avons cherché à décrire le concept d'architecture de communication par des formules d'une logique mathématique. Nous avons identifié les composantes essentielles communes à *toute* structure d'interconnexion ainsi que les propriétés inhérentes à chacune d'entre elles. Nous avons formalisé l'interaction de ces composantes et exprimé la propriété induite par leurs propriétés essentielles. Aussi, toute interaction entre des composantes particulières satisfaisant les propriétés essentielles de nos composantes abstraites, préserve la propriété vérifiée par l'interaction de ces dernières. La validation de toute architecture particulière est réduite à la preuve que chacune de ses composantes satisfait les propriétés génériques. En décrivant

notre concept dans la logique d'un assistant "automatique" de preuve nous fournissons un outil pour spécifier et valider les descriptions des réseaux sur puce à un haut niveau d'abstraction. Pour toute architecture concrète, l'assistant de preuve génère automatiquement les propriétés qui doivent être satisfaites pour établir la conformité de cette architecture avec notre concept.

Première partie

Conception, vérification et communications

Chapitre 1

Introduction : conception et vérification formelle des systèmes digitaux

Les systèmes sur puces (*SoC* pour *System on Chip*) constituent de nos jours l'enjeu principal de la micro- et de la nano-électronique. Dans nos téléphones portables, dans nos voitures et jusque dans notre propre corps, ces systèmes intègrent une part croissante de notre quotidien. En moins d'un siècle, les ordinateurs, réservés à un public restreint des grandes universités et industries, ont envahi l'univers de tout quidam. En 1936, A. Turing définit un modèle de calculateur universel. En 1946, J. Von Neumann définit l'architecture des ordinateurs actuels. Le premier transistor, élément de base de tout système électronique, est inventé par les américains S. Bardeen, N. Brattain et W. Shockley en 1948. Les circuits intégrés n'apparaîtront que dix ans plutard dans les laboratoires de Texas Instrument, grâce à J. Kilby. Depuis, l'évolution de la microélectronique suit la loi dite de Moore : le nombre de transistors intégrés sur une même puce double tous les dix huit mois. Les technologies actuelles permettent d'intégrer, sur une même surface de silicium, plusieurs centaines de millions de transistors d'une longueur de quelques nanomètres (25 nm).

L'évolution rapide de la microélectronique est parfaitement illustrée par l'histoire d'un composant crucial : le microprocesseur. En 1971, Intel présente le premier "microcalculateur" [1] dénommé 4004 parce qu'il traite quatre bits d'information à la fois. En 1991, la même entreprise fabrique le microprocesseur Pentium 4 avec une architecture à 64 bits. Les quelques chiffres suivants sont illustratifs, les premiers correspondent au 4004 et les seconds au Pentium 4 : le nombre de transistors est passé de 2 300 à plus de 5 millions, le nombre d'instructions par seconde de 60 000 à 1 milliard, le nombre de phases de fabrication de 25 à plus de 200, la superficie d'une usine de tranches de silicium de 1 500 mètres carrés environ à 24 000 (une fois et demie celle d'un terrain de football), le coût de construction et d'équipement d'une usine de 1 million de dollars environ à 1,5 milliards de dollars.

Les progrès des technologies d'intégration sont accompagnés par l'évolution des techniques et outils de conception. En effet, le secteur de la microélectronique est extrêmement concurrentiel, et pour obtenir les marchés le temps de conception est de plus en plus court. Un retard de trois mois entraîne la perte d'un quart des revenus attendus [Bue05]. De plus, la complexité croissante des circuits augmente les sources d'erreurs ; et comme nous le verrons

[1] Le terme de microprocesseur est inventé plus tard.

plus tard, la vérification occupe la grande majorité du temps de conception.

Ce chapitre introductif du contexte et des motivations de cette thèse est structuré comme suit. Tout d'abord, la section suivante présente le flot de conception des systèmes monopuces comme il est principalement admis par les acteurs industriels ou académiques de la microélectronique. Cette section est notamment inspirée par un récent ouvrage édité sous la direction de A.A. Jerraya [Jer02]. Dans un deuxième temps nous présentons un état de l'art sur les méthodes formelles de vérification des architectures matérielles. Nous récapitulons les principes des méthodes actuelles ainsi que les limites et possibilités de chacune par rapport à l'évolution du domaine. Finalement, nous situons nos travaux dans le flot de conception et dans le spectre des méthodes formelles.

1.1 Méthodes de conception des systèmes sur puce

Le flot de conception actuel est divisé en deux processus. Le processus de conception part d'une spécification et produit un masque de transistors. La spécification est souvent un document informel exprimant par des diagrammes, schémas et phrases ce qui est attendu de la puce finale. Cette spécification est affinée progressivement jusqu'au masque de transistors. Le processus de fabrication grave ce masque sur le silicium. Ces deux processus font partie intégrante de la micro- et la nano-électronique. Dans ce manuscrit, nous nous focalisons sur le premier processus et comme nous le verrons par la suite, principalement sur la première phase de celui-ci. La figure 1.1 présente le flot de conception et détaille le processus de conception.

Le processus de conception se divise en cinq étapes. Chaque étape considère un aspect particulier d'un système et définit un niveau d'abstraction.

Le **niveau transistors** est le moins abstrait dans le sens où il considère un grand nombre de paramètres liés à la technologie de fabrication. La plupart des choix sur la structure du système ont été arrêtés. À ce niveau, les concepteurs choisissent une technologie particulière (CMOS, biCMOS, portes dominos, ...) et déterminent comment placer les transistors pour minimiser la surface de silicium (*layout*). Les principes électroniques mis en jeu à ce niveau relèvent du domaine analogique. Le résultat de ce niveau est une bibliothèque de cellules qui consituent les briques utilisées au niveau supérieur. Par exemple, la figure 1.2(a) montre le diagramme d'une porte "NON-ET" (*NAND gate*) à deux entrées en technologie CMOS.

Au **niveau portes**, les circuits sont décrits en un ensemble de portes logiques, comme la porte "NAND" précédente. En fonction du choix de la technologie, les portes seront transformées en un schéma de transistors particulier. À ce niveau, l'électronique est numérique. Les considérations analogiques sont abstraites. Les ensembles de portes sont regroupés en fonctions logiques constitutives d'une bibliothèque pour le niveau supérieur. Par exemple, la figure 1.2(b) montre le diagramme d'une bascule D.

Au **niveau transfert de registres** ou RTL (*Register Transfer Level*) l'enjeu est de décrire le comportement d'un circuit au cours du temps. À ce niveau, les synchronisations (signaux d'horloge et de reset) sont considérées. Le niveau RTL se divise en deux niveaux : le niveau structurel où les composants sont décrits par leur structure, on est frontalier du niveau portes ; le niveau comportemental où les comportements des composants sont décrits

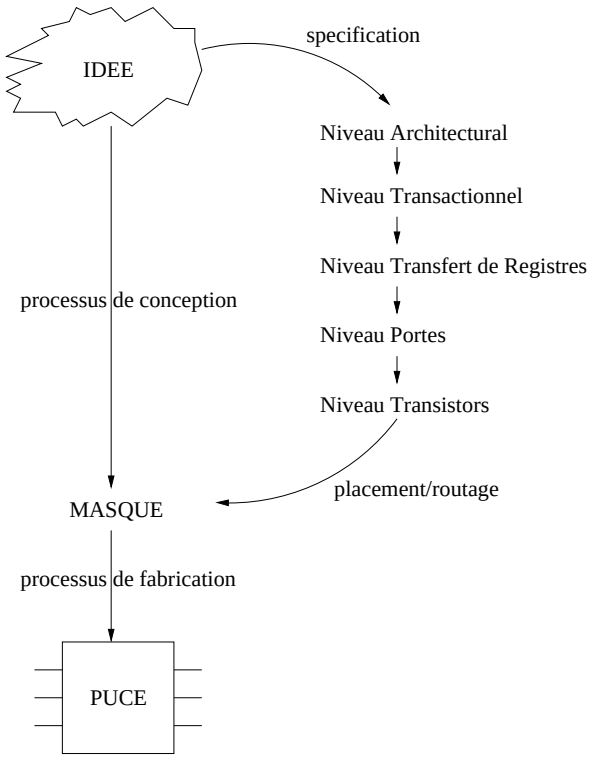
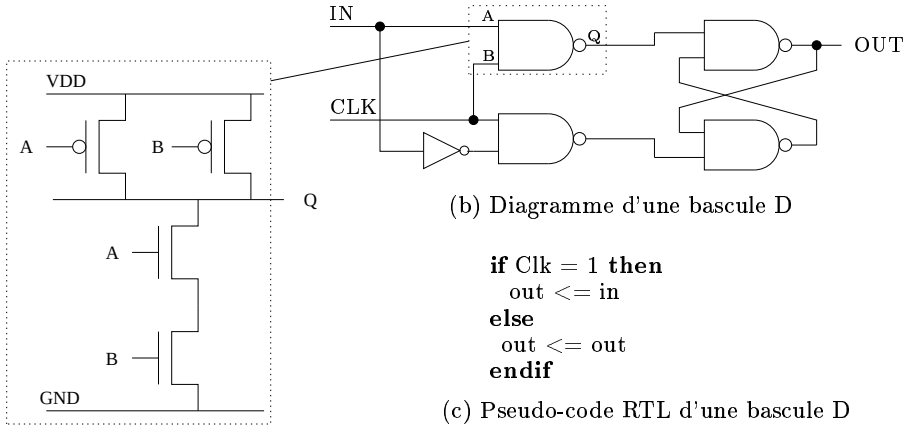


FIG. 1.1 – Niveaux d'abstraction du flot de conception



(a) porte NAND, CMOS

(b) Diagramme d'une bascule D

```
if Clk = 1 then
  out <= in
else
  out <= out
endif
```

(c) Pseudo-code RTL d'une bascule D

FIG. 1.2 – La bascule D et la porte NAND

sans leur donner de structure. La figure 1.2(c) montre un pseudo-code RTL comportemental.

Le **niveau transactionnel** est récent. Les systèmes étant de plus en plus complexes, le temps d'exécution nécessaire à la simulation numérique des descriptions aux niveaux *RTL* et inférieurs explose. L'objectif du niveau transactionnel est d'abstraire une partie des descriptions pour en accélérer la simulation. La principale abstraction concerne les communications. Au niveau *RTL*, les différents composants d'un système communiquent au moyen de signaux. L'outil de simulation propage les bits le long de ces signaux. Au niveau transactionnel, les communications sont abstraites à des appels de fonctions, nécessitant moins de calculs. On parle de *canaux abstraits*. À ce niveau, la préoccupation des concepteurs est la définition et la validation des interfaces entre les composants. Celles-ci sont décrites à un haut niveau d'abstraction puis raffinée jusqu'au niveau *RTL*. La simulation de descriptions précises de ces interfaces est possible grâce à l'abstraction du fonctionnement interne des composants. Les détails de ces calculs ne constituant pas la préoccupation majeure de cette étape de conception, leur ommission permet de réduire le temps de simulation.

Au **niveau architectural**, l'enjeu est de concevoir un système dans son ensemble. On détermine ici les fonctionnalités nécessaires à la réalisation de la spécification initiale. Il s'agit d'assembler des composants préconçus, standards ou spécifiques, implémentant des fonctionnalités particulières (traitement ou contrôle).

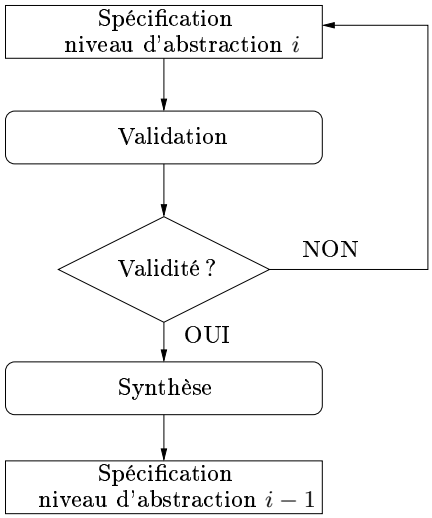


FIG. 1.3 – Méthodologie de conception

Le passage d'un niveau à un autre suit une méthodologie structurée comme le montre la figure 1.3. À partir d'une *spécification* - soit une description à un niveau d'abstraction i donné - le but est de produire une *implémentation* - soit une description à un niveau plus détaillé $i-1$ - qui réalise les comportements spécifiés. Chaque étape est caractérisée par deux actions : la validation et la synthèse. La première consiste dans l'assurance de la conformité de la spécification avec les fonctionnalités désirées et les diverses contraintes de coût et de performance. Cette analyse procède souvent par de nombreuses simulations ou, comme nous le verrons plus tard, par les outils mathématiques de la vérification formelle. Si les contraintes sont satisfaites, une implémentation est - dans la mesure du possible de

manière automatique - générée.

Cette méthode débute au niveau architectural et descend jusqu'au niveau transistors. Le niveau de détails est accru progressivement pour maîtriser la complexité du système à concevoir. Aujourd'hui, les outils industriels permettent le passage automatique et successif du niveau RTL au niveau transistors. La decription RTL est obtenue manuellement. Un des enjeux du domaine est de fournir des méthodes et des outils pour obtenir automatiquement cette description RTL à partir des niveaux plus abstraits [Spi04].

1.2 Les méthodes formelles

L'obtention d'un circuit exempt d'erreur dans le délai le plus bref est la préoccupation majeure des industriels de la microelectronique. Actuellement, de 60 à 70 % du temps de conception est consacré à la vérification et le nombre d'ingénieurs en charge de cette dernière est au moins deux fois supérieur au nombre de concepteurs [Nob02].

Aujourd'hui, pour en vérifier le bon fonctionnement, un système est testé sur un grand nombre des valeurs possibles de ses entrées. Mais la complexité des systèmes actuels - d'un processeur Pentium par exemple - rend impossible une simulation exhaustive et donc l'assurance d'une absence d'erreur. Par ailleurs, le nombre de stimuli nécessaire pour couvrir dans un pourcentage acceptable les erreurs probables d'une conception dépasse les capacités des simulateurs actuels.

Au milieu des années quatre-vingt dix, le processeur Pentium d'Intel effectue un calcul erronné sur son instruction de division flottante [Mar94]. La compagnie doit rappeler ces machines. Le coût se chiffre à plus de 400 millions de dollars. La méthode reine pour la vérification des conceptions est la simulation numérique. Aujourd'hui, le coût d'un tel bug serait fatal à l'entreprise.

Les méthodes formelles proposent des solutions complémentaires et efficaces pour la validation des systèmes digitaux. Si elles apparaissent aujourd'hui comme une nécessité, la notion de preuve de circuits habitait les pionners de l'informatique. A. Turing publiait à la fin des années quarante [Tur49] la preuve d'une *large routine*, en l'occurence un circuit réalisant la factorielle par additions successives. Il écrit :

> "How can one check a large routine in the sense that it's right ?
> In order than the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows."

Turing considère ici une vérification *humaine*. La *revue de code* consiste dans la relecture de programmes par des personnes autres que leurs auteurs. Cette technique constitue encore aujourd'hui un moyen efficace de vérification [Bue05].

Au début des années soixantes, J. McCarthy [McC62] formulait les principes de la vérification formelle *automatique* :

> "Instead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program."

Ce concept de "machines raisonnant sur des machines" fut un des moteurs pour le développement des méthodes formelles. Dans cette section, nous présentons les principales

techniques de vérification actuelles : la vérification de modèles (*model checking*), la démonstration de théorèmes (*theorem proving*) et la simulation symbolique. De plus amples détails sont disponibles dans de précédents ouvrages [CW96, CF98, Ber02].

1.2.1 Vérification de modèles

La vérification de modèles (en anglais *model checking*), a été proposée de manière indépendente par Clarke et Emerson [CE81] et par Queille et Sifakis [QS82]. Les modèles des systèmes à vérifier sont généralement des machines d'états finis (chap. 5 de [Jer02]). Une machine d'états finis est définie par un quintuplet $(S, I, O, \delta, \lambda)$ où :

- S est l'ensemble des états. On identifie un état particulier comme étant l'état initial s_0.
- I est l'ensemble des signaux d'entrée.
- O est l'ensemble de signaux de sortie.
- $\delta : S \times I \rightarrow O$ est une fonction calculant les nouvelles valeurs des sorties en fonction de l'état courant et des entrées.
- $\lambda : S \times I \rightarrow S$ est une fonction calculant le nouvel état en fonction de l'état courant et des entrées. Cette fonction est dite *fonction de transition*.

Les algorithmes de *model checking* utilisent la fonction de transition pour parcourir l'espace des états atteignables. Les algorithmes décident si une propriété est toujours ou inévitablement vraie sur un chemin d'exécution.

Les propriétés expriment les comportements attendus d'un système et sont classées dans deux catégorie : la sûreté et la vivacité. La première consiste en la vérification que *quelque chose de mauvais n'arrivera jamais*, la seconde que *quelque chose de bon se produira inévitablement*. Les propriétés *un système n'atteindra jamais un état d'où il ne peut plus sortir* et les *exclusions mutuelles* sont des propriétés de sûreté. La propriété spécifiant que *toute requête recevra nécessairement une réponse* constitue une propriété de vivacité.

Les propriétés sont exprimées dans une logique temporelle linéaire (*LTL* pour *Linear Temporal Logic*) [Pnu77, BAPM83] ou arborescente (*CTL* pour *Computation Tree Logic*) [CES86]. Ces logiques sont constituées d'une logique classique du premier ordre à laquelle ont été ajoutées des modalités temporelles. Ce dernier n'est pas référencé explicitement (en secondes) mais implicitement par les successions d'états constitutifs d'un chemin d'exécution.

Les deux opérateurs principaux des logiques linéaires sont l'opérateur *next* ("au prochain coup"), noté $\bigcirc$, et l'opérateur *until* ("jusqu'à"), noté $\mathcal{U}$. La formule $\bigcirc\varphi$ exprime le fait que φ est vrai au prochain état atteignable. La formule $\varphi \,\mathcal{U}\, \psi$ exprime le fait que φ est vrai jusqu'à ce que ψ soit vrai. La propriété "vrai jusqu'à ce que ϕ", définit l'opérateur *inévitablement*, noté $\diamond$.

$$\diamond\varphi = true \,\mathcal{U}\varphi$$

La formule $\diamond\varphi$ exprime le fait que la propriété φ est inévitablement satisfaite dans un état futur.

La négation de cet opérateur défini l'opérateur *toujours*, noté $\Box$.

$$\Box\varphi = \neg \diamond \neg\varphi$$

La formule $\Box\varphi$ exprime le fait que φ est vrai pour tous les états atteignables à partir de l'état initial s_0.

L'opérateur *toujours* est utilisé pour exprimer des propriétés de sûreté, l'opérateur *inévitablement* pour des propriétés de vivacité.

Les logiques arborescentes permettent d'exprimer la potentialité des comportements, ce qui n'est pas possible dans le cas linéaire. Les logiques arborescentes reposent sur les logiques linéaires. Chaque "branche" représente un comportement linéaire et il est possible de quantifier sur ces branches. La quantification universelle ($\forall$) prend en compte toutes les branches. La quantification existentielle ($\exists$) spécifie l'existance d'une branche satisfaisant un comportement linéaire particulier. La propriété $\forall \bigcirc \varphi$ exprime que la propriété φ est satisfaite dans le prochain état de toutes les branches. La propriété $\exists(\varphi \mathcal{U} \psi)$ exprime le fait qu'il existe une branche pour laquelle φ est satisfaite jusqu'à ce que ψ le soit. La propriété $\forall\Box\varphi$ exprime le fait que la propriété φ est *toujours* satisfaite sur *toute* les branches. Elle exprime une propriété de sûreté. La propriété $\forall \diamond \varphi$ exprime le fait que la propriété φ est *inévitablement* satisfaite sur *toutes* les branches. Elle exprime une propriété de vivacité.

Les logiques linéaires et arborescentes ont été regroupées dans une logique plus expressive, la logique $CTL*$ (voir le livre de Clarke *et al.* [CGP99] pour une présentation détaillée des algorithmes, des différentes logiques et leur relation).

Un atout des vérificateurs de modèles est leur capacité à produire des contre-exemples. Pour prouver qu'une formule temporelle ϕ (linéaire ou arborescente) est satisfaite par un modèle M ($M \vDash \phi$), un vérificateur de modèles va chercher à prouver que M ne satisfait pas la négation de ϕ ($M \nvDash \neg\phi$). Par exemple, pour prouver $\forall\Box f$, le vérificateur va chercher à construire la preuve de $\neg(\forall\Box f)$, soit $\exists \diamond \neg f$. Si cette construction aboutit, elle constitue un contre-exemple pour $\forall\Box f$. Les contre-exemples sont aujourd'hui utilisés pour générer des vecteurs de test (chapitre 5 de [Jer02]).

La principale limitation de la vérification de modèles provient de l'explosion combinatoire : le nombre d'états croît de manière exponentielle avec la taille des registres du modèle à vérifier. Une solution à ce problème est d'utiliser les diagrammes de décision binaire [Bry86] pour représenter de façon symbolique un ensemble d'état [McM93]. D'autres solutions cherchent à réduire, de façon automatique ou manuelle, les modèles à vérifier. La réduction d'une formule à son *cône d'influence* restreint le modèle à vérifier aux parties concernées par la propriété [BCC98]. D'autres techniques réduisent le modèle en utilisant des symmétries [CFJ93, ES96]. Une technique très employée est celle dite "vous supposez, nous garantissons" (*you assume, we guarantee*) [HQR98]. Intuitivement, l'utilisateur pose comme hypothèse une série de contraintes sur l'environnement d'un système. Les algorithmes de vérification garantissent la satisfaction de propriétés sous ces contraintes. Finalement, la partition fonctionnelle [Dum03] amène un système dans un mode de fonctionnement particulier au moyen d'une étape de simulation symbolique. Le parcours d'états est effectué à partir de ce mode spécifique.

Dans le domaine des techniques algorithmiques de vérification, le cas des systèmes paramétrés est connu sous le nom de *problème de la vérification de modèles paramétrés* (en anglais, *Parameterized Model Checking Problem*, soit PMCP). Ce problème consiste dans la vérification de propriétés temporelles pour toutes les valeurs possibles des paramètres, *e.g.* la taille du système. Si le PMCP est indécidable en général [AK86], il est décidable pour

des classes restreintes de systèmes [GS92]. Pour ces classes de circuits, diverses méthodes plus ou moins automatiques ont été développées. Pour les systèmes synchrones constitués d'un processus de contrôle et d'un nombre arbitraire de processus utilisateurs identiques, Emerson et Namjoshi [EN96] montrent qu'il existe des algorithmes pouvant décider - sous certaines hypothèses de symétrie - des propriétés exprimées :

- sur le processus de contrôle (de la forme $\forall f$ ou $\exists f$, où f est une formule temporelle linéaire exprimée sur les états du processus de contrôle),
- sur les processus utilisateurs (de la forme $\bigwedge_i \forall f(i)$ ou $\bigwedge_i \exists f(i)$, où $f(i)$ est une formule linéaire exprimée sur les états du processus de contrôle et sur les variables du processus utilisateur indexé par i),
- sur des paires de processus utilisateurs distincts (de la forme $\bigwedge_{i \neq j} \forall f(i,j)$ ou $\bigwedge_{i \neq j} \exists f(i,j)$, où $f(i,j)$ est une formule linéaire exprimée sur les états du processus de contrôle et sur les variables du processus utilisateur indexé par i ou j).

La vérification de systèmes paramétrés au moyen de la technique de vérification de modèles est possible [HLR92, CGJ97, EK03]. Néanmoins, chaque solution proposée s'applique à une famille de problèmes particulière, souvent restreinte.

Aujourd'hui, la vérification de modèles (non paramétrés) s'est imposée dans le quotidien de nombreuses entreprises grâce à son niveau d'automatisation. En effet, des outils industriels - produits par les entreprises phares de la microélectronique (Cadence, Synopsis, IBM, Intel, ...) implémentent les algorithmes de *model checking*. Les optimisations des algorithmes, l'introduction de techniques de satisfaisabilité (SAT) et bien évidemment l'augmentation des performances des machines ont permis à la vérification de modèles de traiter des circuits et surtout des sous-circuits réalistes. Les outils et techniques de *model checking* s'appliquent aujourd'hui principalement au niveau transfert de registres (RTL) et au niveau portes.

1.2.2 Démonstration de théorèmes

Les logiques temporelles et les algorithmes de vérification de modèles permettent d'exprimer et de vérifier des propriétés de manière efficace sur les parties contrôles des circuits. La démonstration de théorèmes est une technique mieux adaptée pour s'attaquer aux aspects fonctionnels des systèmes, c'est à dire exprimer et vérifier des propriétés sur le traitement des données. Ces deux techniques sont souvent associées. Par exemple, on peut appliquer la vérification de modèles à une abstraction du circuit. L'espace d'état de l'abstraction est moins large que celui du circuit initial. La démonstration de théorèmes est utilisée pour établir une relation entre le circuit et une abstraction de ce circuit. Cette relation permet de déduire que les propriétés établies sur l'abstraction sont satisfaites par le circuit initial (voir par exemple [MNS99]).

La démonstration de théorèmes est une technique où le système et les propriétés sont décrits par des formules mathématiques d'une logique donnée. Cette dernière est constituée d'axiomes et de règles d'inférences. La preuve d'une formule est la dérivation de celle-ci par application des règles à partir des axiomes. Les *assistants de preuve* ou "démonstrateurs" de théorèmes ont pour objet l'automatisation des raisonnements - recherche d'une preuve - ou de la vérification d'une preuve dans une logique donnée. Ces outils se différencient selon leur degré d'automatisation et l'ordre de la logique employée. Aujourd'hui, aucun outil n'est

complètement automatique, mais le degré de maturité est tout de même suffisant pour traiter des problèmes complexes.

Le premier assistant de preuve appliqué à des problèmes réalistes est le système ACL2, développé par Moore et Kaufmann à partir du système *Nqthm*. Ces deux outils sont fondés sur la logique du premier ordre de Boyer-Moore [BM88]. À la suite du "bug" du Pentium d'Intel, AMD fait appel à Moore et son équipe pour vérifier formellement que leur opération de division flottante est correcte. À l'aide de ACL2, Moore et son équipe prouvent un théorème montrant que l'algorithme d'AMD est équivalent au standard IEEE [MLK98]. Ces résultats seront appliqués par Russinoff à d'autres opérateurs flottants [Rus98]. L'outil ACL2 [KMM00a] conserve les principes fondamentaux de *Nqthm* et intègre de nouvelles possibilités, notamment une vitesse d'exécution des modèles de la logique proche de celle de programmes C. Cette efficacité est propre à ACL2 et constitue un atout important [WGH01].

D'autres assistants de preuve existent aujourd'hui. Gordon a developpé l'outil HOL [Gor87] pour des logiques d'ordre supérieur. Dans ces logiques d'ordre supérieur, les démonstrateurs les plus connus sont Isabelle [Pau89, Pau90, NPW02], PVS [ORS92, SOSC01] et Coq [Coq]. Les applications des différents démonstrateurs sont diverses : vérification de microprocesseurs (en utilisant ACL2 [Hun89, Saw99], HOL [Fox03] ou PVS [BJK+03]), preuve d'opérateurs flottants (en utilisant ACL2 [SG02, Rus98] ou PVS [BJ01]), vérification de protocoles (en utilisant ACL2 [Moo93], PVS [PMMG04] ou Isabelle [Pau98, Bel03]), ...

Au contraire des vérificateurs de modèles, les assistants de preuve peuvent - au moyen de raisonnements par induction - traiter directement des systèmes avec un nombre d'états non borné, voire infini. Néanmoins, ces outils nécessitent une intervention humaine importante dans la recherche des preuves. Les techniques de *theorem proving* s'appliquent sur un spectre large allant du niveau portes au niveaux les plus abstraits. Au niveau RTL et aux niveaux inférieurs, les techniques algorithmiques sont automatiques ; leur utilisation est plus répandue. Par contre, la puissance de raisonnement font des assistants de preuves les outils les plus prometteurs lorsque des niveaux plus abstraits sont considérés.

1.2.3 Simulation Symbolique

Le principe de la *simulation symbolique* est d'introduire des symboles aux entrées d'un circuit au lieu de valeurs numériques. Cette technique a été proposée pour la vérification de circuits en 1979 par Darringer [Dar79]. À l'époque, elle était limitée à des circuits de faible taille pour les raisons suivantes :

1. l'expression obtenue croît exponentiellement en fonction de nombre de cycles de simulation.
2. En présence de branchements conditionnels, les deux branches doivent être simulées. Le simulateur construit un arbre d'exécution qui peut lui aussi croître de manière exponentielle.
3. Sans l'aide d'une simplification ou d'une réduction automatique, le résultat de la simulation symbolique est trop grand pour être "humainement" compréhensible.

Depuis 1979, des travaux de recherches se sont attaqués à une ou plusieurs de ces raisons. Au niveau portes, les travaux de Bryant et Seger ont donné lieu à la technique STE (en

Anglais *Symbolic Trajectory Evaluation*) [BS95]. STE peut être considérée comme une technique de vérification de modèles particulière à une classe de formules : les états (représentés par des BDD's) sont abstraits dans une logique à trois valeurs (0, 1 ou X) et des symboles sont utilisés pour représenter une partie des entrées ou des états initiaux. L'inclusion de calculs de satisfaisabilité (SAT) a permis de généraliser la technique STE et d'en accroître les performances [YS02]. À partir de ces recherches, la société Innologic a commercialisé un simulateur symbolique capable de traiter des circuits réalistes décrits au niveau portes.

Ces techniques ne sont pas applicables à des niveaux d'abstraction élevés où une spécification fait appel à des types de données abstraits au lieu de leur codage booléen. Pour simplifier les expressions algébriques ou contrôler l'expansion de l'arbre d'exécution, la plupart des solutions proposées reposent sur l'utilisation d'outils de preuves. Les outils PVS et ACL2 ont été utilisés pour simuler symboliquement le processeur Java JEM1 développé par Rockwell Collins [Gre98, WGH01]. Une technique systématique pour utiliser ACL2 comme un simulateur symbolique a été proposée par J Strother Moore [Moo98]. Reposant sur cette technique, une thèse de l'équipe VDS du laboratoire TIMA a défini la sémantique de simulation d'un sous-ensemble de VHDL dans ACL2 afin de permettre la simulation symbolique de circuits décrits dans ce langage [BGR01, Geo01].

Une nouvelle approche [STSB03, SBO+03] de simulation symbolique a été développée au sein de l'équipe VDS. Cette approche allie le logiciel de calculs symboliques Mathematica et l'outil de preuves ACL2. Le principe de cette nouvelle méthode est de séparer la partie calcul de la partie raisonnement. Les simplifications sont effectuée par Mathematica et ACL2 est utilisé pour affirmer ou infirmer des conditions de branchement. Ces preuves sont effectuées sous certaines contraintes : soit déduites des informations sur les types de données, soit fournies par un utilisateur. Cette technique, baptisée *simulation symbolique contrainte*, est le support théorique d'un simulateur symbolique développé par l'équipe VDS et nommé *TheoSim* [SST+04, Sam05].

1.3 Positionnement et contribution

Positionnement. Le flot de conception et le flot de vérification des architectures matérielles sont résumés sur la figure 1.4. Le point de départ est une spécification informelle qui exprime par des phrases, des diagrammes et des schémas, les fonctionnalités et les comportements attendus d'un système. Comme nous l'avons écrit en section 1.1 de ce chapitre, ce document initial est raffiné manuellement jusqu'à obtenir une description au niveau transfert de registres.

Toute application de méthodes formelles nécessite un modèle mathématique du circuit. Il s'agit le plus souvent d'une machine d'états finis. Ce modèle est extrait automatiquement à partir de code écrit dans des langages VHDL ou Verilog. Le but de la vérification est de prouver la conformité entre la description RTL et sa spécification informelle. Une description formelle de cette dernière est indispensable. Elle est obtenue manuellement. Si la preuve s'effectue par vérification de modèles, la spécification formelle est un ensemble de propriétés temporelles. La description RTL est conforme si son modèle formel - soit la machine d'états finis - satisfait cet ensemble de propriétés. Dans ce contexte, le langage PSL [Acc]

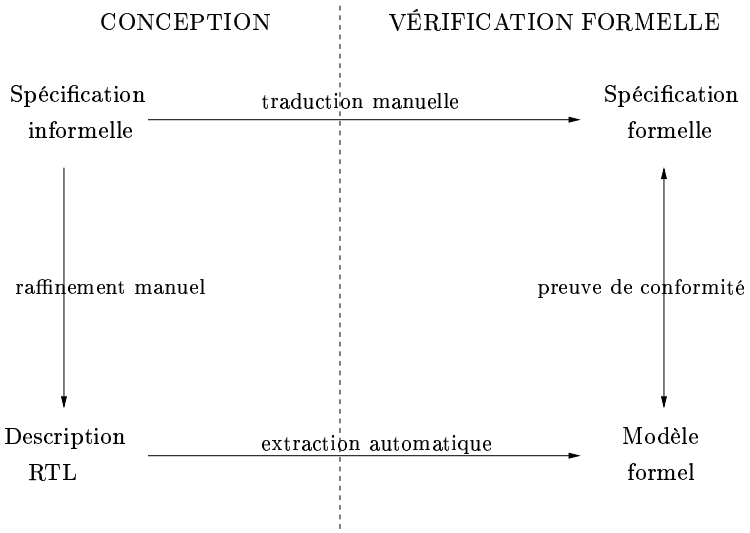


FIG. 1.4 – Flot de conception et de vérification

facilite l'écriture de propriétés complexes. Lorsque la vérification est effectuée par preuve de théorèmes, la spécification et la représentation formelle du code RTL sont tout deux des ensembles de fonctions dans une logique donnée. La conformité est exprimée par un théorème qui montre que les résultats produits par les deux ensembles de fonctions sont équivalents. L'écriture et la validation de spécifications complexes demande une connaissance approfondie des logiques, des langages et des fonctionnements internes des outils de preuves. Les langages d'entrée de ces systèmes sont peu connus des concepteurs et seuls de puissants industriels - comme Intel ou IBM - ou des industriels de domaines sensibles (Rockwell-Collins dans le domaine militaire, par exemple) peuvent s'offrir les services d'experts en démonstration de théorèmes.

Dans cette thèse, nous considérons la première étape d'une conception lorsqu'il s'agit de spécifier les principaux concepts et algorithmes d'un système. Au niveau d'abstraction considéré, les enjeux principaux sont les interactions entre les différents composants - qui sont généralement numériques et analogiques - et les interconnexions [Spi04]. Dans ce manuscrit, seules les interconnexions sont abordées.

La description initiale d'un système est indépendante de paramètres architecturaux - la topologie d'un réseau ou la taille des mémoires, par exemple - qui seront déterminés plus tardivement. Cette description est abstraite et l'espace d'état est trop large pour les algorithmes de vérification de modèles. La technique la mieux adaptée est la démonstration de théorèmes.

Dans nos recherches, nous avons choisi l'outil de preuves ACL2 pour les raisons suivantes.

Contrairement à la plupart des assistants de preuve, la logique ACL2 est exécutable. Il est possible de simuler les modèles sur des jeux d'essais numériques permettant ainsi un déboggage rapide. De plus, ACL2 propose des mécanismes permettant d'optimiser cette exécution jusqu'à une vitesse proche de celle d'un programme C. Cette efficacité est un réel atout d'ACL2 sur ses concurrents [WGH01]. En effet, pour vérifier une description VHDL ou Verilog, par exemple, celle-ci est traduite - manuellement ou de manière automatique - dans

la syntaxe ACL2. Cette traduction peut introduire des erreurs, et l'outil effectuant la transformation n'est la plupart du temps pas "prouvé correct". La "solution ACL2" est d'exécuter les deux modèles sur les même jeux d'essais et de vérifier la conformité des signatures entre les modèles.

L'apprentissage de l'utilisation d'un assistant de preuve est longue et difficile. Pour être efficace, la connaissance des mécanismes internes de l'outil est nécessaire. Les contacts solides entre l'équipe VDS et l'équipe texane où ACL2 est développé constituent une justification importante dans le choix de l'outil.

Contribution. Cette thèse considère l'expression de concepts informels dans la logique d'un démonstrateur de théorèmes. Notre principale contribution est la définition d'un modèle générique des communications. Nous avons identifié les principales composantes d'une architecture de communication ainsi que les propriétés essentielles de chacune d'entres elles. Chaque composante est formalisée par une fonction "contrainte" de satisfaire les propriétés essentielles. À partir de ces composantes, nous avons formalisé leur interaction par une fonction qui représente *toute* architecture de communication. Cette fonction, nommée *GeNoC* (pour *Generic Network on Chip*), est validée par la preuve d'un théorème montrant que chaque message émis sur l'architecture atteint la destination attendue sans modification de l'information qu'il transporte. La preuve de ce théorème est directement déduite des propriétés essentielles de chacune des composantes. La spécification d'une architecture de communication particulière revient à donner une définition *explicite* à chacune des composantes et à prouver que ces définitions satisfont les propriétés essentielles. En décrivant notre modèle dans la logique d'un démonstrateur "automatique" de preuve nous fournissons un outil pour spécifier et valider les descriptions des réseaux sur puce à un haut niveau d'abstraction. Pour toute architecture concrète, le démonstrateur génère automatiquement les propriétés qui doivent être satisfaites pour établir la conformité de cette architecture avec notre modèle.

Cette première partie se termine par un chapitre introduisant les principaux concepts relatifs aux architectures de communications. Ces principes sont applicables aux architectures de communication en général. Nous les illustrons sur quelques exemples de structures déployées sur la puce.

La deuxième partie est le coeur de ce manuscrit. Elle présente notre formalisation, sa réalisation dans la logique d'un démonstrateur de théorèmes et une série de diverses concrétisations de notre *concept* d'architecture de communication. Dans le chapitre trois, nous formalisons dans le langage des mathématiques la fonction *GeNoC* représentant une architecture générique. Nous en identifions les fonctions essentielles et leurs propriétés. Nous définissons et prouvons à partir de ces composantes essentielles la correction de la fonction *GeNoC*. Cette formalisation est ensuite exprimée dans la logique du démonstrateur ACL2. Notre formalisation repose sur de nombreux quantificateurs et des constructions au second ordre. La logique ACL2 est du premier ordre et nous montrons comment utiliser les mécanismes de ce système pour simuler la logique du second ordre. Finalement, nous montrons différentes concrétisations de notre architecture générique au chapitre cinq. Ces illustrations sont constituées d'architectures conçues par des industriels et d'autres plus académiques.

La dernière partie présente nos conclusions et les prolongements possibles des recherches menées durant ces travaux de thèse.

Chapitre 2

Les communications sur la puce

Ce chapitre introduit les notions élémentaires relatives aux architectures de communication qu'elles soient sur la puce ou pas. Nous rappelons brièvement les notions de protocoles et de topologies. Nous détaillons ensuite les principes liés aux architectures de type bus et aux réseaux d'interconnexion. Nous illustrons ces principes sur des architectures déployées sur la puce. Pour les réseaux nous abordons les théories assurant la prévention des interblocages. Finalement, nous exposons les travaux consacrés à l'application des méthodes formelles relative aux systèmes d'interconnexion, et plus particulièrement à ceux utilisés dans les systèmes sur puce.

2.1 Introduction : protocoles et topologies

Le *medium de transmission*, ou simplement *medium*, est le moyen physique sur lequel est transmis l'information. Un *protocole* est l'ensemble des mécanismes régissant la construction et la lecture de la séquence de signaux transmise sur le medium. Cette séquence est une *trame*. Les protocoles ont été structurés en sept couches définies par un standard international OSI. La figure 2.1 représente deux exemplaires des couches OSI [DZ83] reliés par un medium et échangeant une trame. Pour transmettre un message donné, chaque couche ajoute une information - une *entête* - au message. À la réception de la trame, chaque couche décode l'entête correspondante. Une trame est constituée des différentes entêtes, du message et d'un indicateur de fin de trame ou d'un code permettant de détecter d'éventuelles altérations du contenu de la trame. Nous rappelons brièvement les tâches des différentes couches.

La **couche physique** (couche 1) réalise l'interface entre le medium et la machine. Elle lit et émet les trames bit par bit.

La **couche liaison de données** (couche 2) gère les liaisons entre les différentes unités d'un système. Elle établit et maintient les connexions. Elle contrôle la longueur des trames ainsi que leur débit.

La **couche réseau** (couche 3) est responsable de l'acheminement du contenu du message. Dans la plupart des cas, cette couche s'occupe de la segmentation du message en paquets (*c.f.* section 2.3) lors de l'émission, et de la reconstruction du message lors de la réception.

La **couche transport** (couche 4) garantit la qualité de service du système. Elle assure que le message est correctement transmis aux destinataires.

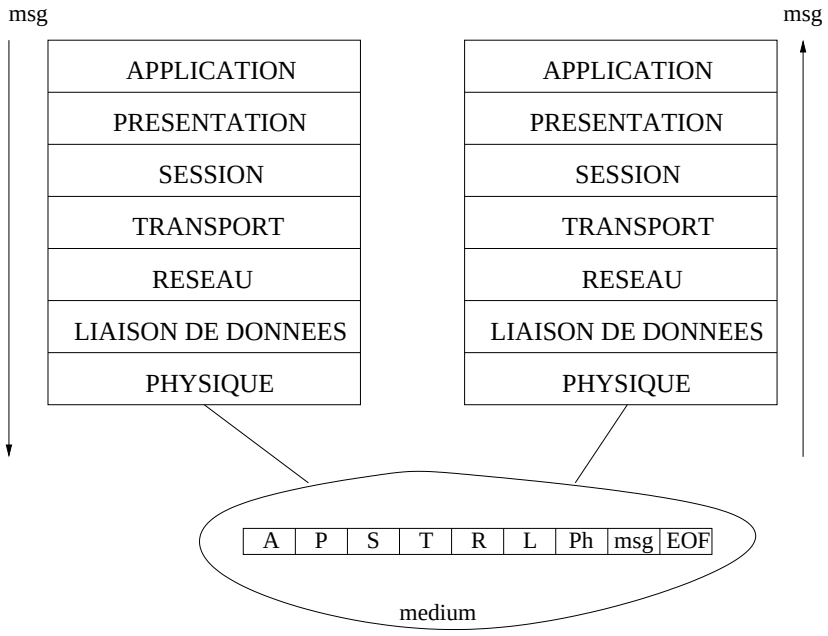


FIG. 2.1 – Couches du système OSI et échange d'un message entre deux unités

La **couche session** (couche 5) organise et synchronise les différents processus communicants. Elle orchestre le dialogue entre ces différents processus.

La **couche présentation** (couche 6) formate le message initial pour le rendre compréhensible par les différents acteurs en cours de communication. Généralement, elle crypte, décrypte, compresse, décompresse, etc ...

La **couche application** (couche 7) réalise le lien avec l'utilisateur. Elle offre les services essentiels du réseau, comme le transfert de fichiers, la messagerie, etc ...

La réalisation de ces sept couches forme une *unité de communication*. Selon la méthodologie proposée par Rowson et Sangiovanni-Vincentelli [RSV97], chaque unité peut être divisée en une partie "interface" et une partie "application". L'application regroupe les couches quatre à sept du modèle OSI, et les interfaces les couches une à quatre. La couche quatre étant une couche charnière, elle s'intègre soit dans la partie interface, soit dans la partie application. Une *communication* est l'échange d'une trame entre plusieurs unités. L'interconnexion de plusieurs unités constitue un *système d'interconnexion*. L'organisation des interconnexions entre les unités définit la *topologie* du système. Deux systèmes sont connectés au moyen d'un *pont*. Un nœud est l'entité visible d'un système. Si un système n'est connecté à aucun autre, un nœud et une unité désigne le même objet. Si ce système est connecté à d'autres systèmes, au niveau de l'interconnexion des systèmes, un nœud désigne un système. La figure 2.2 montre l'interconnexion de trois systèmes comportant chacun une unité et un pont. Chaque système constitue un nœud. Si le pont et les unités avec lesquelles il interagit appartiennent à un même nœud, le pont est équivalent à une unité. Les communications intervenants entre les unités d'un nœud sont dites *locales*. Les communications entre les unités de nœuds différents sont dites *distantes*.

Pour les systèmes sur puce, les topologies se différencient en deux grands groupes : les

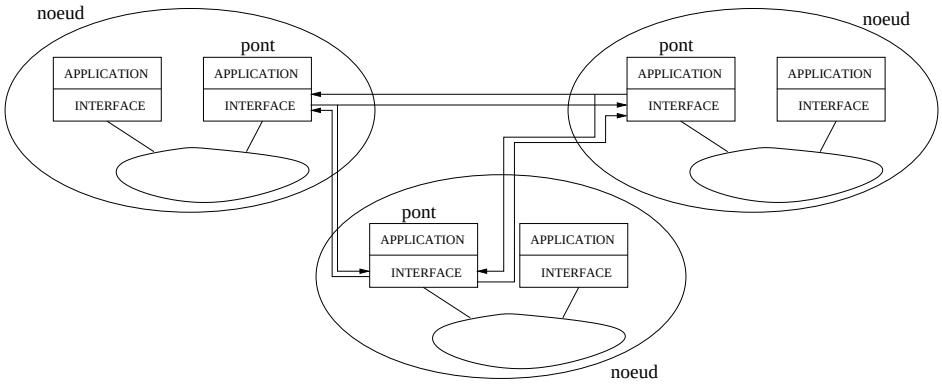


FIG. 2.2 – Interconnexion de trois systèmes

bus et les réseaux. La section suivante s'intéresse aux architectures de type bus. Le reste du chapitre est consacré aux réseaux.

2.2 Les bus

Le principe du bus est de connecter plusieurs unités autour d'un unique medium. Cette section présente les bus comme ils sont actuellement utilisés dans les systèmes sur puce.

2.2.1 Les topologies des bus

La principale topologie utilisée dans les systèmes embarqués est le bus *partagé* (*c.f.* figure 2.3(a)). Toutes les unités communiquent par l'intermédiaire d'un unique medium de transmission. Quand le nombre d'unités devient important, le système est décomposé en plusieurs bus. On peut ainsi hiérarchiser les communications, en fonction du traffic par exemple. L'inconvénient majeur des bus partagés est leur faible parallélisme. Le bus devient vite saturé. Cette architecture n'est pas facilement extensible.

Pour améliorer les performances, plusieurs unités ou bus peuvent être connectés au moyen d'une *matrice d'interconnexion*. Cette dernière établit une connexion entre différents niveaux en fonction des besoins. Par exemple (*cf.* Fig. 2.3(b)), la matrice peut connecter le nœud a au nœud d et le nœud b au nœud c dans un même temps. Le niveau de parallélisme est accru. La migration d'une architecture partagée vers une architecture matricielle est effectuée en remplaçant l'unique medium par une matrice. Ainsi, la bande passante d'un système augmente tout en conservant les unités et les protocoles conçus pour les architectures partagées.

Les bus sont aujourd'hui les architectures les plus utilisées par les industriels. Les plus célèbres sont les bus AMBA (AHB et APB) de ARM, le bus CoreConnect de IBM et le ST-Bus de STMicroelectronics. ARM propose une version hiérarchisée du bus AHB. De plus, les industries se sont regroupées pour définir des protocoles standards. Les groupes les plus actifs sont VSI Alliance et ses protocoles VCI, OCP-IP et plus récemment le consortium FlexRay concernant les systèmes embarqués, en particulier ceux incorporés dans les automobiles.

(a) bus partagé (b) bus matriciel

FIG. 2.3 – Topologies partagées et matricielles des bus

2.2.2 Arbitrage des accès

Généralement, les protocoles utilisés sur les bus sont de type maître/esclave. Seul un maître peut initier une communication. Les unités esclaves ne peuvent que répondre aux requêtes des maîtres. Parce que le medium est unique, l'accès au bus est alloué de manière exclusive à un seul maître. L'arbitre de bus est le composant chargé de cette règlementation. Les politiques d'attribution sont diverses : systèmes de priorités fixes, politique du "tourniquet" (*round-robin*), attribution aléatoire, etc ...

Les principales propriétés que tout arbitre doit vérifier sont l'exclusion mutuelle des accès au bus et l'absence de famine, *i.e.* toute requête d'accès est invévitablement satisfaite.

Pour illustrer les architectures de type bus, l'architecture AMBA AHB est brièvement présentée.

2.2.3 Le bus AMBA AHB

Le système AHB (pour Advanced High-performance Bus) comporte trois bus : HADDR pour les adresses, HWDATA pour l'écriture en mémoire, initiée par les maîtres, et HRDATA pour la lecture, par les esclaves, des données. Ce procédé fournit une bande passante importante pour les échanges d'informations sur une structure pouvant supporter jusqu'à 16 processeurs. Une architecture AMBA-AHB se compose, comme le montre la figure 2.4, des éléments suivants :

- un décodeur d'adresse : il reçoit une adresse et active l'unité désignée par celle-ci au moyen du signal HSEL,
- un arbitre : il reçoit les demandes d'accès au bus d'adresses des maîtres par le signal HREQ et accorde l'accès à l'un d'entre eux par le signal HGRANT,
- des esclaves : répondent aux demandes des maîtres,
- des maîtres : initient des échanges d'information avec les esclaves,

Les maîtres sont dans la plupart des cas des processeurs et les esclaves des mémoires.

Le protocole de communication du bus est du type Maître/Esclave et l'unité qui dirige le transfert est dite maître par opposition à l'unité Esclave qui se contente de fournir les informations demandées. Une communication se déroule comme suit :

- le maître demande à l'arbitre l'accès aux bus d'adresse et d'écriture,
- si le maître a accès aux bus, il envoie l'adresse qu'il souhaite lire ou écrire aux autres unités et au décodeur (phase d'adressage),
- le décodeur active l'unité ayant accès à l'adresse souhaitée,

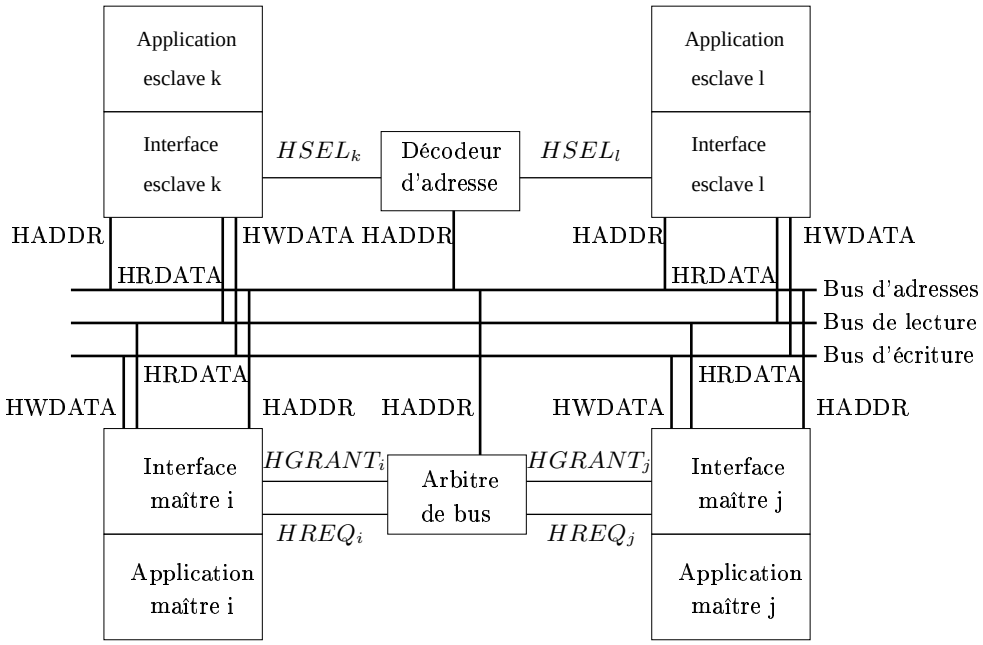


FIG. 2.4 – Synoptique d'une architecture AMBA AHB

– l'esclave répond à la demande du maître (phase des données),

Ces opérations sont pipelinées et la phase d'adressage d'un transfert s'effectue pendant la phase des données du précédent. De plus, l'esclave a la possibilité de libérer le bus et de demander le découpage du transfert. Cette fonction de *split* permet à un esclave d'être disponible pour un autre maître : un esclave peut avoir plusieurs maîtres.

La société ARM propose une version matricielle du bus AHB [ARM01]. La figure 2.5 représente une matrice avec trois maîtres et quatre esclave. Dans cet exemple, chaque maître constitue un niveau à lui seul. Un niveau peut être constitué de plusieurs maîtres ou de maîtres et d'esclaves. Chaque niveau possède un étage d'entrée, un décodeur d'adresse et un multiplexeur. Chaque décodeur est relié à tous les multiplexeurs. En fonction de l'adresse de destination, il commande les multiplexeurs pour connecter le niveau à l'esclave. Si deux niveaux demandent le même esclave au même moment, la politique interne d'arbritage à la matrice sélectionne un niveau. Les signaux de contrôle et de données du niveau en attente sont mémorisés dans l'étage d'entrée jusqu'à la connexion avec l'esclave. La politique d'arbitrage de la matrice peut être distribuée. Chaque esclave est attribué selon une politique qui lui est spécifique.

Les bus sont aujourd'hui les architectures les plus répandues dans les systèmes embarqués. Pour un nombre restreint d'unités, les bus constituent des architectures performantes pour une faible surface de silicium. Malheureusement, ces structures sont difficilement extensibles et ne conviennent plus lorsque le nombre d'unités devient important. Les réseaux sont plus complexes et plus coûteux en silicium, mais permettent d'interconnecter un grand nombre d'unités. Ils constituent les solutions de demain.

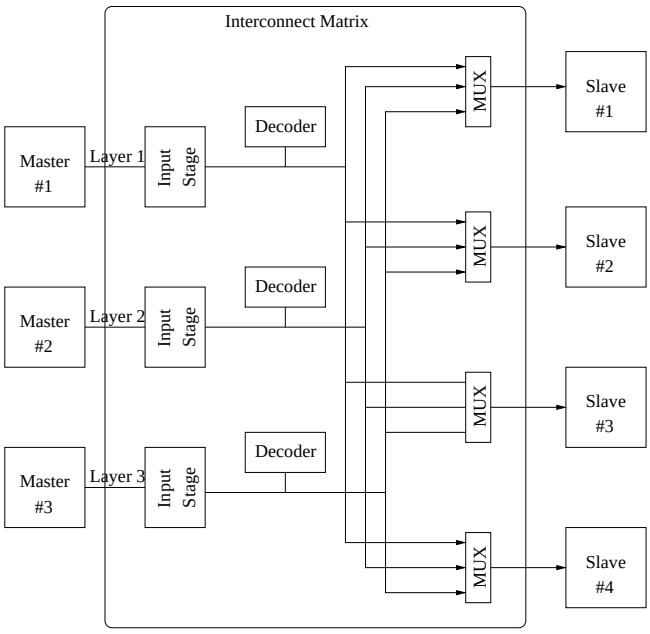


FIG. 2.5 – Exemple d'une matrice avec trois maîtres et quatre esclaves

2.3 Les réseaux (sur puce)

Les structures déployées sur les systèmes sur puce reposent sur les concepts déjà établis pour les réseaux d'ordinateurs. Néanmoins, deux problématiques, non pertinentes jusqu'ici, deviennent prépondérantes [Wol04]. Les systèmes embarqués fonctionnent en **temps réel**. Ces applications nécessitent des fonctionnalités et des performances prévisibles. Par exemple, la perte de messages est critique pour les réseaux intégrés ; elle ne l'est pas pour les réseaux de type Ethernet. Les systèmes embarqués opèrent sous de fortes **contraintes d'énergie et de puissance**. Dans les réseaux d'ordinateurs ou les stations multiprocesseurs, ces problèmes sont limités par des solutions mécaniques, comme l'ajout de volumineux ventilateurs ou radiateurs, par exemple.

Dans cette section, nous rappelons les principes généraux des réseaux d'interconnexion. Cette section s'inspire notamment des articles de Ni et McKinley [NM93], de Mohapatra [Moh98] et de l'ouvrage de Dally et Towles [DT04]. Nous définissons tout d'abord les notions de nœuds, topologies, routes et paquets. Nous abordons ensuite les algorithmes de routage et les techniques de commutation. Finalement, nous illustrons les différents concepts développés dans cette section par quelques exemples de réseaux "sur puce".

2.3.1 Nœuds, topologies, routes et paquets

Généralement, un nœud contient un processeur, une unité de mémoire et d'autres fonctionnalités qui peuvent être, par exemple, des co-processeurs ou des éléments de mémoire additionnels. Le pont connectant le nœud au reste du réseau est souvent réalisé par un composant spécifique, nommé *routeur*. Ce composant analyse les paquets entrants. Sa tâche est de déterminer si le paquet est arrivé à destination ou à quel voisin il doit être transmis. Un

schéma d'un modèle générique de nœud est donné figure 2.6 [NM93]. Le routeur possède en général autant de canaux d'entrée que de canaux de sortie. Les canaux externes permettent de recevoir et d'émettre vers les nœuds voisins. Les canaux internes permettent la communication avec les unités locales du nœud. Un canal est généralement unidirectionnel. Deux nœuds sont reliés par deux canaux pour permettre l'échange d'information dans les deux sens. Pour ne pas surcharger les figures, nous faisons apparaître ces deux canaux uniquement si cela est nécessaire. Sinon, nous ne dessinons qu'un seul lien. Ces unités sont reliées entre elles par un système local d'interconnexion qui est la plupart du temps un bus.

