Mise en œuvre de techniques de démonstration automatique pour la vérification formelle des NoCs

A. Helmy

Institut Polytechnique de Grenoble

N° attribuée par la bibliothèque
|978-2-84813-152-8|

THESE

pour obtenir le grade de
DOCTEUR DE L'Institut polytechnique de Grenoble
Spécialité : Micro et Nano Électronique

préparée au laboratoire **TIMA**
dans le cadre de **l'École Doctorale « Électronique, Électrotechnique, Automatique et Traitement du Signal »**

présentée et soutenue publiquement par
Amr HELMY
le 30 Avril 2010

Titre :

Mise en œuvre de techniques de démonstration automatique pour la vérification formelle des NoCs

Directrice de thèse : Prof. Laurence PIERRE

JURY

Dominique BORRIONE, Pr. Université Joseph Fourier	Présidente
Michaël RUSINOWITCH, DR INRIA Lorraine	Rapporteur
Lionel TORRES, Pr. Université Montpellier 2	Rapporteur
Laurence PIERRE, Pr. Université Joseph Fourier	Directrice de thèse
Patrick BELLOT, Pr. Télécom ParisTech	Examinateur
Dominique HOUZET, Pr. Institut Polytechnique de Grenoble	Examinateur

Remerciments

En premier lieu, je remercie Michaël Rusinowitch et Lionnel Torres pour avoir accepté la tâche de rapporteur. Je remercie également les membres du jury Dominique Borrione, Patrick Bellot, et Dominique Houzet.

Cette thèse n'aurait jamais vu le jour sans Laurence Pierre, ma directrice de thèse. Je la remercie pour sa compréhension, son support et son guidage. Elle m'a toujours poussé à aller plus loin depuis l'époque où elle était responsable de mon stage de master de recherche. Laurence a su m'apprendre quand être exigeant avec moi même et surtout me forcer à être "organisé" dans ma vie et dans ma réflexion. Je remercie Dominique Borrione pour son encouragement durant toute la période que j'ai passé à Grenoble. Je les remercie toutes les deux pour la confiance qu'elles m'ont accordée, leurs encouragements, et leur soutien sur le plan professionnel et personnel.

Je remercie Julien Schmaltz pour ses conseils et suggestions ainsi que la clarté de son manuscrit et de son code. Un grand merci aux membres de l'équipe VDS pour leur soutien scientifique, humour, critiques, écoutes et de m'avoir supporté durant la période de rédaction, je sais que j'étais insupportable : Alexandre, Eric, Florent, Jérôme, Katel, Luca, Renaud, et Yann. Je remercie Cédric pour son modèle LaTeX. Je remercie également tous mes collègues chercheurs du laboratoire Tima, ce fut une période agréable de ma vie (je ne citerai personne pour ne pas en oublier). J'ai appris chaque jour une nouvelle chose : scientifique ou culturelle. Un grand merci aux personnels administratifs du laboratoire, sans qui je me serai perdu dans la vie "administrative".

Un grand merci à tous mes proches et mes amis pour leurs blagues, les sorties et les soirées. Un merci spécial à Maëva d'avoir été toujours là pour moi, j'en suis reconnaissant. Merci aux membres du Hockey Club Grenoble les entrainements et les matchs étaient un souffle d'air nécessaire pour reprendre des forces. Merci de m'avoir accepté dans votre équipe et m'avoir permis de jouer au Hockey à la "grenobloise" : ce fut un honneur de jouer avec vous.

Enfin merci à ma mère pour son soutien, et ses encouragements, c'est grâce à elle que je suis arrivé au bout de mon rêve de doctorat. Je remercie mon père pour tout ce qu'il m'a appris. Je remercie mes grand-pères de m'avoir fourni un modèle à suivre dans ma vie. Je regrette qu'un seul a la chance de me voir Docteur.

La présence de toutes ces personnes mentionnées ci-dessus dans ma vie, dans les moments de doute avant les moments de bonheur, m'a aidé à prendre des décisions que je ne regretterai jamais.

Encore une fois merci à tous.
Amr Helmy

Le but grand de toute la science est de couvrir le plus grand nombre de faits empiriques par déduction logique du plus petit nombre d'hypothèses ou d'axiomes.
Humphrey Davy

Table des matières

Première partie

Introduction

Chapitre 1

Introduction et motivations

Compte tenu de l'importance des applications des circuits micro-électroniques, la conception correcte de ces systèmes s'avère indispensable. Le travail présenté dans ce manuscrit fait partie de l'effort de recherche international dont le but est la vérification d'une classe spécifique de circuits : les *systèmes-sur-puces*.

Les *SoCs (systems-on-chip)* sont passés d'une version simple contenant un processeur et le minimum de mémoire nécessaires (les micro-contrôleurs) à des solutions plus élaborées qui sont le fruit de l'intégration de différents composants hétérogènes d'un système électronique entier sur une même puce : des processeurs, de la mémoire, circuit RF, des interfaces analogiques/digitales, etc. Les domaines d'application sont nombreux : aviation, automobile, géo-positionnement par satellite, lecteurs multimédia, téléphonie portable, etc. Les performances requises ne cessent d'augmenter : 10 MOPS [1] pour les applications audio jusqu'à 3 GOPS [2] pour des systèmes de vidéo [BB05]. Selon le *ITRS*[3], les SoCs du futur auront des centaines d'éléments de traitement construits avec plus de 4 milliards de transistors et fonctionneront à 10GHz [Dua03]. Les systèmes sur puces proposent une méthodologie très efficace et attractive pour créer des circuits aussi performants dans les délais les plus courts grâce à l'utilisation de blocs préconçus (*IP : intellectual property*) comme des processeurs, des mémoires, interconnectés à l'aide d'un système de communication (figure 1.1).

Les performances des processeurs ne cessent d'augmenter : les processeurs ARM2 avaient une fréquence d'opération de 8MHz en 1986, le processeur StrongARM fonctionnait à 233MHz en 1996. Intel se base sur l'architecture StrongARM pour créer Xscale avec une fréquence de 1,25 GHz. Intel commercialise en 2009 un cœur de processeur opérant à une fréquence de 3,3 GHz qui sera utilisé dans un processeur de 6 cœurs début 2010, et une micro architecture "Larrabee" avec un nombre de cœurs entre 10 et 100 [4]. L'entreprise dévoile en décembre 2009 un processeur expérimental de 48 cœurs [5]. AMD lance HYDRA, une révision de son "*Shanghai CPU*", qui supportera jusqu'à 8 cœurs, avec 6MB L3 de cache, et 1MB L2 de cache par cœur. Avec de tels systèmes, les temps de

1. Millions of Operations Per Second
2. Giga Operations Per Second
3. International Technology Roadmap for Semiconductors
4. http ://www.intel.com/technology/visual/microarch.htm
5. http ://www.intel.com/pressroom/archive/releases/2009/20091202comp_sm.htm

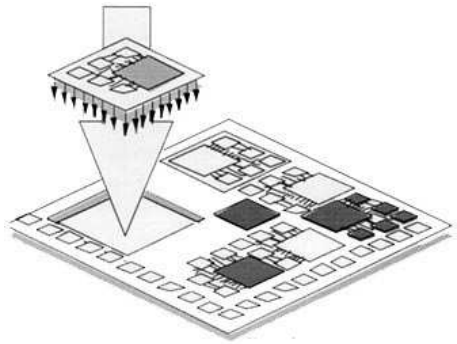

FIGURE 1.1 – Utilisation des IPs pour la création des systèmes

source :http ://www.dolphin.fr/corporate/investors/investors_design.html

calcul deviennent inférieurs aux temps de communication [BM06]. Afin de construire des systèmes rapides et efficaces, les dispositifs de communication ont eux aussi évolué pour répondre à ce problème [LRD01]. Des connexions point-à-point sont utilisées entre les différents composants comme première solution. L'utilisation d'un bus était l'étape logique suivante. Les bus sont des architectures de communication communes avec une politique d'arbitrage pour résoudre les conflits entre les parties communicantes. Les bus sont faciles à construire et à comprendre, et tout IP peut y être connecté. Mais avec l'augmentation du nombre de composants, les bus deviennent problématiques : des problèmes d'arbitrage, et des problèmes de capacitance apparaissent, qui plus est les arbitres nécessitent une personnalisation spécifique à l'application développée. D'autres solutions ont été proposées pour remédier à ces problèmes. Le bus *AMBA* est un bus segmenté développé par ARM pour l'interconnexion de ses cœurs de processeur [ARM99]. Il différencie deux bus système à haute performance (*AHB* : AMBA high-speed bus et *ASB* : Advanced system bus) et un bus de périphériques (*APB* : Advanced peripheral bus). Diverses variantes ont été proposées. Le *multi-layer AHB* [ARM01] permet des communications parallèles entre différents maîtres et esclaves à l'aide d'une matrice d'interconnexion. *AMBA AXI* [ARM03] est une fabrique de communication dont le maître et l'esclave sont découplés. Une interface est définie pour les maîtres et une autre symétrique est définie pour les esclaves. Des solutions avec des bus hiérarchisés peuvent être utilisées. Ce type de réseau utilise une structure arborescente de bus pour acheminer les communications [MADHRW00]. L'architecture *COMA* est une des propositions utilisant ce principe afin de diminuer le temps d'accès à la mémoire cache, ce qui améliore les performances du système [HS95]. [Mah94] compare les performances des systèmes avec des bus hiérarchisés et les bus classiques.

Sans aucun doute, parmi les applications phares de la micro-électronique se comptent celles du multimédia sur systèmes mobiles. Systèmes déjà difficiles à implémenter, le fameux *Time-To-Market* ajoute des contraintes sur le temps de développement. De nombreux standards et algorithmes de traitement, nécessitant des systèmes largement parallélisés, sont utilisés dans de telles applications. Dans les dernières années, l'introduction des *MPSoCs (Multi Processors Systems-on-Chip)* a apporté une solution qui permet la création de telles applications dans le moins de temps possible. Plusieurs exemples industriels existent : le *CELL* [GHF+06] développé par IBM, Toshiba et Sony, et l'architecture *Davinci* de Texas instruments [Yos05]. Le besoin en communications efficaces s'accroît, spécialement dans les systèmes massivement parallèles *(MP²SoCs : Massively Parallel*

Multi Processors Systems-on-Chip) [KPP06] [PFT+07]. Des besoins de broadcast et de multicast apparaissent. La notion de QoS (*Quality of service*) est indispensable pour tout SoC multimédia. La quantité de données échangées entre les différents composants d'un système rend difficile l'utilisation de connexions de type bus et point-à-point.

Des systèmes d'interconnexion ont existé dans l'histoire de l'informatique afin de construire des machines multiprocesseurs pour les applications largement parallélisées. Des architectures de type grille et tore notamment ont été utilisées. *Solomon*, *Illiac* utilisaient des grilles et des tores. *AmetekS14*, *Cosmic Cube* et *nCube* utilisaient un hypercube. Ces réseaux n'étaient pas sur la puce, ils connectaient différents processeurs au sein d'une machine. Dans [DL99], l'implémentation d'un système avec 64 processeurs et leurs mémoires sur une seule puce est prévue avec l'aide d'un système de communication par commutation de paquet. Avec la mise en œuvre de plusieurs processeurs sur le même substrat de silicium, la question suivante se pose : ce type d'architecture de communication est-il adaptable sur la puce ? Les réseaux sur la puce *(NoC : Network on Chip)* peuvent apporter une solution aux problèmes causés par les limitations des bus [BDM02]. Un réseau sur puce est une fabrique de communication multi-hop avec un système de commutation de paquet, dans la plupart des cas intégrée sur la puce [BB05][DT03]. Les composants sont connectés au réseau à travers des connexions point-à-point à l'aide d'une interface de réseau *(NI : network interface)*. Des standards et des protocoles pour les NIs ont été proposés pour simplifier la création de systèmes communicants sur la puce. SCI (Scalable Coherent Interface) [IEE92] [HR99] a été initialement prévu pour les bus. OCP (Open Core Protocol) [WD00] est un autre exemple de tels protocoles. Plusieurs fabriques d'interconnexion existent déjà : Hermes [MCM+04], Xpipes [BB04], SonicsMX [6], Nostrum [KJM+02], SPIN [AG03], et Spidergon [CLM+04] en sont des exemples. Un système permettant de créer un réseau sur puce paramétré NetMaker [BMM07] a été élaboré sous forme de bibliothèque de composants.

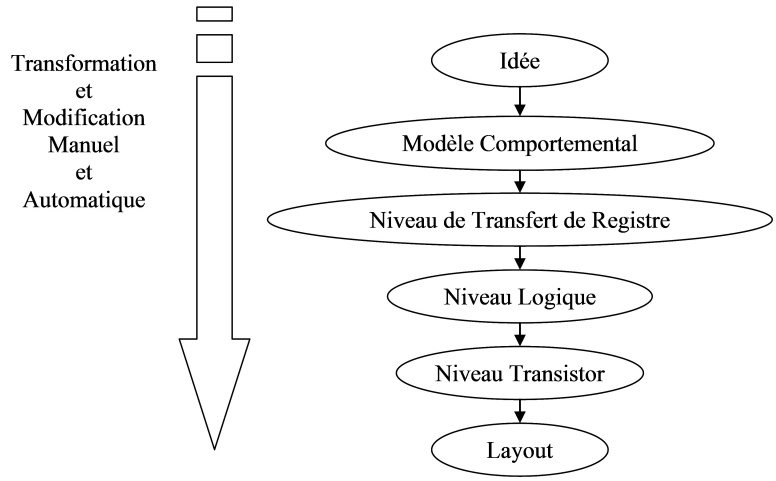

FIGURE 1.2 – Flot de conception matériel

Sur le marché des produits micro-électroniques, il est indispensable de proposer des

6. http ://www.sonicsinc.com/sonicsMX.htm

produits innovants et performants dans les plus brefs délais pour exploiter la fenêtre de commercialisation. Le système créé doit respecter les spécifications attendues. Avec l'augmentation de la complexité des systèmes, de plus en plus de temps est consacré à la vérification (jusqu'à 70% [Nob02]). Dans le flot de conception matériel (figure 1.2), le passage d'un niveau au suivant est susceptible d'introduire des erreurs de conception. Même avec le support d'un large éventail d'outils, ce passage n'est pas toujours sans problème. Une phase de vérification élaborée doit être réalisée. Nous avons vu qu'un SoC regroupe des blocs fonctionnels (IPs) reliés au moyen d'un système d'interconnexion, qui pourra être un NoC [MPCJ08]. Les IPs doivent avoir été vérifiés lors de leur conception. L'aspect crucial du point de vue de la validation reste donc celui du bon fonctionnement des communications. Cette thèse se place dans le contexte de la vérification des infrastructures de communication de type NoC.

1.1 Vérification des réseaux sur puces

Le but des travaux effectués durant cette thèse est la vérification des communications dans les SoCs. [OHM05] expose les challenges majeurs dans la conception des NoCs. [Goo05] évoque quelques aspects de la vérification des réseaux où les méthodes formelles seront utiles.

Les études consacrées aux NoCs se sont majoritairement concentrées sur les performances [PDMG+05] [BM06] [SSM+01], la latence [Jan06], la bande passante [MNTJ04], l'estimation de consommation [NS01], la détection et la correction d'erreurs [GIS+06] [MDMB+05], et la surface utilisée. D'autres proposent des méthodes de routage tendant à éviter des problèmes de deadlock [SHZG05] [GVZ+05] [MMA+06] en visant la caractérisation du trafic [LRD01].

En ce qui concerne la vérification de bon fonctionnement, des environnements ont été proposés pour la simulation ou l'émulation de réseaux sur la puce [RAP+05], de même que pour des solutions orientées test [PO07] [ABC+05]. De telles approches ne permettent pas une vérification exhaustive, et peuvent être complémentées par des méthodes de preuve formelle. Il est souhaitable que celles-ci permettent un raisonnement paramétré sur certaines données comme la taille du réseau, la longueur des messages,... Les méthodes formelles peuvent être classifiées en deux grandes catégories : la démonstration des théorèmes et la vérification des modèles (*model checking*).

La démonstration des théorèmes. Les démonstrateurs sont des outils généraux qui procèdent par déduction à partir d'axiomes et de règles d'inférence pour une logique donnée. Nous pouvons citer ACL2 [KMM02], HOL [GM93], Isabelle[7], Lambda [Lim90] , PVS[OSR93] et Coq [BC04].

La vérification des modèles. Proposée par Clarke et Emerson [CE81] et par Queille et Sifakis [QS82], la méthode consiste à vérifier si un modèle donné, le système ou une abstraction de celui-ci, satisfait une spécification, souvent en terme de logique temporelle. L'intérêt majeur de cette méthode est sa totale automatisation. La limitation principale

7. http ://isabelle.in.tum.de/

est l'explosion exponentielle de nombre d'états avec la taille et le nombre des registres du modèle. Divers model-checkers sont proposés commercialement (IBM, Intel, Cadence,...).

Un des points forts des démonstrateurs des théorèmes par rapport aux outils de type model-checkers est que l'on peut raisonner sur un circuit vu sur plusieurs niveaux d'abstraction. Ils sont également très efficaces pour les raisonnements sur les systèmes orientés traitements de données (*data-path dominated systems*). Néanmoins, leur utilisation nécessite un utilisateur expérimenté.

Une grande quantité de travaux s'est intéressée à l'utilisation des méthodes formelles pour vérifier des systèmes de communications et leur protocoles. La plupart utilisent le model-checking, ou une composition de celui-ci et de démonstration des théorèmes. Les travaux de Clarke et al, publiés dans [CGJ97], permettent de vérifier des propriétés temporelles de réseaux paramétrés en anneau et en arbre binaire. Une première étape consiste à utiliser une grammaire de réseau sans contexte pour modéliser les systèmes de communication. Ensuite des propriétés temporelles sont vérifiées à l'aide d'un model-checker. Dans [Amj04], Amjad utilise un model-checker implémenté dans HOL pour vérifier les protocoles AMBA APB et AHB. Bharadwaj et al. vérifient un protocole de diffusion dans un réseau en arbre binaire à l'aide du model-checker SPIN et du démonstrateur Coq [BFS95]. Dans [Cur94], Curzon développe un modèle structurel du commutateur *ATM Fairsile* et le compare à sa spécification comportementale à l'aide de HOL. L'absence de deadlock dans le réseau Æthereal a été vérifiée par Gebremichael et al. en utilisant l'outil PVS [GVZ+05].

Tous ces travaux portent sur la vérification d'un protocole spécifique. Des approches orientées vers des modèles plus généraux ont été proposées dans [Rus99], [Pik07], et [MGPM04]. Dans [Moo93], Moore utilise le démonstrateur Boyer-Moore pour créer un modèle formel *générique* pour les communications asynchrones. Il démontre l'utilisation de ce modèle pour prouver la conformité du protocole Bi-phase. Un travail plus général est exposé dans [HB05]. Un modèle générique de protocole est proposé avec les opérateurs et les conditions nécessaires pour passer d'une couche à l'autre de ce protocole.

Quelques travaux basés sur des méthodes (semi-) formelles ont aussi été proposés. Ils visent essentiellement la détection et le debug des défaillances. Chenard et al proposent dans [CBA+07] d'intégrer des vérificateurs d'assertions PSL [IEE05] synthétisés à l'aide de l'outil MBAC [BZ08] dans un réseau sur puce. Ils illustrent la méthode sur un NoC à anneaux hiérarchiques décrit dans [BCZ06]. Les propriétés PSL décrivent des obligations à satisfaire au niveau des interfaces. En cas d'échec d'une de ces propriétés, des paquets spéciaux sont propagés vers une station dédiée pour les analyser. Dans [GVVSB07], Goossens et al. présentent une méthode centrée sur les communications, qui surveille les interactions entre les IPs. Le système d'interconnexion est debogué de cette façon. Ils définissent deux fabriques d'interconnexion, une dédiée aux données de l'application, l'autre pour celles du debug. Le standard IEEE1149.1 (*boundary test*) est utilisé. Néanmoins, des changements dans les NIs sont nécessaires, pénalisant ainsi la surface utilisée.

Dans les travaux présentés ci-dessus, la phase de vérification intervient après que tous les choix d'implémentation ont été faits. Des méthodes plus génériques, qui visent à la vérification des réseaux dans les phases initiales de la conception, sont nécessaires. Les travaux

préconisés dans ce manuscrit traitent cette problématique en se basant sur des travaux déjà menés dans l'équipe VDS [8] du laboratoire TIMA. Un méta-modèle pour les architectures de communication sur la puce implémenté dans le démonstrateur de théorème ACL2 a été présenté dans [Sch06]. Baptisé *GeNoC*, et situé au niveau de la couche réseau du modèle ISO-OSI, le modèle identifie les constituants principaux communs à toute architecture de communication (topologie, technique de routage et de commutation) ainsi que des propriétés essentielles *(obligations de preuve)* à partir desquelles la correction du système peut être déduite par preuve formelle. Le théorème de correction globale stipule que tout message arrivé atteint la bonne destination sans modification de son contenu. La taille du réseau ainsi que la taille des messages font partie des paramètres de la preuve.

La principale limitation de ce modèle est que la granularité d'une communication est le voyage entier (transfert de la source à la destination), interdisant de prendre en compte les étapes intermédiaires (hops) et donc de modéliser certains concepts, comme par exemple des algorithmes de routage adaptatifs. Cette caractéristique permettait de s'affranchir de la représentation du temps et de l'état du réseau durant les communications, mais impliquait beaucoup trop de limitations, comme le simple fait de ne considérer que des départs des messages simultanés.

1.2 Contribution et Organisation du Manuscrit

La première version du modèle (décrite succinctement ci-dessus) était un premier pas vers l'obtention d'une théorie générique pour la vérification des réseaux sur la puce. Cependant, ses limitations pénalisaient la précision des résultats obtenus. Le méta-modèle présenté dans ce manuscrit est une extension du modèle *GeNoC* initial afin de remédier à ses restrictions.

La granularité du mouvement dans le modèle était le trajet entier, un message part de sa source pour atteindre sa destination immédiatement. Cette granularité donnait des résultats finaux partiellement corrects. Les résultats reflétaient correctement les messages arrivés, cependant ils n'étaient pas forcement ceux obtenus dans les réseaux lors d'une simulation au niveau RTL [9]. La route entière d'un message devait être calculée empêchant ainsi la considération des algorithmes adaptatifs (non-minimaux plus spécifiquement), à cause de l'impossibilité de calculer toutes les routes possibles entre une source et une destination. En outre, un message réservait sa route entière empêchant ainsi un autre de partir même si cette contrainte n'existait as dans la réalité. La modification du modèle afin de permettre un déroulement en étape (pas-à-pas, un pas étant le saut d'un nœud à un autre) a permis de remédier à ce problème de granularité pour obtenir des résultats plus fidèles et précis comme ceux obtenus par la simulation d'un réseau. Les messages se déplacent maintenant d'un pas à la fois (*network hop*) dans *un* cycle d'exécution. Les cycles d'exécution constituent une modélisation explicite du temps qui a été ajoutée au modèle.

8. Verification & Modeling of Digital Systems
9. Register Transfer Level

Qui plus est, l'ordre dans lequel les messages étaient considérés par le modèle, influençait le résultat obtenu. Les priorités entre les messages ainsi que plusieurs autre aspects n'apparaissaient pas dans le modèle. Une extension importante consistait à prendre en compte les couches transport et liaison de données du modèle ISO-OSI (section 2.1). Cette étape a permis la modélisation de plusieurs aspects importants qui existent dans les réseaux sur la puce : contrôle de flux, les priorités entre les messages, les protocoles d'échange point-à-point, le contrôle d'accès au réseau et les priorités dans les choix des routes à suivre.

L'absence d'une modélisation explicite du temps était la cause principale de l'hypothèse que tous les messages étaient injectés dans le réseau à l'instant initial. La modélisation du temps à l'aide de cycles d'exécution a permis d'enlever cette hypothèse. La nouvelle version permet le départ des messages à tout instant.

L'absence d'une modélisation explicite de l'état du réseau empêchait l'utilisateur de représenter les buffers et les ports sur un nœud. L'introduction d'un tel module a été faite. Son utilisation pour la détection des deadlocks et des livelocks est envisageable dans le futur.

La propriété de correction du modèle correspondait à s'assurer que les messages arrivés n'était pas modifiés par le réseau et qu'ils parvenaient à leur bon destinataire. Une nouvelle propriété a été ajoutée qui garantie l'absence de perte.

Dans ce contexte, les travaux présentés ici, contrairement aux travaux existant, fournissent une méthodologie générique de vérification formelle ciblée vers les réseaux dans les phases initiales de la conception des systèmes sur la puce. Néanmoins, quelques limitations existent encore : preuve de l'absence de deadlock, livelock et famine dans un réseau. Des extensions, traitant ces points, sont encore nécessaires pour obtenir une théorie générique générale portant sur les infrastructures de communications sur la puce. [VS09] propose une extension du modèle présenté dans ce manuscrit traitant l'absence de deadlock. En outre, le modèle traite les réseaux à un niveau d'abstraction très élevé. Un effort de recherche doit être consacré afin de pouvoir s'approcher du niveau RTL.

Organisation du manuscrit. Le manuscrit est divisé en trois partie :

- *Introduction* : Dans le chapitre 2, les principaux concepts des architectures de communication seront exposés. La présentation du modèle *GeNoC* de départ [Sch06], se trouve dans le chapitre 3.
- *Vérification Formelle des communications sur la puce :* le modèle actuel, résultant des travaux de la thèse, ainsi qu'une brève introduction de ACL2 seront présentés dans le chapitre 4.
- *Expérimentation :* le chapitre 5 exposera les cas d'études considérés et l'application de notre modèle. Le chapitre 6 conclura ce manuscrit avec une discussion des prolongations possibles des travaux.

Chapitre 2

Les réseaux et systèmes de communication

Le but de ce chapitre est d'introduire le vocabulaire et les principes relatifs aux systèmes de communication. Nous montrerons les protocoles, les architectures ainsi que les algorithmes utilisés dans les réseaux sur la puce. La plupart des méthodes utilisées dans les systèmes de communication sur la puce proviennent des réseaux d'interconnexion utilisés dans les machines multiprocesseurs et des réseaux informatiques classiques. Les techniques de routage, de commutation et de contrôle de flux des réseaux classiques ne peuvent pas toujours être utilisées telles quelles à cause des limitations de ressources et de performances requises. Nous présentons ces notions générales, et certaines de leurs adaptations. Finalement, dans la section 2.6 nous décrirons trois réseaux sur puce qui seront utilisés comme cas d'étude pour l'évaluation de notre modèle.

2.1 Modèle de référence ISO-OSI

Un *protocole* de communication est l'ensemble des règles nécessaires pour accomplir une communication entre deux interlocuteurs. Un moyen de transport physique pour les données est indispensable entre les parties communicantes : le *médium de transmission*. Les informations seront transmises sous forme de séquences de signaux, *trames*.

Avec des systèmes de communication qui peuvent connecter des entités avec des normes de fonctionnement différentes, le besoin d'un standard de protocoles s'est imposé. Développé par l'ISO [1], le modèle OSI [2] (figure 2.1) structure sous forme de couches les différents protocoles utilisés dans les systèmes de communication. Les principaux services sont brièvement décrits ci-dessous [Puj04], [Tan95].

- La **couche application** est le point de contact entre l'utilisateur et le réseau. Elle contient les protocoles requis par les utilisateurs.

- La **couche présentation** s'intéresse à la syntaxe des données transmises pour permettre aux parties ayant des représentations de données différentes de communiquer.

1. International Standardization Organization
2. Open Systems Interconnection

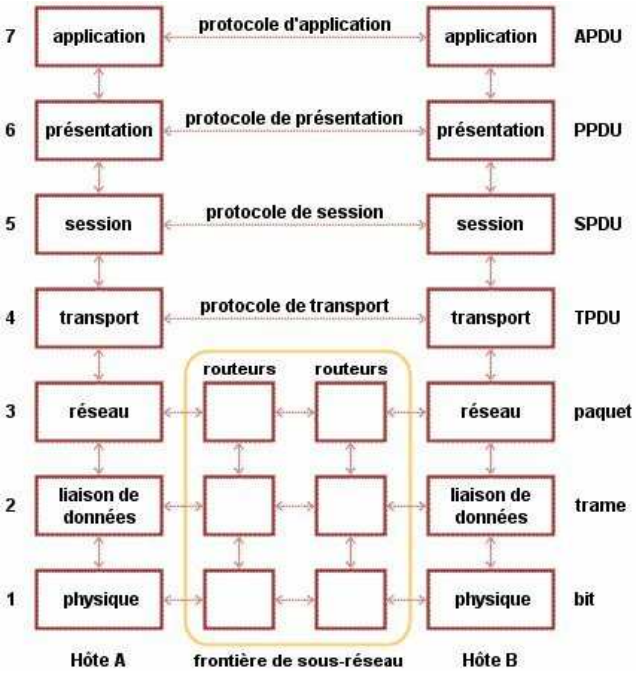

FIGURE 2.1 – Modèle OSI

source : http ://www.frameip.com/osi/

- La **couche session** sert à synchroniser et orchestrer les parties communicantes.

- La **couche transport** gère les communications de bout en bout. Elle prend les messages de la couche session, les découpe si nécessaire et les passe à la couche réseau. A la réception, elle est responsable de réassembler les messages. L'optimisation des ressources et de l'infrastructure du réseau est la tâche principale de ce niveau. Ceci est fait à l'aide d'un système de contrôle de flux ou un mode de multiplexage.

- La **couche réseau** est responsable de l'acheminement des paquets à travers le réseau. Elle s'occupe aussi de la segmentation des messages en paquets. Dans un routeur, cette couche sera responsable de la gestion des conflits entre les messages en concurrence pour ses ressources (notion de priorité). Le routage des messages est effectué au sein de ce niveau. Le contrôle de flux entre l'émetteur et le récepteur ainsi que le mode de commutation sont aussi essentiellement gérés à ce niveau.

- La **couche liaison de données** est aussi appelée la couche trame. Son rôle principal est de transformer la liaison entre deux entités en une connexion exempte d'erreur. Elle fractionne les trames en séquences plus petites. Elle doit être capable de reconnaître les messages et leurs frontières à la réception. La détection et la correction d'erreurs susceptibles d'intervenir sur la couche physique est aussi un rôle important. Des méthodes de régulation de flux doivent exister (e.g. protocole de poignée de main).

- La **couche physique** implémente les règles et les procédures pour réaliser l'acheminement des données binaires sur le médium physique. Les interfaces mécaniques et

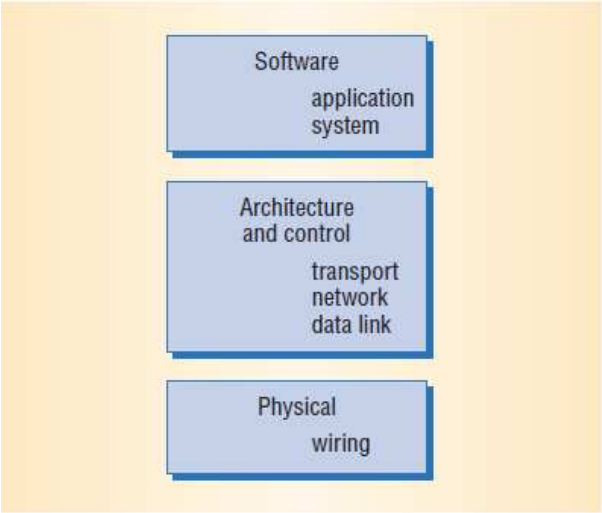

FIGURE 2.2 – Pile de protocoles proposée par Benini et al. [BDM02]

électriques des routeurs avec les fils d'interconnexion sont définies dans cette couche. La topologie et l'architecture du réseau affectent le comportement de cette couche.

La définition de ce modèle a été mise en place pour les réseaux informatiques où l'interaction avec un utilisateur est généralement présente. Dans les réseaux sur puces, le contexte se différencie notamment par le fait que sur les différents nœuds d'une infrastructure d'interconnexion il n'existe pas d'utilisateur à proprement parler. Les couches application et présentation n'apparaissent pas en tant que telles. La couche session n'est pas indispensable : les tâches de cette couche sont effectuées dans l'interface réseau (*Network Interface, NI*).

Dans [BDM02], les auteurs réorganisent le modèle OSI en trois parties (figure 2.2) pour les réseaux sur la puce. Nous pouvons constater que : les couches présentation et session n'existent pas. Les quatre couches inférieures du modèle OSI sont toujours là. Elles seront implémentées dans la partie matérielle du système sous forme de réseau sur puce. Les couches présentation et session sont fusionnées dans la nouvelle couche "*system*". Cette dernière et la couche application seront implémentées dans la partie logiciels embarqués du système sur la puce.

Voyons maintenant les principales caractéristiques que nous associerons aux réseaux sur puce : topologie, routage, technique de communication et contrôle de flux.

2.2 Topologie

Un réseau d'interconnexion permet de connecter différentes entités (IPs : processeur, mémoire, co-processeur, circuit RF), reparties sur différents nœuds. L'acheminement des informations entre nœuds (sous forme de messages) est réalisé par des routeurs auxquels les différents composants sont connectés via les interfaces réseaux (*Network Interface, NI*) (figure 2.3). L'assemblage statique de ces routeurs est la topologie du réseau. Le choix de la

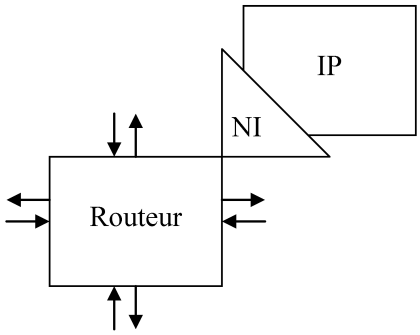

FIGURE 2.3 – Modéle d'un nœud

topologie dépend de son coût et de ses performances (latence et débit) [DT03]. Ci-dessous sont exposées les principales topologies utilisées pour les réseaux sur puces.

Grille. Le nombre de noeuds dans une grille dépend du nombre de dimensions de celle-ci et du nombre de nœuds dans chacune de ces dimensions. Si la dimension i contient K_i nœuds, le nombre total de nœuds pour une grille à n dimensions sera $K_0 \times K_1 \times K_2 ... \times K_{n-1}$. La figure 2.4.b illustre une grille à deux dimensions (grille 2D). Dans cette topologie, chaque nœud est connecté au moins à deux autres nœuds. L'adresse d'un nœud dans un tel réseau est définie par l'ensemble des coordonnées dans chaque dimension. Un nœud dans notre grille 2D de la figure 2.4.b aura pour adresse un vecteur à deux éléments.

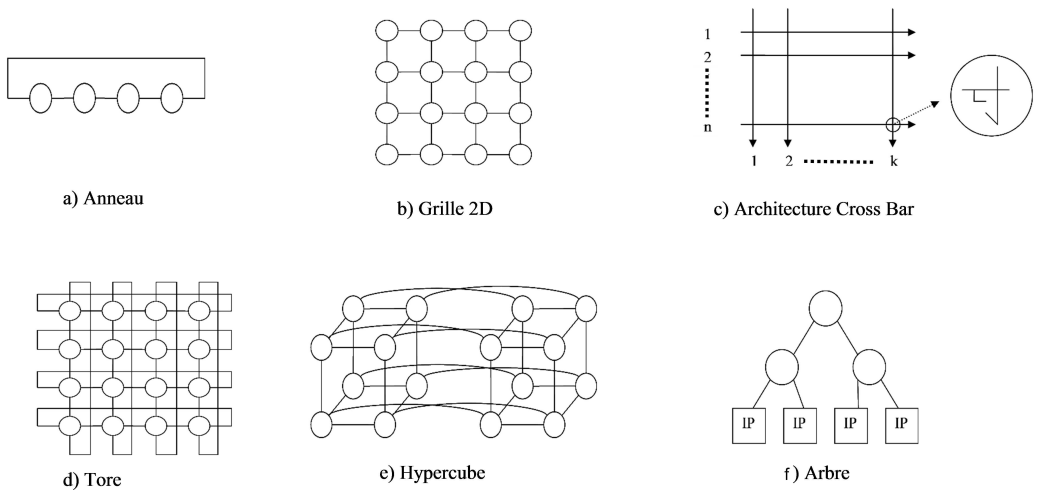

FIGURE 2.4 – Topologie des réseaux sur puces

Tore. Un tore est en quelque sorte une grille dont les bords sont rebouclés sur eux mêmes (figure 2.4.d). L'adressage dans un tore est fait de la même manière que pour une grille.

Anneau. Un tore uni-dimensionnel est un anneau (figure 2.4.a). L'adresse d'un nœud dans un anneau est formée d'une seule coordonnée.

Arbre. Dans cette architecture (figure 2.4.f), Les feuilles sont les IPs tandis que les nœuds supérieurs sont des routeurs qui acheminent les messages.

Cube. Un Kn-cube est une structure à n dimensions et K nœuds par dimension. Chaque nœud sera identifié par un vecteur de n éléments. Si K=2, chaque nœud aura n voisins seulement, si K>2, il aura 2n voisins. Un hypercube (figure 2.4.e) est un 2-ary n-cube. L'hypercube de la figure 2.4.e est un 2-ary 4-cube.

Cross Bar. Un crossbar $n \times K$ permet de connecter, d'une façon matricielle, n composants en entrée à K composants en sortie d'un façon parallèle (figure 2.4).c. Il est particulièrement utilisé dans les réseaux dynamiques et dans les configurations multi-étages.

2.3 Routage

Le routage définit l'acheminement des paquets dans le réseau. Nous réalisons une présentation succincte des méthodes de routage qui peuvent être utilisées sur la puce. Plusieurs classifications existent pour les algorithmes de routage. Ils peuvent être classés selon l'endroit où la décision est prise. Dans les algorithmes de routage *à la source*, les décisions de routage sont prises à la source et encodées dans le message pour guider les routeurs sur le chemin. Un routage *distribué* permet le calcul de la route au fur et à mesure de l'avancement du message sur chaque nœud intermédiaire. A l'arrivée dans un routeur, une décision est prise pour expédier le message à un nœud voisin ou au composant associé.

Un algorithme de routage *minimal* choisit un plus court chemin, sinon il est *non-minimal*. Un algorithme renvoyant toujours la même route, en fonction du nœud courant et de la destination, est un algorithme de routage *déterministe*. Si l'état du réseau rentre dans le processus de la prise de décision de routage, ceci est un algorithme *adaptatif*. Nous adoptons la classification des algorithmes de routage en algorithmes déterministes ou adaptatifs ainsi que minimaux ou non-minimaux.

2.3.1 Algorithmes de routage déterministes minimaux

Ce type d'algorithme calculera toujours la même route entre le nœud courant et la destination d'un message. Le chemin calculé sera toujours parmi les plus courts. Adoptés dans les premiers systèmes d'interconnexion, ces algorithmes sont simples et faciles à implémenter. Plus particulièrement, ils sont largement utilisés dans les réseaux avec une topologie irrégulière parce que la création d'une méthodologie adaptative est compliquée dans ce type d'architecture [DT03]. Néanmoins, leur utilisation peut causer des congestions dans des parties du réseau, ainsi qu'une charge déséquilibrée. Nesson et Johnsson proposent une méthode de routage remédiant à ce problème dans les réseaux ayant une topologie en grille et tore [NJ95].

Le routage par étiquetage de destination [Law75] (*destination-tag routing*) fait partie de ce type d'algorithmes. C'est un routage à la source où une étiquette est mise dans

l'entête du message qui guidera les routeurs pour choisir le port de sortie sur lequel le paquet continuera son chemin.

Une famille d'algorithmes utilisant l'ordonnancement des dimensions a largement été utilisée dans les réseaux multidimensionnels (*dimension-ordered routing)*. C'est une méthodologie de routage distribuée. Un ordre strict est défini pour les dimensions. Chaque message est acheminé sur une dimension puis passera à une autre dans cet ordre. Cependant, des situations d'interblocages peuvent survenir.

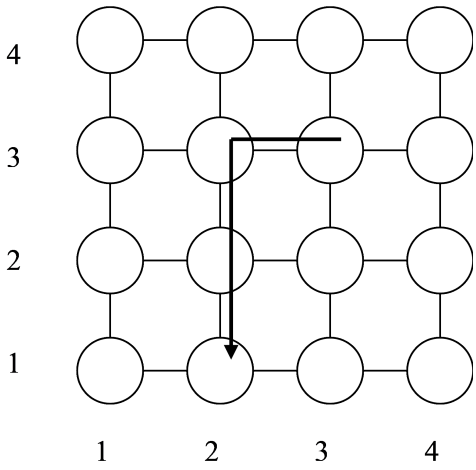

FIGURE 2.5 – Routage déterministe minimal dans une grille 2D

Exemple 2.1 Routage XY.

Voyons un exemple de routage XY (i.e. l'ordre sur les dimensions est X< Y) dans une grille 2D (figure 2.5). Un message est émis sur le nœud (3 3) avec une destination qui est le (2 1). Le message sera d'abord routé le long de l'axe des X. La coordonnée X de la destination étant atteinte, l'algorithme commencera à acheminer le message dans la direction des Y en passant par (2 2) pour arriver à la destination (2 1).

2.3.2 Algorithmes de routage adaptatifs

Un algorithme adaptatif utilise des informations sur l'état du réseau pour décider le chemin suivi par un message. Des informations locales peuvent être utilisées comme le nombre de places libres dans les mémoires tampons du nœud. L'état global du réseau peut être pris en compte, par exemple en considérant la charge des routeurs voisins.

2.3.2.1 Algorithmes de routage adaptatifs minimaux

Ce type d'algorithme choisira toujours une des routes les plus courtes entre le nœud courant et la destination, autrement dit chaque décision de routage rapproche le message de sa destination.

Exemple 2.2 Routage YX adaptatif.

Considérons l'algorithme routage YX suivant utilisé dans un réseau ayant une topologie de grille bi-dimensionnelle. Cet algorithme privilégie un routage YX mais permet des adaptations selon la charge du réseau.

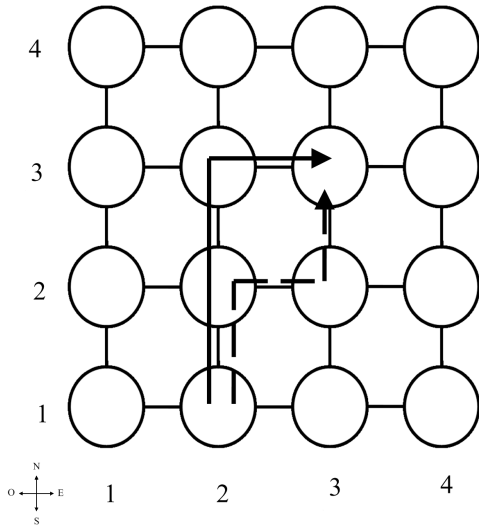

FIGURE 2.6 – Routage adaptatif minimal dans une grille 2D

Tant que le message n'est pas arrivé :

1. *Si les coordonnées X et Y du nœud actuel et de la destination sont identiques, le message est arrivé et transmis au composant attaché.*
2. *Si la coordonnée Y du nœud courant est plus petite que celle de la destination, le message avance vers le nord. Si elle est plus grande, le paquet sera transmis vers le sud. Dans le cas d'une congestion sur la route choisie, ou de l'égalité entre les deux coordonnées Y, l'algorithme passe à l'étape suivante.*
3. *Les coordonnées X sont comparées. Si celle de la destination est plus grande que celle du nœud courant, le message avance vers l'est. Sinon, le message se dirige vers l'ouest. En cas de congestion sur la route calculée dans cette étape ou si le mouvement vers l'est ou l'ouest éloigne le message de sa destination, le message ne bouge pas et attend de pouvoir faire un mouvement qui le rapproche de sa destination.*

Fin tant que.

Sur l'exemple de la figure 2.6, un message est envoyé du nœud (2 1) à destination du nœud (3 3). Sans congestion le message procédera par la route en trait plein : (2 1),(2 2),(2 3) et (3 3). Avec un engorgement sur le lien entre (2 2) et (2 3), l'algorithme calculera le chemin en pointillés : (2 1), (2 2), (3 2), puis (3 3). Si le lien entre (3 2) et (3 3) est bloqué le message attedndra sur le nœud (3 2).

2.3.2.2 Algorithmes de routage adaptatifs non-minimaux

Les algorithmes de type adaptatifs non-minimaux acheminent les messages sur des routes qui ne sont pas forcément les plus courtes. Un pas de routage peut éloigner le

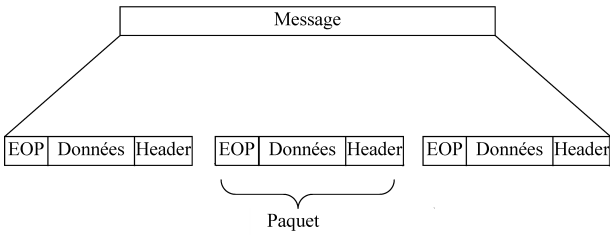

FIGURE 2.7 – Découpage d'un message en paquet

paquet de sa destination.

Par exemple, le réseau Nostrum [KJM+02][Nil03] est muni de routeurs sans buffers. Le stockage des messages est donc impossible et l'algorithme de routage utilisé est un algorithme dit "hot-potato" : tout message doit immédiatement quitter le nœud sur lequel il est arrivé et est donc envoyé à un nœud voisin, même si la route choisie ne le rapproche pas de sa destination.

Des solutions au problème du livelock (le message se déplace dans le réseau mais sans jamais atteindre sa destination) ont été proposées comme le passage à un routage minimal après un certain nombre de pas de routage pour chaque message. D'autres solutions pour les deadlocks (blocage des messages) ont été proposées comme l'interdiction d'une des directions de mouvements [GN92].

2.4 Commutation

Les techniques de commutation définissent l'allocation des ressources (*i.e. les mémoires tampons, et les liens*) du réseau aux messages arrivant sur les nœuds. Une bonne méthode optimise l'utilisation des ressources. Des techniques ont été utilisées dans les réseaux informatiques et elles le sont également dans les réseaux sur la puce. Les méthodes diffèrent principalement sur leurs besoins en mémoire et sur la *latence* (temps entre l'envoi et la réception d'un message).

Au niveau de la couche réseau, les données échangées sont assimilées à des paquets. La figure 2.7 montre cette division. Les informations nécessaires pour le routage d'un paquet (*e.g.* la source et/ou la destination) sont présentes dans le *Header*, tandis que le EOP (*trailer*) signale la fin d'un paquet (*End Of Packet*).

2.4.1 Commutation par paquet

Avec ce type de commutation, les paquets sont acheminés indépendamment les uns des autres. Deux modes existent : *store-and-forward* (figure 2.8), et *virtual cut-through* (figure 2.9). Pour le mode *store-and-forward*, les besoins en mémoire sont plus importants qu'avec les autres méthodes : le paquet entier doit être mémorisé à l'arrivée dans chaque routeur. Une mémoire pouvant garder le plus grand paquet est indispensable. Le mode *virtual cut-through* permet de diminuer le temps nécessaire en transmettant le paquet dès que le *header* est reçu, et les ressources nécessaires redeviennent disponibles (mémoire et canal de sortie) au plus vite. Le parallélisme obtenu par cette méthode est plus important :

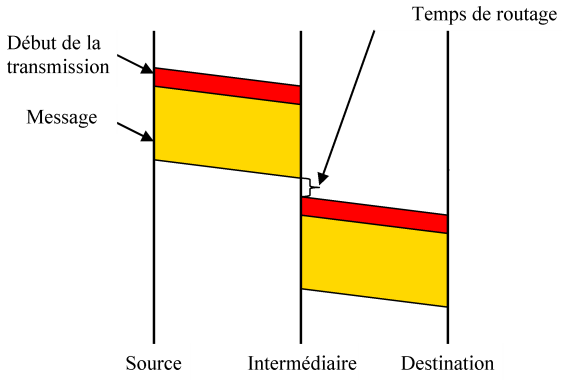

FIGURE 2.8 – Commutation par paquets *(store-and-forward)*

les ressources ne sont pas réservées durant tout l'envoi, elles le sont juste lors du besoin ce qui permet une meilleure utilisation.

Le temps d'envoi (temps de voyage entre la source et la destination) sera proportionnel à la distance entre la source et la destination dans le mode *store-and-forward* (équation 2.1 [DT03]). T_h est le temps de voyage du header entre deux nœuds voisins. Le temps nécessaire pour envoyer le reste du paquet est exprimé par $Size/Bdth$, tandis que D est la distance entre les deux parties communicantes (nombre de nœuds sur la route). *Size* est la taille du reste du message et *Bdth* est la bande passante du lien [3].

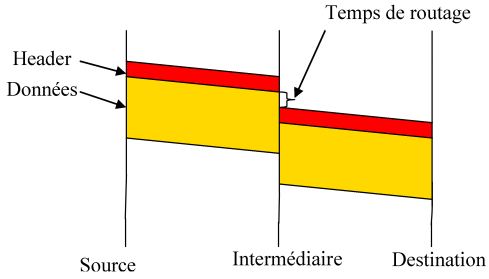

FIGURE 2.9 – Commutation par paquet *(virtual cut-through)*

Dans le mode *store-and-forward*, le message doit être reçu entièrement avant d'être retransmis. Le paquet entier est envoyé entre deux nœuds, le temps nécessaires sera de $(T_h + Size/Bdth)$. Ceci sera répété D fois (la distance entre la source et la destination).

$$Temps\ total = D \times (T_h + Size/Bdth) \tag{2.1}$$

L'utilisation du mode *cut-through* permettra de diminuer ce temps (équation 2.2 [DT03]) qui sera plus petit que le temps de commutation par circuit. Dès que le *header* est analysé, le nœud transmet le paquet avant même de le recevoir en entier. Le temps total devient donc la somme des temps d'analyse et de routage pour le paquet

3. Notons que les termes T_h, D, *Size* et *Bdth* seront utilisés le long de cette section pour les différents modes de commutation avec la même interprétation.

