Modélisation, Validation et Présynthèse de Circuits Asynchrones en SystemC

C. Koch-Hofer

Institut Polytechnique de Grenoble

N° attribuée par la bibliothèque
978-2-84813-131-3

THÈSE

pour obtenir le grade de
DOCTEUR DE L'INP 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
Cédric Koch-Hofer
le 26 mars 2009

Titre :

Modélisation, Validation et Présynthèse de Circuits Asynchrones en SystemC

Directeur de thèse : M. Marc Renaudin
Co-directeur de thèse : Mme. Dominique Borrione

JURY

M. Frédéric Pétrot	Président
Mme. Emmanuelle Encrenaz-Tiphène	Rapporteur
M. Bruno Rouzeyre	Rapporteur
M. Pascal Vivet	Examinateur
M. Bernard Tourancheau	Examinateur
Mme. Dominique Borrione	Co-directeur

Remerciements

Au terme de cette très longue et très enrichissante aventure, je tiens, en premier lieu, à remercier mon directeur de thèse M. Marc Renaudin qui m'a permis de réaliser cette thèse et qui m'a fait profiter de ses grandes compétences.

J'adresse aussi tous mes remerciements à ma co-directrice de thèse Mme. Dominique Borrione sans qui ce manuscrit n'aurait probablement jamais existé.

Je remercie aussi mes deux rapporteurs de thèse Mme. Emmanuelle Encrenaz-Tiphène et M. Bruno Rouzeyre pour avoir eu le courage de relire cette thèse.

Je remercie également M. Pascal Vivet et M. Bernard Tourancheau pour avoir accepté de faire partie de mon jury en tant qu'examinateurs et M. Frédéric Pétrot pour avoir présidé ce jury.

Pour finir et pour faire court, je remercie toutes les personnes du laboratoire TIMA que j'ai côtoyées au cours de ces quatre années et demie de thèse.

Finalement, je tiens à remercier mes deux correcteurs orthographiques humains préférés.

Table des matières

Introduction

Avec l'avancée des progrès technologiques dans la conception des circuits numériques, les méthodes de conception traditionnelles « tout synchrone » atteignent leurs limites. Le signal d'horloge devant parcourir des distances de plus en plus grandes, il devient très difficile de respecter les contraintes temporelles qui lui sont associées : les temps de propagation du signal d'horloge d'un circuit synchrone vers ses différents blocs logiques doivent être uniformes. Une solution de plus en plus populaire pour résoudre ce problème est de diviser un circuit en plusieurs domaines d'horloge indépendants. Toutefois, cette nouvelle méthode de conception entraîne une complexification significative du réseau d'interconnexion employé pour faire communiquer les différents composants logiques d'un tel circuit. En plus de son rôle de medium de communication, le réseau d'interconnexion d'un circuit, possédant plusieurs domaines d'horloges, doit assurer la synchronisation des communications entre les blocs logiques appartenant à des domaines d'horloge différents.

Pour connecter efficacement les différents composants d'un circuit composé de plusieurs domaines d'horloge, un nouveau type de réseau d'interconnexion, appelé réseau sur puce ou NoC (*Network on Chip*), a vu le jour au cours de ces dernières années. Les contraintes spécifiques d'un NoC sont particulièrement bien adaptées à l'emploi de la logique asynchrone car elle permet de s'affranchir du problème de la distribution du signal d'horloge et de synchroniser correctement des communications asynchrones. Cependant, l'utilisation des NoC asynchrones est actuellement limitée par le manque d'outils de conception dédiés aux circuits asynchrones.

Le principal objectif de cette thèse est de contribuer à l'adoption des circuits asynchrones en développant des méthodes et des outils de conception qui puissent être facilement intégrés dans les flots de conception des industriels. A cette fin, une bibliothèque de modélisation ASC (*Asynchronous SystemC*) dédiée au circuit asynchrone a été développée. Avec cette bibliothèque, un circuit asynchrone est modélisé par un ensemble de processus qui communiquent via des canaux de communication point à point unidirectionnels et bloquants. En outre, ASC introduit de nouvelles facilités permettant de modéliser fidèlement des arbitres asynchrones qui sont des composants essentiels des réseaux sur puce asynchrones. La sémantique du langage de modélisation dérivé de ASC est, en fait, très proche de celle du langage CHP (*Concurrent Hardware Processes*) employé par l'outil de conception TAST (*TIMA Asynchronous Synthesis Tools*), car le but final de ce travail est de pouvoir utiliser ASC comme un nouveau format d'entrée de cet outil (cf. figure 1).

Le premier chapitre de ce manuscrit définit le contexte dans lequel se situe ce travail. Il présente, dans une première partie, les concepts et les outils liés aux circuits asynchrones, et s'attarde plus particulièrement sur la classe des circuits quasi insensibles aux délais.

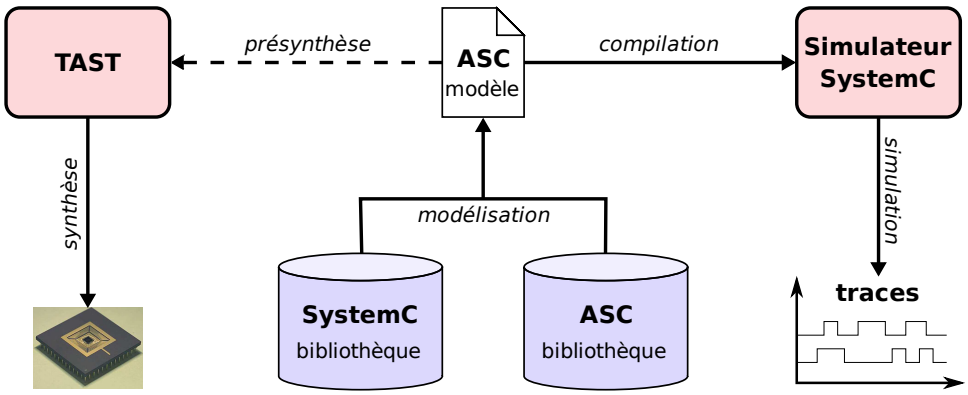


FIGURE 1 – *Flot de conception ASC*

La deuxième partie de ce chapitre présente les réseaux sur puce et les problèmes de synchronisation qui leur sont associés pour faire communiquer les composants d'un circuit constitué de plusieurs domaines d'horloge.

Le deuxième chapitre présente les bibliothèques SystemC et TLM. Au cours de cette thèse les atouts et les faiblesses de ces deux bibliothèques, pour modéliser et concevoir des circuits asynchrones, y ont notamment été répertoriés en détails. Cette étude se base, entre autre, sur des exemples de réseaux sur puce asynchrones Octagon que j'ai spécifiés en SystemC/TLM.

Le troisième chapitre présente en détails la bibliothèque ASC que j'ai développée pour pouvoir modéliser fidèlement des circuits asynchrones en SystemC. Ce chapitre répertorie aussi des exemples d'utilisation pratique de cette bibliothèque. Celle-ci a notamment été utilisée par le CEA-LETI pour modéliser le nœud de routage du réseau sur puce asynchrone FAUST et l'architecture superscalaire SERAPHIN.

Le quatrième chapitre présente le modèle de temps distribué AST que j'ai défini pour dater les différents événements qui surviennent dans un circuit asynchrone. Ce modèle de temps a été notamment employé, au cours de cette thèse, pour implémenter les facilités de traçage de la bibliothèque ASC qui permettent de visualiser l'activité de ses canaux de communication sous la forme de chronogrammes.

Le cinquième chapitre présente l'algorithme de présynthèse des structures de choix ASP que j'ai développé : cet algorithme, dirigé par la syntaxe, est formellement défini à l'aide d'une grammaire attribuée et peut être, par exemple, employé avec les structures de choix ASC. Le principal intérêt de cet algorithme est de supporter des structures de choix qui ne sont généralement pas ou mal supportées par les outils de synthèse de circuit asynchrone.

Finalement, le dernier chapitre de ce manuscrit présente les conclusions et les perspectives de ces différents travaux.

Chapitre 1

Les Circuits Asynchrones et les Réseaux sur Puce

Introduction

La majorité des circuits intégrés numériques conçus actuellement est basée sur les hypothèses de discrétisation des valeurs et de discrétisation du temps. La discrétisation des valeurs permet d'associer des valeurs binaires à tous les signaux. Les circuits vérifiant cette hypothèse sont appelés des circuits numériques. La discrétisation du temps consiste à définir une base de temps commune à tous les composants d'un circuit. Cette base de temps est généralement définie à l'aide d'un signal d'horloge distribué dans tout le circuit. Les circuits qui respectent cette hypothèse temporelle forment la classe des circuits dits synchrones.

L'hypothèse de discrétisation du temps permet de beaucoup simplifier la conception d'un circuit. Un circuit synchrone peut être vu comme un ensemble de blocs logiques qui effectuent leurs calculs entre chaque front d'horloge et qui communiquent entre eux à chaque front d'horloge. Un concepteur de circuit synchrone n'a donc pas à se soucier des valeurs transitoires que peuvent prendre les signaux de son circuit entre deux fronts d'horloge. Cependant l'utilisation d'une horloge globale engendre aussi les contraintes suivantes.

Premièrement, la période de temps entre deux fronts d'horloge doit être supérieure ou égale au temps nécessaire au bloc logique le plus lent pour effectuer son calcul. Un circuit synchrone fonctionne donc à la vitesse de son composant le plus lent.

Deuxièmement, la distribution du signal d'horloge aux différents blocs logiques doit respecter des contraintes temporelles fortes. En effet, il faut que les temps de propagation de ce signal vers les différents blocs logiques soient identiques. Avec l'avancée des progrès technologiques, cette propriété devient de plus en plus difficile à respecter. D'une part, la tolérance au décalage temporel du signal d'horloge est inversement proportionnelle à sa fréquence. D'autre part, l'intégration d'un nombre de plus en plus important de blocs logiques sur une même puce provoque une augmentation importante de la distance à parcourir par le signal d'horloge. L'augmentation de la longueur du signal d'horloge entraîne que le décalage temporel maximal entre deux blocs logiques d'un circuit est de plus en plus grand.

Troisièmement, l'utilisation d'un signal d'horloge global nécessite une énergie de plus

en plus importante. L'augmentation en fréquence et du nombre de transistors d'un circuit nécessitent d'utiliser des réseaux de distribution d'horloge de plus en plus complexes et qui consomment beaucoup d'énergie.

Quatrièmement, l'intégration de composants hétérogènes dans un même circuit n'est pas adaptée à la conception de circuit globalement synchrone. Les circuits actuels sont devenus des systèmes complexes, appelés système sur puce ou SoC (*System on Chip*), qui intègrent un ensemble de composants hétérogènes. Afin de réduire le coût de développement et le temps de conception, les industriels ont de plus en plus recours à des composants matériels propriétaires ou IPs (*Intellectual Properties*) pour réaliser ces SoCs. Ces IPs ont le plus souvent des fréquences de fonctionnement différentes et il est donc très difficile d'utiliser un signal d'horloge global pour synchroniser le fonctionnement de ces différents composants matériels.

Pour pallier ces différents problèmes, un nouveau paradigme de conception, appelé Globalement Asynchrone Localement Synchrone ou GALS [KGGV07] (*Globaly Asynchronous Localy Synchronous*), a été développé. Le principe de base de cette méthode de conception est de diviser le circuit en plusieurs domaines d'horloge distincts. Chaque composant d'un circuit GALS est toujours synchronisé par une horloge, cependant chaque composant possède sa propre horloge locale. Ces différents composants fonctionnent donc de façon asynchrone et se synchronisent localement lorsqu'ils communiquent. Néanmoins, l'introduction de plusieurs domaines d'horloge entraîne de nouvelles contraintes. En effet, la synchronisation des composants de ces circuits est assurée par leurs éléments de communication et leurs performances est donc en grande partie déterminée par ces éléments de communication : s'ils ne sont pas assez performants, les composants de calcul passent la majeure partie du temps à attendre des données à traiter ou à attendre de pouvoir envoyer le résultat de leurs traitements.

La méthode traditionnelle d'interconnexion des composants d'un circuit est d'utiliser un bus de communication centralisé. Elle ne permet cependant pas de faire communiquer efficacement un nombre important de composants et n'est donc pas adaptée pour relier les différents composants d'un circuit GALS. Pour résoudre ce problème d'interconnexion, les concepteurs de circuit GALS se sont inspirés des réseaux de communication utilisés en informatique par les calculateurs hautes performances [DYL02] qui permettent d'interconnecter efficacement un nombre important de composants.

Ce nouveau type d'interconnexion, appelé réseau sur puce ou NoC [DT01, BM02, MB06] (*Network on Chip*), est constitué d'un ensemble de routeurs interconnectés par des liens de communication point à point. Pour relier les différents composants d'un circuit, le réseau d'interconnexion d'un NoC doit être réparti sur l'ensemble du circuit. Le signal d'horloge de ces composants de communication doit donc parcourir une grande distance et est donc difficile à concevoir convenablement. Une solution simple et efficace pour résoudre ce problème est d'utiliser de la logique asynchrone [SE89, Ren00]

1.1 Les Circuits Asynchrones

La principale différence entre un circuit synchrone et un circuit asynchrone est que ce dernier n'utilise pas de signal global pour synchroniser l'exécution de ses composants. L'ensemble des circuits asynchrones est divisé en plusieurs familles qui sont définies en fonction des hypothèses temporelles qu'elles respectent. Ce manuscrit s'intéresse plus particulièrement à la classe des circuits asynchrones QDI [Mar90b] car leurs caractéristiques

sont bien adaptées à la conception des réseaux sur puce. Cependant, l'adoption de ces circuits par les industriels est pour l'instant limitée à cause du manque d'outils de conception adaptés aux flots de conception actuels.

1.1.1 Les Principes et les Propriétés des Circuits Asynchrones

La différence fondamentale entre un circuit synchrone et un circuit asynchrone est que ce dernier ne fait pas d'hypothèse sur la discrétisation du temps. Les composants logiques d'un circuit asynchrone se synchronisent localement en utilisant des protocoles basés sur des rendez vous. Cette synchronisation locale, permet à un circuit asynchrone de ne pas utiliser un signal d'horloge global. Pour qu'un circuit synchrone fonctionne correctement, il est nécessaire que les délais de ses composants soient inférieurs à la période de l'horloge. Les circuits asynchrones n'ont en général pas besoin de faire d'hypothèse temporelle aussi forte. Comme l'illustre la figure 1.1, il existe différentes classes de circuit asynchrone qui sont définies en fonction des hypothèses temporelles qu'ils doivent respecter pour fonctionner correctement.

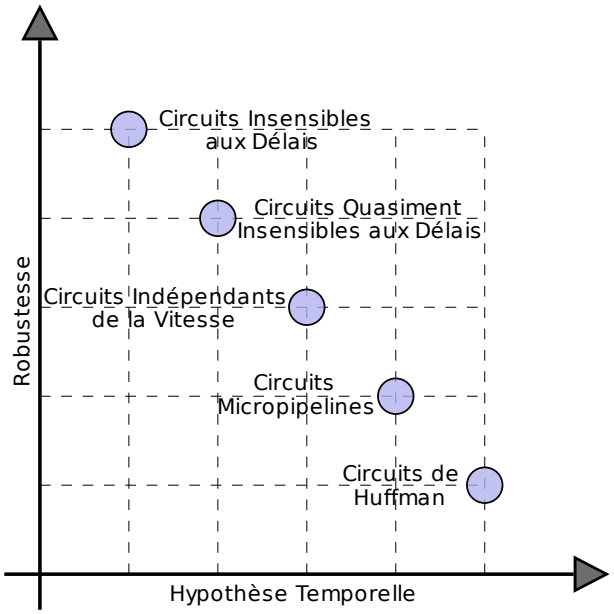
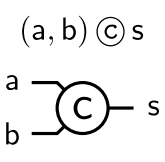


FIGURE 1.1 – *Classification des circuits asynchrones*

1.1.1.1 Les Circuits Insensibles aux Délais

Les circuits insensibles aux délais ou DI (*Delay Insensitive*) ne font aucune hypothèse temporelle. En effet, le comportement fonctionnel de ces circuits est indépendant des valeurs des délais de leurs fils et de leurs portes. L'utilisation pratique de ces circuits est cependant limitée [Mar90a] car ils ne permettent pas de réaliser n'importe quelle fonctionnalité. Ceci provient du fait que la seule porte à une sortie et plusieurs entrées qui peut être utilisée pour implémenter ces circuits est la porte de Müller ou « C-Element » (cf. figure 1.2).



b \ a	0	1
0	0	s^{-1}
1	s^{-1}	1

FIGURE 1.2 – *Porte de Müller à deux entrées*

1.1.1.2 Les Circuits Quasi Insensibles aux Délais

Les circuits quasi insensibles aux délais ou QDI (*Quasi Delay Insensitive*) utilisent le même modèle de délai que la classe de circuit DI en ajoutant l'hypothèse de fourche isochrone [Mar90a]. Une fourche est un signal qui réplique son entrée sur plusieurs sorties. Une fourche est isochrone (cf. figure 1.3) lorsque les délais de ses branches sont identiques. La différence entre un circuit DI et un circuit QDI est que certaines fourches d'un circuit QDI doivent être isochrones. L'intérêt de ces derniers est qu'ils permettent d'utiliser toutes les portes à une sortie et plusieurs entrées et donc d'implémenter n'importe quel type de circuit [MM95].

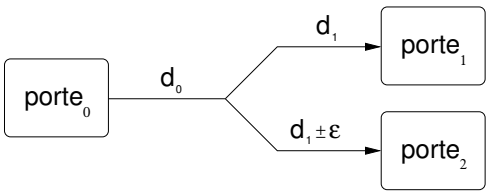


FIGURE 1.3 – *Fourche isochrone*

1.1.1.3 Les Circuits Indépendants de la Vitesse

L'hypothèse temporelle qui caractérise la classe des circuits indépendants de la vitesse ou SI (*Speed Independent*) est que les délais des fils sont négligeables. Cette hypothèse temporelle revient à assumer que toutes les fourches d'un circuit SI sont isochrones. La conception des circuits SI est donc plus contrainte que celle des circuits QDI. En effet, un concepteur de circuit QDI n'a pas à assurer que toutes les fourches de son circuit sont isochrones. Le modèle QDI lui permet d'identifier les fourches qui doivent respecter cette propriété d'isochronisme.

1.1.1.4 Les Circuits Micropipelines

La classe des circuits micropipelines [Sut89] est plutôt définie en fonction d'un modèle d'architecture que d'un modèle de délai. Comme l'illustre la figure 1.4, ces circuits sont constitués d'une partie contrôle insensible aux délais et d'une partie combinatoire identique à celle des circuits synchrones. Des retards sont ajoutés dans les parties de contrôle pour synchroniser le fonctionnement des parties combinatoires et des parties de contrôle. Un inconvénient de ces circuits est que leurs conceptions requièrent une analyse des délais de chacun de leurs blocs combinatoires pour pouvoir fixer correctement le retard des blocs de contrôle.

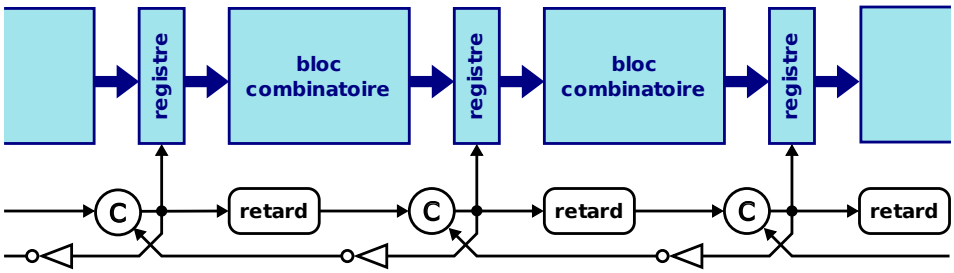


FIGURE 1.4 – *Structure de base des circuits Micropipelines*

1.1.1.5 Les Circuits de Huffman

Les circuits de Huffman forment la classe de circuit asynchrone qui fait les hypothèses temporelles les plus fortes. Cette classe de circuit suppose que les délais de toutes les portes et tous les fils sont bornés et connus.

1.1.2 Les Circuits Asynchrones QDI

Les circuits asynchrones QDI sont la classe de circuit qui est principalement étudiée dans cette thèse. Un circuit asynchrone QDI est un assemblage de composants indépendants, qui communiquent via des canaux de communication. Afin de préserver l'hypothèse temporelle d'insensibilité aux délais propres aux circuits asynchrones QDI, les canaux de communication de ces circuits emploient des codes spécifiques. Cette propriété d'insensibilité aux délais procure aux circuits asynchrones QDI de nombreux avantages pour concevoir des réseaux sur puce.

1.1.2.1 Une Synchronisation Locale par Communication

Comme l'illustre la figure 1.5, un circuit QDI est constitué d'un ensemble de composants indépendants qui sont interconnectés les uns aux autres via des canaux de communication unidirectionnels. L'activité de ces composants est synchronisée par les données qui sont présentes sur les canaux de communication. Le comportement générique d'un composant asynchrone est dans un premier temps d'attendre des données sur ses entrées, dans un deuxième temps de traiter les données reçues et dans un dernier temps d'envoyer le résultat de ce traitement. La synchronisation de l'émission et de la réception de données entre deux composants est effectuée par les canaux de communication grâce à un protocole à poignée de main. Ce protocole à poignée de main est réalisé par un canal à l'aide de ses deux signaux suivants :

- le signal de requête qui indique qu'un composant est prêt à communiquer,
- le signal d'acquittement qui permet à un composant de signaler que la communication est terminée.

Le protocole de communication à poignée de main n'est pas symétrique. En effet, il permet de distinguer les deux types de participants suivants :

- le participant actif qui initie la communication via le signal de requête,
- le participant passif qui termine la communication avec le signal d'acquittement.

Les acteurs d'une communication se distinguent aussi en fonction du fait qu'ils émettent ou qu'ils reçoivent des données. Ces deux critères permettent de définir les deux protocoles à poignée de main [SF01] « *push* » et « *pull* » qui sont illustrés par la figure 1.6.

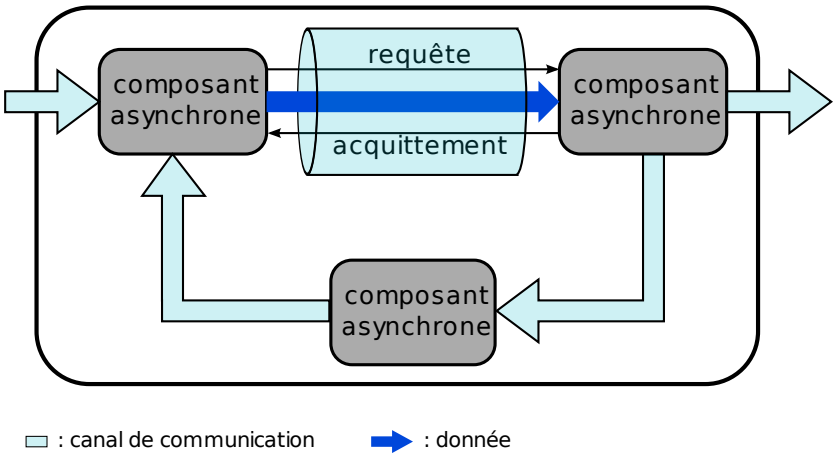


FIGURE 1.5 – *Structure de base des circuits QDI*

Avec le protocole à poignée de main *push*, l'émetteur et le récepteur sont respectivement actif et passif. Les rôles sont inversés pour le protocole à poignée de main *pull*.

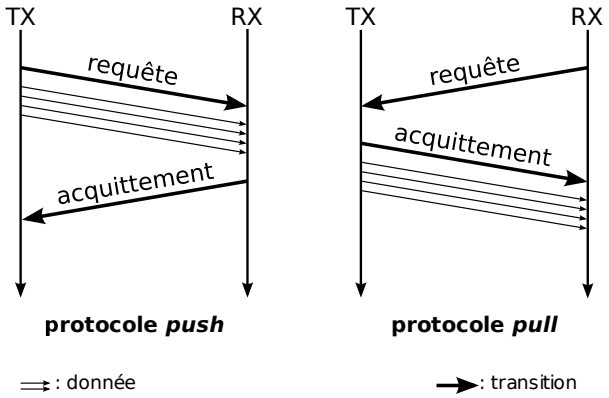


FIGURE 1.6 – *Protocole de communication à poignée de mains*

Une propriété intéressante du protocole à poignée de main est qu'un composant passif peut savoir à tout instant si le composant actif auquel il est connecté est prêt à communiquer. Cette propriété est exploitée par les circuits QDI avec le mécanisme de *probe* [Mar84]. Cette primitive de communication non bloquante permet à un composant passif de tester l'état d'un canal sans terminer l'action de communication. Lorsqu'une communication a été initiée sur un canal de type *push*, le composant passif relié à ce canal peut aussi tester (sans la consommer) la valeur de la donnée communiquée. L'introduction de cette primitive de communication a de fortes implications sur la conception d'un circuit QDI (cf. paragraphe 3.1.8).

1.1.2.2 Le Codage des Données

Pour fonctionner correctement, le modèle de canal utilisé par les circuits QDI doit respecter l'hypothèse temporelle suivante : le délai de propagation du signal de requête doit être supérieur ou égal au délai de propagation des données. En effet, pour fonctionner correctement, un composant passif (respectivement actif) récepteur ne doit jamais recevoir un signal de requête (respectivement d'acquittement) d'un canal avant que les données ne

soient disponibles sur ce canal. Cependant, le modèle de délai des circuits QDI implique que l'on ne peut pas faire d'hypothèse temporelle sur les délais de propagation des fils d'un canal. Pour respecter ces hypothèses temporelles conflictuelles, les circuits QDI ont recours au codage des données insensible aux délais [Ver88]. La caractéristique principale de cette famille de code est d'intégrer le signal de requête dans les données. Cette propriété permet donc d'assurer le comportement correct d'un canal de communication sans faire d'hypothèse temporelle sur les délais de propagation de ses fils.

Le code insensible aux délais le plus utilisé pour concevoir des circuits QDI est le code « *multi-rails* ». Ce code est aussi appelé code « 1 parmi N » car il permet de représenter N valeurs différentes. Comme l'illustre la figure 1.7, une donnée est représentée avec ce code par N fils qui représentent chacun une des valeurs différentes que peut prendre ce code. Une contrainte à respecter pour utiliser ce code est que au plus un des N fils représentant une donnée soit activé en même temps. Lorsque tous les fils sont inactivés, la valeur est dite invalide. Au contraire lorsqu'un des fils est activé, la valeur est dite valide. Les états valide et invalide sont utilisés par un canal de communication pour représenter respectivement l'activation et la désactivation du signal de requête.

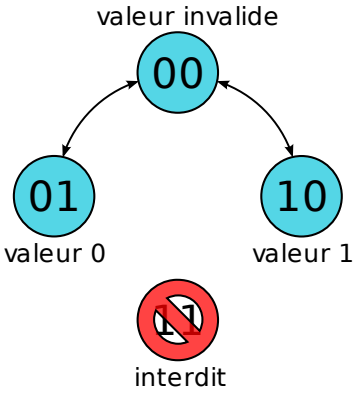


FIGURE 1.7 – *Exemple de code 1 parmi 2*

1.1.2.3 Les Avantages et les Limitations des Circuits QDI

Les propriétés des circuits asynchrones QDI sont bien adaptées pour résoudre les problèmes liés à la conception des NoCs. En plus de résoudre le problème de la distribution du signal d'horloge, la propriété d'insensibilité aux délais procure de nombreux avantages.

Premièrement, ils sont très robustes aux variations de la température et/ou de la tension d'alimentation. La principale conséquence de ces phénomènes est de modifier les délais du circuit. Pour un circuit synchrone, ceci peut avoir des conséquences importantes. Ces phénomènes risquent en effet d'amener des blocs combinatoires à ne pas respecter les délais de l'horloge et donc à ne plus pouvoir garantir la fonctionnalité du circuit. Pour un circuit asynchrone QDI, la conséquence de ces phénomènes est seulement de modifier sa vitesse de fonctionnement. Par contre, quelles que soient ces variations de température et/ou de tension [1] la fonctionnalité du circuit est préservée.

Deuxièment, ils sont peu sensibles aux variations technologiques. La fonctionnalité d'un circuit QDI n'est pas affectée par les différences entre les délais réels des portes et les délais théoriques des portes qui sont utilisées pour concevoir le circuit.

[1] Dans la limite de non destruction du circuit.

Troisièmement, ils permettent de réaliser des arbitres et des synchroniseurs parfaitement fiables. Cette propriété très intéressante pour la conception d'un NoC est présentée en détail au paragraphe 1.2.3.

Malgré ces propriétés intéressantes, les circuits QDI ont du mal à s'imposer dans le monde industriel. En effet, l'utilisation de code insensible aux délais et de protocoles de communication évolués nécessite une surface importante.

Les contraintes [Mye01] liées à la conception des circuits QDI sont un autre frein à leur adoption par les industriels. Contrairement à un circuit synchrone, un circuit asynchrone doit être sans aléa pour fonctionner correctement. En effet, dans un circuit asynchrone tout changement de valeur d'un signal est considéré comme un événement et doit donc être pris en compte. Comme l'illustre la figure 1.8, l'utilisation de portes logiques standards doit donc être effectuée avec précaution car elle peut engendrer des aléas qui sont incompatibles avec le bon fonctionnement d'un circuit asynchrone.

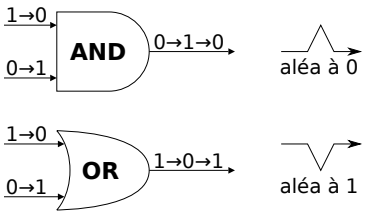


FIGURE 1.8 – *Exemple d'aléas sur des portes « AND » et « OR »*

Une autre contrainte forte à l'adoption des circuits QDI est le manque d'outils de conception [ET04] adaptés pour concevoir des systèmes complexes tels que des réseaux sur puce.

1.1.3 Les Flots de Conception

Deux grandes familles d'outils sont actuellement disponibles pour concevoir des circuits asynchrones. Une première famille est constituée des outils qui s'appuient sur des structures de graphes. Les outils les plus populaires qui appartiennent à cette famille sont Petrify, 3D et Minimalist. La deuxième famille se compose des outils qui utilisent des langages de programmation. Les principaux outils de cette famille sont TiDE, CAST et TAST.

1.1.3.1 Petrify

Petrify [CKK+97] est un outil de synthèse de circuits asynchrones indépendants de la vitesse. Un circuit est modélisé pour cet outil sous la forme d'un graphe de transition de signaux ou STG (*State Transition Graph*). Un STG est un réseau de Pétri qui respecte un ensemble de propriétés permettant de modéliser correctement un circuit asynchrone. Par exemple, un STG doit être « sûr », c'est-à-dire que chacune de ses places ne doit contenir au plus qu'un jeton. Le principal avantage de cet outil est de synthétiser des circuits très performants (vitesse, surface, ...). Son utilisation est cependant limitée à la conception de petits contrôleurs, car les STG ne permettent pas de modéliser facilement les parties opératives d'un circuit. De plus, le nombre d'états à traiter par l'outil augmente exponentiellement en fonction du nombre de signaux. En pratique, il est limité à la conception de circuits ayant au plus une vingtaine de signaux. Une dernière limitation de cet outil tient

au format d'entrée. Décrire un circuit sous la forme d'un STG est en effet fastidieux et sujet à erreurs.

1.1.3.2 3D et Minimalist

3D [YD92] et Minimalist [FNT+99] sont des outils de conception de circuits asynchrones de Huffman. Ces deux outils représentent un circuit à l'aide de formalismes dérivés des machines à états asynchrones. La principale différence entre une machine à états synchrone et une machine à états asynchrone est la condition provoquant un changement d'état. Avec une machine à états synchrone, un changement d'état survient uniquement à chaque front d'horloge. Avec une machine à états asynchrone, les changements de valeurs des signaux d'entrées provoquent un changement d'état. Ces deux outils ont les mêmes limitations que Petrify (explosion combinatoire, format d'entrée inadapté). De plus les circuits produits par ces outils ne sont pas très robustes car ils sont très sensibles aux variations des délais.

1.1.3.3 TiDE

TiDE [2] [BKR+91] (*Timeless Design Environment*) est un environnement de conception complet permettant de concevoir des circuits asynchrones micropipelines. Ce flot de conception utilise le langage Haste pour modéliser des circuits asynchrones. Haste est un langage basé sur CSP [Hoa78] (*Concurrent Sequential Processes*) qui modélise un circuit asynchrone comme un ensemble de processus concurrents qui communiquent via des canaux de communication. Un modèle Haste peut être synthétisé à l'aide de l'outil de synthèse de TiDE. La méthode de synthèse de cet outil est dirigée par la syntaxe. Cette méthode permet de synthétiser des circuits de façon simple et efficace. Elle consiste à associer un composant matériel pour chaque construction du langage possible. Le principal défaut de cette méthode est de ne pouvoir que très faiblement optimiser le circuit obtenu.

1.1.3.4 CAST

CAST [Mar91] (*Caltech Asynchronous Synthesis Tools*) n'est pas un outil mais une méthode de conception de circuit asynchrone QDI. Tout comme TiDE cette méthode de conception repose sur un langage de programmation basé sur CSP appelé CHP (*Concurrent Hardware Processes*). Cependant CAST utilise une technique de synthèse différente dite par compilation. Le principe de cette méthode consiste à raffiner en plusieurs étapes successives la spécification initiale. L'objectif final étant d'obtenir une spécification du circuit qui soit directement synthétisable sous forme de portes logiques. L'avantage de cette méthode est qu'elle permet de bien mieux optimiser les circuits obtenus qu'une approche dirigée par la syntaxe. De plus, cette méthode ne souffre pas du problème d'explosion combinatoire des techniques de synthèse basées sur des graphes. La principale limitation à l'adoption de cette technique est qu'il n'existe pas à l'heure actuelle d'outil automatique de conception.

[2] http://www.handshakesolutions.com

1.1.3.5 TAST

TAST [Bre07, Fol07] (*TIMA Asynchronous Synthesis Tools*) est un environnement de conception de circuits asynchrones QDI. Comme l'illustre la figure 1.9, TAST est composé de plusieurs outils qui définissent un flot de conception complet. Tout comme CAST, ce flot de conception prend en entrée un programme CHP. Un programme CHP peut être compilé vers différents formats en fonction des outils que l'on veut utiliser.

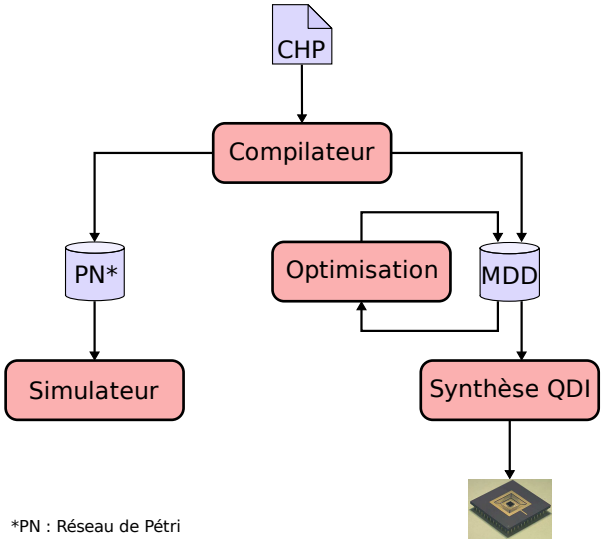


FIGURE 1.9 – *Flot de conception TAST*

L'outil de synthèse utilise notamment des diagrammes de décisions multi-valuées ou MDD (*Multi-Decision Diagram*). La structure de MDD est une généralisation de la structure de diagramme de décision binaire ou BDD (*Binary-Decision Diagram*) qui est couramment utilisée pour la synthèse des circuits synchrones et pour le model checking. Cette structure offre de nombreux avantages et notamment celui de permettre la synthèse de circuits respectant les hypothèses d'insensibilité aux délais propres aux circuits QDI. TAST permet donc de combiner les avantages des deux familles d'outils dédiées à la synthèse des circuits asynchrones. En effet, il permet de synthétiser des circuits performants (grâce à l'utilisation des MDDs) à partir d'un langage de programmation de haut niveau.

L'utilisation de CHP comme format d'entrée ne permet toutefois pas à TAST d'être facilement adopté par le monde industriel. Les concepteurs de circuits synchrones ne sont en effet pas familiers avec ces langages. De plus, ils ne permettent pas de s'interfacer facilement avec les outils utilisés dans les flots de conception des circuits synchrones. Ce dernier point est pourtant essentiel pour concevoir un réseau sur puce d'un circuit GALS. Pour valider par simulation une architecture GALS, il est nécessaire que les différents langages de description de matériels utilisés pour spécifier ses composants puissent facilement s'interfacer. De plus, les langages utilisés doivent permettre de spécifier ces composants à différents niveaux d'abstraction afin de disposer rapidement d'une plateforme de simulation. La solution développée dans cette thèse pour utiliser le flot de conception TAST efficacement est de définir un nouveau format d'entrée basé sur SystemC (cf. chapitre 3).

1.2 Les Réseaux sur Puce

Pour pouvoir connecter efficacement les différents composants d'un circuit GALS, un nouveau type d'interconnexion appelé réseau sur puce ou NoC (*Network on Chip*) a été développé. En plus de permettre la communication entre les différents composants d'un circuit GALS, un réseau sur puce est aussi chargé d'assurer la synchronisation des différents domaines d'horloge. Contrairement aux circuits synchrones, les circuits asynchrones QDI permettent de résoudre de manière plus fiable cette contrainte. Ceci est l'une des raisons principales pour laquelle l'utilisation des circuits asynchrones QDI est de plus en plus en populaire pour réaliser des réseaux sur puce.

1.2.1 Les Composants d'un Réseau sur Puce

Comme l'illustre la figure 1.10, un réseau sur puce est constitué de trois types de composants. Les interfaces réseau sont les éléments qui permettent d'accéder au réseau sur puce. Les nœuds de routage sont chargés de diriger les données qu'ils reçoivent vers leurs destinataires. Les liens de communication assurent le transfert des données entre deux composants du réseau sur puce (interfaces réseau ou nœuds de routage).

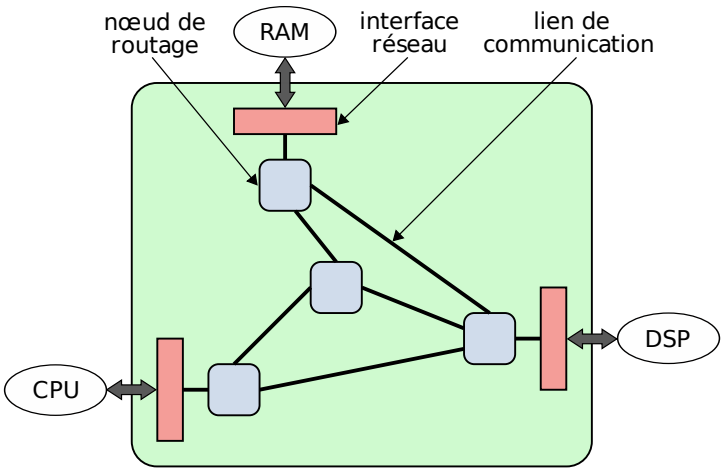


FIGURE 1.10 – *Structure d'un Réseau sur Puce*

1.2.1.1 Les Interfaces Réseau

Les interfaces réseau sont les points d'entrée-sortie d'un réseau sur puce. Une interface réseau fournit un ensemble de services de communication de haut niveau (e.g. paquétisation, contrôle de flux, ordonnancement des communications, ...) aux composants qui lui sont connectés. Elle se sert des primitives de communication de base du réseau sur puce pour répondre aux appels à ses services de communication.
Comme l'illustre la figure 1.11, une interface réseau est divisée en deux parties :

- une partie frontale (de l'anglais *front end*) qui est reliée à un composant fonctionnel du circuit,
- une partie arrière (de l'anglais *back end*) qui est reliée au réseau sur puce.

Pour faciliter la réutilisation des composants fonctionnels, la partie frontale d'une interface réseau implémente généralement un protocole de communication point à point

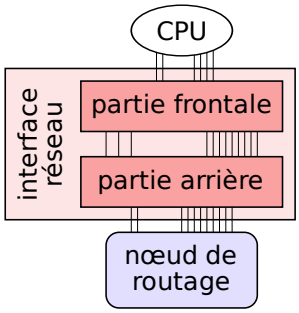


FIGURE 1.11 – *Architecture d'une interface réseau*

standardisé. Les principaux standards actuels sont OCP [3] (*Open Core Protocol*), AXI [4] (*Advanced eXtensible Interface*) et DTL (*Device Transaction Level*). Un standard tel que OCP définit notamment les différents signaux matériels que doit posséder la partie frontale d'une interface réseaux. Cela étant, l'utilisation de tels standards génériques ne permet généralement pas d'exploiter pleinement les capacités d'un réseau sur puce particulier. L'utilisation de ces standards est donc le résultat d'un compromis que doit réaliser un concepteur entre la performance et la flexibilité.

Dans un circuit GALS, les interfaces réseau sont aussi chargées d'assurer la synchronisation entre les domaines d'horloge du réseau sur puce et des composants qui lui sont connectés. L'utilisation de circuit asynchrone permet de résoudre efficacement ce problème (cf. paragraphe 1.2.3.1).

1.2.1.2 Les Nœuds de Routage

Le rôle d'un nœud de routage est d'aiguiller les données qu'il reçoit vers leurs destinataires. La figure 1.12 illustre l'architecture générique d'un nœud de routage qui est constituée des éléments suivants.

Les files d'attente qui sont utilisées pour stocker les données en transit. Elles permettent aussi de lisser le trafic sans bloquer les émetteurs (tant que les files ne sont pas pleines).

Le commutateur qui connecte les files d'attente d'entrée aux files d'attente de sortie.

L'unité de routage et d'arbitrage qui définit comment doit être configuré le commutateur pour que les données contenues dans les tampons d'entrée soient correctement aiguillées. De plus, il doit gérer les conflits d'accès lorsque plusieurs files d'attente d'entrée doivent être connectées à la même file de sortie.

1.2.1.3 Les Liens de Communication

Le rôle principal d'un lien de communication est de transférer des données entre deux composants d'un réseau sur puce. Avec l'avancement des progrès technologiques, cette tâche est de plus en plus difficile à accomplir sans erreur. Les fils utilisés pour réaliser les liens de communication d'un réseau sur puce sont de plus en plus fins mais les distances qu'ils doivent parcourir ne diminuent pas. Ces fils sont donc de plus en plus sensibles aux

[3] http://www.ocpip.org/home
[4] http://www.arm.com/products/solutions/AMBA3AXI.html

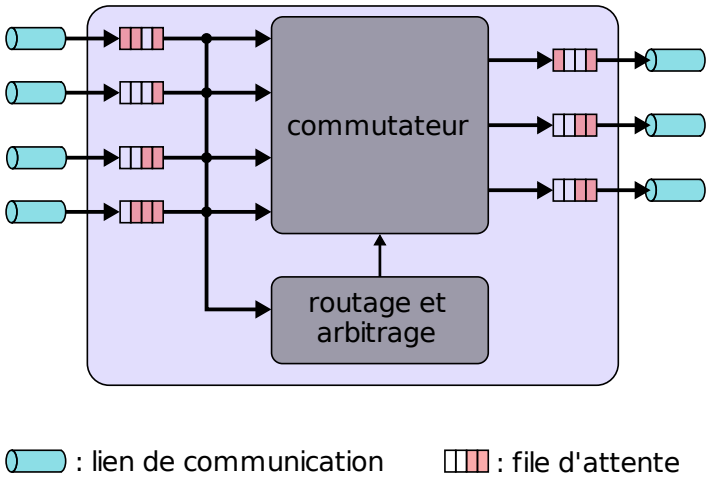


FIGURE 1.12 – *Structure d'un nœud de routage*

phénomènes de bruits, de diaphonie, ... et il est donc difficile de garantir l'intégrité des valeurs qu'ils communiquent.

Une solution permettant de résoudre ce problème est d'utiliser des répéteurs ou des FIFOs (*First In First Out*) asynchrones pour segmenter les fils ayant de longues distances à parcourir. Une autre solution complémentaire est d'utiliser des codes correcteurs et/ou détecteurs d'erreurs.

1.2.2 Les Caractéristiques d'un Réseau sur Puce

Tout comme les réseaux de communication utilisés par les calculateurs haute performance, les réseaux sur puce sont caractérisés par un ensemble de paramètres qui permettent de les adapter aux besoins d'une application donnée. En fonction de ces besoins, les premiers paramètres à fixer sont de définir comment interconnecter les composants du réseau sur puce et de préciser comment ils communiquent entre eux. Un dernier paramètre à déterminer est de choisir un algorithme de routage permettant à un nœud de routage de décider vers quel lien de communication il doit propager les données qu'il reçoit pour qu'elles atteignent leurs destinations.

1.2.2.1 La Topologie

La topologie d'un réseau sur puce définit comment sont interconnectés ses différents composants. Elle est souvent modélisée sous la forme d'un graphe où les nœuds de routage et les liens de communication sont respectivement associés aux sommets et aux arrêtes de ce graphe. Cette modélisation permet d'utiliser les propriétés suivantes des graphes pour caractériser un réseau sur puce :

- le diamètre qui est égal à la distance (en nombre de liens de communication) maximale qui sépare deux nœuds de routage du réseau,
- le degré d'un nœud de routage qui est égal aux nombre de liens de communication qui lui sont connectés,
- la largeur de bissection qui est égale au nombre minimal de liens de communication nécessaires pour séparer le réseau en deux ensembles disjoints de même taille (en nombre de nœuds de routage),

– la régularité qui est respectée lorsque tous les nœuds de routage ont le même degré,
...

De manière générale, un réseau sur puce est dit régulier lorsque sa construction suit une loi géométrique. Par exemple, une grille à deux dimensions est considérée comme une topologie régulière bien que les degrés de ses nœuds ne soient pas tous identiques. En effet cette topologie peut être facilement définie à partir de ses propriétés géométriques [DYL02].

L'utilisation des propriétés géométriques des topologies régulières permet de réaliser des algorithmes de routage simples et efficaces. Cela étant, pour fonctionner efficacement, le trafic transitant sur des réseaux sur puce ayant une topologie régulière doit être homogène. Ces réseaux sur puce sont donc bien adaptés à la famille des circuits multiprocesseurs ou CMP [CK05] (*Chip MultiProcessors*) qui sont composés d'unités fonctionnelles semblables pouvant toutes communiquer les unes avec les autres. Si le trafic généré par un circuit CMP n'est pas prédictible, il est souvent réparti de manière uniforme. En effet, les composants fonctionnels d'un CMP ont des motifs de communication qui sont symétriques les uns par rapport aux autres car ils ont tous des comportements similaires.

Lorsque le trafic est prédictible mais non homogène, il est préférable d'utiliser des topologies non régulières. L'intérêt de ces topologies est de pouvoir être facilement adaptées en fonction des débits requis localement. Elles sont notamment très populaires pour réaliser les réseaux sur puce des circuits dédiés aux applications multimédia. Ces circuits sont généralement constitués de composants fonctionnels hétérogènes dont les motifs de communication sont hétérogènes et souvent prédictibles statiquement.

1.2.2.2 Les Techniques de Commutation

Les techniques de commutation définissent comment sont utilisées les différentes ressources d'un réseau sur puce pour acheminer des données. Les deux techniques de commutation suivantes sont principalement utilisées par les réseaux sur puce.

La commutation de circuits

Cette technique de commutation consiste dans un premier temps à réserver les ressources nécessaires du réseau sur puce pour établir une connexion logique entre l'émetteur et le récepteur. Puis dans un deuxième temps à utiliser cette connexion logique pour transférer les données depuis l'émetteur vers le récepteur.

Le principal avantage de ce mécanisme de réservation est d'assurer que deux flots de communication ne rentreront jamais en conflit une fois leurs connexions logiques établies. Ce type de commutation permet donc à un réseau sur puce de facilement offrir des garanties de service sur les latences et les débits. Un désavantage important de cette méthode de commutation est d'avoir une latence importante à cause du temps nécessaire à l'établissement de la connexion.

La commutation de paquets

Le principe de cette technique de commutation est de découper un message en un ensemble de paquets indépendants. Pour que chaque paquet puisse atteindre sa destination, des informations dites de routage leurs sont ajoutées. Ces informations de routage sont utilisées par les nœuds de routage pour aiguiller les paquets qu'ils reçoivent vers leurs destinations. Contrairement à la commutation de circuits, les ressources nécessaires à l'acheminement d'un paquet ne sont pas réservées avec la commutation de paquets.

Il peut donc survenir que plusieurs paquets transitants par le même nœud de routage veuillent être aiguillés vers le même lien de communication. Ce type de conflit d'accès à une ressource partagée est appelé contention. Pour résoudre cette situation, les nœuds de routage emploient des composants appelés arbitres qui déterminent quel paquet peut emprunter le lien de communication.

Il existe différents modes de commutation de paquets qui définissent comment sont transférés les paquets entre deux nœuds de routage via un lien de communication. Cependant, les réseaux sur puce utilisent essentiellement le mode de commutation de paquets dit par « ver de terre » ou « *wormhole* ». Ce mode de commutation de paquets consiste à diviser un paquet en un ensemble d'éléments de taille fixe appelé *flit* (*FLow control unIT*). Un *flit* est transféré via un lien de communication dès que la place nécessaire à son stockage est disponible dans le nœud de routage connecté à la sortie de ce lien de communication. Lors de son transit dans le réseau sur puce, un paquet peut donc être réparti sur plusieurs nœuds de routage. Cependant tous les *flits* d'un paquet doivent suivre le même chemin et les *flits* de différents paquets ne peuvent pas être entrelacés aux cours de leur parcours.

Les principaux avantages du mode de commutation par ver de terre sont d'avoir une faible latence en moyenne et surtout de permettre d'utiliser des files d'attentes de faible taille. Cette dernière propriété est la principale raison pour laquelle ce mode de commutation de paquets est utilisé. En effet, la surface occupée par les points de mémorisation est une des contraintes les plus fortes des réseaux sur puce. Cependant, ce mode de commutation de paquets doit être employé avec précaution à cause des risques importants d'interblocage qui lui sont associés.

1.2.2.3 Les Algorithmes de Routage

Un algorithme de routage est utilisé par un nœud de routage pour déterminer vers quel lien de communication aiguiller les paquets qu'il reçoit. Le rôle d'un algorithme de routage est donc de définir le chemin (i.e. la suite des liens de communication à utiliser) que doit suivre un paquet pour atteindre sa destination. En fonction des caractéristiques d'un réseau sur puce et de ses besoins (e.g. qualité de service, robustesse, ...) différents types d'algorithmes de routage peuvent être utilisés. Ces différentes familles d'algorithmes de routage sont classées en fonction des propriétés suivantes.

Routage déterministe et routage adaptatif : un algorithme de routage est déterministe (ou statique) lorsqu'il ne dépend que de la localisation de l'expéditeur et du récepteur d'un paquet. A l'opposé, un algorithme de routage adaptatif (ou dynamique) prend en compte l'état du réseau (e.g. charge des liens de communication, liens de communication défectueux, ...) pour prendre ses décisions.

Routage par la source et routage distribué : avec un routage par la source, la suite des liens de communication à utiliser est indiquée dans les informations de routage de chaque paquet. Le rôle d'un nœud de routage se limite donc à extraire et à appliquer les informations de routage contenues dans un paquet. Au contraire, avec un routage distribué, un paquet ne contient que l'adresse de sa destination. Un nœud de routage doit donc calculer pour chaque paquet vers quel lien de communication il doit l'aiguiller.

En plus de définir le chemin que doit suivre un paquet pour atteindre sa destination, un algorithme de routage doit aussi assurer que l'ensemble des chemins suivis par les

paquets ne crée pas des situations d'interblocage statique (ou *deadlock*). Comme l'illustre la figure 1.13, ce phénomène survient lorsque plusieurs paquets se bloquent mutuellement à cause d'une dépendance cyclique. La commutation de paquets par ver de terre est très sensible à ce phénomène car un paquet peut être réparti sur plusieurs nœuds de routage. Différentes méthodes et théories ont été développées pour éviter l'apparition de ce phénomène. Les méthodes les plus populaires sont de limiter les chemins que peut emprunter un paquet [GN92] et/ou d'utiliser des canaux virtuels [DS87].

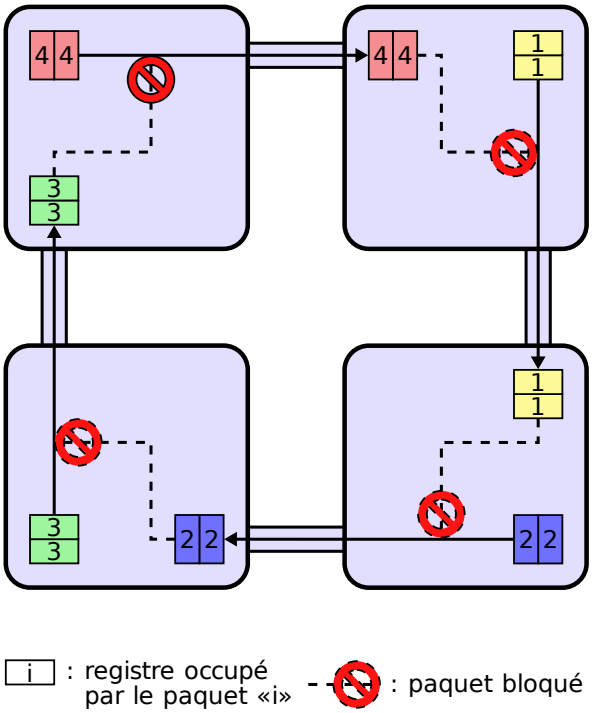


FIGURE 1.13 – *Exemple d'interblocage statique*

Les canaux virtuels sont des files d'attente d'un nœud de routage et sont utilisés pour partager l'accès à un lien de communication. Comme l'illustre la figure 1.14, les canaux virtuels permettent d'éviter les interblocages statiques en autorisant un paquet temporairement bloqué à se faire doubler par un paquet utilisant les mêmes liens de communication mais des canaux virtuels différents. Les canaux virtuels sont aussi employés pour améliorer les performances [Dal92] d'un réseau sur puce car ils permettent de diminuer la contention de ses nœuds de routage. Finalement, les canaux virtuels peuvent être utilisés pour offrir différents niveaux de qualité de service [BCGK04] en attribuant différents niveaux de priorité à chaque canal virtuel.

