2M11.3011.5

Université de Montréal

**Évaluation de la Performance dans la Modélisation SystemC de Systèmes Multiprocesseur à base de Processeur Réseau**

Par
Nadir Boudina

Département d'Informatique et de Recherche Opérationnelle
Faculté des Études Supérieures

Mémoire présenté à la Faculté des Études Supérieures
En vue de l'obtention du grade de Maître Ès Sciences
en M.Sc.Informatique

Septembre, 2002

©, Nadir Boudina, 2002

QA
76
N54
2002
no.053

Université de Montréal

Faculté des Études Supérieures

Ce mémoire intitulé :

**Évaluation de la Performance dans la Modélisation SystemC de Systèmes Multiprocesseur à base de Processeur Réseau**

Présenté par :

Nadir Boudina

a été évalué par un jury composé des personnes suivantes :

Gena Hahn , président-rapporteur

El Mostapha Aboulhamid, membre et directeur de recherche

François Major , membre du jury

Mémoire accepté le 6 novembre 2002

# RÉSUMÉ

Les problèmes introduits par la nouvelle technologie de design de systèmes matériel/logiciel sont nombreux. L'augmentation de la complexité de conception exige de très rapides spécifications exécutables pour valider les concepts système et seul *C/C++* peut fournir à la fois les niveaux adéquats d'abstraction, l'intégration matériel/logiciel et la performance. Les ingénieurs de logiciel emploient *C/C++*. Les architectes de système emploient C/C++. La plupart des concepteurs de matériel sont familiers avec *C/C++*. Cela fait de *C/C++* le langage naturel pour la conception de systèmes embarqués.

*SystemC* est un standard émergent de plate-forme de modélisation basée sur *C++* qui adresse les problèmes posés par la conception *SoC* et prend en charge l'abstraction de conception aux niveaux *RTL*, comportemental et système.

Actuellement, le moteur de simulation de *SystemC* n'implémente pas de mécanisme permettant de réaliser de la simulation distribuée. La complexité des systèmes à simuler ainsi que le besoin de performance, sans cesse croissants, devraient mener *SystemC* à proposer, comme cela a été le cas pour les autres *HDL*, de telles fonctionnalités dans les meilleurs délais.

Le travail présenté par ce mémoire a pour but principal de montrer qu'avec les nouvelles fonctionnalités introduites par *SystemC 2.0* permettant de profiter pleinement des puissantes capacités de la programmation orientée objet, il devient alors possible de croire à une future implémentation d'un environnement *SystemC* offrant de la simulation distribuée.

Pour arriver à cette conclusion, nous avons utilisé un modèle *SystemC* existant d'un multiprocesseur constitué par un anneau de processeurs *DLX* et n'offrant aucune fonctionnalité de simulation distribuée. Nous avons alors développé un nouveau canal de communication *SystemC* utilisant le protocole *TCP/IP* et l'avons utilisé pour distribuer le modèle sur un processeur réseau. Nous avons pu alors simuler plusieurs configurations de répartition des processeurs DLX sur le processeur

réseau et ainsi évaluer l'accroissement de performance introduit par la distribution de la simulation en fonction de la complexité du modèle,. Nous avons ensuite profité de la disponibilité de ce modèle distribué pour évaluer la différence de performance (toujours en fonction de la complexité du modèle) qui existe entre deux différentes méthodologies de modélisation du parallélisme disponibles dans *SystemC* (`SC_THREAD` et `SC_METHOD`).

Enfin, pour montrer qu'une simulation distribuée utilisant des canaux natifs de communication est bien plus performante qu'une implémentation de fonctionnalités étendues de communication à l'aide de librairies logicielles de parallélisation déjà existantes, comme *MPI*, nous avons développé une version *MPI* de notre programme et avons simulé le même modèle dans les mêmes conditions.

Mots clés : **SystemC, canal TCP/IP, simulation distribuée synchrone, modélisation multiprocesseur, processeur réseau.**

# ABSTRACT

Problems introduced by the new technology of hardware/software system design are numerous. The increase of the complexity of conception requires very fast executable specifications to confirm system concepts and only *C/C++* can supply at the same moment the adequate levels of abstraction, the hardware/software integration and the performance. The software engineers as well as system architects use *C/C++*. Most of the designers of hardware are familiar with *C/C++*. This makes *C/C++* the natural language for the embedded systems design.

*SystemC* is an emerging standard modeling platform based on *C++* that addresses the issues discussed above, and supports design abstraction at the *RTL*, behavioral, and system levels.

At present, the simulation engine of *SystemC* does not implement a mechanism allowing the supply of distributed simulation. The complexity of systems to be simulated as well as the need of performance ceaselessly increasing should lead *SystemC* to propose, as it was the case for the other *HDL*, such features as soon as possible.

Work presented by this report aims, first of all, to show that with new features introduced by *SystemC 2.0* allowing completely to take advantage of powerful capacities of the object oriented programming, it becomes then possible to consider in an intended implementation of a `SystemC` environment offering distributed simulation. To arrive at this conclusion, we used an existing `SystemC` model of a multi-processor constituted with a ring of `DLX` processors and not offering any feature of distributed simulation. We then developed a new *SystemC* communication channel using the *TCP/IP* protocol and used it to distribute the model on a network processor. We were able to then simulate several configurations of distribution of *DLX* processors on network processor and so to estimate the increase of performance, according to the complexity of the model, introduced by the distribution of the simulation. We took advantage then of the availability of this distributed model also

to estimate the difference of performance, always according to the complexity of the model, which exists between two various methodologies to model concurrency offered by *SystemC*: `SC_METHOD` and `SC_THREAD`.

Finally, to show that a distributed simulation using native channels of communication is more successful than an implementation of such widened communication functionalities by external software library like *MPI*, we developed a *MPI* version of our program and simulated the same model in the same conditions.

Keywords: **SystemC, TCP/IP chanel, synchronous distributed simulation, multi-processor modeling, network processor**.

# TABLE DES MATIÈRES

RÉSUMÉ .......... iv

ABSTRACT .......... vi

TABLE DES MATIÈRES .......... viii

LISTE DES TABLEAUX .......... x

LISTE DES FIGURES .......... xi

LISTE DES ABRÉVIATIONS .......... xii

LEXIQUE .......... xiv

DÉDICACE .......... xvii

REMERCIEMENTS .......... xviii

INTRODUCTION .......... 1

CHAPITRE 1 : ÉTAT DE L'ART .......... 5

1.1. SystemC 1.0 .......... 11
1.2. SystemC 2.0 .......... 14
1.3. La simulation distribuée .......... 28
1.3.1 La simulation .......... 28
1.3.2 La simulation à évènements discrets .......... 31
1.3.3 Accélération par parallélisation .......... 33
1.3.3.1 Méthodes de distribution .......... 34
1.3.3.2 Le principe de la causalité .......... 35
1.3.4 Evolution synchrone des processus .......... 37
1.3.5 Evolution asynchrone des processus .......... 38
1.3.5.1 Approches conservatives (ou pessimistes) .......... 40
1.3.5.1.1 Principe .......... 40
1.3.5.1.2 Prévention des interblocages .......... 43
1.3.5.1.3 Détection/Résolution des interblocages .......... 43
1.3.5.1.4 Optimisations .......... 44
1.3.5.1.5 Difficultés .......... 45
1.3.5.2 Approche spéculative (ou optimiste) .......... 46
1.3.5.2.1 Principe .......... 46

1.3.5.2.2 Optimisations ... 52
1.3.5.2.3 Difficultés ... 52
1.3.5.3 Approches hybrides ... 53
1.3.5.3.1 Approches mixtes ... 54
1.3.5.3.2 Approche adaptative ... 55
1.3.6 Performances ... 57
1.3.6.1 Performances analytiques ... 57
1.3.6.2 Performances expérimentales ... 58
1.3.6.2.1 Approches conservatives ... 58
1.3.6.2.2 Approches spéculatives ... 59
1.3.7 Conclusion ... 60
1.4 Le moteur de simulation de SystemC ... 61

CHAPITRE 2 : MÉTHODOLOGIE ... 64
2.1 Le modèle du *DLX* ... 64
2.2 Le modèle multiprocesseur ... 66
2.3 Le modèle multiprocesseur distribué par extension du canal élémentaire de communication *sc_signal*: Canal natif de communication TCP/IP ... 72
2.3.1 Un exemple : simulation d'un protocole simple de communication .. 72
2.3.2 Extension du canal primaire de communication : *Simplex* version distribuée (Figure 28) ... 76

CHAPITRE 3 : EXPÉRIMENTATION, PRÉSENTATION ET ANALYSE DES RÉSULTATS ... 85
3.1 Utilisation d'une machine Linux à 450 MHz [CAPP02] ... 85
3.2 Simulation distribuée : Utilisation d'un processeur réseau de 32 machines Linux à 1,2 GHz connectées à travers un commutateur Ethernet 1 Gbit ... 89

DISCUSSION GÉNÉRALE ET CONCLUSION ... 94

BIBLIOGRAPHIE ... 96

ANNEXE : L'ORDONNANCEUR SYSTEMC 2.0 ... xix

# LISTE DES TABLEAUX


Tableau I Actions entreprises lors de la réception d'un message ........................ 50
Tableau II Performance du modèle DLX monoprocesseur................................. 85
Tableau III Temps de simulation du multiprocesseur non distribué...................... 86
Tableau IV Fréquence combinée du multiprocesseur non distribué ...................... 86
Tableau V Temps de simulation du modèle multiprocesseur distribué ............... 89
Tableau VI Temps de simulation des SC_THREAD du modèle distribué............. 90
Tableau VII Fréquence « combinée » du modèle multiprocesseur distribué.......... 91
Tableau VIII Temps de simulation avec *MPI* ........................................................ 93

# LISTE DES FIGURES


Figure 1 Méthodologie de Design « Description et Synthèse » ... 6
Figure 2 Processus de Synthèse/ Implémentation ... 8
Figure 3 Méthodologie de design avec *SystemC* ... 9
Figure 4 La simulation dans `SystemC` ... 10
Figure 5 Illustration des modules, ports, signaux, et de la hiérarchie ... 11
Figure 6 Interfaces et Canaux ... 15
Figure 7 *SystemC 2.0* : Architecture du Langage ... 17
Figure 8 Les différentes techniques de simulation ... 30
Figure 9 Structure d'un processus logique conservatif ... 42
Figure 10 Un processus *Time Warp* ... 47
Figure 11 Structure des messages échangés entre processus optimistes ... 48
Figure 12 État du processus après réception d'un message en retard ... 49
Figure 13 Aspects temporels des protocoles ... 54
Figure 14 Evaluation conservative dans *COMPOSITE ELSA* ... 56
Figure 15 Evaluation optimiste dans *COMPOSITE ELSA* ... 56
Figure 16 Organigramme de l'ordonnanceur *SystemC* ... 63
Figure 17 Diagramme de classes du modèle *DLX* ... 65
Figure 18 Architecture de l'anneau de *DLX* ... 66
Figure 19 Intervalle d'adressage ... 68
Figure 20 Le module `Bridge` ... 68
Figure 21 Le module `device_info` ... 69
Figure 22 Le module `RingDevice` ... 69
Figure 23 Le module `RingStuff` ... 70
Figure 24 Le module `RingManager` ... 71
Figure 25 Diagramme de classe du modèle multiprocesseur ... 72
Figure 26 *Simplex* : Exemple d'architecture non distribuée ... 73
Figure 27 Sortie du programme de simulation non distribuée ... 75
Figure 28 *Simplex* : Exemple de simulation distribuée ... 76
Figure 29 Exemple d'architecture distribuée avec canal serveur et canal client ... 77
Figure 30 Exemple d'architecture d'un canal client *TCP/IP* ... 77
Figure 31 Exemple d'architecture d'un canal serveur *TCP/IP* ... 78
Figure 32 Extension *TCP/IP* du canal primaire de communication ... 79
Figure 33 Exemple d'architecture d'un canal client/serveur *TCP/IP* ... 80
Figure 34 *Simplex* : Architecture distribuée avec canal *TCP/IP* client/serveur ... 80
Figure 35 *Simplex* : Exemple de sortie du programme « émetteur » ... 83
Figure 36 *Simplex* : Exemple de sortie du programme « récepteur » ... 83
Figure 37 `SystemC` 1.2 vs. 2.0 : Performance d'un *DLX* monoprocesseur ... 87
Figure 38 Courbes des temps de simulation du multiprocesseur non distribué ... 88
Figure 39 Chute de performance introduite par la version 2.0 ... 88
Figure 40 Différence des temps de simulation : `SC_THREAD` et `SC_METHOD` ... 90
Figure 41 Temps de simulation des *SC_THREAD* du modèle distribué ... 91
Figure 42 Accélération *superlinéaire* (kHz) obtenue par la distribution ... 92
Figure 43 Canal natif vs *MPI* ... 93

# LISTE DES ABRÉVIATIONS

| | |
|---|---|
| **API** | *Application Programming Interface* ou spécification qui définit comment le programmeur peut accéder aux méthodes et aux variables d'un ensemble de classes |
| **ASIC** | *Application Specific Integrated Circuit* ou circuit intégré à une application |
| **CPU** | *Control Processing Unit* ou microprocesseur |
| **DLX** | Processeur à architecture de chargement/rangement simple |
| **DSP** | *Digital Signal Processing unit* ou processeur numérique de traitement de signal |
| **FIFO** | *First In First Out* ou premier entré, premier sorti |
| **GHz** | *GigaHertz* ou un milliard de Hertz, Hertz étant l'unité de fréquence |
| **HDL** | *Hardware Description Language* ou langage de description de matériel |
| **HW** | *Hardware* ou matériel |
| **I/O** | *In/Out* ou entrée/sortie |
| **IP** | *Intellectual Property module* ou module de propriété intellectuelle |
| **LIFO** | *Last In First Out* ou dernier entré, premier sorti |
| **MIMD** | *Multiple Instructions stream Multiple Data stream*, plusieurs processeurs qui opèrent de façons indépendantes ou semi-indépendantes sur leurs données |
| **MOC** | *Model Of Computation* ou modèle de calcul ou modèle de traitement |
| **MPI** | *Message Passing Interface* ou librairie C portable permettant d'échanger des messages entre processus repartis sur un ensemble de processeurs |
| **O.O** | *Object Oriented* ou orienté objet |
| **PDES** | *Parallel Discrete Event Simulation* ou simulation parallèle à évènements discrets |
| **PL** | Processus Logique |
| **PP** | Processus Physique |
| **RAM** | *Random Access Memory* ou mémoire à accès aléatoire |
| **RISC** | *Reduced Instruction Set Computers* ou ordinateur à jeu d'instructions réduit |
| **ROM** | *Read Only Memory* ou mémoire à lecture seule |
| **RTL** | *Registre Transfert Level* ou description au niveau registre |
| **RTOS** | *Real Time Operating System* ou système d'exploitation temps réel |
| **SIMD** | *Single Instruction stream Multiple Data stream*, plusieurs processeurs qui exécutent en parallèle les mêmes instructions sur plusieurs données |
| **SISD** | *Single Instruction stream Single Data*, uniprocesseur |
| **SoC** | *System on Chip* ou « système sur puce » ou « système embarqué » |
| **SW** | *Software* ou logiciel |
| **TCP/IP** | *Transmission Control Protocol / Internet Protocol* ou protocoles réseau standard utilisés sur Internet |
| **TVG** | Temps Virtuel Global |

| | |
|---|---|
| **Verilog** | ***Verif**ying **Log**ic*, langage de description de matériel |
| **VHDL** | ***V**ery **H**igh Speed Integrated Circuit **D**escription **L**anguage* ou langage de description de circuit intégré de très haute vitesse |

# LEXIQUE

| | |
|---|---|
| **Accélération superlinéaire** | Dans le cas de la simulation parallèle : Le facteur multiplicatif de la vitesse de simulation (*Speedup*) dépasse le nombre de processeurs parallèles utilisés |
| **Accusé de réception** | **Ang** : *Acknowledge* |
| **Asynchrone** | Dans le domaine de la conception et de la synthèse, signifie sans horloge. Dans le domaine des communications, signifie communication non bloquante. |
| **Batteries de test** | voir *testbenches* |
| **Bit** | Abréviation de ***Binary digiT***, unité élémentaire d'information. Un bit est l'information de base d'un ordinateur basé sur le binaire. Un bit correspond à 0 ou 1 soit un circuit fermé ou un circuit ouvert. Un ordinateur traite les informations au niveau du bit qui sert ainsi d'unité |
| **Buffer** | Tampon de données |
| **Canal étendu** | Canal à méthodologie de communication spécifique; dans le cadre de ce document canal de communication utilisant le protocole *TCP/IP*, par opposition au canal élémentaire de communication de *SystemC* de type signal (`sc_signal`) |
| **Canal natif** | Canal de communication implémenté avec les fonctionnalités SystemC seules, par opposition à des fonctionnalités de communication étendue, implémentées avec des librairies logicielles externes à SystemC (*MPI*, par exemple) |
| **Connection Machine CM-1** | 1986 : La société *Thinking Machines* commercialise le premier super ordinateur massivement parallèle d'un nouveau type : la *Connection Machine CM-1* pouvant comporter jusqu'à 65536 processeurs ! La machine est un peu conçue comme le cerveau humain car chaque processeur effectue un travail très réduit mais ce qui compte, c'est la façon dont sont reliés les processeurs entre eux. La machine reconfigure les connexions internes entre les processeurs pour résoudre un problème donné. L'inconvénient de cette architecture est, bien sur, l'extrême complexité de la programmation et surtout de l'optimisation des programmes pour la vitesse. |
| **Couche** | Couche ou niveau d'abstraction<br>**Ang** : *Layer* |
| **Déboguer** | **Ang :** *Debugging*. Action de « déverminage » ou de localisation et correction des erreurs d'un programme à l'aide du débogueur |

| | |
|---|---|
| **Débogueur** | **Ang** : Debugger. Voir Déboguer |
| **Échéancier** | Dans le domaine de la *PDES* : liste chronologique des évènements du système |
| **Envoi/réception** | **Ang** : *Send/receive* |
| **File** | Structure de donnés de type Queue (type abstrait de données) |
| **Grappe** | Ensemble de processeurs *DLX* simulés sur le même nœud du processeur réseau et représentant un fragment de l'anneau de *DLX*.<br>**Ang**: *cluster* |
| **Hardware** | voir *software* |
| **Initiateur** | Voir Maître |
| **Machine Sequent** | Machine multi-processeur à mémoire partagée |
| **Maître** | Composant ayant un pouvoir d'exécution<br>**Ang** : Master, Initiator ou Server |
| **Multi-thread** | Capacité d'effectuer, en parallèle, plusieurs tâches légères, petits programmes ou routines |
| **Mutex** | Un *mutex* permet de gérer des exclusions mutuelles et permet de protéger des données, des zones d'exécution et de synchroniser des tâches. Il possède deux états, *unlocked* c'est à dire qu'il n'est pas attribué à une tâche donnée ou *locked* c'est à dire qu'il appartient à une tâche. Une tâche qui tente de verrouiller un *mutex* pris par une autre tâche est suspendue jusqu'à la libération du *mutex* par son propriétaire |
| **Paratran** | Mécanisme run-time pour exécuter du code Scheme (un dialecte de LISP) séquentiel standard sur un ordinateur multiprocesseur |
| **Requête** | Code indiquant une demande de transmission ou d'utilisation |
| **Sémaphore** | Méchanisme de synchronisation permettant de coordonner l'accès aux ressources partagées |
| **Snapshot algorithm** | Le but d'un *distributed snapshot algorithm* est de fournir une vue cohérente de l'état d'un réseau de noeuds et de canaux. L'état cohérent global est construit en combinant les instantanés locaux des divers processeurs |
| **Socket** | Norme de mode de communication sur réseau, mis au point à Berkeley, qui permet à une application de dialoguer en utilisant un protocole. Un socket est l'équivalent du lien qui permet à deux téléphones de communiquer. Il suit les même règles (composition du numéro, sonnerie, réponse, etc..). Les sockets furent conçus à l'origine pour le système d'exploitation Unix.Les sockets couramment utilisés communiquent en utilisant `TCP/IP` ou `UDP/IP` |
| **Software** | Logiciel par opposition à *hardware* matériel |

| | |
|---|---|
| **Speedup** | Dans le cas de la simulation parallèle, c'est le facteur multiplicatif de la vitesse de simulation obtenu par la distribution de la simulation |
| **Synchrone** | En conception et synthèse : avec une horloge<br>En communication : communication bloquante |
| **Testbenches** | Batteries de test ou bancs de test |
| **Thread** | Tâche légère (*fil d'exécution*): processus correspondant à l'exécution d'un petit programme ou d'une routine d'un programme plus important |

# DÉDICACE

À ma femme et à mes enfants, pour leur amour et leur patience.

À mon frère Kamel pour son soutien dans les moments difficiles.

# REMERCIEMENTS

Je tiens à remercier mon directeur de recherche El Mostapha Aboulhamid et Luc Charest pour m'avoir fourni une base de travail pour ce mémoire et pour leur aide.