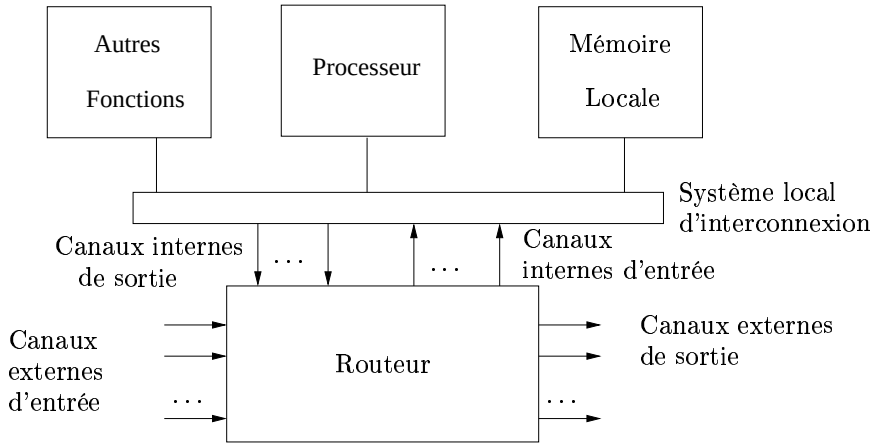


FIG. 2.6 – Modèle générique d'un nœud

Pour les réseaux, les deux topologies les plus répandues sont les grilles et les cubes.

Définition 2.1 Topologie en grille.
Une grille de dimension n est une structure d'interconnexion ayant $k_0 \times k_1 \times \cdots \times k_{n-1}$ nœuds, où k_i représente le nombre de nœuds dans la dimension i. Chaque nœud de la grille est identifié par un vecteur de n coordonnées $(x_0, x_1, \ldots, x_{n-1})$, où $0 \leq x_i \leq k_i - 1$. Deux nœuds $(x_0, x_1, \ldots, x_{n-1})$ et $(y_0, y_1, \ldots, y_{n-1})$ sont connectés si et seulement s'il existe un i tel que $x_i = y_i \pm 1$ et $x_j = y_j$ pour tout $j \neq i$. Le nombre de voisins d'un nœud varie de n à $2n$ en fonction de sa localisation.

Définition 2.2 Topologie en cube
Un kn-cube est une structure d'interconnexion à n dimensions ayant k nœuds dans chaque dimension. Chaque nœud est identifié par un vecteur de n coordonnées $x_0, x_1, \ldots, x_{n-1}$, où $0 \leq x_i \leq k - 1$. Deux nœuds $(x_0, x_1, \ldots, x_{n-1})$ et $(y_0, y_1, \ldots, y_{n-1})$ sont connectés si et seulement s'il existe un i tel que $(x_i = y_i \pm 1) \bmod k$ et $x_j = y_j$ pour tout $j \neq i$. Si $k = 2$, chaque nœud a n voisins, soit un dans chaque dimension. Si $k > 2$, chaque nœud a $2n$ voisins, soit deux dans chaque dimension.

La figure 2.7 montre une grille 8×8. La figure 2.8 montre un hypercube de dimension 3. Les hypercubes sont des cas particuliers des grilles et des kn-cubes. Un hypercube de dimension n est équivalent à une grille de dimension n où $k_i = 2$ pour tout i, $0 \leq i \leq n - 1$, ou à un $2n$-cube. Si $n = 2$, un kn-cube est un *tore*. Un tore est obtenu en connectant les

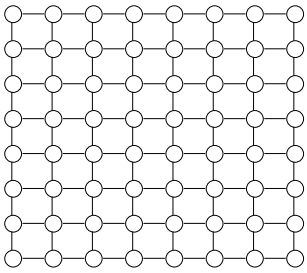


FIG. 2.7 – Grille 8x8

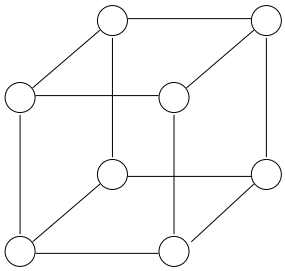


FIG. 2.8 – Hypercube de dimension 3

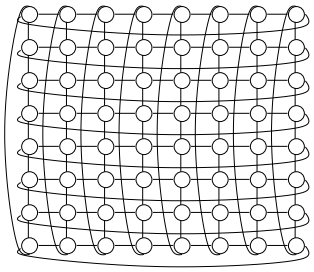


FIG. 2.9 – Tore

nœuds en bordure d'une grille aux nœuds de la bordure opposée (*c.f.* figure 2.9). Notons finalement que les *kn*-cubes sont des topologies symétriques et les grilles sont des topologies asymétriques.

Les nœuds échangent des messages qui sont, pour des raisons d'efficacité et de partage des ressources du réseau, souvent découpés en paquets. Un paquet étant plus court qu'un message, sa transmission est plus rapide. Les routeurs utilisés lors de la transmission d'un paquet sont disponibles plus rapidement pour d'autres paquets. Pour une topologie et un paquet donnés, une *route* (ou un *chemin*) entre un nœud source s et un nœud destination d est la liste des nœuds permettant de relier s à d. Si plusieurs routes existent entre s et d, les différents paquets d'un même message peuvent emprunter différentes routes.

Comme le montre la figure 2.10, un paquet est constitué de trois parties : une partie contenant les données à transmettre (*data*), une partie contenant les informations nécessaires au calcul de la route du paquet (*header*) et une partie délimitant la fin du paquet (*EOP* pour *End of Packet*). Un paquet constitue la plus petite entité possédant les informations sur le routage.

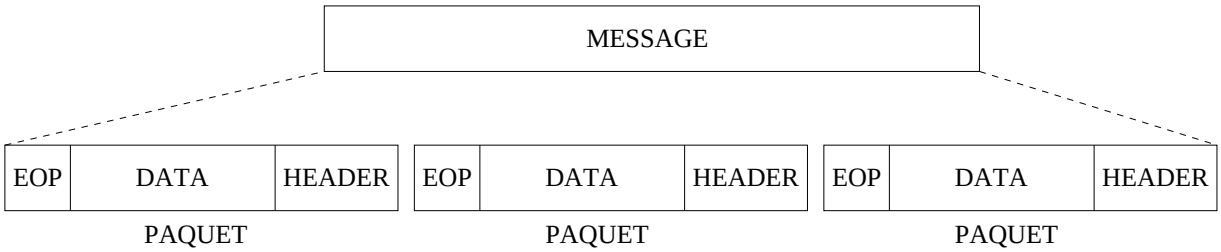


FIG. 2.10 – Division d'un message en paquets

2.3.2 Le routage

Les algorithmes de routage peuvent être classés selon différents critères. Selon la localisation des choix des routes, les algorithmes sont dits *à la source* ou *distribués*. Dans les algorithmes *à la source*, le chemin d'un message est fixé au nœud source, avant que le message ne soit envoyé. Dans les algorithmes *distribués*, le calcul de la route est effectué au fur et à mesure par chaque nœud intermédiaire visité par le message. À la réception d'un paquet,

le routeur décide si le message est arrivé à sa destination ou vers quel voisin le paquet doit être transmis.

Il est aussi possible de différencier les algorithmes comme étant *déterministes* ou *adaptatifs*. Si le chemin d'un message est uniquement fonction du nœud courant et du nœud destinataire, l'algorithme est dit *déterministe*. Les algorithmes adaptatifs prennent en compte la dynamique du réseau et fournissent plusieurs routes pour une même paire de nœuds.

Si l'algorithme de routage produit toujours une route correspondant à un des plus courts chemins entre la source et la destination, il est dit *minimal*. Sinon, il est dit *non-minimal*. Dans ce cas, des situations de "famine" (*livelock*) sont possibles : un message peut voyager indéfiniment dans le réseau sans jamais atteindre sa destination. Il faut prendre soin de concevoir un système prévenant ce phénomène.

Dans cette sous-section, ces notions sont illustrées à travers un algorithme déterministe minimal, un algorithme adaptatif minimal et un algorithme adaptatif non-minimal.

2.3.2.1 Un algorithme de routage déterministe et minimal

Une famille d'algorithmes souvent employée sur les grilles à n dimensions est celle du routage par ordonnancement des dimensions (*dimension-ordered routing*) [DS87]. Les dimensions sont ordonnées selon un ordre strict. Entre un nœud source et un nœud destination donné, un paquet effectue ses déplacements en suivant l'ordre donné aux dimensions. Ces algorithmes sont minimaux.

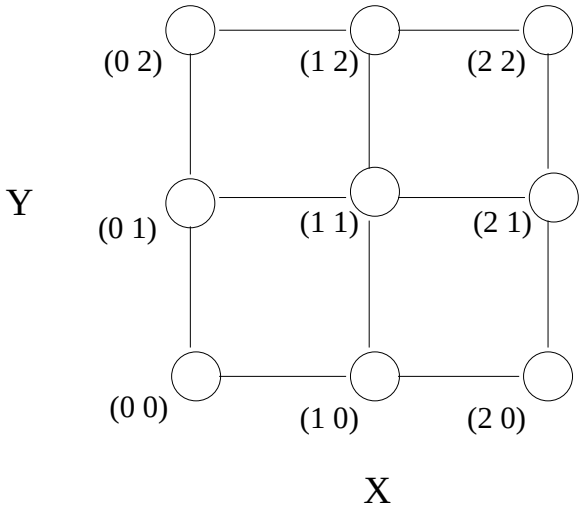


FIG. 2.11 – Routage en XY

Dans une grille à deux dimensions (Fig. 2.11), le routage par ordonnancement des dimensions s'appelle *routage en XY*. On identifie une dimension comme l'axe X et l'autre comme l'axe Y et l'ordre choisi est "X est plus petit que Y". Un paquet est d'abord routé selon l'axe X jusqu'à atteindre l'abscisse de la destination. Ensuite, il est routé selon l'axe Y jusqu'à atteindre l'ordonnée de la destination.

Pour les topologies en k*n*-cube il est impossible de fournir, pour des valeurs de k supérieures à 4, un algorithme minimal et déterministe garantissant l'absence d'interblocage

[NM93]. Les interblocages sont discutés en détail dans la section 2.4. L'algorithme de routage introduit des dépendances entre les canaux. Si un canal c_i est atteignable par la fonction de routage à partir d'un canal c_j, le routage crée une dépendance de c_j à c_i. La globalité des dépendances est représentée par un graphe orienté, dit graphe de dépendance. L'absence d'interblocage est garantie par l'absence de cycle dans le graphe de dépendance des canaux. Considérons l'anneau à une dimension et cinq nœuds de la figure 2.12. Si tous les canaux sont utilisables, leur dépendence est cyclique. Pour briser le cycle, il est nécessaire d'interdire un canal (le canal noté c par exemple). Dans ce cas, les paquets du nœud 2 destinés au nœuds 1 devront passer par les nœuds 3, 4 et 0. Ils ne peuvent emprunter un chemin minimal.

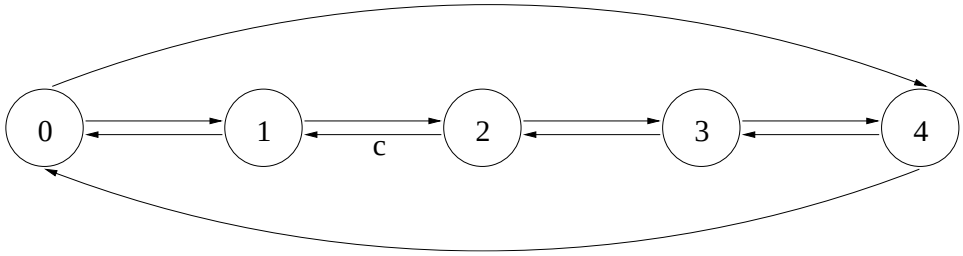


FIG. 2.12 – Anneau à une dimension et cinq nœuds

2.3.2.2 Algorithmes de routage adaptatifs minimaux

Parmi les algorithmes minimaux et adaptatifs, une technique classique consiste dans le découpage d'un réseau en sous-réseaux. Ce découpage est purement théorique. Les sous-réseaux ne présentent aucune réalité physique. En fonction de sa destination, un paquet se déplace dans un sous-réseau particulier. Dans chaque sous-réseau, plusieurs chemins minimaux sont possibles.

Un exemple typique est l'algorithme en double Y (*double Y-channel routing*). Cet algorithme considère une grille à deux dimensions où les nœuds sont reliés par un canal bidirectionnel dans la dimension X et par deux canaux bidirectionnels dans la dimension Y (Fig. 2.13). Le réseau est découpé en deux sous-réseaux. Chaque sous-réseau possède un canal dans la dimension Y. Chaque sous-réseau n'utilise le canal de la dimension X que dans une seule direction. Le sous-réseau $+X$ (en pointillés sur la figure 2.13) utilise ce canal dans le sens ascendant de la dimension X, le sous-réseau $-X$ (en traits continus épais sur la figure 2.13) dans le sens descendant. Soient s et d, les nœuds source et destination d'un paquet. Leurs coordonnées selon l'axe X sont notées s_x et d_x. Si à un nœud donné la destination du paquet courant est à droite (si $d_x > s_x$), le paquet utilise le réseau $+X$; si c'est à gauche (si $d_x < s_x$), le paquet utilise le réseau $-X$. Si $d_x = s_x$, on choisit de manière arbitraire. Dans chaque sous-réseau, plusieurs chemins minimaux sont possibles. Un paquet utilise le même sous-réseau pour la totalité de son voyage.

Exemple 1 *Soit la figure 2.13 et un paquet p_1 émis au nœud $s = (2\ 1)$ et destiné au nœud $d_1 = (0\ 0)$. Comme la destination de p_1 est vers la gauche, il va utiliser le sous-réseau $-X$. Les chemins possibles sont les suivants : $(2\ 1)\ (1\ 1)\ (0\ 1)\ (0\ 0)$ ou $(2\ 1)(1\ 1)(1\ 0)(0\ 0)$*

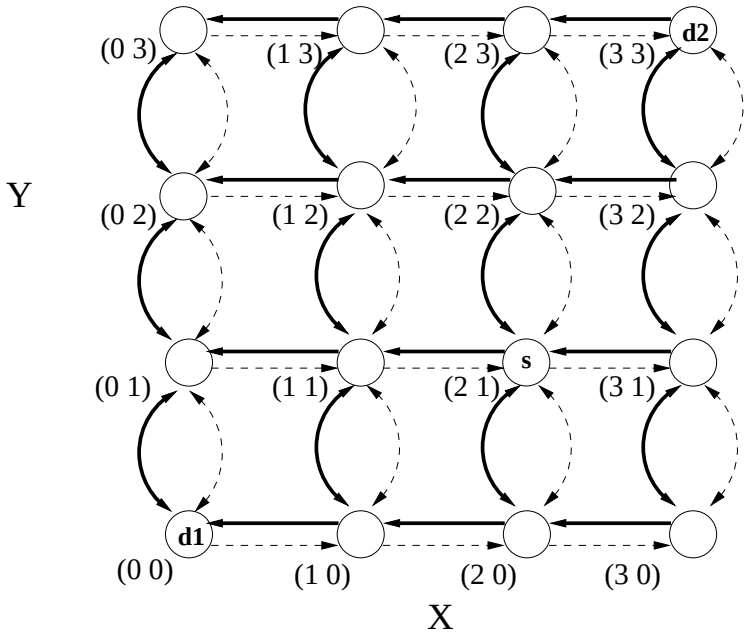


FIG. 2.13 – Routage en double Y

ou encore $(2\ 1)\ (2\ 0)\ (1\ 0)\ (0\ 0)$. *Pareillement, un paquet* p_2 *émis au nœud* s *et destiné au nœud* $d_2 = (3\ 3)$ *utilise un des chemin suivants du sous-réseau* $+X$: $(2\ 1)(3\ 1)(3\ 2)(3\ 3)$ *ou* $(2\ 1)(2\ 2)(3\ 2)(3\ 3)$ *ou encore* $(2\ 1)\ (2\ 2)\ (2\ 3)\ (3\ 3)$.

De manière générale, fournir des algorithmes minimaux et adaptatifs pour des topologies en kn-cube demande des canaux supplémentaires. Le nombre de canaux augmentent de manière exponentielle avec le nombre de dimensions du réseau [LH91].

2.3.2.3 Algorithmes de routage adaptatifs non-minimaux

Si un algorithme minimal n'est pas nécessaire, les algorithmes non-minimaux nécessitent moins de canaux additionels. Par exemple, si chaque nœud est relié par r paires de canaux à chacun de ses voisins, les deux algorithmes suivants, proposés par Dally et Aoki [DA90], peuvent être appliqués à des topologies de type grille et de type "kn-cube".

Dans l'algorithme de **routage par retour statique en dimension** (*static dimension reversal routing algorithm*), le réseau est divisé en r sous-réseaux. Le sous-réseau de classe i $(0 \leq i \leq r-1)$ contient toutes les i-èmes paires de canaux. L'entête (*header*) des paquets contient une information sur la classe c, initiallement $c = 0$. Tout paquet où $c < r-1$, peut être routé dans toutes les directions du sous-réseau de classe c. Le chemin peut ne pas être minimal. À chaque fois que le paquet est routé d'une dimension "haute" vers une dimension "basse" (soit en descendant l'ordre des dimensions), c est augmenté de 1. Quant $c = r-1$, le paquet est routé selon un algorithme par ordonnancement des dimensions présenté précédemment. Le paramètre r donne le degré d'adaptabilité du réseau.

Exemple 2 *Soit la figure 2.14 représentant une grille de dimension deux où deux nœuds adjacents sont reliés par trois paires de canaux* $(r = 3)$. *Les dimensions sont ordonnées de*

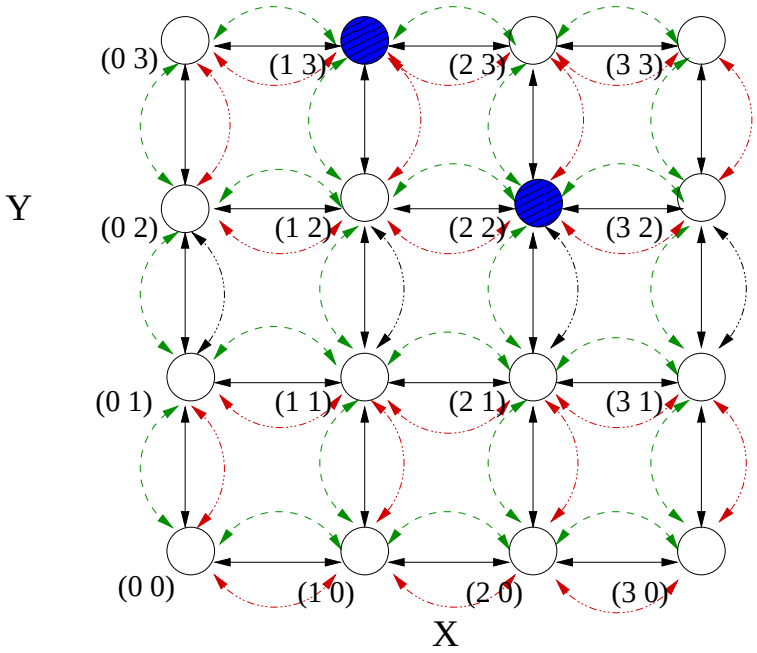


FIG. 2.14 – Routage par retour statique.

sorte que X soit la dimension "basse" et Y la dimension "haute". Ceci définit trois sous-réseaux : le sous-réseau de classe 0 représenté par les canaux en ligne pleine, le sous-réseau de classe 1 par les lignes en gros pointillés et le sous-réseau de classe 2 par les lignes en petits pointillés. Les nœuds noirs sont indisponibles (occupés ou en panne). Soit un paquet p émis au nœud $(0\ 3)$ et destiné au nœud $(3\ 3)$. Puisque le nœud $(1\ 3)$ est indisponible, p est envoyé au nœud $(0\ 2)$ par le canal du sous-réseau de classe 0. En utilisant un canal de ce même sous-réseau p est envoyé au nœud $(1\ 2)$. Ce virage se fait de la dimension Y vers la dimension X, soit d'une dimension "haute" vers une dimension "basse". Le paramètre c de p passe de 0 à 1. Comme le nœud $(2\ 2)$ est indisponible, p est envoyé au nœud $(1\ 1)$. Puis, toujours par le sous-réseau de classe 1, p est envoyé au nœud $(2\ 1)$ et c passe de 1 à 2. Par conséquent, $c = r - 1$ et p est routé selon un algorithme déterministe, l'algorithme en XY par exemple. Donc, p va tout d'abord atteindre la coordonnée X de sa destination au moyen des canaux du sous-réseau de classe 2. Une fois arrivé au nœud $(3\ 1)$, p va rejoindre sa destination dans la dimension Y.

Dans l'algorithme de **routage par retour dynamique en dimension** (*dynamic dimension reversal routing algorithm*), les canaux sont répartis en deux classes non-vides : la classe adaptative et la classe déterministe. Les paquets voyagent initiallement en utilisant les canaux de la classe adaptative et peuvent être routés dans toutes les directions sans limite sur le nombre de fois où ils passent d'une dimension "haute" à une dimension "basse". Le champ c est néanmoins toujours incrémenté. Si un paquet arrive à un nœud où tous les canaux possibles sont occupées par des paquets ayant un c inférieur ou égal au sien, il va être routé selon un algorithme déterministe et utilise les canaux du sous-réseau déterministe. Une fois qu'un paquet utilise les canaux de la classe déterministe, il ne peut plus utiliser

ceux de la classe adaptative. La répartition des canaux d'un réseau entre la classe adaptative et la classe déterministe est cruciale dans la réalisation des ces algorithmes.

Exemple 3 *Reprenons l'exemple précédent et associons les canaux du sous-réseau 0 à la classe déterministe et les autres à la classe adaptative. Supposons que lorsque p arrive au nœud $(1\ 1)$ (pour p, $c = 1$), deux autres paquets occupent les canaux des sous-réseaux adaptatifs. Supposons aussi que pour chacun de ces paquets $c = 0$. Le paquet p va rejoindre sa destination $(2\ 1)$ par le canal 0, soit le canal déterministe.*

2.3.3 Les techniques de commutation

La commutation désigne l'ensemble des mécanismes plaçant les données présentes à un canal d'entrée d'un routeur sur un canal de sortie. Il existe quatre techniques : commutation par paquets, commutation par circuits, commutation par canaux virtuels et la commutation par "vers de terre". Ces quatre techniques ont toutes été mises en oeuvre dans les réseaux d'ordinateurs et sont toutes présentes dans les architectures proposées aujourd'hui pour les systèmes sur puce. Le réseau Æthereal de Philips (*c.f.* sous-section 2.3.4.2 ci-dessous) propose deux modes de fonctionnement : la commutation par circuits ou la commutation par vers de terre [GvMPW02, RGR+03]. Le réseau Octagon [KND02] de STMicroelectronics (*c.f.* sous-section 2.3.4.3 ci-dessous) fonctionne en commutation par paquets ou par circuits. Les récents développements de STMicroelectronics se dirigent vers des réseaux implémentant la commutation par vers de terre. Le réseau SPIN (*c.f.* sous-section 2.3.4.1 ci-dessous) développé à l'université Pierre et Marie Curie implémente la commutation par vers de terre [AG03].

Nous exposons brièvement les principes de chacune de ces techniques. Elles se différencient principalement par leur *temps de latence*. Le temps de latence représente le temps requis par un paquet pour atteindre sa destination. Ce temps est généralement décomposé en deux parties [DT04]. Une partie représentant le temps, t_h, de transmission de l'entête du paquet. La décomposition de ce temps est complexe et prend en compte les fonctionnements internes des routeurs. Pour notre propos, ces détails ne sont pas pertinents. Soit L la longueur du paquet privée de son entête. Ce reste du paquet est transmis en un temps égal au rapport de cette longueur sur la bande passante b, soit $\frac{L}{b}$. Nous notons $dist(s, d)$ la distance entre un nœud source s et un nœud destination d.

2.3.3.1 Commutation par paquets

En *commutation par paquets* (ou *store-and-forward*), chaque paquet d'un message est routé de manière indépendante. Lorsqu'un paquet arrive à un nœud, il est entièrement mémorisé par le nœud. Il est ensuite analysé et délivré au nœud courant ou transmis à un nœud voisin si ce dernier est disponible. Dans cette configuration, un nœud n'est occupé que pendant le temps où il traite un paquet. De plus, en fonction de l'encombrement du réseau, la route du paquet peut être modifiée. Les inconvénients de cette technique sont que le temps de transmission est proportionnel à la longueur du chemin et des éléments de mémorisation sont requis à chaque nœud. Ces éléments doivent pouvoir mémoriser le plus long paquet qui puisse être émis sur le réseau.

La figure 2.15 illustre la transmission en commutation par paquets entre un nœud source s, un nœud destination d et à travers un nœud intermédiaire i. Le paquet est entièrement mémorisé avant d'être transmis. Le temps de latence est directement proportionnel à la distance entre la source et la destination.

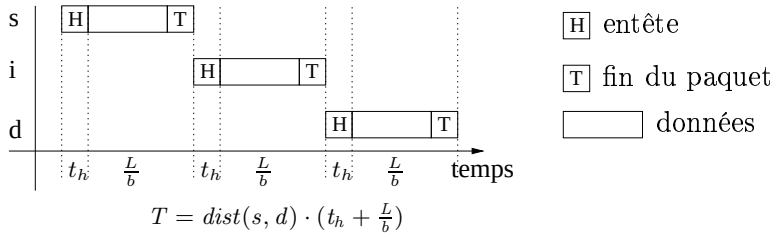


FIG. 2.15 – Transfert en commutation par paquets

2.3.3.2 Commutation par canaux virtuels

Pour réduire le temps nécessaire à la mémorisation des paquets, Kermani et Kleinrock [KK79] ont introduit la *commutation par canaux virtuels*. Un *canal virtuel* consiste en une série d'éléments de mémorisation associée à un canal reliant deux nœuds adjacents. Le canal est virtuel dans le sens où il n'est pas visible par les autres nœuds du réseau. Lorsqu'un paquet est transmis d'un nœud a à un nœud b à travers un canal c, seule son entête est analysée. Si le canal est indisponible, tout le paquet est mémorisé dans un canal virtuel associé à c. Sinon le paquet est transmis directement au nœud b. Ainsi, le temps de latence est en général plus court que pour la commutation par paquets ou équivalent dans le pire cas. Malheureusement, un nœud doit pouvoir mémoriser un grand nombre de messages, et de volumineux éléments de mémorisation sont nécessaires.

La figure 2.16 illustre la transmission en commutation par canaux virtuels entre un nœud source s, un nœud destination d et à travers un nœud intermédiaire i. Une fois que l'entête est analysée les données sont tout de suite transmises.

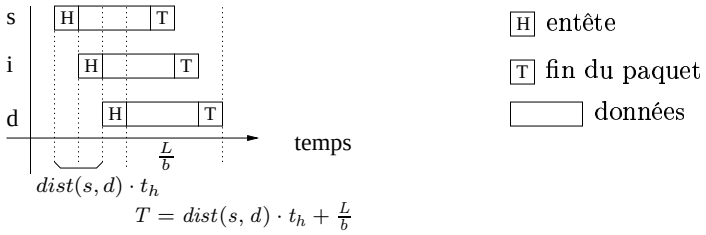


FIG. 2.16 – Transfert en commutation par canaux virtuels

Exemple 4 *Soient a, b, c et d quatre nœuds. Soient p_a, p_b, p_c trois paquets arrivant des nœuds a, b et c à un nœud intermédiaire i comme le montre la figure 2.17.*

Supposons que p_a, destiné au nœud d, soit le premier paquet reçu par i. Le nœud i analyse l'entête de p_a et transmet ce dernier vers d sans mémorisation. Le port de sortie de i vers d est indisponible pour p_b et p_c. En commutation par paquets, les paquets p_b et p_c restent

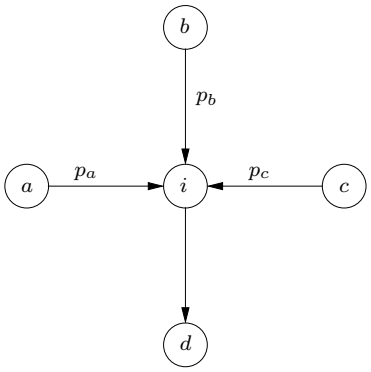


FIG. 2.17 – Figure pour l'exemple

aux nœuds b et c et attendent que i transmette p_a. En commutation par canaux virtuels, les paquets p_b et p_c sont mémorisés au nœud i dans les canaux virtuels associés au canal de i vers d.

Dans le mode paquet, p_b et p_c ont été entièrement mémorisés par les nœuds b, c et i (lors du passage vers d). Dans la commutation par canaux virtuels, ils ne sont entièrement mémorisés que par le nœud i. Par contre, ce dernier doit avoir suffisamment de mémoire pour les accepter.

2.3.3.3 Commutation par vers de terre

La *commutation par vers de terre* (*wormhole routing*) est une variante de la commutation par canaux virtuels et a pour but de limiter la taille des éléments de mémorisation. Un paquet est divisé en *morceaux* (en *flits*) comme le montre la figure 2.18. Les trois parties constituantes d'un paquet se retrouvent dans les morceaux. Le premier morceau (la tête du vers) reprend la partie *header* et des données. Le dernier morceau (la queue du vers) reprend la partie *EOP* et des données. Les autres morceaux ne contiennent que des données. Lorsque la tête du vers franchit un nœud, les morceaux restants du vers suivent en *pipeline*. La figure 2.19 montre une situation où un vers occupe cinq nœuds, le corps du vers suit la tête. Puisque les informations sur la destination du paquet sont contenues uniquement dans la tête du vers, il n'est pas possible de couper un vers. À partir du moment où un nœud accepte la tête d'un vers, il ne peut en accepter une autre tant que la queue du vers courant n'est pas passée.

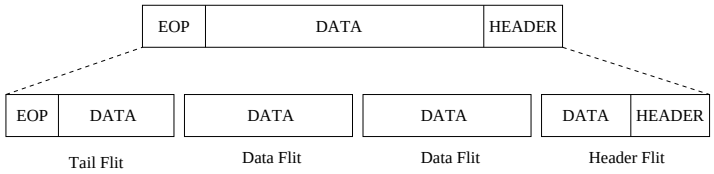


FIG. 2.18 – Découpage d'un paquet en *flits*.

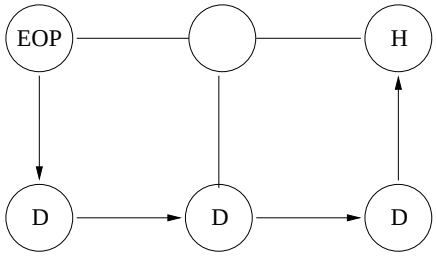


FIG. 2.19 – Déplacement du vers.

En utilisant cette technique, les éléments de mémorisation sont réduits puisque seule la

mémorisation des morceaux est nécessaire. Les paquets ne sont plus mémorisés dans leur intégralité. Par contre, comme nous allons le voir dans la section 2.4, cette technique est propice aux interblocages.

2.3.3.4 Commutation par circuits

En *commutation par circuits*, le chemin complet entre la source s et la destination d est réservé avant de transmettre les données. La figure 2.20 illustre la transmission en commutation par circuits à travers un nœud intermédiaire i. Tout d'abord, une entête est envoyée pour réserver le circuit. La réservation est confirmée par des messages d'aquittement. Ensuite, les données sont transmises sans aucune mémorisation. Finalement, à la fin du paquet les nœuds sont libérés. Dans ce mode, le niveau de parallélisme est limité. Néanmoins, seuls l'entête et l'aquittement sont analysés par la fonction de routage des nœuds du circuit. Ce sont les seuls paquets à être mémorisés. La distance entre la source et la destination est parcourue trois fois : une fois pour transférer la requête, une fois pour l'acquittement et une troisième fois pour envoyer le premier morceau des données.

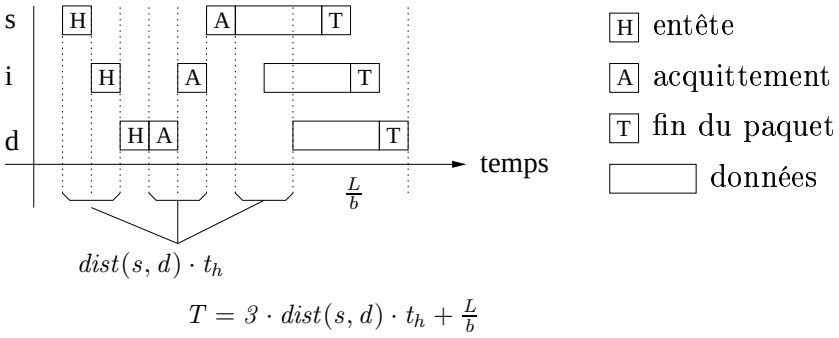


FIG. 2.20 – Transfert en commutation par circuits

L'intérêt de ce mode est de transmettre les paquets "en rafale" une fois le circuit créé. La création du circuit peut nécessiter plusieurs tentatives en fonction du traffic sur le réseau. Plus le nombre de paquets transmis en rafale est grand, plus la contribution de la distance au temps total de transmission est faible. Ainsi, on obtient un temps de latence inférieure à celui des techniques précédentes.

La commutation par paquets est difficilement applicable pour des réseaux de grande taille. La commutation par canaux virtuels permet de réduire l'impact de la distance sur le temps de latence. La commutation de circuits est la technique idéale lorsqu'il s'agit de transmettre un grand nombre de paquets à une destination lointaine. Ces trois techniques requièrent une bande passante élevée pour un débit important. La commutation par vers de terre joue plutôt sur la longueur des séquences à transmettre pour augmenter ce débit. Comme le montre la sous-section suivante, les deux techniques qui semblent s'imposer aujourd'hui dans les systèmes sur puce sont les commutations par circuits et par vers de terre. La commutation par circuits offre des transmissions sûres et limitant les interblocages. La commutation par vers de terre offre un plus grand débit mais elle est plus complexe à mettre en oeuvre et plus propice aux interblocages.

2.3.4 Réalisations de NoC

Les réseaux sur puce constituent un paradigme récent [BM02]. À ce titre, ils ont fait l'objet de publications sur les principes ([YBM03] par exemple). Nous nous intéressons dans cette section plus particulièrement aux expérimentations relatives à l'implémentation physique des réseaux. Nous donnons un bref aperçu des différents travaux menés à ce sujet. Une présentation plus complète est disponible dans la littérature [MCM+04]. À l'heure actuelle, seules deux études proposent un protypage sur FPGA : les travaux de Marescaux et Bartic *et al.* [MBV+02, BMN+03] et les travaux sur le réseau Hermes [MMM+03, MTCM05]. Il existe aussi quatre réalisations en technologie ASIC : le réseau SPIN [AG03], le réseau aSOC [LST00], le réseau Æthereal de Philips [RGR+03] et le réseau Proteo [SAN03]. Aucune de ces quatre publications ne donne de précision quant à la réalisation de ces conceptions sur silicium. La plupart des études concernent des modèles de réseaux sans considérer leur implémentation physique [For02, MNT+04, SSM+01, DBG+03, KND02, DT01].

Parmi ces diverses études, nous avons sélectionné une réalisation universitaire et deux réalisations industrielles. Nous les présentons brièvement.

2.3.4.1 Le réseau SPIN

Le réseau SPIN (*Scalable, Programmable, Integrated Network*) [GG00, ACG+03], est un réseau synchrone développé à l'université Pierre et Marie Curie, Paris VI. La topologie de ce réseau est un arbre élargi (*c.f.* figure 2.21). Un arbre élargi est constitué de l'interconnexion de plusieurs arbres simples composés uniquement d'une racine et de feuilles. La racine est un routeur connecté à plusieurs applications - actives ou passives - au moyen d'une paire de canaux internes par application. Les racines sont répétées sur plusieurs étages, par exemple la figure 2.21 comporte un seul étage de doublons. Ces doublons offrent quatre routes possibles entre deux racines. Par exemple, la racine 1 est reliée à la racine 3 par quatre routes, chacune empruntant un doublon différent.

Tous les liens sont divisés en deux canaux unidirectionnels indépendants de 32 bits. La technique de commutation est celle du vers de terre. Le flot de données est contrôlé selon la méthode des crédits [Tan96, DT04]. Pour chaque connexion entre deux unités, l'unité émettrice traque l'espace de mémorisation disponible chez le récepteur à l'aide d'un compteur. À chaque paquet envoyé, ce compteur est décrémenté de l'espace occupé par les données de ce paquet. Si le compteur atteint la valeur zéro, l'envoi de paquet cesse. Lorsque de l'espace se libère, le récepteur rend les crédits correspondants à l'émetteur. Ceci prévient la surcharge au niveau des unités réceptrices.

2.3.4.2 Le réseau Æthereal de Philips

L'allocation des ressources (*e.g.* canaux et buffers) aux communications est effectuée selon une politique précise attribuant un "service de communication" d'une certaine "qualité". La qualité de service proposée sur un réseau se divise en deux catégories. La catégorie *service garanti* assure un niveau de performance tant qu'une série de contraintes est satisfaite. Par exemple, on peut garantir que 99,999% des paquets atteignent leur destination en 1μs tant que le débit est inférieur à 100Kbits par période de 100ns. *A contrario*, la catégorie *au*

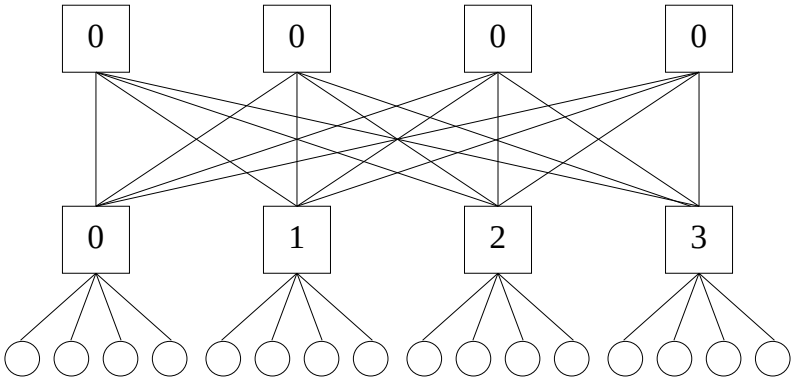


FIG. 2.21 – L'arbre élargi du réseau SPIN

mieux, ne garantit aucun service. Le réseau fait de son mieux pour acheminer les paquets. En fonction du traffic, les paquets sont retardés ou même perdus.

Le principe du protocole Æthereal [RDP+05, GDR05] proposé par Philips est de combiner ces deux classes de qualité. C'est un protocole de type maître/esclave. Son objectif est de créer des connexions *garanties* entre les maîtres et les esclaves au moyen de paquets *au mieux*.

Le fonctionnement est illustré sur la figure 2.22. Pour établir une connexion avec un esclave s, un maître m envoie, *au mieux*, un premier paquet, noté *SETUP*. Ce paquet *SETUP* tente de réserver les liens d'un chemin de s à m. À chaque nœud intermédiaire i, si $i = s$, la connexion est établie et un paquet d'aquittement (ACK) est envoyé *au mieux* au maître m. Ensuite, les paquets utilisent les liens réservés garantissant l'acheminement des paquets. Si $i \neq s$ et que la réservation est possible, le paquet *SETUP* est passé au prochain nœud intermédiaire. Sinon, la création de la connexion a échoué et un paquet signalant cet échec ($NACK$) est envoyé *au mieux* au maître m. Ce dernier annule les réservations effectuées jusqu'au nœud i au moyen d'un paquet *TDOWN* (*tear-down*). Ce dernier emprunte le même chemin que le paquet *SETUP* initial.

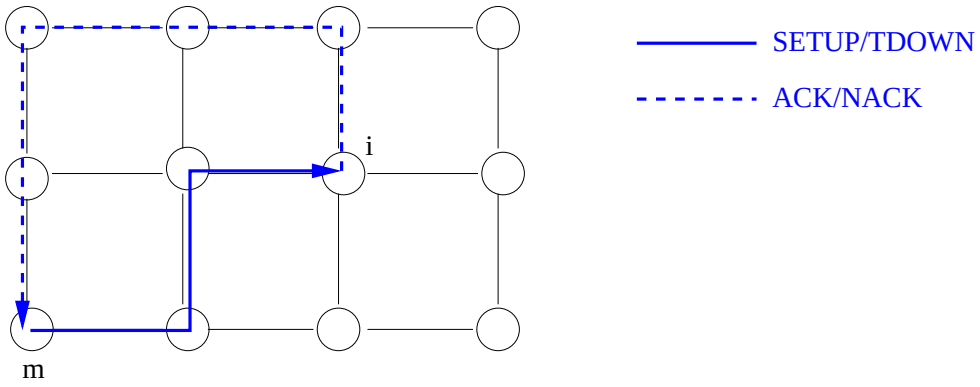


FIG. 2.22 – Protocole Æthereal de Philips

Pour les paquets de la classe *au mieux*, la commutation s'effectue par vers de terre. Pour la classe de *service garanti*, la commutation s'effectue par circuits avec partage du temps (*Time Division Multiplexing*). Le principe du TDM est de découper le temps en *fenêtres* qui

se répètent de manière périodique. Les liens sont réservés dans l'espace (entre deux nœuds) et dans le temps (dans une fenêtre). Chaque routeur maintient une table de réservation associant un circuit et une *fenêtre temporelle*. La figure 2.23 illustre ce partage du temps. Le nœud s a créé un circuit (noté x) avec le nœud d et un circuit avec le nœud e (noté o). Pendant le temps alloué par la première fenêtre le nœud s envoi des paquets destinés au nœud d et le nœud i envoie des paquets au nœud e. Dans le temps alloué par la deuxième fenêtre, le nœud s envoie des paquets destinés au nœud e et le nœud i des paquets destinés au nœud d. Les différents liens sont partagés dans le temps entre plusieurs connexions.

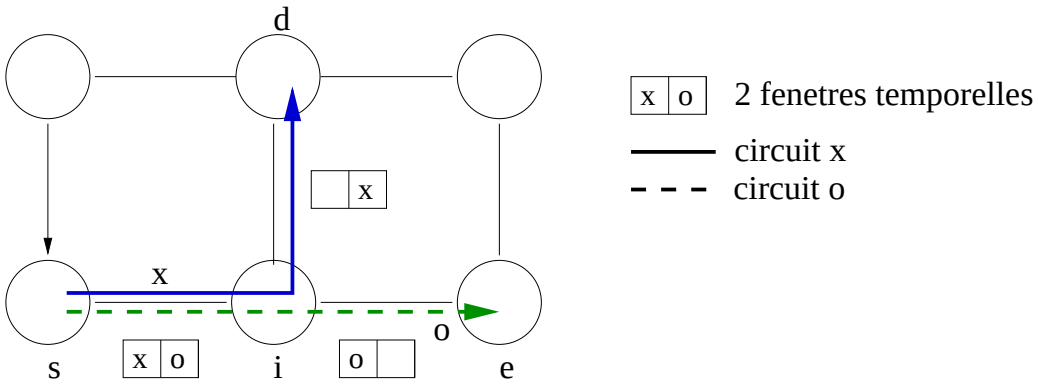


FIG. 2.23 – Partage du temps dans le réseau Æthereal

Le réseau est une architecture synchrone. Tous les éléments du réseau fonctionnent à la même vitesse (500MHz en l'occurence). Ce protocole est au niveau logiciel et indépendant de la topologie. Par exemple, Gangwal *et al*, proposent une grille 2×3 et une grille 1×3 pour une application de codage MPEG-2 [GRG+05]. Le flot de donné est contrôlé par la méthode des crédits.

2.3.4.3 Le réseau Octagon de STMicroelectronics

Le réseau Octagon est une architecture synchrone. L'unité primaire du réseau (*cf.* Fig. 2.24) est constituée de huit nœuds et douze liens birectionnels [KNDR01, KND02]. Les deux propriétés principales de cette architecture sont :

1. pour toute communication entre deux nœuds, au plus un seul nœud intermédiaire est nécessaire,
2. l'algorithme de routage est simple et minimal.

La route d'un paquet est calculée de la manière suivante. Chaque nœud compare l'adresse de destination ($PackAd$) avec sa propre adresse ($NodeAd$). Chaque nœud calcule une adresse relative ($RelAd$) définie comme suit :

$$RelAd = (PackAd - NodeAd) \; mod \; 8 \tag{2.1}$$

À chaque nœud, la route d'un paquet est fonction de cette adresse :
- $RelAd = 0$, le paquet a atteint sa destination,

- $RelAd = 1$ ou 2, le paquet est envoyé au prochain nœud dans le sens des aiguilles d'une montre,
- $RelAd = 6$ ou 7, le paquet est envoyé au prochain nœud dans le sens inverse des aiguilles d'une montre,
- sinon le paquet est envoyé au nœud "d'en face".

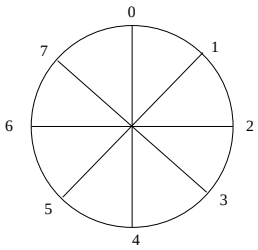


FIG. 2.24 – Unité primaire de l'Octagon.

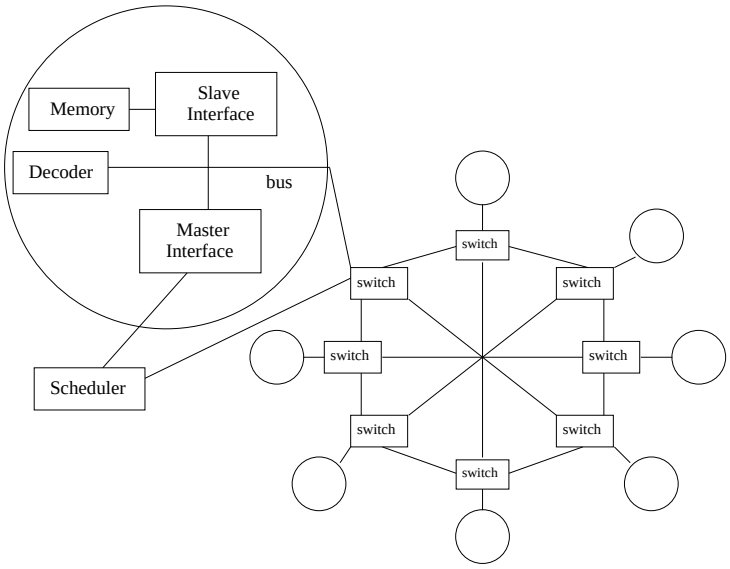


FIG. 2.25 – Noeuds connectés aux commutateurs et à l'arbitre central.

Exemple 5 *Soit Pack un paquet émis au nœud 2 et destiné au nœud 5. Tout d'abord, le nœud 2 calcule* $5 - 2 \bmod 8 = 3$*, et Pack est envoyé au nœud "d'en face", soit le nœud 6. Ensuite, le nœud 6 calcule* $5 - 6 \bmod 8 = 7$*, et Pack est envoyé vers le prochain nœud dans le sens des aiguilles d'une montre, soit le nœud 5. Finallement,* $5 - 5 \bmod 8 = 0$*, et Pack a atteint sa destination.*

Chaque nœud est un système construit autour d'un bus contenant un décodeur d'adresses, des interfaces maîtres et esclaves ainsi qu'une unité de mémoire. Ces systèmes sont connectés à l'Octagon *via* des *switches* (Fig. 2.25). Les maîtres et les *switches* sont connectés à un arbitre central qui régule les communications.

Si la mémoire et l'interface maître impliquées dans une communication appartiennent au même nœud, la requête est dite *locale* et la sortie du décodeur est mise à 1 ("vrai") pour activer l'interface esclave locale. Sinon, la requête est dite *distante* et la sortie du décodeur est mise à 0 ("faux"). Dans le réseau "Octagon", deux modes sont possibles : commutation par circuits ou par paquets. En fonction de ces modes et de l'état d'occupation des files, l'arbitre central va autoriser ou non l'émission de paquets vers un nœud donné.

Actuellement, STMicroelectronics s'oriente vers un réseau à seize nœuds (*c.f.* figure 2.26) implémentant la commutation par vers de terre et un routage similaire à celui de l'Octagon. L'arbitre central n'est plus utilisé.

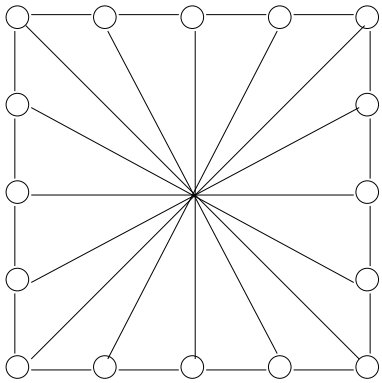


FIG. 2.26 – Le réseaux Spidergon de STMicroelectronics

2.4 Théorie des algorithmes sans interblocage

Une situation d'interblocage (*deadlock*) survient dans un réseau lorsqu'un paquet est bloqué indéfiniment à un nœud du réseau. La figure 2.27 montre une situation d'interblocage. Aucun paquet ne peut plus avancer car tous les nœuds aux alentours sont occupés et leur libération ne surviendra jamais. En commutation par vers de terre, si un vers est bloqué, tous les nœuds occupés par ses *flits* le sont aussi, engendrant autant de sources d'interblocage.

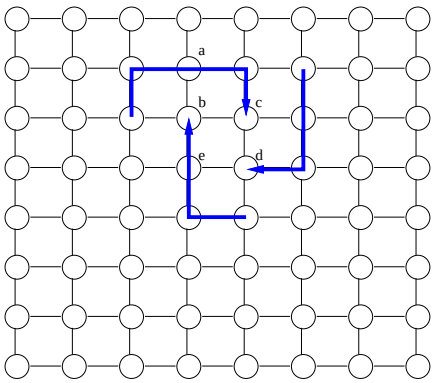


FIG. 2.27 – Exemple d'interblocage.

Pour pallier ce problème, deux solutions sont possibles : soit on corrige une situation de "deadlock", soit on l'évite. La première solution suppose de fournir des mécanismes capables de détecter et de corriger ces situations. Par exemple, un arbitre central peut être relié à tous les routeurs et contrôler l'émission des messages. Les analyse nécessaires à la détection et à la récupération des interblocages dégradent les performances du réseau dans le pire cas. Ces techniques sont applicables lorsque les interblocages sont peu fréquents et que les performances moyennes importent plus que celles du pire cas. Une autre solution est de définir des algorithmes ne produisant pas d'interblocage. Comme nous allons le voir dans cette section, l'absence d'interblocage est garantie par l'absence de cycle de dépendance entre les canaux. Pour casser les cycles, des canaux supplémentaires ou des restrictions sur le routage sont nécessaires. Les performances moyennes sont altérées. La plupart des réseaux

sur puce reposent sur la technique du vers de terre propice aux interblocages. La solution retenue est la prévention des interblocages plutôt que leur détection.

Plusieurs théories ont été developpées dans le but de prouver formellement qu'un algorithme ne produira pas de telle situation. Le routage peut être vu comme un problème d'allocation de ressources (*e.g.* les canaux). Ce problème est très général et a été notamment rencontré dans les systèmes d'exploitation des ordinateurs. Les travaux pionniers sont ceux de Havender [Hav68], Habermann [Hab69] et Holt [Hol72]. Ces travaux constituent aujourd'hui le contenu de livres destinés à l'enseignement dans les universités (*e.g.* [Kra85]). Ces résultats ont été appliqués aux réseaux d'ordinateurs et sont applicables aux architectures multiprocesseurs et aux réseaux sur puce. À propos des réseaux d'ordinateurs et des architectures multiprocesseurs, les principales contributions sont celles de Dally et Seitz [DS86, DS87], Duato [Dua93, Dua94, Dua96], Schwiebert et Jayasimha [SJ95], Fleury et Fraigniaud [FF98]. Cette section en expose brièvement les principes élémentaires. L'objet de cette thèse n'est pas l'étude de ces théories. Nous discutons leur connexion avec notre formalisme lors de l'exposition de nos conclusions.

Les théories sus-citées reposent sur les notions de ressource et de dépendance de ressources. Dans les réseaux, les ressources sont des canaux, des files ou dans certains cas les routeurs. Les détenteurs des ressources sont les paquets dans le cas des commutations par paquets ou canaux virtuels. Les ressources sont attribuées à un circuit ou un flit si la commutation s'effectue par circuits ou par "trou de vers". La fonction de routage crée des dépendances entre les ressources. La relation de dépendance est une relation d'ordre partiel, notée $\succ$. Par exemple, sur la figure 2.27 la fonction de routage au nœud b propose le nœud a, il y a une dépendance de b vers a, soit $b \succ a$. De même le nœud c dépend du nœud d qui lui dépend du nœud e en relation de dépendance avec le nœud b. Par transitivité, on a une relation entre b et lui même, soit $b \succ a \succ c \succ d \succ e \succ b$. La relation de dépendance peut être représentée par un graphe orienté. Les sommets sont les ressources et un arc du sommet i au sommet j représente une dépendance de i vers j. De manière générale, une condition suffisante pour prouver l'absence d'interblocage est de prouver que le graphe de dépendance est acyclique. Dans le cas des réseaux, cette condition est aussi nécessaire pour les algorithmes déterministes [DS87]. Par contre, pour les algorithmes adaptatifs, une fonction de routage peut introduire des cycles sans nécessairement introduire des interblocages. Le routage adaptatif permet un choix parmi les routes possibles. Si pour chaque cycle possible, il existe une voie de secours qui soit sans interblocage, la fonction de routage globale est sans interblocage. Dans ce contexte, l'idée est de considérer la sous-fonction de routage qui n'utilise que les voies de secours, et de montrer qu'elle n'introduit pas d'interblocage. Son graphe de dépendance reflète les dépendances *directes* entre les ressources qu'elle manipule. Un paquet qui utilise une voie de secours peut *a priori* en sortir. Il peut ensuite revenir sur une voie de secours pour briser un nouveau cycle. Cette situation n'est pas prise en compte si l'on ne considère que les dépendances directes de la sous-fonction de routage. En effet, celle-ci ne propose que les voies de secours. Dans ce cas, il faut prendre en compte les dépendances *indirectes* qui apparaissent entre les ressources de la sous-fonction de routage possiblement reliées par des ressources de la fonction de routage. Le graphe de dépendance *étendu* d'une sous-fonction de routage est son graphe de dépendance augmenté des dépendances indirectes[1].

[1]Il se peut que la fonction de routage et la sous-fonction de routage partagent des ressources. Dans ce

Plus formellement, deux théorèmes résument les théories. Le premier constitue une condition suffisante dans le cas adaptatif et une condition suffisante et nécessaire dans le cas déterministe.

Théorème 2.1 Condition suffisante.
Une fonction de routage connexe pour un réseau d'interconnexion est sans interblocage s'il n'existe pas de cycle dans son graphe de dépendance.

Le second est une condition suffisante et nécessaire dans le cas adaptatif.

Théorème 2.2 Condition nécessaire et suffisante.
Une fonction de routage connexe pour un réseau d'interconnexion est sans interblocage si et seulement s'il existe une sous-fonction de routage telle que cette sous-fonction soit connexe et qu'il n'existe pas de cycle dans son graphe de dépendance étendu.

Pour prolonger les illustrations de ces théorèmes, nous les appliquons aux algorithmes présentés dans la section 2.3.2.

Exemple 6 *L'algorithme de routage en XY n'a pas de cycle dans son graphe de dépendance car une fois passé de la dimension X à la dimension Y on ne revient jamais en X. Puisque le graphe est acyclique, le théorème 2.1 permet de conclure que cet algorithme est sans interblocage. La preuve de l'algorithme en double Y est similaire. Un paquet se déplace uniquement dans un sous-réseau et il n'y a pas de cycle dans le graphe de dépendance.*

Exemple 7 *L'algorithme statique de Dally et Aoki autorise un seul virage susceptible de conduire à une situation d'interblocage (soit d'une dimension haute vers une dimension basse) dans chaque sous-réseau. Donc, pour $c < r - 1$ le théorème 2.1 permet de conclure à l'absence d'interblocage. Lorsque $c = r - 1$, l'algorihme déterministe doit être sans interblocage. Si le choix se porte sur l'algorithme en XY, il y a absence d'interblocage. Concernant l'algorithme dynamique de Dally et Aoki des cycles sont possibles dans le sous-réseau adaptatif. Cela dit, toute situation d'interblocage dans le réseau adaptatif va être résolu car un paquet possédant la valeur de c la plus forte va passer dans le réseau déterministe. Si la sous-fonction déterministe est sans cycle dans son graphe de dépendance, par application du théorème 2.2, l'algorithme est sans interblocage.*

2.5 Vérification formelle des communications

La vérification des communications consiste la plupart du temps en la preuve de propriétés exprimant l'absence de famine ou d'interblocage, l'équité et l'exclusion mutuelle des accès à une section critique, la cohérence des accès mémoires, etc ...Les méthodes les plus utilisées pour prouver ces propriétés reposent sur la vérification de modèles. Les approches basées sur la démonstration de théorèmes permettent de vérifier une partie de ces propriétés ainsi

cas, il apparaît des dépendances *croisées*. En pratique, cette situation se produit très rarement (*c.f.* [DT04] page 275) et nous n'abordons pas ces dépendances. Elles sont discutées par Duato [Dua96].

que des aspects fonctionnels comme la preuve que la composition d'un récepteur et d'un émetteur est une identité.

Les architectures de communication sont récemment devenues critiques dans le flot de conception. Les efforts de vérification formelle dans ce domaine sont eux aussi récents. Avant de présenter les efforts de vérification concernant les bus et les réseaux spécifiques aux systèmes embarqués, nous rappelons brièvement les travaux au sujet des protocoles.