$D \times T_h$ et le temps d'envoi du reste du paquet qui procède d'une manière comparable à un pipeline $Size/Bdth$.

$$Temps\ total = D \times T_h + Size/Bdth \tag{2.2}$$

De ce fait, cette méthode est plus efficace pour les réseaux de taille modeste pour envoyer une quantité de données relativement petite.

2.4.2 Commutation par circuit

Le *circuit switching* consiste à réserver le chemin entier entre la source d'un message et sa destination au début de la communication. L'entête du message est envoyée d'abord pour réserver la route. Arrivée à destination, un acquittement est renvoyé à la source.

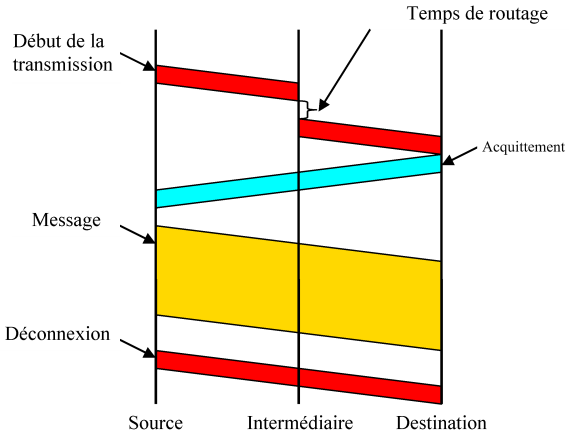

FIGURE 2.10 – Commutation par circuit

La Figure 2.10 montre le déroulement d'un envoi d'un message entre une source et une destination à travers un nœud intermédiaire. Après l'établissement du circuit, les données et le *tailer* du message sont envoyés. Après l'envoi de toutes les informations, un paquet de déconnexion est envoyé pour la libération des ressources réservées. La réservation de la route entière limite le parallélisme des transmissions dans ce type de commutation. Néanmoins, les besoins en mémoire sont faibles : seuls les entêtes et les acquittements sont analysés et mémorisés. Cette méthode est intéressante pour les grands réseaux lors de l'envoi d'une grande quantité de données à un destinataire lointain. Si D est la distance entre la source et la destination, le temps total de l'envoi est exprimé dans l'équation 2.3 [DT03]. Le terme $3 \times D \times T_h$ traduit le temps nécessaire pour que le header atteigne la destination, puis pour le retour de l'acquittement, puis le temps nécessaire pour l'envoi du signal du déconnexion (la distance D est parcourue trois fois). La communication fonctionne ensuite comme un pipeline entre les nœuds intermédiaires.

$$Temps\ total = 3 \times D \times T_h + Size/Bdth \tag{2.3}$$

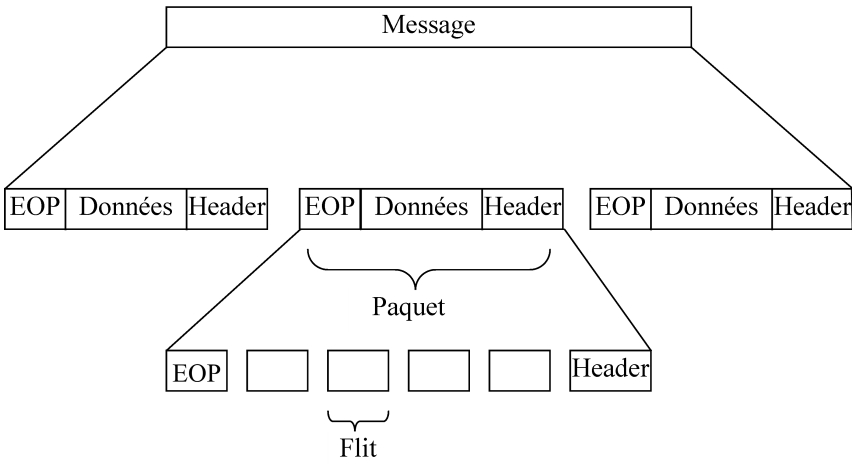

FIGURE 2.11 – Découpage d'un message en flits

2.4.3 Commutation par ver de terre

Proposée par Dally et Seitz [DS86], puis raffinée par Dally dans [Dal92], cette méthode fonctionne de la même façon que le *cut-through*, mais avec des flits (figure 2.11) au lieu des paquets. Ceci implique la diminution de la taille de la mémoire nécessaire. Généralement, le *header* contient les informations pour effectuer le routage, tandis que le *tailer* (le flit EOP) marque la fin d'un message.

Le *header* (la tête du ver) passe en premier et réserve la route un pas à la fois. Le reste des flits du message le suit comme dans un *pipeline*. Chaque fois que le *tailer* traverse un nœud les ressources réservées sont libérées. Le temps d'envoi sera égal à celui de la méthode *cut-through* de la commutation par paquet dans les meilleurs des cas. Les flits appartenant au même message se suivent et peuvent résider sur des nœuds différents. Un flit avance quand le nœud prochain possède de la place pour le recevoir. Les flits des autres messages ne peuvent pas être acceptés sur un port tant que le ver n'est pas passé en entier à travers ce dernier (un seul chemin virtuel peut exister sur un lien physique). La limitation principale de cette méthode est qu'elle est plus susceptible de créer des interblocages.

2.4.4 Commutation par canaux virtuels

Cette technique s'apparente en fait au contrôle de flux, généralement associé à la commutation par ver de terre. Elle permet de surmonter les problèmes d'interblocages de cette dernière en permettant la présence de plusieurs canaux virtuels sur le même lien physique. Un canal virtuel est un ensemble d'éléments mémorisants associés à un lien. Lors de l'arrivée d'un paquet sur un nœud, l'entête est analysée. Si le lien nécessaire est indisponible, le paquet est mémorisé dans les éléments mémorisants du canal auquel il est associé. Sinon il est transmis directement sans mémorisation. La présence de plusieurs canaux sur le même lien physique permet aux flits d'un paquet de passer devant un autre paquet si ce dernier est bloqué, optimisant ainsi la bande passante du réseau.

2.5 Contrôle de flux

Des techniques de contrôle de flux peuvent être utilisées. Dans les réseaux d'interconnexion, elles se basent sur les espaces libres dans les buffers des voisins pour décider d'envoyer un flit ou un paquet. Nous présentons ici quelques méthodes utilisées.

Contrôle de flux par crédit. Chaque routeur en amont tient un compte des places libres dans les buffers des canaux virtuels des voisins avec lesquels il communique (le cas d'un seul canal par lien est possible). Chaque fois qu'un flit est transmis, le compteur du canal utilisé est décrémenté. Arrivé à zéro, aucun flit ne pourra utiliser le canal virtuel en question. Lorsque le routeur aval (de l'autre côté du canal) aura de la place pour recevoir des nouveaux flits, il enverra un *crédit* qui incrémentera le compteur (de un ou plus selon le cas).

Contrôle de flux *On/Off*. Un signal est envoyé au routeur en amont pour annoncer la présence de place dans le routeur aval. Ceci est le signal *On*, le routeur peut transmettre des données. Lorsque les places disponibles chez le récepteur chute au-dessous qu'un seuil défini, un signal *Off* est envoyé. L'émetteur arrête les envois dans ce cas. Il suffit d'un bit pour garder l'information de disponibilité de place dans le routeur émetteur.

Il existe d'autres méthodes de contrôle de flux que nous ne présenterons pas ici. Ces méthodes ne sont pas utilisées dans la littérature des réseaux sur la puce.

2.6 Réalisations de réseaux

Dans le chapitre 1, nous avons présenté les réseaux sur puces comme un nouveau paradigme apportant une réponse aux problématiques de conception des circuits [BDM02]. Plusieurs études ont été menées donnant lieu à différentes réalisations et descriptions. Les travaux de l'équipe BONE[4] ont fait l'objet d'implémentations de réseaux à basse consommation et latence. Plusieurs applications multimédia ont été implementées à l'aide de leurs réseaux, [SJKSJHJ05],[KSJY+08]. D'autres réseaux sur puces ont été conçus comme : Hermes [MCM+04], Xpipes [BB04], SonicsMX, Nostrum [KJM+02][Nil03], Spidergon [CLM+04], et DSPIN [MPGS06]. Hermes, Faust [CVL05] [DBL05] et Nostrum ont été réalisés sur FPGA et des mesures de performances ont été publiées. SonicsMX est une solution d'interconnexion industrielle proposé par Sonics. D'autres réseaux ont été réalisés en technologie ASIC comme Æthereal[RGR+03], SPIN[And03], DSPIN. Des SoCs construits à l'aide de NoCs existent déjà comme l'architecture Faust. L'architecture Faust est une plateforme de télécommunications respectant les normes IEEE802.11a et le MC-CDMA implémentée avec les réseaux aSOC et DSPIN [MPCVG08].

Dans cette section, nous allons présenter trois implémentations de réseaux sur puces qui ont été utilisées comme cas d'études pour notre modèle : Hermes, Nostrum et Spidergon.

4. http ://ssl.kaist.ac.kr/∼ocn/

2.6.1 Hermes

Hermes [5] est développé à l'université catholique de Rio Grande do Sol (Brésil) avec la collaboration du Laboratoire d'Informatique de Robotique et de Micro-électronique de Montpellier (LIRMM). Ce réseau utilise la commutation par ver de terre dans une grille à deux dimensions avec un routage XY sans contrôle de flux. Un routeur contient cinq ports bidirectionnels : *N*ord, *S*ud, *E*st, *O*uest, et *L*ocal. Chaque IP est couplé à un routeur à l'aide du port local (figure 2.12), l'interconnexion avec les voisins se fait à travers les quatre autres ports.

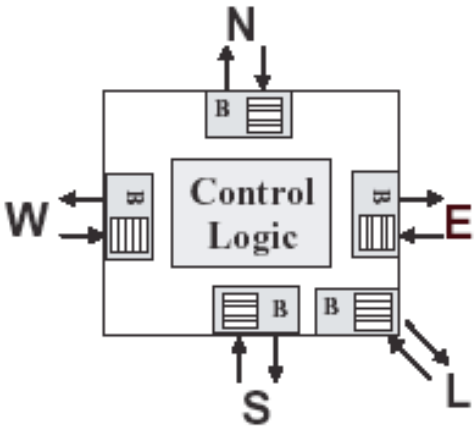

FIGURE 2.12 – Routeur du réseau Hermes [MCM+04]

Chaque routeur contient principalement : la logique de contrôle (composée de la logique de routage et d'arbitrage) et les ports de communications. Chaque port d'entrée mémorise les données reçues dans une FIFO. La communication asynchrone entre les routeurs est faite à l'aide d'un protocole de poignée de main, alors que dans les routeurs les transactions sont synchrones (Globalement Asynchrone Localement Synchrone). Un code de correction d'erreur est utilisé pour s'affranchir des erreurs éventuelles. Un arbitrage à priorité tournante (*round robin*) est utilisé pour arbitrer l'accès aux ports. Les flits reçus ou bloqués sont stockés dans le buffer dans un port de communication (ceci limite la chute des performances). Des buffers à FIFO circulaire paramétrable sont utilisés.

La figure 2.13 décrit l'interface entre deux routeurs. Quand un routeur a des données à envoyer à un de ses voisin il active le signal Tx et attend l'acquittement de l'autre partie sur la ligne ack_tx. Suite à cet échange, le routeur peut procéder à l'envoi des données.

La logique de contrôle est l'ensemble de la logique de routage et d'arbitrage. Lors de la présence d'un flit à envoyer sur un port d'entrée, ce dernier envoie une requête à la logique d'arbitrage. Lorsque celle-ci termine l'arbitrage selon la politique *round robin*, elle transférera les données nécessaires à la logique de routage. Cette dernière décidera alors du port de sortie selon l'adresse et l'état du tableau du routage. Si le port est occupé, les flits de ce paquet sont bloqués sur le port d'entrée et le signal de requête à la logique

5. http ://www.inf.pucrs.br/∼gaph/Projects/Hermes/Hermes.html

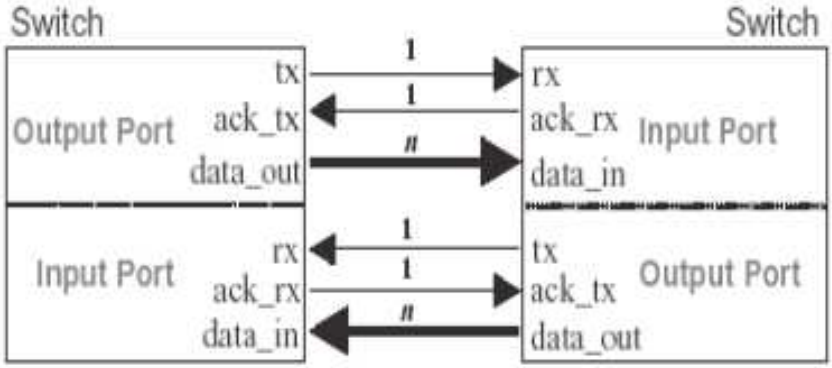

FIGURE 2.13 – Interface entre deux routeurs [MCM+04]

d'arbitrage reste actif. Au cycle suivant le processus recommencera jusqu'au moment où le message pourra obtenir le port de sortie.

2.6.2 Nostrum

Nostrum [6] est une architecture de type GALS ayant une topologie de grille 2D, couplée à un algorithme de routage adaptatif non-minimal et une commutation par paquets et par circuits virtuels. Chaque routeur contient 5 ports bidirectionnels : Nord, Sud, Est, Ouest et Local (figure 2.14). Les besoins en mémoire sont minimes, un paquet n'est jamais mémorisé ni bloqué dans un routeur.

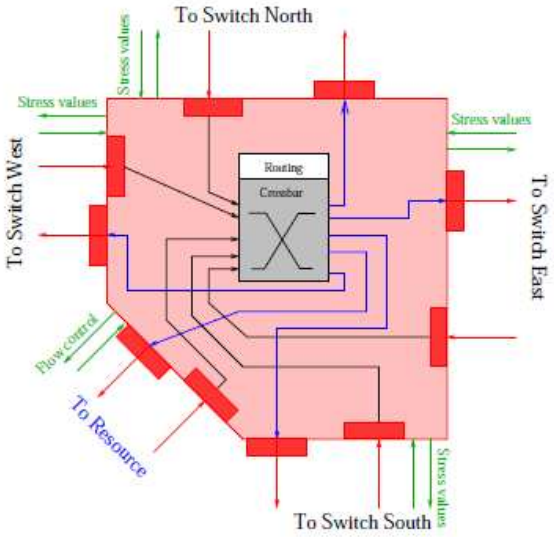

FIGURE 2.14 – Routeur de Nostrum

L'utilisation de deux modes de commutation permet de définir deux types de trafic : le trafic normal est acheminé via commutation par paquet, et les circuits virtuels seront utilisés pour satisfaire des besoins de QoS (*guaranteed service*). La taille des paquets est

6. http ://www.ict.kth.se/nostrum/

paramétrable. L'entête contient l'adresse de la destination sous forme d'adresse relative sur les deux dimensions (X et Y) ainsi qu'un compteur de saut (*Hop Counter*) qui est incrémenté lors du passage sur un routeur.

Le routage est un routage déflectif, dit "hot potato" : chaque paquet est immédiatement retransmis à un routeur voisin, ce qui permet de n'avoir qu'un buffer à une place sur chaque port d'entrée et de sortie. Pour décider du voisin auquel le paquet sera transmis, les paquets sur les cinq ports d'entrée sont analysés et selon les valeurs de la destination, une ou deux directions favorites sont choisies pour chaque message. Si les adresses relatives X et Y sont égales à 0 la direction favorite est le port local. Sinon, si la coordonnée X est supérieure à 0, l'*est* est parmi les directions favorites. Sinon, c'est la direction *ouest*. Si la coordonnée Y est supérieure à 0, le *sud* est parmi les directions favorites, autrement c'est la direction *nord*. Le table 2.1 résume le choix des directions favorites. Quand un message a deux directions favorites, l'algorithme spécifie de suivre la direction qui lui permet de voyager le plus longtemps au long de la même dimension, e.g. si les adresses relatifs sont x=-3 ey y=2, le message aura comme directions favorites l'ouest et le sud. L'algorithme spécifie dans ce cas de suivre la direction ouest puisque $|x| > |y|$. Avec l'adressage relatif utilisé, il suffit de comparer le X et le Y, et suivre la direction ayant la composante la plus grande.

X	Y	Direction(s) Favorite(s)
0	0	Locale
0	0<	Sud
0	<0	Nord
<0	0	Ouest
<0	0<	Sud et Ouest
<0	<0	Nord et Ouest
0<	0	Est
0<	0<	Sud et Est
0<	<0	Nord et Est

Table 2.1 – Choix des directions favorites

Exemple 2.3 Routage du réseau Nostrum.

La figure 2.15 montre un message au départ du nœud (1 1) avec une destination (2 3). L'adresse de la destination dans l'entête sera de (1 2) qui représente la distance entre la position actuelle du message et la destination : un pas vers l'est et deux pas vers le sud. Cette adresse sera modifiée à chaque pas du message pour refléter le déplacement en calculant la nouvelle adresse relative. Dans une situation idéale (pas de conflit), le message suivra tout d'abord la direction sud (X < Y). L'adresse est modifiée pour devenir (1 1). A ce routeur, le message sortira par le port sud, l'adresse devient (1 0). Ensuite, le message sera transmis dans la direction de l'est pour arriver à destination.

Il se peut que plusieurs messages aient les mêmes préférences de directions. Si l'un des deux appartient à un circuit virtuel, il gagne l'arbitrage pour le port de sortie. Sinon, un système de priorité est utilisé dans lequel le message le plus ancien (ayant le *hop counter* le plus grand) gagne l'arbitrage. Dans le cas d'égalité, le message le plus proche de sa

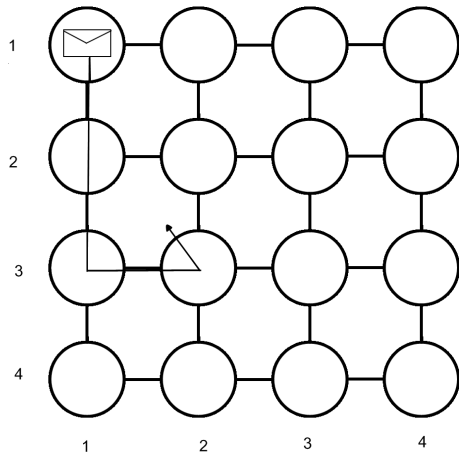

FIGURE 2.15 – Exemple de routage de Nostrum

destination l'emporte. Ceci tend à éviter des situations de livelock. Le message perdant empruntera l'autre direction favorite (dans le cas où il avait le luxe d'en avoir deux), ou sera transmis dans une direction non favorite.

Le choix des ports de sortie non désirés pour un message se fait selon deux critères : la charge du routeur dans chaque direction et la situation du routeur. Chaque routeur communique à ses voisins la charge qu'il a eue durant les quatre derniers cycles. Le choix favorise toujours le routeur ayant la moindre charge. Le deuxième critère consiste à éloigner les messages loin du centre du réseau dans le but de diminuer la congestion dans cette zone. Si un routeur est dans le sud-est du centre du réseau, la préférence sera d'envoyer les messages qui perdent l'arbitrage dans la direction du sud ou de l'est.

2.6.3 Spidergon

Développé par STMicroelectronics, le réseau Spidergon [CLM+04] a une topologie en anneau dans laquelle chaque routeur est relié directement à son correspondant disposé en diagonale. Cette connexion permet de minimiser le nombre de nœuds qu'un paquet traversera pour parvenir à sa destination. Une commutation par ver de terre est utilisée, couplée à un algorithme de routage minimal. Le réseau est une extension du réseau Octagon [KND02], permettant de construire des systèmes avec un nombre de nœuds qui est multiple de quatre.

La figure 2.16 montre l'interconnexion dans le cas de 16 nœuds. Chaque nœud possède quatre ports qui le connectent à trois voisins, un dans le sens des aiguilles d'une montre (port *clockwise*), un dans le sens contraire des aiguilles d'une montre (port *anticlockwise*), et un en face (port *across*), ainsi qu'à un IP (port *local*). L'algorithme de routage utilisé décrit ci-dessous permet aux paquets d'atteindre leur destination en un nombre de pas inférieur ou égal à $Nodenum/4$, où $Nodenum$ est le nombre de nœuds. Lors de l'arrivée d'un message sur un nœud, l'adresse relative de la destination ($RelAd$) est calculée à l'aide de l'adresse locale du nœud ($LocAd$) et l'adresse de la destination ($DestAd$). L'équation 2.4 utilisée dans le calcul est le même que pour le réseau Octagon ; N vaut $Nodenum/4$,

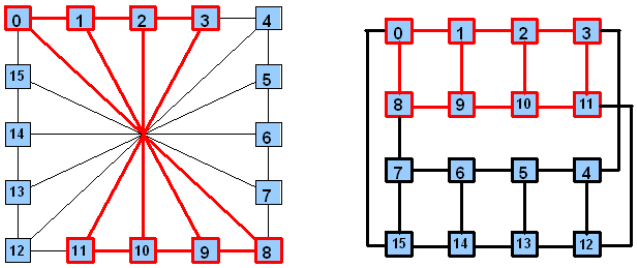

FIGURE 2.16 – Réseau Spidergon

http ://www.st.com/stonline/press/news/year2005/fra/t1741tfra.htm

soit 4 dans le cas d'un réseau à 16 nœuds.

$$RelAd = \mid DestAd \text{ - } LocAd \mid \mod (4 \times N) \tag{2.4}$$

La décision de routage est prise suivant le résultat du calcul de $RelAd$:
- $RelAd = 0$, le paquet est transmis à l'IP par le biais du port local,
- Si $0 < RelAd \leq N$, le message est transmis par le port *clockwise*,
- Si $3 \times N < RelAd \leq 4 \times N$, le port *anticlockwise* est utilisé,
- Sinon le message sortira par le port *across*.

Exemple 2.4 Routage du réseau Spidergon.

Un message ayant pour destination le nœud 12 arrive sur le routeur 2. L'adresse relative est de 10. Le routeur enverra le paquet par le port across pour atteindre le nœud 10. Sur le nœud 10, l'adresse relative sera 2. Dans ce cas, le message voyage dans la direction clockwise. Sur le nœud 11, l'adresse relative est de 1, le message passe par le port clockwise pour atteindre le routeur 12. Sur le nœud 12, le paquet sera transmis à l'IP à travers le port local.

2.7 Conclusion

Nous avons illustré à travers ce chapitre les notions de base des réseaux et des systèmes d'interconnexion : topologie, routage, et commutation. Trois réseaux qui vont être utilisés dans la suite en tant que cas d'étude ont été décrits.

Le réseau Hermes utilise une commutation par ver de terre. La modélisation de ce type de commutation n'avait jamais été réalisée avec le modèle *GeNoC* original. C'est par cette étude que nos travaux ont débuté, afin d'évaluer l'adaptabilité du modèle dans ce contexte. Ce même réseau a ensuite été utilisé pour évaluer notre extension du modèle.

Le réseau Spidergon est un réseau ayant des propriétés assez simples, un algorithme de routage déterministe mais original et une commutation par paquets. Son utilisation comme cas d'étude visait à démontrer l'applicabilité de notre approche dans le cas de systèmes industriels réalistes, ce réseau étant issu de STMicroelectonics.

Le réseau Nostrum, avec son algorithme de routage adaptatif non-minimal, a permis de prouver que notre modèle est suffisamment élaboré et réaliste pour prendre en compte de telles caractéristiques non triviales.

Chapitre 3

Modèle *GeNoC* V1.0

Dans le chapitre précédent, nous avons présenté les concepts principaux relatifs aux architectures de communication. Nous les avons illustrés avec différents réseaux déployés sur la puce qui seront utilisés comme cas d'étude de l'application du nouveau modèle *GeNoC* défini durant cette thèse. Dans ce chapitre, nous présentons les travaux de formalisation des systèmes de communication décrits dans la thèse de Julien Schmaltz [Sch06]. Ces travaux furent la première étape établissant le fondement des travaux présentés dans ce manuscrit. La plupart des informations décrites dans ce chapitre sont issues du troisième chapitre de [Sch06].

3.1 Principe de la méta-modélisation

Les travaux que nous présentons dans ce chapitre exposent un méta-modèle générique formant une représentation de toute architecture de communication, notamment celle déployée sur la puce. Dans ce cadre, une architecture de communication générique arbitraire est représentée mathématiquement par la fonction *GeNoC* paramétrée par les principaux composants présentés dans le chapitre précédent (routage, ordonnancement, et la topologie). La figure 3.1 montre le principe de fonctionnement du modèle. Les fonctions associées aux composantes du réseau n'ont pas de définition. Elles sont contraintes par des obligations de preuve (représentées par P1, P2, P3, et P4) qui expriment des propriétés caractéristiques. La preuve de correction de la fonction *GeNoC* s'appuie sur les obligations de preuve vérifiées sur chacune des fonctions. Autrement dit, la correction du réseau s'appuie sur la correction des différents modules. La propriété de correction $\mathcal{P}$ est ici la suivante : *Chaque message arrivé a atteint son bon destinataire sans altération dans son contenu.* Nous pouvons représenter ce principe de la manière suivante.

$$\forall T, I, R, S, P1(T) \wedge P2(I) \wedge P3(R) \wedge P4(S) \Rightarrow \mathcal{P}(GeNoC(T, I, R, S)) \tag{3.1}$$

La définition du modèle et de ses obligations de preuve consistait à trouver les conditions suffisantes pour garantir le bon fonctionnement d'un réseau. Cette définition a été faite une fois, la preuve de correction du modèle générique a été effectuée à partir de ces conditions. Une fois ces conditions trouvées, la puissance du modèle vient du fait que le

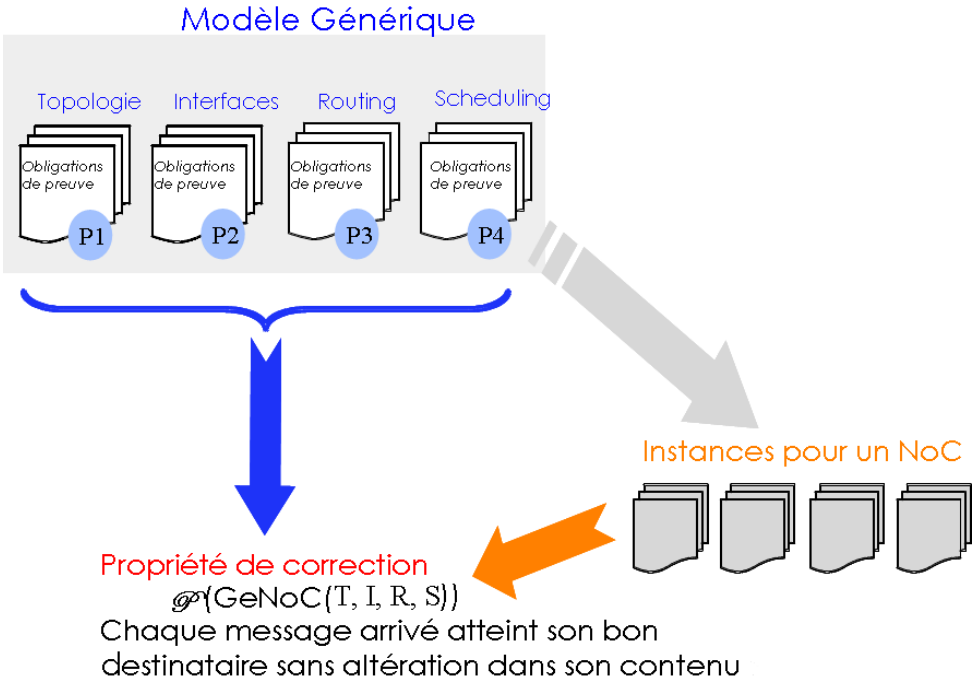

FIGURE 3.1 – Principe de *GeNoC*

processus de vérification d'un réseau consistera seulement à créer une instance de ce modèle pour chaque réseau à vérifier. Une instance est la modélisation du réseau sous forme de fonctions. La preuve des différentes obligations de preuve de chaque module doit être effectuée. La correction globale du réseau est garantie dès le moment où les obligations de preuve des différents modules ont été vérifiées (conditions suffisantes).

3.2 Vision de l'infrastructure de communication

La formalisation haut niveau d'une infrastructure de communication nécessite la mise en place d'une abstraction de tout le système de communication. Un modèle unique pour les systèmes de communication est proposé (figure 3.2). Chaque nœud est divisé en deux parties : la partie application et la partie interface (modèle de Rowson et Sangiovanni-Vincentelli [RSV97]). Ces deux parties communiquent entre elles à l'aide de messages. Les interfaces utilisent l'architecture de communication pour envoyer ces messages à leurs destinations respectives. Un message sera transformé en trames (*frames*) afin d'être transféré par les interfaces de communications.

L'architecture de communication représente le système d'interconnexion et permet aux interfaces de fournir un service de communication aux applications. Elle se caractérise par plusieurs éléments comme la topologie, la politique de routage, le mode de commutation, etc... Le modèle permet de traiter un nombre arbitraire de nœuds avec la seule condition qu'il soit fini.

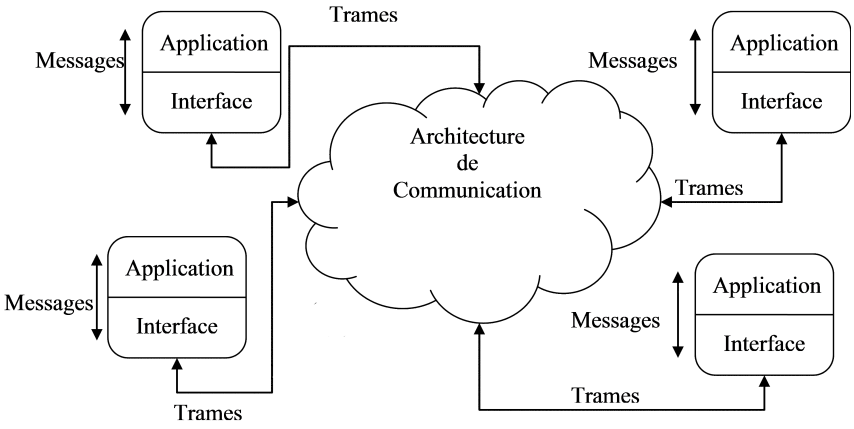

FIGURE 3.2 – Abstraction de communications [Sch06].

3.3 Les types

Les messages subissent plusieurs transformations nécessitant la définition de quatre types différents. Dans cette section, nous présentons ces différents types ainsi que leurs utilisations dans le modèle.

Id	Source	Msg	Destination

FIGURE 3.3 – Une transaction

Transactions. Une opération de communication émise par une application est modélisée sous forme de "*transaction*". Chacune des transactions est identifiée par un naturel *Id*. Elle contient le message *msg* sur lequel aucune restriction n'existe, ainsi que l'origine du message et sa destination (figure 3.3). *NodeSet* est l'ensemble des nœuds d'un réseau. La source et la destination d'une transaction doivent appartenir à cet ensemble. Une liste de transaction est reconnue par le prédicat $\mathcal{T}_{lstp}$. $\mathcal{T}_{ids}$ est la liste des identificateurs d'une liste de transactions. $Org_{\mathcal{T}}$ est une fonction qui renvoie l'ensemble des origines de toutes les transactions, tandis que la fonction $Dest_{\mathcal{T}}$ retourne l'ensemble des destinataires. Le prédicat $\mathcal{T}_{lstp}$ précise que dans une liste valide de transactions :

1. les identifiants des transactions (éléments de $\mathcal{T}_{ids}$) sont des naturels distincts,
2. les ensembles des nœuds origines et destinations sont des sous-ensembles des nœuds du réseau,
3. pour chaque transaction, l'origine et la destination ne sont pas le même nœud.

Si le domaine des nœuds est *NodeSet*, et $\mathcal{D}_{msg}$ est le domaine des messages, alors le domaine des transactions $\mathcal{D}_{\mathcal{T}}$ sera défini par : $\mathbb{N}$ x *NodeSet* x $\mathcal{D}_{msg}$ x *NodeSet*.

Id	Source	Frame	Destination

FIGURE 3.4 – Une Missive

Missives. Lors de l'envoi d'un message, celui-ci est transmis à une interface qui injectera le message dans le réseau. Ceci est le rôle de la fonction *send* (voir section 3.6). Cette fonction transforme les "*transactions*" en "*missives*" (*i.e.* trames).

La différence entre ces deux types est le contenu du message : une missive (figure 3.4) contient le message sous forme de *frame* (message après encodage pour l'envoi sur le réseau). Le domaine des missives $\mathcal{D}_{\mathcal{M}}$ est donc : $\mathbb{N}$ x *NodeSet* x $\mathcal{D}_{frm}$ x *NodeSet*. Une liste de missives est reconnue par le prédicat $\mathcal{M}_{lstp}$. Les contraintes sur la définition des transactions s'appliquent également pour les missives.

Voyages. Les voyages (figure 3.5) sont le résultat de la fonction *Routing*. Cette dernière calcule les routes possibles pour chaque missive pour créer le voyage correspondant. La fonction *Routing* proposera une ou plusieurs routes (sous forme d'une liste de routes) suivant que l'algorithme soit déterministe ou adaptatif. Le domaine des voyages $\mathcal{D}_{\mathcal{V}}$ est : $\mathbb{N}$ x $\mathcal{D}_{frm}$ x $\mathcal{D}_{routes}$, où $\mathcal{D}_{routes}$ est le domaine des routes. Une liste de voyages est notée $\mathcal{V}$ et est reconnue par le prédicat $\mathcal{V}_{lstp}$. Ce prédicat vérifie que :

1. la liste des identificateurs de l'ensemble des voyages $\mathcal{V}_{ids}$ est incluse dans $\mathbb{N}$ et ne contient pas de duplication,
2. pour chaque voyage, la liste de routes est bien formée. Une liste de routes bien formée ne contient que des routes bien construites. Une route bien formée est une liste contenant au moins deux éléments.

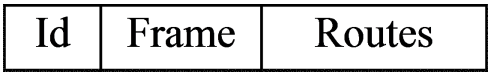

FIGURE 3.5 – Un Voyage

Résultats. A l'arrivée d'un message à sa destination, une interface sera responsable de la réception et du décodage du message. La fonction *recv* (*cf.* section 3.6) est responsable, dans la formalisation, de cette procédure [1]. Le produit de ce décodage est appelé "*résultat*" (figure 3.6). Le domaine des résultats est $\mathcal{D}_{\mathcal{R}} = \mathbb{N}$ x *NodeSet* x $\mathcal{D}_{msg}$. Une liste de résultats est reconnue par le prédicat $\mathcal{R}_{lstp}$, qui vérifie que :

1. la liste des identificateurs de l'ensemble des résultats $\mathcal{R}_{ids}$ est incluse dans $\mathbb{N}$ et ne contient pas de duplication.
2. la destination de chacun des résultats appartenant à la liste doit appartenir à *NodeSet*.

FIGURE 3.6 – Un Résultat

1. La conversion d'un message, sous forme de "*voyage*", en un "*résultat*".

3.4 La fonction *GeNoC*

Cette fonction est une représentation générique des communications dans les architectures de communication. Elle prend en entrée une liste de messages émis aux nœuds sources et modélise leur transmission ainsi que leurs "voyages" jusqu'à leurs destinations respectives. Dans le cadre de la définition de cette fonction, la topologie, la politique de routage, et la technique de commutation restent génériques. Quatre fonctions principales constituent les blocs de construction de *GeNoC*. Deux fonctions (*send* et *recv*) modélisent les opérations des interfaces de chaque nœud. La fonction *send* représente l'encodage du contenu du message, tandis que *recv* sert au décodage de ce contenu. La fonction *Routing* représente la technique de routage dans une topologie donnée. La fonction *Scheduling* encode le mode de commutation utilisé. Ces fonctions sont caractérisées par les *obligations de preuve*. Une présentation détaillée de chacune de ces fonctions sera donnée dans leurs sections respectives.

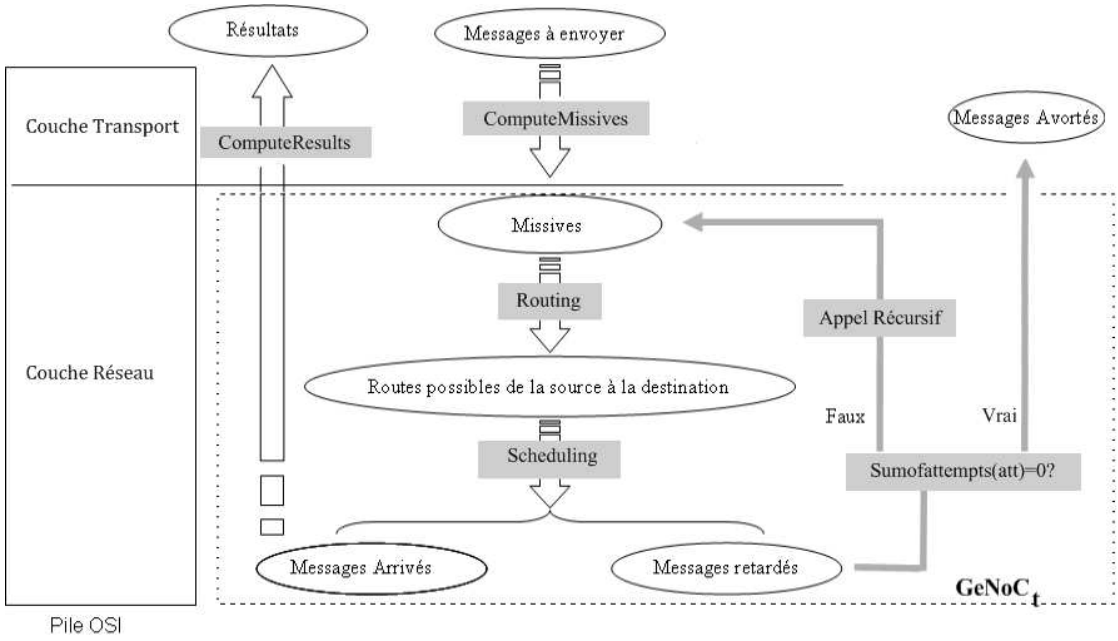

FIGURE 3.7 – Déroulement du modèle *GeNoC*

La figure 3.7 montre le déroulement de *GeNoC*. La fonction prend comme entrées : la liste des messages à envoyer (les transactions $\mathcal{T}$), l'ensemble de nœuds du réseau *NodeSet*, et une liste du nombre de tentatives *att*. Elle retournera deux listes : celle des messages arrivés et celle des messages avortés. Quelques détails ne sont pas explicités ici pour ne pas affecter la clarté de la description, mais le seront dans les sections appropriées.

- La liste des messages à envoyer (les transactions $\mathcal{T}$) est traitée par la fonction *ComputeMissives*. Cette étape génère une liste des *missives* dans lesquelles les messages ont été encodés à l'aide de la fonction *send*.
- *GeNoC* appelle à ce stade la fonction récursive $GeNoC_t$ en lui passant la liste des missives.
- La fonction $GeNoC_t$ invoque la fonction *Routing* en lui passant la liste de ces missives. *Routing* implémente l'algorithme de routage sur chaque élément de la liste.

Elle calcule la liste des routes possibles entre la source de chaque message et sa destination. Le résultat est donné sous forme d'une liste de *voyages*.
- $GeNoC_t$ invoque la fonction *Scheduling* en lui passant cette liste. *Scheduling* tente d'ordonnancer ces voyages selon le mode de commutation utilisé par le réseau. Les messages ayant une route complètement libre [2] entre leur source et leur destination sont ordonnancés (ils atteignent ainsi leur destination dans un seul cycle). Ils forment une liste des messages arrivés à leur destination. Les messages qui ne peuvent pas obtenir leur route pour arriver à destination sont regroupés dans une liste de messages retardés.
- Les messages retardés sont utilisés dans l'appel récursif de la fonction $GeNoC_t$ afin de tenter un ordonnancement au prochain cycle d'exécution.
- Une fois que $GeNoC_t$ a fini son exécution, elle retourne la liste des messages arrivés et la liste des messages avortés à $GeNoC$. Cette dernière invoque la fonction *ComputeResults* sur la liste des messages arrivés pour obtenir la liste des *résultats* (elle contient les messages décodés par la fonction *recv*). La preuve de correction de $GeNoC$ est effectuée sur cette liste.

Un nombre de tentatives est donné à chaque nœud afin de garantir la terminaison de la fonction $GeNoC_t$. A chaque appel récursif, le nombre de tentatives est décrémenté pour chaque nœud. L'appel récursif de la fonction $GeNoC_t$ est ainsi contrôlé. Une fois que la somme des tentatives est nulle, les transactions restantes sont dites avortées, et l'appel récursif de la fonction est arrêté.

La propriété de correction $\mathcal{P}$ de l'équation 3.1 peut se traduire par le théorème 3.1 qui signifie que : *pour chaque message reçu, il existe un message envoyé ayant le même identificateur, le même contenu et une destination identique.* Ceci permet de vérifier que les messages arrivés ont atteint le bon destinataire, sans altération de leur contenu.

Théorème 3.1 Correction de $GeNoC$**.**

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

3.5 Nœuds et paramètres

L'ensemble des nœuds d'un réseau est noté *NodeSet*. Il est généré par la fonction *NodeSetGen* à partir d'un ensemble de paramètres noté *pms*. Le domaine des paramètres est arbitraire et est noté *GenParams*, tandis que le domaine des nœuds valides est *NodeSet*. L'ensemble des paramètres valides est reconnu par la fonction *ValidParamsp*. Le prédicat *ValidNodep* reconnait un nœud valide. *Nodesetp* reconnait un ensemble de nœuds valides en invoquant la fonction *ValidNodep* sur chaque élément de l'ensemble. L'obligation de preuve 3.1 est la seule obligation pour les nœuds. Elle stipule que : tout élément produit par *NodeSetGen* est un nœud valide, si les paramètres *pms* le sont.

Obligation de preuve 3.1 .

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

2. Il n'existe aucun autre message qui utilise un nœud faisant partie de la route en question.

A travers tout ce chapitre et le chapitre suivant, chaque utilisation de *NodeSet* dans une obligation de preuve, un théorème ou un lemme, désigne $NodeSetGen(pms)$ et l'ajout de l'hypothèse $ValidParamsp(pms)$.

3.6 Les Interfaces

Deux fonctions, *send* et *recv*, modélisent les interfaces . La première encode le contenu du message qui sera mis dans le champ *frame* d'une *missive* à partir du champ *msg* d'une *transaction*. La deuxième construit le champ *msg* d'un *résultat* à partir du *frame* d'un *voyage*. La seule contrainte sur ces deux fonctions (obligation de preuve 3.2) est que leur composition est l'identité. *Pour un message appartenant au domaine des messages, le résultat de la composition de send et recv est le message initial.*

Obligation de preuve 3.2 Composition de *send* et *recv*.

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

3.7 Le routage

Le routage est responsable du calcul de la route entre la source s d'un message et sa destination d. La logique de routage dans un réseau est responsable du calcul de la prochaine étape tout au long du chemin entre s et d. Cette logique a été représentée dans le modèle par la fonction $RoutingLogic(x, y)$ où x et y sont deux nœuds quelconques. Pour obtenir le chemin entier, cette fonction est appliquée successivement jusqu'à atteindre la destination. Si N_1, le premier nœud de la route, est obtenu par l'évaluation de l'expression $RoutingLogic(s, d)$, N_2 (le deuxième nœud) sera obtenu à partir de $RoutingLogic(N_1, d)$. La route entre s et d est donc :

$$s, N_1, N_2, ..., d$$

La route entière sera calculée à l'aide de la fonction *RoutingCore* qui applique la fonction *RoutingLogic* de la source à la destination. Elle est définie comme suit :

Définition 3.1 Définition de *RoutingCore*.

$$RoutingCore(s, d) \triangleq \begin{cases} d & \textbf{\textit{si}}\ s = d \\ append(s, RoutingCore(RoutingLogic(s, d), d)) & \textbf{\textit{sinon}} \end{cases}$$

La fonction *RoutingCore* est récursive : il faut donc prouver sa terminaison. Cette preuve permet de s'assurer qu'aucun message ne voyagera indéfiniment dans le réseau. La preuve de la terminaison de la fonction est faite en montrant qu'une *mesure* décroit pour tout appel récursif de la fonction. Une fonction *measure* est ainsi définie pour la fonction de routage *RoutingCore*. Cette fonction prendra en entrée deux nœuds s et d. Considérons un ensemble S muni d'une relation bien fondée $\prec_S$ (dans la plupart des cas, il s'agit de l'ensemble des entiers naturels $\mathbb{N}$). La fonction *measure* est de type $NodeSet \times NodeSet \rightarrow S$. Elle est définie de telle façon que sa valeur décroisse à chaque appel récursif. Ici, dans l'unique appel récursif présent dans la définition de *RoutingCore* (qui correspond au cas $s \neq d$), l'expression $RoutingLogic(s, d)$ prend la place du paramètre s, le paramètre d étant inchangé. Il faut donc garantir que $measure(RoutingLogic(s, d), d) \prec_S measure(s, d)$ dans le cas $s \neq d$, d'où l'obligation de preuve suivante.

Obligation de preuve 3.3 Terminaison de *RoutingCore*.
$\forall s, d \in NodeSet, \exists measure : NodeSet \times NodeSet \rightarrow S,$
$s \neq d \Rightarrow measure(RoutingLogic(s,d),d) \prec_S measure(s,d)$

Exemple 3.1 Routage XY et sa terminaison.

Nous utilisons l'exemple de la section 2.3.1 pour illustrer le principe de terminaison : un réseau avec un routage XY et un message au départ du nœud (3 3), à destination du nœud (2 1). La mesure $measure(s,d) = |d_x - s_x| + |d_y - s_y|$ *détermine une valeur dans l'ensemble des naturels. A chaque application de* $RoutingLogic_{XY}$*, cette mesure décroit. Le message traversera dans l'ordre les nœuds (3 3), (2 3), (2 2), et finalement (2 1).*

s	d	$RoutingLogic(s,d)$	$measure$(s,d)
(3 3)	(2 1)	(2 3)	|2 - 3| + |1 - 3| = 3
(2 3)	(2 1)	(2 2)	|2 - 2| + |1 - 3| = 2
(2 2)	(2 1)	(2 1)	|2 - 2| + |1 - 2| = 1
(2 1)	(2 1)	-	|2 - 2| + |1 - 1| = 0

Table 3.1 – Exemple de mesure du routage XY

La deuxième obligation de preuve consiste à prouver la correction de la route par rapport à la missive m pour laquelle elle est calculée. Une route correcte débute à l'origine de la missive m, et finit par sa destination. Elle ne contient que des nœuds qui appartiennent au *NodeSet*, contient au moins deux nœuds, et la source et la destination sont différentes. Le prédicat *ValidRoutep* vérifie ces conditions :

Définition 3.2 Définition de *ValidRoutep*.

$$ValidRoutep(r,m,NodeSet) \triangleq \begin{cases} & First(r) = Org_{\mathcal{M}}(m) \\ \wedge & Last(r) = Dest_{\mathcal{M}}(m) \\ \wedge & Org_{\mathcal{M}}(m) \neq Dest_{\mathcal{M}}(m) \\ \wedge & r \subseteq NodeSet \wedge Len(r) \geq 2 \end{cases}$$

Toute route retournée par une fonction de routage *RoutingCore* doit satisfaire ce prédicat. L'obligation de preuve suivante doit donc être prouvée :

Obligation de preuve 3.4 Validité des routes produites par *RoutingCore*.
$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet)$
$\Rightarrow \forall m \in \mathcal{M}, \forall r \in RoutingCore(Org_{\mathcal{M}}(m), Dest_{\mathcal{M}}(m)), ValidRoutep(r,m,NodeSet)$

La fonction *Routing*. La fonction *Routing* représente l'algorithme de routage utilisé par la fonction *GeNoC*. Elle prend en entrée une liste de missives $\mathcal{M}$ ainsi que l'ensemble *NodeSet* [3]. Elle retournera une liste de voyages : une liste dans laquelle chaque missive est associée à une liste de routes. Un voyage est construit à partir de l'identifiant, de la trame, et de la route construite à partir de l'origine et de la destination de la missive correspondante. La définition de cette fonction est donc :

3. Dans la définition générique *NodeSet* n'est pas utilisé mais il se trouve que dans certains réseaux le besoin de *NodeSet* se justifie, notamment pour Nostrum.

Définition 3.3 Définition de *Routing*.
$Routing(\mathcal{M}, NodeSet) \triangleq$
$Append_{\forall m \in \mathcal{M}}(List(Id_{\mathcal{M}}(m), Frm_{\mathcal{M}}(m), RoutingCore(Org_{\mathcal{M}}(m), Dest_{\mathcal{M}}(m))))$

Une obligation de preuve sur le type de *Routing* précise que le résultat de la fonction doit être une liste de voyages.

Obligation de preuve 3.5 Validité des voyages produits par *Routing*.
$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet) \Rightarrow \mathcal{V}_{lstp}(Routing(\mathcal{M}, NodeSet))$

La fonction *Routing* préserve les propriétés prouvées sur la fonction *RoutingCore*. La fonction termine, et les routes de chaque voyage satisfont le prédicat *ValidRoutep*. Deux autres propriétés sont nécessaires, illustrées ci-dessous dans les théorèmes 3.2 et 3.3. Les preuves de ces propriétés sont obtenues en se basant uniquement sur les obligations de preuve. Pour toute fonction de routage qui satisfait les obligations de preuve précédentes, la preuve des propriétés ci-dessous est triviale [4].