# INTRODUCTION

Limites des méthodologies actuelles de conception système

Le niveau d'abstraction auquel le hardware est conçu, a augmenté significativement avec l'adoption répandue de Langages de Description de Matériel (*HDL*) comme format de spécification ou comme point d'entrée de conception, ce qui a mené à une énorme augmentation de la productivité par rapport à la précédente méthodologie de conception.

Ce saut quantitatif de la production fut possible parce que des *HDL* comme *VHDL* et *Verilog* ont permis aux designers de spécifier des fonctionnalités complexes au niveau comportemental (*Behavioral level*) et registre (*RTL level*) d'une façon relativement succincte comparée à la méthode précédente du type « vue structurelle-seule » (*structural-only view*).

Cependant, après une décennie de déploiement couronné de succès, il apparaît que la génération actuelle des *HDL* est insuffisamment équipée pour faire face à une conception hardware de complexité toujours croissante et à une conception de niveau système. Il n'est plus productif pour les designers de modéliser au niveau *bit* comme cela est imposé par les *HDL* ; des capacités d'abstraction de données plus sophistiquées sont nécessaires.

Ce problème de conception est à la base des récents changements de paradigme dans la conception système : la propriété intellectuelle (*IP*), la réutilisation, la conception basée plate-forme et le haut niveau d'abstraction des spécifications couvrant le matériel et le logiciel embarqué.

En plus, le hardware n'est plus conçu comme une entité indépendante. Les modules matériels coexistent fréquemment sur la même puce avec des noyaux de processeurs, du logiciel embarqué et d'autres blocs *IP* complexes, qui obligent les designers à exécuter des co-simulations lentes et inefficaces du matériel et des parties logicielles quand ils essayent de simuler le système en entier.

Un mécanisme mieux adapté pour le traitement du logiciel et des composants matériels dans le même environnement est absolument nécessaire.

Pourquoi *SystemC* ?

Plusieurs plates-formes de modélisation ont été proposées dans les années passées pour augmenter le niveau d'abstraction et permettre le co-design logiciel/matériel (*hardware-software co-design*). Une spécification à des niveaux d'abstraction plus hauts est possible dans des environnements comme *SpecC* [GZR+00]. Une approche unifiée et intégrée au co-design logiciel/matériel est possible si la description de modélisation du hardware est basée sur du *C/C++*, langages populaires dans la communauté du logiciel [VSB99]. *Hardware-C* [Gup95] est un exemple d'une telle approche.

*SystemC* [LTG97, GR00, SS01, FSSy20] est un standard émergent de plate-forme de modélisation basée sur *C++* qui répond aux problèmes cités plus haut et qui supporte une abstraction de conception aux niveaux *RTL*, comportemental et système. Le moteur de simulation de *SystemC* utilise un algorithme « piloté par évènement » (*event-driven*) comme un simulateur *RTL* classique. Une liste de sensibilité aux évènements est associée à chaque composant.

Description des limites du projet

Le travail présenté par ce mémoire a pour but, tout d'abord, de montrer qu'avec les nouvelles fonctionnalités introduites par *SystemC 2.0* permettant de profiter pleinement des puissantes capacités de la programmation orientée objet, il devient alors possible de croire à une future disponibilité d'un environnement *SystemC* offrant de la simulation distribuée. Pour arriver à cette conclusion, nous avons utilisé un modèle *SystemC* existant d'un multiprocesseur constitué par un anneau de processeurs *DLX*, modèle développé par [CAPP02] et n'offrant aucune fonctionnalité de simulation distribuée. Nous l'avons alors adapté pour pouvoir le distribuer sur un processeur réseau. L'anneau original était constitué d'un maximum de 128 *DLX* communiquant entre eux, localement, à l'aide de canaux de

communication élémentaires de type `sc_signal`. Pour distribuer les *DLX* sur le processeur réseau tout en leur permettant de continuer à communiquer, on a alors développé un nouveau canal *SystemC* de communication étendue utilisant le protocole *TCP/IP*. Nous avons aussi modifié le modèle pour passer d'une architecture en forme d'anneau de simples *DLX* locaux (l'anneau était dans sa totalité implémenté sur une seule machine) à une architecture en forme d'anneau de grappes (ou *cluster* en anglais, ensemble de processeurs *DLX* simulés sur un même nœud du processeur réseau) de *DLX* distribuées sur un processeur réseau. Nous avons pu alors simuler plusieurs configurations de répartition des processeurs *DLX* sur le processeur réseau et ainsi évaluer l'accroissement de performance introduit par la distribution de la simulation, en fonction de la complexité du modèle et ce, en augmentant à chaque fois la complexité de ce dernier par l'augmentation du nombre total de processeurs *DLX* ( de 2 à 128) dans l'anneau et par la variation du nombre de *DLX* par nœud de processeur réseau (de 1 à 64). Nous avons ensuite profité de la disponibilité de ce modèle distribué pour évaluer la différence de performance (toujours en fonction de la complexité du modèle) qui existe entre deux différentes méthodologies de modélisation du parallélisme disponibles dans *SystemC* (`SC_THREAD` et `SC_METHOD`).

Dans le cadre de notre expérimentation, plusieurs approches d'implémentation de la simulation distribuée étaient possibles. L'une d'elles est la distribution des éléments du modèle, quand il s'y prête, sur le processeur réseau. Une autre aurait été la conception et l'implémentation d'un moteur de simulation distribuée et donc la modification du code source de la librairie *SystemC*. Ayant en notre possession un modèle multiprocesseur conçu pour une simulation centralisée, se prêtant à une distribution « quasi-naturelle » de ses éléments (éléments très faiblement couplés, de charge équivalente, distribués sur des nœuds de traitement similaires), pratiquement sans aucun déséquilibre de charge impliquant des cycles inactifs nuisant aux performances et désirant exploiter son parallélisme, nous avons alors opté pour la première approche, plus précisément pour une simulation distribuée synchrone avec distribution des éléments du modèle. La synchronisation des différents éléments distribués s'est alors faite « naturellement » (sans implémentation

de protocoles de synchronisation) et ce, grâce à un échange simple de données et à une latence négligeable de la communication sur le réseau.

Enfin, pour montrer qu'une simulation distribuée utilisant des canaux natifs de communication distante est bien plus performante que l'implémentation de telles fonctionnalités étendues de communication à l'aide de librairies logicielles externes de parallélisation comme *MPI* [MPI], nous avons développé une version *MPI* de notre programme et avons simulé le même modèle dans les mêmes conditions.

Plan du mémoire

Le chapitre 1 fait un tour d'horizon des méthodologies de conception de systèmes pour arriver à *SystemC* et nous en présenter la version 1.0 avant de nous décrire, par l'exemple, les nouvelles fonctionnalités introduites par la version 2.0. La simulation distribuée est ensuite abordée dans ces détails par la description des ses différentes techniques et des algorithmes correspondants. Enfin, le moteur de simulation de *SystemC 2.0* est présenté. Le chapitre 2 commence par décrire le modèle multiprocesseur utilisé avant de proposer des exemples d'architecture de canaux *SystemC* de communication *TCP/IP*. Le chapitre 3 décrit l'expérimentation effectuée sur le modèle, en présente les résultats et les analyse. Une annexe proposant du pseudo-code du moteur de simulation de *SystemC 2.0* vient boucler le mémoire.

# CHAPITRE 1 : ÉTAT DE L'ART

La venue de l'ère des systèmes embarqués ou « systèmes sur puce » (*System On Chip* ou *SoC*) crée, à toutes les étapes du processus de conception, beaucoup de nouveaux défis. Au niveau du système, les ingénieurs sont entrain de reconsidérer comment les conceptions sont spécifiées, partitionnées et vérifiées. Aujourd'hui, avec des ingénieurs systèmes programmant en *C/C++* et leurs homologues du matériel travaillant avec des langages de description de matériel comme *VHDL* et *Verilog*, les problèmes résultant de l'utilisation de langages de conception différents et d'outils incompatibles deviennent courants.

On entend ici par systèmes embarqués (ou "System on Chip") des systèmes informatiques réalisés :

- pour une seule application prédéterminée,
- plutôt sur une seule puce,
- en grande série,
- avec des interfaces spécifiques de l'application (par exemple radio dans le cas des objets portables communicants),
- à base d'un ou plusieurs processeurs spécialisés, ressemblant aux processeurs d'usage général, mais pouvant présenter des aspects spécifiques à l'application,
- et comportant un logiciel sur mesure pour l'application visée, ce qui inclut, par exemple un système d'exploitation spécifique.

De tels systèmes se retrouvent dans les téléphones portables, les consoles de jeux de poches, les assistant personnels de poche, les contrôleurs de TV satellite, etc. La conception de ces systèmes informatiques suppose des méthodes particulières puisque le matériel et le logiciel sont développés simultanément. On ne peut pas

"essayer" les programmes mais seulement les simuler conjointement avec la simulation du matériel.

Initialement, la description et la simulation des systèmes se faisaient au niveau des portes logiques. Cette méthodologie se nomme *Saisie et Simulation* (*Capture and Simulate*). Ensuite, les langages de haut niveau, *VHDL* [Ashe96] et *Verilog* [Ver93], ainsi que leurs outils de synthèse ont vu le jour; ce qui a permis d'augmenter le niveau d'abstraction de la description des circuits et donc la conception de circuits plus complexes en moins de temps. L'industrie utilise actuellement une méthodologie de *Description et Synthèse* (*Describe and Synthesize*) (Figure 1). Cependant, les pressions du marché et l'intervalle toujours croissant entre la complexité disponible et la nécessité de réduire le temps de développement poussent l'industrie à modifier de nouveau sa méthodologie.

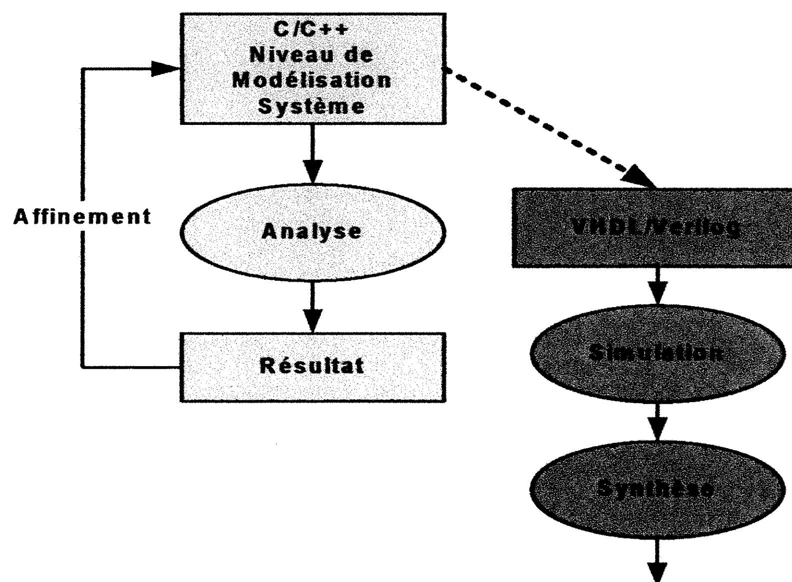


**Figure 1** Méthodologie de Design « Description et Synthèse »

Pour combler l'écart entre la productivité actuelle et la productivité désirée, l'industrie doit se tourner vers le développement de systèmes sur une seule puce ou *SoC* et la réutilisation de modules, aussi appelés « *core* » ou *IP* (module de propriété intellectuelle). La réutilisation est essentielle au leadership d'une entreprise à court

terme et à sa survie à moyen terme [CCH+99]. Le passage de la réutilisation de puces sur différentes cartes à la réutilisation d'*IP* sur différentes puces est conditionnel à l'établissement de normes, ainsi qu'à l'adoption d'outils et de méthodes au niveau de l'industrie. Les approches de la conception *SoC* sont discutées dans [GDG00].

Les problèmes introduits par la nouvelle technologie de design sont nombreux. L'augmentation de la complexité de conception exige de très rapides spécifications exécutables pour valider les concepts système et seul *C/C++* peut fournir à la fois les niveaux adéquats d'abstraction, l'intégration matériel/logiciel et la performance. Les ingénieurs de logiciel emploient *C/C++*. Les architectes de système emploient *C/C++*. La plupart des concepteurs de matériel sont familiers avec *C/C++*. Cela fait de *C/C++* le langage naturel pour la conception de systèmes embarqués. Avec la loi de Moore [Moore65] qui prédit un nombre toujours plus élevé de transistors, il est essentiel de commencer la conception à un niveau d'abstraction significativement plus haut que celui offert par les langages de description de matériel (*VHDL* ou *Verilog*). L'approche qui consiste à décrire toutes les fonctions du système en *C* permet aux architectes de plus facilement passer d'une implantation logicielle des fonctionnalités à leur réalisation par du matériel (Figure 2 : La conception peut se faire à des niveaux d'abstraction différents : système (*System level*), comportemental (*Behavioral level*) ou registre (*RTL level*). La même batterie de test peut être utilisée pour valider les modèles à ces différents niveaux d'abstraction).

*SystemC* est le langage standard de conception et de vérification écrit en *C++* qui permet de passer du concept à l'implémentation du matériel et du logiciel. *SystemC*, comme norme, permet et accélère l'échange des modèles de propriété intellectuelle de niveau système et des spécifications exécutables en employant une plate-forme de modélisation commune basée sur le langage *C*. Avec le langage *SystemC* et une mise en oeuvre à code source ouvert, les concepteurs peuvent créer, valider et partager des modèles et des spécifications exécutables avec d'autres sociétés employant la norme *ANSI C++*.

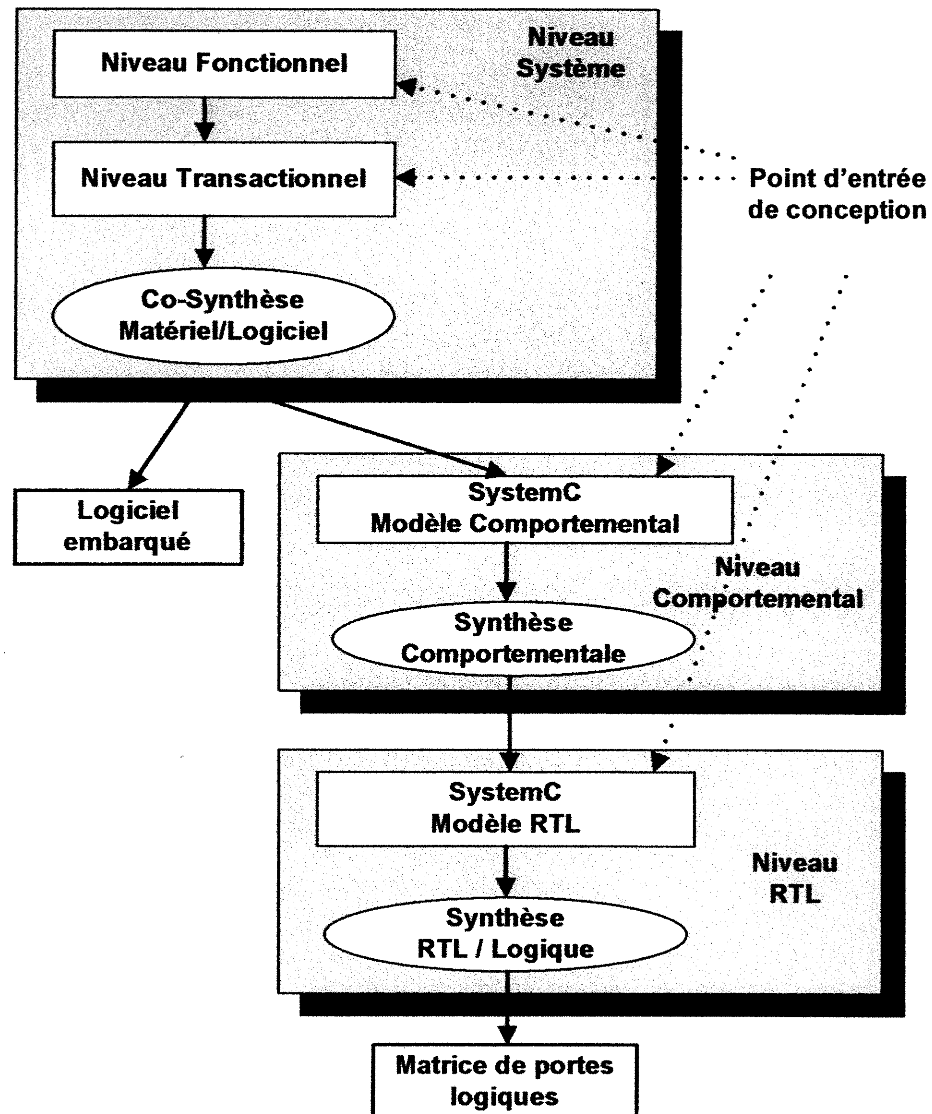


**Figure 2** Processus de Synthèse/ Implémentation

Parce que C/C++ ne possède pas la syntaxe nécessaire pour décrire le parallélisme et le cadencement (*concurrency and clocking*), plusieurs intervenants ont développé, de façons différentes, leurs propres extensions à *C/C++*. Cela a fragmenté le marché des outils de conception de niveau système et des modèles *IP* nécessaires au soutien d'un flux de conception du matériel basé sur *C*. Le but de l'*OSCI* (*Open SystemC Initiative*) [OSCI] est d'aligner tout le monde sur une plate-forme de modélisation simple pour éliminer le besoin de modèles multiples et permettre des flux de conception système vers silicium. Pour relever le défi d'une complexité de

conception croissante, les concepteurs de matériel se tournent vers *C/C++* pour modéliser des systèmes matériels à un plus haut niveau d'abstraction.

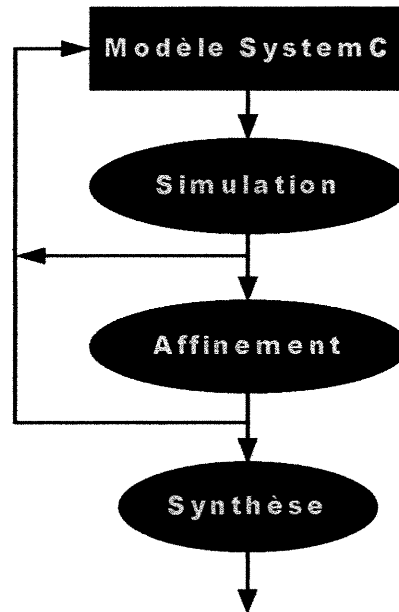


**Figure 3** Méthodologie de design avec *SystemC*

L'*OSCI* est une organisation indépendante à but non lucratif composée d'une large gamme de sociétés, d'universités et d'individus consacrés au soutien et à l'avancement de *SystemC* comme une norme industrielle ouverte (*industry open source standard*) pour la conception de niveau système. Par l'intermédiaire d'une organisation active comme l'*OSCI*, la communauté des utilisateurs peut cultiver et former cette plate-forme pour l'avantage de tous. *SystemC 2.0* est le résultat direct du travail réalisé par les membres techniques de cette organisation et qui a consisté à développer des extensions de modélisation de niveau système pour *SystemC*. *SystemC 2.0* est un sur-ensemble de *SystemC 1.0* : Toutes les conceptions *SystemC 1.0* sont toujours valables dans *SystemC 2.0* [FSSy20].