1.2.3 Les Réseaux sur Puce Asynchrones et les Architectures GALS

Le principe des architectures GALS est de diviser un circuit en plusieurs domaines d'horloge distincts. Un circuit GALS utilise généralement un réseau sur puce pour faire communiquer ses différents composants car il permet de résoudre efficacement les contraintes suivantes.

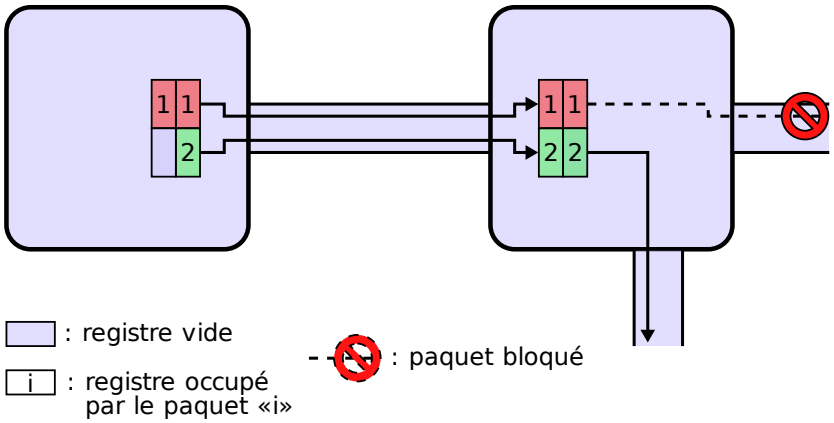


FIGURE 1.14 – *Exemple d'utilisation de deux canaux virtuels*

- Le passage à l'échelle : l'utilisation de liens de communication point à point partagés permet d'adapter la bande passante en fonction du nombre de composants à interconnecter.
- La flexibilité : l'utilisation d'interfaces réseau standardisées permet de remplacer facilement les composants connectés à un NoC.
- La qualité de service : l'utilisation de protocoles de communication plus ou moins complexes permet d'adapter la qualité de service en fonction des besoins des applications.

L'émergence de ces nouvelles méthodes de conception entraîne aussi un intérêt croissant pour les technologies asynchrones et notamment QDI. En effet, les propriétés spécifiques des circuits asynchrones sont bien adaptées aux réseaux sur puce. Un premier avantage des réseaux sur puce asynchrones est de ne pas avoir à résoudre le problème difficile de la distribution d'un signal d'horloge. Un deuxième avantage des circuits asynchrones est d'avoir une faible consommation dynamique. En effet, la seule énergie consommée par les composants d'un circuit asynchrone non utilisés est celle due aux courants de fuite. Un troisième avantage propre aux familles des circuits asynchrones insensibles aux délais (DI, QDI, SI), est d'utiliser des codes pour représenter les données qui sont robustes aux problèmes de diaphonie et de bruit [BF01]. Cette propriété est très avantageuse pour un réseau sur puce car ses liens de communication doivent parcourir de longues distances et sont donc très sensibles à ces problèmes. Un dernier avantage des réseaux sur puce asynchrones est d'utiliser des synchroniseurs et des arbitres parfaitement fiables [Qua04].

1.2.3.1 La Synchronisation des Communications

La principale contrainte d'un circuit GALS est de synchroniser les communications entre des composants appartenant à des domaines d'horloge différents [Kin08]. Cette synchronisation s'effectue à l'aide de circuits spécifiques appelés synchroniseurs synchrones. La tâche d'un synchroniseur synchrone est d'échantillonner les données qui lui parviennent en fonction d'un signal périodique. Un synchroniseur synchrone permet donc à un composant synchrone de recevoir des données émises à des dates indépendantes de son domaine d'horloge. Une limitation importante de ces synchroniseurs est de n'être pas totalement fiables à cause du phénomène de métastabilité [CM73, Gin03].

Un état métastable se définit comme un état indéfini et instable qui converge vers un

état stable au bout d'une période de temps finie mais non bornée. Une situation pouvant amener un circuit à entrer dans un état métastable est d'échantillonner un signal à une valeur intermédiaire. Pour limiter l'impact de ce phénomène, un synchroniseur synchrone est généralement réalisé avec deux bascules en série. Cela étant, ce synchroniseur n'est pas parfaitement fiable car il ne fonctionne correctement que si sa première bascule n'est jamais dans un état métastable pendant une période de temps supérieure à la période d'échantillonnage. Un autre désavantage de ce synchroniseur est d'introduire une latence importante (double échantillonnage) au temps de transmission d'une donnée.

Comme l'illustre la figure 1.15, l'utilisation d'un réseau sur puce asynchrone permet de diviser par deux le nombre de synchroniseurs synchrones. En effet, un réseau sur puce asynchrone utilise des synchroniseurs asynchrones sans horloge qui sont parfaitement fiables et qui ont des temps de latence moyens nettement inférieurs [NM02]. Les réseaux sur puce asynchrones permettent donc d'améliorer la fiabilité d'un circuit GALS et de diminuer la latence de ses communications [SMPG07].

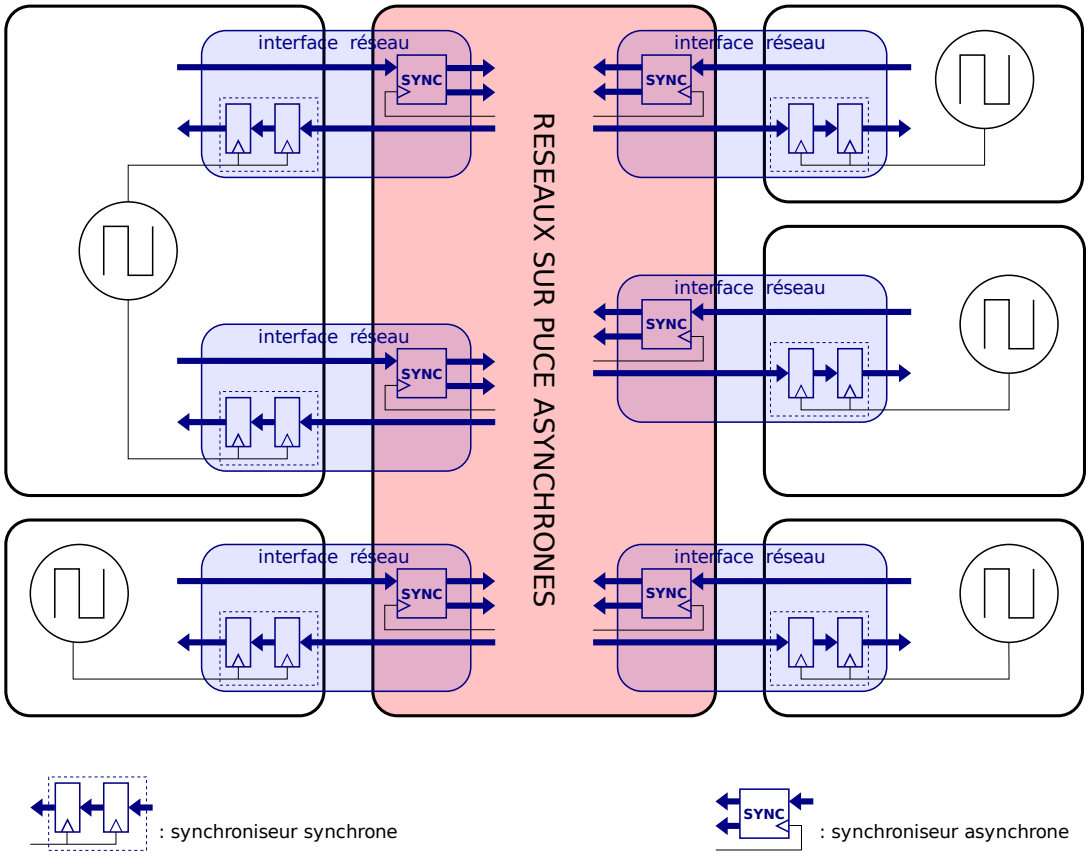


FIGURE 1.15 – *Synchronisation d'un circuit GALS avec un Réseau sur Puce Asynchrone*

1.2.3.2 L'Arbitrage

Le rôle d'un arbitre dans une architecture distribuée comme un réseau sur puce est de recevoir des requêtes et d'allouer des ressources partagées (e.g. l'accès aux liens de communication) en fonction de ces requêtes [Kin08]. Pour être performant, un arbitre synchrone ne vérifie pas ses entrées à chaque cycle d'horloge mais attend qu'on lui soumette des requêtes pour accéder à une ressource partagée. Les signaux de requête qui parviennent

à un arbitre synchrone sont donc asynchrones vis-à-vis de son horloge. Ces signaux doivent donc être échantillonnés par des synchroniseurs synchrones qui ne sont pas complètement fiables.

Un point fort d'un arbitre asynchrone est de traiter une requête dès qu'il la reçoit sans avoir à l'échantillonner. Contrairement à un arbitre synchrone, ce n'est pas un signal d'horloge qui déclenche le traitement d'une requête mais sa réception. Cependant, une contrainte de ce mode de fonctionnement est de faire apparaître le phénomène de métastabilité lorsque plusieurs requêtes arrivent dans un petit intervalle de temps. En effet, une autre situation qui peut conduire un circuit dans un état métastable est de devoir déterminer parmi plusieurs signaux lequel a effectué en premier une transition. Pour résoudre ce problème, les arbitres asynchrones ont recours à un composant matériel appelé opérateur d'exclusion mutuelle ou *mutex*. Une limitation de cet opérateur est d'avoir un temps de traitement non borné car il doit résoudre le phénomène de métastabilité. Cependant, cette contrainte permet de rendre son comportement parfaitement fiable et de pouvoir concevoir des arbitres asynchrones qui soient parfaitement fiables [MN06].

Une conséquence importante du phénomène de métastabilité pour les arbitres asynchrones est de rendre leurs comportements non déterministes. En effet, lorsqu'un arbitre asynchrone entre dans un état métastable, il devient très sensible aux phénomènes extérieurs (bruit électrique, température, ...) et la valeur de sa sortie n'est donc pas prédictible.

1.2.4 Exemples de Réseau sur Puce

L'étude des réseaux sur puce est à l'heure actuelle un domaine de recherche très dynamique qui a été le sujet de nombreuses publications et a entraîné la réalisation de nombreux prototypes. Des états de l'art complets sur ces différents travaux de recherches [BM06] et réalisations [Rom06, MCM+04] sont disponibles.

1.2.4.1 Octagon

L'Octagon [KNDR01] est une architecture de réseau sur puce développée par STMicroelectronics et l'université de San Diego qui adopte une topologie octogonale à cordes (cf. figure 1.16). Ce réseau sur puce a notamment été utilisé au cours de cette thèse pour évaluer les bibliothèques SystemC/TLM (cf. paragrapahe 2.2.4) et ASC (cf. paragraphe 3.2.1).

Un paquet qui transite dans un Octagon peut être acheminé soit en utilisant une commutation de circuit, soit en employant une commutation de ver de terre. Cette topologie octogonale procure à ce réseau sur puce les deux avantages suivants :

- un faible diamètre (distance maximale de deux liens de communication),
- un algorithme de routage (déterministe et distribué) simple.

L'algorithme de routage de l'Octagon se base sur les propriétés géométriques de sa topologie. Le principe de cet algorithme est de calculer une adresse relative qui est utilisée pour définir vers quel lien de communication un paquet doit être aiguillé. L'adresse relative $adRel$ d'un paquet est calculée en fonction de son adresse de destination $adDest$ et de l'adresse du nœud qu'il a atteint $adNd$ avec la formule suivante :

$$adRel = (adDest - adNd) \bmod 8$$

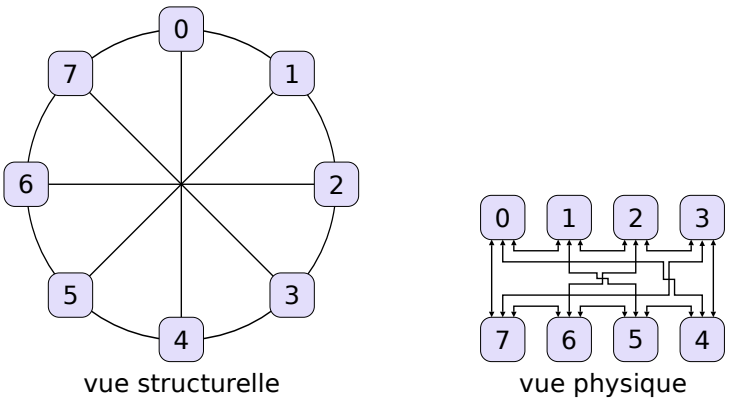


FIGURE 1.16 – *Topologie du réseau sur puce Octagon*

Pour déterminer comment aiguiller un paquet ayant une adresse relative *adRel*, un nœud de routage utilise les règles suivantes :

- $adRel = 0$, le paquet est arrivé à sa destination,
- $adRel \in \{1, 2\}$, le paquet est aiguillé dans le sens horaire,
- $adRel \in \{6, 7\}$, le paquet est aiguillé dans le sens anti-horaire,
- $adRel \in \{3, 4, 5\}$, le paquet est aiguillé vers le nœud diamétralement opposé.

Pour pouvoir passer à l'échelle, ce réseau sur puce peut être étendu en associant plusieurs Octagons par l'intermédiaire de nœuds jouant le rôle de ponts [KND02]. Cette topologie octogonale a aussi été généralisée à une topologie polygonale à cordes appelée Spidergon [CLM+04].

1.2.4.2 Æthereal

Æthereal [GDR05] est un réseau sur puce synchrone développé par Philips ayant une topologie non régulière et utilisant un algorithme de routage par la source. Les interfaces réseau de Æthereal supportent différents protocoles de communication standard (OCP, AXI et DTL). La caractéristique principale de Æthereal est d'avoir des routeurs qui offrent deux niveaux de qualité de service différents. Le niveau « service garanti » assure des garanties sur les débits et sur les latences contrairement au niveau « meilleur effort » qui n'en offre aucune.

Les paquets ayant un niveau de qualité de service meilleur effort sont acheminés par les routeurs en utilisant la technique de commutation par ver de terre. Une technique de commutation de circuit basée sur du multiplexage temporel ou TDM (*Time Division Multiplexing*) est utilisée pour transporter les paquets de type service garanti. Cette technique de commutation consiste à créer un circuit logique en réservant des périodes de temps dans des routeurs. Ce mécanisme permet d'assurer que les paquets qui empruntent un de ces circuits arriveront à destination sans entrer en conflit avec d'autres paquets. Le principal désavantage de cette approche est que la latence garantie est inversement proportionnelle au débit garanti. Ce mécanisme de commutation ne permet donc pas d'établir une connexion ayant un faible débit et une faible latence. Ce type de connexion est toutefois important car il permet de supporter correctement les communications liées aux interruptions.

1.2.4.3 MANGO

MANGO [Bje05] (*Message-passing Asynchronous Network-on-Chip providing Guaranteed services over OCP interfaces*) est un réseau sur puce asynchrone développé par l'Université Technique du Danemark qui adopte une topologie en forme de grille et qui possède des interfaces réseau [BMOS05] conforme à la norme OCP. Les liens de communication de MANGO sont réalisés en utilisant des codes insensibles aux délais pour diminuer leur consommation et augmenter leur robustesse [BF01]. A l'inverse, les nœuds de routage sont conçus comme des circuits micropipelines afin de diminuer leur surface. Comme Æthereal, les nœuds de routage [BS05a] de MANGO fournissent les deux niveaux de qualité de services meilleur effort et service garanti.

Une connexion à service garanti est établie en réservant une suite de canaux virtuels [BS04] et en appliquant une politique d'arbitrage appelé ALG [BS05b] (*Asynchronous Latency Guarantees*). L'objectif de cette politique d'arbitrage est d'attribuer l'accès à un lien de communication de manière à respecter les latences et les débits associés aux canaux virtuels connectés à ce lien de communication. Le principal avantage de cette technique par rapport au multiplexage temporel employé par Æthereal est que les garanties sur les latences sont découplées des garanties sur les débits. Une limitation de cette technique est de ne pas permettre d'utiliser complètement la bande passante disponible pour les connections à service garanti.

1.2.4.4 Chain

Chain [BF02] (*CHip Area INterconnect*) est un réseau sur puce conçu par l'Université de Manchester ayant une topologie non régulière et utilisant un routage par la source. Chain définit un ensemble de composants de base (arbitres, routeurs, bascules, ...) qui doivent être assemblés les uns avec les autres pour former un réseau sur puce complet. Contrairement à MANGO, tous les composants de Chain sont réalisés avec des circuits asynchrones QDI utilisant un codage insensible aux délais 1 parmi 4. Les travaux de recherche sur ce réseau sur puce ont donné naissance à la start-up silistix [5] qui propose des outils automatisant l'assemblage des composants de base de Chain. Ces outils permettent de réaliser rapidement un réseau sur puce adapté à une application donnée.

1.2.4.5 SPIN

SPIN [GG00, AG03] (*Scalable Programmable Integrated Network*) est un réseau sur puce développé par le laboratoire LIP6 (Laboratoire d'Informatique de Paris 6) qui employait initialement une topologie régulière « d'arbre élargi » (de l'anglais *fat tree*). Si cette topologie permet d'utiliser efficacement les ressources matérielles d'un réseau sur puce, elle n'est toutefois pas bien adaptée pour interconnecter les différents composants d'un Système sur Puce [MPGA06]. Pour remédier à cette limitation, les deux nouvelles versions de SPIN emploient une topologie en forme de grille et l'algorithme de routage déterministe distribué « X-first » [DS87].

La première nouvelle version de ce réseau sur puce, appelée DSPIN [MPGS06] (*Distributed Scalable Predictable Interconnect Network*), est constituée de nœuds de routage qui possèdent chacun leur propre horloge de synchronisation et qui sont reliés par des FIFOs bi-synchrones. Une restriction de cette implémentation est d'introduire des latences

[5] http://www.silistix.com

importantes pour synchroniser les communications entre les différents domaines d'horloge des nœuds de routage.

La deuxième nouvelle version de SPIN, appelée ASPIN [SMPG07]((*Asynchronous Scalable Predictable Interconnect Network*), est composée de nœuds de routage asynchrones qui communiquent entre eux en utilisant un protocole à poignée de main insensible aux délais. Un avantage de cette implémentation est de réduire la latence de transmission d'un paquet car seules ses interfaces réseau nécessitent d'être synchronisées avec des composants synchrones [SG08].

1.2.4.6 ANOC

ANOC [BCV+05] (*Asynchronous Network on Chip*) est un réseau sur puce asynchrone développé par le CEA-LETI qui est entièrement réalisé en logique asynchrone QDI. Ce réseau sur puce a une topologie régulière en forme de grille (cf. figure 1.17) et emploie un routage par la source déterministe. Les paquets sont transportés dans ce réseau sur puce en utilisant une commutation de paquets par ver de terre. Les nœuds de routage de ANOC sont composés de deux canaux virtuels ayant des priorités différentes. Ces canaux virtuels permettent à un paquet transporté par le canal virtuel prioritaire de doubler les paquets empruntant le canal virtuel à faible priorité. Contrairement à MANGO, ces canaux virtuels ne fournissent pas des garanties dures sur les latences et les débits car les paquets circulant sur le canal prioritaire peuvent entrer en contention. Les interfaces réseau [BV06] de ANOC utilisent un protocole de communication spécifique. Ces interfaces réseau utilisent des files d'attente basées sur des codes de Grey, pour permettre une synchronisation robuste entre les différents domaines d'horloge qu'elles relient.

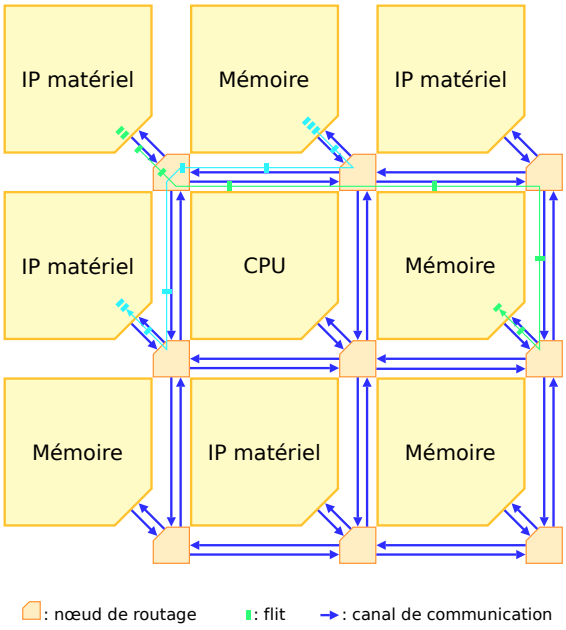


FIGURE 1.17 – *Architecture du réseau sur puce ANOC*

Le réseau sur puce ANOC a été utilisé pour réaliser le circuit GALS FAUST [DBL05] (*Flexible Architecture of Unified System for Telecom*) dédié aux applications de télécommunication 4G. Afin de disposer d'une plateforme de simulation efficace, le réseau sur puce ANOC a été modélisé à différents niveaux d'abstraction à l'aide des bibliothèques

de modélisation SystemC et SystemC/TLM. Cette plateforme de simulation a permis de valider tôt dans le cycle de conception les différents composants du circuit FAUST et aussi de valider les nouveaux protocoles de communication du réseau sur puce ANOC. Une des contributions de cette thèse a été de développer une extension de la bibliothèque SystemC qui a permis de modéliser fidèlement le comportement des nœuds de routage de ce réseau sur puce (cf. paragraphe 3.2.2).

Chapitre 2

Les Bibliothèques SystemC et TLM

Introduction

SystemC est une bibliothèque du langage de programmation C++ [Str97] permettant de modéliser des systèmes complexes tels que des Systèmes sur Puce. Un modèle SystemC est donc un programme C++ qui peut être compilé avec un compilateur C++ afin d'obtenir un modèle exécutable simulant son comportement. Un des principaux avantages de SystemC est de pouvoir modéliser un système à différents niveaux d'abstraction. La version 2.1 de SystemC est définie par un standard IEEE [IEE06] et une version *open-source* (source ouverte en français) est disponible sur le site de l'OSCI [1] (*Open SystemC Initiative*).

Le paragraphe 2.1 de ce chapitre présente les concepts et les propriétés de SystemC qui sont nécessaires à la compréhension de cette thèse. Il présente aussi les limitations de SystemC pour concevoir des circuits asynchrones (principalement la modélisation et la validation). En effet, SystemC a été initialement prévu pour réaliser des circuits synchrones et utilise donc des mécanismes de synchronisation globale des communications. Cependant, cette bibliothèque n'est pas seulement prévue pour modéliser les composants matériels d'un système mais aussi ses composants logiciels. Cette dernière caractéristique propre à SystemC permet de modéliser des circuits asynchrones fidèlement. En effet, les circuits asynchrones et les composants logiciels d'un système ont des propriétés communes telles que l'utilisation de mécanismes de synchronisation locale des communications. Néanmoins, les primitives de communication utilisées par des composants logiciels ne sont pas les mêmes que celles utilisées par les circuits asynchrones. Pour pouvoir modéliser correctement des circuits asynchrones, la bibliothèque SystemC de base doit donc être étendue.

La bibliothèque TLM [CG03] (*Transaction Level Modeling*) est une extension de SystemC qui permet de modéliser des réseaux sur puce à des niveaux d'abstraction élevés. Cette bibliothèque permet notamment d'avoir un modèle SystemC purement fonctionnel d'un réseau sur puce. Cette extension de SystemC a été étudiée en détails au cours de cette thèse car elle possède des propriétés (hypothèse temporelle, mode de communication, ...) similaires à celles des circuits asynchrones (cf. paragraphe 2.2).

[1] Association à but non lucratif qui est chargée de spécifier et de diffuser SystemC : `http://www.systemc.org`

2.1 La Bibliothèque SystemC

SystemC définit un ensemble de classe C++ qui permet de modéliser un système de manière hiérarchisée en séparant ses aspects fonctionnels de ses aspects de communication. De plus, SystemC est fourni avec un noyau de simulation et des facilités de traçage qui permettent de valider par simulation le comportement d'un modèle SystemC. Un des points forts de SystemC est d'être un projet *open-source* et donc de pouvoir être facilement étendu. Cette propriété fondamentale permet notamment de pouvoir s'affranchir des limitations de la bibliothèque SystemC standard pour modéliser des circuits asynchrones en développant ses propres extensions.

2.1.1 La Modélisation d'un Système

Comme l'illustre la figure 2.1, un modèle SystemC est un modèle hiérarchique composé d'un ensemble de modules (TOP, BUS, ARBITRE, ...) interconnectés les uns aux autres via des ports, des canaux et des exports. La fonctionnalité d'un module est modélisée par ses processus qui sont exécutés de façon concurrente. La synchronisation de leurs exécutions est réalisée grâce aux événements (non représentés sur la figure). Les interfaces spécifient les primitives de communication utilisées par les processus pour communiquer entre eux. Les ports sont les points d'entrée-sortie des modules qui permettent aux processus d'accéder aux primitives de communication des canaux. Les canaux (ch_1, ch_2, ...) modélisent les aspects de communication d'un système en implémentant les primitives de communication des interfaces (non représentées sur la figure). Un export permet de connecter des ports et des canaux qui sont localisés dans des modules disjoints.

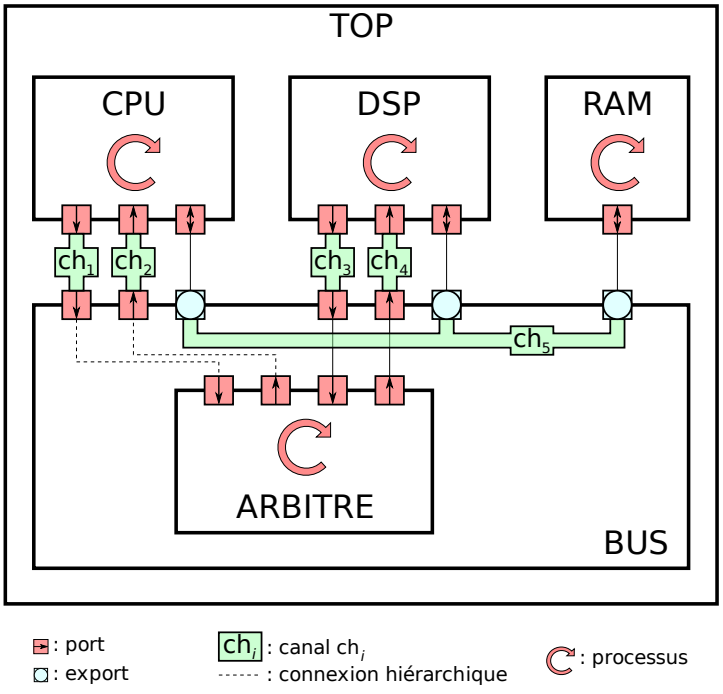


FIGURE 2.1 – *Exemple de modèle SystemC*

2.1.1.1 Les Modules

Les modules sont les éléments de base d'un modèle SystemC. Ils permettent de décrire un système de manière hiérarchique en le décomposant en plusieurs sous modules. Un module peut contenir de manière générale n'importe quelle structure de données C++, tels que des entiers (`int`, `char`, ...), des modules SystemC, des ports SystemC, ...

La réalisation d'un module s'effectue en dérivant la classe de base `sc_module`. En plus d'initialiser ses différentes structures de données, le constructeur d'un module est aussi chargé de connecter ses différents éléments (canaux, ports et exports) et de déclarer ses processus.

2.1.1.2 Les Processus

Les processus expriment les aspects fonctionnels et concurrents d'un système. Le comportement d'un processus SystemC est défini par une procédure C++ qui lui est associée. La simulation d'un modèle SystemC consiste donc à exécuter de façon pseudo concurrente le code C++ de chacun de ses processus (cf. paragraphe 2.1.2.2). La version actuelle de SystemC distingue les deux catégories de processus suivantes :

- les processus sans contexte qui s'exécutent dans le contexte de l'ordonnanceur du simulateur,
- les processus avec contexte qui ont leurs propres contextes d'exécution (pile, tas, ...).

Les processus se distinguent aussi en fonction de la manière dont ils sont créés.

Un processus statique est créé par l'ordonnanceur du simulateur durant la phase d'élaboration en appelant le constructeur du module qui contient la méthode C++ associée à ce processus. Par exemple sur le programme 2.1, on peut voir que le constructeur du module `foo` crée à l'aide de l'instruction `SC_THREAD` un processus avec contexte dont le comportement est défini par la méthode `bar`.

```
class foo: public sc_module {
    public:
        // Constructor of the foo module
        foo(void) {
            // Initialisation of the bar process
            SC_THREAD(bar)
        }

        // Method defining the behavior of the bar process
        void bar(void) { ... }
};
```

PROGRAMME 2.1 – *Exemple de processus SystemC*

Un processus dynamique est créé en appelant la fonction `sc_spawn` soit par l'ordonnanceur au cours de la phase d'élaboration, soit par un autre processus durant la phase de simulation. Cette dernière méthode de création de processus a notamment été utilisée pour autoriser un processus SystemC à effectuer plusieurs communications en parallèle (cf. paragraphe 3.1.4).

L'ordonnancement de l'exécution des processus est assuré par le noyau de simulation en fonction de l'état courant de chaque processus. Comme l'illustre la figure 2.2, un processus

peut être dans l'un des trois états suivants : « élu », « éligible » ou « endormi ». Lorsqu'un processus est dans l'état élu, il est exécuté jusqu'à ce qu'il demande explicitement à passer dans l'état endormi. Lorsqu'un processus passe dans l'état endormi sa liste de sensibilité est mise à jour. Une liste de sensibilité définit les événements qui doivent survenir pour qu'un processus endormi puisse devenir éligible. Lorsqu'un processus devient éligible, cela signifie qu'il est prêt à être exécuté. Cependant, il n'est exécuté que lorsqu'il passe dans l'état élu. La politique d'élection d'un processus n'est pas définie par le standard et dépend donc de l'implémentation de la bibliothèque SystemC utilisée.

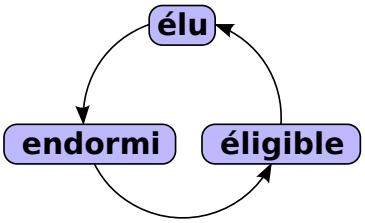


FIGURE 2.2 – *Diagramme d'état d'un processus SystemC*

Un processus sans contexte s'exécute dans le contexte de l'ordonnanceur et ne peut donc pas être suspendu au cours de son exécution : si on suspend un processus de ce type, la simulation s'arrête car l'ordonnanceur est aussi suspendu et il ne peut donc pas élire un nouveau processus éligible. Le code d'un processus sans contexte est donc toujours complètement exécuté (i.e. jusqu'à la première instruction `return`) et sa liste de sensibilité définit ses conditions de re-exécution. Comme le montre la figure 2.3, l'instruction `next_trigger` permet à un processus sans contexte de mettre à jour sa liste de sensibilité sans l'interrompre. Sur cet exemple le processus P_0 spécifie que l'événement `e` doit être notifié pour qu'il soit re-exécuté.

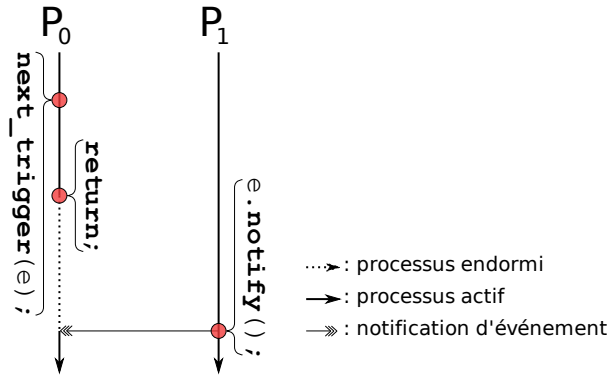


FIGURE 2.3 – *Exemple de réveil de processus sans contexte*

A l'opposé, un processus avec contexte peut être suspendu au cours de son exécution en appelant la méthode `wait`. Comme l'illustre la figure 2.4, cette méthode permet aussi de mettre à jour sa liste de sensibilité. Dans cet exemple, le processus avec contexte P_0 suspend son exécution en appelant la méthode `wait` et il définit la condition de son réveil avec la liste d'événements qu'il donne en paramètre à cette méthode. Cette liste construite avec l'opérateur C++ « `&` » indique que les événements e_1 et e_2 doivent être notifiés pour que le processus P_0 soit réveillé.

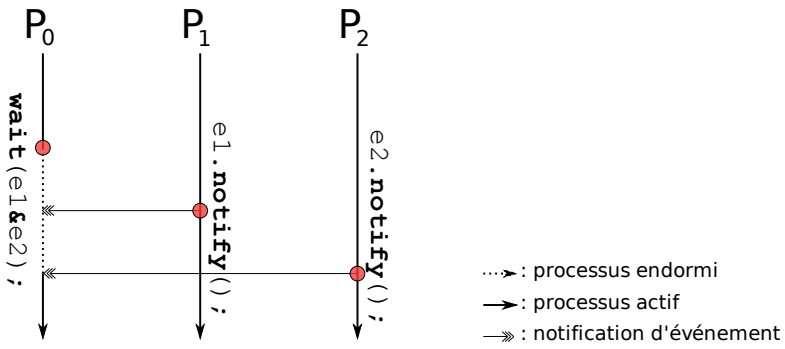


FIGURE 2.4 – *Exemple de réveil de processus avec contexte*

Lorsqu'un processus avec contexte a fini de s'exécuter, il passe dans l'état terminé et ne peut plus être re-exécuté. Ces processus permettent de modéliser un système plus facilement mais nécessitent un espace mémoire (pile, tas, ...) et des temps de simulation (rétablissement du contexte) plus importants.

2.1.1.3 Les Événements

Les événements sont les éléments de base utilisés pour synchroniser l'exécution des processus. Pour qu'un processus en attente d'un événement soit réveillé, il faut que cet événement soit notifié. La notification d'un événement s'effectue en deux étapes : la demande de notification et le déclenchement de la notification.

La demande de notification d'un événement consiste à définir à quelle date le déclenchement de la notification aura lieu. Une demande de notification s'effectue avec la méthode `notify` d'un événement qui peut être appelée des trois façons suivantes.

- `e.notify()` est une demande de notification immédiate de l'événement `e`. Le déclenchement de la notification s'effectue au même instant de simulation que la demande de notification.
- `e.notify(SC_ZERO_TIME)` est une demande de notification retardée de l'événement `e`. Le déclenchement de la notification s'effectuera au prochain delta-cycle (cf. paragraphe 2.1.2.1).
- `e.notify(200, SC_NS)` est une demande de notification temporisée de l'événement `e`. Le déclenchement de la notification s'effectuera dans 200 nanosecondes.

Un événement ne peut avoir, à tout instant, qu'une seule demande de notification en attente. Les règles suivantes permettent de résoudre les notifications multiples à un événement qui n'ont pas encore été déclenchées : « Une demande de notification plus récente va toujours écraser une demande plus tardive, sachant que :

- Une notification immédiate est plus récente qu'une notification retardée,
- Une notification retardée est plus récente qu'une notification temporisée. »

La méthode `cancel` d'un événement permet d'annuler une demande de notification temporisée ou retardée. Par contre il est impossible d'annuler une demande de notification immédiate.

Lorsque la date de déclenchement de notification d'un événement est atteinte, toutes les listes de sensibilité des processus en attente de cet événement sont mises à jour. En outre, les processus endormis, dont la liste de sensibilité ne contient plus d'événement en attente, sont réveillés (cf. figure 2.3 et figure 2.4).

Une différence fondamentale entre SystemC et un langage de description de matériel tel

que VHDL (*Very high speed integrated circuit Hardware Description Language*) est de ne pas avoir un comportement complètement déterministe : malgré les règles de résolution des notifications multiples, le comportement d'un modèle SystemC, utilisant des notifications immédiates, peut dépendre de l'ordonnancement de ses processus (page 10 de [OSC03]). Dans le cadre de la modélisation de circuits asynchrones, cette contrainte est un avantage car elle permet de modéliser fidèlement leurs comportements (cf. paragraphe 3.1.3).

2.1.1.4 Les Interfaces

Une interface définit un ensemble de primitives de communication, mais ne définit pas leurs implémentations. Les interfaces permettent d'appliquer le patron de conception [GHJV95] (de l'anglais *design pattern*) d'appel d'interface ou IMC [OSC02] (*Interface Method Call*) qui permet de relier un port à un canal. Le rôle d'une interface est de définir les primitives de communication d'un canal qui sont accessibles par un port. Avec ce patron de conception, le rôle d'un port se limite à transmettre les appels de ses méthodes vers le canal auquel il est connecté. Ce mécanisme permet de pouvoir facilement remplacer un canal par un autre canal qui implémente les mêmes interfaces et donc de rendre indépendante les parties fonctionnelles et de communication d'un modèle SystemC.

Le schéma de la figure 2.5 décrit l'utilisation du motif de programmation IMC. Dans cet exemple, le processus du module CONSOMMATEUR appelle la méthode `get` de son port *IP*. Ce port utilise l'interface `interface_in` qui définit une méthode abstraite `get`. On peut voir sur le schéma que le corps de cette méthode ne consiste qu'à appeler la méthode `get` de l'objet `this` en utilisant l'opérateur « `->` ». Comme nous le verrons au paragraphe 2.1.1.5, cet opérateur est surchargé et permet d'accéder aux méthodes du canal *CH* qui est connecté au port *IP* et qui dérive de la classe `interface_in`.

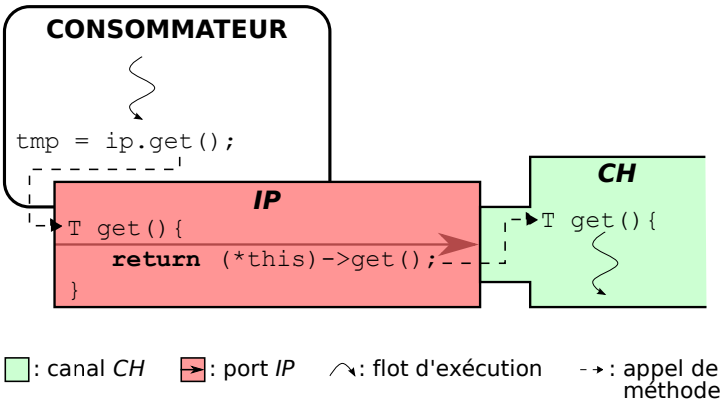


FIGURE 2.5 – *Exemple d'utilisation du motif de programmation IMC*

2.1.1.5 Les Ports

Les ports sont les points d'entrée-sortie des modules. Ils permettent aux processus de différents modules de communiquer via des canaux. Afin de pouvoir accéder à des canaux se trouvant à des niveaux hiérarchiques différents, il est possible de connecter un port directement à un autre port (cf. connexion hiérarchique de la figure 2.1).

Le diagramme de classe UML [PP05] (*Unified Modeling Language*) de la figure 2.6 illustre comment est réalisé la classe `port_in` du port *IP* vu au paragraphe précédant.

Cette classe hérite de la classe générique `sc_port` qui définit les propriétés de base des ports SystemC. Le paramètre générique `IF` de cette classe définit l'interface qui lui est associée. La classe `interface_in` est une interface qui hérite de la classe `sc_interface` et qui déclare une méthode abstraite `get`. Le comportement de cette méthode est défini par la classe `port_in` qui hérite de la classe générique `sc_port` en affectant au paramètre `IF` la classe `interface_in`. La classe `sc_port` redéfinit aussi l'opérateur `->` qui renvoie un pointeur sur le canal connecté à ce port.

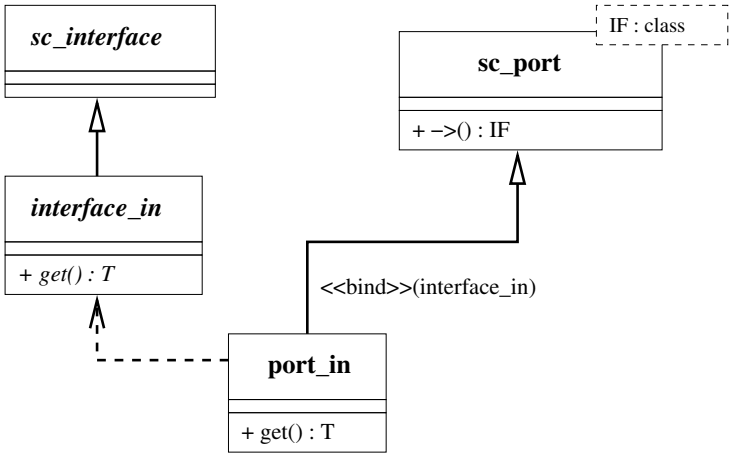


FIGURE 2.6 – *Diagramme de classe UML d'un port SystemC*

2.1.1.6 Les Canaux

Les canaux de communication sont les objets utilisés pour modéliser les aspects de communication d'un système. La bibliothèque SystemC définit un ensemble de canaux de communication de base tels que les `sc_signal` ou les `sc_fifo`. Un `sc_signal` est un canal qui modélise le comportement d'un fil en se basant sur les hypothèses de discrétisation des valeurs et du temps des circuits numériques synchrones. Un `sc_fifo` est un canal modélisant une file d'attente FIFO qui peut être instanciée avec une taille supérieure ou égale à un.

Un canal a recours à des événements et à la technique « *request-update* » pour synchroniser l'exécution des processus qui font appel à ses primitives de communication. Cette dernière technique de synchronisation consiste à décomposer l'accès à un canal en deux étapes successives afin de pouvoir résoudre convenablement les accès concurrents à ce canal (cf. paragraphe 2.1.2.1). Si cette dernière technique de synchronisation permet aux canaux SystemC de respecter l'hypothèse de discrétisation du temps des circuits synchrones, elle n'est toutefois pas adaptée pour modéliser des circuits asynchrones (cf. paragraphe 2.1.3.1).

Un des points forts de la bibliothèque SystemC est de pouvoir être facilement étendue en ajoutant des nouveaux canaux de communication à l'aide de la classe de base `sc_prim_channel` qui définit les propriétées de base des canaux de communication SystemC (mécanisme de connection à un port, méthode *request-update*, ...). Le diagramme de classe UML de la figure 2.7 montre comment est définie la classe `my_channel` qui représente un nouveau canal de communication. En plus de la classe `sc_prim_channel`, la classe `my_channel` hérite des deux interfaces `interface_in` et `interface_out` qui lui

permettent de se connecter avec n'importe quel port qui implémente une de ces deux interfaces (e.g. le port *IP* de la figure 2.5).

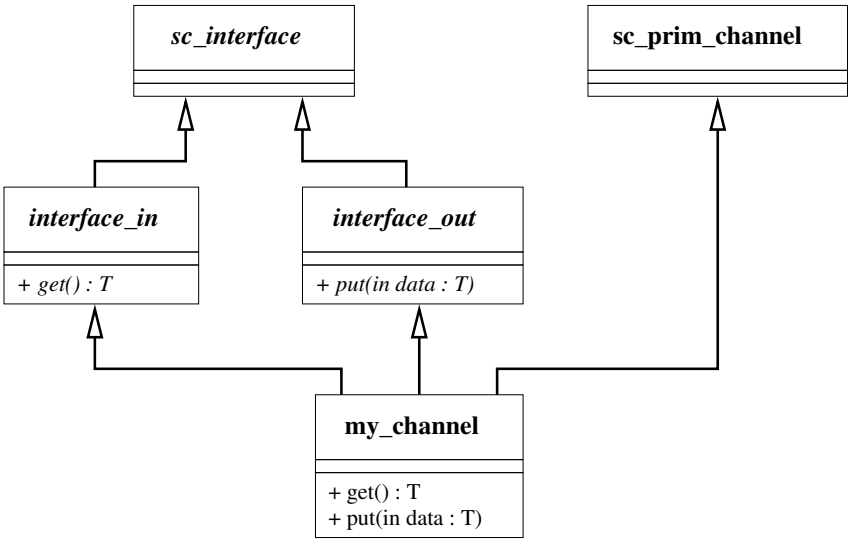


FIGURE 2.7 – *Diagramme de classe UML d'un canal SystemC*

Le schéma de la figure 2.8 décrit comment est transférée la donnée `data` entre les processus `prod` et `cons` via le canal *CH*. Ce canal possède un événement `e` qui lui permet de synchroniser la communication entre les deux processus de la manière suivante :

1. le processus `cons` appelle la méthode `get` et il est interrompu par l'appel de la méthode `wait`,
2. lorsque le processus `prod` a produit une donnée, il la transmet en appelant la méthode `put`,
3. la donnée `data` passée en paramètre de la méthode `put` est copiée dans la variable `buffer` du canal, puis l'événement `e` est notifié,
4. la notification de l'événement `e` entraîne le réveil du processus `cons` qui récupère la donnée dans la variable `buffer`.

Si la valeur de la donnée `data` est bien transférée avec ce nouveau canal, celui-ci n'est cependant pas réaliste car sa fonctionnalité dépend de l'ordonnancement des processus `prod` et `cons`. Les notifications d'événements ne sont, en effet, pas persistantes et elles sont donc à utiliser avec précaution pour synchroniser l'exécution des processus d'un modèle SystemC.

Des limitations importantes d'un canal de communication obtenu avec la classe `sc_prim_channel` sont de ne pas pouvoir contenir des structures de données complexes telles que des processus, des canaux, des modules, ... et de ne pas être autorisé à faire directement appel aux primitives de communication d'un autre canal. Pour remédier à ces limitations, il est possible de définir un canal de communication (appelé canal de communication hiérarchique) en remplaçant la classe `sc_prim_channel` par la classe `sc_module`. Une restriction importante à l'utilisation des canaux de communication hiérarchiques est de ne pas avoir accès à la technique *request-update* qui est limitée aux canaux de communication dérivant de la classe `sc_prim_channel`.

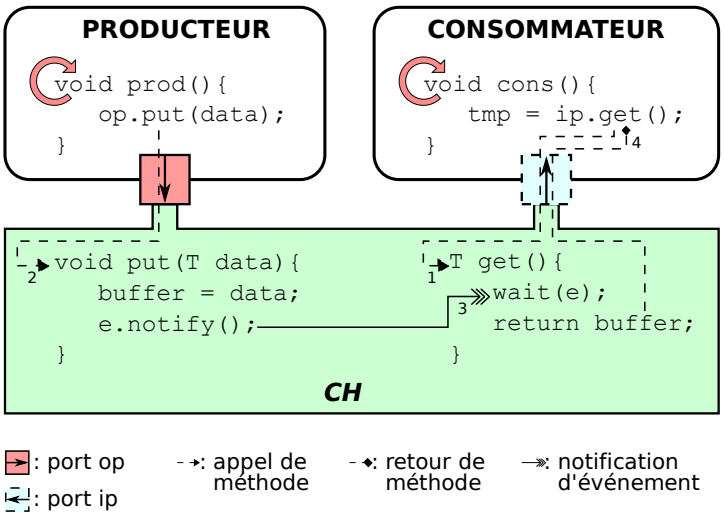


FIGURE 2.8 – *Exemple de canal SystemC*

2.1.1.7 Les Exports

Le rôle d'un export est de rendre visible un canal en dehors de son module. Un export permet à un port de se connecter à un canal se trouvant dans un module disjoint. Comme pour les ports, il est possible de connecter directement deux exports. Les exports permettent donc de rendre plus flexible la description hiérarchique d'un système. Ils sont notamment utilisés pour réaliser les ports cibles de la bibliothèque SystemC/TLM.

Sur la figure 2.1 on peut, par exemple, voir que les exports sont employés pour rendre visible le canal CH_5 à l'extérieur du module BUS. Les modules CPU, RAM et DSP peuvent ainsi se connecter à ce canal qui se trouve à un niveau hiérarchique différent.

2.1.2 La Simulation et la Validation

Le simulateur SystemC permet d'exécuter de façon concurrente les processus d'un modèle SystemC afin de valider son comportement. Ce simulateur utilise un modèle de temps discret adapté à la validation de modèles de composants matériels synchrones. La simulation d'un modèle SystemC se déroule en plusieurs étapes au cours desquelles des facilités de traçage peuvent être utilisées pour enregistrer l'activité de ces canaux.

2.1.2.1 Le Modèle de Temps

Le modèle de temps discret du simulateur SystemC utilise une échelle de temps physique (seconde, microseconde, nanoseconde, ...), mais chaque instant de simulation peut être décomposé en un nombre arbitraire de « delta-cycles ». Un delta-cycle peut être vu comme un intervalle de temps infiniment petit qui ne fait pas progresser le temps physique du simulateur. Comme l'illustre la figure 2.9, un delta-cycle d'un simulateur SystemC est constitué des trois phases suivantes.

La phase d'évaluation : pendant cette phase tous les processus qui sont dans l'état éligible sont exécutés séquentiellement. Cette phase se termine lorsqu'il n'y a plus de processus éligible.

La phase de mise à jour : durant cette phase, l'ordonnanceur du simulateur exécute les méthodes `update` des canaux primaires dont les méthodes `request_update` ont été appelées au cours de la phase d'évaluation précédente.

La phase de notification retardée : Au cours de cette phase, l'ordonnanceur déclenche les demandes de notification retardées qui ont été effectuées pendant les phases d'évaluation et de mise à jour précédentes.

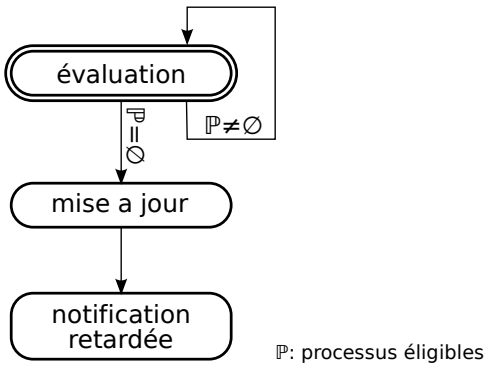


FIGURE 2.9 – *Diagramme d'état d'un delta-cyle SystemC*

La notion de delta-cycle est généralement utilisée pour rendre déterministe le comportement d'un modèle décrit à l'aide d'un langage de description de matériel. Les phases d'évaluation et de mise à jour sont par exemple utilisées en SystemC avec la technique *request-update* pour résoudre de manière déterministe les accès concurrents à un canal `sc_signal_resolve`. Comme l'illustre l'exemple de la figure 2.10, la phase de mise à jour est utilisée par l'ordonnanceur pour définir la valeur de sortie d'un canal `sc_signal_resolve` en fonction des écritures effectuées par les processus P_0 et P_1 pendant la phase d'évaluation. Cette technique permet donc d'affecter une valeur à un canal indépendamment de l'ordre d'exécution des processus P_0 et P_1.

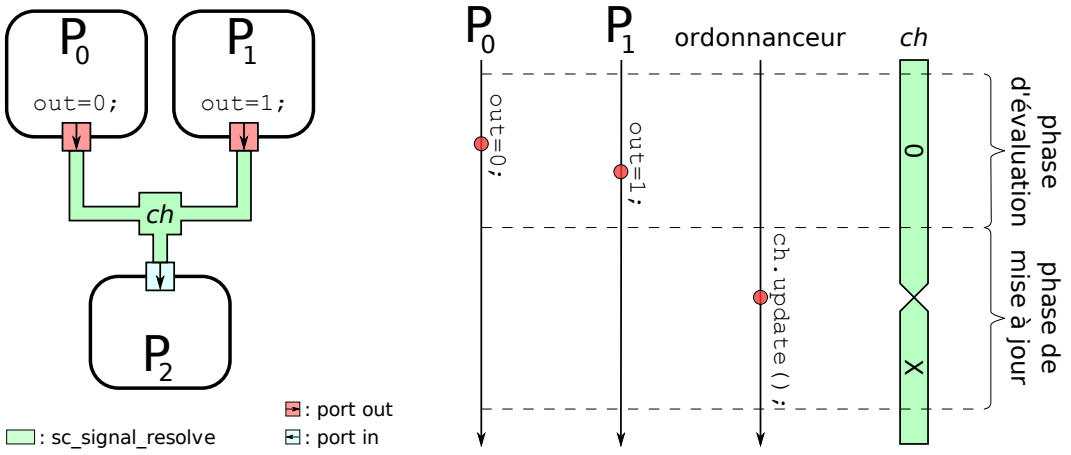


FIGURE 2.10 – *Exemple d'écritures concurrentes dans un canal* `sc_signal_resolve`

De manière générale, les delta-cycles sont utilisés par les canaux SystemC pour que leurs valeurs de sortie restent constantes au cours d'une phase d'évaluation. Cette utilisation des delta-cyles permet notamment de respecter les hypothèses de discrétisation du temps des circuits synchrones. Elle ne permet cependant pas de modéliser correctement le comportement d'un circuit asynchrone (cf. paragraphe 2.1.3.1).

2.1.2.2 Le Simulateur SystemC

Le simulateur SystemC est un simulateur à événements ayant un ordonnanceur collaboratif (non préemptif). Un processus ne peut donc être interrompu au cours de son exécution que s'il appelle explicitement sa méthode `wait` et qu'il possède son propre contexte d'exécution. Comme l'illustre la figure 2.11, la simulation d'un modèle SystemC se décompose en trois grandes étapes qui sont détaillées dans les paragraphes suivants.