La première propriété stipule que chaque voyage doit correspondre à une missive unique (le voyage et la missive ont le même identificateur et le même contenu du frame).

Théorème 3.2 Correspondance Voyage/Missive.
$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet) \Rightarrow$
$\forall v \in Routing(\mathcal{M}, NodeSet), \exists! m \in \mathcal{M}, Id_{\mathcal{V}}(v) = Id_{\mathcal{M}}(m) \wedge Frm_{\mathcal{V}}(v) = Frm_{\mathcal{M}}(m)$

Le deuxième théorème signifie que si les voyages produits par la fonction *Routing* sont convertis en missives à l'aide de la fonction *ToMissives*, nous obtiendrons les missives initiales passées en entrée à la fonction *Routing*. La fonction *ToMissives* reconstruit chaque missive en prenant l'identificateur et la trame du voyage. L'origine et la destination de la missive seront respectivement le premier et dernier élément de la route.

Théorème 3.3 Conversion du résultat de *Routing*.
$\forall \mathcal{M}, \mathcal{M}_{lstp}(\mathcal{M}, NodeSet) \Rightarrow ToMissives \circ Routing(\mathcal{M}, NodeSet) = \mathcal{M}$

La preuve de ce théorème est obtenue par les définitions des fonctions *Routing* et *ToMissives* ainsi que par l'obligation de preuve 3.4.

3.8 L'ordonnancement

La politique d'ordonnancement définit l'ensemble des communications qui peuvent être effectuées simultanément. La formalisation de la politique d'ordonnancement est faite par la fonction *Scheduling*. Elle prend une liste des voyages $\mathcal{V}$ (produite par la fonction *Routing*) et la divise en deux listes : une liste des voyages *Scheduled* qui peuvent être ordonnancés, et une autre des voyages retardés *Delayed*. Elle prend en entrée aussi une liste de tentatives qui sera mise à jour selon les décisions d'ordonnancement. *Scheduling* préservera une seule route pour chaque message ordonnancé et effacera les autres routes du voyage correspondant.

4. Notons la différence entre une "obligation de preuve" et un "théorème". Un théorème peut être obtenu à partir des obligations de preuve, alors la propriété ne fait pas partie des contraintes sur la fonction. C'est une conséquence de ces contraintes.

Rappelons que ce modèle original ne prend pas en compte le temps ni les différentes étapes du déplacement d'un message. Il n'est donc pas possible de considérer avec réalisme les éventuels conflits entre messages. La décision de "*faire partir*" les messages est prise à priori, dans un ordre arbitraire. Tout message retenu par *Scheduling* dans la liste *Scheduled* interdira, au cours du même appel récursif de $GeNoC_t$, l'insertion dans cette même liste de tout autre message pouvant présenter un conflit de routes. Cet autre message est alors placé dans la liste *Delayed* pour être considéré ultérieurement, au prochain appel récursif de la fonction $GeNoC_t$.

Pour garantir le bon fonctionnement de cette fonction, nous devons garantir que les deux listes *Scheduled* et *Delayed* sont des listes de voyages si l'entrée de la fonction est une liste de voyages. Ceci constitue la première obligation de preuve sur la fonction *Scheduling* :

Obligation de preuve 3.6 Validité des listes *Scheduled* et *Delayed*.

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

La deuxième obligation de preuve stipule qu'une tentative est consommée à chaque nœud. Ceci se traduit par le fait que *la somme de la liste des tentatives "att" passée en entrée de Scheduling doit être supérieure à la somme de la liste des tentatives "natt" récupérée en sortie de la fonction.* La somme des tentatives est calculée par la fonction *SumOfAtt*.

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

$$SumOfAtt(att) \neq 0 \Rightarrow SumOfAtt(natt) < SumOfAtt(att)$$

Delayed doit être un sous-ensemble de la liste des voyages $\mathcal{V}$ (l'entrée de la fonction *Scheduling*). *Pour chaque voyage "dv" dans Delayed, il existe un seul voyage dans la liste $\mathcal{V}$, tel que l'identificateur, la trame et les routes des deux voyages sont égaux.* Les voyages de *Delayed* seront reconvertis en missives dans l'appel récursif de la fonction *GeNoC*. Le résultat de cette conversion doit être un sous-ensemble de l'ensemble des missives initial passé en tant qu'entrée à la fonction *Routing* d'où la raison pour cette obligation de preuve.

Obligation de preuve 3.8 Correction de *Delayed*.

$$\forall \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}) \Rightarrow \forall dv \in Delayed, \exists! v \in \mathcal{V} \begin{cases} & Id_{\mathcal{V}}(dv) = Id_{\mathcal{V}}(v) \\ \wedge & Frm_{\mathcal{V}}(dv) = Frm_{\mathcal{V}}(v) \\ \wedge & Routes_{\mathcal{V}}(dv) = Routes_{\mathcal{V}}(v) \end{cases}$$

Une obligation similaire existe pour la liste *Scheduled*, cependant pour un message ordonnancé seule la route empruntée par le message est gardée dans la liste des routes possibles. De ce fait, la route d'un message ordonnancé doit simplement être présente dans la liste des routes du voyage dans $\mathcal{V}$.

Obligation de preuve 3.9 Correction de *Scheduled*.

$$\forall \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}) \Rightarrow \forall sv \in Scheduled, \exists! v \in \mathcal{V} \begin{cases} & Id_{\mathcal{V}}(sv) = Id_{\mathcal{V}}(v) \\ \wedge & Frm_{\mathcal{V}}(sv) = Frm_{\mathcal{V}}(v) \\ \wedge & Routes_{\mathcal{V}}(sv) \in Routes_{\mathcal{V}}(v) \end{cases}$$

Si les routes de $\mathcal{V}$ [5] sont correctes, les deux obligations de preuve précédentes nous

5. L'entrée de la fonction *Scheduling*.

permettent de garantir que la correction des routes est conservée pour les deux listes *Scheduled* et *Delayed*.

Finalement, les deux listes doivent être mutuellement exclusives. Un message est ordonnancé ou retardé, il ne peut pas appartenir aux deux listes en même temps. *Les identificateurs des deux listes sont donc comparés pour prouver que leur intersection se réduit à l'ensemble vide.*

Obligation de preuve 3.10 Exclusion mutuelle entre *Scheduled* **et** *Delayed* **.**
$$\mathcal{V}_{ids}(Scheduled) \cap \mathcal{V}_{ids}(Delayed) = \emptyset$$

3.9 Validation de *GeNoC*

La fonction *GeNoC* est définie à l'aide de trois fonctions : $GeNoC_t$, *ComputeMissives*, et *ComputeResults*. *ComputeMissives* utilise la fonction *send* afin de produire les missives à partir des transactions une fois au début de l'exécution du modèle.

Définition 3.4 Définition de *ComputeMissives***.**
$ComputeMissives(\mathcal{T}) \triangleq$
$Append_{\forall t \in \mathcal{T}}(List(Id_{\mathcal{T}}(t), Org_{\mathcal{T}}(t), send(Msg_{\mathcal{T}}(t)), Dest_{\mathcal{T}}(t)))$

ComputeResults construit les résultats à partir des messages arrivés à l'aide de la fonction *recv* à la fin de l'exécution de la fonction $GeNoC_t$.

Définition 3.5 Définition de *ComputeResults***.**
$ComputeResults(\mathcal{V}) \triangleq$
$Append_{\forall v \in \mathcal{V}}(List(Id_{\mathcal{V}}(v), Last(Routes_{\mathcal{V}}(v)), recv(Frm_{\mathcal{V}}(v))))$

La fonction récursive $GeNoC_t$ met en jeu les fonctions *Routing* et *Scheduling*. Elle prend en entrée principalement la liste des missives $\mathcal{M}$ obtenue après le passage par la fonction *ComputeMissives*, les nœuds du réseau *NodeSet*, la liste des tentatives *att* et un accumulateur pour les messages arrivés $\mathcal{V}$ initialement vide. Elle renvoie deux listes à sa terminaison (gouvernée par la condition $SumOfAtt(att)=0$) : une liste des messages arrivés (transformés en résultats dans la fonction *GeNoC*), et une liste des messages avortés.

Définition 3.6 Définition de $GeNoC_t$**.**
$GeNoC_t(\mathcal{M}, NodeSet, att, \mathcal{V}) \triangleq$
If $SumOfAtt(att) = 0$ **then**
  $return\ List(\mathcal{V}, \mathcal{M})$
else
  Let $(ScheduledRtg,\ DelayedRtg,\ natt)$ **be**
    $Scheduling(Routing(\mathcal{M}, NodeSet), att)$ **in**
      $GeNoC_t(ToMissives(DelayedRtg), NodeSet, natt, ScheduledRtg \cup \mathcal{V})$

La fonction *GeNoC* est chargée de calculer les missives, d'appeler la fonction $GeNoC_t$ au départ, et de calculer les résultats à partir des messages arrivés à la fin de l'exécution de $GeNoC_t$. La définition de la fonction est la suivante :

Définition 3.7 Définition de *GeNoC*.
$GeNoC(\mathcal{T}, NodeSet, att) \triangleq$
Let$(A\mathcal{M},\ Aborted)$ **be**
$(GeNoC_t(ComputeMissives\ \mathcal{T})\ , NodeSet\ , att,\ nil)$ **in**
$return\ List(ComputeResults\ (A\mathcal{M})\ Aborted)$

La correction de la fonction $GeNoC_t$ est exprimée dans le théorème 3.4. Il stipule que *pour chaque membre cv de la liste des messages arrivés* $A\mathcal{M}$ *retournée par la fonction* $GeNoC_t$*, il existe* ***une et seulement une*** *missive dans la liste initiale* $\mathcal{M}$ *passée à la fonction en entrée ayant les mêmes identificateur, trame et destinataire (dernier élément de la route dans le cas de cv)*[6].

Théorème 3.4 Correction de $GeNoC_t$.

$$\forall cv \in A\mathcal{M}, \exists! m \in \mathcal{M}, \begin{cases} & Id_{\mathcal{V}}(cv) = Id_{\mathcal{M}}(m) \\ \wedge & Frm_{\mathcal{V}}(cv) = Frm_{\mathcal{M}}(m) \\ \wedge & \forall r \in Routes_{\mathcal{V}}(cv), Last(r) = Dest_{\mathcal{M}}(m) \end{cases}$$

La correction de la fonction *GeNoC*, exprimée dans le théorème 3.1, repose sur ce théorème 3.4 ainsi que sur la définition de la fonction *ComputeResults*.

3.10 Conclusion

Dans ce chapitre nous avons présenté la formalisation initiale pour les réseaux sur la puce. Ces travaux ont été la base sur laquelle les travaux qui seront présentés dans la prochaine partie ont été bâtis. La fonction principale *GeNoC* utilise d'autres fonctions qui représentent les différentes fonctionnalités qui existent dans un réseau. Les fonctions *send* et *recv* modélisent l'encodage et le décodage des messages à l'entrée et à la sortie du réseau. Le routage et l'ordonnancement sont modélisés par les fonctions *Routing* et *Scheduling* respectivement. Chacune de ces fonctions est contrainte par des propriétés (obligations de preuve et théorèmes) suffisantes pour garantir le bon fonctionnement du réseau. La validité de *GeNoC* ne dépend que de ces propriétés. Qui plus est, la modularité du modèle permet des vérifications indépendantes les unes des autres. Ce modèle a été implémenté dans la logique de ACL2 et a été utilisé pour vérifier le routage du réseau Octagon de STMicroelectronics ainsi que le routage XY dans une grille à deux dimensions. Il a aussi été utilisé pour modéliser la commutation par circuit et par paquet, ainsi que l'ordonnancement du bus AMBA AHB. Le protocole Bi-ϕ-M a aussi été vérifié. Dans le chapitre suivant, nous présenterons les extensions et les améliorations apportées durant cette thèse.

6. Les détails de cette preuve peuvent être trouvés dans le manuscrit de thèse de Julien SCHMALTZ pages 71-74.

Deuxième partie

Extension et application du modèle *GeNoC*

Chapitre 4

Modèle *GeNoC* enrichi

Dans le chapitre précédent, nous avons présenté la version initiale du modèle *GeNoC* qui a constitué le point de départ des travaux de cette thèse. Nous allons exposer maintenant le nouveau modèle *GeNoC*. Cette version est la dernière obtenue, plusieurs versions intermédiaires ont été produites en ajoutant progressivement de nouvelles fonctionnalités.

Nous avons expliqué succinctement dans la section 1.2 les limitations de la version initiale du modèle *GeNoC*. A travers ce chapitre, nous montrerons les améliorations apportées. Comme dans le chapitre précédent, nous donnons pour chaque module les principales obligations de preuve. Nous donnerons également un aperçu de la mise en œuvre du modèle dans ACL2 [1]. Nous commençons donc par une très courte présentation d'ACL2.

4.1 Rapide présentation d'ACL2

4.1.1 *A Computational Logic for Applicative Common Lisp*

"*A Computational Logic for Applicative Common Lisp*" ou "*ACL2*", est utilisable en tant que langage de programmation (Common LISP), de logique mathématique formelle, ou de démonstrateur de théorèmes. Qui plus est, les modèles formels des systèmes peuvent être exécutés à des vitesses comparables au langage C ; ceci représente un atout important qui se révèle très utile dans les applications de grande taille. Un modèle d'un système peut ainsi être utilisé à la fois pour des vérifications formelles et peut être exécuté ou simulé [KMM02].

Une mise en œuvre remarquable de modélisation et de vérification grâce à ACL2 est celle de l'algorithme de division du processeur AMD-K5 de Advanced Micro Devices [MLK98]. [KMM00] expose des différents exemples de l'utilisation de ACL2 dans le domaine de la vérification des systèmes complexes. La vérification du micro-code du processeur DSP CAP de Motorola à l'aide de ACL2 en est un exemple. Elle consistait à créer un interpréteur de micro-code sur lequel le micro-code en question était évalué. Dans le processeur AMD-K7, les opérations d'addition, de division, de multiplication et de racine carrée en virgule-flottante ont été vérifiées avec ACL2.

1. Pour l'implémentation entière du modèle générique, consultez l'annexe A.

Dans [BKM96], un modèle ACL2 d'une machine pipelinée à trois étages est défini, ainsi qu'un modèle pour la machine séquentielle correspondante. Une preuve de l'équivalence entre les deux machines est présentée. Une méthode générale applicable à des architectures plus compliquées a été définie dans le cadre de cette vérification.

Différents autres exemples de l'utilisation d'ACL2 dans la vérification des systèmes complexes existent. Russinoff et Flatau ont fourni une méthodologie de vérification pour les systèmes représentés au niveau RTL. La première étape est de traduire automatiquement ces descriptions RTL vers ACL2, puis on vérifie des propriétés sur le modèle ACL2 équivalent. La correction d'un multiplieur à virgule-flottante simple est ainsi prouvée [KMM00]. De plus, Moore expose une méthodologie de simulation symbolique des modèles de processeurs à l'aide d'ACL2 dans [Moo98]. Dans [BY96], Boyer montre la preuve automatique des micro-instructions des processeurs.

4.1.2 La logique de ACL2[KMM02]

Dans ACL2, l'utilisateur peut fournir des définitions de fonctions qui peuvent être simples ou récursives, à l'aide de la commande "*defun*", ainsi que les propriétés à prouver sous forme de théorèmes (à l'aide de la commande "*defthm*"). Lors d'un "événement" (par exemple la preuve d'un théorème, ou la définition d'une fonction), celui-ci est ajouté au monde logique en tant que règle. Il existe un état initial qui ne contient que les règles fournies avec ACL2. Comme son prédécesseur Nqthm [2], ACL2 utilise la logique du premier ordre, sans quantificateur, et avec égalité. ACL2 est non typé, mais des prédicats peuvent être utilisés pour tester le typage des fonctions et des variables.

4.1.2.1 Le moteur de raisonnement

La Figure 4.1 montre l'organisation du démonstrateur. Les formules à prouver sont déposées dans un "pool". La conjecture à prouver est initialement la seule formule dans le pool. Cette formule sera considérée comme le but à prouver. Chacune des méthodes sera utilisée afin d'essayer de prouver le but. La technique utilisée à chaque étape peut réussir à prouver la formule, générer plusieurs sous-formules, ou échouer. Dans le cas de la génération de plusieurs sous-formules, elles seront déposées dans le pool. Chacune sera considérée comme étant un but (sous-but à proprement parler). La preuve du but principal reviendra donc à la vérification de chacun de ces sous buts. Les techniques utilisées successivement sont :

- La simplification : Elle est le cœur du démonstrateur des théorèmes. Les principales méthodes utilisées pour la simplification sont :
  - Les règles du calcul propositionnel, l'égalité, et les procédures de décision de l'arithmétique linéaire rationnelle.
  - Les informations de typage.
  - La réécriture de chaque sous terme dans le contexte approprié en utilisant les définitions, les réécritures conditionnées et les méta-fonctions.
  - L'utilisation de la normalisation du calcul propositionnel.

2. http ://www.computationallogic.com/software/nqthm/

– L'élimination des destructeurs : Des fonctions dites destructives en ACL2 (notamment "CAR" et "CDR"[3]) sont éliminées au profit de nouvelles variables.
Exemple : si dans une fonction les termes (CAR A) et (CDR A) sont utilisés, A est une liste : A sera remplacée par (cons A1 A2), (CAR A) sera remplacé par A1 et (CDR A) par A2.

– L'utilisation des équivalences :
Cette étape essaie d'utiliser les équivalences qui apparaissent dans l'hypothèse pour les appliquer au but. Si la formule contient l'hypothèse (equal LHS RHS), chaque fois que LHS sera rencontré, il pourra être remplacé par RHS.

– La généralisation :
Cette technique généralise le but. L'heuristique de base consiste à trouver un sous terme qui apparaît dans l'hypothèse et la conclusion, ou dans deux hypothèses différentes, ou des deux côtés d'une équivalence puis le remplacer avec une nouvelle variable ce qui généralise la formule.

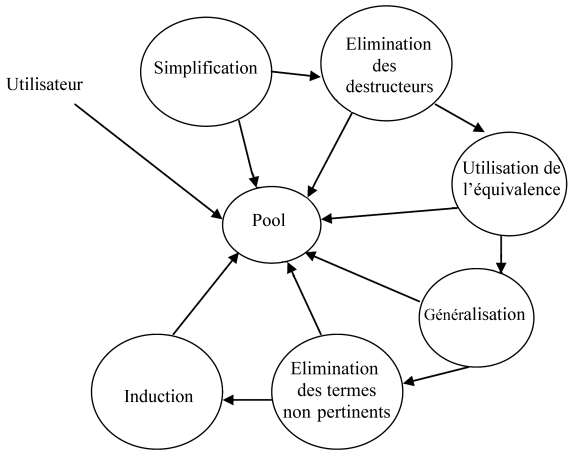

FIGURE 4.1 – Moteur du raisonnement [KMM02]

– L'élimination des termes non pertinents :
Elle est la dernière à être utilisée avant l'induction pour éliminer les hypothèses inutiles (par exemple des hypothèses sur des variables qui ont disparu), en particulier après une étape de généralisation.

– L'induction :
Une induction mathématique est utilisée : il existe un (ou plusieurs) cas de base, et le pas d'induction. Les fonctions récursives utilisées dans la formule à prouver guident le démonstrateur pour choisir les variables sur lesquelles l'induction pourra être effectuée.

3. CAR est une fonction primitive de Common LISP qui renvoie le premier élément d'un CONS, et CDR retourne le reste.

4.1.2.2 Le principe de l'encapsulation

L'encapsulation est un principe d'extension, largement utilisé dans le modèle *GeNoC*, qui permet d'introduire des symboles de fonctions et de construire des théories pour ces fonctions, sans avoir à les définir. Ceci nous permet dans une certaine mesure de simuler le raisonnement et la quantification sur des fonctions. Il permet de présenter une classe de fonctions ayant un ensemble de propriétés en commun. Ces propriétés sont considérées comme des contraintes.

Une macro, *"defspec"* (voir exemple listing 4.1), existe pour la construction d'une telle "encapsulation". Dans le listing 4.1, un seul symbole de fonction f (ici d'arité 3) est introduit. Autant de symboles de fonctions que nécessaire peuvent être introduits.

```
(defspec nom1
 (((f * * *)=>*)) ;;symbole de la fonction et sa signature
  (local
   (defun f (x y z)
            (+ x y z))) ;;définition du témoin
  (defthm ctr1 ...)) ;;Contrainte
```

Listing 4.1: Forme Générique d'un *Defspec*

Pour conserver la cohérence de la logique d'ACL2, la présentation d'un témoin satisfaisant la(les) contrainte(s), est obligatoire. Ceci garantit qu'il existe au moins une fonction qui satisfait les contraintes. Cette fonction sera définie en tant qu'événement local (elle ne sera pas connue en dehors du "defspec" dans lequel elle a été définie.).

L'exemple précédent du *defspec* (listing 4.1) sera admis comme événement et ajouté au monde logique d'ACL2 sous réserve que la fonction témoin f soit admise, et que le théorème *crt1* soit un théorème valide dans le monde logique étendu par la définition de f (il pourrait y avoir plusieurs théorèmes spécifiant les contraintes sur le ou les symboles de fonctions).

Les seules connaissances sur f en dehors du defspec, sont sa signature (plus exactement son arité) et les contraintes définies en tant qu'événements non-locaux. Dans [KM01], Les auteurs précisent que pour l'exemple ci-dessus, tout théorème *thm1* prouvé sur f hors du defspec sera prouvé en se basant sur la contrainte *ctr1*. Ainsi, toute fonction g faisant partie de la classe des fonctions définie par le defspec (qui satisfait la contrainte *ctr1*), satisfera *thm1*. L'utilisation de la macro *"definstance"* permet de vérifier qu'une vraie fonction appartient à la classe de fonctions définie par le "defspec". Ceci est fait en vérifiant les contraintes du "defspec".

Exemple 4.1 Définition et instanciation d'un defspec.
Définissons une classe de fonctions qui reçoivent en paramètre une liste, et renvoient en résultat une liste qui est une permutation du paramètre. La fonction IsPerm est un reconnaisseur de permutation. Elle prend en entrée deux listes et vérifie que la première est une permutation de la deuxième.

Le listing 4.2 montre la spécification de cette théorie sous forme de defspec. Dans la première partie, la signature de la fonction ainsi que le témoin sont définis. Puis, le théorème

Isperm-Fgen spécifie que le résultat renvoyé par la fonction Fgen est une permutation de son entrée x, si x est une liste bien formée.

```
(defspec Permutation
 (((Fgen *)=> *)) ;;signature de la fonction Fgen
 (local
  (defun Fgen (x)
     x)) ;;définition du témoin
 (defthm Isperm-Fgen  ;;Contrainte
   (implies (True-listp x)
            (IsPerm (Fgen x) x))))))
```

Listing 4.2: Exemple d'un *Defspec*

En général, le témoin est choisi le plus simple possible (ici l'identité) pour faciliter les preuves des contraintes. Nous utiliserons dorénavant le terme obligation de preuve pour décrire les contraintes qui doivent être satisfaites pour être une instance d'un "defspec".

Supposons maintenant que nous avons défini une fonction de tri OwnSort et que nous voulons vérifier qu'elle respecte l'obligation de preuve du defspec "Permutation". La construction definstance (listing 4.3) permet de demander l'instanciation du defspec "Permutation" pour la fonction Ownsort à la place de la fonction Fgen. Ceci se fait grâce à la construction :functional-substitution. Lors de l'introduction de ce definstance, ACL2 tente de prouver la véracité du théorème Isperm-Fgen pour Ownsort.

```
(defun Ownsort (x)
 .
 .
 .
 );; définition de la fonction Ownsort

(definstance Permutation Sort-is-valid-perm
   :functional-substitution
      ((Fgen OwnSort)))
```

Listing 4.3: Exemple de l'instanciation d'un defspec

4.2 La nouvelle version du modèle *GeNoC*

La figure 4.2 reprend la figure 3.1 avec une caractérisation du principe de la nouvelle version de *GeNoC*. Nous pouvons voir que de nouveaux modules ont été ajoutés, en particulier le module *état du réseau*. Le module *Scheduling* utilise maintenant deux nouveaux modules : celui de la *synchronisation* et celui de la *priorité*. Chacun de ces nouveaux modules possède ses propres obligations de preuve. Les anciennes obligations sont toujours utilisées (parfois avec des modifications). Une nouvelle propriété de correction du réseau a été ajoutée : *aucun message n'est perdu.* La vérification des propriétés de correction d'un réseau donné s'appuie seulement sur les obligations de preuve de chaque module. Le

principe utilisé pour les instances de réseaux est toujours le même : une fois les obligations des preuves des modules prouvées, la preuve des propriétés de correction en est déduite comme conséquence logique.

Il est à noter qu'un utilisateur peut raisonner sur un réseau à différents niveaux d'abstraction. Si les modules *synchronisation* et *priorité* ne sont pas instanciés, nous raisonnons à un niveau d'abstraction plus élevé où certains détails ne sont pas considérés. Dans ce cas, la preuve de correction du réseau dépend seulement des propriétés prouvées sur les cinq autres modules. Lors de l'utilisation des sept modules, les obligations de preuve du *Scheduling* ne peuvent être vérifiées qu'avec la validation des propriétés des modules de *synchronisation* et *priorité*. Comme la correction du réseau dépend des obligations de preuve du *Scheduling*, elle dépend des modules *synchronisation* et *priorité* par association.

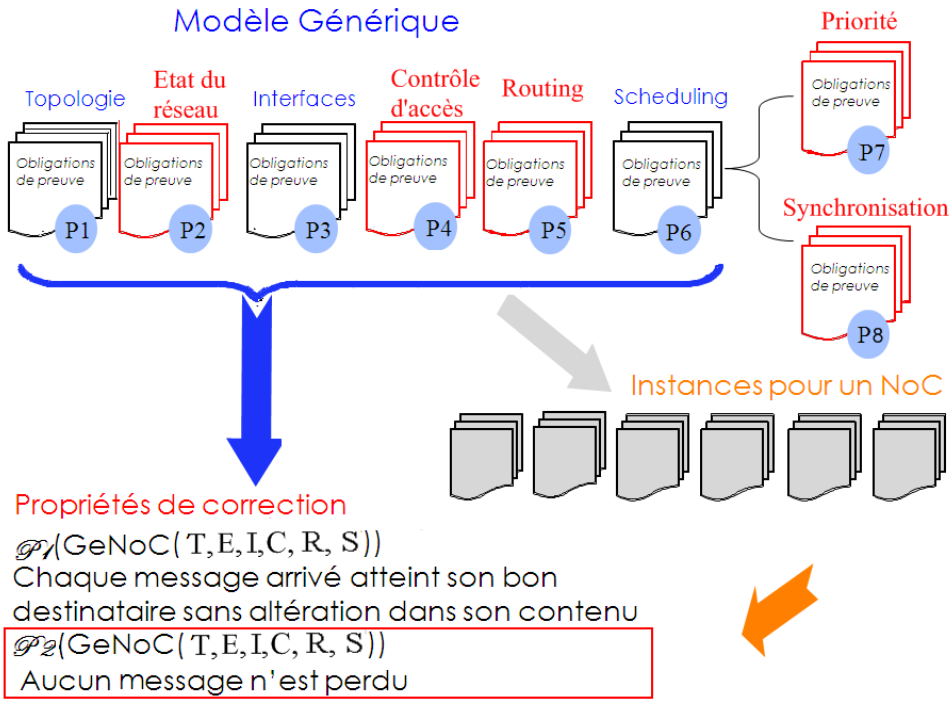

FIGURE 4.2 – Principe de *GeNoC* dans la nouvelle version

La figure 4.3 montre la nouvelle version de l'implémentation du modèle générique *GeNoC* qui permet une modélisation des transmissions pas à pas. La différence majeure entre cette version et celle illustrée dans la figure 3.7, est la granularité du mouvement. Dans la version initiale, la granularité était réduite au déplacement du message de sa source à sa destination en un seul pas. Dans la version présentée dans ce chapitre, la progression des messages dans le réseau est effectuée en avançant le message d'un pas à la fois (*hop*), du nœud courant à un de ses voisins.

L'ancien modèle se positionnait essentiellement dans la couche réseau du modèle OSI : il modélisait l'encodage et le décodage des messages, le routage, et l'ordonnancement du réseau. La nouvelle version implique trois couches du modèle OSI : la couche transport, la couche réseau, et la couche liaison de données. La couche transport nous permet toujours de modéliser l'encodage et le décodage des messages mais aussi le contrôle d'accès au

réseau. La couche réseau couvre le routage, l'ordonnancement ainsi que les politiques de priorités utilisées. Au niveau de la couche liaison de données nous pouvons modéliser la technique de synchronisation entre les routeurs.

La fonction *GeNoC* a été modifiée. Sept fonctions constituent maintenant la modélisation du réseau. Les quatre fonctions de l'ancien modèle (*send*, *recv*, *Routing*, et *Scheduling*) existent toujours. La fonction *Ready4Departure* a été ajoutée, elle représente la politique de contrôle d'accès. La fonction *Priority* (utilisée dans le module *Scheduling*) permet de modéliser la politique de priorité dans les routeurs. La fonction *testroutes* représente le protocole de transmission entre deux nœuds.

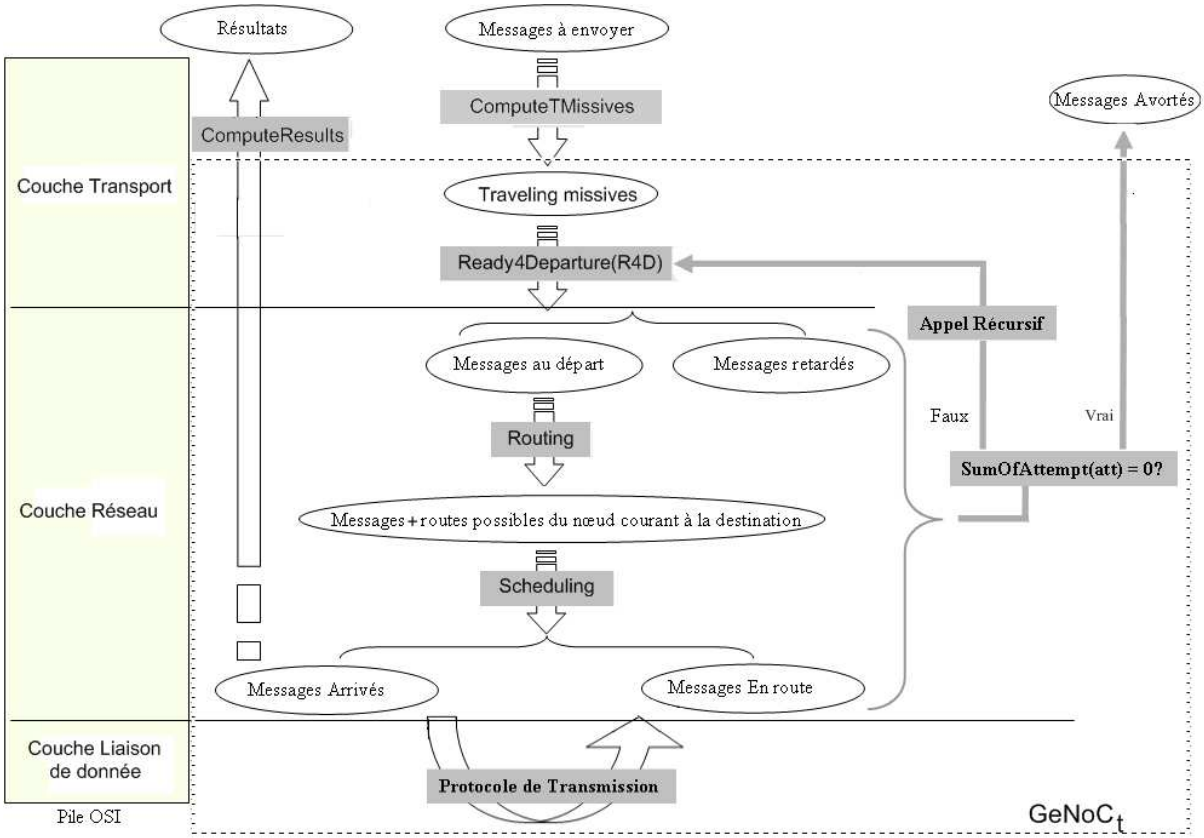

FIGURE 4.3 – Nouveau Modèle *GeNoC*

Les messages au départ sont transformés en traveling missives à l'aide de la fonction *ComputeTMissives* (cette dernière utilise la fonction *send* pour l'encodage). La fonction *Ready4Departure* décide ensuite, pour chaque message, s'il peut être autorisé à circuler (ou à continuer de circuler) dans le réseau. Elle renvoie deux listes de messages : la liste des messages au départ et celle des messages retardés.

Les messages au départ sont passés à la fonction *Routing* qui calcule les routes pour chaque message. La fonction retourne pour chaque message toutes les routes possibles de l'emplacement actuel du message jusqu'à sa destination. Nous montrerons dans la section 5.4.3.1 qu'il pourra être possible d'adapter le raisonnement pour prendre en compte des algorithmes de routage adaptatifs non-minimaux.

Le résultat de la fonction *Routing* est passé à la fonction *Scheduling* qui essaye d'ordonnancer les messages. La fonction *Scheduling* utilise (s'ils sont présents) les modules sur la priorité et la synchronisation entre les nœuds pour prendre ses décisions. Après le traitement de la liste des messages entière, la fonction retourne :

- la liste des messages arrivés *Arrived* qui contient l'ensemble des messages qui ont atteint leur destination.
- la liste des messages *EnRoute* qui contient les messages qui sont encore sur leur chemin pour atteindre leur destination, qu'ils soient bloqués ou en mouvement.

La liste des messages *EnRoute* est utilisée dans l'appel récursif de la fonction $GeNoC_t$ tandis que la liste *Arrived* est accumulée avec les anciens messages arrivés pour construire les *résultats* de la fonction $GeNoC$. A la fin de l'exécution de $GeNoC_t$, la fonction renvoie une liste contenant les messages arrivés à destination et une liste des messages *avortés* qui n'ont pas réussi à atteindre leur destination. Dans la fonction $GeNoC$, la liste des résultats est construite par la fonction *ComputeResults* (qui utilise la fonction *recv* pour décoder les messages).

Pour la preuve de la nouvelle propriété de correction (pas de perte de message), nous devons prouver que l'union des messages avortés et des messages arrivés (les résultats) contient les mêmes messages que ceux injectés au début de l'exécution (les messages à envoyer sur la figure 4.3). Cela ne revient pas à prouver l'égalité entre les deux listes parce que l'ordre des messages a été modifié durant l'exécution. Pour garantir la propriété, il suffit de prouver que l'union de la liste des messages avortés et arrivés est une permutation de la liste des messages à envoyer. Grâce au différentes obligations de preuve qui garantissent la correction des différents modules, nous avons juste besoin de prouver cette propriété sur les identificateurs des messages.

La première étape consiste à prouver que les identificateurs de l'union de la liste des messages au départ et des messages retardés est une permutation des identificateurs de la liste d'entrée de la fonction *Ready4Departure* (obligation de preuve 4.11). Ensuite, nous prouvons que les identificateurs de la sortie de la fonction *Routing* est une permutation des identificateurs de son entrée (obligation de preuve 4.16). Finalement, nous prouvons que les identificateurs de l'union des deux listes à la sortie de la fonction *Scheduling* (messages arrivés et messages en route) sont une permutation des identificateurs de l'entrée de la fonction (obligation de preuve 4.23). Nous avons ainsi prouvé que les identificateurs de l'union des trois listes messages arrivés, en route et retardés est une permutation de l'entrée de la fonction *Ready4Departure* (qui possède les mêmes identificateurs que les messages à envoyer par la définition de la fonction *ComputeTMissives*). La liste des résultats possède les mêmes identificateurs que la liste contenant les messages arrivés (par la définition de la fonction *ComputeResults*). Et la liste des messages avortés est l'union des listes des messages retardés et des messages en route lorsque la fonction s'arrête. Donc, les identificateurs de l'union des listes résultats et messages avortés est une permutation des identificateurs des messages à envoyer.

Dans les sections suivantes, nous exposons les détails de chacun des modules, et ensuite nous montrerons en détails le fonctionnement de $GeNoC$. Nous commencerons en présen-

tant les différents types utilisés dans cette version, en montrant la différence avec l'ancien modèle et en expliquant les raisons de ces ajouts. Puis nous présentons la modélisation des nœuds et de l'état du réseau.

Nous présentons nos différents modules selon la couche OSI correspondante. Les détails de la fonction *Ready4Departure* seront présentés dans la section 4.2.3 (couche transport). La section 4.2.4 contient les détails de l'implémentation de la couche réseau (module *Routing*, celui de l'ordonnancement et le module de la politique de priorité). Et finalement, le module synchronisation (couche liaison de données) sera présenté dans la section 4.2.5. Finalement, la fonction *GeNoC* sera exposée en détails.

Dans la section 3.2, nous avons expliqué brièvement les abstractions utilisées dans la première version de *GeNoC*. Ces mêmes abstractions existent toujours dans la nouvelle version, nous ne les exposons pas à nouveau.

4.2.1 Types

Les types utilisés dans la version initiale existent toujours : les transactions, les voyages et les résultats. Un nouveau type a dû être ajouté : les *traveling missives*. Les tentatives restent exactement identiques à la première version du modèle. Pour les anciens types, de nouveaux champs apparaissent : *Flit*, *Temps* et *hopcount*. Le premier champ permet d'indiquer le nombre de flits dans un message, ceci nous permettra de modéliser les réseaux qui utilisent l'ordonnancement par ver de terre (dont la modélisation est maintenant possible grâce à ce modèle pas-à-pas). Le champ *temps* permet de spécifier l'instant de départ d'un message. En effet, la limitation de l'ancienne version qui consistait à supposer que tout message était expédié à l'instant initial de l'exécution du modèle n'existe plus ici. Ces deux champs seront ajoutés dans tous les types utilisés. Le champ *hopcount* sera ajouté seulement dans les voyages et les traveling missives. Il sert à indiquer le nombre de sauts que le message a effectués (ce qui s'avère utile dans les réseaux qui ont une politique de priorité basée sur le nombre de sauts effectués). Pour tous les types les mêmes contraintes sont encore valides, nous les exposerons en tant que rappel.

Transactions. Les transactions (figure 4.4.b) gardent leur fonctionnalité ainsi que la spécification de la section 3.3 à l'exception des deux champs ajoutés (*flit* et *temps*) qui sont tous les deux des naturels. Le domaine des transactions $\mathcal{D}_{\mathcal{T}}$ sera défini par : $\mathbb{N}$ x *NodeSet* x $\mathcal{D}_{msg}$ x *NodeSet* x $\mathbb{N}$ x $\mathbb{N}$. La définition d'une liste valide de transactions devient donc :

1. l'ensemble des identificateurs des transactions $\mathcal{T}_{ids}$ sont des naturels distincts,
2. les ensembles des nœuds origines et destinations sont des sous-ensembles des nœuds du réseau,
3. pour chaque transaction, l'origine et la destination ne sont pas le même nœud,
4. Les champs *flit* sont des naturels,
5. Les champs *temps* sont des naturels.

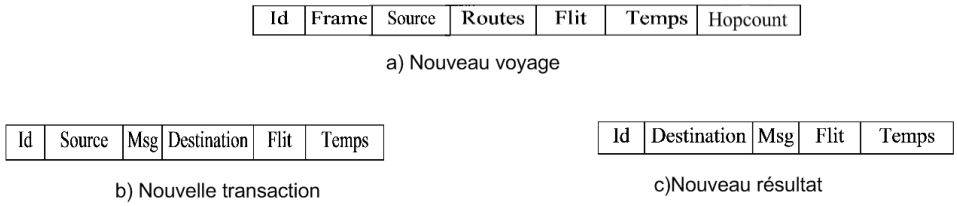

FIGURE 4.4 – Nouveaux Types du modèle *GeNoC*

Voyages. La figure 4.4.a montre la nouvelle forme des voyages. Quatre champs ont été ajoutés *source*, *flit*, *temps*, et *hopcount*. Pour prouver la correction des voyages vis-à-vis des traveling missives dans le module du routage, nous devons vérifier que chaque voyage et sa traveling missive correspondante ont la même source. Dans la version initiale cette preuve de correction s'effectuait entre les voyages et les missives correspondantes. Les routes contenues dans un voyage commençaient toujours par la source du message, or ce n'est plus le cas dans notre version : elles commencent par le nœud courant. Nous avons donc ajouté le champs *source* pour pouvoir vérifier la correction des voyages vis-à-vis des traveling missives correspondantes. Le domaine de définition des voyages est $\mathbb{N}$ x $\mathcal{D}_{frm}$ x *NodeSet* x $\mathcal{D}_{routes}$ x $\mathbb{N}$ x $\mathbb{N}$ x $\mathbb{N}$, et il est noté $\mathcal{D}_{\mathcal{V}}$. Une liste de voyages est notée $\mathcal{V}$ et est reconnue par le prédicat $\mathcal{V}_{lstp}$. Quelques conditions supplémentaires ont été ajoutées au prédicat par rapport à la version initiale. Ce prédicat vérifie maintenant que :

1. La liste des identificateurs de l'ensemble des voyages $\mathcal{V}_{ids}$ est incluse dans $\mathbb{N}$ et ne contient pas de duplication.
2. Les ensembles des nœuds origines (le champ *source*) sont des sous-ensembles des nœuds du réseau.
3. Pour chaque voyage, la liste de routes contient seulement des routes bien formées. Une route bien formée ne contient que des nœuds appartenant à *NodeSet*, sa longueur doit être supérieure ou égale à 2, et le premier élément et le denier de la route doivent être différents.
4. Les champs *flit* sont des naturels.
5. Les champs *temps* sont des naturels.
6. Les champs *hopcount* sont des naturels.

Résultats. Le domaine de définition d'un résultat $\mathcal{D}_{\mathcal{R}}$ (figure 4.4.c) est $\mathbb{N}$ x *NodeSet* x $\mathcal{D}_{msg}$ x $\mathbb{N}$ x $\mathbb{N}$. Les résultats sont reconnus par le prédicat $\mathcal{R}_{lstp}$, qui vérifie que :

1. la liste des identificateurs de l'ensemble des résultats $\mathcal{R}_{ids}$ est incluse dans $\mathbb{N}$ et ne contient pas de duplication.
2. la destination de chacun des résultats appartenant à la liste doit appartenir à *NodeSet*.
3. Les champs *flit* sont des naturels,
4. Les champs *temps* sont des naturels.

Traveling Missives. Ce type a dû être ajouté afin de permettre la modélisation pas-à-pas. Une *traveling missive* est illustrée dans la figure 4.5. Elle contient quatre champs

de plus que la missive : le champ *courant*, le champ flit, le champ temps, et le champ *hopcount*. *Courant* contient l'adresse du nœud où le message réside à l'itération courante. Le domaine des traveling missives est donc $\mathbb{N}$ x *NodeSet* x *NodeSet* x $\mathcal{D}_{frm}$ x *NodeSet* x $\mathbb{N}$ x $\mathbb{N}$ x $\mathbb{N}$. Le domaine de définition des traveling missives est noté $\mathcal{D}_{\mathcal{TM}}$. Le prédicat $\mathcal{TM}_{lstp}$ reconnait une liste valide de traveling missives, il vérifie que :

1. l'ensemble des identificateurs $\mathcal{TM}_{ids}$ contient des naturels distincts,
2. les ensembles des nœuds origines, courants et destinations sont des sous-ensembles des nœuds du réseau,
3. l'origine et la destination de chaque message ne sont pas le même nœud,
4. les champs courant et destination de chaque message sont différents,
5. Les champs *flit* sont des naturels,
6. Les champs *temps* sont des naturels,
7. Les champs *hopcount* sont des naturels.

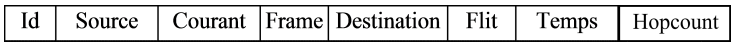

FIGURE 4.5 – Une traveling missive

4.2.2 Nœuds et état du réseau

L'ensemble des nœuds du réseau garde la modélisation de l'ancien modèle. Les obligations de preuves sont identiques. Nous ne détaillerons pas ce point.

Un nouveau module correspondant à l'état du réseau a été ajouté. Il représente l'état global du réseau et la disponibilité des ressources, *i.e.* les éléments mémorisants. Pour chacun des ports d'un nœud, une entrée dans l'état du réseau est créée, qui représente l'état local du port. La collection de ces états locaux des ports forme l'état global du réseau. Une forme à respecter a été définie, et l'état de chaque port sera de la forme : *((Coor (...)) (Buffers ...) (Optional))*. Le mot clef *Coor* introduit les coordonnées (l'identificateur) du port/nœud. Le champ *Buffers* représente les éléments mémorisants du port et leur état. Le champ *Optional* pourra permettre de prendre en compte des informations supplémentaires dans le cas où le module de synchronisation est utilisé [4]. Dans le cas où les détails de synchronisation ne sont pas modélisés (niveau d'abstraction plus élevé, cf. page 48), seuls les champs $Coor$ et $Buffers$ seront utilisés.

La fonction *BasicEntryp* [5] permet de contrôler la correction de l'état par rapport à la représentation de base d'un état local valide d'un nœud (chaque état local doit contenir les champs $Coor$ et $Buffers$). Elle utilise les prédicats *validCoordp* et *validBufferp*. La fonction *validCoordp* vérifie que la partie coordonnées est bien formée tandis que *validBufferp* s'assure de la bonne formation de la partie modélisant l'état des éléments mémorisants.

4. Nous verrons par exemple dans la section 5.2 l'utilisation du champ *Optional* pour représenter des signaux de synchronisation.

5. La fonction *First* retourne le premier élément de son unique paramètre d'entrée, et *Second* renvoie le deuxième.

Définition 4.1 Définition de *BasicEntryp*.
$$BasicEntryp(e) \triangleq validCoordp(First(e)) \wedge validBufferp(Second(e))$$

La possibilité d'utilisation d'un champ *Optional* nécessite de pouvoir étendre la définition d'un état valide. Nous considérons donc le prédicat vérifiant la validité d'un état local est *ValidEntryp*, et nous lui associons l'obligation de preuve 4.1 qui permet de garantir que la forme de base est respectée. Nous allons dans la suite caractériser un état global valide par le prédicat *Validstatep*, où il sera admis que cette fonction est définie selon le modèle de la définition 4.2.

Obligation de preuve 4.1 Validité de l'état local.
$$ValidEntryp(ntkstate) \Rightarrow BasicEntryp(ntkstate)$$

Définition 4.2 Définition de *Validstatep*.
$$Validstatep(ntkst) \triangleq \bigwedge_{ent \in ntkst} ValidEntryp(ent)$$

Quatre fonctions servent à la manipulation de l'état du réseau. Ces fonctions ne sont pas définies mais elles sont contraintes par des obligations de preuve.

- La fonction *StateGenerator* génère l'état global initial du réseau. Elle construit pour chaque port/nœud une entrée de la forme spécifiée ci-dessus, en initialisant les différents champs à leurs valeurs initiales. Elle prend en entrée deux paramètres : le premier spécifie la taille du réseau, et le deuxième représente le nombre de places de mémorisation dans les ports/nœuds.
- La fonction *ValidstateParamsp* vérifie que les paramètres passés à la fonction précédente sont valides.
- La fonction *loadbuffers* permet de remplir ou de libérer une place dans le buffer d'un port (ou d'un nœud). La fonction prend trois paramètres en entrée, l'adresse du nœud considéré, un message et l'état actuel du réseau. La fonction met le message qu'elle a reçu en entrée dans les buffers du nœud considéré dans l'état actuel du réseau en marquant que le nombre de places libres a diminué. Si le deuxième paramètre est *nil*, la fonction libère une place dans les buffers du nœud qu'elle a reçu comme premier paramètre. Cette libération est reflétée sur l'état actuel du réseau en enlevant le contenu qui était dans la place libérée et en augmentant le nombre de places vides. Dans les deux cas, le nouvel état du réseau résultant de ces opérations est renvoyé en résultat de la fonction.
- La fonction *readbuffers* permet de consulter l'état d'un nœud (port) du réseau. Cette dernière fonction prendra deux paramètres en entrée : l'adresse du nœud (port) à consulter et l'état actuel du réseau. Elle consulte l'entrée du nœud considéré (adresse reçue en entrée) dans l'état actuel du réseau, et le renvoie en résultat.

La première obligation de preuve vérifie que le résultat de la fonction *StateGenerator* est un état valide. La deuxième et la troisième sont utilisées respectivement pour vérifier que les résultats obtenus par les fonctions *loadbuffers* et *readbuffers* sont du bon type.

Obligation de preuve 4.2 Validité de l'état généré par *StateGenerator*.
$$ValidstateParamsp(pm1, pm2) \Rightarrow Validstatep(StateGenerator(pm1, pm2))$$

L'obligation suivante consiste à vérifier que : si l'état actuel *ntkst* du réseau est valide et *adr* (l'adresse du nœud où le message *msg* réside) est valide, le résultat obtenu par la fonction *loadbuffers* est un nouvel état valide.

Obligation de preuve 4.3 Validité du résultat de la fonction *loadbuffers*.
$$Validstatep(ntkst) \wedge adr \in NodeSet \Rightarrow Validstatep(loadbuffers(adr, msg, ntkst))$$

Le résultat de la fonction *readbuffers* doit être une entrée valide d'état local d'un nœud. L'obligation de preuve suivante vérifie : si *ntkst* est valide et l'adresse *adr* appartient à l'ensemble des adresses du réseau *NodeSet*, alors le résultat de *readbuffers* est un état local valide d'un nœud.