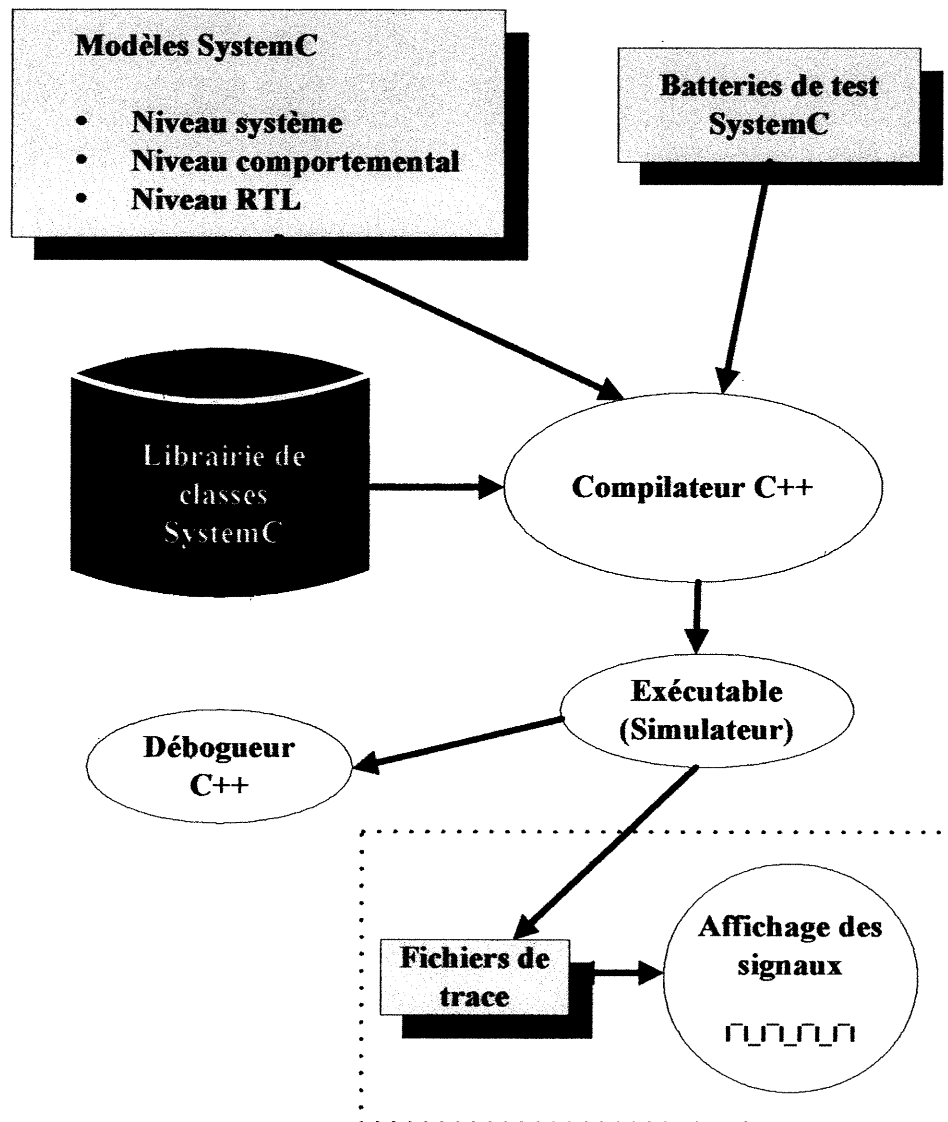


**Figure 4** La simulation dans `SystemC`

Ce chapitre passera d'abord en revue les composants primaires de modélisation de *SystemC 1.0*. Ensuite, les nouvelles possibilités de modélisation de niveau système de *SystemC 2.0* sont présentées. On montrera comment ces nouveaux composants de modélisation permettent aux utilisateurs de « proprement » modéliser (au sens de la programmation objet) la communication et la synchronisation dans les systèmes et permettent même aux utilisateurs de mettre en oeuvre de nouveaux modèles de calcul (*MOC)* dans *SystemC*. Les nouvelles possibilités de modélisation de la communication et de la synchronisation sont si suffisamment générales que tous

les mécanismes déjà existants dans *SystemC 1.0* pour la communication et la synchronisation peuvent maintenant être reconstruits autour de ces nouvelles fonctionnalités de *SystemC 2.0*. On parlera enfin de la simulation distribuée en présentant ses différentes techniques, ses algorithmes et ses limites.

## 1.1. SystemC 1.0

*SystemC 1.0* fournit un jeu de composants de modélisation semblables à ceux employés pour la modélisation *RTL* et fonctionnelle dans un *HDL* comme *Verilog* ou *VHDL*. Comme dans ces *HDL*, les utilisateurs peuvent construire des conceptions structurelles en utilisant des modules, des ports et des signaux (Figure 5). Les modules peuvent être instanciés, au sens OO, dans d'autres modules, permettant ainsi de construire des hiérarchies structurelles de conception. Les ports et les signaux permettent la communication de données entre les modules, et sont déclarés par l'utilisateur pour accepter un type spécifique de données. Les types de données généralement employés incluent les *bits*, les vecteurs de *bits*, les caractères, les entiers, les nombres à virgule flottante, les vecteurs d'entiers, etc. *SystemC 1.0* inclut aussi le support des signaux à quatre états de logique (c'est-à-dire des signaux qui modélisent `0`, `1`, `X` et `Z`).

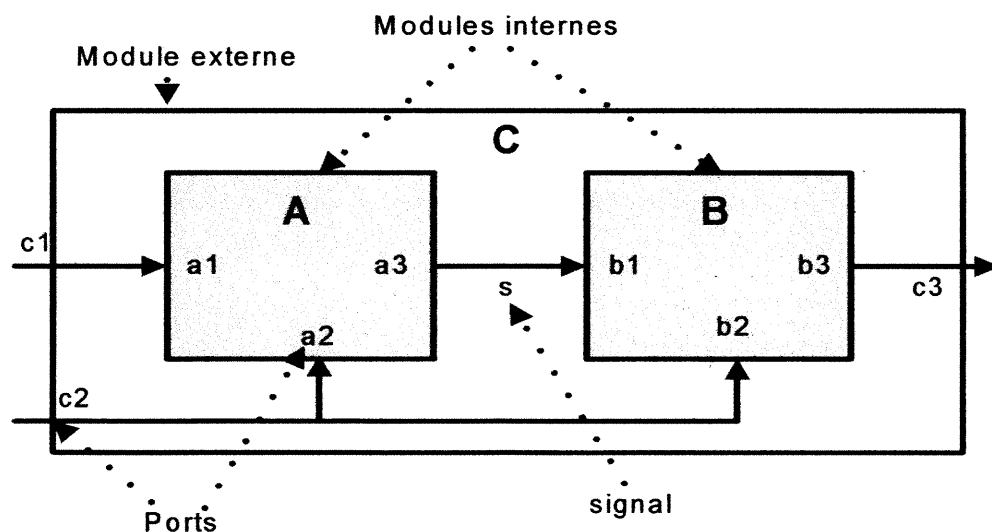


**Figure 5** Illustration des modules, ports, signaux, et de la hiérarchie

Un type de données important qu'on trouve dans *SystemC 1.0*, mais pas dans les autres *HDL* est le type numérique à précision fixe (*Fixed Precision Integers*). Les nombres à précision fixe sont fréquemment employés dans des applications *DSP* qui ciblent aussi bien des applications matérielles que logicielles puisque les opérations à virgule flottante consomment généralement trop de ressources. Un exemple d'opération à précision fixe pourrait être l'addition de deux nombres signés qui ont trois *bits* de précision entière et quatre *bits* de précision fractionnaire et l'affectation du résultat à un nombre à précision fixe similaire. Souvent les utilisateurs veulent spécifier les modes d'arrondi et de débordement (*overflow*) lors de l'emploi des nombres à précision fixe. C'est facile et naturel de modéliser les nombres à précision fixe dans *SystemC*, mais par contre c'est très difficile de le faire dans les autres *HDL*.

Dans *SystemC 1.0*, les comportements simultanés sont modélisés par des processus. Un processus peut être pensé comme un fil de contrôle indépendant (*thread*) qui reprend son exécution quand un certain évènement arrive ou quand un certain signal change de valeur et suspend ensuite son exécution après l'exécution de quelque action. Dans *SystemC 1.0*, il y a une capacité limitée pour spécifier la condition sous laquelle un processus reprend l'exécution : Le processus peut seulement être sensible aux changements de valeur de signaux particuliers et le jeu de signaux auxquels le processus est sensible doit être pré-spécifié avant le démarrage de la simulation.

Puisque les processus s'exécutent parallèlement et peuvent suspendre et reprendre leur exécution aux points indiqués par l'utilisateur, les instances de processus *SystemC* exigent généralement leur propre pile indépendante d'exécution (une situation équivalente dans le monde du logiciel surgit dans les applications multi-tâches où chaque *thread* exige sa propre pile d'exécution.) Certains processus *SystemC* qui suspendent leur exécution en des points limités n'exigent pas en réalité une pile indépendante d'exécution. Ces types de processus sont nommés *SC_METHOD*. L'optimisation de la conception en *SystemC* par l'emploi de SC_METHOD fournit de grandes améliorations de performance de la simulation quand le nombre de processus est grand.

Les signaux hardware ont plusieurs propriétés qui compliquent la tâche de leur modélisation par du logiciel. D'abord, les utilisateurs veulent souvent simuler des signaux hardware et des registres comme étant initialisé à `X` quand la simulation commence. C'est utile pour la détection de problèmes de réinitialisation (*reset*) dans les conceptions via des techniques de propagation `X` dans la simulation. Dans *SystemC 1.0*, on fournit cette particularité dans les types de données `sc_logic` et `sc_lv`.

Deuxièmement, les signaux hardware ont parfois de multiples « générateurs » (*drivers*). Dans ce cas, une fonction est nécessaire pour calculer une valeur résolue basée sur chacune des valeurs des différents générateurs. Cette fonction doit automatiquement être appelée quand une quelconque valeur générée change. Par exemple, quand on génère un signal avec `1` et `Z`, la valeur résolue doit être `1`, mais quand on le génère avec `1` et `0`, la valeur résolue doit être `X`. Dans *SystemC 1.0*, on fournit des signaux logiques résolus pour ce type de modélisation.

Troisièmement, les signaux hardware ne changent pas immédiatement leur valeur de sortie quand on leur affecte une nouvelle valeur, que ce soit en simulation ou dans le monde réel. Il y a toujours quelque retard (peut-être très petit) avant que la nouvelle valeur affectée à un signal ne soit disponible à d'autres processus de la conception. Ce retard est crucial à la modélisation appropriée du matériel, puisqu'il permet, par exemple, à deux registres d'échanger leurs valeurs sur un top d'horloge. Par comparaison, deux variables logicielles ne peuvent pas échanger leurs valeurs sans l'introduction d'une troisième variable provisoire.

Comme *VHDL* et *Verilog*, *SystemC 1.0* supporte le concept de « *signal à affectation différée* » (la nouvelle valeur du signal ne sera disponible qu'au début du prochain cycle delta) en employant des *cycles delta* pour correctement modéliser les signaux hardware. Un *cycle delta* peut être pensé comme un très petit pas de temps dans la simulation qui n'augmente pas le temps visible par l'utilisateur. Des *cycles delta* multiples peuvent avoir lieu à un moment donné. Quand on affecte une nouvelle valeur à un signal, les autres processus ne la « voient » pas et ce, jusqu'au prochain

*cycle delta.* Les processus qui sont sensibles à ce signal reprennent alors l'exécution si la valeur du signal a changé par rapport à sa valeur précédente.

## 1.2. SystemC 2.0

Un des buts primordiaux de la version *SystemC 2.0* est de permettre la modélisation au niveau système de systèmes pouvant être implantés par du logiciel, du matériel ou par une quelque combinaison des deux. Un des défis dans la fourniture d'un langage de conception de niveau système est qu'il existe un grand choix de modèles de conception, de niveaux d'abstraction de conception et de méthodologies de conception, employés dans la conception système. Pour adresser ce défi dans *SystemC 2.0,* une petite mais très générale couche de fonctionnalités de modélisation de base a été ajoutée au langage. Au-dessus de cette base du langage on peut alors ajouter d'autres modèles spécifiques de calcul, de bibliothèques de conception, de directives de modélisation et de méthodologies de conception, exigés pour la conception des systèmes.

La petite et générale couche de fonctionnalités de modélisation de base dans *SystemC 2.0* est nommée *le noyau du langage* (*core language*) et est le composant central de la norme *SystemC 2.0.*

Les autres composants de la norme *SystemC 2.0* sont les modèles élémentaires de bibliothèque, largement utilisés, qui sont construits au-dessus du noyau du langage (par exemple les réveils (*timers*), les *FIFO*, les signaux, etc.) et qui sont appelés *canaux élémentaires.* Il est reconnu que beaucoup de modèles différents de calcul et de méthodologies de conception, peuvent être employés en conjonction avec *SystemC.* Pour cette raison, les bibliothèques de conception et les modèles nécessaires au support de ces méthodologies spécifiques de conception sont considérés comme étant séparés de la norme du noyau du langage.

Beaucoup des fonctionnalités du langage, déjà présentes dans *SystemC 1.0,* sont aussi très utiles pour la modélisation de niveau système. Les fonctionnalités de description structurelle disponibles dans *SystemC 1.0* (les *modules* et les *ports*) sont

aussi utiles pour la conception système, comme le sont le vaste jeu de types de données et la capacité d'exprimer la concurrence (ou le parallélisme) en employant les processus. Cependant, dans *SystemC 1.0*, le mécanisme élémentaire de communication et de synchronisation (*signal*) n'est pas suffisamment général pour la modélisation de niveau système. Par exemple, dans une conception de niveau système, un concepteur pourrait vouloir spécifier que plusieurs modules communiquent en utilisant des files d'attente ou que plusieurs processus s'exécutent concurremment et gèrent l'accès à des données globales partagées en utilisant des *mutex*.

*SystemC 2.0* présente un nouveau jeu de fonctionnalités pour la modélisation généralisée de la communication et de la synchronisation. Ceux-ci sont les *canaux*, les *interfaces* et les *évènements*. Les Ports des modules sont connectés aux canaux à travers les interfaces (Figure 6).

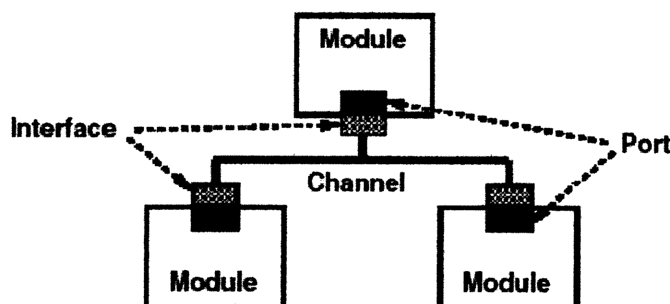


**Figure 6** Interfaces et Canaux

Un canal est un objet qui sert comme un conteneur pour la communication et la synchronisation. Les canaux mettent en oeuvre une ou plusieurs interfaces. Une interface spécifie un jeu de méthodes d'accès implémenté dans un canal, mais l'interface elle-même ne fournit pas l'implémentation de ces méthodes. Un évènement est une synchronisation primitive flexible et à bas niveau qui est employé pour construire d'autres formes de synchronisation. Les canaux, les interfaces et les évènements permettent aux concepteurs de modéliser le grand choix de type de communication et de synchronisation, rencontré dans la conception de systèmes. Les exemples incluent les signaux *HW*, les files d'attente (*FIFO*, *LIFO*, les files d'attente

de message, etc.), les sémaphores, les mémoires et les bus (au niveau *RTL* et au niveau transactionnel).

La Figure 7 ci-après récapitule l'architecture du langage *SystemC 2.0*. Il y a plusieurs concepts importants à comprendre de ce diagramme :

- Tout *SystemC* est fondé sur *C++*.
- Les couches supérieures dans le diagramme sont « proprement » construites (au sens OO) par-dessus les couches inférieures.
- Le noyau du langage *SystemC* fournit seulement un jeu minimal de composants de modélisation pour la description structurelle, la concurrence, la communication et la synchronisation.
- Les types de données sont séparés du noyau du langage et les types de données définis par l'utilisateur sont entièrement supportés.
- Les mécanismes de communication généralement employés comme les signaux et les *FIFO* peuvent être construits par-dessus le noyau du langage. Les modèles de calcul généralement employés peuvent aussi être construits sur le sommet du noyau du langage.
- Si désiré, les couches inférieures dans le diagramme peuvent être utilisées sans avoir besoin des couches supérieures.

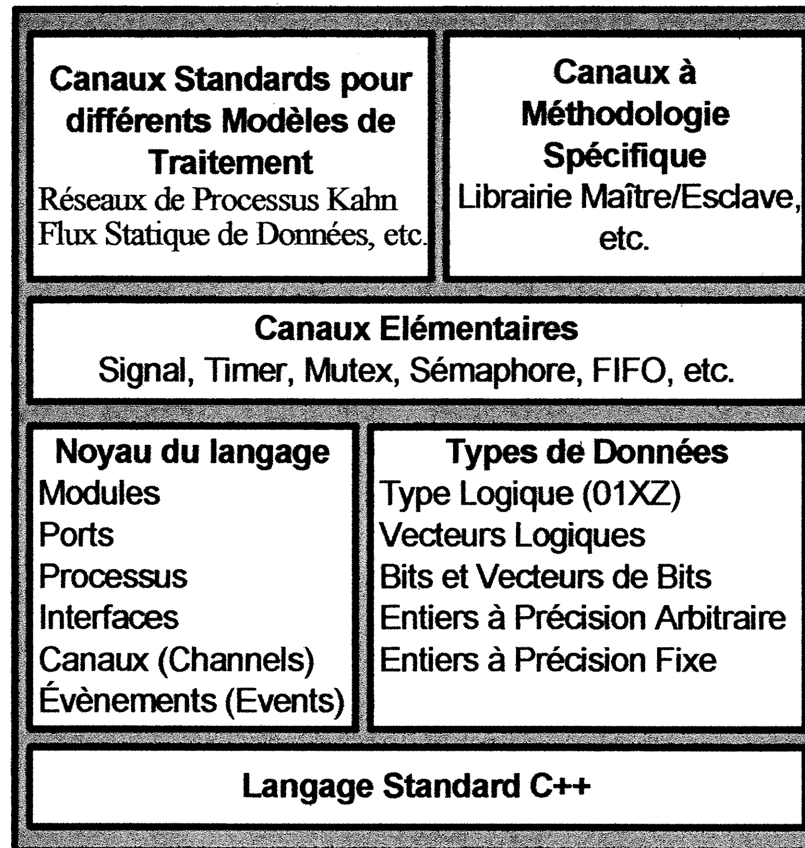


**Figure 7** *SystemC 2.0* : Architecture du Langage

Considérons un exemple simple de modélisation d'une communication: un *FIFO* qui stocke dix (10) caractères. Le *FIFO* a des interfaces de lecture et écriture bloquantes de sorte que les caractères soient toujours sûrement « livrés ». (Une version exécutable de cet exemple est incluse dans le répertoire "examples/simple_fifo" dans la distribution *SystemC 2.0* [OSCI]). D'abord, nous spécifions les interfaces séparées de lecture et d'écriture du *FIFO*. Pour illustrer cela, disons que l'interface d'écriture permet la réinitialisation (`reset`) du *FIFO*, alors que l'interface de lecture permet une requête non bloquante de demande de lecture du nombre de caractères actuellement présents dans le FIFO.

```
class write_if : virtual public sc_interface {
      public:
      virtual void write(char) = 0;
      virtual void reset() = 0;
};
```

```
class read_if : virtual public sc_interface {
      public:
      virtual void read(char &) = 0;
      virtual int num_available() = 0;
};
```

Comme indiqué ci-dessus, une interface est une classe *C++* abstraite qui hérite de `sc_interface`. Les interfaces spécifient un jeu de méthodes d'accès pour un canal, mais ne fournissent aucune implémentation de ces méthodes.

Ensuite, spécifions un modèle de haut niveau pour le *FIFO*. Le *FIFO* utilise l'héritage multiple de *C++* pour hériter les deux interfaces de lecture et d 'écriture et aussi les propriétés des canaux de `sc_channel`. (On déconseille parfois l'utilisation de l'héritage multiple dans *C++*, mais la forme employée dans *SystemC* pour les canaux et les interfaces est "sûre" puisque les interfaces ne mettent jamais en œuvre de méthodes ou ne contiennent de membres de données).

```
class fifo : public sc_channel, public write_if, public read_if {
      public:
      SC_CTOR(fifo) { // constructeur du canal
            num_elements = first = 0;
      }
      void write(char c) {
            if (num_elements == max) wait(read_event);
            data[(first + num_elements) % max ] = c;
            ++ num_elements;
            write_event.notify();
      }
      void read(char& c) {
            if (num_elements == 0) wait(write_event);
            c = data[first];
            -- num_elements;
            first = (first + 1) % max;
            read_event.notify();
      }
      void reset() {
            num_elements = first = 0;
      }
      int num_available() {
            return num_elements;
      }
      private:
      enum e { max = 10 };//max: constante visible dans la classe
      char data[max];
      int num_elements, first;
      sc_event write_event, read_event;
};
```

Le canal fournit une implémentation pour les méthodes d'accès indiquées dans les interfaces de lecture et d'écriture. D'abord, il a un constructeur qui met à zéro le nombre de caractères disponibles dans le *FIFO*. Ensuite, il implémente la méthode `write` qui attend, si nécessaire, jusqu'à ce qu'il y ait un espace disponible dans le *FIFO*, et ajoute ensuite le nouveau caractère à la place appropriée dans le tableau de données. La fonctionnalité *FIFO* est implémentée en employant un *buffer* circulaire dans le tableau de données. Finalement la méthode `write` notifie l'évènement `write_event`, qui permettra alors la reprise d'exécution de n'importe quelle demande de lecture en attente d'une nouvelle entrée dans le *FIFO*.

L'implémentation de la méthode `read` est semblable. Elle attend, si nécessaire, jusqu'à ce qu'il y ait des données dans le *FIFO*, lit ensuite le premier article et le supprime du tableau de données par incrémentation de l'index `first`. Finalement, elle notifie l'évènement `read_event`, qui permettra alors la reprise d'exécution de n'importe quelle requête d'écriture qui attend qu'un espace de stockage soit disponible.