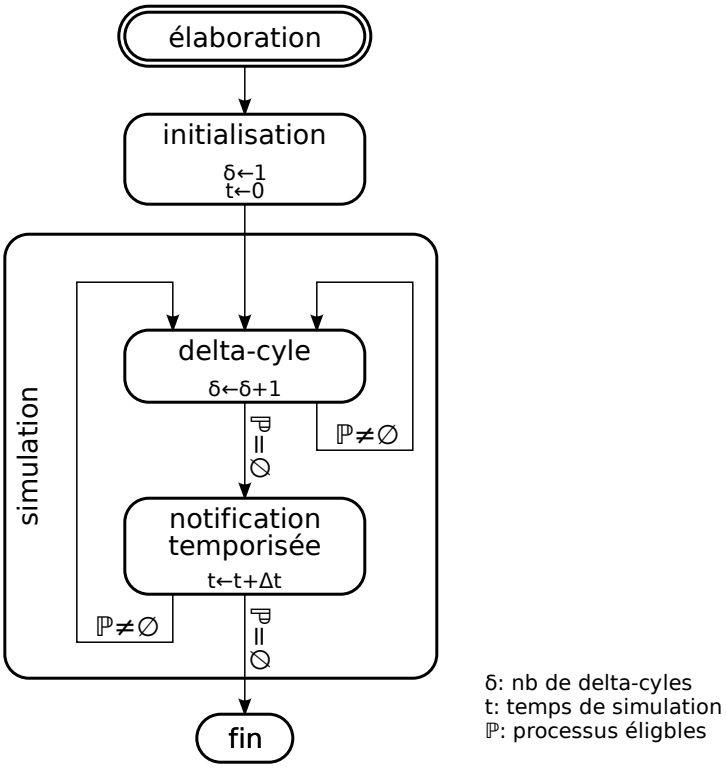


FIGURE 2.11 – *Automate de l'ordonnanceur SystemC*

La Phase d'Élaboration

La phase d'élaboration est la première étape d'une simulation. Elle est chargée de créer et de connecter les différentes structures de données d'un modèle SystemC (e.g. modules, canaux, ...). Durant cette phase sont aussi créées et initialisées les différentes structures de données internes du simulateur (e.g. liste des processus, liste des canaux, ...). Parmi les structures de données servant à la modélisation d'un système, seuls les processus peuvent être créés dynamiquement. Toutes les autres structures de données doivent être créées durant la phase d'élaboration.

La Phase d'Initialisation

L'objectif de la phase d'initialisation est de préparer les processus pour la phase de simulation en mettant à jour leurs états. En effet, durant cette phase, tous les processus sont initialisés à l'état éligible pour être exécutés au début de la phase de simulation. Si un processus ne veut pas être exécuté au début de la phase de la simulation, il doit le signaler au cours de la phase d'élaboration à l'aide de la méthode `dont_initialize`. L'état d'un

processus ayant appelé cette méthode est alors initialisé à l'état endormi pendant la phase d'initialisation.

La Phase de Simulation

La phase de simulation simule le comportement d'un modèle SystemC en exécutant de manière pseudo concurrente ses différents processus. Elle est constituée de séquences de delta-cyles qui sont séparées par des phases de déclenchement des notifications temporisées. La phase de simulation s'arrête lorsqu'il n'y a plus de processus éligible et d'événement à notifier.

Pour gérer l'avancement du temps de simulation, le simulateur SystemC possède un échéancier des notifications temporisées à déclencher. Cet échéancier est une liste de notifications d'événements temporisées qui est triée en fonction de leurs dates de déclenchements. Lorsqu'il n'y a plus de processus à exécuter après une séquence de delta-cycle, le simulateur SystemC avance le temps de simulation à la date de déclenchement la plus proche de son échéancier. Il effectue ensuite les déclenchements de toutes les notifications temporisées enregistrées à cette date.

Au cours de la phase d'évaluation d'un delta-cycle les processus sont exécutés séquentiellement dans un ordre qui dépend de l'implémentation de la bibliothèque SystemC utilisée (cf. paragraphe 2.1.1.3). Des processus peuvent aussi être ajoutés à l'ensemble des processus éligibles durant la phase d'évaluation grâce aux notifications immédiates.

2.1.2.3 La Génération de Traces

Le format de trace VCD [IEE04] (*Value Change Dump*) est le seul format de trace supporté par le standard SystemC de base. Ce format de trace enregistre les changements de valeur d'un canal au cours du temps. Le standard SystemC définit une interface de programmation ou API (*Application Programming Interface*) complète permettant de choisir les canaux que l'on veut tracer et le fichier dans lequel on veut enregistrer leurs activités durant une simulation. Ce fichier de trace peut ensuite être utilisé avec des outils tels que GTKWave [2], SimVision, ... qui permettent de visualiser l'activité des canaux sous forme de chronogrammes.

Le principal avantage de ce format de trace est d'être supporté par la majorité des outils de conception de circuits intégrés. Il n'est cependant pas adapté pour enregistrer les activités de n'importe quel modèle SystemC (cf. paragraphe 2.1.3.4).

2.1.3 Les Limitations

Une première limitation de la bibliothèque SystemC pour modéliser des circuits asynchrones est de posséder des canaux de communication qui ne sont pas adaptés pour modéliser ces circuits. En effet, ces canaux de communication emploient des mécanismes de synchronisation globale qui ne respectent pas les propriétés de base des protocoles employés par les canaux de communication matériels des circuits asynchrones.

Une deuxième limitation de cette bibliothèque est de ne pas posséder des structures de choix non déterministes qui sont néanmoins nécessaires pour modéliser fidèlement le comportement des arbitres d'un circuit asynchrone.

[2] http://home.nc.rr.com/gtkwave

Une troisième limitation concerne les types de données disponibles en SystemC qui ne correspondent pas à ceux employés généralement par les circuits asynchrones insensibles aux délais.

Une dernière limitation est due aux choix d'implémentation effectués par l'OSCI pour réaliser le simulateur de référence et au format de trace VCD qui ne sont pas adaptés pour valider des modèles SystemC de circuits asynchrones.

2.1.3.1 La Modélisation des Communications

Comme les processus SystemC, chaque bloc d'un circuit asynchrone s'exécute de façon concurrente et communique avec d'autres processus via des canaux de communication. Cependant, l'utilisation de la technique *request-update* par les canaux de communication SystemC (pour synchroniser les valeurs de leurs sorties) ne permet pas de modéliser correctement des circuits asynchrones. Avec cette méthodologie, les communications se déroulant au cours de la même phase d'évaluation d'un delta-cycle, sont implicitement synchronisées sur la fin de ce delta-cyle. Si cette méthode de synchronisation permet de respecter les hypothèses temporelles d'un circuit synchrone (les delta-cyles jouant le rôle d'une horloge logique), elle n'est cependant pas adaptée à un circuit asynchrone qui ne fait pas d'hypothèse temporelle aussi forte sur les délais de propagation.

Le problème de la modélisation d'un circuit asynchrone avec des canaux SystemC de base est illustré par la figure 2.12. Cet exemple montre comment un arbitre peut être modélisé en SystemC à l'aide de canaux `sc_signal`. Dans cet exemple le processus P_0 (respectivement P_1) envoie une requête au processus `arbitre` via le canal ch_0 (respectivement ch_1). Les écritures dans ces canaux provoquent les appels de leurs méthodes `update`. Le rôle de cette méthode est de mettre à jour la valeur de son canal et d'effectuer une notification retardée de l'événement associé à ce canal (non représenté sur la figure). Cette notification retardée a pour effet de réveiller le processus `ARBITRE` endormi précédemment en appelant la méthode `wait`. Ce modèle d'un arbitre ne respecte donc pas la propriété d'insensibilité aux délais des circuits asynchrones insensibles aux délais (DI et QDI) car il suppose que les délais des canaux ch_0 et ch_1 sont identiques.

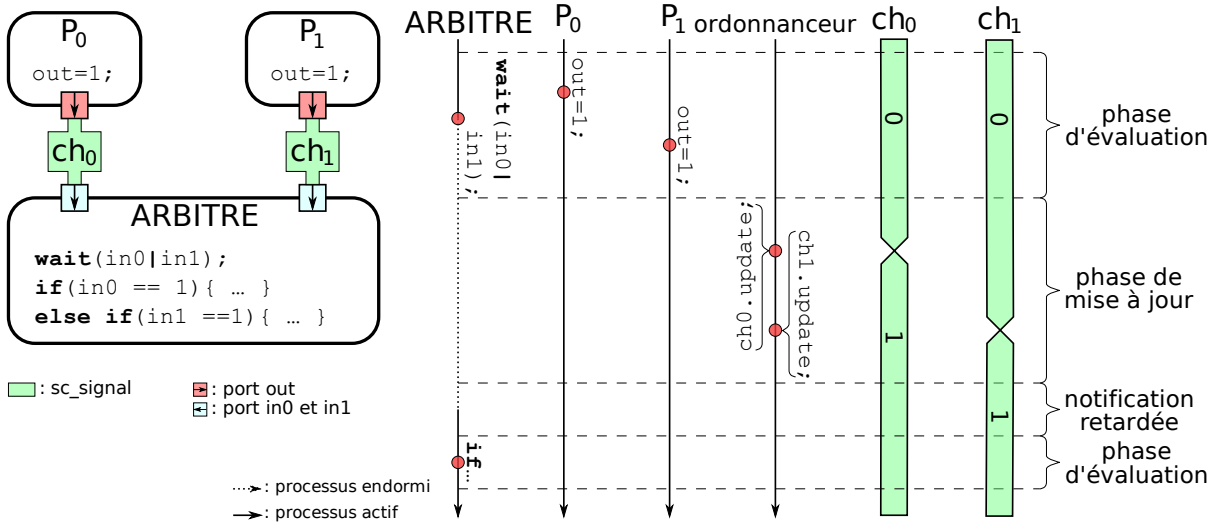


FIGURE 2.12 – *Modèlisation d'un arbitre en SystemC*

Les canaux de communication reliant les blocs matériels d'un circuit asynchrone insen-

sible aux délais sont les seuls éléments permettant à ses différents blocs de se synchroniser. Un modèle SystemC d'un circuit asynchrone insensible aux délais doit donc employer des canaux de communication SystemC qui respectent les propriétés de base des protocoles de communication utilisés par les canaux de communication matériels d'un circuit asynchrone insensible aux délais. Les propriétés de base de ces protocoles de communication sont d'être synchrones (bloquants) et sans mémoire.

Ces propriétés sont, par exemple, utilisées lorsqu'un bloc matériel d'un circuit asynchrone insensible aux délais veut accéder séquentiellement à un bloc matériel A puis à un autre bloc matériel B. Avec un canal de communication respectant les propriétés précédentes, la fin de la communication avec A est une condition suffisante pour pouvoir accéder à B. Si le canal de communication employé ne respecte pas toutes ces propriétés alors des mécanismes de synchronisation supplémentaires doivent être employés pour assurer le séquençage des accès à A puis à B. L'emploi de canaux de communication tels qu'un `sc_signal` (qui n'est pas bloquant) ou d'un `sc_fifo` (qui n'est pas sans mémoire) n'est donc pas adapté pour modéliser un circuit asynchrone.

La solution à ces problèmes de modélisation des communications, développée au cours de cette thèse, a été de définir des nouveaux canaux de communication respectant les propriétés des circuits asynchrones insensibles aux délais (cf. paragraphe 3.1.3).

2.1.3.2 La Modélisation des Structures de Choix

Une spécificité des circuits asynchrones insensibles aux délais est d'être composés d'arbitres parfaitement fiables mais qui n'ont pas un comportement déterministe : le choix d'une requête parmi un ensemble de requêtes concurrentes est effectué de manière non déterministe par un arbitre asynchrone (cf. paragraphe 1.2.3.2). Pour modéliser correctement des arbitres asynchrones, il est donc nécessaire de disposer de structure de choix ayant un comportement non déterministe.

La bibliothèque SystemC de base ne permet pas de modéliser facilement des structures de choix non déterministes. Cette bibliothèque ne dispose en effet d'aucune autre structure de choix que celles du C++ (`if...then...else` et `switch...case`) qui ne sont pas adaptées pour les raisons suivantes :

- leurs conditions sont évaluées séquentiellement,
- les instructions, protégées par une de ces conditions, sont exécutées dès que cette condition a été vérifiée.

En outre, l'ordre d'évaluation, imposé par ces structures de choix, modélise implicitement un arbitrage à priorité qui n'est pas toujours nécessaire.

Une solution envisageable pour modéliser une structure de choix non déterministe en C++ est de la modéliser à l'aide de l'instruction `goto`. Cette solution n'est cependant pas satisfaisante car il est difficile, voir impossible, d'utiliser TAST pour synthétiser un modèle SystemC qui utilise des `goto`. De plus, l'utilisation de l'instruction `goto` est d'une manière générale fortement déconseillée [Dij79].

Pour être en mesure de modéliser fidèlement le comportement d'un arbitre asynchrone en SystemC, de nouvelles structures de choix ont été ajoutées au C++ (cf. paragraphe 3.1.5).

2.1.3.3 Les Types de Données

Les valeurs entières utilisées par les circuits intégrés ont souvent des tailles qui sont supérieures à celles des entiers C++. Pour pallier cette limitation, SystemC définit les types d'entiers `sc_signed` et `sc_unsigned` qui peuvent avoir une taille arbitraire. Cela étant, ces entiers ne sont pas adaptés pour spécifier des modèles synthétisables de circuits asynchrones insensibles aux délais car ils ne permettent pas de prendre en compte les propriétés spécifiques des codes qui sont utilisés par ces circuits pour représenter les valeurs de leurs données (cf paragraphe 1.1.2.2). L'utilisation du code *multi-rails* a notamment pour effet de rajouter la valeur `null` et de permettre d'utiliser une base différente de deux pour représenter des entiers. De plus, pour synthétiser les canaux de communication et les registres d'un circuit asynchrone, il est nécessaire de connaître leurs tailles et donc de connaître précisément le type des données qu'ils manipulent.

Pour avoir la possibilité de synthétiser des circuits asynchrones insensibles aux délais à partir de modèles SystemC, de nouveaux types de données (entier, booléen, ...) basés sur le code *multi-rails* ont été développés (cf. paragraphe 3.1.7).

2.1.3.4 La Validation par Simulation

Pour vérifier la propriété d'insensibilité aux délais du modèle SystemC d'un circuit asynchrone insensible aux délais, il faut tester tous les ordonnancements possibles de ses processus. Par exemple, les différents entrelacements possibles de deux processus accèdant à une ressource partagée telle qu'un arbitre doivent pouvoir être simulés. Un des avantages de la spécification de l'ordonnanceur du simulateur SystemC est d'être non déterministe et donc de permettre en théorie de simuler les différents entrelacements des processus d'un modèle SystemC. Néanmoins, la plupart des implémentations du simulateur SystemC ont un ordonnanceur déterministe et notamment l'implémentation de référence fournie par l'OSCI. Afin de résoudre cette limitation, un correctif (de l'anglais *patch*) du simulateur OSCI a été développé (cf. paragraphe 4.2.3).

Les facilités de traçage standard de SystemC sont basées sur le format de trace VCD. Pour qu'une action de communication soit enregistrée par ce format de trace, il est nécessaire que la valeur échangée soit différente de la valeur courante du canal et que ce changement de valeur ait une durée au moins égale à un delta-cycle. La figure 2.13 montre que ce format de trace n'est pas adapté à un canal utilisant des notifications immédiates pour synchroniser ses communications car il peut en effectuer plusieurs au cours d'un seul delta-cycle. Pour permettre d'enregistrer correctement l'activité de tels canaux de communication, de nouvelles fonctionnalités de traçage basées sur un modèle de temps distribué ont été réalisées (cf. chapitre 4).

2.2 La Bibliothèque SystemC/TLM

La bibliothèque TLM est une extension de SystemC prévue pour modéliser des systèmes à haut niveau d'abstraction. L'objectif principal de cette bibliothèque est de permettre la modélisation rapide et efficace des média de communication (réseau sur puce, bus partagé, ...) employés par les systèmes sur puce [Par02]. Comme SystemC, le développement et la standardisation de cette bibliothèque sont assurés par l'OSCI. La première [RSPF05] version de cette bibliothèque définit principalement une méthodologie de modélisation en spécifiant des interfaces de communication et des patrons de conception.

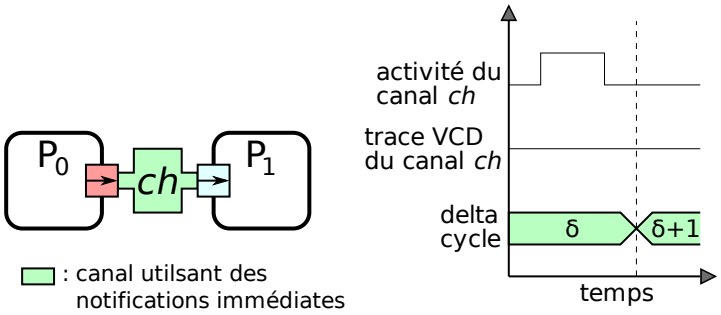


FIGURE 2.13 – *Trace VCD d'un canal utilisant des notifications immédiates*

Une deuxième version [OSC05] de la bibliothèque TLM a été récemment rendue disponible sur le site de l'OSCI. Les principaux apports de cette nouvelle version sont de fixer les types de données qui peuvent être échangés et d'introduire une nouvelle classe de modèle temporisé (les modèles faiblement temporisés ou *loosely timed*).

Dans le cadre de cette thèse, seule la première version de la bibliothèque TLM a été étudiée en détails afin de déterminer si elle est adaptée pour concevoir des circuits asynchrones. Les deux types de modèles TLM-PV (*TLM-Programmer View*) et TLM-PVT (*TLM-Programmer View plus Timing*), proposés par STMicroelectronics [Ghe06], ont notamment été évalués. Dans un premier temps, cette évaluation a consisté à étudier les cadres d'application (de l'anglais *framework*) TAC (*Transaction Accurate Communication*) et BASIC_PVT. Dans un deuxième temps, des modèles d'Octagons TLM-PV ont été réalisés pour définir si cette famille de modèles pouvait être utilisée efficacement pour concevoir des circuits asynchrones.

2.2.1 Les Concepts

La bibliothèque SystemC/TLM est constituée d'un ensemble d'interfaces de communication et de patrons de conception qui permettent de modéliser un système à différents niveaux d'abstraction. Comme l'illustre la figure 2.14, le niveau d'abstraction d'un modèle TLM est généralement caractérisé en fonction des deux paramètres suivants :

- la granularité des données qui détermine la précision et/ou la taille des données véhiculées par une transaction,
- la précision temporelle qui définit la fidélité du comportement temporel d'un modèle par rapport au système réel.

Un modèle TLM appartenant au niveau PV ne contient pas d'information temporelle liée à l'implémentation (logicielle ou matérielle) du système qu'il modélise. Ces modèles peuvent donc être disponibles très tôt dans un flot de conception. De plus, cette abstraction des détails d'implémentation, associée à la faible granularité (i.e. gros grain) des données échangées, permet d'obtenir des plateformes de simulation très efficaces. Si ces modèles sont parfaitement adaptés pour valider fonctionnellement un système, ils ne sont cependant pas adaptés pour évaluer ses performances.

Pour résoudre cette limitation des modèles PV, la bibliothèque TLM dispose des modèles approximativement temporisés PVT. Ces modèles contiennent des annotations temporelles qui permettent d'estimer les performances d'un système. En outre, un modèle PVT décompose généralement une transaction d'un modèle PV en un ensemble de sous transactions afin de mieux prendre en compte les mécanismes de synchronisation em-

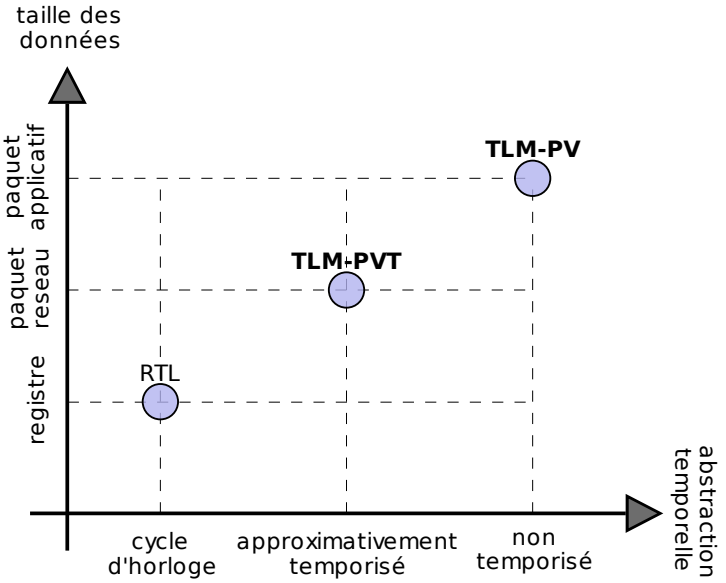


FIGURE 2.14 – *Classification des niveaux d'abstraction*

ployés par le système réel. Ceci permet par exemple de pouvoir modéliser plus fidèlement le comportement temporel d'un système employant des mécanismes de transmission par rafales.

Comme l'illustre la figure 2.15, la structure générique d'un modèle TLM est composée de modules initiateurs et cibles qui sont interconnectés via un ou plusieurs modules d'interconnexion. Un module initiateur est un module dont un processus initie une transaction vers un module cible. Une transaction est équivalente à un appel de procédure distante (de l'anglais *remote procedure call*) où un processus d'un module initiateur appelle une méthode d'un module cible via un ou plusieurs modules d'interconnexion (un module initiateur et un module cible peuvent aussi être directement reliés). Ces modules sont reliés par des ports initiateurs et des ports cibles qui sont chargés de propager les appels de méthodes d'une transaction. Ces ports peuvent aussi convertir les appels de méthodes en fonction des modules qu'ils connectent. Par exemple, le port initiateur `ip` du module initiateur i_1 convertit l'appel de la méthode `write` du module cible c_1 en un appel à la méthode `transport` du module d'interconnexion. Ce mécanisme de conversion des appels de méthodes permet de rendre les modules d'interconnexion indépendants des modules initiateurs et cibles qu'ils connectent. Un module d'interconnexion peut être ainsi facilement utilisé pour connecter des modules initiateurs et cibles utilisant des appels de méthodes différents.

Un des principaux objectifs de la bibliothèque TLM est de standardiser les méthodes employées par un module d'interconnexion pour propager des transactions afin de pouvoir facilement le remplacer et le réutiliser. La bibliothèque TLM définit donc un ensemble d'interfaces de communication qu'un module d'interconnexion doit implémenter pour propager les transactions qui lui parviennent. De plus, cette bibliothèque présente aussi des patrons de conception à utiliser pour obtenir un modèle TLM efficace et facilement réutilisable.

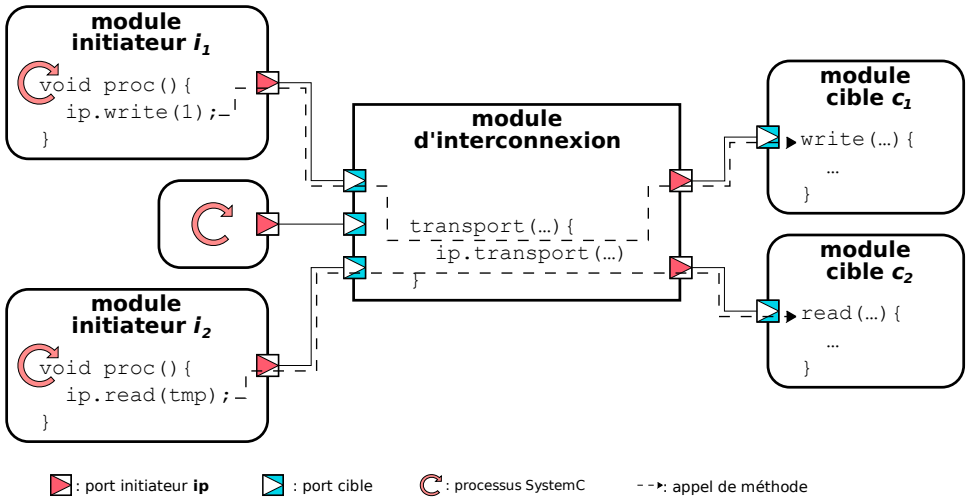


FIGURE 2.15 – *Exemple de modèle SystemC/TLM*

2.2.1.1 Les Interfaces

La bibliothèque SystemC/TLM définit différentes interfaces de communication adaptées aux besoins des différents niveaux d'abstraction auxquels un système peut être modélisé. Elle distingue notamment les deux types d'interfaces suivantes :

- les interfaces « bidirectionnelles » qui permettent d'envoyer et de recevoir des données en une seule transaction,
- les interfaces « unidirectionnelles » qui autorisent une transaction soit à envoyer soit à recevoir des données.

L'interface de communication `tlm_transport_if` est la principale interface bidirectionnelle de la bibliothèque TLM (cf. programme 2.2). Cette interface est généralement utilisée par les modèles TLM-PV car elle est bien adaptée pour acheminer les transactions employées par ces modèles telles que des écritures bloquantes ou des lectures bloquantes. Cette interface est composée de la seule méthode générique abstraite `transport` . Cette méthode prend en paramètre la requête d'une transaction émise par un module initiateur et renvoie la réponse du module cible.

```
template < typename REQ , typename RSP >
class tlm_transport_if : public virtual sc_interface
{
  public:
    virtual RSP transport( const REQ & ) = 0;
};
```

PROGRAMME 2.2 – *Interface bidirectionnelle* ***transport***

Les deux interfaces de communication `tlm_blocking_put_if` et `tlm_blocking_get_if` sont les interfaces de communication unidirectionnelle essentielles de la bibliothèque TLM (cf. programme 2.3). Ces interfaces de communication définissent les deux méthodes génériques abstraites `put` et `get` qui sont respectivement employées pour envoyer et recevoir des données. Ces méthodes sont bien adaptées pour acheminer les transactions des modèles TLM-PVT car elles permettent de prendre en compte les propriétés temporelles qui sont associées à ces transactions telles que les notions de pipeline ou de transmission

par rafales.

```
template < typename T >
class tlm_blocking_get_if : public virtual sc_interface
{
    public:
        virtual T get( void ) = 0;
};

template < typename T >
class tlm_blocking_put_if : public virtual sc_interface
{
    public:
        virtual void put( const T &t ) = 0;
};
```

PROGRAMME 2.3 – *Interfaces unidirectionnelles* `put` *et* `get`

2.2.1.2 Les Patrons de Conception

La bibliothèque TLM définit une méthode de modélisation segmentée en trois couches qui repose sur les deux patrons de conception « *intiator_port* » et « *slave_base* ».

La couche « utilisateur » contient la partie comportementale du système (processeur, mémoire, DSP, ...). Pour communiquer, ces composants appellent les méthodes des modules cibles qui sont spécifiques au système modélisé.

La couche « transport » est constituée des éléments de communication du système. Ces éléments utilisent les méthodes des interfaces de communication définies par la bibliothèque TLM (`transport`, `put`, `get`, ...) pour communiquer entre eux et avec les éléments de la couche protocole.

La couche « protocole » a pour rôle d'interfacer la couche utilisateur et la couche transport en utilisant les deux patrons de conception *initiator_port* et *slave_base*.

Un avantage de cette méthode de modélisation est de rendre indépendante la couche utilisateur de la couche transport. Un autre avantage de cette méthode de modélisation est d'améliorer la vitesse de simulation en diminuant le nombre de processus et le nombre de changements de contexte pour respectivement modéliser et simuler un système. Par exemple, une action de communication entre deux modules TLM nécessite un seul processus et s'effectue sans changement de contexte, alors qu'une action de communication entre deux modules SystemC utilisant un `sc_signal` requiert deux processus et au moins un changement de contexte.

Le Patron de Conception *initiator_port*

Pour appeler une méthode d'un module cible, le processus d'un module initiateur utilise un port initiateur qui implémente le patron de conception *initiator_port*. Ce patron de conception consiste à définir un port initiateur qui convertit les appels aux méthodes des modules cibles en appels aux méthodes des interfaces de communication de la bibliothèque TLM.

La figure 2.16 présente un exemple de port initiateur *IP* qui respecte le patron de conception *initiator_port*. Ce port convertit l'appel à sa méthode `read` appartenant à la couche utilisateur en un appel à la méthode `transport` de la couche transport. Dans un

premier temps, il encapsule dans la requête `req` l'adresse de la donnée à lire et l'emplacement mémoire où la valeur de cette donnée doit être écrite. Dans un deuxième temps il appelle la méthode `transport` du port cible auquel il est connecté en utilisant le patron de conception IMC (cf. paragraphe 2.1.1.4).

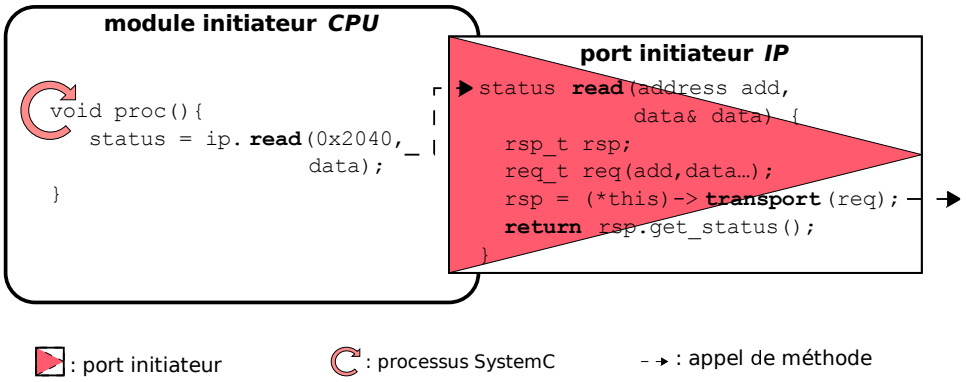


FIGURE 2.16 – *Exemple d'un port TLM respectant le patron de conception iniator_port*

Le Patron de Conception *slave_base*

L'objectif de ce patron de conception est dual avec celui du patron de conception *initiator_port*. Son principe de conception est de définir une classe de base abstraite pour les modules cibles. La tâche de cette classe abstraite est de convertir les appels à ses méthodes de la couche transport en appels aux méthodes de la couche utilisateur de ses classes filles.

Un exemple d'implémentation (diagramme de classe UML) et d'utilisation du patron de conception *slave_base* est représentés par la figure 2.17. La classe `ram` représentant le module cible *RAM* est en fait un canal de communication hiérarchique car elle hérite de la classe `sc_module` et des deux interfaces de communications `tlm_transport_if` et `transaction_if`. Elle possède un port cible `tp` qui est un export chargé de propager les appels à sa méthode `transport` provenant du port initiateur connecté à `tp`. Le corps de la méthode `transport` (appartenant à la couche transport) est implémenté par la classe de base abstraite `passive_component` qui convertit son appel en un appel à la méthode `send` (appartenant à la couche utilisateur) de la classe `ram`.

2.2.2 Les Cadres d'Application TAC et BASIC_PVT

La bibliothèque TLM de SystemC et les circuits asynchrones insensibles aux délais ont de nombreuses propriétés de base communes (e.g. hypothèse temporelle, mode de communication, ...). Ces points communs font de cette bibliothèque une candidate potentiellement bien adaptée pour modéliser des circuits asynchrones insensibles aux délais.

Pour évaluer si cette bibliothèque est réellement bien adaptée, les deux cadres d'application TAC et BASIC_PVT ont été étudiés en détails. Ces deux cadres d'application reposant sur la bibliothèque TLM permettent de modéliser un système à un niveau d'abstraction élevé. L'objectif principal de ces deux cadres d'application est de pouvoir disposer d'une plateforme de simulation efficace tôt dans le développement d'un circuit complexe tel qu'un Système sur Puce.

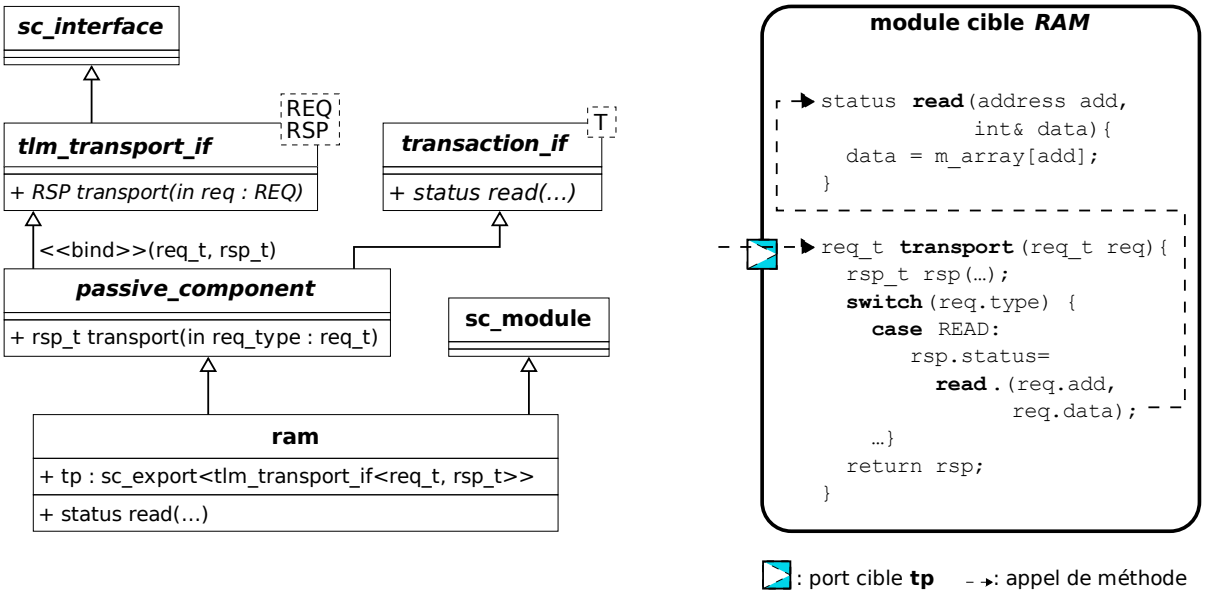


FIGURE 2.17 – *Exemple d'un module cible respectant le patron de conception slave_base*

2.2.2.1 Le Cadre d'Application TAC

Le cadre d'application TAC [Gre05] est un projet *open-source* développé par STMicroelectronics qui est disponible sur le site de GreenSocs [3]. Ce cadre d'application fournit un ensemble de classes C++ qui permet de modéliser un système en SystemC TLM au niveau PV.

Comme l'illustre la figure 2.18, la méthode de modélisation employée par ce cadre d'application respecte les patrons de conception vus précédemment. Cette méthode de modélisation consiste à définir les éléments de la couche utilisateur d'un système à modéliser et à les relier avec le routeur TAC. Ces éléments de la couche utilisateur communiquent entre eux via des transactions de type accès mémoire basées sur les méthodes `read` et `write`.

Le routeur TAC est chargé d'acheminer les transactions de la couche utilisateur à l'aide de la méthode `transport` de l'interface bidirectionnelle TLM. Ce routeur représente un modèle idéal d'interconnexion où les transactions sont acheminées sans contention et sans délai (modèle non temporisé). Les transactions parvenant à ce routeur sont aiguillées en utilisant une table de routage (appel de la méthode `decode`). Cette table de routage définit les ports initiateurs à utiliser pour propager une transaction en fonction des plages d'adresses accessibles.

Un routeur TAC peut aussi être utilisé comme medium d'interconnexion d'un modèle approximativement temporisé. Comme l'illustre la figure 2.18, une annotation temporelle peut être définie dans le corps de la méthode `transport` du routeur TAC en utilisant la méthode `pv_wait` . Cependant, ces annotations ne permettent pas d'obtenir des modèles temporisés très précis. Il est, par exemple, difficile de représenter le phénomène de contention qui peut survenir dans un réseau sur puce ou de modéliser le mécanisme de transmission par rafales traditionnellement employé par un bus partagé.

[3] http://www.greensocs.org

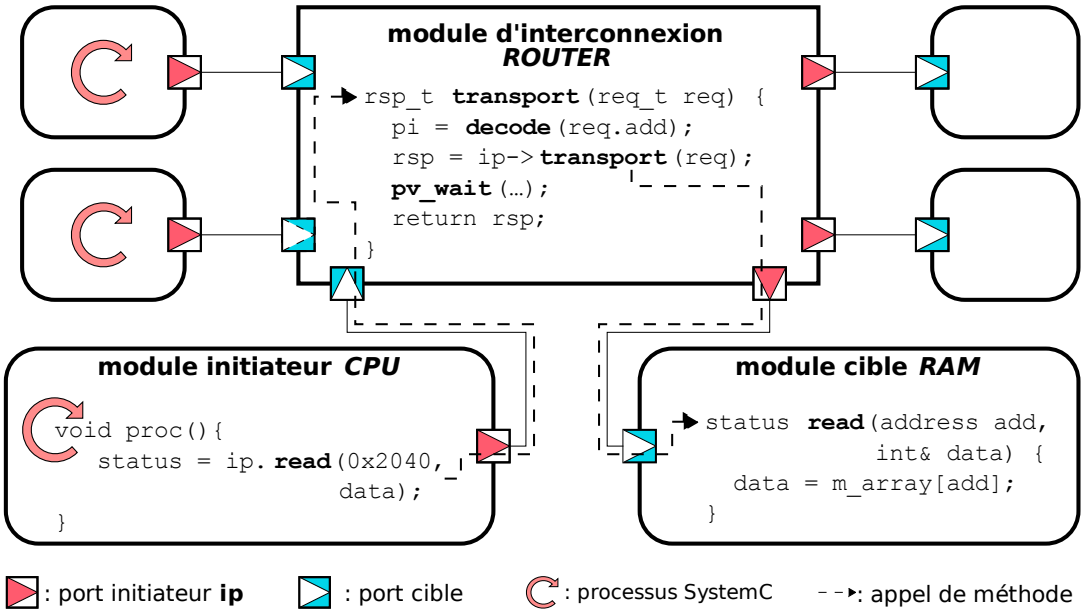


FIGURE 2.18 – *Exemple de modèle utilisant le cadre d'application TAC*

2.2.2.2 Le Cadre d'Application BASIC_PVT

Le cadre d'application propriétaire BASIC_PVT est prévu pour modéliser des systèmes en SystemC/TLM au niveau PVT. Ce cadre d'application est une version allégée du cadre d'application ST_BUS développé par STMicroelectronics et qui en reprend la méthode de modélisation. Cette dernière repose sur l'utilisation de la méthode `put` de l'interface unidirectionnelle TLM `tlm_blocking_put_if` (cf. figure 2.19). Les primitives de communication de la couche utilisateur du cadre d'application BASIC_PVT (`read_req`, `read_rsp`, `write_req` et `write_rsp`) sont similaires à celles du TAC (`read` et `write`). Le routeur employé pour interconnecter les éléments d'un modèle BASIC_PVT est aussi très proche de celui utilisé par le cadre d'application TAC. Comme pour ce dernier, les transactions parvenant à un routeur BASIC_PVT sont propagées sans contention et sont routées en utilisant une table de routage.

La figure 2.19 illustre comment une écriture mémoire est accomplie avec le cadre d'application BASIC_PVT. Le principe de base est similaire à celui du TAC (appel de méthode distante), à la différence que chacune de ses transactions est divisée en deux parties distinctes : une partie « requête » et une partie « réponse ». Le principal intérêt de diviser une transaction en plusieurs sous transactions est de pouvoir modéliser plus fidèlement le comportement d'un système. Il est, par exemple, facile de représenter le mécanisme de transmission par rafales en autorisant l'envoi de plusieurs transactions de requête pour une transaction de réponse.

Un désavantage important de la méthode de modélisation employée par le cadre d'application BASIC_PVT par rapport à celle du cadre d'application TAC est d'être beaucoup plus complexe et de produire des modèles simulables beaucoup moins efficaces. Un modèle BASIC_PVT requiert beaucoup plus de processus qu'un modèle TAC équivalent et simuler les transactions d'un modèle BASIC_PVT nécessite beaucoup plus de changements de contextes de processus que de simuler les transactions d'un modèle TAC équivalent.

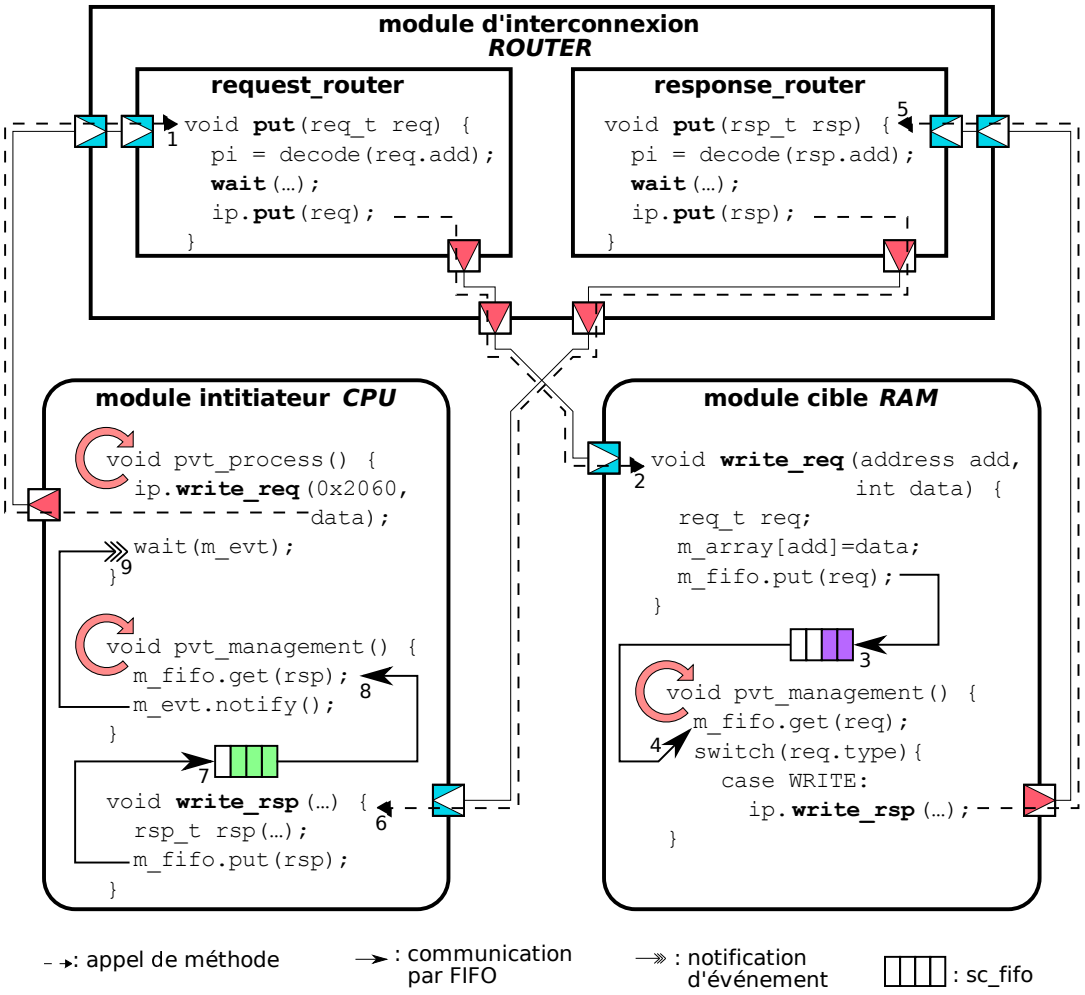


FIGURE 2.19 – *Exemple de modèle utilisant le cadre d'application BASIC_PVT*

2.2.3 Les Avantages et les Limitations

Le principal avantage des modèles TLM-PV pour modéliser des circuits asynchrones insensibles aux délais est de partager les mêmes hypothèses temporelles. En effet, un modèle TLM-PV (tel que le routeur du cadre d'application TAC) est un modèle purement fonctionnel qui ne fait aucune hypothèse sur les délais. De plus, ces modèles utilisent généralement des mécanismes de synchronisation locale et n'ont donc pas recours à des mécanismes de synchronisation globale tels que les delta-cycles. Un dernier avantage des modèles TLM, en général, est d'utiliser un mode de communication proche de celui employé par les circuits asynchrones. En effet les modes de communication par transaction et par canal de communication synchrone sont assez proches.

Cependant, les abstractions importantes (à l'exception des abstractions temporelles) employées par les modèles TLM ne permettent pas de pouvoir les utiliser directement comme des modèles synthétisables. Par exemple, les routeurs des cadres d'application TAC et BASIC_PVT ne contiennent aucun détail d'implémentation permettant de pouvoir les synthétiser. En effet, pour synthétiser un medium d'interconnexion tel qu'un réseau sur puce, il est nécessaire de préciser des informations de base telles que sa topologie, sa technique de commutation, son algorithme de routage, ... qui ne sont pas présents dans ces modèles de routeur.

La principale différence entre les deux modèles de routeur TAC et BASIC_PVT est que ce dernier permet de mieux modéliser les aspects temporels d'un médium d'interconnexion. Ce dernier routeur est divisé en deux parties (cf. figure 2.19), requête et réponse qui sont chargées d'acheminer respectivement les transactions de requête et d'acquittement. Si ce découpage en deux parties permet de mieux représenter les aspects temporels d'un système, il ne procure cependant pas d'information supplémentaire concernant son implémentation réelle (topologie, routage, ...). Cette méthode de modélisation est donc beaucoup plus complexe que celle du TAC et n'apporte pas d'information utile pour la synthèse d'un réseau sur puce asynchrone. Cette thèse s'est donc limitée à déterminer si la méthode de modélisation employée par le cadre d'application TAC pouvait être adaptée pour réaliser des modèles synthétisables avec TAST.

2.2.4 Modélisation d'un Octagon

Dans le cadre de cette thèse, seule la méthode de modélisation de la couche transport du cadre d'application TAC a été étudiée pour déterminer si elle pouvait être adaptée pour modéliser des circuits synthétisables avec TAST. En effet, parmi les trois couches du cadre d'application TAC, seuls les composants de la couche transport utilisent des interfaces de communication standardisées. De plus, les composants appartenant à cette couche sont les composants qui sont principalement ciblés par cette thèse car ils modélisent les éléments d'interconnexion d'un système tel qu'un réseau sur puce.

La principale limitation à l'utilisation du routeur TAC comme modèle d'entrée pour TAST est son niveau d'abstraction trop élevé. Une solution pour pallier ce problème est d'utiliser un modèle d'interconnexion moins générique et plus proche d'une implémentation réelle. Pour évaluer l'intérêt de cette approche, deux modèles SystemC/TLM du réseau sur puce Octagon (cf. paragraphe 1.2.4.1) ont été réalisés. Une des raisons qui a dirigé le choix d'utiliser ce réseau sur puce est qu'il supporte la commutation de circuit et de ver de terre. Cette propriété a notamment permis d'observer que l'utilisation de la méthode `transport` était plus adaptée pour modéliser des réseaux sur puce à commutation de circuits.

Pour vérifier le comportement de ces modèles SystemC/TLM de NoC Octagon, une plateforme de simulation reposant sur le cadre d'application TAC a été définie. Pour faciliter notre étude, le cadre d'application TAC a été simplifié en ne conservant que le code nécessaire pour générer du trafic (e.g. suppression du code pour la génération de trace, ...) et en modifiant les signatures des méthodes `read` et `write` de la couche utilisateur. Celles-ci ne prennent en paramètres que la donnée à lire ou à écrire et l'identifiant du nœud de routage (i.e. un entier entre 0 et 7) auquel est directement connecté le module cible. En outre, la classe de base abstraite des modules cibles et la classe des ports initiateurs ont dûes être modifiées en fonction de cette nouvelle interface. Finalement, un module initiateur qui génère du trafic et un module cible qui en consomme ont été réalisés.

2.2.4.1 Modèle Sans Contention

Comme le routeur TAC, le premier modèle de réseau sur puce d'Octagon utilise la méthode bidirectionnelle TLM `transport` pour acheminer les transactions qui lui parviennent. Contrairement au routeur TAC, la topologie et l'algorithme de routage employés par le réseau sur puce Octagon sont explicitement modélisés.

Comme l'illustre la figure 2.20, ce routeur est constitué d'un ensemble de nœuds de routage qui sont interconnectés en respectant la topologie de l'Octagon. Cette figure montre aussi que la méthode `transport` de ces nœuds de routage utilise bien l'algorithme de routage de l'Octagon pour aiguiller les transactions qui lui parviennent.

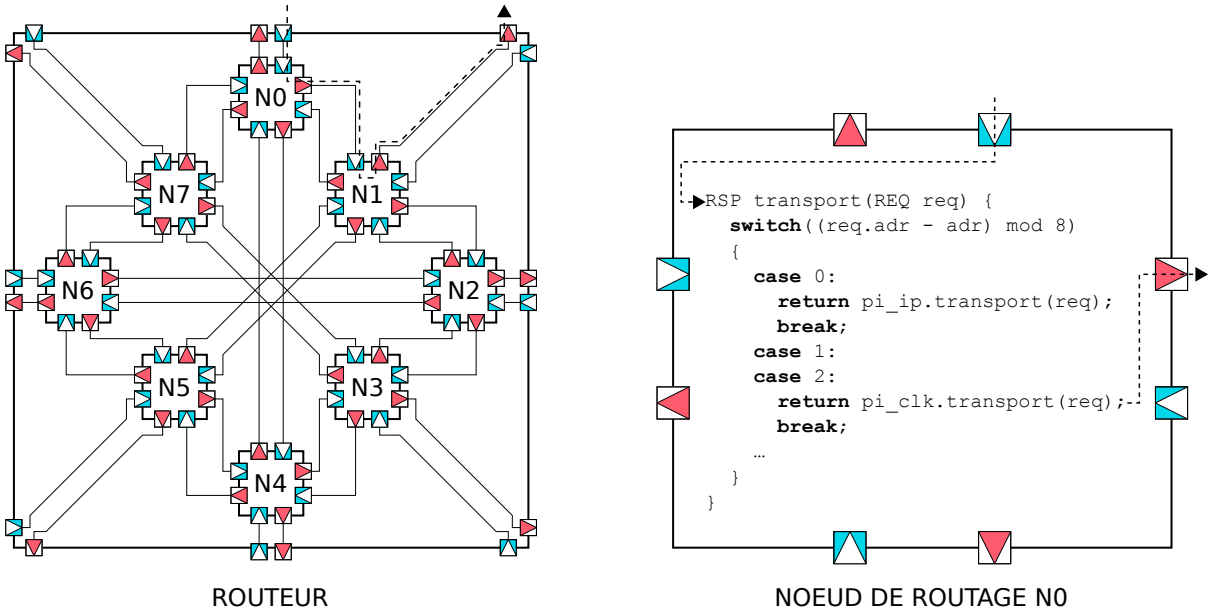


FIGURE 2.20 – *Routeur SystemC/TLM d'un réseau sur puce Octagon*

Ce modèle de routeur ne peut pas pour autant être utilisé comme modèle synthétisable car il n'est pas réaliste. Comme pour le routeur TAC, les transactions sont acheminées sans contention par ce routeur. La méthode `transport` peut en effet être appelée au même instant par plusieurs processus. Ceci implique qu'une réalisation matérielle de ce nœud de routage doit disposer de ressources de calcul et de mémorisation infinies afin de pouvoir traiter en même temps un nombre arbitraire de transactions.

2.2.4.2 Modèle Avec Contention

Pour avoir un modèle plus réaliste et donc plus facilement synthétisable, le nœud de routage du réseau sur puce Octagon étudié précédemment a été modifié. Comme l'illustre le programme 2.4, la principale modification apportée à ce nœud de routage a été d'encadrer le corps de sa méthode `transport` avec un sémaphore.

Cette modification permet donc de limiter l'accès à cette méthode à un nombre fixe de processus. Ceci revient, en fait, à modéliser la contention en utilisant une fonction d'arbitrage implicite. Une contrainte forte pour synthétiser ce type de modèle est que la modélisation de l'arbitrage n'est pas explicite. Avec ces modèles, un outil de synthèse doit donc inférer le matériel à générer à partir d'une description purement comportementale (code d'accès au sémaphore). L'outil TAST n'est pas capable à l'heure actuelle de réaliser ce type d'inférence qui constitue un sujet de recherche à part entière (i.e. problématique de la synthèse comportementale) et qui sort du cadre de cette thèse.

Une autre solution, pour résoudre ce problème, serait de faire de la synthèse par composants (instanciation de composants matériels complets pour des macros blocs de code). Cependant, ces méthodes de synthèse ne sont pas efficaces et ne sont pas adaptées au flot de synthèse TAST.

```
RSP transport(REQ req) {
  // P on semaphore
  while(concurency_level != 0 \
        and transaction_nb >= concurency_level)
    wait(concurent_event)

  // Octagon routing algorithm
  switch((req.adr - adr) mod 8)
  {...}

  // V on semaphore
  --transaction_nb;
  concurent_event.notify();

  return rsp;
}
```

PROGRAMME 2.4 – *Modèle TLM d'un nœud de routage Octagon avec contention*

Une dernière solution envisageable serait de rendre la fonction d'arbitrage explicite en ajoutant dans chaque nœud de routage un processus qui soit chargé d'en contrôler l'accès. Une limitation importante de cette solution est de ne pas être conforme à la méthode de modélisation TLM (i.e. ajout de processus dans la couche transport) et d'en perdre les avantages. En effet, le principal avantage d'un modèle TLM est de pouvoir être simulé efficacement car il possède peu de processus et qu'il nécessite peu de changements de contexte pour être simulé.

2.2.4.3 Les Limitations

En plus des limitations vues avec les exemples précédents, l'utilisation de modèle TLM comme entrée du flot de synthèse TAST est restreinte par les points suivants.

Premièrement, le type exact d'une transaction est encapsulé dans le paramètre et dans la valeur de retour de la méthode `transport`. Il est donc difficile, voir impossible, de connaître statiquement le type exact des données qui vont transiter entre un port initiateur et un port cible.

Deuxièmement, un réseau sur puce à commutation de paquets peut difficilement être modélisé sans ajouter de processus dans les nœuds de routage. Pour modéliser explicitement ce mécanisme de commutation, il est en effet nécessaire qu'une transaction corresponde à la transmission d'un paquet entre deux nœuds de routage. Un nœud de routage doit donc posséder son propre processus pour pouvoir transmettre les paquets qu'il reçoit.