2.5.1 Vérification des protocoles

Dans la plupart des cas, les approches de vérification reposent sur l'utilisation des logiques temporelles. Les propriétés sont vérifiées soit par *model checking* [CGJ97, HLR92], soit par une combinaison de cette technique et d'assistants de preuves [BFS95, HS96]. Les protocoles sont spécifiés par des automates, notamment les automates "I/O" de Lynch et Tutle [Lyn96, DGRV00].

Ces travaux considèrent la vérification d'invariants inductifs d'un réseau de processus identiques. Pour prouver une formule ϕ, on définit (et on prouve) un invariant I pour un ensemble de processus P de sorte que cet invariant implique ϕ, soit :

$$I \vDash \phi \Longrightarrow \forall p \in P, p \vDash \phi \tag{2.2}$$

La difficulté est de trouver les invariants (s'ils existent) et d'exprimer des propriétés sans tenir compte du nombre de composants. La génération d'invariants constitue un domaine de recherche complet et hors du cadre de ce manuscrit.

Dans d'autres approches, basées sur la démonstration de théorèmes [Moo93, PMMG04], la modélisation est faite de manière fonctionnelle. Moore [Moo93] formalise les phénomènes inhérents au décalage entre les horloges de deux unités communicantes par une fonction *async* dans la logique de Boyer-Moore [BM88]. Il modélise ensuite un protocole - en l'occurence Bi-ϕ-M - par une fonction *send* représentant le codage d'un message et une fonction *recv* représentant le décodage de messages. Il prouve, en utilisant l'outil de preuve *Nqthm*, que la composition $recv \circ async \circ send$ de ces trois fonctions est une identité. Pike *et al.* [PMMG04] représentent les communications par une relation dans la logique d'ordre supérieur HOL, et définissent les conditions pour que la relation soit valide.

2.5.2 Vérification des bus

Concernant les protocoles dédiés aux architectures de type bus, Roychoudhury *et al.* utilisent l'outil SMV de vérification de modèles [McM93] pour "débogger" une implémentation universitaire du protocole AMBA AHB [RMK03]. Leur modèle est décrit au niveau *transferts de registres* et sans aucun paramètre. Roychoudhury *et al.* détectent un scénario de famine qui provient plus de leur propre arbitre que du protocole lui-même.

Récemment, Amjad [Amj04] utilise un vérificateur de modèles implémenté dans l'outil de preuves HOL pour vérifier les protocoles AMBA APB et AMBA AHB et un système incluant les deux protocoles. Des propriétés de sûreté sont vérifiées sur chaque protocole par vérification de modèle. L'outil HOL est utilisé pour la vérification de leur composition. Le modèle est là aussi de bas niveau et sans paramètre.

2.5.3 Vérification des réseaux

Concernant les réseaux, par l'aspect extrêmement novateur de ces architectures, très peu de travaux existent aujourd'hui. Gebremichael *et al.* [GVZ05a, GVZ+05b] ont récemment spécifié le protocole Æthereal de Philips dans la logique de PVS. La principale propriété vérifiée est l'absence d'interblocage quels que soient le nombre de maîtres et le nombre d'esclaves.

La prévention des interblocages repose sur la séparation des files en deux classes. Les files de la classe *système* acceptent les paquets de type *SETUP* et *TDOWN* ; celles de la classe *acquittement* acceptent les paquets de type *ACK* et *NACK*. On montre ensuite qu'il n'existe pas de cycle dans le graphe de dépendance de chaque classe. L'envoi de paquets d'acquittement crée une dépendance des files de la classe *système* vers celles de la classe *acquittement*. Comme le montre la figure 2.22 (page 40), un cycle est crée de la classe *acquittement* vers la classe *système* dans le cas où un paquet *TDOWN* est nécessaire pour annuler les réservations. Dans ce cas, le contrôle par crédits prévient de toute situation d'interblocage. Une situation d'interblocage signifie qu'il y a sur le réseau plus de paquets que d'espace disponible pour les accueillir. Cette situation est absurde puisqu'elle implique qu'au moins un maître possède un nombre négatif de crédits.

Par rapport aux travaux présentés dans cette section, notre approche se situe comme suit.

Concernant les architectures de type bus, les contributions de cette thèse sont la spécification et la validation dans la logique d'ACL2 du protocole AMBA AHB [SB03a, SB03c, SB03b]. À la différence des travaux de Roychoudhury *et al.* et de Amjad, notre modèle est plus abstrait et complètement paramétré, (*e.g.* taille du bus, nombre d'unités, taille de la mémoire, ...). Notre représentation est fonctionnelle et nous nous focalisons sur des propriétés de sûreté (exclusion mutuelle des accès au bus, respect des priorités), d'équité d'arbitrage et de cohérence des accès mémoires.

Concernant les réseaux, la contribution majeure de cette thèse est la formalisation dans un style fonctionnel d'un réseau générique [Sch04, SB04b, SB04a, SB05]. À la différence de Gebremichael *et al.* , nous focalisons notre travail sur la validation des spécifications. Leur architecture peut être considérée comme un cas particulier de notre formalisation.

2.6 Vers une formalisation générique

L'enjeu de nos recherches est de formaliser autour d'un modèle générique les différents concepts des architectures de communication. La modélisation générique a pour but d'identifier les principaux composants d'une architecture et les propriétés essentielles inhérentes à chaque composant et à leurs interactions. La formalisation mathématique des concepts permet de prouver formellement la validité des composants et de leur intégration au sein d'un système. Le tout fournit un cadre facilitant la spécification et la validation des architectures de communication lors de leur première phase de conception.

La partie suivante présente la généralisation et la formalisation des communications décrites dans ce chapitre. Elle en présente aussi l'expression dans la logique ACL2.

Deuxième partie

Formalisation fonctionnelle des communications

Chapitre 3

Une formalisation fonctionnelle et générique des communications sur la puce : *GeNoC*

Dans le chapitre précédent, nous avons présenté les notions relatives aux architectures de communication : interfaces, topologie, routage et technique de commutation. Nous les avons illustré sur différents exemples, notamment des systèmes déployés sur la puce. Dans ce chapitre, nous formalisons ces notions dans un style fonctionnel. Le style fonctionnel signifie que tous les composants sont modélisés par des fonctions et les imbrications des appels de ces fonctions représentent les interconnexions des composants. Notre formalisme s'articule autour d'une fonction, nommée *GeNoC*, qui représente une architecture générique de communication. Le caractère générique signifie que *GeNoC* ne modélise pas *une* architecture particulière, mais un *ensemble* d'architectures. La topologie de *GeNoC* est *arbitraire*. Les notions de routage, de commutation et d'interfaces sont représentées chacune par une fonction qui n'a pas de définition *explicite* mais qui est contrainte de satisfaire une série de propriétés. La fonction *GeNoC* est construite à partir de ces fonctions. Sa correction est exprimée par un théorème montrant que chaque message atteint sa destination sans modification de son contenu. Ce théorème est prouvé à partir des propriétés sur les fonctions contraintes.

3.1 Abstractions

Les choix de modélisation et les abstractions définissent le cadre de notre formalisation. Dans un premier temps, les communications sont généralisées à un modèle unique. Dans un second temps, nous présentons la régularisation des topologies.

3.1.1 Abstraction des communications

Pour aborder les différentes architectures de communication avec un même formalisme, ces dernières sont généralisées à un modèle unique représenté sur la figure 3.1. Le modèle des communications est composé de trois parties. Selon la méthodologie proposée par Rowson et Sangiovanni-Vincentelli [RSV97], chaque nœud est séparé en une partie *application*

et une partie *interface*. Cette dernière partie est connectée à l'architecture de communication. Pour différencier les échanges entre une application et une interface d'une part, et les échanges entre deux interfaces d'autre part, les premières communiquent par l'intermédiaire de *messages*, les secondes par l'intermédiaire de *trames* (en anglais, *frames*).

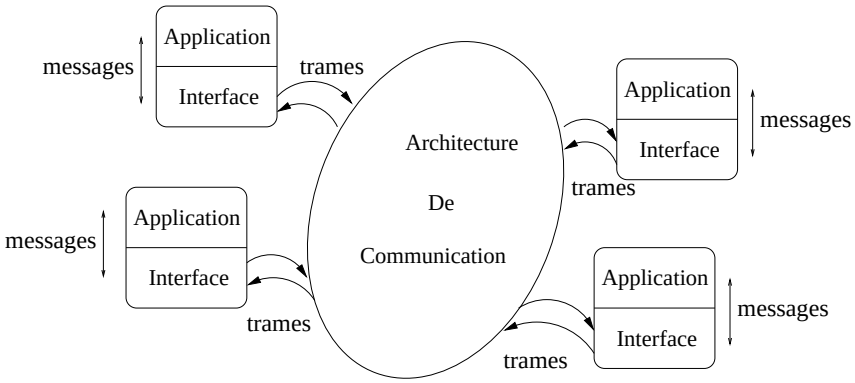


FIG. 3.1 – Modèle des communications.

Les interfaces permettent aux applications de communiquer au moyen de protocoles. L'architecture de communication représente la structure des interconnexions. Elle comprend la topologie, l'algorithme de routage et la politique d'ordonnancement. Le nombre de couples application-interface est arbitraire mais fini.

Les applications représentent les aspects calculatoires et les fonctionnalités d'un nœud. Elles sont classées en deux catégories : actives ou passives. Les applications actives sont généralement des processeurs. Les applications passives sont le plus souvent des mémoires. Dans les réseaux "peer-to-peer", les nœuds ont tous la même structure et possèdent des applications actives et passives. Dans les architectures de type maître/esclave, seuls les nœuds dits "maîtres" possèdent des applications actives et peuvent initier des communications. Les autres nœuds sont dits "esclaves" et se contentent de répondre aux ordres des maîtres. Les communications maître/esclave reposent sur les structures "peer-to-peer". Dans un premier temps, ces dernières sont les seules considérées. Nous discutons en conclusion de ce manuscrit comment étendre notre formalisme pour rendre compte des communications maître/esclave.

Notre considération principale étant les communications entre des nœuds distants, nous ne prenons en compte que celles-ci. Nous supposons que pour toute communication, la destination est différente du nœud d'émission. En ne considérant pas les communications de type maître/esclave, le traîtement de l'information reçue lors de la réception est abstrait et les applications passives (fonctionnalités et mémoire) le sont aussi. Concernant les applications actives, seules leurs opérations de communication sont pertinentes. Les applications actives sont réduites à la liste des opérations de communication "en attente" d'être réalisées.

3.1.2 Régularisation des architectures

Une topologie est dite régulière si tous les nœuds ont la même connectivité ; sinon elle est dite irrégulière. La plupart des topologies sont irrégulières. Soit la grille 4×4 de la figure

3.2, le centre de la grille est régulier. Tous les nœuds sont connectés à quatre voisins et possèdent la même logique de routage. *A contrario*, les nœuds en bordure de la grille ont des connectivités et des algorithmes de routage différents.

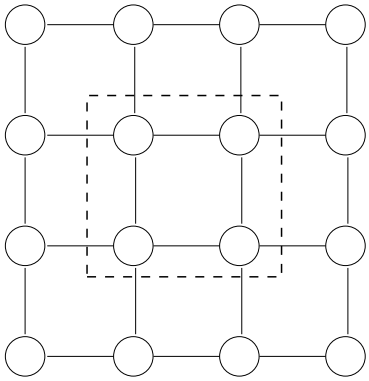


FIG. 3.2 – Réseau irrégulier

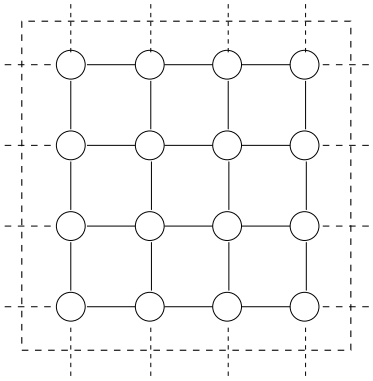


FIG. 3.3 – Réseau régulier

Le principe est de "régulariser" les réseaux irréguliers en généralisant les interconnexions et la logique de routage des nœuds à plus forte connectivité à tous les nœuds du réseau (*cf.* figure 3.3). Le réseau irrégulier de la figure 3.2 est régularisé en généralisant la connectivité et la logique de routage de la partie régulière aux nœuds situés en bordure de la topologie.

Soit $NodeSet$ l'ensemble des nœuds du réseau irrégulier. La régularisation est représentative du réseau originel si toute communication entre deux nœuds appartenant à l'ensemble $NodeSet$, passe uniquement par des nœuds appartenant à $NodeSet$.

3.2 Notations et opérateurs sur les listes

Les listes sont les éléments omniprésents de la formalisation. Nous présentons brièvement les notations et les fonctions les manipulant.

Les lettres l ou L sont utilisées pour représenter une liste ou une liste de listes. Les éléments d'une liste sont représentés par des minuscules, le plus souvent par la lettre e. La liste vide est notée ϵ. Une liste l est une suite finie de k valeurs indexées de 0 à $k-1$. L'élément d'indice i d'une liste l est noté $l[i]$.

$$l = (l[i])_{i \in [0;k-1]} \tag{3.1}$$

La longueur d'une liste est son nombre de valeurs et est obtenue par la fonction $Len(l)$.

Le dernier élément d'une liste l est obtenu par la fonction $Last(l)$.

Le typage d'une liste est défini par l'appartenance de tous ses éléments à un ensemble donné. Le typage des éléments d'une liste l_1 sur un ensemble E est notée par un opérateur $\subseteq_l$ défini comme suit :

$$l_1 \subseteq_l E \equiv \forall i \in [0; Len(l_1) - 1], l_1[i] \in E \tag{3.2}$$

Dans la suite, on dira que l_1 a ses éléments dans E, ou que l_1 est une liste de E.

L'ajout d'un élément e à une liste l crée une liste l', notée $l' = e.l$. L'élément e prend l'indice 0 dans l'. Tout élément de l' d'indice i supérieur à 0 est l'élément de l d'indice $i - 1$. Dans le cas d'une liste de listes, e est une liste. L'opérateur "." est défini de la manière suivante :

$$l' = e.l \equiv l'[0] = e \wedge \forall i \in [1; Len(l)], l'[i] = l[i-1] \tag{3.3}$$

Le prédicat $NoDuplicatesp(l)$ reconnait une liste ne contenant que des éléments distincts.

La concaténation de deux listes l_1 et l_2 de même type est notée $l_1 \sqcup l_2$. La concaténation est une liste de ce type. L'opérateur de concaténation est défini de la manière suivante :

$$l' = l_1 \sqcup l_2 \equiv \left\{ \begin{array}{ll} & \forall i \in [0; Len(l_1) - 1], l'[i] = l_1[i] \\ \wedge & \forall j \in [0; Len(l_2) - 1], l'[Len(l_1) + j] = l_2[j] \end{array} \right. \tag{3.4}$$

Si les listes sont de type différent, leur juxtaposition est obtenue par la fonction $List(l_1, l_2)$.

Un élément e est un élément d'une liste l si et seulement si e est une valeur de l. L'appartenance est notée par l'opérateur $\in_l$ défini comme suit :

$$e \in_l l_1 \Leftrightarrow \exists i \in [0; Len(l_1) - 1], l_1[i] = e \tag{3.5}$$

Une liste l_1 est *incluse* dans une liste l_2 si et seulement si tout élément de l_1 est un élément de l_2. La liste vide, ϵ, est incluse dans toute liste. L'opérateur d'*inclusion* est noté $\sqsubseteq$ et défini de la manière suivante :

$$l_1 \sqsubseteq l_2 \Leftrightarrow \forall i \in [0; Len(l_1) - 1], l_1[i] \in_l l_2 \vee l_1 = \epsilon \tag{3.6}$$

Cette notion d'inclusion est similaire à celle de sous-ensemble. La liste $(1\ 1\ 1)$ est incluse dans la liste (1). La liste $(3\ 2)$ est incluse dans la liste $(1\ 2\ 3)$.

La liste l privée de la première occurence d'un élément e est notée $l \setminus e$. La définition de l'opérateur $\setminus$ est la suivante :

$$l \setminus e \triangleq \left\{ \begin{array}{ll} l & \textbf{si } e \notin_l l \\ l_1 & \textbf{si } l = e.l_1 \\ e_1.(l_1 \setminus e) & \textbf{si } l = e_1.l_1 \wedge e_1 \neq e \end{array} \right. \tag{3.7}$$

La liste l' contenant les éléments communs à deux listes l_1 et l_2 est notée $l' = l_1 \sqcap l_2$. Cette liste conserve l'ordre des éléments de l_1. Par exemple, $(1\ 2\ 5\ 3) \sqcap (1\ 2\ 1\ 3\ 4) = (1\ 2\ 3)$.

$$l_1 \sqcap l_2 \triangleq \left\{ \begin{array}{ll} \epsilon & \textbf{si } l_1 = \epsilon \vee l_2 = \epsilon \\ l'_1 \sqcap l_2 & \textbf{si } l_1 = e.l'_1 \wedge e \notin_l l_2 \\ e.(l'_1 \sqcap (l_2 \setminus e)) & \textbf{si } l_1 = e.l'_1 \wedge e \in_l l_2 \end{array} \right. \tag{3.8}$$

Dans le cas où les éléments e d'une liste L sont eux même des listes, la liste de tous les éléments de même indice i dans e est notée $L_{\lfloor i}$. La plupart du temps, la signification des éléments de e est donnée par un identifiant. Pour ne pas avoir à mémoriser les indices de chaque élément de e, on utilise plutôt l'identifiant que l'indice. Considérons que e est une liste composée d'une clé, d'un nom et d'un prénom, soit $e = (cle\ nom\ prenom)$. Soit L une liste d'éléments e de cette sorte, la liste des clés est notée $L_{\lfloor cle}$, la liste des noms $L_{\lfloor nom}$ et celle des prénoms $L_{\lfloor prenom}$.

Souvent, une liste est construite par application d'une fonction f à chaque élément d'une liste l. Cette manipulation correspond à une fonction φ d'ordre supérieur qui prend pour arguments une fonction f et une liste l. La fonction φ retourne la liste des résultats de l'application de f à chaque élément de l. La définition de φ est la suivante. Si la liste l est vide, φ produit ϵ. Sinon, la liste l contient au moins un élément e et s'écrit comme l'ajout de e à une liste l', soit $e.l'$. La fonction φ crée la liste composée du résultat de f appliqué à l'élément e et du résultat de l'application de φ au reste de l, soit l' [1]. La fonction f pouvant être compliquée, il n'est pas toujours pratique de la formuler explicitement. Souvent, il est suffisant d'exprimer la modification apportée à chaque élément. Pour alléger la notation, l'application de la fonction φ est notée par l'opérateur Λ défini comme suit :

$$\bigwedge_{e\in l} f(e) \equiv \varphi(l,f) \triangleq \begin{cases} \epsilon & \textbf{si } l = \epsilon \\ f(e).\varphi(l',f) & \textbf{sinon } l = e.l' \end{cases} \tag{3.9}$$

Par exemple, soit l une liste dont chaque élément e est consitué de deux entiers, $e = (x_1\ x_2)$. La liste l' dont chaque élément est la somme $x_1 + x_2$ est obtenue par l'application, à chaque élément de l, de la composée de la fonction $+$ avec les fonctions permettant d'extraire x_1 et x_2. Avec l'opérateur Λ, on autorisera d'écrire simplement :

$$l' = \bigwedge_{e\in l}(e[0] + e[1])$$

Les notations concernant les listes sont récapitulées dans le tableau 3.1 [2].

Nom	Rôle
$e.l$	ajout de l'élément e à la liste l
$l_1 \subseteq_l E$	typage des éléments de l_1 sur l'ensemble E
$l_1 \sqcup l_2$	concaténation de l_1 et l_2
$e \in_l l_1$	e est un élément de la liste l_1
$l_1 \sqsubseteq l_2$	inclusion de l_1 dans l_2
$l_1 \sqcap l_2$	extraction de l_1 par rapport à l_2
$List(l_1, l_2)$	juxtaposition des listes l_1 et l_2
$Len(l)$	retourne le nombre d'éléments contenus dans l
$Last(l)$	retourne le dernier élément de l
$NoDuplicatesp(l)$	identifie une liste l sans doublon
ϵ	liste vide
$l[i]$	élément de la liste l, $0 \leq i \leq len(l) - 1$

TAB. 3.1 – Notations et fonctions pour manipuler les listes

3.3 La fonction *GeNoC*

La fonction *GeNoC* représente une architecture de communication régulière et générique. Tous les nœuds sont identiques, leur nombre est arbitraire et fini. La topologie, l'algorithme

[1] Cette fonction correspond à la fonction MAPCAR du langage LISP.

[2] Les tableaux résumant les objets manipulés dans notre modèle sont rassemblés en annexe A.

de routage, la technique de commutation sont quelconques. La fonction *GeNoC* modélise la transmission de messages de leur source à leur destination. Elle prend pour principal argument la liste des messages émis aux nœuds sources et retourne la liste des résultats reçus aux nœuds destinations. Sa définition repose principalement sur les fonctions suivantes :

1. deux fonctions représentant les interfaces ; une fonction (notée *send*) pour l'injection de trames sur le réseau, une fonction (notée *recv*) pour les réceptionner,
2. une fonction (notée *Routing*) représentant l'algorithme de routage et la topologie,
3. une fonction (notée *Scheduling*) représentant la technique de commutation.

Le caractère générique est obtenu en ne définissant pas ces fonctions de manière explicite. Ces fonctions sont uniquement définies par une série de propriétés, appelées *obligations de preuve* ou simplement *contraintes*.

Avant de continuer, nous donnons brièvement les principes sous-jacents la définition de chacune de ces fonctions.

Interfaces. Leur modélisation est simple. La fonction *send* représente l'encapsulation d'un message dans une trame, selon les différentes couches du modèle OSI. La fonction *recv* représente le décodage de cette trame pour récupérer le message émis. La principale contrainte associée à ces fonctions exprime que la partie réceptrice doit être capable d'extraire l'information codée, soit que la composition de la fonction *recv* avec la fonction *send* ($recv \circ send$) est une identité.

Routage. Le routage est représenté par l'application successive de déplacements unitaires. Pour tout couple formée d'une source et d'une destination, la fonction de routage calcule *toutes* les routes possibles autorisées par les déplacements unitaires. La principale contrainte associée au routage exprime le fait que toute route entre une source s et une destination d débute effectivement en s et emprunte uniquement des nœuds existants pour aboutir en d.

Ordonnancement. La technique de commutation reflète le niveau de parallélisme du réseau. La commutation par paquets permet plus de parallélisme que la commutation par circuits dans le sens où dans la première un paquet n'utilise qu'une ressource (canal ou file) et la deuxième réserve plusieurs ressources pour un même paquet. La technique de commutation est ainsi généralisée à la notion d'*ordonnancement*. La fonction d'ordonnancement extrait d'une liste de trames, la sous-liste de trames qui peuvent voyagées simultanément dans le réseau. La principale contrainte associée à la fonction d'ordonnancement exprime l'exclusion mutuelle entre les trames ordonnancées et celles retardées.

Dans la sous-section suivante, nous exposons les différentes étapes pour passer d'un message à un résultat. Nous précisons les objets consommés et produits à chaque étape. Ensuite, nous montrons comment ces différents objets et fonctions sont traités par la fonction *GeNoC*.

3.3.1 Du message au résultat

Transactions. Les applications actives sont formalisées par la liste des opérations de communication "en attente". Une opération de communication est modélisée par une *transaction*.

Une transaction précise le message, msg, à envoyer, le nœud, A, d'où il est émis et le nœud, B, auquel il est destiné. Nous considérons qu'il n'y a qu'une seule application active et qu'une seule application passive par nœud. Par contre, plusieurs transactions peuvent être associées à une seule application active. Une transaction représente l'intention de l'application A d'envoyer le message msg à l'application B. Pour pouvoir suivre un message de sa source à sa destination, chaque transaction est identifiée de manière unique par un naturel id. Aucune restriction n'est imposée sur la structure d'un message qui est défini sur un domaine arbitraire, noté $\mathcal{D}_{msg}$. Soit $GenNodeSet$ le domaine de définition des nœuds (*c.f.* sous-section suivante), une transaction, t, est un quadruplet $(id\ A\ msg\ B)$ défini sur le domaine $\mathbb{N} \times GenNodeSet \times \mathcal{D}_{msg} \times GenNodeSet$. Les accesseurs des éléments d'une transaction sont notés par des fonctions. L'origine A et la destination B d'une transaction t sont notées $Org_{\mathcal{T}}(t)$ et $Dest_{\mathcal{T}}(t)$. De même, l'identifiant est noté $Id_{\mathcal{T}}(t)$ et le message $Msg_{\mathcal{T}}(t)$.

Une liste $\mathcal{T}$ de transactions représente toutes les transactions de toutes les applications actives de tous les nœuds du réseau. Une liste de transactions est valide si :

1. les éléments de la liste $\mathcal{T}_{\lfloor id}$ des identifiants sont des naturels distincts deux à deux,
2. l'ensemble des nœuds origines est un sous-ensemble des nœuds du réseau, de même pour l'ensemble des nœuds destinations,
3. l'origine est différente de la destination.

Formellement, une liste valide de transactions est reconnue par le prédicat $\mathcal{T}_{lstp}$:

Définition 3.1 Définition de $\mathcal{T}_{lstp}$.

$$\mathcal{T}_{lstp}(\mathcal{T}, NodeSet) \triangleq \left\{ \begin{array}{ll} & \mathcal{T}_{\lfloor id} \subseteq_l \mathbb{N} \wedge NoDuplicatesp(\mathcal{T}_{\lfloor id}) \\ \wedge & \forall t \in_l \mathcal{T}, Org_{\mathcal{T}}(t), Dest_{\mathcal{T}}(t) \in NodeSet \\ \wedge & \forall t \in_l \mathcal{T}, Org_{\mathcal{T}}(t) \neq Dest_{\mathcal{T}}(t) \end{array} \right.$$

Le domaine de définition $\mathcal{D}_{\mathcal{T}}$ d'une liste de transactions est l'ensemble des listes satisfaisant ce prédicat.

Les fonctions et objets concernant la formalisation des applications actives sont récapitulés sur le tableau 3.2.

Nom	Objet
transaction	quadruplet $t = (Id\ Org\ Msg\ Dest)$
$\mathcal{D}_{msg}$	domaine de définition d'un message
$\mathcal{D}_{\mathcal{T}}$	domaine de définition de $\mathcal{T}$
$\mathcal{T}_{lstp}(\mathcal{T}, NodeSet)$	identifie une liste valide de transactions
$\mathcal{T}$	liste de transactions
$Id_{\mathcal{T}}(t)$, $Org_{\mathcal{T}}(t)$, $Msg_{\mathcal{T}}(t)$, $Dest_{\mathcal{T}}(t)$	accesseurs des éléments d'une transaction t
$\mathcal{T}_{\lfloor id}$	liste des identifiants de $\mathcal{T}$
$NodeSet$	ensemble des nœuds du réseau

TAB. 3.2 – Fonctions et objets concernant les applications actives

Missives. Chaque message est transmis à une interface pour être envoyé sur le réseau. La trame ainsi envoyée est construite par la fonction *send* de l'interface (voir section 3.5 page 63 pour plus de détails sur les interfaces). Aucune restriction n'est imposée sur la structure des trames. Le domaine de définition, $\mathcal{D}_{frm}$, d'une trame est arbitraire. Une *missive* est une transaction où le message a été converti en une trame. Une missive m est définie sur le domaine $\mathbb{N} \times \mathit{GenNodeSet} \times \mathcal{D}_{frm} \times \mathit{GenNodeSet}$. Les fonctions $Id_{\mathcal{M}}(m)$, $Org_{\mathcal{M}}(m)$, $Frm_{\mathcal{M}}(m)$ et $Dest_{\mathcal{M}}(m)$ permettent d'obtenir les différents éléments d'une missive m. Une liste valide de missives est reconnue par un prédicat $\mathcal{M}_{lstp}$ similaire à $\mathcal{T}_{lstp}$. La seule différence est que les messages sont remplacés par des trames. Une liste de missives est notée $\mathcal{M}$. Le domaine de définition d'une liste de missives est l'ensemble des listes satisfaisant le prédicat $\mathcal{M}_{lstp}$. Il est noté $\mathcal{D}_{\mathcal{M}}$.

Nom	Objet
missive	$m = (Id\ Org\ Frm\ Dest)$
$\mathcal{D}_{frm}$	domaine de définition d'une trame
$\mathcal{M}_{lstp}(\mathcal{M}, NodeSet)$	identifie une liste valide de missives
$\mathcal{D}_{\mathcal{M}}$	domaine de définition de $\mathcal{M}$
$\mathcal{M}$	liste de missives
$Id_{\mathcal{M}}(m)$, $Org_{\mathcal{M}}(m)$, $Frm_{\mathcal{M}}(m)$, $Dest_{\mathcal{M}}(m)$	accesseurs des éléments de m

TAB. 3.3 – Fonctions et objets concernant les missives

Voyages. Pour chaque trame, la fonction de routage (*c.f.* section 3.6 page 64) propose une ou plusieurs routes. Une route r (ou un chemin) est une liste de nœuds, soit une liste qui a ses éléments dans $\mathit{GenNodeSet}$ ($r \subseteq_l \mathit{GenNodeSet}$). Le domaine de définition d'une liste de routes est noté $\mathcal{C}$. L'association d'une liste de routes à une trame constitue un *voyage*. Un *voyage* est un triplet constitué d'un identifiant, d'une trame et de la liste de ses routes possibles dans le réseau, soit $(\mathit{Id\ Frm\ Routes})$. Une liste de voyages est notée $\mathcal{V}$. Un voyage est défini sur le domaine $\mathbb{N} \times \mathcal{D}_{frm} \times \mathcal{C}$. Les éléments d'un voyage v sont obtenus par les fonctions $Id_{\mathcal{V}}(v)$, $Frm_{\mathcal{V}}(v)$ et $Routes_{\mathcal{V}}(v)$. Une liste de voyages est valide si la liste $\mathcal{V}_{\lfloor id}$ des identifiants est une liste de naturels et ne comporte aucun doublon. Le prédicat suivant reconnaît les listes valides de voyage :

Définition 3.2 Définition de $\mathcal{V}_{lstp}$.

$$\mathcal{V}_{lstp}(\mathcal{V}) \triangleq \left\{ \begin{array}{ll} & \mathcal{V}_{\lfloor id} \subseteq_l \mathbb{N} \\ \wedge & \mathit{NoDuplicatesp}(\mathcal{V}_{\lfloor id}) \end{array} \right.$$

Le domaine de définition d'une liste de voyages est l'ensemble des listes satisfaisant ce prédicat. Le domaine de définition d'une liste de voyages est notée $\mathcal{D}_{\mathcal{V}}$.

Les fonctions et objets concernant les voyages sont récapitulés sur le tableau 3.4.

Nom	Objet
voyage	triplet $v = (Id\ Frm\ Routes)$
$\mathcal{V}$	liste de voyages
$\mathcal{D}_{\mathcal{V}}$	domaine de définition de $\mathcal{V}$
$\mathcal{C}$	domaine de définition d'une liste de routes
$\mathcal{V}_{lstp}(\mathcal{V})$	reconnaît une liste valide de voyages
$Id_{\mathcal{V}}(v)$, $Frm_{\mathcal{V}}(v)$,$Routes_{\mathcal{V}}(v)$	accesseurs des éléments de v

TAB. 3.4 – Fonctions et objets concernant les voyages

Tentatives. La liste, $\mathcal{V}$, des voyages est transmise à la fonction d'ordonnancement qui la divise en une liste de voyages à effectuer, notée *Scheduled*, et une liste de voyage à retarder, notée *Delayed* (*c.f.* section 3.7 page 67). Pour assurer la vivacité du modèle, il est nécessaire de prouver qu'il n'existe pas de message qui soit indéfiniment retardé. Une solution possible est que la fonction d'ordonnancement effectue toujours au moins un voyage, soit que la liste *Scheduled* ne soit jamais vide. Cette solution restreint notre modèle. Il existe des politiques d'ordonnancement, comme dans le protocole Ethernet par exemple, pour lesquelles il est possible de retarder toutes les trames émises à un instant donné. Ceci se traduit par une liste *Scheduled* vide. Dans Ethernet, la solution est de donner à chaque station un nombre limité de *tentatives* pour envoyer une trame. Si toutes les tentatives sont consommées, le transfert est abandonné. Nous optons pour une solution similaire. Dans *GeNoC*, un nombre de tentatives n'est pas associé à une trame ou une transaction mais à un nœud. Chaque nœud possède un nombre arbitraire mais fini de tentatives pour exécuter toutes les transactions en attente à ce nœud. Une fois toutes les tentatives consommées à un nœud, les transactions restantes de ce nœud sont dites *avortées*. Une liste de transactions avortées est notée $\mathcal{A}$. Une liste de nombres de tentatives est notée *att*. L'indice dans *att* du nombre de tentatives disponibles pour le nœud n est donné par une fonction injective $MapAtt : GenNodeSet \hookrightarrow [0; Len(att) - 1]$. Formellement, ceci est noté $att[MapAtt(n)]$. Par abus de notation, nous le notons simplement $att[n]$. Le domaine de définition des listes de nombres de tentatives est noté *AttLst*. La fonction $SumOfAttempts(att)$ additionne tous les éléments de *att*. Elle calcule la somme de toutes les tentatives disponibles dans le réseau.

Les fonctions et objets concernant les tentatives sont récapitulés sur le tableau 3.5.

Nom	Objet
att	liste des tentatives de tous les nœuds
$MapAtt(n)$	donne l'indice i dans *att* des tentatives pour le nœud n
$AttLst$	domaine de définition de *att*
$SumOfAttempts(att)$	additionne toutes les tentatives restantes de tous les nœuds
$\mathcal{A}$	liste de transactions avortées

TAB. 3.5 – Fonctions et objets concernant les tentatives

Résultats. Chaque voyage effectué est transmis à une interface. Cette dernière décode la trame du voyage et produit un message. Ceci est modélisé par la fonction *recv* (*c.f.* section 3.5 à propos des interfaces). Un résultat modélise l'information reçue à un nœud. Un résultat est un triplet constitué d'un identifiant, d'une destination et d'un message, soit $(Id\ Dest\ Msg)$. Un résultat est défini sur le domaine $\mathbb{N} \times GenNodeSet \times \mathcal{D}_{msg}$. Les éléments d'un résultat rst sont obtenus par les fonctions $Id_{\mathcal{R}}(rst)$, $Dest_{\mathcal{R}}(rst)$ et $Msg_{\mathcal{R}}(rst)$. L'identifiant permet d'associer un résultat à une transaction. Si l'architecture de communication est correcte, le résultat est la transaction initiale moins l'origine ; le message initial est reçu sans modification par la destination prévue.

Une liste de résultats est notée $\mathcal{R}$. Une liste de résultats est valide si :

1. la liste $\mathcal{R}_{\lfloor id}$ des identifiants est une liste de naturels et ne comporte pas de doublon,
2. la destination de chaque résultat appartient à l'ensemble des nœuds du réseau.

Le prédicat suivant reconnaît une liste valide de résultats :

Définition 3.3 Définition de $\mathcal{R}_{lstp}$.

$$\mathcal{R}_{lstp}(\mathcal{R}, NodeSet) \triangleq \left\{ \begin{array}{ll} & \mathcal{R}_{\lfloor id} \subseteq_l \mathbb{N} \wedge NoDuplicatesp(\mathcal{R}_{\lfloor id}) \\ \wedge & \forall rst \in_l \mathcal{R}, Dest_{\mathcal{R}}(rst) \in NodeSet \end{array} \right.$$

Le domaine de définition d'une liste de résultat est l'ensemble des listes satisfaisant ce prédicat. Le domaine de définition de $\mathcal{R}$ est noté $\mathcal{D}_{\mathcal{R}}$.

Les fonctions et objets concernant les résulats sont récapitulés sur le tableau 3.6.

Nom	Objet
résultat	triplet $rst = (Id\ Dest\ Msg)$
$\mathcal{R}$	liste de résultats
$\mathcal{D}_{\mathcal{R}}$	domaine de définition de $\mathcal{R}$
$\mathcal{R}_{lstp}(\mathcal{R}, NodeSet)$	reconnaît une liste valide de résultats
$Id_{\mathcal{R}}(rst)$, $Dest_{\mathcal{R}}(rst)$,$Msg_{\mathcal{R}}(rst)$	accesseurs des éléments de rst

TAB. 3.6 – Fonctions et objets concernant les résultats

3.3.2 Déroulement de la fonction *GeNoC*

La fonction *GeNoC* est représentée sur la figure 3.4. Ce schéma présente deux nœuds parmi l'ensemble des nœuds du réseau. Chaque nœud est divisé en une application et une interface ; elle-même représentée par les fonctions *send* et *recv*. Au centre du modèle, l'architecture de communication est divisée en deux parties constituant chacune une sous-fonction de *GeNoC*. La fonction *Routing* représente le calcul des routes et la fonction *Scheduling* la politique d'ordonnancement.

La fonction *GeNoC* prend pour arguments une liste $\mathcal{T}$ non vide de transactions, l'ensemble *NodeSet* des nœuds d'un réseau et une liste de nombres de tentatives *att*. Elle retourne une liste $\mathcal{R}$ de résultats et une liste $\mathcal{A}$ de missives avortées.

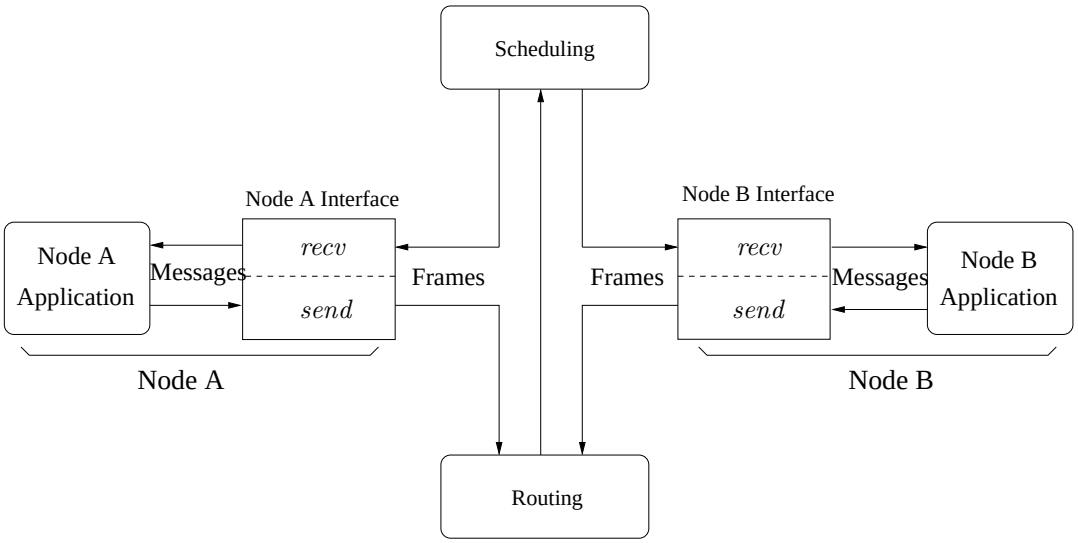


FIG. 3.4 – *GeNoC* : un réseau générique

Le déroulement de la fonction *GeNoC* est illustré par la figure 3.5. Pour chaque message de la liste des transactions $\mathcal{T}$, la fonction *ComputeMissives* applique la fonction *send* produisant une liste $\mathcal{M}$ de missives. La fonction *Routing* associe une liste de routes à chaque trame de $\mathcal{M}$ produisant une liste $\mathcal{V}$ de voyages. La fonction *Scheduling* divise $\mathcal{V}$ en une liste *Scheduled* de voyages à effectuer et une liste *Delayed* de voyages qui sont retardés. Cette séparation est fonction des priorités des transactions et du nombre de tentatives disponibles à chaque nœud. Les messages reçus sont calculés par la fonction *ComputeResults* par application de la fonction *recv* à toutes les trames de *Scheduled*. La fonction *ComputeResults* produit la liste $\mathcal{R}$ des résultats. Les voyages de la liste *Delayed* sont convertis en une liste de missives. Si la somme des tentatives restantes n'est pas nulle, ces missives sont passées de nouveau à la fonction *Routing* [3]. Sinon, elles constituent la liste $\mathcal{A}$ des missives avortées.

La correction de cette fonction est exprimée par deux propriétés. Tout d'abord, les messages reçus sont identiques aux messages envoyés. Ensuite, un message est reçu par la bonne destination. Formellement, ceci se traduit par l'expression ci-dessous montrant que tout résultat rst est obtenu à partir d'une unique transaction t ayant un identifiant, un message et une destination identiques à ceux de rst.

$$\forall rst \in_l \mathcal{R}, \exists! t \in_l \mathcal{T}, \begin{cases} & Id_{\mathcal{R}}(rst) = Id_{\mathcal{T}}(t) \\ \wedge & Msg_{\mathcal{R}}(rst) = Msg_{\mathcal{T}}(t) \\ \wedge & Dest_{\mathcal{R}}(rst) = Dest_{\mathcal{T}}(t) \end{cases} \tag{3.10}$$

[3] Dans notre modélisation la fonction *Routing* calcule toutes les routes possibles entre une source et une destination *indépendamment* des routes déjà calculées. Revenir à la fonction *Routing* semble inutile, on pourrait revenir directement à la fonction *Scheduling*. De manière générale, on peut imaginer une évolution de *GeNoC* dans laquelle le calcul des routes pourrait dépendre des routes déjà calculées, du temps ou de tout autre paramètre. Revenir à la fonction *Routing* ne complique que légèrement la preuve finale, et nous avons opté pour cette solution plus facilement évolutive.

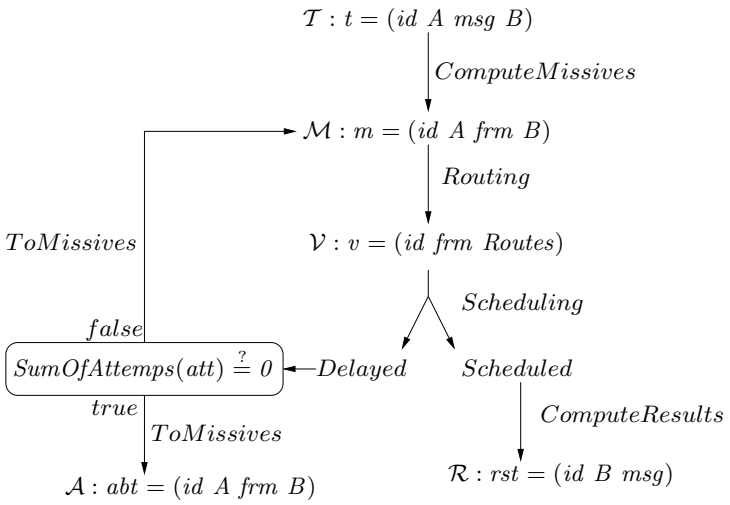


FIG. 3.5 – Fonctions de *GeNoC*

3.4 Nœuds et paramètres

Le domaine de définition des nœuds est arbitraire et noté $GenNodeSet$. Les éléments de ce domaine sont reconnus par le prédicat $ValidNodep$, qui en est la fonction caractéristique :

$$\forall x,\ ValidNodep(x) \Leftrightarrow x \in GenNodeSet \tag{3.11}$$

L'ensemble des nœuds d'un réseau particulier est noté $NodeSet$. Il est généré à partir des paramètres pms définis sur un domaine arbitraire $GenParams$ et à partir de la fonction $NodeSetGen$. Les paramètres sont reconnus par le prédicat $ValidParamsp$ et constituent une base génératrice pour $NodeSet$. La fonctionnalité de $NodeSetGen$ est la suivante :

$$NodeSetGen : GenParams \rightarrow \mathcal{P}(GenNodeSet) \tag{3.12}$$

Ces fonctions sont valides si quels que soient les paramètres reconnus par $ValidParamsp$, tout élément produit par $NodeSetGen$ appartient au domaine $GenNodeSet$ (c'est à dire satisfait le prédicat $ValidNodep$) :

Obligation de preuve 3.1 Définition de $NodeSet$.

$$\forall pms,\ ValidParamsp(pms) \Rightarrow \forall x \in NodeSetGen(pms),\ ValidNodep(x)$$

Les fonctions et termes utilisés concernant les nœuds sont résumés dans le tableau 3.7. Dans la suite de ce chapitre, pour chaque obligation de preuve, lemme ou théorème dans lequel intervient $NodeSet$, il faut lire $NodeSetGen(pms)$ à la place de $NodeSet$ et ajouter comme hypothèse $ValidParamsp(pms)$.

Nom	Objet
$ValidNodep(x)$	identifie x comme faisant partie des nœuds du réseau
$NodeSetGen(pms)$	construit l'ensemble des nœuds d'un résau
$GenNodeSet$	domaine de définition des nœuds d'un réseau
$NodeSet$	ensemble des nœuds d'un réseau particulier
$ValidParamsp(pms)$	identifie pms comme paramètres d'un réseau
$GenParams$	domaine de définition de pms

TAB. 3.7 – Fonctions et objets concernant les nœuds et les paramètres

3.5 Formalisation des Interfaces

Les interfaces communiquent avec deux types de composants, une application et une autre interface, elles sont modélisées par deux fonctions. La fonction *send* construit une *trame* à partir d'un message et la fonction *recv* construit un *message* à partir d'une *trame*.

La fonctionnalité de *send* est la suivante :

$$send : \mathcal{D}_{msg} \rightarrow \mathcal{D}_{frm} \tag{3.13}$$

La fonctionnalité de *recv* est la suivante :

$$recv : \mathcal{D}_{frm} \rightarrow \mathcal{D}_{msg} \tag{3.14}$$

La fonction *send* code un message dans une trame, la fonction *recv* décode la trame pour récupérer le message. La contrainte sur les fonctions représentant les interfaces est que cette composition est une identité. Au niveau des interfaces, l'obligation de preuve suivante doit être levée :

Obligation de preuve 3.2 Validité des fonctions des interfaces.

$$\forall msg \in \mathcal{D}_{msg}, recv \circ send(msg) = msg$$

Les fonctions et termes utilisés pour les interfaces sont résumés dans le tableau 3.8.

Nom	Objet
message	(noté msg) information entre une application et une interface
$\mathcal{D}_{msg}$	domaine de définition de $message$
frame	(ou $trame$ ou simplement frm) information entre les interfaces
$\mathcal{D}_{frm}$	domaine de définition de frm
$send$	construit une trame à partir d'un message
$recv$	construit un message à partir d'une trame

TAB. 3.8 – Fonctions et objets concernant les interfaces

3.6 Formalisation du routage

3.6.1 Principes et critères de correction

Formalisation du routage. Dans le cas des algorithmes déterministes, pour toute trame contenue à un nœud s et destinée à un nœud d, la logique de routage d'un réseau régulier fournit un unique nœud constituant la prochaine étape dans la route de s à d. Cette logique est représentée par une fonction $\mathcal{L}(s, d)$. La liste des nœuds visités pour tout voyage d'un nœud s vers un nœud d est obtenue par applications successives de la fonction $\mathcal{L}$ jusqu'à ce que la destination soit atteinte, soit tant que $\mathcal{L}(s, d) \neq d$. La route de s à d est :

$$s, \mathcal{L}(s, d), \mathcal{L}(\mathcal{L}(s, d), d), \mathcal{L}(\mathcal{L}(\mathcal{L}(s, d), d), d), \ldots, d$$

Une route est calculée par une fonction ρ_{det} qui applique récursivement la fonction $\mathcal{L}$ de la source à la destination. La fonction ρ_{det} est définie comme suit :

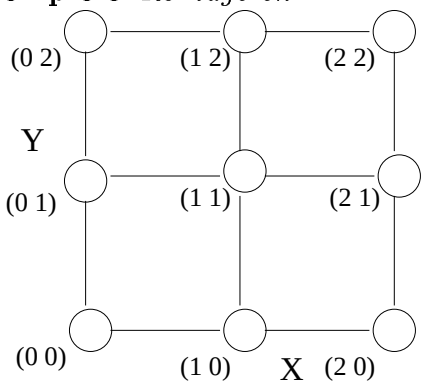
$$\rho_{det}(s, d) \triangleq \begin{cases} d & \textbf{si } s = d \\ s.\rho_{det}(\mathcal{L}(s, d), d) & \textbf{sinon} \end{cases} \tag{3.15}$$

Dans le cas adaptatif, la logique de routage propose à chaque nœud intermédiaire plusieurs "prochains" nœuds. Plusieurs routes sont possibles entre une source s et une destination d. Dans ce cas, le routage est représenté par une fonction ρ_{ndet} qui calcule *toutes* les routes possibles entre les nœuds s et d. Nous illustrons le cas adaptif au chapitre 5, section 5.1.2, page 108. Le cas adaptatif est aussi discuté en conclusion de ce manuscrit.

De manière générale, l'algorithme de routage est représenté par une fonction ρ prenant pour arguments un nœud source s et un nœud destination d. Cette fonction retourne la liste des routes possibles entre s et d. Sa fonctionnalité est la suivante :

$$\rho : GenNodeSet \times GenNodeSet \rightarrow \mathcal{C} \tag{3.16}$$

Exemple 8 *Routage en XY.*



Considérons la grille 3 × 3 ci-contre. Dans cette configuration, la logique de routage $\mathcal{L}_{xy}$ fonctionne comme suit. À un nœud s donné, une trame destinée au nœud d est routée vers la gauche si l'abscisse de s est plus petite que celle de d ; vers la droite si elle est plus grande. Si s et d ont la même abscisse, la trame est routée vers le haut si l'ordonnée de s est plus petite que celle de d ; vers le bas si elle est plus grande. Si s et d sont identiques la logique produit d. La définition de $\mathcal{L}_{xy}$ est donnée ci-dessous.

$$\mathcal{L}_{xy}(s, d) \triangleq \begin{cases} d & \textbf{si } s = d \\ (s_x + 1, s_y) & \textbf{si } s_x < d_x \\ (s_x - 1, s_y) & \textbf{si } s_x > d_x \\ (s_x, s_y + 1) & \textbf{si } s_x = d_x \wedge s_y < d_y \\ (s_x, s_y - 1) & \textbf{si } s_x = d_x \wedge s_y > d_y \end{cases}$$

La fonction ρ_{xy} est définie comme l'application récursive de la fonction $\mathcal{L}_{xy}$:

$$\rho_{xy}(s,d) \triangleq \begin{cases} d & \textbf{si } s = d \\ s.\rho_{xy}(\mathcal{L}_{xy}(s,d),d) & \textbf{sinon} \end{cases}$$

Terminaison du routage. La fonction ρ est le plus souvent récursive et il est important d'en prouver la terminaison. La raison est double. Tout d'abord, montrer que le routage termine assure qu'aucune trame ne voyage indéfiniment dans le réseau. Cette preuve est nécessaire pour assurer la vivacité du réseau. Ensuite, les logiques sur lesquelles reposent la plupart des assistants de preuve imposent la preuve de la terminaison de toute fonction récursive.

Soient S un ensemble et $\prec_S$ une relation d'ordre total sur S. La relation $\prec_S$ est une relation de *bon ordre* sur S si tout sous-ensemble de S admet un élément minimum. On dira aussi que $(S, \prec_s)$ forme une structure bien ordonnée. Typiquement, la preuve de terminaison d'une fonction est effectuée en montrant qu'une *mesure* est décroissante sur une structure bien ordonnée pour tout appel récursif de la dite fonction. Une mesure est une fonction définie sur le produit des domaines des arguments de la fonction vers S. Considérons le cas déterministe et la fonction ρ_{det}. Pour la preuve de la terminaison de ρ_{det}, nous considérons les mesures définies sur $GenNodeSet \times GenNodeSet$ vers S (dans la plupart des cas S est l'ensemble des naturels). Soit $(S, \prec_S)$ une structure bien ordonnée et $mes(s,d)$ une mesure sur S. Pour prouver que ρ_{det} termine, il faut prouver que la condition "gouvernant" l'appel récursif implique que mes décroit. La condition gouvernant l'appel récursif est $s \neq d$. L'obligation de preuve suivante doit être satisfaite :

Obligation de preuve 3.3 Conditions de terminaison de ρ_{det}.

$$\forall s, d \in GenNodeSet, \exists mes : GenNodeSet \times GenNodeSet \rightarrow S,$$
$$s \neq d \Rightarrow mes(\mathcal{L}(s,d),d) \prec_S mes(s,d)$$

La généralisation de ce résultat au cas adaptatif est discutée en conclusion de ce manuscrit.

Exemple 9 *Considérons la fonction ρ_{xy} de l'exemple précédent. La structure composée de la relation strictement inférieure et des naturels, $(\mathbb{N}, <)$, est bien ordonnée. La mesure $m(s,d) = |d_x - s_x| + |d_y - s_y|$ appartient aux naturels et décroit à chaque application de $\mathcal{L}_{xy}(s,d)$.*

Correction du routage. La correction d'une route est définie par rapport à une missive. Une route est correcte par rapport à une missive m donnée si elle débute par l'origine de m, finit par la destination de m et que chaque nœud de la route appartient à l'ensemble des nœuds du réseau. Toute route correcte contient au moins deux nœuds. Le prédicat suivant définit ces conditions :

Définition 3.4 ValidRoutep.

$$ValidRoutep(r,m,NodeSet) \triangleq \begin{cases} & r[0] = Org_{\mathcal{M}}(m) \\ \wedge & Last(r) = Dest_{\mathcal{M}}(m) \\ \wedge & r \subseteq_l NodeSet \wedge Len(r) \geq 2 \end{cases}$$

Que le routage soit déterministe ou adaptatif, toutes les routes produites par la fonction de routage ρ doivent satisfaire ce prédicat. L'obligation de preuve suivante doit être levée :

Obligation de preuve 3.4 Validité des routes produites par ρ.

$$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet)$$
$$\Rightarrow \forall m \in_l \mathcal{M}, \forall r \in_l \rho(Org_{\mathcal{M}}(m), Dest_{\mathcal{M}}(m)), ValidRoutep(r, m, NodeSet)$$

3.6.2 Définition et validation de la fonction *Routing*

L'algorithme de routage de *GeNoC* est représenté par la fonction *Routing* qui prend comme arguments une liste de missives et l'ensemble *NodeSet* des nœuds du réseau. Elle retourne une liste de voyages dans laquelle une liste de routes est associée à chaque missive. La fonctionalité de *Routing* est la suivante :

$$Routing : \mathcal{D}_{\mathcal{M}} \times \mathcal{P}(GenNodeSet) \rightarrow \mathcal{D}_{\mathcal{V}} \tag{3.17}$$

La fonction *Routing* construit une liste de voyages à partir de l'identifiant, la trame, l'origine et la destination des missives.

Définition 3.5 Fonction Routing
$Routing(\mathcal{M}, NodeSet) \triangleq$

$$\bigwedge_{m \in_l \mathcal{M}} List(Id_{\mathcal{M}}(m), Frm_{\mathcal{M}}(m), \rho(Org_{\mathcal{M}}(m), Dest_{\mathcal{M}}(m)))$$

Au niveau des types de données il faut prouver que la fonction *Routing* produit une liste valide de voyages si la liste de missives est elle-même valide.

Obligation de preuve 3.5 Type de Routing.

$$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet) \Rightarrow \mathcal{V}_{lstp}(Routing(\mathcal{M}, NodeSet))$$

La définition de la fonction *Routing* préserve les propriétés établies sur la fonction ρ précédentes. La fonction *Routing* termine et les routes de chaque voyage satisfont le prédicat $ValidRoutep$. Dans une liste de missives, les identifiants sont tous uniques. À chaque voyage v produit par la fonction *Routing*, il correspont une unique missive m telle que l'identifiant de m soit identique à celui de v et la trame de v est identique à celle de m.

Théorème 3.1 Correspondance Missive/Voyage.

$$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet) \Rightarrow$$
$$\forall v \in_l Routing(\mathcal{M}, NodeSet), \exists! m \in_l \mathcal{M}, Id_{\mathcal{V}}(v) = Id_{\mathcal{M}}(m) \wedge Frm_{\mathcal{V}}(v) = Frm_{\mathcal{M}}(m)$$

Preuve: Par définition de *Routing*. □

Les voyages retardés par la fonction d'ordonnancement - mais produits par la fonction *Routing* - sont converties en missives par la fonction $ToMissives$. Cette dernière construit les missives de la manière suivante. Elle reprend l'identifiant et la trame des voyages. L'origine et la destination d'une missive sont le premier et le dernier nœud d'une route. La fonction $ToMissives$ est la réciproque de la fonction *Routing*.

Théorème 3.2 Routing Missives.

$$\forall \mathcal{M}, \mathcal{M}_{lstp} \Rightarrow ToMissives \circ Routing(\mathcal{M}, NodeSet) = \mathcal{M}$$

Preuve: La trame n'est pas modifiée par la fonction *Routing*. Puisque cette dernière satisfait le prédicat $ValidRoutep$ pour toutes les routes de tous les voyages qu'elle produit, le premier et le dernier nœud de toute route sont égaux à l'origine et à la destination de la missive initiale. □

Les fonctions et prédicats concernant le routage sont récapitulés dans le tableau 3.9.