Rappelons que les processus dans *SystemC* ont leurs propres *threads* d'exécution. Quand un processus « producteur » invoque la méthode `write` du *FIFO* ci-dessus, le processus producteur peut devoir faire suspendre son exécution jusqu'à ce qu'il y ait un espace disponible dans le *FIFO*. L'appel `wait()` dans la méthode `write` réalise cette suspension du processus producteur. Il est important de noter que cette suspension est complètement gérée par le canal et n'est pas visible au processus producteur. Si le processus producteur est suspendu, il sera relancé la prochaine fois que le processus « consommateur » lit un caractère du *FIFO*, ce qui aboutira à une notification de l'évènement `read_event`, signalant que cet espace est maintenant disponible dans le *FIFO*.

*SystemC 1.0* fournissait aussi un appel `wait()`, mais cet appel ne pouvait pas prendre d'arguments parce qu'un processus était toujours sensible à un jeu fixe de signaux. Dans l'exemple ci-dessus, nous voyons une extension significative de *SystemC 1.0* : l'appel `wait()` peut maintenant avoir des arguments spécifiques et les

processus peuvent être sensibles aux évènements et aux signaux. La capacité de fournir des arguments à l'appel `wait()` est nommée *la sensibilité dynamique*. Ce terme est employé parce que quand les processus s'exécutent, ils peuvent dynamiquement choisir le jeu des évènements qui causeront leur reprise d'exécution par l'emploi de l'appel `wait()`. Non seulement les processus peuvent attendre sur des évènements spécifiques dans *SystemC 2.0*, mais ils peuvent aussi attendre un temps spécifique. Par exemple, un processus peut appeler `wait(200, SC_NS)`, qui causerait que le processus suspende son exécution pour exactement 200 nanosecondes ou bien il pourrait appeler `wait(200, SC_NS, e)` ce qui causerait alors l'attente de la notification de l'évènement `e` ou de l'écoulement d'un temps mort de 200 nanosecondes.

Les évènements sont les primitives fondamentales de la synchronisation dans *SystemC 2.0* et ils ont plusieurs différences importantes avec les signaux (`sc_signal`), qui étaient les seuls mécanismes de synchronisation disponibles dans *SystemC 1.0*. À la différence des signaux, un évènement n'a pas de type et ne transmet pas de valeur (les évènements transfèrent seulement le contrôle d'un *thread* à un autre). Aussi, une notification d'évènement causera toujours la relance des processus sensibles, alors que l'affectation d'une valeur à un signal ne cause la relance des processus sensibles que si la nouvelle valeur du signal diffère de la valeur précédente. Finalement, les notifications d'évènement peuvent être spécifiées pour survenir immédiatement au bout d'un *cycle delta* ou au bout d'un temps déterminé. Les affectations de signal deviennent disponibles toujours au bout d'exactement un *cycle delta* dans l'avenir.

Pour compléter l'exemple de modélisation d'une communication, montrons maintenant comment le *FIFO* peut être employé comme canal de communication entre un module producteur et un module de consommateur :

```
// le module producteur
class producer : public sc_module {
      public:
      sc_port<write_if> out; //port de sortie du module producteur
      SC_CTOR(producer) {// le constructeur du module
            SC_THREAD(main); // démarrage du processus producteur
      }


      void main ( ){ // le processus producteur
            char c;
            while (true) {
                  ...
                  out->write(c); // écrit c dans le fifo
                  if (...) out->reset(); // réinitialise le fifo
            }
      }
};

// le module consommateur
class consumer : public sc_module {
      public:
      sc_port<read_if> in; // port d'entrée du consommateur
      SC_CTOR(consumer)  {// le constructeur du module
            SC_THREAD(main); // démarrage du processus consommateur
      }
      void main() {// le processus consommateur
            char c;
            while (true) {
                  in->read(c); // lire c à partir du fifo
                  // pour accélérer le processus…
                  if (in->num_available() > 5) ...;
            }
      }
};

// le module principal
class top : public sc_module {
      public:
      fifo *fifo_inst;          // une instance du module fifo
      producer *prod_inst;      // une instance du module producteur
      consumer *cons_inst;      // une instance du module consommateur
      SC_CTOR ( top ) {         // le constructeur du module
            fifo_inst = new fifo ("Fifo1");
            prod_inst = new producer("Producer1");
            // connecter le fifo au port de sortie du producteur
            prod_inst->out(fifo_inst);
            cons_inst = new consumer("Consumer1");
            // connecter le fifo au port d'entrée du consommateur
            cons_inst->in(fifo_inst);
      }
};
```

Il est important de noter que le port de sortie du producteur possède un argument de type générique qui est spécifié, dans ce cas-ci, comme étant l'interface d'écriture `write_if` du *FIFO*. De la même façon le port d'entrée du consommateur a un argument de type générique qui est l'interface de lecture `read_if` du *FIFO*. À cause de cela, seules les méthodes déclarées par l'interface d'écriture sont disponibles pour une utilisation dans le producteur, tandis que seules les méthodes déclarées par l'interface de lecture sont disponibles pour le consommateur. Ainsi, le consommateur ne peut pas appeler la méthode `fifo.reset()` comme le fait le producteur, par exemple.

Dans cet exemple, l'instance *FIFO* est reliée au port de sortie du producteur et au port d'entrée du consommateur. Cependant, le port de sortie du producteur n'est pas « conscient » qu'il est relié à un *FIFO*. Il sait seulement qu'actuellement il est relié à un objet qui implémente l'interface d'écriture `write_if`. De la même façon l'entrée du consommateur n'est pas « consciente » qu'elle est reliée à un *FIFO*. Elle sait seulement qu'actuellement elle est reliée à un objet qui implémente l'interface de lecture `read_if`. Cette dissimulation de l'implémentation du canal est intentionnelle et est un ingrédient clef de *SystemC 2.0* dans la fourniture d'une modélisation flexible et d'un affinement de la communication.

Voici plusieurs exemples qui montrent comment la dissimulation de l'implémentation du canal mentionnée ci-dessus facilite la modélisation et l'affinement de la communication:

- Imaginons que le concepteur voudrait expérimenter une spécification fonctionnelle légèrement différente qui utilise un *FIFO* qui détruit les caractères quand il est plein, plutôt que de suspendre l'exécution du processus producteur. Tout ce que le concepteur a à faire est de réécrire l'implémentation du canal pour ce nouveau *FIFO* qui réutilise les interfaces existantes de lecture et d'écriture. Alors le seul changement qu'aura subie la conception originale est la substitution du *FIFO* original par le nouveau *FIFO* dans le module supérieur.

- Imaginons que le concepteur voudrait affiner cette conception en une implémentation s'exécutant sur un *RTOS*. Les processus producteur et consommateur seraient implémentés par des *threads* s'exécutant sur le *RTOS*, tandis que le *FIFO* original serait remplacé par un nouveau dans lequel les méthodes de lecture et d'écriture ont directement accès aux services de queue incorporés du *RTOS*. De nouveau, le seul changement nécessaire dans la conception originale est la substitution du nouveau *FIFO* à l'original dans le module supérieur.

- Imaginons que le concepteur voudrait raffiner cette conception en une implémentation personnalisée (propriétaire) qui emploie un *FIFO* hardware. De nouveau, un nouveau canal *FIFO* est écrit pour remplacer l'existant dans le module supérieur. Les canaux dans *SystemC* peuvent contenir d'autres canaux et des modules, de même que les modules peuvent contenir des modules enfants multiples. Dans ce cas nous écrivons un nouveau canal *FIFO* qui instancie le *FIFO* hardware dans son propre module et qui contient les signaux hardware nécessaires pour l'interfaçage avec lui. Nous implémentons alors, dans ce canal, les méthodes des interfaces de lecture et d'écriture pour contrôler les signaux hardware avec le protocole exigé et donc pour faire en sorte que les données soient correctement écrites et lues, respectivement, dans et à partir du *FIFO* hardware. Cela nous permet alors de simuler la conception en employant un modèle réel du *FIFO* hardware. Comme étape finale du processus d'affinement de la communication, le code qui exécute les protocoles de lecture et d'écriture pour accéder au *FIFO* hardware dans le canal peut être écrit sous forme de macros (*inlining*) dans le module producteur et le module consommateur, respectivement, lui permettant ainsi d'être synthétisé et optimisé avec le code source de ces derniers (le langage *SystemC* n'automatise pas cette étape d'*inlining*, mais il fournit les composants nécessaires pour que d'autres outils puissent la réaliser).

Une note finale sur cet exemple : Pour raison de simplicité, le canal *FIFO* qui a été présenté est seulement capable de stocker des caractères. En pratique, les canaux comme celui-ci seraient écrits en employant des types de données génériques *C++* pour permettre au type de données « véhiculé » d'être spécifié seulement au moment de la déclaration du canal. En employant cette technique, un canal simple *FIFO* pourrait stocker n'importe quel type de données *C++*, incluant les types de données définis par l'utilisateur. *SystemC 2.0* supporte entièrement cette technique de conception basée sur les types génériques.

Il y a eu beaucoup de discussion sur les *MOC* ces dernières années. C'est peut-être parce qu'il y a beaucoup de modèles différents de calcul et qu'il n'est pas toujours évident lequel est le meilleur candidat pour une tâche de conception système particulière. Dans le sens le plus large, un *MOC* est défini par les propriétés suivantes:

- Le modèle de temps employé (à valeur réelle, à valeur entière, non prévu) et les contraintes d'ordonnancement des évènements dans le système (globalement ordonné, partiellement ordonné, etc.).

- La méthode de communication utilisée entre processus simultanés.

- Les règles d'activation de processus.

Dans *SystemC 2.0* les capacités simples et flexibles de synchronisation fournies par les évènements et l'appel `wait()` permettent à une large gamme de types de canal différents d'être implémentés sans devoir changer le moteur de simulation sous-jacent. Toute la fonctionnalité exigée est déjà présente dans le noyau de simulation. Ainsi, *SystemC 2.0* supporte un modèle générique de traitement très puissant. Alors que le modèle global du temps est fixé à un modèle à valeur entière, les concepteurs peuvent construire des canaux spécifiques pour implémenter de façon précise leurs règles de communication interprocessus ainsi que leurs règles d'activation de processus et pour « puiser » dans un choix large d'ordonnancement des évènements du système.

Bien que des modèles de temps continu comme ceux employés par exemple dans la modélisation analogique ne puissent pas encore être construits dans *SystemC*, pratiquement n'importe quel système de temps discret peut être modélisé dans *SystemC*. Quelques modèles bien connus de traitement qui peuvent être tout à fait naturellement modélisés dans *SystemC 2.0*, incluent :

- Les réseaux de processus *Kahn* (*Kahn Process Networks*),
- Le flux statique multi-taux de données (*Static Multi-rate Dataflow*),
- Le flux dynamique multi-taux de données (*Dynamic Multi-rate Dataflow*),
- Les processus séquentiels de communication,
- L'évènement discret comme employé dans la :
    - Modélisation hardware *RTL*,
    - Modélisation de Réseau (par exemple modèles stochastiques ou modèles de type "waiting room"),
    - Modélisation de plate-forme *SoC* Basée sur transaction.

Un exemple de comment il est possible de réaliser cet empilement de modèles spécifiques de calcul par-dessus le noyau de *SystemC 2.0* est le signal hardware (`sc_signal`). Dans *SystemC 1.0*, le signal hardware était le seul mécanisme disponible pour la communication et la synchronisation entre processus. Dans *SystemC 2.0*, le signal hardware est maintenant complètement implémenté par-dessus les canaux, les interfaces et les évènements. Le noyau de simulation *SystemC 2.0* n'a aucun support spécifique pour les signaux hardware et n'est pas « conscient » si un signal hardware est employé dans une conception particulière. Notez que cette approche par couches a été introduite dans *SystemC* d'une façon discrète pour permettre aux conceptions réalisées avec *SystemC 1.0* de continuer à fonctionner dans *SystemC 2.0* sans aucune modification.

Parce que la norme *SystemC* est contrôlée par quelques sociétés et que son développement dépend de contributions ouvertes, son évolution future peut être difficile à prévoir. Néanmoins, un consensus existe concernant quelques extensions probables à *SystemC*. Ces extensions peuvent être séparées dans deux catégories.

Le premier jeu d'extensions inclut de nouvelles particularités dans le noyau du langage. Celles-ci vont probablement inclure :

- La capacité de création de *threads* fils (*fork*), le rendez-vous entre *threads* (*join*) ainsi que la création dynamique de *threads*.
- La capacité d'interrompre (*interrupt*) ou d'annuler (*abort*) un *thread* et ses enfants.
- La spécification et la vérification des contraintes de temps.
- Le support pour la modélisation abstraite de *RTOS* et la modélisation de l'ordonnanceur (*scheduler*).
- Le support possible pour la modélisation de circuits analogiques et à signaux mixtes (*ASIC*s).

Le deuxième jeu d'extensions inclura les particularités qui peuvent être développées comme des bibliothèques par-dessus le noyau du langage.

Celles-ci peuvent inclure :

- Des canaux standardisés pour des modèles de calcul divers comme le flux statique de données et les réseaux de processus Kahn.
- Des bibliothèques pour faciliter le développement de testbenches, comme des structures de données qui facilitent la génération de stimulus et la vérification de réponse, des fonctions qui facilitent la production de stimulus aléatoires, etc.

- Des directives de modélisation au niveau système et des bibliothèques de code qui aide les utilisateurs à créer leurs propres modèles d'après ces directives.

- Des API standards pour interfacer *SystemC* avec d'autres simulateurs, émulateurs, etc.

Les nouvelles fonctionnalités de modélisation de *SystemC 2.0* permettent aux utilisateurs de *SystemC* de modéliser un grand choix de méthodes de communication et de synchronisation dans la conception de système et leur permet même d'implémenter de nouveaux modèles de calcul. De plus, ces fonctionnalités de modélisation rendent facile l'exploration de différents procédés de communication dans la conception de systèmes et la réalisation d'un affinement de la communication pour une implémentation matérielle ou logicielle. *SystemC 2.0* permet aux tâches suivantes d'être réalisées par l'utilisation d'un seul langage de modélisation:

- Des spécifications de système complexes peuvent être développées et simulées.

- Des spécifications de système peuvent être affinées pour une implantation mixte matériel/logiciel.

- Des implémentations de matériel peuvent être modélisées entièrement au niveau *RTL*.

- Des types de données complexes peuvent être facilement modélisés et un type numérique flexible à précision fixe est supporté.

- La vaste connaissance, l'infrastructure et la base de code construite autour de *C* et *C++* peut être démultipliée.

La capacité de soutenir un tel grand choix de tâches de conception système par l'utilisation d'un seul langage de modélisation est unique à *SystemC*.

## 1.3. La simulation distribuée

### 1.3.1 La simulation

La simulation parallèle de langages de description de matériel n'est plus seulement un sujet d'intérêt académique; c'est maintenant une opportunité intéressante pour les designers des systèmes et du matériel. L'équipement requis et la densité de fabrication des systèmes excèdent souvent la capacité de simulation monoprocesseur, tandis que les multiprocesseurs à mémoire partagée, les réseaux ultrarapides et les processeurs parallèles (des plates-formes) sont largement accessibles. Les simulateurs parallèles sont une réalité technique et commerciale, grâce en partie à la nature intrinsèquement parallèle des *HDL*. L'efficacité de simulation réalisée en employant de tels simulateurs dépend largement du style de modélisation utilisé pour décrire le modèle source *HDL*.

Dans l'automatisation de la conception électronique, le besoin de vitesse et de capacité de simulation excède presque toujours ce qui est disponible. Pour augmenter rentablement la performance de la simulation *HDL*, des techniques de simulation parallèles peuvent être adoptées. De nombreux groupes ont développé ou développent des simulateurs parallèles *HDL*. L'exploitation de techniques de traitement parallèles pour améliorer la performance est en particulier prometteuse parce que les *HDL* supportent explicitement et de manière cohérente la notion de processus simultanés.

La recherche de simulation parallèle a commencé il y a plus de 20 ans [BRE77, ChMis79] comme moyen d'amélioration du temps d'exécution de la simulation. En raison de l'importance de ce domaine, un intérêt significatif s'est concentré sur lui, avec de nombreux algorithmes et des implémentations développées pour réaliser des simulations plus rapides. L'automatisation de la conception électronique représente un des secteurs primordiaux de concentration de la recherche en raison de la demande énorme de cycles de simulation pour assurer des implémentations correctes. Parce que la simulation parallèle continue à être un secteur si actif pour la recherche, il existe une richesse au niveau de la disponibilité du matériel bibliographique qui y est lié. Pour une vue d'ensemble du travail réalisé

sur la simulation logique parallèle, une excellente revue par *Bailey* et d'autres [BMB94] est disponible. Pour plus de traitement général sur la simulation parallèle, *Fujimoto* et *Misra* donne de bonnes vues d'ensemble des problèmes et des techniques [Fuj90, Misra86]. Les avancées générales de technologie de simulation parallèle ont depuis lors été principalement les affinements des techniques associées de modélisation et pour de l'information plus récente sur la technologie générale de simulation parallèle, il faut voir [PWPDS].

L'intérêt économique de la simulation est actuellement manifeste. Des secteurs d'application de plus en plus nombreux font appel à la simulation: évaluation de phénomènes naturels (prévisions météo et climatiques, évolution démographique, etc.), mécanique des fluides (aérodynamique, hydrodynamique), prévisions et planifications d'opérations diverses (modèles économiques, files d'attentes), etc. La demande importante pour disposer rapidement de résultats pour différents modèles d'un modèle physique à étudier a ouvert un axe de recherche fortement orienté vers le domaine particulier que constitue la simulation distribuée (ou parallèle).

L'ensemble de ce qui suit dans cette section est consacré aux différentes techniques logicielles utilisées pour l'accélération des applications de simulation. Avant de présenter les orientations de ces recherches, nous introduisons brièvement la simulation à évènements discrets. Les deux sections suivantes de ce chapitre sont consacrées aux grands types d'approches pour la simulation distribuée : l'évolution synchrone de processus est présentée dans la section 1.3.4, et les différentes approches asynchrones sont ensuite détaillées. Les méthodes conservatives sont l'objet de la section 1.3.5.1 et la stratégie spéculative celui de la section 1.3.5.2. Les approches hybrides combinant les deux options précédentes sont introduites dans la section 1.3.5.3. La section suivante est consacrée aux performances obtenues avec différents simulateurs distribués.

La simulation permet d'étudier le comportement dynamique d'un système réel à travers un modèle plus ou moins complexe sur lequel on peut effectuer des expériences. Elle permet de reproduire à plus ou moins grande vitesse, un grand nombre de fois et à moindre coût, les situations que l'on veut étudier. Le temps est

simulé, ce qui permet soit de le dilater, soit de l'accélérer pour faciliter l'observation. Deux grandes classes de modèles sont utilisées (Figure 8):

- Le modèle discret qui fournit une représentation sous la forme d'une séquence de transitions d'états. Chaque changement d'état correspond à un évènement qui est daté.

- Le modèle continu qui fournit une représentation sous la forme d'un système d'équations différentielles traduisant l'évolution continue des variables représentatives.

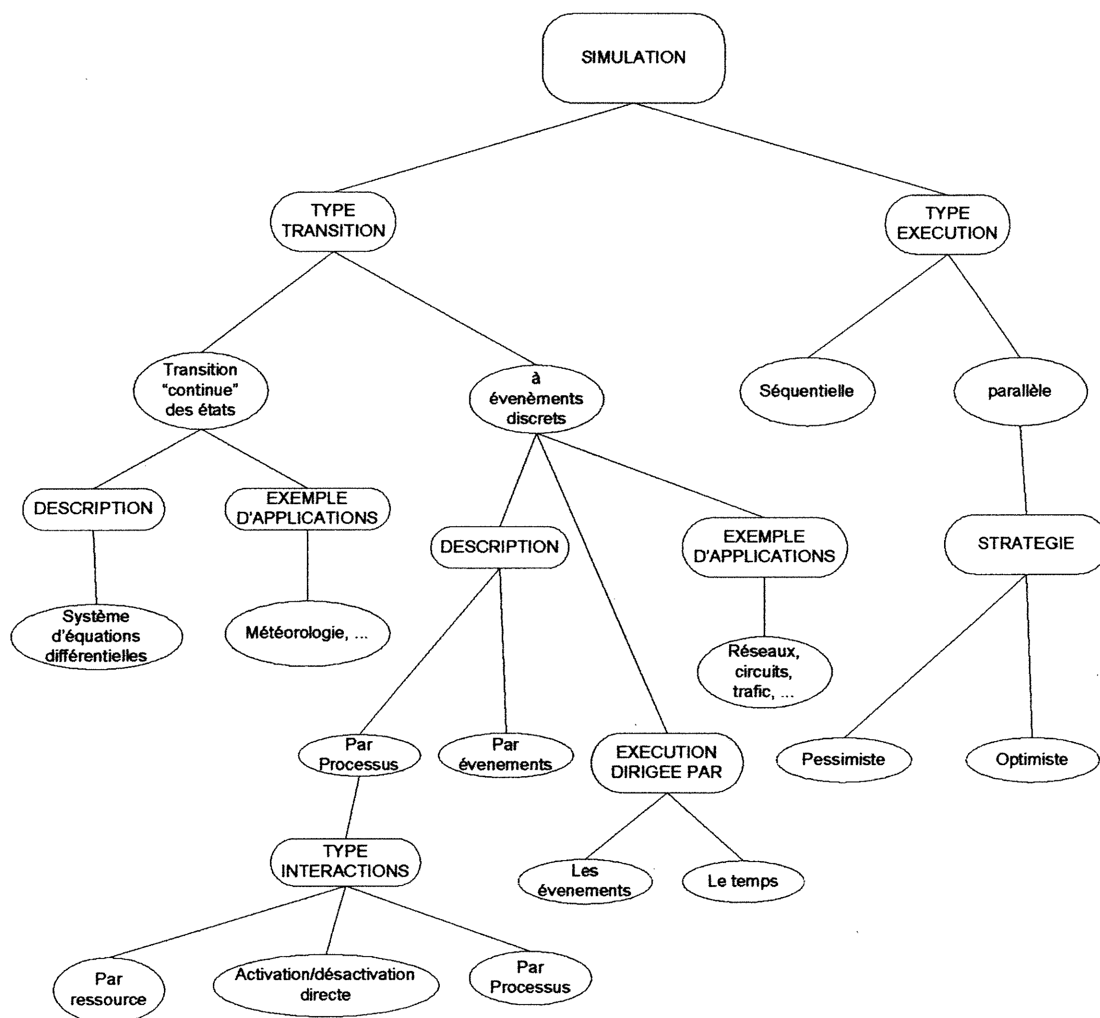


**Figure 8** Les différentes techniques de simulation

La simulation constitue un champ d'application particulier pour l'expérimentation de problèmes liés au parallélisme. La nécessité constamment affirmée de disposer de résultats de plus en plus rapidement sur des modèles de plus en plus complexes a conduit à rechercher toutes les voies possibles d'accélération des traitements.