Obligation de preuve 4.4 Validité du résultat de la fonction *readbuffers*.
$$Validstatep(ntkst) \wedge adr \in NodeSet \Rightarrow ValidEntryp(readbuffers(adr, ntkst))$$

Les deux dernières obligations de preuve sont les plus importantes : elles nous permettent de faire le lien entre *NodeSet* et l'état du réseau. La première obligation spécifie que si les paramètres de la génération de l'état initial sont valides, le premier sera donc un paramètre valide pour la génération de *NodeSet* [6].

Obligation de preuve 4.5 Correspondance entre la validité des paramètres.
$$ValidstateParamsp(pm1, pm2) \Rightarrow ValidParamsp(pm1)$$

La dernière obligation de preuve spécifie que l'ensemble des coordonnées dans l'état du réseau est identique à celui généré avec la fonction *NodeSetGen*. L'ensemble des coordonnées de l'état est extrait à l'aide de la fonction *getcoordinates*.

Obligation de preuve 4.6 Correspondance entre *StateGenerator* **et** *NodeSetGen*.
$$ValidstateParamsp(pm1, pm2)$$
$$\Rightarrow getcoordinates(StateGenerator(pm1, pm2)) = NodeSetGen(pm1)$$

4.2.3 Couche transport - Contrôle d'accès au réseau

Certains systèmes de communications possèdent une méthode pour contrôler l'accès des messages au réseau. La spécification de cette méthode est le rôle du module que nous présentons dans cette section. La politique de contrôle d'accès est modélisée par la fonction *Ready4Departure*. Dans le cas le plus simple, il peut s'agir seulement de vérifier que le temps courant est le temps de départ du message. Plus généralement, nous pourrons avoir un vrai contrôle d'accès, comme par exemple un contrôle de flux par crédits.

Ready4Departure prend en entrée 4 paramètres : une liste de traveling missives $\mathcal{TM}$, deux accumulateurs vides au départ, et un paramètre utilisé pour le contrôle d'accès (le temps courant par exemple). Le premier accumulateur *traveling* sera utilisé pour les messages qui peuvent accéder au réseau, tandis que le deuxième *delayed* est pour les messages qui verront leur accès au réseau refusé. La fonction effectue un test sur chaque message dans la liste des traveling missives, et selon le résultat du test il est ajouté à l'un des deux

6. Les fonctions *ValidParamsp* et *NodeSetGen* ont été décrites en détails dans la section 3.5.

accumulateurs. Si le message peut partir, il est ajouté à *traveling* sinon à *delayed*.

Plusieurs obligations de preuve sont nécessaires pour cette fonction. La première consiste à vérifier que si le premier paramètre $\mathcal{TM}$ est bien une liste de traveling missives, les deux listes *traveling* et *delayed* seront de ce même type. La deuxième et la troisième obligation de preuve vérifient que les éléments des deux listes de sortie proviennent de la liste $\mathcal{TM}$.

Obligation de preuve 4.7 Validité des listes *traveling* et *delayed*.
$$\forall \mathcal{TM}, \mathcal{TM}_{lstp}(\mathcal{TM}, NodeSet) \Rightarrow$$
$$\mathcal{TM}_{lstp}(traveling, NodeSet) \wedge \mathcal{TM}_{lstp}(delayed, NodeSet)$$

Obligation de preuve 4.8 Correction de *delayed*.
$$\forall \mathcal{TM}, \mathcal{TM}_{lstp}(\mathcal{TM}, NodeSet) \Rightarrow \forall dtm \in delayed, \exists! tm \in \mathcal{TM}, dtm = tm$$

Obligation de preuve 4.9 Correction de *traveling*.
$$\forall \mathcal{TM}, \mathcal{TM}_{lstp}(\mathcal{TM}, NodeSet) \Rightarrow \forall deptm \in traveling, \exists! tm \in \mathcal{TM}, deptm = tm$$

Une propriété importante est qu'un message ne peut pas être dans les deux listes en même temps.

Obligation de preuve 4.10 Exclusion mutuelle entre *traveling* et *delayed* .
$$traveling \cap delayed = \emptyset$$

Finalement, une obligation de preuve importante consiste à vérifier que l'union des identificateurs des deux listes est une permutation de la liste des identificateurs des messages d'entrée de la fonction *Ready4Departure*. La fonction *IsPerm* prend deux paramètres en entrée et vérifie que le premier est une permutation du deuxième. La fonction $\mathcal{TM}_{ids}$ retourne les identificateurs de son paramètre. Cette obligation de preuve est indispensable pour la preuve de la nouvelle propriété de correction que nous avons introduite avec ce modèle amélioré.

Obligation de preuve 4.11 Permutation des identificateurs de $\mathcal{TM}$.
$$IsPerm(\mathcal{TM}_{ids}(\mathcal{TM}), \mathcal{TM}_{ids}(traveling) \cup \mathcal{TM}_{ids}(delayed))$$

4.2.4 Couche réseau

4.2.4.1 Routage

Nous avons montré, dans la section 4.1, un exemple de l'utilisation du "defspec" pour définir une fonction générique et puis nous l'avons instancié pour créer une instance d'un système simple. Le module du routage servira comme exemple pour décrire la mise en œuvre du modèle dans ACL2. Nous exposerons les obligations de preuve sous forme logique et puis sous forme de code ACL2. Remarquons que la même méthodologie de modélisation est utilisée pour chacun des différents modules de notre modèle générique présentés dans ce chapitre.

Une modification dans ce module du routage par rapport à la version du départ, nous a permis de modéliser l'exécution pas-à-pas. Dans la version de départ du modèle, la fonction *Routing* calculait la route entre l'origine du message et sa destination. Dans notre

version, les routes sont calculées à partir du nœud courant (le champ *courant* d'une traveling missive) jusqu'à la destination du message.

Nous utilisons la même modélisation que dans le chapitre précédent pour la fonction de routage. La fonction $RoutingLogic(x, y)$ modélise la logique de routage responsable de calculer la prochaine étape tout au long du chemin entre deux nœuds x et y. Pour obtenir la route entière, cette fonction est appliquée successivement jusqu'à atteindre la destination. *RoutingCore* (définition 4.3) applique la fonction *RoutingLogic* à partir du nœud courant jusqu'à la destination pour obtenir la route entière.

Définition 4.3 Définition de *RoutingCore*.

$$RoutingCore(c, d) \triangleq \begin{cases} d & \textbf{\textit{si}}\ c = d \\ append(c, RoutingCore(RoutingLogic(c, d), d)) & \textbf{\textit{sinon}} \end{cases}$$

La fonction *RoutingCore* étant récursive, la première obligation de preuve consiste à prouver sa terminaison. Cette preuve est effectuée de la même façon que dans le chapitre précédent. Considérons un ensemble S muni d'une relation bien fondée $\prec_S$ (dans la plupart des cas, il s'agit de l'ensemble des entiers naturels $\mathbb{N}$). Nous définissons la fonction *measure*. Elle est de type $NodeSet \times NodeSet \rightarrow S$. Elle est définie de telle façon que sa valeur décroisse à chaque appel récursif. Il faut donc garantir que $measure(RoutingLogic(c, d), d) \prec_S measure(c, d)$ dans le cas $c \neq d$, d'où l'obligation de preuve suivante.

Obligation de preuve 4.12 Terminaison de *RoutingCore*.

$$\forall c, d \in NodeSet, \exists measure : NodeSet \times NodeSet \rightarrow S,$$
$$c \neq d \Rightarrow measure(RoutingLogic(c, d), d) \prec_S measure(c, d)$$

Lors de la définition d'une fonction récursive (comme *RoutingCore*) dans ACL2, ce dernier essaye de prouver sa terminaison automatiquement. Si la preuve réussit, la fonction est acceptée sinon elle est refusée. Lors de la définition des instances de la fonction *RoutingCore*, ACL2 tâche donc de prouver cette obligation sans que l'utilisateur le demande (il pourra toutefois être amené à fournir la mesure, si celle-ci n'est pas triviale). Grâce à cette propriété de ACL2, nous verrons que dans le defspec associé au module *Routing* cette obligation de preuve n'est pas explicitée.

Une fois la terminaison prouvée, nous devons prouver la correction de la route par rapport à la traveling missive *tm* pour laquelle elle est calculée. Une route correcte débute à l'emplacement courant de *tm*, et finit par sa destination. Elle ne contient que des nœuds qui appartiennent au *NodeSet*, contient au moins deux nœuds, la source et la destination sont différentes, et le nœud courant et la destination sont différents. Le prédicat *ValidRoutep* qui vérifie ces conditions devient donc :

Définition 4.4 Définition de *ValidRoutep*.

$$ValidRoutep(r, tm, NodeSet) \triangleq \begin{cases} & First(r) = curr_{\mathcal{TM}}(tm) \\ \wedge & Last(r) = Dest_{\mathcal{TM}}(tm) \\ \wedge & curr_{\mathcal{TM}}(tm) \neq Dest_{\mathcal{TM}}(tm) \\ \wedge & Org_{\mathcal{TM}}(tm) \neq Dest_{\mathcal{TM}}(tm) \\ \wedge & r \subseteq NodeSet \wedge Len(r) \geq 2 \end{cases}$$

La fonction *RoutingTop* utilise la fonction *RoutingCore* pour calculer l'ensemble des routes possibles entre la position actuelle d'un message et sa destination. Elle prend en entrée le nœud courant et la destination du message. Il existe deux types de routage comme nous l'avons expliqué dans le chapitre 2 : les routages déterministes et les routages adaptatifs. Dans le cas d'un routage déterministe, une seule route est calculée à l'aide de la fonction *RoutingCore*. Dans le cas adaptatif, la fonction *RoutingTop* calcule plusieurs routes en invoquant toutes les variantes nécessaires de la fonction *RoutingCore*. Elle présente ces routes sous forme d'une liste de routes en résultat.

Comme dans la première version du modèle, toute route doit satisfaire le prédicat *ValidRoutep* ci-dessus. L'obligation de preuve suivante doit être prouvée. Elle stipule que pour chaque *tm* membre de la liste $\mathcal{TM}$, toute route calculée par la fonction de routage *RoutingTop* doit satisfaire le prédicat *ValidRoutep* [7].

Obligation de preuve 4.13 Validité des routes produites par *RoutingTop*.
$\forall \mathcal{TM}, \mathcal{TM}_{lstp}(\mathcal{TM}, NodeSet)$
$\Rightarrow \forall tm \in \mathcal{TM}, \forall r \in RoutingTop(curr_{\mathcal{TM}}(tm), Dest_{\mathcal{TM}}(tm)), ValidRoutep(r, tm, NodeSet)$

La fonction *Routing*. La fonction *Routing* représente l'algorithme de routage utilisé par la fonction *GeNoC*. Dorénavant, elle prend comme paramètre d'entrée une liste de traveling missives $\mathcal{TM}$ ainsi que l'ensemble *NodeSet*. Elle retournera une liste de voyages. Un voyage est construit à partir de l'identifiant, de l'origine, de la trame, de la(les) route(s) construite(s) à partir du nœud courant et de la destination, le nombre de flit, le temps et le hopcount de la traveling missive correspondante. La définition de cette fonction est :

Définition 4.5 Définition de *Routing*.
$Routing(\mathcal{TM}, NodeSet) \triangleq$
$Append_{\forall tm \in \mathcal{TM}}(List(Id_{\mathcal{TM}}(tm), Org_{\mathcal{TM}}(tm), Frm_{\mathcal{TM}}(tm), RoutingTop(curr_{\mathcal{TM}}(tm),$
$Dest_{\mathcal{TM}}(tm)), Flit_{\mathcal{TM}}(tm), Time_{\mathcal{TM}}(tm), Hopcount_{\mathcal{TM}}(tm)))$

Une obligation de preuve sur le type de *Routing* précise que le résultat de la fonction doit être une liste de voyages.

Obligation de preuve 4.14 Validité des voyages produits par *Routing*.
$\forall \mathcal{TM}, \mathcal{TM}_{lstp}(\mathcal{TM}, NodeSet) \Rightarrow \mathcal{V}_{lstp}(Routing(\mathcal{TM}, NodeSet), NodeSet)$

Nous devons prouver maintenant la correction de la fonction de routage. Il s'agit de prouver que les voyages calculés par la fonction de routage correspondent aux traveling missives que la fonction a reçues en entrée. Nous vérifions donc que pour chaque voyage il existe une traveling missive avec le même identificateur, origine, contenu du message, nombre de flit, temps, et hopcount.

7. Comme nous avons mentionné dans le chapitre précédent, l'utilisation de *NodeSet* désignera en fait *NodeSetGen*(*pms*) et sous-entendra l'ajout de l'hypothèse *ValidParamsp*(*pms*) dans la modélisation ACL2.

Obligation de preuve 4.15 Correspondance Voyage/Missive.
$\forall \mathcal{TM}, \mathcal{TM}_{lstp}(\mathcal{TM}, NodeSet) \Rightarrow$

$$\forall v \in Routing(\mathcal{TM}, NodeSet), \exists! tm \in \mathcal{TM}, \begin{cases} & Id_{\mathcal{V}}(v) = Id_{\mathcal{TM}}(tm) \\ \wedge & Org_{\mathcal{V}}(v) = Org_{\mathcal{TM}}(tm) \\ \wedge & Frm_{\mathcal{V}}(v) = Frm_{\mathcal{TM}}(tm) \\ \wedge & Flit_{\mathcal{V}}(v) = Flit_{\mathcal{TM}}(tm) \\ \wedge & Time_{\mathcal{V}}(v) = Time_{\mathcal{TM}}(tm) \\ \wedge & Hopcount_{\mathcal{V}}(v) = Hopcount_{\mathcal{TM}}(tm) \end{cases}$$

Dans le cadre de la définition de la nouvelle version du modèle, une nouvelle obligation de preuve a été ajoutée : la liste des identificateurs des voyages résultants de la fonction de routage est une permutation des identificateurs de la liste de traveling missives passée à la fonction en entrée.

Obligation de preuve 4.16 Permutation des identificateurs du Routage.
$\mathcal{TM}_{lstp}(\mathcal{TM}, NodeSet) \Rightarrow IsPerm(\mathcal{TM}_{ids}(\mathcal{TM}), \mathcal{V}_{ids}(Routing(\mathcal{TM}, NodeSet))$

La fonction *Routing* préserve les propriétés prouvées sur la fonction *RoutingTop*. La fonction termine, et les routes de chaque voyage satisfont le prédicat *ValidRoutep*. Le théorème 4.1 signifie que si les voyages produits par la fonction *Routing* sont reconvertis en traveling missives à l'aide de la fonction *ToTravMis*, nous obtiendrons les traveling missives initiales passées en entrée à la fonction *Routing*. La fonction *ToTravMis* reconstruit chaque missive en prenant l'identificateur, l'origine, la trame du voyage, le nombre de flit, le champ temps et le nombre de sauts *hopcount*. Le nœud courant et la destination de la traveling missive seront respectivement le premier et dernier élément de la route. La preuve de ce théorème est obtenue par les définitions des fonctions *Routing* et *ToTravMis* ainsi que par l'obligation de preuve 4.13.

Théorème 4.1 Conversion du résultat de *Routing*.
$\forall \mathcal{TM}, \mathcal{TM}_{lstp}(\mathcal{TM}, NodeSet) \Rightarrow ToTravMis \circ Routing(\mathcal{TM}, NodeSet) = \mathcal{TM}$

Nous présentons maintenant la modélisation de ce module dans ACL2. Il s'agit de la mise en œuvre du module du *Routing* avec ses obligations de preuve P4 de la figure 4.2. Cette implémentation, comme celle des autres modules, repose sur le principe d'encapsulation décrit dans la section 4.1.2.2. Le "defspec" correspondant est donné dans le listing 4.4. Nous précisons d'abord que la fonction *Routing* admet deux paramètres. Un témoin local au defspec est ensuite fourni, cette fonction *Routing* fait appel à une fonction auxiliaire *RoutingTop* qui utilise la fonction *RoutingCore* (les deux fonctions sont également définies localement au "defspec"). La définition de cette dernière est la plus simple possible avec une route uniquement formée du nœud courant et de la destination du message (il s'agit du routage dans un simple bus).

Une fois le témoin défini, nous introduisons les contraintes liées à la fonction *Routing*. L'obligation de preuve 4.13 correspond au théorème *TrLstp-routing*. Elle vérifie que si les paramètres passés à la fonction *Routing* sont bien typés, le résultat de la fonction est une liste de voyages bien formée. La fonction $\mathcal{V}_{lstp}$ est appelé *Trsltp* dans le code ACL2 tandis que la fonction $\mathcal{TM}_{lstp}$ porte le nom *TMissivesp*. Nous remarquons la caractérisation de *NodeSet*, comme nous l'avons mentionné plutôt.

```
(defspec GenericRouting
 (((Routing * *) => *))

 (local
  (defun RoutingCore (current destination)
    ;; Le routage dans un bus
   (list current destination)))

 (local
  (defun RoutingTop (current destination)
   (list (RoutingCore current destination))))

 (local
   (defun Routing (TM Nodeset)
           (declare (ignore Nodeset))
           (if (endp TM)
               nil
            (let* ((msv (car TM))
                   (Id (IdTM msv))
                   (frm (FrmTM msv))
                   (origin (OrgTM msv))
                   (current (CurTM msv))
                   (destination (DestTM msv))
                   (flit (FlitTM msv))
                   (time (TimeTM msv))
                   (hopcount (HopcountTM msv)))
              (cons (list Id origin frm
                          (RoutingTop current destination)
                          Flit time hopcount)
                    (Routing (cdr TM) Nodeset))))))

(defthm TrLstp-routing ;; obligation de preuve 4.13
  (let ((NodeSet (NodeSetGenerator Params)))
    (implies (and (TMissivesp TM NodeSet)
                  (ValidParamsp Params))
          (TrLstp (Routing TM NodeSet) NodeSet))))

(defthm Routing-CorrectRoutesp
;; ce théorème correspond aux obligations
;; de preuve 4.12 et 4.14
   (let ((NodeSet (NodeSetGenerator Params)))
     (implies (and (TMissivesp TM NodeSet)
                   (ValidParamsp Params))
              (CorrectRoutesp (Routing TM NodeSet) TM NodeSet))))

 (defthm Is-perm-routing
 ;;obligation de preuve 4.15
       (implies (tmissivesp TM nodeset)
                (isperm (V-ids (routing TM nodeset))
                        (tm-ids m)))))
```

Listing 4.4: Définition du defspec *Routing*

Nous avons défini trois fonctions pour faciliter les preuves et la modélisation des obligations de preuve 4.12 et 4.14 (associées au théorème *Routing-CorrectRoutesp*). La fonction *CorrectRoutesp* (listing 4.5) prend trois paramètre, une liste de voyages, une liste de traveling missives et l'ensemble des nœuds du réseau. Pour chaque voyage et sa traveling missive correspondante, elle vérifie les égalités de l'obligation de preuve 4.14, puis elle appelle la fonction *Checkroutes* en lui passant l'ensemble des routes du voyage.

La fonction *Checkroutes* prend trois paramètres en entrées : la liste des routes d'un voyage, la traveling missive correspondante au voyage, et l'ensemble des nœuds du réseau. Cette fonction est récursive, elle invoque la fonction *ValidRoutep* pour chacune des routes qu'elle a reçues en entrée. La fonction *ValidRoutep* prend une route, une traveling missive, et l'ensemble des nœuds du réseau et vérifie les conditions de la définition 4.4.

```
(defun Validroutep (r tm Nodeset)
 (and (equal (car r) (curTM tm))
      (equal (car (last r)) (destTM tm))
      (not (equal (car r) (car (last r))))
      (not (equal (OrgTM tm) (car (last r))))
      (<= 2 (len r)))
      (subsetp r Nodeset))

(defun Checkroutes (routes tm Nodeset)
 (if (endp route)
     t
    (let ((r (car routes)))
         (and (Validroutep r tm Nodeset)
              (Checkroute (cdr routes) tm Nodeset))

(defun CorrectRoutesp (TrLst TM NodeSet)
 (if (endp TrLst)          ;;Si la liste des voyages est finie
     (if (endp TM)         ;;la liste de traveling missives
         t                 ;;doit être finie aussi
         nil)
     (let* ((v (car TrLst))
            (tms (car TM))
            (routes (RoutesV v)))
           (and (equal (IdV v) (IdTM tms))
                (equal (OrgV v) (OrgTM tms))
                (equal (FrmV v) (FrmTM tms))
                (equal (FlitV v) (FlitTm tms))
                (equal (timeV v) (TimeTm tms))
                (equal (HopcountV tr) (hopcountTM tms))
                (CheckRoutes routes tms NodeSet)
                (CorrectRoutesp (cdr TrLst) (cdr TM) NodeSet)))))
```

Listing 4.5: Définition de *ValidRoutep*, *CheckRoutes* et *CorrectRoutesp*

Finalement, l'obligation de preuve 4.15 est associée au théorème *Is-perm-routing*. Ceci conclut notre module de routage. A noter que les preuves de ces contraintes pour la fonction témoin ont nécessité l'introduction de quelques lemmes intermédiaires, volontairement omis ici pour simplifier l'exposé.

4.2.4.2 L'ordonnancement

Les propriétés de correction du module de l'ordonnancement ont été modifiées pour réaliser la modélisation pas-à-pas. Qui plus est, il est maintenant possible de prendre en compte les schémas de priorité utilisés par les arbitres pour résoudre les compétitions entre les messages pour l'obtention des ressources des nœuds. Nous faisons la différence entre les priorités dépendantes et indépendantes de l'état courant du réseau. Certains mécanismes, comme round robin, attribuent systématiquement une priorité à chaque port, quel que soit l'état courant. Dans certains cas, le mécanisme de priorité peut tenir compte des caractéristiques des messages dans l'état courant (nous verrons avec l'exemple de Nostrum que "l'âge" des messages est pris en compte).

Pour le cas où l'état n'a pas à être considéré, une nouvelle fonction *getnextpriority* a été ajoutée au module d'ordonnancement. La fonction prend en entrée un paramètre qui spécifie l'ordre de priorité entre les ports dans le cycle d'exécution courant (ou un élément qui permet de le déduire comme le port le plus prioritaire), et calcule une relation d'ordre qui sera utilisée par la fonction *PrioritySort* pour faire un tri des messages suivant cet ordre (cf. section 4.2.4.3). Elle ne sera instanciée que dans le cas où le réseau utilise ce type de priorité.

La fonction principale demeure *Scheduling*. Elle prend cinq paramètres en entrée :
- $\mathcal{V}$: la liste des voyages résultants du module du routage,
- *att* : la liste des tentatives,
- *NodeSet* : l'ensemble des adresses valides du réseau,
- *NtkState* : l'état actuel du réseau pour consulter l'état des nœuds et décider si un ordonnancement est possible,
- *Order* : une liste qui contient les ports dans leur ordre décroissant de priorité, ou bien le port le plus prioritaire (dans le cas où la relation d'ordre de priorité peut en être déduite). Ce paramètre sera utilisé dans l'appel de la fonction principale du module de la priorité.

Elle renvoie quatre éléments : la liste des messages encore en route *EnRoute*, la liste des messages arrivés à leur destination *Arrived*, la nouvelle liste de tentative *natt*, et le nouvel état du réseau *newNtkState* modifié selon les décisions d'ordonnancement des messages [8].

La première obligation de preuve est une contrainte de typage : si l'entrée $\mathcal{V}$ est une liste de voyages valide, alors la liste *Arrived* est une liste de voyages bien formée et la liste *EnRoute* est une liste de traveling missives. A noter que la liste *EnRoute* sera réinjectée lors du prochain appel récursif de la fonction $GeNoC_t$, cette liste doit donc être une liste de traveling missives.

Obligation de preuve 4.17 Validité des listes *Arrived* **et** *EnRoute*.
$\forall \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}, NodeSet) \Rightarrow \mathcal{V}_{lstp}(Arrived, NodeSet) \wedge \mathcal{TM}_{lstp}(EnRoute, NodeSet)$

8. Par exemple, désactivation des signaux de requête des communications effectuées, occupation et libération de place dans les buffers des nœuds...

Le nouvel état du réseau *newNtkState* doit être bien typé. Il doit satisfaire le prédicat *BasicStatep* si l'état *NtkState* passé en entrée à la fonction était bien typé. Ceci se traduit par :

Obligation de preuve 4.18 Validité du nouvel état.
$$\forall \mathit{NtkState}, \mathit{Validstatep}(\mathit{NtkState}) \Rightarrow \mathit{Validstatep}(\mathit{newNtkState})$$

La prochaine obligation de preuve garantit la vivacité de la fonction *Scheduling*. Elle garantit qu'une tentative, au moins, sera consommée à chaque cycle. Comme dans le chapitre précédent, la fonction *SumOfAtt* calcule la somme des tentatives.

Obligation de preuve 4.19 *Scheduling* **consomme au moins une tentative.**
$$\mathit{SumOfAtt}(\mathit{att}) \neq 0 \Rightarrow \mathit{SumOfAtt}(\mathit{natt}) < \mathit{SumOfAtt}(\mathit{att})$$

Les deux listes *Arrived* et *EnRoute* doivent être mutuellement exclusives : l'intersection des identificateurs des deux listes doit être l'ensemble vide. Ceci est exprimé par la propriété suivante :

Obligation de preuve 4.20 Exclusion mutuelle entre *Arrived* **et** *EnRoute*.
$$\mathcal{V}_{ids}(\mathit{Arrived}) \cap \mathcal{TM}_{ids}(\mathit{EnRoute}) = \emptyset$$

La correction de la liste *EnRoute* doit être contrôlée par rapport à la liste des voyages $\mathcal{V}$ (l'entrée de la fonction *Scheduling*). *Pour chaque traveling missive "en" dans EnRoute, il existe un voyage unique dans la liste* $\mathcal{V}$*, ayant l'identificateur, l'origine, le trame, la destination, le nombre de flit et le temps identiques à ceux de "en"* (le champ *current* peut, lui, être différent si le message s'est déplacé). Les traveling missives de *EnRoute* seront utilisées dans l'appel récursif de la fonction GeNoC_t. *EnRoute* doit être un sous-ensemble de l'ensemble initial de traveling missives passé en entrée à la fonction *Ready4Departure* d'où la raison pour cette obligation de preuve. Elle nous permet de vérifier que le module ne crée pas des messages et que toute traveling missive correspond à un voyage qui appartient à la liste qui a été passée à la fonction *Scheduling* [9].

Obligation de preuve 4.21 Correction de *EnRoute*.
$$\forall \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}, \mathit{NodeSet}) \Rightarrow \forall \mathit{en} \in \mathit{EnRoute}, \exists! v \in \mathcal{V},$$
$$\begin{cases} \quad \mathit{Id}_{\mathcal{TM}}(\mathit{en}) = \mathit{Id}_{\mathcal{V}}(v) \\ \wedge \quad \mathit{Org}_{\mathcal{TM}}(\mathit{en}) = \mathit{Org}_{\mathcal{V}}(v) \\ \wedge \quad \mathit{Frm}_{\mathcal{TM}}(\mathit{en}) = \mathit{Frm}_{\mathcal{V}}(v) \\ \wedge \quad \mathit{Dest}_{\mathcal{TM}}(\mathit{en}) = \mathit{Last}(\mathit{First}(\mathit{Routes}_{\mathcal{V}}(v))) \\ \wedge \quad \mathit{Flit}_{\mathcal{TM}}(\mathit{en}) = \mathit{Flit}_{\mathcal{V}}(v) \\ \wedge \quad \mathit{Time}_{\mathcal{TM}}(\mathit{en}) = \mathit{Time}_{\mathcal{V}}(v) \end{cases}$$

Une obligation similaire existe pour la liste *Arrived*, cependant pour un message arrivé seule la route empruntée par le message est gardée dans la liste des routes possibles. De ce fait, la route d'un message arrivé doit simplement être présente dans la liste des routes du voyage dans $\mathcal{V}$.

9. A noter que *Last(First(Routes))* est le dernier nœud de la première route de v, qui est utilisé pour reconstruire la traveling missive. Nous avons prouvé que les derniers nœuds des différentes routes d'un voyage sont tous identiques dans la section précédente (égales à la destination).

Obligation de preuve 4.22 Correction de *Arrived*.

$$\forall \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}, NodeSet) \Rightarrow \forall arr \in Arrived, \exists! v \in \mathcal{V},$$
$$\begin{cases} & Id_{\mathcal{V}}(arr) = Id_{\mathcal{V}}(v) \\ \wedge & Org_{\mathcal{V}}(arr) = Org_{\mathcal{V}}(v) \\ \wedge & Frm_{\mathcal{V}}(arr) = Frm_{\mathcal{V}}(v) \\ \wedge & Routes_{\mathcal{V}}(arr) \in Routes_{\mathcal{V}}(v) \\ \wedge & Flit_{\mathcal{V}}(arr) = Flit_{\mathcal{V}}(v) \\ \wedge & Time_{\mathcal{V}}(arr) = Time_{\mathcal{V}}(v) \end{cases}$$

Si les routes de l'entrée de la fonction *Scheduling* sont correctes, les deux obligations de preuve précédentes nous permettent de garantir la correction des listes *Arrived* et *EnRoute*.

Finalement, une nouvelle obligation de preuve a été ajoutée dans cette version afin de prouver la nouvelle propriété de correction de *GeNoC*. Cette obligation de preuve exprime que l'union des identificateurs des deux listes *Arrived* et *EnRoute* est une permutation des identificateurs de la liste de voyages passée en paramètre à la fonction *Scheduling* :

Obligation de preuve 4.23 Permutation des identificateurs de $\mathcal{V}$.

$$IsPerm(\mathcal{V}_{ids}(Trlst), \mathcal{V}_{ids}(Arrived) \cup \mathcal{TM}_{ids}(EnRoute))$$

Ceci conclut les obligations de preuve du module *Scheduling* dans la nouvelle version de *GeNoC*.

4.2.4.3 Priorité d'entrée

Le module de priorité d'entrée modélise la politique d'arbitrage utilisée au sein des routeurs d'un réseau. La politique d'arbitrage décide l'allocation des ressources du nœud aux messages selon un certain ordre de priorité. Un message ayant une priorité plus élevée doit être servi avant un autre avec une priorité plus basse. Pour modéliser les priorités entre les messages, nous utilisons une fonction de tri qui ordonnera les messages dans l'ordre décroissant de leur priorité. Le critère utilisé pour ordonner les messages peut être une combinaison de l'ordre des ports calculé par *getnextpriority* et d'un ordre dynamique induit par l'état courant du réseau. Une fois ces messages ordonnés au début d'un cycle d'ordonnancement, l'algorithme d'ordonnancement n'a plus qu'à traiter les messages dans l'ordre de façon à allouer les ressources aux messages prioritaires d'abord. L'union de ces ordres partiels sur chaque nœud forme un ordre global sur tout le réseau.

La fonction principale du module *Priority* est *PrioritySort*. Elle prend en entrée deux paramètres :

- la liste des messages (sous forme de voyages) à trier,
- l'ordre de priorité entre les ports sur le même nœud, ou un élément qui permet de le construire. Il s'agit de la relation d'ordre calculée par la fonction *getnextpriority*.

La liste triée est renvoyée en résultat de la fonction. La bonne formation de la liste renvoyée en sortie est essentielle. La liste doit être une liste valide de voyages pour garantir le bon fonctionnement de la fonction *Scheduling* qui utilisera cette liste pour effectuer son travail d'ordonnancement. La liste renvoyée par *PrioritySort* doit être une liste de voyages valide si l'entrée l'était.

Obligation de preuve 4.24 Validité de la sortie de *PrioritySort*.

$$\forall \mathcal{V}, \mathcal{V}_{lstp}(\mathcal{V}, NodeSet) \Rightarrow \mathcal{V}_{lstp}(PrioritySort(\mathcal{V}, order), NodeSet)$$

La deuxième obligation de preuve est la plus importante. Elle exprime que les identificateurs des messages dans la liste de sortie de la fonction est une permutation de ceux de la liste d'entrée.

Obligation de preuve 4.25 Permutation des identificateurs de *PrioritySort*.

$$IsPerm(\mathcal{V}_{ids}(\mathcal{V}), \mathcal{V}_{ids}(PrioritySort(\mathcal{V}, order))$$

Cette dernière obligation de preuve est indispensable pour obtenir la preuve de la nouvelle propriété de correction finale de *GeNoC*.

4.2.5 Couche liaison de donnée - Module de synchronisation

Ce module permet de modéliser le protocole de communication entre routeurs voisins. Il permet d'établir les décisions d'envoi et de réception. Par exemple, si un réseau utilise un protocole de synchronisation par poignée de main, ce protocole sera modélisé dans cette étape. Les fonctions de ce module consultent l'état local des nœuds impliqués dans la communication dans l'état actuel du réseau afin de prendre les décisions.

Afin d'accomplir ces tâches, plusieurs fonctions génériques ont été définies. La fonction *reqtrans* est utilisée dans le cas où des signaux de requêtes sont mis en jeu. Elle prend en entrée l'état actuel du réseau et active les signaux de requête de tous les nœuds ayant des données sur leurs ports de sortie. L'état résultant de cette manipulation est renvoyé en résultat. Le résultat de la fonction *reqtrans* doit être un état valide :

Obligation de preuve 4.26 Correction de *reqtrans*.

$$Validstatep(ntkst) \Rightarrow Validstatep(reqtrans(ntkst))$$

Le prédicat *processreq* prend en entrée deux paramètres : l'état actuel du réseau et l'adresse d'un nœud. Il vérifie si l'état local du nœud en question permet de recevoir des données et renvoie un *vrai* le cas échéant.

La fonction *chkavail* prend quatre paramètres : l'état actuel du réseau, l'adresse de départ du message (nœud courant), la destination du message, et une route. La fonction est responsable de tester si la route peut être empruntée ou non. Elle utilise la fonction *processreq* pour savoir la faisabilité du transfert au prochain nœud sur la route. Une obligation de preuve stipule que la longueur de la route doit être supérieure ou égale à 2, que le nœud courant doit être différent du dernier élément de la route, et que la destination passée en paramètre est égale au dernier élément de la route. Cette obligation de preuve est indispensable pour garantir les obligations de preuve du module *Scheduling* (plus spécifiquement les obligations de la correction de la liste *Arrived* et *EnRoute*).

Obligation de preuve 4.27 Correction de *chkavail*.

$$chkavail(st, cur, dest, route) \Rightarrow \begin{cases} & cur \neq Last(route) \\ \wedge & dest = Last(route) \\ \wedge & Len(route) \geq 2 \end{cases}$$

La fonction *GoodRoute*? est responsable d'invoquer le prédicat précédent (*chkavail*) sur toutes les routes envisageables pour un même message, afin de trouver la *première* route possible. *GoodRoute*? prend quatre paramètres : l'état actuel du réseau, l'adresse de départ du message (nœud courant), la destination du message, et la liste des routes du message. L'ordre dans lequel les routes sont passées à *chkavail* est important. La modélisation des priorités de sorties (préférence de directions de sortie) si elle existe, se fait dans cette fonction en triant les routes selon l'ordre décroissant induit par les priorités. La fonction *GoodRoute*? renvoie le nouvel état calculé selon le résultat de la fonction *chkavail* (l'état reste inchangé si *chkavail* renvoie qu'aucune route ne peut être empruntée), et la route choisie (la route qui a réussi le test de *chkavail*). La seule obligation de preuve vérifie que la première constituante du résultat de la fonction (l'état du réseau obtenu par l'application de la fonction *first*) est un état valide :

Obligation de preuve 4.28 Correction de *GoodRoute*?**.**

$$Validstatep(ntkst) \Rightarrow Validstatep(first(GoodRoute?(ntkst, cur, dest, routes)))$$

La fonction *testroutes* est la fonction principale de ce module appelée par la fonction *Scheduling*. Elle prend en entrée l'état actuel du réseau et un message (sous forme de voyage). Elle est responsable d'invoquer *GoodRoute*? en lui passant les routes dans le bon ordre. A noter que les algorithmes de routage adaptatif mettent en jeu un ordre de préférence entre les routes (comme nous verrons avec le réseau Nostrum) et que ceci relève de la définition du module de routage. Cependant un ordre dépendant de l'état du réseau peut aussi exister. Ce dernier dépend seulement de l'état du réseau et des interactions et échanges de données entre les routeurs. Puisque les décisions de cette priorité entre les routes possibles dépendent des échanges et des communications entre les différents nœuds, nous considérons que cette tâche relève des responsabilités du module de synchronisation. Dans une instance d'un réseau donné, si des priorités de sortie sont considérées, un tri des routes doit être effectué selon les informations de synchronisation échangées entre les routeurs(routeur occupé, deadlock sur le routeur, buffers remplis...).

La fonction *testroutes* retourne à la fonction *Scheduling* le nouvel état de réseau et la route choisie (les deux étant les sorties de la fonction *GoodRoute*?). Une seule obligation de preuve existe, la fonction doit retourner un état valide. Le terme v ci-dessous représente le voyage.

Obligation de preuve 4.29 Correction de *testroutes***.**

$$Validstatep(ntkst) \Rightarrow Validstatep(first(testroutes(ntkst, v)))$$

Il faut noter que la gestion des acquittements n'est pas spécifiée dans le modèle générique. Généralement ces fonctions doivent être spécifiées quand le besoin l'impose. La fonction d'activation d'acquittement n'effectue que la mise à jour des valeurs de signaux dans l'état, et ne nécessite pas de représentation particulière dans notre modèle. Ceci conclut la section de synchronisation ainsi que la couche liaison de donnée. Nous nous penchons maintenant sur la fonction *GeNoC*.

4.2.6 *GeNoC*

Une fonction $GeNoC_t$ a été mise en place comme dans la première version du modèle. Elle prend sept paramètres en entrée :

- $\mathcal{TM}$: la liste de traveling missives à envoyer,
- *NodeSet* : la liste des adresses valides du réseau,
- *att* : la liste des tentatives,
- $\mathcal{V}$: un accumulateur pour les messages arrivés,
- *Time* : le cycle d'exécution actuel,
- *NtkState* : l'état actuel du réseau,
- *order* : l'ordre de priorité entre les ports sur le même nœud, ou un élément qui permet de le construire (dans le cas d'une priorité statique).

La fonction renvoie à la fin de son exécution deux listes : l'accumulateur $\mathcal{V}$, et la liste $\mathcal{TM}$ des messages encore en route et qui n'ont pas réussi à atteindre leur destination. La fonction *Ready4Departure* est invoquée sur la liste $\mathcal{TM}$ pour calculer la liste "traveling" des messages ayant le droit de voyager sur le réseau. Ce sont les messages qui seront passés à la fonction de routage. Les messages toujours en route (renvoyés par *Scheduling* dans *EnRoute*) et les messages qui n'avaient pas le droit de partir sur le réseau (renvoyés par *Ready4Departure* dans *delayed*), formeront la liste $\mathcal{TM}$ dans l'appel récursif de la fonction $GeNoC_t$. Le paramètre *order* est remplacé dans l'appel récursif par la fonction *getnextpriority* qui prend en entrée la valeur de *order* dans le cycle courant pour calculer celui du prochain cycle. La définition de la version de la fonction est la suivante :

Définition 4.6 Définition de $GeNoC_t$.
$GeNoC_t(\mathcal{TM}, NodeSet, att, \mathcal{V}, Time, NtkState, order) \triangleq$

If $SumOfAtt(att) = 0$ **then**
  $List(\mathcal{V}, \mathcal{TM})$
else
  Let $(delayed, traveling)$ **be**
    $Ready4Departure(\mathcal{TM}, nil, nil, Time)$ **in**
  Let $(Scheduled,\ EnRoute,\ natt,\ newNtkState)$ **be**
    $Scheduling(Routing(traveling, NodeSet), att, NodeSet, NtkState, order)$ **in**
    $GeNoC_t(EnRoute \cup delayed, NodeSet, natt, Scheduled \cup \mathcal{V},$
      $Time + 1, newNtkState, getnextpriority(order))$

La fonction *GeNoC* a été légèrement modifiée. Sa définition est illustrée ci-dessous. L'appel à la fonction *ComputeMissives* est remplacée par un appel à *ComputeTMissives*. *ComputeTMissives* calcule une liste de traveling missives. Nous pouvons voir le nouveau champ *courant* qui représente le nœud courant ainsi que les champs *flit*, *temps*, et *hopcount*. Le champ *courant* est initialisé à l'origine du message (le troisième paramètre). Le nombre de flits et *temps* sont obtenus directement à partir de la transaction. Le champ *hopcount* est initialisé à 0.

Définition 4.7 Définition de *ComputeTMissives*.
$ComputeTMissives(\mathcal{T}) \triangleq$
$Append_{t\in\mathcal{T}}(List(Id_{\mathcal{T}}(t), Org_{\mathcal{T}}(t), Org_{\mathcal{T}}(t),$
$send(Msg_{\mathcal{T}}(t)), Dest_{\mathcal{T}}(t), Flit_{\mathcal{T}}(t), Temps_{\mathcal{T}}(t), 0))$

La fonction *GeNoC* prend quatre paramètres : la liste des transactions $\mathcal{T}$, $P1$ et $P2$ qui sont utilisés pour générer l'ensemble des nœuds du réseau (cf section 4.2.2) ainsi que son état initial , et la liste des tentatives *att*. Elle renvoie deux sorties : la liste $A\mathcal{M}$ convertie en liste de résultat (*Result*), et la liste des messages qui n'ont pas atteint leur destination *Aborted*. La fonction invoque $GeNoC_t$ en lui passant les transactions converties en traveling missives, l'ensemble des nœuds du réseau (calculé par la fonction *NodeSetGen*), la liste des tentatives *att* suivi d'un accumulateur vide (pas de messages arrivés avant l'exécution du modèle), le temps initial (0), l'état initial du réseau et finalement une constante *order* qui représente l'ordre des priorités au premier cycle le cas échéant.

Définition 4.8 Définition de *GeNoC*.
$GeNoC(\mathcal{T}, att, P1, P2, order) \triangleq$
Let$(A\mathcal{M},\ Aborted)$ **be**
$GeNoC_t(ComputeTMissives(\ \mathcal{T}),\ NodeSetGen(P1),\ att,$
$nil,\ 0,\ StateGenerator(P1, P2),\ order)$
$return(ComputeResults\ (A\mathcal{M}),\ Aborted)$

La première propriété de correction n'a pas été modifiée. Elle énonce toujours que *pour chaque message reçu il existe un seul message envoyé ayant le même identificateur, le même contenu ainsi que la même destination* (théorème 3.1), elle se base toujours sur la lemme 3.4 du chapitre précédent.

La nouvelle propriété de correction stipule qu'*aucun message n'est perdu*. Elle est obtenue en prouvant que l'union des deux résultats (la liste *Aborted*, et la conversion de la liste $A\mathcal{M}$ en résultat) de *GeNoC* est une permutation de la liste des messages passée à la fonction au début de l'exécution. La preuve de cette propriété se base sur les obligations de preuve : 4.11, 4.16, 4.23, et 4.25. Le théorème final sera de la forme suivante. Nous appelons la conversion de la liste $A\mathcal{M}$ en résultat *Result*.

Théorème 4.2 *GeNoC* **conserve les messages.**
$IsPerm(\mathcal{T}_{ids}(\mathcal{T}), \mathcal{TM}_{ids}(Aborted) \cup \mathcal{R}_{ids}(Result))$

4.2.6.1 Version simulable de *GeNoC*

Nous avons parlé à plusieurs reprises de la possibilité de simulation du réseau pas-à-pas ce qui permet de voir l'avancement des messages étape par étape et de confronter les résultats de simulation obtenus avec ce modèle formel avec ceux obtenus par exemple à partir d'une description VHDL du réseau. La version que nous venons de présenter permet seulement de connaître les messages arrivés et ceux qui ne le sont pas. En ajoutant un accumulateur *AccuSt* dans lequel on accumule les états du réseau à la fin de chaque cycle d'exécution, nous sommes capable de renvoyer de plus en résultat le descriptif des états successifs pendant la simulation (exécution LISP proprement dit). Il suffit de changer la définition de la fonction en ajoutant l'accumulateur comme sortie de la fonction. La définition de la fonction $GeNoC_t$ devient donc :

Définition 4.9 Définition de $GeNoC_t$**.**
$GeNoC_t(\mathcal{TM}, NodeSet, att, \mathcal{V}, Time, NtkState, order, AccuSt) \triangleq$
If $SumOfAtt(att) = 0$ **then**
$List(\mathcal{V}, \mathcal{TM}, AccuSt)$
else
Let $(delayed, traveling)$ **be**
$Ready4Departure(\mathcal{TM}, nil, nil, Time)$ **in**
Let $(Scheduled,\ EnRoute,\ natt,\ newNtkState)$ **be**
$Scheduling(Routing(traveling, NodeSet), att, NodeSet, NtkState, order)$ **in**
$GeNoC_t(EnRoute \cup delayed, NodeSet, natt, Scheduled \cup \mathcal{V},$
$Time + 1, newNtkState, getnextpriority(order), AccuST \cup newNtkState)$

Nous devons maintenant changer la fonction $GeNoC$ pour accommoder les changements de la fonction $GeNoC_t$. Nous devons initialiser l'accumulateur des états du réseau $AccuSt$ avec l'état initial du réseau. La fonction $GeNoC_t$ renvoie une sortie en plus maintenant. Cette sortie sera aussi une sortie de la fonction $GeNoC$ et sera utilisée pour la simulation.

Définition 4.10 Définition de $GeNoC$**.**
$GeNoC(\mathcal{T}, P1, P2, att) \triangleq$
Let$(A\mathcal{M},\ Aborted, AccuSt)$ **be**
$GeNoC_t(ComputeTMissives(\ \mathcal{T}),\ \mathcal{D}_{NodeSet}(P1),\ att,\ nil,$
$0,\ StateGenerator(P1, P2),\ order, List(StateGenerator(P)))$
$return(ComputeResults\ (A\mathcal{M}),\ Aborted,\ AccuSt)$

Cette modification et la nouvelle version du modèle, nous permettent d'avoir des résultats de simulation réalistes et fidèles aux descriptions RTL des réseaux contrairement à la version présentée dans le chapitre 3.

4.3 Conclusion

Nous avons montré à travers ce chapitre la nouvelle version du modèle $GeNoC$. De nouvelles obligations de preuve ont dû être ajoutées dont certaines pour satisfaire la nouvelle propriété de correction qui consiste à démontrer qu'*il n'existe pas de perte de message dans le réseau.* Cette version du modèle permet une modélisation pas-à-pas.

Les réseaux avec un mode d'ordonnancement par ver de terre peuvent maintenant être modélisés. Le départ des messages à des instants différents peut également être pris en compte. Une modélisation de l'état global du réseau a été faite. Ceci nous permet de prendre en compte les éléments mémorisants du réseau. Cet étape est aussi un pas vers la formalisation de l'absence des deadlocks dans les réseaux [10]. Plusieurs nouveaux modules génériques ont été ajoutés au modèle : les priorités, la synchronisation, et le contrôle d'accès au réseau. Nous avons maintenant un modèle formel fidèle au comportement du réseau, qui permet à la fois des vérifications de propriétés mais aussi une première phase de debug par simple simulation si l'utilisateur le souhaite.

10. Nous exposons ce point avec plus de détails dans la conclusion du manuscrit.

Plusieurs réseaux utilisent une méthode de contrôle d'accès (e.g. le contrôle de flux par crédit). L'ajout de ce module était indispensable pour avoir une véritable modélisation générique des réseaux. Les routeurs utilisent toujours une méthode d'arbitrage selon une priorité (*i.e.* premier arrivé premier servi, tourniquet...). Le module de priorité d'entrée était donc également nécessaire. Certains réseaux (comme Nostrum) utilisent aussi une priorité de sortie : dans le cas de l'impossibilité de suivre la sortie souhaitée pour un message, ce dernier se verra dévié vers une autre direction. Cette direction sera choisie selon un mode de priorité de sortie. L'existence du module de synchronisation complète le modèle avec une première solution de modélisation de la couche liaison de données.

Ceci conclut la présentation de notre nouvelle version du modèle générique *GeNoC*. Dans le chapitre suivant, nous montrerons les cas d'études utilisés pour tester notre modèle. Nous montrerons l'instanciation de chacun des modules exposés dans ce chapitre pour chacun des réseaux étudiés.

Chapitre 5

Études de cas

Dans le chapitre précédent, nous avons illustré les modifications effectuées pour obtenir la version actuelle du modèle *GeNoC*. Les différents composants de notre modèle ainsi que leurs obligations de preuve ont été exposés . Durant la mise au point de cette version, nous avons effectué des études de cas afin d'identifier les modules indispensables des réseaux et leurs propriétés.

A travers ce chapitre, nous allons montrer les modélisations des réseaux Hermes, Spidergon et Nostrum décrits dans la section 2.6. Pour le réseau Spidergon, nous avons réutilisé des parties déjà décrites durant la modélisation des autres réseaux. Nous ne montrerons pas tous les détails de la modélisation mais seulement les parties conçues pour ce réseau [1].

En effet, l'un des avantages de ce modèle est que tout constituant déjà instancié et validé pour un NoC est réutilisable pour d'autres. Par exemple, une fois formalisées et vérifiées, les caractéristiques de la commutation par paquets sont directement réutilisables.

5.1 Les interfaces

Nous avons utilisé la même modélisation des interfaces pour les trois réseaux. Nous montrerons la modélisation dans cette section. Les interfaces encodent et décodent les messages durant l'injection des messages dans les réseaux et leur sorties des réseaux respectivement par les fonctions *send* et *recv*. Aucun détail d'implémentation de ces interfaces n'a été fourni pour les trois réseaux utilisés pour l'évaluation de notre modèle. En effet, les processus d'encodage et de décodage relèvent essentiellement de la couche transport, voire des couches applications sur lesquelles nous n'avions pas d'information. La modélisation des fonctions a donc simplement été réalisée par des fonctions d'identité (listing 5.1) [2].