Ces nombreuses limitations sont les raisons pour lesquelles a été abandonnée l'idée d'utiliser la bibliothèque SystemC TLM pour modéliser des circuits asynchrones insensibles aux délais qui puissent être utilisés par le flot de synthèse TAST. Toutefois, les concepts d'utiliser des modèles SystemC non temporisés et des appels de méthodes transactionnelles pour communiquer ont été réutilisés au cours de cette thèse. Ils ont notament été employés pour réaliser l'extension de la bibliothèque SystemC ASC (cf. chapitre 3) dédiée à la modélisation des systèmes asynchrones.

Chapitre 3

La Modélisation de Circuits Asynchrones en SystemC

Introduction

Comme l'a montré le chapitre 2, la bibliothèque SystemC standard ne permet pas de modéliser fidèlement des circuits asynchrones insensibles aux délais mais elle peut facilement être étendue. Une extension de cette bibliothèque, dénommée « *DTU Channel Package* », a notamment été proposée pour pouvoir modéliser des circuits asynchrones à différents niveaux d'abstraction. Cette extension [BMS04] possède un canal de communication abstrait qui offre des primitives de communication similaires à celles du CHP (`send`, `receive` et `probe`). Ce canal de communication est un modèle de haut niveau des canaux de communication matériels qui sont employés par les circuits asynchrones. Les principales abstractions réalisées par ce canal de communication sont les suivantes :

- ne pas définir le protocole de communication à poignée de main (*push* ou *pull*) employé pour synchroniser une communication,
- ne pas préciser le codage des données (donnée groupée, *multi-rails*, ...) utilisé pour représenter les données transmises.

Cette extension possède aussi des canaux de communication physiques qui définissent explicitement ces différentes propriétés en spécifiant les signaux matériels utilisés pour le contrôle et pour l'acheminement des données. Finalement, cette extension permet de facilement modéliser un système à un niveau d'abstraction mixte grâce au mécanisme d'héritage de C++. En effet, un processus communiquant via un canal physique peut aussi utiliser les primitives de communication `send`, `receive` et `probe` car la classe `abstract_channel` représentant un canal abstrait est une classe de base des canaux physiques.

Malgré leurs nombreux avantages, les canaux de communication SystemC de cette extension n'ont pas été utilisés au cours de cette thèse. Premièrement, le niveau d'abstraction de ces canaux ne correspond pas à celui requis pour utiliser convenablement le flot de synthèse TAST : celui du canal abstrait et celui des canaux physiques sont respectivement trop élevés et trop bas. Les canaux physiques sont, par exemple, trop proches du matériel pour permettre d'utiliser efficacement certains algorithmes d'optimisation du flot de synthèse TAST (e.g. choix du protocole de séquençage [YR06] : WCHB, PCFB, ...). Deuxièmement, cette extension de SystemC n'est plus maintenue à l'heure actuelle.

De manière générale, les différentes extensions de SystemC qui sont actuellement disponibles ne sont pas adaptées au flot de synthèse TAST. Une extension de SystemC appelée ASC [KHRTV07] (*Asynchronous SystemC*) a donc été développée au cours de cette thèse pour pallier cette limitation.

3.1 La Bibliothèque ASC

Comme l'illustre le schéma de la figure 3.1, la bibliothèque SystemC peut être divisée en trois parties (partie structurelle, types de données et simulation) qui s'appuient sur le langage C++. La bibliothèque ASC (cf. éléments en italique de la figure 3.1) est une extension de SystemC qui permet de modéliser des circuits asynchrones. Comme la bibliothèque de canaux du DTU (*Technical University of Denmark*, la bibliothèque ASC définit de nouveaux canaux de communication qui sont proches de ceux employés par le langage CHP mais qui indiquent explicitement le protocole de communication qu'ils emploient sans fixer des détails d'implémentation (cf. paragraphe 3.1.3). Cette bibliothèque définit aussi de nouveaux types de données qui utilisent un codage *multi-rails* pour représenter leurs valeurs (cf. paragraphe 3.1.7). Finalement, cette bibliothèque étend le langage C++ avec des macros définissant de nouvelles structures de choix qui permettent notamment de modéliser fidèlement des composants non déterministes tels que des arbitres asynchrones (cf. paragraphe 3.1.5 et 3.1.6).

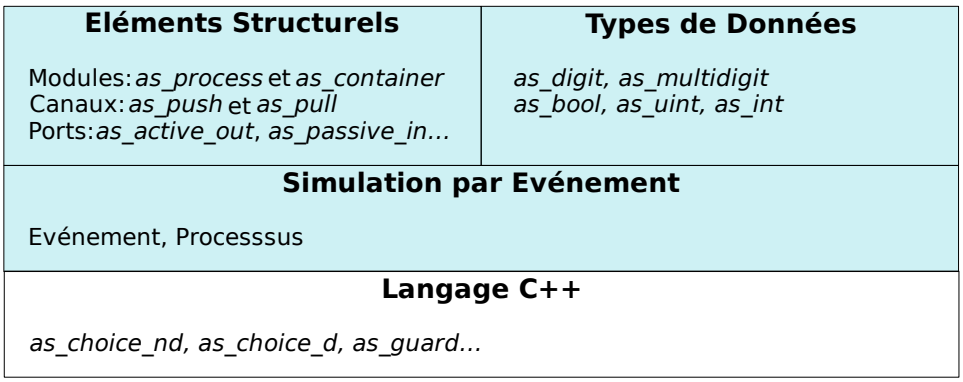


FIGURE 3.1 – *Architecture de la bibliothèque ASC*

Un point fort de ASC est d'être compatible avec n'importe quelle implémentation de la bibliothèque SystemC respectant le standard IEEE–1666 [IEE06]. La modélisation et la simulation d'un circuit asynchrone en ASC ne requierent donc qu'un compilateur C++ et ces deux bibliothèques. Cette compatibilité s'accompagne, toutefois, d'une certaine lourdeur syntaxique (e.g. structure de choix `as_choice_d`, `as_choice_nd`, ...). Il est à noter que pour améliorer la robustesse et la réutilisation du code, la bibliothèque ASC a été développée en respectant aux mieux les règles d'écriture (de l'anglais *coding standard*) C++ fixées par [SA04].

3.1.1 Les Modules

Un module SystemC standard peut contenir un nombre arbitraire de processus, de modules, de canaux, ... ; il est donc utilisé pour modéliser les aspects comportementaux

et structurels d'un système. La bibliothèque ASC définit pour sa part deux types de modules différents qui permettent de clairement séparer les parties comportementales et structurelles d'un système. Ce choix de séparer explicitement les parties comportementales et structurelles a été motivé pour des raisons d'implémentation et pour rendre plus facile la synthèse d'un modèle ASC.

3.1.1.1 Les Modules Structurels

Un module structurel permet de décrire un système de manière hiérarchique. Il peut en effet contenir les éléments suivants :

- d'autres modules ASC (structurels ou comportementaux),
- des canaux de communication ASC,
- des ports de communication ASC.

Un module structurel peut aussi posséder les mêmes types d'objets que ceux qui sont autorisés pour les modules SystemC standard. Dans l'objectif de pouvoir utiliser des modèles ASC comme des entrées du flot de synthèse TAST, il n'est toutefois pas recommandé d'inclure des éléments tels que des variables partagées ou des processus SystemC. Comme pour les modules SystemC standard, un module structurel ASC est obtenu en dérivant la classe de base `as_container`.

3.1.1.2 Les modules Comportementaux

Les modules comportementaux sont utilisés pour définir le comportement d'un système. Un module comportemental est prévu pour contenir un processus ASC et un nombre arbitraire de ports ASC, mais il peut aussi posséder les mêmes autres types d'éléments qu'un module structurel (canaux, processus SystemC, ...). Toujours dans l'objectif de faciliter la synthèse d'un modèle ASC, il n'est pas recommandé d'inclure de tels éléments dans un module comportemental. Un module comportemental est créé en dérivant la classe de base abstraite `as_process` qui est constituée des méthodes suivantes.

`process :` méthode virtuelle pure qui définit le comportement du processus qui est associé à un module comportemental. L'implémentation de cette méthode est donc à la charge des classes qui héritent de la classe de base abstraite `as_process`. Cette méthode associée à la méthode `sc_thread` implémente le patron de conception « *Template Method* » [GHJV95] : la méthode `sc_thread` est chargée d'appeler la méthode `process` et de gérer l'initialisation et la destruction des structures de données qui sont associées à un processus (cf. paragraphe 4.2.1).

`idle :` méthode qui permet de synchroniser l'exécution d'un processus ASC en fonction d'un ensemble de ports passifs ASC ou d'un ensemble de processus dynamiques. Le comportement et le rôle de cette méthode sont détaillés dans le paragraphe 3.1.2 et le paragraphe 3.1.4.

Un module comportemental hérite aussi des méthodes d'un module SystemC standard et son processus peut donc faire appel à la méthode `wait` pour modéliser un délai.

3.1.2 Les Ports

Comme pour la bibliothèque SystemC standard, les ports ASC sont les points d'entrée-sortie des processus qui leur permettent de communiquer via des canaux de communication. Ces ports sont unidirectionnels (entrée ou sortie) et peuvent être connectés à au plus

un autre port via un canal du paragraphe 3.1.3. Pour permettre à un processus d'accéder à un canal se trouvant à un niveau hiérarchique différent, il est possible de connecter directement deux ports ASC identiques (cf. connexion hiérachique du paragraphe 2.1.1.5).

Comme pour les circuits asynchrones, une communication entre deux processus s'effectue en respectant un protocole à poignée de main de type *push* ou *pull* (cf. paragraphe 1.1.2.1). Afin de distinguer le processus qui initie une communication de celui qui l'acquitte, la bibliothèque ASC dispose des deux types de ports suivants : les ports « actifs » et les ports « passifs ». Un port passif possède une méthode de test `probe` qui lui permet de vérifier si le port actif, auquel il est connecté, a initié une communication ou non. La bibliothèque ASC définit les deux ports passifs `as_passive_in` et `as_passive_out` qui peuvent être connectés respectivement aux deux ports actifs `as_active_out` et `as_active_in`.

En plus de son rôle d'interface de communication, un port passif peut aussi être utilisé avec la méthode `idle` pour synchroniser l'exécution d'un processus. Cette méthode peut prendre en paramètre une liste de ports passifs construite soit avec l'opérateur C++ `|` soit avec l'opérateur C++ `&` . Ces deux opérateurs permettent de construire respectivement des « *or list* » et des « *and list* » de ports passifs. Comme l'illustre la figure 3.2, les conditions de suspension et de réveil d'un processus sont modifiées lorsqu'il utilise la méthode `idle` avec une *and list* ou une *or list*.

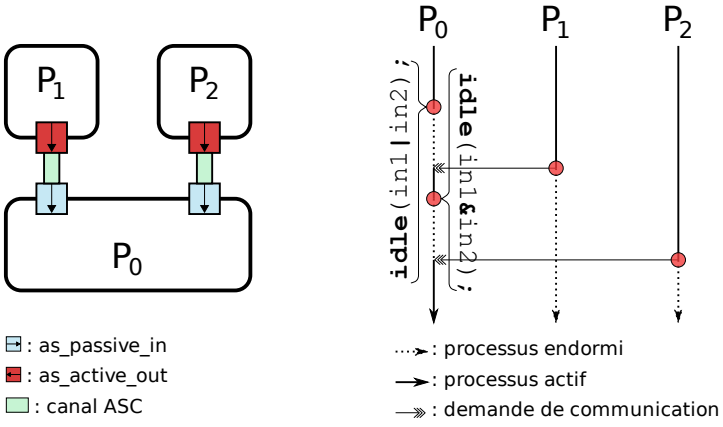


FIGURE 3.2 – *Exemple d'utilisation de la méthode* `idle` *avec des ports passifs*

Un processus appelant la méthode `idle` avec une *or list* est suspendu si et seulement si aucune demande de communication n'a été reçue par l'un des ports passifs de cette liste. Un processus suspendu sur une *or list* est réveillé quand au moins une demande de communication parvient à l'un des ports passifs de cette liste.

Avec une *and list*, un processus est suspendu si et seulement si au moins un port passif de cette liste n'a pas reçu une demande de communication. Un processus suspendu sur une *and list* est réveillé dès que tous les ports passifs de cette liste ont reçu une demande de communication.

Afin de rendre l'utilisation de la méthode `idle` plus flexible, il est aussi possible de créer une *or list* avec une liste de la bibliothèque STL [Aus98] (*Standard Template Library*) de C++. Ceci permet de pouvoir fixer dynamiquement au cours d'une simulation les ports passifs d'une *or list* que l'on souhaite passer en paramètres de la méthode `idle`. L'introduction de cette fonctionnalité est le résultat d'une coopération avec le CEA-LETI [1]

[1] http://www-leti.cea.fr

et plus particulièrement avec Florent Rotenberg qui l'a utilisée pour réaliser son modèle ASC de SERAPHIN (cf. paragraphe 3.2.3).

3.1.3 Les Canaux

Les canaux de communication sont des éléments essentiels de la bibliothèque ASC car ce sont eux qui permettent aux processus de communiquer et de se synchroniser. La bibliothèque ASC définit les deux canaux suivants :

- `as_push` qui relie un port `as_active_out` avec un port `as_passive_in` et qui implémente le protocole à poignée de main *push* ,
- `as_pull` qui connecte un port `as_passive_out` avec un port `as_active_in` et qui réalise le protocole à poignée de main *pull* .

Ces deux canaux de communication ont une méthode `send` qui est utilisée par les ports de sortie pour envoyer une donnée. Une donnée transmise par un de ces canaux doit définir un opérateur de copie car ceux-ci ne transfèrent pas la donnée elle même mais une copie de celle-ci. La méthode `receive` de ces canaux est utilisée par les ports d'entrée pour recevoir les données émises à l'aide de la méthode `send` précédente. Ces canaux possèdent en outre une méthode `probe` qui permet à un port passif de vérifier si un canal est prêt à communiquer ou non. Lorsqu'un canal `as_push` est prêt à communiquer, la méthode `probe` peut être aussi utilisée pour consulter (sans la consommer) la valeur de la donnée transférée.

Comme l'illustre l'annexe A, les canaux de communication ASC respectent les propriétés fondamentales des circuits asynchrones. Premièrement, les méthodes `send` et `receive` de ces canaux de communication ne font aucune hypothèse sur les délais nécessaires pour qu'elles réalisent une action de communication car ces méthodes n'utilisent que des variables locales et des notifications immédiates pour se synchroniser. Deuxièmement, les protocoles à poignée de main qui sont définis par ces méthodes sont synchrones et sans mémoire. En effet, deux processus qui réalisent une communication via ces deux méthodes sont bloqués jusqu'à la terminaison de celle-ci et la valeur des données qui est échangée n'est pas mémorisée par le canal.

Si par défaut, une communication est accomplie sans délai par un canal de communication ASC, il n'en reste pas moins que, après avoir validé fonctionnellement un modèle, il est nécessaire de pouvoir évaluer ses performances. Pour répondre à ce besoin, des délais peuvent être associés aux phases de requête et/ou d'acquittement du protocole de communication employé par un de ces canaux de communication. La possibilité d'avoir des canaux ASC temporisés a été initialement ajoutée pour permettre à Yvain Thonnart du CEA-LETI d'évaluer son modèle ASC du réseau sur puce ANOC (cf. paragraphe 3.2.2).

Un avantage de ces canaux de communication est de permettre de modéliser un système à différents niveaux d'abstraction : le type des données communiquées par ces canaux n'est pas fixé (paramètre générique) et peut donc être adapté en fonction du niveau d'abstraction visé. Ces canaux de communication ont notamment été utilisés pour modéliser les réseaux sur puce ANOC et Octagon (cf paragraphe 3.2.1) à des niveaux d'abstraction qui sont respectivement équivalents au niveau RTL (*Register Transfer Level*) des circuits synchrones et au niveau TLM-PV. Un autre avantage de ces canaux de communication est d'être très proches de ceux employés par le langage CHP du flot de conception TAST. Ils pourraient donc être facilement synthétisés par l'outil de synthèse de TAST.

3.1.4 Les Communications Parallèles

Lorsqu'on modélise un circuit asynchrone, il est souvent nécessaire d'effectuer des communications parallèles au sein d'un même processus. Ce type de construction est, par exemple, très utile pour modéliser facilement et fidèlement une fourche d'un circuit asynchrone insensible aux délais. Malheureusement, un processus SystemC est prévu pour effectuer des séries d'actions séquentiellement. Pour pallier cette limitation, les canaux de communication ASC définissent les deux méthodes `par_send` et `par_receive`.

Comme l'illustre la figure 3.3, chacune de ces méthodes crée un nouveau sous processus pour paralléliser leurs actions de communication respectives. La synchronisation de l'exécution de ces sous processus de communication avec leur processus père est réalisée à l'aide de l'opérateur `||` et de la méthode `idle`. Le rôle de l'opérateur `||` est de créer une liste des sous processus de communication qui puisse être utilisée par la méthode `idle`. Pour sa part, la méthode `idle` suspend le processus principal jusqu'à ce que tous les sous processus de la liste renvoyée par l'opérateur `||` soient terminés.

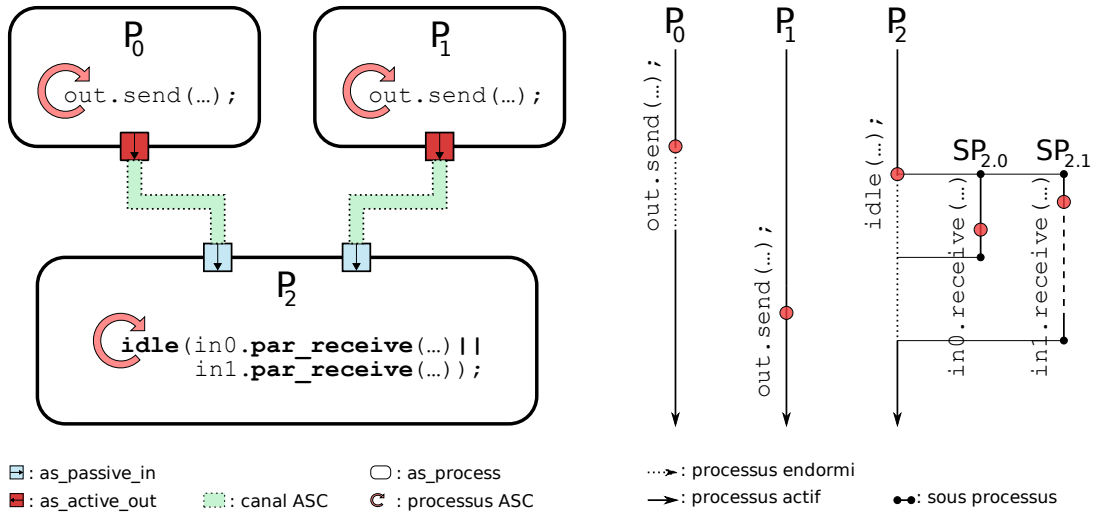


FIGURE 3.3 – *Modélisation d'une fourche en ASC avec des communications parallèles*

Pour qu'un modèle ASC, utilisant des communications parallèles, soit cohérent, il est nécessaire qu'il respecte les règles d'écriture suivantes :

- toujours utiliser les méthodes `par_send` et `par_receive` conjointement avec la méthode `idle`,
- ne jamais utiliser le même port pour effectuer plusieurs communications parallèles,
- utiliser avec précaution des variables partagées entre des communications parallèles.

De même que pour les ports passifs, la méthode `idle` peut aussi prendre en paramètre une liste de sous processus créée à l'aide de la bibliothèque STL de C++. Cette fonctionnalité a aussi été introduite pour améliorer la flexibilité du modèle ASC de SERAPHIN.

3.1.5 Les Structures de Choix non Déterministes

Pour avoir la possibilité de modéliser fidèlement le comportement non déterministe d'un arbitre asynchrone en SystemC (cf. paragraphe 2.1.3.2), la bibliothèque ASC définit

les deux nouvelles instructions `as_choice_nd` et `as_guard`. Ces deux instructions permettent de définir des structures de choix non déterministes similaires à celles du langage « *Guarded Command* » de Dijstra [Dij75]. La syntaxe de ces nouvelles instructions est définie par les pseudo règles EBNF (*Extended Backus Normal Form*) suivantes.

$$
\begin{aligned}
select\_seq &::= \texttt{as\_choice\_nd}(\ guard\_seq\ )\{\ case\_seq_{opt}\ default\_st_{opt}\ \}\\
guard\_seq &::= guard \mid guard\_seq\ ,\ guard\\
guard &::= \texttt{as\_guard}(\ exp\ ,\ const\_exp\ )\\
case\_seq &::= case\_st \mid case\_seq\ case\_st\\
case\_st &::= \texttt{case}\ const\_exp : statement\_seq_{opt}\\
default\_st &::= \texttt{default} : statement\_seq_{opt}
\end{aligned}
$$

La définition d'une structure de choix, à l'aide de ces instructions, est divisée en deux parties distinctes. Une première partie est chargée de définir les expressions booléennes (symbole non terminal *exp*) des gardes. En outre, une étiquette (symbole non terminal *const_exp*) identifiant un bloc d'instructions (symbole non terminal *case_st*) est associée à chacune de ces expressions booléennes. La deuxième partie est constituée des séquences de blocs d'instructions (symbole non terminal *case_seq*) qui sont exécutées en fonction de l'évaluation des expressions booléennes de la première partie. De même que pour les structures de choix standard du C++, il est possible d'imbriquer des structures de choix non déterministes ASC.

Lorsque plusieurs gardes d'un choix non déterministe sont vraies, un générateur de nombres aléatoires est utilisé pour déterminer quel ensemble d'instructions gardé doit être exécuté. Le générateur de nombres aléatoires utilisé par un choix non déterministe ASC s'appuie sur la fonction standard C++ `random`.

Les générateurs de nombres aléatoires de chaque choix non déterministe ne sont pas corrélés car ils utilisent chacun une graine (de l'anglais *seed*) indépendante. En effet, un générateur aléatoire ne peut être créé que par l'unique fabrique [Ale01] de générateurs aléatoires qui garantit que les graines qu'elle attribue ne sont pas corrélées. Cette fabrique utilise aussi la fonction `random` pour calculer les valeurs des graines qu'elle attribue. La graine utilisée par la fabrique peut pour sa part soit être calculée automatiquement (en fonction de la date courante), soit être fixée par l'utilisateur. Le principal intérêt de cette dernière fonctionnalité est de permettre de rejouer une simulation plusieurs fois (nécessaire pour le débogage).

Comme l'illustre l'exemple de la figure 3.4, ces nouvelles instructions ajoutées à la méthode `idle` permettent de modéliser fidèlement un arbitre asynchrone. L'attente d'une communication étant modélisée avec la méthode `idle` et le comportement non déterministe par l'instruction `as_choice_nd` (cf. paragraphe 1.2.3.2).

Cette structure de choix non déterministes doit toutefois être employée avec précaution car elle permet aussi de diriger la synthèse. Leur présence indique notamment que du matériel supplémentaire doit être synthétisé afin de contrôler le phénomène de métastabilité et assurer l'exclusion mutuelle des gardes (cf. paragraphe 5.2).

3.1.6 Les Structures de Choix Déterministes

Lorsque les gardes d'une structure de choix sont naturellement exclusives, il est préférable de ne pas employer une structure de choix non déterministe pour éviter de générer

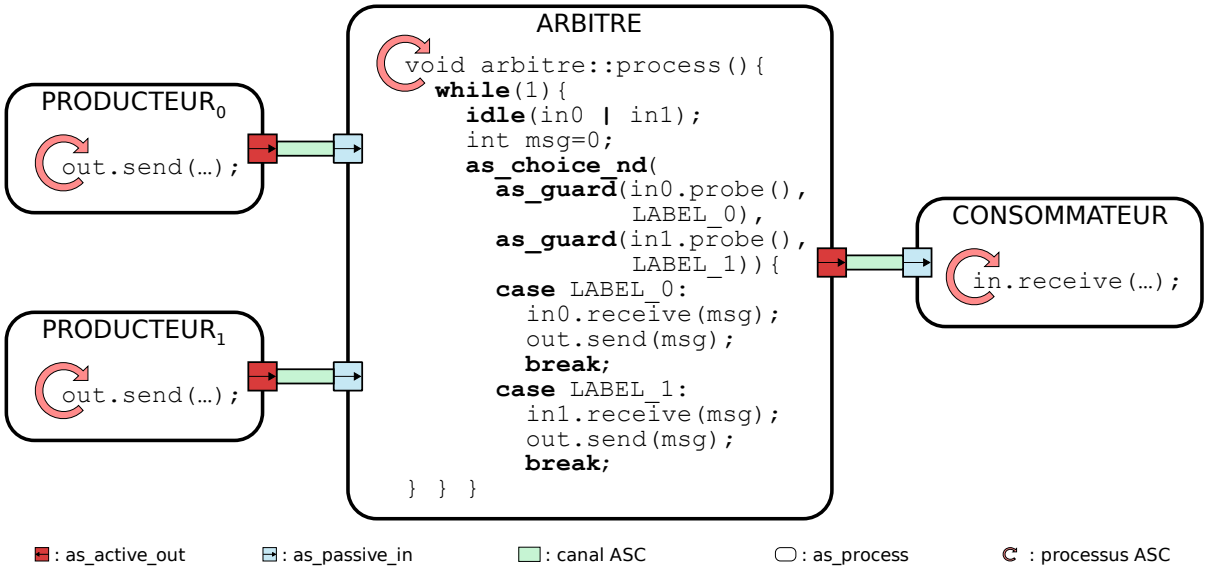


FIGURE 3.4 – *Modélisation d'un arbitre en ASC*

du matériel inutile lors de sa synthèse. Les structures de choix C++ standard ne sont pas non plus adaptées car, de même que pour les structures de choix non déterministes, elles imposent un ordre d'évaluation qui n'est pas toujours nécessaire. De plus, elles ne permettent pas de vérifier facilement que toutes leurs gardes sont bien mutuellement exclusives car ces dernières ne sont pas toutes obligatoirement évaluées. Il est pourtant très utile de pouvoir vérifier dynamiquement (lors d'une simulation) que les gardes d'une structure de choix déterministe sont bien mutuellement exclusives afin de ne pas obtenir un circuit qui soit fonctionnellement incorrect.

Pour résoudre ce problème, la bibliothèque ASC possède une nouvelle instruction `as_choice_d`. Comme l'illustre la figure 3.5, la syntaxe adoptée pour définir un choix déterministe avec cette instruction est identique à celle employée avec l'instruction `as_choice_nd`. Comme pour un choix non déterministe, toutes les gardes d'un choix déterministe sont évaluées mais si plus d'une de ces gardes sont vraies alors une notification d'erreur est envoyée au gestionnaire d'erreur SystemC (cf. pages 378–385 de [IEE06]).

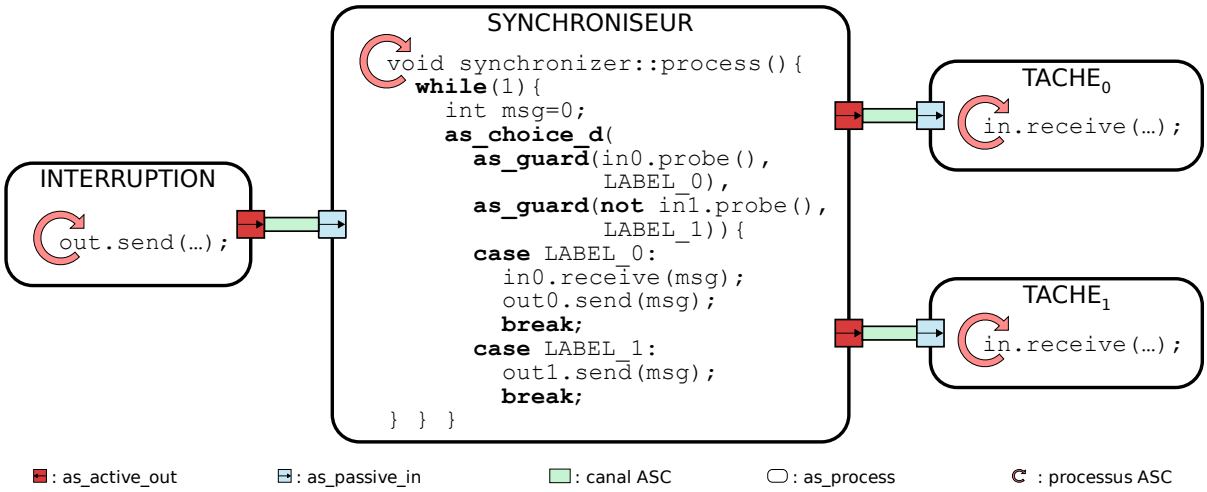


FIGURE 3.5 – *Modélisation d'un synchroniseur en ASC*

3.1.7 Les Types de Données Insensibles aux Délais

Pour être en mesure de synthétiser des circuits asynchrones insensibles aux délais à partir de modèles ASC, de nouveaux types de données qui utilisent le code *multi-rails* ont été définis (cf. paragraphe 2.1.3.3). Comme l'illustre le diagramme de classe UML de la figure 3.6, ces nouveaux types de données reposent sur la classe `as_digit` qui représente un ensemble de fils pouvant prendre les valeurs `0` ou `1`. Afin de respecter le code *multi-rails*, une instance de la classe `as_digit`, appelé *digit*, est autorisée à avoir, à tout instant, au plus un seul de ses fils à la valeur `1` (cf. paragraphe 1.1.2.2). La bibliothèque ASC définit aussi des opérateurs (`&`, `[]`, `<<`, ...) et des méthodes (`set`, `none`, `reset`, ...) qui permettent de manipuler un *digit* de manière similaire à un vecteur de bits.

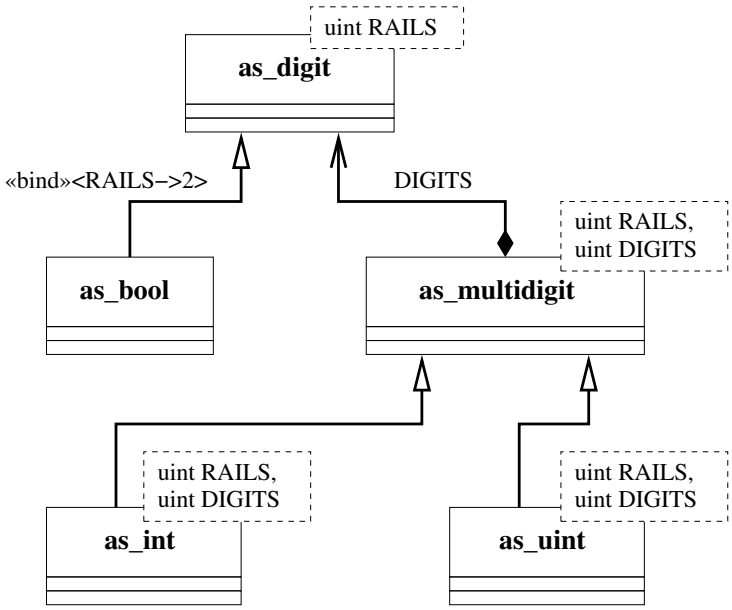


FIGURE 3.6 – *Diagramme de classe UML des types de données insensibles aux délais ASC*

Une particularité des circuits asynchrones insensibles aux délais est d'utiliser le codage des données pour synchroniser leurs communications (cf. paragraphe 1.1.2.2). L'absence de communication est notamment signalée, par l'absence de fil ayant la valeur `1`, dans un canal de communication d'un circuit asynchrone insensible aux délais. Il est donc fonctionnellement incorrect d'envoyer un *digit* `null` (i.e. un *digit* dont tous les fils sont à `0`) avec un canal de communication ASC. Pour s'assurer que cette contrainte est bien respectée, les ports de sortie ASC vérifient que les données qui lui sont transmises ne contiennent pas de *digit* `null`

3.1.7.1 Les Booléens

Un type de base redéfini par la bibliothèque ASC est le type booléen qui est représenté à l'aide de la classe `as_bool`. Un objet de type `as_bool` est en fait un *digit* composé de deux fils avec lequel on peut utiliser les opérateurs booléens suivants : `and`, `or`, `xor` et `not`.

En plus des valeurs booléenes traditionnelles `true` et `false`, une instance de la classe `as_bool` peut être égale à la valeur `null` lorsque ses deux fils sont à `0`. Une propriété intéressante de la valeur `null` est d'indiquer qu'une variable n'a pas de valeur définie (e.g.

une variable non initialisée) et qu'il est donc incorrect de la lire pour effectuer des calculs (utilisation des opérateurs booléens).

3.1.7.2 Les Entiers

Les autres types de base redéfinis par la bibliothèque ASC sont les entiers signés et les entiers non signés qui sont respectivement modélisés par les classes `as_int` et `as_uint`. La valeur d'un entier modélisé avec l'une ou l'autre de ces deux classes est définie à l'aide de la classe `as_multidigit` qui est constituée d'un ensemble de *digits*. La valeur d'un entier est représentée par un objet de type `as_multidigit`, appelé *multidigit*, en respectant la convention petit-boutiste (de l'anglais *little-endian*). Le *digit* de poids faible d'un entier est donc indexé à la position `0` de son *multidigit*.

La classe `as_multidigit` possède des méthodes (`begin`, `end`, ...) et des opérateurs (`[]`, `|`, `>>`, ...) permettant de manipuler ses *digits*. Il est aussi possible d'accéder aux *digits* d'un *multidigit* en utilisant des itérateurs [GHJV95] (`as_iterator`, `as_const_iterator`, ...). L'implémentation de ces itérateurs est basée sur celle présentée dans [Aus01] et leurs principaux intérêts sont de vérifier qu'ils pointent toujours sur un espace mémoire valide (pages 561–566 de [Str97]).

Un entier ASC est égal à la valeur `null` lorsque tous ses *digits* sont `null`. Un entier ASC ne peut pas être constitué à la fois de *digits* `null` et de *digits* non `null`. Il n'est pas non plus autorisé d'utiliser un opérateur arithmétique avec un entier `null`.

Les Entiers Non Signés

Les entiers non signés sont modélisés en ASC à l'aide de la classe `as_uint` qui possède deux paramètres génériques `RAILS` et `DIGITS` définissant respectivement sa base et sa taille. La base d'un entier non signé `as_uint` est égale au nombre de fils que possède chacun de ses *digits*. La taille d'un entier non signé est pour sa part définie par le nombre de *digits* qu'il possède.

Etant donné que la base et la taille d'un entier `as_uint` ne sont pas fixées, il n'est pas possible d'utiliser directement les opérateurs arithmétiques standards du C++. Les opérateurs arithmétiques associés à la classe `as_uint` ont donc été réalisés en utilisant les algorithmes d'arithmétique standard [Fri75] suivants :

- l'opérateur d'addition utilise l'algorithme d'un additionneur complet à propagation de retenue,
- l'opérateur de soustraction se sert d'un algorithme à propagation de retenue similaire à celui de l'additionneur complet,
- L'opérateur de multiplication emploie l'algorithme traditionnel de multiplication longue (i.e. addition plus décalage),
- Les opérateurs de division euclidienne (division et modulo) ont recours à l'algorithme de la division longue.

Une propriété intéressante de ces opérateurs arithmétiques est de signaler les dépassements de capacité en envoyant une notification d'avertissement au gestionnaire d'erreur SystemC.

En plus des opérateurs arithmétiques, la bibliothèque ASC définit des opérateurs de décalage de *digits*. Ces opérateurs réalisent des décalages « logiques », car des *digits* de valeur `0` sont introduits dans les positions vacantes (cf. figure 3.7). Ils peuvent, en outre, être employés pour effectuer des divisions ou des multiplications par la base employée. Sur l'exemple de la figure 3.7, le décalage à droite de deux digits équivaut à une division

par $base^2 = 3^2$. Toutefois, ces manipulations doivent être accomplies avec précaution car, contrairement aux opérateurs arithmétiques, elles ne vérifient pas les dépassements de capacité (cf. décalage à gauche de la figure 3.7).

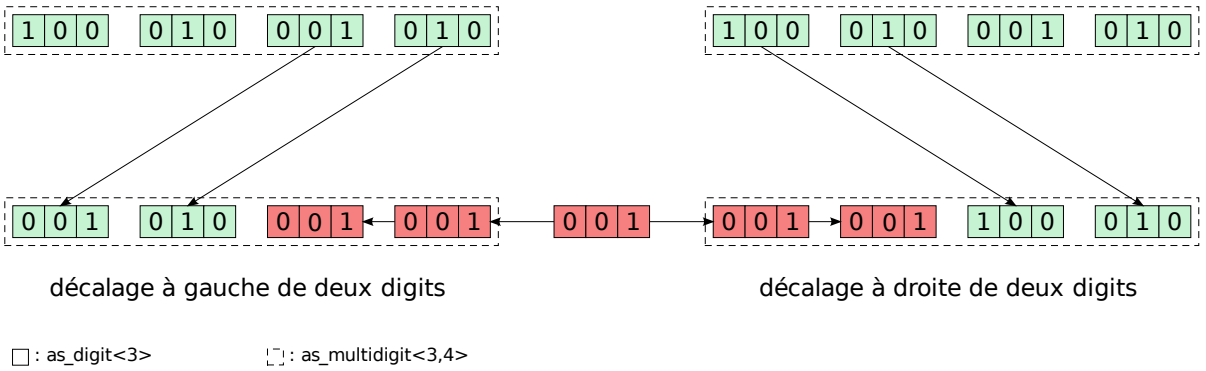


FIGURE 3.7 – *Exemples de décalage d'un entier non signé en ASC*

Les Entiers Signés

De même que la classe `as_uint`, la classe `as_int` servant à modéliser des entiers signés possède deux paramètres génériques `RAILS` et `DIGITS` qui déterminent respectivement sa base et sa taille. Le signe d'un entier `as_int` est représenté en utilisant la méthode du complément à « n » où n est égal à la base de cet entier. Le principal intérêt de cette méthode de représentation est de permettre de réutiliser facilement les algorithmes d'arithmétiques employés par les opérateurs arithmétiques de la classe `as_uint`. Comme pour la classe `as_uint`, les opérateurs arithmétiques de la classe `as_int` vérifient qu'ils ne génèrent pas des dépassements de capacité.

Contrairement aux opérateurs de décalage de *digits* de la classe `as_uint`, les opérateurs de décalage de la classe `as_int` effectuent des décalages « arithmétiques ». Comme l'illustre la figure 3.8, le décalage à gauche arithmétique est identique au décalage à gauche logique. Le décalage à droite arithmétique est pour sa part différent car les positions vacantes sont remplies avec des *digits* ayant une valeur égale à celle du *digit* de poids fort initial.

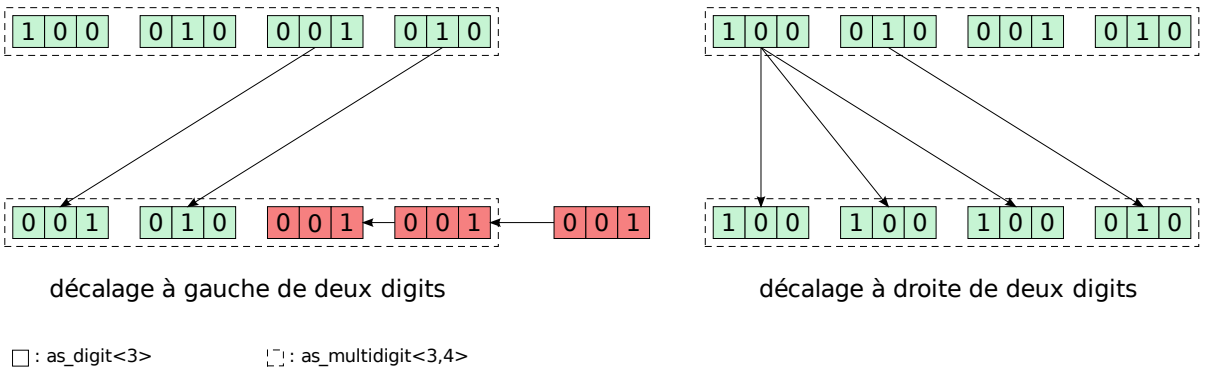


FIGURE 3.8 – *Exemples de décalage d'un entier signé en ASC*

3.1.8 Les Limitations et les Améliorations

Si la version actuelle de ASC permet de modéliser correctement un circuit asynchrone en SystemC, elle possède néanmoins quelques limitations. Une première limitation est liée à l'utilisation des types de données insensibles aux délais de ASC qui entraînent une dégradation importante du temps de simulation d'un modèle ASC. Ce sont plus particulièrement les opérateurs arithmétiques des entiers insensibles aux délais qui sont très coûteux en temps de calcul. Une solution pour pallier ce problème est de déléguer le calcul de ces opérations à des entiers SystemC non bornés (`sc_signed` et `sc_unsigned`) et d'effectuer les conversions de base nécessaires.

L'ajout de la fonctionnalité de « *probe* » aux canaux de communication est la principale différence entre CSP et ASC. L'introduction de cette nouvelle fonctionnalité a des conséquences importantes et notamment celle de faire apparaître du non déterminisme lié à l'emploi de variables partagées. Contrairement aux primitives de communication bloquantes `send` et `receive`, le résultat de la primitive de communication non bloquante `probe` dépend de l'ordonnancement des processus (cf figure 3.9). Ce phénomène est, en fait, équivalent à celui vu au paragraphe 2.1.3.4 concernant le respect de la propriété d'insensibilité aux délais des circuits asynchrones et les solutions présentées dans ce paragraphe permettent donc d'y répondre.

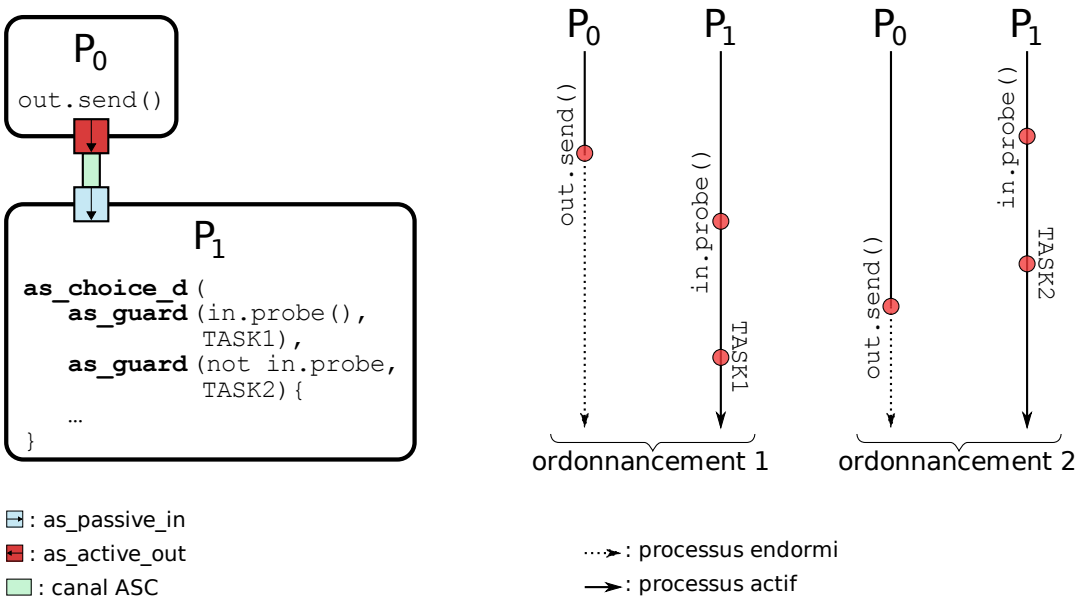


FIGURE 3.9 – *Exemples de non déterminisme en ASC avec les probes*

Une limitation de l'implémentation actuelle des *probes* est de ne pas modéliser le non déterminisme lié à l'emploi de *probes* sucessifs d'un même canal au sein d'un même processus. Contrairement à ASC, le résultat du *probe* d'un canal par un processus après avoir réalisé un premier *probe* négatif n'est, en effet, pas obligatoirement négatif, car un autre processus peut avoir initié une communication sur ce canal entre ces *probes*. Une solution simple pour résoudre cette limitation est de suspendre temporairement l'exécution d'un processus effectuant un *probe* en le faisant passer de l'état élu à l'état éligible (cf. pages 46–47 de [Hel07]).

Les limitations, vues aux paragraphe 2.1.3.4, sur les facilités de traçage standard de SystemC affectent aussi ASC car ses canaux de communication utilisent des notifications

immédiates pour se synchroniser. Une solution à cette limitation est détaillée dans le chapitre 4.

Comme l'indique le paragraphe 5.1, une structure de choix d'un circuit asynchrone est aussi un point de rendez vous (synchronisation). Pour des raisons d'implémentation, les structures de choix ASC (`as_choice_nd` et `as_choice_d`) et leurs points de rendez vous (appel de la méthode `idle` avec des ports passifs) sont découplés. Un désavantage de ce découplage est de pouvoir obtenir des modèles ASC qui n'ont pas d'équivalent matériel. En outre, il ne permet pas de modéliser n'importe quelle structure de choix d'un circuit asynchrone. Il est par exemple impossible de mélanger des *and list* et des *or list* de ports passifs dans une méthode `idle`. Une première solution pour résoudre ce problème est d'augmenter le pouvoir d'expression de la méthode `idle` et de fixer des règles d'écriture. Une deuxième solution, plus propre mais plus difficilement réalisable, est d'inclure l'appel de la méthode `idle` dans les gardes des structures de choix ASC.

3.2 Modèles de Circuits Asynchrones en ASC

En plus des différentes suites de tests qui ont été développées pour la valider, la bibliothèque ASC a aussi servi à modéliser des systèmes plus réalistes. Le nœud de routage de la plateforme de simulation TLM du réseau sur puce Octagon a été modélisé avec cette bibliothèque. Une version ASC du nœud de routage du réseau sur puce ANOC a aussi été spécifiée. Finalement, cette bibliothèque a été utilisée pour modéliser l'architecture super scalaire SERAPHIN.

3.2.1 Modélisation de Réseaux sur Puce Octagon

Dans le cadre de cette thèse, la bibliothèque ASC a été utilisée pour modéliser deux versions différentes du réseau sur puce asynchrone Octagon [KHTM07] utilisant respectivement une technique de commutation par paquets et une technique de commutation par circuits. Ces deux modèles de réseau sur puce ont été développés pour déterminer si la bibliothèque ASC est adaptée pour modéliser des réseaux sur puce qui soient synthétisables avec de la logique asynchrone et qui puissent être intégrés dans un flot de conception industriel. Pour vérifier ce dernier point, ces modèles ont été intégrés à la plateforme de simulation TLM vue au paragraphe 2.2.4. L'intégration de ces modèles de réseaux sur puce ASC a consistée d'une part à remplacer le réseau d'interconnexion TLM et d'une autre part à définir des « *transactors* » [KHR07].

Habituellement, un *transactor* est employé pour interfacer des composants modélisés à des niveaux d'abstraction différents. Dans le cas présent, les composants TLM et ASC sont spécifiés à des niveaux d'abstraction identiques mais emploient des protocoles de communication différents : les modules ASC et TLM utilisent respectivement des protocoles de communication à poignée de main et par appel de méthode qui ne sont pas directement compatibles. Le rôle des *transactors* employés par ces deux modèles d'Octagons est donc uniquement d'effectuer de la conversion de protocole. Comme l'illustre la figure 3.10, un de ces *transactors* est par exemple chargé de convertir un appel à la méthode `transport` de la bibliothèque TLM en un appel à la méthode `send` ou `receive` de la bibliothèque ASC. Un *transactor* est aussi chargé de convertir la réception d'un paquet de type `request` (cf. processus `asc_in_process`) en un appel à la méthode `transport`.

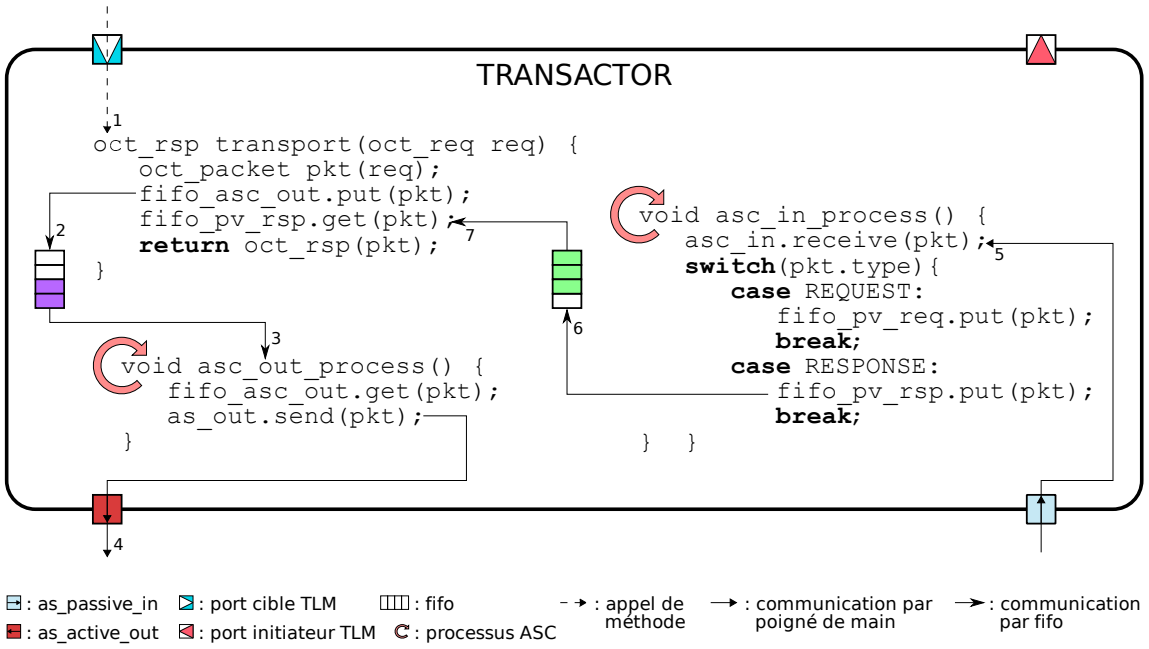


FIGURE 3.10 – *Exemple d'utilisation d'un transactor ASC-TLM*

Un point intéressant de cette méthode de modélisation à base de *transactor* est d'être proche de celle employée pour connecter des composants matériels à un réseau sur puce. En effet, un *transactor* joue un rôle identique à celui d'une interface réseau et est conçu d'une manière similaire. Cette méthode de modélisation pourrait donc servir de base intéressante pour la synthèse automatique d'interface de réseaux sur puce asynchrones.

3.2.1.1 Acheminement par Commutation de Paquets

Le premier modèle ASC de réseau sur puce Octagon utilise une technique de commutation de paquets. Selon ce modèle les nœuds de routage ne font que propager les paquets qu'ils reçoivent en fonction de leur adresse de destination. L'exemple de la figure 3.11 détaille, par exemple, comment un de ces nœuds de routage propage dans le sens horaire (via le port out_clk) un paquet qu'il a reçu de son interface réseau (via le port in_ip).

Contrairement aux nœuds de routage des modèles TLM, ces nœuds de routage modélisent explicitement la contention car leur fonction d'arbitrage est clairement spécifiée par l'instruction `as_choice_nd`. Ce type de modèle peut donc être plus facilement synthétisé que les modèles TLM vus précédemment.

3.2.1.2 Acheminement par Commutation de Circuits

Le deuxième modèle ASC de réseau sur puce Octagon qui a été implémenté utilise une technique de commutation de circuits. Contrairement au modèle précédent, il supporte la notion de transaction en distinguant les deux types de paquets suivants :

– les paquets de type « *request* » qui correspondent aux requêtes des transactions,
– les paquets de type « *response* » qui correspondent aux réponses des transactions.

Comme l'illustre la figure 3.12, le comportement initial du processus `as_process` est similaire à celui des nœuds de routage à commutation de parquets. La méthode `receive_request` attend de manière identique (utilisation de la méthode `probe` et d'un

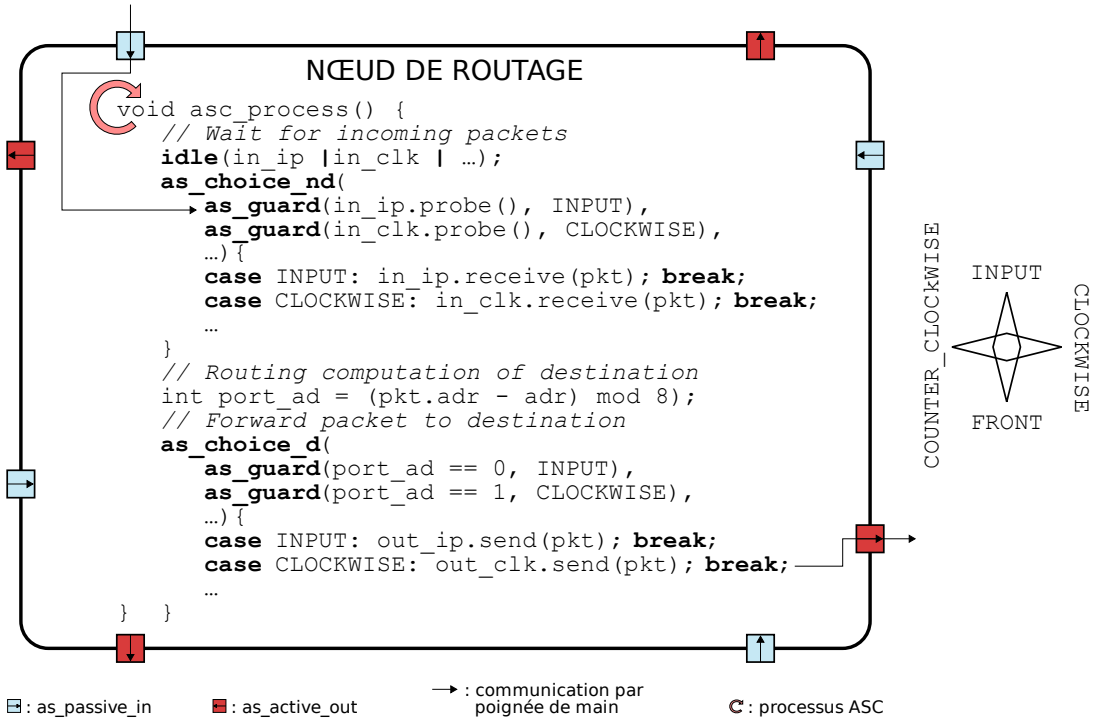


FIGURE 3.11 – *Nœud de routage ASC d'un Octagon à commutation de paquets*

choix non déterministe) qu'un paquet de type *request* soit transmis sur l'un des ports d'entrée. Elle mémorise en outre dans la variable `port_in_req` le port d'entrée par lequel elle a reçu un paquet. Ce dernier paquet de type *request* est ensuite aiguillé vers sa destination en appelant la méthode `route` qui implémente l'algorithme de routage standard de l'Octagon. Finalement, le processus de ces nœuds de routage attend la réponse à cette requête et la transmet vers le processus qui a initié cette transaction.