L'article de *Misra* présente les approches pessimistes avec prévention ou détection et résolution des interblocages [Misra86]. *Ingels* et *Raynal* dans [InRa90] présentent les deux grands schémas de la simulation repartie, d'un point de vue algorithmes sous-jacents. Enfin, *Fujimoto* en 1990 [Fuj90], puis en 1994 avec *Nicol* [NiFu94], effectuent une étude quasi-exhaustive des algorithmes existants pour la simulation distribuée à évènements discrets.

Nous présenterons tout d'abord rapidement la simulation de systèmes discrétisés avec ses différents schémas d'exécution. Ensuite, nous nous intéresserons à un problème crucial dans la simulation : *la contrainte de causalité*.

### 1.3.2 La simulation à évènements discrets

Nous considérons ici le modèle d'un système physique constitué par un ensemble de processus physiques (PP). Ces processus interagissent à des dates discrétisées : ces interactions constituent les évènements. Dans un tel modèle, un processus logique (PL) modélise un ou plusieurs des PP, et les évènements du système physique sont représentés par des échanges de messages entre les PL du modèle correspondant.

La simulation à évènements discrets est présentée par *Rakotoarisoa* et *Mussi* dans [RaMu91]. On appelle évènement le changement d'état du système provoqué par l'apparition de données venant de l'extérieur ou résultant de ses traitements internes.

La simulation à évènements discrets a les caractéristiques suivantes :

- Les variables d'état, qui décrivent l'état du système physique, sont discrètes. Elles sont regroupées dans des vecteurs d'état ;

- Les états sont discrets, c'est à dire qu'ils apparaissent à des instants discrets (i.e., l'état du système ne peut changer de manière continue). Les instants d'occurrence de ces évènements sont appelés *date* des évènements;

- Les variables continues sont discrétisées : elles ne sont prises en compte qu'aux instants où elles sont utilisées, c'est à dire au moment d'apparition des évènements. Le temps physique est ainsi remplacé par le temps logique.

On appelle noyau de synchronisation l'ensemble des procédures d'entretien de l'échéancier (ajout, suppression et modification d'éléments). L'occurrence d'un évènement suivi de son traitement, modifiant l'état du système (donc l'échéancier et la date de la simulation) constitue une activité de la simulation. La simulation à évènements discrets peut être décrite de deux façons distinctes :

- D'une part, grâce à une technique basée sur la notion d'évènements, où le type de l'évènement définit l'action de la simulation. Après un changement d'état, l'évènement en tête de l'échéancier provoque l'exécution de la procédure correspondante qui est fournie par le noyau de synchronisation. Ceci peut éventuellement créer d'autres évènements.

- D'autre part, par une méthode basée sur la notion de processus, où chaque activité de la simulation est considérée comme l'activité d'un processus. Ainsi, la simulation est constituée d'un ensemble de processus s'exécutant en parallèle. Le noyau de synchronisation n'est plus un ensemble de procédures comme dans le cas précédent, mais un ensemble de primitives de manipulation de processus (activer, suspendre,...). Elle permet la distinction entre deux entités dans le système: les entités passives (clients) qui sont les messages circulant entre les processus, et les entités actives (serveurs) qui sont les processus traitant ces messages.

Il existe deux types de simulation à évènements discrets :

- La simulation dirigée par le temps entretient une horloge centrale qui progresse par pas de temps de longueur arbitraire fixée par le problème

traité. A chaque pas de temps, la liste des évènements est parcourue et chaque évènement portant la date courante, est simulé. Le choix d'un pas de temps approprié pour le problème est primordial: il permet d'éviter d'avoir à parcourir la liste alors qu'il n'y a aucun évènement à traiter pour cette date;

- Dans la simulation dirigée par les évènements, il n'y a pas d'horloge centrale. Les paramètres qui représentent le temps sont les dates d'apparition des évènements. Le temps progresse donc dans la simulation au fur et à mesure que les évènements sont traités. Pour gérer la chronologie des évènements, ceux-ci sont triés par ordre croissant dans une liste appelée échéancier.

### 1.3.3 Accélération par parallélisation

La simulation informatique de grands systèmes complexes reste un élément important de la recherche et des développements actuels. Les besoins en calcul continuent de croître et dépassent largement les capacités des ordinateurs monoprocesseur. Durant les quinze premières années de l'existence de la simulation distribuée, les techniques introduites se sont avérées plus fondamentales qu'une application directe du problème des horloges logiques de *Lamport* [Lam78]. La disponibilité actuelle de technologie de calcul parallèle ou distribué et de communication lui a donné un sens qui n'avait pas été envisagé à ses débuts. En effet, la recherche menée sur les parties critiques de la simulation distribuée commence à produire des avancées dans des domaines informatiques différents de la simulation dans sa signification classique.

A titre d'exemple, l'exécution simulée de certains programmes *SIMD* dans un environnement asynchrone peut accélérer leur exécution [ShKl92], et il a été démontré que des simulations parallèles de programmes parallèles communicants par échanges de messages sont possibles [DHN94]. D'autres travaux ont montré que la collecte non intrusive de traces d'exécution de programmes parallèles utilisant de la mémoire distribuée est réalisable en superposant l'exécution et un protocole de

simulation à évènements discrets [TuCai93]. Le difficile problème de « déverminage » (*debugging*) de programmes parallèles pourrait trouver une solution avec de telles approches.

#### 1.3.3.1 Méthodes de distribution

Afin de distribuer l'exécution de la simulation sur plusieurs processeurs (on parlera alors de simulation distribuée à évènements discrets ou *PDES*), cinq grandes options pour la décomposition ont été proposées dans [RW89]. Ces possibilités de décomposition s'intéressent, soit au simulateur lui-même, soit au modèle à simuler, soit aux deux.

**Parallélisation automatique :** L'utilisation d'un compilateur parallélisant permet de distribuer l'exécution directement à partir du code séquentiel, en détectant les portions indépendantes de code. Cette méthode, transparente pour l'utilisateur, ne permet pas de détecter, ni d'exploiter le parallélisme intrinsèque du modèle à simuler, eu égard à la nature irrégulière du modèle et aux multiples dépendances de données. Les accélérations obtenues grâce à ces techniques restent faibles [ChaBro83].

**Distribution des expériences :** L'exécution d'un même code séquentiel de la simulation est affectée à chaque processeur de la machine. Les exécutions diffèrent d'un processeur à l'autre par les paramètres d'entrée fournis. Les différentes simulations n'interagissant quasiment pas entre elles, cette technique parait efficace. Cependant, elle reste une méthode séquentielle avec les mêmes problèmes que la simulation sur monoprocesseur (performances, limitations mémoire, etc.).

**Distribution des fonctions du simulateur :** Les différentes fonctions du simulateur sont ici distribuées. La génération de nombres aléatoires peut être affectée à un ou plusieurs processeurs (spécialisé(s) ou non), les entrées/sorties à un autre groupe, l'évaluation d'un type de composants du modèle à un autre, etc. Cette approche reste transparente à l'utilisateur, et un nombre limité de processeurs peut

être utilisé efficacement. De plus, une telle approche permet l'utilisation de composants spécifiques ou spécialisés pour certaines fonctions.

**Distribution des évènements :** Dans cette approche, l'exécution des évènements est distribuée à chaque processeur à partir d'une liste d'évènements ou d'un échéancier global, similaire à celui de la simulation séquentielle. Ce principe est bien adapté aux machines à mémoire partagée et est surtout intéressant quand le volume d'informations partagées par les processus et/ou d'informations globales est important.

**Distribution des éléments du modèle :** Pour cette dernière méthode de distribution, le modèle à simuler est décomposé en éléments : ceux-ci sont faiblement couplés et leur simulation est affectée à autant de processus (un processeur pouvant exécuter plusieurs processus). Les différents processus interagissent entre eux en échangeant des messages estampillés. Cette approche permet d'exploiter le parallélisme du modèle à simuler. Mais le couplage faible entre les différents éléments implique l'utilisation de protocoles de synchronisation que nous introduisons dans la suite du chapitre.

Nous venons de décrire rapidement les solutions envisageables pour la distribution de la simulation à évènements discrets. Il convient cependant de souligner que toutes ces voies ne sont pas mutuellement exclusives et débouchent sur une multitude d'approches « mixtes » combinant plusieurs des options précédentes.

Avant de présenter les méthodes de simulation distribuée dirigées par les évènements, nous introduisons la notion importante de *causalité* pour l'exécution distribuée.

#### 1.3.3.2 Le principe de la causalité

Un des aspects importants des modèles de simulation est que les systèmes physiques obéissent toujours au *principe de causalité*. Ceci signifie simplement que le futur ne peut influencer le passé. La causalité impose un ordre partiel des transitions entre états dans les systèmes physiques. Si une transition a des

conséquences sur une autre, cette dernière doit toujours se produire après la première, c'est-à-dire que la cause doit toujours précéder l'effet. Les transitions démunies de toute relation de cause à effet, directe ou indirecte, ne sont soumises à aucune contrainte d'ordonnancement et n'ont pas besoin de se produire dans des créneaux bien définis du temps physique. Par conséquent, la causalité impose un ordre partiel entre les transitions. *Lamport* [Lam78], puis *Mattern* [Mat88] et *Fidge* [Fid88], ont défini des méthodes basées sur l'utilisation d'estampilles, scalaires puis vectorielles, pour permettre aux processus de respecter la causalité.

Cet ordre partiel imposé aux transitions du système physique induit un ordre partiel entre les évènements du modèle de simulation. En particulier, l'ordre dans lequel le simulateur produit des évènements doit être cohérent avec cet ordre partiel, pour que le simulateur soit le juste reflet du comportement du système physique. Par exemple, si la transition A du système physique intervient à la date 3 et a une influence sur la transition B intervenant à la date 5, le simulateur doit générer l'évènement modélisant la transition A avant celui modélisant la transition B.

Pour une implémentation séquentielle, la causalité est aisément assurée en ordonnant les évènements par temps simulé croissant dans un échéancier : le prochain évènement (i.e., celui avec la plus petite estampille) sera le prochain à être généré. Le programme efface itérativement le prochain évènement de l'échéancier et appelle la procédure modélisant cet évènement. Cette procédure met à jour les variables d'état et insère, au besoin, dans l'échéancier de nouveaux évènements.

La distribution de la simulation sur un réseau de processeurs pose un problème pour le maintien de cette relation de causalité. Plusieurs évènements causalement liés peuvent en effet se produire simultanément sur des processeurs distants. Le schéma de distribution posant le plus de problèmes est celui où ce sont les éléments du modèle qui sont affectés à différents processeurs. En effet, l'échéancier est alors distribué, et chacun des processus en détient une instance dans laquelle sont exclusivement stockés les évènements à évaluer localement. Les autres approches utilisent principalement des simulateurs séquentiels ou un seul échéancier.

Quand les éléments du modèle sont distribués sur des processeurs distants, gérant chacun son échéancier, il faut introduire des protocoles supplémentaires pour assurer une exécution causalement correcte.

La suite de cette section est consacrée à la présentation des principaux protocoles de synchronisation proposés pour la simulation à évènements discrets.

### 1.3.4 Evolution synchrone des processus

La simulation synchrone implémente le temps simulé comme une horloge globale, qui sera représentée soit explicitement comme une structure de donnée centralisée, soit implicitement comme une procédure dont l'exécution est séquencée par le temps.

La caractéristique fondamentale de l'exécution est qu'à chaque instant (dans le temps physique), tous les processus logiques simulent le même temps logique (i.e., on assure que tous les PL avancent ensemble et à la même vitesse).

Dans ce type de simulation synchrone, toutes les horloges logiques des PL possèdent une valeur identique à chaque instant dans le temps physique. En d'autres termes, la simulation s'exécute en fonction d'une pseudo horloge globale, puisque toutes les horloges locales possèdent la même valeur. Ainsi, chaque PL doit évaluer tous les évènements de date `i` avant que les PL ne soient autorisés à exécuter les évènements de date `i+1` et suivants. Entre les horloges logiques, on pourra en fait noter un léger décalage qui est lié à la diffusion du signal de synchronisation. On est cependant sûr que, dès qu'un site évalue la date `d`, plus aucun site n'évalue la date `d-1`. Une telle stratégie simplifie considérablement l'implantation de simulateurs distribués respectant la contrainte de causalité, car elle supprime les problèmes d'interblocage, de retours-arrière et de gestion mémoire, rencontrés dans les approches asynchrones (présentées dans les sections suivantes). Par contre, le déséquilibre de charge à certaines dates logiques implique des cycles inactifs qui nuisent aux performances.

Des implantations centralisées et distribuées des horloges globales ont été réalisées.

Une implantation centralisée, où un processeur dédié gère l'horloge, a été proposée dans [VRL86]. Afin d'éviter de passer par des étapes où il n'y a pas d'évènements à évaluer, des algorithmes ont été développés pour déterminer la prochaine date à laquelle un évènement va effectivement se produire. Lorsque cette prochaine date `d'` a été déterminée, tous les PL avancent leur horloge locale jusqu'à cette date.

Pour une implantation distribuée, on utilise un synchroniseur [Awe85] qui permet l'exécution synchrone sur une machine asynchrone. Ce synchroniseur génère une pulsation à chaque cycle d'horloge globale. Il attend que toute activité ait cessé sur les processus dans la pulsation courante pour avancer l'horloge globale et passer à la pulsation suivante en autorisant les sites propriétaires d'évènements estampillés par celle-ci à les exécuter en parallèle. Cette approche est appelée *liste multiple d'évènements synchronisés* [PLH88] et a été proposée par *Peacock* et *al.* dans [PWM79]. Une organisation hiérarchique des PL peut également être utilisée [Con89] pour déterminer la prochaine date d'évaluation. Une autre possibilité consiste à utiliser un algorithme de *distributed snapshot* pour éviter le goulot d'étranglement du synchroniseur [CL85].

### 1.3.5 Evolution asynchrone des processus

Nous venons de voir, dans la section précédente, que les méthodes synchrones nécessitent un nombre important d'opérations globales pour la synchronisation des PL, à la fin unique de maintenir une seule valeur d'horloge sur l'ensemble des sites. Si la densité d'évènements à évaluer à chaque étape de temps simulé est faible, le surcoût introduit devient trop élevé pour espérer obtenir des gains en performance intéressants.

Une autre approche, dite asynchrone, laisse la responsabilité à chaque processus de gérer sa propre horloge locale. Le respect de la contrainte de causalité

devient alors un problème crucial, car ces horloges peuvent être différentes d'un site à l'autre. Afin de maintenir cette relation, des protocoles de synchronisation sont introduits. Ils permettent de gérer les potentiels interblocages entre les PL. Ces interblocages sont induits par les techniques de distribution de la simulation et ne découlent pas directement des modèles représentant le système physique à étudier. Dans ce contexte, trois stratégies, que nous présentons par la suite, prennent en charge le problème des interblocages du système :

- En les évitant par l'utilisation de messages *NULL*;
- En les autorisant, mais en offrant un mécanisme pour les détecter et les résoudre;
- En les ignorant en cherchant à conserver actif un nombre maximal de processeurs. On autorise les processus à traiter les évènements dont ils ont la connaissance, même s'il y a des canaux d'entrée vides sur lesquels peuvent encore arriver des messages. Ceci peut conduire au non-respect de la contrainte de causalité, donc il faut ajouter un mécanisme de retour arrière (*rollback*) afin de restaurer la causalité en cas d'erreur.

Les deux premières stratégies sont dites *conservatives* (ou *pessimistes*), tandis que la troisième est dite *spéculative* (ou *optimiste*).

Avant de présenter en détail les trois stratégies que nous venons d'introduire, il convient de faire certaines hypothèses concernant le comportement du système et des communications :

- Les canaux de communication sont *FIFO* (i.e., les messages sont reçus dans l'ordre où ils ont été émis) ;
- Les communications sont sûres (i.e., pas de perte, ni d'altération des messages) ;
- Chaque processeur dispose de suffisamment de mémoire (i.e., on considère la mémoire comme infinie) ;

- Les messages d'un processus vers un autre sont estampillés avec des valeurs non décroissantes ;

- Toute évaluation est le résultat de la réception d'un message (i.e., pas de « génération spontanée »). Ceci implique que si une évaluation sur un site $P_i$ génère un autre évènement dans le futur pour un site $P_j$, alors un message est échangé entre $P_i$ et $P_j$ (dans le cas où $i=j$, on envoie virtuellement ce message);

- L'estampille d'un message doit être au moins égale à celle du message qui a provoqué son envoi.

Parmi les trois premières hypothèses (qui portent directement sur l'architecture utilisée), seule la dernière peut être discutable : si on considérait la mémoire comme infinie d'un point de vue théorique, on se restreindra, dans la pratique, à des simulations dont les besoins seront compatibles avec la mémoire physiquement disponible. Les deux dernières hypothèses de travail sont directement intégrées dans le noyau de simulation utilisé.

#### 1.3.5.1 Approches conservatives (ou pessimistes)

Ces approches sont dites conservatives en ce sens que les processus n'avancent leur horloge qu'avec précaution, seulement quand ils sont surs que la contrainte de causalité locale ne sera pas violée.

##### 1.3.5.1.1 Principe

En 1979, *Chandy*, *Misra* [ChMis79] et *Bryant* [Bry79] ont développé, indépendamment, certains des premiers algorithmes de simulation à évènements discrets asynchrones. Ici, on ne dispose plus d'une horloge globale : chaque processus dispose d'une horloge locale, le problème étant de maintenir la cohérence des horloges et la relation de causalité.

La majeure partie des approches conservatives est basée sur l'algorithme de *Chandy* et *Misra*. Celui-ci est donne par l'algorithme 1.1 avec les notations suivantes:

- P représente le nombre de processeurs ;
- N représente le nombre de PL ;
- $\{PL_k\}$ représente l'ensemble des PL simulés par le processeur k;
- $PL_i.t$ indique la date courante dans la simulation pour le $PL_i$ quelque soit $i \in 1..N$;
- $PL_i.numCanal$ est le nombre de canaux en entrée pour le $PL_i$ quelque soit $i \in 1..N$;
- $PL_i.h_j$ contient la date du dernier message reçu par le $PL_i$ (quelque soit $i \in 1..N$) sur le canal j (quelque soit $j \in 1..\ PL_i.numCanal$);
- 0..dateFin est l'intervalle de temps sur lequel on exécute la simulation.
- TVL est le Temps Virtuel Local.

On retrouve dans la Figure 9 la structure d'un noyau du simulateur distribué.