L'unique obligation de preuve sur les interfaces est que leur composition est l'identité. Il est clair que la preuve en est triviale en se basant sur la définition des fonctions.

1. Pour la modélisation entière consultez [BHPS09].
2. Dans le cas où l'utilisateur du modèle aurait une définition effective des interfaces, le processus de définition sera le même : définition des fonctions suivie de l'unique obligation de preuve de ce module.

```
(defun send (m)
      m)
(defun recv (m)
  m)
```

Listing 5.1: Définition de *send* et *recv*

5.2 Hermes

Nous avons présenté le réseau Hermes dans la section 2.6.1. L'ajout de nombre de flit dans les différents types du modèle a été effectué. Celui-ci nous a permis de modéliser le mode d'ordonnancement par ver de terre utilisé dans ce réseau.

Nous commencerons par la modélisation du *NodeSet*, suivie de l'état du réseau. Le contrôle d'accès au réseau est l'étape suivante. Le routage et l'ordonnancement suivront. Nous montrerons l'instanciation finale de *GeNoC*. Enfin, nous montrerons un exemple de simulation en comparant le résultat de la simulation du code VHDL (niveau Register Transfer Level) et l'exécution de notre modèle.

5.2.1 *NodeSet* et l'état global du réseau Hermes

5.2.1.1 *NodeSet*

Le réseau Hermes est une grille 2D. Chaque nœud du réseau contient 5 ports bidirectionnels. Pour identifier avec exactitude chacun de ces ports, il nous faut la représentation suivante $< XYPD >$ ou X est la coordonnée X du nœud, Y est la coordonnée Y, P est l'identificateur du port et finalement D est la direction du port. X et Y varient entre 0 et le nombre de nœuds moins un sur leur axes respectifs. P représente le port, et peut avoir une des valeurs suivantes : N pour *north*, S pour *south*, E pour *east*, W pour *west* et L pour *local*. Finalement, D peut prendre la valeur I pour une direction d'entrée ou O pour le cas d'un port de sortie. Le *NodeSet*, ensemble des nœuds de communication, est en fait l'ensemble des ports.

Le *NodeSet* du réseau Hermes est généré par la fonction *MeshNodeSetGen* (instanciation de la fonction *NodeSetGen*). Cette fonction prend un paramètre : un couple qui spécifie la taille du réseau dans la direction de l'axe X et la taille dans la direction de l'axe Y. Elle génère tout les tuples de la forme $< XYPD >$ du réseau. La vérification des paramètres de *MeshNodeSetGen* est effectuée par la fonction *MeshHyps* (instance de la fonction *ValidParamsp*). Cette dernière prend le paramètre de la fonction *MeshNodeSetGen* et vérifie que le premier et le deuxième élément sont des naturels. La fonction *ValidNodep*, qui vérifie qu'un nœud est valide, est instanciée par la fonction *Meshnodep* qui vérifie qu'un nœud fait partie du domaine des adresses de Hermes noté $MeshNodeSet$. La seule obligation de preuve 3.1 devient donc : si les paramètres *pms* sont valides (reconnus par *MeshHyps*), tout élément produit par *MeshNodeSetGen* appartient au domaine $MeshNodeSet$. Cette modélisation a nécessité 10 fonctions, 9 théorèmes, et un temps de preuve de 3,5 secondes [3].

3. Les preuves sont effectuées sur une machine Intel Core DUO 1,6 GHz, avec 1,5 GB de RAM, un

Obligation de preuve 5.1 Instanciation de l'obligation 3.1.
$\forall pms, MeshHyps(pms) \Rightarrow \forall x \in MeshNodeSetGen(pms), Meshnodep(x)$

5.2.1.2 L'état global du réseau Hermes

Pour chaque adresse (port), il existe une entrée dans l'état global du réseau. L'état global est ainsi le rassemblement des états locaux des différents ports. Un état local doit permettre d'identifier à quel port il appartient, ainsi que l'état et le contenu des éléments de mémorisation, et l'état des signaux de synchronisation qui seront utilisés par les ports voisins pour effectuer les poignées de main pour initier les communications. La forme choisie est la suivante :
((Coor (X Y P D)) (Buffers (Max ..)(Free ..)(Contents ..)) (Signals (Tx ..) (Ackrx ..))).
Le mot clef *Coor* est associé à l'adresse du port en question. Le mot clef $Buffers$ est associé à trois autres mots clefs : le premier Max correspond au nombre total de places dans la mémoire tampon du port. Le deuxième $Free$ donne le nombre de places libres dans cette mémoire. Le troisième *Contents* donne le contenu de la mémoire sous forme de couples <identificateur de message, identificateur de flit>. Finalement, *Signals* est un parfait exemple de champ *Optional*. Ce champ contient les champs Tx et *Ackrx* (qui correspondent au signaux Tx et *Ackrx* de la figure 2.13). Tx est mis à 1 quand ce port contient des données à envoyer (dans le cas d'un port de sortie). *Ackrx* est activé pour signaler qu'un port d'entrée peut recevoir des données.

Les fonctions reconnaisseuses d'un état bien formé sont $HerValidEntryp$ et $HerValidstatep$ (listing 5.2) (instances des fonctions *ValidEntryp* et *Validstatep*). $HerValidEntryp$ étend *BasicEntryp* avec la caractérisation de la bonne formation du champ optionnel. Diverses fonctions ont été définies pour la génération et la manipulation de l'état du réseau Hermes. La fonction *HerStGenerator* construit l'état initial du réseau, elle prend en entrée deux paramètres. Le premier est un couple qui définit la taille du réseau, tandis que le deuxième est le nombre du buffers sur les ports d'entrées. Elle génère pour chaque port du réseau une entrée locale dans l'état en initialisant le nombre de places libres avec la valeur du deuxième paramètre. Les valeurs des champs Tx et *Ackrx* sont initialisées à 0. La fonction *HermesValidstateParamsp* est l'instanciation de la fonction *ValidstateParamsp*. Elle vérifie la validité des paramètres de *HerStGenerator*(le premier est un couple de naturels comme pour la fonction *MeshHyps*, le deuxième paramètre est un naturel).

La fonction *HermesLoadBuffer* et *HermesReadBuffer* sont les instanciations respectives des fonctions *loadbuffers* et *readbuffers*. La présence des signaux Tx et *Ackrx* nécessite la présence d'un mécanisme pour les manipuler. La fonction $modifyReq$ manipule le signal Tx tandis que $modifyAck$ modifie le signal *Ackrx*. Chacune de ces deux fonctions prend trois entrées, l'état actuel du réseau, l'adresse du nœud où le changement est à effectuer, et le mode du changement (activer/déactiver le signal). Elles renvoient chacune en sortie l'état du réseau après le changement du signal. Puisque les fonctions manipulent l'état du réseau, elles doivent renvoyer en sortie un état valide du réseau. Nous prouvons cette propriété pour chacune des fonctions.

Les différentes obligations de preuve de la section 4.2.2 ont été prouvées. Elles corres-

système d'exploitation Windows XP et ACL2 version 3.3.

```
(defun HerValidEntryp (entry)
 (if (endp entry)
      t
      (and (BasicEntryp entry)
           (equal (caadar (cddr entry)) 'Tx)
           (equal (caaddr (caddr entry)) 'Ackrx)
           (equal (caaddr entry) 'Signals))))

(defun HerValidStatep (ntkstate)
  (if (endp ntkstate)
      t
    (and (HerValidEntryp (car ntkstate))
         (HerValidStatep (cdr ntkstate)))))
```

Listing 5.2: Définition de la fonction *HermesReady4Dep*

pondent à prouver que l'état retourné est valide après sa manipulation par les fonctions *HermesLoadBuffer* et *HermesReadBuffer*. Finalement, la preuve de correspondance entre les adresses du réseau et l'état du réseau (obligation de preuve 4.6) et la preuve de correspondance entre les paramètres de la construction de l'état et ceux la génération de l'ensemble des nœuds du réseau ont été vérifiées. Au total, la modélisation est faite avec 18 fonctions, 16 théorèmes, et 0,13 secondes de temps de preuve.

5.2.2 Couche Transport - Le contrôle d'accès au réseau Hermes

Dans ce réseau, il n'existe aucune méthode de contrôle d'accès spéciale. Nous utilisons donc ce module afin de contrôler l'accès des messages au réseau par rapport au temps seulement. Ceci est accompli à l'aide de la fonction *HermesReady4Dep* illustrée dans le listing 5.3.

```
(defun HermesReady4Dep (TM delayed traveling time)
  (if (endp TM)
      (mv delayed traveling)
    (let ((mundertest (car TM)))
      (if (< time (TimeTM mundertest))
               (HermesReady4Dep (cdr TM) (cons mundertest delayed)
                                         traveling  time)
               (HermesReady4Dep (cdr TM) delayed (cons mundertest
                                         traveling) time)))))
```

Listing 5.3: Définition de la fonction *HermesReady4Dep*

La fonction prend quatre entrées, la liste $\mathcal{TM}$ à tester, deux accumulateurs vides au début et le temps actuel du modèle. Elle teste le temps de départ de chaque membre de la liste $\mathcal{TM}$ par rapport au temps actuel du réseau. Si le temps actuel est inférieur au temps de départ du message, ceci veut dire que le temps d'envoi du message n'est pas encore arrivé et le message est ajouté à la liste *delayed*. Sinon, il est ajouté à la liste des messages au départ *traveling*.

Les obligations de preuve 4.7, 4.8, 4.9, 4.10 et 4.11 ont toutes été prouvées. Les deux listes sont mutuellement exclusives, elles sont toutes les deux incluses dans la liste d'entrée $\mathcal{TM}$, et sont toutes les deux du type traveling missives si $\mathcal{TM}$ l'était. Finalement, nous prouvons que l'union des identificateurs des deux listes est une permutation des identificateurs de la liste d'entrée. 3 fonctions, 26 théorèmes ont été nécessaires pour un temps de preuve de 5,12 secondes.

5.2.3 Couche Réseau

5.2.3.1 Le routage

Hermes utilise un mode de routage déterministe minimal, le routage XY (voir section 2.3.1). Pour un couple source destination, il existe une seule route. Dans cette section nous montrerons notre modélisation de cet algorithme de routage.

La première étape consiste à trouver une mesure décroissante. A priori, la distance qu'un message doit parcourir entre sa source et sa destination est la distance entre les deux routeurs correspondants. Cependant, rappelons que nous n'assimilons pas les nœuds de communication aux routeurs, mais à leurs ports. Un "hop" dans la fonction de routage est un saut entre deux ports. Passer d'un port d'entrée d'un routeur au port d'entrée du suivant représente donc 2 hops (passage d'un port d'entrée au port de sortie dans la bonne direction, puis passage par le lien externe au port d'entrée voisin). Compte tenu du fait qu'à son arrivée à destination, tout message emprunte nécessairement un port de sortie (sortie du réseau par un port "Local"), la mesure définissant la distance entre un nœud courant et la destination est donnée par la fonction *XY-measure*, listing 5.4.

```
(defun XY-measure (current to)
 (let ((x_d (car to))
       (y_d (cadr to))
       (x_c (car current))
       (y_c (cadr current))
       (direction_o (car (last current ))))
      (if (equal direction_o 'I)
          (1+ (* 2 (+ (abs (- x_d x_c)) (abs (- y_d y_c)))))
          (* 2 (+ (abs (- x_d x_c)) (abs (- y_d y_c)))))))
```

Listing 5.4: Définition de la mesure pour le routage XY

Nous avons identifié des fonctions génériques dans les deux chapitres précédents : *RoutingLogic*, *RoutingCore* et *Routing*. Le processus de l'instanciation du module du routage consiste à définir les fonctions correspondantes pour le réseau Hermes. La fonction *XYLogic* est l'instanciation de *RoutingLogic* qui calcule un pas sur la route (voir section 4.2.4.1). Pour simplifier la définition de cette fonction, nous définissons une fonction de calcul pour chaque direction possible, utilisée par la fonction *XYLogic*.

Le listing 5.5 illustre la définition de ces fonctions et la fonction *XYLogic*. Chacune des cinq fonctions calcule un pas de mouvement dans une des cinq directions possibles de mouvements (nord, sud, est, ouest, et local) à partir d'une position donnée. La fonction *XYLogic* utilise ces fonctions pour calculer le prochain pas. Elle prend deux entrées

current et *to* (le nœud où l'entête du message réside pour l'instant et la destination). Le choix de la fonction à utiliser est fait selon la situation des nœuds *current* et *to*.

```
(defun move-north (current)
  (if (equal (car (last current)) 'I)
      (list (car current) (cadr current) 'N 'O)
    (list (car current) (1- (cadr current)) 'S 'I)))
(defun move-south (current)
  (if (equal (car (last current)) 'I)
      (list (car current) (cadr current) 'S 'O)
    (list (car current) (1+ (cadr current)) 'N 'I)))
(defun move-east (current)
  (if (equal (car (last current)) 'I)
      (list (car current) (cadr current) 'E 'O)
    (list (1+ (car current)) (cadr current) 'W 'I)))
(defun move-west (current)
  (if (equal (car (last current)) 'I)
      (list (car current) (cadr current) 'W 'O)
    (list (1- (car current)) (cadr current) 'E 'I)))
(defun move-local (current to)
  (cons current (cons to nil)))

(defun XYLogic (current to)
  (let ((x_d (car to))
        (y_d (cadr to))
        (x_c (car current))
        (y_c (cadr current)))
    (if (not (equal x_d x_c))
        (if (< x_d x_c)
            (move-west current)
          (move-east current))
      (if (< y_d y_c)
          (move-north current)
        (move-south current)))))
```

Listing 5.5: Définition de la fonction *XYLogic* et les différentes fonctions nécessaires

La prochaine étape consiste à définir l'instance de la fonction générique *RoutingCore*. Dans notre cas, la fonction portera le nom *XYCore*. Cette fonction est responsable du calcul de la route entière entre l'emplacement actuel de l'entête du message et sa destination en appliquant la fonction *XYLogic* : elle s'appelle récursivement à partir du nœud obtenu par *XYLogic* (même mode de fonctionnement décrit pour la fonction générique dans la section 3.7). Le listing 5.6 montre la définition de cette fonction. La deuxième ligne déclare que la mesure à utiliser pour prouver la terminaison est celle décrite par la fonction *XYmeasure* appelée sur les deux entrées *current* et *to*. Le fait que ACL2 accepte la définition de cette fonction en utilisant cette mesure est une preuve implicite de l'obligation de preuve 4.12.

Afin de satisfaire l'obligation de preuve 4.13, trois lemmes intermédiaires ont dû être prouvés : toute route calculée par *XYCore* débute au nœud *current*, elle se termine au nœud *to*, et les éléments constituants la route calculée par *XYCore* appartiennent tous à l'ensemble des nœuds valides du réseau.

```
(defun XYCore (current to)
 (declare (xargs :measure (XY-measure current to)))
 (if (or (not (2dnodep current))
         (not (2dnodep to)))
       nil
      (let ((x_d (car to))
                  (y_d (cadr to))
                  (x_c (car current))
                  (y_c (cadr current)))
           (if (and (equal x_d x_c)
                    (equal y_d y_c))
           (move-local current to)
           (cons current (XYCore (XYLogic current to) to))))))
```

Listing 5.6: Définition de la fonction *XYCore*

Nous définissons maintenant la fonction *XYRouting* (listing 5.7) comme l'instanciation de la fonction générique *Routing*. Cette fonction prend deux paramètres en entrées : la liste des messages à acheminer et l'ensemble d'adresses du réseau. La fonction construit le voyage correspondant à chaque traveling missives dans sa liste d'entrée avec le résultat du routage calculé par la fonction *XYCore*. L'ensemble des nœuds n'est pas utilisé dans cette algorithme de routage, il est donc déclaré à ignorer sur la deuxième ligne.

```
(defun XYRouting (TM nodeset)
  (declare (ignore nodeset))
  (if (endp TM)
      nil
    (let* ((miss (car TM))
           (From (OrgTM miss))
           (current (CurTM miss))
           (to (DestTM miss))
           (id (IdTM miss))
           (frm (FrmTM miss))
           (flits (FlitTM miss))
           (Time (TimeTM miss))
           (hopcount (hopcountTm miss)))
      (cons
       (list id from frm (list (XYCore current to)) flits time hopcount)
       (XYRouting (cdr TM) nodeset)))))
```

Listing 5.7: Définition de la fonction *XYRouting*

Après la définition de cette fonction, la preuve de la correction de la fonction de routage doit être effectuée. La première étape consiste à prouver que la fonction renvoie une liste de voyages valide (obligation de preuve 4.14). Ceci est obtenu en s'appuyant sur la bonne formation des routes calculées par *XYCore* et la définition de la fonction *XYRouting*. Un autre lemme est nécessaire, il stipule que si un nœud e appartient à un ensemble x qui est un ensemble de nœuds valides, donc e est un nœud valide. Nous prouvons ensuite que la liste des identificateurs du résultat de la fonction *XYRouting* est une permutation de de la liste des identificateurs de son paramètre (obligation de preuve 4.16). La dernière

obligation de preuve est la correction des routes (obligation de preuve 4.15). Nous omettons les directives données à ACL2 pour réussir les preuves. Le listing 5.8 donne ces trois obligations de preuve.

```
(defthm TrLstp-XYRouting
  (let ((NodeSet (MeshNodeSetGen Params)))
    (implies (and (TMissivesp TMissives NodeSet)
                  (meshhyps Params))
             (TrLstp (XYRouting TMissives nodeset) nodeset))))

(defthm is-perm-xy-routing
  (let ((NodeSet (MeshNodeSetGen Params)))
    (implies (and (TMissivesp TMissives NodeSet)
                  (meshhyps Params))
           (is-perm (V-ids (XYRouting TMissives nodeset))
                    (tm-ids TMissives)))))

(defthm CorrectRoutesp-XYRouting
  (let ((NodeSet (MeshNodeSetGen Params)))
    (implies (and (meshhyps Params)
                  (TMissivesp TMissives NodeSet))
             (CorrectRoutesp (XYRouting TMissives nodeset)
                                        TMissives NodeSet)))
```

Listing 5.8: Obligation de preuve de *XYRouting*

Maintenant que les différentes obligations de preuve du module du routage ont été prouvées pour la fonction *XYRouting*, il est facile d'établir que cette fonction est une instance de notre module générique défini à l'aide du *defspec* dans le chapitre précédent. Ceci est accompli dans le listing 5.9. Nous faisons le lien entre les fonctions génériques et les fonctions instanciées dans un premier temps. Puis nous aidons ACL2 en lui donnant quelques indices, notamment pour le sous-but 5, nous permettons à ACL2 l'expansion de la fonction *meshhyps*. Dans la version générique du routage les obligations de preuve utilisent les fonctions *NodeSetGen*, *Nodesetp* et *ValidParamsp*. Nous devons les instancier puisque les preuves dépendent de ces fonctions.

```
(definstance GenericRouting checkcomlpiancexyrouting
  :functional-substitution
  ((NodeSetGenerator MeshNodeSetGen)
   (NodeSetp Meshnodep)
   (ValidParamsp Meshhyps)
   (Routing XYRouting))
  :rule-classes nil
  :hints (("GOAL" :in-theory (disable ToMissives-Routing
                                      meshnodesetgen trlstp meshhyps
                                      TMissivesp))
          ("Subgoal 5"
           :in-theory (enable meshhyps)))
  :otf-flg t)
```

Listing 5.9: Preuve de conformité de *XYRouting*

Ceci conclut notre modélisation du module du routage du réseau Hermes. Nous avons défini 12 fonctions, 47 théorèmes pour un temps de preuve de 393,11 secondes.

5.2.3.2 L'ordonnancement

Hermes utilise un mode d'ordonnancement par ver de terre. Nous présentons notre modélisation de ce mode d'ordonnancement qui est l'instanciation du module *Scheduling*. Pour accomplir sa tâche, ce module en utilise deux autres : un module pour la politique de priorité entre les différents messages, et un autre pour le mode de synchronisation pour effectuer les envois entre les voisins (détaillé dans la section 5.2.4).

Comme nous l'avons indiqué dans le chapitre précédent, la fonction *getnextpriority* calcule le port qui sera prioritaire dans le cycle suivant. Nous verrons la définition de son instance *HermesGetNextPriority* dans la section 5.2.3.3 et son utilisation dans la section 5.2.5.

Notre instance de *Scheduling* utilise la fonction *WHS* (listing 5.10) qui est responsable de la prise des décisions d'ordonnancement. La fonction prend en entrée sept paramètres :
- $Trlst$: la liste des voyages (messages routés) à ordonnancer,
- $EnRoute$: l'accumulateur des messages qui n'ont pas atteint leur destination (soit encore en mouvement ou bloqués),
- *NtkState* : l'état du réseau au début de l'ordonnancement,
- $Arracc$: l'accumulateur qui contient les messages qui ont atteint leur destination,
- $2Bkept$: un accumulateur des messages bloqués (les entêtes sont bloqués) sur leur nœud courant,
- $2Bmoved$: un accumulateur des messages qui ont le droit de se déplacer,
- *nonodes* : un accumulateur pour connaître les nœuds (ports) desquels un message est déjà parti.

Les accumulateurs $2Bkept$ et $2Bmoved$ sont nécessaires pour connaître les messages qui ont les entêtes bloquées et les messages qui peuvent effectuer un saut. La mise à jour de l'état diffère selon le cas : si le message a l'entête bloquée, ses autres flits peuvent se déplacer jusqu'au nœud où l'entête réside (sous condition de la présence de place suffisante dans les buffers). Sinon, tous les flits du message peuvent avancer. Le dernier accumulateur *nonodes* permet d'empêcher qu'un port reçoive deux flits dans le même cycle. Si le port est dans la liste, il a déja reçu un flit dans ce cycle.

La fonction commence en vérifiant que la liste *trlst* n'est pas vide, si elle l'est la fonction retourne les accumulateurs (*enroute*, *arracc*, $2Bkept$ et $2Bmoved$) et l'état du réseau *ntkstate*. La fonction *HermesTestRoutes* prend un voyage et l'état actuel du réseau (voir la section 5.2.4). Elle vérifie la présence d'une route valide[4] pour le message selon l'état actuel du réseau. Si une telle route existe, elle retourne la route choisie $r?$ et l'état du réseau *intst* après avoir activé le signal d'acquittement nécessaire.

La fonction *WHS* vérifie ensuite que cette route $r?$ est utilisable (lignes 7- 10). Nous vérifions que le prochain pas sur la route ne fait pas partie des nœuds interdits (*nonodes*)

4. Valide signifie que la route est bien formée et que le port destinataire du premier saut est libre.

```
(defun WHS (trlst enroute ntkstate arracc 2bkept
                2bmoved nonodes)
  (if (endp trlst)
      (mv enroute arracc ntkstate 2bkept 2bmoved )
    (mv-let (intst r?)
            (HermesTestRoutes ntkstate (car trlst))
        (if (and (not (member-equal (second r?) nonodes))
                        r? (check_ack_wh intst (first r?) (second r?))
                 (or (destination-empty (second r?) ntkstate)
                             (dest-fin-prev-msg (second r?) ntkstate)))
          (if (equal (len r?) 2)
            (WHS (tail trlst) enroute
                 (dsblreqack (myupdatestate intst
                                 (list (update_route (first trlst) r?)))
                             (first r?) (second r?))
                 (cons (first trlst) arracc) 2bkept
                 (cons (first trlst) 2bmoved)
                 (cons (second r?) nonodes))
            (WHS (tail trlst)
                 (cons (update_route (first trlst) r?) enroute)
                 (dsblreqack (myupdatestate intst
                                   (list (update_route (first trlst) r?)))
                                     (first r?) (second r?))
                 arracc 2bkept (cons (first trlst) 2bmoved)
                 (cons (second r?) nonodes)))
          (WHS (tail trlst)(cons (first trlst) enroute)
                    intst arracc (cons (first trlst) 2bkept)
                    2bmoved nonodes)))))
```

Listing 5.10: Définition de la fonction *WHS*

et que $r?$ n'est pas vide, ce qui signifie la présence d'une route valide. De plus, les signaux de requête et d'acquittement doivent tous les deux être actifs (indiquant ainsi la possibilité du transfert). Enfin, pour respecter le mode de fonctionnement de la commutation par ver de terre, un des deux cas suivants doit être vrai :

- soit le buffer destination du saut est entièrement vide (ligne 9),
- soitle buffer n'est pas vide et le dernier élément occupé contient le dernier flit d'un message [5] (ligne 10).

Si les tests précédents sont satisfaits, deux cas sont possibles : le message atteindra sa destination dans ce cycle, ou le message est encore en route. Si le message atteint sa destination dans ce cycle (ligne 12-18), sa route aura la longueur deux (testée sur la ligne 11). Dans ce cas, nous effectuons un appel récursif de la fonction WHS en ajoutant le message à la liste des messages arrivés, en mettant à jour l'état du réseau en occupant la destination du saut et désactivant les signaux de requête et d'acquittement (fonction *dsblreqack*). Ensuite, il faut mettre l'identificateur du message dans la liste des identificateurs des messages ayant des entêtes qui ont bougé $2Bmoved$, et finalement ajouter la destination du saut à *nonodes*.

La même procédure sera effectuée dans le cas où la longueur de la route est supérieure à 2 (le message est encore en route pour sa destination). Le même mode opératoire est utilisé à l'exception de l'ajout du message à l'accumulateur *enroute* après l'avoir modifié avec la fonction *UpdateRoute* (listing 5.11) modélisant ainsi l'avancement de l'entête du message d'un port au prochain. Ceci est fait en éliminant le premier nœud sur la route (l'ancien emplacement de l'entête), ce qui transforme le deuxième nœud sur la route du voyage en l'emplacement actuel de l'entête.

```
(defun UpdateRoute (tr route)
 (list (idv tr) (orgv tr) (frmv tr) (list (cdr route))
       (flitv tr) (timev tr) (hopcountV tr)))
```

Listing 5.11: Définition de la fonction *UpdateRoute*

Si les tests sur les lignes 7- 10 échouent le message est simplement ajouté à la liste des messages *enroute* et à l'accumulateur $2Bkept$ signifiant ainsi que son entête ne peut pas bouger.

La fonction *WHS* est utilisée par la fonction *WormHoleScheduling* (listing 5.12) qui est l'instanciation de la fonction *Scheduling*. Elle prend en entrée la liste des voyages à ordonnancer *trlst*, la liste des tentatives *att*, l'ensemble des ports du réseau *NodeSet*, l'état actuel du réseau *NtkState*, et le port le plus prioritaire dans ce cycle qui induit l'ordre du priorité *order*.

La fonction vérifie que la somme des tentatives n'est pas nulle (à l'aide de la fonction *SumOfAtt*). Le cas échéant, elle renvoie ses entrées sans effectuer d'ordonnancement (après conversion de la liste des voyages en liste de traveling missives). Sinon, la fonction

5. Les flits appartenant à d'autres messages ne peuvent pas être acceptés sur un port tant que le ver n'est pas passé en entier à travers ce dernier.

WHS est invoquée sur : l'état actuel du réseau après activation des requêtes à l'aide de la fonction *WHreqtrans* (voir section 5.2.4), la liste des voyages après avoir appliqué la politique de priorité du tourniquet à l'aide de la fonction *roundrobin* (voir section 5.2.3.3). Les différents accumulateurs sont initialisés dans cet appel avec la valeur *nil* puisqu'ils sont vides initialement.

```
(defun WormHoleScheduling (trlst att NodeSet ntkstate order)
 (if (zp (SumOfAtt att))
     (mv nil (totmissives trlst) att ntkstate)
     (mv-let (enroute arrived newst 2Bkept 2Bmoved)
       (WHS (roundrobin trlst order) nil
                  (WHreqtrans ntkstate) nil nil nil nil)
     (mv (totmissives enroute) arrived
                    (consumeattempts att)
                    (deltacycleimitation 2Bkept 2Bmoved newst)))))
```

Listing 5.12: Définition de la fonction *WormHoleScheduling*

Le résultat de la fonction sera :
- la liste des voyages *enroute* après l'avoir convertie en liste de traveling missives,
- la liste des messages qui ont atteint leur destination *arrived*,
- la liste des tentatives après en avoir consommé un (avec la fonction *consumeattempts*),
- l'état actuel du réseau après l'accomplissement de l'ordonnancement qui sera générer à l'aide de la fonction *deltacycleimitation*.

La fonction *deltacycleimitation* fait avancer les flits des différents messages tout en respectant le cas des entêtes de message bloquées. Elle prend en entrée l'état actuel du réseau généré par *WHS* et les deux listes $2Bkept$ et $2Bmoved$. Elle effectue la mise à jour de l'état en respectant les deux listes puis renvoie l'état obtenu.

Les différentes obligations de preuve de la section 4.2.4.2 ont été prouvées. La preuve que l'instance d'ordonnancement satisfait les obligations de preuve de *Scheduling* a été effectuée à l'aide de 16 fonctions, et 64 théorèmes en 331,71 secondes.

5.2.3.3 Les priorités

Le réseau Hermes utilise un mode de priorité par tourniquet. Un port est le plus prioritaire à un cycle et la priorité diminue dans le sens des aiguilles d'une montre. Le cycle suivant le port à sa droite sera le plus prioritaire. La fonction *HermesGetNextPriority* [6] (listing 5.13) est responsable du calcul du port le plus prioritaire du cycle suivant. La fonction prend en entrée un port (celui qui est prioritaire dans le cycle actuel) et renvoie le port d'entrée qui sera prioritaire au cycle suivant. Cette fonction sera invoquée par l'instance de la fonction $GeNoC_t$. La définition de cette fonction est simple, l'évolution du port le plus prioritaire est fixe.

6. Nous utilisons la directive *cond*. Cette directive fonctionne comme un *switch case*.

```
(defun HermesGetNextPriority (port)
 (cond ((equal port 'L) 'E)
       ((equal port 'E) 'S)
       ((equal port 'S) 'W)
       ((equal port 'W) 'N)
       ((equal port 'N) 'L)))
```

Listing 5.13: Définition de la fonction *HermesGetNextPriority*

La fonction *roundrobin* est la modélisation de la politique d'ordonnancement. C'est une fonction de tri qui met les messages dans l'ordre de leur priorité décroissante selon le port où ils résident. L'ordre partiel sur chaque nœud engendre ainsi un ordre global. La fonction prend en entrée la liste de messages à trier, et le port le plus prioritaire donné par *HermesGetNextPriority*. Selon ce port, l'ordre de priorité des ports est établi et il est utilisé pour trier les messages.

Les obligations de preuve de ce module ont été prouvées à l'aide de 11 fonctions et 23 théorèmes en 39,19 secondes. Les propriétés prouvées sont principalement que la liste des identificateurs du résultat est une permutation de la liste des identificateurs du paramètre, et que le résultat est une liste de voyages.

5.2.4 Couche liaison de donnée - synchronisation

Hermes utilise un mode de synchronisation par poignée de main. Un port ayant un message à envoyer active son signal de requête. Si le port voisin peut recevoir le message, il active son signal d'acquittement. La fonction *WHreqtrans* active les signaux de requête de tous les ports de sortie ayant des données dans leurs buffers. Elle prend une entrée (l'état du réseau actuel) et renvoie l'état après l'activation des signaux de requêtes. Elle est appelée par la fonction *WormHoleScheduling*. La seule contrainte sur cette fonction est que la sortie est un état de réseau valide si l'entrée est un état valide. Dans la fonction *WormHoleScheduling*, nous utilisons la fonction *dsblreqack* pour désactiver le signal de requête de l'origine du saut et l'acquittement de sa destination.

La fonction principale dans ce module est *HermesTestRoutes*. Elle prend un voyage et l'état actuel du réseau comme entrée et tâche de trouver une route valide libre que le message peut suivre. Si une route est trouvée, elle est renvoyée avec l'état du réseau après l'activation de l'acquittement du port destinataire du mouvement (deuxième port sur la route). Elle utilise la fonction *HermesGoodRoutes* pour trouver une route valide.

Cette dernière prend en entrée l'état actuel et la liste des routes du message ainsi que l'origine et la destination finale du message. Elle tâche de trouver si le prochain port sur la route a de la place libre dans ses buffers à l'aide de la fonction *Herchkavail* (listing 5.14) (instance de de la fonction *chkavail*). *Herchkavail* utilise la fonction *herprocessreq* pour tester s'il existe de la place libre dans le buffer de la destination du saut. Si c'est le cas, le signal d'acquittement de la destination du saut dans l'état du réseau est activé. *Herchkavail* vérifie aussi les conditions liées à l'obligation de preuve 4.27. La route choisie et l'état modifié sont retournés à la fonction *HermesTestRoutes* qui les renvoie en résultat à son tour.

```
(defun Herchkavail (st cur dest route)
 (and (Herprocessreq (cadr route))
      (<= 2 (len route))
       (not (equal cur (car (last route))))
       (equal (car (last (cdr route))) dest)))
```

Listing 5.14: Définition de la fonction *Herchkavail*

La définition de ce module a nécessité 6 fonctions et 10 théorèmes, pour un temps CPU de 0,66 secondes.

5.2.5 L'instanciation de *GeNoC*

La dernière étape afin de valider notre réseau consiste à créer l'instance des fonctions *GeNoC* et $GeNoC_t$. L'instance $HermesGeNoC\_t$ (listing 5.15) est notre instance de $GeNoC_t$. Nous pouvons voir que cette fonction possède le même squelette de la version générique de la section 4.2.6. Chaque fonction est remplacée par son instance pour le réseau. L'étape suivante est de créer l'instance de la fonction *GeNoC*. Dans notre cas nous l'appellerons $HermesGeNoC$ (listing 5.16).

```
(defun HermesGeNoC_t (m nodeset att trlst time ntkstate order accup)
 (declare (xargs :measure (sumofattempts att)))
 (if (zp (sumofattempts att))
     (mv trlst m accup )
   (mv-let (delayed departing )
           (HermesReady4Dep m nil nil time)
           (let ((v (XYRouting departing nodeset))
             (mv-let (newtrlst arrived newatt newntkstate)
                     (WormHoleScheduling v att nodeset ntkstate order)
                     (HermesGeNoC_t (append newtrlst delayed)
                               nodeset newatt (append arrived trlst)
                                (+ 1 time) newntkstate
                               (HermesGetNextPriority order)
                               (append accup (list newp)))))))))
```

Listing 5.15: Définition de la fonction $HermesGeNoC\_t$

La procédure de vérification des propriétés de correction consiste à créer les instances de ces deux fonctions, ensuite les théorèmes de correction sont obtenus directement. Le processus d'instanciation nécessite la définition de deux fonctions et 4 théorèmes. Nous avons pu ainsi prouver en 13,88 secondes, que :

- *les messages arrivés atteignent leur destination sans altération dans leur contenu,*
- *le réseau ne perd pas de message.*

5.2.6 La simulation

Nous avons mentionné dans la section 4.1.2 que ACL2 fournit à la fois un démonstrateur de théorèmes et un environnement d'exécution.Grâce à nos spécifications exécutables,

```
(defun HermesGeNoC (trs att p1 p2 order)
 (mv-let (responses aborted accup )
         (HermesGeNoC_t (computetmissives trs) (MeshNodeSetGen p1)
         att nil '0 (2d-state-generator-with-ports p1 p2) order
                    (list (2d-state-generator-with-ports p1 p2)))
         (mv (computeresults responses) aborted accup)))
```

Listing 5.16: Définition de la fonction *HermesGeNoC*

nous avons pu comparer les résultats d'exécution avec ACL2 et les simulations des descriptions VHDL [7] au niveau RTL. Nous commençons en montrant la simulation VHDL du réseau Hermes sur un exemple simple puis nous exposerons l'exécution ACL2 correspondante.

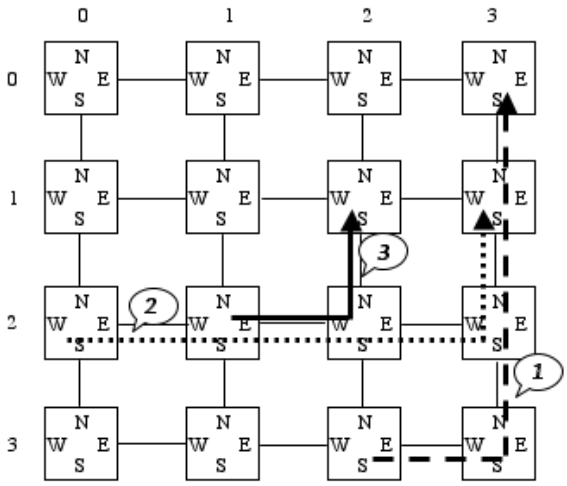

FIGURE 5.1 – Exemple de simulation du réseau Hermes

Dans nos expérimentations nous avons utilisé une version 4x4 du réseau Hermes. Le tableau 5.1 décrit les 3 messages de notre exemple de simulation (figure 5.1). Le message 1 est divisé en 6 flits (donc le nombre total de flits est 8 puisque Hermes en ajoute deux pour le contrôle), le message 2 est divisé en 3 flits (5 flits au total), et le message 3 est découpé en 5 flits (7 flits au total). Nous supposons que tous les messages seront injectés à l'instant 0 de l'exécution du modèle.

ID	Source	Destination	Contenu	Nb de flits	Temps	Couleur
1	(2 3)	(3 0)	11 21 31 41 51 61	8	0	vert
2	(0 2)	(3 1)	12 22 32	5	0	rose
3	(1 2)	(2 1)	13 23 33 43 53	7	0	gris

Table 5.1 – Résumé des messages de l'exemple de simulation du réseau Hermes

7. Nous avons obtenu le code VHDL grâce au laboratoire LIRMM. Le code est maintenant disponible publiquement sur le site http ://www.inf.pucrs.br/~gaph/Projects/Hermes/Hermes.html.

Simulation VHDL. Nous commençons par la simulation de la modélisation VHDL du réseau. Le résultat est visible dans la figure 5.2. Nous montrons seulement les ports de sortie des routeurs qui sont pertinents pour notre exemple. Chaque nœud est représenté par un signal data_showXY(P), où X et Y sont les coordonnées du nœud. Chaque signal est composé de cinq vecteurs 8-bits, P peut prendre une des cinq valeurs :E pour l'est, W pour l'ouest, N pour le nord, S pour sud, et L pour le port local.

Dans la figure 5.2, nous pouvons voir l'évolution des trois messages. Chacun est composé d'un premier flit qui contient sa destination (21 pour le message 3 par exemple), un deuxième qui contient le nombre total des flits du message (7 dans le cas du message 3), et le reste est le contenu du message. le message 3 passera sur les signaux : data_show12(E) *i.e.*, le port *est* du nœud (1,2), data_show22(N) *i.e.*, le port nord du nœud (2,2), et data_show21(L) *i.e.*, le port local du nœud (2,1) pour atteindre sa destination. Le message 1 passera sur les signaux : data_show23(E), data_show33(N), data_show32(N), data_show31(N), and data_show30(L).

Pour le message 2, il est censé passer par : data_show02(E), data_show12(E), data_show22(E), data_show32(N), et data_show31(L). Lors de l'arrivée du message sur le port ouest du nœud (1,2) l'algorithme de routage calcule une sortie par le port *est* du même nœud, or le port *est* est déjà occupé par le message 3. Le message 2 sera donc bloqué dans le buffer du port ouest jusqu'au passage du message 3 entièrement (signal data_show12(E) repères 6 à 8). Une fois tous les flits du message 3 passés, le message 2 continuera son voyage le long de la route prévue. Cependant une autre contention aura lieu entre le message 1 et le message 2 pour l'allocation du port nord du nœud (3 2) (signal data_show32(N)). Encore une fois le message 2 aura la priorité la plus basse. Ainsi le message 1 obtiendra le port. Dès que le message libère le port, le message 2 continue sa route pour la destination (signal data_show32(N) entre les repères 11 et 12).

Exécution ACL2. ACL2 nous donne le résultat de l'exécution sous forme textuelle LISP difficile à interpréter (tableau 5.2). Nous avons donc créé un outil en Java qui permet d'animer cette sortie sous forme graphique. Dans la sortie textuelle, chaque flit est identifié par un couple *(m f)*, où m est l'identificateur du message et f et l'identificateur du flit. L'identificateur d'un flit a une valeur entre nombre de flit moins un et zéro ($0 < identificateurdeflit < nb\_flit - 1$). Les flits sont numérotés de façon décroissante. Par exemple, le flit 3 du message 1 sera noté *(1 2)*. Notre modèle s'exécute en cycles : un cycle d'exécution correspond à faire avancer un flit d'un seul saut, d'un port à un autre. D'une façon générale, deux cycles d'exécution correspondent à un cycle de simulation VHDL.

La fonction de routage du réseau calcule pour chacun des messages la route entre sa source et sa destination. Le message 1 suivra la route (2 3),(3 3), (3 2),(3 1) et (3 0). La route du message 2 sera (0 2), (1 2), (2 2), (3 2), et (3 1). Le message 3 suivra la route : (1 2),(2 2), et (2 1). Le tableau 5.2 donne une partie du résultat obtenu à l'exécution du modèle ACL2.

Nous pouvons voir qu'à l'instant 0 (premier cycle d'exécution) le premier flit de chacun

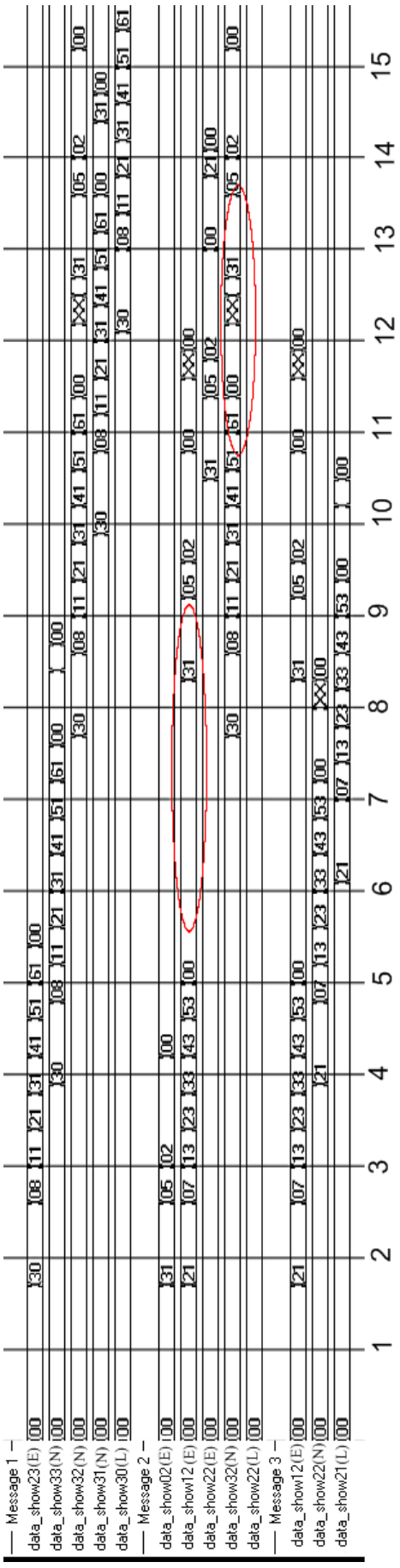

FIGURE 5.2 – Résultat de la simulation VHDL du réseau Hermes

Cycle	Evolution des messages
1er cycle	(((3 6) (1 2 L I)) ((2 4)(0 2 L I)) ((1 7) (2 3 L I)))
2ème cycle	(((3 6) (1 2 E O)) ((2 4) (0 2 E O)) ((1 7) (2 3 E O)) ((1 6) (2 3 L I)) ((3 5) (1 2 L I)) ((2 3) (0 2 L I)))
3ème cycle	(((3 6) (2 2 W I)) ((3 5) (1 2 E O)) ((2 2) (0 2 L I))((3 4) (1 2 L I)) ((2 4) (1 2 W I)) ((2 3) (0 2 E O)) ((1 7) (3 3 W I)) ((1 5) (2 3 L I)) ((1 6) (2 3 E O)))
4ème cycle	(((3 6) (2 2 N O)) ((1 7) (3 3 N O)) ((3 5) (2 2 W I)) ((1 5) (2 3 E O)) ((1 6) (3 3 W I)) ((3 4) (1 2 E O)) ((1 4) (2 3 L I)) ((2 2) (0 2 E O)) ((3 3) (1 2 L I)) ((2 1) (0 2 L I)) ((2 4) (1 2 W I)) ((2 3) (1 2 W I)))
...	....
9ème cycle	(((1 7) (3 0 S I)) ((3 2) (2 1 S I)) ((2 4) (1 2 W I)) ((3 3) (2 1 L O)) ((1 5) (3 1 S I)) ((1 6) (3 1 N O)) ((2 2) (1 2 W I)) ((2 1) (1 2 W I)) ((2 3) (1 2 W I)) ((2 0) (1 2 W I)) ((3 0) (2 2 W I)) ((3 1) (2 2 N O)) ((1 3) (3 2 S I)) ((1 4) (3 2 N O)) ((1 0) (2 3 E O))((1 1) (3 3 W I)) ((1 2) (3 3 N O)))
10ème cycle	(((1 6)(3 0 S I)) ((1 7) (3 0 L O)) ((1 3) (3 2 N O)) ((3 1) (2 1 S I)) ((3 2) (2 1 L O)) ((1 0) (3 3 W I)) ((1 4) (3 1 S I)) ((1 5) (3 1 N O)) ((2 3) (1 2 W I)) ((1 2) (3 2 S I)) ((2 2) (1 2 W I)) ((2 1) (1 2 W I)) ((2 0) (1 2 W I)) ((2 4) (1 2 E O)) ((3 0) (2 2 N O)))((1 1) (3 3 N O)))
...	....
13ème cycle	(((1 3) (3 0 S I)) ((1 0) (3 2 N O)) ((1 4) (3 0 L O)) ((1 1) (3 1 S I)) ((1 2) (3 1 N O)) ((2 2) (2 2 W I)) ((2 0) (1 2 W I)) ((2 1) (1 2 E O)) ((2 3) (2 2 E O)) ((2 4) (3 2 W I)))
14ème cycle	(((1 2) (3 0 S I)) ((1 3) (3 0 L O)) ((2 0) (1 2 E O)) ((1 0) (3 1 S I)) ((1 1) (3 1 N O)) ((2 2) (2 2 E O)) ((2 4) (3 2 W I)) ((2 1) (2 2 W I)) ((2 3) (3 2 W I)))
15ème cycle	(((1 1) (3 0 S I)) ((2 0) (2 2 W I)) ((1 2) (3 0 L O)) ((1 0) (3 1 N O)) ((2 1) (2 2 E O)) ((2 3) (3 2 W I)) ((2 2) (3 2 W I)) ((2 4) (3 2 N O)))
...	....
19ème cycle	(((2 1) (3 1 S I)) ((2 2) (3 1 L O)) ((2 0) (3 2 N O)))
20ème cycle	(((2 0) (3 1 S I)) ((2 1) (3 1 L O))
21ème cycle	(((2 0) (3 1 L O))

Table 5.2 – Sortie de la simulation ACL2 du réseau Hermes

des trois messages (1 7) (2 4) et (3 6) est sur le port local de sa source. Au cycle 2, les messages suivront la même route que dans l'implémentation VHDL. Au cycle 3, le premier flit du message 2 (flit (2 4)) atteint le port ouest du nœud (1 2) dans la direction de l'entrée.

Le flit devrait sortir par le port *est* du nœud dans le cycle 4. Or, le message 3 occupe déjà ce port. Au cycle 4 (figure 5.3.a), le premier flit du message 2 reste donc bloqué sur le port d'entrée ouest. Le deuxième flit du message (2 3) arrivera sur le port ouest et sera sauvegardé dans le buffer derrière son flit de *header* (2 4). Au cycle 5, le troisième flit arrivera sur le même port et sera traité de la même façon. Le reste des flits du message deux arriveront, l'un derrière l'autre, sur le port ouest où le header est bloqué. Ils resteront bloqués jusqu'au cycle 9 quand le message 3 passera en entier libérant ainsi le port *est* du nœud (1 2).