Nom	rôle
$\mathcal{L}(i,j)$	déplacements unitaires autorisés dans le réseau
$\rho(i,j)$	construit toutes les routes de i à j
$Routing(\mathcal{M}, NodeSet)$	construit toutes les routes pour toutes les missives
$ValidRoutep(r, m, NodeSet)$	identifie une route valide par rapport à la missive m
$ToMissives(\mathcal{V})$	convertit la liste de voyages $\mathcal{V}$ en une liste de missives

TAB. 3.9 – Fonctions et notations pour le routage

3.7 Politique d'ordonnancement

La politique d'ordonnancement participe à la gestion des conflits dans un réseau. Elle définit l'ensemble des communications pouvant être effectuées *simultanément*. Formellement, les communications en cours satisfont un *invariant*. Une communication est *ordonnancée* si son ajout préserve l'invariant. Une politique d'ordonnancement est correcte si elle préserve cet invariant à *tout instant* et *quel que soit l'état d'occupation du réseau*. À chaque politique correspond un invariant. La formalisation de la politique d'ordonnancement suppose l'existence de cet invariant qui n'est pas représenté de manière explicite. La fonction d'ordonnancement, *Scheduling*, extrait d'une liste de voyages, $\mathcal{V}$, la sous-liste, *Scheduled*, des voyages *ordonnancés*, c'est à dire ceux qui satisfont l'invariant. Le reste de $\mathcal{V}$ définit la liste, *Delayed*, des voyages *retardés*.

3.7.1 Définition de la fonction *Scheduling*

La fonction *Scheduling* prend pour arguments la liste de voyages produite par la fonction *Routing* et la liste des nombres de tentatives *att*. Elle retourne deux listes de voyages : la liste *Scheduled* et la liste *Delayed*. Elle met aussi à jour la liste des nombres de tentatives *att*. La fonctionnalité de *Scheduling* est la suivante :

$$Scheduling : \mathcal{D}_{\mathcal{V}} \times AttLst \rightarrow \mathcal{D}_{\mathcal{V}} \times \mathcal{D}_{\mathcal{V}} \times AttLst \tag{3.18}$$

Pour tout voyage ordonnancé possédant plusieurs routes possibles, la fonction d'ordonnancement ne conserve qu'une seule route. Pour ne pas introduire un nouveau type de donnée, nous considérons les voyages ordonnancés comme des voyages "classiques", c'est à dire contenant une liste de routes, même si cette liste n'a qu'un seul élément.

3.7.2 Validation de la fonction *Scheduling*

La projection d'un vecteur selon une de ses dimensions est notée par une fonction π_i^j qui a la fonctionnalité suivante :

$$\pi_i^j : \mathcal{D}_1 \times \mathcal{D}_2 \times \cdots \times \mathcal{D}_j \rightarrow \mathcal{D}_i \tag{3.19}$$

Cette fonction extrait la valeur sur le domaine $\mathcal{D}_i$ d'une liste de j valeurs définies sur autant de domaines. Par exemple, soit une fonction $f : \mathcal{D}_1 \times \mathcal{D}_2 \rightarrow \mathcal{D}_3 \times \mathcal{D}_4$. Nous noterons $\pi_1^2 \circ f(x_1, x_2)$ et $\pi_2^2 \circ f(x_1, x_2)$ les projections de f selon $\mathcal{D}_3$ et $\mathcal{D}_4$.

Tout d'abord, si la liste $\mathcal{V}$ est une liste valide de voyages, les listes *Scheduled* et *Delayed* le sont aussi.

Obligation de preuve 3.6 Type de *Scheduled* et *Delayed*.

Soient $Scheduled = \pi_1^3 \circ Scheduling(\mathcal{V}, att)$ *et*
$Delayed = \pi_2^3 \circ Scheduling(\mathcal{V}, att)$, *alors :*

$$\forall \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}) \Rightarrow \mathcal{V}_{lstp}(Scheduled) \wedge \mathcal{V}_{lstp}(Delayed)$$

À chaque "tour" d'ordonnancement, tous les voyages de $\mathcal{V}$ sont analysés. Si plusieurs voyages sont associés à un même nœud, ce nœud consomme une tentative pour l'ensemble de ces voyages. À chaque appel de *Scheduling*, une tentative est consommée à chaque nœud. Si toutes les tentatives n'ont pas été consommées, la somme des tentatives restantes après application de la fonction *Scheduling* est strictement inférieure à la somme des tentatives avant application de *Scheduling*. Ceci se traduit par l'obligation de preuve suivante :

Obligation de preuve 3.7 La fonction *Scheduling* consomme au moins une tentative.

Soit $natt = \pi_3^3 \circ Scheduling(\mathcal{V}, att)$, *alors :*

$$\begin{array}{ll} & SumOfAttempts(att) \neq 0 \\ \rightarrow & SumOfAttempts(natt) < SumOfAttempts(att) \end{array}$$

Les voyages *reportés* sont convertis en missives dans l'appel récursif de *GeNoC*. Le résultat de cette conversion doit être une sous-liste de la liste de missives initiale. Pour que cette conversion soit valide il faut que les informations contenues dans les voyages retardés soient identiques à celles contenues initialement dans la liste $\mathcal{V}$. La liste des voyages retardés doit être une sous-liste de $\mathcal{V}$. De manière formelle, on s'assure que pour tout voyage retardé *dtr*, il existe un unique voyage initial v tel que *dtr* et v ont le même identifiant, la même trame et les mêmes routes. Soit, l'obligation de preuve suivante :

Obligation de preuve 3.8 Correction des voyages retardés.
Soit $Delayed = \pi_2^3 \circ Scheduling(\mathcal{V}, att)$, *alors :*

$$\forall \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}) \Rightarrow \forall dtr \in_l Delayed, \exists! v \in_l \mathcal{V}, \left\{ \begin{array}{ll} & Id_{\mathcal{V}}(dtr) = Id_{\mathcal{V}}(v) \\ \wedge & Frm_{\mathcal{V}}(dtr) = Frm_{\mathcal{V}}(v) \\ \wedge & Routes_{\mathcal{V}}(dtr) = Routes_{\mathcal{V}}(v) \end{array} \right.$$

Généralement, la fonction d'ordonnancement ne garde qu'une seule route pour chaque voyage ordonnancé. De ce fait, la liste *Scheduled* n'est pas exactement une sous-liste de la liste initiale de voyages $\mathcal{V}$. Les identifiants et les trames ne sont pas changés. On vérifie que la route, ou de manière plus générale, les routes d'un voyage ordonnancé appartiennent aux routes associées au voyage initial correspondant. De manière formelle, on s'assure que pour tout voyage ordonnancé *str*, il existe un unique voyage initial v tel que *str* et v ont le même identifiant et la même trame et les routes associées à *str* sont parmi les routes associées à v.

Obligation de preuve 3.9 Correction des voyages ordonnancés.
Soit $Scheduled = \pi_1^3 \circ Scheduling(\mathcal{V}, att)$, *alors :*

$$\forall \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}) \Rightarrow \forall str \in_l Scheduled, \exists! v \in_l \mathcal{V}, \left\{ \begin{array}{ll} & Id_{\mathcal{V}}(str) = Id_{\mathcal{V}}(v) \\ \wedge & Frm_{\mathcal{V}}(str) = Frm_{\mathcal{V}}(v) \\ \wedge & Routes_{\mathcal{V}}(str) \sqsubseteq Routes_{\mathcal{V}}(v) \end{array} \right.$$

Puisque les routes des voyages de *Scheduled* sont des routes de $\mathcal{V}$, la fonction *Scheduling* préserve la correction des routes. Si les routes de $\mathcal{V}$ satisfont le prédicat $ValidRoutep$, les routes de la liste *Scheduled* aussi.

Le but de l'ordonnancement est de séparer une liste de voyages en deux listes. Cette séparation doit être exclusive et un voyage ne peut appartenir à la liste *Scheduled et* à la liste *Delayed*. L'intersection des identifiants de ces deux listes doit être vide.

Obligation de preuve 3.10 Exclusion mutuelle entre *Delayed* **et** *Scheduled*.
Soient $Scheduled = \pi_1^3 \circ Scheduling(\mathcal{V}, att)$ *et*
$Delayed = \pi_2^3 \circ Scheduling(\mathcal{V}, att)$, *alors :*

$$\forall \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}) \Rightarrow Delayed_{\lfloor id} \sqcap Scheduled_{\lfloor id} = \epsilon$$

3.8 Définition et validation de *GeNoC*

La définition de la fonction *GeNoC* suit la figure 3.5 (page 62) qui est reprise sur la figure 3.6. L'appel récursif dans *GeNoC* ne concerne que les fonctions *Routing* et *Scheduling*. Ces deux fonctions sont regroupées dans une fonction récursive nommée $GeNoC_t$. La preuve de la correction de la fonction *GeNoC* repose sur un lemme (le Lemme 3.1 ci-après) établissant la correction de la fonction $GeNoC_t$. Cette fonction prend pour principal argument une liste de missives et retourne une liste de voyages *effectués* et une liste de voyages *avortés*. Les premiers sont transformés en résultats dans la fonction *GeNoC*, et les seconds en missives. Le lemme 3.1 constitue le point central de la preuve de *GeNoC* et ne mentionne que les voyages effectués. Il prouve que pour chaque voyage effectué il existe une unique missive de

la liste initiale telle que cette missive ait le même identifiant, la même trame que ce voyage et que la destination de cette missive soit égale au dernier nœud de la route de ce voyage. Pour simplifer la preuve, nous définissons la fonction $GeNoC_t^{nt}$ qui est la sous-fonction de $GeNoC_t$ calculant les voyages effectués.

La fonction $GeNoC_t^{nt}$ accumule récursivement les voyages ordonnancés par la fonction *Scheduling*. Les voyages retardés constituent le principal argument de l'appel récursif. Intuitivement, la correction du résultat de $GeNoC_t^{nt}$ repose sur la correction des voyages accumulés et sur la correction de ceux qui le seront dans l'appel récursif. Le lemme 3.1 établit une formule ϕ à propos des voyages effectués, soit le résultat de la fonction $GeNoC_t^{nt}$. La preuve est faite par induction sur la structure de la définition de $GeNoC_t^{nt}$. Concernant les voyages ordonnancés, la preuve est directement déduite des contraintes et d'un théorème. Pour les voyages retardés, leur correction est obtenue en utilisant l'hypothèse d'induction, une contrainte et un théorème.

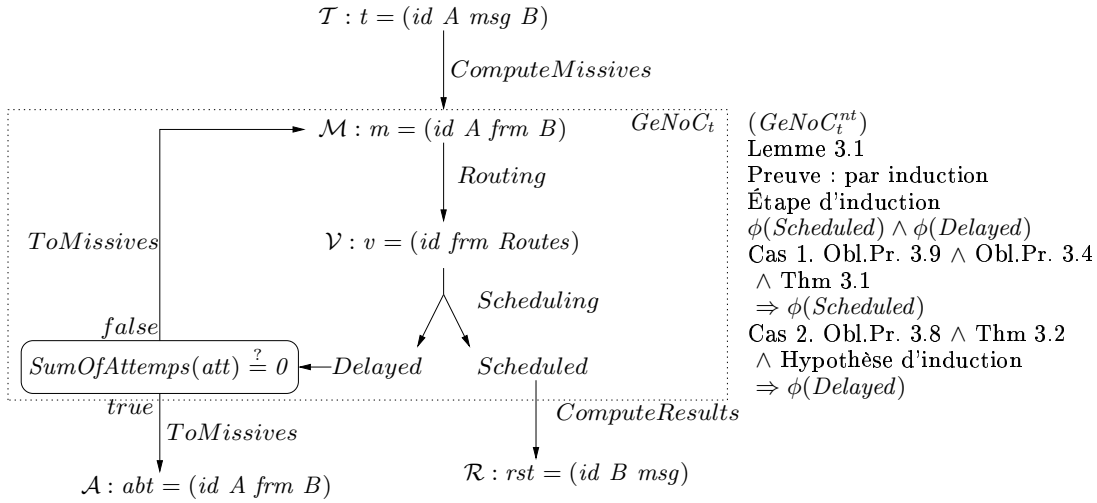


FIG. 3.6 – Preuve de *GeNoC*

3.8.1 Composition des fonctions *Routing* et *Scheduling*

La composition des fonctions *Routing* et *Scheduling* est définie par la fonction $GeNoC_t$. Elle prend pour arguments une liste $\mathcal{M}$ de missives, l'ensemble *NodeSet* des nœuds du réseau, la liste *att* des nombres de tentatives et une liste $\mathcal{V}$ de voyages qui est initialement vide. Elle retourne deux listes : une liste de voyages contenant les trames reçues par les nœuds destinataires des missives de $\mathcal{M}$ et une liste contenant les missives avortées. Sa fonctionnalité est la suivante :

$$GeNoC_t : \mathcal{D}_{\mathcal{M}} \times \mathcal{P}(GenNodeSet) \times AttLst \times \mathcal{D}_{\mathcal{V}} \rightarrow \mathcal{D}_{\mathcal{V}} \times \mathcal{D}_{\mathcal{M}} \tag{3.20}$$

$GeNoC_t$ est une fonction récursive définie comme suit. Si toutes les tentatives ont été consommées, $GeNoC_t$ retourne les voyages accumulés dans la liste $\mathcal{V}$ et la liste des missives restantes, *i.e.* des missives avortées. Sinon, les voyages obtenus par la fonction *Routing* sont

passés à la fonction *Scheduling*. Les voyages ordonnancés sont ajoutés à la liste $\mathcal{V}$. Les voyages retardés sont convertis en missives et consituent un argument de l'appel récursif de $GeNoC_t$. Les autres arguments sont les listes att et $\mathcal{V}$ mises à jour.

Définition 3.6 $GeNoC_t$
$GeNoC_t(\mathcal{M}, NodeSet, att, \mathcal{V}) \triangleq$
if $SumOfAttempts(att) = 0$ **then**
$List(\mathcal{V}, \mathcal{M})$
else
Let$(ScheduledRtg\ DelayedRtg\ att_1)$ **be**
$Scheduling(Routing(\mathcal{M},NodeSet),att)$ **in**
$GeNoC_t(ToMissives(DelayedRtg),\ NodeSet,\ att_1, ScheduledRtg \sqcup \mathcal{V})$
endif

Dans la suite, nous nous intéressons principalement à la liste $\mathcal{R}$ des voyages terminés, soit le premier argument retourné par la fonction $GeNoC_t$. Pour alléger la lecture, nous utilisons l'abréviation suivante :

$$G = \pi_1^2 \circ GeNoC_t(\mathcal{M}, NodeSet, att, \epsilon) \tag{3.21}$$

La correction de la fonction $GeNoC_t$ est obtenue si pour tout élément ctr de la liste $\mathcal{R}$ des voyages terminés la trame et le dernier nœud de la route de ctr [4] sont égaux à la trame et à la destination de la missive m de $\mathcal{M}$ ayant le même identifiant que ctr. Soit, le lemme ci-dessous :

Lemme 3.1 Correction de $GeNoC_t$**.**

$$\forall ctr \in_l G, \exists! m \in_l \mathcal{M}, \left\{ \begin{array}{ll} & Id_{\mathcal{V}}(ctr) = Id_{\mathcal{M}}(m) \\ \wedge & Frm_{\mathcal{V}}(ctr) = Frm_{\mathcal{M}}(m) \\ \wedge & \forall r \in_l Routes_{\mathcal{V}}(ctr), Last(r) = Dest_{\mathcal{M}}(m) \end{array} \right.$$

Preuve: La preuve est faite par induction sur la structure de $GeNoC_t^{nt}$ (voir ci-après). Le principe est que l'hypothése d'induction, grâce à une partie des obligations de preuve, permet de prouver ce lemme pour les voyages retardés. Pour les voyages terminés, la preuve est déduite d'une partie des obligations de preuve. Avant d'attaquer ces preuves, nous définissons quelques abréviations.

Abréviations. Pour rendre la preuve plus lisible, la conjonction du lemme est notée $\Theta(ctr, m)$:

$$\Theta(ctr, m) = \left\{ \begin{array}{ll} & Id_{\mathcal{V}}(ctr) = Id_{\mathcal{M}}(m) \\ \wedge & Frm_{\mathcal{V}}(ctr) = Frm_{\mathcal{M}}(m) \\ \wedge & \forall r \in_l Routes_{\mathcal{V}}(ctr), Last(r) = Dest_{\mathcal{M}}(m) \end{array} \right.$$

[4] Notons que pour rester homogène dans les notations, un voyage est toujours constitué d'une liste de routes, même si cette liste n'a qu'un seul élément.

L'appel à la fonction *Routing* est abrégé par *Rtg* :

$$Rtg = Routing(\mathcal{M}, NodeSet)$$

La composition de *Routing*, *Scheduling* et π_2^3 est notée *DelayedRtg* :

$$DelayedRtg = \pi_2^3 \circ Scheduling(Rtg, att)$$

La composition de *Routing*, *Scheduling* et π_1^3 est notée *ScheduledRtg* :

$$ScheduledRtg = \pi_1^3 \circ Scheduling(Rtg, att)$$

Fonctions sans accumulateurs. La fonction $GeNoC_t$ est directement récursive (en anglais, *tail recursive*). De plus, le dernier argument de $GeNoC_t$ (soit ϵ) ne permet aucune substitution dans l'hypothèse d'induction. Une solution est de généraliser le lemme en remplaçant ϵ par une variable ayant certaines particularités. Une autre solution est d'écrire une fonction qui ne soit pas directement récursive mais qui calcule le premier élément de $GeNoC_t$. Dans ce cas, l'accumulateur n'est pas nécessaire. Nous choisissons cette dernière solution et utilisons la fonction $GeNoC_t^{nt}$ ci-dessous :

Définition 3.7 $GeNoC_t^{nt}$
$GeNoC_t^{nt}(\mathcal{M}, NodeSet, att) \triangleq$
if $SumOfAttempts(att) = 0$ **then**
ϵ
else
Let$(ScheduledRtg\ DelayedRtg\ att_1)$ **be**
$Scheduling(Routing(\mathcal{M}, NodeSet), att)$ **in**
$GeNoC_t^{nt}(ToMissives(DelayedRtg), NodeSet, att_1) \sqcup ScheduledRtg$
endif

On prouve facilement par induction que cette fonction calcule effectivement le premier argument de $GeNoC_t$:

$$GeNoC_t(\mathcal{M}, NodeSet, att, \mathcal{V}) = GeNoC_t^{nt}(\mathcal{M}, NodeSet, att) \sqcup \mathcal{V}$$

En particulier, si $\mathcal{V} = \epsilon$, on a :

$$G = GeNoC_t^{nt}(\mathcal{M}, NodeSet, att)$$

Pour prouver le lemme 3.1, il suffit de prouver que la fonction $GeNoC_t^{nt}$ satisfait la propriété suivante :

$$\forall ctr \in_l GeNoC_t^{nt}(\mathcal{M}, NodeSet, att), \exists! m \in_l \mathcal{M}, \Theta(ctr, m) \tag{3.22}$$

Cas de base. Cette formule est prouvée par induction sur la structure de la fonction $GeNoC_t^{nt}$. Le cas de base suppose que toutes les tentatives ont été consommées. Dans ce cas, $GeNoC_t^{nt}$ produit la liste vide ϵ. Tout voyage *ctr* est égal à ϵ. Le cas de base est trivialement satisfait pour $m = \epsilon$: il se réduit à $\Theta(\epsilon, \epsilon)$.

Étape d'induction. L'hypothèse d'induction est obtenue en substituant, dans la formule 3.22, $\mathcal{M}$ par $ToMissives(DelayedRtg)$ et att par att_1. Cette hypothèse suppose aussi que la somme des tentatives restantes n'est pas nulle. Notons G_σ^{nt} le résultat de l'application de ces subsitutions à $GeNoC_t^{nt}$, soit :

$$G_\sigma^{nt} = GeNoC_t^{nt}(ToMissives(DelayedRtg), NodeSet, att_1)$$

Finalement, l'étape d'induction consiste dans la preuve de la formule suivante :

$$\begin{array}{ll} & SumOfAttempts(att) \neq 0 \\ \wedge & \forall ctr \in_l G_\sigma^{nt}, \exists! m \in_l ToMissives(DelayedRtg), \Theta(ctr, m) \\ \Rightarrow & \forall ctr \in_l GeNoC_t^{nt}(\mathcal{M}, NodeSet, att), \exists! m \in_l \mathcal{M}, \Theta(ctr, m) \end{array} \tag{3.23}$$

Puisque la somme des tentatives n'est pas nulle, $GeNoC_t^{nt}$ prend la forme de son appel récursif, soit :

$$GeNoC_t^{nt}(\mathcal{M}, NodeSet, att) = G_\sigma^{nt} \sqcup ScheduledRtg$$

La formule 3.23 devient :

$$\begin{array}{ll} & \forall ctr \in_l G_\sigma^{nt}, \exists! m \in_l ToMissives(DelayedRtg), \Theta(ctr, m) \\ \Rightarrow & \forall ctr \in_l (G_\sigma^{nt} \sqcup ScheduledRtg), \exists! m \in_l \mathcal{M}, \Theta(ctr, m) \end{array} \tag{3.24}$$

Cette formule est satisfaite si sa conclusion est prouvée pour les voyages ordonnancés, soit ceux de *Scheduled* et pour les voyages retardés, soit ceux de G_σ^{nt}. Considérons tout d'abord les voyages ordonnancés.

Sous-but 1 : voyages ordonnancés.

$$\forall ctr \in_l ScheduledRtg, \exists! m \in_l \mathcal{M}, \Theta(ctr, m) \tag{3.25}$$

L'obligation de preuve 3.9 (page 69) est valable pour toute liste, $\mathcal{V}$, valide de voyages. En particulier, elle est valable si $\mathcal{V}$ est le résultat de la fonction *Routing*. Dans ce cas, la route de chaque voyage ordonnancé est une route produite par la fonction *Routing*. D'après l'obligation de preuve 3.4 (page 66), cette route satisfait le prédicat $ValidRoutep$. La fonction *Routing* préserve la correspondance entre les voyages et les missives (théorème 3.1 page 66), chaque voyage de *Scheduled* correspond bien à une unique missive.

Sous-but 2 : voyages retardés.

$$\begin{array}{ll} & \forall ctr \in_l G_\sigma^{nt}, \exists! m \in_l ToMissives(DelayedRtg), \Theta(ctr, m) \\ \Rightarrow & \forall ctr \in_l G_\sigma^{nt}, \exists! m \in_l \mathcal{M}, \Theta(ctr, m) \end{array} \tag{3.26}$$

Ce sous-but revient à montrer que chercher une missive dans $ToMissives(DelayedRtg)$ est équivalent à chercher une missive dans $\mathcal{M}$. L'obligation de preuve 3.8 (page 69) est valable pour toute liste de voyage, $\mathcal{V}$. En particulier, elle est valable si $\mathcal{V}$ est produite par la fonction *Routing*. Il est équivalent de chercher un voyage dans la liste $DelayedRtg$ et dans la liste Rtg, soit dans la liste $ToMissives(DelayedRtg)$ et la liste $ToMissives(Rtg)$. D'après, le théorème 3.2 (page 67) $ToMissives(Rtg)$ produit $\mathcal{M}$.

Les preuves de ces deux sous-but concluent la preuve du lemme 3.1. □

3.8.2 Définition et validation de *GeNoC*

La fonction *GeNoC* prend pour arguments une liste, $\mathcal{T}$, de transactions, l'ensemble, *NodeSet*, des nœuds du réseau, la liste *att* des nombres de tentatives. Elle retourne la liste, $\mathcal{R}$ des résultats et la liste, $\mathcal{A}$ contenant les missives avortées. Sa fonctionnalité est la suivante :

$$GeNoC : \mathcal{D}_{\mathcal{T}} \times \mathcal{P}(GenNodeSet) \times AttLst \rightarrow \mathcal{D}_{\mathcal{R}} \times \mathcal{D}_{\mathcal{M}} \tag{3.27}$$

La fonction *ComputeMissives* applique la fonction *send* au message de chaque transaction de la liste $\mathcal{T}$. Cette fonction produit une liste de *missives* à partir des transactions initiales. Sa fonctionnalité est la suivante :

$$ComputeMissives : \mathcal{D}_{\mathcal{T}} \rightarrow \mathcal{D}_{\mathcal{M}} \tag{3.28}$$

Elle est définie comme suit :

Définition 3.8 ComputeMissives.
$ComputeMissives(\mathcal{T}) \triangleq$

$$\bigwedge_{t \in_l \mathcal{T}} List(Id_{\mathcal{T}}(t), Org_{\mathcal{T}}(t), send(Msg_{\mathcal{T}}(t)), Dest_{\mathcal{T}}(t))$$

La fonction *ComputeResults* applique la fonction *recv* à chaque trame d'une liste de voyages pour produire une liste des résultats. Sa fonctionnalité est la suivante :

$$ComputeResults : \mathcal{D}_{\mathcal{V}} \rightarrow \mathcal{D}_{\mathcal{R}} \tag{3.29}$$

Elle est définie comme suit :

Définition 3.9 ComputeResults.
$ComputeResults(\mathcal{V}) \triangleq$

$$\bigwedge_{tr \in_l \mathcal{V}} List(Id_{\mathcal{V}}(tr), Last(Routes_{\mathcal{V}}(tr)), recv(Frm_{\mathcal{V}}(tr)))$$

La fonction *GeNoC* est définie au moyen de ces fonctions et de $GeNoC_t$. La fonction *ComputeMissives* fournit le premier argument de $GeNoC_t$ à partir de la liste de transactions $\mathcal{T}$. Le dernier argument de $GeNoC_t$ est la liste vide. La liste des résultats est obtenue par la fonction *ComputeResults*. Les missives avortées sont produites par la fonction $GeNoC_t$. La définition de *GeNoC* est la suivante :

Définition 3.10 *GeNoC*.
GeNoC($\mathcal{T}$, NodeSet, att) $\triangleq$
Let (*Responses Aborted*) **be**
$GeNoC_t$(ComputeMissives($\mathcal{T}$), NodeSet, att, ϵ) **in**
List(ComputeResults(Responses), Aborted)

La correction de *GeNoC* est définie par l'expression 3.10 définie en section 3.4.

Théorème 3.3 Correction de $GeNoC$**.**
Soit $\mathcal{R} = \pi_1^2 \circ GeNoC(\mathcal{T}, NodeSet, att)$*, alors :*

$$\forall rst \in_l \mathcal{R}, \exists! t \in_l \mathcal{T}, \left\{ \begin{array}{ll} & Id_{\mathcal{R}}(rst) = Id_{\mathcal{T}}(t) \\ \wedge & Msg_{\mathcal{R}}(rst) = Msg_{\mathcal{T}}(t) \\ \wedge & Dest_{\mathcal{R}}(rst) = Dest_{\mathcal{T}}(t) \end{array} \right.$$

Preuve: Le dernier terme de la conjonction est directement obtenu par le lemme 3.1. Ce même lemme permet de déduire que les trames produites par la fonction $ComputeMissives$ sont identiques à celles converties en message par la fonction $ComputeResults$. L'obligation de preuve 3.2 sur les interfaces permet d'affirmer que les messages des résultats sont identiques aux messages des transactions initiales. □

3.9 Conclusion

Dans ce chapitre, les communications sur la puce ont été exprimées selon une fonction nommée $GeNoC$. Cette fonction est structurée en sous-fonctions représentant les principaux aspects des communications : encapsulation et "décapsulation" des informations, routage et ordonnancement. Ces composantes d'un système sont représentées de façon générique dans le sens où leur définition n'est pas donnée explicitement. Des contraintes - exprimées sous forme d'obligations de preuve - sont associées à chaque sous-fonction. Toute définition satisfaisant les contraintes est une particularisation valide du modèle générique. La validité de $GeNoC$, ne dépendant que des contraintes, est maintenue. De plus, les sous-fonctions de $GeNoC$ étant indépendantes les unes des autres, l'approche proposée est modulaire tant au sujet des définitions que des preuves.

Dans le chapitre suivant, nous présentons l'expression $GeNoC$ dans la logique ACL2.

Chapitre 4

Expression de $GeNoC$ dans la logique d'ACL2

Dans le chapitre précédent nous avons présenté notre formalisme qui s'articule principalement autour de la fonction $GeNoC$. Nous en avons exprimé les fonctions et les théorèmes dans un formalisme mathématique classique. Ce chapitre aborde la réalisation de $GeNoC$ dans le système ACL2. Ce modèle est légèrement différent de celui présenté au chapitre précédent, en raison des spécificités de la logique ACL2. Le formalisme du chapitre précédent fait appel aux quantificateurs et à la logique du second ordre. La logique ACL2 étant du premier ordre et sans quantificateurs, des adaptations sont nécessaires. Afin d'éviter toute ambiguité, l'implémentation de $GeNoC$ dans ACL2 est notée $GeNoC^\flat$.

Ce chapitre est destiné à une personne ayant une expérience avec un démonstrateur de théorèmes mais n'étant pas forcément une experte. L'objectif de ce chapitre est d'une part de présenter rapidement les spécificités du système ACL2 - son fonctionnement et sa logique - et d'autre part d'illustrer l'expression de constructions du second ordre dans ACL2. Ce dernier propose des mécanismes permettant de "simuler" certains aspects de la logique du second ordre, comme la quantification sur des fonctions, par exemple.

4.1 ACL2 : un langage, une logique, un outil de preuves

Les initiales ACL2 signifient "A Computational Logic for Applicative Common LISP" soit "une logique calculatoire pour un sous-ensemble applicatif de Common LISP". ACL2 est le nom d'un langage fonctionnel, d'une logique du premier ordre et d'un outil d'aide à la preuve. ACL2 est l'évolution du système de Boyer-Moore vers un outil plus propice aux applications industrielles [KM97a]. La principale évolution fut l'introduction de mécanismes permettant l'exécution des programmes ACL2 à une vitesse proche de l'exécution de programmes C. L'efficacité d'exécution d'ACL2 s'est avérée un atout important dans de nombreux projets industriels (*e.g.* [Rus98, WGH01]).

Cette section apporte une connaissance très partielle du système ACL2. Pour une plus grande compréhension, la littérature propose des ouvrages et articles introduisant les principes d'ACL2 [KMM00a, KM05], ainsi que de nombreuses applications (*e.g.* [KMM00b, BKM96]).

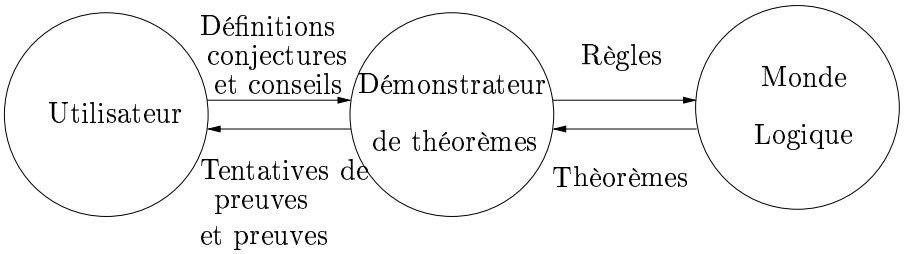


FIG. 4.1 – Interaction avec ACL2

Dans les sous-sections suivantes, nous présentons brièvement le système ACL2 dans son ensemble avant d'en présenter la logique. Nous insistons sur un principe utilisé intensément dans $GeNoC^b$: le principe d'encapsulation.

4.1.1 Le système ACL2

Le système ACL2 est constitué de trois entités : l'utilisateur, le démonstrateur et le "monde logique" (*c.f.* figure 4.1[1]). Ce monde est constitué de règles et de stratégies qui ont été développées précédemment. L'utilisateur soumet la formule qu'il souhaite démontrer à ACL2 qui utilise les règles du monde logique pour trouver une preuve du théorème. Si la preuve de la formule existe, ACL2 transforme ce théorème en une règle de preuve et ajoute celle-ci au monde logique. Dans le futur, ACL2 pourra utiliser cette règle pour une autre preuve. L'utilisateur modifie le monde logique du démonstrateur.

Le langage ACL2 est un sous-ensemble sans effet de bord de Common LISP. La notation utilisée est préfixée. L'expression mathématique $x * f(y, z)$ est écrite `(* x (f y z))`. ACL2 fonctionne par sessions. Le contenu du monde logique définit l'état d'une session. Le monde logique est constitué de l'ensemble des fonctions, constantes, variables et les règles connues par ACL2. Initialement, le monde logique est constitué d'une théorie contenant les fonctions usuelles, les axiomes initiaux de la logique et quelques règles élémentaires. Cette théorie est nommée par les auteurs "ground zero".

ACL2 se présente comme un interpréteur de commandes. À l'invite d'un "prompt"[2], l'utilisateur soumet une commande ; celle-ci est interprétée par ACL2 qui imprime une réponse. Par exemple, la concaténation des listes `(1 2 3)` et `(4 5 6)` s'effectue par la commande suivante :

```
ACL2 !>(append '(1 2 3) '(4 5 6))
(1 2 3 4 5 6)
ACL2 !>
```

L'apostrophe indique à ACL2 de traiter l'expression qui suit comme une constante et non comme un appel de fonction.

Cette commande ne modifie par l'état de la session. Modifier l'état d'une session ou étendre la théorie "ground zero" constitue un évènement. Les évènements les plus courants

[1] La figure est issue de [KMM00a].

[2] Le prompt ACL2 est `ACL2 !>`.

sont la définition d'une nouvelle fonction (commande `defun`), la preuve d'un théorème (commande `defthm`), ou l'encapsulation (commande `encapsulate`, décrite en section 4.1.3). Les évènements peuvent être regroupés dans une bibliothèque. L'importation d'une bibliothèque constitue aussi un évènement et s'effectue par la commande `include-book`.

La commande "defthm" est la clé pour l'interaction avec l'outil de preuves. Lorsque cette commande est soumise, ACL2 applique une série de procédures pour tenter de prouver le théorème à partir du contenu du monde logique. Les procédures les plus employées sont la réecriture, des procédures de décision pour l'arithmétique linéaire, la généralisation et l'induction. ACL2 est complètement automatique dans le sens où une fois qu'une tentative de preuve est lancée, aucune interaction avec ACL2 n'est possible. Cette tentative échoue dans la plupart des cas et l'utilisateur doit alors aider ACL2 en ajoutant des conseils aux "defthm", ou par la preuve de règles supplémentaires.

Voici, par exemple, une fonction qui "copie" un arbre : [3]

```
(defun treecopy (x)
  (if (consp x)
      (cons (treecopy (car x))
            (treecopy (cdr x)))
    x))
```

Cette définition est un évènement qui étend la théorie "ground zero" avec l'axiome constitué de l'égalité entre le terme `(treecopy x)` et le corps de sa définition. Cette extension de la logique est permise par le principe de définition d'ACL2. Ce principe impose que pour ajouter une fonction récursive, on doit en prouver la terminaison. Nous ne détaillons pas ici ce principe. Pour notre exemple, à chaque appel récursif x est une paire et la fonction `treecopy` est appelée sur le premier élément ou le reste de cette paire, mais dans l'un et l'autre cas l'appel récursif a pour paramètre x privé de un ou plusieurs éléments. Le calcul s'arrête quand tous les éléments de x ont été consommés. La fonction `treecopy` est admissible dans la logique ACL2.

Pour prouver que cette fonction est une identité, l'utilisateur soumet l'évènement suivant :

```
(defthm treecopy-id
   (equal (treecopy x) x))
```

La réponse complète d'ACL2 est donnée en annexe B. ACL2 fait une preuve par induction selon la structure de la définition de `treecopy`. Il présente tout d'abord le schéma d'induction utilisé, avant de décomposer la preuve en un cas de base (`Subgoal *1/2`) :

```
Subgoal *1/2
(IMPLIES (NOT (CONSP X))
         (EQUAL (TREECOPY X) X))
```

ce qui signifie :

$$x \text{ n'est pas une liste} \Rightarrow \mathit{treecopy}(x) = x$$

et une étape d'induction (`Subgoal *1/1`) :

[3] L'exemple est issu de [KM05] avec la permission des auteurs.

```
Subgoal *1/1
(IMPLIES (AND (CONSP X)
              (EQUAL (TREECOPY (CAR X)) (CAR X))
              (EQUAL (TREECOPY (CDR X)) (CDR X)))
         (EQUAL (TREECOPY X) X))
```

qui signifie :

$$\begin{array}{ll} & x \text{ est une liste} \\ \wedge & treecopy(car(x)) = car(x) \\ \wedge & treecopy(cdr(x)) = cdr(x) \\ \Rightarrow & treecopy(x) = x \end{array}$$

Il est important de noter qu'ACL2 imprime une grande quantité de texte. Ceci est extrêment utile pour analyser les échecs de preuve et déduire les lemmes intermédiaires nécessaires. Notons aussi que cette preuve n'est pas une *preuve formelle*. Les auteurs d'ACL2 garantissent plutôt qu'une preuve formelle *existe* pour chaque `Q.E.D` de ACL2 [KM97c]. Nous revenons sur ce point dans la section suivante.

Nous décrivons maintenant avec un peu plus de détails la logique ACL2.

4.1.2 La logique ACL2

La logique est du premier ordre avec égalité et sans quantificateur[4]. Sa syntaxe est donnée par le sous-ensemble de Common LISP constitutif du langage ACL2. La logique comporte un ensemble d'axiomes et de règles d'inférence.

La syntaxe décrit des termes composés de variables, de constantes et de symboles de fonctions. Les fonctions sont appliquées à un nombre fixe de termes passés comme arguments. La logique ACL2 n'est pas typée. Les fonctions sont *totales*. Elles sont définies sur tout l'univers de la logique ACL2.

Les axiomes d'ACL2 décrivent les propriétes des primitives de Common LISP. La figure 4.2 montre quelques axiomes; notons que `t` et `nil` signifient respectivement *vrai* et *faux*. Mis à part `if` et `equal`, les fonctions importantes sont `cons`, `car` et `cdr`. `cons` construit une paire ordonnée, `car` retourne la partie gauche d'une telle paire, et `cdr` renvoie la paire moins son premier élément. Le prédicat `consp` reconnaît une paire. Le prédicat `endp` reconnaît l'élément vide, soit la constante `nil`.

Revenons sur la preuve précédente, notamment sur le sous-but `*1/1`. ACL2 ne donne pas une preuve formelle, mais une preuve formelle existe. En voici une par exemple[5] qui utilise les axiomes de la figure 4.2 :

```
(treecopy x)
=                          {def treecopy}
(if (consp x)
    (cons (treecopy (car x))
```

[4]La logique ACL2 admet les quantificateurs universel et existentiel du premier ordre. Leur utilisation n'est pas aisée et complètement inadaptée aux problèmes rencontrés dans cette thèse [K05]. Dans ce manuscrit, la logique ACL2 est donc considérée comme sans quantificateurs.

[5]L'exemple est issu de [KM05] avec la permission des auteurs.

1. $x = y \leftrightarrow (equal\ x\ y) = t$
2. $x \neq nil \rightarrow (if\ x\ y\ z) = y)$
3. $x = nil \rightarrow (if\ x\ y\ z) = z$
4. $(consp\ (cons\ x\ y)) = t$
5. $(car\ (cons\ x\ y)) = x$
6. $(cdr\ (cons\ x\ y)) = y$
7. $(consp\ x) = t \rightarrow (cons\ (car\ x)\ (cdr\ x)) = x$

FIG. 4.2 – Quelques axiomes d'ACL2.

```
          (treecopy (cdr x)))
    x)
=                        {hypothèse 1 et Axiome 6}
(if t
    (cons (treecopy (car x))
          (treecopy (cdr x)))
    x)
=                        {Axiomes 2 et 1}
(cons (treecopy (car x))
      (treecopy (cdr x)))
=                        {hypothèse 2}
(cons (car x)
      (treecopy (cdr x)))
=                        {hypothèse 3}
(cons (car x)
      (cdr x))
=                        {Axiome 11 et hypothèse 1}
x
```

L'existence de preuves formelles et, de manière plus générale, la cohérence de la logique ACL2 sont démontrées par les auteurs [KM97c].

Les règles d'inférence du noyau de la logique sont celles du calcul propositionel avec égalité. À ce noyau, sont ajoutés : un principe d'induction et des principes d'extensions. Le principe d'induction et les principes d'extensions ne sont pas l'objet de ce manuscrit. Leur définition précise est disponible dans la littérature [KMM00a, KM05] ainsi que la description de la logique ACL2 [KM97b].

Parmi les principes d'extension, nous faisons un usage intensif de celui d'encapsulation. Nous en détaillons le fonctionnement dans la sous-section suivante.

4.1.3 Le second ordre dans ACL2 : principe d'encapsulation

Dans le chapitre précédent, les fonctions *Routing*, *Scheduling*, *recv* et *send* sont des fonctions non-définies mais contraintes à satisfaire une série de propriétés. La correction de

GeNoC est déduite de ces propriétés. Ce théorème est valable pour *toute* fonction satisfaisant les propriétés. Cette quantification du second ordre n'est pas permise dans la logique ACL2 qui est du premier ordre. Néanmoins, ACL2 fournit un mécanisme permettant de simuler cette quantification.

Le *principe d'encapsulation* autorise l'introduction de fonctions non-définies. L'évènement suivant introduit une nouvelle fonction, nommée f, prenant n arguments et retournant un argument. La contrainte ϕ, nommée *contrainte*, est associée à cette fonction.

```
(encapsulate (((f x1 ... xn) ⇒ *))   ; signature de f
    (local (defun f (x1 ... xn) body))   ; fonction f témoin
    (defthm contrainte φ))   ; théorème à prouver sur la fonction témoin
```

L'encapsulation est *admissible* si l'évènement `defun` est admissible et que ϕ est un théorème pour la logique étendue par ce `defun`. Si l'encapsulation est admissible l'axiome ϕ est ajouté à la logique. L'encapsulation modifie la logique ACL2 et pour en conserver la cohérence, ACL2 traite celui-ci en deux temps. Tout d'abord, ACL2 considère tous les évènements (même locaux) et tente de les ajouter à la logique. Si cette tentative réussit, ACL2 conserve uniquement les évènements non-locaux. Sinon, il rapporte une erreur. Les évènements locaux servent de témoin et assurent l'existence, pour chaque signature, d'au moins une définition satisfaisant les théorèmes exportés. Autrement dit, il existe au moins une extension de la logique qui en préserve la cohérence.

Puisque ϕ est le seul axiome connu à propos de f, il parait naturel de déduire que tout théorème établi sur f est aussi un théorème pour toute fonction $\hat{f}$ qui satisfait ϕ. Soit ψ une formule et $\hat{\psi}$ la formule obtenue en remplaçant f par $\hat{f}$ dans ψ. La règle d'inférence d'ACL2, dite *functional instantiation* [BGKM91], permet de déduire de tout théorème θ, le théorème $\hat{\theta}$ pourvu que $\hat{\phi}$ soit un théorème. Autrement dit, on peut déduire un théorème à propos d'une fonction $\hat{f}$ à partir d'un théorème analogue établi à propos d'une fonction f, si $\hat{f}$ satisfait les mêmes contraintes que f.

Nous illustrons ce principe sur les interfaces, décrites en section 3.5 (page 63) du chapitre précédent. Le code ACL2 complet est donné en Annexe C.1.

La fonction *send* prend pour unique argument un message et retourne une unique trame. Ci-dessous, nous en rappelons la fonctionnalité :

$$send : \mathcal{D}_{msg} \rightarrow \mathcal{D}_{frm} \tag{4.1}$$

Aucune restriction n'est faite sur les domaines $\mathcal{D}_{msg}$ et $\mathcal{D}_{frm}$ des messages et des trames ; ils sont quelconques. La fonction *send*, pour la logique ACL2, est simplement une fonction prenant un argument et retournant un argument. La *signature* de la fonction est la suivante :

```
((send *) ⇒ *)
```

La fonctionnalité de la fonction *recv* est rappelée ci-dessous :

$$recv : \mathcal{D}_{frm} \rightarrow \mathcal{D}_{msg} \tag{4.2}$$

La signature de cette fonction, comme pour la fonction précédente, est :

$$((recv\ *) \Rightarrow *)$$

La principale contrainte sur ces deux fonctions est que leur composition est une identité. Ces deux fonctions doivent satisfaire le théorème ACL2 suivant :

```
(defthm p2pCorrectness
  ; ; recv ∘ send(msg) = msg
  (equal (recv (send msg)) msg))
```

Deux autres contraintes sont associées à la fonction *send*. La première spécifie que si rien ne doit être envoyé, la fonction *send* retourne la liste vide. La liste vide est représentée par `nil` dans la logique ACL2. La constante `nil` est aussi interprétée comme "faux" dans la logique. La contrainte est que la fonction *send* retourne `nil` lorqu'on lui passe `nil` comme argument. Soit le théorème suivant :

```
(defthm send-nil
  (not (send nil)))
```

La seconde contrainte spécifie que si le message n'est pas la liste vide, la fonction *send* produit un message différent de la liste vide. Dans la logique ACL2, tout ce qui n'est pas égal à la constante `nil` est interprété comme "vrai". "Être différent de la liste vide" signifie "être différent de `nil`", soit "être vrai". On obtient l'expression suivante :

```
(defthm send-not-nil
  (implies msg (send msg)))
```

Les fonctions *send* et *recv*, et leurs contrainte sont regroupées dans l'`encapsulate` suivant :

```
(encapsulate
 (((send *) ⇒ *)
  ((send *) ⇒ *))
 (local (defun send (msg) msg)) ; ; temoin local
 (local (defun recv (frm) frm)) ; ; temoin local
 (defthm p2pCorrectness
  (equal (recv (send msg)) msg))
 (defthm send-nil
  (not (send nil)))
 (defthm send-not-nil
  (implies msg (send msg))))
```

Après son exécution, les trois contraintes sont ajoutées comme axiomes à la logique ACL2. Les deux définitions locales ne servent que de témoins. Leur définition est la plus simple possible.

Le système ACL2 génère - et prouve quand cela est possible - automatiquement les obligations de preuve à remplir pour appliquer la règle de *functional instantiation*. Par exemple,

redéfinissons les deux témoins en dehors de l'encapsulate. Considérons une fonction my_{send} qui débute une communication par l'envoie d'une constante permettant de se synchroniser avec l'unité réceptrice. Cette constante est la liste `(0 1 0 1 0 1 0 1)`, nommée `*start*`. Pour satisfaire la contrainte *send*-`nil`, la fonction my_{send} retourne la constante `nil` si msg est égal à `nil`. La définition de la fonction my_{send} est la suivante :

```
(defun my_send (msg)
  (if (not msg)
     nil
   (append *start* msg))
```

La fonction my_{recv} lit une liste, `lst`, de bits. Si cette liste est la liste vide, elle retourne la constante `nil`. Si les huit premiers bits forment la constante `*start*`, elle retourne `lst` moins ces huit premiers bits. Sinon, elle consomme un bit et lit le reste de `lst`. La définition de my_{recv} est la suivante :

```
(defun my_recv (lst)
  (if (endp lst)
     nil
    (if (equal (firstn 8 lst) *start*)
       (nthcdr 8 lst) ; ; lst moins *start*
     (my_recv (cdr lst)))))
```

Le principe est ensuite de prouver une propriété quelconque (la constante vraie `t` par exemple) en donnant pour conseil (*hint*) à ACL2 d'utiliser la propriété de l'`encapsulate` précédent. Ceci constitue l'application de la règle de *functional instantiation* de ACL2. Le code ACL2 ci-dessous demande à ACL2 de prouver la constante vrai `t`. Le *hint* force ACL2 à ajouter comme hypothèse la contrainte *p2p*`Correctness` ci-dessus en remplaçant les fonctions *send* et *recv* par my_{send} et $my_{p2precv}$. Avant d'ajouter cette hypothèse, ACL2 prouve que les fonctions concrètes vérifient le théorème *p2p*`Correctness`.

```
(defthm check-instance-p2p
  t                    ; on veut prouver "vrai"
  :rule-classes nil    ; on n'ajoute pas de règle
  :hints (("GOAL"      ; on force ACL2 à utiliser p2pCorrectness
                       ; en substituant recv par my_recv
                       ; et send par my_send
           :use ( :functional-instance p2pCorrectness
                                       (recv my_recv)
                                       (send my_send)))))
```

Les principes d'`encapsulation` et de *functional instantiation* sont utilisés de manière analogue pour l'expression dans ACL2 de tous les autres modules de *GeNoC*.

4.2 Principes de $GeNoC^\flat$

La logique ACL2 n'admettant pas de quantificateurs, les formules présentées au chapitre précédent n'ont pas une traduction directe vers ACL2. Le principe est d'exprimer les quantificateurs par des fonctions récursives. Considérons la formule $\forall x \in E, p(x)$, signifiant que tout élément de l'ensemble E satisfait le prédicat p. Dans ACL2, ceci s'exprime en considérant une liste d'éléments de E et en définissant une fonction f_p vérifiant que chaque élément d'une liste vérifie p. La définition de f_p est la suivante[6] :

$$f_p(l) \triangleq \begin{cases} t & \textbf{si } l = \epsilon \\ p(e) \wedge f_p(l') & \textbf{sinon } l = e.l' \end{cases} \tag{4.3}$$

La propriété $\forall x \in E, p(x)$ devient $\forall l, l \subseteq_l E, f_p(l)$. Dans la syntaxe ACL2, ceci s'exprime par une implication :

```
(defthm foo
  (implies (Ep l) (f_p l)))
```

où `Ep` est un prédicat reconnaissant une liste dont tous les éléments appartiennent à E.

De manière générale, les formules principales de $GeNoC$ expriment une propriété liant une liste et le résultat de l'application d'une fonction sur cette liste. Ces propriétés expriment que pour tout élément e' d'une liste obtenue par application d'une fonction $\mathcal{F}$ à une liste L, il existe un unique élément e de L tel que e et e' satisfont une propriété $\psi(e, e')$. Soit une expression de la forme suivante :

$$\forall e' \in_l \mathcal{F}(L), \exists! e \in_l L, \psi(e, e') \tag{4.4}$$

Les listes L et $\mathcal{F}(L)$ sont des listes de missives, voyages, transactions, etc ... La formule ψ contient toujours l'égalité entre l'identifiant de e et celui de e'. L'unicité de l'élément e est assurée par le typage qui garantit l'unicité des identifiants des éléments de L et ceux de $\mathcal{F}(L)$. Si on filtre la liste L par rapport aux identifiants de $\mathcal{F}(L)$, on obtient une liste comparable avec $\mathcal{F}(L)$ élément par élément. Le quantificateur existentiel n'est plus utile.

L'opérateur de filtrage est défini comme suit. La liste $\mathcal{V}/l_{ids}$ est une sous-liste de la liste de voyages $\mathcal{V}$ obtenue en filtrant $\mathcal{V}$ par rapport à une liste d'idenfiants l_{ids} donnée.

Exemple 10 *Si $\mathcal{V}$ est* ((123 m_1 (1 3 9)) (212 m_2 (12 4 25)) (313 m_3 (1 12 3))), *alors* $\mathcal{V}/(123\ 313)$ *est* ((*123 m_1 (1 3 9)*) (*313 m_3 (1 12 3)*)).

Soit f_ψ, une fonction qui prend deux listes pour arguments. Si ces deux listes sont vides, la fonction f_ψ retourne la constante "vrai" (notée t). Si une seule des deux listes est vide, la fonction f_ψ retourne la constante "faux" (notée nil). Sinon, la fonction retourne la conjonction de la propriété ψ appliquée entre les premiers éléments de chaque liste et l'application de f_ψ au reste de chaque liste. La définition de f_ψ est la suivante :

$$f_\psi(l_1, l_2) \triangleq \begin{cases} t & \textbf{si } l_1 = \epsilon \wedge l_2 = \epsilon \\ nil & \textbf{si } l_1 \neq \epsilon \wedge l_2 = \epsilon \vee l_1 = \epsilon \wedge l_2 \neq \epsilon \\ \psi(e, e') \wedge f_\psi(l'_1, l'_2) & \textbf{sinon } l_1 = e.l'_1 \wedge l_2 = e'.l'_2 \end{cases}$$

[6]Pour le quantificateur existentiel, la conjonction est remplacée par une disjonction.

Soit $\mathcal{D}_L$ le domaine de définition d'une liste L. Les expressions de la forme 4.4, "pour tout e' de $\mathcal{F}(L)$, il existe un unique e de L tel que $\psi(e, e')$", se traduisent par "pour toute liste L prenant ses valeurs sur $\mathcal{D}_L$, la fonction f_ψ appliquée à la liste $\mathcal{F}(L)$ et au filtrage de L par rapport aux identifiants de $\mathcal{F}(L)$ n'est jamais infirmée". Soit, l'expression suivante :

$$\forall L \subseteq_l \mathcal{D}_L, f_\psi(\mathcal{F}(L), L/\mathcal{F}(L)_{\lfloor id}) \tag{4.5}$$

Finalement, le quantificateur universel est remplacé par une implication et on obtient la forme suivante :

$$L \subseteq_l \mathcal{D}_L \Rightarrow f_\psi(\mathcal{F}(L), L/\mathcal{F}(L)_{\lfloor id}) \tag{4.6}$$

Dans la syntaxe ACL2, la partie gauche de l'implication est traduite par la fonction caractéristique du domaine $\mathcal{D}_L$, notée $\mathcal{D}_L$`-p`. Soient `extract-sublst` la fonction ACL2 réalisant le filtrage, et `ids` la fonction collectant les identifiants, on obtient le code ACL2 suivant :

```
(defthm bar
  (implies (D_L-p L)
           (f_ψ (F L)
                (extract-sublst L (ids (F L))))))
```

Considérons par exemple l'obligation de preuve 3.9 (page 69) concernant l'ordonnancement et qui a la forme suivante :
Soit $Scheduled = \pi_1^3 \circ Scheduling(TrLst, att)$, *alors :*

$$\forall str \in_l Scheduled, \exists! tr \in_l TrLst, \left\{ \begin{array}{ll} & Id\mathcal{V}(str) = Id\mathcal{V}(tr) \\ \wedge & Frm\mathcal{V}(str) = Frm\mathcal{V}(tr) \\ \wedge & Routes\mathcal{V}(str) \sqsubseteq Routes\mathcal{V}(tr) \end{array} \right.$$

Si nous notons $\theta(str, tr)$ la conjonction à propos de str et tr, la formule s'écrit sous une forme similaire à l'expression 4.4 ci-dessus :

$$\forall str \in_l Scheduled, \exists! tr \in_l TrLst, \theta(str, tr)$$

En appliquant des transformations similaires à celles qui nous ont permises de passer de l'expression 4.4 à l'expression 4.6, on obtient :

$$TrLst \subseteq_l \mathcal{D}_\mathcal{V} \Rightarrow f_\theta(Scheduled, TrLst/Scheduled_{\lfloor id})$$

4.3 Listes et types de données

Dans cette section, nous montrons comment les opérateurs définis dans la section 3.2 (page 53) s'expriment dans ACL2. Nous montrons aussi le principe de l'expression des types de données définis dans la section 3.3.1 (page 56) du chapitre précédent.

Notation classique	Notation ACL2	Rôle
$e.l$	`(cons e l)`	ajoute e à l
$l_1 \sqcup l_2$	`(append l1 l2)`	concaténation
$Len(l)$	`(len l)`	longueur de l
$l[i]$	`(nth i l)`	ième élément de l
$NoDuplicatesp(l)$	`(no-duplicatesp l)`	absence de doublon
ϵ	`nil`	liste vide
$l_1 \subseteq_l E$	`(subsetp l1 E)` `(ValidTypep E)`	inclusion de l_1 dans E et typage
$l_1 \sqsubseteq l_2$	`(subsetp l1 l2)`	inclusion de l_1 dans l_2

TAB. 4.1 – Traductions des opérateurs sur les listes

4.3.1 Opérateurs sur les listes

Le tableau 4.1 récapitule l'expression des opérateurs sur les listes dans ACL2. Toutes les fonctions sont des fonctions standards de LISP et/ou ACL2.

L'ajout d'un élément à une liste est traduit par un appel à la fonction `cons`, la concaténation à la fonction `append`. La longueur d'une liste est calculée par la fonction `len`, un élément d'une liste est accédé par la fonction `nth`. Le prédicat `no-duplicatesp` reconnaît une liste sans doublon. La liste vide est représentée par la constante `nil`. L'inclusion est représentée par la fonction `subsetp` à laquelle il faut ajouter une condition de typage pour obtenir l'équivalent de l'opérateur $\subseteq_l$.

Les opérateurs de projection des éléments d'une liste de liste $l_{\lfloor x}$ sont traduits par une fonction pour chaque élément des listes. Par exemple, si $TrLst$ est une liste de voyages alors $TrLst_{\lfloor id}$ correspond à `(V-ids TrLst)`, ou encore la liste des trames $\mathcal{M}_{\lfloor frm}$ d'une liste $\mathcal{M}$ de missives est obtenue par la fonction `(M-frms M)`.

4.3.2 Types de données

La définition formelle des objets à analyser est une tâche importante. Sa principale difficulté réside dans la traduction de concepts informels et imprécis dans un formalisme doté d'une impitoyable précision. Considérons une transaction. Au chapitre précédent, une transaction est décrite comme un quadruplet. Formellement, ceci signifie qu'une transaction `trans` est une paire (*i.e.* reconnue par le prédicat `consp`), ainsi que son reste (`cdr`), le reste de son reste (`cddr`), et le reste du reste de son reste (`cdddr`). Mais ce qui est important est de savoir aussi que le reste du reste du reste de son reste (`cddddr`) est égal à `nil`. Ce détail, même sur un objet aussi simple qu'une transaction, est crucial pour les preuves futures. Le prédicat suivant reconnaît une transaction :

```
(defun validfield-transactionp (trans)
  ;; trans = (id A msg B)
  (and (consp trans)
       (consp (cdr trans))          ;; (A msg B)
       (consp (cddr trans))         ;; (msg B)
```

```
(consp (cdddr trans))        ;; (B)
(null (cddddr trans))))
```

Tous les objets définis dans la section 3.3.1 sont ainsi modélisés dans ACL2. Le code complet est donné en Annexe C.2.