---

**Algorithme 1.1 : Algorithme pessimiste de simulation distribuée**

```
1.1.1   début
1.1.2     ▷ Initialisation
1.1.3     pour k=1 à P faire en parallèle
1.1.4       pour tous PLi ∈ {PLk} faire
1.1.5         PLi.t ← 0 ;
1.1.6         pour j=1 à PLi.numCanal faire
1.1.7           PLi.hj ← 0 ;
1.1.8         fin pour
1.1.9       fin pourTous
1.1.10    fin pour
1.1.11    ▷ Simulation
1.1.12    pour k=1 à P faire en parallèle
1.1.13      tant que min_{PLi∈{PLk}}(PLi.t) ≠ dateFin faire
1.1.14        pour tous PLi ∈ {PLk} faire
1.1.15          pour tous messages reçus avec une estampille inférieure à PLi.t faire
1.1.16            Exécuter l'événement associé et mettre à jour l'échéancier ;
1.1.17            Envoyer les messages générés s'il y en a ;
1.1.18          fin pourTous
1.1.19          pour j=1 à PLi.numCanal faire
1.1.20            Recevoir les messages sur le canal j ;
1.1.21            Mettre à jour PLi.hj ;
1.1.22          fin pour
1.1.23          PLi.t ← min_{1≤j≤PLi.numCanal}(PLi.hj) ;
1.1.24        fin pourTous
1.1.25      fin tantQue
1.1.26    fin pour
1.1.27  fin
```

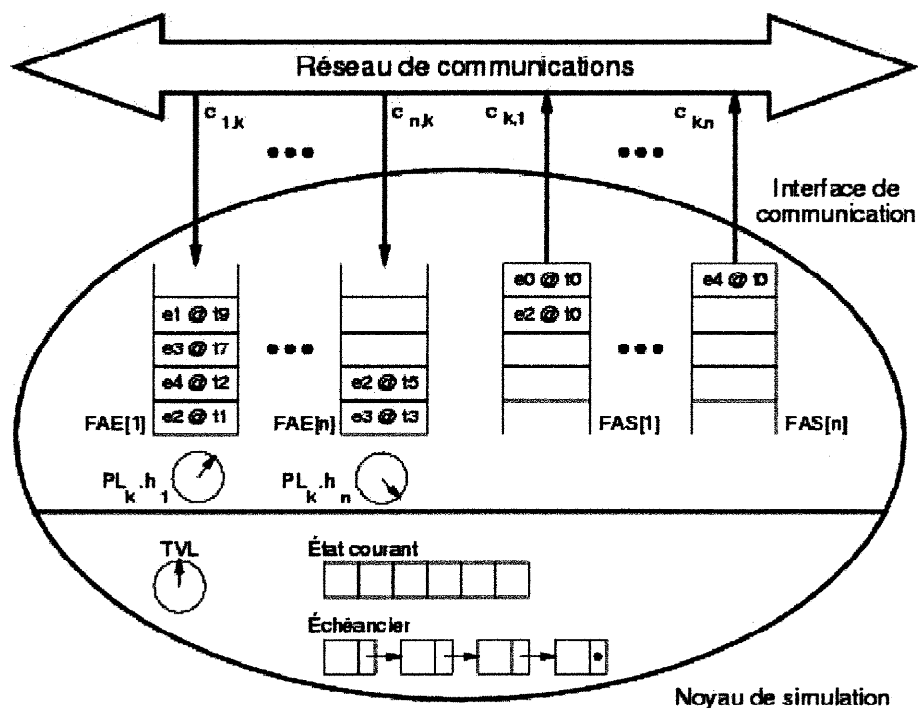


**Figure 9** Structure d'un processus logique conservatif

Cependant, cet algorithme est sujet à de fréquents interblocages lors de la simulation de réseaux cycliques de PP. En fait, des interblocages peuvent survenir également dans le cas de réseaux acycliques. Ce type de situation est essentiellement lié à la méthode de distribution employée. L'ensemble de ces interblocages potentiels impose donc la présence d'un protocole pour le contrôle de l'exécution distribuée et assurer ainsi, la sûreté causale et la vivacité de la simulation. Les actions correspondant à la ligne 1.1.23 de l'algorithme précédent nécessitent, de plus, des calculs de réduction globale, dont le coût est élevé (surcharge sur le réseau et interruption des calculs de la simulation sur les processeurs).

Ces approches nécessitent que l'on définisse statiquement les liens qui permettent de communiquer entre les processus. À chaque lien est associée une horloge qui est égale à la plus petite estampille des messages qu'il contient ou à l'estampille du dernier message reçu si le lien est vide. Le processus prend le message ayant la plus petite estampille sur le lien avec la plus petite horloge et le traite. Si ce lien est vide, le processus se bloque. Ceci garantit que les messages seront traités

selon un ordre non décroissant des estampilles, assurant ainsi la contrainte de causalité locale.

#### 1.3.5.1.2 Prévention des interblocages

Pour éviter les interblocages, on utilise des messages NULL. Ces messages NULL servent uniquement pour des besoins de synchronisation et ne correspondent à aucune activité du système physique. Un message NULL d'estampille $t_{NULL}$ envoyé de $PL_A$ vers $PL_B$ est une information indiquant que $PL_A$ n'enverra pas plus tard à $PL_B$ un message avec une estampille inférieure à $t_{NULL}$. On détermine la valeur de l'estampille en considérant le minimum des estampilles des messages d'entrée, augmenté de la durée de traitement de l'évènement par le PL. Ceci nous donne alors une borne inférieure de l'estampille des prochains messages envoyés sur les liens de sortie. Quand un processus a fini l'évaluation d'un évènement, il envoie un message NULL sur chacune de ses sorties. Les receveurs de ces messages peuvent alors déterminer une borne inférieure, l'envoyer à leurs voisins, et ainsi de suite. *Peacock* et *al.* montrent dans [PWM79] que ce mécanisme évite les interblocages tant que l'on n'a pas de cycle dans le système où l'incrément de l'estampille des messages circulant dans le cycle serait 0 (*réseau à rétroaction sans prévision*).

#### 1.3.5.1.3 Détection/Résolution des interblocages

En 1981, *Chandy* et *Misra* suggèrent de laisser le système s'interbloquer, et de fournir un mécanisme pour détecter et résoudre cet interblocage [ChMis81]. Pour ce faire, un processus de contrôle doit tout d'abord détecter l'interblocage grâce à un algorithme réparti tels ceux de *Dijkstra* et *Scholten* [DS80] ou *Chandy* et *Misra* [ChMis82].

Une fois l'interblocage détecté, le mécanisme en lance la résolution. Le but de ce nouveau calcul est de déterminer les processus qui peuvent redémarrer, sans risquer de violer la relation de causalité. De façon triviale, la simulation pourra être relancée sur au moins un processus car, parmi tous les messages en attente sur les canaux de tous les processus, celui qui a la plus petite estampille $t_{min}$ peut être

consommé sans risque par son destinataire. En effet, aucun évènement ne peut plus modifier le système avant cette date $t_{min}$. La procédure de résolution de l'interblocage consiste donc simplement à déterminer cette date minimum et à autoriser les processus à consommer les messages estampillés avec $t_{min}$.

#### 1.3.5.1.4 Optimisations

Des variations de l'approche avec messages NULL ont été proposées. L'une d'entre elles, due à *Misra* [Misra86] cherche à diminuer le nombre de messages NULL: lorsqu'un processus se bloque parce qu'un de ses liens devient vide, il demande au processus du côté émetteur du lien d'envoyer le message suivant (NULL ou non) et, dès qu'il le reçoit, reprend l'exécution. Cette méthode permet de réduire sensiblement le nombre de messages NULL, mais nécessite un délai plus important pour recevoir les messages NULL à cause du transfert de la requête (i.e., deux messages au lieu d'un seul).

*Lubachevsky* utilise une fenêtre temporelle pour réduire le surcoût nécessaire à la validation du choix d'évaluation d'un évènement [Lub89]. La borne inférieure de la fenêtre est définie par le minimum des estampilles des évènements en attente d'être évalués. Les seuls évènements évaluables sont ceux dont l'estampille appartient à cette fenêtre temporelle. L'utilité de cette dernière est qu'elle permet de réduire le domaine de recherche que l'on doit parcourir pour déterminer si on est sûr de pouvoir évaluer un évènement.

La prédiction sur les sorties revêt un aspect important dans la résolution des interblocages. Supposons que la date de simulation d'un processus logique $PL_i$ est $t_i$, c'est-à-dire que l'évènement en cours de traitement porte cette date. Il est alors possible pour $PL_i$ de déduire la liste des messages qu'il enverra vers ses voisins jusqu'à la date $t'_i$ en fonction de ce qu'il a reçu auparavant jusqu'à la date $t_i$, et indépendamment des messages qui lui parviendront entre les dates $t_i$ et $t'_i$. On appelle *prévision* (*lookahead*) sur un canal de $PL_i$ vers $PL_j$ la valeur :

$$L_{ij} = t'_i - t_i$$

De façon plus formelle, la date jusqu'à laquelle $PL_i$ peut prédire les évènements qu'il enverra vers $PL_j$ est :

$$t_{OUT\ ij} = t_{IN\ i} + L_{ij}$$

où $L_{ij}$ est la *prévision* et $t_{IN\ i} = \min_{1 \leq k \leq PL_i.numCanal} (PL_i.h_k)$, c'est-à-dire le minimum des dates des files d'entrée de $PL_i$. Nous pouvons ainsi faire suivre vers les PL voisins la date minimum à laquelle un message pourra arriver sur ce canal.

#### 1.3.5.1.5 Difficultés

L'inconvénient le plus important des approches conservatives est leur incapacité à exploiter totalement tout le parallélisme possible dans la simulation. Si un évènement $E_A$ peut affecter $E_B$, les approches conservatives vont exécuter $E_A$ et $E_B$ séquentiellement dans tous les cas, alors que si $E_A$ affecte rarement $E_B$, les évènements pourraient être évalués la plupart du temps en parallèle.

Un autre problème important est que ces approches requièrent une configuration statique: on ne peut pas dynamiquement créer de processus et l'interconnexion entre processus doit être définie statiquement. Des techniques existent pour résoudre ce problème, mais elles conduisent généralement à des surcharges excessives de calcul.

L'utilisation de messages NULL dans la prévention de l'interblocage implique que certains types de simulation ne peuvent pas être réalisés. C'est le cas des simulations où le système à étudier contient des cycles dans lesquels l'incrément de l'estampille des messages est nul (par exemple les réseaux de file d'attente dans lesquels le délai minimum pour un service serait nul).

Enfin, la plupart des schémas conservatifs nécessitent pour le fonctionnement du protocole de synchronisation qu'une connaissance du comportement du processus logique soit donnée par le programmeur. Mais des informations, comme l'incrément minimal des estampilles ou l'assurance que certains évènements n'auront pas d'effet sur d'autres, peuvent être difficiles à donner pour des simulations complexes.

### 1.3.5.2 Approche spéculative (ou optimiste)

Le second type d'approche asynchrone est dit *spéculatif* (ou *optimiste*) en ce sens qu'il autorise les processus à progresser aussi vite qu'ils le peuvent. L'objectif est d'obtenir un degré maximum de parallélisme dans l'exécution. Cependant, le risque dérivant de l'évaluation d'évènements non sûrs est une simulation erronée. Nous avons en effet vu, dans le paragraphe précédent, que les algorithmes de contrôle conservatifs bloquent les processus qui n'ont pas d'évènement sûr à évaluer. Ceci diminue le parallélisme obtenu dans l'exécution. On va donc autoriser les processus à exécuter tous les évènements dont ils disposent (tout en respectant la règle de causalité locale) sans attendre la certitude qu'ils sont bien ordonnés par rapport aux exécutions sur les autres processeurs. Lorsqu'une erreur de causalité se produit, le processus doit payer le prix de son optimisme en retournant dans l'état consistant précédant l'erreur. Ceci est réalisé au moyen d'un mécanisme de retour arrière (*rollback*).

L'idée essentielle des approches optimistes réside dans l'hypothèse que le coût des corrections de telles erreurs sera inférieur au temps durant lequel les processeurs seraient restés inactifs.

#### 1.3.5.2.1 Principe

Le mécanisme de distorsion du temps (*Time Warp*), basé sur le principe du temps virtuel décrit dès 1985 par *Jefferson* [Jef85], est le plus connu des protocoles optimistes. Il est également utilisé dans d'autres cadres que la simulation : système d'exploitation [JBW+87], exécution parallèle de code séquentiel avec *ParaTran* [TK88].

Le principe de base de cette approche est d'exploiter tout le parallélisme possible dans la simulation, afin de saturer de calculs les processeurs. Les hypothèses de base pour l'approche optimiste sont moins contraignantes que pour la stratégie conservative : on n'a pas besoin du fonctionnement *FIFO* des canaux de communication, même si un processus doit toujours émettre une séquence de

messages d'estampilles croissantes vers un autre processus. La communication reste asynchrone.

Chaque processus logique contient (Figure 10 ) :

- un vecteur d'état modélisant l'état du processus physique,
- une horloge locale indiquant la valeur locale du temps virtuel,
- une liste des messages en entrée, triée suivant les dates de réception croissante,
- une liste des messages envoyés, triée suivant les dates d'émission croissante,
- une liste, utilisée pour les retours-arrière, des vecteurs d'état et leur date associée.

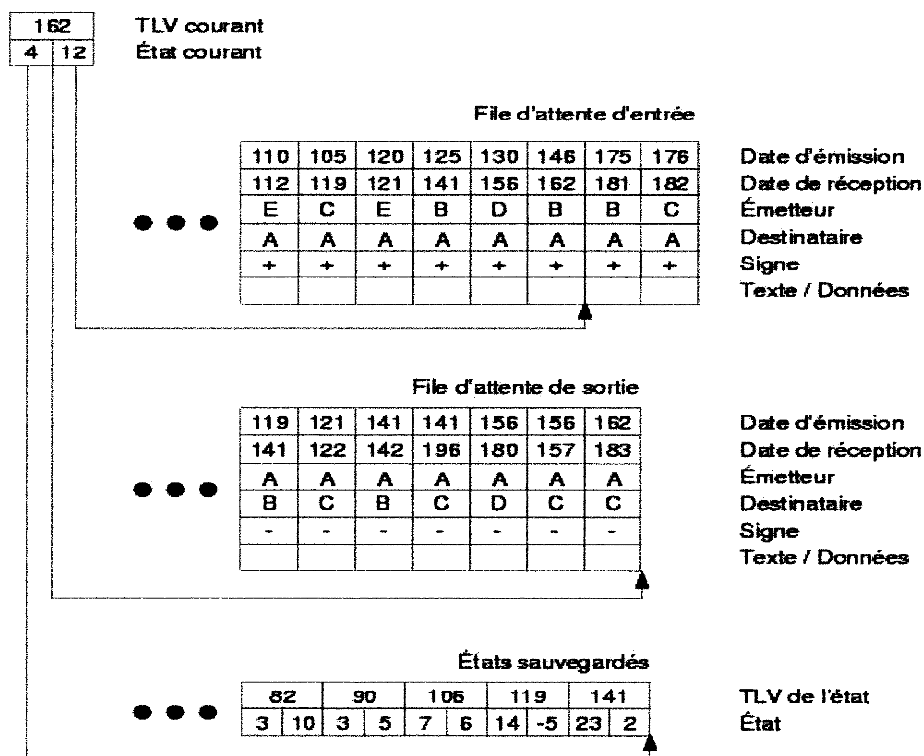


**Figure 10** Un processus *Time Warp*

Un processus traite les messages de sa liste d'entrée un à un, et ce, sans se soucier des horloges locales des autres processus. À chaque message traité, l'horloge

locale est mise à jour avec la valeur de l'estampille du message (Figure 11). Si la liste d'entrée est vide, l'horloge locale prend la valeur $+\infty$.

Un processus peut détecter une violation de la causalité locale si l'estampille du message reçu est inférieure à celle du dernier message traité (i.e., la valeur de l'horloge locale). On doit alors corriger cette erreur en effectuant un retour-arrière pour rétablir la causalité. Cette causalité a été respectée (à priori) jusqu'à la date précédant immédiatement l'estampille du message en retard que l'on vient de recevoir.

Le mécanisme de retour-arrière permet de régresser dans le temps local virtuel jusqu'à une situation causalement correcte d'estampille antérieure à celle du message reçu. On doit alors annuler les évaluations erronées, ainsi que les effets de bord.

| Nom Expéditeur | Nom Destinataire | Date virtuelle d'émission | Date virtuelle de réception | Signe | Texte / Données |
|---|---|---|---|---|---|

**Figure 11** Structure des messages échangés entre processus optimistes

Soit $t_f$ l'estampille du message ayant provoqué l'erreur de causalité. L'exécution de la simulation doit donc retourner à une date antérieure à $t_f$. Ceci est réalisé en utilisant le vecteur d'état de plus grande date $t_c$ inférieure à $t_f$. Une fois dans cet état consistant de date $t_c$, on peut insérer le message dans la liste d'entrée, effacer les vecteurs d'état de date supérieure à $t_c$, annuler les messages envoyés d'estampille supérieure à $t_c$ par des antimessages et reprendre normalement l'exécution (Figure 12).

Un antimessage est identique au message auquel il est associé, sauf pour le champ signe (Figure 11) qui permet ainsi de distinguer la copie positive (i.e., le message) de la copie négative (i.e., l'antimessage).

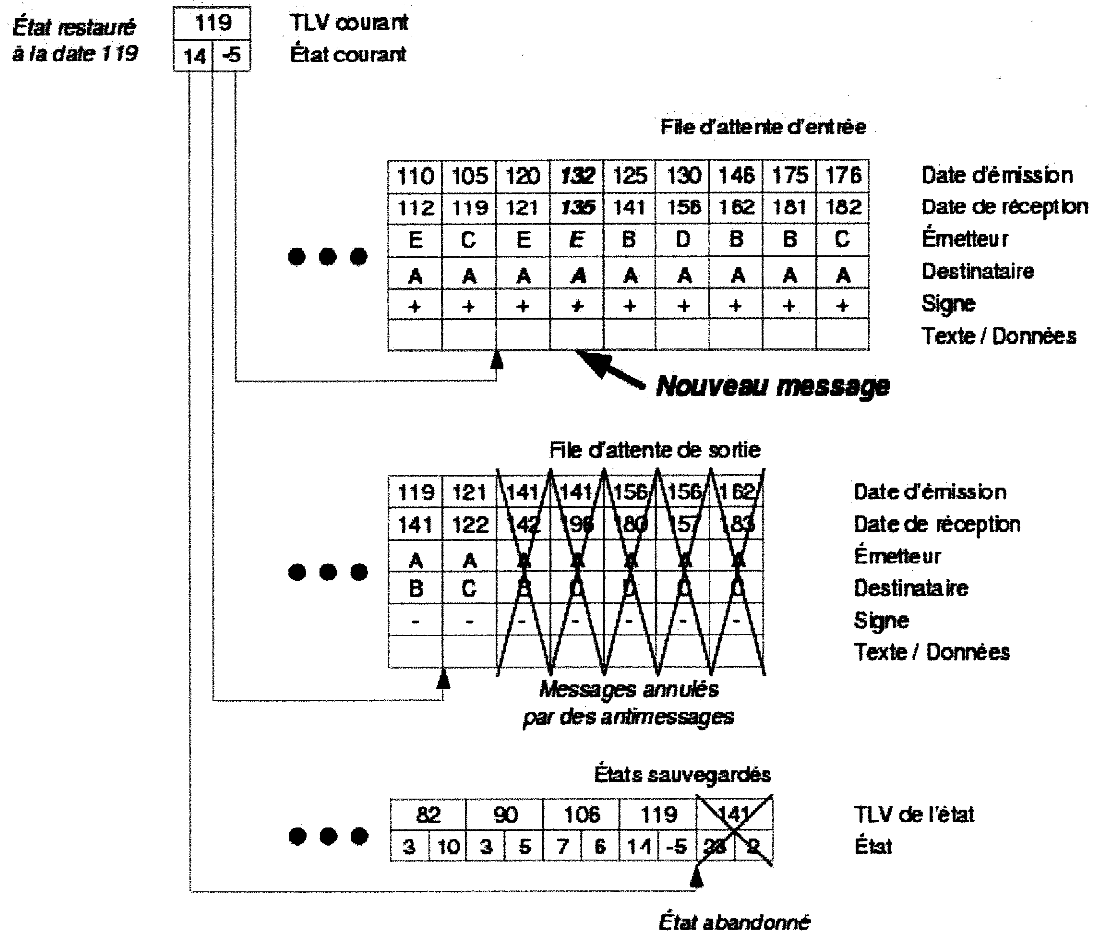


**Figure 12** État du processus après réception d'un message en retard

Lorsqu'un message $+m$ et son antimessage $-m$ se trouvent dans une même file, ils s'annulent :

$$+ m - m = 0$$

$$Insert\ (\ Insert\ (file,\ +m),\ -m\ ) = file$$

Le Tableau I donne les actions à entreprendre lors de la réception des deux types de messages, en fonction de l'estampille portée par le message.