Si ce modèle de réseau sur puce Octagon est bien équivalent au modèle TLM initial, il n'est néanmoins pas très réaliste car chaque circuit qui est établi par un paquet de type *request* ne peut être utilisé que par un seul paquet de type *response*. Ce modèle de réseau sur puce permet tout de même de montrer que la bibliothèque ASC peut être utilisée indifféremment pour modéliser des réseaux sur puce à commutation de circuits ou à commutation de paquets.

3.2.2 Modélisation du Réseau sur Puce ANOC

Afin de disposer d'une plateforme de simulation efficace pour la conception du circuit GALS FAUST, le réseau sur puce ANOC (cf. paragraphe 1.2.4.6) a été initialement modélisé en SystemC avec des `sc_fifo` [BCV+05]. Cependant l'utilisation du SystemC standard ne permet pas de modéliser fidèlement le comportement d'un circuit asynchrone QDI (protocoles à poignée de main, canaux de communication sans mémoire, choix non déterministe, ...). Pour pallier cette limitation, le CEA-LETI a utilisé la bibliothèque ASC pour modéliser fidèlement le comportement d'un noeud de routage de ce réseau sur puce [KHRTV07].

Comme l'illustre la figure 3.13, ce modèle ASC du nœud de routage ANOC a pu être intégré dans la plateforme de simulation du CEA-LETI. Le principal intérêt de cette plateforme de simulation est de pouvoir modéliser un circuit GALS avec des composants

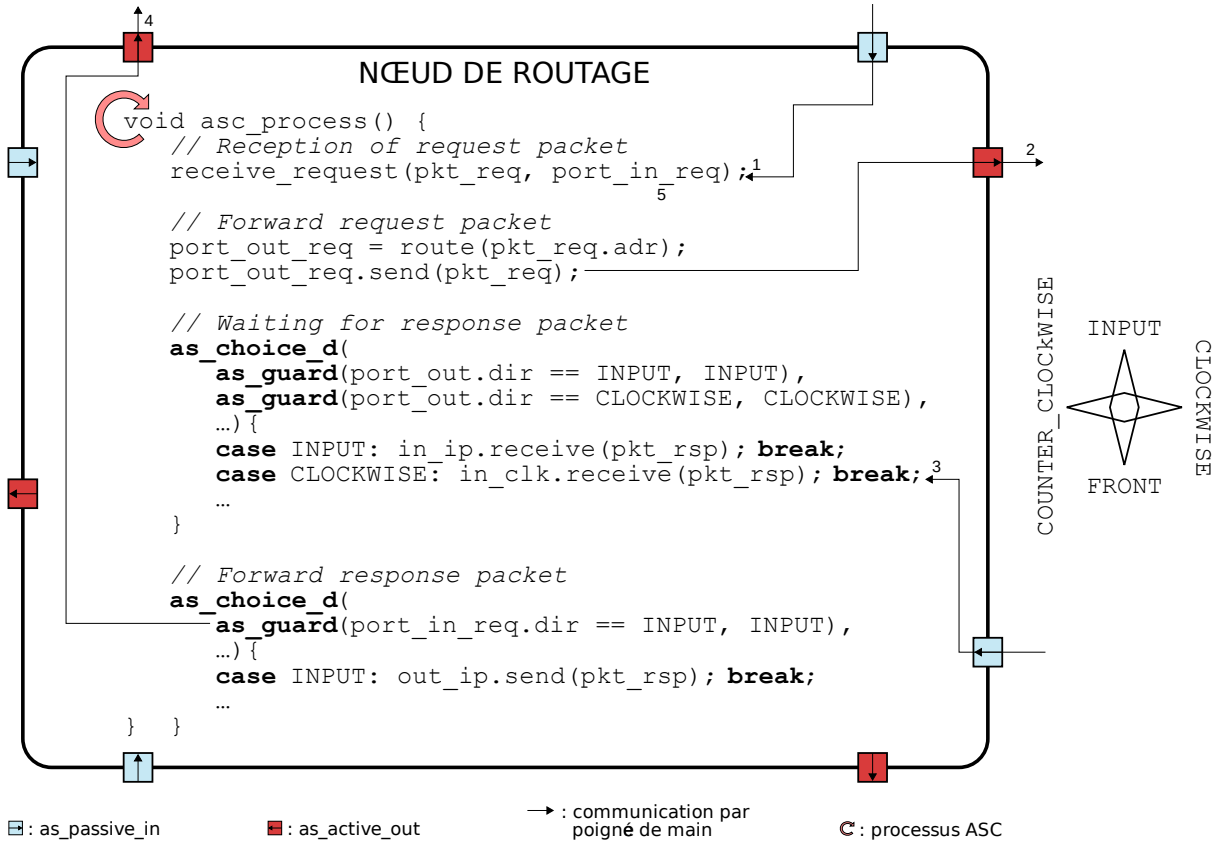


FIGURE 3.12 – *Nœud de routage ASC d'un Octagon à commutation de circuits*

(CPU, RAM, nœud de rouage, ...) décrits à des niveaux d'abstraction différents. Les nœuds du réseau sur puce ANOC peuvent être notamment modélisés en SystemC aux deux niveaux d'abstraction suivants.

TLM-PV : les modèles spécifiés à ce niveau d'abstraction permettent d'obtenir des plateformes de simulation très efficaces et sont donc principalement utilisés pour valider fonctionnellement les applications qui utilisent le réseau sur puce. Les modèles appartenant à ce niveau d'abstraction sont modélisés avec le cadre d'application TLM_ANOC [DDVC07] qui est dérivé du cadre d'application TAC de STMicroelectronics.

Handshake : ce niveau d'abstraction est équivalent au niveau RTL utilisé pour les circuits synchrones et est employé pour la validation et le débogage du réseau sur puce asynchrone. Ce type de modèle est aussi utilisé pour effectuer des simulations plus précises temporellement car ces modèles possèdent des annotations temporelles qui modélisent fidèlement le comportement temporel des composants matériels asynchrones qu'ils représentent.

Ce partenariat avec le CEA-LETI a donc permis de vérifier que la bibliothèque ASC est bien adaptée pour modéliser des réseaux sur puce asynchrones réalistes et que ces modèles pouvaient être facilement utilisés dans les flots de conception actuels.

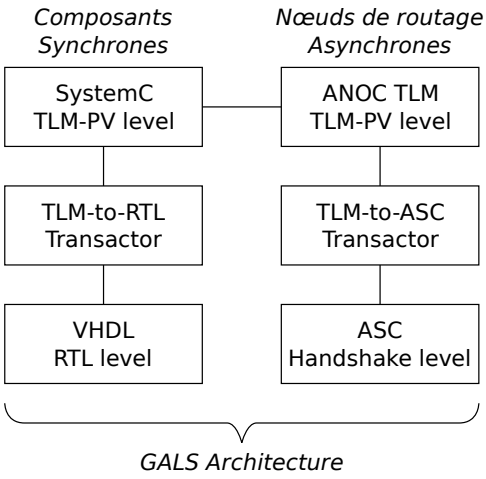


FIGURE 3.13 – *Plateforme de simulation mixte du réseau sur puce ANOC*

3.2.3 Modélisation du Processeur Superscalaire SERAPHIN

SERAPHIN [Rot07] est un processeur asynchrone superscalaire dédié au traitement du signal qui a été développé par le CEA-LETI en utilisant la bibliothèque ASC. Une des principales originalités de ce processeur est d'adopter une architecture qui soit à la fois dirigée par les opérations ou OTA (*Operation Triggered Architecture*) et à la fois dirigée par les données ou TTA [Cor94] (*Transport Triggered Architecture*). Avec une architecture traditionnelle de type OTA, ce sont les opérations (e.g. addition, soustraction, ...) qui vont déclencher les transferts de données. A l'inverse, avec une architecture de type TTA ce sont les transferts de données qui vont déclencher les opérations.

Comme l'illustre l'exemple ci-dessous, une opération d'addition d'une architecture OTA est équivalente pour une architecture TTA à la suite des opérations suivantes :

1. transférer respectivement les valeur des registres R_1 et R_2 vers les entrées $\mathsf{O}_{\mathsf{Add}}$ et $\mathsf{T}_{\mathsf{Add}}$ d'un additionneur Add,
2. transférer la valeur de la sortie $\mathsf{R}_{\mathsf{Add}}$ de cet additionneur vers le registre R_3.

$$\mathsf{Add}\ \mathsf{R}_1,\ \mathsf{R}_2,\ \mathsf{R}_3 \implies \begin{array}{l} \mathsf{R}_1 \rightarrow \mathsf{O}_{\mathsf{Add}};\ \mathsf{R}_2 \rightarrow \mathsf{T}_{\mathsf{Add}} \\ \mathsf{R}_{\mathsf{Add}} \rightarrow \mathsf{R}_3 \end{array}$$

De même qu'une architecture OTA standard, SERAPHIN possède des instructions qui permettent de configurer sa fonctionnalité. Cependant, comme pour une architecture TTA, ces instructions ne sont exécutées par SERAPHIN qu'à partir du moment où les données nécessaires à leur exécution sont disponibles. Ce type de fonctionnement hybride permet de pleinement exploiter les avantages de la logique asynchrone qui, contrairement à la logique synchrone, n'est pas contrainte d'attendre un front d'horloge pour commencer à traiter une nouvelle instruction.

Comme l'illustre la figure 3.14, l'architecture SERAPHIN est composée d'un ensemble de composants fonctionnels concurrents qui communiquent via un réseau point à point. Un composant fonctionnel est constitué d'un élément spécifique (e.g. cache de données, unité arithmétique, ...) et des trois éléments génériques suivants : une mémoire d'instructions, un cache d'instructions et un séquenceur local. Les tâches à effectuer par ces composants fonctionnels sont définies par l'unité de contrôle qui charge dans leurs mémoires d'instructions les groupes d'instructions qu'ils doivent exécuter.

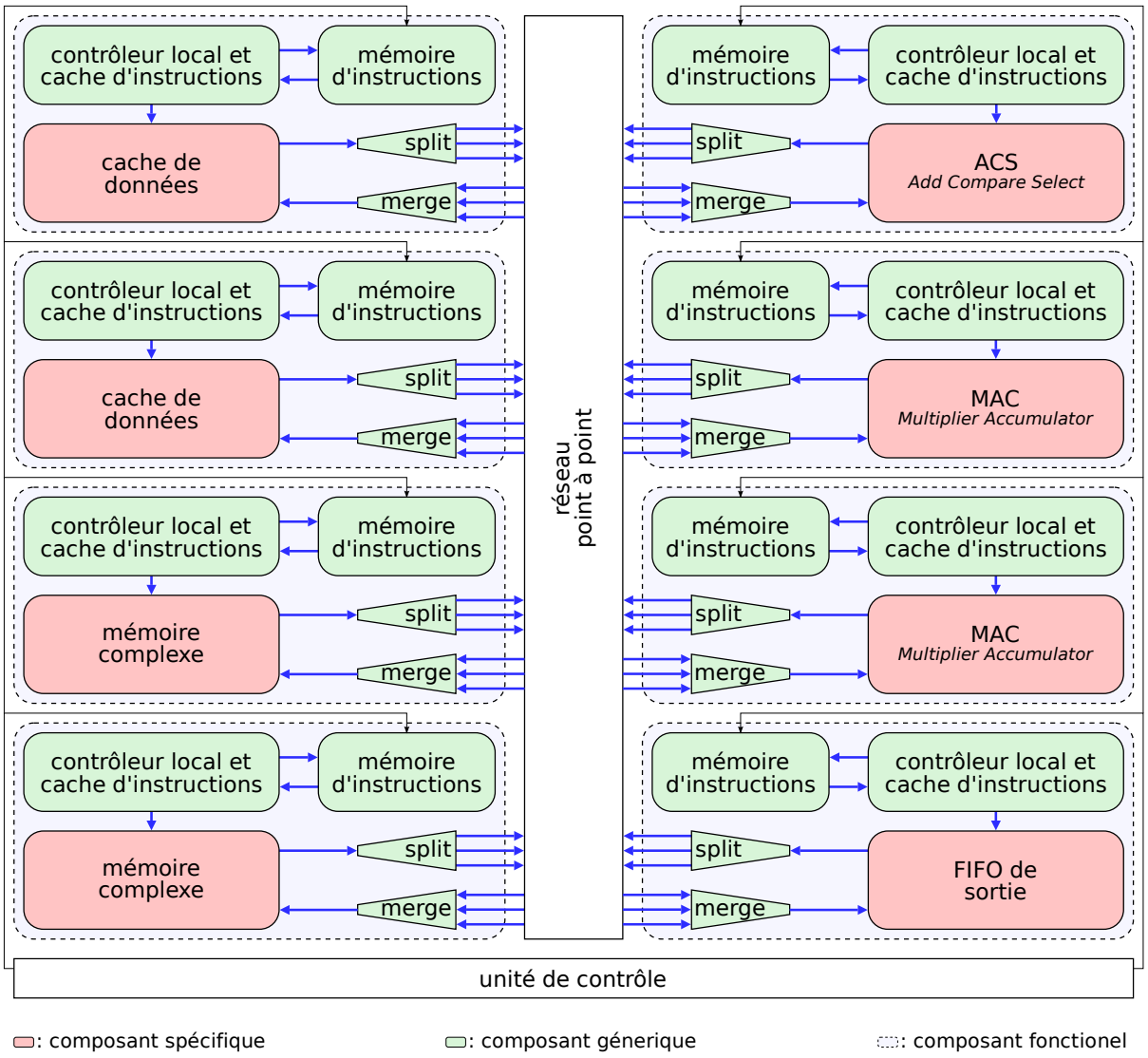


FIGURE 3.14 – *Architecture du processeur superscalaire SERAPHIN*

Cette architecture originale a été complètement modélisée en ASC, par Florent Rotenberg du CEA-LETI, pour valider les choix de spécification architecturale qui ont été adoptés pour répondre aux besoins spécifiques du domaine applicatif visé (i.e. traitement du signal dans les télécommunications). Florent Rotenberg a réalisé cette validation en effectuant des simulations et des mesures de performance du modèle ASC. Ces simulations ont consisté à faire exécuter par le modèle ASC l'algorithme de synchronisation de trame d'une chaîne de réception MC-CDMA (*Multiple Carrier-Coded Division Multiple Acces*).

Au cours de cette collaboration, il est apparu que les facilités de traçage standard de la bibliothèque SystemC ne sont pas adaptées pour valider des modèles ASC. Suite à cette collaboration de nouvelles facilités de traçage basées sur un modèle de temps distribué ont donc été développées (cf. chapitre 4).

Chapitre 4

La Validation de Circuits Asynchrones

Introduction

Pour respecter l'hypothèse d'insensibilité aux délais d'un circuit asynchrone, les canaux de communication de la bibliothèque ASC emploient des mécanismes de synchronisation basés sur des notifications immédiates. Comme l'a montré le paragraphe 2.1.3.4, les facilités de traçage standard de SystemC ne sont pas adaptées pour enregistrer l'activité de tels canaux. Plus précisément, c'est le modèle de temps centralisé sur lequel reposent ces facilités de traçage qui n'est pas approprié, car il suppose que les différents événements survenant au cours d'une simulation sont synchronisés en fonction de l'horloge globale du simulateur (delta-cyle et temps physique).

Les modèles de temps [AEF+94, CDYC97, KR99] qui ont été précédemment étudiés sont dédiés à la validation de modèles de circuits asynchrones ayant un niveau d'abstraction très faible (niveau porte). Ces modèles de temps sont principalement utilisés pour effectuer des analyses sur les latences des différents composants d'un circuit. Ils sont, par exemple, employés pour vérifier que les délais d'un circuit ne dépassent pas certaines limites en utilisant des simulations « monte-carlo » ou l'algorithme « min-max ». Ces modèles de temps sont donc basés sur une représentation physique concrète du temps qui n'est pas adaptée aux modèles ASC où la précision temporelle est fortement abstraite.

« Les horloges de Lamport » [Lam78] sont un modèle de temps couramment employé pour synchroniser l'activité des différents processus d'un système distribué : chaque processus possède sa propre horloge locale qui est synchronisée avec celle d'un autre processus lorsqu'ils communiquent. Ce modèle de temps est donc bien approprié pour ordonner les événements survenant au cours de la simulation d'un modèle ASC car ils ont des propriétés communes (processus concurrents, synchronisation par communication). Ce modèle de temps ne peut pas être néanmoins directement utilisé pour implémenter des facilités de traçage pour ASC car il ne prend pas en compte certaines spécificités des canaux de communication ASC (protocoles de communication à poignée de main et *probe*).

Dans [VPG06], un cadre d'application SystemC basé sur les horloges de Lamport est présenté pour améliorer la vitesse de simulation. Les horloges de Lamport sont employées pour coordonner efficacement l'exécution des processus SystemC sur une plateforme de simulation distribuée. L'exécution de ces processus est en effet ordonnancée en fonction des étiquettes temporelles qu'ils s'échangent au cours de leurs communications. Ce cadre d'application n'est cependant pas adapté car ses étiquettes temporelles ne permettent

pas de prendre facilement en compte les mécanismes de communication spécifiques des circuits asynchrones tels que les *probe*.

Pour être en mesure de valider correctement la fonctionnalité d'un modèle ASC (cf. paragraphe 4.2), un nouveau modèle de temps, appelé AST (*Asynchronous SystemC Time*), basé sur les horloges de Lamport a été développé (cf. paragraphe 4.1). Ces nouvelles facilités de validation ont notamment été utilisées pour valider les modèles ASC du réseau sur puce Octagon (cf. paragraphe 4.3).

4.1 Le Modèle de Temps AST

AST est un modèle de temps discret qui permet de dater de manière cohérente les différents événements survenant dans un circuit asynchrone. Pour cela, ce modèle de temps étend la relation d'ordre « survenue avant » [Lam78] (de l'anglais « *happened before* ») afin de fixer un ordre partiel entre les différents événements survenant dans un circuit asynchrone (cf. paragraphe 4.1.1). De plus, ce modèle de temps définit une fonction d'estampillage temporel qui assigne une date à chacun de ces événements en respectant cette relation d'ordre (cf. paragraphe 4.1.2 et paragraphe 4.1.3).

La figure 4.1 illustre comment cette fonction d'estampillage peut être utilisée pour dater les événements survenant dans un modèle de circuit asynchrone basé sur un langage de modélisation tel que CHP ou ASC. Ce modèle de circuit asynchrone est composé de trois processus p_0, p_1 et p_2 qui sont respectivement initialisés par les événements $init_0$, $init_1$ et $init_2$. Les événements sca_i^l (*Start Communication Active*) et scp_j^l (*Start Communication Passive*) correspondent respectivement au début actif et passif d'une communication com_l entre deux processus p_i et p_j. Les événements eca_i^l (*End Communication Active*) et ecp_i^l (*End Communication Passive*) sont, pour leur part, respectivement associés à la fin active et passive de cette communication. Cet exemple montre aussi que tous les événements ne sont pas comparables (relation d'ordre partiel) : l'événement scp_2^1 n'est pas comparable à l'événement ecp_1^0 car ils surviennent dans deux processus distincts et appartiennent à des actions de communication concurrentes.

Une propriété particulièrement intéressante du modèle de temps AST est de pouvoir être facilement étendu. Une extension de ce modèle de temps (relation d'ordre et fonction d'estampillage) a notamment été développée pour prendre en compte l'occurrence d'événements globaux tels que les fronts d'horloge d'un circuit synchrone (cf. paragraphe 4.1.4).

4.1.1 Définition d'un Ordre Partiel

Contrairement à un circuit synchrone, les événements survenant dans un circuit asynchrone ne peuvent pas être triés en fonction d'un signal global. La solution proposée par le modèle de temps AST est de modéliser le comportement d'un circuit asynchrone en fonction des mécanismes de communication locaux employés par ses composants (cf. paragraphe 4.1.1.1). Le principal intérêt de cette modélisation est de permettre d'ordonner partiellement les différents événements survenant dans un circuit asynchrone en respectant sa propriété d'insensibilité aux délais (cf. paragraphe 4.1.1.2).

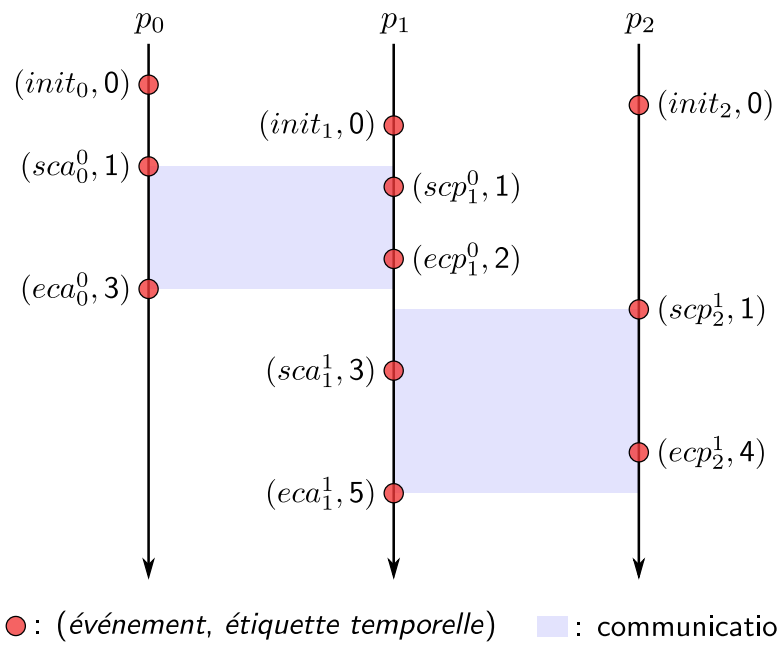


FIGURE 4.1 – *Exemple d'estampillage temporel d'un modèle de circuit asynchrone*

4.1.1.1 Représentation Evénementielle d'un Circuit Asynchrone

Avec le modèle de temps AST, le comportement d'un circuit asynchrone est représenté formellement par les trois ensembles suivants.

L'ensemble de ses processus $\mathbb{P} = \{p_0, p_1, \ldots\}$ Un processus p_i est défini par la séquence d'événements $(e_i, e'_i, \ldots)$ qui surviennent au cours de son exécution. Le premier et le dernier événement d'un processus p_i sont respectivement son initialisation $init_i$ et sa terminaison end_i.

L'ensemble de ses canaux de communication $\mathbb{C} = \{ch_0, ch_1, \ldots\}$ Un canal de communication ch_k est spécifié par un triplet (k, p_i, p_j) où k est un entier identifiant ce canal et p_i et p_j sont deux processus distincts qui peuvent communiquer via ce canal. Les processus p_i et p_j sont chargés respectivement d'initier et d'acquitter les communications réalisées via ce canal. Il est à noter qu'avec ce formalisme le sens des données communiquées par un canal n'est pas pertinent.

L'ensemble de ses communications $\mathbb{T} = \{com_0, com_1, \ldots\}$ Une communication com_l entre deux processus p_i et p_j est caractérisée par un quadruplet d'événements $(sca_i^l, scp_j^l, ecp_j^l, eca_i^l)$ et par le canal de communication ch_k employé. Les événements sca_i et eca_i marquent respectivement le début actif et la fin active de la communication com_l pour le processus p_i qui initie la communication. De même, les événements scp_j et ecp_j identifient respectivement le début passif et la fin passive de la communication com_l pour le processus p_j qui acquitte la communication.

Les propriétés de base des canaux de communication d'un circuit asynchrone étant d'être bloquant et sans mémoire, il est légitime de supposer qu'un modèle AST respecte l'hypothèse suivante :

Hypothèse 1 — Communication Synchrone
Lorsqu'un processus débute une communication, le prochain événement de ce processus est obligatoirement la fin de cette communication.

La figure 4.2 présente les différents événements survenant au cours d'une communication com_0 entre deux processus p_0 et p_1. Ces événements correspondent aux phases suivantes des protocoles à poignée de main employés par un circuit asynchrone :

- l'événement sca_0^0 (*Start Communication Active*) correspond à l'envoi d'une demande de communication et au début de l'attente de l'acquittement de cette demande de communication,
- l'événement scp_1^0 (*Start Communication Passive*) marque le début de l'attente d'une demande de communication, [1]
- l'événement ecp_1^0 (*End Communication Passive*) indique la réception d'une demande de communication et l'envoi de l'acquittement de cette demande de communication,
- l'événement eca_0^0 (*End Communication Active*) signale la réception de l'acquittement d'une demande de communication.

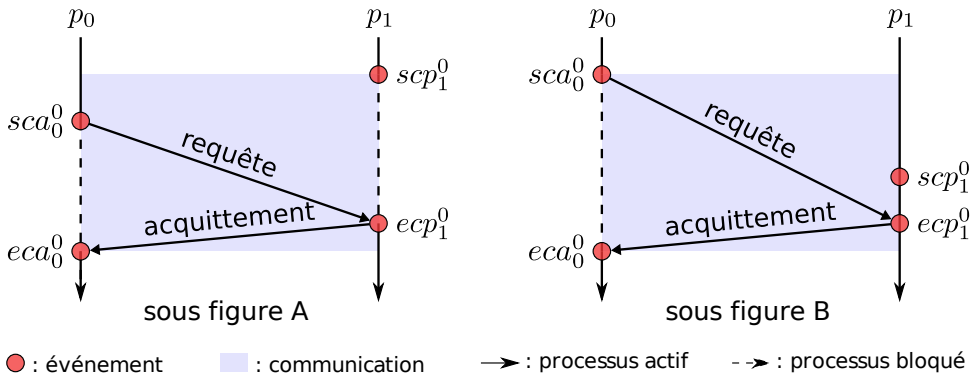


FIGURE 4.2 – *Représentation événementielle du protocole à poignée de main*

Lorsqu'un processus p_j désire communiquer avec un processus p_i via un canal $ch_k = (k, p_i, p_j)$, il peut utiliser le mécanisme de *probe* pour vérifier si p_i a initié une communication sur ch_k. L'action de *probe* est atomique et génère un et un seul des deux événements suivants :

- pp_j^k (*Positive Probe*) : cet événement, appelé « *probe* positif », survient si et seulement si p_i a initié une communication sur le canal ch_k (i.e. après l'envoi d'une demande de communication sur ch_k et avant l'émission de l'acquittement de cette communication),
- np_j^k (*Negative Probe*) : cet événement, appelé « *probe* négatif », se produit si et seulement si p_i n'a pas initié de communication sur le canal ch_k (i.e. après l'envoi de l'acquittement d'une communication sur ch_k et avant l'émission de la prochaine demande de communication sur ch_k).

Il est à noter que le mécanisme de *probe* ne peut être utilisé par un processus p_j avec un canal $ch_k = (k, p_i, p_j)$ que si p_j est chargé d'acquitter les communications de ch_k. Cette propriété, plus le fait que les communications sont synchrones (i.e. bloquantes), entraîne qu'une action de *probe* ne peut jamais survenir entre des événements marquant le début d'une demande de communication et l'envoi de l'acquittement de cette communication.

Une limitation du mécanisme de *probe* est de déterminer si il est positif ou négatif lorsqu'une communication est en cours d'initialisation. Avec ce formalisme, ceci revient à déterminer si un *probe* survient avant ou après l'événement marquant l'initialisation d'une

[1] Cet événement est signalé même si une demande de communication est déjà disponible (cf. sous figure 4.2.B)

communication. Pour être à même de répondre sans ambiguïté à cette question, ce modèle de temps effectue l'hypothèse suivante :

Hypothèse 2 — Non Simultanéité des Événements
Deux événements distincts ne peuvent jamais survenir exactement au même instant.

La figure 4.3 illustre le résultat de deux actions de *probe* (np_1^0 et pp_1^0) effectuées par le processus p_1 pour savoir si une communication a été initiée sur le canal $ch_0 = (0, p_0, p_1)$. Le résultat du premier *probe* (événement np_1^0) est négatif car le processus p_0 n'a pas encore initié une communication sur le canal ch_0 (événement sca_0^0). Au contraire le résultat du deuxième probe pp_1^0 est positif car il survient après sca_0^0 mais avant ecp_1^0 (acquittement de la communication).

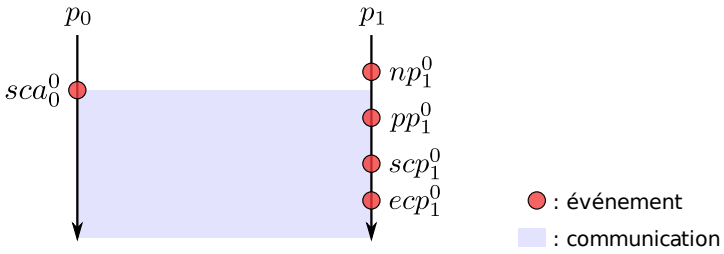


FIGURE 4.3 – *Représentation événementielle du mécansime de probe*

Pour ne pas rendre trop fastidieuse la description du comportement d'un circuit asynchrone, il est nécessaire de pouvoir effectuer plusieurs tâches en parallèle au sein d'un même processus. Par exemple, le langage de modélisation CHP ajoute un opérateur de composition parallèle au langage CSP qui lui sert de base.[2] Pour représenter le comportement de tels opérateurs, AST dispose des événements suivants :

- un événement de création de sous processus $csp_i^{n,n+1,\dots}$ (*Create Sub Processes*) qui apparaît lorsqu'un processus p_i crée les sous processus $p_n, p_{n+1}, \dots$ pour exécuter un ensemble de tâches en parallèle,
- un événement de terminaison de sous processus $esp_i^{n,n+1,\dots}$ (*End Sub Processes*) qui se manifeste lorsque les sous processus $p_n, p_{n+1}, \dots$ créés par un processus p_i se sont terminés.

Afin de préserver la cohérence de ce modèle de temps, un processus créant des sous processus doit respecter l'hypothèse suivante :

Hypothèse 3 — Sous Processus Bloquant
Un processus créant des sous processus est bloqué jusqu'à ce que ses sous processus se soient tous terminés.

Pour avoir la possibilité d'effectuer des communications en parallèle au sein d'un même processus, les sous processus d'un processus sont autorisés à accéder à ses canaux de communication. Pour éviter que des communications et des *probe* puissent être effectués en parallèle via un même canal, l'accès aux canaux de communication d'un processus par ses sous processus doit respecter l'hypothèse suivante :

Hypothèse 4 — Canal de Communication Séquentiel
L'accès à un canal de communication d'un processus est limité à un seul et même sous processus parmi l'ensemble des sous processus de ce processus.

[2] Le comportement d'un processus CSP standard est purement séquentiel.

La figure 4.4 présente comment un processus p_1 réalise en parallèle les deux communications com_0 et com_1 avec les deux processus p_0 et p_2. En réalité, le processus p_0 ne réalise pas directement ces communications mais les délègue à ses sous processus p_3 et p_4. La création et la terminaison de ces sous processus sont respectivement indiquées par les événements $csp_1^{3,4}$ et $esp_1^{3,4}$.

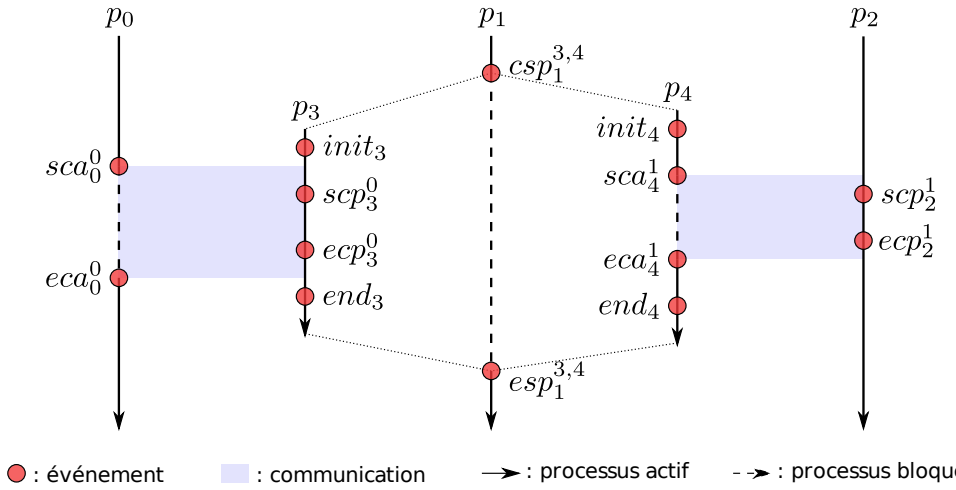


FIGURE 4.4 – *Représentation événementielle de deux communications parallèles*

4.1.1.2 Définition de la Relation « Survenue Avant »

Par définition, la séquence des événements $(e_i, e_i', \ldots)$ d'un processus p_i respecte leur ordre d'occurrence. Il est aussi raisonnable de supposer que deux événements distincts d'un même processus ne peuvent jamais survenir en même temps. La séquence des événements $(e_i, e_i', \ldots)$ définissant un processus p_i respecte donc une relation d'ordre total. L'objectif du modèle de temps AST est toutefois plus général car il consiste à définir une relation d'ordre sur les ensembles d'événements $\mathbb{E} = \{e, e', \ldots\}$ qui caractérisent le comportement d'un circuit asynchrone. A cette fin, le modèle de temps AST définit la relation « survenue avant » $\prec : \mathbb{E} \rightarrow \mathbb{E}$ qui respecte les propriétés de base d'un circuit asynchrone et qui est formellement définie dans ce paragraphe à l'aide de ces propriétés. La définition de la relation $\prec$ repose, en effet, essentiellement sur les dépendances entre les événements dues aux mécanismes de synchronisation employés par un circuit asynchrone (e.g. communication synchrone, *probe*, ...). Par exemple, l'événement marquant l'envoi d'une demande de communication survient avant l'événement marquant la réception de cette demande de communication (cf. propriété 6).

En plus de respecter les propriétés des circuits asynchrones, la relation $\prec$ doit aussi respecter les propriétés de la physique classique. Un événement ne peut donc pas survenir avant lui-même (cf. propriété 1) et la relation $\prec$ doit être transitive (cf. propriété 2). Toutefois ces différentes propriétés qui définissent la relation $\prec$ ne permettent pas de comparer tous les événements qui surviennent dans un circuit asynchrone (cf. definition 1). Par exemple, les événements d'initialisation des processus ne sont pas comparables entre eux (cf. figure 4.1). La relation $\prec$ définit donc un ordre partiel strict de $\mathbb{E}$ (ou préordre de $\mathbb{E}$).

Une caractéristique intéressante de la relation $\prec$ est de respecter la propriété d'insensibilité aux délais des circuits asynchrones DI, car les propriétés qui la définissent ne font

aucune hypothèse sur les délais qui séparent les événements appartenant à $\mathbb{E}$.

Définition 1 — Événements Concurrents
Deux événements e et e' sont dit concurrents, lorsque

$$e \not\prec e' \ \wedge \ e' \not\prec e$$

Propriété 1 — Irréflexivité
$\forall e \in \mathbb{E}, e \not\prec e$

Propriété 2 — Transitivité
$\forall e, e', e'' \in \mathbb{E}, (e \prec e' \wedge e' \prec e'') \Rightarrow e \prec e''$

Propriété 3 — Processus Séquentiel
Si e_i et e'_i sont deux événements distincts appartenant à la séquence des événements caractérisant un processus p_i, et que e_i apparaît avant e'_i dans cette séquence, alors

$$e_i \prec e'_i$$

Remarque. Cette propriété assure la séquentialité du comportement d'un processus. Elle garantit en outre que les événements d'un processus sont totalement ordonnés par la relation $\prec$ car la séquence des événements d'un processus est totalement ordonnée.

Propriété 4 — Initialisation d'un Processus
Si $init_i$ est le début d'un processus p_i, et que e_i est un événement de p_i autre que $init_i$, alors

$$init_i \prec e_i$$

Remarque. Cette propriété garantit que le premier événement survenant dans un processus est son initialisation.

Propriété 5 — Terminaison d'un Processus
Si end_i est la fin d'un processus p_i, et que e_i est un événement de p_i autre que end_i, alors

$$e_i \prec end_i$$

Remarque. Cette propriété assure que le dernier événement survenant dans un processus est sa terminaison.

Propriété 6 — Protocole à Poignée de Main
Si $(sca^l_i, scp^l_j, ecp^l_j, eca^l_i)$ est le quadruplet d'événements caractérisant une communication com_l entre les processus p_i et p_j, alors

$$sca^l_i \prec ecp^l_j \ \wedge \ scp^l_j \prec ecp^l_j \ \wedge \ ecp^l_j \prec eca^l_i$$

Remarque. Cette propriété vérifie que les protocoles à poignée de main employés par un circuit asynchrone sont bien respectés (cf. figure 4.2). Le fait que sca^l_i survienne avant ecp^l_j garantit que la requête d'une communication est reçue après son envoi. De même, requérir que ecp^l_j apparaisse avant eca^l_i assure que l'acquittement d'une communication est reçu après son envoi. Finalement, demander à scp^l_j de se manifester avant ecp^l_j interdit à une communication de se terminer avant d'avoir commencé.

Propriété 7 — *Probe* Positif
Si un processus p_h réalise un probe positif pp^k_h d'un canal ch_k, car une communication $com_l = (ch_k, (sca^l_i, scp^l_j, ecp^l_j, eca^l_i))$ a été initiée, alors

$$sca^l_i \prec pp^k_h \prec ecp^l_j$$

Remarque. L'objectif de cette propriété est d'assurer que le résultat du *probe* d'un canal est positif si et seulement si une communication a bien été initiée sur ce canal et qu'elle n'a pas encore été acquittée.

Propriété 8 — *Probe* Négatif
Si un processus p_h effectue un probe négatif np_h^k d'un canal ch_k, et si $com_l = (ch_k, (sca_i^l, scp_j^l, ecp_j^l, eca_i^l))$ est une communication via ce canal, alors

$$np_h^l \prec sca_i^l \ \vee \ ecp_j^l \prec np_h^l$$

Remarque. Le rôle de cette propriété est de garantir que le *probe* d'un canal est négatif si et seulement si aucune communication n'a été initialisée sur ce canal. L'hypothèse 2 garantit que la valeur d'un *probe* peut toujours être déterminée à l'aide de cette propriété et de la propriété 7.

Propriété 9 — Création de Sous Processus
Si $csp_i^{n,n+1,\dots}$ est un événement d'un processus p_i qui indique la création de ses sous processus $p_n, p_{n+1}, \dots$, et que les événements $init_n, init_{n+1}, \dots$ marquent respectivement leur initialisation, alors

$$csp_i^{n,n+1,\dots} \prec init_n \ \wedge \ csp_i^{n,n+1,\dots} \prec init_{n+1} \ \wedge \ \dots$$

Remarque. Cette propriété garantit que les événements d'un sous processus sont postérieurs à sa création.

Propriété 10 — Terminaison de Sous Processus
Si $esp_i^{n,n+1,\dots}$ est un événement d'un processus p_i qui indique la terminaison de ses sous processus $p_n, p_{n+1}, \dots$, et que les événements $end_n, end_{n+1}, \dots$ marquent respectivement leur fin, alors

$$end_n \prec esp_i^{n,n+1,\dots} \ \wedge \ end_{n+1} \prec esp_i^{n,n+1,\dots} \ \wedge \ \dots$$

Remarque. Cette propriété garantit que l'événement marquant la terminaison d'une action de création de sous processus est postérieure à la terminaison de ces sous processus.

4.1.2 Fonction d'Estampillage Temporel

La date t à laquelle se déroule un événement e est définie avec le modèle de temps AST par la fonction date : $\mathbb{E} \rightarrow \mathbb{N}$. Cette fonction, qui représente le temps logique du système modélisé, est définie par l'ensemble des règles présentées dans ce paragraphe. L'objectif de ces règles est d'associer à chaque événement survenant dans un circuit asynchrone une étiquette temporelle représentant sa date d'apparition. La valeur d'une étiquette temporelle est calculée par ces règles en fonction des étiquettes temporelles qu'elles ont précédemment allouées. Afin de pouvoir attribuer une date à n'importe quel événement, une règle est définie pour chaque type d'événement qui peut survenir.

Règle 1 — Événement Nul
Si e est un événement qui ne s'est pas produit, alors

$$\text{date}(e) = 0$$

Remarque. Le rôle de cette règle n'est pas d'être directement utilisée, mais de faciliter la définition des autres règles. Elle est, par exemple, nécessaire pour associer, en toute circonstance, une date à l'événement np_h^k de la règle 5.

Règle 2 — Initialisation d'un processus
Si $init_i$ est le début du procesus p_i, et que ce processus n'est pas un sous processus, alors

$$\text{date}(init_i) = 0$$

Règle 3 — Initialisation d'un sous processus
Si $init_m$ est le début d'un sous procesus p_m qui a été créé par l'événement $cti_i^{n,\ldots,m,\ldots}$ du processus p_i, alors

$$\text{date}(init_m) = \text{date}(cti_i^{n,\ldots,m,\ldots}) + 1$$

Règle 4 — Terminaison d'un Processus
Si end_i est la fin du procesus p_i, et que le_i (Last Event) est le dernier événement de p_i avant end_i, alors

$$\text{date}(end_i) = \text{date}(le_i) + 1$$

Règle 5 — Début Actif de Communication
Si sca_i^l est le début actif d'une communication com_l entre les processus p_i et p_j via un canal ch_k, et que np_h^k est le dernier probe négatif de ce canal avant sca_i^l, et que le_i est le dernier événement de p_i avant sca_i^l, alors

$$\text{date}(sca_i^l) = \max(\text{date}(le_i), \text{date}(np_h^k)) + 1$$

Règle 6 — Début Passif de Communication
Si scp_j^l est le début passif d'une communication com_l entre les processus p_i et p_j, et que le_j est le dernier événement de p_j avant scp_j^l, alors

$$\text{date}(scp_j^l) = \text{date}(le_j) + 1$$

Règle 7 — Fin Passive de Communication
Si ecp_j^l est la fin passive d'une communication $com_l = (ch_k, (sca_i^l, scp_j^l, ecp_j^l, eca_i^l))$ entre les processus p_i et p_j via le canal ch_k, alors

$$\text{date}(ecp_j^l) = \max(\text{date}(scp_j^l), \text{date}(sca_i^l)) + 1$$

Règle 8 — Fin Active de Communication
Si eca_i^l est la fin active d'une communication $com_l = (ch_k, (sca_i^l, scp_j^l, ecp_j^l, eca_i^l))$ entre les processus p_i et p_j, alors

$$\text{date}(eca_i^l) = \text{date}(ecp_j^l) + 1$$

Règle 9 — *Probe* Positif
Si pp_h^k est un probe positif d'un canal ch_k, car une communication $com_l = (ch_k, (sca_i^l, scp_j^l, ecp_j^l, eca_i^l))$ a été initiée sur ce canal, et que le_h est le dernier événement de p_h avant pp_h^k, alors

$$\text{date}(pp_h^k) = \max(\text{date}(sca_i^l), \text{date}(le_h)) + 1$$

Règle 10 — *Probe* Négatif
Si np_h^k est un probe négatif d'un canal ch_k, et que le_h est le dernier événement de p_h avant np_h^k, alors

$$\text{date}(np_h^k) = \text{date}(le_h) + 1$$

Règle 11 — Création de Sous Processus
Si $csp_i^{n,n+1,\ldots}$ est un événement du processus p_i qui marque la création de ses sous processus $p_n, p_{n+1}, \ldots$, et que le_i est le dernier événement de p_i avant $csp_i^{n,n+1,\ldots}$, alors

$$\text{date}(csp_i^{n,n+1,\ldots}) = \text{date}(le_i) + 1$$

Règle 12 — Terminaison de Sous Processus
Si $esp_i^{n,n+1,\ldots}$ *est un événement du processus* p_i *qui marque la terminaison de ses sous processus* $p_n, p_{n+1}, \ldots$, *et que* $end_n, end_{n+1}, \ldots$ *sont les derniers événements de ces sous processus, alors*

$$\text{date}(esp_i^{n,n+1,\ldots}) = \max(\text{date}(end_n), \text{date}(end_{n+1}), \ldots) + 1$$

La figure 4.5 présente comment ces règles permettent de calculer les dates auxquelles surviennent les différents événements d'une simulation d'un modèle ASC. Ce modèle est constitué des deux modules FOO et BAR qui communiquent via le canal ch_0. Les comportements des processus ASC `foo::proc` et `bar::proc` sont respectivement représentés, avec le modèle de temps AST, par les processus p_0 et p_1. La date à laquelle survient l'événement ecp_1^0 est, par exemple, calculée en utilisant la règle 7 de la manière suivante :

$$\begin{aligned} \text{date}(ecp_1^0) &= \max(\text{date}(scp_1^0), \text{date}(sca_0^0)) + 1 \\ &= \max(3, 1) + 1 = 4 \end{aligned}$$

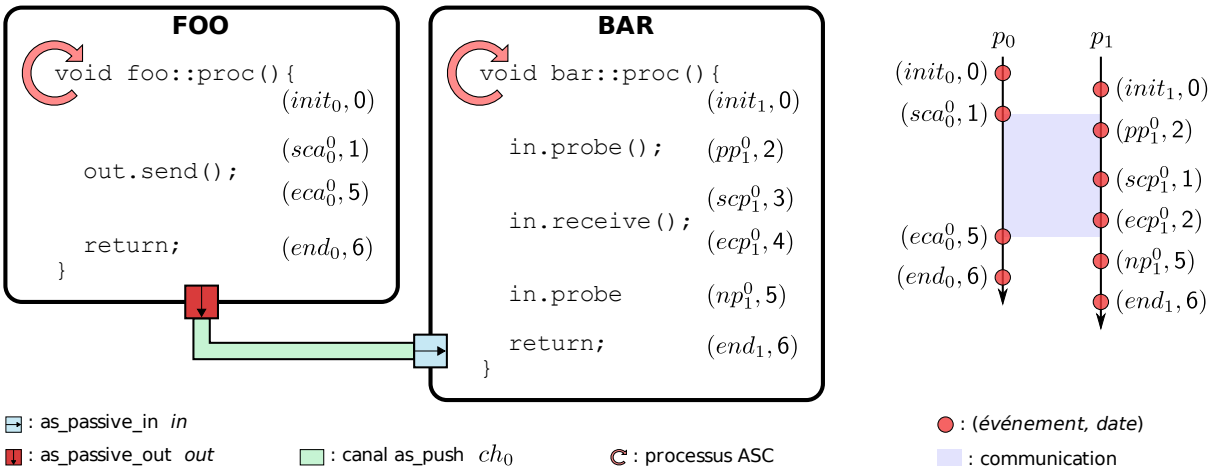


FIGURE 4.5 – *Estampillage temporel des communications d'un modèle ASC*

4.1.3 Cohérence du Modèle de Temps

Comme l'explique [Lam78], la fonction date : $\mathbb{E} \to \mathbb{N}$ respecte la relation d'ordre partiel strict $\prec : \mathbb{E} \to \mathbb{E}$ si la condition suivante est respectée :

Condition 1 — Condition d'Horloge

$$\forall e, e' \in \mathbb{E}(e \prec e' \Rightarrow \text{date}(e) < \text{date}(e'))$$

La démonstration de la validité de cette condition s'effectue en deux étapes. La première étape se limite à prouver que la condition d'horloge est vraie pour des modèles dont les processus ne créent pas de sous processus (cf. paragraphe 4.1.3.1). La deuxième étape généralise cette « condition d'horloge restreinte » au système employant des opérateurs de composition parallèle (cf. paragraphe 4.1.3.2).

4.1.3.1 Condition d'Horloge Restreinte

L'énoncé de la « condition d'horloge restreinte » est identique à celui de la condition d'horloge initiale à l'exception qu'elle est valable sur les ensembles d'événements $\mathbb{E}^* \subseteq \mathbb{E}$ qui définissent le comportement d'un circuit asynchrone n'employant pas de sous processus (i.e. $\mathbb{E}^*$ ne contient pas d'événement de création et de terminaison de sous processus). Pour démontrer cette condition, il suffit de prouver que les propriétés définissant la relation d'ordre $\prec$ sont préservées par la fonction date.

Lemme 1. *Si e_i et e'_i sont deux événements consécutifs d'un processus p_i tel que $e_i \prec e'_i$, alors*

$$\text{date}(e_i) < \text{date}(e'_i)$$

Preuve. Soit e_f et e'_f deux événements consécutifs d'un processus p_f, tel que $e_f \prec e'_f$. Par définition e'_f ne peut pas être l'événement marquant le début de ce processus (contradiction avec le fait que e_f soit apparu avant e'_f).
Si e'_f est l'événement marquant la fin du processus p_i, alors ce lemme est vérifiée, car la règle 4 implique que $\text{date}(e'_f) \geq \text{date}(e_f) + 1$.
De même, les règles 5, 6, 9 et 10 permettent d'obtenir la même inégalité si e'_f est respectivement un événement marquant :

- le début actif d'une communication $com_l = (ch_k, (sca^l_i, scp^l_j, ecp^l_j, eca^l_i))$ où $e'_f = sca^l_i$,
- le début passif d'une communication $com_l = (ch_k, (sca^l_i, scp^l_j, ecp^l_j, eca^l_i))$ où $e'_f = scp^l_j$,
- un *probe* positif,
- un *probe* négatif.

Si e'_f est la fin passive d'une communication $com_l = (ch_k, (sca^l_i, scp^l_j, ecp^l_j, eca^l_i))$ où $e'_f = ecp^l_j$, alors l'hypothèse 1 implique que $e_f = scp^l_j$. Dans ce cas, le lemme est vérifié, car la règle 7 permet de conclure que $\text{date}(e'_f) \geq \text{date}(e_f) + 1$.
De même, si e'_f est la fin active d'une communication $com_l = (ch_k, (sca^l_i, scp^l_j, ecp^l_j, eca^l_i))$ où $e'_f = eca^l_i$, alors l'hypothèse 1 implique que $e_f = sca^l_i$. Les règles 8 et 7 permettent alors de déduire que $\text{date}(e'_f) = \text{date}(ecp^l_j) + 1 \geq \text{date}(e_f) + 2$. □

Condition 2 — Processus Séquentiel
Si e_i et e'_i sont deux événements distincts d'un processus p_i tel que $e_i \prec e'_i$, alors

$$\text{date}(e_i) < \text{date}(e'_i)$$

Preuve. Soit $\mathbb{S}_n$ l'ensemble des sous séquences d'événements de longueur n de l'ensemble des processus $\mathbb{P}^* \subseteq \mathbb{P}$ n'employant pas de sous processus. On pose comme hypothèse d'induction que si e et e' sont respectivement les premiers et les derniers éléments d'une séquence $s_n = (e, \ldots, e') \in \mathbb{S}_n$, alors $\text{date}(e) < \text{date}(e')$.

Base. Pour $n = 2$, l'hypothèse d'induction est directement vérifiée avec le lemme 1.

Induction. Soit n un entier > 2 et $s_n = (e, \ldots, e', e'') \in \mathbb{S}_n$. Par hypothèse d'induction $\text{date}(e) < \text{date}(e')$ car la sous séquence $s_{n-1} = (e, \ldots, e')$ préfixe de s_n appartient à $\mathbb{S}_{n-1}$. De plus, le lemme 1 implique $\text{date}(e') < \text{date}(e'')$, et donc que l'hypothèse d'induction est vraie au rang n. □

Condition 3 — Initialisation et Terminaison d'un Processus
Si $init_i$ et end_i sont respectivement le début et la fin d'un processus p_i, et que e_i est un événement p_i autre que $init_i$ ou end_i, alors

$$\text{date}(init_i) < \text{date}(e_i) < \text{date}(end_i)$$

PREUVE. Cette propriété se déduit directement de la condition 2 car les événements de début et fin de processus sont le premier et le dernier élément de la séquence d'événements d'un processus (cf. propriété 4 et 5). □

Condition 4 — Protocole à Poignée de Main
Si $(sca_i^l, scp_j^l, ecp_j^l, eca_i^l)$ *est le quadruplet d'événements caractérisant une communication* com_l *entre les processus* p_i *et* p_j*, alors*

$$\text{date}(sca_i^l) < \text{date}(ecp_j^l) \ \wedge \ \text{date}(scp_j^l) < \text{date}(ecp_j^l) \ \wedge \ \text{date}(ecp_j^l) < \text{date}(eca_i^l)$$

PREUVE. Les règles 7 et 8 permettent de facilement démontrer la validité de cette propriété. □

Condition 5 — *Probe* Positif
Si un processus p_h *réalise un probe positif* pp_h^k *d'un canal* ch_k *car une communication* $com_l = (ch_k, (sca_i^l, scp_j^l, ecp_j^l, eca_i^l))$ *a été initiée, alors*

$$\text{date}(sca_i^l) < \text{date}(pp_h^k) < \text{date}(ecp_j^l)$$

PREUVE. Soit pp_h^k un *probe* positif du canal ch_k car des processus p_i et p_j ont initié une communication $com_l = (ch_k, (sca_i^l, scp_j^l, ecp_j^l, eca_i^l))$ via ce canal.
D'après la règle 9, on obtient directement que $\text{date}(sca_i^l) < \text{date}(pp_h^k)$.
D'après l'énoncé de la condition d'horloge restreinte, un processus ne peut pas créer de sous processus. La mécanisme de *probe* implique donc que un seul et même processus peut acquitter une communication ou effectuer un *probe* d'un canal de communication donné (cf. page 74), et par conséquent que les processus p_j et p_h sont un seul et même processus. La propriété 7 implique que $pp_h^k \prec ecp_j^l$, et donc la condition 2 implique que $\text{date}(pp_h^k) < \text{date}(ecp_j^k)$. □

Condition 6 — *Probe* Négatif
Si un processus p_h *effectue un probe négatif* np_h^k *d'un canal* ch_k*, et si* $com_l = (ch_k, (sca_i^l, scp_j^l, ecp_j^l, eca_i^l))$ *est une communication via ce canal, alors*

$$\text{date}(np_h^l) < \text{date}(sca_i^l) \ \vee \ \text{date}(ecp_j^l) < \text{date}(np_h^l)$$

PREUVE. Soit np_h^k un *probe* négatif du canal ch_k et soient $(sca_i^l, scp_j^l, ecp_j^l, eca_i^l)$ les événements d'une communication com_l via ce canal. Comme on l'a démontré pour la condition 5, il est facile de démontrer que les processus p_j et p_h sont, en fait, un seul et même processus et que seul ce processus peut effectuer un *probe* du canal ch_k.
D'après la définition du mécanisme de *probe* (cf. propriété 8) : $np_h^k \prec sca_i^l \vee ecp_j^l \prec np_h^k$.
Dans un premier temps, on suppose que $np_h^k \prec sca_i^l$. Si $np_h^{k'}$ est le dernier *probe* négatif avant sca_l^i, alors la condition 2 et la règle 5 implique que $\text{date}(np_h^k) < \text{date}(np_h^{k'}) < \text{date}(sca_i^l)$.
Dans un deuxième temps, on suppose que $ecp_j^l \prec np_h^k$. La condition 2 implique donc que $\text{date}(ecp_j^l) < \text{date}(np_h^k)$. □

4.1.3.2 Condition d'Horloge

Pour prouver que la condition d'horloge est valide, il faut démontrer d'une part que les conditions 7 et 8 sont vraies et d'autre part, il faut vérifier que les propriétés de la condition d'horloge restreinte sont toujours vérifiées sur $\mathbb{E}$.