Au cycle 10 (figure 5.3.b) le port est libre, et le flit de *header* du message 2 peut donc repartir. Le reste des flits reste bloqué et repartira un à la fois dans les cycles suivants. Les messages continuent leurs trajets comme prévu par la fonction de routage. Au cycle 13 (figure 5.4), le message 2 veut emprunter le port nord du nœud (3 2) dans la direction

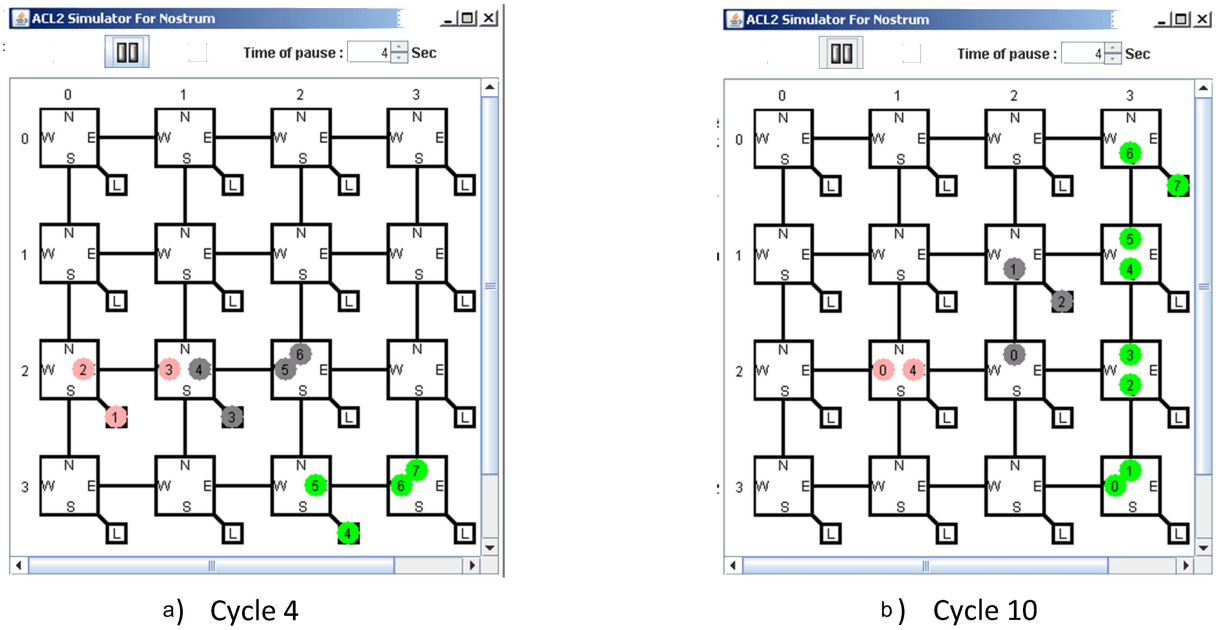

FIGURE 5.3 – Exécution de Hermes - cycle 4 et 10

de la sortie. Il est déjà occupé par le dernier flit du message 1. Le message 2 est encore une fois bloqué. Le dernier flit du message 1 libère le port au cycle 14, et le message 2 reprend son chemin dans le cycle 15. Six cycles plus tard le dernier flit du message 2 atteint sa destination.

Ceci conclut la simulation de notre modélisation du réseau Hermes. Nous avons montré les différents étapes à suivre pour vérifier un réseau à l'aide du modèle *GeNoC*. Nous avons vu également sur un exemple que les simulations de la modélisation VHDL et ACL2 du réseau Hermes produisent les mêmes résultats. Nous nous penchons maintenant sur la modélisation du réseau Spidergon.

5.3 Spidergon

Nous avons présenté ce réseau dans la section 2.6.3. Dans cette section, nous montrons la modélisation utilisée pour vérifier ce réseau, ainsi que la réutilisation des spécifications que notre modèle permet. Le routage de ce réseau a été modélisé durant la thèse de Julien Schmaltz [Sch06] avec le modèle *GeNoC* d'origine. Les autres modules ont été étudiés et modélisés dans le cadre de cette thèse. Notons q'un avantage de notre approche *GeNoC* et que tout constituant de réseau déjà modélisé et validé pour un réseau donné peut être réutilisé pour d'autres réseaux. C'est le cas ici pour les modules correspondant au contrôle d'accès, à la commutation par ver de terre et à la synchronisation par handshake. Ceux-ci ayant déjà été validés pour Hermes sont réutilisés ici puisqu'on les retrouve dans les caractéristiques de Spidergon. Nous ne reviendrons donc pas sur eux.

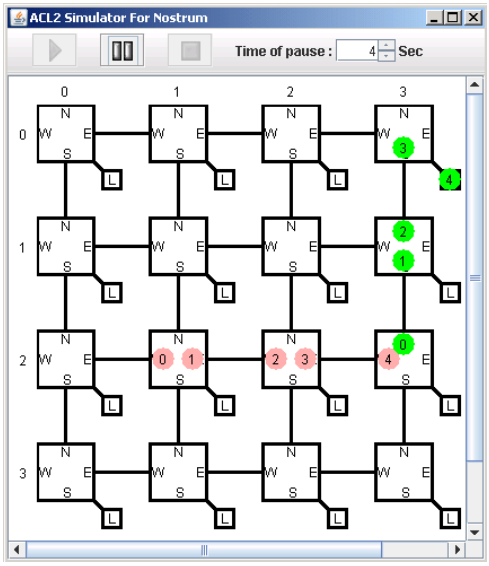

FIGURE 5.4 – Exécution de Hermes - cycle 13

5.3.1 *NodeSet* et l'état global du réseau Spidergon

5.3.1.1 *NodeSet*.

La modélisation des nœuds de ce réseau *SpiderNodeSet*, suit la même méthodologie que celle utilisée pour Hermes. Le réseau est sous forme d'anneau dans lequel chaque nœud possède 4 ports bidirectionnels. Nous utilisons la représentation suivante pour identifier avec exactitude chacun de ces ports $< NPD >$, où N, un naturel, est l'identificateur du nœud sur lequel le port se trouve. P, l'identificateur du port, peut prendre les valeurs suivantes : Cw pour designer le port "ClockWise", Ccw pour un port "CounterClockWise", Acr pour le port "Across" et l pour le port local connecté à un IP. D peut prendre la valeur I pour une direction d'entrée ou O pour le cas d'un port de sortie.

La fonction *SpiderNodeSetGen* génère le *NodeSet* du réseau Spidergon : ceci est l'instanciation de la fonction *NodeSetGen*. La vérification des paramètres passés en entrée à la fonction *SpiderNodeSetGen* est effectuée par la fonction *SpiderHyps* qui forme ainsi l'instanciation de la fonction *ValidParamsp*. La fonction *ValidNodep*, qui vérifie qu'un nœud est valide, est instanciée par la fonction *Spidernodep* qui vérifie qu'un nœud fait partie du domaine des nœuds de Spidergon noté *SpiderNodeSet*. La seule obligation de preuve 3.1 devient donc à : tout élément produit par *SpiderNodeSetGen* appartient au domaine *SpiderNodeSet*, si les paramètres de *pms* sont valides (les paramètres utilisés sont reconnus par *SpiderHyps*).

5.3.1.2 L'état global du réseau Spidergon.

Dans le cas de Spidergon, la forme choisie pour l'état local d'un port est : *((Coor (N P D)) (Buffers..) (Signals (Tx..) (Ackrx..)))*. Le mot clef *Coor* est suivi par la coordonnée du port. Nous avons implémenté ce réseau avec des ports à une seule place[8], le mot

8. Par manque d'informations sur les détails d'implémentation, nous avons choisi de considérer simplement des buffers à une place.

clé$Buffers$ est suivi par le contenu du buffer (l'identificateur du message et l'identificateur du flit) ou *nil* si le buffer est libre. Finalement, $Signals$ contient les champs Tx et $Ackrx$. Tx est mis à 1 quand ce port contient des données à envoyer (dans le cas d'un port de sortie). $Ackrx$ est activé pour signaler qu'un port d'entrée peut recevoir des données.

Les fonctions reconnaisseuses d'un état bien formé sont *Spivalidentry* et $SpiValidstate$. La fonction *SpiStGenerator* construit l'état du réseau, elle prend en entrée les paramètres reconnus valides par la fonction *SpiValidstateParamsp*. Dans le cas du Spidergon, le premier paramètre est le nombre des nœuds du réseau, tandis que le deuxième est fixé à un puisque les buffers sont à une place.

SpiLoadBuffer et *SpiReadBuffer* sont les instanciations respectives des fonctions *loadbuffers* et *readbuffers*. La présence des signaux Tx et $Ackrx$ nécessite la présence d'un mécanisme pour les manipuler, la fonction $modifyReq$ manipule le signal Tx tandis que $modifyAck$ modifie le signal $Ackrx$. Chacune de ces deux fonctions prend trois entrées, l'état actuel du réseau, l'adresse du port où le changement est à effectuer, et le mode du changement (activer/désactiver le signal).

Les différentes obligations de preuve de la section 4.2.2 ont été prouvées. 17 fonctions et 13 théorèmes étaient nécessaires pour cette modélisation pour un temps de preuve de 0,2 secondes.

5.3.2 Couche Transport - Le contrôle d'accès au réseau Spidergon

Spidergon n'utilise aucune méthode de contrôle d'accès. L'instanciation de la fonction *Ready4Departure* est donc identique à celle du réseau Hermes *HermesReady4Dep*.

5.3.3 Couche Réseau

5.3.3.1 Le routage

Le réseau Spidergon utilise un routage propriétaire. Nous l'avons décrit dans la section 2.6.3. C'est un routage déterministe minimal. Comme dans le réseau précédent, la première étape consiste à trouver une mesure décroissante pour cette fonction de routage. Nous utilisons une mesure similaire à celle utilisée pour Hermes, nous l'appelons *Spidergon-measure* (listing 5.17).

```
(defun Spidergon-measure (current to)
 (let ((n_d (car to))
       (n_c (car current))
       (direction_o (car (last current))))
   (if (equal direction_o 'I)
       (nfix (1+ (* 2  (abs (- n_d n_c)))))
     (nfix (* 2  (abs (- n_d n_c)))))))
```

Listing 5.17: Définition de la mesure pour le routage XY

Nous définissons maintenant les mouvements unitaires de routage. Quatre fonctions sont définies (listing 5.18 : *clockwise*, *counterclockwise*, *across*, et *local*). La définition est directe, si un message est sur un port d'entrée, il suivra le port nécessaire sur le même nœud pour sortir du nœud. S'il est sur un port de sortie, il faut calculer l'adresse du nœud sur lequel le message va arriver selon le port de sortie :

- *clockwise* : (nœud courant + 1) mod 16,
- *counterclockwise* : (nœud courant - 1) mod 16,
- *across* : (nœud courant + 8) mod 16.

```
(defun clockwise (from)
 (If (equal (get_dir from) 'I)
        (list (get_id From) 'Cw '0)
  (list (mod (+ (get_id from) 1) 16) 'ccw 'i)))

(defun counterclockwise (from)
        (If (equal (get_dir from) 'I)
                (list (get_id from) 'Ccw 'o)
  (list (mod (- (get_id from) 1) 16) 'cw 'i)))

(defun across (from )
        (If (equal (get_dir from) 'I)
                (list (get_id from) 'acr 'o)
                (list (mod (+ (get_id from) 8) 16) 'acr 'i)))

(defun local (from to)
                (cons from (cons to nil)))
```

Listing 5.18: Définition des mouvements unitaires du routage Spidergon

Nous définissons maintenant la fonction *SRLogic* (listing 5.19). Elle utilise les quatre fonctions de mouvements unitaires afin de calculer un pas selon la définition de la fonction de routage définie dans la section 2.6.3. La fonction *SCore* calcule la route entière en invoquant récursivement la fonction *SRLogic*.

Pour prouver les obligations de preuve du module de routage, plusieurs lemmes intermédiaires ont dû être prouvés, la plupart étant liés à l'utilisation de la fonction *mod*. Voyons les obligations de preuve du module de routage. Nous prouvons que : toute route calculée par *SCore* débute au nœud *current*, se termine au nœud *To*, et les éléments constituants la route calculée par *SCore* sont tous des nœuds valides appartenant au réseau.

Nous définissons ensuite la fonction *SpiderRouting* (listing 5.20), instance de la fonction *Routing*. Une fois cette fonction définie, il ne reste qu'à suivre les mêmes étapes que celles effectuées pour l'algorithme de routage du réseau Hermes. La modélisation de ce routage a été faite à l'aide de 28 fonctions, 79 théorèmes, en 1150,25 secondes.

5.3.3.2 L'ordonnancement

Comme nous avons dit au début de cette section, nous réutilisons l'ordonnancement par ver de terre défini pour le réseau Hermes pour le réseau Spidergon. Il nous faut juste

```
(defun SRLogic (from dest)
 (let* ((id_c (get_id from))
        (id_d (get_id dest))
        (RelAd (id_d - id_c) mod 16))
    (if (and (< 0 RelAd)
             (< RelAd 4))
        (clockwise from)
        (if (and (< 12 RelAd)
                 (< RelAd 12))
            (counterclockwise from)
            (across from)))))

(defun SCore (from dest)
 (let ((id_c (get_id from))
       (id_c (get_id dest)))
 (if (equal id_c id_d)
     (local from dest)
     (cons from (SCore (SRLogic from dest) dest)))))
```

Listing 5.19: Définition des fonctions *SRLogic* et *Score*

```
(defun SpiderRouting (Missives nodeset)
 (declare (ignore nodeset))
 (if (endp Missives)
     nil
   (let* ((miss (car Missives))
          (From (OrgTM miss))
          (current (CurTM miss))
          (to (DestTM miss))
          (id (IdTM miss))
          (frm (FrmTM miss))
          (flits (FlitTM miss))
          (Time (TimeTM miss))
          (hopcount (hopcountTm miss)))
     (cons
      (list id from frm (list (SCore current to)) flits time hopcount)
      (SpiderRouting (cdr Missives) nodeset)))))
```

Listing 5.20: Définition des fonctions *SpiderRouting*

modifier l'instanciation de la fonction *getnextpriority*. La fonction *Spidergetnextpriority* (listing 5.21) prend en entrée le port prioritaire dans le cycle actuel et renvoie le port qui sera prioritaire dans le prochain cycle. Le nombre de fonctions, de théorèmes et le temps de preuve reste le même que dans le cas du réseau Hermes.

```
(defun SpiderGetNextPriority (port)
 (cond ((equal port 'cw) 'acr)
       ((equal port 'acr) 'ccw)
       ((equal port 'ccw) 'l)
       ((equal port 'l) 'cw))
```

Listing 5.21: Définition de la fonction *Spidergetnextpriority*

5.3.3.3 Les priorités

Le réseau Spidergon utilise une priorité round robin. La priorité diminue à partir du port le plus prioritaire dans le sens des aiguilles d'une montre comme dans le cas de Hermes.

La fonction *Spiderroundrobin* est la modélisation de la politique d'ordonnancement. Elle suit une modélisation similaire à celle expliquée en section 5.2.3.3, en utilisant la fonction *Spidergetnextpriority* pour obtenir le port le plus prioritaire.

Les obligations de preuve de ce module ont été prouvées à l'aide de 11 fonctions et 23 théorèmes en 23,51 secondes.

5.3.4 Couche Liaison de Données - la synchronisation

Le réseau Spidergon utilise le même mode de synchronisation que réseau Hermes. Aucune modification n'est nécessaire dans ce module, nous réutilisons donc la modélisation faite pour Hermes.

5.3.5 L'instanciation de *GeNoC*

La dernière étape consiste à créer l'instance des fonctions *GeNoC* et $GeNoC_t$. L'instance $SpiderGeNoC\_t$ (listing 5.22) est notre instance de $GeNoC_t$. Comme pour le reseau Hermes, chaque fonction générique est juste remplacée par son instance pour le réseau Spidergon. La fonction *Ready4Departure* sera instanciée par la fonction *HermesReady4Dep* (c'est le même mode d'accès au réseau que Hermes). Le routage sera instancié dans la fonction *SpiderRouting* tandis que l'ordonnancement est instancié par la fonction *WormHoleScheduling*. La fonction *getnextpriority* sera instanciée par $SpiderGetNextPriority$.

L'instance de la fonction *GeNoC* s'appelle *SpiderGeNoC* (listing 5.23). L'instance de la fonction de génération de l'état du réseau est *SpiStGenerator* et les adresses des ports du réseau seront générées par la fonction. Comme dans le cas de Hermes, nous créons les instances de ces deux fonctions et les théorèmes de corrections sont obtenus directement.

```
(defun SpiderGeNoC_t (m nodeset att trlst time p order accup)
 (declare (xargs :measure (sumofattempts att)))
 (if (zp (sumofattempts att))
     (mv trlst m accup )
   (mv-let (delayed departing )
    (HermesReady4Dep m nil  nil time)
    (let ((v (SpiderRouting departing nodeset))
     (mv-let (newtrlst arrived  newatt newp)
            (WormHoleScheduling  v  att nodeset p order)
            (SpiderGeNoC_t (append newtrlst delayed)
                        nodeset newatt
                        (append arrived trlst)
                                (+ 1 time) newp
                                (SpiderGetNextPriority order)
                                (append accup (list newp))))))))
```

Listing 5.22: Définition de la fonction *SpiderGeNoC_t*

La définition de deux fonctions et 4 théorèmes est nécessaire pour un temps de preuve de 147 secondes.

```
(defun SpiderGeNoC (trs att p1 p2 order)
 (declare (ignore p2))
 ;; main function
 (mv-let (responses aborted accup )
         (SpiderGeNoC_t (computetmissives trs) (SpiderNodeSetGen p1)
                        att  nil '0 (SpiSTGenerator p1 '1)
                        order (list (SpiSTGenerator p1 '1)))
         (mv (computeresults responses) aborted accup)))
```

Listing 5.23: Définition de la fonction *SpiderGeNoC*

5.3.6 La Simulation

Voyons maintenant comment cette spécification peut être utilisée à des fins de simulation. Nous n'avions pas accès à une modélisation simulable de ce réseau. Nous ne pouvons donc pas comparer nos résultats à ceux produits en utilisant une modélisation RTL du réseau.

Id	Source	Contenu	Destination	Flits	Temps	Couleur
1	(0 Loc i)	(11 12)	(8 Loc o)	4	2	vert
2	(1 Loc i)	(21 22 23)	(8 Loc o)	5	1	rose
3	(4 Loc i)	(31)	(3 Loc o)	3	3	gris
4	(5 Loc i)	(41 42)	(3 Loc o)	4	1	Fuchsia

Table 5.3 – Exemple de simulation du réseau Spidergon

Nous utilisons 4 messages dans notre exemple (tableau 5.3 et figure 5.5). Le message 1 part du nœud 0 à l'instant 2. Il emprunte ensuite le port *Acr* pour atteindre sa destination :

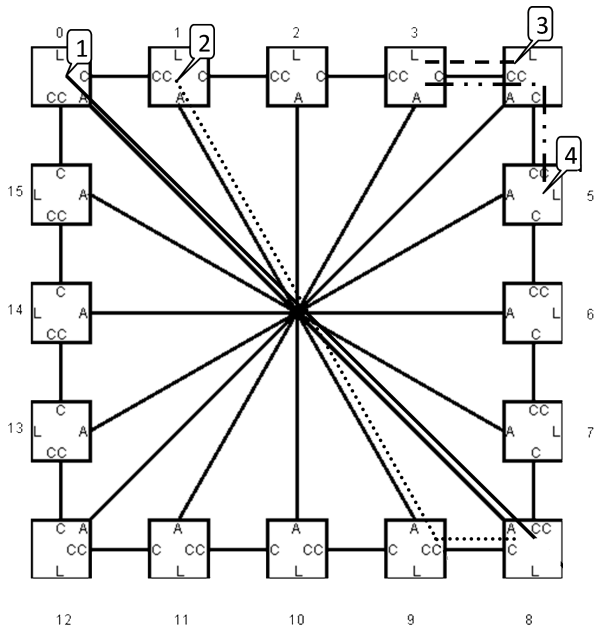

FIGURE 5.5 – Illustration des messages de l'exemple de simulation

le nœud 8. Le message 2 part a l'instant 1 du nœud 1. Il emprunte le port *Acr* pour arriver au nœud 9. Il emprunte ensuite le port *Ccw* pour arriver au nœud 8 (sa destination). Il reste bloqué sur le port d'entrée *Cw* de ce nœud jusq'au cycle 10 quand le message 1 libère le port local du nœud 8. Le message 3 part à l'instant 3 du nœud 4. Il emprunte le port *Ccw* pour atteindre le nœud 3 (sa destination). Le message 4 part du nœud 5 à l'instant 1. Il emprunte le port *Ccw* pour arriver au nœud 4. Il reste bloqué sur le port de sortie *Ccw* du nœud 4 jusqu'au cycle 9 quand le message 3 libère le port d'entrée *Cw* du nœud 3. Au cycle d'aprés le message 4 atteint sa destination (le nœud 3) Pour voir les routes des messages, consultez le tableau 5.4, où nous associons cycles d'exécution et positions de l'entête du message.

Id	Route
1	(2, (0 Loc i)) (3, (0 Acr o)) (4, (8 Acr i)) (5, (8 Loc o))
2	(1, (1 Loc i)) (2, (1 Acr o)) (3, (9 Acr i)) (4, (9 Ccw o)) (5, (8 Cw i)) (10, (8 Loc o))
3	(3, (4 Loc i)) (4, (4 Ccw o)) (5, (3 Cw i)) (6, (3 Loc o))
4	(1, (5 Loc i)) (2, (5 Ccw o)) (3, (4 Cw i)) (8, (4 Ccw o)) (9, (3 Cw i)) (10, (3 Loc o))

Table 5.4 – Routes suivies par les messages - exemple Spidergon

Au premier cycle, les messages 2 et 4 sont injectés dans le réseau. Le message 1 est injecté au cycle 2 et arrivera au port local du nœud 8 (sa destination) avant le message 2. En effet, le message 2 est bloqué sur le port *cw* du nœud 8 comme nous pouvons le voir sur la figure 5.6 puisque le message 1 est arrivé avant lui. Le message 2 restera bloqué sur ce port jusqu'au cycle 10. Le message 4 arrive sur le port *cw* du nœud 4 à l'instant où le message 3 part du port local de ce nœud. Les deux messages sont en compétition

pour obtenir le même port de sortie *ccw* du nœud 4. Le message 3 gagne selon l'arbitrage Round-Robin bloquant ainsi le message 4. Ce dernier continuera son voyage quand le message 3 aura libéré le port (cycle 8). Ceci est bien le comportement attendu selon la modélisation du réseau qui a été faite à partir des descriptions fournies dans la littérature.

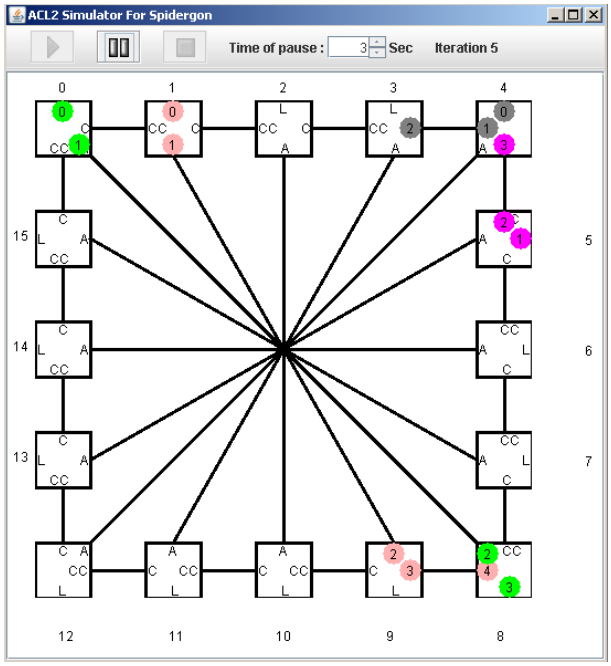

FIGURE 5.6 – Exécution Spidergon - cycle 5

5.4 Nostrum

Nous avons présenté le réseau Nostrum dans la section 2.6.2. Dans cette section, nous montrerons la modélisation du réseau utilisée pour sa vérification. L'implémentation pas-à-pas a permis la modélisation de l'algorithme de routage adaptatif non-minimal utilisé dans ce réseau, ce qui était impossible avec l'ancienne version du modèle.

5.4.1 *NodeSet* et l'état global du réseau Nostrum

Nodeset. Comme le réseau Hermes, Nostrum est une grille bi-dimensionnelle, nous utiliserons donc la même modélisation des nœuds que pour Hermes (section 5.2.1.1).

L'état global du réseau Nostrum. Comme nous l'avons indiqué dans la section 2.6.2, chaque port contient un buffer à une place. Nous avons noté aussi que chacun des nœuds communique à ses voisins la moyenne de sa charge durant les quatre derniers cycles. Dans ce cas, la forme d'une entrée d'état local d'un port est la suivante : *((Coor (X Y P D)) (Buffers ..) (Load (..)(..)(..)(..)))*. Le mot clef *Coor* est suivi par la coordonnée du port en question. $Buffers$ est suivi d'un message si le port contient un message ou de *nil*

dans le cas inverse. Finalement, "*Load*" est associé aux charges du nœud dans les quatre derniers cycles. Nous retrouvons ici un autre contexte d'utilisation du champ *Optional*.

NosValidstatep et *NosValidEntryp* reconnaissent un état global et un état local bien formé respectivement. La fonction *NostrumStGenerator* construit l'état initial du réseau, elle prend en entrée les paramètres reconnus valides par la fonction *HermesValidstateParamsp* Le premier paramètre est un couple qui définit la taille du réseau (donc deux naturels), tandis que le deuxième est fixé à un. Nous devons pourtant définir la fonction avec deux paramètres pour respecter la signature de la fonction générique. La fonction *NostrumLoadBuffer* et *NostrumReadBuffer* sont les instanciations respectives des fonctions *loadbuffers* et *readbuffers*.

A chaque cycle, la liste des charges passées de chaque port doit être décalée éliminant ainsi la plus ancienne, et la charge du cycle courant doit être mémorisée. Ce travail est effectué à l'aide de plusieurs fonctions. La première $ShiftLoad$ effectue le décalage pour un seul nœud, éliminant ainsi la charge la plus ancienne et met la première (celle du cycle courant) à 0. La fonction *incrementload* effectue la mise à jour de la charge d'un nœud en incrémentant la charge de 1. Il faut noter que la charge est calculée par nœud et non par port. Dans ce contexte, la fonction incrémente la charge de chacun des quatre ports de sortie du nœud : Nord, Est, Ouest, et Sud. La fonction *CalculateLoads* du module de la synchronisation (section 5.4.4) utilise *incrementload* en l'appelant chaque fois que la première trouve un message sur un des ports appartenant au même nœud [9].

Les différentes obligations de preuve de la section 4.2.2 ont été prouvées : l'état retourné après manipulation par les fonctions *NostrumLoadBuffer* et *NostrumReadBuffer* est valide. Nous avons défini les fonctions $IncrementLoad$, et $ShiftLoad$. Ces deux fonctions n'existaient pas dans le module générique puisqu'elles font référence à une caractéristique optionnelle de l'état. Il est néanmoins nécessaire de prouver que chacune retourne en résultat un état valide du réseau.

Finalement, la preuve de correspondance entre les adresses du réseau et l'état du réseau (obligation de preuve 4.6) et la preuve de correspondance entre les paramètres de la construction de l'état et ceux de la génération de l'ensemble des nœuds du réseau ont été vérifiées. 13 fonctions ont dû être définies, et 11 théorèmes sont vérifiés en 0,14 secondes.

5.4.2 Couche Transport - Le contrôle d'accès au réseau Nostrum

Le réseau utilise une méthode de contrôle d'accès qui interdit l'accès d'un message dans le réseau à travers un port local si quatre messages se trouvent déjà sur le nœud (un par port d'entrée). Cette méthode est due au fait que chaque message doit être capable d'emprunter un port de sortie (routage par déflection, pas de mémorisation possible des messages). Afin de simplifier le processus de modélisation et de preuve, nous n'avons pas modélisé cette option grâce à la présence d'une autre particularité.

Dans la section 5.4.3.3, nous verrons que durant l'ordonnancement des messages, le

9. *CalculateLoads* prend en paramètre la liste des voyages dans le réseau.

réseau utilise une priorité par âge des messages (*hopcount*) pour résoudre les contentions entre les messages comme cela a été décrit dans la section 2.6.2. Or, un message qui rentre dans le réseau par le port local aura toujours un *hopcount* égal à 0 et est donc le moins prioritaire. Nous n'avons pas donc besoin de modéliser la limite de quatre messages par nœud, parce que le message sur le port local ne pourra être ordonnancé que si un port de sortie est libre après l'ordonnancement des autres messages (ceci signifie qu'il y avait moins de quatre messages sur le nœud). L'implémentation de la fonction du contrôle d'accès est donc la même que celle utilisée pour Hermes et Spidergon.

5.4.3 Couche Réseau

5.4.3.1 Le routage

Nostrum utilise un algorithme de routage adaptatif non-minimal. Dans la nouvelle version du modèle *GeNoC*, la fonction de routage calcule une ou plusieurs routes entre la position actuelle du message et sa destination puis l'ordonnancement en choisira une et le message effectuera un seul pas sur cette route. Au cycle suivant, le processus recommence. Nous pouvons ainsi nous contenter de prendre en compte uniquement les quatre routes qui commencent par les quatre voisins du nœud courant et sont ensuite construites selon la politique décrite dans la section 2.6.2. Plus précisément, quatre routes (quatre ports : nord, est, ouest et sud) sont calculées dans le cas général, et cinq routes (les quatre ports précédents et le port local) si le message est sur le nœud de destination [10].

A partir de chaque nœud, nous avons donc cinq mouvements possibles. Nous définissons cinq fonctions, chacune calcule un pas dans l'une des cinq directions possibles : *MoveNorth* (listing 5.24 [11]), *MoveSouth*, *MoveEast*, *MoveWest*, et *MoveLocal*. Chacune des fonctions prend en entrée l'adresse du port actuel et renvoie le prochain pas. Nous rappelons que le prochain pas diffère selon la direction du port où le message réside : si c'est un port d'entrée, le prochain pas sera un port de sortie du même nœud sinon il sera un port d'entrée d'un voisin. Pour la fonction *MoveLocal*, un seul cas existe : c'est le port de sortie local du nœud *current*.

```
(defun MoveNorth (current)
  (if (equal (car (last current)) 'I)   ;; Si direction= I
      ;;prendre le port de sortie nord du noeud
      (list (car current) (cadr current) 'N 'O)
      ;;sinon entrer par le port sud du voisin
      (list (car current) (1+ (cadr current)) 'S 'I)))
```

Listing 5.24: Définition de la fonction *MoveNorth*

Ces fonctions seront utilisées dans la fonction *NosRoutingCore* [12](listing 5.25.). Cette

10. Il faut prendre en compte le cas où deux messages avaient le même destinataire et arrivent au même instant. Les deux seront en conflit, et un seul obtiendra l'accès au port local. L'autre sera envoyé à un des voisins.

11. (car (last current)) est la direction, (car current) est la coordonnée X, (cadr current) est la coordonnée Y.

12. Elle constitue l'instance de la fonction *RoutingCore*.

fonction calcule le prochain pas sur la route. Elle le calcule selon les règles de routage de Nostrum. Elle commence en comparant les distances le long de l'axe des X et le long de l'axe des Y entre le nœud courant et la destination. Si la distance le long de l'axe des Y est plus grande et si la coordonnée Y de la destination est supérieure à celle du nœud courant, la fonction calcule le prochain pas vers le nord. Si elle est inférieure le pas sera vers le sud. Les coordonnées X sont comparées si la destination est atteinte en Y. Si celle de la destination est inférieure, le pas sera vers l'ouest, et vers l'est sinon. Enfin, on opère symétriquement dans le cas où la distance le long de l'axe des X est la plus grande.

```
(defun NosRoutingCore (current to)
 (let* ((x_d (car to))
        (y_d (cadr to))
        (x_c (car current))
        (y_c (cadr current))
        (xdist (abs (- x_d x_c)))
        (ydist (abs (- x_d x_c))))
       (If (< xdist ydist)
           (if (< y_c y_d)
               (move-north current)
               (if (> y_c y_d )
                   (move-south current)
                   (if (< x_d x_c)
                       (move-west current)
                       (move-east current))))
           (if (< x_c x_d)
               (move-east current)
               (if (> x_c x_d )
                   (move-west current)
                   (if (< y_d y_c)
                       (move-south current)
                       (move-north current)))))))
```

Listing 5.25: Définition de la fonction *NosRouting*

La fonction précédente sera utilisée par la fonction *NosRoutingLogic* qui calculera une route entière selon le routage décrit dans la section 2.6.2 en appliquant *NosRoutingCore* récursivement. Dans le routage *Hot-potato*, à l'arrivée d'un message sur un nouveau nœud (le message est sur un port d'entrée), quatre mouvements sont possibles. Nous définissons alors quatre fonctions qui utilisent *NosRoutingLogic* pour calculer une route qui commence par le port courant et a comme prochaine escale un des quatre voisins possibles du nœud : $CalculateNRouteIn$, $CalculateSRouteIn$, $CalculateERouteIn$, et $CalculateWRouteIn$. Si le port où se trouve le message est un port de sortie, une seule route existe , celle qui conduit au voisin "en face". Les fonctions $CalculateNRouteOut$, $CalculateSRouteOut$, $CalculateERouteOut$, et $CalculateWRouteOut$ gèrent ce dernier cas selon le port de sortie où le message se trouve.

Dans le listing 5.26, nous donnons la définition de $CalculateNRouteIn$. La fonction commence en appelant la fonction $CalculateNorthNode$ qui calcule le voisin immédiat du nœud dans la direction du nord. Si le nœud est sur la bordure nord et le message réside sur le port de sortie nord, la fonction retourne le même port mais dans la direction

de l'entrée, en raison du rebouclage. La fonction *CalculateNRouteIn* invoque la fonction *NosRoutingLogic* en lui passant le voisin calculé par la fonction *CalculateNorthNode*.

```
(defun CalculatenNRouteIn (current to nodeset)
(let* ((x_c (car current))
       (y_c (cadr current))
       (north_output (list x_c y_c 'N 'O))
       (north_node (calculateNorthnode current nodeset))
       (north_route (NosRoutingLogic north_node to)))
       (cons current (cons north_output north_route))))
```

Listing 5.26: Définition de la fonction *CalculateNRouteIn*

Pour chacune des fonctions de calcul de la route, nous prouvons que la route commence par le nœud courant, finit bien par la destination, et tous les ports qui forment cette route sont inclus dans l'ensemble *NodeSet* de Nostrum.

Voyons maintenant la définition de la fonction principale de routage *HPLogic* (listing 5.27). Dans le cas où le message se trouve sur un port de sortie, il ne peut passer qu'au port d'entrée du voisin "en face". Dans ce cas, aucun autre message n'est en compétition pour ce port (ligne 35-42).

```
(defun HPLogic (current to nodeset)
 (if (or (not (2dnodep current))
         (not (2dnodep to)))
     nil
     (let* ((x_d (car to))
            (y_d (cadr to))
            (x_c (car current))
            (y_c (cadr current))
            (port_c (caddr current))
            (direction_c (car(last current)))
            (x_distance (abs (- x_d x_c)))
            (y_distance (abs (- y_d y_c)))
            (nri (CalculatenNRouteIn current to nodeset))
            (sri (CalculateSRouteIn current to nodeset))
            (eri (CalculatenERouteIn current to nodeset))
            (wri (CalculatenWRouteIn current to nodeset)))
            (if (equal direction_c 'I)
                (if (and (equal y_d y_c)(equal x_d x_c))
                    (list (movelocal current to) nri eri sri wri)
                    (if (< x_distance y_distance)
                        (if (< y_c y_d)
                            (if (< x_d x_c)
                                (list nri wri sri eri)
                                (list nri eri sri wri))
                            (if (< x_d x_c)
                                (list sri wri nri eri)
                                (list sri eri nri wri)))
                        (if (< y_c y_d)
                            (if (< x_d x_c)
                                (list wri nri sri eri)
                                (list eri nri sri wri))
                            (if (< x_d x_c)
```

```
33                              (list wri sri nri eri)
                                (list eri sri nri wri)))) )
          (let ((nro (calculate_north_route_output current to nodeset))
                (sro (calculate_south_route_output current to nodeset))
                (ero (calculate_east_route_output current to nodeset))
38              (wro (calculate_west_route_output current to nodeset)))
                (cond  ((equal port_c 'N)  (list nro))
                       ((equal port_c 'S)  (list sro))
                       ((equal port_c 'E)  (list ero))
                       (t                  (list wro))))))))
```

Listing 5.27: Définition de la fonction *HPLogic*

Dans le cas de la présence du message sur un port d'entrée (ligne 17-34), nous vérifions tout d'abord si le message a atteint sa destination (ligne 18-19). Dans ce cas, la seule route prioritaire est celle qui permet au message de sortir par le port local. C'est elle qui est donc placée en tête de la liste des routes possibles, les autres routes étant placées dans un ordre quelconque arbitraire.

Dans les autres cas, nous devons respecter le critère décrit dans la section 2.6.2, et les routes sont donc triées dans un ordre de préférence qui favorise si possible que chaque message se rapproche de sa destination tout en prenant la route qui va lui permettre de voyager le plus longtemps sur l'axe choisi. Nous comparons la distance sur l'axe des X et l'axe des Y (ligne 20). Si la distance sur l'axe des Y est supérieure (ligne 21-27), la route prioritaire est toujours celle qui permet un mouvement le long de cet axe. Le message peut aussi avoir une deuxième route prioritaire (avec une priorité plus basse) le long de l'axe des X. L'ordre des autres routes est quelconque. Dans le cas inverse (où la distance le long de l'axe des X est plus grande) la route la plus prioritaire sera celle qui permet un mouvement au long de l'axe des X. Le tableau 5.5 résume pour chacun des cas la route (ou les deux routes) le(s) plus prioritaire(s) dans chacun des cas de figure. L'ordre des routes prend en compte le tableau 2.1 (section 2.6.2).

Distances	Coordonnées	Lignes	Ordre des routes
Dist X $<$ Dist Y	Xc $>$ Xd et Yd $>$ Yc	23	Nord, Ouest, Sud, Est
Dist X $<$ Dist Y	Xc $\leq$ Xd et Yd $>$ Yc	24	Nord, Est, Sud, Ouest
Dist X $<$ Dist Y	Xc $>$ Xd et Yd $\leq$ Yc	26	Sud, Ouest, Nord, Est
Dist X $<$ Dist Y	Xc $\leq$ Xd et Yd $\leq$ Yc	27	Sud, Est, Nord, Ouest
Dist X $\geq$ Dist Y	Xc $>$ Xd et Yd $>$ Yc	30	Ouest, Nord, Sud, Est
Dist X $\geq$ Dist Y	Xc $\leq$ Xd et Yd $>$ Yc	31	Est, Nord, Sud, Ouest
Dist X $\geq$ Dist Y	Xc $>$ Xd et Yd $\leq$ Yc	33	Ouest, Sud, Nord, Est
Dist X $\geq$ Dist Y	Xc $\leq$ Xd et Yd $\leq$ Yc	34	Est, Sud, Nord, Ouest

Table 5.5 – Ordre des routes

La fonction *HPLogic* utilise les fonctions définies au début de cette section pour calculer les routes [13]. De ce fait, toutes les propriétés prouvées pour ces fonctions sont valides pour *HPLogic*.

13. *CalculateNRouteIn*, *CalculateSRouteIn*, *CalculateERouteIn*, *CalculateWRouteIn*

```
(defun HPRouting (Missives nodeset)
  (if (endp Missives)
      nil
   (let* ((miss (car Missives))
          (From (OrgTM miss))
          (current (CurTM miss))
          (to (DestTM miss))
          (id (IdTM miss))
          (frm (FrmTM miss))
          (flits (FlitTM miss))
          (Time (TimeTM miss))
          (hopcount (hopcountTM miss)) )
     (cons (list id from frm  (HPLogic current to nodeset)
                 flits time hopcount)
           (HPRouting (cdr Missives) nodeset)))))
```

Listing 5.28: Définition de la fonction *HPRouting*

L'instance de la fonction *Routing* s'appelle *HPRouting* (listing 5.28). Nous prouvons alors les obligations de preuve du module du routage. Nous prouvons les propriétés de correction du résultat de la fonction du routage *HPRouting* (la preuve est triviale, elle se base sur les propriétés prouvées pour les fonctions précédentes). Finalement, nous pouvons établir que la fonction *HPRouting* est une instance de la fonction *Routing* à l'aide de la directive $definstance$ (listing 5.29).

```
(defthm TrLstp-HPRouting
  (let ((NodeSet (MeshNodeSetGen Params)))
    (implies (and (TMissivesp TMissives NodeSet)
                  (NostrumStGenerator Params))
             (TrLstp (HPRouting TMissives nodeset) nodeset))))

(defthm CorrectRoutesp-HPRouting
  (let ((NodeSet (MeshNodeSetGen Params)))
    (implies (and (meshhyps Params)
                  (TMissivesp TMissives NodeSet))
             (CorrectRoutesp (HPRouting TMissives nodeset)
                       TMissives NodeSet)))

(definstance GenericRouting checkcomplianceHProuting
  :functional-substitution
  ((NodeSetGenerator MeshNodeSetGen)
   (NodeSetp 2Dnodesetp)
   (ValidParamsp MeshHyps)
   (Routing HPRouting))
  :rule-classes nil
  :otf-flg t)
```

Listing 5.29: Obligations de preuve et preuve de conformité de *HPRouting*

CalculateNRouteOut, *CalculateSRouteOut*, *CalculateERouteOut*, et *CalculateWRouteOut*.

5.4.3.2 L'ordonnancement

Le réseau Nostrum utilise un mode de commutation par paquet. Comme précédemment, la fonction *Scheduling* effectue trois étapes : (1)la première consiste à s'assurer que le pas à effectuer du nœud courant au voisin est possible, (2)le saut est effectué, et (3)finalement nous libérons les places que les messages ont quittées dans l'étape (2). Nous détaillons chacune de ces étapes ci-dessous. Dans la modélisation de Nostrum, la fonction *getnextpriority* n'est pas nécessaire puisque le réseau n'utilise qu'une priorité basée sur l'age des messages (pas de composante indépendante de l'état courant dans le mécanisme de priorité). Nous ne la définissons pas.

La fonction *HPS* (listing 5.30) est le cœur de notre implémentation. Elle est responsable de la prise des décisions d'ordonnancement. Elle reçoit six paramètres :

- *trlst* : la liste des messages à ordonnancer,
- *enroute* : l'accumulateur des messages qui n'ont pas encore atteint leur destination (initialement vide),
- *ntkstate* : l'état courant du réseau,
- *arracc* : l'accumulateur des messages arrivés (initialement vide),
- $2Bfreed$: la liste des ports à libérer (suite au départ d'un message de ce port),
- *nonodes* : un accumulateur des ports qui viennent de recevoir un message durant le cycle courant.

Si la liste *trlst* est vide, la fonction retourne les accumulateurs *enroute*, *ntkstate*, *arracc*, $2Bfreed$ et *nonodes*. Autrement, la fonction vérifie la présence d'une route valide que le message peut emprunter à l'aide de la fonction *NostrumTestRoutes* (voir section 5.4.4) qui choisit une route libre et valide s'il en existe une. Si une route $r?$ est trouvée, elle est renvoyée en sortie de cette fonction *NostrumTestRoutes* ainsi que l'état du réseau sans modification.

La fonction *HPS* effectue ensuite un test pour vérifier que le hop peut être effectué. La première condition est que $r?$ n'est pas vide, ce qui signifie la présence d'une route valide. La fonction vérifie ensuite que le message ne réside pas sur un port qui a déjà vu un message partir dans ce cycle. La définition est ainsi générale et peut être utilisée pour des réseaux ayant des ports contenant des buffers à plusieurs places. La définition n'a pas besoin d'être changée même si le nombre de place du buffer change (ligne 8). La troisième condition (ligne 9) consiste à vérifier que le port destinataire de saut n'a pas déjà reçu un message durant ce cycle. Ce test évite qu'un port reçoive plusieurs messages durant le même cycle dans le cas des buffers à plusieurs places.

Les lignes 10-20 implémentent le cas où ces trois conditions sont satisfaites donc le message peut être ordonnancé. Si le message atteint sa destination (*i.e*, la route a une longueur de deux), le message est ajouté à l'accumulateur *arracc*. Sinon, le message sera ajouté à *enroute* après la mise à jour de la route pour simuler un saut à l'aide de la fonction *UpdateRoute* (listing 5.11) (comme dans le cas de Hermes). Le port duquel le message est parti sera ajouté à l'accumulateur $2Bfreed$ pour le libérer à la fin du cycle. Tandis que le port auquel le message arrive sera ajouté à *nonodes*. Si le test des lignes 7-9 échoue, le message est simplement ajouté à *enroute* (lignes 21-23).

```
(defun HPS (trlst enroute ntkstate arracc 2BFreed nonodes)
 (if (endp TrLst)
     (mv enroute arracc ntkstate 2BFreed nonodes)
     (let ((tr (car trlst)))
      (mv-let (intermediatest r?)
              (NostrumTestRoutes ntkstate (car trlst))
        (if (and r?
                 (not (member-equal (car r?) 2bfreed))
                 (not (member-equal (cadr r?) nonodes)))
            (HPS (cdr Trlst)
                 (if (equal (len r?) 2)
                     enroute
                     (cons (update_route tr r?) enroute))
                 (MyUpdateState intermediatest
                                (list (update_route tr r?)))
                 (if (equal (len r?) 2)
                     (cons tr arracc)
                     arracc)
                 (cons (car r?) 2BFreed)
                 (cons (cadr r?) nonodes))
            (HPS (cdr TrLst)
                 (cons tr enroute) intermediatest arracc
                 2BFreed nonodes))))))
```

Listing 5.30: Définition de la fonction HPS

La fonction *PacketScheduling* (listing 5.31) est l'instance de la fonction générique *Scheduling*. Elle prend en entrée la liste des voyages à ordonnancer *trlst*, la liste des tentatives *att*, l'ensemble des ports du réseau *NodeSet*, l'état actuel du réseau *NtkState*, et *order* [14].

```
(defun PacketScheduling (TrLst att NodeSet NtkState order)
(declare (ignore NodeSet))
 (if (zp (SumOfAttempts att))
     (mv (totmissives Trlst) nil Att NtkState)
   (mv-let (enroute arracc newNtkState 2BFreed nonodes)
   (HPS (NostrumPriority TrLst order) nil
              (calculate_loads trlst (shift_loads NtkState)) nil nil nil)
   (mv (totmissives enroute) arracc  (consume-attempts att)
       (treat-arrived-messages (freeNodes newNtkState 2BFreed)
                               nonodes (freeNodes newNtkState 2BFreed)))))))
```

Listing 5.31: Définition de la fonction *PacketScheduling*

La fonction vérifie que la somme des tentatives n'est pas nulle (à l'aide de la fonction *SumOfAtt*). Le cas échéant, elle renvoie ses entrées sans effectuer d'ordonnancement (après conversion de la liste des voyages en liste de traveling missives). Sinon la fonction HPS est appelée avec les entrées suivantes : la liste des voyages *trlst* après avoir appliqué la politique de priorité par âge à l'aide de la fonction *NostrumPriority* (voir section 5.4.3.3)

14. Ce dernier paramètre est présent pour respecter le modèle générique mais ne sera pas nécessaire, compte tenu de l'absence de la fonction *getnextpriority* (roir page 104).

et l'état actuel du réseau *ntkstate* après avoir effectué la mise à jour des charges à l'aide des fonctions $ShiftLoads$ et $CalculateLoads$ (voir section 5.4.4).

La fonction rend en résultat :
- La liste des voyages en route *enroute* après l'avoir convertie en liste de traveling missives,
- la liste des messages qui ont atteint leur destination *arracc*,
- la liste des tentatives après en avoir consommé à l'aide de la fonction *consumeattempts*,
- le nouvel état du réseau après diverses mises à jour, dont la libération des buffers que les messages ont quittés, à l'aide des fonctions *treat-arrived-messages* et $freenodes$.

La fonction $freenodes$ se charge de la libération des ports que les messages ont quittés durant le cycle actuel. La fonction *treat-arrived-messages* parcourt l'état du réseau et libère les ports locaux de sortie qui contiennent un message arrivé sous la condition que le port ne soit pas dans la liste *nonodes* [15].

Les différentes obligations de preuve de la section 4.2.4.2 ont été prouvées à l'aide de 10 fonctions, et 69 théorèmes en 103,61 secondes.

5.4.3.3 La priorité

Le réseau Nostrum se base sur l'âge des messages (hopcount) pour décider de leur priorité durant la résolution des contentions, les plus anciens messages étant prioritaires. L'implémentation de ce mode de priorité ne nécessite pas l'instanciation de la fonction *getnextpriority* car le port d'arrivée n'est pas concerné.

La fonction $NostrumPriority$ est la modélisation de cette politique. Comme dans le cas de Hermes, elle est implémentée sous forme de fonction de tri. La fonction trie les messages selon le nombre de sauts effectués. En cas d'égalité, celui qui est le plus proche de sa destination remporte l'arbitrage. La fonction prend en entrée la liste de messages à trier, et la deuxième entrée est ignorée.

Les obligations de preuve de ce module ont été prouvées à l'aide de 10 fonctions et 22 théorèmes en 4,93 secondes.

5.4.4 Couche liaison de donnée- la synchronisation

Chaque routeur communique la moyenne de sa charge durant les quatre derniers cycles à ses voisins. Ce calcul de moyenne et la mise à jour de l'état sont effectués par des fonctions définies au sein du module de synchronisation. Deux fonctions (listing 5.32) sont nécessaires pour la gestion des charges. La première, $ShiftLoads$ est responsable de décaler les charges en éliminant la charge la plus ancienne afin de pouvoir mémoriser une nouvelle valeur au début de chaque cycle d'exécution. Elle prend en entrée l'état du réseau, le parcourt en décalant les charges de tous les ports, puis elle renvoie le nouvel état. Elle utilise la fonction $Shiftload$ qui décale les charges et remplit la case de la charge la

15. Un message sur un port local de sortie est un message qui a atteint sa destination. Si port est dans la liste *nonodes*, le message vient donc d'occuper ce port. Nous ne devons pas libérer le port dans ce cycle, nous le libérons dans le cycle suivant.

plus récente avec 0.

```
(defun ShiftLoads (st)
  ;; shifting the loads of the states
  (if (endp st)
      nil
    (cons (ShiftLoad (car st)) (ShiftLoads (cdr st)))))

(defun CalculateLoads (trlst st)
  (if (endp trlst)
      st
    (let* ((tr (car trlst))
           (routes (routesV tr))
           (current (caar routes)))
      (CalculateLoads (cdr trlst) (incrementLoad st current)))))
```

Listing 5.32: Définition des fonctions *ShiftLoads* et *CalculateLoads*

La deuxième fonction, *CalculateLoads*, est responsable de la mise à jour de la charge de chaque port. La fonction prend en entrée la liste des messages dans le réseau ainsi que son état courant. Pour chaque message, elle invoque la fonction *incrementLoad* du module de l'état global du réseau, en lui passant le port où réside le message pour incrémenter la charge du nœud.