4.4 Définition générique des nœuds

Dans cette section, nous présentons la définition générique de l'ensemble des nœuds du réseau dans la logique ACL2. Cette définition diffère de celle donnée en section 3.4 (page 62) du chapitre précédent par différents aspects. Le code complet est donné en annexe C.3.

Tout d'abord, la logique ACL2 n'est pas typée. Le domaine de définition $GenericNodeSet$ n'est pas considéré.

Dans le chapitre 3 (section 3.4 page 62) nous considérons l'ensemble $NodeSet$ des nœuds d'un réseau particulier. Dans $GeNoC^\flat$, cet ensemble est une liste de nœuds. La fonction $NodeSetGenerator$ produit une liste de nœuds et la contrainte correspondante à l'obligation de preuve 3.1 assure que chaque élément de cette liste est bien un nœud, c'est à dire est reconnu par le prédicat $ValidNodep$. Comme nous l'avons vu en section 4.2 de ce chapitre, la quantification universelle est remplacée par une fonction récursive (*c.f.* équation 4.3 page 85) $f_{ValidNodep}$ définie comme suit :

$$f_{ValidNodep}(l) \triangleq \begin{cases} t & \textbf{si } l = \epsilon \\ ValidNodep(e) \wedge f_{ValidNodep}(l') & \textbf{sinon } l = e.l' \end{cases} \tag{4.7}$$

Si on considère $ValidNodep$ comme une fonction non-définie, elle devient un paramètre dans la définition de la fonction $f_{ValidNodep}$. La construction de $f_{ValidNodep}$ est une opération du second ordre ; elle n'est pas possible dans la logique ACL2. Dans $GeNoC^\flat$, nous considérons la fonction $f_{ValidNodep}$, notée simplement `NodeSetp`, comme la fonction non-définie. L'équation 4.7 ci-dessus donne la construction de toute fonction `NodeSetp` à partir d'un prédicat $ValidNodep$ particulier à un réseau.

La fonction `NodeSetp` prend une liste et retourne un Booléen. Dans la logique non typée d'ACL2 sa signature est la suivante :

```
((NodeSetp *) => *)
```

Comme témoin local, un nœud est un naturel. Le prédicat `NodeSetp` reconnaît donc une liste de naturels. La définition de son témoin est la suivante :

```
(local (defun NodeSetp (l)
         (if (endp l) t
           (and (natp (car l)) (NodeSetp (cdr l))))))
```

Les paramètres sont représentés par une fonction dont la signature est la suivante :

```
((ValidParamsp *) => *)
```

Pour simplifier, le prédicat `ValidParamsp` a pour témoin une fonction qui retourne t quel que soit son argument.

```
(local (defun ValidParamsp (x) (declare (ignore x)) t))
```

La fonction génératrice est représentée par une fonction dont la signature est la suivante :

```
((NodesetGenerator *) => *)
```

Le témoin de la fonction génératrice `NodeSetGenerator` prend comme argument un naturel x et construit la liste de tous les naturels allant de x jusqu'à zéro.

```
(local (defun NodesetGenerator (x)
         (if (zp x) nil
           (cons x (NodesetGenerator (1- x))))))
```

La contrainte concernant les nœuds est que tout nœud produit par la fonction génératrice est un nœud valide ; soit le théorème suivant :

```
(defthm nodeset-generates-valid-nodes
  (implies (ValidParamsp params)
           (NodeSetp (NodesetGenerator params))))
```

Dans les preuves futures, il est nécessaire de prouver que toute sous-liste d'une liste de nœuds valides est une liste de nœuds valides, soit :

```
 (defthm subsets-are-valid
   (implies (and (NodeSetp x) (subsetp y x))
            (NodeSetp y)))
```

La preuve de ce lemme dépend de la définition de `NodeSetp`. Il doit être prouvé pour chaque concrétisation de `NodeSetp`. Il est ajouté à l'encapsulation.

Finalement, l'`encapsulate` pour les nœuds est le suivant :

```
(encapsulate  ;; GenericNodeSet
 (((NodesetGenerator *) => *)
  ((ValidParamsp *) => *)
  ((NodeSetp *) => *))
 ;; local witnesses
 (local (defun ValidParamsp (x) (declare (ignore x)) t))
 (local (defun NodesetGenerator (x)
          (if (zp x) nil
            (cons x (NodesetGenerator (1- x))))))
 (local (defun NodeSetp (l)
          (if (endp l) t
            (and (natp (car l))
                 (NodeSetp (cdr l))))))
 (defthm nodeset-generates-valid-nodes
```

```
   (implies (ValidParamsp params)
            (NodeSetp (NodesetGenerator params))))
 ;; we add a generic lemma
 (defthm subsets-are-valid
   ;; this lemma is used to prove that routes are made of valid nodes
   (implies (and (NodeSetp x)
                 (subsetp y x))
            (NodeSetp y)))
) ;; end GenericNodeSet
```

4.5 Définition générique des fonctions du routage

Dans cette section, nous présentons la définition générique du routage. La définition de la fonction *Routing* dépend de la définition de la fonction ρ. Cette dépendance, tout comme pour les fonction `NodeSetp` et $ValidNodep$ des nœuds, introduit des constructions à l'ordre supérieur qui sont impossibles dans la logique ACL2. Pour $GeNoC^b$, on ne considère que la fonction *Routing*. Par conséquent, une partie des théorèmes associés à $GeNoC$ devient ici des obligations de preuve.

Avant de présenter la fonction *Routing* et ses contraintes, nous définissons les prédicats exprimant la validité des routes. Le code complet concernant le routage est donné en annexe C.4.

4.5.1 Validité des routes

Comme cela a été défini au chapitre précédent, la correction des routes est une notion générale à toute architecture de communication. La traduction du prédicat `ValidRoutep` est directe. Une route r est correcte par rapport à une missive m donnée si le premier élément de r - soit `(car r)` - est égal à l'origine de m, le dernier élément - soit `(car (last r))`[7] - est égal à la destination de m et enfin si chaque élément de r appartient à la liste des nœuds existants dans l'architecture. La longueur d'une route est supérieure ou égale à deux. Parmi ces propriétés, une seule est dépendante de la liste `NodeSet`. Pour éviter les variables libres dans certains théorèmes, il est nécessaire de séparer explicitement cette propriété. Les propriétés restantes sont définies par le prédicat suivant :

```
(defun ValidRoutep (r m)
  (and (equal (car r) (OrgM m)) (equal (car (last r)) (DestM m))
       (<= 2 (len r))))
```

La fonction `CheckRoutes` vérifie que chaque route d'une liste de routes satisfait les prédicats `ValidRoutep` et que chaque route est une sous-liste de `NodeSet`. Elle exprime l'obligation de preuve 3.4 page 66.

```
(defun CheckRoutes (routes m NodeSet)
```

[7] Dans ACL2, la fonction `(Last l)` retourne une liste contenant le dernier élément de l.

```
(if (endp routes)
    t
  (let ((r (car routes)))
    (and (ValidRoutep r m)
         (subsetp r NodeSet)
         (CheckRoutes (cdr routes) m NodeSet)))))
```

Le prédicat `CorrectRoutesp` vérifie que tout voyage est correct par rapport à une missive donnée, soit que chaque route associée au voyage satisfait le prédicat `CheckRoutes` ci-dessus et que l'identifiant et la trame du voyage sont identiques à ceux de la missive. On s'assure aussi que la liste de missives et celle des voyages contiennent le même nombre d'éléments. Ce prédicat exprime l'obligation de preuve 3.4 et le théorème 3.1 page 66 garantissant la correspondance entre les voyages et les missives.

```
(defun CorrectRoutesp (TrLst M NodeSet)
  (if (endp TrLst)
      (if (endp M)
          t                  ;; len(M) = len(TrLst)
        nil)
    (let* ((tr (car TrLst))
           (msv (car M))
           (routes (RoutesV tr)))
      (and (CheckRoutes routes msv NodeSet)
           (equal (IdV tr) (IdM msv))
           (equal (FrmV tr) (FrmM msg))
           (CorrectRoutesp (cdr TrLst) (cdr M) NodeSet)))))
```

Si ce prédicat est satisfait, il implique que la conversion de la liste de voyages $TrLst$ en une liste de missives produit la liste $\mathcal{M}$. Soit le théorème suivant :

```
(defthm correctroutesp-=>-tomissives
    (implies (and (CorrectRoutesp TrLst M NodeSet)
                  (Missivesp M NodeSet) ; liste valide de missives[8]
                  (TrLstp TrLst)) ; liste valide de voyages[9]
             (equal (ToMissives TrLst) M)))
```

4.5.2 Fonction générique de routage dans ACL2

La fonction générique de routage prend deux arguments : une liste de missives et la liste des nœuds existants dans le réseau. Elle retourne une liste de voyages. Sa signature est la suivante :

```
(((Routing * *) => *))
```

[8]Le prédicat `Missivesp` correspond au prédicat $\mathcal{M}_{lstp}$ du chapitre précédent.

[9]Le prédicat `TrLstp` correspond au prédicat $\mathcal{V}_{lstp}$ du chapitre précédent.

La fonction témoin pour l'`encapsulate` correspond à un routage dans un bus. La liste des routes contient simplement la liste constituée de l'origine et de la destination.

```
;; local witnesses
(local (defun route (Missives)
         (if (endp Missives)
             nil
           (let* ((msv (car Missives))
                  (Id (IdM msv))
                  (frm (FrmM msv))
                  (origin (OrgM msv))
                  (destination (DestM msv)))
             (cons (list Id frm (list (list origin destination)))
                   (route (cdr Missives)))))))
(local (defun routing (Missives NodeSet)
         (declare (ignore NodeSet))
         (route Missives)))
```

Les contraintes associées à la fonction ACL2 correspondent aux théorèmes de la section 3.6, page 64, du chapitre précédent.

La contrainte principale est que la fonction `Routing` satisfait le prédicat `CorrectRoutesp`.

```
(defthm Routing-CorrectRoutesp
  (let ((NodeSet (NodeSetGenerator Params)))
    (implies (and (Missivesp M NodeSet)
                  (ValidParamsp Params))
             (CorrectRoutesp (Routing M NodeSet) M NodeSet))))
```

Cette contrainte correspond à toutes les obligations de preuve définies en section 3.6 du chapitre précédent, sauf une. La seconde contrainte correspond à cette dernière obligation de preuve (obligation 3.5, page 66). Elle vérifie que pour toute liste valide de missives la fonction `Routing` retourne une liste valide de voyages.

```
(defthm TrLstp-routing
  (let ((NodeSet (NodeSetGenerator Params)))
    (implies (and (Missivesp M NodeSet)
                  (ValidParamsp Params))
             (TrLstp (routing M NodeSet)))))
```

Nous avons donné ici les principales contraintes. Des lemmes locaux sur la fonction témoin sont nécessaires. De plus, deux contraintes supplémentaires sont ajoutées. La première vérifie que la fonction `Routing` retourne une liste propre. La seconde vérifie que `Routing` retourne `nil` si la liste de missives est vide, c'est à dire égale à `nil`.

4.6 Définition générique de l'ordonnancement

Définition. La fonction générique définie dans ACL2 est une traduction directe de la définition donnée au chapitre précédent. La fonctionnalité de la fonction *Scheduling* (*c.f.* section 3.7 du chapitre précédent) est :

$$Scheduling : \mathcal{D}_{\mathcal{V}} \times AttLst \rightarrow \mathcal{D}_{\mathcal{V}} \times \mathcal{D}_{\mathcal{V}} \times AttLst \tag{4.8}$$

Dans ACL2, les fonctions qui retournent une liste de valeurs sont définies à l'aide de la fonction `mv` (*multi-valued*). Il est de même pour les signatures. La fonction *Scheduling* prend trois arguments et retourne une liste de trois valeurs. Sa signature est la suivante :

```
((scheduling * *) => (mv * * *))
```

La fonction témoin est très simple et elle doit juste vérifier les contraintes. La fonction choisie n'a pas un comportement réaliste mais joue pleinement son rôle de témoin. Sa définition est la suivante. Si la somme de toutes les tentatives est vide, tous les voyages sont retardés. Sinon, tous les voyages sont ordonnancés et une tentative est consommée par chaque nœud qui en possède au moins une (fonction `consume-attempts`).

```
(local (defun scheduling (TrLst att)
          ;; local witness
          (mv
           ;; scheduled frames
           (if (zp (SumOfAttempts att))
               nil  ;; no attempt left -> nothing is scheduled
             TrLst)  ;; otherwise everything is scheduled
           ;; delayed frames
           (if (zp (SumOfAttempts att))
               TrLst ;; no attempt left -> everything is delayed
             nil) ;; otherwise nothing is delayed
           (if (zp (SumOfAttempts att))
               att ;; no attempt left -> no modification on att
             (consume-attempts att))))) ;; consume attempts
```

Contraintes. L'obligation de preuve 3.6 (page 68) exprime que les deux premièrs arguments retournés par la fonction *Scheduling* sont des listes valides de voyages si la liste $TrLst$ est une liste valide de voyage. Le i-ème argument d'une fonction définie par `mv` est obtenu par l'expression `(mv-nth i F)`. Cette obligation se traduit de la manière suivante :

```
(defthm trlstp-scheduled-delayed
  (implies (trlstp TrLst)
           (and (trlstp (mv-nth 0 (scheduling TrLst att)))
                (trlstp (mv-nth 1 (scheduling TrLst att))))))
```

L'obligation de preuve 3.7 (page 68) assure que la fonction *Scheduling* consomme au moins une tentative à chaque "tour" d'ordonnancement. Si la somme des tentatives n'est pas nulle, cette somme diminue à chaque appel de la fonction *Scheduling*. La fonction `mv-let` permet de récupérer une liste d'arguments et de les utiliser ensuite dans une expression. Si des arguments ne sont pas mentionnés dans cette expression, il faut déclarer explicitement qu'on les ignore au moyen de la commande *ignore*. L'obligation de preuve 3.7 se traduit de la manière suivante :

```
(defthm consume-at-least-one-attempt
  (mv-let (Scheduled Delayed newatt) ;; trois arguments de Scheduling
          (scheduling TrLst att)
          (declare (ignore Scheduled Delayed)) ;; deux  sont ignores
          (implies (not (zp (SumOfAttempts att)))
                   (< (SumOfAttempts newatt) (SumOfAttempts att)))))
```

L'obligation de preuve 3.8 (page 69) vérifie que la liste *Delayed* des voyages retardés est une sous-liste de la liste $TrLst$ initiale. Elle est exprimée au moyen de quantificateurs. La preuve ACL2 du théorème final sur $GeNoC^\flat$ étant différente de celle sur *GeNoC* (voir section 4.7 ci-après), nous formulons cette obligation de preuve d'une manière différente. Nous exprimons que la liste *Delayed* est égale au filtrage de $TrLst$ par rapport aux identifiants de *Delayed*. Nous rappelons que l'opérateur de filtrage est réalisée par la fonction `extract-sublst` et que la liste des identifiants d'une liste de voyages est obtenue par la fonction `v-ids`. On obtient l'expression ACL2 suivante :

```
(defthm delayed-travel-correctness
  (mv-let (Scheduled Delayed newatt) ;; trois arguments de Scheduling
          (scheduling TrLst att)
          (declare (ignore newatt scheduled)) ;; 2 ignores
          (implies (and (ValidParamsp Params) (TrLstp TrLst))
                   (equal Delayed
                          (extract-sublst TrLst  ;; filtrage de TrLst
                                          (v-ids Delayed)))))
  :rule-classes nil)
```

Cette règle est à utiliser avec précaution car elle réécrit la liste *Delayed* par une expression dans laquelle *Delayed* apparaît. Ceci crée une boucle dans le système de réécriture de ACL2. Cette règle ne doit pas être appliquée automatiquement, il faut la désactiver. La commande " `:rule-classes nil`" demande à ACL2 de ne pas générer de règle à partir de ce théorème. L'utilisation de cette règle sera faite manuellement au moyen d'un conseil.

L'obligation de preuve 3.9 (page 69) vérifie que la liste *Scheduled* des voyages ordonnancés est une sous-liste de la liste de voyages initiale $TrLst$. La liste des routes de chaque voyage ordonnancé est une sous-liste de celles du voyage initial correspondant. Dans la plupart des cas, la liste des routes de chaque voyage ordonnancé ne contient qu'une seule route. Cette obligation de preuve est exprimée au moyen de quantificateurs. Elle ne peut se traduire directement dans ACL2. Le quantificateur universel est remplacé par une fonction récursive comme nous l'avons vu en section 4.2 de ce chapitre.

Le prédicat `s-travel-correctness` prend pour arguments deux listes de voyages et vérifie que les éléments ayant la même position dans chaque liste possèdent le même identifiant, la même trame et que les routes du premier sont aussi des routes du second (il correspond à la fonction f_θ de la section 4.2). Sa définition est la suivante :

```
(defun s-travel-correctness (sTrLst TrLst/sids)
  (if (endp sTrLst) ; si la liste sTrLst est vide
      (if (endp TrLst/sids) ; si la liste TrLst/sids l'est aussi
          t ; alors les deux listes on la même longueur
        nil) ; sinon il y a une erreur, on retourne "faux"
    (let* ((str (car sTrLst))
           (tr (car TrLst/sids)))
      (and (equal (FrmV str) (FrmV tr)) ; Frm_V(str) = Frm_V(tr)
           (equal (IdV str) (IdV tr)) ; Id_V(str) = Id_V(tr)
           (subsetp (RoutesV str) (RoutesV tr)) ; Routes_V(str) ⊑ Routes_V(tr)
           (s-travel-correctness (cdr sTrLst) (cdr TrLst/sids))))))
```

L'obligation de preuve ACL2 exprimant la correction des voyages ordonnancés est la suivante. Le prédicat ci-dessus est vérifié si son premier argument est la liste *Scheduled* des voyages ordonnancés, l'autre est le filtrage de $TrLst$ par rapport aux identifiants de *Scheduled*.

```
(defthm scheduled-travels-correctness
  (mv-let (Scheduled Delayed newatt) ; trois résultats de scheduling
          (scheduling TrLst att)
          (declare (ignore Delayed newatt)) ; on en ignore 2
          (implies (and (TrLstp TrLst) (ValidParamsp Params))
                   (s-travel-correctness ; relation entre
                    ; les trames, les identifiants et les routes
                    scheduled ; des voyages ordonnancés
                    (extract-sublst TrLst ; et ceux de la liste initiale
                                          ; filtrée par les identifiants
                                          ; de Scheduled
                                    (V-ids Scheduled))))))
```

L'obligation de preuve 3.10 (page 69) vérifie qu'un voyage ne peut être ordonnancé et retardé, soit l'exclusion mutuelle entre les listes *Scheduled* et *Delayed*. Le prédicat `not-in` prend deux listes pour arguments et est vérifié si aucun élément de la première liste appartient à la seconde.

```
(defthm not-in-delayed-scheduled
  (mv-let (scheduled delayed newatt)
          (scheduling TrLst att)
          (declare (ignore newatt))
          (implies (TrLstp TrLst)
                   (not-in (v-ids delayed) (v-ids scheduled)))))
```

Nous avons exposé ici les principales contraintes sur la fonction *Scheduling*. D'autres contraintes sont nécessaires. Par exemple, pour pouvoir appliquer la fonction `mv-nth` à la fonction *Scheduling* il est nécessaire de savoir qu'elle retourne une liste de valeurs. Cette propriété n'est pas ajoutée par ACL2 à partir de la signature. Ou encore, il est important de savoir que les listes *Scheduled* et *Delayed* sont des listes propres. Finalement, nous montrons, en dehors de l'encapsulation, que la fonction *Scheduling* préserve le prédicat `CorrectRoutesp`.

Le code complet concernant la fonction *Scheduling* est donné en annexe C.5.

4.7 Définition et validation de $GeNoC^\flat$

La définition de $GeNoC^\flat$ est identique à celle de *GeNoC*. La différence se situe dans le théorème final dont la formulation ne mentionne pas de quantificateurs. Nous rappelons le théorème sur la fonction *GeNoC* donné au chapitre précédent. Il vérifie que pour tout résultat rst, il existe une unique transaction t telle que rst et t aient le même identifiant, le même message et la même destination. Soit, la formule suivante :

Soit $\mathcal{R} = \pi_1^2 \circ GeNoC(\mathcal{T}, NodeSet, att)$ *dans*

$$\forall rst \in_l \mathcal{R}, \exists! t \in_l \mathcal{T}, \left\{ \begin{array}{ll} & Id_{\mathcal{R}}(rst) = Id_{\mathcal{T}}(t) \\ \wedge & Msg_{\mathcal{R}}(rst) = Msg_{\mathcal{T}}(t) \\ \wedge & Dest_{\mathcal{R}}(rst) = Dest_{\mathcal{T}}(t) \end{array} \right.$$

Dans la "version ACL2" de ce théorème, nous effectuons les transformations exposées plus haut dans ce chapitre. Nous filtrons la liste des transactions initiales par rapport aux identifiants de la liste des résultats. Nous comparons ces deux listes élément par élément. Cette comparaison est définie par le prédicat `genoc-correctness`. Ce dernier prend pour arguments une liste de résultats et une liste de transactions. Il vérifie que les messages sont identiques dans les deux listes. Il vérifie aussi que les destinations correspondent. La liste des messages d'une liste de transactions est obtenue par la fonction `T-msgs`, celle des destinations par la fonction `T-dests`. Les messages d'une liste de résultats sont obtenus par la fonction `R-msgs`, celle des destinations par la fonction `R-dests`. La définition de `genoc-correctness` est la suivante :

```
(defun GeNoC♭-correctness (Results Trs/ids)
  (and (equal (R-msgs Results)
              (T-msgs Trs/ids))
       (equal (R-dests Results)
              (T-dests Trs/ids))))
```

Le théorème final établi sur la fonction $GeNoC^\flat$ prouve que le prédicat ci-dessus est vérifié si la liste `Results` est produite par la fonction $GeNoC^\flat$ et la liste de transactions `Trs` est le principal argument de cette fonction. Soit, le théorème ACL2 suivant :

```
(defthm GeNoC♭-is-correct
 (let ((NodeSet (NodeSetGenerator Params)))
```

```
(mv-let (results aborted)
        (GeNoC♭ Trs NodeSet att)
        (declare (ignore aborted))
        (implies (and (Transactionsp Trs NodeSet)
                      (ValidParamsp Params))
                 (GeNoC♭-correctness
                  results
                  (extract-sublst Trs (R-ids results)))))))
```

La preuve de ce théorème est similaire à celle présentée au chapitre trois. La fonction $GeNoC^\flat$ est définie par la fonction $GeNoC^\flat_t$ qui produit, à partir d'une liste de missives, une liste de voyages "terminés" et une liste de missives "avortées". Tout comme pour la preuve du chapitre trois, nous définissons une fonction $GeNoC^\flat_{nt}$ qui calcule la liste des voyages "terminés" sans accumulateur. Dans le théorème exprimant sa correction, les quantificateurs sont supprimés et remplacés par une fonction récursive, nommée $GeNoC^\flat_{nt}$-`correctness`. Cette fonction prend pour arguments une liste $TrLst$ de voyages et une liste $\mathcal{M}$ de missives. Pour chaque élément tr de position i dans la liste $TrLst$, et pour chaque élément m de même position i dans $\mathcal{M}$, elle vérifie que la trame de tr est identique à celle de m, et le dernier nœud de chaque route de tr est égal à la destination de m. Sa définition est la suivante :

```
(defun GeNoC♭nt-correctness (TrLst M/TrLst)
  (if (endp TrLst)
    (if (endp M/TrLst)
        t
      nil)
   (let* ((tr (car TrLst)) ; ; voyage courant
          (v-frm (FrmV tr)) ; ; trame du voyage
          (routes (RoutesV tr)) ; ; routes du voyage
          (m (car M/TrLst)) ; ; missive courante
          (m-frm (FrmM m)) ; ; trame de la missive
          (m-dest (DestM m))) ; ; destination de la missive
         (and (equal v-frm m-frm) ; ; les trames sont identiques
              (GoodRoutes routes m-dest)
              (GeNoC♭nt-correctness (cdr TrLst) (cdr M/TrLst))))))
```

où le prédicat `GoodRoutes` est défini comme suit :

```
(defun GoodRoutes (routes m-dest)
  (if (endp routes)
    t
  (let ((r (car routes)))
     (and (equal (car (last r)) m-dest)
          (GoodRoutes (cdr routes) m-dest)))))
```

Le prédicat ci-dessus doit être satisfait si `TrLst` est la liste, notée `Cplt`, produite par la fonction $GeNoC^\flat_{nt}$ et la liste `M/TrLst` le filtrage de la liste de missives initiale par rapport aux identifiants de la liste produite par $GeNoC^\flat_{nt}$. Soit, le théorème ACL2 suivant :

```
(defthm GeNoC♭nt-is-correct
  (let ((NodeSet (NodeSetGenerator Params)))
   (implies (and (Missivesp M NodeSet)
                 (ValidParamsp Params))
            (let ((Cplt (GeNoC♭nt M NodeSet att)))
              (GeNoC♭nt-correctness Cplt
                                    (extract-sublst M (v-ids Cplt)))))))
```

La preuve de ce théorème est similaire à celle exposée au chapitre trois. Elle procède par induction sur la structure de la définition de la fonction $GeNoC^{\flat}_{nt}$. Le cas de base est trivial. L'étape d'induction se divise en deux sous-buts. Le premier considère les voyages ordonnancés et est prouvé uniquement à partir des contraintes. L'autre considère les voyages retardés et sa preuve consiste dans la réecriture de l'hypothèse d'induction vers la conclusion en utilisant les contraintes. Le code complet relatif à la définition et à la validation de la fonction $GeNoC$ est donné en annexe C.6.

4.8 Conclusion

Dans ce chapitre, nous avons exprimé le formalisme du chapitre précédent dans la logique du démonstrateur de théorèmes ACL2. La réalisation de $GeNoC$ dans ACL2 ne résulte pas d'une traduction directe. Pour respecter les contraintes de la logique ACL2, de légères modifications ont dû être apportées. Dans une logique d'ordre supérieur, une traduction directe aurait été possible.

Pour exprimer $GeNoC$ dans ACL2, nous avons utilisé le principe d'encapsulation qui permet de "simuler" une quantification sur des fonctions. Une encapsulation contient des signatures, des contraintes et des fonctions témoins. Les fonctions témoins assurent qu'il existe au moins une extension de la logique ACL2 qui en préserve la sûreté et la cohérence. Après l'encapsulation, seules les contraintes sont exportées sous forme d'axiomes. En utilisant la règle d'inférence *functional instantiation*, tout théorème à propos de fonctions "encapsulées" est préservé par la substitution d'une fonction par une autre qui satisfait les contraintes de l'encapsulation. Les fonctions représentant les interfaces, le routage et l'ordonnancement sont des fonctions "encapsulées". Pour montrer que toute définition explicite de ces fonctions préserve la correction de $GeNoC^{\flat}$, il faut prouver que cette définition satisfait les contraintes de l'encapsulation correspondante. Nous avons montré comment ACL2 fournit automatiquement ces contraintes. Notre objectif a été d'obtenir le moins de contraintes possible, soit de déduire le maximum à partir des contraintes pour minimiser la preuve de conformité entre une fonction concrète et une fonction de $GeNoC^{\flat}$.

Le tableau 4.2 récapitule le nombre de fonctions, de théorèmes et le temps de preuve pour la définition et la validation de $GeNoC^{\flat}$ et de ses différents modules. L'exécution est effectuée sur une machine équipée d'un Pentium IV 1.6 GHz et Linux Mandrake 9.2. La version d'ACL2 est v2.9. Nous importons la librairie "arithmetic-3" développée par R. Krug et les librairies "list-defuns" et "list-defthms" développées par B. Bevier. Ces librairies sont distribuées avec ACL2.

	Nbre de fonctions	Nbre de théorèmes	Temps de preuve (en secondes)	Taille
types	45	3	~ 0	320 lignes
définitions diverses	7	24	~ 3	251 lignes
interfaces	2	3	~ 0	42 lignes
nœuds	3	2	< 0.1	39 lignes
routage	2	11	~ 1	116 lignes
ordonnancement	3	26	~ 3	310 lignes
$GeNoC^\flat$	10	50	~ 25	786 lignes
Total	71	119	< 30	1864 lignes

TAB. 4.2 – Fonctions, théorèmes et temps de preuve pour l'ensemble du modèle

Le modèle présenté au chapitre précédent manipule de nombreux objets. Leur définition (ligne "types" dans le tableau), ainsi que celle des prédicats et fonctions (ligne "définitions diverses") occupent une part importante du modèle ACL2. Les différents modules (interfaces, nœuds, routage et ordonnancement) ont une taille modeste. Dans l'ordonnancement, des théorèmes sont prouvés en dehors de l'encapsulation. Cette dernière est consituée d'environ 200 lignes. Les encapsulations contiennent les définitions et les obligations de preuve que toute concrétisation doit satisfaire. Leur taille raisonnable les rend facilement accessibles. La preuve de $GeNoC^\flat$, c'est à dire que les contraintes impliquent la correction du système global, est beaucoup plus importante. C'est ici que se situe un avantage important de notre approche. Cette preuve, ainsi que les types de données et les fonctions diverses ne nécessitent aucune modification pour toute concrétisation des différentes encapsulations. L'ensemble des encapsulations constituent un peu plus du quart du modèle complet.

Dans le chapitre suivant, nous montrons plusieurs concrétisations de la fonction $GeNoC^\flat$.

Chapitre 5

Méthodologie et études de cas

Dans les chapitres précédents, nous avons exposé une représentation fonctionnelle d'un modèle générique des communications sur la puce. Ce formalisme fournit une méthodologie pour la validation des spécifications du routage, de l'ordonnancement et des interfaces. Dans ce chapitre, nous présentons des *concrétisations* des fonctions de *GeNoC* selon une méthodologie définie à partir des fonctions et obligations de preuve définies au chapitre 3.

La définition générique du routage de *GeNoC* est illustrée autour d'un exemple industriel (le réseau *Octagon* de *STMicroelectronics*) et de deux autres plus académiques, l'algorithme de routage en XY et un algorithme en double Y qui est adaptatif. La définition générique de l'ordonnancement est illustrée sur un exemple industriel (l'arbitrage du bus AMBA AHB) et sur les techniques de commutation par circuits et par paquets. Concernant les interfaces, nous illustrons la définition générique par un protocole à marquage de phase situé à la couche 1 du système OSI.

Les illustrations proposées concernent tous les aspects de *GeNoC* : routage, ordonnancement et interfaces. Dans chaque illustration, nous précisons les données relatives à celle-ci. Un tableau récapitule des données chiffrées sur les différentes applications. Il présente le nombre de fonctions, de théorèmes, le temps de preuve et la taille des différents fichiers. La machine utilisée est un Pentium IV à 1.6 GHz, avec 256MB de mémoire et fonctionnant sous Linux.

5.1 Réalisations de la fonction *Routing*

La topologie d'un réseau détermine la numérotation des nœuds et les déplacements élémentaires autorisés entre nœuds directement connectés. La fonction de routage est définie par l'application successive de ces déplacements. Avant de définir une fonction particulière de routage, il faut définir l'ensemble des nœuds.

Définition des nœuds. Avant tout, il faut définir le domaine de définition des nœuds, soit une particularisation du prédicat *ValidNodep*, notée $ValidNodep_\sharp$. Le domaine de définition générique *GenNodeSet* devient un domaine particulier $GenNodeSet_\sharp$, comme les naturels

par exemple. On définit une concrétisation de la définition 3.11 (page 62), soit :

$$\forall x,\ ValidNodep_\sharp(x) \Leftrightarrow x \in GenNodeSet_\sharp \tag{5.1}$$

La topologie du réseau fixe les paramètres à partir desquels l'ensemble des nœuds est généré. Les paramètres particuliers à un réseau, $pms_\sharp$, constituent l'argument de la fonction génératrice, $NodeSetGen_\sharp$ de l'ensemble des nœuds particuliers du réseau, noté $NodeSet_\sharp$. Les paramètres particuliers $pms_\sharp$ sont reconnus par le prédicat $ValidParamsp_\sharp$.

Finalement, pour valider la concrétisation de l'ensemble des nœuds, il faut prouver que les fonctions concrètes satisfont la concrétisation de la contrainte 3.1 (page 62), soit :

$$\forall pms_\sharp,\ ValidParamsp_\sharp(pms_\sharp) \Rightarrow \forall x \in NodeSetGen_\sharp(pms_\sharp),\ ValidNodep_\sharp(x) \tag{5.2}$$

Définition du routage Tout d'abord, il faut identifier les déplacements autorisés entre nœuds directements connectés. Puisque nous considérons des réseaux réguliers (ou une régularisation d'un réseau irrégulier), ces déplacements sont tous identiques en tout point du réseau. L'identification de ces déplacements unitaires définit une concrétisation, $\mathcal{L}_\sharp$, de la logique de routage $\mathcal{L}$. La fonction de routage résulte de l'application successive de ces déplacements unitaires, soit[1] :

$$\rho_\sharp(s,d) \triangleq \begin{cases} d & \textbf{si } s = d \\ s.\rho_\sharp(\mathcal{L}_\sharp(s,d),d) & \textbf{sinon} \end{cases} \tag{5.3}$$

À partir de la topologie, on déduit la distance entre la localisation d'un message et sa destination. La distance entre un nœud s un nœud d est notée $dist(s,d)$. Cette distance est la mesure la plus couramment employée pour prouver que la fonction de routage termine. Il suffit de montrer que chaque déplacement unitaire réduit la distance entre un message et sa destination. La distance est une fonction qui retourne un naturel pour toute paire de nœuds, soit la fonctionnalité suivante :

$$dist : GenNodeSet_\sharp \times GenNodeSet_\sharp \rightarrow \mathbb{N} \tag{5.4}$$

Pour prouver la terminaison de la fonction de routage, $\rho_\sharp$, il faut prouver que celle-ci satisfait la concrétisation de l'obligation de preuve 3.3 (page 65), soit :

$$\forall s,d \in GenNodeSet_\sharp,\ s \neq d \Rightarrow dist(\mathcal{L}_\sharp(s,d),d) < dist(s,d) \tag{5.5}$$

La validité d'une route est définie par le prédicat $ValidRoutep$ (page 65). Cette définition est valable quel que soit le réseau. Il n'est pas nécessaire de redéfinir ce prédicat. Nous en rappelons la définition :

$$ValidRoutep(r,m,NodeSet) \triangleq \begin{cases} & r[0] = Org_{\mathcal{M}}(m) \\ \wedge & Last(r) = Dest_{\mathcal{M}}(m) \\ \wedge & r \subseteq_l NodeSet \wedge Len(r) \geq 2 \end{cases} \tag{5.6}$$

[1]Nous rappelons que l'ajout d'un élément e à une liste l est noté $e.l$.

Finalement, pour valider la fonction concrète de routage, il suffit de prouver qu'elle satisfait le prédicat *ValidRoutep* pour l'ensemble, $NodeSet_\sharp$, des nœuds du réseau concret, soit :

$$\begin{array}{l}\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet_\sharp) \\ \Rightarrow \forall m \in_l \mathcal{M}, \forall r \in_l \rho_\sharp(Org_\mathcal{M}(m), Dest_\mathcal{M}(m)), ValidRoutep(r, m, NodeSet_\sharp)\end{array} \tag{5.7}$$

La fonction compatible avec la définition générique, $Routing_\sharp$, calcule la liste des routes pour chaque missive d'une liste $\mathcal{M}$, soit :

$$Routing_\sharp(\mathcal{M}, NodeSet_\sharp) \triangleq \bigwedge_{m \in \mathcal{M}} List(Id_\mathcal{M}(m), Frm_\mathcal{M}(m), \rho_\sharp(Org_\mathcal{M}(m), Dest_\mathcal{M}(m))) \tag{5.8}$$

Pour valider la conformité de cette fonction avec $GeNoC$, il reste à montrer que $Routing_\sharp$ produit une liste valide de voyages si la liste initiale, $\mathcal{M}$, est une liste valide de missives. Soit l'expression suivante :

$$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet_\sharp) \Rightarrow \mathcal{V}_{lstp}(Routing_\sharp(\mathcal{M}, NodeSet_\sharp)) \tag{5.9}$$

Dans cette section, nous montrons l'application de cette méthodologie sur trois réalisations de la fonction *Routing*. La première est issue de l'industrie. Les deux suivantes montrent un algorithme déterministe et un algorithme adaptatif.

5.1.1 L'Octagon

Le réseau *Octagon* a été présenté au chapitre 2 de ce manuscrit (section 2.3.4.3, page 41). Nous en rappelons ici brièvement les principales caractéristiques (*c.f.* figure 5.1).

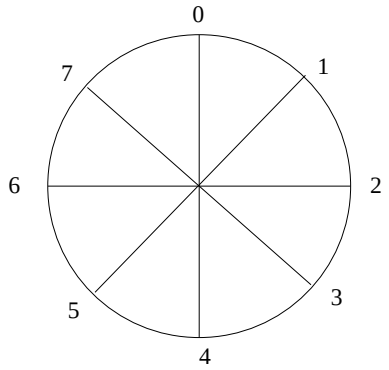


FIG. 5.1 – Unité primaire de l'Octagon.

L'unité principale de l'Octagon est constituée de huit nœuds et douze liens bidirectionnels. La route d'un paquet est calculée de la manière suivante. Chaque nœud compare l'adresse de destination (*PackAd*) avec sa propre adresse (*NodeAd*). Chaque nœud calcule une adresse relative (*RelAd*) définie comme suit :

$$RelAd = (PacketAd - NodeAd) \bmod 8 \tag{5.10}$$

À chaque nœud, la route d'un paquet est fonction de cette adresse :

- $RelAd = 0$, le paquet a atteint sa destination,
- $RelAd = 1$ ou 2, le paquet est envoyé au prochain nœud dans le sens des aiguilles d'une montre,
- $RelAd = 6$ ou 7, le paquet est envoyé au prochain nœud dans le sens inverse des aiguilles d'une montre,
- sinon le paquet est envoyé au nœud "d'en face".

5.1.1.1 Définition des nœuds de l'Octagon

Le modèle de l'*Octagon* considère un nombre arbitraire mais fini de nœuds. Ce nombre est noté *NumNode*, c'est un entier naturel et un multiple de 4 [2]. Une partie des propriétés de l'*Octagon* dépendent du quart de *NumNode*. Ce nombre est défini à partir d'un naturel N selon l'égalité $NumNode = 4N$. Le prédicat $ValidNodep_{Oct}$ prend comme arguments un nœud x et le nombre N :

$$\forall N \in \mathbb{N}, \forall x,\ ValidNodep_{Oct}(x, N) \Leftrightarrow x \in \mathbb{N} \wedge x < 4N \tag{5.11}$$

Le nombre N est aussi le paramètre dont dépend la fonction génératrice des nœuds. Le prédicat $ValidParamsp_{Oct}(N)$ vérifie que N est un entier positif non nul. L'ensemble des nœuds est celui de tous les naturels strictement inférieurs à $4N$. Cet ensemble est généré par la fonction $NodeSetGen_{Oct}(N)$. Pour valider la définition des nœuds de l'Octagon, on prouve le théorème suivant :

Théorème 5.1 Validité des nœuds de l'Octagon.

$$\forall N,\ ValidParamsp_{Oct}(N) \Rightarrow \forall x \in NodeSetGen_{Oct}(N),\ ValidNodep_{Oct}(x, N)$$

Une fois l'ensemble des nœuds définis, nous pouvons modéliser l'algorithme de routage.

5.1.1.2 Algorithme de routage de l'Octagon

Dans un premier temps, nous définissons les trois mouvements unitaires possibles dans l'Octagon. Soit s un nœud recevant un message. Le mouvement dans le sens des aiguilles d'une montre (en anglais, *clockwise*) est défini comme l'incrémentation de s modulo le nombre de nœuds dans le réseau :

$$Clockwise(s, NumNode) \triangleq (s + 1) \bmod NumNode$$

Le mouvement dans le sens inverse des aiguilles d'une montre (an anglais, *counterclockwise*) est défini comme la décrémentation de s modulo le nombre de nœuds dans le réseau :

$$CounterClockwise(s, NumNode) \triangleq (s - 1) \bmod NumNode$$

Enfin, la traversée de l'Octagon (*across*) est défini comme l'ajout à s de la moitié du nombre de nœuds dans le réseau, le tout, modulo le nombre de nœuds dans le réseau :

$$Across(s, NumNode) \triangleq (s + \frac{NumNode}{2}) \bmod NumNode$$

[2] Le raisonnement dans ACL2 est simplifié si l'on ne considère que des entiers au lieu des rationnels.

Ces mouvements sont regroupés dans la fonction $\mathcal{L}_{Oct}$, définie de la manière suivante. Soient s le nœud courant, d le nœud destination. L'adresse relative est $RelAd = (d - s) \bmod 4N$. Si le nœud courant est la destination, le message est consommé. Si l'adresse relative est strictement positive et inférieure ou égale au quart du nombre de nœuds dans le réseau, c'est à dire inférieure ou égale à N, le mouvement a lieu dans le sens des aiguilles d'une montre. Si l'adresse relative est comprise entre $3N$ et $4N$, le mouvement a lieu dans le sens inverse des aiguilles d'une montre. Sinon, le mouvement est une traversée du réseau. La définition de $\mathcal{L}_{Oct}$ est la suivante :

Définition 5.1 Déplacements unitaires dans l'Octagon.

$$\mathcal{L}_{Oct}(s, d, N) \triangleq \begin{cases} s & \textbf{si } RelAd = 0 \\ Clockwise(s, 4N) & \textbf{si } 0 < RelAd \leq N \\ CounterClockwise(s, 4N) & \textbf{si } 3N \leq RelAd < 4N \\ Across(s, 4N) & \textbf{sinon} \end{cases}$$

La fonction de routage ρ_{Oct} est définie comme l'application successive des déplacements unitaires :

Définition 5.2 Fonction de routage de l'Octagon, ρ_{Oct}.

$$\rho_{Oct}(s, d, N) \triangleq \begin{cases} d & \textbf{si } s = d \\ s.\rho_{Oct}(\mathcal{L}_{Oct}(s, d, N), d, N) & \textbf{sinon} \end{cases}$$

Pour l'Octagon, puisqu'au moins deux sens de parcours sont possibles, il existe deux distances différentes entre deux nœuds. La distance dans le sens des aiguilles d'une montre, et celle dans le sens inverse. La mesure utilisée pour prouver la terminaison de la fonction de routage de l'Octagon est le minimum entre ces deux distances, soit l'expression suivante :

$$mes_{Oct}(s, d, NumNode) = Min[(d - s) \bmod NumNode, (s - d) \bmod NumNode]$$

Pour prouver que la fonction de routage termine, il suffit de prouver que les déplacements unitaires réduisent cette "distance", soit le théorème suivant :

Théorème 5.2 Terminaison de la fonction de routage de l'Octagon.

$$\forall s, d \in GenNodeSet_{Oct}, s \neq d \Rightarrow mes_{Oct}(\mathcal{L}_{Oct}(s, d), d, NumNode) < mes_{Oct}(s, d, NumNode)$$

Preuve: La preuve est décomposée selon les différents mouvements possibles. Chacun d'entre eux doit réduire la mesure. Pour chacun de ces mouvements, les fonctions Min et mod introduisent de nombreux cas. La preuve est découpée en plus de 1200 cas par ACL2. La preuve de ces cas nécessite 10 lemmes sur la fonction modulo en plus de ceux importés avec la plus récente bibliothèque ACL2 concernant l'arithmétique [RKM03]. Deux lemmes sont aussi nécessaires pour amener ACL2 à décomposer la preuve correctement. Ensuite, la preuve est entièrement automatique. Elle est effectuée en moins de 100 secondes sur une machine équipée d'un Pentium IV 1,6 GHz et de 256 MB de mémoire. □

Pour montrer que la fonction de routage de l'Octagon, ρ_{Oct}, constitue une concrétisation valide de la fonction de routage de $GeNoC$, il faut montrer qu'elle produit des routes vérifiant le prédicat $ValidRoutep$, soit le théorème suivant :

Théorème 5.3 Validité des routes de l'Octagon.

$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet_{Oct})$
$\Rightarrow \forall m \in_l \mathcal{M}, \forall r \in_l \rho_{Oct}(Org_{\mathcal{M}}(m), Dest_{\mathcal{M}}(m)), ValidRoutep(r, m, NodeSet_{Oct})$

Preuve: Les raisonnements arithmétiques, en particulier à propos de la fonction mod, sont difficiles dans la logique ACL2. La fonction ρ_{Oct} utilise le modulo. Pour simplifier les raisonnements sur cette fonction, nous définissons une fonction équivalente, notée ρ^*_{Oct}, qui n'utilise pas le modulo. Dans cette fonction, l'adresse relative est simplement définie par la différence entre la localisation, s, d'un message et sa destination, d, soit :

$$ReAd^* = d - s$$

Les valeurs de cette adresse sont comprises entre $4N$ et $-4N$. Cet intervalle est découpé en huit intervalles de longueur N.

À partir d'une source s donnée, les destinations possibles se situent dans quatre zones - ou quartiers - différentes (*c.f.* figure 5.2). Ces quartiers correspondent aux nœuds atteignables à partir de s :

- en au plus N déplacements dans le sens positif (*clockwise*) (Quartier 1),
- en au plus N déplacements dans le sens négatif (*counterclockwise*) (Quartier 4),
- en une traversée (*accross*) et moins de N déplacements dans le sens positif (Quartier 3),
- en une traversée et moins de N déplacements dans le sens négatif (Quartier 2).

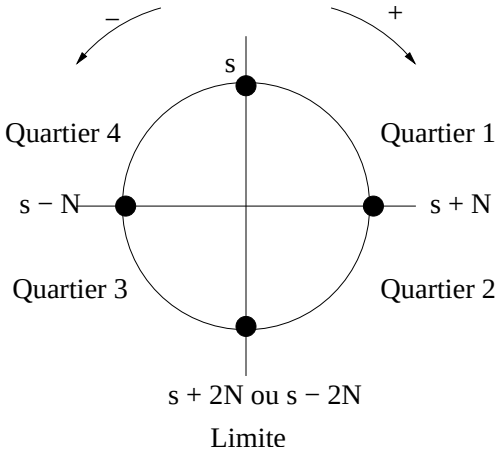


FIG. 5.2 – Décomposition de l'Octagon

Chaque zone est atteinte pour deux intervalles parmi les huit intervalles de l'adresse $RelAd^*$. Chaque intervalle définit un ensemble de destinations atteignables à partir d'une source donnée. Le quartier 1 correspond à l'ensemble des destinations d_i telles que l'adresse relative à cette destination, $d_i - s$, soit comprise entre 0 et N (ensemble noté Q_1^+) ou entre $-4N$ et $-3N$ (ensemble noté Q_1^-). Le quartier 2 correspond à l'ensemble des destinations d_i telles que l'adresse relative à cette destination soit comprise entre N et $2N$ (ensemble noté Q_2^+) ou entre $-3N$ et $-2N$ (ensemble noté Q_2^-). Nous considérons la valeur $\pm 2N$ à part, en tant que "limite". Les différents ensembles sont les suivants :

1. **Quartier 1.** $Q_1^+ = \{d_i : 0 \leq d_i - s \leq N\}$ et $Q_1^- = \{d_i : -4N < d_i - s \leq -3N\}$
2. **Quartier 2.** $Q_2^+ = \{d_i : N < d_i - s < 2N\}$ et $Q_2^- = \{d_i : -3N < d_i - s < -2N\}$
3. **Quartier 3.** $Q_3^+ = \{d_i : 2N < d_i - s < 3N\}$ et $Q_3^- = \{d_i : -2N < d_i - s < -N\}$
4. **Quartier 4.** $Q_4^+ = \{d_i : 3N \leq d_i - s < 4N\}$ et $Q_4^- = \{d_i : -N \leq d_i - s < 0\}$
5. **Limite.** $L_+ = \{d_i : d_i - s = 2N\}$ et $L_- = \{d_i : d_i - s = -2N\}$

La fonction ρ^*_{Oct} est définie par une fonction de routage particulière pour chacun de ces ensembles. Chacune de ces fonctions est très simple. Elles se construisent les unes à partir des autres. La preuve des propriétés définies par le prédicat $ValidNodep_{Oct}$ est aussi très simple. Par exemple, sur l'ensemble Q_1^+, le déplacement unitaire est simplement une incrémentation. On obtient la fonction $\rho_{Q_1^+}$ suivante :

$$\rho_{Q_1^+}(s,d) \triangleq \begin{cases} d & \textbf{si } s = d \\ s.\rho_{Q_1^+}(s+1,d) & \textbf{sinon} \end{cases}$$

On montre ensuite l'équivalence sur Q_1^+ entre cette fonction et la fonction initiale ρ_{Oct}, soit l'expression suivante :

$$\forall d_i \in Q_1^+ \Rightarrow \rho_{Oct}(s,d_i) = \rho_{Q_1^+}(s,d_i)$$

La fonction de routage sur l'ensemble Q_3^+ réalise une traversée du réseau puis une série de déplacements dans le sens des aiguilles d'une montre, c'est à dire les mêmes déplacements que sur l'ensemble Q_1^+. On obtient la définition suivante :

$$\rho_{Q_3^+}(s,d) \triangleq s.(s+2N).\rho_{Q_1^+}(s+2N+1,d)$$

Pour toute destination de Q_3^+, si on ajoute $2N+1$ à la source, on arrive dans Q_1^+. Puisque $\rho_{Q_1^+}$ est équivalente à ρ_{Oct}, $\rho_{Q_3^+}$ l'est aussi. Les huit fonctions sont construites sur ce même schéma. La principale interaction avec ACL2 consiste dans la décomposition de la preuve selon les différents cas ci-dessus. La preuve des propriétés spécifiées par $ValidNodep_{Oct}$ sur chacune des fonctions est très simple. Ces preuves sont la plupart du temps automatiques dans ACL2. □

Finalement, La définition de la fonction $Routing_{Oct}$ suit la définition de la fonction générique *Routing*, soit la définition suivante :

Définition 5.3 Routage de l'Octagon, fonction *OctagonRouting*
$Routing_{Oct}(\mathcal{M}, NodeSet_{Oct}) \triangleq$

$$\bigwedge_{m \in \mathcal{M}} List(Id\mathcal{M}(m), Frm\mathcal{M}(m), List(\rho_{Oct}(Org\mathcal{M}(m), Dest\mathcal{M}(m))))$$

Il reste à montrer que cette fonction produit une liste valide de voyages. On prouve facilement le théorème suivant :

Théorème 5.4 Type des routes de l'Octagon.

$$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet_{Oct}) \Rightarrow \mathcal{V}_{lstp}(Routing_{Oct}(\mathcal{M}, NodeSet_{Oct}))$$

Le tableau 5.1 détaille les différents aspects du code ACL2. La spécification et la validation de l'Octagon demande une vingtaine de fonction et une soixantaine de théorèmes. Cette concision est une caractéristique importante pour une spécification initiale. Le temps de preuve provient principalement des raisonnements sur l'arithmétique.

	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

TAB. 5.1 – Fonctions, théorèmes et temps de preuve pour la définition et la validation de l'Octagon

5.1.2 Routage en XY dans une grille à deux dimensions

Le routage en XY dans une grille à deux dimensions a servi d'illustration au chapitre 3 (voir page 64). Avant d'aborder cet algorithme plus en détails, nous définissons et validons les fonctions associées à l'ensemble des nœuds dans une grille à deux dimensions.

5.1.2.1 Définition des nœuds de la grille

Dans une grille, les nœuds sont représentés par des coordonnées. Puisque nous considérons une grille à deux dimensions, un nœud est représenté par une paire de coordonnées. Cette dernière est une liste composée de deux naturels : une abscisse et une ordonnée. Une coordonnée est reconnue par le prédicat $ValidNodep_{2D}$ qui est la fonction caractéristique du domaine de définition $GenNodeSet_{2D}$:

$$\forall x, ValidNodep_{2D}(x) \Leftrightarrow x \in GenNodeSet_{2D} \tag{5.12}$$

Les paramètres de la grille sont le nombre de nœuds dans chacune de ses dimensions. Le prédicat $ValidParamsp_{2D}$ reconnaît les paramètres de la grille. Soient N_X et N_Y, le nombre de nœuds selon la première et la seconde dimension. L'ensemble des nœuds est l'ensemble des coordonnées allant de $(0, 0)$ à $((N_X - 1), (N_Y - 1))$. La fonction $NodeSetGen_{2D}$ génère l'ensemble des nœuds de la grille. Elle est définie de la manière suivante. Soit la fonction $XGen(N_X, y)$ qui prend pour arguments le nombre N_X de nœuds selon la première dimension et une ordonnée y constante. Elle génère toutes les paires possibles pour ce y fixé. La fonction $NodeSetGen_{2D}$ calcule les coordonnées en appliquant la fonction $XGen$ pour toutes les valeurs de y comprises entre zéro et $N_Y - 1$. Pour valider la définition des nœuds de la grille, on prouve le théorème suivant :

Théorème 5.5 Validation des nœuds de la grille.

$$\forall pms, ValidParamsp_{2D}(pms) \Rightarrow \forall x \in NodeSetGen_{2D}(pms), ValidNodep_{2D}(x)$$

5.1.2.2 Algorithme de routage en XY

L'algorithme de routage en XY est décrit au chapitre 3 (*c.f.* exemple de la section 3.6.1, page 64). Nous rappelons ici son fonctionnement. Soit $s = (s_x, s_y)$ un nœud contenant un paquet destiné au nœud $d = (d_x, d_y)$. Dans cet algorithme, les abscisses sont prioritaires. Si l'abscisse de la destination d est supérieure - respectivement inférieure - à celle de l'origine s, le prochain nœud est le nœud supérieur $(s_x + 1, s_y)$ - ou inférieur $(s_x - 1, s_y)$ selon l'axe X. Sinon, les abscisses sont égales et on compare les ordonnées. Si l'ordonnée de d est supérieure - ou inférieure - à celle de s, le prochain nœud est le nœud supérieur $(s_x, s_y + 1)$ - ou inférieur $(s_x, s_y - 1)$ - selon l'axe Y. Finalement, les déplacements unitaire sont définis par la fonction suivante :

$$\mathcal{L}_{xy}(s,d) \triangleq \begin{cases} d & \textbf{si} \quad s = d \\ (s_x + 1, s_y) & \textbf{si} \quad s_x < d_x \\ (s_x - 1, s_y) & \textbf{si} \quad s_x > d_x \\ (s_x, s_y + 1) & \textbf{si} \quad s_x = d_x \wedge \ s_y < d_y \\ (s_x, s_y - 1) & \textbf{si} \quad s_x = d_x \wedge s_y > d_y \end{cases}$$

La fonction ρ_{xy} est définie comme l'application successive des déplacements unitaires :

Définition 5.4 Fonction de routage en XY, fonction ρ_{xy}.

$$\rho_{xy}(s,d) \triangleq \begin{cases} d & \textbf{si } s = d \\ s.\rho_{xy}(\mathcal{L}_{xy}(s,d), d) & \textbf{sinon} \end{cases}$$

La distance entre deux nœuds est définie par la somme des valeurs absolues des différences entre les coordonnées, soit l'expression suivante :

$$dist_{xy}(s,d) = |d_x - s_x| + |d_y - s_y| \tag{5.13}$$

Pour prouver que la fonction de routage ρ_{xy} termine, il suffit de prouver que les déplacements unitaires réduisent cette distance, soit le théorème suivant :

Théorème 5.6 Terminaison de la fonction ρ_{xy}.

$$\forall s, d \in GenNodeSet_{2D}, s \neq d \Rightarrow dist_{xy}(\mathcal{L}_{xy}(s,d), d) < dist_{xy}(s,d)$$

Preuve: Par définition de $\mathcal{L}_{xy}$. □

Pour montrer que la fonction de routage ρ_{xy} constitue une concrétisation valide de la fonction de routage de *GeNoC*, toute route produite par la fonction ρ_{xy} doit satisfaire le prédicat $ValidRoutep$, soit le théorème suivant :

Théorème 5.7 Validité des routes pour le routage en XY.

$$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet_{2D})$$
$$\Rightarrow \forall m \in_l \mathcal{M}, \forall r \in_l \rho_{xy}(Org_{\mathcal{M}}(m), Dest_{\mathcal{M}}(m)), ValidRoutep(r, m, NodeSet_{2D})$$

Preuve: La plupart des propriétés définies dans $ValidNodep$ sont triviales à établir. Les preuves ACL2 sont automatiques. La preuve que chaque route n'utilise que des nœuds de l'ensemble $NodeSet_{2D}$ est la seule nécessitant une interaction avec le démonstrateur. L'ensemble des nœuds est généré par la fonction $NodeSetGen_{2D}$ et est constitué de toutes les paires de naturels (x, y) telles que $0 \leq x < N_X$ et $0 \leq y < N_Y$. La stratégie est de montrer que tout ensemble de coordonnées satisfaisant ces inégalités est un sous-ensemble $NodeSet_{2D}$. Ensuite, il suffit de montrer que les routes produites par la fonction ρ_{xy} satisfont ces inégalités. La validation de cette tactique nécessite 5 lemmes et 2 fonctions. La preuve de la "fermeture" de ρ_{xy} sur $NodeSet_{2D}$ nécessite ensuite 30 lemmes. Pour seulement deux d'entre eux, ACL2 a besoin d'un conseil. □

On définit ensuite la fonction $Routing_{xy}$, concrétisation de la fonction $Routing$:

Définition 5.5 Routage en XY, fonction $Routing_{xy}$
$Routing_{xy}(\mathcal{M}, NodeSet_{2D}) \triangleq$

$$\bigwedge_{m \in \mathcal{M}} List(Id\mathcal{M}(m), Frm\mathcal{M}(m), List(\rho_{xy}(Org\mathcal{M}(m), Dest\mathcal{M}(m))))$$

Il reste à montrer que cette fonction produit une liste valide de voyages. On prouve facilement la théorème suivant :

Théorème 5.8 Type des routes en XY.

$$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet_{2D}) \Rightarrow \mathcal{V}_{lstp}(Routing_{xy}(\mathcal{M}, NodeSet_{2D}))$$

Le tableau 5.2 détaille les différents aspects des fichiers ACL2. Notre approche propose une spécification et une validation très concises de cet algorithme.