Condition 7 — Création de Sous Processus
Si $csp_i^{n,n+1,\dots}$ *est un événement d'un processus* p_i *qui indique la création de ses sous processus* $p_n, p_{n+1}, \dots$, *et que les événements* $init_n, init_{n+1}, \dots$ *marquent respectivement leur début, alors*

$$\text{date}(csp_i^{n,n+1,\dots}) < \text{date}(init_n) \ \wedge \ \text{date}(csp_i^{n,n+1,\dots}) < \text{date}(init_{n+1}) \ \wedge \ \dots$$

Preuve. La règle 3 permet de facilement démontrer la validité de cette propriété. □

Condition 8 — Terminaison de Sous Processus
Si $esp_i^{n,n+1,\dots}$ *est un événement d'un processus* p_i *qui indique la terminaison de ses sous processus* $p_n, p_{n+1}, \dots$, *et que les événements* $end_n, end_{n+1}, \dots$ *marquent respectivement leur fin, alors*

$$\text{date}(end_n) < \text{date}(esp_i^{n,n+1,\dots}) \ \wedge \ \text{date}(end_{n+1}) < \text{date}(esp_i^{n,n+1,\dots}) \ \wedge \ \dots$$

Preuve. La règle 12 permet de facilement démontrer la validité de cette propriété. □

Preuve — Condition 4 : *Protocole à Poignée de Main*
Démonstration identique à celle de la condition d'horloge restreinte (cf. page 82). □

Preuve — Lemme 1
Soit $\mathbb{P}_n$ l'ensemble des processus composés de n sous processus. On pose comme hypothèse d'induction que si e_i et e'_i sont deux événements consécutifs d'un processus $p_i \in \mathbb{P}_n$ tels que $e_i \prec e'_i$, alors $\text{date}(e_i) < \text{date}(e'_i)$.

Base. Pour $n = 0$, l'hypothèse d'induction est directement vérifiée par le lemme 1 de la condition d'horloge restreinte.

Induction. Soit n un entier > 0 et soient e_i et e'_i deux événements consécutifs d'un processus $p_i \in \mathbb{P}_n$ tels que $e_i \prec e'_i$. Si e'_i n'est pas un événement de création ou de terminaison de processus, alors l'inégalité $\text{date}(e_i) < \text{date}(e'_i)$ peut être démontrée avec la même preuve que pour le lemme 1 de la condition d'horloge restreinte.
Si e'_i est un événement de création de sous processus, alors la règle 11 implique que $\text{date}(e_i) < \text{date}(e'_i)$.
Si $e'_i = esp_i^{m,m+1,\dots}$ est un événement marquant la fin des sous processus $p_m, p_{m+1}, \dots$, alors l'hypothèse 3 implique que $e_i = csp_i^{m,m+1,\dots}$ est l'événement qui a créé ces sous processus. Soit $init_m, init_{m+1}, \dots$ et $end_m, end_{m+1}, \dots$ les événements marquants respectivement le début et la fin des processus $p_m, p_{m+1}, \dots$, alors les règles 12 et 3 impliquent respectivement que :

- $\max(\text{date}(end_m), \text{date}(end_{m+1}), \dots) < \text{date}(e'_i)$,
- $\text{date}(e_i) < \text{date}(init_m) \wedge \text{date}(e_i) < \text{date}(init_{m+1}) \wedge \dots$.

Par hypothèse d'induction, les processus p_m, $p_{m+1}, \dots$ respectent le lemme 1, et donc par conséquence les conditions 2 et 3 (cf. preuves suivantes). Au final, ceci implique que $\text{date}(init_m) < \text{date}(end_m) \wedge \text{date}(init_{m+1}) < \text{date}(end_{m+1}) \wedge \dots$, et donc que $\text{date}(e_i) < \text{date}(e'_i)$. L'hypothèse d'induction est donc bien vérifiée au rang n. □

Preuve — Condition 2 : *Processus Séquentiel*
Démonstration identique à celle de la condition d'horloge restreinte (cf. page 81). □

Preuve — Condition 3 : *Initialisation et Terminaison d'un Processus*
Démonstration identique à celle de la condition d'horloge restreinte (cf. page 82). □

Lemme 2. *Si e_i et un événement d'un processus p_i survenant avant un événement e_j d'un processus p_j, et que p_j est un sous processus de p_i, et que $csp_i^{j,\dots}$ est l'événement marquant la création du sous processus p_j, alors*

$$e_i \prec csp_i^{j,\dots}$$

Preuve. Soient p_i et p_j deux processus tels que p_j soit un sous processus de p_i. Soient e_i et e_j des événements des processus p_i et p_j tels que $e_i \prec e_j$. Soient $csp_i^{j,\dots}$ et $esp_i^{j,\dots}$ les événements de p_i marquant la création et la terminaison du sous processus p_j.
Pour prouver par l'absurde que $e_i \prec csp_i^{j,\dots}$, on suppose que $e_i \not\prec csp_i^{j,\dots}$.
Si $e_i \not\prec csp_i^{j,\dots}$, alors $csp_i^{j,\dots} \prec e_i$ car les événements de p_i sont totalement ordonnés par la relation $\prec$ (cf. propriété 3). L'hypothèse 3 implique alors que $csp_i^{j,\dots} \prec esp_i^{j,\dots} \prec e_i$.
Si end_j marque la fin du processus p_j, alors les propriétés 5 et 10 impliquent que $e_j \prec end_j \prec esp_i^{j,\dots}$.
La propriété 2 permet finalement d'obtenir la contradiction $e_j \prec e_i$ □

Lemme 3. *Si e_i est un événement d'un processus p_i survenant après un événement e_j d'un processus p_j, et que p_j est un sous processus de p_i, et que $esp_i^{j,\dots}$ est l'événement marquant la terminaison du sous processus p_j, alors*

$$esp_i^{j,\dots} \prec e_i$$

Preuve. Soient p_i et p_j deux processus tels que p_j soit un sous processus de p_i. Soient e_i et e_j des événements des processus p_i et p_j tels que $e_j \prec e_i$. Soient $csp_i^{j,\dots}$ et $esp_i^{j,\dots}$ les événements de p_i marquant la création et la terminaison du sous processus p_j.
Pour prouver par l'absurde que $esp_i^{j,\dots} \prec e_i$, on suppose que $esp_i^{j,\dots} \not\prec e_i$.
Si $esp_i^{j,\dots} \not\prec e_i$, alors $e_i \prec esp_i^{j,\dots}$ car les événements de p_i sont totalement ordonnés par la relation $\prec$ (cf. propriété 3). L'hypothèse 3 implique alors que $e_i \prec csp_i^{j,\dots} \prec esp_i^{j,\dots}$.
Si $init_j$ marque le début du processus p_j, alors les propriétés 4 et 9 impliquent que $csp_i^{j,\dots} \prec init_j \prec e_j$.
La propriété 2 permet finalement d'obtenir la contradiction $e_i \prec e_j$ □

Définition 2 — Ancêtre d'un Processus
Un processus p_i est un ancêtre d'un processus p_j si et seulement si p_j est un sous processus de p_i, ou si il existe un processus p_k qui soit un ancêtre de p_j lui même sous processus de p_i.

Lemme 4. *Les événements d'un processus et ceux de ses ancêtres sont totalement ordonnés par la relation $\prec$.*

Preuve. Soit $\mathbb{P}_n$ l'ensemble des processus ayant n ancêtres. Soit $\mathbb{A}_n$ l'ensemble des ancêtres des processus appartenant à $\mathbb{P}_n$. Soit $\mathbb{L}_n$ l'ensemble des événements des processus appartenant à $\mathbb{P}_n \cup \mathbb{A}_n$. On pose comme hypothèse d'induction que $\forall e, e' \in \mathbb{L}_n (e \prec e' \vee e' \prec e)$.

Base. Par définition les événements d'un processus sont totalement ordonnés par la relation $\prec$ (cf. propriété 3). l'hypothèse d'induction est donc vraie au rang 0.

Induction. Soient n un entier > 0 et p_j un processus appartenant à $\mathbb{P}_n$. Soit p_i le processus qui a créé p_j (i.e. p_j est un sous processus de p_i). Soient csp_i^j et esp_i^j les événements de p_i marquant respectivement la création et la terminaison du processus p_j. Soient $init_j$ et end_j les événements de p_j indiquant respectivement le début et la fin de ce processus.
Si A_j est l'ensemble des événements des ancêtres du processus p_j alors par hypothèse d'induction A_j est totalement ordonné par la relation $\prec$, et donc l'hypothèse 3 implique

$\forall e \in A_j \setminus \{csp_i^j, esp_i^j\}(e \prec csp_i^j \vee esp_i^j \prec e)$.
Les propriétés 9 et 10 permettent respectivement de conclure que $csp_i^j \prec init_j$ et que $end_j \prec esp_i^j$. Les propriétés 4 et 5 impliquent alors que $\forall e_j \in p_j(csp_i^j \prec e_j \prec esp_i^j)$.
L'hypothèse d'induction est donc vraie au rang n, car les résultats précédents et la propriété 2 permettent de conclure que $\forall e \in A_j, \forall e_j \in p_j(e \prec e_j \vee e_j \prec e)$. □

Lemme 5. *Si p_i est un ancêtre d'un processus p_j, et que $init_i$ et end_i sont respectivement les événements marquant le début et la fin de p_i, et que e_j est un événement de p_j, alors*

$$init_i \prec e_j \prec end_i$$

Preuve. Soit $\mathbb{P}_n$ l'ensemble des processus ayant n ancêtres. On pose comme hypothèse d'induction que si e_j est un événement d'un processus p_j appartenant à $\mathbb{P}_n$, et que $init_i$ et end_i sont les événements de début et de fin d'un processus p_i qui est un ancêtre de p_j, alors $init_i \prec e_j \prec end_i$.

Base. Soit p_j un processus appartenant à $\mathbb{P}_1$ et soit p_i l'ancêtre de p_j (i.e. p_j est un sous processus de p_i). Soit e un événement de p_i.
Les propriétés 4, 5, 9 et 10 permettent de déduire que $init_i \prec csp_i^j \prec init_j \prec e_j \prec end_j \prec esp_i^j \prec end_i$. La propriété 2 permet finalement de conclure que $init_i \prec e_j \prec end_i$, et donc que l'hypothèse d'induction est vraie au rang 1.

Induction. Soit n un entier > 1, et soit p_j un processus appartenant à $\mathbb{P}_n$, et soient p_i et p_k deux ancêtres de p_j tels que p_k soit un sous processus de p_i.
Si $init_k$ et end_k sont les événements marquant respectivement le début et la fin de p_k, alors par hypothèse d'induction $\forall e_j \in p_j(init_k \prec e_j \prec end_k)$.
Si $init_i$ et end_i sont les événements marquant respectivement le début et la fin de p_i, alors les propriétés 4, 5, 9, 10 impliquent que $init_i \prec csp_i^k \prec init_k \wedge end_k \prec esp_i^k \prec end_i$.
L'hypothèse d'induction est donc vraie au rang n, car les résultats précédents et la propriété 2 impliquent $\forall e_j \in p_j(init_i \prec e_j \prec end_i)$. □

Lemme 6. *Si e_i est un événement d'un processus p_i survenant avant un événement e_j d'un processus p_j, et que p_i est un ancêtre du processus p_j, alors*

$$\text{date}(e_i) < \text{date}(e_j)$$

Preuve. On pose comme hypothèse d'induction que si p_i est un ancêtre d'un processus p_j appartenant à $\mathbb{P}_n$, alors $\forall e_i \in p_i, \forall e_j \in p_j(e_i \prec e_j \Rightarrow \text{date}(e_i) < \text{date}(e_j))$.

Base. Soit p_j un processus appartenant à $\mathbb{P}_1$ et soit p_i l'ancêtre de p_j (i.e. p_j est un sous processus de p_i). Soient e_i et e_j deux événements appartenant respectivement aux processus p_i et p_j tels que $e_i \prec e_j$.
Si $csp_i^{j,\cdots}$ est l'événement marquant la création du sous processus p_i, alors le lemme 2 implique que $e_i \prec csp_i^j$ et la condition 2 implique donc que $\text{date}(e_i) < \text{date}(csp_i^{j,\cdots})$.
Si $init_j$ est l'événement marquant le début du processus p_j, alors les conditions 7 et 3 impliquent que $\text{date}(csp_i^{j,\cdots}) < \text{date}(init_j) < \text{date}(e_j)$, et donc que l'hypothèse d'induction est vraie un rang 1.

Induction. Soit n un entier > 1, et soit p_j un processus appartenant à $\mathbb{P}_n$, et soient p_i et p_k deux ancêtres de p_j tels que p_k soit un sous processus de p_i. Soient e_i et e_j des événements des processus p_i et p_j tels que $e_i \prec e_j$.
Si $init_k$ est l'événement marquant le début du processus p_k, alors le lemme 5 implique que $init_k \prec e_j$, et donc par hypothèse d'induction que $\text{date}(init_k) < \text{date}(e_j)$.
Pour prouver par l'absurde que $e_i \prec init_k$, on suppose que $e_i \not\prec init_k$, et donc d'après le

lemme 4 que $init_k \prec e_i$. Si $esp_i^{k,\cdots}$ est l'événement marquant la terminaison de p_k, alors le lemme 3 implique que $esp_i^{k,\cdots} \prec e_i$. Si end_k est l'événement marquant la fin du processus p_k, alors la propriété 10 implique que $end_k \prec esp_i^{k,\cdots}$. Le lemme 5 implique, en outre, que $e_j \prec end_k$. La propriété 2 permet finalement d'obtenir la contradiction $e_j \prec e_i$.
Etant donné que $e_i \prec init_k$, si $csp_i^{k,\cdots}$ est l'événement marquant la création du processus p_k, alors le lemme 2 et la propriété 9 impliquent que $e_i \prec csp_i^{k,\cdots} \prec init_k$.
L'hypothèse d'induction est donc vraie au rang n, car les conditions 2 et 7 permettent d'affirmer que $\text{date}(e_i) < \text{date}(csp_i^{k,\cdots}) < \text{date}(init_k)$, et donc que $\text{date}(e_i) < \text{date}(e_j)$. □

Lemme 7. *Si e_i et un événement d'un processus p_i survenant après un événement e_j d'un processus p_j, et que p_i est un ancêtre du processus p_j, alors*

$$\text{date}(e_j) < \text{date}(e_i)$$

Preuve. On pose comme hypothèse d'induction que si p_i est un ancêtre d'un processus p_j appartenant à $\mathbb{P}_n$, alors $\forall e_i \in p_i, \forall e_j \in p_j (e_j \prec e_i \Rightarrow \text{date}(e_j) < \text{date}(e_i))$.

Base. Soit p_j un processus appartenant à $\mathbb{P}_1$ et soit p_i l'ancêtre de p_j (i.e. p_j est un sous processus de p_i). Soient e_i et e_j deux événements appartenant respectivement aux processus p_i et p_j tels que $e_j \prec e_i$.
Si $esp_i^{j,\cdots}$ est l'événement marquant la terminaison du sous processus p_i, alors le lemme 3 implique que $esp_i^j \prec e_i$ et la condition 2 implique donc que $\text{date}(esp_i^{j,\cdots}) < \text{date}(e_i)$.
Si end_j est l'événement marquant la fin du processus p_j, alors les conditions 3 et 8 impliquent que $\text{date}(e_j) < \text{date}(end_j) < \text{date}(esp_i^{j,\cdots})$, et donc que l'hypothèse d'induction est vraie un rang 1.

Induction. Soit n un entier > 1, et soit p_j un processus appartenant à $\mathbb{P}_n$, et soient p_i et p_k deux ancêtres de p_j tels que p_k soit un sous processus de p_i. Soient e_i et e_j des événements des processus p_i et p_j tels que $e_j \prec e_i$.
Si end_k est l'événement marquant la fin du processus p_k, alors le lemme 5 implique que $e_j \prec end_k$, et donc par hypothèse d'induction que $\text{date}(e_j) < \text{date}(end_k)$.
Pour prouver par l'absurde que $end_k \prec e_i$, on suppose que $end_k \not\prec e_i$, et donc d'après le lemme 4 que $e_i \prec end_k$. Si $csp_i^{k,\cdots}$ est l'événement marquant la création de p_k, alors le lemme 2 implique que $e_i \prec csp_i^{k,\cdots}$. Si $init_k$ est l'événement marquant le début du processus p_k, alors la propriété 9 implique que $csp_i^{k,\cdots} \prec init_k$. Le lemme 5 implique, en outre, que $init_k \prec e_j$. La propriété 2 permet finalement d'obtenir la contradiction $e_i \prec e_j$.
Etant donné que $end_k \prec e_i$, si $esp_i^{k,\cdots}$ est l'événement marquant la terminaison du processus p_k, alors la propriété 10 et le lemme 3 impliquent que $end_k \prec esp_i^{k,\cdots} \prec e_i$.
L'hypothèse d'induction est donc vraie au rang n, car les conditions 8 et 2 permettent d'affirmer que $\text{date}(init_k) < \text{date}(esp_i^{k,\cdots}) < \text{date}(e_i)$, et donc que $\text{date}(e_j) < \text{date}(e_i)$. □

Preuve — Condition 5 : *Probe positif*
Soit pp_h^k un *probe* positif du canal ch_k car des processus p_i et p_j ont initié une communication $com_l = (ch_k, (sca_i^l, scp_j^l, ecp_j^l, eca_i^l))$ via ce canal.
D'après la règle 9, on obtient directement que $\text{date}(sca_i^l) < \text{date}(pp_h^k)$.
D'après l'hypothèse 4, le processus p_h est, soit identique au processus p_j, soit un ancêtre du processus p_j, soit un descendant du processus p_j (i.e. p_j est un ancêtre de p_h).
Si p_h et p_j sont un seul et même processus, alors la démonstration est identique à celle de la condition d'horloge restreinte (cf. page 82).
Si p_h est un ancêtre de p_j, alors le lemmes 6 implique $\text{date}(pp_h^k) < \text{date}(ecp_j^l)$, car par

définition du mécanisme de *probe* $pp_h^k \prec ecp_j^l$ (cf. propriété 7).
De même, si p_j est un ancêtre de p_h, alors le lemme 7 implique $\text{date}(pp_h^k) < \text{date}(ecp_j^l)$. □

Preuve — Condition 6 : *Probe Négatif*
Soit np_h^k un *probe* négatif du canal ch_k et soient $(sca_i^l, scp_j^l, ecp_j^l, eca_i^l)$ les événements d'une communication com_l via ce canal. D'après l'hypothèse 4, le processus p_h est, soit identique au processus p_j, soit un ancêtre du processus p_j, soit un descendant du processus p_j (i.e. p_j est un ancêtre de p_h).
La fin de la démonstration est alors similaire à celle de la condition d'horloge restreinte à la différence que l'on emploie les lemmes 6 et 7 à la place de la condition 2 lorsque p_h et p_j ne sont pas les mêmes processus. □

4.1.4 Interfaçage avec le Monde Synchrone

Pour valider un réseau sur puce asynchrone reliant des composants synchrones, il est nécessaire de simuler leurs comportements conjointement. Le modèle de temps AST de base ne permet pas de dater les événements survenant dans un tel système car il ne dispose pas d'événement de synchronisation globale pour modéliser les fronts d'horloge d'un circuit synchrone. Pour pallier cette limitation, un modèle AST est défini par un nouvel ensemble de processus $\mathbb{P}^+ = \mathbb{P} \cup \mathbb{P}^\Delta$, où $\mathbb{P}^\Delta = \{p_\Delta, p'_\Delta, \ldots\}$ est l'ensemble des processus globaux. Un processus global p_Δ est constitué d'une séquence d'événements globaux $(ge_\Delta, ge'_\Delta, \ldots)$ dont les occurrences sont visibles par les autres processus de $\mathbb{P}^+$. Un processus global peut notamment être utilisé pour modéliser l'horloge d'un composant synchrone en employant ses événements pour représenter les fronts d'horloge.

Avec cette extension du modèle de temps AST, un processus $p \in \mathbb{P}^+$ est caractérisé par la séquence des événements survenant au cours de son exécution et de ceux qui surviennent au cours de l'exécution des processus globaux. Etant donné que deux événements distincts ne peuvent jamais survenir simultanément (cf. hypothèse 2), la séquence des événements qui définit un processus $p \in \mathbb{P}^+$ respecte toujours une relation d'ordre total.

Pour prendre en compte l'occurrence des événements globaux, la relation « survenue avant » $\prec$ est étendue à l'ensemble $\mathbb{E}^+ \supseteq \mathbb{E}$ qui caractérise le comportement des composants asynchrones d'un circuit GALS. Cette relation est définie sur $\mathbb{E}^+$ à l'aide des propriétés la définissant sur $\mathbb{E}$ et des propriétés suivantes.

Propriété 11. *Si e et ge_Δ (Global Event) sont des événements survenant dans deux processus distincts $p \in \mathbb{P}$ et $p_\Delta \in \mathbb{P}^\Delta$, et que e apparaît avant ge_Δ dans la séquence des événements caractérisant p, alors*

$$e \prec ge_\Delta$$

Propriété 12. *Si e et ge_Δ sont des événements survenant dans deux processus distincts $p \in \mathbb{P}$ et $p_\Delta \in \mathbb{P}^\Delta$, et que ge_Δ apparaît avant e dans la séquence des événements caractérisant p, alors*

$$ge_\Delta \prec e$$

De même que pour la relation $\prec$, la fonction d'estampillage date est étendue à l'ensemble $\mathbb{E}^+$ en modifiant certaine de ses règles et en ajoutant la règle 13. Les règles définissant la fonction date vue au paragraphe 4.1.2 doivent être modifiées car l'étiquette temporelle allouée à un événement $e \in \mathbb{E}$ doit être cohérente avec celles allouées aux événements globaux. Par exemple, pour la règle 4, si on conserve les mêmes hypothèses, et si

$le_\Delta, le'_\Delta, \ldots$ sont les derniers événements étant survenus dans les processus $p_\Delta, p'_\Delta, \ldots \in \mathbb{P}_\Delta$ avant end_i, alors

$$\text{date}(end_i) = \max(\text{date}(le_i), \text{date}(le_\Delta), \text{date}(le'_\Delta), \ldots) + 1$$

Le détail des modifications des autres règles (à l'exception de la règle 1 qui ne doit pas être modifiée) du paragraphe 4.1.2 n'est pas explicitement indiqué car il est identique à celui de cette règle .

Règle 13 — Synchronisation Globale
Si ge_Δ est un événement global survenant dans un processus global p_Δ, et que $le, le', \ldots$ sont les derniers événements survenant dans les processus $p, p', \ldots \in \mathbb{P}^+$ avant ge_Δ, alors

$$\text{date}(ge_\Delta) = \max(\text{date}(le), \text{date}(le'), \ldots) + 1$$

De même que précédemment, la fonction d'estampillage date : $\mathbb{E}^+ \to \mathbb{N}$ respecte la relation d'ordre partiel strict $\prec : \mathbb{E}^+ \to \mathbb{E}^+$ si l'extension de la condition d'horloge sur $\mathbb{E}^+$ (appelée « condition d'horloge étendue ») est vraie. Pour démontrer la validité de cette condition, il faut en premier lieu prouver que les propriétés de la condition d'horloge sont toujours valides sur $\mathbb{E}^+$. En deuxième lieu, il faut montrer que les conditions 9 et 10 sont vraies.

Preuve — Lemme 1
Soient e et e' deux événements consécutifs d'un processus p, tels que e survienne avant e'.
Si $p \in \mathbb{P}^\Delta$, alors la règle 13 implique que $\text{date}(e) < \text{date}(e')$.
Si $p \in \mathbb{P}$ et si il n'y pas d'événement global qui est survenu entre e et e', alors la condition d'horloge implique que $\text{date}(e) < \text{date}(e')$.
Si $p \in \mathbb{P}$ et si il y a n événements globaux consécutifs $ge^1, \ldots, ge^n$ qui sont survenus entre e et e', alors la règle 13 implique que $\text{date}(e) < \text{date}(ge^1) < \ldots < \text{date}(ge^n)$. En outre, les modifications apportées aux règles pour les événements non globaux impliquent que $\text{date}(ge^n) < \text{date}(e')$. □

Preuve — Conditions 2 et 3
Démonstration identique que pour la condition d'horloge. □

Preuve — Conditions 4, 5, 6, 7 et 8
Les démonstrations de ces propriétés s'effectuent, de manière similaire à celle du lemme 1, en se servant des propriétés démontrées pour la condition d'horloge et en considérant les cas où des événements globaux surviennent entre les événements considérés. □

Condition 9. *Si e et ge_Δ sont des événements de deux processus distincts $p \in \mathbb{P}$ et $p_\Delta \in \mathbb{P}^\Delta$, et que $e \prec ge_\Delta$, alors*

$$\text{date}(e) < \text{date}(ge_\Delta)$$

Preuve. Soient e et ge_Δ des événements de deux processus distincts $p \in \mathbb{P}$ et $p_\Delta \in \mathbb{P}_\Delta$ tels que $e \prec e_\Delta$.
Si e est le dernier événement de p survenant avant ge_Δ, alors la règle 13 implique que $\text{date}(e) < \text{date}(ge_\Delta)$.
Si e n'est pas le dernier événement de p survenant avant ge_Δ, alors il existe un événement $le \in p$ qui remplit cette condition et qui se produit après e. La condition 2 et la règle 13 permettent alors de conclure que $\text{date}(e) < \text{date}(le) < \text{date}(ge_\Delta)$. □

Condition 10. *Si e et ge_Δ sont des événements de deux processus distincts $p \in \mathbb{P}$ et*

$p_\Delta \in \mathbb{P}^\Delta$, et que $ge_\Delta \prec e$, alors

$$\text{date}(ge_\Delta) < \text{date}(e)$$

PREUVE. Soient e et ge_Δ des événements de deux processus distincts $p \in \mathbb{P}$ et $p_\Delta \in \mathbb{P}_\Delta$ tels que $ge_\Delta \prec e$.
Si ge_Δ est le dernier événement de p_Δ survenant avant e, alors les modifications apportées aux règles pour les événements non globaux impliquent que $\text{date}(ge_\Delta) < \text{date}(e)$.
Si ge_Δ n'est pas le dernier événement de p_Δ survenant avant e, alors il existe un événement $lge_\Delta \in p_\Delta$ (*Last Global Event*) qui remplit cette condition et qui se produit après ge_Δ. La condition 2 et les modifications apportées aux règles pour les événements non globaux permettent alors de conclure que $\text{date}(ge_\Delta) < \text{date}(lge_\Delta) < \text{date}(e)$. □

4.1.5 Les Limitations et les Améliorations

Si la fonction d'estampillage date permet d'assigner une date respectant la relation $\prec$, elle perd néanmoins de l'information. Par exemple sur la figure 4.6, l'étiquette temporelle assignée à l'événement scp_2^1 ne montre pas explicitement qu'il est concurrent avec les événements scp_1^0 et ecp_1^0. Une solution, proposée par l'équipe VDS [3] (*Verification and Modeling of Digital Systems*), pour résoudre ce problème est d'utiliser des intervalles indiquant la date à laquelle a put survenir un événement. Sur cet exemple l'étiquette temporelle de l'événement scp_2^1 serait alors égale à l'intervalle $[1, 3]$.

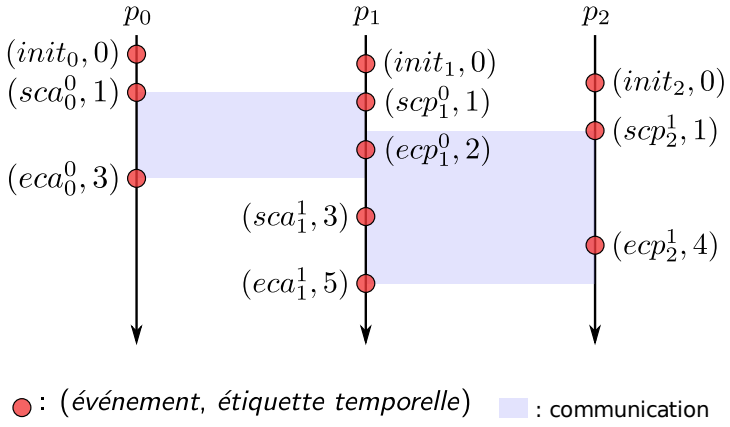


FIGURE 4.6 – *Estampillage temporel d'événements concurrents*

4.2 La Validation de Modèle ASC

Une des limitations de la bibliothèque ASC est de ne pas pouvoir utiliser convenablement les facilités de traçage standard de la bibliothèque SystemC (cf. paragraphe 2.1.3.4). Pour remédier à ce problème, la bibliothèque ASC définit ses propres facilités de traçage qui sont basées sur le modèle de temps AST (cf. paragraphe 4.2.1). Toutefois, les fichiers de trace générés par ces nouvelles facilités de traçage ne sont pas compatibles avec les outils de conception standards. Un outil a donc été développé pour convertir au format VCD les fichiers de trace obtenus avec ces nouvelles facilités de traçage (cf. paragraphe 4.2.2).

[3] http://tima.imag.fr/vds

Pour vérifier qu'un modèle ASC est fonctionnellement correct, il est indispensable de s'assurer que son comportement est correct quel que soit l'ordonnancement de ses processus (cf. paragraphe 2.1.3.4). Un correctif de l'ordonnanceur du simulateur SystemC OSCI a donc été développé pour le rendre non déterministe (cf. paragraphe 4.2.3).

4.2.1 La Génération de Trace avec ASC

Les facilités de traçage ASC permettent d'enregistrer l'activité des canaux de communication qui sont employés pour relier les composants d'un modèle ASC. Ces facilités de traçage respectent le standard IEEE–1666 (cf. pages 371–374 de [IEE06]) qui définit comment doivent être étendues les fonctions de traçage SystemC. Un fichier de trace est créé avec la fonction `as_create_ast_file` qui prend en paramètre le nom du fichier où les traces sont enregistrées. Cette fonction retourne aussi un pointeur qui est utilisé par la fonction `as_trace` pour indiquer les canaux ASC à observer. Finalement, un fichier de trace doit être fermé en se servant de la fonction `as_close_ast_trace_file`. Le programme 4.1 montre comment l'activité d'un canal `ch` peut être enregistrée dans un fichier `trace.ast` à l'aide de ces différentes fonctions.

```
class foo: public as_trace_generic {
   void serialize(ostream& a_ostream) const {
      a_ostream << "Hello world!!!";
   }
};

class top: public as_container {
   public:
      as_push< foo > ch;
      ...
};

int sc_main(int argc, char* argv) {
   // Create a new trace file
   as_trace_file* l_file_pt = as_create_ast_trace_file("trace.ast");

   // Record generic data type
   l_trace_file_pt->add_generic_type(foo());

   // Elaboration of the system
   top l_top("top");
   as_trace(l_file_pt, l_top.ch, "ch");

   // Simulation of the system
   sc_start();
   as_close_ast_trace_file(l_file_pt);

   return 0;
}
```

Programme 4.1 – *Exemple d'utilisation des fonctions de traçage ASC*

Le type des données échangées par un canal ASC est un paramètre générique de la classe qui le définit. Si ces facilités de traçage permettent de tracer l'activité de n'importe quel canal ASC, elles ne permettent néanmoins d'enregistrer les valeurs échangées au cours

d'une communication que si elles sont du type C++ suivant : `bool`, `char`, `short`, `int`, `long`, `long long`, `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, `unsigned long long`, `float`, `double`.

La bibliothèque ASC dispose en outre d'une classe de base abstraite `as_trace_generic` qui peut être employée pour enregistrer les valeurs de n'importe quel type de données. Cette classe possède une méthode virtuelle pure `serialize` qui permet d'enregistrer une chaîne de caractères représentant la valeur d'une donnée. Il est en outre nécessaire d'indiquer, à l'aide de la méthode `add_generic_type`, les types de données qui utilisent cette classe de base et dont les valeurs doivent être enregistrées dans un fichier de trace. Par exemple, dès qu'une communication est effectuée via le canal `ch`, la classe `foo` du programme 4.1 utilise la classe `as_trace_generic` pour enregistrer la chaine de caractères `Hello world!!!` dans le ficher de trace `trace.ast`.

En plus des valeurs échangées, les facilités de traçage ASC enregistrent les événements (et leurs dates d'occurrence) qui surviennent lorsque des processus accèdent à des canaux de communication observés. Le calcul des étiquettes temporelles, qui sont associées à chaque événement ASC, est effectué en appliquant les règles définissant la fonction d'estampillage étendue (cf. paragraphe 4.1.4).

4.2.2 Génération de Fichiers de Trace VCD

Avec ces facilités de traçage, il est difficile de générer directement un fichier VCD, car le modèle de temps AST ne date pas les événements de manière chronologique au cours d'une simulation et le format de trace VCD exige que les dates des changements de valeurs de ses signaux soient ordonnées chronologiquement. La solution à ce problème, qui a été développée au cours de cette thèse, a été d'employer un format de trace intermédiaire qui soit adapté au modèle de temps AST et qui puisse être facilement converti vers du VCD. Ce format de trace, aussi appelé AST (*Asynchronous SystemC Trace*), est un format textuel dont la syntaxe est détaillée dans l'annexe B.1.

Le format de trace AST n'est pas compatible avec les outils de conception standards car il est spécifique à la bibliothèque ASC. L'outil de conversion ast2vcd permet cependant de pouvoir utiliser des visionneuses de chronogrammes (de l'anglais *wave form viewer*) standards, telles que GTKWave [4], pour visualiser le résultat d'une simulation d'un modèle ASC. L'activité d'un canal ASC est représentée dans un fichier VCD obtenu avec cet outil par les signaux suivants :

- `data` : représente la valeur des données transférées au cours d'une communication,
- `sca`, `scp`, `eca`, `ecp` : représentent les événements définissant une communication via un canal ASC,
- `probe` : représente le résultat du*probe* d'un canal ASC.

La figure 4.7 montre le chronogramme des signaux VCD représentant l'activité du canal `ch` qui relie les deux modules FOO et BAR. Ce chronogramme a été obtenu en utilisant l'outil GTKWave avec la trace VCD présentée dans l'annexe B.2. Cette trace a elle-même été générée par l'outil ast2vcd à partir de la trace AST (cf. annexe B.3) résultant de la simulation du modèle ASC présenté sur cette figure. Finalement, le diagramme temporel de cette figure représente comment sont ordonnés, avec le modèle de temps AST, les principaux événements survenant dans les processus p_0 et p_1 des modules FOO et BAR.

[4] http://home.nc.rr.com/gtkwave

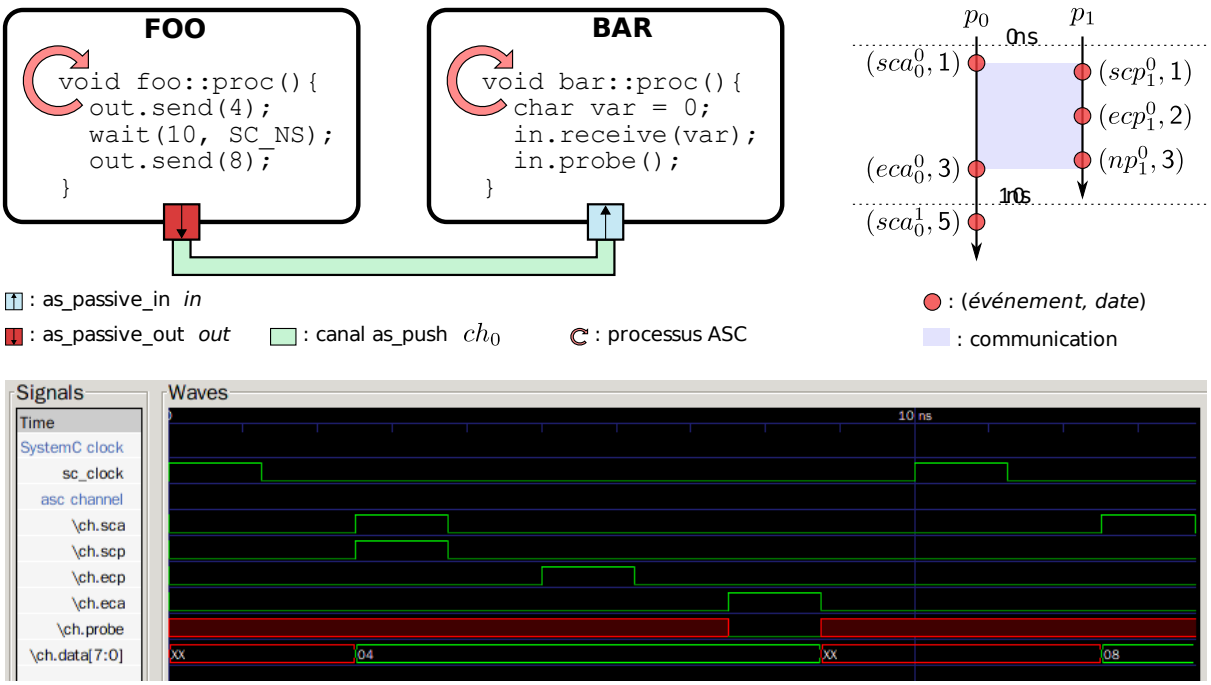


FIGURE 4.7 – *Chronogramme représentant l'activité d'un canal ASC*

Tous les événements représentés sur la figure 4.7, à l'exception de sca_0^1, surviennent à l'instant 0 ns (nanoseconde) du temps de simulation SystemC. Sur la trace VCD obtenue, ces événements surviennent toutefois à des dates différentes pour représenter la valeur de leurs étiquettes temporelles AST. En effet, l'outil ast2vcd sépare par des ϵ pas de temps les événements qui ont des étiquettes temporelles différentes, mais qui surviennent à un temps de simulation SystemC identique. L'horloge du simulateur SystemC est représentée par le signal `sc_clock` afin de savoir à quel instant du temps de simulation un événement survient. La valeur du ϵ pas de temps est en effet calculée en garantissant que la date VCD d'un événement survient toujours après son temps de simulation et avant le prochain front du signal `sc_clock`.

4.2.3 Simulateur SystemC Non Déterministe

Si la spécification de l'ordonnancement des processus SystemC au cours de la phase d'évaluation d'un delta-cycle n'est pas définie par le standard IEEE–1666 (cf. page 16 de [IEE06]), il n'en reste pas moins que l'implémentation de référence du simulateur OSCI est déterministe.

Cette implémentation utilise deux pseudo-fifos pour gérer l'ensemble des processus qui sont prêts à être exécutés. Une de ces deux pseudo-fifos est dédiée aux processus sans contexte et l'autre aux processus avec contexte. Ces pseudo-fifos sont divisées en deux listes appelées respectivement `get_list` et `push_list`. La `get_list` est utilisée par l'ordonnanceur pour sélectionner un nouveau processus à exécuter. La `push_list` est employée pour insérer des nouveaux processus éligibles dans une pseudo-fifo.

Au cours d'une phase d'évaluation d'un delta-cycle, tous les processus qui se trouvent dans la `get_list` de la pseudo-fifo des processus sans contexte sont dans un premier temps exécutés. Dans un deuxième temps, tous les processus avec contexte se trouvant dans la `get_list` de l'autre pseudo-fifo sont exécutés. Finalement, les processus se trouvant

dans les `push_list` sont transférés dans leurs `get_list` respectives. Ces trois étapes sont répétées en boucle au cours d'une phase d'évaluation jusqu'à ce que les `get_list` soient vides au début de la première étape. Cet algorithme d'ordonnancement des processus est donc bien complètement déterministe et ne permet donc pas de vérifier correctement la fonctionnalité d'un modèle ASC (cf. paragraphe 2.1.3.4).

Comme l'illustre la figure 4.8, le correctif de cet ordonnanceur fusionne les deux pseudo-fifos en une seule file à priorité pour que l'ordre d'exécution des processus soit aléatoire. Une nouvelle classe de base commune à tous les processus (i.e. avec ou sans contexte) a été développée pour définir leurs priorités d'exécution. Cette priorité est en particulier utilisée pour définir où doit être ajouté un processus dans la file à priorité lorsqu'il devient éligible. Ce générateur aléatoire est créé par la fabrique de générateur aléatoire de la bibliothèque ASC (cf. paragraphe 3.1.5), ce qui permet de facilement rejouer une simulation. Finalement, la priorité d'un processus est employée comme critère d'élection d'un processus par ce nouvel ordonnanceur : le processus élu parmi un ensemble de processus éligibles est le processus ayant la plus forte priorité.

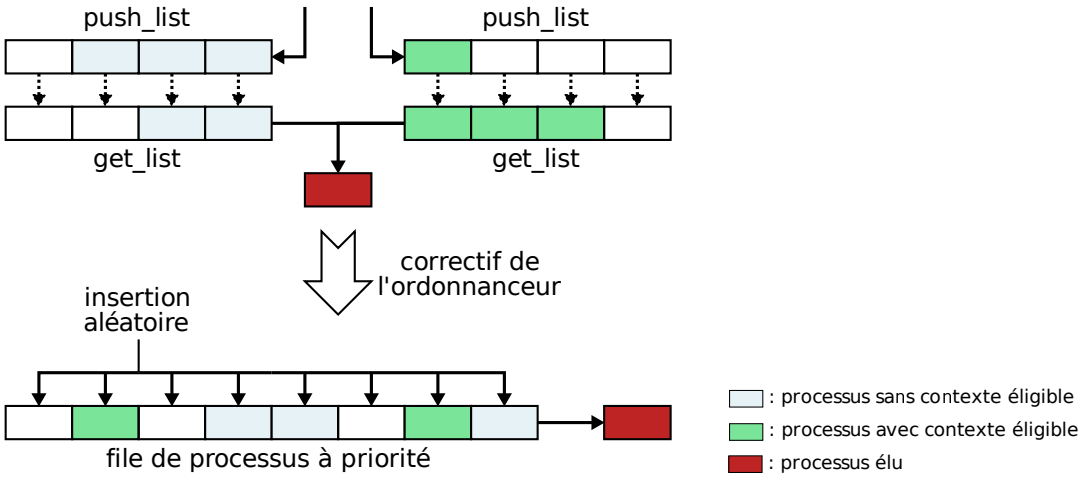


FIGURE 4.8 – *Ordonnanceur du simulateur SystemC non déterministe*

Une autre solution, très intéressante, à ce problème est présenté dans [HMMCM06, Hel07]. Une méthode et des outils sont proposés pour générer efficacement différents ordonnancements. Cette méthode utilise des techniques de réduction d'ordre partiel qui évitent de générer plusieurs fois des ordonnancements amenant le système au même état final.

4.3 Validation des Modèles ASC d'Octagons

Ces différents outils, dédiés à la validation de modèles ASC, ont été utilisés avec les modèles de NoC Octagon vus au paragraphe 3.2.1. Des modules TLM_PV de génération et de consommation de trafic ont en outre été développés pour avoir la possibilité de tester facilement différents motifs de communication. Ces modules TLM_PV sont connectés aux modèles ASC d'Octagons via des *transactors* basés sur celui présenté au paragraphe 3.2.1. Leurs comportements sont en outre paramétrés à l'aide d'un fichier de configuration dont la syntaxe est donnée dans l'annexe C.1

Ces plateformes de simulation, ont dans un premier temps, été employées pour s'assurer que les paquets étaient bien acheminés à leur destination en respectant l'algorithme de

routage de l'Octagon. La figure 4.9 montre comment la version à commutation de paquets de l'Octagon ASC achemine les paquets d'une transaction de lecture entre le producteur connecté au nœud de routage Nord (N) et le consommateur connecté au nœud de routage Sud Ouest (SW). Les valeurs affichées sur le chronogramme de cette figure pour les paquets de requête (*Request Packets*) ne sont pas pertinentes car elles sont indéfinies dans le cas d'une transaction de lecture. Un point intéressant, qui est mis en évidence par ce chronogramme, est que le paquet de requête et celui de réponse empruntent des nœuds de routage différents pour arriver à leur destination.

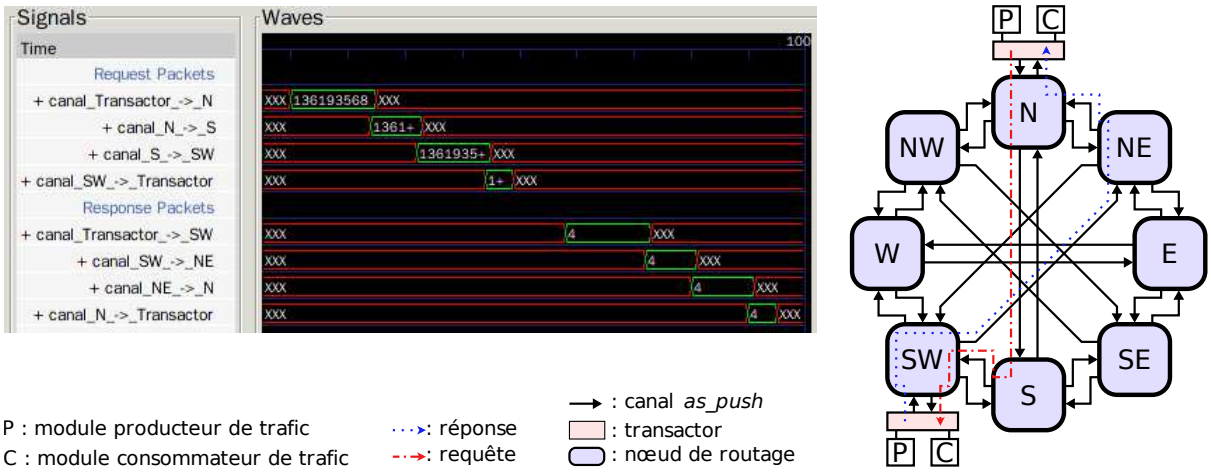


FIGURE 4.9 – *Résultat de simulation de l'Octagon à commutation de paquets*

Dans un deuxième temps, des latences ont été introduites dans ces plateformes de simulation pour, par exemple, vérifier le mécanisme de réservation des nœuds de routage qui est employé par la version de l'Octagon à commutation de circuits. La figure 4.10 présente deux chronogrammes obtenus en simulant, cette dernière version de l'Octagon, avec le fichier de configuration de l'annexe C.2 et en ajoutant les latences suivantes :

- 2 ns pour les canaux de communication : 1 ns pour la phase de requête et 1 ns pour la phase d'acquittement,
- 4 ns pour la fonction de routage des nœuds de routage,
- 5 ns pour qu'un consommateur de trafic envoie la réponse à une requête.

Le chronogramme se situant au bas de la figure 4.10 montre que, contrairement à la version à commutation de paquets, la réponse d'une transaction emprunte obligatoirement le chemin inverse de celui suivi par la requête. La trace du canal NE → N indique, en outre, que l'accès au routeur N est réservé jusqu'à la terminaison de la transaction N → SW.

Le chronogramme disposé en haut de la figure 4.10 détaille les raisons du blocage de la communication entre les routeurs NE et N. Si une communication est bien initiée par le routeur NE sur le canal NE → N (cf. trace SCA de ce canal) au bout de 5 ns (cf. marqueur A), il n'en reste pas moins que le routeur N ne peut y répondre (cf. marqueur D de la trace SCP du canal NE → N) avant d'avoir renvoyé la réponse (cf. marqueur D de la trace ECA du canal N → Transactor) à la requête de son *transactor* (cf. canal Transactor → N du précédent chronogramme). En effet, ce routeur est bloqué par la communication qu'il a initiée sur le canal S → N (cf. marqueur B de la trace SCP de ce canal) pour recevoir la réponse à la requête de son *transactor*.

Dans un dernier temps, les producteurs de trafic ont été modifiés pour générer des transactions de manière aléatoire (destination et latence entre deux transactions). Au

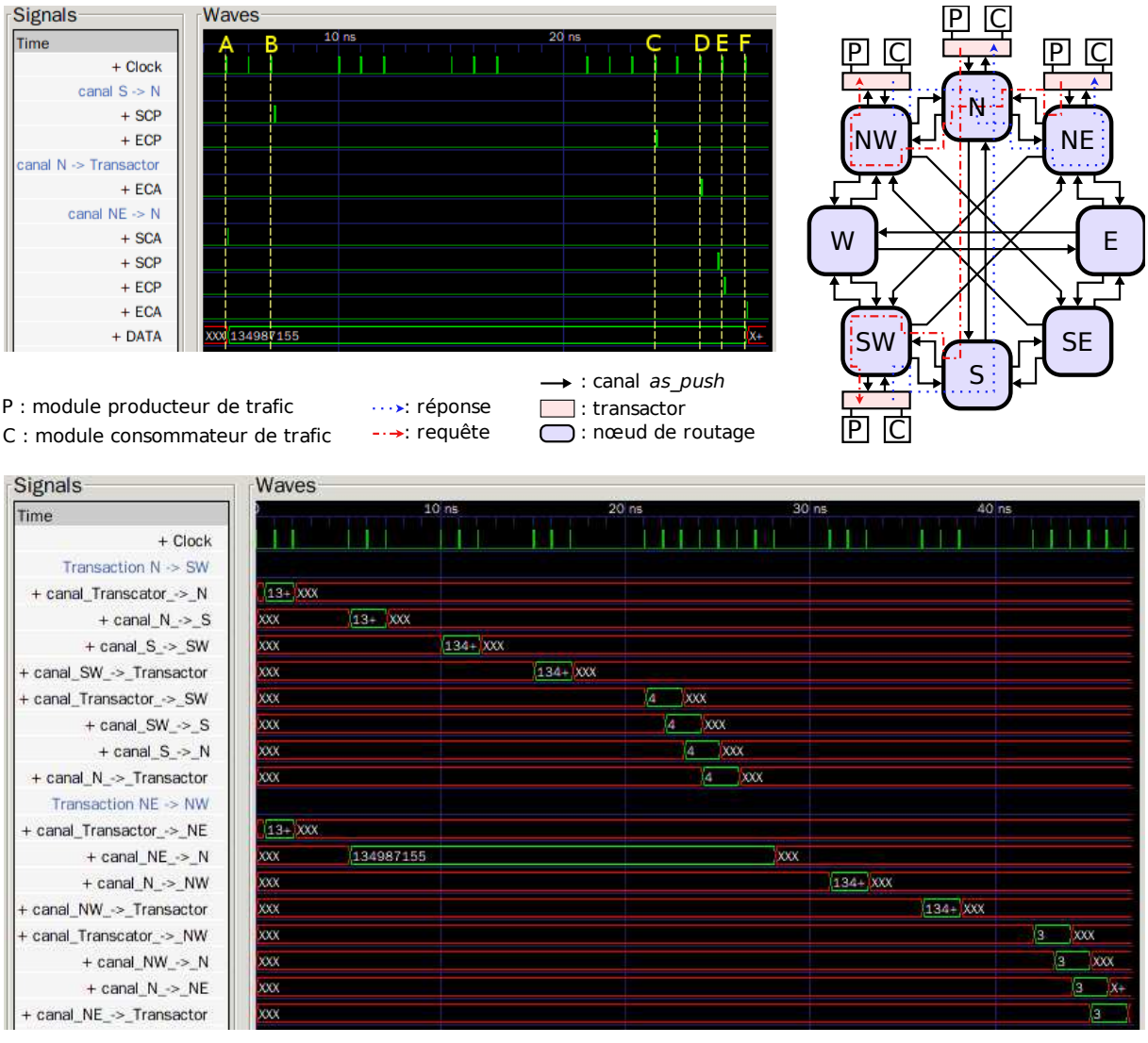


FIGURE 4.10 – *Résultat de simulation de l'Octagon à commutation de circuits*

cours de cette étape, il est apparu que des interblocages statiques pouvaient apparaître dans ce réseau sur puce. Une solution pour résoudre ce problème serait d'utiliser des canaux virtuels (cf. paragraphe 1.2.2.3).