La fonction principale *NostrumTestRoutes* (listing 5.33), qui sera utilisée par la fonction d'ordonnancement, prend un voyage et l'état actuel du réseau en entrée. Elle essaie de trouver une route libre valide que le message peut suivre. Si une route existe, elle est renvoyée en résultat de la fonction ainsi que l'état du réseau. Elle utilise la fonction *NostrumGoodRoutes* afin d'effectuer sa tâche.

```
(defun NostrumGoodRoutes (st org dest routes)
  (let ((current (caar routes))
        (destination (car (last (car routes)))))
    (if (endp routes)
        (mv st nil)
      (if (or (equal (car current) (car destination))
              (equal (cadr current) (cadr destination)))
          (only_first_counts st org dest routes)
        (first_two_count st org dest routes)))))

(defun NostrumTestRoutes (st tr)
  (let* ((routes  (routesv tr))
         (dest (car (last (car routes))))
         (org (orgv tr)))
    (mv-let (newst r?)
            (NostrumGoodRoutes st org dest routes)
            (mv newst r?))))
```

Listing 5.33: Définition des fonctions *NostrumTestRoutes* et *NostrumGoodRoutes*

Selon le positionnement du message par rapport à sa destination, une ou deux routes

sont prioritaires. C'est la fonction *NostrumGoodRoutes* qui prend ce point en considération. Les routes ont été ordonnées selon les règles définies dans le tableau 2.14 par la fonction de routage. La fonction *NostrumGoodRoutes* (listing 5.33) teste la position actuelle par rapport à la destination afin de connaître s'il existe une route préférée ou deux. Si le port courant possède une coordonnée en commun avec la destination, c'est seulement la première route dans la liste des routes qui est prioritaire, sinon ce sont les deux premières.

Si une seule route est favorisée, la fonction $only\_first\_counts$ est appelée en lui passant l'état du réseau, l'origine du message, sa destination et la liste des routes du message. Elle teste si la route préférée (la première dans la liste) du message peut être empruntée. Si c'est possible, cette route ainsi que l'état du réseau sont renvoyés à la fonction *NostrumGoodRoutes* qui les renvoie à son tour à la fonction *NostrumTestRoutes* . Sinon, toutes les routes seront ordonnées dans l'ordre croissant des charges du deuxième [16] port sur la route et la fonction essayera de trouver parmi ces routes ordonnées la première route libre empruntable. La fonction $first\_two\_count$ fonctionne de la même façon, mais considère les deux premières routes avant d'ordonner les routes selon les charges.

Les fonctions $only\_first\_counts$ et $first\_two\_count$ utilisent la fonction $Noschkavail$ qui est l'instanciation de la fonction *chkavail*. La fonction verifie les trois conditions décrites dans le chapitre précédent et vérifie que le buffer du deuxième port sur la route est libre à l'aide de la fonction $Nosprocessreq$ qui est l'instance de la fonction *processreq*. La fonction *reqtrans* n'est pas instanciée puisque le réseau n'utilise pas de protocole de communication point à point.

Une fois ces fonctions définies, les différentes obligations de preuve de ce module sont prouvées. Pour modéliser le module de synchronisation du réseau Nostrum, 17 fonctions ont été nécessaires ainsi que 14 théorèmes pour un temps de preuve de 4,69 secondes.

5.4.5 L'instanciation de *GeNoC*

Comme dans le cas de Hermes, nous créons les instances des fonctions $GeNoC_t$ et $GeNoC$: $NostrumGeNoC\_t$ (listing 5.34) et $NostrumGeNoC$ (listing 5.35) respectivement.

La procédure de vérification des propriétés de correction consiste à créer les instances de ces deux fonctions et les théorèmes de corrections sont obtenus directement. Le processus d'instanciation nécessite la définition de deux fonctions et 4 théorèmes pour un temps de preuve de 13,17 secondes. Comme pour les 2 autres NoCs, nous avons pu ainsi prouver que :

- *les messages arrivés atteignent leur destination sans altération dans leur contenu,*
- *le réseau ne perd pas de message.*

16. Le premier étant la position actuelle, le deuxième est la destination du saut.

```
(defun NostrumGeNoC_t (m nodeset att trlst time p order accup)
 (declare (xargs :measure (sumofattempts att)))
 (if (zp (sumofattempts att))
     (mv trlst m accup )
   (mv-let (delayed departing )
           (Hermesreadyfordeparture m nil  nil time)
           (let ((v (HPRouting departing nodeset))
             (mv-let (newtrlst arrived  newatt newp)
                     (PacketScheduling  v  att nodeset p order)
                     (NostrumGeNoC_t (append newtrlst delayed)
                               nodeset newatt
                                       (append arrived trlst)
                                       (+ 1 time) newp
                                       (GetNextPriority order)
                                       (append accup (list newp)))))))))
```

Listing 5.34: Définition de la fonction *NostrumGeNoC_t*

```
(defun NostrumGeNoC (trs nodeset att p1 p2 order)
 (declare (ignore p2)
 ;; main function
 (mv-let (responses aborted accup )
         (NostrumGeNoC_t (computetmissives trs) nodeset att nil '0
                   (2d-state-generator-with-ports p1 '1) order
                   (list (2d-state-generator-with-ports p1 '1)))
         (mv (computeresults responses) aborted accup)))
```

Listing 5.35: Définition de la fonction *NostrumGeNoC*

5.4.6 La simulation

Comme dans le cas du réseau Hermes, nous avions accès au code VHDL du réseau Nostrum qui nous a été fourni par Axel Jantsch (Royal Institute of Technology, Stockholm). Axel Jantsch nous a également apporté un support pour la compréhension de certaines caractéristiques du réseau et certains choix dans la mise en œuvre. Nous avons donc pu comparer les résultats de la simulation VHDL et l'exécution de notre modèle ACL2 du réseau. Nous utilisons 3 messages dans l'exemple présenté ici (tableau 5.6 et figure 5.7). Les messages 2 et 3 vont se trouver en compétition sur le routeur (3 3), c'est la raison pour laquelle le message 3 va se trouver dévié par le nœud (4 3).

ID	Source	Destination	Contenu	Temps	Couleur
1	(1 3)	(4 3)	111111111111111	0	vert
2	(3 4)	(3 1)	22222222222222	0	rose
3	(3 3)	(3 1)	33333333333333	1	gris

Table 5.6 – Résumé des messages de la simulation du réseau Nostrum

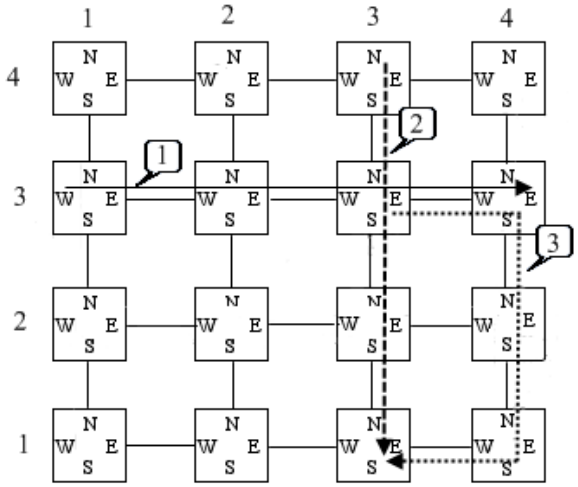

FIGURE 5.7 – Exemple de simulation du réseau Nostrum

Simulation VHDL. La figure 5.8 montre le résultat de la simulation VHDL. Nous montrons pour chaque message le port local d'entrée du message au réseau ainsi que le port où il sortira du réseau. Pour l'évolution du message dans le réseau nous montrons seulement les ports d'entrée sur lesquels le message réside à un instant donné. Le port de sortie, emprunté au cycle d'avant, est dans ce cas trivial à trouver. Chaque signal porte le nom *Data_ShowXY(port)*, X et Y sont les coordonnées X et Y du nœud, tandis que *port* est le port d'entrée.

Le tableau 5.7 et la figure 5.7 montrent pour chacun des messages la route suivie pour arriver à sa destination. Nous montrons tous les ports où le message transitera. Le message 1 suit sa route tel qu'elle est dans le tableau sans contention. Le messages 2 et 3 vont être

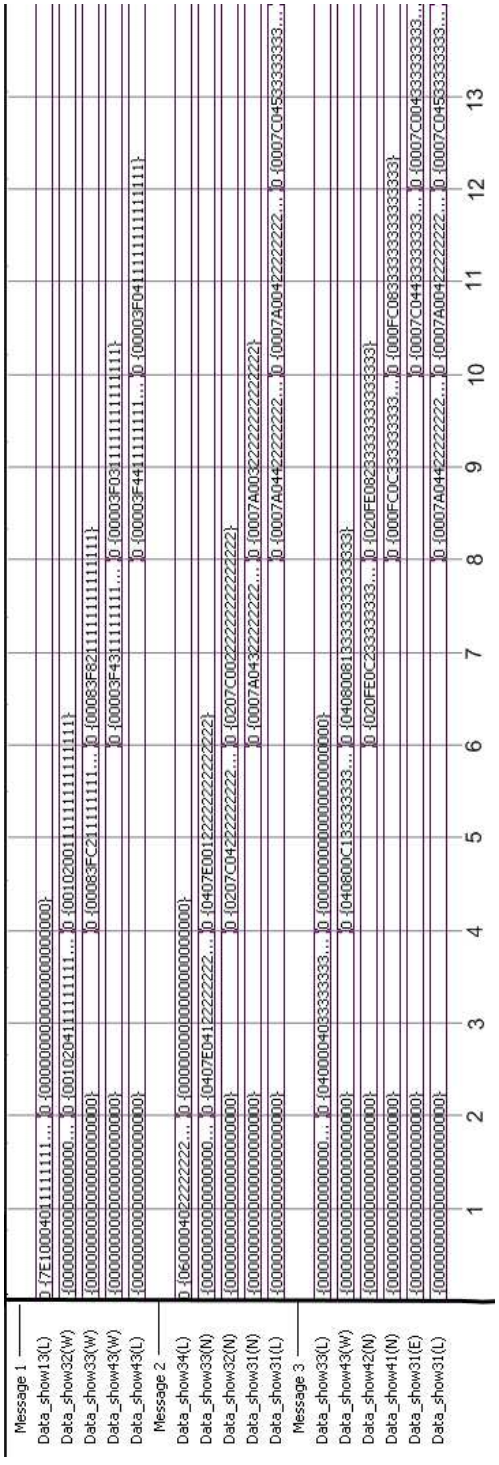

FIGURE 5.8 – Résultat de la simulation VHDL du réseau Nostrum

en compétition pour le port (3 3 S o). Sur la figure 5.8 (repère 2) nous pouvons voir le message 2 arriver sur le nœud (3 3) par le port ouest, et le message 3 par le port local. La route favorisée pour les deux messages passe par le port sud du nœud (3 3). Le message 2, étant plus vieux, gagne l'arbitrage et aura accès au port de sortie sud du nœud (3 3). Il poursuit son voyage pour atteindre sa destination (3 1) (repère 8) en transitant par le nœud (3 2) (repère 6). Le message 3 ayant perdu l'arbitrage s'éloigne de sa destination et emprunte le port de sortie *est* du nœud[17]. Le message 3 arrive donc sur le nœud (4 3). Il se dirige ensuite vers le sud passant ainsi sur les nœuds (4 2) (repère 6), et (4 1) (repère 8). Pour atteindre le nœud (3 1), le message quitte le nœud (4 1) par le port ouest (repère 10).

ID	Route
1	(1 3 L i) (1 3 E o) (2 3 W i) (2 3 E o) (3 3 W i) (3 3 E o) (4 3 W i) (4 3 L o)
2	(3 4 L i) (3 4 S o) (3 3 N i) (3 3 S o) (3 2 N i) (3 2 S o) (3 1 N i) (3 1 L o)
3	(3 3 L i) (3 3 E o) (4 3 W i) (4 3 S o) (4 2 N i) (4 2 S o) (4 1 N i) (4 1 W O) (3 1 E i) (3 1 L o)

Table 5.7 – Routes suivies par les messages (Simulation VHDL du réseau Nostrum)

Simulation ACL2. Nous avons exécuté ce même exemple dans notre modèle ACL2. Nous obtenons les mêmes résultats. Les messages 1 et 2 sont injectés au premier cycle d'exécution. Le message 3 est injecté au deuxième cycle. Les messages avancent dans le réseau en suivant les routes du tableau 5.7. Au cycle 2 (figure 5.9),les messages 2 et 3 atteignent le nœud (3 3). Les deux essayent d'obtenir le port de sortie sud. Le message 2 gagne l'arbitrage comme dans le cas du modèle VHDL. Au cycle 3 (figure 5.10), le message 2 emprunte le port de sortie sud du nœud (3 3) tandis que le message 3 emprunte le port *est*. Le message 3 atteint ainsi le nœud (3 4) au cycle suivant à travers le port d'entrée *ouest* (figure 5.11). Il continue son voyage en se dirigeant vers le sud pour atteindre le nœud (4 1). Le message emprunte ensuite le port ouest de ce nœud pour atteindre le nœud (3 1). Il atteint ainsi sa destination au cycle 12.

5.5 Conclusion

Nous avons exposé dans ce chapitre l'application de notre modèle pour la vérification de plusieurs réseaux. Hermes est un réseau issu de travaux académiques. Il a constitué notre première étude de cas. Le choix du réseau Spidergon permet de démontrer que *GeNoC* est utilisable pour vérifier des réseaux industriels. Enfin, Nostrum, un autre réseau académique, a été choisi pour illustrer que le modèle est capable de couvrir des algorithmes de routage adaptatifs non-minimaux.

17. C'est la direction où la charge est la plus basse puisque le message 1 est sur le nœud (3 2) à cet instant.

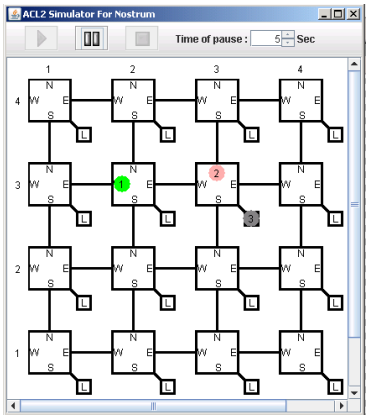

FIGURE 5.9 – Simulation Nostrum Cycle 2

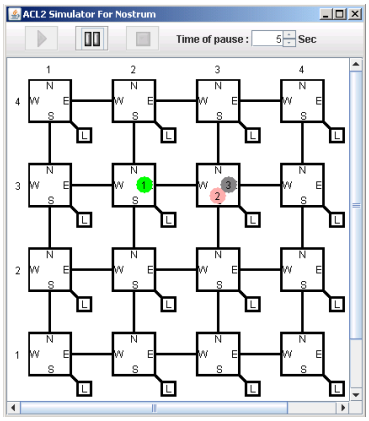

FIGURE 5.10 – Simulation Nostrum Cycle 3

Le tableau 5.8 résume le nombre de fonctions et de théorèmes ainsi que les temps de preuve de chaque module. La machine utilisée est un Core Duo à 1,6 GHz, avec 1 GB de mémoire et fonctionnant sous Windows XP.

Le temps humain, très difficile à évaluer, varie selon le niveau d'expertise de l'utilisateur et la complexité du réseau considéré. Le temps nécessaire pour la modélisation d'un réseau par un utilisateur expérimenté est évalué à 2 à 4 semaines selon le réseau. Pour un nouvel utilisateur, le temps nécessaire pour la prise en main du modèle varie entre 2 et 3 semaines. Dans ce chapitre, nous avons montré les étapes à suivre pour la vérification de nouveaux réseaux selon la méthodologie fournie par *GeNoC*. L'approche est systématique. *GeNoC* propose un cadre formel ainsi qu'un ensemble minimal d'obligations de preuve, suffisantes pour la vérification de deux propriétés de correction. Même si les instantiations pour un réseau donné doivent être réalisées manuellement, ce chapitre a montré que le même schéma de raisonnement se reproduit pour chaque NoC, et en outre que la réutilisabilité de composantes validées est possible.

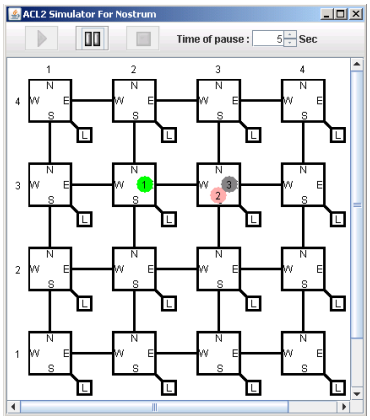

FIGURE 5.11 – Simulation Nostrum Cycle 4

Réseau	Module	Fonctions	Théorèmes	Temps
Hermes	Nodeset	10	9	3,5 s
	État du réseau	18	16	0,13 s
	Interfaces	4	15	1,01 s
	Accès au réseau	3	26	5,12 s
	Routage	12	47	393,11 s
	Ordonnancement	16	64	331,71 s
	Priorité	11	23	39,19 s
	Synchronisation	7	6	0,66 s
	GeNoC	2	4	13,88 s
	Total	86	204	788,4 s
Spidergon	Nodeset	10	4	0.71s
	État du réseau	17	13	0,2 s
	Interfaces	4	15	1,01 s
	Accès au réseau	3	26	5,12 s
	Routage	28	79	1150,28 s
	Ordonnancement	16	63	98,23 s
	Priorité	7	22	23,51 s
	Synchronisation	6	3	0,09 s
	GeNoC	2	5	147 s
	Total	96	227	1426,12 s
Nostrum	Nodeset	10	9	3,5 s
	État du réseau	13	11	0,14 s
	Interfaces	4	15	1,01 s
	Accès au réseau	3	26	5,12 s
	Routage	30	140	524,67 s
	Ordonnancement	10	69	103,61 s
	Priorité	10	22	4,93 s
	Synchronisation	17	14	4,69 s
	GeNoC	2	4	13,17 s
	Total	102	308	660,83 s

Table 5.8 – Fonctions, théorèmes, et temps de preuve pour l'application de *GeNoC*

Chapitre 6

Conclusion et perspectives

Nous avons exposé dans le chapitre 3 la première version d'une formalisation des architectures de communication. Cette première version présentée dans la thèse de Julien Schmaltz [Sch06], s'articulait autour de quatre fonctions principales *send*, *recv*, *Routing*, et *Scheduling*. Chacune d'elles est associée à une série de propriétés indispensables pour garantir le bon fonctionnement de l'architecture de communication. La fonction *GeNoC* est la modélisation d'une architecture de communication générique. L'ensemble des obligations de preuve implique le théorème de correction de *GeNoC* : *tout message arrivé a atteint son bon destinataire sans altération de son contenu.* La vérification d'un réseau donné revient donc à instancier les différents composants et prouver que les instances satisfont les obligations de preuve.

Nous avons ensuite présenté notre nouvelle version du modèle *GeNoC*. Cette version nous permet de modéliser une grande quantité des caractéristiques des couches transport, réseau, et liaison de donnée du modèle OSI. Nous avons ajouté aux modules déjà existants dans l'ancienne version un module permettant de modéliser l'état global de l'architecture de communication, un module de contrôle d'accès au réseau (couche transport), un module pour les priorités (couche réseau), et un module de synchronisation entre les nœuds (couche liaison de donnée). Une nouvelle propriété de correction a été ajoutée : *aucun message n'est perdu.* De nouvelles obligations de preuve ont été ajoutées en conséquence.

Grâce à la notion explicite du temps et à la modélisation pas à pas qui ont été rajoutées, nous pouvons maintenant avoir une preuve/simulation plus réaliste. Comme nous l'avons montré dans le chapitre 5, cela nous permet en particulier de modéliser des algorithmes de routage adaptatifs non-minimaux.

Finalement, nous avons exposé différents cas d'études permettant de vérifier les capacités de notre modèle. Le réseau Hermes nous a permis d'illustrer la prise en compte du mode de commutation par ver de terre. Le réseau Spidergon montre l'adaptabilité de notre modèle aux applications industrielles. La preuve du réseau Nostrum offrait plusieurs particularités complexes : algorithmes de routage adaptatif non-minimal, priorité d'entrée entre les messages, une priorité de sortie, et une synchronisation par communication de la moyenne de la charge des routeurs à leurs voisins.

Discussion et perspectives

NoC et *GeNoC*. Nous avons déjà évoqué dans les chapitres 1 et 2 l'importance croissante des NoCs. Hormis leur intérêt grandissant dans les milieux purement industriels, on peut aussi noter par exemple l'attention que leur porte l'agence spatiale européenne, qui a organisé en Septembre dernier une table ronde dédiée aux réseaux sur puce [1]. L'objectif annoncé est d'envisager l'utilisation de cette technologie dans les SoCs spatiaux.

Dans ce contexte, nos travaux sur la vérification des communications dans les NoCs prennent de plus en plus de pertinence. Il est cependant clair que notre modélisation à très haut niveau d'abstraction permet des preuves de propriétés de correction essentielles mais n'est pas conçue pour la vérification de certaines caractéristiques très fines, comme certaines propriétés sur le protocole de communication point à point ou la gestion de canaux virtuels, qui requièrent de se placer à un niveau plus proche des signaux de la description VHDL par exemple. Il pourrait donc être étudié comment complémenter nos résultats par exemple par des vérifications de propriétés au niveau RTL comme le permet l'outil HORUS développé dans notre équipe [OMAB08] : des assertions logico-temporelles exprimées en PSL [IEE05] peuvent être vérifiées dynamiquement, en cours de simulation ou émulation sur FPGA. De telles vérifications peuvent être menées sur des descriptions VHDL de NoCs.

Interblocage. Les travaux présentés dans [VS09] visent à proposer des solutions pour l'étude des interblocages. Il s'agit d'une application directe de notre modèle *GeNoC*. Au lieu d'imposer l'arrêt de la fonction *GeNoC* par une décroissance systématique du nombre de "tentatives", la terminaison est en quelque sorte laissée à la responsabilité de la fonction *Scheduling*, et le simple fait de pouvoir définir cette fonction sera associé à l'absence d'interblocage. Plus exactement, une mesure implicitement associée à la fonction d'ordonnancement est introduite dans notre modèle. La fonction $GeNoC_t$ termine si (1) tous les messages sont arrivés, ou (2) la mesure n'est pas valide, ou (3) aucun message ne peut être ordonnancé. Dans le cas où aucun message ne peut être ordonnancé un interblocage existe. La preuve de l'absence d'interblocage repose sur l'idée de vérifier que les messages dans le réseau sont tous évacués (ils atteignent tous leurs destinations). Dans ce cas, aucun interblocage existe.

Dans [VS10], les auteurs raffinent leurs méthode pour la vérification de l'absence d'interblocage dans les algorithmes de routage déterministes en codant dans *GeNoC* la condition de Dally et Seitz : un algorithme de routage ne présente pas de risque d'interblocage *si et seulement si* il n'y a pas de cycle dans le graphe de dépendance des canaux. Si la preuve de l'absence d'interblocage est réussie, la terminaison de la fonction récursive $GeNoC_t$ permet de prouver l'évacuation des messages.

Ces travaux qui démarrent sur les aspects liés aux interblocages ont été rendus possibles grâce à notre apport au modèle *GeNoC*. La modélisation d'origine était à bien trop gros grain pour permettre la prise en compte de telles propriétés. Notre modèle plus fin a donc ouvert un certain nombre de perspectives sur la vérification de propriétés de base dans les réseaux.

1. http ://conferences.esa.int/01C25/NoC

Extraction de modèles. Le modèle actuel raisonne à un niveau d'abstraction très élevé. Il faudrait pouvoir le rapprocher du modèle RTL. Un outil capable d'extraire la modélisation ACL2 au moins en partie automatiquement à partir de la modélisation RTL faciliterait la tâche de raisonnement et de preuve. A cette fin, un outil comme VSYML [OBMAP09], développé dans notre équipe, pourrait permettre d'extraire certaines parties du modèle à partir du code VHDL.

Troisième partie

Annexes

Annexe A

Code ACL2 pour *GeNoC*

A.1 Code pour les types de données

```
Filename:GeNoC-types.lisp
(in-package "ACL2")
(include-book "data-structures/list-defuns"  :dir :system)
(include-book "data-structures/list-defthms"  :dir :system)
;;-------------------------------------------------------------------
;;                          TRANSACTIONS
;;-------------------------------------------------------------------
;; A transaction is a tuple t = (id A msg B flits time)
;; Accessors are IdT, OrgT, MsgT, destT, FlitT, TimeT
(defun Idt   (trans) (car trans))
(defun OrgT  (trans) (nth 1 trans))
(defun MsgT  (trans) (nth 2 trans))
(defun DestT (trans) (nth 3 trans))
(defun FlitT (trans) (nth 4 trans))
(defun TimeT (trans) (nth 5 trans))
;; The function that grabs the ids of a list of transactions.
(defun T-ids (Transts)
  (if (endp Transts)
      nil
     (cons (list (caar Transts)) (T-ids (cdr Transts)))))
;; The function that grabs the messages of a list of transactions.
(defun T-msgs (Trs)
  (if (endp Trs)
      nil
     (cons (MsgT (car trs)) (T-msgs (cdr Trs)))))
;; The function that grabs the origins of a list of transactions.
(defun T-orgs (Trs)
  (if (endp Trs)
      nil
     (cons (OrgT (car trs)) (T-orgs (cdr Trs)))))
```

```
;; The function that grabs the destinations of a list of transactions.
(defun T-dests (Trs)
  (if (endp Trs)
      nil
     (cons (DestT (car trs)) (T-dests (cdr Trs)))))
;; The following predicate checks that each transaction has
;; the right number of arguments.
;; trans = (id A msg B flits time)
(defun validfield-transactionp (trans)
  (and (consp trans)
       (consp (cdr trans))            ;; (A msg B flits Time)
       (consp (cddr trans))           ;; (msg B flits Time)
       (consp (cdddr trans))          ;; (B flits Time)
       (consp (cddddr trans))         ;; (Flits Time)
       (consp (cdr (cddddr trans)))   ;; (Time)
       (null (cddr (cddddr trans))))) ;; nil
;; The following predicate recognizes a valid list of transactions(partially).
(defun Validfields-T (Transts)
  (if (endp Transts)
      t
      (let ((trans (car Transts)))
       (and (validfield-transactionp trans)
            (natp (Idt trans))                      ;; id is a natural
            (MsgT trans)                            ;; msg /= nil
            (natp (FlitT trans))                    ;; flit is a natural
            (natp (timeT trans))                    ;; time is a natural
            (not (equal (OrgT trans) (DestT trans))) ;; A /= B
            (Validfields-T (cdr Transts))))))
;; 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)
         (true-listp Transts)
         ;; the origins are members of the nodeset
         (subsetp (T-orgs Transts) NodeSet)
         ;; the destinations are members of the nodeset
         (subsetp (T-dests Transts) NodeSet)
         ;;No duplicates in the identifiers
         (No-Duplicatesp T-ids))))
;;------------------ end of Transactions ------------------------------
;;---------------------------------------------------------------------
;;                          MISSIVES
;;---------------------------------------------------------------------
;; We keep this type because it simplifies the coding of the
;; correctness of the enroute messages in the scheduling module.
;; A missive contains the same information in a traveling missive except the
```

```
;; current field and the hopcount. These are the information used to
;; verify the correctness of the enroute list.
;; A missive is a tuple m = (id A frm B Flit Time)
;; Accessors are IdM, OrgM, FrmM, DestM, FlitM and TimeM
(defun IdM (m) (car m))
(defun OrgM (m) (nth 1 m))
(defun FrmM (m) (nth 2 m))
(defun DestM (m) (nth 3 m))
(defun FlitM (m) (nth 4 m))
(defun TimeM (m) (nth 5 m))
;; The function that grabs the ids of a list of missives.
(defun M-ids (M)
  (if (endp M)
      nil
     (cons (list (caar M)) (M-ids (cdr M)))))
;; The function that grabs the origins of Missives.
(defun M-orgs (M)
  (if (endp M)
      nil
      (cons (list (OrgM (car M))) (M-orgs (cdr M)))))
;; The same for the destinations
(defun M-dests (M)
  (if (endp M)
      nil
    (cons (list (DestM (car M))) (M-dests (cdr M)))))
;; The function that grabs the frames of a list of missives.
(defun M-frms (M)
  (if (endp M)
      nil
    (let* ((msv (car M))
           (m-frm (FrmM msv)))
      (cons (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 flit time)
  (and (consp m)
       (consp (cdr m))              ;; (A frm B flits time )
       (consp (cddr m))             ;; (frm B flits time )
       (consp (cdddr m))            ;; (B flits time)
       (consp (cddddr m))           ;; (Flits Time)
       (consp (cdr (cddddr  m)))    ;; (time)
       (null (cddr  (cddddr  m))))) ;; nil
;; The following predicate recognizes a valid list of missives (partially).
(defun Validfields-M (M)
  (if (endp M)
      t
```

```
    (let ((msv (car M)))
      (and (validfield-missivep msv)
           (natp (IdM msv))                      ;; id is a natural
           (FrmM msv)                            ;; frm /= nil
           (natp (FlitM msv))                    ;; flit is a natural
           (natp (TimeM msv))                    ;; time is a natural
           (not (equal (OrgM msv) (DestM msv))) ;; A /= B
           (Validfields-M (cdr M))))))
;; now we define the predicate that recognizes a valid list of
;; missives
(defun Missivesp (M NodeSet)
  (let ((M-ids (M-ids M)))
    (and (Validfields-M M)
         ;;origins subset of nodeset
         (subsetp (M-orgs M) NodeSet)
         ;;destinations subset of nodeset
         (subsetp (M-dests M) NodeSet)
         (true-listp M)
         ;;No duplicates in the identifiers
         (No-Duplicatesp M-ids))))
;;-------------------- end of Missives --------------------------------
;;---------------------------------------------------------------------
;;                     TRAVELING MISSIVES
;;---------------------------------------------------------------------
;; A traveling missive is a tuple
;; tm = (id A current frm B Flit time hopcount)
;; Accessors are IdTM, OrgTM, curTM, FrmTM, DestTM and flitm
(defun IdTM (m) (car m))
(defun OrgTM (m) (nth 1 m))
(defun CurTM (m) (nth 2 m))
(defun FrmTM (m) (nth 3 m))
(defun DestTM (m) (nth 4 m))
(defun FlitTM (m) (nth 5 m))
(defun TimeTM (m) (nth 6 m))
(defun HopcountTM(m) (nth 7 m))
;; The function that grabs the ids of a list of traveling missives.
(defun TM-ids (M)
  (if (endp M)
      nil
    (append (list (IdTM (car M))) (TM-ids (cdr M)))))
;; The function that grabs the origins of a list of traveling missives.
(defun TM-orgs (M)
  (if (endp M)
      nil
    (append (list (OrgTM (car M))) (TM-orgs (cdr M)))))

;; The same for the currents.
```

```
(defun TM-curs (M)
  (if (endp M)
      nil
     (append (list (CurTM (car M))) (TM-curs (cdr M)))))
;; The same for the destinations.
(defun TM-dests (M)
  (if (endp M)
      nil
     (append (list (DestTM (car M))) (TM-dests (cdr M)))))
;; A function that grabs the frames of a list of traveling missives.
(defun TM-frms (M)
  ;; grabs the frames of M
  (if (endp M)
      nil
     (let* ((msv (car M))
            (m-frm (FrmTM msv)))
       (append (list m-frm) (TM-frms (cdr M))))))
;; The following predicate checks that a traveling missive has
;; the right number of arguments.
;; m = (id A current frm B time hopcount)
(defun validfield-Tmissivep (m)
  (and (consp m)
       (consp (cdr m))              ;;(A current frm B flits time hopcount)
       (consp (cddr m))             ;;(current frm B flits time hopcount)
       (consp (cdddr m))            ;;(frm B flits time hopcount)
       (consp (cddddr m))           ;;(B flits time hopcount)
       (consp (cdr (cddddr  m)))    ;;(flits Time hopcount)
       (consp (cddr (cddddr m)))    ;;(time hopcount)
       (consp (cdddr (cddddr m)))   ;;(hopcount)
       (null (cddddr (cddddr m))))) ;;nil
;; The following predicate recognizes a valid list of missives (partially).
(defun Validfields-TM (M)
  (if (endp M)
      t
    (let ((msv (car M)))
      (and (validfield-Tmissivep msv)
           (natp (IdTM msv))                      ;; id is a natural
           (FrmTM msv)                            ;; frm /= nil
           (natp (FlitTM msv))                    ;;flit is a natural
           (natp (TimeTM msv))                    ;;time is a natural
           (natp (HopcountTM msv))                ;;hopcount is a natural
           (not (equal (OrgTM msv) (DestTM msv))) ;; A /= B
           (not (equal (CurTM msv) (DestTM msv))) ;; current /= B
           (Validfields-TM (cdr M))))))
;; now we define the predicate that recognizes a valid list of
;; traveling missives.
(defun TMissivesp (M NodeSet)
```

```
  (let ((M-ids (TM-ids M)))
    (and (Validfields-TM M)
         (subsetp (TM-orgs M) NodeSet) ;;origines subset nodeset
         (subsetp (TM-curs M) NodeSet) ;;current subset nodeset
         (subsetp (TM-dests M) NodeSet) ;;destination subset nodeset
         (true-listp M)
         (No-Duplicatesp M-ids))))
;;-------------------- end of Traveling Missives ----------------------
;;---------------------------------------------------------------------
;;                          TRAVELS
;;---------------------------------------------------------------------
;; We add the origin since it will not always be in the routes.
;; A travel is a tuple tr = (id org frm Routes flits time).
;; Accessors are IdV, OrgV, FrmV, RoutesV, FlitV, TimeV and HopcountV.
(defun IdV (tr) (car tr))
(defun OrgV (m) (nth 1 m))
(defun FrmV (tr) (nth 2 tr))
(defun RoutesV (tr) (nth 3 tr))
(defun FlitV (tr) (nth 4 tr))
(defun TimeV (tr) (nth 5 tr))
(defun HopcountV(tr) (nth 6 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 function grabs the hopcounts of a list of travels.
(defun V-hop (TrLst)
  (if (endp TrLst)
       nil
      (append (list (hopcountV TrLst)) (V-hop (cdr TrLst)))))
;; We need a function that grabs the orgs of a list of travels.
(defun V-orgs (TrLst)
  (if (endp TrLst)
      nil
     (append (list (OrgV (car TrLst))) (V-orgs (cdr TrLst)))))
;;The function grabs the frames of a list of travels.
(defun V-frms (TrLst)
  (if (endp TrLst)
      nil
    (let* ((tr (car Trlst))
           (v-frm (FrmV tr)))
      (append (list v-frm) (V-frms (cdr TrLst))))))
;; The following predicate checks that each route of routes
;; has at least two elements and all are members of nodeset.
(defun validfield-route (routes nodeset)
  (if (endp routes)
```

```
      t
      (let ((r (car routes)))
        (and (consp r)
             (consp (cdr r))
             (subsetp r nodeset)
             (validfield-route (cdr routes) nodeset)))))
;; The following predicate checks that each travel has
;; the right number of arguments and that the origin and the current
;; are different from the last element of the routes.
;; tr = (id org frm Routes)
(defun validfield-travelp (tr nodeset)
  (and (consp tr)
       (consp (cdr tr))           ;; (org frm Routes flits time hipcount)
       (consp (cddr tr))          ;; (frm Routes flits time hopcount)
       (consp (cdddr tr))         ;; (Routes flits time hopcount)
 ;; (Routes flits time hopcount)  = (((R1) (R2) ...) flits time hopcount)
       (consp (routesv tr))
       (consp (cddddr tr))        ;;(Flits Time hopcount)
       (consp (cdr (cddddr tr)))  ;; (time hopcount)
       (consp (cddr (cddddr tr))) ;;(hopcount)
       (null (cdddr (cddddr tr)))
 ;; the origin is different from the last element of the route.
       (not (equal (orgv tr)
                   (car (last (car (routesv tr))))))
 ;; the first element of the route is different from the last.
       (not (equal (caar (routesv tr))
                   (car (last (car (routesv tr))))))
 ;;the origin is a member of nodeset
     (member-equal (orgv tr) nodeset)
     (validfield-route (routesv tr) nodeset)))
;; The following predicate recognizes a valid list of missives (partially).
(defun Validfields-TrLst (TrLst nodeset)
  (if (endp TrLst)
      t
    (let ((tr (car TrLst)))
      (and (validfield-travelp tr nodeset)
           (natp (IdV tr))            ;; id is a natural
           (FrmV tr)                  ;; frm /= nil
           (natp (FlitV tr))          ;; flit is a natural
           (natp (TimeV tr))          ;; time is a natural
           (natp (HopcountV tr))      ;; hopcount is a natural
           (true-listp (RoutesV tr))  ;; routes is a valid list
           (Validfields-TrLst (cdr TrLst) nodeset)))))
;; now we define the predicate that recognizes a valid list of
;; travels.

(defun TrLstp (TrLst nodeset)
```

```
  (let ((V-ids (V-ids TrLst)))
    (and (Validfields-TrLst TrLst nodeset)
         (true-listp TrLst)
         (No-Duplicatesp V-ids))))
;;-------------------- end of Travels ------------------------------------
;;------------------------------------------------------------------------
;;                          RESULTS
;;------------------------------------------------------------------------
;; A result is a tuple rst = (Id Dest Msg flit Time)
;; Accessors are IdR, DestR, MsgR, FlitR and TimeR.
(defun IdR (rst) (car rst))
(defun DestR (rst) (nth 1 rst))
(defun MsgR (rst) (nth 2 rst))
(defun FlitR (rst) (nth 3 rst))
(defun TimeR (rst) (nth 4 rst))
;; The function that grabs the ids of a list of results.
(defun R-ids (Results)
  (if (endp R)
      nil
     (append (IdR (car Results)) (R-ids (cdr Results)))))
;; The function grabs the destinations of a list of results.
(defun R-dests (Results)
  (if (endp Results)
      nil
     (append (DestR (car Results)) (R-dests (cdr Results)))))
(defun R-msgs (Results)
  ;; function that grabs the messages of a list of results.
  (if (endp Results)
      nil
     (append (MsgR (car results)) (R-msgs (cdr Results)))))
;; 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 Flit time)
       (consp (cddr rst))          ;; (Msg Flit time)
       (consp (cdddr rst))         ;; (flit time)
       (consp (cddddr rst))        ;;(time)
       (null (cdr(cddddr rst)))))  ;; nil
;; 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
```

```
             (MsgR rst)                          ;; msg /= nil
             (natp (FlitR rst))                  ;;flit is a natural
             (natp (TimeR rst))                  ;;time is a natural
             ;;destination is a member of nodeset
             (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))))
;;-------------------- end of Results -------------------------------------
;;-----------------------------------------------------------------------
;;                          ATTEMPTS
;;-----------------------------------------------------------------------
;; we just need to define the sum of the attempts.
(defun SumOfAttempts (att)
  ;; att has the following form:
  ;; ( (i RemAtt) (j RemAtt) ...)
  ;;where i and j are nodes.
  (if (endp att)
      0
      (let* ((top (car att))
             (rem-att (cadr top)))
        (+ (rem-att)
           (SumOfAttempts (cdr att))))))

;;-------------------- end of Attempts -------------------------------------
;;-----------------------------------------------------------------------
;;                          STATE
;;-----------------------------------------------------------------------
;; A state is a list of the form ((coor (node_id)) (buffers ...)) where
;; node_id a an element of NodeSet. We define accessors and predicates
;;defining valid state entries. We need accessors to the state elements.
;; st_entry =  ((coor (id)) (buffers ...) (optional))This function returns id.
(defun get_coor (st_entry)
   (cadar st_entry))
;; st_entry =  ( (coor (id)) (buffers x) (optional))This function returns x.
(defun get_buff (st_entry)
  (cadadr st_entry))
;; The function grabs the coordinates of all the state entries.
(defun getcoordinates (ntkstate)
  (if (endp ntkstate)
      nil
      (cons (get_coor (car ntkstate))(getcoordinates (cdr ntkstate)))))
```

```
;; The function grabs the buffers of all the state entries.
(defun get-buffers (ntkstate)
  (if (endp ntkstate)
      nil
    (cons (get_buff (car ntkstate)) (get-buffers (cdr ntkstate)))))
;; The function recognises a valid coordinate part of a state entry.
(defun validCoordp (x)
  (and (equal (car x) 'Coor)
       (consp x)
       (consp (cdr x))
       (null (cddr x))))
;; The function recognises a valid buffers part of a state entry.
(defun ValidBufferp (x)
  (and (equal (car x) 'Buffers)
       (consp x)
       (consp (cdr x))))
;; The function grabs all the coordinates of the state of a network and
;; verify that they are all valid coordinate part of a state entry.
(defun ValidCoordlist (x)
  (if (endp x)
      t
     (and (Validcoordp (caar x))
          (Validcoordlist (cdr x)))))
;; The function grabs all the bufferss of the state of a network and
;; verify that they are all valid buffers part of a state entry.
(defun ValidbuffersList (x)
  (if (endp x)
      t
    (and (ValidBufferp (cadar x))
         (Validbufferslist (cdr x)))))
;; The function verifies a state entry is valid using the validcoord and
;; validbuffer functions.
(defun Basic-entryp (st_entry)
  (if (endp st_entry)
      t
    (and (Validcoord (car st_entry))
         (Validbuffer (cadr st_entry)))))
```

A.2 Module des interfaces

```
Filename:GeNoC-interfaces.lisp
(in-package "ACL2")
(include-book "make-event/defspec" :dir :system)

(defspec GenericInterfaces
  (;; Any peer has an interface that can send and receive messages.
```

```
   ;; Function send
   ;; argument: a message msg
   ;; output: a frame frm
   ((send *) => *)
   ;; Function recv
   ;; argument: a frame frm
   ;; output: a message msg
   ((recv *) => *))
;;---------------------- Witness Functions -----------
  (local (defun send (msg) msg))
  (local (defun recv (frm) frm))
;;-------------------end Witness Functions -----------
  (defthm p2p-Correctness
    ;; the composition of recv and send
    ;; is the identity function.
    (equal (recv (send msg)) msg))
  ;; The following theorems are not proof obligations
  ;; but they will simplify proofs later on so we put them here.
  ;; A user will not need to proof them they are trivial to prove.
  (defthm send-nil
    ;; if msg is nil then send is nil too
    (not (send nil)))
  (defthm send-not-nil
    ;; if msg is not nil then send is not nil too
    (implies msg
           (send msg)))
  (defthm recv-nil
    ;; if frm is nil then recv is nil too
    (not (recv nil)))
  (defthm recv-not-nil
    ;; if frm is not nil then recv is not nil too
    (implies frm
           (recv frm))))
```

A.3 Module des nœuds

```
Filename:GeNoC-nodeset.lisp
(in-package "ACL2")
(include-book "make-event/defspec"  :dir :system)

(defspec GenericNodeSet
  ;; abstract set of nodes
  ;; the set is generated by the following function
  ;; its argument is the parameters
  (((NodesetGen *) => *)
   ;; the following predicate recognizes valid parameters
   ((ValidParamsp *) => *)
```

```
   ;; the following predicate recognizes a valid node
   ((ValidNodep *) => *)
   ((NodeSetp *) => *))
;;---------------------- Witness Functions -----------
  (local
   (defun ValidParamsp (pms)
   (declare (ignore pms))
   t))
  (local
   (defun NodesetGen (pms)
     (if (zp x)
        nil
       (cons x (NodesetGen (1- pms))))))
  (local
   (defun ValidNodep (n)
     (natp n)))
  (local
   (defun NodeSetp (l)
      (if (endp l)
        t
       (and (ValidNodep (car l))
            (NodeSetp (cdr l))))))
;;-------------------end Witness Functions -----------
  (defthm nodeset-generates-valid-nodes
    ;; the result of the nodesetgen is a valid list of nodes
    (implies (ValidParamsp pms1)
           (NodeSetp (NodesetGen pms1))))
  ;; we add a generic lemma only to simplify
  ;; the proofs of a given instance.
  ;; we force the user to prove it only to simplify the proofs later.
  ;; It is not part of the sufficient conditions to prove the
  ;; network correction.
  (defthm subsets-are-valid
     (implies (and (NodeSetp x)
                   (subsetp y x))
            (NodeSetp y)))
  ;; We prove that the result of NodeSetGen is a valid list
  (defthm true-listp-nodesetgen
    (implies (Validparamsp pms1)
           (true-listp (NodesetGen pms1))))
)
```

A.4 Module de l'état du réseau

```
Filename:GeNoC-ntkstate.lisp
(in-package "ACL2")
(include-book  "GeNoC-nodeset")
```

```
(include-book "GeNoC-misc")
(include-book "GeNoC-types")
(include-book "make-event/defspec" :dir :system)

(defspec GenericNodesetbuffers
  (;; Function StateGenerator generates
   ;; a state from two parameters
   ;; the first one is the parameter used to
   ;; generate the list of the nodes of the network
   ((StateGenerator * *) => *)
   ;; recognizer for valid parameters
   ((ValidstateParamsp * *) => *)
   ;; update the state
   ;; inputs = node_id (in NodeSet), a message and a state
   ;; returns a new state
   ((loadbuffers * * *)=> *)
   ;; read the state
   ;; input = node_id (in NodeSet) and a state
   ;; returns the state entry corresponding to node_id
   ((readbuffers * *) => *))
   ;; This function takes one parameter as input
   ;; and decides if it is a valid state entry
   ((ValidEntryP *) => *)
   ;; This function uses the previous function
   ;; to decide if a state is a valid state.
   ((ValidStatep *) => *)
   )
   ;; A network state is a list of node representing the state of the network
   ;; a state entry  has the form :
   ;;    (      (coor (...))
   ;;     (Buffers ...)  (optional)   )
   ;;
   ;;example of a network state:
   ;; ( ((coor (2 3)) (buffers 4 3) (optional))
   ;;    ((coor (3 2)) (buffers 5 3) (optional))
   ;;    ((coor (5 4)) (buffers 2 3) (optional)) )
   ;;
   ;; n: is the number of buffers on this specific coordinate
   ;; m: is the actual numbr of free buffers on this specific coordinate
   ;; example : ((coord (2 3)) (buffers 4 2))
   ;; This means that the node coordinate is (2 3) and that it has 4 buffers
   ;; of which only 2 are free.
;;---------------------- Witness Functions -----------
   ;; Local functions for the next witness function.
   ;; This function does not have to be instantiated.
   (local
    (defun stategeneratorlocal (nodeset y)
```

```
      (if (endp nodeset)
        nil
         (append (list (list (list 'Coor  (car nodeset)) (list 'Buffers y)))
               (StateGeneratorlocal (cdr nodeset) y)))))
   ;; A function taking a natural as input and generating a list of nodes.
   (local
    (defun StateGenerator (pms1 pms2)
     (let ((nodes (nodesetgenerator pms1)))
        (stategeneratorlocal nodes pms2))))
   ;; A function that verifies the input parameters of the state genration
   ;; function.
   (local
    (defun ValidStateParamsp (pms1 pms2)
      (declare (ignore pms2))
      (validparamsp pms1)))
   ;; In an instance, this function takes as input the coordinates of a node
   ;; and puts a message (msgid) in its buffers in case there are still free
   ;; buffers and modify the network state accodingly.
   ;; The witness is the identity function.
   (local
    (defun loadbuffers (coordinates msgid ntkstate)
      (declare (ignore coordinates msgid))
      ntkstate))
   ;; The function returns the local state of the node designated by
   ;; coordinates. In the witness, we send the first state entry.
   (local
   (defun readbuffers (coordinates ntkstate)
     (declare (ignore coordinates))
     (car ntkstate)))
   ;; The function detects a valid state entry.
   (local
   (defun ValidEntryP (entry)
    (Basic-entryp entry)))
  ;; The function MUST use the previous function to detect
  ;; the validity of a state by checking the validity of
  ;; each entry (with the help of ValidEntryP).
  (local
   (defun ValidStatep (ntkstate)
       (if (endp ntkstate)
      t
    (and (ValidEntryP (car ntkstate))
     (ValidStatep (cdr ntkstate))))))

;;----------------------end Witness Functions -----------
    (defthm Validentry-implies-Basicentry
      (implies (ValidEntryP ntkstate)
               (Basic-entryp ntkstate)))
```

```
;; Local theorem needed for the next one.
;; It states that the result of stategeneratorlocal is a valide state.
(local
 (defthm ValidStatep-stategenerator
   (ValidStatep (stategeneratorlocal listx pms2))))
;; Theorem to prove the correctness of the state generated by
;; StateGenerator.
(defthm nodeset-generates-valid-resources
  (implies (ValidStateParamsp pms1 pms2)
           (ValidStatep (StateGenerator pms1 pms2))))
;; The funciton loadbuffers returns a ValidStatep.
(defthm ValidStatep-loadbuffers-statep
  (implies (ValidStatep ntkstate)
           (ValidStatep (loadbuffers coordinates msgid ntkstate))))
;; We prove that reading a valid state for a valid node address
;; returns a valid state entry.
(defthm Readbuffers-valid-entryp
  (let ((ntkstate (StateGenerator pms1 pms2))
      (NodeSet (NodeSetGenerator pms1)))
    (implies (and (ValidStateParamsp pms1 pms2)
                  (member-equal node_id NodeSet))
             (Validentryp (Readbuffers node_id ntkstate)))))
;; This proof obligation is important to do the link between the
;; nodesetgenerator and the stategenerator.
;; It states that The Validity of the stategenerator inputs must imply the
;; validity of the input of the nodeset
(defthm Validstateparamsp-implies-validparamsp
  (implies (ValidStateparamsp  pms1 pms2)
           (Validparamsp  pms1)))
;; This is an intermediate theorem used for the next one
(local
  (defthm getcoodrdinates-stategenloc
    (implies (true-listp listx)
             (equal (getcoordinates (stategeneratorlocal listx pms2))
                    listx))))
 ;; We prove the equality between the nodeset and the coordinates in the
 ;; stategenerator.
 (defthm nodesetp-coordinates
    (implies (ValidStateparamsp pms1 pms2)
             (equal (getcoordinates (StateGenerator  pms1 pms2))
                    (nodesetgenerator pms1)))
    :hints (("Goal" :do-not '(generalize))))
```