	Nbre de fonctions	Nbre de théorèmes	Temps de preuve (en secondes)	Taille
$NodeSet_{2D}$	8	5	~ 2	115 lignes
$Routing_{xy}$	6	44	~ 20	500 lignes
Total	14	49	~ 25	615 lignes

TAB. 5.2 – Fonctions, théorèmes et temps de preuve pour la définition et la validation du routage en XY

5.1.3 Routage en double Y dans une grille à deux dimensions

Cet algorithme de routage a été présenté au chapitre 2 (section 2.3.2.2, page 32). Il est construit sur l'algorithme précédent. La définition de l'ensemble des nœuds est identique. Nous reprenons la définition de ρ_{xy} ainsi que les lemmes et théorèmes qui lui sont associés.

Le caractère adaptatif signifie que plusieurs routes sont possibles entre deux nœuds. Un message emprunte *une* route parmi *toutes* les routes proposées. La fonction de routage définie dans *GeNoC* calcule toutes les routes possibles entre une source et une destination. La fonction concrète représentant l'algorithme en double Y calcule toutes les routes minimales

possibles entre deux coordonnées d'une grille. En pratique, une seule route est empruntée. En prouvant toutes les routes correctes, *la* route empruntée par un message est correcte.

Nous rappelons ici brièvement le fonctionnement de cet algorithme. Le principe est d'avoir, entre chaque paire de nœuds, deux canaux selon l'axe Y et un canal selon l'axe X. Le réseau est découpé en deux sous réseaux. Un premier sous-réseau pour les paquets voyageant dans le sens ascendant de l'axe X, un second pour ceux voyageant dans le sens descendant de l'axe X. Chaque sous-réseau partage le canal selon l'axe X et possède son propre canal selon l'axe Y. Cet algorithme revient à appliquer à chaque nœud, soit l'algorithme en XY, soit l'algorithme symétrique en YX. Le routage en double Y permet d'utiliser tous les chemins minimaux entre deux points. Par exemple, entre le nœud *(0 0)* et le nœud *(2 2)*, il propose six routes possibles (*c.f.* figure 5.3) ; entre les nœuds *(0 0)* et *(3 2)* il propose dix routes possibles (*c.f.* figure 5.4) [3].

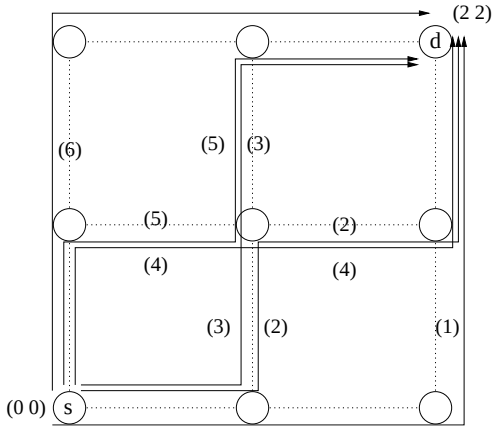


FIG. 5.3 – Chemins minimaux possibles dans une grille 3×3

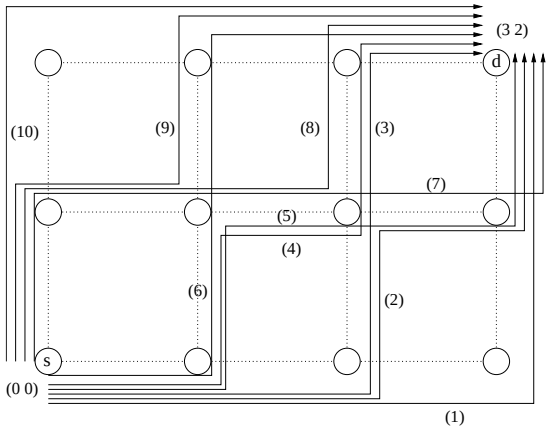


FIG. 5.4 – Chemins minimaux possibles dans une grille 4×3

Principe de la modélisation. Nous modélisons l'algorithme en double Y par une fonction qui applique alternativement l'algorithme en XY et en YX. Remarquons d'abord que pour tout nœud possédant une coordonnée c commune avec la destination d, aucun choix n'est possible. Le chemin minimal est d'effectuer les déplacements selon l'axe X si c est une ordonnée ou selon l'axe Y si c est une abscisse. Seuls les nœuds n'ayant aucune coordonnée commune avec la destination offre un choix entre les deux algorithmes. Soit $\mathcal{R}_{xy}(s, d)$ la fonction qui calcule l'ensemble des routes à partir de s en appliquant d'abord l'algorithme en XY. On définit de manière similaire la fonction $\mathcal{R}_{yx}(s, d)$. Pour un nœud source s donné, l'ensemble des routes possibles en appliquant d'abord l'algorithme en XY est constitué de la route r obtenue par cet algorithme à partir de s et de l'ensemble des routes obtenues par la fonction $\mathcal{R}_{yx}$ à partir de chaque nœud de r ayant une abscisse différente de celle de d.

[3] Notre modélisation de l'algorithme en double Y revient à calculer l'ensemble des combinaisons composées des déplacements selon l'axe X et l'axe Y nécessaires entre deux nœuds. La modélisation du routage en tant que problème combinatoire est une généralisation de notre approche. Elle est discutée en conclusion de ce manuscrit.

$$\mathcal{R}_{xy}(s,d) = \rho_{xy}(s,d) \cup \{(s \ldots s^{i-1}) \cup r_i, \forall r_i \in \mathcal{R}_{yx}(s^i,d), \forall s^i \in \rho_{xy}(s,d), s^i_x \neq d_x\}$$

On définit de manière analogue l'ensemble des routes possibles en appliquant d'abord l'algorithme en YX, soit :

$$\mathcal{R}_{yx}(s,d) = \rho_{yx}(s,d) \cup \{(s \ldots s^{i-1}) \cup r_i, \forall r_i \in \mathcal{R}_{xy}(s^i,d), \forall s^i \in \rho_{yx}(s,d), s^i_y \neq d_y\}$$

L'ensemble des routes possibles entre deux nœuds est l'union de ces deux ensembles de routes :

$$\mathcal{R}_{dy}(s,d) = \mathcal{R}_{xy}(s,d) \cup \mathcal{R}_{xy}(s,d)$$

Algorithme en YX. L'algorithme en YX déplace les paquets selon l'axe Y puis selon l'axe X. Son résultat entre une source s et une destination d est obtenu à partir de l'algorithme en XY en inversant le résultat du routage en XY de d vers s :

Définition 5.6 Routage en YX.

$$\rho_{yx}(s,d) = Rev(\rho_{xy}(d,s))$$

La fonction ρ_{yx} n'est pas récursive, le problème de l'arrêt des calculs ne se pose pas. La validité des routes produites par cette fonction est déduite de celle des routes produites par l'algorithme en XY.

Théorème 5.9 Validité des routes de l'algorithme en YX.

$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet_{2D})$
$\Rightarrow \forall m \in_l \mathcal{M}, \forall r \in_l \rho_{yx}(Org_{\mathcal{M}}(m), Dest_{\mathcal{M}}(m)), ValidRoutep(r, m, NodeSet_{2D})$

Preuve: Par définition de la fonction ρ_{yx} et la validité des routes de l'algorithme en XY (théorème 5.7). □

Réalisation de l'algorithme en double Y. L'algorithme en double Y est principalement réalisé par la fonction ζ_{dy} qui prend pour arguments une liste, *sources*, de nœuds, une destination d, un drapeau flg et une liste, $prefixes$, de listes de nœuds. Elle retourne une liste de routes. Chaque route de cette liste part d'un nœud initialement présent dans la liste *sources* et arrive à d. Le drapeau flg est un booléen ayant la signification suivante. Si flg est "vrai", la fonction ζ_{dy} applique l'algorithme en XY. Sinon, elle applique l'algorithme en YX. À chaque appel récursif la valeur du drapeau est inversée. La liste des routes possibles entre s et d est constituée des deux listes de routes obtenues pour les deux valeurs possibles du drapeau. La liste $prefixes$ permet de mémoriser les calculs. L'algorithme s'arrête si la liste *sources* est vide ou si son premier nœud possède une coordonnée de la destination d. La définition de cette fonction, explicitée ci-après, est la suivante :

Définition 5.7 Algorithme en Double Y, fonction ζ_{dy}.
$\zeta_{dy}(sources, d, flg, prefixes) \triangleq$
if $sources = \epsilon \vee \neg CloserListp(sources, d)$ **then** ϵ
else $sources = s.sources' \wedge prefixes = p.prefixes'$
if $s_x = d_x \vee s_y = d_y$ **then** ϵ
else
let *routes* **be** $\Psi(s, d, p, flg)$ **in**
$routes \sqcup \zeta_{dy}(sources', d, flg, prefixes')$
endif
endif
où $\Psi(s, d, p, flg)$ *est :*
if flg **then**
let r_{xy} **be** $\rho_{xy}(s, d)$ **and** *NewPrefixes* **be** $ComputePrefixes(r_{xy}, d)$ **in**
$(p \sqcup r_{xy}).\zeta_{dy}(GetCandidatesXY(r_{xy}, d), d, \neg flg, \Lambda_{e \in NewPrefixes} \sqcup(p, e)))$
else
let r_{yx} **be** $\rho_{yx}(s, d)$ **and** *NewPrefixes* **be** $ComputePrefixes(r_{yx}, d)$**in**
$(p \sqcup r_{yx}).\zeta_{dy}(GetCandidatesYX(r_{yx}, d), d, \neg flg, \Lambda_{e \in NewPrefixes} \sqcup(p, e))$
endif

La terminaison des calculs est assurée par le fait que chaque nouvelle source est plus proche de d que les précédentes. Autrement dit, plus la position d'un nœud dans *sources* est élevée plus ce nœud est proche de d. Cette propriété est un invariant et doit être vérifié par la liste *sources* initiale. Le prédicat $CloserListp(lst, d)$ vérifie qu'une liste, lst, contient des nœuds se rapprochant de d. Le premier test de la fonction ζ_{dy} est d'assurer que cet invariant est préservé et que *sources* contient au moins un nœud.

Passé ce premier test, on peut écrire *sources* comme la concaténation d'un nœud s et du reste $sources'$ de cette liste. Nous écrivons la liste *prefixes* d'une manière similaire. (Si *prefixes* est vide, nous obtenons simplement $\epsilon.\epsilon$.) Si le nœud s possède une coordonnée commune avec la destination, le calcul s'arrête. Sinon, on calcule l'ensemble des routes pour le nœud s. Ce calcul, pour des raisons de présentation, est abrégé par la fonction Ψ que nous détaillons plus tard. Les routes produites par Ψ sont ajoutées aux routes obtenues pour les nœuds restants de la liste *sources* et le reste des préfixes.

La fonction Ψ applique, en fonction de la valeur de flg, soit la fonction ρ_{xy}, soit la fonction ρ_{yx}. La fonction $ComputePrefixes(r, d)$ calcule tous les préfixes suggérés par la route r pour la destination d. Les nouvelles sources sont calculées par la fonction $GetCandidateXY$ ou la fonction $GetCandidatesYX$ en fonction de l'algorithme de routage utilisé. À cette route obtenue par ρ_{xy} ou ρ_{yx}, est ajouté le préfixe courant. La route ainsi obtenue est ajoutée aux routes produites par la fonction ζ_{dy} appliquée aux nouvelles sources, à la valeur opposée du drapeau, et aux nouveaux préfixes obtenus par $ComputePrefixes$ auxquels a été ajouté le préfixe courant.

Pour prouver que cette fonction est une concrétisation valide, il faut prouver que chacune des routes produites satisfait le prédicat *ValideNodep*, soit le théorème suivant :

Théorème 5.10 Validité des routes de l'algorithme en double Y.

$$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet_{2D})$$
$$\Rightarrow \forall m \in_l \mathcal{M}, \forall r \in_l \zeta_{dy}(Org_{\mathcal{M}}(m), Dest_{\mathcal{M}}(m)), ValidRoutep(r, m, NodeSet_{2D})$$

Preuve: La preuve est faite par induction sur la structure la définition de ζ_{dy}. Le cas de base est trivial. La fonction ζ_{dy} applique alternativement les fonctions ρ_{xy} et ρ_{yx}. La preuve de l'étape d'induction est effectuée en faisant appel à la correction de ces fonctions (théorèmes 5.7 et 5.9). □

La totalité des routes est l'union des routes obtenues en appliquant d'abord l'algorithme en XY ou YX.

Définition 5.8 Fonction de routage en doubleY, ρ_{dy}

$$\rho_{dy}(s, d) = \zeta_{dy}((s), d, true, \epsilon) \sqcup \zeta_{dy}((s), d, false, \epsilon)$$

Exemple 11 *Considérons la figure 5.4 ; soit le calcul des routes possibles entre les nœuds (0 0) et (3 2) dans une grille 4 × 3. Pour l'exemple, nous considérons que le drapeau est initialement "vrai". La liste* **sources** *contient initialement une unique source s. La liste* **prefixes** *contient initialement aucun préfixe. La première route, $r_{xy,0}$ est obtenue par la fonction ρ_{xy} appliquée à s (c.f. figure 5.5). Cette route suggère deux nouvelles "sources" : les nœuds s_1 et s_2. Le nouveau préfixe est uniquement constitué de s.*

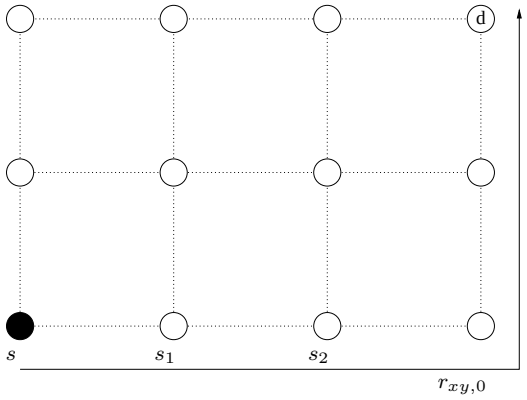


FIG. 5.5 – Première itération de l'algorithme : application de $\rho_{xy}(s, d)$

L'algorithme est appliqué récursivement sur s_1 puis s_2. L'application sur s_1 produit trois routes (c.f. figure 5.6). La première est obtenue par l'ajout de s au résultat de $\rho_{yx}(s_1, d)$, la deuxième par l'ajout de s et s_1 au résultat de $\rho_{xy}(s_3, d)$, et la troisième par l'ajout de s, s_1 et s_3 au résultat de $\rho_{yx}(s_4, d)$. L'application sur s_2 produit les deux dernières routes (c.f. figure 5.7). La première est obtenue par l'ajout de s et s_1 au résultat de $\rho_{yx}(s_2, d)$, la seconde par l'ajout de s, s_1 et s_2 au résultat de $\rho_{xy}(s_4, d)$. Le résultat de cet algorithme dans le cas où le drapeau est initialement "faux" produit de manière similaire quatre routes. Ainsi, nous obtenons la totalité des routes entre s et d.

Finalement, on définit ensuite une fonction compatible avec la définition générique.

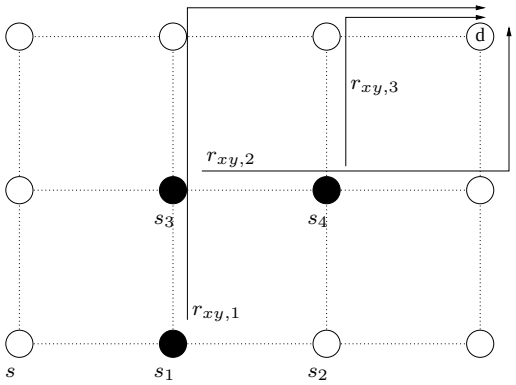


FIG. 5.6 – Itérations 2,3 et 4 : application de $\rho_{yx}(s_1, d)$, $\rho_{xy}(s_3, d)$ et $\rho_{yx}(s_4, d)$

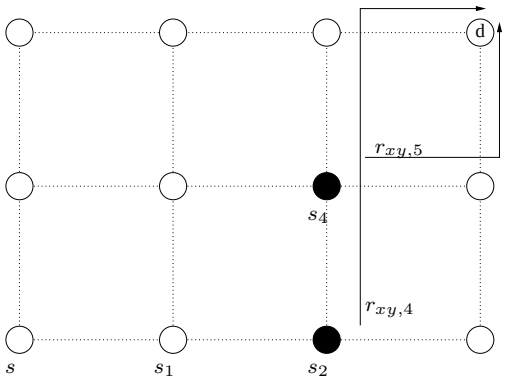


FIG. 5.7 – Itérations 5 et 6 : application de $\rho_{yx}(s_2, d)$ et $\rho_{xy}(s_4, d)$

Définition 5.9 Routage en double Y, fonction $Routing_{dy}$**.**

$$Routing_{dy}(\mathcal{M}, NodeSet_{2D}) \triangleq \bigwedge_{m \in_l \mathcal{M}} List(Id_{\mathcal{M}}(m), Frm_{\mathcal{M}}(m), \rho_{dy}(Org_{\mathcal{M}}(m), Dest_{\mathcal{M}}(m)))$$

Pour terminer la validation, nous prouvons que cette fonction produit une liste valide de voyages à partir d'une liste valide de missives, soit le théorème suivant :

Théorème 5.11 Type du routage en double Y.

$$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet_{2D}) \Rightarrow \mathcal{V}_{lstp}(Routing_{dy}(\mathcal{M}, NodeSet_{2D}))$$

Preuve: Par définition de la fonction ρ_{dy}. □

5.2 Réalisations de la fonction *Scheduling*

La méthodologie pour valider toute concrétisation de la politique d'ordonnancement suit le formalisme présenté au chapitre 3 (section 3.7, page 67). Elle est très simple et consiste en la définition d'une fonction concrète $Scheduling_\sharp$ ayant la fonctionnalité de *Scheduling* et satisfaisant les obligations de preuve 3.6, 3.7, 3.8, 3.9 et 3.10 définies sur la fonction *Scheduling*. Ces contraintes sont indépendantes les unes des autres. L'ordre de leur preuve est sans importance.

Dans cette section, nous exposons trois réalisations de la fonction *Scheduling*. Les deux premières sont deux exemples de commutation dans les réseaux ; la dernière présente un arbitrage de bus.

5.2.1 Commutation par circuits

Modélisation de la commutation par circuits. Comme nous l'avons décrit en section 2.3.3.4 page 38, en commutation par circuits le chemin entre le nœud source et le nœud

destination est réservé pendant toute la durée de la communication. Autrement dit, pour qu'un voyage soit ordonnancé, les nœuds de sa route ne doivent pas être utilisés par les voyages déjà ordonnancés.

La fonction $ExistsValidRoutep(routes, prev)$ prend pour arguments une liste $routes$ de routes et une liste $prev$ de nœuds déjà utilisés. Elle retourne une route n'utilisant aucun nœud de la liste $prev$. Si une telle route n'existe pas, elle retourne la constante $faux$. Formellement, la fonction $ExistsValidRoutep$ exprime la propriété suivante :

$$ExistsValidRoutep(routes, prev) \triangleq \exists r \in routes, \forall i \in [0; Len(r) - 1], r[i] \notin prev$$

La fonction $CircuitScheduler$ prend pour arguments une liste $TrLst$ de voyages, la liste $Scheduled$ des voyages ordonnancés, la liste $Delayed$ des voyages retardés et la liste $prev$ des nœuds déjà utilisés. Les trois dernières listes sont initialement vides. Elle retourne la liste des voyages ordonnancés et la liste des voyages retardés. Les voyages sont accumulés dans l'ordre décroissant des identifiants. Le voyage avec le plus faible identifiant se retrouve en dernière position dans la liste $Scheduled$ ou $Delayed$. L'ordre de ces voyages reflète la priorité entre ceux-ci. Pour conserver l'ordre des identifiants dans ces listes, la fonction $CircuitScheduler$ inverse ses accumulateurs à la fin du calcul. La liste $TrLst$ des voyages est parcourue dans l'ordre des identifiants. À chaque étape de calcul, si la fonction $ExistsValidRoutep$ fournit une route pour le voyage courant, ce voyage est ordonnancé. Seule cette route est conservée. (La fonction $UpdateRoute(tr, route)$ associe l'unique $route$ au voyage tr). Les nœuds de cette route sont ajoutés à la liste $prev$. Sinon, le voyage est ajouté à la liste $Delayed$. La définition de $CircuitScheduler$ est la suivante :

Définition 5.10 Fonction $CircuitScheduler$**.**
$CircuitScheduler(TrLst, Scheduled, Delayed, prev) \triangleq$
if $TrLst = \epsilon$ **then**
$List(Rev(Scheduled), Rev(Delayed))$
else $TrLst = tr.T'$
let $r?$ **be** $ExistsValidRoutep(Routes\mathcal{V}(tr), prev)$ **in**
if $r? \neq faux$ **then**
$CircuitScheduler(T', UpdateRoute(tr, r?) \sqcup Scheduled, Delayed, r? \sqcup prev)$
else
$CircuitScheduler(T', Scheduled, tr \sqcup Delayed, prev)$
endif
endif

Une fonction conforme à la fonctionnalité générique $Scheduling$ est définie comme suit. Si toutes les tentatives ont été utilisées, aucun voyage n'est ordonnancé ; la liste $scheduled$ est vide et la liste $delayed$ est la liste $TrLst$. Sinon, la fonction $CircuitScheduler$ est appelée et chaque nœud du réseau consomme une tentative.

Définition 5.11 Fonction d'ordonnancement par circuits.
$Scheduling_{ct}(TrLst, NodeSet, att) \triangleq$
if $SumOfAttempts(att) = 0$ **then**
$List(\epsilon, TrLst, att)$

else
  Let $(Scheduled\ Delayed)$ **be** $CircuitScheduler(TrLst, \epsilon, \epsilon, \epsilon)$ **in**
    $List(Scheduled, Delayed, ConsumeAttempts(att))$
endif

Cette fonction présente la même fonctionnalité que la fonction *Scheduling*. Il reste à prouver qu'elle satisfait les obligations de preuve.

Validation et conformité de la commutation par circuits. Avant de montrer la conformité de $Scheduling_{ct}$ avec la fonction générique *Scheduling*, nous en prouvons la correction. L'ordonnancement est correct si les routes des voyages ordonnancés ne partagent aucun nœud. Soit *CtScheduled* la liste des voyages ordonnancés produite par la fonction $CircuitScheduler(TrLst, att, prev)$. Pour tout voyage de la liste *CtScheduled*, l'intersection de sa route avec celle de chaque autre voyage est vide. Ceci se traduit par le théorème suivant :

Théorème 5.12 Correction de l'ordonnancement par circuits.
$\forall i \in [0; Len(CtScheduled) - 2]$,

$$\bigwedge_{j=i+1}^{Len(CtScheduled)-1} Routes_{\mathcal{V}}(CtScheduled[i])[0] \sqcap Routes_{\mathcal{V}}(CtScheduled[j])[0] = \epsilon$$

Preuve: La fonction *CircuitScheduler* utilise des accumulateurs pour calculer les listes *CtScheduled* et *CtDelayed* des voyages ordonnancés et des voyages retardés. Ce lemme ne concerne que la première liste. Pour simplifier la preuve, on définit une fonction qui calcule cette liste sans utiliser d'accumulateur. Un voyage est ordonnancé uniquement si sa route a une intersection vide avec les nœuds des routes précédentes. À partir de cette propriété, il vient par induction que les routes des voyages ordonnancés n'ont aucun nœud en commun. □

La première obligation de preuve à satisfaire est de montrer que la fonction $Scheduling_{ct}$ produit des listes valides de voyages à partir d'une liste valide de voyages, soit le théorème suivant :

Théorème 5.13 Type des voyages pour la commutation par circuits.
Soient $Scheduled = \pi_1^3 \circ Scheduling_{ct}(\mathcal{V}, NodeSet, att)$ *et*
$Delayed = \pi_2^3 \circ Scheduling_{ct}(\mathcal{V}, NodeSet, att)$, *alors* :

$$\forall \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}) \Rightarrow \mathcal{V}_{lstp}(Scheduled) \wedge \mathcal{V}_{lstp}(Delayed)$$

Preuve: Une liste de voyages est valide si la liste des identifiants est une liste de naturels et sans doublon. Pour montrer l'absence de doublon, il suffit de montrer que si un identifiant n'est pas un identifiant d'un voyage de la liste, $TrLst$, initiale, il n'est un identifiant ni de la liste *Scheduled*, ni de la liste *Delayed*. La preuve est ensuite faite par induction sur la structure de la définition de $Scheduling_{ct}$. □

La fonction *ConsumeAttempts* décrémente le nombre de tentatives de chaque nœud du réseau d'une unité. Par induction, il vient trivialement qu'elle réduit la somme des tentatives

et que l'obligation de preuve 3.7 (consommation d'au moins une tentative, page 68) est satisfaite.

Les obligations de preuve restantes sont toutes prouvées par induction sur la structure de la fonction $Scheduling_{ct}$.

5.2.2 Commutation par paquets

Modélisation de la commutation par paquets Comme nous l'avons décrit en section 2.3.3.4 (voir page 38), en commutation par paquets, un nœud n'est occupé que pendant le temps où il traîte un paquet. Autrement dit, pour qu'un voyage soit ordonnancé, le nœud de chaque étape de sa route ne doit pas être utilisé par d'autres paquets.

La fonction *ExistsValidRoutep* prend pour arguments une liste, *routes*, de routes pas encore élues, et une liste de listes de nœuds, *prev*. La liste *prev* est une liste dont chaque élément $prev[i]$ est la liste des nœuds qui traîtent un paquet à l'étape i parmi les routes des voyages déjà ordonnancés. Une route de la liste *routes* est élue si cette route r n'utilise pas de nœuds appartenant à $prev[j]$ pour chacune de ses étapes j. La fonction *ExistsValidRoutep* retourne une route de la liste *routes* qui satisfait cette condition, ou la constante *faux* si une telle route n'existe pas. Formellement, cette fonction exprime l'expression suivante :

$$ExistsValidRoutep(routes, prev) \triangleq \exists r \in routes, \forall i \in [0; Len(r) - 1], r[i] \notin prev[i]$$

La fonction d'ordonnancement par paquet *PacketScheduler* prend pour arguments une liste *TrLst* de voyages et trois accumulateurs. Les deux premiers - nommés *Scheduled* et *Delayed* - accumulent les voyages ordonnancés et retardés. Le troisième mémorise les nœuds utilisés. Cette fonction retourne la valeur des accumulateurs *Scheduled* et *Delayed*. Les voyages sont accumulés dans l'ordre décroissant des identifiants. Le voyage avec le plus faible identifiant se retrouve en dernière position dans la liste *Scheduled* ou *Delayed*. L'ordre de ces voyages reflète la priorité entre ceux-ci. Pour conserver l'ordre des identifiants dans ces listes, la fonction *PacketScheduler* inverse ses accumulateurs à la fin du calcul. La fonction *PacketScheduler* parcourt la liste de voyages et pour chacun d'entre eux prend la décision suivante. Si une route du voyage satisfaitt *ExistsValidRoutep*, le voyage est ordonnancé et les nœuds de la route choisie sont ajoutés dans les listes de *prev* (fonction *UpdatePrev*). Sinon, le voyage est retardé. La définition de *PacketScheduler* est la suivante :

Définition 5.12 Fonction *PacketScheduler*.
$PacketScheduler(TrLst, scheduled, delayed, prev) \triangleq$
if $TrLst = \epsilon$ **then**
  $List(Rev(Scheduled), Rev(Delayed))$
else $TrLst = tr.T'$
  let $r?$ **be** $ExistsValidRoutep(Routes\mathcal{V}(tr), prev)$ **in**
   if $r? \neq faux$ **then**
    $PacketScheduler(T', UpdateRoute(tr, r?) \sqcup Scheduled, Delayed, r? \sqcup prev)$
   else
    $PacketScheduler(T', Scheduled, tr \sqcup Delayed, prev)$
   endif
endif

Une fonction conforme à la fonctionnalité générique *Scheduling* est définie. Si toutes les tentatives ont été utilisées, aucun voyage n'est ordonnancé ; la liste *Scheduled* est vide et la liste *Delayed* est la liste *TrLst*. Sinon, la fonction *PacketScheduler* est appelée et chaque nœud consomme une tentative.

Définition 5.13 Fonction d'ordonnancement par paquets.
$Scheduling_{pt}(TrLst, NodeSet, att) \triangleq$
if $SumOfAttempts(att) = 0$ **then**
$List(\epsilon, TrLst, att)$
else
Let $(Scheduled\ Delayed)$ **be** $PacketScheduler(TrLst, \epsilon, \epsilon, \epsilon)$ **in**
$List(Scheduled, Delayed, ConsumeAttempts(att))$
endif

Validation et conformité de la commutation par paquets Avant de prouver la conformité de la fonction $Scheduling_{pt}$ avec la fonction générique *Scheduling*, nous en montrons la correction. Soit *PktScheduled* la liste des voyages ordonnancés produite par la fonction *PacketScheduler*. En commutation par paquets, l'ordonnancement est correct si pour chaque étape des routes de *PktScheduler* un nœud n'est utilisé qu'une seule fois. Soit $CreateSteps(TrLst)$ une fonction prenant pour argument une liste de voyages et construisant la liste des nœuds utilisés à chaque étape des routes de *TrLst*. Soit *steps* la liste $CreateSteps(PktScheduled)$. La liste $steps[i]$ représente la liste des nœuds utilisés à l'étape i des routes de *PktScheduled*. La correction de l'ordonnancement par paquets s'exprime par l'absence de doublon dans chaque liste de *steps*, soit le théorème suivant :

Théorème 5.14 Correction de l'ordonnancement par paquets.

$$\bigwedge_{i=0}^{Len(steps)-1} NoDuplicatesp(steps[i])$$

Preuve: La fonction *PacketScheduler* utilise des accumulateurs pour calculer les listes *PktScheduled* et *PktDelayed* des voyages ordonnancés et des voyages retardés. Ce lemme ne concerne que la première liste. Pour simplifier la preuve, on définit une fonction qui calcule cette liste sans utiliser d'accumulateur. Une route est élue si chacune de ses étapes j n'ajoute pas de doublon aux nœuds déjà utilisés à l'étapes j. À partir de cette propriété et par induction, il vient que les listes des nœuds utilisés à chacune des étapes des voyages ordonnancés n'ont pas de doublon. □

La validation de la fonction $Scheduling_{pt}$ est identique à la fonction $Scheduling_{ct}$. Elle repose sur les mêmes lemmes et les mêmes preuves.

5.2.3 Arbitrage du bus AMBA AHB

Nous montrons ici qu'un arbitre de bus est aussi représenté par la fonction *Scheduling*. Le bus AMBA AHB a été présenté en section 2.2.3 du chapitre 2 (voir page 26). Nous considérons ici uniquement l'arbitrage du bus.

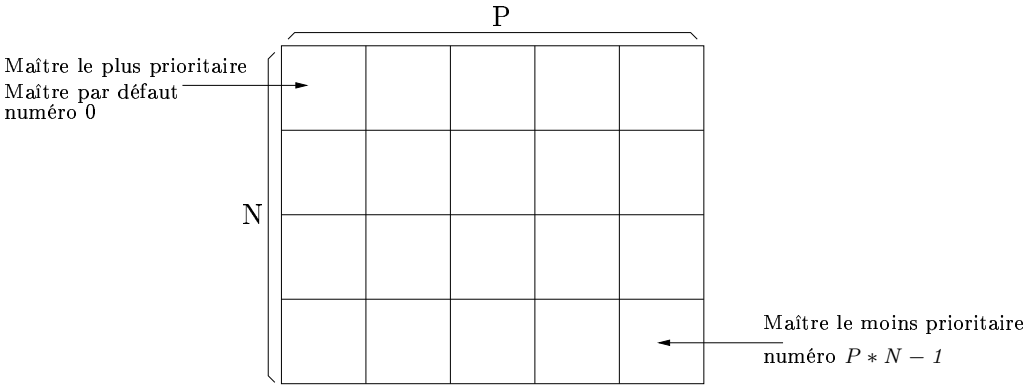


FIG. 5.8 – Matrice des priorités

Politique d'arbitrage. Le bus AHB comporte toujours un seul arbitre dont le rôle est d'attribuer l'accès au bus à un unique maître. Pour élire celui-ci, l'arbitre utilise un schéma de priorité qui n'est pas imposé par la spécification [ARM00]. L'arbitre doit respecter des règles, la principale étant d'assurer l'exclusion mutuelle des accès au bus. Une autre restriction, due à l'absence d'état "haute impédance" du bus, est la définition d'un maître par défaut qui accède au bus lorsque aucune requête n'est émise. Le schéma des priorités est le suivant.

La hiérarchie entre les maîtres est un point central du protocole. Les requêtes des maîtres sont rassemblées dans une matrice définissant le schéma de priorité (*c.f* Figure 5.8). Le nombre P définit le nombre de niveaux de priorité. Les niveaux sont numérotés de 0 à $P - 1$ en commençant par le haut ; l'étage 0 est considéré comme le plus prioritaire. À chaque niveau, le bus est attribué en suivant une politique de "tourniquet". Le nombre N définit le nombre d'unités par étage de priorité. Sur un même étage, les éléments sont numérotés de 0 à $N - 1$. Le nombre de maîtres est égal au produit $P * N$. Le numéro $MASTER\_NUMBER$ de l'unité élue est calculé selon l'équation suivante :

$$MASTER\_NUMBER = \text{numéro de l'étage} * N + \text{numéro de la colonne} \tag{5.14}$$

La maître numéro 0 est considéré comme le plus prioritaire et défini comme le maître par défaut.

Modélisation fonctionnelle et validation de cette politique. La fonction $Scheduling_{ahb}$ correspond à la fonction générique *Scheduling*. Elle repose sur une fonction plus élémentaire - la fonction $Arbiter_{ahb}$ - modélisant la politique d'arbitrage.

La fonction $Arbiter_{ahb}$ prend pour arguments une matrice contenant les requêtes, le nombre N d'unités par niveaux de priorité et le numéro du dernier maître détenteur du bus. Elle retourne le numéro du maître auquel le bus a été attribué.

Les étapes de calcul de la fonction $Arbiter_{ahb}$ sont les suivantes :

1. déterminer le premier étage contenant au moins une requête,
2. extraction de cet étage hors de la matrice,
3. déterminer l'élu de cet étage,
4. à partir de la position de l'étage, calculer le numéro de l'élu.

Les deux étapes les plus importantes sont la première et la troisième. Nous détaillons uniquement celles-ci.

Tout d'abord, des prédicats permettent de définir les concepts utilisés dans la modélisation. Le prédicat $Zerosp(l)$ reconnaît une liste composée uniquement de 0 ; $01Bitp(l)$ reconnaît l comme une liste de bits. Une matrice est représentée par une liste de listes. Le prédicat $NoRequestMatrixp(M)$ reconnaît une matrice ne contenant que des 0, soit aucune requête.

Si la matrice MREQ des requêtes ne contient que des zéros, le bus est attribué au maître par défaut. Sinon, la première ligne de la matrice contenant au moins une requête est le premier élément de MREQ tel que la négation du prédicat $Zerosp$ est satisfaite. La fonction $FindStage(MREQ)$ réalise la première étape de $Arbiter_{ahb}$; elle retourne le numéro de l'étage le plus prioritaire contenant au moins une requête. Sa définition est la suivante :

Définition 5.14 Détermination de l'étage le plus prioritaire.
$FindStage(MREQ) \triangleq$
if $MREQ = \epsilon \vee NoRequestMatrixp(MREQ)$ **then** *0*
else $MREQ = m.MREQ'$
  if $\neg Zerosp(m)$ **then** *0* *;; on a trouvé l'étage*
  else
    $1 + FindStage(MREQ')$
endif

Soit S_{gtd} le numéro fourni par la fonction *FindStage*. Cette fonction est correcte si tout étage plus prioritaire que S_{gtd} ne contient aucune requête. Et, si au moins une requête a été émise, l'étage S_{gtd} contient au moins une requête. La première condition s'éxprime par le fait que pour tout i inférieur à S_{gtd}, MREQ[i] ne contient que des zéros.

Théorème 5.15 Correction de *FindStage* 1.

$$\forall MREQ, \forall i \in \mathbb{N}, i < FindStage(MREQ) \Rightarrow Zerosp(MREQ[i])$$

La seconde s'éxprime par le fait que si $MREQ$ contient au moins une requête, l'étage S_{gtd} aussi.

Théorème 5.16 Correction de *FindStage* 2.

$$\forall MREQ, \neg NoRequestMatrixp(MREQ) \Rightarrow \neg Zerosp(MREQ[FindStage(MREQ)]$$

Les preuves de ces théorèmes par ACL2 ne demandent aucun lemme.

L'étape importante suivante consiste dans la détermination du prochain maître dans cet étage sélectionné par la fonction *FindStage*. À un niveau de priorité donné, les maîtres ont tous la même priorité et l'accés suit une politique du "tourniquet".

Le calcul est effectué par un parcours circulaire de l'étage à partir de la position, p_{last}, du dernier détenteur du bus (*c.f.* Figure 5.9). L'étage est coupé en deux parties au niveau de la position p_{last}. Si la fin de l'étage contient au moins une requête, le parcours débute à la position suivante de p_{last}. Sinon, le parcours débute par le premier élément de l'étage. La

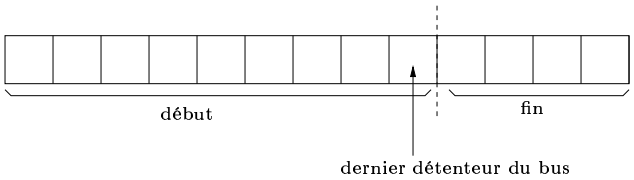


FIG. 5.9 – Calcul du "round-robin".

première partie de l'étage est obtenue par la fonction $Firstn(n, l)$ qui retourne les n premiers éléments de n (soit les éléments $l[0] \ldots l[n-1]$). La dernière partie est obtenue par la fonction $Lastn(n, l)$ qui retourne les derniers éléments de l (soit les éléments $l[n] \ldots l[Len(l) - 1]$). Nous définissons la fonction $FindNextReq(l)$ qui retourne la position (comptée de 0 et à partir du premier élément de l) de la première requête rencontrée dans l. La fonction $RoundRobin(Stage, p_{last})$ prend pour arguments un étage et la position du dernier possesseur du bus dans cet étage. Elle retourne la position, dans cet étage, du prochain maître. Sa définition est la suivante :

Définition 5.15 Fonction *RoundRobin*
$RoundRobin(Stage, p_{last}) \triangleq$
if $Zerosp(Stage)$ **then** 0
else
if $Zerosp(Lastn(p_{last} + 1, Stage))$[4] **then**
$FindNextReq(Firstn(p_{last} + 1, Stage))$
else
$p_{last} + 1 + FindNextReq(Lastn(p_{last} + 1, Stage))$
endif
endif

Nous montrons sur cette fonction une propriété de sûreté exprimant que l'accès au bus change de main. Supposons que le maître m_{last} anciennement détenteur du bus émet toujours une requête et qu'il existe au moins une autre requête au même étage, le prochain maître détenteur du bus n'est pas le maître m_{last}.

Théorème 5.17 Propriété sur la fonction *RoundRobin*.
$\forall Stage, 01Bitp(Stage), \forall i, m_{last} \in \mathbb{N}, i \neq m_{last},$

$$Stage[m_{last}] = 1 \wedge Stage[i] = 1 \rightarrow RoundRobin(Stage, m_{last}) \neq m_{last}$$

Ceci conclut la validation de la politique d'arbitrage, soit de la fonction $Arbiter_{ahb}$. Nous définissons et validons maintenant la fonction $Scheduling_{ahb}$.

Conformité de l'arbitre avec *Scheduling*. La fonction $Scheduling_{ahb}$ prend pour arguments une liste $TrLst$ de voyages, l'ensemble $NodeSet$ des nœuds, la liste des tentatives et une liste, $opts$, contenant le nombre N ($= opts[0]$) de niveaux de priorité, le nombre P (=

[4]Dans le cas où le dernier détenteur du bus est le dernier nœud de la liste *Stage*, l'appel à la fonction *Lastn* retourne `nil`, et l'appel à *Zerosp* retourne `t`. La fonction *Firstn* retourne l'intégralité de la liste *Stage*.

$opts[1]$) de maîtres par niveau et le numéro $LastGranted$ ($= opts[2]$) du dernier maître détenteur du bus. Elle retourne une liste contenant un voyage dont la route débute par le maître détenteur du bus. Les autres voyages de $TrLst$ constituent la liste des voyages retardés. Les calculs effectués par $Scheduling_{ahb}$ sont répartis en trois étapes :

1. créer la matrice $MREQ$ à partir de $TrLst$
2. appeler $Arbiter_{ahb}$
3. extraire de $TrLst$ le bon voyage

La matrice est créée en deux étapes. Premièrement, $MREQ$ est initialisée à une matrice ne contenant que des zéros, Ensuite, pour chaque premier nœud n_0 de la première route de chaque voyage de $TrLst$, un 1 est inséré dans la matrice à la position correspondant à celle de n_0 (*c.f.* Figure 5.10). La fonction initialisant la matrice est nommée $InitREQ$. Le remplissage de la matrice est effectuée par la fonction $FillREQ$.

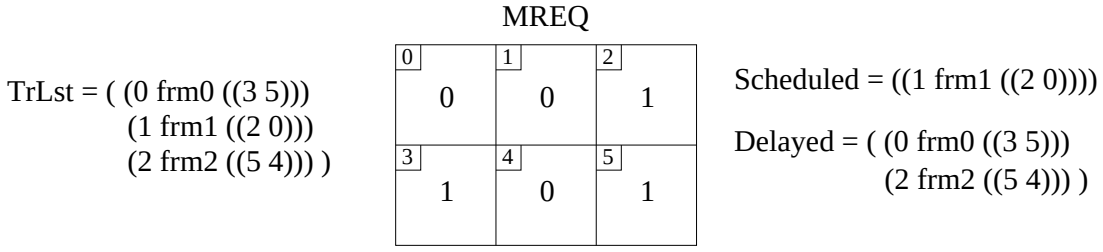


FIG. 5.10 – Exemple de création de la matrice MREQ

La fonction $Arbiter_{ahb}$ fournit le prochain maître. Le voyage ordonnancé est le premier voyage de $TrLst$ tel que sa première route débute par le prochain maître (*c.f.* Figure 5.10). La fonction $GetGranted$ construit les listes $Scheduled$ et $Delayed$. La définition de la fonction $Scheduling_{ahb}$ est la suivante :

Définition 5.16 Arbitrage du bus.
$Scheduling_{ahb}(TrLst, NodeSet, att, opts) \triangleq$
if $SumOfAttempts(att) = 0$ **then**
$List(\epsilon, TrLst, att)$
else
let $MREQ$ **be** $FillREQ(TrLst, opts[0], InitREQ(opts[0], opts[1]))$ **and**
$Granted$ **be** $Arbiter_{ahb}(MREQ, opts[0], opts[2] \bmod opts[1])$ **and**
$(Scheduled\ Delayed)$ **be** $GetGranted(TrLst, Granted, \epsilon)$ **in**
$List(Scheduled, Delayed, ConsumeAttempts(att))$
endif

Comme le montre le tableau 5.3, la définition et la preuve de la conformité de $Scheduling_{ahb}$ nécessitent 9 fonctions et 25 théorèmes. La preuve de tous ces théorèmes est automatique. Le tableau 5.3 récapitule le nombre de fonctions, de théorèmes, le temps de preuve et le nombre de lignes des différents fichiers concernant l'arbitre de bus.

	Nbre de fonctions	Nbre de théorèmes	Temps de preuve (en secondes)	Taille
Utilitaires	12	22	~ 1	100 lignes
$Arbiter_{ahb}$	6	47	~ 5	550 lignes
$Scheduling_{ahb}$	9	25	~ 10	330 lignes
Total	27	94	~ 15	980 lignes

TAB. 5.3 – Fonctions, théorèmes et temps de preuve pour la définition et la validation du bus AHB

5.3 Réalisation des interfaces

Dans le formalisme présenté au chapitre 3, les interfaces sont représentées par deux fonctions : une fonction injectant des trames dans le réseau et une fonction permettant de les lire. La principale obligation de preuve à relever est de prouver que la composition de la lecture avec l'écriture est une identité. Dans cette section, nous montrons une concrétisation des fonctions des interfaces : un protocole à marquage de phase au niveau 1 des couches OSI.

5.3.1 Le protocole Bi-Φ-M

Le protocole Bi-Φ-M a été étudié par J Strother Moore [Moo93]. Dans son article, Moore présente le protocole et la modélisation des phénomènes inhérants aux communications entre deux unités ayant deux horloges différentes. Dans cette section, nous reprenons uniquement le protocole. Les définitions et la preuve sont fortement inspirées de l'article de J Strother Moore. La figure 5.11 est issue de l'article de Moore.

Le protocole Bi-ϕ-M permet à deux unités ayant des horloges différentes, d'échanger des signaux à travers une "ligne". Ce protocole fonctionne de la manière suivante (*c.f.* Figure 5.11). Chaque bit est codé dans une cellule divisée en deux sous-cellules : une cellule de marquage et une cellule de codage. La cellule de marquage permet de synchroniser les unités réceptrice et émettrice. La cellule de codage permet de décoder le bit émis.

La valeur de la cellule de marquage est l'inverse de la valeur précédente de la ligne. Ainsi, une marque (soit un "front" d'horloge) sur lequel le récepteur peut se synchroniser est créée. La valeur de la cellule de codage dépend de celle de la cellule de marquage. Si le bit à coder est "t", la cellule de codage prend la valeur opposée à celle de la cellule de marquage. Sinon, les valeurs des deux cellules sont identiques, codant ainsi un "f".

Le principe de la transmission est qu'une fois que la marque est détectée par le récepteur, celui-ci attend un nombre de cycles donné (*distance de lecture*) pour lire le signal présent sur la ligne. La distance de lecture est définie de sorte que la lecture se fasse dans la cellule de codage. La valeur du bit initialement émis est le "ou exclusif" entre le signal lu et la valeur de la marque.

Dans les sections suivantes, nous définissons un émetteur et un récepteur. Nous prouvons que leur composition est une identité. Nous adoptons les notations suivantes. La longueur de la cellule de marquage est notée n, celle de la cellule de codage k. Généralement, une convention fixe la valeur de la ligne lorsqu'aucun message n'est émis. Nous notons flg cette

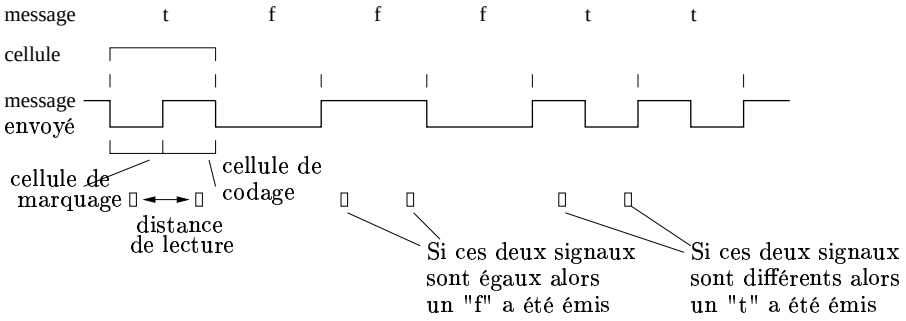


FIG. 5.11 – Codage par Bi-ϕ-M

valeur. La distance de lecture est notée *rclock*. Nous notons $Listn(n, e)$ la fonction réalisant une liste composée de n copies de l'élément e.

5.3.2 Modélisation de la partie émettrice

Le concept fondamental du protocole est celui de cellule. Une cellule est une liste de $n + k$ signaux codant un bit b donné. Le codage de ce bit dépend du signal s précédent la cellule. Soit $csig(s, b)$ la fonction réalisant le codage de b. Elle retourne s si b est "t". Sinon elle retourne $\neg s$. Une cellule est donc définie comme une liste de n copies de $\neg s$ (la marque), suivie d'une liste de k copies de $csig(s, b)$ (le code).

Définition 5.17 Codage d'un bit, fonction *cell*.
$cell(s, n, k, b) \triangleq Listn(n, \neg s) \sqcup Listn(k, csig(s, b))$

Le codage d'une liste de bits, msg, est réalisée par la concaténation des cellules codant chacun des bits. Le signal précédent la transmission est la valeur flg.

Définition 5.18 Codage d'une liste de bits, fonction *cells*.
$cells(flg, n, k, msg) \triangleq$
if $msg = \epsilon$ **then** ϵ
else $msg = b.msg'$
$cell(flg, n, k, b) \sqcup cells(csig(flg, b), n, k, msg')$
endif

Une liste de p signaux de valeur flg est initialement envoyée pour être conforme à la convention. L'unité émettrice est représentée par la fonction $send_\Phi$ ci-dessous :

Définition 5.19 Unité émettrice, fonction $send_\Phi$.
$send_\Phi(msg, p, n, k, flg) \triangleq Listn(p, flg) \sqcup cells(flg, n, k, msg)$

Par exemple, le message *(f t t f)* est envoyée par l'appel $send_\Phi$*((f t t f), 2,1,3,f)*. Le résultat est donné sur la figure 5.12.

send((f t t f), 2, 1 ,3 , f)

= (f f | t t t t | f | t t t | f | t t t | f f f f)

FIG. 5.12 – Envoi du message *(f t t f)* avec des cellules de taille $1+3$

5.3.3 Modélisation de la partie réceptrice

La fonction *scan* prend un signal s et une liste de signaux lst. Elle parcours lst jusqu'à rencontrer un signal différent de s. La fonction *scan* modélise la synchronisation sur la marque.

Définition 5.20 Détection de la marque, fonction $scan$**.**
$scan(s, lst) \triangleq$)
if $lst = \epsilon$ **then** ϵ
else $lst = e.lst'$
  if $xor(s, e)$ **then** lst
  else $scan(s, lst')$
  endif
endif

Par exemple, *scan(f,(f f t t t f))* retourne *(t t t f)*.

La fonction *recvbit* décode le bit d'une cellule. Cette fonction prend deux arguments. Le premier est la distance de lecture *rclock* à laquelle le signal de la cellule de codage doit être lu. Le second est une liste de signaux commençant par le premier signal de la cellule de marquage. Le bit reçu est t si le premier élément de lst est opposé à celui lu dans la cellule de codage. Sinon, le bit reçu est f.

Définition 5.21 Décodage d'une cellule, fonction $recvbit$**.**
$recvbit(rclock, lst) \triangleq$
if $xor(lst[0], lst[rlock])$ **then** t
else f
endif

La partie réceptrice est modélisée par la fonction $recv_\Phi$ définie à partir des fonctions ci-dessus. Elle prend pour arguments, le nombre, i, de bits à décoder, la valeur, flg, du signal émis juste avant la transmission, la distance de lecture, $rlock$, et la liste, lst, de signaux à analyser. Si i est 0, la fonction $recv_\Phi$ retourne le message vide. Sinon, le premier bit est décodé par la fonction *recvbit* et concaténé au décodage récursif des bits restants de lst.

Définition 5.22 Partie réceptrice, fonction $recv_\Phi$**.**
$recv_\Phi(i, flg, rclock, lst) \triangleq$
if $i = 0$ **then** ϵ
else let Sc **be** $scan(flg, lst)$ **in**
  $recvbit(rclock, Sc).recv_\Phi(i-1, Sc[rclock], rclock, Lastn(rclock, Sc))$
endif

Soit *lst* la liste produite par l'appel de la fonction $send_\Phi$ de la figure 5.12. Alors, $recv_\Phi(4, f, 2, lst)$ retourne le message original *(f t t f)*.

5.3.4 Preuve de la composition de $send_\Phi$ et $recv_\Phi$

Pour prouver la conformité des fonctions $send_\Phi$ et $recv_\Phi$ avec les fonctions génériques des interfaces, il faut prouver que la composition de $send_\Phi$ et $recv_\Phi$ est une identité. Cette preuve fait intervenir une série de lemmes exprimant les propriétés essentiels du protocole.

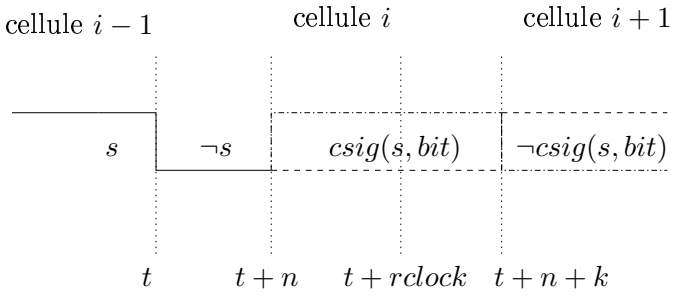


FIG. 5.13 – Structure des cellules

La figure 5.13 montre une cellule ainsi que les cellules suivantes et précédentes. La marque de la cellule i est la négation du dernier signal de la cellule $i-1$. Le bit est codé par la fonction *csig*. La cellule suivante commence par la négation de ce codage. Ce diagramme montre aussi les hypothèses sur les différents paramètres. Pour permettre la synchronisation et le décodage, la distance des cellules de marquage et de codage ne peuvent être nulles. La distance de lecture doit permettre de lire dans la cellule de codage. Pour des raisons de présentation, l'hypothèses suivante n'est pas explicitement mentionnée dans les lemmes et théorèmes ci-après.

$$\forall k, n, rclock, \in \mathbb{N}, 0 < n \wedge 0 < k \wedge n \leq rclock < n + k$$

Tout d'abord, nous lions les fonctions *scan* et *cells*. La première cellule construite par la fonction *cells* débute par la négation du dernier signal émis, s. Autrement dit, le ou exclusif entre s et le premier signal de la première cellule est toujours vrai. Le premier lemme montre que la fonction *scan* permet bien de synchroniser sur le début d'une cellule.

Lemme 5.1 Correction de *scan* avec *cells*.

$$scan(s, cells(s, n, k, msg)) = cells(s, n, k, msg)$$

Une autre propriété importante de la fonction *scan* est que son résultat ne dépend pas du nombre de signaux s. L'application de *scan* à la concaténation d'une série de signaux s et d'une liste, *lst*, est égal à l'application de *scan* à *lst*.

Lemme 5.2 Correction de *scan*.

$$scan(s, Listn(n, s) \sqcup lst) = scan(s, lst)$$

Nous montrons maintenant deux propriétés importantes de la fonction $cells$. La première concerne la lecture du signal de la cellule de codage. Si la distance de lecture est correctement ajustée, le premier signal lu est le codage du premier bit émis.

Lemme 5.3 Lecture dans la cellule de codage.

$$cells(s, n, k, msg)[rclock] = csig(s, msg[0])$$

Supposons que le message, msg, a codé contient au moins un bit. Le message s'écrit comme la concaténation d'un bit, b, et du reste du message msg'. Notons cs le résultat du codage de b par rapport au signal s, $cs = csig(s, b)$. La séquence de signaux suivants l'instant de lecture est la concaténation d'une série de cs et du codage du reste du message.

Lemme 5.4 Structure des cellules.

$$Lastn(rclock, cells(s, n, k, msg)) = Listn(n + k - rclock, cs) \sqcup cells(cs, n, k, msg')$$

Nous montrons maintenant que la composition de $recv_\Phi$ et $send_\Phi$ est une identité.