**Tableau I** Actions entreprises lors de la réception d'un message

| | **Arrivée d'un message +m** | **Arrivée d'un antimessage -m** |
|---|---|---|
| **Estampille(m) >= LVT**<br>**(i.e., dans le futur local)** | L'antimessage -m est présent dans la file d'entrée<br>Annule l'antimessage -m<br>Insere(liste, +m)<br>L'antimessage -m n'existe pas | Le message +m est présent dans la file d'entrée (mais pas encore évalué)<br>Annule le message +m<br>Insere(liste, -m)<br>Le message +m n'existe pas |
| **Estampille(m) < LVT**<br>**(i.e., dans le passé local)** | L'antimessage -m est présent dans la file d'entrée<br>Annule l'antimessage -m<br>Rollback<br>Insere(liste, +m)<br>L'antimessage -m n'existe pas | Le message +m est présent dans la file d'entrée (et a été évalué)<br>Rollback<br>Annule le message +m<br>Insere(liste, -m)<br>Le message +m n'existe pas |

Lors du retour-arrière d'un processus de l'instant $t_f$ dans un état consistant $t_c$, il convient d'annuler tous les messages envoyés postérieurs à $t_c$ : On envoie donc les antimessages associés dont on a conservé une copie dans la liste des messages envoyés (File d'attente de sortie de la Figure 10).

Le mécanisme de contrôle local laisse irrésolus un certain nombre de problèmes importants : Comment être sûr que, malgré les retours-arrière, le système progresse globalement ? Comment éviter de manquer de mémoire sur les noeuds de la machine quand le mécanisme de contrôle local requiert la sauvegarde de deux copies de chaque message et plusieurs copies des états du processus ? Comment détecter la terminaison ? Comment gérer les entrées/sorties en tenant compte des retours-arrière ?

Le mécanisme de contrôle global permet de résoudre tous ces problèmes.

Le concept principal du mécanisme de contrôle global est le *temps virtuel global* (TVG). Le temps virtuel global est défini comme le minimum de :

1. tous les temps virtuels de toutes les horloges virtuelles à la date physique t,

2. des dates d'expédition de tous les messages envoyés et non encore reçus à cette même date physique t.

Il a été montré par induction sur le nombre d'actions de communication, que le TVG ne décroît jamais. Ainsi le TVG sert de valeur minimum pour le temps virtuel jusqu'auquel un processus peut, au pire, revenir en arrière. *Samadi* présente un algorithme permettant l'évaluation du TVG dans [Sam85].

Le TVG permet ainsi de pouvoir valider (*commit*) les évènements qui se sont produits à une date inférieure. Il est en effet inutile de conserver les états passés, les messages envoyés ou reçus à une date antérieure, car il est impossible de revenir en arrière à une date inférieure au TVG.

La détection de la terminaison de la simulation, dans ce contexte distribué optimiste, ne pose pas de difficulté. En effet, nous avons vu précédemment qu'un processus dont la liste d'attente est vide (i.e., qui n'a plus, pour l'instant, d'évènement à traiter) prend une valeur d'horloge locale égale à $+\infty$. Si le TVG devient égal à $+\infty$, cela signifie que toutes les horloges locales ont pour valeur $+\infty$, et donc que tous les processus n'ont plus d'évènement à traiter. Comme on est également sûr qu'aucun message n'est en transit, aucun processus ne peut alors plus redémarrer. La terminaison est ainsi conclue.

En ce qui concerne les entrées/sorties, si un processus commande un périphérique extérieur, il est important que cette action extérieure ne soit pas validée tout de suite, car elle peut être annulée, par la suite, par un retour-arrière. On n'exécutera donc physiquement cette action que lorsque le TVG devient supérieur à sa date d'exécution, étant ainsi sûr qu'aucun retour-arrière ne pourra plus l'annuler.

#### 1.3.5.2.2 Optimisations

Des optimisations pour limiter les « dégâts » causés par un message en retard ont été proposées [Gaf88], en observant que l'arrivée d'un message en retard ne modifie pas suffisamment les calculs pour que les messages déjà envoyés soient faux.

Dans l'*annulation paresseuse*, les processus n'envoient pas immédiatement les antimessages lors d'un retour-arrière. En fait, ils attendent de savoir si la nouvelle exécution produit les mêmes messages; si les mêmes messages sont crées, il n'y a pas besoin d'annuler les premiers messages envoyés. Un antimessage n'est envoyé que si le processus en dépasse l'estampille sans régénérer le message associé.

Une autre optimisation consiste dans la *réévaluation paresseuse*. Celle-ci est similaire à l'*annulation paresseuse* mais considère les états sauvegardés plutôt que les messages. Considérons le cas où l'état d'un processus est le même après le traitement d'un message en retard que lors de l'exécution qui a précédé son arrivée : si aucun autre message n'est arrivé entre temps, il semble évident que la nouvelle exécution donnera les mêmes résultats que l'exécution précédente (l'exécution est déterministe). On peut donc ignorer la réévaluation pour reprendre l'exécution là où elle avait été interrompue lors du retour-arrière. Cette optimisation a été implémentée dans les premières versions du *Time Warp Operating System* [JBW+87].

Ces deux méthodes génèrent une propriété intéressante pour l'exécution distribuée. En effet, il devient théoriquement possible d'atteindre des accélérations *superlinéaires* (i.e., le *speedup* est plus élevé que le nombre de processeurs impliqués dans le calcul). En effet, l'exécution optimiste peut permettre de dépasser le chemin critique du modèle (i.e., la partie séquentielle minimale de l'exécution distribuée) [Gun94].

#### 1.3.5.2.3 Difficultés

Une des questions importantes est de savoir si on peut rencontrer des comportements dans lesquels le mécanisme passerait plus de temps à effectuer des retours-arrière qu'à exécuter effectivement la simulation. *Jefferson* dans [Jef85]

annonce que le temps passé par un processus à effectuer des retours-arrière dans l'approche optimiste est équivalent au temps durant lequel le processus est bloqué dans le cas conservatif.

Un problème plus sérieux réside dans le fait que l'approche optimiste nécessite une sauvegarde périodique des états du système. Cette sauvegarde implique une dégradation des performances de l'exécution et devient très pénalisante dans le cas de programmes qui nécessitent une allocation dynamique de la mémoire, car on doit alors traverser des structures de données complexes pour sauvegarder l'état du processus.

Les sauvegardes multiples, nécessaires à la gestion des retours-arrière, font que l'approche optimiste a besoin de beaucoup plus de mémoire que ses équivalents conservatifs.

L'approche optimiste doit également être capable de tolérer et de corriger n'importe quelle erreur qui pourrait être causée par une exécution erronée et annulée par un retour-arrière: si, lors d'une exécution, un processus « tombe » dans une boucle infinie, le mécanisme optimiste doit être capable d'interrompre matériellement le processus pour pouvoir reprendre l'exécution. Si une exécution erronée modifie la valeur d'un pointeur ou la valeur pointée, le mécanisme doit pouvoir restaurer le système dans l'état consistant d'origine. Actuellement, ces tâches sont laissées à la charge de l'utilisateur par la majeure partie des systèmes optimistes.

Enfin, il faut également noter que le mécanisme de distorsion du temps est beaucoup plus difficile à implémenter, à mettre au point ou à calibrer (fréquence des sauvegardes, gestion mémoire,...) que ses homologues conservatifs. La fréquence des sauvegardes, la gestion mémoire, etc., sont des paramètres dont l'impact sur les performances du système est primordial.

#### 1.3.5.3 Approches hybrides

Des tentatives diverses ont été menées pour combiner les deux grands protocoles précédents, considérés par *Aahlad* et *Browne* [AB89] comme les deux

solutions extrêmes dans l'ensemble des protocoles envisageables pour la simulation distribuée.

#### 1.3.5.3.1 Approches mixtes

Les efforts menés pour cela consistent soit à ajouter une quantité d'optimisme dans les protocoles conservatifs, soit à limiter l'optimisme des protocoles spéculatifs. Dans la première catégorie, on peut citer les approches de *Dickens* et *Reynolds* [DR91] et de *Steinman* [Ste92] qui calculent les évènements de façon optimiste (i.e., un évènement est calculé même s'il n'est pas sûr), mais les messages potentiellement erronés ne sont pas expédiés avant d'avoir été validés par une exécution sûre.

La seconde catégorie d'approches cherche à réduire le nombre de *rollbacks* et les surcoûts de la sauvegarde des états dans l'exécution optimiste, en évitant à certains PL d'anticiper trop loin dans le futur [SBW88, JRW89, LSW89]. L'utilisation de fenêtres temporelles permet uniquement aux évènements concernés d'être exécutés.

L'approche *Moving Time Window* (*MTW* (Figure 13.a)) [SBW88] autorise l'évaluation des évènements compris entre `t` et `t + F`, où `t` est le temps virtuel global, et `F` la taille de la fenêtre. Dans le même ordre d'idée, le *Bounded Time Warp* (*BTW* (Figure 13.b)) [TX92] de *Turner* et *Xu* découpe la durée de la simulation en intervalles. Un évènement sera exécuté uniquement s'il est dans le même intervalle que le temps virtuel global.

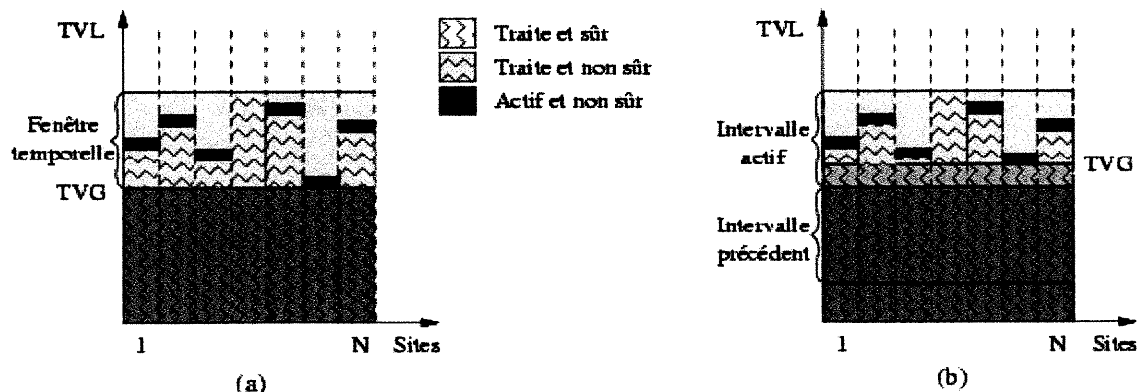


**Figure 13** Aspects temporels des protocoles

*Rajaei* et *al.* proposent dans [RAT93] une troisième approche, appelée *Local Time Warp*. Cette dernière combine les deux grands types de protocoles à deux niveaux distincts. Le modèle à simuler est partitionné en un ensemble de sous-systèmes comportant plusieurs éléments. Chaque composant dans un sous-système est simulé avec le protocole *Time Warp*, alors que les échanges entre les différents sous-systèmes obéissent à un contrôle pessimiste utilisant des fenêtres. Une telle méthode permet de limiter les effets de cascade lors de *rollbacks*, de diminuer les besoins en mémoire pour la sauvegarde des états et de limiter les calculs de réduction à des sous-ensembles du système.

Ces algorithmes à base de fenêtre offrent l'avantage d'être relativement plus simples à implémenter sur les machines de type *SIMD*. Ainsi, des simulations *SIMD* de réseaux d'interconnexion [BA91] et de réseaux commutés ou de centraux téléphoniques [GGN93] ont été menées avec succès.

#### 1.3.5.3.2 Approche adaptative

Comme le choix d'un protocole pour une simulation distribuée particulière reste un problème non trivial, il semble intéressant de retarder le plus possible ce choix, voire d'offrir la possibilité de modifier le mécanisme de synchronisation durant la simulation. Ainsi, ce concept débouche sur des protocoles adaptatifs où le degré d'agressivité ou d'optimisme peut varier, et où différents éléments du modèle simulé peuvent obéir à des protocoles différents.

Une première avancée vers un tel modèle d'exécution est apportée par *COMPOSITE ELSA* [AS92]. Chaque nœud du modèle pourra être évalué soit de façon conservative, soit de façon optimiste. Le type d'exécution est directement fonction des messages en entrée. Si l'on considère un nœud à deux entrées et une sortie :

- Si un message est sûr pour chaque entrée, on peut exécuter l'évènement de façon conservative. Le résultat est sûr et le message résultant également (Figure 14),

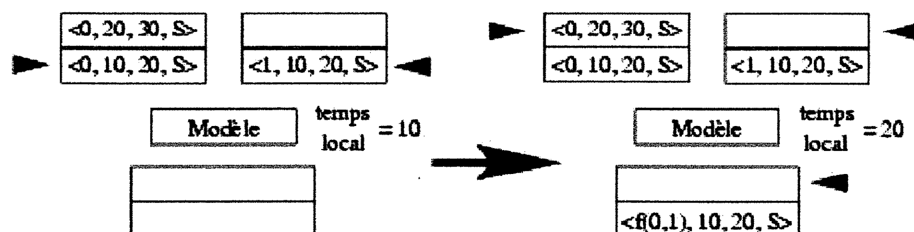


**Figure 14** Evaluation conservative dans *COMPOSITE ELSA*

- Si un seul message est sûr (avec soit une entrée vide, soit un message non sûr), on exécute l'évènement de façon optimiste (Figure 15). En conséquence, le résultat de l'évaluation et le message envoyé ne sont pas sûrs, et on doit conserver une sauvegarde du message pour pouvoir l'annuler si le résultat est faussé par un message en entrée.

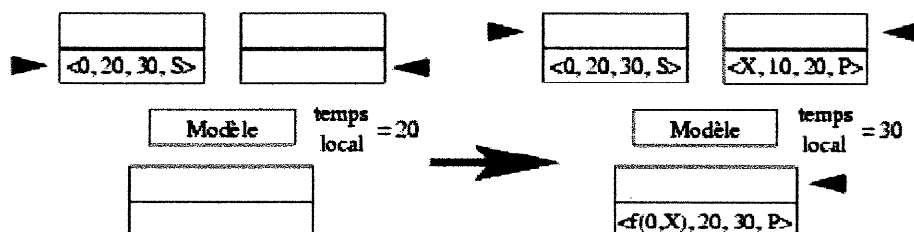


**Figure 15** Evaluation optimiste dans *COMPOSITE ELSA*

Après l'évaluation, on supprime des buffers d'entrée et de sortie tous les évènements antérieurs à la dernière évaluation sûre (i.e., dont toutes les entrées sont sûres). Comme chaque canal d'entrée pour un nœud doit être connu a priori, il apparaît clairement que l'interconnexion entre les composants du modèle simulé doit être définie statiquement.

### 1.3.6 Performances

La présentation des performances obtenues avec les différentes approches exposées tout au long de ce chapitre nécessite un avant-propos important : les aspects quantitatifs que l'on peut trouver dans la littérature montrent pour une grande majorité des accélérations, plus ou moins significatives. Cependant, tous les auteurs convergent vers un point: obtenir de bons résultats est difficile, et dépend de paramètres nombreux et variés (prédictions, granularité du modèle, performances des calculateurs et du réseau d'interconnexion), et parfois inconnus à priori (e.g., les stimuli en entrée du modèle). L'opinion commune est, actuellement, qu'il n'existe pas de solution universelle valable pour tous les domaines de simulation et tous leurs modèles : une simulation synchrone pourra donner de très bons résultats pour beaucoup de modèles, alors que d'autres tireront bénéfice de solutions plus complexes (optimistes ou pessimistes).

Il reste de plus extrêmement difficile de définir, d'après la littérature, si une solution donne toujours de meilleurs résultats, car il n'y a pas de benchmark spécifique à la simulation, ce qui rend les comparaisons entre approches difficiles, voire impossibles.

Dans la suite de cette section, nous essaierons de donner quelques indications sur les performances actuellement obtenues.

#### 1.3.6.1 Performances analytiques

*Lin* montre dans [Lin90] que les approches pessimistes de type *Chandy-Misra* peuvent utiliser beaucoup plus de mémoire qu'un simulateur séquentiel. Les nombres d'interblocages et de messages NULL dans ces approches dépendent uniquement du graphe de précédence des évènements (i.e., ces nombres sont indépendants de paramètres architecturaux, tels que le temps d'exécution des évènements et des délais d'envoi des messages). De telles propriétés simplifient les études de performances. *Lin* montre également que l'approche par détection/résolution des interblocages n'est pas efficace pour la simulation de systèmes sans prévision.

Pour les approches optimistes de type *Time-Warp*, les performances du protocole sont directement affectées par les nombres de messages et calculs erronés (i.e., tous les calculs annulés par l'arrivée d'un message dans le passé de l'exécution locale). *Fleishmann* et *Wilsey* montrent dans [FW95] que la fréquence de la sauvegarde des états revêt également une importance primordiale dans les exécutions optimistes. Cette fréquence est généralement difficile à déterminer a priori, et les résultats exposés montrent que si elle n'est pas constante, mais déterminée dynamiquement, les performances s'améliorent.

En conclusion, l'approche optimiste est démontrée comme surpassant les approches pessimistes dans *presque tous* les cas dans le cadre de simulation de réseaux à rétroaction sans prévision, mais *toujours* dans le cadre de graphes acycliques orientés.

#### 1.3.6.2 Performances expérimentales

##### 1.3.6.2.1 Approches conservatives

*Thomas* et *Zahorjan* proposent dans [TZ91] un protocole spécialisé pour l'étude des réseaux de Petri. De tels modèles, fréquemment utilisés pour les évaluations de performances, utilisent un ensemble de jetons circulant sur un réseau d'étapes. Avec leur protocole, ils mesurent une accélération de 10 sur 12 processeurs d'une machine *Sequent*.

Dans le domaine de la simulation de files d'attente, les performances reportées par *Reed*, *Malony* et *McCredie* sont extrêmement pauvres : L'approche pessimiste ne donne aucune accélération sur 5 processeurs (les temps d'exécution se dégradant). Pour la simulation de 292 véhicules se déplaçant sur 292 routes, un réseau de 33 Transputers atteint un speedup de 19 [MRR90].

Dans son article de référence sur le *Bounded Lag*, *Lubachevsky* simule un modèle composé d'atomes qui changent de spin. À chaque étape, le nouveau spin d'un atome est calculé en fonction de celui de ses voisins. Une accélération de 1900 sur 16 kiloprocesseurs d'une *Connection Machine CM-1* est annoncée [Lub88].

Les deux derniers domaines très souvent étudiés sont la simulation de circuits logiques et de réseaux d'interconnexion. Pour la simulation logique, *Soulé* et *Gupta* annoncent une accélération de 7 à 9 sur 64 processeurs pour l'algorithme distribué s'exécutant sur un seul processeur. Ces chiffres sont ramenés à 5 si l'on considère un simulateur séquentiel [SG91]. Pour *Wood* et *Turner* [WT94], la simulation d'un microprocesseur de 5068 portes logiques ne tire presque jamais de bénéfice d'une exécution distribuée (meilleure accélération de 1,6 sur 9 processeurs). *Bagrodia* et *al.* simulent des circuits du benchmark ISCAS85 [BPH85] en utilisant différentes options de partitionnement des modèles. L'impact sur les performances est important : les temps d'exécution peuvent varier de 1 à 5. Les gains obtenus sur 8 processeurs oscillent entre 2,5 et 3 pour des circuits de quelques milliers de portes [BCJS95].

Dans le cas des réseaux d'interconnexion, l'accélération obtenue dans [AR90] est de 14 sur 15 processeurs. *Cleary* et *Tsai* mettent empiriquement en évidence dans [CT96] que les performances obtenues sont fonction de la densité des messages simulés, ainsi que de la granularité des évènements du modèle. Ainsi, les meilleurs résultats sont obtenus avec un grand nombre de messages simulés et une granularité importante (7,5 sur 8 processeurs). De façon plus réaliste par rapport au système physique, l'approche distribuée conservative est seulement trois (3) fois plus rapide. De la même façon, *Bagrodia* et *al.* montrent dans [BCG+96] que la charge sur le réseau simulé influe directement sur les performances du simulateur. Les améliorations rapportées varient entre 5,3 et 6,6 sur 12 processeurs.

##### 1.3.6.2.2 Approches spéculatives

Une quantité importante de travaux a été menée autour des simulations de combat avec le *Time Warp Operating System* [JBW+87]. La simulation d'un monde de fourmis, de leur comportement pour se nourrir et prévenir leur nid donne une accélération de 13 sur 32 processeurs [EDP+89]. Les mouvements de palets se déplaçant et se rencontrant sur une surface sans frottement sont également étudiés et leur simulation sur 32 nœuds est accélérée par un facteur 12 [HBD+89]. Enfin, la simulation d'un combat opposant deux armées composées de divisions blindées est étudiée. Elle se compose de trois phases principales : la mise en place des troupes, le

combat sur le champ de bataille, et la fin des hostilités. L'utilisation de 48 processeurs permet une exécution 12 fois plus rapide que dans le cas séquentiel [WHF+89].

Dans le contexte de la simulation logique, *Briner* rapporte des accélérations allant jusqu'à 25 sur une machine *BBN Butterfly* de 32 nœuds [Bri91], tandis que *Bagrodia* et *al.* obtiennent uniquement un facteur proche de 2 sur 8 processeurs [BCJS95]. Les études sur les réseaux d'interconnexion ont été également menées avec des approches optimistes. Les gains sont plus faibles que dans les cas conservatifs : [CT96] rapporte des accélérations de 3 à 3,9 sur 12 processeurs, alors que [BCG+96] ne met en évidence aucune amélioration sur 8 nœuds.