Un point fort de ces facilités de traçage, qui a été constaté au cours de ce travail de validation, est de pouvoir être utilisées aussi bien à un niveau macroscopique qu'à un niveau microscopique. Au niveau macroscopique, la possibilité d'afficher les données échangées par les canaux a permis d'étudier facilement le trafic des paquets à l'intérieur de ces réseaux sur puce. Au niveau microscopique, les différents signaux caractérisant l'état d'un canal ont servi à comprendre finement comment sont synchronisées les différentes communications entre les nœuds de routage et comment des interblocages statiques peuvent survenir.

4.4 Les Limitations et les Améliorations

Pour rendre les traces plus lisibles, il serait pertinent que le signal VCD, qui représente les données échangées par un canal au cours d'une communication, prenne une valeur spéciale lorsqu'une communication a débuté, mais que la valeur de la données échangée n'est pas encore connue. Cela permettrait notamment d'obtenir le chronogramme de la figure 4.11 à partir de l'exemple d'Octagon à commutation de circuits présenté au paragraphe 4.3. Contrairement au chronogramme initial (cf. figure 4.10), le mécanisme de réservation employé par cette version de l'Octagon est explicitement modélisé.

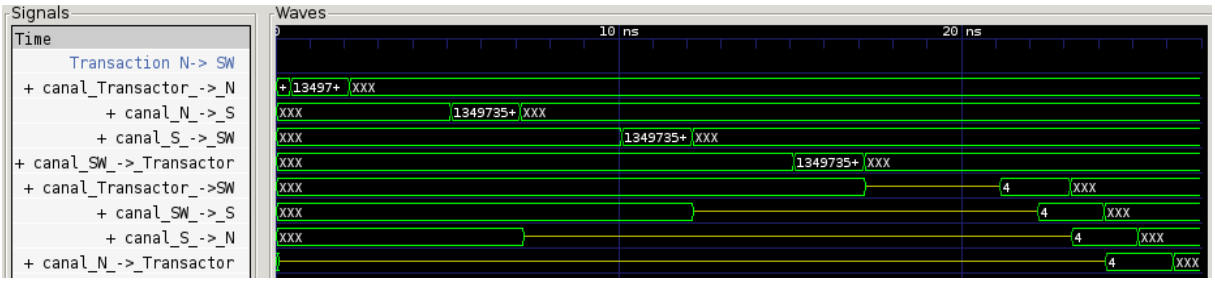


FIGURE 4.11 – *Chronogrammes avec signaux VCD améliorés*

Une autre amélioration possible, serait d'avoir la possibilité de générer d'autres formats de trace, tel que celui adopté par le cadre d'application TLM_TAC. Ce dernier est particulièrement bien adapté pour enregistrer l'activité des canaux de communication d'un circuit asynchrone, car la notion de transaction sur lequel il est basé est similaire aux protocoles à poignée de main employés par les canaux des circuits asynchrones. En outre, ce format de trace permet d'enregistrer facilement le contenu d'une structure de données complexe. Toutefois, une limitation importante à l'usage de ce format de trace est de ne pas être supporté par beaucoup d'outils de conception et notamment les visionneuses de chronogrammes libres (de l'anglais *freeware*) tels que GTKWave.

Chapitre 5

La Présynthèse des Structures de Choix

Introduction

La qualité de la méthode employée pour synthétiser les structures de choix d'un modèle ASC (cf. paragraphe 3.1) est fondamentale pour obtenir un circuit asynchrone de ce modèle qui soit fonctionnellement correct et efficace. En effet, le comportement de la partie de contrôle d'un circuit asynchrone est essentiellement spécifié en ASC à l'aide de ses structures de choix (i.e. `as_choice_nd` et `as_choice_d`). Ces structures de choix sont, par exemple, utilisées pour modéliser les fonctions d'arbitrage des nœuds des réseaux sur puce asynchrones Octagon présentés aux paragraphe 3.2.1.

Même s'il existe des méthodes et des outils permettant de synthétiser des circuits asynchrones à partir de langages de description de matériel similaire à ASC, il n'en reste pas moins que la synthèse des structures de choix de ces langages est soit fortement limitée, soit peu efficace.

Avec le langage Balsa[1] [SF01, EAJ+06], il est, par exemple, impossible d'inclure des *probe* dans les structures de choix : `if then else end` et `case end`. Ce langage ne dispose que des instructions `select` et `arbitrate` pour sélectionner une action à effectuer en fonction des canaux de communication prêts à communiquer. Le pouvoir d'expression de ces instructions est très limité car leurs gardes ne peuvent contenir que des listes de canaux de communication à tester, et elles ne permettent ni d'utiliser de *probe* négatif ni de tester la valeur d'un canal actif.

Un avantage du langage CHP développé par le CALTECH est de posséder des structures de choix qui puissent contenir des *probes* (positifs ou négatifs) et qui soient synthétisables. Une méthode basée sur la syntaxe de ce langage permet notamment de synthétiser des circuits de type QDI [BM88b]. Le principal désavantage de cette méthode est d'être inefficace [BM88a], car elle ne permet pas d'extraire suffisamment de parallélisme. Pour pallier cette restriction, une technique basée sur les dépendances de données a aussi été développée [WM01, WM03], mais elle ne résout pas les problèmes de la synthèse des structures de choix CHP. Le but de cette méthode est d'augmenter le parallélisme d'un programme CHP en décomposant ses processus en plusieurs sous processus.

De même que la première méthode du CALTECH, la méthode ASP (*Asynchronous SystemC Presynthesis*) présentée dans ce chapitre définit un ensemble de règles de présynthèse à l'aide d'une grammaire attribuée [Paa95] (cf. paragraphe 5.4) qui permet de

[1] http://intranet.cs.man.ac.uk/apt/projects/tools/balsa

synthétiser les structures de choix de langages de modélisation tels que CHP ou ASC (cf. paragraphe 5.1). Contrairement à la méthode du CALTECH, celle-ci ne permet pas de synthétiser directement un circuit QDI, mais elle identifie les composants matériels (cf. paragraphe 5.2 et 5.3) nécessaires à la synthèse d'une structure de choix et elle optimise l'utilisation de ces composants (cf. paragraphe 5.5). Les composants matériels identifiés par cette méthode ne sont pas en général correctement supportés par les outils de synthèse qui ne sont pas dirigés par la syntaxe. L'objectif, à terme, de cette méthode est d'être couplée avec l'outil de synthèse TAST (cf. paragraphe 1.1.3.5) afin d'obtenir des circuits performants (surface, vitesse, consommation, ...) à partir de modèles comportant des structures de choix complexes. Toutefois, cette méthode possède encore de nombreuses limitations (formalisation, généralisation, ...) car elle est encore actuellement en cours de développement.

5.1 La Modélisation des Structures de Choix

Une structure de choix est spécifiée avec la méthode ASP comme un ensemble de commandes gardées en respectant la grammaire hors-contexte $G = (V_{nt}, V_t, R, S)$ qui est définie de la manière suivante :

$$V_{nt} = \{choice, guard\_cmd\_seq, guard\_cmd, exp\_or, exp\_and, exp\}$$
$$V_t = \{\Box, \sqcap, (,), :, ,, \wedge, \vee, \neg, \mathtt{cmd}_{i\in\mathbb{N}}, \overline{\mathtt{ch}_{i\in\mathbb{N}}}\}$$
$$R = \left\{ \begin{array}{rcl} choice & \rightarrow & \Box\ (\ guard\_cmd\_seq\ ) \mid \sqcap\ (\ guard\_cmd\_seq\ ) \\ guard\_cmd\_seq & \rightarrow & guard\_cmd\ ,\ guard\_cmd\_seq \mid guard\_cmd \\ guard\_cmd & \rightarrow & exp\_or : \mathtt{cmd}_{i\in\mathbb{N}} \\ exp\_or & \rightarrow & exp\_or \vee exp\_and \mid exp\_and \\ exp\_and & \rightarrow & exp\_and \wedge exp \mid exp \\ exp & \rightarrow & \neg\overline{\mathtt{ch}_{i\in\mathbb{N}}} \mid \overline{\mathtt{ch}_{i\in\mathbb{N}}} \mid (\ exp\_or\ ) \end{array} \right\}$$
$$S = choice$$

Les opérateurs $\Box$ et $\sqcap$ définissent respectivement des choix déterministes et non déterministes. Une commande gardée *guard_cmd* est composée d'une expression booléenne *exp_or* qui conditionne l'exécution de l'action $\mathtt{cmd}_{i\in\mathbb{N}}$ qui lui est associée. Une expression booléenne n'accepte comme élément terminal que des probes de canal : l'opération de probe d'un canal $\mathtt{ch}_{i\in\mathbb{N}}$ est noté $\overline{\mathtt{ch_i}}$. A l'heure actuelle, cette méthode n'a pas encore été généralisée aux structures de choix contenant des tests de valeurs sur des variables, des appels de fonctions, ... Si l'intégration de ces structures de choix à cette méthode est un travail fastidieux, il ne présente néanmoins pas de réelle difficulté car ces dernières sont déjà bien supportées par l'outil de synthèse TAST.

De même qu'en CHP, les opérateurs $\Box$ et $\sqcap$ sont bloquants : l'évaluation d'une structure de choix, spécifiée à l'aide de l'un de ces deux opérateurs, est bloquée jusqu'à ce qu'au moins une de ses gardes soit vérifiée. Cette restriction a notamment des conséquences importantes pour rendre synthétisable un modèle ASC à l'aide de cette méthode. En effet, les structures de choix ASC n'étant pas bloquantes, il est nécessaire d'utiliser la méthode `idle` à bon escient pour les rendre synthétisables. Pour remédier à cette limitation, une des améliorations envisagées de ASC est de rendre ses structures de choix bloquantes (cf. paragraphe 3.1.8).

Comme l'illustre la sous figure 5.1.A, cette méthode de présynthèse produit en sortie un ensemble d'équations logiques qui est assimilable à un composant matériel (cf. schéma logique de la sous figure 5.1.B). Ce composant matériel prend en entrée des signaux $\mathsf{in}_0, \ldots, \mathsf{in}_n$ qui définissent l'activité des canaux de communication $\mathtt{ch}_0, \ldots, \mathtt{ch}_n$ des gardes de la structure de choix initiale. Les signaux de sortie $\mathsf{out}_0, \ldots, \mathsf{out}_m$ de ce composant pilotent l'activation des actions $\mathtt{cmd}_0, \ldots, \mathtt{cmd}_m$. En outre, ce composant matériel prend en entrée un signal d'activation act. Il est à noter que le composant matériel produit par cette méthode ne gère pas le chemin d'acquittement des composants auquel il transmet des requêtes d'activation. Si cette tâche est essentielle pour être en mesure de synthétiser une structure de choix, elle ne présente cependant pas de réelle difficulté et c'est pourquoi j'ai préféré concentrer mes efforts sur la présynthèse du chemin de requête.

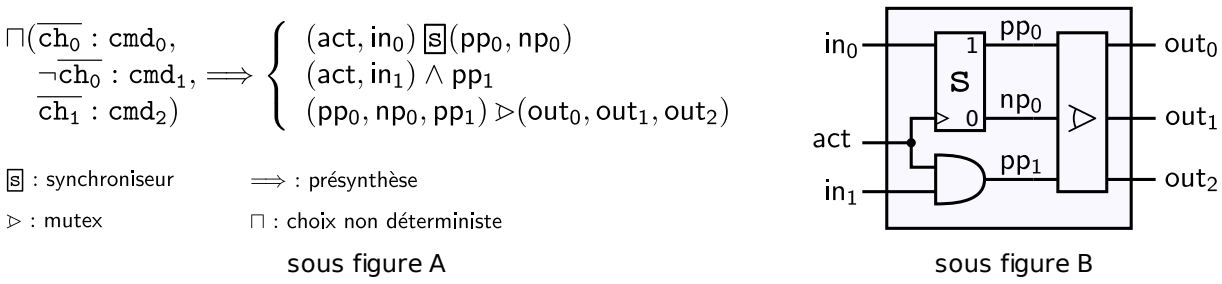


FIGURE 5.1 – *Exemple de présyntèse d'une structure de choix non déterministe*

Comme le montre la sous figure 5.1.A, le système d'équations logiques obtenu utilise un opérateur logique d'exclusion mutuelle ou *mutex* (cf. paragraphe 1.2.3.2) et d'échantillonnage ou synchroniseur (cf. paragraphe 1.2.3.1) pour respectivement résoudre les problèmes de non exclusivité des gardes (cf. paragraphe 5.2) et d'instabilité des *probes* (cf. paragraphe 5.3).

5.2 L'Exclusion Mutuelle des Gardes

Pour qu'un composant matériel modélisé par une équation logique $(\mathsf{act}, \mathsf{in}_0, \ldots, \mathsf{in}_n) \sqcap (\mathsf{out}_0, \ldots, \mathsf{out}_n)$ respecte le comportement d'un choix non déterministe, il est nécessaire qu'à tout instant au plus une des sorties $\mathsf{out}_{i \in [0,n]}$ soit vraie. Cette propriété est, par exemple, essentielle pour assurer le bon fonctionnement d'un arbitre asynchrone (cf. paragraphe 3.1.5).

Une solution employée pour résoudre ce problème est de renforcer les gardes d'un choix non déterministe : pour une garde donnée on ajoute les négations des gardes qui ne sont pas exclusives avec celle-ci [BM88b]. Une limitation importante de cette technique est d'introduire une priorité entre des gardes qui ne sont pas initialement corrélées.

Une autre solution consiste à employer des composants matériels d'exclusion mutuelle, appelés *mutex*, sur les signaux de sortie des gardes d'un choix non déterministe (cf. figure 5.1). Comme son nom l'indique, la principale caractéristique de ce composant est d'assurer l'exclusion mutuelle de ses signaux d'entrée. De manière plus formelle, un *mutex* à $n + 1$ entrées $(\mathsf{in}_0, \ldots, \mathsf{in}_n) \triangleright (\mathsf{out}_0, \ldots, \mathsf{out}_n)$ respecte à tout instant les propriété suivantes :

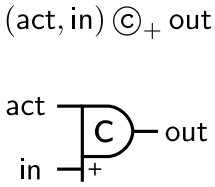
$$\begin{cases} (\mathsf{in}_0 \vee \ldots \vee \mathsf{in}_n) \Rightarrow (\mathsf{out}_0 \vee \ldots \vee \mathsf{out}_n) \\ \forall i, j \in [0, n]\, ((i \neq j \wedge \mathsf{out}_i) \Rightarrow (\mathsf{in}_i \wedge \neg \mathsf{out}_j)) \end{cases}$$

Une étude détaillée des *mutex* à n entrées et de leurs implémentations est présentée dans le chapitre 13 de [Kin08]. Une limitation importante de l'opérateur logique d'exclusion mutuelle est d'avoir une complexité non linéaire en fonction de son nombre d'entrées, et donc d'aboutir à des implémentations matérielles inefficaces (i.e. surface, consommation, latence, ...). Pour atténuer cette limitation, des techniques d'optimisation ont été développées pour diminuer le nombre d'entrées de cet opérateur (cf. paragraphe 5.5.1).

5.3 La Synchronisation des *Probes*

Comme l'indique le paragraphe 1.1.2.1, les circuits asynchrones utilisent des protocoles de communication locale à poignée de main pour synchroniser leurs fonctionnements. L'activation d'un bloc matériel spécifié par une équation logique $(\mathsf{act}, \mathsf{in}_0, \ldots) \,\square\, (\ldots)$ ou $(\mathsf{act}, \mathsf{in}_0, \ldots) \sqcap (\ldots)$ est modélisée par le signal act. Les valeurs des entrées $\mathsf{in}_0, \ldots$ de ces équations logiques doivent donc être synchronisées avec ce signal. Par exemple, l'évaluation de l'activité d'un canal $\mathtt{ch}_i$ (i.e. *probe* positif) est réalisée par un « et logique » entre le signal in_i et le signal act (cf. canal $\mathtt{ch}_1$ de la figure 5.1). Un *probe* positif ne requiert que l'emploi d'un simple « et logique » car les protocoles de communication employés par les circuits asynchrones assurent que le signal in_i ne peut effectuer la transition $1 \rightarrow 0$ que si le signal act est à 0. Cette propriété permet de garantir que si une garde contenant un *probe* positif déclenche un bloc matériel, alors elle reste vraie au moins jusqu'à ce que ce bloc matériel ait terminé son calcul.

Contrairement au *probe* positif, la valeur d'un *probe* négatif d'une structure de choix n'est pas stable vis à vis de son signal d'activation et nécessite donc l'emploi d'une porte logique mémorisante appelée « Müller dissymétrique » [Ren02] (cf. figure 5.2). Comme



in \ act	0	1
0	0	act^{-1}
1	0	1

FIGURE 5.2 – *Porte de Müller dissymétrique*

l'illustre la figure 5.3, un *probe* négatif ne peut pas être réalisé avec un simple « et logique » car le signal d'activation d'un canal peut effectuer la transition $0 \rightarrow 1$ à tout instant et donc rendre le résultat d'un *probe* négatif faux alors que le bloc matériel qu'il a déclenché n'a pas encore terminé son calcul.

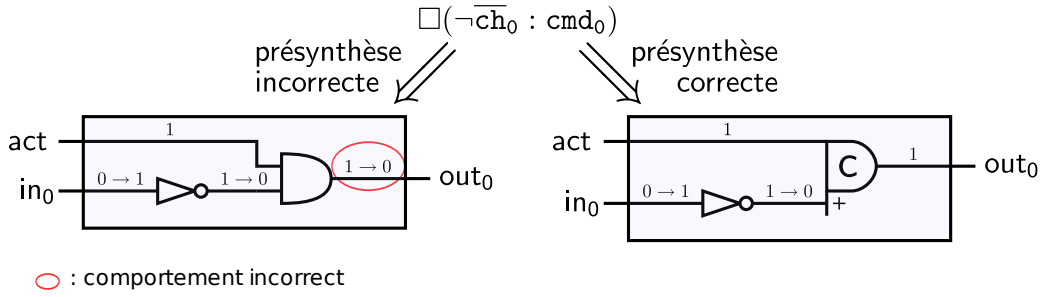


FIGURE 5.3 – *Exemple de présynthèse de probe négatif*

Lorsque des opérations de *probes* positifs et négatifs sont réalisées au sein d'une même structure de choix, alors les opérateurs logiques vus précédemment doivent être remplacés par un synchroniseur asynchrone (cf. figure 5.4). Si ce remplacement n'est pas effectué,

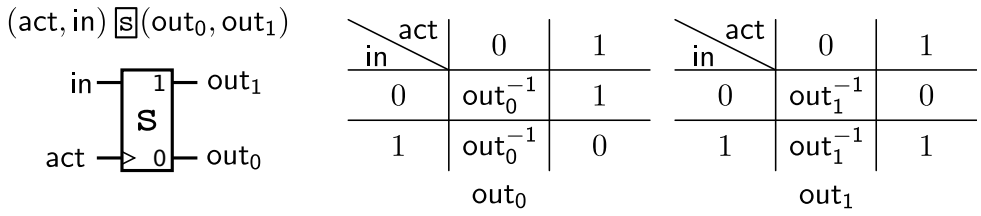


FIGURE 5.4 – *Synchroniseur Asynchrone*

alors le comportement du composant matériel obtenu peut être incorrect : un choix déterministe peut par exemple avoir plusieurs de ses sorties vraies au même instant (cf. figure 5.5). De manière plus générale, cette transformation est nécessaire pour assurer que les sorties d'une structure de choix sont stables jusqu'à la complétion des actions qu'elles déclenchent (i.e. que son signal d'activation effectue la transition $1 \rightarrow 0$).

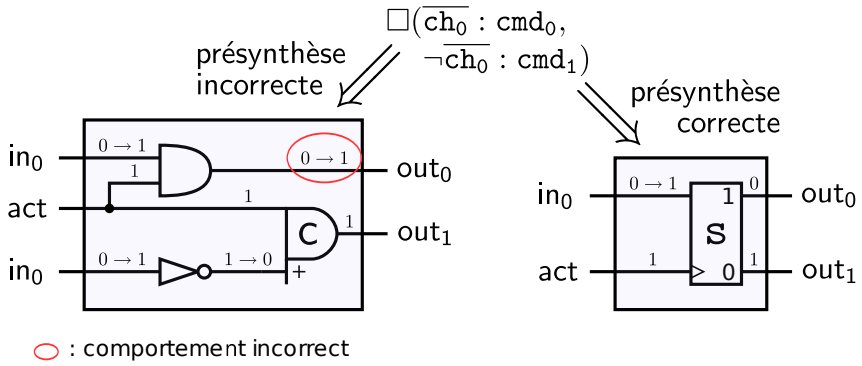


FIGURE 5.5 – *Exemple de présynthèse de probe négatif et positif*

5.4 Grammaires Attribuées de Présynthèse

Les paragraphes précédents ont présenté de manière informelle les différents opérateurs logiques nécessaires à la présynthèse d'une structure de choix, ce paragraphe définit, pour sa part, comment les utiliser pour réaliser cette présynthèse. La méthode ASP, dirigée par la syntaxe, est définie formellement à l'aide de la grammaire attribuée $GA = (G, A, R)$ où G est la grammaire hors-contexte vue au paragraphe 5.1, A est un ensemble d'attributs qui est présenté aux paragraphes 5.4.1 et 5.4.2, et R est un ensemble de règles d'affectation des attributs (cf. paragraphe 5.4.3).

5.4.1 Les Domaines des Attributs

Les domaines de base utilisés par les attributs sont ceux de l'ensemble des signaux logiques $\mathbb{S} = \{\mathsf{act}, \mathsf{in}_0, \ldots\}$ et de l'ensemble des opérateurs logiques $\mathbb{O} = \{\wedge, \boxed{\mathsf{s}}, \ldots\}$. Un signal logique peut prendre la valeur 0 ou 1 et permet d'interconnecter des opérateurs logiques. Le rôle d'un opérateur logique $op : \mathbb{S}^n \rightarrow \mathbb{S}^m$ est d'affecter des valeurs à ses

signaux de sortie en fonction des valeurs de ses signaux d'entrée. Les opérateurs logiques suivants sont notamment utilisés par la grammaire attribuée GA :

- les opérateurs de « et logique » $\wedge : \mathbb{S}^{n\geq 2} \rightarrow \mathbb{S}$, de « ou logique » $\vee : \mathbb{S}^{n\geq 2} \rightarrow \mathbb{S}$, de « non logique » $\neg : \mathbb{S} \rightarrow \mathbb{S}$,
- les opérateurs d'exclusion mutuelle $\triangleright : \mathbb{S}^{n\geq 2} \rightarrow \mathbb{S}^n$ (cf. paragraphe 5.2), d'échantillonnage $\boxed{\mathsf{s}} : \mathbb{S}^2 \rightarrow \mathbb{S}^2$ (cf. figure 5.4), de Müller dissymétrique $\copyright_+ : \mathbb{S}^2 \rightarrow \mathbb{S}$ (cf. figure 5.2),
- l'opérateur « fil » $\bullet\!\!-\!\!\bullet : \mathbb{S} \rightarrow \mathbb{S}$ qui permet de relier directement deux signaux.

Pour faciliter l'identification des signaux appartenant à $\mathbb{S}$ qui sont utilisés par la grammaire attribuée GA, un indice $i \in \mathbb{N}$ et un exposant constitué d'une liste de signaux peuvent être ajoutés à leur nom. Par exemple act, out_1 et $\mathsf{sig}^{\mathsf{act},\mathsf{out}_1}$ sont des identificateurs de signaux valides utilisant cette convention de notation. En plus de cette convention d'écriture, les types des signaux employés par la grammaire GA sont identifiés par les noms suivants :

- act : signal d'activation d'une structure de choix,
- out_i : signal d'activation d'une action gardée $\mathtt{act}_i$,
- in_i : signal indiquant l'état d'un canal de communication $\mathtt{ch}_i$,
- pp_i, np_i : signal modélisant respectivement le résultat d'un *probe* positif et négatif d'un canal $\mathtt{ch}_i$
- $\mathsf{and}^{s,s'}$, $\mathsf{or}^{s,s'}$, not_i : signaux connectés respectivement à la sortie d'un opérateur logique $\wedge$, $\vee$, $\neg$ et utilisés pour construire la représentation d'une garde par un ensemble d'équations logiques.

Une équation logique définit les listes de signaux qui sont connectés aux entrées et aux sorties d'un opérateur logique. Une équation logique est décrite en utilisant une notation infixée où les listes de signaux d'entrée et de sortie sont respectivement disposées à gauche et à droite d'un opérateur. Pour modéliser un composant matériel d'un circuit asynchrone, la grammaire attribuée GA emploie les domaines de l'ensemble des équations logiques $\mathbb{E}$ et l'ensemble des ensembles d'équations logiques $\mathcal{P}(\mathbb{E})$. Elle utilise, en outre, le domaine de l'ensemble des listes de signaux $\mathbb{L}$ pour être en mesure de synthétiser des équations logiques dont les signaux sont définis sur différents nœuds de l'arbre syntaxique.

5.4.2 Les Attributs

L'ensemble des attributs A de la grammaire attribuée GA est constitué des attributs suivants.

- $i \in \mathbb{N}$: cet entier qui est synthétisé par les symbole terminaux $\mathtt{act}_{i\in\mathbb{N}}$ et $\mathtt{ch}_{i\in\mathbb{N}}$ sert à identifier des signaux logiques. Il est, par exemple, employé pour identifier le signal d'activation de l'action gardée associée au symbole terminal $\mathtt{act}_{i\in\mathbb{N}}$.
- $s, s', s'' \in \mathbb{S}$: ces signaux sont synthétisés par les symboles non terminaux *exp_or*, *exp_and* et *exp*. Ils sont notamment employés pour relier les différentes équations logiques représentant la garde d'une structure de choix.
- $(s, s', \ldots), (s, \ldots), (s) \in \mathbb{L}$: ces listes de signaux sont synthétisés par le symbole non terminal *guard_cmd_seq*. Chaque signal de ces listes représente le résultat de l'évaluation d'une garde.
- $o \in \mathbb{S}$: ce signal qui est synthétisé par le symbole non terminal *guard_cmd* sert à activer une action gardée.
- $(o, o', \ldots), (o, \ldots), (o) \in \mathbb{L}$: ces listes de signaux qui sont synthétisés par le sym-

bole non terminal *guard_cmd_seq* permettent d'activer les actions gardées par une structure de choix.
- $E, E', E'', E''' \in \mathcal{P}(\mathbb{E})$: ces attributs sont synthétisés ou hérités par les différents symboles non terminaux pour construire la spécification d'une structure de choix sous la forme d'un ensemble d'équations logiques. L'équation logique finale est notamment produite par l'attribut E associé à l'axiome *choice*.

5.4.3 Les Règles d'Affectation des Attributs

Les règles d'affectation des attributs présentées dans ce paragraphe définissent comment synthétiser un ensemble d'équations logiques $E \in \mathbb{E}$ à partir d'une structure de choix respectant la grammaire G. Les attributs synthétisés et hérités par les symboles de cette grammaire sont respectivement préfixés par $\uparrow$ et $\downarrow$.

$$\begin{aligned} &choice\uparrow E \rightarrow \square\ (\ guard\_cmd\_seq\downarrow\emptyset\uparrow E'\uparrow(s, s', \ldots)\uparrow(o, o', \ldots)\ ) \\ &\quad E := E' \cup \{s \bullet\!\!-\!\!\bullet o, s' \bullet\!\!-\!\!\bullet o', \ldots\} \end{aligned} \tag{5.1}$$

$$\begin{aligned} &choice\uparrow E \rightarrow \sqcap\ (\ guard\_cmd\_seq\downarrow\emptyset\uparrow E'\uparrow(s, s', \ldots)\uparrow(o, o', \ldots)\ ) \\ &\quad E := E' \cup \{(s, s', \ldots) \triangleright (o, o', \ldots)\} \end{aligned} \tag{5.2}$$

Les règles 5.1 et 5.2 produisent l'ensemble d'équations logiques E qui implémente une structure de choix respectant la grammaire hors contexte G. Cet ensemble d'équations logiques est défini à l'aide de l'ensemble d'équations logiques E' qui est généré par le symbole non terminal *guard_cmd_seq* et qui contient les équations logiques des différentes gardes de la structure de choix à présynthétiser. Ce symbole non terminal synthétise aussi les listes de signaux suivantes :
- $(s, s', \ldots)$: chaque signal de cette liste indique si la garde qui lui est associée est vraie ou non,
- $(o, o', \ldots)$: les signaux de cette liste activent les commandes protégées par les gardes de la structure de choix.

La règle 5.1 emploie de simples fils pour relier les signaux de ces deux listes car les gardes des structures de choix déterministe sont mutuellement exclusives. La règle 5.2 utilise, pour sa part, un *mutex* pour relier ces signaux car les structures de choix non déterministe ne sont pas mutuellement exclusives.

$$\begin{aligned} &guard\_cmd\_seq\downarrow E\uparrow E''\uparrow(s, s', \ldots)\uparrow(o, o', \ldots) \rightarrow \\ &\quad guard\_cmd\downarrow E\uparrow E'\uparrow s\uparrow o\ \mathbf{,}\ guard\_cmd\_seq\downarrow E'\uparrow E''\uparrow(s', \ldots)\uparrow(o', \ldots) \end{aligned} \tag{5.3}$$

$$guard\_cmd\_seq\downarrow E\uparrow E'\uparrow(s)\uparrow(o) \rightarrow guard\_cmd\downarrow E\uparrow E'\uparrow s\uparrow o \tag{5.4}$$

$$guard\_cmd\downarrow E\uparrow E'\uparrow s\uparrow \texttt{out}_i \rightarrow exp\_or\downarrow E\uparrow E'\uparrow s\ \texttt{:}\ \texttt{cmd}_{i\in\mathbb{N}}\uparrow i \tag{5.5}$$

L'objectif des règles 5.3 et 5.4 est de construire récursivement l'ensemble d'équations logiques des différentes gardes et les deux listes de signaux vues précédemment. Cette définition récursive se base sur la règle 5.5 qui produit pour une commande gardée les attributs suivants :
- le signal d'activation de sa commande $\texttt{out}_i$,
- le signal de validité de sa garde s,

– l'ensemble d'équations logiques E' qui est obtenu en ajoutant les équations logiques de sa garde à l'ensemble d'équations logiques E des gardes précédemment présynthétisées.

$$exp\_or{\downarrow}E{\uparrow}E'''{\uparrow}s \rightarrow exp\_or{\downarrow}E{\uparrow}E'{\uparrow}s' \vee exp\_and{\downarrow}E'{\uparrow}E''{\uparrow}s'' \tag{5.6}$$
$$\text{si } (s', s'') \vee \mathsf{or}^{s',s''} \in E'' \text{ alors } E''' := E'' \text{ et } s := \mathsf{or}^{s',s''}$$
$$\text{sinon si } (s'', s') \vee \mathsf{or}^{s'',s'} \in E'' \text{ alors } E''' := E'' \text{ et } s := \mathsf{or}^{s'',s'}$$
$$\text{sinon } E''' := E'' \cup \left\{(s', s'') \vee \mathsf{or}^{s',s''}\right\} \text{ et } s := \mathsf{or}^{s',s''}$$

$$exp\_or{\downarrow}E{\uparrow}E'{\uparrow}s \rightarrow exp\_and{\downarrow}E{\uparrow}E'{\uparrow}s \tag{5.7}$$

La présynthèse des gardes contenant des disjonctions d'expression booléenne est définie par les régles 5.6 et 5.7. La règle 5.6 est accompagnée de conditions sur les affectations des attributs qu'elle synthétise pour éviter de dupliquer inutilement du matériel. Ces affectations conditionnelles permettent, par exemple, d'obtenir un ensemble d'équations logiques ne contenant que l'équation logique $(\mathsf{in}_0, \mathsf{in}_1) \vee \mathsf{or}^{\mathsf{in}_0,\mathsf{in}_1}$ ou $(\mathsf{in}_1, \mathsf{in}_0) \vee \mathsf{or}^{\mathsf{in}_1,\mathsf{in}_0}$ à partir d'une structure de choix contenant plusieurs instances des expressions $\overline{\mathtt{in}_0} \vee \overline{\mathtt{in}_1}$ et $\overline{\mathtt{in}_1} \vee \overline{\mathtt{in}_0}$.

$$exp\_and{\downarrow}E{\uparrow}E'''{\uparrow}s \rightarrow exp\_and{\downarrow}E{\uparrow}E'{\uparrow}s' \wedge exp{\downarrow}E'{\uparrow}E''{\uparrow}s'' \tag{5.8}$$
$$\text{si } (s', s'') \wedge \mathsf{and}^{s',s''} \in E'' \text{ alors } E''' := E'' \text{ et } s := \mathsf{and}^{s',s''}$$
$$\text{sinon si } (s'', s') \wedge \mathsf{and}^{s'',s'} \in E'' \text{ alors } E''' := E'' \text{ et } s := \mathsf{and}^{s'',s'}$$
$$\text{sinon } E''' := E'' \cup \left\{(s', s'') \wedge \mathsf{and}^{s',s''}\right\} \text{ et } s := \mathsf{and}^{s',s''}$$

$$exp\_and{\downarrow}E{\uparrow}E'{\uparrow}s \rightarrow exp{\downarrow}E{\uparrow}E'{\uparrow}s \tag{5.9}$$

Les règles 5.8 et 5.9 ont une fonction identique à celle des règles 5.6 et 5.7 mais pour des gardes contenant des conjonctions d'expressions booléennes.

$$exp{\downarrow}E{\uparrow}E'{\uparrow}\mathsf{pp}_i \rightarrow \overline{\mathtt{ch}_{i\in\mathbb{N}}}{\uparrow}i \tag{5.10}$$
$$\text{si } \mathsf{in}_i \neg \mathsf{not}_i \in E \text{ alors}$$
$$E' := E \setminus \left\{\mathsf{in}_i \neg \mathsf{not}_i, (\mathsf{act}, \mathsf{not}_i)\, ©_+\, \mathsf{np}_i\right\} \cup \{(\mathsf{act}, \mathsf{in}_i)\, \boxed{\mathsf{s}}(\mathsf{pp}_i, \mathsf{np}_i)\}$$
$$\text{sinon si } (\mathsf{act}, \mathsf{in}_i) \wedge \mathsf{pp}_i \in E \text{ alors } E' := E$$
$$\text{sinon si } (\mathsf{act}, \mathsf{in}_i)\, \boxed{\mathsf{s}}(\mathsf{pp}_i, \mathsf{np}_i) \in E \text{ alors } E' := E$$
$$\text{sinon } E' := E \cup \{(\mathsf{act}, \mathsf{in}_i) \wedge \mathsf{pp}_i\}$$

La règle 5.10 est dédiée à la présynthèse des *probes* positifs des gardes. De même que pour la règle 5.6, les affectations des attributs synthétisés par cette règle sont conditionnées. Ces affectations conditionnelles servent aussi à définir le type d'opérateur logique à employer. Par exemple, lorsque l'ensemble d'équations logiques E contient un non logique du signal in_i cela implique qu'au moins une garde de la structure de choix contient un *probe* négatif du canal $\mathtt{ch}_i$ et qu'il est donc nécessaire d'utiliser un synchroniseur pour réaliser ce *probe*

négatif et le *probe* positif de cette règle.

$$exp{\downarrow}E{\uparrow}E'{\uparrow}\mathsf{np}_i \rightarrow \neg\overline{\mathsf{ch}_{i\in\mathbb{N}}}{\uparrow}i \tag{5.11}$$
si $(\mathsf{act},\mathsf{in}_i) \wedge \mathsf{pp}_i \in E$ alors
$E' := E \setminus \{(\mathsf{act},\mathsf{in}_i) \wedge \mathsf{pp}_i\} \cup \{(\mathsf{act},\mathsf{in}_i)\ \boxed{\mathsf{s}}(\mathsf{pp}_i,\mathsf{np}_i)\}$
sinon si $\mathsf{in}_i\ \neg\,\mathsf{not}_i \in E$ alors $E' := E$
sinon si $(\mathsf{act},\mathsf{in}_i)\ \boxed{\mathsf{s}}(\mathsf{pp}_i,\mathsf{np}_i) \in E$ alors $E' := E$
sinon $E' := E \cup \left\{\mathsf{in}_i\ \neg\,\mathsf{not}_i, (\mathsf{act},\mathsf{not}_i)\ \textcircled{c}_+\ \mathsf{np}_i\right\}$

La règle 5.11 a un rôle identique à celui de la règle 5.10 mais pour le *probes* positifs.

$$exp{\downarrow}E{\uparrow}E'{\uparrow}s \rightarrow (\ exp\_or{\downarrow}E{\uparrow}E'{\uparrow}s\ ) \tag{5.12}$$

La règle 5.12 sert uniquement à modifier l'ordre d'évaluation des expressions logiques d'une garde.

La figure 5.6 présente un exemple d'arbre de dérivation obtenu, avec la grammaire attribuée GA, à partir de la structure de choix non déterministe présentée par la figure 5.1 de la page 99. Il est à noter que l'ensemble d'équations logiques synthétisé par la racine de cet arbre de dérivation est identique à celui présenté par la figure 5.1.

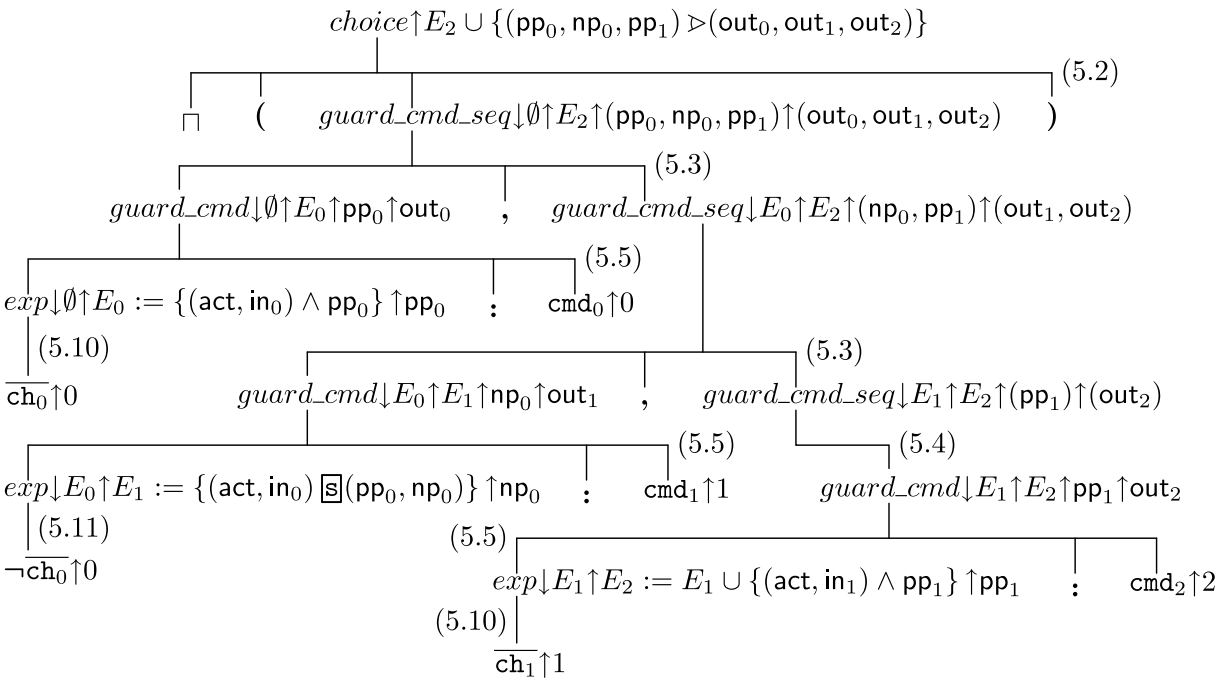


FIGURE 5.6 – *Exemple d'arbre de dérivation d'une structure de choix non déterministe*

5.5 Les Optimisations

Comparés aux opérateurs logiques standards (i.e. $\wedge$, $\vee$, ...), les opérateurs logiques de synchronisation $\triangleright$ et $\boxed{\mathsf{s}}$ ont des coûts importants : surface, consommation, ... Pour diminuer le coût de la synchronisation des structures de choix, des optimisations spécifiquement dédiées aux opérateurs $\triangleright$ (cf. paragraphe 5.5.1) et $\boxed{\mathsf{s}}$ (cf. paragraphe 5.5.2) ont été élaborées.

5.5.1 Simplification de l'Opérateur d'Exclusion Mutuelle

Une optimisation permettant d'améliorer significativement le coût d'une structure de choix $\sqcap$ est de diminuer le nombre d'entrées de l'opérateur $\triangleright$ appartenant à un ensemble d'équations logiques obtenu avec la grammaire attribuée GA. Une première étape pour réaliser cette optimisation est de déterminer les gardes qui sont mutuellement exclusives (cf. paragraphe 5.5.1.1). Une deuxième étape est de diminuer le nombre d'entrées de l'opérateur $\triangleright$ en fonction des gardes qui sont mutuellement exclusives (cf. paragraphe 5.5.1.2).

5.5.1.1 Identification des Gardes Exclusives

Une première solution pour réaliser cette tâche est de considérer les *probes* des canaux comme des variables propositionnelles et donc de modéliser une garde comme une formule propositionnelle. La propriété 13 permet alors d'identifier facilement des gardes qui sont mutuellement exclusives [ASBG02].

Propriété 13. *Si les formules propositionnelles de deux gardes distinctes sont insatisfaisables, alors ces deux gardes sont mutuellement exclusives.*

Cette méthode permet, par exemple, d'identifier que les gardes $\overline{\mathsf{ch}_0}$ et $\neg\overline{\mathsf{ch}_0}$ de la structure de choix présentée par la figure 5.1 de la page 99 sont mutuellement exclusives.

Une limitation de cette méthode est de ne pas permettre d'identifier toutes les gardes mutuellement exclusives d'une structure de choix, car la propriété 13 définit une condition suffisante mais pas nécessaire. Pour atténuer cette limitation, une solution complémentaire est d'autoriser un concepteur à ajouter des annotations aux structures de choix non déterministes pour désigner les gardes qui sont mutuellement exclusives.

5.5.1.2 Optimisation des Gardes Exclusives

Dans la suite de ce paragraphe, les optimisations présentées sont toutes définies en fonction d'un ensemble d'équations logiques E obtenu à partir d'une structure de choix $\sqcap(g_0 : \mathsf{out}_0, \ldots, g_n : \mathsf{out}_n)$. En outre, ces optimisations supposent que E est composé de l'équation logique $e = (s_0, \ldots, s_n) \triangleright(\mathsf{out}_0, \ldots, \mathsf{out}_n)$ où les signaux $s_0, \ldots s_n$ correspondent respectivement à l'évaluation des gardes $g_0, \ldots, g_n$.

Une première optimisation de base, qui peut être appliquée sur E, est de retirer de l'équation logique e les signaux des gardes qui sont mutuellement exclusives. Si on définit $\mathbb{G}$ comme étant l'ensemble des gardes des structures de choix, et $\ominus : \mathbb{G} \rightarrow \mathbb{B}$ comme l'opérateur d'exclusion mutuelle entre deux gardes, alors pour une garde $g_i \in G = \{g_0, \ldots, g_n\}$ donnée, cette optimisation se traduit plus formellement de la manière suivante :

$$\begin{array}{l}\textbf{si } \forall g_j \in G \setminus \{g_i\}\ (g_i \ominus g_j) \textbf{ alors} \\ \quad E := E \setminus \{e\} \cup \{s_i \bullet\!\!\rightarrow \mathsf{out}_i, (\ldots, s_{i-1}, s_{i+1}, \ldots) \triangleright(\ldots, \mathsf{out}_{i-1}, \mathsf{out}_{i+1}, \ldots)\}\end{array} \tag{5.13}$$

Une deuxième optimisation de base, qui peut être appliquée sur E, est de rassembler les signaux de e dont les gardes sont mutuellement exclusives entre elles. Pour un sous

ensemble $\{g_x, \ldots\} \subset G$ donné, cette optimisation s'applique de la manière suivante :

$$\textbf{si } \forall g_i, g_j \in \{g_x, \ldots\}\,(i \neq j \wedge g_i \ominus g_j) \textbf{ alors} \tag{5.14}$$
$$E := E \setminus \{e\} \cup \left\{ \begin{array}{l} (s_x, \ldots) \vee \mathsf{or}^{s_x,\cdots} \\ (\ldots, s_{x-1}, s_{x+1}, \ldots, \mathsf{or}^{s_x,\cdots}) \triangleright (\ldots, \mathsf{out}_{x-1}, \mathsf{out}_{x+1}, \ldots, \mathsf{ex}^{s_x,\cdots}) \\ (\mathsf{ex}^{s_x,\cdots}, s_x) \,©\, \mathsf{out}_x \\ \ldots \end{array} \right\}$$

Il est à noter que l'opérateur logique $© : \mathbb{S}^2 \to \mathbb{S}$ employé par cet algorithme correspond à la porte de Müller standard vue au paragraphe 1.1.1.1. Cette optimisation doit toutefois être utilisée avec précaution et cela plus particulièrement lorsque les intersections des sous ensembles de gardes exclusives ne sont pas vides. La figure 5.7 illustre, par exemple, comment cet algorithme peut être convenablement utilisé avec l'ensemble d'équations logiques présenté par la figure 5.1 de la page 99.

$$\left\{ \begin{array}{l} (\mathsf{act}, \mathsf{in}_0)\,\boxed{\mathsf{s}}\,(\mathsf{pp}_0, \mathsf{np}_0) \\ (\mathsf{act}, \mathsf{in}_1) \wedge \mathsf{pp}_1 \\ (\mathsf{pp}_0, \mathsf{np}_0, \mathsf{pp}_1) \triangleright (\mathsf{out}_0, \mathsf{out}_1, \mathsf{out}_2) \end{array} \right. \Longrightarrow \left\{ \begin{array}{l} (\mathsf{act}, \mathsf{in}_0)\,\boxed{\mathsf{s}}\,(\mathsf{pp}_0, \mathsf{np}_0) \\ (\mathsf{act}, \mathsf{in}_1) \wedge \mathsf{pp}_1 \\ (\mathsf{pp}_0, \mathsf{np}_0) \vee \mathsf{or}^{\mathsf{pp}_0,\mathsf{np}_0} \\ (\mathsf{or}^{\mathsf{pp}_0,\mathsf{np}_0}, \mathsf{pp}_1) \triangleright (\mathsf{ex}^{\mathsf{pp}_0,\mathsf{np}_0}, \mathsf{out}_2) \\ (\mathsf{ex}^{\mathsf{pp}_0,\mathsf{np}_0}, \mathsf{pp}_0) \,©\, \mathsf{out}_0 \\ (\mathsf{ex}^{\mathsf{pp}_0,\mathsf{np}_0}, \mathsf{np}_0) \,©\, \mathsf{out}_1 \end{array} \right.$$

$\Longrightarrow$: optimisation

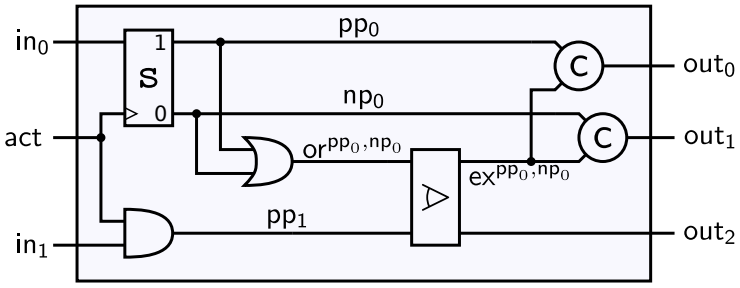


FIGURE 5.7 – *Exemple d'optimisation de gardes exclusives*

5.5.2 Fusion des Équations Logiques d'Échantillonnage

Une autre optimisation permettant de fortement diminuer le coût d'une structure de choix est de diminuer le nombre d'équations logiques employant des opérateurs de synchronisation $\boxed{\mathsf{s}}$. Pour parvenir à cette fin, une optimisation de base consiste à fusionner les équations logiques dont les sorties sont utilisées pour réaliser des conjonctions de *probes* positifs ou négatifs. Plus précisément, la fusion de la conjonction de *probes* positifs

s'effectue à l'aide de l'algorithme suivant :

$$
\begin{array}{l}
\textbf{si } (\mathsf{pp}_i, \ldots) \wedge \mathsf{and}^{\mathsf{pp}_i,\cdots}, (\mathsf{act}, \mathsf{in}_i)\ \boxed{\mathsf{s}}(\mathsf{pp}_i, \mathsf{np}_i), \ldots \in E \\
\quad \forall e \in E \setminus \{(\mathsf{pp}_i, \ldots) \wedge \mathsf{and}^{\mathsf{pp}_i,\cdots}, (\mathsf{act}, \mathsf{in}_i)\ \boxed{\mathsf{s}}(\mathsf{pp}_i, \mathsf{np}_i)\}\ (\mathsf{pp}_i \notin e) \\
\quad \ldots \textbf{ alors} \\
\quad E := E \setminus \{(\mathsf{pp}_i, \ldots) \wedge \mathsf{and}^{\mathsf{pp}_i,\cdots}, (\mathsf{act}, \mathsf{in}_i)\ \boxed{\mathsf{s}}(\mathsf{pp}_i, \mathsf{np}_i), \ldots\} \\
\qquad\qquad \cup \left\{ \begin{array}{l}
\mathsf{in}_i \neg \mathsf{not}_i \\
\ldots \\
(\mathsf{in}_i, \ldots) \wedge \mathsf{and}^{\mathsf{in}_i,\cdots} \\
(\mathsf{act}, \mathsf{and}^{\mathsf{in}_i,\cdots})\ \boxed{\mathsf{s}}(\mathsf{and}^{\mathsf{pp}_i,\cdots}, \mathsf{sync}^{\mathsf{np}_i,\cdots}) \\
(\mathsf{sync}^{\mathsf{np}_i,\cdots}, \mathsf{not}_i)\ \textcircled{c}_+\ \mathsf{np}_i \\
\ldots
\end{array} \right\}
\end{array}
\tag{5.15}
$$

Comme le montre la figure 5.8, cet algorithme peut être notamment employé avec l'ensemble d'équations logiques produit par la grammaire attribuée GA à partir de la structure de choix $\sqcap(\neg\overline{\mathtt{ch}_0} : \mathtt{cmd}_0, \neg\overline{\mathtt{ch}_1} : \mathtt{cmd}_1, \overline{\mathtt{ch}_0} \wedge \overline{\mathtt{ch}_1} : \mathtt{cmd}_2)$.

$$
\left\{ \begin{array}{l}
(\mathsf{act}, \mathsf{in}_0)\ \boxed{\mathsf{s}}(\mathsf{pp}_0, \mathsf{np}_0) \\
(\mathsf{act}, \mathsf{in}_1)\ \boxed{\mathsf{s}}(\mathsf{pp}_1, \mathsf{np}_1) \\
(\mathsf{pp}_0, \mathsf{pp}_1) \wedge \mathsf{and}^{\mathsf{pp}_0,\mathsf{pp}_1} \\
(\mathsf{np}_0, \mathsf{np}_1, \mathsf{and}^{\mathsf{pp}_0,\mathsf{pp}_1}) \rhd (\mathsf{out}_0, \mathsf{out}_1, \mathsf{out}_2)
\end{array} \right.
\quad \Longrightarrow \quad
\left\{ \begin{array}{l}
\mathsf{in}_0 \neg \mathsf{not}_0 \\
\mathsf{in}_1 \neg \mathsf{not}_1 \\
(\mathsf{in}_0, \mathsf{in}_1) \wedge \mathsf{and}^{\mathsf{in}_0,\mathsf{in}_1} \\
(\mathsf{act}, \mathsf{and}_{\mathsf{in}_0,\mathsf{in}_1})\ \boxed{\mathsf{s}}(\mathsf{and}^{\mathsf{pp}_0,\mathsf{pp}_1}, \mathsf{sync}^{\mathsf{np}_0,\mathsf{np}_1}) \\
(\mathsf{sync}^{\mathsf{np}_0,\mathsf{np}_1}, \mathsf{not}_0)\ \textcircled{c}_+\ \mathsf{np}_0 \\
(\mathsf{sync}^{\mathsf{np}_0,\mathsf{np}_1}, \mathsf{not}_1)\ \textcircled{c}_+\ \mathsf{np}_1 \\
(\mathsf{np}_0, \mathsf{np}_1, \mathsf{and}^{\mathsf{pp}_0,\mathsf{pp}_1}) \rhd (\mathsf{out}_0, \mathsf{out}_1, \mathsf{out}_2)
\end{array} \right.
$$

$\Longrightarrow$: optimisation

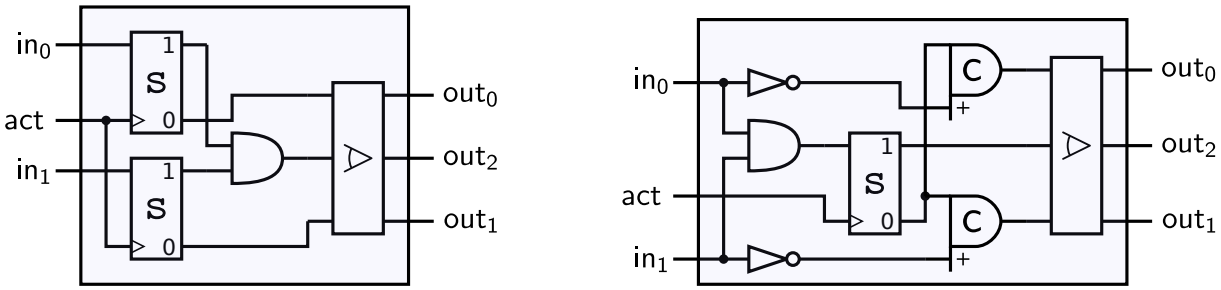


FIGURE 5.8 – *Exemple de fusion d'opérateurs d'échantillonnage*

De manière symétrique, la conjonction de *probes* négatifs se fusionnent avec l'algorithme suivant :

$$
\begin{array}{l}
\textbf{si } (\mathsf{np}_i, \ldots) \wedge \mathsf{and}^{\mathsf{np}_i,\cdots}, (\mathsf{act}, \mathsf{in}_i)\ \boxed{\mathsf{s}}(\mathsf{pp}_i, \mathsf{np}_i), \ldots \in E \\
\quad \forall e \in E \setminus \{(\mathsf{np}_i, \ldots) \wedge \mathsf{and}^{\mathsf{np}_i,\cdots}, (\mathsf{act}, \mathsf{in}_i)\ \boxed{\mathsf{s}}(\mathsf{pp}_i, \mathsf{np}_i)\}\ (\mathsf{np}_i \notin e) \\
\quad \ldots \textbf{ alors} \\
\quad E := E \setminus \{(\mathsf{np}_i, \ldots) \wedge \mathsf{and}^{\mathsf{np}_i,\cdots}, (\mathsf{act}, \mathsf{in}_i)\ \boxed{\mathsf{s}}(\mathsf{pp}_i, \mathsf{np}_i), \ldots\} \\
\qquad\qquad \cup \left\{ \begin{array}{l}
\mathsf{in}_i \neg \mathsf{not}_i \\
\ldots \\
(\mathsf{not}_i, \ldots) \wedge \mathsf{and}^{\mathsf{not}_i,\cdots} \\
(\mathsf{act}, \mathsf{and}^{\mathsf{not}_i,\cdots})\ \boxed{\mathsf{s}}(\mathsf{and}^{\mathsf{np}_i,\cdots}, \mathsf{sync}^{\mathsf{pp}_i,\cdots}) \\
(\mathsf{sync}^{\mathsf{pp}_i,\cdots}, \mathsf{in}_i)\ \textcircled{c}_+\ \mathsf{pp}_i \\
\ldots
\end{array} \right\}
\end{array}
\tag{5.16}
$$

Ces deux algorithmes ne doivent pas être employés conjointement pour fusionner des conjonctions de *probes* qui ne sont pas disjointes. Par exemple, une seule des deux conjonctions de *probes* de la structure de choix $\sqcap(\overline{\texttt{ch}_0} \wedge \overline{\texttt{ch}_1} : \texttt{cmd}_0, \neg\overline{\texttt{ch}_0} \wedge \neg\overline{\texttt{ch}_1} : \texttt{cmd}_1)$ peut être fusionnée : soit la conjonction $\overline{\texttt{ch}_0} \wedge \overline{\texttt{ch}_1}$ avec l'algorithme 5.15, soit la conjonction $\neg\overline{\texttt{ch}_0} \wedge \neg\overline{\texttt{ch}_1}$avec l'algorithme 5.16.