Théorème 5.18 Composition de $recv_\Phi$ et $send_\Phi$.

$$recv_\Phi(Len(msg), flg, rclock, send_\Phi(msg, p, n, k, flg)) = msg$$

Preuve: La preuve est faite par induction. Nous considérons deux cas de bases. Le premier suppose que le message initiale est vide. Dans ce cas, sa longueur est nulle. Par définition, $recv_\Phi$ retourne le message vide. Le second suppose que le message ne contient qu'un seul bit. La preuve de ce cas est relativement simple. Nous détaillons plutôt l'étape d'induction. Cette étape suppose la négation des deux cas précédents, et le message contient donc au moins deux bits. Il peut s'écrire comme la concaténation d'un bit et du reste du message, $msg = b.msg'$. L'hypothèse d'induction est obtenue en substituant, dans la formule initiale, le message msg par le reste msg' et l'argument flg par le bit codé $csig(flg, b)$. On obtient l'implication suivante :

$$\begin{array}{l} recv_\Phi(Len(msg'), csig(flg, b), rclock, send_\Phi(msg', p, n, k, csig(flg, b)) = msg' \\ \rightarrow \\ recv_\Phi(Len(msg), flg, rclock, send_\Phi(msg, p, n, k, flg) = msg \end{array}$$

En utilisant la définition de $send_\Phi$, et en notant $csig(flg, b)$ par cs, cette formule se réecrite comme suit :

$$\begin{array}{l} recv_\Phi(Len(msg'), cs, rclock, Listn(p, cs).cells(cs, n, k, msg') = msg' \\ \rightarrow \\ recv_\Phi(Len(msg), flg, rclock, Listn(p, flg) \sqcup cells(flg, n, k, msg) = msg \end{array}$$

En utilisant le lemme 5.2, on peut facilement montrer que la lecture d'une liste de signaux commençant par un signal s donné, ne dépend pas du nombre de copies de ce signal précédent la cellule de marquage. On peut donc "effacer" les appels à $Listn$ dans les arguments de la

fonction $recv_\Phi$. Nous notons $cells(flg, n, k, msg)$ par β. En utilisant la définition de $recv_\Phi$ et le lemme 5.1, on obtient :

$$\begin{array}{l} recv_\Phi(Len(msg'), cs, rclock, cells(cs, n, k, msg') = msg' \\ \rightarrow \\ recvbit(flg, \beta) \\ .recv_\Phi(Len(msg'), \beta[rclock], rclock, Lastn(n + k - rclock, \beta) = msg \end{array}$$

Les lemmes 5.3 et 5.4 permettent de réecrire la deuxième partie de la conclusion vers l'hypothése d'induction. □

Nous avons modélisé un protocole de niveau 1 dans le système OSI. La modélisation de protocoles à d'autres niveaux est similaire. Par exemple, l'encapsulation de trames selon le protocole Ethernet peut être représentée par une fonction $send_{eth}$ et la réception de ces trames par une fonction $recv_{eth}$. La validation de ces fonctions consiste dans la preuve que leur composition est une identité, soit la formule suivante :

$$recv_{eth} \circ send_{eth}(msg) = msg$$

Il est possible de combiner ces deux protocoles. L'envoi d'un message est le résultat de la composition $send_\Phi \circ send_{eth}$. La réception d'une trame est le résultat de la composition $recv_{eth} \circ recv_\Phi$. Ces deux compositions produisent des fonctions respectant la contrainte définie sur les fonctions des interfaces. En simplifiant un peu, on a bien :

$$recv_{eth} \circ recv_\Phi \circ send_\Phi \circ send_{eth}(msg) = msg$$

Nous avons modélisé les fonctions $send_{eth}$ et $recv_{eth}$ à partir du standard IEEE [ref]. Nos fonctions font abstraction des aspects temporisations mais comportent la détection des collisions. Elles sont capables de produire et de lire des trames qui pourraient circuler sur un réseau Ethernet *réel*. La preuve de leur composition est encore inachevée. Elle fait l'objet d'un sujet de stage proposé aux étudiants du magistère première année d'informatique.

5.4 Conclusion

Les illustrations présentées dans ce chapitre sont résumées sur la figure 5.14. Ces réalisations sont variées montrant ainsi l'adéquation entre le modèle générique et la réalité. Toute combinaison de ces différentes concrétisations est définie et validée par la fonction générique $GeNoC$, soit sans le moindre travail supplémentaire.

Le tableau 5.4 récapitule des données chiffrées sur les différentes applications. La machine utilisée est un Pentium IV à 1.6 GHz, avec 256MB de mémoire et fonctionnant sous Linux. Le tableau montre le nombre de fonctions, de théorèmes, le temps de preuve et la taille des fichiers.

Les temps de preuve sont peu significatifs. L'algorithme de routage de l'Octagon fait appel à de nombreux raisonnements arithmétiques peu favorables à ACL2. Néanmoins, le temps n'est qu'un peu plus de 10 minutes. Concernant les approches basées sur la démonstration de théorèmes, le temps important est le temps humain consacré au développement des preuves.

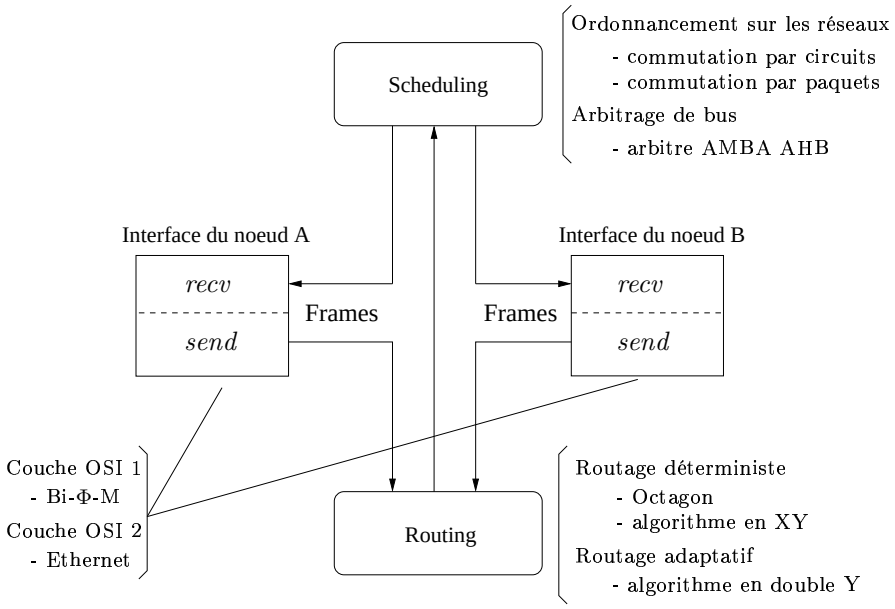


FIG. 5.14 – Concrétisations de *GeNoC*

	Nbre de fonctions	Nbre de théorèmes	Temps de preuve (en secondes)	Taille
$NodeSet_{Oct}$	5	4	< 1	70 lignes
$Routing_{Oct}$	21	64	< 740	1325 lignes
$NodeSet_{2D}$	8	5	~ 2	115 lignes
$Routing_{xy}$	6	44	~ 20	500 lignes
$Routing_{dy}$	14	49	~ 30	768 lignes
$Scheduling_{ct}$	11	26	~ 10	310 lignes
$Scheduling_{pt}$	13	24	~ 9	430 lignes
$Scheduling_{ahb}$	27	94	~ 15	980 lignes
Ethernet	57	26	~ 15	1300 lignes
BiPhiM	13	24	~ 6	330 lignes
Total	171	360	~ 850	6118 lignes

TAB. 5.4 – Fonctions, théorèmes et temps de preuve pour les concrétisations de *GeNoC*

Ce temps est difficile à évaluer et fortement dépendant du niveau d'expertise de l'utilisateur tant au niveau du système à valider que de l'outil de preuve utilisé.

La preuve de l'algorithme de routage de l'Octagon et d'un système plus global a été développée en trois mois avant l'existance de *GeNoC* et sans une connaissance préalable de ce réseau [5]. Aujourd'hui, nous estimons le temps de modélisation et de validation de cet algorithme à au plus deux semaines.

De plus certaines illustrations se construisent sur d'autres. La définition et la validation de la commutation par paquets reposent sur les fonctions et les lemmes développés pour la commutation par circuits. De même, le développement du modèle et de la preuve du routage en double Y est construit à partir de la définitions du routage en XY.

En conclusion, ces études de cas nécessitent des fichiers d'une taille relativement faible, tant sur le nombre de fonctions que de théorèmes. La modélisation et la validation, aux premiers pas d'une conception, devant être rapides, *GeNoC* fournit un cadre correspondant à ces besoins.

[5] La construction du modèle a été effectuée à partir des descriptions partielles fournies par la littérature [KNDR01, KND02]. Après discussions avec des concepteurs de *STMicroelectronics* le modèles ACL2 s'est avéré proche du leur.

Troisième partie

Conclusions

Conclusion et perspectives

Conclusion

Nous avons formalisé le concept d'architecture de communication. Celui-ci s'articule autour d'interfaces, d'un algorithme de routage et d'une politique d'ordonnancement. Les interfaces permettent de coder et décoder l'information transmise au travers de l'architecture ; l'algorithme de routage crée un chemin dans celle-ci reliant deux acteurs d'une communication ; la politique d'ordonnancement élit les communications qui peuvent être simultanées et retarde les autres.

Chacun de ces éléments doit possèder une série de propriétés essentielles pour garantir un fonctionnement correct de l'ensemble. Concernant les interfaces, la partie réceptrice doit être capable de *décoder l'information codée* par la partie émettrice. Tout chemin proposé par l'algorithme de routage doit relier les acteurs *attendus* d'une communication sans louvoyer indéfiniment dans l'architecture. Les communications simultanées élues par la politique d'ordonnancement doivent être *mutuellement exclusives* des communications retardées. L'interaction de ces différents éléments forme une architecture garantissant que tout message en attente est, soit jamais envoyé, soit atteint sa destination sans modification de son contenu.

La fonction *GeNoC* modélise une architecture de communication générique. Elle appelle les fonctions des différents éléments. Les propriétés essentielles d'un élément sont représentées par une série d'obligations de preuve attachées à la fonction - ou aux fonctions dans le cas des interfaces - les représentant. Ces obligations de preuve impliquent le théorème principal sur *GeNoC* qui montre que pour toute communication non avortée *la bonne destination* reçoit *l'information attendue* de *la bonne source*. La spécification d'une architecture de communication particulière revient à donner une définition *explicite* à chacun des éléments et à prouver que ces définitions satisfont les propriétés essentielles. Chaque élément se suffit à lui même. On peut valider une particularisation d'un élément sans nécessairement donner une définition explicite aux autres. Le cadre défini par *GeNoC* est modulaire tant sur le plan de la conception que de la validation.

En pratique, ce modéle a été mis en œuvre dans la logique du démonstrateur de théorèmes ACL2. Le formalisme repose sur de nombreux quantificateurs et des constructions du second ordre. La logique ACL2 est du premier ordre et sans quantificateur, la réalisation de *GeNoC* n'y est pas triviale. Nous avons montré comment utiliser le principe d'encapsulation d'ACL2 et la règle de *functional instantiation* pour simuler une partie de la logique du second ordre et comment remplacer les quantificateurs par des fonctions récursives. Notre formalisation n'est pas intrinsèquement liée au démonstrateur ACL2, elle pourrait être réalisée dans d'autres

systèmes, comme HOL, PVS ou Isabelle, par exemple.

Finalement, nous avons exhibé différentes architectures constituant autant de concrétisations de notre concept. Ces concrétisations comprennent des systèmes industriels, comme le bus AMBA ou le réseau Octagon, et des exemples plus académiques, comme le routage en XY ou en double Y dans une grille, les techniques de commutation par paquets et par circuits ou le protocole à marquage de phase Bi-ϕ-M.

Notre principale contribution est d'avoir identifié les composantes essentielles communes à *toute* structure d'interconnexion ainsi que les propriétés inhérentes à chacune d'entre elles. En décrivant ce concept dans la logique d'un assistant "automatique" de preuve nous fournissons un outil pour spécifier et valider les descriptions des réseaux sur puce à un haut niveau d'abstraction.

En dépit du très haut niveau d'abstraction du modèle, le problème traité dans cette thèse et la solution proposée sont réels. Aujourd'hui, la complexité des systèmes impose l'utilisation de démonstrateurs de théorèmes pour garantir la correction des conceptions. Les travaux utilisant la démonstration de théorème et menés dans l'industrie - comme à Rockwell&Collins ou Infineon - et les recherches universitaires ont vérifié des systèmes hors de portée des autres techniques de vérification formelle. Le prix de cette technique est une forte intervention humaine. La spécification formelle dans la logique d'un démonstrateur est délicate. La structure d'un modèle et sa définition ont des conséquences importantes sur les preuves futures. En formalisant le concept d'architecture de communication nous identifions une structure. La preuve de la correction de celle-ci étant déduite des contraintes associées aux fonctions de base, la validation formelle de *toute* concrétisation est réduite à la preuve de ces contraintes.

Discussion et perspectives

Applications esclaves. Pour rendre compte des communications de type maître/esclave, la formalisation des applications passives (ou esclaves) est nécessaire. Le principe d'un esclave est de produire une *réponse* à la *requête* d'un maître. Dans *GeNoC*, la requête et la réponse sont représentées simplement par des messages. La distinction entre une requête et une réponse permet d'identifier le sens de la communication. Elle n'a aucune influence sur la modélisation initiale. Il est possible de formaliser un esclave par une fonction qui manipule son état. À chaque réception d'un message, cette fonction calcule une réponse et un nouvel état. L'ensemble des applications passives est modélisé par une fonction qui répond à une requête en renvoyant un nouveau message et une nouvelle valeur de la variable d'état. L'échange entre un maître et un esclave serait modélisé en deux temps : un premier pour l'envoi des requêtes, un second pour mettre à jour la variable d'état et renvoyer les réponses.

En pratique, les protocoles maître/esclaves fonctionnement en deux modes. Dans le premier mode la réponse emprunte la même route que la requête ; dans le second la réponse est acheminée de manière indépendante.

Au début des travaux de thèse, nous avons modélisé des architectures de type maître-/esclave où les esclaves sont des mémoires : un système autour d'un bus AMBA [SB03a, SB03c] et un système autour de l'Octagon [SB04a, Sch04]. Ces résultats n'ont pas été intégrés

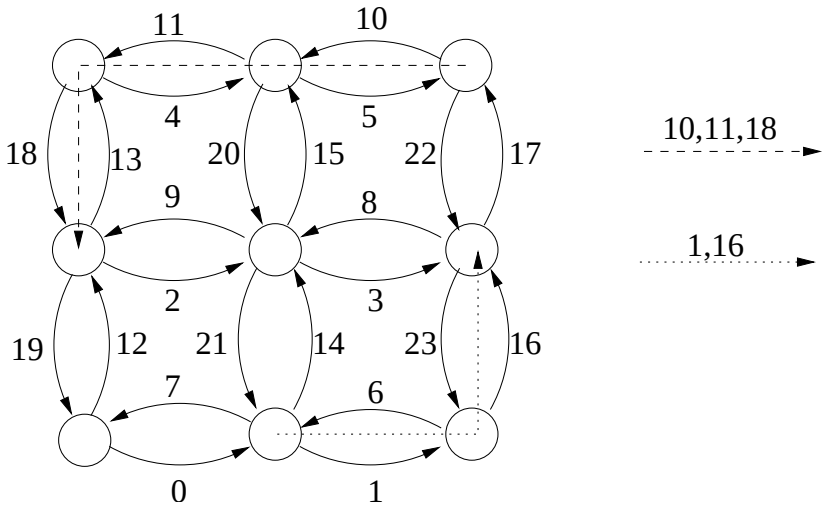


FIG. 5.15 – Énumération des canaux dans une grille 3×3 et l'algorithme en XY.

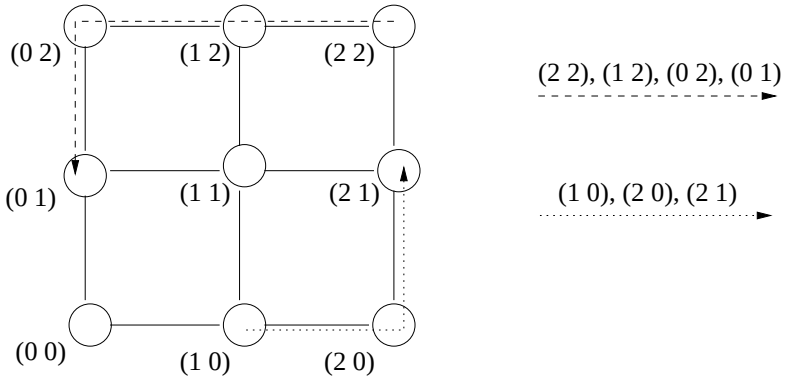


FIG. 5.16 – Dans *GeNoC*, la relation d'ordre est sur les nœuds.

dans le modèle *GeNoC* présenté dans ce mémoire et pourraient servir de point de départ pour une extension de *GeNoC* explicitant les protocoles maître/esclave.

Interblocages et *GeNoC*. Comme nous l'avons vu au chapitre 2 (page 43), une condition nécessaire et suffisante pour garantir l'absence d'interblocage est la preuve de l'absence de cycle dans le graphe de dépendance des canaux. De manière pratique, la preuve de l'absence de cycle est effectuée en exhibant une numérotation des canaux telle que ceux-ci apparaissent de manière ordonnée - ordre croissant ou décroissant - pour toute route du réseau. Considérons une grille 3×3 sur laquelle est appliqué l'algorithme en XY. Considérons aussi que chaque nœud est relié dans chaque dimension par deux canaux. La figure 5.15 montre une numérotation des canaux telle que ceux-ci apparaissent toujours dans l'ordre croissant. Cette figure montre deux routes et les numéros des canaux empruntés.

Dans *GeNoC*, nous considérons un réseau abstrait dans lequel les nœuds sont explicitement représentés et numérotés, non les canaux. Il est possible d'exhiber une numérotation des nœuds telle que les nœuds apparaissent de manière ordonnée dans les routes. Ainsi, sur

l'exemple de la grille 3×3 précédent, une numérotation des nœuds est donnée sur la figure 5.16. Dans toute route calculée selon l'algorithme en XY, l'axe X est parcouru avant l'axe Y. Autrement dit, chaque route est constituée d'une série de nœuds qui ont la même ordonnée puis d'une série de nœuds qui ont la même abscisse. En fonction du sens de parcours de l'axe X, les abscisses croissent ou décroissent. Il n'est pas possible de trouver une numérotation et *une* relation d'ordre. Dans le cas du routage en XY - et aussi en YX et double Y - il y a deux relations d'ordre. Le problème des interblocages doit être approfondi pour déterminer une condition nécessaire et suffisante, basée sur les nœuds, garantissant l'absence d'interblocage dans *GeNoC*.

Algorithmes non-minimaux. Dans ce manuscrit, nous avons considéré uniquement les algorithmes minimaux. Ces algorithmes sont représentés par des fonctions définies comme l'application récursive de déplacements unitaires. La mesure utilisée pour prouver leur terminaison est la plupart du temps la distance entre la localisation d'une trame et sa destination. Puisque l'algorithme est minimal, chaque déplacement unitaire réduit cette distance. La terminaison des algorithmes non-minimaux est assurée en bornant l'éloignement des chemins non-minimaux par rapport au chemin minimal. Une définition récursive de ces algorithmes est possible. Le problème particulier à *GeNoC* est lié à la régularisation des topologies. En généralisant un algorithme non-minimal à tous les nœuds d'un réseau irrégulier, il devient impossible de garantir que les chemins du réseau "régularisé" ne passent que par des nœuds appartenant au réseau irrégulier initial. Cet aspect ne semble pas critique car la généralisation propose plus de routes qu'il en existe réellement. Des études complémentaires sont nécessaires mais l'extension de *GeNoC* aux algorithmes de routage non-minimaux semble possible.

Applications "maîtres". Rendre compte des applications "maîtres" semble possible. Tout comme pour les applications esclaves, nous pouvons associer des applications actives aux nœuds. Cette étude se rapproche des recherches menées au niveau logiciel sur la vérification d'applications parallèles. L'étude des interactions possibles entre *GeNoC* et ce domaine de recherche constitue un prolongement possible des travaux présentés dans ce manuscrit.

Notion explicite du temps. Dans *GeNoC*, le temps est représenté de manière *implicite*. La fonction d'ordonnancement de *GeNoC* sépare les trames en deux parties ; celles qui sont envoyées *maintenant*, et celles qui sont envoyées *plus tard* et qui sont possiblement avortées. Une extension possible de *GeNoC* serait d'ajouter une variable représentant le temps de manière *explicite*. Une fonctionnalité suppémentaire de la fonction d'ordonnancement serait de mettre à jour cette variable, "de faire avancer le temps". Cette représentation explicite du temps a déjà été utilisée pour la spécification et la vérification d'algorithmes "temps réels" [AL94, Lam05]. L'étude d'une version "temporisée" de *GeNoC* pourrait s'appuyer sur ces travaux.

Raffinements de *GeNoC*. Dans *GeNoC*, seuls les nœuds sont considérés. Le nombre de canaux ou de files reliant les nœuds n'est pas précisé. Il serait sûrement intéressant

d'étendre *GeNoC* de façon à considérer les files ou les canaux de manière explicite. On pourrait considérer cette modification de *GeNoC* comme une opération de raffinement. Il serait intéressant d'étudier les propriétés que cette opération doit satisfaire pour préserver les théorèmes établis au niveau de *GeNoC*. Ce raffinement pourrait être mis en relation avec la preuve d'absence d'interblocage au niveau des nœuds et déterminer les conditions à satisfaire pour préserver cette propriétés après ajout de files et de canaux.

À terme, une ou plusieurs étapes de raffinement devraient conduire à un modèle suffisamment détaillé pour qu'un lien direct avec le niveau RTL puisse être établi. La modélisation du routage comme l'application successive de déplacements unitaires reflète les passages successifs au travers de la logique de routage de chaque nœud. La modélisation de la politique d'ordonnancement sous entend la préservation d'un invariant par les mécanismes mis en œuvre pour contrôler les flux sur le réseau. Ces deux aspects constituent deux points de départ potentiels pour l'étude des liens entre *GeNoC* et des réseaux "physiques", prototypés sur FPGA par exemple. À la date de parution de ce manuscrit, un étudiant de M2R entrepend les études prélimaires à ce travail. Des travaux connexes visant à extraire automatiquement des descriptions RTL de réseaux sur puce vers la logique de démonstrateurs ont été menés en collaboration avec de précédents travaux de thèse [SSB04, SSB05]. Ces travaux sont aussi à considérer pour faire le lien entre le RTL et *GeNoC*.

Généralisation du routage. Dans ce manuscrit, la fonction de routage est définie comme l'application récursive de déplacements unitaires. Une autre approche du routage est de considérer que l'ensemble des routes possibles entre deux nœuds est l'ensemble des combinaisons possibles des différents déplacements requis pour parcourir la distance qui sépare ces deux nœuds. Si on associe un entier à chaque déplacement, il est ensuite facile de convertir toute combinaison d'entiers en une route. On applique à partir du nœud source le déplacement correspondant à chaque élément d'une combinaison. L'idée est de développer la preuve des contraintes associées au routage autour de cette définition. On obtiendrait ainsi une méthode très systèmatique pour vérifier le routage. Si on abstrait un algorithme de routage à ce problème de combinatoire, on va produire des routes qui ne sont pas autorisées initialement. Il faut ajouter une fonction qui restreint les combinaisons possibles. Un challenge intéressant serait de déduire cette fonction du code RTL décrivant la logique de routage d'un routeur.

Quatrième partie

Annexes

Annexe A

Termes, objets et fonctions

A.1 Listes

Nom	Rôle
$e.l$	ajout de l'élément e à la liste l
$l_1 \subseteq_l E$	typage des éléments de l_1 sur l'ensemble E
$l_1 \sqcup l_2$	concaténation de l_1 et l_2
$e \in_l l_1$	e est un élément de la liste l_1
$l_1 \sqsubseteq l_2$	inclusion de l_1 dans l_2
$l_1 \sqcap l_2$	extraction de l_1 par rapport à l_2
$List(l_1, l_2)$	juxtaposition des listes l_1 et l_2
$Len(l)$	retourne le nombre d'éléments contenus dans l
$Last(l)$	retourne le dernier élément de l
$NoDuplicatesp(l)$	identifie une liste l sans doublon
ϵ	liste vide
$l[i]$	élément de la liste l, $0 \leq i \leq len(l) - 1$

TAB. A.1 – Notations et fonctions pour manipuler les listes

Nom	Objet
transaction	quadruplet $t = (Id\ Org\ Msg\ Dest)$
$\mathcal{D}_{msg}$	domaine de définition d'un message
$\mathcal{D}_{\mathcal{T}}$	domaine de définition de $\mathcal{T}$
$\mathcal{T}_{lstp}(\mathcal{T}, NodeSet)$	identifie une liste valide de transactions
$\mathcal{T}$	liste de transactions
$Id_{\mathcal{T}}(t)$, $Org_{\mathcal{T}}(t)$, $Msg_{\mathcal{T}}(t)$, $Dest_{\mathcal{T}}(t)$	accesseurs des éléments d'une transaction t
$\mathcal{T}_{\lvert id}$	liste des identifiants de $\mathcal{T}$
$NodeSet$	ensemble des nœuds du réseau

TAB. A.2 – Fonctions et objets concernant les applications actives

A.2 Transactions

A.3 Missives

A.4 Voyages

A.5 Tentatives

A.6 Résultats

A.7 Nœuds et paramètres

A.8 Interfaces

A.9 Routage

Nom	Objet
missive	$m = (Id\ Org\ Frm\ Dest)$
$\mathcal{D}_{frm}$	domaine de définition d'une trame
$\mathcal{M}_{lstp}(\mathcal{M}, NodeSet)$	identifie une liste valide de missives
$\mathcal{D}_{\mathcal{M}}$	domaine de définition de $\mathcal{M}$
$\mathcal{M}$	liste de missives
$Id_{\mathcal{M}}(m)$, $Org_{\mathcal{M}}(m)$, $Frm_{\mathcal{M}}(m)$, $Dest_{\mathcal{M}}(m)$	accesseurs des éléments de m

TAB. A.3 – Fonctions et objets concernant les missives

Nom	Objet
voyage	triplet $v = (Id\ Frm\ Routes)$
$\mathcal{V}$	liste de voyages
$\mathcal{D}_{\mathcal{V}}$	domaine de définition de $\mathcal{V}$
$\mathcal{C}$	domaine de définition d'une liste de routes
$\mathcal{V}_{lstp}(\mathcal{V})$	reconnaît une liste valide de voyages
$Id_{\mathcal{V}}(v)$, $Frm_{\mathcal{V}}(v)$,$Routes_{\mathcal{V}}(v)$	accesseurs des éléments de v

TAB. A.4 – Fonctions et objets concernant les voyages

Nom	Objet
att	liste des tentatives de tous les nœuds
$MapAtt(n)$	donne l'indice i dans att des tentatives pour le nœud n
$AttLst$	domaine de définition de att
$SumOfAttempts(att)$	additionne toutes les tentatives restantes de tous les nœuds
$\mathcal{A}$	liste de transactions avortées

TAB. A.5 – Fonctions et objets concernant les tentatives

Nom	Objet
résultat	triplet $rst = (Id\ Dest\ Msg)$
$\mathcal{R}$	liste de résultats
$\mathcal{D}_{\mathcal{R}}$	domaine de définition de $\mathcal{R}$
$\mathcal{R}_{lstp}(\mathcal{R}, NodeSet)$	reconnaît une liste valide de résultats
$Id_{\mathcal{R}}(rst)$, $Dest_{\mathcal{R}}(rst)$,$Msg_{\mathcal{R}}(rst)$	accesseurs des éléments de rst

TAB. A.6 – Fonctions et objets concernant les résultats

Nom	Objet
$ValidNodep(x)$	identifie x comme faisant partie des nœuds du réseau
$NodeSetGen(pms)$	construit l'ensemble des nœuds d'un résau
$GenNodeSet$	domaine de définition des nœuds d'un réseau
$NodeSet$	ensemble des nœuds d'un réseau particulier
$ValidParamsp(pms)$	identifie pms comme paramètres d'un réseau
$GenParams$	domaine de définition de pms

TAB. A.7 – Fonctions et objets concernant les nœuds et les paramètres

Nom	Objet
message	(noté msg) information entre une application et une interface
$\mathcal{D}_{msg}$	domaine de définition de $message$
frame	(ou $trame$ ou simplement frm) information entre les interfaces
$\mathcal{D}_{frm}$	domaine de définition de frm
$send$	construit une trame à partir d'un message
$recv$	construit un message à partir d'une trame

TAB. A.8 – Fonctions et objets concernant les interfaces

Nom	rôle
$\mathcal{L}(i,j)$	déplacements unitaires autorisés dans le réseau
$\rho(i,j)$	construit toutes les routes de i à j
$Routing(\mathcal{M}, NodeSet)$	construit toutes les routes pour toutes les missives
$ValidRoutep(r,m,NodeSet)$	identifie une route valide par rapport à la missive m
$ToMissives(\mathcal{V})$	convertit la liste de voyages $\mathcal{V}$ en une liste de missives

TAB. A.9 – Fonctions et notations pour le routage

Annexe B

Réponse d'ACL2 au "defthm"

```
ACL2 !>(defthm treecopy-id
          (equal (treecopy x) x))


Name the formula above *1.

Perhaps we can prove *1 by induction.
One induction scheme is suggested by this conjecture.

We will induct according to a scheme suggested by (TREECOPY X).
This suggestion was produced using the :induction rule TREECOPY.
If we let  (:P X) denote *1 above then the induction scheme
we'll use is
(AND (IMPLIES (NOT (CONSP X)) (:P X))
     (IMPLIES (AND (CONSP X)
                   (:P (CAR X))
                   (:P (CDR X)))
              (:P X))).
This induction is justified by the same argument
used to admit TREECOPY, namely, the measure (ACL2-COUNT X) is
decreasing according to the relation O< (which is known to be
well-founded on the domain recognized by O-P).
When applied to the goal at hand the above induction scheme
produces the following two nontautological subgoals.

Subgoal *1/2
(IMPLIES (NOT (CONSP X))
         (EQUAL (TREECOPY X) X)).

But simplification reduces this to T, using
the :definition TREECOPY and primitive type reasoning.
```

```
Subgoal *1/1
(IMPLIES (AND (CONSP X)
              (EQUAL (TREECOPY (CAR X)) (CAR X))
              (EQUAL (TREECOPY (CDR X)) (CDR X)))
         (EQUAL (TREECOPY X) X)).

But simplification reduces this to T, using
the :definition TREECOPY, primitive type reasoning and
the :rewrite rule CAR-CDR-ELIM.

That completes the proof of *1.

Q.E.D.

Summary
Form:  ( DEFTHM TREECOPY-ID ...)
Rules: ((:DEFINITION TREECOPY)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:INDUCTION TREECOPY)
        (:REWRITE CAR-CDR-ELIM))
Warnings:  None
Time:  0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
 TREECOPY-ID
ACL2 !>
```

Annexe C

Code ACL2 pour *GeNoC*

C.1 Code pour les interfaces

```
;; Interface Module of GeNoC
(in-package "ACL2")
(encapsulate
 (;; Any peer has an interface that can send and receive messages
  ;; Function p2psend
  ;; argument: a message msg
  ;; output: a frame frm
  ((p2psend *) => *)
  ;; Function p2precv
  ;; argument: a frame frm
  ;; output: a message msg
  ((p2precv *) => *))
 (local (defun p2psend (msg) msg))
 (local (defun p2precv (frm) frm))
 (defthm p2p-Correctness
   ;; the composition of p2precv and p2psend
   ;; is the identity function
   (equal (p2precv (p2psend msg)) msg))
 (defthm p2psend-nil
   ;; if msg is nil then p2psend is nil too
  (not (p2psend nil)))
(defthm p2psend-not-nil
  ;; if msg is not nil then p2psend is not nil too
  (implies msg
           (p2psend msg)))
) ;; end of interfaces
```

C.2 Code pour les types de données

```
;; Julien Schmaltz
;; Definition of the data-types used in GeNoC:
;; Transactions, missives, travels, results and attempts
;; June 20th 2005
;; File: GeNoC-types.lisp
(in-package "ACL2")
(include-book
 "/home/julien/softs/acl2-v29/acl2-sources/books/data-structures/list-defuns")
(include-book
 "/home/julien/softs/acl2-v29/acl2-sources/books/data-structures/list-defthms")
(include-book
 "/home/julien/softs/acl2-v29/acl2-sources/books/textbook/chap11/qsort")
;;---------------------------------------------------------------------
;;
;;                          TRANSACTIONS
;;
;;---------------------------------------------------------------------
;; A transaction is a tuple t = (id A msg B)
;; Accessors are IdT, OrgT, MsgT and DestT
(defun Idt (trans) (car trans))
(defun OrgT (trans) (nth 1 trans))
(defun MsgT (trans) (nth 2 trans))
(defun DestT (trans) (nth 3 trans))
;; We need a function that grabs the ids of a list of transactions
(defun T-ids (Transts)
  (if (endp Transts)
      nil
    (append (list (caar Transts)) (T-ids (cdr Transts)))))
;; The following predicate checks that each transaction has
;; the right number of arguments
(defun validfield-transactionp (trans)
  ;; trans = (id A msg B)
  (and (consp trans)
       (consp (cdr trans))          ;; (A msg B)
       (consp (cddr trans))         ;; (msg B)
       (consp (cdddr trans))         ;; (B)
       (null (cddddr trans))))
;; The following predicate recognizes a valid list of transactions
(defun Validfields-T (Transts NodeSet)
  (if (endp Transts)
      t
    (let ((trans (car Transts)))
```

```
      (and (validfield-transactionp trans)
           (natp (Idt trans))                        ;; id is a natural
           (member (OrgT trans) NodeSet)
           (member (DestT trans) NodeSet)
           (not (equal (OrgT trans) (DestT trans))) ;; A /= B
           (Validfields-T (cdr Transts) NodeSet)))))
;; now we define the predicate that recognizes a valid list of
;; transactions
(defun Transactionsp (Transts NodeSet)
  (let ((T-ids (T-ids Transts)))
    (and (Validfields-T Transts NodeSet)
         (true-listp Transts)
         (No-Duplicatesp T-ids)
         (Orderedp T-ids))))
;;------------------ end of Transactions --------------------------------

;;-----------------------------------------------------------------------
;;
;;                          MISSIVES
;;
;;-----------------------------------------------------------------------
;; A missive is a tuple m = (id A frm B)
;; Accessors are IdM, OrgM, FrmM and DestM
(defun IdM (m) (car m))
(defun OrgM (m) (nth 1 m))
(defun FrmM (m) (nth 2 m))
(defun DestM (m) (nth 3 m))
;; We need a function that grabs the ids of a list of missives
(defun M-ids (M)
  (if (endp M)
      nil
    (append (list (caar M)) (M-ids (cdr M)))))
;; We need a function that grabs the origins of Missives
(defun M-orgs (M)
  (if (endp M)
      nil
    (append (list (OrgM (car M))) (M-orgs (cdr M)))))
;; The same for the destinations
(defun M-dests (M)
  (if (endp M)
      nil
    (append (list (DestM (car M))) (M-dests (cdr M)))))
;; We also need a function that grabs the frames of
;; a list of missives
```

```
(defun M-frms (M)
  ;; grabs the frames of M
  (if (endp M)
      nil
    (let* ((msv (car M))
           (m-frm (FrmM msv)))
      (append (list m-frm) (M-frms (cdr M))))))
;; The following predicate checks that each missive has
;; the right number of arguments
(defun validfield-missivep (m)
  ;; m = (id A frm B)
  (and (consp m)
       (consp (cdr m))         ;; (A frm B)
       (consp (cddr m))        ;; (frm B)
       (consp (cdddr m))        ;; (B)
       (null (cddddr m))))
;; The following predicate recognizes a valid list of missives (partially)
(defun Validfields-M (M NodeSet)
  (if (endp M)
      t
    (let ((msv (car M)))
      (and (validfield-missivep msv)
           (natp (IdM msv))                    ;; id is a natural
           (member (OrgM msv) NodeSet)
           (member (DestM msv) NodeSet)
           (not (equal (OrgM msv) (DestM msv))) ;; A /= B
           (Validfields-M (cdr M) NodeSet)))))
;; now we define the predicate that recognizes a valid list of
;; transactions
(defun Missivesp (M NodeSet)
  (let ((M-ids (M-ids M)))
    (and (Validfields-M M NodeSet)
         (true-listp M)
         (No-Duplicatesp M-ids)
         (Orderedp M-ids))))
;;-------------------- end of Missives -----------------------------------

;;-------------------------------------------------------------------------
;;
;;                          TRAVELS
;;
;;-------------------------------------------------------------------------
;; A travel is a tuple tr = (id frm Routes)
;; Accessors are IdV, FrmV and RoutesV
```

```
;; (JS: V comes from the french word for travel, voyage :-)
(defun IdV (tr) (car tr))
(defun FrmV (tr) (nth 1 tr))
(defun RoutesV (tr) (nth 2 tr))
;; We need a function that grabs the ids of a list of travels
(defun V-ids (TrLst)
  (if (endp TrLst)
      nil
    (append (list (caar TrLst)) (V-ids (cdr TrLst)))))
;; The following predicate checks that each route of routes
;; has at least two elements
(defun validfield-route (routes)
  ;; checks that every route has at least two elements
  (if (endp routes)
      t
    (let ((r (car routes)))
      (and (consp r)
           (consp (cdr r))
           (validfield-route (cdr routes))))))
;; The following predicate checks that each travel has
;; the right number of arguments
(defun validfield-travelp (tr)
  ;; tr = (id frm Routes)
  (and (consp tr)
       (consp (cdr tr))          ;; (frm Routes)
       (consp (cddr tr))         ;; (Routes) = ( ((R1) (R2) ...))
       (consp (caddr tr))        ;; ((R1) (R2) ...)
       (validfield-route (caddr tr))
       (null (cdddr tr))))
;; The following predicate recognizes a valid list of missives (partially)
(defun Validfields-TrLst (TrLst)
  (if (endp TrLst)
      t
    (let ((tr (car TrLst)))
      (and (validfield-travelp tr)
           (natp (IdV tr))                    ;; id is a natural
           (Validfields-TrLst (cdr TrLst))))))
;; now we define the predicate that recognizes a valid list of
;; transactions
(defun TrLstp (TrLst )
  (let ((V-ids (V-ids TrLst)))
    (and (Validfields-TrLst TrLst)
         (true-listp TrLst)
         (No-Duplicatesp V-ids)
```

```
         (Orderedp V-ids))))
(defun V-frms (TrLst)
  ;; grabs the frames of TrLst
  (if (endp TrLst)
      nil
    (let* ((tr (car Trlst))
           (v-frm (FrmV tr)))
      (append (list v-frm) (V-frms (cdr TrLst))))))
;;-------------------- end of Travels -----------------------------------
;;-------------------------------------------------------------------------
;;
;;                          RESULTS
;;
;;-------------------------------------------------------------------------
;; A result is a tuple rst = (Id Dest Msg)
;; Accessors are IdR, DestR and MsgR
(defun IdR (rst) (car rst))
(defun DestR (rst) (nth 1 rst))
(defun MsgR (rst) (nth 2 rst))
;; We need a function that grabs the ids of a list of results
(defun R-ids (R)
  (if (endp R)
      nil
    (append (list (caar R)) (R-ids (cdr R)))))
;; The following predicate checks that each result has
;; the right number of arguments
(defun validfield-resultp (rst)
  ;; tr = (Id Dest Msg)
  (and (consp rst)
       (consp (cdr rst))         ;; (Dest Msg)
       (consp (cddr rst))        ;; (Msg)
       (null (cdddr rst))))
;; The following predicate recognizes a valid list of results (partially)
(defun Validfields-R (R NodeSet)
  (if (endp R)
      t
    (let ((rst (car R)))
      (and (validfield-resultp rst)
           (natp (IdR rst))                    ;; id is a natural
           (member (DestR rst) NodeSet)
           (Validfields-R (cdr R) NodeSet)))))
;; now we define the predicate that recognizes a valid list of
;; results
(defun Resultsp (R NodeSet)
```

```
  (let ((R-ids (R-ids R)))
    (and (Validfields-R R NodeSet)
         (true-listp R)
         (No-Duplicatesp R-ids)
         (Orderedp R-ids))))
;;-------------------- end of Results -----------------------------------

;;-------------------------------------------------------------------------
;;
;;                          ATTEMPTS
;;
;;-------------------------------------------------------------------------
;; we just need to define the sum of the attempts
(defun SumOfAttempts (att)
  ;; this is the measure for sigma-t
  ;; att has the following form:
  ;; ( ... (i RemAtt) ...)
  (if (endp att)
      0
    (let* ((top (car att))
   (rem-att (cadr top)))
      (+ (nfix rem-att)
         (SumOfAttempts (cdr att))))))
;;-------------------- end of Attempts -----------------------------------
```

C.3 Code pour les noeuds

```
;; Julien Schmaltz
;; Generic Set of Nodes
;; June 17th 2005
;; File: GeNoC-nodeset.lisp
(in-package "ACL2")
(encapsulate  ;; GenericNodeSet
 ;; abstract set of nodes
 ;; the set is generated by the following function
 ;; its argument is the parameters
 (((NodesetGenerator *) => *)
  ;; the following predicate recognizes valid parameters
  ((ValidParamsp *) => *)
  ;; the following predicate recognizes a valid node
  ((NodeSetp *) => *))
 ;; local witnesses
 (local (defun ValidParamsp (x) (declare (ignore x)) t))
```

```
 (local (defun NodesetGenerator (x)
          (if (zp x) nil
            (cons x (NodesetGenerator (1- x))))))
 (local (defun NodeSetp (l)
          (if (endp l) t
            (and (natp (car l))
                 (NodeSetp (cdr l))))))
 (defthm nodeset-generates-valid-nodes
   (implies (ValidParamsp params)
            (NodeSetp (NodesetGenerator params))))
 ;; we add a generic lemma
 (defthm subsets-are-valid
   ;; this lemma is used to prove that routes are made of valid nodes
   (implies (and (NodeSetp x)
                 (subsetp y x))
            (NodeSetp y)))
) ;; end GenericNodeSet
```

C.4 Code pour le routage

```
;; Julien Schmaltz
;; Generic Routing Module of GeNoC
;; June 20th 2005
;; File: GeNoC-routing.lisp
(in-package "ACL2")
;; we import the books for the set of nodes and about the data-types
(include-book "GeNoC-nodeset")
(include-book "GeNoC-misc") ;; import also GeNoC-types
(encapsulate
 ;; Routing computes the route of each message within the network
 ;; It takes as arguments: M and NodeSet
 ;; It outputs a list of travel TrLst = (... (Id frm Route) ...)
 ;; Constraints:
 ;; 1/ If the input is a list of valid missives, the output must be
 ;;    a list of valid travels (Ids are still unique)
 ;; 2/ Every route of every travel must be correct
 ;; 3/ Frms of the output must be equal to the frms of the input
 (((Routing * *) => *))
 ;; local witnesses
 (local (defun route (M)
          ;; this would be the routing in a bus
          (if (endp M)
              nil
```

```
           (let* ((msv (car M))
                  (Id (IdM msv))
                  (frm (FrmM msv))
                  (origin (OrgM msv))
                  (destination (DestM msv)))
             (cons (list Id frm (list (list origin destination)))
                   (route (cdr M)))))))
(local (defun routing (M NodeSet)
         (declare (ignore NodeSet))
         (route M)))
;; 1/ If the input is a list of valid missives, the output must be
;;    a list of valid travels (Ids are still unique)
;; local lemmas
(local (defthm route-ids
         ;; ids of the output TrLst are equal to the ids of
         ;; the initital missives
         (equal (V-Ids (route M)) (M-Ids M))))
(local (defthm validfields-route
         (implies (Validfields-M M)
                  (validfields-TrLst (route M)))))
(local (defthm TrLstp-route
         (implies (and (Validfields-M M)
                       (No-duplicatesp (M-ids M)))
                  (TrLstp (route M)))))
(defthm TrLstp-routing
  ;; 1st constraint
  ;; the travel list is recognized by TrLst
  ;; Params is a free variable
  (let ((NodeSet (NodeSetGenerator Params)))
    (implies (and (Missivesp M NodeSet)
                  (ValidParamsp Params))
             (TrLstp (routing M NodeSet)))))
;; 2/ Routes must satisfy the predicate CorrectRoutesp
(local
 (defthm correctroutesp-route
   (implies (Missivesp M NodeSet)
            (CorrectRoutesp (route M) M NodeSet))))
(defthm Routing-CorrectRoutesp
  ;; 2nd constraint
  ;; The routes produced by routing are correct
  (let ((NodeSet (NodeSetGenerator Params)))
    (implies (and (Missivesp M NodeSet)
                  (ValidParamsp Params))
             (CorrectRoutesp (Routing M NodeSet) M NodeSet))))
```

```
 ;; some additional constraints
 (defthm true-listp-routing
   (true-listp (routing M NodeSet))
   :rule-classes :type-prescription)
 (defthm routing-nil
   ;; the routing has to return nil if the list of missives is nil
   (not (routing nil NodeSet)))
 ) ;; end of routing
(defthm tomissives-routing
  (let ((NodeSet (NodeSetGenerator Params)))
    (implies (and (Missivesp M NodeSet)
                  (ValidParamsp Params))
             (equal (ToMissives (routing M NodeSet)) M)))
  :hints (("GOAL"
           :use (:instance correctroutesp-=>-tomissives
                           (TrLst (Routing M (NodeSetGenerator Params)))
                           (NodeSet (NodeSetGenerator Params)))
           :in-theory (disable Missivesp
                               correctroutesp-=>-tomissives)
           :do-not-induct t))))
;; A useful theorem.
(defthm ids-routing
  ;; the ids of the output of routing are equal to the ids
  ;; of the initial list of missives
  (let ((NodeSet (NodeSetGenerator Params)))
    (implies (and (Missivesp M NodeSet)
                  (ValidParamsp Params))
             (equal (V-ids (routing M NodeSet))
                    (M-ids M))))
  :hints (("GOAL"
           :use ((:instance ToMissives-Routing))
           :in-theory (disable ToMissives-Routing))))
```

C.5 Code pour l'ordonnancement

```
; Julien Schmaltz
;; Generic Scheduling Module of GeNoC
;; Feb 16th 2005
;; File: GeNoC-scheduling.lisp
;; Rev. Nov. 22nd 2005
;; JS: NodeSet is not necessary in practice and fundamentally
;; so I remove it from the scheduling definition
(in-package "ACL2")
```

```
(include-book "GeNoC-routing")
;; import also GeNoC-types, GeNoC-nodeset, GeNoC-misc

;; Inputs: TrLst = ( ... (Id frm route) ...)
;; outputs: Scheduled and Delayed (which are both TrLst)
(encapsulate
 ;; Function Scheduling represents the scheduling policy of the
 ;; network.
 ;; arguments: TrLst att
 ;; outputs: Scheduled Delayed newatt
 (((scheduling * *) => (mv * * *)))

 (local (defun consume-attempts (att)
          ;; function that consumes one attempt for each node
          ;; which has still at least one left
  (if (zp (SumOfAttempts att))
              att
            (let* ((top (car att))
                   (node (car top))
                   (atti (cadr top)))
              (if (zp atti)
                  (cons top
                        (consume-attempts (cdr att)))
                (cons (list node (1- atti))
                      (consume-attempts (cdr att))))))))

 (local (defun scheduling (TrLst att)
          ;; local witness
          (mv
           ;; scheduled frames
           (if (zp (SumOfAttempts att))
               nil  ;; no attempt left -> nothing is scheduled
             TrLst)  ;; otherwise everything is scheduled
           ;; delayed frames
           (if (zp (SumOfAttempts att))
               TrLst ;; no attempt left -> everything is delayed
             nil) ;; otherwise nothing is delayed
           (if (zp (SumOfAttempts att))
               att ;; no attempt left -> no modification on att
             (consume-attempts att))))) ;; newatt has less attempts than att

 ;; Proof obligations (also named constraints)
 ;; ----------------------------------------
```

```
;; 1/Type of Scheduled and Delayed travels
;; ---------------------------------------
;; The list of scheduled travels is a valid travel list
(defthm trlstp-scheduled
  (implies (trlstp TrLst)
           (trlstp (mv-nth 0 (scheduling TrLst att)))))

;; so is the list of delayed travels
(defthm trlstp-delayed
  (implies (trlstp TrLst)
           (trlstp (mv-nth 1 (scheduling TrLst att)))))

;; 2/ consume at least one attempt to ensure termination.
;; ------------------------------------------------------
(defthm consume-at-least-one-attempt
  ;; the scheduling policy should consume at least one attempt
  ;; this is a sufficient condition to prove that
  ;; the full network function terminates
  (mv-let (Scheduled Delayed newatt)
  (scheduling TrLst att)
  (declare (ignore Scheduled Delayed))
  (implies (not (zp (SumOfAttempts att)))
   (< (SumOfAttempts newatt) (SumOfAttempts att)))))

;; 3/ Correctness of the scheduled travels
;; ---------------------------------------------------------------------
;; For any scheduled travel sched-tr, there exists a unique travel
;; tr in the initial TrLst, such that IdV(sched-tr) = IdV(tr)
;; and FrmV(sched-tr) = FrmV(tr) and RoutesV(sched-tr) is a
;; sublist of RoutesV(tr).
;; In ACL2, the uniqueness of the ids is given by the predicate
;; TrLstp.
;; ---------------------------------------------------------------------

;; First, let us define this correctness
(defun s/d-travel-correctness (sd-TrLst TrLst/sd-ids)
  ;; sd-TrLst is the list of scheduled or delayed travels
  ;; TrLst/sd-ids is the filtering of the initial TrLst
  ;; according to the ids of the scheduled or delayed travels
  (if (endp sd-TrLst)
      (if (endp TrLst/sd-ids)
          t
        nil)
    (let* ((sd-tr (car sd-TrLst))
```

```
           (tr (car TrLst/sd-ids)))
      (and (equal (FrmV sd-tr) (FrmV tr))
           (equal (IdV sd-tr) (IdV tr))
           (subsetp (RoutesV sd-tr) (RoutesV tr))
           (s/d-travel-correctness (cdr sd-TrLst)
                                   (cdr TrLst/sd-ids))))))

(defthm scheduled-travels-correctness
  (mv-let (Scheduled Delayed newatt)
          (scheduling TrLst att)
          (declare (ignore Delayed newatt))
          (implies (TrLstp TrLst)
                   (s/d-travel-correctness
                    scheduled
                    (extract-sublst TrLst (V-ids Scheduled))))))

(defthm subsetp-scheduled-delayed-ids
  ;; this should be provable from the two lemmas above
  ;; but it will always be trivial to prove, and it is
  ;; useful in subsequent proofs.
  (mv-let (scheduled delayed newatt)
          (scheduling TrLst att)
          (declare (ignore newatt))
          (implies (TrLstp TrLst)
                   (and (subsetp (v-ids scheduled) (v-ids Trlst))
                        (subsetp (v-ids delayed) (v-ids TrLst))))))

;; 4. Correctness of the delayed travels
;; -------------------------------------
;; the correctness of the delayed travels differs from
;; the correctness of the scheduled travels because,
;; for the scheduled travels we will generally keep only
;; one route, but for the delayed travels we will not modify
;; the travels and keep all the routes. In fact, by
;; converting a travel back to a missive we will remove the
;; routes.
;; ----------------------------------------------------------

;; the list Delayed is equal to filtering the initial
;; TrLst according to the Ids of Delayed

(defthm delayed-travel-correctness
  ;; this is a looping rule !!! should be disabled and used
  ;; at the right time
```

```
  (mv-let (Scheduled Delayed newatt)
          (scheduling TrLst att)
          (declare (ignore newatt scheduled))
          (implies (TrLstp TrLst)
                   (equal Delayed
                          (extract-sublst TrLst (V-ids Delayed)))))
  :rule-classes nil)


;; 6/ if the initial AttLst contains only 0, we do not modify it
(defthm mv-nth-2-scheduling-on-zero-attlst
  ;; if every attempt has been consumed
  ;; the new att is equal to the initial one
  (implies (and (zp (SumOfAttempts att))
                (TrLstp trlst))
           (equal
            (mv-nth 2 (scheduling TrLst att));; new att
            att)))

(defthm mv-nth-1-scheduling-on-zero-attlst
  ;; if every attempt has been consumed
  ;; the set of delayed orders is equal to the initial TrLst
  (implies (zp (SumOfAttempts att))
           (equal
            (mv-nth 1 ;; delayed travels
                    (scheduling TrLst att))
            TrLst)))

;; 7/ The intersection of the ids of the scheduled travels and those
;; of the delayed travels is empty
;; -----------------------------------------------------------------

(defthm not-in-delayed-scheduled
  (mv-let (scheduled delayed newatt)
          (scheduling TrLst att)
          (declare (ignore newatt))
          (implies (TrLstp TrLst)
                   (not-in (v-ids delayed) (v-ids scheduled)))))

;; some constraints required because we do not have a definition
;; for scheduling
(defthm consp-scheduling
  ;; for the mv-nth
  (consp (scheduling TrLst att))
```

```
   :rule-classes :type-prescription)

 (defthm true-listp-car-scheduling
   (implies (true-listp TrLst)
            (true-listp (mv-nth 0 (scheduling TrLst att))))
   :rule-classes :type-prescription)

 (defthm true-listp-mv-nth-1-sched
   (implies (TrLstp TrLst)
            (true-listp (mv-nth 1 (scheduling TrLst att))))
   :rule-classes :type-prescription)
) ;; end of scheduling

(defthm checkroutes-member-equal
  (implies (and (checkroutes routes m NodeSet)
                (member-equal r Routes))
           (validroutep r m)))

(defthm checkroutes-subsetp-validroute
  (implies (and (checkroutes routes m NodeSet)
                (consp r)
                (subsetp r routes))
           (and (validroutep (car r) m)
                (subsetp (car r) NodeSet))))

(defthm checkroutes-subsetp
  (implies (and (checkroutes routes m NodeSet)
                (subsetp routes1 routes))
           (checkroutes routes1 m NodeSet)))

(defthm correctroutesp-s/d-travel-correctness
  (implies (and (CorrectRoutesp TrLst/ids (ToMissives TrLst/ids) NodeSet)
                (s/d-travel-correctness TrLst1 TrLst/ids))
           (CorrectRoutesp TrLst1
                           (ToMissives TrLst/ids) NodeSet)))


(defthm scheduling-preserves-route-correctness
  (mv-let (Scheduled Delayed newatt)
          (scheduling TrLst att)
          (declare (ignore Delayed newatt))
          (implies (and (CorrectRoutesp TrLst (ToMissives TrLst) NodeSet)
                        (TrLstp TrLst))
                   (CorrectRoutesp Scheduled
```

```
                                    (ToMissives
                                     (extract-sublst
                                        TrLst
                                        (V-ids scheduled)))
                                    NodeSet)))
  :otf-flg t
  :hints (("GOAL"
           :do-not '(eliminate-destructors generalize)
           :do-not-induct t
           :in-theory
           (disable mv-nth ;; to have my rules used
                    tomissives-extract-sublst TrLstp))))

(defthm scheduling-preserves-route-correctness-delayed
  (mv-let (Scheduled Delayed newatt)
          (scheduling TrLst att)
          (declare (ignore scheduled newatt))
          (implies (and (CorrectRoutesp TrLst (ToMissives TrLst) NodeSet)
                        (TrLstp TrLst))
                   (CorrectRoutesp Delayed
                                   (ToMissives Delayed)
                                   NodeSet)))
  :otf-flg t
  :hints (("GOAL"
           :do-not '(eliminate-destructors generalize)
           :do-not-induct t
           :use
           ((:instance delayed-travel-correctness))
           :in-theory (disable tomissives-extract-sublst TrLstp))))

;; the three defthms are similar to the three used to prove
;; missivesp-extract-sublst in GeNoC.lisp ... (proof by analogy)

(defthm valid-trlstp-assoc-equal
  (implies (and (TrLstp L)
                (member-equal e (V-ids L)))
           (TrLstp (list (assoc-equal e L)))))

(defthm TrLstp-cons
  ;; lemma used in the next defthm
  ;; if we cons a valid travel to a filtered valid list
  ;; of travel, we obtain a valid list of travel if the
  ;; consed travel has an id less than the first of the filter
  ;; and this id is not in the filter
```

```
  (implies (and (trlstp (extract-sublst L ids))
                (trlstp (list (assoc-equal e L)))
                (not (member-equal e ids))
                (subsetp ids (V-ids L)))
           (trlstp (cons (assoc-equal e L)
                         (extract-sublst L ids)))))

(defthm trlstp-extract-sublst
  (implies (and (TrLstp TrLst) (subsetp ids (v-ids TrLst))
                (no-duplicatesp ids) (true-listp ids))
           (trlstp (extract-sublst TrLst ids)))
  :hints (("GOAL"
           :in-theory (disable TrLstp))))

(defthm extract-sublst-subsetp-v-ids
  (implies (and (subsetp ids (V-ids l))
                (true-listp ids)
                (TrLstp l))
           (equal (v-ids (extract-sublst l ids))
                  ids)))

(defthm Missivesp-mv-nth-1-scheduling
  (let ((NodeSet (NodeSetGenerator Params)))
    (implies (and (CorrectRoutesp TrLst (ToMissives TrLst) NodeSet)
                  (ValidParamsp Params)
                  (TrLstp TrLst))
             (Missivesp (ToMissives
                         (mv-nth 1
                                 (scheduling TrLst att)))
                        NodeSet)))
  :otf-flg t
  :hints (("GOAL"
           :do-not-induct t
           :use ((:instance delayed-travel-correctness)
                 (:instance trlstp-delayed))
           :in-theory (disable Missivesp trlstp-delayed
                               TOMISSIVES-EXTRACT-SUBLST
                               TrLstp mv-nth))))
```