A.5 Module du contrôle d'accès au réseau

Filename:GeNoC-departure.lisp

```
(in-package "ACL2")
(include-book "GeNoC-nodeset")
(include-book "GeNoC-misc") ;; imports also GeNoC-types
(include-book "GeNoC-ntkstate");
(include-book "needed-perm-theorems")
(defspec GenericR4d
  ;; The only function in this module takes 4 input parameters:
  ;; a list of messages that are trying to travel
  ;; an empty accumulator to accumulate the messages allowed to depart
  ;; an empty accumulator to accumulate the messages are not allowed to depart
  ;; a parameter used in the test of each message to decide its departure.
  (((readyfordeparture * * * *) => (mv * * )))
;;---------------------- Witness Functions -----------
  (local
    (defun readyfordeparture (missives delayed departing time)
     ;;(declare (ignore delayed departing dropped))
     (let ((mundertest (car missives)))
      (mv ;;delayed messages
       (if (< time (TimeTm mundertest))
           (append missives delayed)
          delayed)
       ;;departing messages
      (if (< time (TimeTm mundertest))
          departing
        (append missives departing))))))
;;-------------------end Witness Functions -----------
  ;; this theorem verifies that the first output of ready4departure
  ;; is a valid traveling missives list
  (defthm tmissivesp-ready-4-departure-mv-0
    (implies (tmissivesp m nodeset)
             (tmissivesp (mv-nth 0 (readyfordeparture m nil nil time))
                         nodeset)))
  ;; this theorem verifies that the second output of ready4departure
  ;; is a valid traveling missives list
  (defthm tmissivesp-ready-4-departure-mv-1
    (implies (tmissivesp m nodeset)
             (tmissivesp (mv-nth 1 (readyfordeparture m nil nil time))
                        nodeset)))
  ;; general theorem needed for later the proof of the next theorem
  ;; it will also be useful for the same proof in the instances.
  (defthm subsetp-append-bis
    (implies (subsetp y S)
             (subsetp y (append z S))))
  ;; The theorem proves that all the members of the first output
  ;; of the function are members of the input list m.
  (defthm subset-ready-for-departure-0
    (implies (tmissivesp m nodeset)
```

```
            (subsetp (mv-nth 0 (readyfordeparture m nil nil time))
                     m)))
  ;; The theorem proves that all the members of the second output
  ;; of the function are members of the input list m.
  (defthm subset-ready-for-departure-1
    (implies (tmissivesp m nodeset)
             (subsetp  (mv-nth 1 (readyfordeparture m nil nil time)) m)))
  ;; The theorem proves that that the two outputs of the function
  ;; ready for departure are mutually exclusive.
  (defthm not-in-1-0-ready-for-dept
   (implies (tmissivesp m nodeset)
            (not-in (tm-ids (mv-nth 1 (readyfordeparture m nil nil  time)))
                    (tm-ids (mv-nth 0 (readyfordeparture m nil nil  time))))))
;; The theorem proves that that the union of the identifiers of the
;; two output lists of the function is a permutation of the identifiers
;; of it first input.
  (defthm isperm-missives-r4d
    (let ((delayed (mv-nth 0 (readyfordeparture m nil nil  time)))
        (departing (mv-nth 1 (readyfordeparture m nil nil time))))
      (implies (tmissivesp m nodeset)
               (is-perm (append (tm-ids delayed) tm-ids departing))
                        (tm-ids m))))
    :hints (("Goal" :in-theory (disable tmissivesp)))))
```

A.6 Module du routage

```
Filename:GeNoC-routing.lisp
(in-package "ACL2")
(include-book "GeNoC-nodeset")
(include-book "GeNoC-misc") ;; import also GeNoC-types
(include-book "needed-perm-theorems")
(defspec GenericRouting
  ;; Routing computes the route of each message within the network
  ;; It takes as arguments: TM and NodeSet.
  ;; It outputs a list of travels
  ;; TrLst = (... (Id org frm Route flit time hopcount) ...).
  (((Routing * *) => *))
;;---------------------- Witness Functions -----------
  (local
   (defun RoutingCore (current destination)
     ;; Le routage dans un bus
    (list current destination)))
  (local
   (defun RoutingTop (current destination)
    (list (RoutingCore current destination))))
  (local
    (defun Routing (TM Nodeset)
```

```
    (declare (ignore Nodeset))
    (if (endp TM)
        nil
     (let* ((msv (car TM))
    (Id (IdTM msv))
    (frm (FrmTM msv))
    (origin (OrgTM msv))
    (current (CurTM msv))
    (destination (DestTM msv))
    (flit (FlitTM msv))
    (time (TimeTM msv))
    (hopcount (HopcountTM msv)))
       (cons (list Id origin frm
                   (RoutingTop current destination)
                   Flit time hopcount)
             (Routing (cdr TM) Nodeset))))))
;;-------------------end Witness Functions -----------
  ;; The first theorem proves that if the input is a list of valid traveling
  ;; missives, the output must be a list of valid travels.
  (defthm TrLstp-routing
    ;; 1st constraint
    ;; the travel list is recognized by TrLst
    ;; pms1 is a free variable
    (let ((NodeSet (NodeSetGenerator pms1)))
      (implies (and (TMissivesp M NodeSet)
                    (ValidParamsp pms1))
               (TrLstp (routing M NodeSet) NodeSet))))

  (defthm Routing-CorrectRoutesp
    ;; The routes produced by routing are correct.
    (let ((NodeSet (NodeSetGenerator pms1)))
      (implies (and (TMissivesp M NodeSet)
                    (ValidParamsp pms1))
               (CorrectRoutesp (Routing M NodeSet) M NodeSet))))

;; The identifiers of the output list is a permutation of the
;; identifiers of the input list m.
  (defthm is-perm-routing
  (implies (tmissivesp m nodeset)
           (is-perm (V-ids (routing m nodeset))
                    (tm-ids m))))

;; some additional constraints that are trivial to prove
;; but useful for the instances.
  (defthm true-listp-routing
    (true-listp (routing M NodeSet))
    :rule-classes :type-prescription)
```

```
  ;; the routing has to return nil if the list of missives is nil
  (defthm routing-nil
     (not (routing nil NodeSet)))
) ;; end of routing


;; The next theorem is not a proof obligations, it is a logical result of the
;; obligations in the defspec. It corresponds to theorem 4.1.
(defthm tomissives-routing
  ;; the result of the routing converted into Traveling misssives is equal to
  ;; the original input list of Traveling missives passed to
  ;; the function as input.
  (let ((NodeSet (NodeSetGenerator pms1)))
    (implies (and (TMissivesp M NodeSet)
                  (ValidParamsp pms1))
             (equal (ToTMissives (routing M NodeSet)) M)))
  :hints (("GOAL"
           :use (:instance correctroutesp-=>-toTmissives
                           (TrLst (Routing M (NodeSetGenerator pms1)))
                           (TM M)
                           (NodeSet (NodeSetGenerator pms1)))
           :in-theory (disable TMissivesp
                               correctroutesp-=>-toTmissives)
           :do-not-induct t)))
```

A.7 Module de l'ordonnancement

```
Filename:GeNoC-scheduling.lisp
(in-package "ACL2")
(include-book "GeNoC-nodeset")
(include-book "GeNoC-misc") ;; imports also GeNoC-types
(include-book "GeNoC-ntkstate");
(include-book "needed-perm-theorems")
;; Inputs: TrLst = ( ... (Id org frm routes flit time hopcount) ...), att,
;; NodeSet, and the current network state
;; outputs: enroute, arrived, new state of the network, att updated

(defspec GenericScheduling
  ;; Function Scheduling represents the scheduling policy of the
  ;; network.
  ;; arguments: TrLst att nodeset ntkstate
  ;; outputs:  enroute Arrived newntkstate newatt
  (((scheduling * * * * *) => (mv * * * *))
   ((get_next_priority *)=> *))
;;---------------------- Witness Functions -----------
  (local
   (defun get_next_priority (port)
```

```
     port))
  (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 NodeSet ntkstate order)
   (declare (ignore NodeSet order))
   (mv
    (if (zp (SumOfAttempts att))
        (totmissives TrLst)
         nil)
    (if (zp (SumOfAttempts att))
        nil
        TrLst)
    (if (zp (SumOfAttempts att))
        att
        (consume-attempts att))
    (if (zp (SumOfAttempts att))
         ntkstate
         ntkstate))))
;;-------------------end Witness Functions -----------
  ;; the result of the scheduling function in the case of empty input list
  ;; is equal to nil. This is not a proof obligation but it will simplify
  ;; the proofs of the instances so we write it. ACL2 will prove it always
  ;; on its own if it is true for the instance.
  (defthm scheduled-nil-nil
    (equal (car (scheduling nil att nodeset ntkstate order))
           nil))
  ;; The first output of scheduling is a Tmissivesp if the input was a valid
  ;;travel list.
  (defthm tmissivesp-enroute
    (implies (trlstp TrLst nodeset)
     (tmissivesp (mv-nth 0 (scheduling TrLst att NodeSet ntkstate order))
                 NodeSet)))
  ;; The list Arrived is a valid travel list if the input was a valid one
  (defthm trlstp-Arrived
    (implies (trlstp TrLst nodeset)
```

```
   (trlstp (mv-nth 1 (scheduling TrLst att NodeSet ntkstate order))
           nodeset)))
;; the state list newntkstate is a ValidStatep
(defthm Valid-state-ntkstate
  (implies (ValidStatep ntkstate)
   (ValidStatep (mv-nth 3 (scheduling TrLst att NodeSet ntkstate order)))))
;; the scheduling policy should consume at least one attempt
;; this is a sufficient condition to prove that
;; the full network function terminates
(defthm consume-at-least-one-attempt
  (mv-let (enroute Arrived newatt newState )
          (scheduling TrLst att NodeSet ntkstate order)
          (declare (ignore enroute Arrived newState))
    (implies (not (zp (SumOfAttempts att)))
             (< (SumOfAttempts newatt) (SumOfAttempts att)))))
;; For any arrived missive arr, there exists a unique travel
;; tr in the initial TrLst, such that IdV(arr) = IdV(tr)
;; and FrmV(arr) = FrmV(tr), RoutesV(arr) is a
;; sublist of RoutesV(tr), flitV(arr)=FlitV(tr), timeV(arr)=timeV(tr)
;; and hopcountV(arr)=hopcountV(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 (arr-TrLst TrLst/arr-ids)
  (if (endp arr-TrLst)
    (if (endp TrLst/arr-ids)
        t
        nil)
      (let* ((arr-tr (car arr-TrLst))
           (tr (car TrLst/arr-ids)))
      (and (equal (FrmV arr-tr) (FrmV tr))
           (equal (IdV arr-tr) (IdV tr))
           (equal (OrgV arr-tr) (OrgV tr))
           (equal (FlitV arr-tr) (FlitV tr))
           (equal (timeV arr-tr) (TimeV tr))
           (equal (HopcountV arr-tr) (HopcountV tr))
           (subsetp (RoutesV arr-tr) (RoutesV tr))
           (s/d-travel-correctness (cdr arr-TrLst)
                                   (cdr TrLst/arr-ids))))))
;; we prove a unitary correcteness property.
(defthm s/d-travel-correctness-unitary
  (implies (trlstp x nodeset)
   (s/d-travel-correctness x x)))
;; We prove the correctness of the arrived messages.
;; The list Arrived is equal to filtering the initial
;; TrLst according to the Ids of Arrived
(defthm arrived-travels-correctness
```

```
  (mv-let (enroute Arrived newatt newstate )
          (scheduling TrLst att NodeSet  ntkstate order)
          (declare (ignore  enroute newatt newstate ))
  (implies (TrLstp TrLst nodeset)
           (s/d-travel-correctness Arrived
                                   (extract-sublst TrLst (V-ids Arrived)))))
  :hints (("Goal" :in-theory (disable trlstp))))
;; this should be provable from the two lemmas above
;; but it will always be trivial to prove, and it is
;; useful in subsequent proofs.
(defthm subsetp-arrived-enroute-ids
  (mv-let (enroute Arrived newatt newstate )
          (scheduling TrLst att NodeSet ntkstate order)
          (declare (ignore newatt newstate ))
  (implies (TrLstp TrLst nodeset)
           (and (subsetp (v-ids Arrived) (v-ids Trlst))
                (subsetp (Tm-ids enroute) (v-ids TrLst))))))
;; The correctness of the  enroute travels differs from
;; the correctness of the Arrived travels because,
;; for the Arrived travels we will keep only
;; one route, but for the  enroute 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 enroute is equal to filtering the initial
;; TrLst according to the Ids of  enroute
;; the correctness of enroute is the equivalence of the transformation
;; of the enroute into missives, and the transformation of the initial
;; trlst (input to the scheduling function) into tmissives and then to
;; missives. This rule will cause an infinite number of rewrites that's why
;; it's in rule-classes nil, we have to create an instance to use it.
(defthm  enroute-travel-correctness
   (mv-let (enroute Arrived  newatt newstate )
           (scheduling TrLst att NodeSet  ntkstate order)
           (declare (ignore Arrived newatt newstate))
    (implies (TrLstp TrLst nodeset)
             (equal (tomissives  enroute)
                    (extract-sublst (tomissives(totmissives TrLst))
                                    (Tm-ids enroute)))))
  :rule-classes nil)
;; if every attempt has been consumed
;; the new att is equal to the initial one
(defthm mv-nth-3-scheduling-on-zero-attlst
   (implies (and (zp (SumOfAttempts att))
                 (TrLstp trlst nodeset))
            (equal (mv-nth 2 (scheduling TrLst att NodeSet ntkstate order))
                   att)))
```

```
  ;; if every attempt has been consumed
  ;; the set of  enroute s is equal to the initial TrLst
  (defthm mv-nth-0-scheduling-on-zero-attlst
     (implies (zp (SumOfAttempts att))
              (equal (mv-nth 0 (scheduling TrLst att NodeSet ntkstate order))
                     (totmissives TrLst))))
  ;; The intersection of the ids of the Arrived travels and those
  ;; of the enroute travels is empty.
  ;; -----------------------------------------------------------------
  (defthm not-in-enroute-Arrived
    (mv-let (enroute Arrived  newatt newstate )
            (scheduling TrLst att NodeSet ntkstate order)
            (declare (ignore  newatt newstate ))
    (implies (TrLstp TrLst nodeset)
             (not-in (Tm-ids  enroute) (v-ids Arrived)))))
  ;; Some Typing constraints required because we do not have a definition
  ;; for scheduling.

  (defthm consp-scheduling
    (consp (scheduling TrLst att NodeSet ntkstate order))
    :rule-classes :type-prescription)
  (defthm true-listp-car-scheduling
      (implies (true-listp TrLst)
               (true-listp (mv-nth 0 (scheduling TrLst att NodeSet ntkstate
                                                 order ))))
    :rule-classes :type-prescription)
  (defthm true-listp-mv-nth-1-sched
    (implies (TrLstp TrLst nodeset)
             (true-listp (mv-nth 1 (scheduling TrLst att NodeSet ntkstate
                                               order))))
    :rule-classes :type-prescription)
  ;; We prove now that the union of the identifiers of the two output lists
  ;; enroute and arrived is a permutation of the identifiers of the input
  ;; Trslt.
  (defthm two-results-perm-input-sched
   (let ((arrived (mv-nth 1 (scheduling TrLst att NodeSet ntkstate order)) )
         (enroute (mv-nth 0 (scheduling TrLst att NodeSet ntkstate order)) ))
     (implies (trlstp Trlst nodeset)
               (is-perm (append (v-ids arrived)(tm-ids  enroute))
                        (v-ids trlst))))))

;; the following properties are not proof obligations
;; they are properties that can be proven directly using the proof obligations
;; proved in defspec above.
;; Correctroutesp between trlst/ids and it's transformation into tmissves,
;; and the s/d-travel-correctness, between trlst/ids and trlst1
;; implies the correctroutesp between trlst1 and trlst/ids
```

```
(defthm correctroutesp-s/d-travel-correctness
  (implies (and (CorrectRoutesp TrLst/ids (ToTMissives TrLst/ids) NodeSet)
                (s/d-travel-correctness TrLst1 TrLst/ids))
           (CorrectRoutesp TrLst1 (ToTMissives TrLst/ids) NodeSet)))
;; We prove that sheduling preserve the correctness of the routes after the
;; transformation.
(defthm scheduling-preserves-route-correctness
  (mv-let (enroute Arrived newatt newstate )
          (scheduling TrLst att NodeSet ntkstate order)
          (declare (ignore  enroute newstate newatt ))
    (implies (and (CorrectRoutesp TrLst (ToTMissives TrLst) NodeSet)
                  (TrLstp TrLst nodeset))
             (CorrectRoutesp Arrived
                             (ToTMissives (extract-sublst TrLst
                                                          (V-ids Arrived)))
                              NodeSet)))
    :otf-flg t
    :hints (("GOAL"
             :do-not '(eliminate-destructors generalize)
             :do-not-induct t
             :in-theory (disable mv-nth ToTMissives-extract-sublst TrLstp))))
```

A.8 Module des priorités

```
Filename:GeNoC-priority.lisp
(in-package "ACL2")
(include-book "own-perm")
(include-book "GeNoC-types")
(include-book "GeNoC-misc")

(defspec GenericPriority
  ;; The only function prioritysorting takes two inputs
  ;; a list of travels to sort with respect to the second input order.
  ;; The output is a list of travels.
  (((prioritysorting * *)=> * ))
  ;;---------------------- Witness Functions -----------
  (local
   (defun prioritysorting (trlst order)
     (declare (ignore order))
     trlst))
 ;;-------------------end Witness Functions -----------
  ;; the output of the function is just a permutation of the input
  (defthm isperm-prioritysorting
    (implies (trlstp trlst nodeset)
             (is-perm (v-ids (prioritysorting trlst order))
                      (v-ids trlst))))
  ;;The output is a valid list of travels.
```

```
  (defthm trlstp-prioritysorting
    (implies (trlstp trlst nodeset)
             (trlstp (prioritysorting trlst order) nodeset)))
  ;; The following theorems are not proof obligations, but their proofs will
  ;; help the user during the instanciation phase, so we put them here
  ;; to force the user to define them.
  ;; The output is a subsetp of the input (to be sure no travels are created).
  (defthm subsetp-prioritysorting-trlst
    (implies (trlstp trlst nodeset)
             (subsetp (prioritysorting trlst order)
                      trlst)))
  ;; the identifiers of the output is a subset of those of the input
  ;; remove and put a general relation between is-perm and subsetp
  (defthm subsetp-v-ids-priority-trlst
    (implies (trlstp trlst nodeset)
             (subsetp (v-ids (prioritysorting trlst order))
                      (v-ids trlst))))
  ;; the result of the function is a true-listp.
  (defthm true-listp-priority-sorting
    (implies (trlstp trlst nodeset)
             (true-listp (prioritysorting trlst order))))
  ;; The origins of the output are a subset of those of the input.
  (defthm subsetp-orgs-prioritysort
    (implies (trlstp trlst nodeset)
             (subsetp (v-orgs (prioritysorting trlst order))
                      (v-orgs trlst))))
  ;; The frames of the output are a subset of those of the input.
  (defthm subsetp-frms-prioritysort
    (implies (trlstp trlst nodeset)
             (subsetp (v-frms (prioritysorting trlst order))
                      (v-frms trlst))))
  ;; The destinations of the output after transformation to missives are a
  ;; subset of those of the input after the same transformation.
  (defthm subsetp-prioritysort_mdests
    (implies (trlstp trlst nodeset)
            (subsetp (m-dests
                       (tomissives
                        (totmissives (prioritysorting trlst order))))
                     (m-dests (tomissives(totmissives trlst)))))))
```

A.9 Module de la synchronisation

```
Filename:GeNoC-synchronisation.lisp
(in-package "ACL2")
(include-book "GeNoC-misc")

(defspec genericsynchronisation
```

```
(;;req_tans is the equivalent of requesting a transmission in a
 ;;synchronisation protocol.
 ((req_trans *) => *)
 ;; The next function is the function that checks if we can put the
 ;; acknowledge to 1 or not.
 ((process_req * * ) => *)
 ;; This function checks if it's possible to do the transmission and if so,
 ;; it puts the acknowledge to one.
 ((chk_avail * * * *) => *)
 ;; The next function uses chk_avail to see if it's possible to do a
 ;; transmission it send back the st updated with the activation of the ack
 ;; signal or leaves it as it is. The check to decide wether the
 ;; transmission will be done or not will be done in the scheduling function
 ;; by looking to the ack of the destination
 ;; if it is equal to one it means that the transmission can be done
 ;; otherwise we leave it intact.
 ((good_route? * * * * ) => (mv * *))
 ;; A cover fucntion used from the scheduling instance that calls the
 ;; previous function.
 ((test_routes * *) => (mv * *)))
 ;;---------------------- Witness Functions -----------
(local
 (defun req_trans (st)
   ;;local witness
   st))
 (local
  (defun process_req (st dest)
   (declare (ignore st dest))
    t))
 (local
   (defun chk_avail (st org dest route)
     ;; This local witness is complicated for the simple reason that, one of
     ;; the proof obligations needed to prove necessitate the definition of
     ;; such witness.
     ;; the function checks three major conditions needed to verify that the
     ;; origin of a message is different from its destination, last element
     ;; of a route is equal to a route
     ;; finally if the message is not arrived to its destination
     ;; (route length equal 2), the next hop is not equal to the destination
     ;; in instants of this function the user should provide the necessary
     ;; extra conditions needed to schedule a travel
   (and (process_req st dest)
        (if (equal (len route) 2)
            t
            (not (equal (cadr route) (car (last route)))))
        (not (equal org (car (last route))))
        (equal (car (last (cdr route))) dest))))
```

```
(local
  (defun good_route? (st org dest routes)
      (if (endp routes)
          (mv st nil)
          (let ((route (car routes)))
            (if (chk_avail st org dest route)
                (mv st (car routes))
                  (good_route? st org dest (cdr routes)))))))
(local
 (defun test_routes (st tr)
   (let* ((routes  (routesv tr))
          (dest (car (last (car routes))))
          (org (orgv tr)))
     (mv-let (newst r?)
         (good_route? st org dest routes)
       (mv newst r?) ))))
;;-------------------end Witness Functions -----------
;; We prove that the result of req_trans is a valid state if its input was a
;; valid one.
(defthm state-req-trans
  (implies (ValidStatep ntkstate)
           (ValidStatep (req_trans ntkstate))))
;; We prove that the result of req_trans is a valid state if its input was a
;; valid one.
(defthm chk_avail_obligation-for-scheduling
  (implies (chk_avail st org dest route)
           (and (if (equal (len route) 2)
                    t
                  (not (equal (cadr route) (car (last route)))))
                (not (equal org (car (last route))))
                (equal (car (last (cdr route))) dest)))
  :rule-classes :forward-chaining)
(defthm validdtate-good_route?
  (implies (ValidStatep ntkstate)
           (ValidStatep (mv-nth 0 (good_route? ntkstate org dest routes)))))
(defthm validdtate-test_routes
  (implies (ValidStatep ntkstate)
           (ValidStatep (mv-nth 0(test_routes ntkstate tr))))
:hints (("Goal" :in-theory (disable mv-nth))) )
```

A.10 Module GeNoC

Nous donnons ici une version simplifiée du code source du module $GeNoC$: la définition de la fonction $GeNoC_t$, son théorème de correction, la fonction $GeNoC$, et ses propriétés de correction. Pour des raisons de clarté, nous omettons 72 lemmes intermédiaires ainsi que les directives données à ACL2 pour la preuve des théorèmes présentés ci-dessous.

`Filename:GeNoC.lisp`

```
(include-book "GeNoC-scheduling")
(include-book "GeNoC-routing")
(include-book "GeNoC-departure")
;;-----------------------------------------------------------------------
;;                          genoc_t
;;-----------------------------------------------------------------------
;; tail definition of genoc_t
;; --------------------------
(defun genoc_t (m nodeset att trlst time ntkstate order accup)
  ;; the composition of routing and scheduling is built by function genoc_t.
  ;; it takes as arguments:
  ;; - a list of tmissives, m
  ;; - the set of existing nodes, nodeset
  ;; - the list of attempts, att
  ;; - an accumulator of travels, trlst
  ;; - an accumulator for lost messages
  ;; - an accumulator for network state through the time steps
  ;; - the state of the network
  ;; - an order to specify the priority schema
  ;; 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 accup );;dropped)
    ;; else,
      (mv-let (delayed departing)
         (readyfordeparture m nil nil time)
     (let ((v (routing departing nodeset)))
       ;; we compute the routes. this produces the travel list v.
       (mv-let (enroute arrived newatt newntkstate)
       ;; we call function scheduling
       (scheduling v att nodeset ntkstate order)
       ;; we enter the recursive call and accumulate the
       ;; scheduled travels
       (genoc_t  (append enroute delayed) nodeset newatt
                 (append arrived trlst) (+ 1 time)
                newntkstate (get_next_priority order)
                 (append accup (list newntkstate)))))))
;; 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) = frmtm(m) and (b) forall r in
  ;; routesv(ctr) last(r) = desttm(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))))))
;; before checking correctness we filter m
;; according to the ids of trlst
(defun genoc_t-correctness (trlst m)
   (let ((m/trlst  (extract-sublst (tomissives m) (v-ids trlst))))
    (genoc_t-correctness1 trlst m/trlst)))
;; now we prove the correctness of genoc_t
(defthm genoc_t-thm
  (let ((nodeset (nodesetgenerator pms1)))
     (implies (and (tmissivesp m nodeset)
                   (validparamsp pms1))
              (mv-let (cplt abt simu)
                      (declare (ignore abt simu))
                     (genoc_t m nodeset att nil time ntkstate order accup)
                (genoc_t-correctness cplt m)))))
;;-------------------------------------------------------------
;;      definition and validation of genoc
;;-------------------------------------------------------------
;;ComputetTMissives
;;--------------
(defun computeTMissives (transactions)
 ;; The function applies the function send to build a list of tmissives
 ;; from a list of transactions.
 (if (endp transactions)
      nil
```

```
      (let* ((trans (car transactions))
             (id (idt trans))
             (org (orgt trans))
             (dest (destt trans))
             (flit (flitt trans))
             (time (timet trans))
             (hopcount (hopcountV trans)))
        (cons (list id org org (send msg) dest flit time '0)
              (computetmissives (cdr transactions))))))
;; ComputeResults
;; -------------
(defun computeresults (trlst)
 ;; The function applies the function recv to build a list of results
 ;; from a list of travels.
 (if (endp trlst)
     nil
     (let* ((tr (car trlst))
            (r (car (routesv tr)))
            (dest (car (last r)))
            (frm (frmv tr))
            (flit (flitv tr)))
      (cons (list id dest (recv frm) flit)
            (computeresults (cdr trlst))))))
(defun genoc (trs nodeset att pms1 pms2 order)
  ;; main function
  (mv-let (responses aborted simu)
          (genoc_t (computetmissives trs) nodeset att nil '0
                   (stategenerator pms1 pms2) order
                   (list (stategenerator pms1 pms2)))
  (mv (computeresults responses) aborted simu)))
;; The next function will be used to check the correctness of the
;; function.
(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))))
;; We prove the correctness of GeNoC using the function genoc-correctness.
;; We extract the transactions that bear the same identifier as the result.
;; We then pass the results obtained at the ouptut of GeNoC and the extracted
;; transactions to the function.
(defthm genoc-is-correct
  (let ((nodeset (nodesetgenerator pms1)))
```

```
    (mv-let (results aborted simu)
            (genoc trs nodeset att pms1 pms2 order)
            (declare (ignore aborted simu))
      (implies (and (transactionsp trs nodeset)
                    (validstateparamsp pms1 pms2))
               (genoc-correctness
                      results
                      (extract-sublst trs (r-ids results)))))))
;; New property: the identifiers of all the messages en route and arrived
;; combined are equal to those of the input set of messages
(defthm genoc-is-correct-2
  (let ((nodeset (nodesetgenerator pms1)))
    (mv-let (results aborted simu)
    (declare (ignore simu))
      (genoc trs nodeset att pms1 pms2 order)
    (implies (and (transactionsp trs nodeset)
                  (validstateparamsp pms1 pms2))
             (is-perm (t-ids Trs)
                      (append (tm-ids aborted) (r-ids results))))))
```

Bibliographie

[ABC+05] A.M. Amory, E. Briao, E. Cota, M. Lubaszewski, and F.G. Moraes. A scalable test strategy for network-on-chip routers. In *Proceedings of the International Test Conference, ITC'05*, pages 591–599, November 2005.

[AG03] A. Andriahantenaina and A. Greiner. Micro-Network for SoC: Implementation of a 32-Port SPIN network. *Design, Automation and Test in Europe Conference and Exhibition*, 1:11128, 2003.

[Amj04] H. Amjad. Model Checking the AMBA Protocol in HOL. Technical report, University of Cambridge, Computer Laboratory, September 2004.

[And03] A. Andriahantenaina. SPIN: a scalable, packet switched, on-chip micro-network. In *Proceedings of Conference on Design, Automation and Test in Europe, DATE'03*, pages 70–73, 2003.

[ARM99] ARM. AMBA specification v2.0. Technical report, 1999.

[ARM01] ARM. AMBA Multilayer AHB overview. Technical report, 2001.

[ARM03] ARM. AMBA AXI Protocol Specification. Technical report, 2003.

[BB04] D. Bertozzi and L. Benini. Xpipes: A network-on-chip architecture for gigascale systems-on-chip. *IEEE Circuits and Systems Magazine*, 4:18–31, 2004.

[BB05] L. Benini and D. Bertozzi. Network-on-chip architectures and design methods. In *IEE Proceedings on Computers and Digital Techniques*, volume 152, pages 261–272, March 2005.

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[BCZ06] S. Bourduas, J.-S. Chenard, and Z. Zilic. A RTL-Level Analysis of a Hierarchical Ring Interconnect for Network-on-Chip Multi-Processors. In *Proceedings of International System-on-a-Chip Design Conference, ISOCC'06*, October 2006.

[BDM02] L Benini and G. De Micheli. Networks on chip: a new paradigm for systems on chip design. In *Proceedings of Conference on Design, Automation and Test in Europe, DATE'02*, pages 418–419, 2002.

[BFS95] R. Bharadwaj, A. Felty, and F. Stomp. Formalizing Inductive Proofs of Network Algorithms. In *Proceedings of 1995 Asian Computing Science Conference*, 1995.

[BHPS09] Dominique Borrione, Amr Helmy, Laurence Pierre, and Julien Schmaltz. A Formal Approach to the Verification of Networks on Chip. *EURASIP Journal on Embedded Systems*, 2009.

[BKM96] B. Brock, M. Kaufmann, and J.S. Moore. ACL2 Theorems about Commercial Microprocessors. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design, FMCAD'96*, volume 1166 of *Lecture Notes In Computer Science*, pages 275–293, London, UK, 1996. Springer-Verlag.

[BM06] T. Bjerregaard and S. Mahadevan. A Survey of Research and Practices of Network-on-Chip. *ACM Computer Surveys*, 38(1), 2006.

[BMM07] A. Banerjee, R. Mullins, and S. Moore. A power and energy exploration of network-on-chip architectures. In *Proceedings of the ACM/IEEE Int. Symp. on Networks-on-Chip, NOCS'07*, pages 163–172, Washington, DC, USA, 2007. IEEE Computer Society.

[BY96] R.S. Boyer and Y. Yuan. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43:166–192, 1996.

[BZ08] M. Boulé and Z. Zilic. Automata-based assertion-checker synthesis of PSL properties. *ACM Trans. Des. Autom. Electron. Syst.*, 13(1):1–21, 2008.

[CBA+07] J.S. Chenard, S. Bourduas, N. Azuelos, M. Boulé, and Z. Zilic. Hardware Assertion Checkers in On-line Detection of Network-on-Chip Faults. In *Proceedings of the Workshop on Diagnostic Services in Networks-on-Chips*, 2007.

[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.

[CGJ97] E. M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(5):726 – 750, September 1997.

[CLM+04] M. Coppola, R. Locatelli, G. Maruccia, L. Pieralisi, and A. Scandurra. Spidergon: a novel on-chip communication network. In *Proceedings of the International Symposium on System-on-Chip*, November 2004.

[Cur94] P. Curzon. Experiences formally verifying a network component. In *Proceedings of IEEE Conference on Computer Assurance*. IEEE Press, 1994.

[CVL05] F. Clermidy, D. Varreau, and D. Lattard. A NoC-based communication framework for seamless IP integration in complex systems. In *Proceedings Design & Reuse IP-SoC '05*, Grenoble, France, December 2005.

[Dal92] W. J. Dally. Virtual-channel flow control. *IEEE Trans. Parallel Distrib. Syst.*, 3(2):194–205, 1992.

[DBL05] Y. Durand, C. Bernard, and D. Lattard. FAUST: On-Chip Distributed Architecture for a 4G Baseband Modem SoC. In *Design & Reuse IP-SoC'05*, Grenoble, France, December 2005.

[DL99] W.J. Dally and S. Lacy. VLSI architecture: past, present, and future. In *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, pages 232–241, March 1999.

[DS86] W.J. Dally and C.L. Seitz. The torus routing chip. *Distributed Computing*, 1(4):187–196, December 1986.

[DT03] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[Dua03] M.E. Duarte. Networks on Chip (NOC): Design Challenges. In *International Conference on Computer Architecture, ICCA'03*, pages 121–128, 2003.

[GHF+06] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. volume 26, pages 10–24, Los Alamitos, CA, USA, 2006. IEEE Computer Society Press.

[GIS+06] C. Grecu, A. Ivanov, R. Saleh, E.r Sogomonyan, and P. Pande. On-line Fault Detection and Location for NoC Interconnects. In *Proceedings of the International On-Line Testing Symposium, IOLTS'06*, July 2006.

[GM93] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[GN92] C. J. Glass and L. M. Ni. The Turn Model for Adaptive Routing. In David Abramson and Jean-Luc Gaudiot, editors, *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 278–287, Gold Coast, Australia, May 1992. ACM Press.

[Goo05] K. Goossens. Formal Methods for Networks on Chips. In *Proceedings of the Fifth International Conference on Application of Concurrency to System Design, ACSD'05*, pages 188–189. IEEE Computer Society, 2005.

[GVVSB07] K. Goossens, B. Vermeulen, R. Van Steeden, and M. Bennebroek. Transaction-Based Communication-Centric Debug. In *Proceedings of the International Symposium on Networks-on-Chip, NOC'07*, 2007.

[GVZ+05] B. Gebremichael, F.W. Vaandrager, M. Zhang, K. Goossens, E. Rijpkema, and A. Radulescu. Deadlock Prevention in the Æthereal Protocol. In *Proceedings of CHARME'05*, October 2005.

[HB05] D. Herzberg and D. Broy. Modeling Layered Distributed Communication Systems. *Formal Aspects of Computing*, 17(1):1–18, 2005.

[HR99] H. Hellwagner and A. Reinefeld, editors. *SCI: Scalable Coherent Interface, Architecture and Software for High-Performance Computer Clusters*, London, UK, 1999. Springer-Verlag.

[HS95] T.-L. Hu and F. N. Sibai. Performance analysis and optimal system configuration of hierarchical two-level COMA multiprocessors. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation, Frontiers'95*, page 90, Washington, DC, USA, 1995. IEEE Computer Society.

[IEE92] *IEEE Standard for Scalable Coherent Interface (SCI) IEEE Std 1596*. 1992.

[IEE05] *IEEE Standard for Property Specification Language (PSL) IEEE STD 1850*. 2005.

[Jan06] Axel Jantsch. Models of Computation for Networks on Chip. In *Proceedings of the Sixth International Conference on Application of Concurrency to System Design, ACSD '06*, pages 165–178, Washington, DC, USA, 2006. IEEE Computer Society.

[KJM+02] S Kumar, A. Jantsch, M. Millberg, J. Öberg, J.P. Soininen, M. Forsell, K. Tiensyrjä, and A. Hemani. A network on chip architecture and design methodology. *VLSI, IEEE Computer Society Annual Symposium on*, 0:0117, 2002.

[KM01] M. Kaufmann and J. M. Moore. Structured Theory Development for a Mechanized Logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.

[KMM00] M. Kaufmann, P. Manolios, and J.S. Moore. *Computer Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, 2000.

[KMM02] M. Kaufmann, P. Manolios, and J.S Moore. *Computer Aided Reasoning: an Approach*. Kluwer Academic Press, 2002.

[KND02] F. Karim, A. Nguyen, and S. Dey. An Interconnect Architecture for Networking Systems on Chips. *IEEE Micro*, (5):36–45, September 2002.

[KPP06] M. Kistler, M. Perrone, and F. Petrini. Cell Multiprocessor Communication Network: Built for Speed. *IEEE Micro*, 26(3):10–23, May-June 2006.

[KSJY+08] K. Kwanho, L. Seungjin, K. Joo-Young, K. Minsu, K. Donghyun, W. Jeong-Ho, , and Y. Hoi-Jun. A 125GOPS 583mW Network-on-chip Based Parallel Processor with Bio-inspired Visual Attention Engine. In *Proceedings of the IEEE International Solid-State Circuits Conference, ISSCC'08*, volume 44, pages 136–147, February 2008.

[Law75] D. H. Lawrie. Access and Alignment of Data in an Array Processor. *IEEE Trans. Comput.*, 24(12):1145–1155, 1975.

[Lim90] Abstract Hardware Limited. LAMBDA - Logic and Mathematics behind Design Automation. Technical report, 1990.

[LRD01] K. Lahiri, A. Raghunathan, and S. Dey. Evaluation of the traffic-performance characteristics of system-on-chip communication architectures. In *Proceedings of the The 14th International Conference on VLSI Design, VLSID '01*, pages 29–35, Washington, DC, USA, 2001. IEEE Computer Society.

[MADHRW00] F. Meyer Auf Der Heide, H. Räcke, and M. Westermann. Data management in hierarchical bus networks. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 109–118, New York, NY, USA, 2000. ACM.

[Mah94] S. M. Mahmud. Performance Analysis of Multilevel Bus Networks for Hierarchical Multiprocessors. *IEEE Transactions on Computers*, 43(7):789–805, 1994.

[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, The VLSI Journal*, 38(1):69–93, 2004.

[MDMB+05] S. Murali, G. De Micheli, L. Benini, T. Theocharides, N. Vijaykrishnan, and M.J. Irwin. Analysis of Error Recovery Schemes for Networks on Chips. *Design & Test of Computers*, 22(5), 2005.

[MGPM04] P. Miner, A. Geser, L. Pike, and J. Maddalon. A Unified Fault-Tolerance Protocol. In Y. Lakhnech and S. Yovine, editors, *Proceedings Formal Techniques, Modeling and Analysis of Timed and Fault-Tolerant Systems (FORMATS-FTRTFT'04)*, volume 3253 of *LNCS*, pages 167–182. Springer, 2004.

[MLK98] J S. Moore, T. Lynch, and M. Kaufmann. A Mechanically Checked Proof of the Correctness of the Kernel of the AMD5K86 Floating-Point Division Algorithm. *IEEE Trans. on Computers*, 47(9), 1998.

[MMA+06] S. Murali, P. Meloni, F. Angiolini, D. Atienza, S. Carta, L. Benini, L. Raffo, and G. De Micheli. Designing Message-Dependent Deadlock Free Networks on Chips for Application-Specific Systems-on-Chips. In *4th Annual IFIP International Conference on Very Large Scale Integration (VLSI-SoC)*, 2006.

[MNTJ04] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *Proceedings of Design, Automation and Test in Europe, DATE'04*, 2004.

[Moo93] J S. 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.S. Moore. Symbolic Simulation: An ACL2 Approach. In *Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design, FMCAD '98*, November 1998.

[MPCJ08] E.I. Moreno, K.M. Popovici, N.L.V. Calazans, and A.A. Jerraya. Integrating Abstract NoC Models within MPSoC Design. In *Proceedings of The 19th IEEE/IFIP International Symposium on Rapid System Prototyping, 2008, RSP'08*, pages 65–71, June 2008.

[MPCVG08] I. Miro-Panades, F. Clermidy, P. Vivet, and A. Greiner. Physical Implementation of the DSPIN Network-on-Chip in the FAUST Architecture. In *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip, NoCS 2008.*, volume 7, pages 139 – 148, April 2008.

[MPGS06] I. Miro Panades, A. Greiner, and A. Sheibanyrad. A Low Cost Network-on-Chip with Guaranteed Service Well Suited to the GALS Approach. In *Proceedings of 1st International Conference on Nano-Networks and Workshops, NanoNet '06.*, pages 1–5, September 2006.

[Nil03] Erland Nilsson. Experiments of the proximity congestion awareness with the nostrum backbone. In *Proceedings of teh Swedish System-on-Chip Conference, SSoCC'03.*, 2003.

[NJ95] T. Nesson and S.L. Johnsson. ROMM routing on mesh and torus networks. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures, SPAA '95*, pages 275–287, New York, NY, USA, 1995. ACM.

[Nob02] J.P. Noblanc. EDA and Systems-on-Chip: a Key challenge for MEDEA+. In *MEDEA+ design Automation Conference*, Stressa, Italy, October 2002.

[NS01] S. F. Nielsen and J. Sparsø. Analysis of low-power SoC interconnection networks. In *IEEE 19th Norchip Conference*, pages 77–86, nov 2001.

[OBMAP09] F Ouchet, D. Borrione, K. Morin-Allory, and L. Pierre. High-level symbolic simulation for automatic model extraction. In *Proceedings of the IEEE Symposium on Design and Diagnostics of Electronic Systems*, Liberec (Czech Republic), April 2009.

[OHM05] U. Ogras, J. Hu, and R. Marculescu. Key Research Problems in NoC Design: A Holistic Perspective. In *Proceedings of the international conference on HArdware/Software Codesign and System synthesis, CODES+ISSS'2005*, pages 69–74, September 2005.

[OMAB08] Y. Oddos, K. Morin-Allory, and D. Borrione. Assertion-Based Design with Horus. In *International Conference on Formal Methods and Models for Codesign (MEMOCODE'08)*, pages 75–76. IEEE Computer Society, June 2008.

[OSR93] S. Owre, N. Shankar, and J. Rushby. User Guide for the PVS Specification and Verification System, Language, and Proof Checker. Technical report, Computer Science Laboratory, SRI International, Melno Park, California, february 1993.

[PDMG+05] P. Pande, G. De Micheli, C. Grecu, A. Ivanov, and R. Saleh. Design, Synthesis, and Test of Networks on Chips. *IEEE Design & Test of Computers*, 22(5):404–413, 2005.

[PFT+07] L.A. Plana, S. B. Furber, S. Temple, M. Khan, Y. Shi, J. Wu, and S. Yang. A GALS Infrastructure for a Massively Parallel Multiprocessor. *IEEE Design and Test of Computers*, 24(5):454–463, 2007.

[Pik07] L. Pike. Modeling Time-Triggered Protocols and Verifying Their Real-Time Schedules. In *Proceedings of Formal Methods in Computer Aided Design (FMCAD'07)*. IEEE, 2007.

[PO07] K. Petersén and J. Öberg. Toward a scalable test methodology for 2D-mesh Network-on-Chips. In *Proceedings of the conference on Design, automation and test in Europe, DATE '07*, pages 367–372, San Jose, CA, USA, 2007. EDA Consortium.

[Puj04] G. Pujolle. *Les réseaux*. Eyrolles, 5th edition, 2004.

[QS82] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. *International Symposium on Programming*, pages 337–351, 1982.

[RAP+05] J.P. Ramas, D. Atienza, M. Peon, I. Magan, J. Mendias, and R. Hermida. Versatile FPGA-Based Functional Validation Framework for Networks-on-Chip Interconnections Designs. In *Proceedings of Parallel Computing (mini-symposium NoC), ParCo'2005*, 2005.

[RGR+03] E. Rijpkema, K. G. W. Goossens, A. Radulescu, 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 *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10350, Washington, DC, USA, 2003. IEEE Computer Society.

[RSV97] J.A. Rowson and A. Sangiovanni-Vincentelli. Interface-Based Design. In *Proceedings of the 34th Conference on Design Automation Conference (DAC'96)*, pages 178–183, 1997.

[Rus99] John Rushby. Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, sep 1999.

[Sch06] J. Schmaltz. *Une formalisation fonctionnelle des communications sur la puce*. PhD thesis, Joseph Fourier University, Grenoble, France, January 2006. (In French).

[SHZG05] M. Schäfer, T. Hollstein, H. Zimmer, and M. Glesner. Deadlock-free routing and Component placement for irregular mesh-based networks-on-chip. In *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design, ICCAD '05*, pages 238–245, Washington, DC, USA, 2005. IEEE Computer Society.

[SJKSJHJ05] L. Se-Joong, L. Kangmin, S. Seong-Jun, and Y. Hoi-Jun. Packet-switched on-chip interconnection network for system-on-chip applications. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 52(6):308– 312, June 2005.

[SSM+01] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli. Addressing the System-on-a-Chip Interconnect Woes Through Communication-Based Design. In *Proceedings of the 38th conference on Design Automation, DAC'01*, pages 667–672, New York, NY, USA, 2001. ACM.

[Tan95] A.S. Tanenbaum. *Computer Networks*. Eastern Economy Edition. Prentice-Hall(India) Pvt Ltd, New Delhi, 2 edition, 1995.

[VS09] F. Verbeek and J. Schmaltz. Formal Validation of Deadlock Prevention in Networks-on-Chip. In *Proceedings of the 8th International Workshop on the ACL2 Theorem Prover and its Applications (ACL2'09)*, Boston, MA, USA, May 2009. ACM.

[VS10] F. Verbeek and J. Schmaltz. Formal specification of networks-on-chips: Deadlock and evacuation. In *Proceedings of the conference on Design, automation and test in Europe, DATE'10*, 2010.

[WD00] W. Wolf-Dietrich. Enabling Reuse via an IP Core-centric Communications Protocol: Open Core Protocol. In *Proceedings of IP use in SOC design, IPSoC'00*, 2000.

[Yos05] J. Yoshida. Texas Instruments launches DaVinci platform. *EEtimes*, page online, 2005.

Abstract: The current technology allows the integration on a single die of complex systems-on-chip (SoC's) composed of manufactured blocks (IP's) that can be interconnected through specialized networks-on-chip (NoCs). IP's have usually been validated by diverse techniques (simulation, test, formal verification) and the key problem remains the validation of the communication infrastructure. This thesis addresses the formal verification of NoCs by means of a mechanized proof tool, the ACL2 theorem prover. A meta-model for NoCs has been developed and implemented in ACL2. It satisfies generic correctness statements, which are logical consequences of a set of proof obligations for each one of the NoC constituents (topology, routing, switching technique,…). Thus the verification of a particular NoC instance is reduced to discharging this set of proof obligations. The purpose of this thesis is to extend this meta-model in several directions: more accurate timing modeling, flow control, priority mechanisms,… The methodology is demonstrated on realistic and state-of-the-art NoC designs: Spidergon (STMicroelectronics), Hermes (The Federal University of Rio Grande do Sul, Brazil, and LIRMM), and Nostrum (Royal Institute of Technology, Sweden).

Keywords: formal verification, ACL2, theorem proving, NoCs, SoCs, network on chip, system on chip,automated reasoning, network verification.

Mise en œuvre de techniques de démonstration automatique pour la vérification formelle des NoCs

Résumé: Les technologies actuelles permettent l'intégration sur une même puce de systèmes complexes (SoCs) qui sont composés de blocs préconçus (IPs) pouvant être interconnectés grâce à un réseau sur la puce (NoCs). De manière générale, les IPs sont validés par diverses techniques (simulation, test, vérification formelle) et le problème majeur reste la validation des infrastructures des communications. Cette thèse se concentre sur la vérification formelle des réseaux sur puce à l'aide d'un outil de preuve automatique, le démonstrateur de théorèmes ACL2. Un méta-modèle pour les réseaux sur puce a été développé et implémenté dans ACL2. Il satisfait des propriétés de correction générique, conséquences logiques d'un ensemble d'obligations de preuve sur les constituants principaux du réseau (topologie, routage, technique de commutation,…). La preuve de correction pour une instance spécifique de réseau sur puce est alors réduite à la vérification de ces obligations de preuve. Cette thèse poursuit les travaux entrepris dans ce domaine en étendant ce méta-modèle dans plusieurs directions : prise en compte plus fine de la modélisation temporelle, du contrôle de flux, des mécanismes de priorités,… Les résultats sont démontrés sur plusieurs réseaux actuels : Spidergon (STMicroelectronics), Hermes (Université fédérale du Rio Grande do Sul, Brésil et LIRMM) et Nostrum (Royal Institute of Technology, Suéde).

Mots clés : vérification formelle, ACL2, démonstration de théorèmes, NoCs, SoCs, réseaux sur puce, système sur puces, raisonnement automatique, vérification des réseaux.

Laboratoire TIMA, 46 avenue Félix Viallet, 38031, Grenoble, France

ISBN : 978-2-84813-152-8