Une possibilité pour améliorer significativement l'efficacité de ces deux algorithmes est de normaliser les gardes d'une structure de choix sous forme normale disjonctive avant de les présynthétiser. Cependant, cette transformation doit être appliquée avec précaution car elle peut faire croître exponentiellement la taille des gardes.

5.6 Les Limitations et les Améliorations

Si la méthode de présynthèse des structures de choix présentée dans ce chapitre est dès à présent fonctionnelle, il n'en reste pas moins que celle-ci doit être encore améliorée avant de pouvoir être intégrée au flot de synthèse TAST. Dans un premier temps, un algorithme général d'optimisation, basé sur les algorithmes de base présentés au paragraphe 5.5, doit être développé pour automatiser l'optimisation des ensembles d'équations logiques obtenus avec la grammaire attribuée *GA*. En outre, cette grammaire attribuée doit être étendue pour, d'une part prendre en charge les acquittements des requêtes qu'elle transmet, et d'autre part pour supporter des structures de choix moins restreintes (e.g. comportant des opérateurs arithmétique de comparaisons, des appels de fonction, ...).

Dans un deuxième temps, il serait pertinent de prouver formellement que cette méthode de présynthèse (i.e. grammaire attribuée et optimisation) préserve la sémantique de la structure de choix initiale et que les ensembles d'équations logiques obtenues respectent les hypothèses d'insensibilité aux délais des circuits asynchrones QDI.

Conclusion et Perspectives

Si la classe des circuits asynchrones a longtemps été un domaine de recherche confidentiel, il n'en reste pas moins que l'émergence de nouveaux paradigmes de conception tels que les circuits GALS et les réseaux sur puce entraîne un intérêt croissant pour les circuits asynchrones. Le premier chapitre a montré que les circuits asynchrones, et plus particulièrement les circuits QDI, permettent de répondre efficacement aux nouvelles contraintes qui découlent des progrès technologiques dans la conception des circuits numériques. Les circuits QDI sont notamment bien adaptés pour concevoir les réseaux sur puce des circuits GALS car ils permettent de s'affranchir des contraintes liées à la distribution du signal d'horloge et de prendre en charge de manière fiable et efficace les problèmes de synchronisation et d'arbitrage spécifiques à ces composants. Malgré ces avantages indéniables, l'utilisation de la logique asynchrone ne s'est toujours pas généralisée à cause du manque d'outil de conception adapté au flot de conception standard des industriels.

La solution à ce problème proposée par cette thèse est d'utiliser la bibliothèque SystemC pour concevoir des circuits asynchrones. Une première contribution de cette thèse a donc été d'identifier les limitations et les avantages de cette bibliothèque et de ses extensions TLM (cf. chapitre 2). Cette étude a montré qu'elles ne pouvaient pas être utilisées directement pour modéliser fidèlement le comportement d'un circuit asynchrone.

Un des points forts de la bibliothèque SystemC est d'être facilement extensible et de disposer des primitives de synchronisation (i.e. les notifications immédiates d'événements) nécessaires à la modélisation des circuits asynchrones. Une deuxième contribution de cette thèse a donc été d'exploiter ces propriétés pour développer la bibliothèque ASC qui permet de fidèlement modéliser en SystemC le comportement d'un circuit asynchrone et plus particulièrement celui d'un réseau sur puce asynchrone (cf. chapitre 3). Cette bibliothèque a été développée, en outre, avec la contrainte de pouvoir, à terme, employer un modèle ASC comme une entrée de l'outil de synthèse du flot de conception TAST. Finalement, la collaboration avec le CEA-LETI dans le cadre du projet NEVA a prouvé avec des exemples réels que cette bibliothèque est adaptée pour développer des circuits asynchrones complexes tels que des réseaux sur puce.

Au cours de l'utilisation de la bibliothèque ASC par le CEA-LETI est apparu une lacune importante de cette bibliothèque : les facilités de traçage standard de SystemC ne peuvent pas être employées pour valider le comportement d'un modèle ASC car ses communications sont synchronisées par défaut avec des notifications immédiates. Une troisième contribution de cette thèse a donc été de développer le modèle de temps distribué AST qui permet de dater de manière cohérente les événements survenant dans un circuit asynchrone (cf. chapitre 4). Ce modèle de temps a été formellement spécifié à l'aide d'une

relation d'ordre partiel respectant les propriétés de base des circuits asynchrones et d'une fonction d'estampillage temporel qui respecte cette relation. L'utilité pratique de cette fonction d'estampillage a notamment été démontrée en l'employant pour concevoir des facilités de traçage adaptées à la validation de modèle ASC. En outre, un correctif du simulateur SystemC de référence a été développé pour mieux assurer le respect de la propriété d'insensibilité aux délais d'un modèle ASC. La principale modification de ce correctif est de rendre non déterministe l'ordonnanceur de ce simulateur.

Une dernière contribution de cette thèse est d'avoir défini une méthode de présynthèse des structures de choix d'un langage de description matériel tel que ASC (cf. chapitre 5). Le principal apport de cette méthode est de prendre en charge des structures de choix qui ne sont généralement pas ou mal supportées par les outils de synthèse de circuits asynchrones (i.e. les structures de choix comportant des *probes* positifs et/ou négatifs et dont plusieurs gardes peuvent être simultanément vraies).

Si cette thèse a prouvé qu'un langage de description matériel de haut niveau tel que SystemC pouvait être utilisé efficacement pour concevoir des circuits asynchrones, il n'en reste pas moins qu'un travail important reste à accomplir avant que ces différents outils puissent être intégrés dans les flots de conception des industriels. La principale tâche à accomplir consiste à finir le travail sur la synthèse des modèles ASC qui a été initié. En outre, la bibliothèque ASC doit encore être un peu modifiée afin d'améliorer la vitesse de simulation d'un modèle ASC (e.g. implémentation des entiers insensibles aux délais) et de faciliter sa synthèse (e.g. structure de choix bloquante). Finalement, ce travail pourrait servir de base pour définir une extension de SystemC standard dédiée à la conception des circuits asynchrones. Ce standard pourrait notamment être développé au sein de l'OSCI de manière identique au standard TLM.

Annexe A

Modèle ASC des Canaux de Communication

A.1 Canal de Communication as_push

Extrait du code des méthodes `send` et `receive` du canal de communication `as_push`.

A.1.1 Méthode send

```
template< class T >
void as_push< T >::send(const T& p_data)
{
    // Data transfer
    this->a_data = p_data;

    // Request the transaction
    this->a_request_state = true;
    this->a_request_event.notify();

    // Wait after the transaction acknowledge
    this->wait(this->a_acknowledge_event);

    // Wait the latency of the acknowledge
    if(this->a_latency_ack.is_null() == false)
        this->wait(this->a_latency_ack.get_sc_time());

    // Consume the acknowledge
    this->a_acknowledge_state = false;
}
```

A.1.2 La Méthode receive

```
template< class T >
void as_push< T >::receive(T& p_data)
{
    // Wait after the transaction request
    if(this->a_request_state == false)
        this->wait(this->a_request_event);

    // Wait the latency of the request
    if(this->a_latency_req.is_null() == false)
        this->wait(this->a_latency_req.get_sc_time());
```

```
    // data transfer
    p_data = this->a_data;

    // Acknowledge of the transaction
    this->a_acknowledge_state = true;
    this->a_acknowledge_event.notify();
}
```

A.2 Canal de Communication as_pull

Extrait du code des méthodes `send` et `receive` du canal de communication `as_pull`.

A.2.1 send method

```
template< class T >
void as_pull< T >::send(const T& p_data)
{
    // Wait the transaction request
    if(this->a_request_state == false)
        this->wait(this->a_request_event);

    // Wait the latency of the request
    if(this->a_latency_req.is_null() == false)
        this->wait(this->a_latency_req.get_sc_time());

    // Consume the Request
    this->a_request_state = false;

    // data transfer
    this->a_data = p_data;

    // Acknowledge of the transaction
    this->a_acknowledge_state = true;
    this->a_acknowledge_event.notify();
}
```

A.2.2 receive method

```
template< class T >
void as_pull< T >::receive(T& p_data)
{
    // Request of the transaction
    this->a_request_state = true;
    this->a_request_event.notify();

    // Wait after the transaction acknowledge
    this->wait(this->a_acknowledge_event);

    // Wait the latency of the acknowledge
    if(this->a_latency_ack.is_null() == false)
        this->wait(this->a_latency_ack.get_sc_time());

    // data transfer
    p_data = this->a_data;
```

```
    // Consume the acknowledge
    this->a_acknowledge_state = false;
}
```

Annexe B

Les Formats de Trace Supportés par ASC

B.1 Syntaxe du Format de Trace AST

Un fichier de trace AST est composé d'un ensemble de commandes qui servent à définir la résolution du temps de simulation, les canaux qui sont observés et les événements qui surviennent sur ces canaux. Une commande est définie sur une ligne et son type est spécifié par le premier mot de cette ligne.

La syntaxe de la commande fixant la résolution du temps de simulation est définie par les pseudo règles EBNF suivantes :

$$
\begin{aligned}
\textit{time\_resolution\_command} &::= \texttt{STR}\ \textit{sc\_time} \\
\textit{sc\_time} &::= \textit{integer}\ \textit{time\_unit} \\
\textit{time\_unit} &::= \texttt{s} \mid \texttt{sec} \mid \texttt{ms} \mid \texttt{us} \mid \texttt{ns} \mid \texttt{ps} \mid \texttt{fs}
\end{aligned}
$$

La syntaxe de la commande permettant d'ajouter un canal de communication à observer est spécifiée par les pseudo règles EBNF suivantes :

$$
\begin{aligned}
\textit{channel\_command} &::= \texttt{CH}\ \textit{identifier}\ \textit{name}\ \textit{type}\ \textit{data\_type} \\
\textit{identifier} &::= \textit{integer} \\
\textit{name} &::= \textit{string} \\
\textit{type} &::= \texttt{PUSH} \mid \texttt{PULL} \\
\textit{data\_type} &::= \texttt{BOOL} \mid \texttt{CHAR} \mid \texttt{SHORT} \mid \texttt{INT} \mid \texttt{LONG} \mid \texttt{INT64} \mid \\
&\quad\ \texttt{UCHAR} \mid \texttt{USHORT} \mid \texttt{UINT} \mid \texttt{ULONG} \mid \texttt{UINT64} \mid \\
&\quad\ \texttt{FLOAT} \mid \texttt{DOUBLE} \mid \texttt{VOID}
\end{aligned}
$$

Le type de donnée `VOID` est utilisé pour les canaux qui utilisent des types de données non supportés par les facilités de traçage ASC. Chaque canal qui est observé doit posséder un unique *identifier*.

La commande permettant d'enregistrer l'occurrence d'un événement est spécifiée par les pseudo règles EBNF suivantes :

$$
\begin{aligned}
\textit{event\_command} &::= \textit{type}\ \textit{identifier}\ \textit{sc\_time}\ \textit{ast\_time} \mid \\
&\quad\ \textit{type}\ \textit{identifier}\ \textit{sc\_time}\ \textit{ast\_time}\ \textit{data} \\
\textit{type} &::= \texttt{SCA} \mid \texttt{SCP} \mid \texttt{ECA} \mid \texttt{ECP} \mid \texttt{PP} \mid \texttt{VP} \mid \texttt{NP} \\
\textit{ast\_time} &::= \textit{integer}
\end{aligned}
$$

Le champ *identifier* permet d'indiquer sur quel canal un événement survient. Seuls les événements du type `SCA`, `SCP` et `VP` sont autorisés à avoir un champ *data*. Un événement de type `SCA` ou `VP` possède un champ *data* si et seulement si il survient sur un canal de type `PUSH`. Un événement de type `SCP` possède un champ *data* si et seulement si il survient sur un canal de type `PULL`. Un événement de type `VP` ne peut survenir que sur un canal de type `PUSH`. Chacun des différents types d'événements correspond aux actions ASC suivantes :

- `SCA` : appel de la méthode `send` d'un canal `as_push` ou de la méthode `receive` d'un canal `as_pull`,
- `SCP` : appel de la méthode `receive` d'un canal `as_push` ou de la méthode `send` d'un canal `as_pull`,
- `ECA` : fin de l'appel de la méthode `send` d'un canal `as_push` ou de la méthode `receive` d'un canal `as_pull`,
- SCP : fin de l'appel de la méthode `receive` d'un canal `as_push` ou de la méthode `send` d'un canal `as_pull`,
- `PP` : *probe* positif d'un canal,
- `NP` : *probe* négatif d'un canal,
- `VP` : *probe* positif d'un canal `as_push` et qui renvoi la valeur communiquée par ce canal.

B.2 Exemple de Trace AST

```
STR 1 ps
CH 0 ch PUSH CHAR
SCA 0 0 s 1 4
SCP 0 0 s 1
ECP 0 0 s 2
NP 0 0 s 3
ECA 0 0 s 3
SCA 0 10 ns 5 8
```

B.3 Exemple de Trace VCD

```
$date   26/09/2008 15:54:45   $end

$version   ast2vcd   $end

$comment
VCD trace file generated by ast2vcd from an AST trace file.
$end

$timescale   1 ps   $end

$comment
time step between asynchronous events: 1250 ps
$end
```

```
$scope module SystemC $end
$var wire 1 clk sc_clock $end
$scope module ch $end
$var wire 1 p0 ch.probe $end
$var wire 1 sca0 ch.sca $end
$var wire 1 scp0 ch.scp $end
$var wire 1 eca0 ch.eca $end
$var wire 1 ecp0 ch.ecp $end
$var integer 8 d0 ch.data $end
$upscope $end
$upscope $end

$enddefinitions $end


$dumpvars
b0 clk
bX p0
b0 sca0
b0 scp0
b0 eca0
b0 ecp0
bX d0
$end

#0
b1 clk

#1250
b0 clk

#2500
b1 sca0
b100 d0
b1 scp0

#3750
b0 sca0
b0 scp0

#5000
b1 ecp0

#6250
b0 ecp0
```

```
#7500
b0 p0
b1 eca0

#8750
bX p0
b0 eca0
bX d0

#10000
b1 clk

#11250
b0 clk

#12500
b1 sca0
b1000 d0

#13750
b0 sca0
```

Annexe C

Configuration des Octagons ASC

C.1 Syntaxe du Fichier de Configuration

Un fichier de configuration est un ensemble d'affectations de variables et de commentaires. Une affectation ou un commentaire doit être défini sur une et une seule ligne. Une ligne commençant par le caractère `#` est un commentaire. Une affectation est définie en respectant la syntaxe définie par les pseudo règles EBNF suivantes :

$$
\begin{aligned}
\textit{assignment} &::= \textit{var\_name} := \textit{list}_{opt} \\
\textit{var\_name} &::= \textit{string} \\
\textit{list} &::= \textit{value} \mid \textit{value} \sqcup \textit{list} \\
\textit{value} &::= \textit{integer} \mid \textit{float} \mid \textit{string}
\end{aligned}
$$

Un premier ensemble de variables est destiné à configurer les générateurs de trafic et notamment à définir les transactions qu'ils doivent effectuer au cours d'une simulation. Les chaînes de caractères identifiant ces différentes variables sont définies de la manière suivante :

$$
\begin{aligned}
\textit{var\_generator} &::= \texttt{traffic\_generator\_}\ \textit{localisation}\ \texttt{-}\ \textit{property} \\
\textit{localisation} &::= \texttt{north} \mid \texttt{south} \mid \texttt{east} \mid \texttt{west} \mid \texttt{north\_east} \mid \\
&\quad\ \texttt{north\_west} \mid \texttt{south\_east} \mid \texttt{south\_west} \\
\textit{property} &::= \texttt{type} \mid \texttt{name} \mid \texttt{addresses} \mid \texttt{data}
\end{aligned}
$$

Les champs *localisation* et *property* permettent respectivement d'identifier un générateur et de fixer une de ses propriétés. La propriété `type` définit le type des transactions à effectuer : `READ` ou `WRITE`. Les adresses de destination des transactions à effectuer sont définies par la liste qui est affectée à la propriété `addresses`. Un élément de cette liste indique la localisation d'un module cible : `N`, `S`, `E`, `W`, `NE`, `NW`, `SE` ou `SW`. De même, les données à envoyer ou à recevoir sont spécifiées par la liste qui est affectée à la propriété `datas`. La propriété `name` définit le nom SystemC d'un module générateur.

Les paramètres des consommateurs de trafic sont spécifiés à l'aide d'un deuxième ensemble de variables dont les chaînes de caractères sont définies de la manière suivantes :

$$
\begin{aligned}
\textit{var\_consumer} &::= \texttt{traffic\_consumer\_}\ \textit{localisation}\ \texttt{-}\ \textit{property} \\
\textit{property} &::= \texttt{name} \mid \texttt{datas\_read} \mid \texttt{datas\_write}
\end{aligned}
$$

Les données à envoyer ou à recevoir (en réponse à des transactions de type READ ou WRITE) sont respectivement définies par les propriétés `data_read` et `data_write`.

Finalement, le nom du fichier de trace généré par une simulation est défini par la variable `trace_file_name`. [1]

C.2 Exemple de Fichier de Configuration

```
# Configuration file for the octagon test case


##############################################################################
# GLOBAL SETTINGS
trace_file_name := result-01


##############################################################################
# TRAFFIC GENERATORS

# North
traffic_generator_north-type         := READ
traffic_generator_north-name         := read_generator_north
traffic_generator_north-addresses    := SW
traffic_generator_north-datas        := 4

# South
traffic_generator_south-type         := READ
traffic_generator_south-name         := read_generator_south
traffic_generator_south-addresses    :=
traffic_generator_south-datas        :=

# East
traffic_generator_east-type          := READ
traffic_generator_east-name          := read_generator_east
traffic_generator_east-addresses     :=
traffic_generator_east-datas         :=

# West
traffic_generator_west-type          := READ
traffic_generator_west-name          := read_generator_west
traffic_generator_west-addresses     :=
traffic_generator_west-datas         :=

# North East
traffic_generator_north_east-type          := READ
traffic_generator_north_east-name          := read_generator_north_east
```

[1] l'extension « .ast » est automatiquement ajoutée au nom du fichier de trace.

```
traffic_generator_north_east-addresses     := NW
traffic_generator_north_east-datas         := 3

# North West
traffic_generator_north_west-type          := READ
traffic_generator_north_west-name          := read_generator_north_west
traffic_generator_north_west-addresses     :=
traffic_generator_north_west-datas         :=

# South East
traffic_generator_south_east-type          := READ
traffic_generator_south_east-name          := read_generator_south_east
traffic_generator_south_east-addresses     :=
traffic_generator_south_east-datas         :=

# South West
traffic_generator_south_west-type          := READ
traffic_generator_south_west-name          := read_generator_south_west
traffic_generator_south_west-addresses     :=
traffic_generator_south_west-datas         :=


##############################################################################
# TRAFFIC CONSUMERS

# North
traffic_consumer_north-name        := traffic_consumer_north
traffic_consumer_north-datas_read  :=
traffic_consumer_north-datas_write :=

# South
traffic_consumer_south-name        := traffic_consumer_south
traffic_consumer_south-datas_read  :=
traffic_consumer_south-datas_write :=

# East
traffic_consumer_east-name        := traffic_consumer_east
traffic_consumer_east-datas_read  :=
traffic_consumer_east-datas_write :=

# West
traffic_consumer_west-name        := traffic_consumer_west
traffic_consumer_west-datas_read  :=
traffic_consumer_west-datas_write :=

# North East
traffic_consumer_north_east-name        := traffic_consumer_north_east
```

```
traffic_consumer_north_east-datas_read  :=
traffic_consumer_north_east-datas_write :=

# North West
traffic_consumer_north_west-name        := traffic_consumer_north_west
traffic_consumer_north_west-datas_read  := 3
traffic_consumer_north_west-datas_write :=

# South East
traffic_consumer_south_east-name        := traffic_consumer_south_east
traffic_consumer_south_east-datas_read  :=
traffic_consumer_south_east-datas_write :=

# South West
traffic_consumer_south_west-name        := traffic_consumer_south_west
traffic_consumer_south_west-datas_read  := 4
traffic_consumer_south_west-datas_write :=
```

Bibliographie

[AEF+94] Aaron Ashkinazy, Doug Edwards, Craig Farnsworth, Gary Gendel et Shiv Sikand: *Tools For Validating Asynchronous Digital Circuits*. Dans *ASYNC'01: Proceedings of the 1st IEEE International Symposium on Asynchronous Circuits and Systems*, pages 12–21, Salt Lake City, UT, USA, 1994. ISBN 0-8186-6210-7.

[AG03] Adrijean Andriahantenaina et Alain Greiner: *Micro-network for SoC: Implementation of a 32-Port SPIN network*. Dans *DATE'03: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1128–1129, Munich, DE, 2003. IEEE Computer Society, ISBN 0-7695-1870-2.

[Ale01] Andrei Alexandrescu: *Modern C++ Design: Generic Programming and Design Patterns Applied*. C++ in Depth Series. Addison-Wesley Professional, Boston, MA, USA, 2001, ISBN 0-201-70431-5.

[ASBG02] Jean Marc Alliot, Thomas Schiex, Pascal Brisset et Frédérick Garcia: *Intelligence artificielle et Informatique théorique*, chapitre 6, pages 93–104. Cépadues, Toulouse, FR, deuxième édition, 2002, ISBN 2-85428-578-6.

[Aus98] Matthew Harold Austern: *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998, ISBN 0-201-30956-4.

[Aus01] Matthew Harold Austern: *The Standard Librarian: Defining Iterators and Const Iterators*. C/C++ Users Journal, 19(1):74–79, 2001. http://www.ddj.com/cpp/184401331.

[BCGK04] Evgeny Bolotin, Israel Cidon, Ran Ginosar et Avinoam Kolodny: *QNoC: QoS architecture and design process for network on chip*. Journal of Systems Architecture, 50(2-3):105–128, 2004.

[BCV+05] E. Beigne, F. Clermidy, P. Vivet, A. Clouard et M. Renaudin: *An Asynchronous NOC Architecture Providing Low Latency Service and its Multi-Level Design Framework*. Dans *ASYNC'05: Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 54–63, New York, NY, USA, 2005.

[BF01] W.J. Bainbridge et S.B. Furber: *Delay Insensitive System-on-Chip Interconnect using 1-of-4 Data Encoding*. Dans *ASYNC'01: Proceedings of the 7th International Symposium on Asynchronous Circuits and Systems*, pages 118–126, Salt Lake City, UT, USA, 2001. IEEE Computer Society.

[BF02] John Bainbridge et Steve Furber: *Chain: A Delay-Insensitive Chip Area Interconnect*. Micro, IEEE, 22(5):16–23, 2002, ISSN 0272-1732.

[Bje05] Tobias Bjerregaard: *The MANGO clockless network-on-chip: Concepts and implementation*. Thèse de doctorat, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2005. http://www2.imm.dtu.dk/pubdb/p.php?4025, Supervised by Assoc. Prof. Jens Sparsø, IMM.

[BKR+91] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs et Frits Schalij: *The VLSI-programming language Tangram and its translation into handshake circuits*. Dans *EDAC'91: Proceedings of the European Conference on Design Automation*, pages 384–389, Amsterdam, NH, 1991.

[BM88a] Steven M. Burns et Alain J. Martin: *Syntax-directed Translation of Concurrent Programs into Self-Timed Circuits*. Dans *Proceedings of the fifth MIT conference on Advanced research in VLSI*, pages 35–50, Cambridge, MA, USA, 1988. MIT Press, ISBN 0-262-01100-X.

[BM88b] Steven M. Burns et Alain J. Martin: *Synthesis of Self-Timed Circuits by Program Transformation*. Dans George J. Milne (éditeur): *The Fusion of Hardware Design and Verification*, pages 99–116, Amsterdam, NL, 1988. Elsevier Science Publishers B.V., ISBN 0-444-70532-5.

[BM02] Luca Benini et Giovanni De Micheli: *Networks on chips: a new SoC paradigm*. Computer, 35(1):70–78, 2002, ISSN 0018-9162.

[BM06] Tobias Bjerregaard et Shankar Mahadevan: *A Survey of Research and Practices of Network-on-Chip*. ACM Computing Surveys, 38(1):1–51, 2006, ISSN 0360-0300.

[BMOS05] Tobias Bjerregaard, Shankar Mahadevan, Rasmus Grøndahl Olsen et Jens Sparsø: *An OCP Compliant Network Adapter for GALS-based SoC Design Using the MANGO Network-on-Chip*. Dans *SOC'05: Proceedings of the 3rd International Symposium on System-on-Chip*, pages 171–174, Tampere, FI, 2005.

[BMS04] Tobias Bjerregaard, Shankar Mahadevan et Jens Sparsø: *A Channel Library for Asynchronous Circuit Design Supporting Mixed-Mode Modelling*. Dans *PATMOS'04: Proceedings of the 14th International Workshop on Power and Timing Modeling, Optimization and Simulation*, pages 301–310, Isle of Santorini, GR, 2004. Springer.

[Bre07] Vivian Bregier: *Synthèse automatisée de circuits asynchrones optimisés prouvés Quasi Insensibles aux Délais*. Thèse de doctorat, Institut National Polytechnique de Grenoble (INPG), 2007.

[BS04] Tobias Bjerregaard et Jens Sparsø: *Virtual Channel Designs for Guaranteeing Bandwidth in Asynchronous Network-on-Chip*. Dans *Norchip'04: Proceedings of the 22th Nordic Event in ASIC Design*, pages 269–272, Oslo, NO, 2004.

[BS05a] Tobias Bjerregaard et Jens Sparsø: *A Router Architecture for Connection-Oriented Service Guarantees in the MANGO Clockless Network-on-Chip*. Dans *DATE'05: Proceedings of the conference on Design, automation and test in Europe*, tome 2, pages 1226–1231, Munich, DE, 2005.

[BS05b] Tobias Bjerregaard et Jens Sparsø: *A Scheduling Discipline for Latency and Bandwidth Guarantees in Asynchronous Network-on-Chip*. Dans *ASYNC'05: Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 34–43, New York, NY, USA, 2005. IEEE Computer Society.

[BV06] E. Beigné et P. Vivet: *Design of On-chip and Off-chip Interfaces for a GALS NoC Architecture*. Dans *ASYNC'06: Proceedings of the 12th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 172–181, Grenoble, FR, 2006.

[CDYC97] Supratik Chakraborty, David L. Dill, Kenneth Y. Yun et Kun Yung Chang: *Timing Analysis for Extended Burst-Mode Circuits*. Dans *ASYNC'03: Proceedings of the 3rd IEEE International Symposium on Asynchronous Circuits and Systems*, pages 101–111, Eindhoven, NH, 1997. ISBN 0-8186-7922-0.

[CG03] Lukai Cai et Daniel Gajski: *Transaction Level Modeling: An Overview*. Dans *CODES+ISSS'03: Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 19–24, Newport Beach, CA, USA, 2003. ACM, ISBN 1-58113-742-7.

[CK05] Israel Cidon et Idit Keidar: *Zooming in on Network-on-Chip Architectures*. rapport technique, Technion Department of Electrical Engineering, Haifa, IS, 2005.

[CKK+97] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno et Alex Yakovlev: *Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers*. IEICE Transactions on Information and Systems, E80-D(3):315–325, 1997.

[CLM+04] Marcello Coppola, Riccardo Locatelli, Giuseppe Maruccia, Lorenzo Pieralisi et Alberto Scandurra: *Spidergon: a novel on-chip communication network*. Dans *SOC'04: Proceedings of the 2nd International Symposium on System-on-Chip*, pages 15–15, Tampere, FI, 2004.

[CM73] Thomas J. Chaney et Charles E. Molnar: *Anomalous Behavior of Synchronizer and Arbiter Circuits*. IEEE Transactions on Computers, 22(4):421–422, 1973.

[Cor94] Henk Corporaal: *Design of Transport Triggered Architectures*. Dans *GLSV'94: Proceedings of the 4th Great Lakes Symposium on VLSI*, pages 130–135, Notre Dame, IN, USA, 1994. ISBN 0-8186-5610-7.

[Dal92] William J. Dally: *Virtual-Channel Flow Control*. Parallel and Distributed Systems, IEEE Transactions on Parallel and Distributed Systems, 3(2):194–205, 1992, ISSN 1045-9219.

[DBL05] Yves Durand, Christian Bernard et Didier Lattard: *FAUST: On-Chip Distributed SoC Architecture for a 4G Baseband Modem Chipset*. Proc. Design and Reuse IP-SOC, pages 51–55, 2005.

[DDVC07] Anh Vu Dinh-Duc, Pascal Vivet et Alain Clouard: *A Transaction Level Modeling of Network-on-Chip Architecture for Energy Estimation*. Dans *RIVF'07: Proceedings of the 5th IEEE International Conference on Research, Innovation and Vision for the Future*, pages 58–64, Hanoi, VI, 2007. ISBN 1-4244-0694-3.

[Dij75] Edsger Wybe Dijkstra: *Guarded Commands, Nondeterminacy and Formal Derivation of Programs*. Commununications of the ACM, 18(8):453–457, 1975, ISSN 0001-0782.

[Dij79] Edsger Wybe Dijkstra: *Go To Statement Considered Harmful*. Dans Edward Nash Yourdon (éditeur): *Classics in Software Engineering*, pages 27–33. Yourdon Press, Upper Saddle River, NJ, USA, 1979, ISBN 0-917072-14-6.

[DS87] William J. Dally et Charles L. Seitz: *Deadlock-Free Message Routing in Multiprocessor Interconnection Networks*. Computers, IEEE Transactions on Computers, C-36(5):547–553, 1987, ISSN 0018-9340.

[DT01] William J. Dally et Brian Towles: *Route Packets, Not Wires: On-Chip Inteconnection Networks*. Dans *DAC'01: Proceedings of the 38th conference on Design automation*, pages 684–689, Las Vegas, NV, USA, 2001. ACM, ISBN 1-58113-297-2.

[DYL02] Jose Duato, Sudhakar Yalamanchili et Ni Lionel: *INTERCONNECTION NETWORKS: AN ENGINEERING APPROACH*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002, ISBN 1-55860-852-4.

[EAJ+06] Doug Edwards, Bardsley Andrew, Lilian Janin, Luis Plana et Will Toms: *Balsa: A Tutorial Guide*. Manchester University, 2006. Version V3.5, http://www.cs.manchester.ac.uk/apt/projects/tools/balsa.

[ET04] D. A. Edwards et W. B. Toms: *Design, Automation and Test for Asynchronous Circuits and Systems*. Rapport technique IST-1999-29119, Information Society Technologies (IST) Program Concerted Action Thematic Network Contract, 2004.

[FNT+99] Robert M. Fuhrer, Steven M. Nowick, M. Theobald, Niraj K. Jha, Bill Lin et Luis Plana: *MINIMALIST: An environment for the synthesis, verification and testability of burst-mode asynchronous machines*. Rapport technique TR CUCS-020-99, Department of Computer Science, Columbia University Technical Report, New York, NY, USA, 1999.

[Fol07] Bertrand Folco: *Contribution à la synthèse des circuits asynchrones Quasi Insensibles aux Délais, applications aux systèmes sécurisés*. Thèse de doctorat, Institut National Polytechnique de Grenoble (INPG), 2007.

[Fri75] Arthur D. Friedman: *LOGICAL DESIGN OF DIGITAL SYSTEMS*. W. H. Freeman & Co., New York, NY, USA, 1975, ISBN 0-914894-50-1.

[GDR05] Kees Goossens, John Dielissen et Andrei Rãdulescu: *Æthereal Network on Chip: Concepts, Architectures, and Implementations*. Design & Test of Computers, IEEE, 22(5):414–421, 2005, ISSN 0740-7475.

[GG00] Pierre Guerrier et Alain Greiner: *A Generic Architecture for On-Chip Packet-Switched Interconnections*. Dans *DATE'00: Proceedings of the conference on Design, automation and test in Europe*, pages 250–256, Paris, FR, 2000. ACM, ISBN 1-58113-244-1.

[Ghe06] Frank Ghenassia: *TRANSACTION-LEVEL MODELING WITH SYSTEMC: TLM Concepts and Applications for Embedded Systems*. Springer-Verlag, New York, Inc. Secaucus, NJ, USA, 2006, ISBN 978-0-387-26232-1.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995, ISBN 0-201-63361-2.

[Gin03] Ran Ginosar: *Fourteen Ways to Fool Your Synchronizer*. Dans *ASYNC'03: Proceedings of the 9th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 89–96, Vancouer, CA, 2003.

[GN92] Christopher J. Glass et Lionel M. Ni: *The turn model for adaptive routing*. International Symposium on Computer Architecture, 19(21):278–287, 1992.

[Gre05] GreenSoCs: *TAC : Transaction Accurate Communication/Channel*, 2005. http://www.greensocs.com/en/projects/TACPackage.

[Hel07] Claude Helmstetter: *Validation de Modèles de Systèmes sur Puce en Présence d'Ordonnancements Indéterministes et de Temps Imprécis*. Thèse de doctorat, Institut National Polytechnique de Grenoble (INPG), 2007. http://www-verimag.imag.fr/~helmstet/these-fr.html.

[HMMCM06] Claude Helmstetter, Florence Maraninchi, Laurent Maillet-Contoz et Matthieu Moy: *Automatic Generation of Schedulings for Improving the Test Coverage of Systems-on-a-Chip*. Dans *FMCAD'06: Formal Methods in Computer-Aided Design*, pages 171–178, San Jose, CA, USA, 2006. IEEE Computer Society, ISBN 0-7695-2707-8.

[Hoa78] C.A.R. Hoare: *Communicating Sequential Processes*. Commununications of the ACM, 21(8):666–677, 1978, ISSN 0001-0782.

[IEE04] IEEE Std 1364-2001: *Behavioural languages - Part 4: Verilog hardware description language*, chapitre 18, pages 349–374. New York, NY, USA, 2004, ISBN 2-8318-7675-3.

[IEE06] IEEE Std 1666-2005: *IEEE Standard SystemC Language Reference Manual*. New York, NY, USA, 2006, ISBN 0-7381-4871-7. http://standards.ieee.org/getieee/1666/index.html.

[KGGV07] Milos Krštić, Eckhard Grass, Frank K. Gürkaynak et Pascal Vivet: *Globally Asynchronous, Locally Synchronous Circuits: Overview and Outlook*. Design & Test of Computers, IEEE, 24(5):430–441, 2007, ISSN 0740-7475.

[KHR07] Cédric Koch-Hofer et Marc Renaudin: *Timed Asynchronous Circuits Modeling using SystemC*. Dans *FDL'07: Proceedings of the 9th Forum on specification and Design Languages*, Barcelona, ES, 2007. ECSI.

[KHRTV07] Cédric Koch-Hofer, Marc Renaudin, Yvain Thonnart et Pascal Vivet: *ASC, a SystemC extension for Modeling Asynchronous Systems, and its application to an Asynchronous NoC*. Dans *NOC'07: First International Symposium on Networks-on-Chip*, pages 295–306, Princeton, NJ, USA, 2007. ISBN 0-7695-2773-6.

[KHTM07] Tareq Hasan Khan, Ali Habibi, Sofiène Tahar et Otmane Ait Mohamed: *Automatic Generation of Transactors in SystemC*. Dans *FDL'07: Proceedings of the 9th Forum on specification and Design Languages*, Barcelona, ES, 2007. ECSI.

[Kin08] David J. Kinniment: *Synchronization and Arbitration in Digital Systems*. John Wiley & Sons Inc., New York, NY, USA, 2008, ISBN 0-470-51082-X.

[KND02] Faraydon Karim, Anh Nguyen et Sujit Dey: *An Interconnect Architecture for Networking Systems on Chips*. Micro, IEEE, 22(5):36–45, 2002, ISSN 0272-1732.

[KNDR01] Faraydon Karim, Anh Nguyen, Sujit Dey et Ramesh Rao: *On-Chip Communication Architecture for OC-768 Network Processors*. Dans *DAC'01: Proceedings of the 38th conference on Design automation*, pages 678–683, Las Vegas, NV, USA, 2001. ACM, ISBN 1-58113-297-2.

[KR99] Per Arne Karlsen et Per Torstein Røine: *A Timing Verifier and Timing Profiler for Asynchronous Circuits*. Dans *ASYNC'05: Proceedings of the 5th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 13–23, Barcelona, ES, 1999. ISBN 0-7695-0031-5.

[Lam78] Leslie Lamport: *Time, Clocks, and the Ordering of Events in a Distributed System*. Commununications of the ACM, 21(7):558–565, 1978, ISSN 0001-0782.

[Mar84] Alain J. Martin: *The Probe: An Addition to Communication Primitives*. Rapport technique CaltechCSTR:1984.5124-tr-84, California Institute of Technology, Pasadena, CA, USA, 1984.

[Mar90a] Alain J. Martin: *Limitations to Delay-Insensitivity in Asynchronous Circuits*. Rapport technique CaltechCSTR:1990.cs-tr-90-02, California Institute of Technology, Pasadena, CA, USA, 1990.

[Mar90b] Alain J. Martin: *Programming in VLSI: from communicating processes to delay-insensitive circuits*. Dans C. A. R. Hoare (éditeur): *Developments in concurrency and communication*, The UT Year of Programming Series, chapitre 1, pages 1–64. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA, 1990, ISBN 0-201-17232-1.

[Mar91] Alain J. Martin: *Synthesis of Asynchronous VLSI Circuits*. Rapport technique CaltechCSTR:1991.cs-tr-93-28, California Institute of Technology, Pasadena, CA, USA, 1991.

[MB06] Giovanni De Micheli et Luca Benini: *NETWORKS ON CHIPS: TECHNOLOGY AND TOOLS*. SYSTEMS ON SILICON. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006, ISBN 0-12-370521-5.

[MCM+04] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller et Luciano Ost: *HERMES: an infrastructure for low area overhead packet-switching networks on chip*. INTEGRATION, the VLSI journal, 38(1):69–93, 2004, ISSN 0167-9260.

[MM95] Rajit Manohar et Alain J. Martin: *Quasi-Delay-Insensitive Circuits are Turing-Complete*. Rapport technique CaltechCSTR:1995.cs-tr-95-11, California Institute of Technology, Pasadena, CA, USA, 1995.

[MN06] Alain J. Martin et Mika Nyström: *Asynchronous Techniques for System-on-Chip Design*. Proceedings of the IEEE, 94(6):1089–1120, June 2006, ISSN 0018-9219.

[MPGA06] Ivan Miro Panades, Alain Greiner et Sheibanyrad Abbas: *Micro-réseau sur puce compatible avec l'approche GALS*. Dans *JNRDM'06: Journées Nationales du Réseau Doctoral en Microélectronique*, Rennes, FR, 2006.

[MPGS06] Ivan Miro Panades, Alain Greiner et Abbas Sheibanyrad: *A Low Cost Network-on-Chip with Guaranteed Service Well Suited to the GALS Approach*. Dans *NanoNet'06: proceedings of the first International Conference ON nANO-nETWORKS*, pages 1–5, Lausane, CH, 2006.

[Mye01] Chris J. Myers: *ASYNCHRONOUS Circuit Design*. John Wiley & Sons Inc., New York, NY, USA, 2001, ISBN 0-471-41543-X.

[NM02] Mika Nyström et Alain J. Martin: *Crossing the Synchronous-Asynchronous Divide*. Dans *WCED'02: Proceedings of the Workshop on Complexity-Effective Design*, Anchorage, Alaska, USA, 2002. http://www.ece.rochester.edu/~albonesi/wced02/program.html.

[OSC02] OSCI (Open SystemC Initiative): *Functional Specification for SystemC 2.0*, 2002. Update for SystemC 2.0.1, version 2.0-Q.

[OSC03] OSCI (Open SystemC Initiative): *SystemC 2.0.1 Language Reference Manual*, 2003. Revision 1.0.

[OSC05] OSCI (Open SystemC Initiative): *OSCI TLM2 USER MANUAL*, 2005. http://www.systemc.org/downloads/standards/tlm20/.

[Paa95] Jukka Paakki: *Attribute Grammar Paradigms—A High-Level Methodology in Language Implementation*. ACM Computing Surveys, 27(2):196–255, 1995, ISSN 0360-0300.

[Par02] Sudeep Parsiha: *Transaction level modeling of SoC with SystemC 2.0*. SNUG'02: Synopsys User Group Conference, 2002.

[PP05] Dan Pilone et Neil Pitman: *UML 2.0 in a Nutshell*. In a Nutshell. O'Reilly Media, Inc., Sebastopol, CA, USA, 2005, ISBN 0-596-00795-7.

[Qua04] Jérôme Quartana: *Design of Asynchronous Network on Chip: application to GALS systems*. Thèse de doctorat, Institut National Polytechnique de Grenoble (INPG), 2004.

[Ren00] M. Renaudin: *Asynchronous circuits and systems: a promising design alternative*. Microelectronic Engineering, 54(1-2):133–149, 2000, ISSN 0167-9317.

[Ren02] Marc Renaudin: *Etat de l'art sur la conception des circuits asynchrones: perspectives pour l'intégration des systèmes complexes*. Rapport technique TIMA-RR–02/12-02–FR, Institut National Polytechnique de Grenoble (INPG), Grenoble, FR, 2002.

[Rom06] Lemaire Romain: *Conception et modélisation d'un système de contrôle d'applications de télécommunication avec une architecture de réseau sur puce (NoC)*. Thèse de doctorat, Institut National Polytechnique de Grenoble (INPG), 2006.

[Rot07] Florent Rotenberg: *Conception et modélisation d'une architecture superscalaire et modulaire de traitement du signal fondée sur la logique asynchrone*. Diplôme de recherche technologique, Institut National Polytechnique de Grenoble (INPG), 2007.

[RSPF05] Adam Rose, Stuart Swan, John Pierce et Jean-Michel Fernandez: *Transaction Level Modeling in SystemC*. rapport technique, Open SystemC Initiative, 2005.

[SA04] Herb Sutter et Andrei Alexandrescu: *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. C++ in Depth Series. Addison-Wesley Professional, Boston, MA, USA, 2004, ISBN 0-321-11358-6.

[SE89] Ivan E. Sutherland et Jo Ebergen: *Computers without Clocks*. Communications of the ACM, 32(6):720–738, 1989.

[SF01] Jens Sparsø et Steve Furber: *Principles of Asynchronous Circuit Design: A Systems Perspective*, pages 153–218. Kluwer Academic Publishers, Dordrecht, NL, 2001, ISBN 0-7923-7613-7.

[SG08] Abbas Sheibanyrad et Alain Greiner: *Two efficient synchronous ⇔ asynchronous converters well-suited for networks-on-chip in GALS architectures*. Integration, the VLSI Journal, 41(1):17–26, 2008, ISSN 0167-9260. http://www.sciencedirect.com/science/article/B6V1M-4NNWCBB-1/2/4e874ad9fcdb7aa26e3317c03f3b38c4.

[SMPG07] Abbas Sheibanyrad, Ivan Miro Panades et Alain Greiner: *Systematic Comparison between the Asynchronous and the Multi-Synchronous Implementations of a Network on Chip Architecture*. Dans *DATE'07: Proceedings of the conference on Design, automation and test in Europe*, pages 1090–1095, Nice, FR, 2007. EDA Consortium, ISBN 978-3-9810801-2-4.

[Str97] Bjarne Stroustrup: *The C++ Programming Language, Special Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997, ISBN 0-201-70073-5.

[Sut89] Ivan E. Sutherland: *Micropipelines*. Commununications of the ACM, 32(6):720–738, 1989, ISSN 0001-0782.

[Ver88] Tom Verhoeff: *Delay-insensitive codes – overview*. Distributed Computing, 3(1):1–8, 1988.

[VPG06] Emmanuel Viaud, François Pêcheux et Alain Greiner: *An Efficient TLM/T Modeling and Simulation Environment Based on Conservative Parallel Discrete Event Principles*. Dans *DATE'06: Proceedings of the conference on Design, automation and test in Europe*, pages 94–99, Munich, DE, 2006. European Design and Automation Association, ISBN 3-9810801-0-6.

[WM01] Catherine .G. Wong et Alain J. Martin: *Data-Driven Process Decomposition For Circuit Synthesis*. Dans *ICECS'01: Proceedings of the IEEE International Conference on Electronics, Circuits and Systems*, tome 1, pages 539–546, Malte, MT, 2001.

[WM03] Catherine .G. Wong et Alain J. Martin: *High-Level Synthesis of Asynchronous Systems by Data-Driven Decomposition*. Dans *DAC '03: Proceedings of the 40th conference on Design automation*, pages 508–513, Anaheim, CA, USA, 2003. ACM, ISBN 1-58113-688-9.

[YD92] Keneth Y. Yun et David L. Dill: *Automatic synthesis of 3D asynchronous state machines*. Dans *ICCAD'92: Proceedings of the International Conference on Computer-Aided Design*, pages 576–580, Santa Clara, CA, USA, 1992. IEEE Computer Society Press, ISBN 0-89791-540-2.

[YR06] Eslam Yahya et Marc Renaudin: *QDI Latches Characteristics and Asynchronous Linear-Pipeline Performance Analysis*. Dans *PATMOS'06: Pro-*

ceedings of the 16th International Workshop on Power and Timing Modeling, Optimization and Simulation, pages 583–592, Montpellier, FR, 2006. Springer.

Glossaire

A

act *activation*
ALG *Asynchronous Latency Guarantees*
ALG *Asynchronous Latency Guarantees*
ANOC *Asynchronous Network on Chip*
ASC *Asynchronous SystemC*
ASP *Asynchronous SystemC Presynthesis*
ASPIN *Asynchronous Scalable Predictable Interconnect Network*
AST *Asynchronous SystemC Time*
AST *Asynchronous SystemC Trace*
AXI *Advanced eXtensible Interface*

B

BDD *Binary-Decision Diagram*

C

CAST *Caltech Asynchronous Synthesis Tools*
CEA *Commissariat à l'Énergie Atomique*
CHP *Concurrent Hardware Processes*
ch *Channel*
cmd *Command*
CMP *Chip MultiProcessors*
com *Communication*
CSP *Concurrent Sequential Processes*
csp *Create Sub Processes*

D

DI *Delay Insensitive*
DSPIN *Distributed Scalable Predictable Interconnect Network*
DTU *Technical University of Denmark*

E

EBNF *Extended Backus Normal Form*
eca *End Communication Active*
ecp *End Communication Passive*
esp *End Sub Processes*

F

FAUST *Flexible Architecture of Unified System for Telecom*
FIFO *First In First Out*

G

GALS *Globaly Asynchronous Localy Synchronous*
ge *Global Event*

I

IEEE *Institute of Electrical and Electronics Engineers*
IMC *Interface Method Call*
init *Initialisation*
IPs *Intellectual Properties*

L

le *Last Event*
LETI *Laboratoire d'Électronique et de Technologies de l'Information*
lge *Last Global Event*

M

MANGO *Message-passing Asynchronous Network-on-Chip Guaranteed services over OCP interfaces*
MC-CDMA *Multiple Carrier-Coded Division Multiple Acces*
MDD *Multi-Decision Diagram*

N

NoC *Network on Chip*
np *Negative Probe*

O

OCP *Open Core Protocol*
OSCI *Open SystemC Initiative*
OTA *Operation Triggered Architecture*

P

p *Process*
pp *Positive Probe*
PV *Programmer View*
PVT *Programmer View plus Timing*

Q

QDI *Quasi Delay Insensitive*

S

sca *Start Communication Active*
scp *Start Communication Passive*
SI *Speed Independent*
SoC *System on Chip*
SPIN *Scalable Programmable Integrated Network*
STG *State Transition Graph*

STL *Standard Template Library*

T

TAC *Transaction Accurate Communication*
TAST *TIMA Asynchronous Synthesis Tools*
TDM *Time Division Multiplexing*
TiDE *Timeless Design Environment*
TIMA *Techniques de l'Informatique et de la Microéléctronique pour l'Architecture des systèmes intégrés*
TLM *Transaction Level Modeling*
TLM-PV *TLM-Programmer View*
TLM-PVT *TLM-Programmer View plus Timing*
TTA *Transport Triggered Architecture*

U

UML *Unified Modeling Language*

V

VCD *Value Change Dump*
VHDL *Very high speed integrated circuit Hardware Description Language*

Index

R

S

T

U

V

W

Publications de l'Auteur

Chapitre de Livre

Cédric Koch-Hofer et Marc Renaudin : *Timed Asynchronous Circuits Modeling and Validation using SystemC*. Dans Eugenio Villar (éditeur) : *Embedded Systems Specification and Design Languages : Selected Contributions from FDL'07*, chapitre 2, pages 15–29. Springer Netherlands, 2008, ISBN 978-1-4020-8296-2.

Conférences Internationales

Cédric Koch-Hofer et Marc Renaudin : *Timed Asynchronous Circuits Modeling using SystemC*. Dans *FDL'07 : Proceedings of the 9th Forum on specification and Design Languages*, Barcelona, ES, 2007.

Cédric Koch-Hofer, Marc Renaudin, Yvain Thonnart et Pascal Vivet : *ASC, a SystemC extension for Modeling Asynchronous Systems, and its application to an Asynchronous NoC*. Dans *NOC'07 : First International Symposium on Networks-on-Chip*, Princeton, NJ, USA, 2007. ISBN 0-7695-2773-6

Workshop

Cédric Koch-Hofer et Marc Renaudin : *Modeling and synthesis of Asynchronous Network on Chip using SystemC*, Dans *Special Workshop on Future Interconnects and Networks on Chip*, Munich, DL, 2006.

Modélisation, Validation et Présynthèse de Circuits Asynchrones en SystemC

Résumé – Avec les progrès technologiques en microéléctronique, les méthodes de conception traditionnelles « tout synchrone » atteignent leurs limites. Une solution efficace pour résoudre ce problème est de diviser un circuit en plusieurs domaines d'horloge indépendants et de faire communiquer leurs composants avec un réseau sur puce asynchrone. Toutefois, la généralisation de cette solution est limitée par le manque d'outils adaptés à la conception de circuits asynchrones complexes tels que des réseaux sur puce asynchrones.

Une contribution de cette thèse, pour pallier cette limitation, a été de développer la bibliothèque ASC qui permet de modéliser fidèlement en SystemC des circuits asynchrones insensibles aux délais. Des facilités de traçage basées sur un modèle de temps distribué ont également été développées pour être en mesure de valider par simulation le comportement d'un modèle ASC. Une dernière contribution de cette thèse a été de définir une méthode de présynthèse des structures de choix qui prennent en compte efficacement les primitives de synchronisation spécifiques aux circuits asynchrones.

Mots Clefs : logique asynchrone, réseaux sur puce, modélisation, SystemC, validation, modèle de temps, présynthèse, structure de choix, synchronisation et exclusion mutuelle.

Modeling, Validation and Presynthesis of Asynchronous Circuits in SystemC

Abstract – With the technological advances in microelectronics, the traditional "fully synchronous" design methods of design are reaching their limits. An efficient solution to address this problem is to divide a circuit in several independent clock domains and to interconnect them with an asynchronous network on chip. However, the generalization of this solution is restricted by the lack of tools adapted to the design of complex asynchronous circuits like asynchronous network on chips.

A contribution of this thesis, for lifting this restriction, has been to develop the ASC library to properly model delay insensitive asynchronous circuits in SystemC. Tracing facilities, based on a distributed timing model, were also developed to allow the validation by simulation of an ASC model. The last contribution of this thesis has been to define a presynthesis method for conditional statements which efficiently handles the synchronization mechanisms specific to asynchronous circuits.

Keywords : asynchronous logic, network on chip, modeling, SystemC, validation, timing model, presynthesis, conditional statement, synchronization and mutual exclusion.

Adresse : Laboratoire TIMA, 46 Avenue Félix Viallet, 38031 Grenoble Cedex, France.
ISBN : 978-2-84813-131-3