C.6 Code pour *GeNoC*

```
;; Julien Schmaltz
;; Generic Routing Module of GeNoC
```

```
;; June 20th 2005
;; File: GeNoC-routing.lisp

(in-package "ACL2")

;; we import the previous books
(include-book "GeNoC-scheduling")

(in-theory (disable mv-nth))

;;---------------------------------------------------------------------
;;
;;                         GeNoC_t
;;
;;---------------------------------------------------------------------

;; Tail definition of GeNoC_t
;; --------------------------
(defun GeNoC_t (M NodeSet att TrLst)
  ;; The composition of Routing and Scheduling is built by function GeNoC_t.
  ;; It takes as arguments:
  ;; - a list of missives, M
  ;; - the set of existing nodes, NodeSet
  ;; - the list of attempts, att
  ;; - an accumulator of travels, TrLst
  ;;
  ;; we set the measure to be SumOfAttempts(att)
  (declare (xargs :measure (SumOfAttempts att)))
  (if (zp (SumOfAttempts att))
      ;; if every attempt has been consumed, we return the accumulator
      ;; TrLst and the remaining missives M
      (mv TrLst M)
    ;; else,
    (let ((V (Routing M NodeSet)))
      ;; we compute the routes. This produces the travel list V.
      (mv-let (Scheduled Delayed newAtt)
              ;; we call function scheduling
              (Scheduling V att)
              ;; we enter the recursive call and accumulate the
              ;; scheduled travels
              (GeNoC_t (ToMissives Delayed) NodeSet newAtt
                       (append Scheduled TrLst))))))
```

```
;; Correctness of GeNoC_t
;; ----------------------
(defun CorrectRoutes-GeNoC_t (routes m-dest)
  ;; GeNoC_t is correct if every element ctr of the output list
  ;; is such that (a) FrmV(ctr) = FrmM(m) and (b) forall r in
  ;; RoutesV(ctr) Last(r) = DestM(m). For the m such that
  ;; IdM(m) = IdV(ctr).
  ;; This function checks that (b) holds.
  (if (endp routes)
      t
    (let ((r (car routes)))
      (and (equal (car (last r)) m-dest)
           (CorrectRoutes-GeNoC_t (cdr routes) m-dest)))))

(defun GeNoC_t-correctness1 (TrLst M/TrLst)
  ;; we complement the correctness of GeNoC_t
  (if (endp TrLst)
      (if (endp M/TrLst)
          t
        nil)
    (let* ((tr (car TrLst))
           (v-frm (FrmV tr))
           (routes (RoutesV tr))
           (m (car M/TrLst))
           (m-frm (FrmM m))
           (m-dest (DestM m)))
      (and (equal v-frm m-frm)
           (CorrectRoutes-GeNoC_t routes m-dest)
           (GeNoC_t-correctness1 (cdr TrLst)
                                 (cdr M/TrLst))))))

(defun GeNoC_t-correctness (TrLst M)
  ;; before checking correctness we filter M
  ;; according to the ids of TrLst
  (let ((M/TrLst (extract-sublst M (V-ids TrLst))))
    (GeNoC_t-correctness1 TrLst M/TrLst)))


;; Non tail definition of GeNoC_t
;; ------------------------------
(defun GeNoC_t-non-tail-Comp (M NodeSet att)
  ;; we define a non tail function that computes the
  ;; first output of GeNoC_t, i.e the completed transactions
  (declare (xargs :measure (SumOfAttempts att)))
```

```
  (if (zp (SumOfAttempts att))
      ;; if every attempt has been consumed, we return the accumulator
      ;; TrLst and the remaining missives M
      nil
    ;; else,
    (let ((V (Routing M NodeSet)))
      ;; we compute the routes. This produces the travel list V.
      (mv-let (Scheduled Delayed newAtt)
              ;; we call function scheduling
              (Scheduling V att)
              ;; we enter the recursive call and accumulate the
              ;; scheduled travels
              (append (GeNoC_t-non-tail-Comp (ToMissives Delayed)
                                             NodeSet newAtt)
                      Scheduled)))))

;; we now prove that this function is right

(defthm true-listp-GeNoC_t-non-tail-comp
  (true-listp (GeNoC_t-non-tail-Comp M NodeSet att))
  :rule-classes :type-prescription)

(defthm GeNoC_t-non-tail-=-tail-comp
  (implies (true-listp TrLst)
           (equal (car (GeNoC_t M NodeSet att TrLst))
                  (append (GeNoC_t-non-tail-Comp M NodeSet Att)
                          TrLst))))

;; Proof of GeNoC_t correctness
;; ----------------------------

;; first we add a lemma that tells ACL2 that
;; converting the travels that are routed and delayed
;; produced a valid list of missives
(defthm missivesp-mv-nth1-scheduling-routing
  (let ((NodeSet (NodeSetGenerator Params)))
    (implies (and (Missivesp M NodeSet)
                  (ValidParamsp Params))
             (Missivesp
              (ToMissives (mv-nth 1
                                  (scheduling (routing M NodeSet)
                                              att)))
              NodeSet)))
  :hints (("GOAL"
```

```
           :in-theory (disable
                       TrLstp Missivesp))))



;; the recursive call of genoc_t-non-tail-comp calls append
;; we put the append at the top.
;; to do so we add the two rules below:

(defthm v-ids-append
  ;; the ids of an append is the append of the ids
  (equal (v-ids (append a b))
         (append (v-ids a) (v-ids b))))

(defthm extract-sublst-append
  ;; filtering according to an append is the append
  ;; of the filtering.
  (equal (extract-sublst M (append id1 id2))
         (append (extract-sublst M id1)
                 (extract-sublst M id2))))


;; then to split the proof is two cases, we replace the
;; append by a conjunction.
;; the rule below allows this decomposition:

(defthm correctroutess1-append
  (implies (and (equal (len a) (len c))
                (equal (len b) (len d)))
           (equal (genoc_t-correctness1 (append a b)
                                        (append c d))
                  (and (Genoc_T-Correctness1 a c)
                       (Genoc_T-Correctness1 b d)))))

;; To have this lemma used we need to prove some
;; additional properties between len and extract-sublst
;; and len and v-ids (e.g. a is a call to v-ids)

(defthm len-extract-sublst
  (equal (len (extract-sublst L ids))
         (len ids)))

(defthm len-v-ids
  (equal (len (v-ids x))
```

```
         (len x)))

;; now we need to prove some lemmas so that previous rules
;; (from GeNoC-misc) about extract-sublst, tomissives, etc could
;; fire.

(defthm subsetp-trans
  ;; transitivity of subsetp
  (implies (and (subsetp x y)
                (subsetp y z))
           (subsetp x z)))

(defthm v-ids-GeNoC_t-non-tail-comp
  ;; the ids of the output of genoc_t-non-tail-comp is a
  ;; subset of those of M
  ;; for this theorem the rule ids-routing is useful
  (let ((NodeSet (NodeSetGenerator Params)))
    (implies (and (Missivesp M NodeSet)
                  (ValidParamsp Params))
             (let ((Gnt (Genoc_T-Non-Tail-Comp M NodeSet att)))
               (subsetp (V-ids Gnt) (M-ids M)))))
  :hints (("GOAL"
           :in-theory
           (disable missivesp TrLstp))
          ("Subgoal *1/3"
           :use
           ((:instance subsetp-scheduled-delayed-ids
                       (TrLst (Routing M (NodeSetGenerator Params)))))
           :in-theory
           (disable subsetp-scheduled-delayed-ids Missivesp TrLstp))))

(defthm not-in-no-duplicatesp-equal-append
  ;; if x is not in y and both do not have duplicates then
  ;; their append has no duplicate too
  (implies (and (no-duplicatesp-equal x)
                (not-in x y)
                (no-duplicatesp-equal y))
           (no-duplicatesp-equal (append x y))))

(defthm subsetp-not-in
  ;; if a list y and no element in common with z
  ;; then any sublist x of y has no element in z
  (implies (and (not-in delayed scheduled)
                (subsetp x delayed))
```

```
           (not-in x scheduled)))

(defthm not-in-v-ids-genoc_t-non-tail-comp
  ;; if the ids of a list have no common element with
  ;; another ids then the output of genoc_t-non-tail-comp does
  ;; not introduce any new id
  (let ((NodeSet (NodeSetGenerator Params)))
    (implies (and (not-in (m-ids delayed) Sched-ids)
                  (Missivesp delayed NodeSet)
                  (ValidParamsp Params))
             (not-in (v-ids (genoc_t-non-tail-comp delayed NodeSet att))
                     Sched-ids)))
  :otf-flg t
  :hints (("GOAL"
           :do-not-induct t
           :in-theory (disable missivesp))))

(defthm fwd-trlstp
  ;; because we disable trlstp, this rule adds its content
  ;; as hypotheses
  (implies (TrLstp TrLst)
           (and (validfields-trlst trlst)
                (true-listp trlst)
                (no-duplicatesp-equal (v-ids trlst))))
  :rule-classes :forward-chaining)

(defthm v-ids-GeNoC_t-non-tail-comp-no-dup
  ;; the ids of the output of genoc_t-non-tail-comp have no dup
  (let ((NodeSet (NodeSetGenerator Params)))
    (implies (and (Missivesp M NodeSet)
                  (ValidParamsp Params))
             (let ((Gnt (Genoc_T-Non-Tail-Comp M NodeSet att)))
               (no-duplicatesp-equal (V-ids Gnt)))))
  :otf-flg t
  :hints (("GOAL"
           :do-not '(eliminate-destructors generalize)
           :do-not-induct t
           :induct (genoc_t-non-tail-comp M (nodeSetGenerator Params) att)
           :in-theory (disable missivesp TrLstp))
          ("Subgoal *1/2"
           :use
           ((:instance
             not-in-v-ids-genoc_t-non-tail-comp
             (delayed (tomissives
```

```
                    (mv-nth 1 (scheduling
                               (routing M (nodeSetGenerator Params))
                               att))))
             (att (mv-nth 2 (scheduling
                             (routing M (nodeSetGenerator Params))
                             att)))
             (Sched-ids (v-ids (mv-nth 0 (scheduling
                                          (routing M (nodeSetGenerator Params))
                                          att)))))
            (:instance trlstp-scheduled
                       (TrLst (routing M (Nodesetgenerator Params)))))
           :in-theory
           (disable trlstp-scheduled mv-nth trlstp
                    not-in-v-ids-genoc_t-non-tail-comp Missivesp))))

(defthm extract-sublst-subsetp-m-ids
  ;; filtering a list l according to a subset ids of its identifiers
  ;; produces a list the ident. of which are ids
  (implies (and (subsetp ids (M-ids l))
                (true-listp ids)
                (Validfields-M l))
           (equal (M-ids (extract-sublst l ids))
                  ids)))

(defthm ToMissives-Delayed/Rtg
  ;; we prove that the conversion of the delayed travels
  ;; into a list of missives is equal to the filtering
  ;; of the initial list of missives M according to the ids
  ;; of the delayed travels.
  (let ((NodeSet (NodeSetGenerator Params)))
    (mv-let (Scheduled/Rtg Delayed/Rtg newAtt)
            (scheduling (Routing M NodeSet) att)
            (declare (ignore Scheduled/Rtg newAtt))
            (implies (and (Missivesp M NodeSet)
                          (ValidParamsp Params))
                     (equal (ToMissives Delayed/Rtg)
                            (extract-sublst M (V-ids Delayed/Rtg))))))
  :otf-flg t
  :hints (("GOAL"
           :do-not-induct t
           :do-not '(eliminate-destructors generalize fertilize)
           :use ((:instance ToMissives-extract-sublst
                            (L (Routing M (NodeSetGenerator Params)))
                            (ids
```

```
                            (V-ids (mv-nth 1
                                           (scheduling
                                            (Routing M (NodeSetGenerator Params))
                                            att)))))
                  (:instance delayed-travel-correctness
                             (TrLst (Routing M (NodeSetGenerator Params))))
                  (:instance subsetp-scheduled-delayed-ids
                             (TrLst (Routing M (NodeSetGenerator Params)))))
           :in-theory (disable TrLstp Missivesp
                               ToMissives-extract-sublst
                               subsetp-scheduled-delayed-ids))))

(defthm valid-missive-assoc-equal
  ;; a list of a member of a valid list of missives
  ;; is a valid list of missives
  (implies (and (Missivesp M NodeSet)
                (member-equal e (M-ids M)))
           (Missivesp (list (assoc-equal e M)) NodeSet)))

(defthm missivesp-cons
  ;; lemma used in the next defthm
  ;; if we cons a valid missive to a filtered valid list
  ;; of missives, we obtain a valid list of missives if the
  ;; the id of the consed missive is not in the filter
  (implies (and (missivesp (extract-sublst M ids) nodeset)
                (missivesp (list (assoc-equal e M)) nodeset)
                (not (member-equal e ids))
                (subsetp ids (M-ids M)))
           (missivesp (cons (assoc-equal e M)
                            (extract-sublst M ids))
                      nodeset)))


(defthm missivesp-extract-sublst
  (let ((NodeSet (NodeSetGenerator Params)))
    (implies (and (missivesp M NodeSet)
                  (ValidParamsp Params)
                  (true-listp ids)
                  (no-duplicatesp-equal ids)
                  (subsetp ids (M-ids M)))
             (Missivesp (extract-sublst M ids) NodeSet)))
  :hints (("GOAL"
           :in-theory (disable missivesp))
          ("Subgoal *1/1"
```

```
           :in-theory (enable missivesp))))


(defthm fwd-missivesp
  ;; as missivesp is disabled we prove this rule to add
  ;; the content of missivesp as hypotheses
  (implies (missivesp M NodeSet)
           (and (Validfields-M M)
                (subsetp (M-orgs M) NodeSet)
                (subsetp (M-dests M) NodeSet)
                (True-listp M)
                (No-duplicatesp (M-ids M))))
  :rule-classes :forward-chaining)


(defthm v-ids_G_nt_sigma_subsetp-v-ids-delayed/rtg
  ;; this lemma is used in the subsequent proofs
  ;; it makes a fact "explicit"
  (let ((NodeSet (NodeSetGenerator Params)))
    (mv-let (scheduled/rtg delayed/rtg newAtt)
            (scheduling (routing M NodeSet) att)
            (declare (ignore scheduled/rtg))
            (implies (and (Missivesp M NodeSet)
                          (ValidParamsp Params))
                     (subsetp
                      (V-ids
                       (genoc_t-non-tail-comp
                        (extract-sublst M (v-ids delayed/rtg))
                        NodeSet newAtt))
                      (V-ids delayed/rtg)))))
  :otf-flg t
  :hints (("GOAL"
           :do-not-induct t
           :use
           ((:instance subsetp-scheduled-delayed-ids
                       (TrLst (Routing M (NodeSetGenerator Params))))
            (:instance trlstp-delayed
                       (TrLst (routing M (NodeSetGenerator Params))))
            ;; the following is required because in the conclusion of the
            ;; rule there is no call to extract-sublst
            (:instance v-ids-GeNoC_t-non-tail-comp
                       (M (extract-sublst
                           M
                           (V-ids (mv-nth 1 (scheduling
```

```
                                          (routing M (NodeSetGenerator Params))
                                          att)))))
                      (att (mv-nth 2 (scheduling
                                      (routing M (NodeSetGenerator Params))
                                      att)))))
           :in-theory (disable subsetp-scheduled-delayed-ids
                               trlstp-delayed
                               v-ids-GeNoC_t-non-tail-comp
                               trlstp missivesp))))


;; Scheduled/Rtg does not modify frames
;; ------------------------------------

(defthm s/d-travel-v-frms
  (implies (and (TrLstp sd-trlst)
                (s/d-travel-correctness sd-trlst TrLst/sd-ids))
           (equal (V-frms sd-trlst) (V-frms TrLst/sd-ids))))

(defthm m-frms-to-v-frms
  ;; this rule is only used to rewrite the next theorem to
  ;; the previous one.
  (equal (m-frms (toMissives x))
         (v-frms x)))

(defthm Scheduled-v-frms-m-frms
  ;; we prove that the frames of the scheduled travels
  ;; are equal to the frames of the conversion of the initial list of travels
  ;; filtered according to the ids of the scheduled travels
  (mv-let (Scheduled Delayed newAtt)
          (scheduling TrLst att)
          (declare (ignore Delayed newAtt))
          (implies (and (TrLstp TrLst)
                        (ValidParamsp Params))
                   (equal
                    (V-frms scheduled)
                    (M-frms
                     (ToMissives (extract-sublst TrLst
                                                 (v-ids scheduled)))))))
  :hints (("GOAL"
           :use ((:instance s/d-travel-v-frms
                            (sd-trlst
                             (mv-nth 0 (scheduling TrLst att)))
                            (TrLst/sd-ids
```

```
                          (extract-sublst
                           TrLst
                           (v-ids
                            (mv-nth 0 (scheduling TrLst att))))))
                (:instance scheduled-travels-correctness))
           :in-theory (disable TrLstp s/d-travel-v-frms mv-nth))))

(in-theory (disable m-frms-to-v-frms))

(defthm Scheduled/Rtg_not_modify_frames
  ;; we prove the the frames of the scheduled travels produced
  ;; by scheduling and routing are equal to the frames
  ;; of the initial list of missives
  (let ((NodeSet (NodeSetGenerator Params)))
  (mv-let (Scheduled/Rtg Delayed/Rtg newAtt)
          (scheduling (routing M NodeSet) att)
          (declare (ignore Delayed/Rtg newAtt))
          (implies (and (Missivesp M NodeSet)
                        (ValidParamsp Params))
                   (equal (V-frms Scheduled/Rtg)
                          (M-frms
                           (extract-sublst
                            M (v-ids Scheduled/Rtg)))))))
  :hints (("GOAL"
           :do-not-induct t
           :do-not '(eliminate-destructors generalize fertilize)
           :use ((:instance Scheduled-v-frms-m-frms
                            (TrLst (routing M (NodeSetGenerator Params))))
                 (:instance subsetp-scheduled-delayed-ids
                            (TrLst (routing M (NodeSetGenerator Params)))))
           :in-theory (disable TrLstp Missivesp
                               subsetp-scheduled-delayed-ids
                               scheduled-v-frms-m-frms))))

(defthm correctroutesp-vm-frms-gc1
  ;; the correctness of routes and equality of frames imply
  ;; the main predicate (correctness of genoc_t-non-tail-comp)
  (implies (and (correctroutesp L (extract-sublst M (v-ids L))
                                NodeSet)
                (equal (V-frms L)
                       (m-frms (extract-sublst M (v-ids L)))))
           (Genoc_T-Correctness1 L
                                 (extract-sublst m (v-ids L)))))
```

```
(defthm GC1_scheduled/Rtg
  ;; we prove the correctness of the scheduled travels
  (let ((NodeSet (NodeSetGenerator Params)))
    (implies (And (missivesp M NodeSet)
                  (ValidParamsp Params))
             (mv-let (scheduled/rtg delayed/rtg newAtt)
                     (scheduling (routing M NodeSet) att)
                     (declare (ignore delayed/rtg newAtt))
                     (genoc_t-correctness1
                      scheduled/rtg
                      (extract-sublst M (v-ids scheduled/rtg))))))
  :otf-flg t
  :hints (("GOAL"
           :do-not-induct t
           :do-not '(eliminate-destructors generalize fertilize)
           :use
           ((:instance Scheduled/Rtg_not_modify_frames)
            (:instance subsetp-scheduled-delayed-ids
                       (TrLst (Routing M (NodeSetGenerator Params))))
            (:instance
             scheduling-preserves-route-correctness
             (NodeSet (NodeSetGenerator Params))
             (TrLst (routing M (NodeSetGenerator Params))))
            (:instance correctroutesp-vm-frms-gc1
                       (NodeSet (NodeSetGenerator Params))
                       (L (mv-nth 0 (scheduling
                                     (routing m (NodeSetGenerator Params))
                                     att)))))
           :in-theory (disable TrLstp Missivesp
                               Correctroutesp-Vm-Frms-Gc1
                               subsetp-scheduled-delayed-ids
                               Scheduling-Preserves-Route-Correctness
                               Scheduled/Rtg_not_modify_frames))))

(defthm GeNoC_t-thm
  ;; now we can prove the correctness of GeNoC_t
  (let ((NodeSet (NodeSetGenerator Params)))
    (implies (and (Missivesp M NodeSet)
                  (ValidParamsp Params))
             (mv-let (Cplt Abt)
                     (GeNoC_t M NodeSet att nil)
                     (declare (ignore Abt))
                     (GeNoC_t-correctness Cplt M))))
  :otf-flg t
```

```
  :hints (("GOAL"
           :induct (GeNoC_t-non-tail-comp M (NodeSetGenerator Params) Att)
           :do-not '(eliminate-destructors generalize)
           :in-theory (disable TrLstp Missivesp)
           :do-not-induct t)
          ("Subgoal *1/2"
           :use
           ((:instance trlstp-delayed
                       (TrLst (routing M (NodeSetGenerator Params))))
            (:instance subsetp-scheduled-delayed-ids
                       (TrLst (Routing M (NodeSetGenerator Params))))
            (:instance GC1_scheduled/Rtg))
           :in-theory (e/d (mv-nth)
                           (TrLstp missivesp trlstp-delayed
                                   subsetp-scheduled-delayed-ids
                                   GC1_scheduled/Rtg)))))

;;-------------------------------------------------------------
;;      Definition and Validation of GeNoC
;;-------------------------------------------------------------

;; we load the generic definitions of the interfaces
(include-book "GeNoC-interfaces")

;; ComputeMissives
;; --------------
(defun ComputeMissives (Transactions)
  ;; apply the function p2psend to build a list of missives
  ;; from a list of transactions
  (if (endp Transactions)
      nil
    (let* ((trans (car Transactions))
           (id (IdT trans))
           (org (OrgT trans))
           (msg (MsgT trans))
           (dest (DestT trans)))
      (cons (list id org (p2psend msg) dest)
            (ComputeMissives (cdr Transactions))))))

;; ComputeResults
;; --------------
(defun ComputeResults (TrLst)
  ;; apply the function p2precv to build a list of results
  ;; from a list of travels
```

```
  (if (endp TrLst)
      nil
    (let* ((tr (car TrLst))
           (id (IdV tr))
           (r (car (routesV tr)))
           (dest (car (last r)))
           (frm (FrmV tr)))
      (cons (list id dest (p2precv frm))
            (ComputeResults (cdr TrLst))))))

;; GeNoC
;; -----
(defun GeNoC (Trs NodeSet att)
  ;; main function
  (mv-let (Responses Aborted)
          (GeNoC_t (ComputeMissives Trs) NodeSet att nil)
          (mv (ComputeResults Responses) Aborted)))

;; GeNoC Correctness
;; -----------------
(defun genoc-correctness (Results Trs/ids)
  ;; Trs/ids is the initial list of transactions filtered according
  ;; to the ids of the list of results.
  ;; We check that the messages and the destinations of these two lists
  ;; are equal.
  (and (equal (R-msgs Results)
              (T-msgs Trs/ids))
       (equal (R-dests Results)
              (T-dests Trs/ids))))

(defthm M-ids-computesmissives
  ;; lemma for the next defthm
  (equal (M-ids (computemissives Trs))
         (T-ids trs)))

(defthm missivesp-computemissives
  (implies (transactionsp trs NodeSet)
           (missivesp (ComputeMissives trs) NodeSet)))


(defun all-frms-equal-to-p2psend (TrLst Trs)
  ;; check that every frame of TrLst is equal to the application
  ;; of p2psend to the corresponding message in the list of
  ;; transactions Trs
```

```
  (if (endp TrLst)
      (if (endp Trs)
          t
        nil)
    (let* ((tr (car TrLst))
           (trans (car Trs))
           (tr-frm (FrmV tr))
           (t-msg (MsgT trans)))
      (and (equal tr-frm (p2psend t-msg))
           (all-frms-equal-to-p2psend (cdr TrLst) (cdr Trs))))))

(defthm GC1-=>-all-frms-equal-to-p2psend
  (implies (GeNoC_t-correctness1 TrLst (ComputeMissives Trs))
           (all-frms-equal-to-p2psend TrLst Trs)))

(defthm all-frms-equal-r-msgs-t-msgs
  ;; if frames have been computed by p2psend then
  ;; computeresults applies p2precv. We get thus the initial msg.
  (implies (and (all-frms-equal-to-p2psend TrLst Trs)
                (validfields-trlst TrLst))
           (equal (R-msgs (ComputeResults TrLst))
                  (T-msgs Trs))))

(defthm R-ids-computeresults
  (equal (R-ids (computeresults x))
         (V-ids x)))

(defthm m-dests-computemissives
  (equal (M-dests (computeMissives trs))
         (T-dests trs)))


(defthm GC1-r-dest-m-dests
  (implies (and (GeNoC_t-correctness1 TrLst M/TrLst)
                (validfields-trlst TrLst)
                (Missivesp M/TrLst NodeSet))
           (equal (R-dests (ComputeResults TrLst))
                  (M-dests M/TrLst))))

(defthm validfields-append
  ;; lemma for the next defthm
  (implies (and (validfields-trlst l1)
                (validfields-trlst l2))
           (validfields-trlst (append l1 l2))))
```

```
(defthm validfields-trlst-GeNoC_nt
  ;; to use the lemma all-frms-equal-to-p2psend we need to establish
  ;; that GeNoC_nt contains travels with validfields
  ;; and that it contains no duplicated ids
  (let ((NodeSet (NodeSetGenerator Params)))
    (implies (and (Missivesp M NodeSet)
                  (ValidParamsp Params))
             (validfields-trlst (genoc_t-non-tail-comp M NodeSet att))))
  :otf-flg t
  :hints (("GOAL"
           :do-not-induct t
           :induct (Genoc_T-Non-Tail-Comp M (NodeSetGenerator Params) att))
          ("Subgoal *1/2"
           :use ((:instance trlstp-delayed
                            (TrLst (Routing M (NodeSetGenerator Params))))
                 (:instance trlstp-scheduled
                            (TrLst (Routing M (NodeSetGenerator Params))))
                 (:instance Missivesp-mv-nth-1-scheduling
                            (TrLst (routing M (NodeSetGenerator Params)))))
           :in-theory (disable trlstp-delayed trlstp-scheduled
                               Missivesp-mv-nth-1-scheduling))))

;; the next four lemmas are similar to those used to prove
;; the lemma tomissives-extract-sublst .... (proof by analogy)

(defthm computemissives-append
  (equal (computemissives (append a b))
         (append (computemissives a) (computemissives b))))

(defthm member-equal-assoc-equal-not-nil-t-ids
  ;; if e is an Id of a travel of L
  ;; then (assoc-equal e L) is not nil
  (implies (and (member-equal e (T-ids Trs))
                (Validfields-T Trs))
           (assoc-equal e Trs)))

(defthm computemissives-assoc-equal
  ;; if (assoc-equal e L) is not nil then we can link
  ;; assoc-equal and computemissives as follows:
  ;; (this lemma is needed to prove the next defthm)
  (implies (assoc-equal e L)
           (equal (computemissives (list (assoc-equal e L)))
```

```
                (list (assoc-equal e (computemissives L))))))

(defthm computemissives-extract-sublst
  ;; calls of computemissives are moved into calls
  ;; of extract-sublst
  (implies (and (subsetp ids (t-ids trs))
                (validfields-t trs))
           (equal (ComputeMissives (extract-sublst trs ids))
                  (extract-sublst (ComputeMissives trs) ids)))
  :otf-flg t
  :hints (("GOAL"
           :induct (extract-sublst Trs ids)
           :do-not-induct t
           :in-theory (disable computemissives append))))


(defthm m-dest-t-dests-extract-sublst
  (implies (and (subsetp ids (t-ids trs))
                (validfields-t trs))
           (equal (m-dests (extract-sublst (computemissives Trs) ids))
                  (t-dests (extract-sublst trs ids))))
  :hints (("GOAL"
           :do-not-induct t
           :use (:instance m-dests-computemissives
                           (Trs (extract-sublst Trs ids)))
           :in-theory (disable m-dests-computemissives))))

(defthm fwd-chaining-transactionsp
  (implies (Transactionsp Trs NodeSet)
           (and (validfields-t Trs)
                (true-listp trs)
                (subsetp (t-orgs trs) nodeset)
                (subsetp (t-dests trs) nodeset)
                (no-duplicatesp-equal (t-ids trs))))
  :rule-classes :forward-chaining)

(defthm GeNoC-is-correct
  (let ((NodeSet (NodeSetGenerator Params)))
    (mv-let (results aborted)
            (GeNoC Trs NodeSet att)
            (declare (ignore aborted))
            (implies (and (Transactionsp Trs NodeSet)
                          (equal NodeSet (NodeSetGenerator Params))
                          (ValidParamsp Params))
```

```
                    (GeNoC-correctness
                     results
                     (extract-sublst Trs (R-ids results))))))
  :otf-flg t
  :hints (("GOAL"
           :do-not-induct t
           :use ((:instance GeNoC_t-thm
                            (M (ComputeMissives Trs)))
                 (:instance v-ids-GeNoC_t-non-tail-comp
                            (M (computemissives trs)))
                 (:instance all-frms-equal-r-msgs-t-msgs
                            (TrLst (genoc_t-non-tail-comp
                                    (computeMissives trs)
                                    (Nodesetgenerator params) att))
                            (Trs (extract-sublst Trs
                                                 (v-ids
                                                  (genoc_t-non-tail-comp
                                                   (computeMissives trs)
                                                   (nodesetgenerator params)
                                                   att)))))
                 (:instance GC1-r-dest-m-dests
                            (TrLst (genoc_t-non-tail-comp
                                    (computeMissives trs)
                                    (nodesetgenerator params) att))
                            (M/TrLst
                             (extract-sublst (ComputeMissives Trs)
                                             (V-ids (genoc_t-non-tail-comp
                                                     (computeMissives trs)
                                                     (nodesetgenerator params)
                                                     att))))
                            (NodeSet (Nodesetgenerator params))))
           :in-theory (e/d (mv-nth)
                           (GeNoC_t-thm missivesp trlstp
                            all-frms-equal-r-msgs-t-msgs
                            v-ids-GeNoC_t-non-tail-comp
                            transactionsp GC1-r-dest-m-dests)))))
```

Bibliographie

[Acc] Accellera. *Property Specification Language Reference Manual Version 1.1.*

[ACG+03] A. Adriahantenaina, H. Charlery, A. Greiner, L. Mortiez, and C.A. Zeferino. SPIN : A Scalable, Packet Switched, On-Chip Micro-Network. In *Proc. of Design Automation and Test in Europe Conference and Exhibition (DATE'03)*, 2003.

[AG03] A. Adriahantenaina and A. Greiner. Micro-network for SoC : Implementation of a 32-port SPIN networl. In *Proc. of Design Automation and Test in Europe Conference and Exhibition (DATE'03)*, pages 1129–1130, 2003.

[AK86] K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6) :307–309, 1986.

[AL94] M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5) :1543–1571, September 1994.

[Amj04] H. Amjad. Model Checking the AMBA Protocol in HOL. Technical report, University of Cambridge, Computer Laboratory, September 2004.

[ARM00] ARM. *AMBA Specification Rev 2.0*, 2000.

[ARM01] ARM. *Multi-Layer AHB : Overview*, 2001.

[BAPM83] M. Ben-Ari, A. Pnueli, and Z. Manna. The Temporal Logic of Branching Time. *Acta Informatica*, 20 :207–226, 1983.

[BCC98] S. Berezin, S. Campos, and E. M. Clarke. Compositional Reasoning in Model Checking. In *COMPOS'97 : Revised Lectures from the International Symposium on Compositionality : The Significant Difference*, pages 81–102. Springer-Verlag, 1998.

[Bel03] G. Bella. Inductive Verification of Smart Card Protocols. *Journal of Computer Security*, 11(1) :87–132, 2003.

[Ber02] D. Berry. Formal Methods : The Very Idea. Some Thoughts About Why They Work When They Work. *Science of Computer Programming*, 42(1) :11–27, January 2002.

[BFS95] R. Bharadwaj, A. Felty, and F. Stomp. Formalizing Inductive Proofs of Network Algorithms. In *1995 Asian Computing Conference*, volume 1023 of *LNCS*. Springer-Verlag, 1995.

[BGKM91] R.S. Boyer, D. Goldschlag, M. Kaufmann, and J Strother Moore. Functional Instanciation in First Order Logic. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computations : Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991.

[BGR01] D. Borrione, P. Georgelin, and V. Moraes Rodrigues. Symbolic Simulation of VHDL with ACL2. In P.J. Ashended, J.P. Mermet, and R. Seepold, editors, *System-on-chip Methodologies and Design Languages*, pages 59–70. Kluwer Academic Publisher, 2001.

[BJ01] Christoph Berg and Christian Jacobi. Formal Verification of the VAMP Floating Point Unit. In *Proc. 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *LNCS*, pages 325–339. Springer, 2001.

[BJK+03] S. Beyer, C. Jacobi, D. Kroning, D. Leinenbach, and W.J. Paul. Instantiating Uninterpreted Functional Units and Memory System : Functional Verification of the VAMP. In D. Geist and E. Tronci, editors, *Correct Hardware Design and Verification Methods (CHARME'03)*, volume 2860 of *LNCS*, pages 51–65, L'Aquila, Italy, October 2003. Springer-Verlag.

[BKM96] B. Brock, M. Kaufmann, and J Strother Moore. ACL2 Theorems about Commercial Microprocessors. In M. Srivas and A. Camilleri, editors, *Proceedings of Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *LNCS*, pages 275–293. Springer-Verlag, 1996.

[BM88] R. S. Boyer and J Strother Moore. *A Computation Logic Handbook*. Academic Press, 1988.

[BM02] L. Benini and G. De Micheli. Networks on Chips : A New SoC Paradigm. *Computer*, 35(1) :70–78, 2002.

[BMN+03] A. Bartic, J-Y. Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde, and R. Lauwereins. Highly Scalable Network on Chip for Reconfigurable Systems. In *Proceedings of the International Conference on System-On-Chip 2003 (SoC'03)*, pages 79–82, 2003.

[Bry86] R. E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions in Computers*, 8(35) :677–691, 1986.

[BS95] R. E. Bryant and C. J. H. Seger. Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. *Formal Methods in System Design : An International Journal*, 6(2) :147–189, March 1995.

[Bue05] W. Buettner. Is Formal Verification Bound to Remain a Junior Partner of Simulation ? In D. Borrione and W. Paul, editors, *Correct Hardware Design and Verification Methods (CHARME'05)*, volume 3725 of *LNCS*, 2005. Invited Speaker.

[CE81] E. Clarke and E.A. Emerson. Design and Synthesis of Synchronisation Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs : Workshop*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.*, 8(2) :244–263, 1986.

[CF98] B. Le Charlier and P. Flener. Specifications are Necessarily Informal, or : Some More Myths of Formal Methods. *Journal of Systems and Software, Special Issue on Formal Methods Technology Transfer*, 40(3) :275–296, March 1998.

[CFJ93] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting Symmetry in Temporal Logic Model Checking. In *CAV '93 : Proceedings of the 5th International Conference on Computer Aided Verification*, pages 450–462. Springer-Verlag, 1993.

[CGJ97] E.M. Clarke, O. Grumberg, and S. Jha. Verifying Parameterized Networks. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(5) :726–750, 1997.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[Coq] *LogiCal Project The Coq Development TEAM : The Coq Proof Assistant.*

[CW96] E. Clarke and J. Wing. Formal Methods : State of the Art and Future Directions. *ACM Computing Survey (CSUR)*, 28(4) :626–643, September 1996.

[DA90] W.J. Dally and H. Aoki. Adaptive Routing Using Virtual Channels. Technical report, MIT Laboratory for Computer Science, September 1990.

[Dar79] LA. Darringer. Application of Program Verification Techniques to Hardware Verification. In *Proc. IEEE-ACM Design Automation Conference*, pages 375–381, 1979.

[DBG+03] M. Dall'Osso, G. Biccari, L. Giovannini, D. Bertozzi, and L. Benini. Xpipes : a Latency Insensitive Parameterized Network-on-chip Architecture for Multi-Processor SoCs. In *ICCD*, pages 536–, 2003.

[DGRV00] M. Devillers, D. Griffioen, J. Rmijn, and F. Vaandrager. Verification of a Leader Election Protocol : Formal Methods Applied to IEEE 1394. *Formal Methods in System Design*, 16(3) :307–320, 2000.

[DS86] W.J. Dally and C.L. Seitz. The Torus Routing Chip. *J. Distributed Computing*, 1(3) :187–196, 1986.

[DS87] W.J. Dally and C.L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, C-36(5) :547–553, May 1987.

[DT01] W. J. Dally and B. Towles. Route packets, not wires : on-chip interconnection networks. In *Proceedings of the Design Automation Conference*, pages 684–689, Las Vegas, NV, 2001.

[DT04] W.J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan-Kaufmann Publisher, 2004.

[Dua93] J. Duato. A New Theory of Deadlock-Free Adaptive Routing in Wormhole Networks. *IEEE Transactions on Parallel and Distributed Systems*, 4(12) :1320–1331, 1993.

[Dua94] J. Duato. A Necessary and Sufficient Condition for Deadlock-Free Adaptive Routing in Wormhole Networks. In *International Conference on Parallel Processing*, pages 142–149, 1994.

[Dua96] J. Duato. A Necessary and Sufficient Condition for Deadlock-Free Routing in Cut-Through and Store-and-Forward Networks. *IEEE Transactions on Parallel and Distributed Systems*, 7(8) :841–854, 1996.

[Dum03] E. Dumitrescu. *Construction de modèles réduits et vérification symbolique de circuits industriels décrits au niveau RTL.* PhD thesis, Université Joseph Fourier, Grenoble, France, 2003.

[DZ83] J.D. Day and H. Zimmerman. The OSI Reference Model. *Proceedings of the IEEE*, 71 :1334–1340, 1983.

[EK03] E.A. Emerson and V. Kahlon. Rapid Parameterized Model Checking of Snoopy Cache Coherence Protocols. In H. Garavel and J. Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 144–159, Warsaw, Poland, April 2003. Springer-Verlag.

[EN96] E. A. Emerson and K. S. Namjoshi. Automatic Verification of Parameterized Synchronous Systems. In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 87–98, London, UK, 1996. Springer-Verlag.

[ES96] E. Allen Emerson and A. Prasad Sistla. Symmetry and Model Checking. *Formal Methods in System Design : An International Journal*, 9(1/2) :105–131, August 1996.

[FF98] E. Fleury and P. Fraigniaud. A General Theory for Deadlock Avoidance in Wormhole-Routed Networks. *IEEE Transactions on Parallel and Distributed Systems*, 9(7) :626–??, 1998.

[For02] M. Forsell. A Scalable High-Performance Computing Solution for Networks on Chips. *IEEE Micro*, 22(5) :46–55, 2002.

[Fox03] A.C.J. Fox. Formal Specification and Verification of ARM6. In D. Basin and B. Wolff, editors, *Thoerem Provers in Higher-Order Logics (TPHOLS'03)*, volume 2758 of *LNCS*, pages 24–40. Springer-Verlag, 2003.

[GDR05] K. Goossens, J. Dielissen, and A. Rădulescu. Æthereal Network on Chip : Concepts, Architectures, and Implementations. *IEEE Design and Test of Computers*, 22(5) :414–421, September-October 2005.

[Geo01] P. Georgelin. *Vérification formelle de systèmes digitaux synchrones, basée sur la simulation symbolique.* PhD thesis, Université Joseph Fourier, Grenoble, France, 2001.

[GG00] P. Guerrier and A. Greiner. A Generic Architecture for On-Chip Packet-Switched Interconnections. In *Proc. Design Automation and Test in Europe (DATE'00)*, pages 250–256, 2000.

[Gor87] M.J.C. Gordon. HOL : A Proof Generating System for Higher-Order Logic. In G. Birthwislte and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128, Boston, 1987. Kluwer Academic Publishers.

[Gre98] D. Greve. Symbolic Simulation of the JEM1 Microprocessor. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD '98)*, volume 1522, pages 321–333, Palo Alto, CA, 1998. Springer-Verlag.

[GRG+05] O.P. Gangwal, A. Rădulescu, K. Goossens, S. González Pestana, and E. Rijpkema. *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, volume 3 of *Philips Research Book*, chapter Building Predictable Systems On Chip : An Analysis of Guaranteed Communication in the Æthereal Network on Chip. Springer, 2005.

[GS92] S. M. German and A. P. Sistla. Reasoning About Systems With Many Processes. *Journal of the ACM*, 39(3) :675–735, 1992.

[GvMPW02] K. Goossens, J. van Meerbergen, A. Peeters, and P. Wielage. Networks on Silicon : Combining Best-Effort and Guaranteed Services. In *Proc. of Design Automation and Test in Europe Conference and Exhibition (DATE'02)*, pages 423–425, 2002.

[GVZ05a] B. Gebremichael, F. Vaandrager, and M. Zhang. Formal Models of Guaranteed and Best-Effort Services. Technical report, Institute for Computing and Information Sciences, Radboud University Nijmegen, March 2005.

[GVZ+05b] B. Gebremichael, F. Vaandrager, M. Zhang, K. Goossens, E. Rijpkema, and A. Rădulescu. Deadlock Prevention in the Æthereal protocol. In D. Borrione and W.J. Paul, editors, *Correct Hardware Design and Verification Methods (CHARME'05)*, volume 3725 of *LNCS*, pages 345–348, 2005.

[Hab69] A.N. Habermann. Prevention of System Deadlocks. *Communications of the ACM*, 12(7) :373–377&385, July 1969.

[Hav68] J.W. Havender. Avoiding deadlock in mutlitasking systems. *IBM Syst. J.*, 2 :74–84, 1968.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. An Experience in Proving Regular Networks of Processes by Modular Model Checking. *Acta Informatica*, 29(6-7) :523–543, November 1992.

[Hol72] R.C. Holt. Some Deadlock Properties of Computer Systems. *Computing Surveys*, 4(3) :179–196, September 1972.

[HQR98] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You Assume, We Guarantee : Methodology and Case Studies. In *CAV '98 : Proceedings of the 10th International Conference on Computer Aided Verification*, pages 440–451. Springer-Verlag, 1998.

[HS96] K. Havelung and N. Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In *Formal Methods Europe (FME'96)*, volume 1051 of *LNCS*. Springer-Verlag, 1996.

[Hun89] W. A. Hunt. Microprocessor Design Verification. *Journal of Automated Reasoning*, 5(4) :429–460, 1989.

[Jer02] A. A. Jerraya. *Conception de haut niveau des systèmes monopuces.* Lavoisier, 2002.

[K05] Communication personnelle avec M. Kaufmann.

[KK79] P. Kermani and L. Kleinrock. Virtual Cut-Through : A New Computer Communication Switch Technique. *Computer Network*, 3 :267–286, 1979.

[KM97a] M. Kaufmann and J Strother Moore. An Industrial Strengh Theorem Prover of a Logic Based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4) :203–213, April 1997.

[KM97b] M. Kaufmann and J Strother Moore. A Precise Description of the ACL2 Logic. available at http ://wwww.cs.utexas.edu/users/moore/publications/km97a.ps.gz, 1997.

[KM97c] M. Kaufmann and J Strother Moore. Structured Theory Development for a Mechanized Logic. *Journal of Automated Reasoning*, 26(2) :161–203, 1997.

[KM05] M. Kaufmann and J Strother Moore. How To Prove Theorems Formally. available at http ://www.cs.utexas.edu/users/moore/publications/how-to-prove-thms/index.html, 2005.

[KMM00a] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *ACL2 Computer Aided Reasoning : An Approach.* Klulwer Academic Press, 2000.

[KMM00b] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer Aided Reasoning : ACL2 Case Studies.* Klulwer Academic Press, 2000.

[KND02] K. Karim, A. Nguyen, and S. Dey. An Interconnect Architecture for Networking Systems On Chip. *IEEE Micro*, pages 36–45, September-October 2002.

[KNDR01] F. Karim, A. Nguyen, S. Dey, and R. Rao. On-Chip Communication Architecture for OC-768 Network Processor. In 38^{th} *Design Automation Conference (DAC'01)*, pages 678–683, 2001.

[Kra85] S. Krakowiac. *Principes des systèmes d'exploitation des ordinateurs.* DUNOD informatique, 1985.

[Lam05] L. Lamport. Real-Time Model Checking is Really Simple. In D. Borrione and W. Paul, editors, *Correct Hardware Design and Verification Methods (CHARME'05)*, volume 3725 of *LNCS*, pages 162–175. Springer, 2005.

[LH91] D.H. Linder and J.C. Harden. An Adaptive and Fault-Tolerant Wormhole Routing Strategy for k-ary n-cubes. *IEEE Transactions on Computers*, 40(1) :2–12, 1991.

[LST00] J. Liang, S. Swaminathan, and R. Tessier. asoc : A Scalable, Single-Chip Communications Architecture. In *IEEE PACT*, pages 37–46, 2000.

[Lyn96] N.A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., 1996.

[Mar94] J. Markoff. Circuit Flaw Causes Pentium to Miscalculate : Intel Admits. *New York Times*, 24th November 1994. Voir aussi http ://mathworks.com/company/pentium/index.html.

[MBV+02] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins. Interconnection Networks Enable Fine-Grain Dynamic Multi-tasking on FPGAs. In *FPL*, pages 795–805, 2002.

[McC62] John McCarthy. Towards a Mathematical Science of Computation. In *Proceedings of the Information Processing Cong 62*, pages 21–28, 1962.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Press, 1993.

[MCM+04] F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost. HERMES : an infrastructure for low area overhead packet-switching networks on chip. *Integration*, 38(1) :69–93, 2004.

[MLK98] J Strother Moore, T. Lynch, and M. Kaufmann. A Mechanically Checked Proof of the AMD5k86 Floating-Point Division Program. *IEEE Transactions on Computers*, 47(9) :913–926, September 1998.

[MMM+03] F. G. Moraes, A. Mello, L. Möller, L. Ost, and N. Calazans. A low area overhead packet-switched network on chip : Architecture and prototyping. In *VLSI-SOC*, pages 318–323, 2003.

[MNS99] Panagiotis Manolios, Kedar S. Namjoshi, and Robert Summers. Linking Theorem Proving and Model-Checking with Well-Founded Bisimulation. In *CAV '99 : Proceedings of the 11th International Conference on Computer Aided Verification*, volume 1633 of *LNCS*, pages 369–379. Springer-Verlag, 1999.

[MNT+04] M. Millberg, E. Nilsson, R. Thid, S. Kumar, and A. Jantsch. The Nostrum Backbone - a Communication Protocol Stack for Networks on Chip. In *VLSI Design*, pages 693–696, 2004.

[Moh98] P. Mohapatra. Wormhole Routing Techniques for Directly Connected Multicomputer Systems. *ACM Computing Surveys (CSUR)*, 30(3) :374–410, September 1998.

[Moo93] J Strother Moore. A Formal Model of Asynchronous Communications and Its Use in Mechanically Verifying a Biphase Mark Protocol. *Formal Aspects of Computing*, 6(1) :60–91, 1993.

[Moo98] J Strother Moore. Symbolic Simulation : An ACL2 Approach. In P. Windley G. Gopalakrishnan, editor, *Formal Methods in Computer Aided Design (FMCAD'98)*, volume 1522 of *LNCS*, pages 334–350. Springler-Verlag, 1998.

[MTCM05] A. Mello, L. Tedesco, N. Calazans, and F. Moraes. Virtual Channels in Networks on Chip : Implementation and Evaluation on Hermes NoC. In *SBCCI '05 : Proceedings of the 18th annual symposium on Integrated circuits and system design*, pages 178–183. ACM Press, 2005.

[NM93] L.M. Ni and P.K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *IEEE Computer Mag.*, 26(2) :62–76, February 1993.

[Nob02] J. P. Noblanc. EDA and Systems-on-Chip : A Key Challenge for MEDEA+. In *Proc. MEDEA+ Design Automation Conference*, Stresa, Italy, 23-25 oct 2002.

[NPW02] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL : A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS : A Prototype Verification System. In D. Kapur, editor, *Eleventh International Conference on Automated Deduction (CADE'92)*, volume 607 of *LNAI*, pages 748–752, Saragota, NY, June 1992. Springer-Verlag.

[Pau89] L.C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5 :363–397, 1989.

[Pau90] L.C. Paulson. Isabelle : the next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[Pau98] L.C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6 :85–128, 1998.

[PMMG04] L. Pike, J. Maddalon, P. Milner, and A. Geser. Abstractions for Fault-Tolerant Distributed System Verification. In *Thoerem Proving in Higher-Order Logics (TPHOLS'03)*, volume 3223 of *LNCS*, pages 257–270. Springer-Verlag, 2004.

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *Eighteenth Annual IEEE Symposium on Foundations in Computer Science*, pages 46–57, 1977.

[QS82] J.P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *5^{th} Colloqium on International Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer-Verlag, 1982.

[RDP+05] A. Rădulescu, J. Dielissen, S. González Pestana, O.P. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens. An Efficient On-Chip Network Offering Guaranteed Services, Shared-Memory Abstraction, and Flexible Network Programming. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 24(1), 2005.

[RGR+03] E. Rijpkema, K. Goossens, A. Rădulescu, J. Dielissen, J. Van Meerbergen, P. Wielage, and E. Waterlander. Trade Offs in the Design of a Router with both Guaranteed and Best-Effort Services for Networks on Chip. In *Proc. of Design Automation and Test in Europe Conference and Exhibition (DATE'03)*, pages 350–355, 2003.

[RKM03] W. A. Hunt R. Krug and J Strother Moore. Linear and Nonlinear Arithemetic in ACL2. In D. Geist and E. Tronci, editors, *Correct Hardware Design and Verification Methods (CHARME'03)*, volume 2860 of *LNCS*, pages 51–65, L'Aquila, Italy, October 2003. Springer-Verlag.

[RMK03] A. Roychoudhury, T. Mitra, and S.R. Karri. Using Formal Techniques to Debug the AMBA System-on-Chip Bus Protocol. In *Design Automation and Test Europe (DATE'03)*, pages 828–833, 2003.

[RSV97] J.A. Rowson and A. Sangiovanni-Vincentelli. Interface-Based Design. In *34^{th} Design Automation Conference (DAC'96)*, pages 178–183, 1997.

[Rus98] D. Russinoff. A Mechanically Checked Proof of IEEE Compliance of a Register Transfer Level Specification of the AMD-K7 Floating-Point Multiplication, Division and Square Root Instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1 :148–200, December 1998.

[Sam05] G. Al Sammane. *Simulation symbolique des circuits décrits au niveau algorithmique*. PhD thesis, Université Joseph Fourier, 2005.

[SAN03] I. Saastamoinen, M. Alho, and J. Nurmi. Buffer implementation for Proteo networks-on-chip. In *International Symposium on Circuits and Systems (ISCAS'03)*, pages 113–116, 2003.

[Saw99] J. Sawada. *Formal Verification of an Advanced Pipelined Machine*. PhD thesis, University of Texas at Austin, Texas, USA, 1999.

[SB03a] J. Schmaltz and D. Borrione. Formalization and Verification of the AMBA AHB Communication Architecture Using the ACL2 Theorem Prover. In *6^th IEEE International Workshop on Design and Diagnostics of Electronic Circtuits and Systems*, pages 93–100, Poznań, Poland, April 2003.

[SB03b] J. Schmaltz and D. Borrione. Formalization and Verification of the AMBA AHB Communication Architecture Using the ACL2 Theorem Prover. Research Report TIMA-RR-03/07-03-FR, TIMA Laboratory, 2003.

[SB03c] J. Schmaltz and D. Borrione. Verification of a Parameterized Bus Architecture Using ACL2. In *Proceedings of the Fourth International Workshop on the ACL2 Theorem Prover and its Applications*, April 2003.

[SB04a] J. Schmaltz and D. Borrione. A Functional Approach to the Formal Specification of Networks on Chip. In A.J. Hu and A.K. Martin, editors, *Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *LNCS*, pages 52–66, Austin, Tx, USA, November 2004. Springer-Verlag.

[SB04b] J. Schmaltz and D. Borrione. A Functional Specification and Validation Model for Networks on Chip in the ACL2 Logic. In *Proceedings of the Fith International Workshop on the ACL2 Theorem Prover and its Applications*, November 2004.

[SB05] J. Schmaltz and D. Borrione. A Generic Network on Chip Model. In T. Melham and J. Hurd, editors, *Theorem Proving in Higher Order Logics (TPHOLs'05)*, volume 3603 of *LNCS*, pages 310–325, Oxford, UK, August 2005. Springer-Verlag.

[SBO+03] G. Al Sammane, D. Borrione, P. Ostier, J. Schmaltz, and D. Toma. Combining ACL2 and Mathematica for the Symbolic Simulation of Digital Sytems. In *Proceedings of the Fourth International Workshop on the ACL2 Theorem Prover and its Applications*, April 2003.

[Sch04] J. Schmaltz. Functional Specification and Validation of the Octagon Network on Chip using the ACL2 Theorem Prover. Research Report TIMA-RR–04/01-02–FR, TIMA, 2004.

[SG02] J. Sawada and R. Gamboa. Mechanical Verification of a Square Root Algorithm Using Taylor's Theorem. In *Formal Methods in Computer Aided Design (FMCAD '02)*, LNCS, pages 274–292. Springer-Verlag, 2002.

[SJ95] L. Schwiebert and D.N. Jayasimha. A Universal Proof Technique for Deadlock-Free Routing in Interconnection Networks. In *7^th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'95)*, pages 175–184, 1995.

[SOSC01] N. Shankar, S. Owre, and D.W.J. Stringer-Calvert. PVS Prover Guide. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, December 2001.

[Spi04] G. Spirakis. Beyond Verification : Formal Methods in Design. In A. Hu and A.K. Martin, editors, *Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *LNCS*, Austin, Texas, USA, November 2004. Springer-Verlag. Invited Speaker.

[SSB04] G. Al Sammane, J. Schmaltz, and D. Borrione. Formal Design and Verification of On Chip Networking. In *1st International Conference on Information & Communication Technologies : From Theory to Applications (ICTTA'04)*, 2004.

[SSB05] J. Schmaltz, G. Al Sammane, and D. Borrione. Formal Design and Verification of Networks on Chip. Research Report TIMA-RR–05/12-01–FR, TIMA, 2005.

[SSM+01] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. M. Rabaey, and A. L. Sangiovanni-Vincentelli. Addressing the System-on-a-Chip Interconnect Woes Through Communication-Based Design. In *DAC*, pages 667–672, 2001.

[SST+04] G. Al Sammane, J. Schmaltz, D. Toma, P. Ostier, and D. Borrione. Theosim : Combining Symbolic Simulation and Theorem Proving for Hardware Verification. In *Seventeenth Symposium on Integrated Circuits and Systems Design (SBCCI'04)*, Pernambuco, Brazil, September 2004. ACM.

[STSB03] G. Al Sammane, D. Toma, J. Schmaltz, and D. Borrione. Constrained Symbolic Simulation using Mathematica and ACL2. In *Correct Hardware Design and Verification Methods (CHARME'03)*, volume 2860 of *LNCS*, L'Aquila, Italy, October 2003. Springer-Verlag.

[Tan96] A.S. Tanenbaum. *Computer Networks*. Prentice-Hall, 1996.

[Tur49] Allan M. Turing. Checking a Large Routing. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, Cambridge, England, June 1949. University Mathematical Laboratory.

[WGH01] M. Wilding, D. Greve, and D. Hardin. Efficient Simulation of Formal Processor Models. *Formal Methods in System Design : An International Journal*, 2001.

[Wol04] W. Wolf. The Future of Multiprocessor Systems-on-chips. In *Design Automation Conference (DAC'04)*, pages 681–685, 2004.

[YBM03] T. Tao Ye, L. Benini, and G. De Micheli. Packetized On-Chip Interconnect Communication Analysis for MPSoC. In *DATE '03 : Proceedings of the conference on Design, Automation and Test in Europe*, pages 344–349. IEEE Computer Society, 2003.

[YS02] J. Yang and C.I. Seger. Generalized Symbolic Trajectory Evaluation Abstraction in Action. In *Formal Methods in Computer-Aided Design (FMCAD'02)*, LNCS 2517, pages 70–87, Portland, Or,USA, nov 2002. Springer.

Table des figures

Résumé

Cette thèse présente un modèle formel représentant *toute* architecture de communication sur la puce. Ce modèle est mathématiquement décrit par une fonction nommée *GeNoC*. La correction de *GeNoC* est exprimée par un théorème montrant que tout message émis atteint sa destination sans modification de l'information qu'il transporte. Le modèle identifie les composantes communes à toute architecture et leurs propriétés essentielles, à partir desquelles est déduite la preuve du théorème sur *GeNoC*. Chaque composante est représentée par une fonction sans définition *explicite*, mais *contrainte* de satisfaire ses propriétés essentielles. Ainsi, la validation de toute architecture *particulière* consiste en la preuve que les définitions concrètes de ses composantes satisfont les propriétés essentielles. En pratique, ce formalisme a été réalisé dans la logique du démonstrateur de théorèmes ACL2. Une méthodologie associée au modèle fournit un support systématique pour la spécification et la validation des architectures de communication sur la puce à un haut niveau d'abstraction. Pour valider notre approche, nous avons exhibé différentes architectures constituant autant de concrétisations du modèle générique *GeNoC*. Ces concrétisations comprennent notamment des systèmes industriels, comme le bus AMBA AHB ou le réseau Octagon de ST Microelectronics.

Mots clés : systèmes sur puce, architectures de communication, vérification formelle, démonstration automatique de théorèmes.

Abstract

This thesis presents a formal model that represents *any* on-chip communication architecture. This model is described mathematically by a function, named *GeNoC*. The correctness of *GeNoC* is expressed as a theorem, which states that messages emitted on the architecture reach their expected destination without any modification of their content. The model identifies the key constituents common to *all* communication architectures and their essential properties, from which the proof of the *GeNoC* theorem is deduced. Each constituent is represented by a function, which has no *explicit* definition, but that is constrained to satisfy the essential properties. Thus, the validation of a *particular* architecture is reduced to the proof that its concrete definition satisfies the essential properties. In practice, the model has been defined in the logic of the ACL2 theorem proving system. We defined a methodology that yields a systematic approach to the validation of communication architectures at a high level of abstraction. To validate our approach, we exhibit several architectures that constitute concrete instances of the generic model *GeNoC*. Some of these applications come from industrial designs, such as the AMBA AHB bus or the Octagon network from ST Microelectronics.

Keywords : systems on chip, communication architectures, formal methods, automated theorem proving.