### 1.3.7 Conclusion

L'apparition d'architectures parallèles a revêtu un aspect particulièrement important dans le domaine de la simulation où les besoins en puissance de traitement restent très importants. Cependant, nous venons de voir que la simulation distribuée à évènements discrets est une tâche qui n'est pas triviale. Même si le système que l'on cherche à simuler contient fréquemment du parallélisme intrinsèque, sa modélisation pour une exécution distribuée est un problème ardu.

Le choix d'une approche de simulation distribuée est une des clefs des problèmes rencontrés par la diffusion du principe général. Les différentes approches introduites présentent chacune des côtés positifs, et les performances obtenues sont encourageantes.

Cependant, lors du choix de la méthode de synchronisation de l'algorithme parallèle, le concepteur du simulateur doit vérifier que le système à étudier, et son modèle, vérifient bien des propriétés imposées par le protocole. Du point de vue de l'utilisateur, il est possible que certains des choix faits lors de la modélisation compliquent la simulation parallèle : les informations concernant la prédiction, les temps de service minimum sont des facteurs du modèle qui influent sur les performances du protocole. Il ressort ainsi une particularité de la simulation

distribuée : la définition du modèle à simuler et le protocole de *PDES* ne sont pas indépendants.

Cette interdépendance entre les parties « noyau de simulation » et « modèle à simuler » a limité jusqu'à présent l'attrait des utilisateurs pour les approches distribuées: il est en effet actuellement incontournable de maîtriser le moteur de simulation utilisé afin de pouvoir tirer des bénéfices de la distribution lors de la simulation d'un système.

## 1.4 Le moteur de simulation de SystemC

Actuellement, *SystemC* ne fournit pas de ressources permettant de réaliser une simulation distribuée. Son moteur de simulation (ou ordonnanceur) utilise un algorithme *piloté par évènement* (*event-driven*) comme un simulateur *RTL* classique. Une liste de sensibilité aux évènements est associée à chaque composant.

La tâche de l'ordonnanceur est de déterminer l'ordre d'exécution des processus de la conception en se basant sur la sensibilité aux évènements des processus et sur les notifications d'évènement qui arrivent. L'ordonnanceur *SystemC* supporte les deux types d'orientation de la modélisation, logicielle et matérielle.

Semblablement à *VHDL* et *Verilog*, l'ordonnanceur *SystemC* supporte les *cycles delta*. Un *cycle delta* est constitué de deux phases différentes que sont la *phase d'évaluation* et la *phase de mise à jour* (*evaluate and update*). De multiples *cycles delta* peuvent avoir lieu à un moment précis de la simulation. Les *cycles delta* sont utiles pour la modélisation de traitements entièrement distribués et synchronisés comme on en trouve par exemple dans le matériel *RTL*. Dans *SystemC*, l'utilisation de l'appel `notify()` avec un argument de temps égal à zéro cause la notification de l'évènement dans la phase d'évaluation du *cycle delta* suivant, tandis qu'un appel à `request_update()` cause l'appel de la méthode `update()` dans la phase d'évaluation du *cycle delta* actuel. Par l'utilisation de ces propriétés, des canaux qui modélisent le comportement des signaux hardware, peuvent être construits.

*SystemC* supporte aussi les *notifications datées d'évènements* (*notified events*). Ces notifications sont spécifiées en employant `notifiy()` avec un argument de type temps. Une notification datée cause la notification de l'évènement spécifié au moment indiqué dans l'avenir par l'argument de temps. Les notifications datées sont présentes et dans *VHDL* et dans *Verilog* et sont aussi utiles dans la modélisation de systèmes logiciels.

Finalement, *SystemC* supporte aussi les notifications immédiates d'évènement, qui sont spécifiées en employant `notify()` sans argument. Une notification immédiate d'évènement permet de rendre les processus sensibles à cet évènement immédiatement prêts à être exécutés (c'est-à-dire prêt à s'exécuter dans la phase d'évaluation courante.) Ce type de notification est utile pour la modélisation de systèmes logiciels et de systèmes d'exploitation, qui n'emploient pas le concept de *cycles delta*.

Les étapes suivantes décrivent le fonctionnement de l'ordonnanceur *SystemC*. Du pseudo code plus détaillé est inclus en annexe.

1) **Phase d'initialisation** : Exécuter tous les processus (excepté les `SC_CTHREAD`) dans un ordre non spécifié d'avance.

2) **Phase d'évaluation** : Sélectionner un processus en attente d'exécution et le démarrer. Ceci peut causer l'apparition de notifications immédiates d'évènements, avec pour conséquence un ajout de processus prêts à s'exécuter dans cette même phase.

3) S'il existe de tels processus en attente d'exécution, retourner à l'étape 2.

4) **Phase de mise à jour** : Exécuter tous les appels en attente à `update()`, résultant des appels à `request_update()` faits à l'étape 2.

5) S'il existe des *notifications différées* (*delayed notifications*) en attente, déterminer quels sont les processus prêts à s'exécuter du fait de ces notifications différées et aller à l'étape 2.

6) S'il ne reste plus de *notification datée* (*timed notification*), la simulation est alors terminée.

7) Avancer le temps actuel de la simulation jusqu'à la plus proche date des *notifications datées* en attente.

8) Déterminer quels sont les processus prêts à s'exécuter du fait des évènements qui possèdent des *notifications datées* avec comme date, la date courante. Aller à l'étape 2.

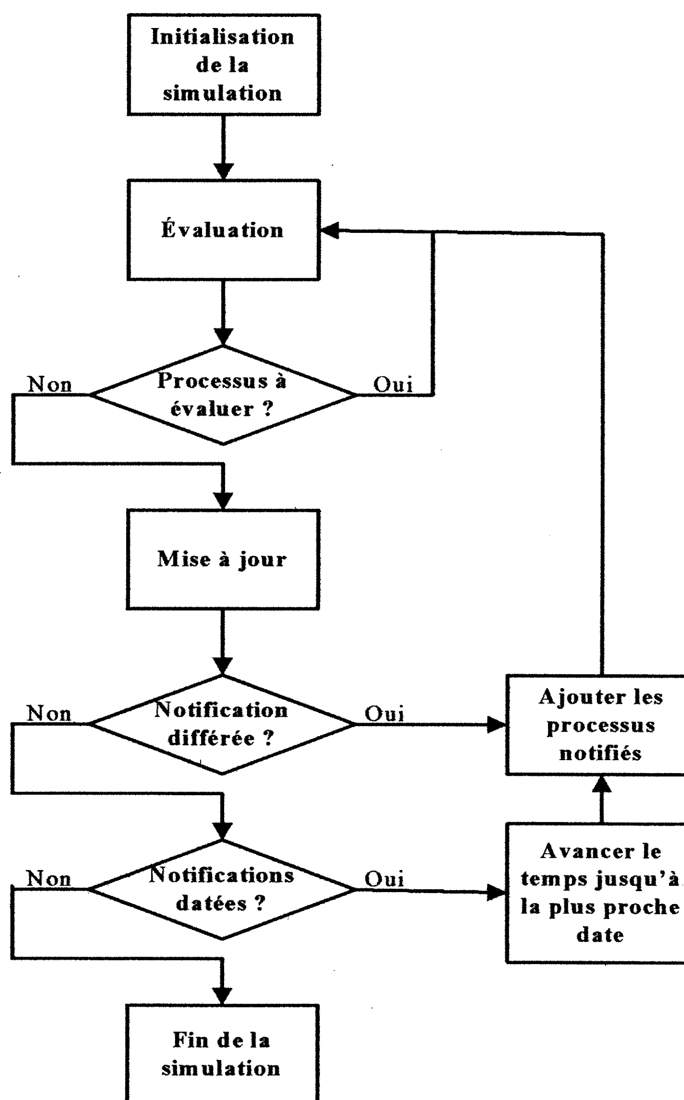


**Figure 16** Organigramme de l'ordonnanceur *SystemC*

# CHAPITRE 2 : MÉTHODOLOGIE

Pour les besoins de notre expérimentation nous avons d'abord développé, avec *SystemC*, un modèle de processeur *DLX*. Ce modèle a été ensuite enrichi grâce l'héritage orienté objet pour aboutir à un système multiprocesseur en forme d'anneau. Enfin, pour distribuer notre modèle sur plusieurs machines nous avons développé, grâce aux nouvelles fonctionnalités introduites par *SystemC 2.0*, un canal *TCP/IP* que nous avons implanté au lieu et place des canaux élémentaires de type `sc_signal` utilisés dans l'anneau de la version non distribuée.

## 2.1 Le modèle du *DLX*

Le *DLX* [HePat95] possède un pipeline de cinq étages qui sont : `IF` (*Instruction Fetch*), `ID` (*Instruction Decode*), `EX` (*Execution*), `MEM` (*Memory*) et `WB` (*Write Back*). Chaque étage a été modélisé par un processus *SystemC* de type `SC_METHOD`. Le premier étage est responsable de la récupération des instructions *DLX* localisées dans la *ROM* (mémoire de programme), le second s'occupe de la sélection des registres opérandes, du décodage de l'instruction et de l'évaluation de la condition de branchement. Le troisième étage se charge aussi bien des instructions arithmétiques et logiques que du calcul de l'adressage mémoire. L'accès à la *RAM* (mémoire de données) est effectué par l'étage `MEM`. Si nécessaire, le résultat est finalement sauvegardé dans le registre destination par l'étage `WB`.

La Figure 17 est un diagramme de classes du modèle *DLX* monoprocesseur. Ce diagramme montre les relations structurales (ou architecturales) entre les différents modules du modèle. Ce diagramme sera, plus tard, enrichi, entre autres par héritage objet, pour obtenir le modèle multiprocesseur.

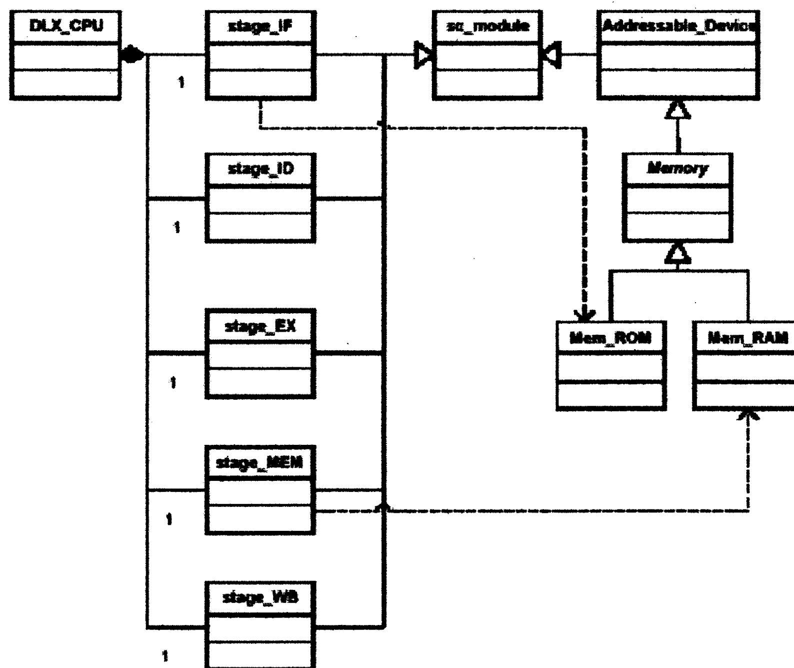


**Figure 17** Diagramme de classes du modèle *DLX*

Pour exemple, voici le code d'implantation de l'étage IF :

```
SC_MODULE(stage_IF)
{
      sc_in_clk clk;                    // l'horloge
      sc_in<int> NPC;                   // compteur d'instruction dans
                                        // le cas d'un branchement
      sc_in<bool> ID_EX_cond;           // la condition de branchement
      sc_in<DData> data;
      sc_inout<int> PC;                 // le compteur d'instruction
      sc_inout<bool> io_request;        // requête de lecture en mémoire
      sc_inout<bool> stalled;
      sc_out<sc_uint<32> > IR;          // vers registre d'instruction
      sc_out<int> op;
      sc_out<bool> write_enable;
      sc_out<DAddr> address;

      int addr;

      void process_IF_request(void){
            if (!stalled)
            {
                  addr = PC.read();
                  address = addr;
                  io_request = true;
            }
      }
```

```
        void process_IF_forward(void) {
              IR.write(data.read());
              op.write(data.read().range(31, 26));
              //par défaut, sauter à l'adresse suivante
              process_update_PC();

        }

        void process_update_PC(void) {
              //mettre à jour le PC quand cond change...
              if (ID_EX_cond.read()) PC.write(NPC.read());
              else PC.write(addr + 4);
        }

        // Le constructeur
        SC_CTOR( stage_IF )     {
              SC_METHOD (process_IF_request);
              sensitive_pos << clk;
              SC_METHOD (process_IF_forward);
              sensitive_neg << io_request;
              SC_METHOD(process_update_PC);
              sensitive << ID_EX_cond;
        }
};
```

## 2.2 Le modèle multiprocesseur

Le modèle multiprocesseur utilisé consiste en un anneau de processeurs *DLX*. A chaque cycle d'horloge, un nœud de l'anneau échange de l'information avec son proche voisin via l'interface `addressable_Device`. L'échange sur l'anneau est unidirectionnel et ne peut être bloqué (Figure 18).

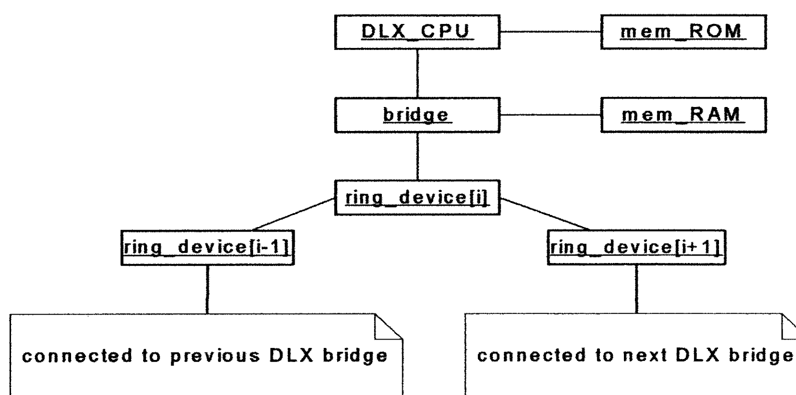


**Figure 18** Architecture de l'anneau de *DLX*

La première chose à faire par un $\mathrm{DLX}_i$ pour envoyer un message à un $\mathrm{DLX}_j$ est d'interroger le registre d'état (*status register*) de son `RingDevice` en attendant la

disponibilité de ses registres de transmission (*transmit registers*). Quand ces derniers deviennent disponibles, la prochaine étape pour le $DLX_i$ est d'écrire le message et son adresse de destination (qui est dans ce cas j) dans les registres de transmission appropriés.

Les messages circulant déjà sur l'anneau sont prioritaires par rapport à ceux nouvellement introduits par les processeurs *DLX* et ce, afin d'éviter tout risque d'interblocage (en supposant que toutes les adresses soient valides). La livraison de chaque message est garantie au bout de n sauts au maximum car il ne peut y avoir plus de n messages sur l'anneau en même temps. Dès qu'un emplacement est disponible sur l'anneau le nouveau message est transféré des registres du RingDevice courant vers le prochain RingDevice dans la boucle. Quand un message atteint sa destination, il est conservé dans les registres de réception du RingDevice de destination (*RingDevice receive registers*) de sorte que le $DLX_j$ qui était entrain d'interroger son RingDevice puisse le lire et ce faisant, libère un « espace » permettant ainsi l'acceptation d'un nouveau message sur l'anneau.

Pour les besoins de conception du modèle multiprocesseur, des modules supplémentaires ont été développés par héritage de l'interface addressable_device. Ces nouveaux modules sont :

- Bridge (Figure 20) : Permet le partage de l'intervalle d'adressage (Figure 19) entre plusieurs modules dérivés de l'interface addressable_device (mem_RAM et ring_device dans notre cas). Deux processus de type SC_METHOD permettent de diriger les requêtes de l'extérieur (en provenance du *CPU*) vers un des modules internes (mem_RAM ou ring_device) et la réponse de ce dernier vers l'extérieur. Contrairement à la version monoprocesseur, le module *RAM* n'est pas directement relié au *CPU* mais est attaché avec un module *I/O* de type ring_device au module Bridge. Quand le *CPU* fait une requête de lecture ou d'écriture, celle-ci est prise en charge par le module Bridge qui

en interprétant correctement l'adresse concernée la dirige vers le bon module et retourne ensuite le résultat au *CPU*.

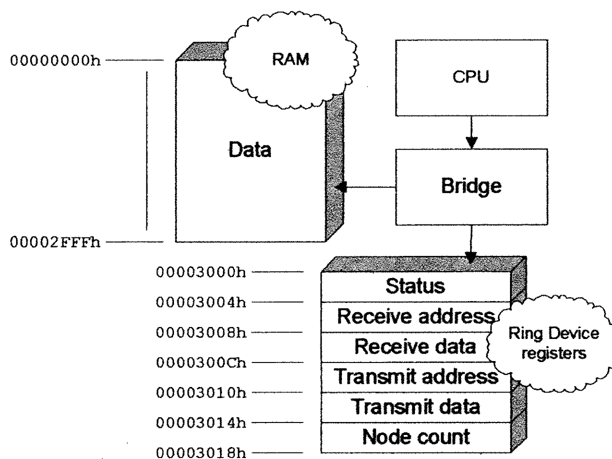


**Figure 19** Intervalle d'adressage

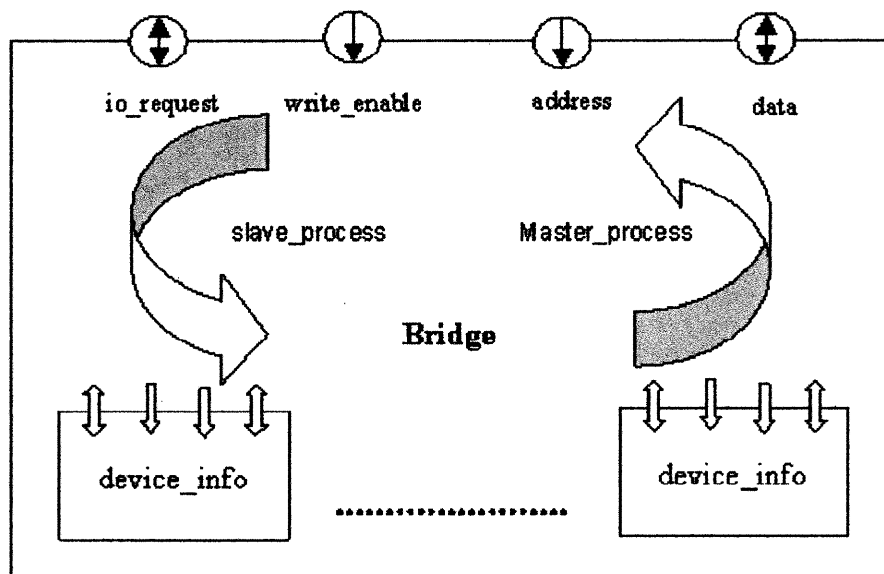


**Figure 20** Le module `Bridge`

- `Device_info` (Figure 21) : Grâce à ses signaux, permet de connecter les ports de modules dérivés de l'interface `addressable_device` (`mem_RAM` et `ring_device` dans ce cas) aux ports du module `Bridge` qui les contient.

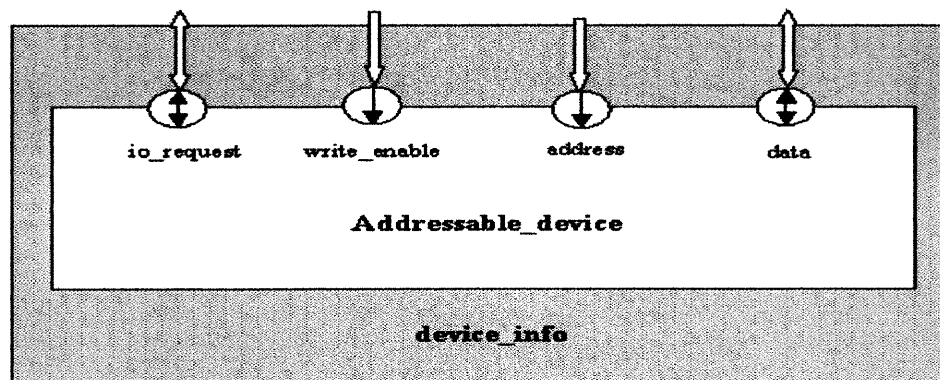


**Figure 21** Le module `device_info`

- `Ring_Device` (Figure 22): Module d'entrée/sortie qui est, avec le module `RingStuff` et le module *`RingManager`*, l'élément constitutif principal de l'anneau. Il est constitué de six registres, trois pour la réception des données et trois pour leur émission. Les données sont constituées par la valeur à lire ou à écrire, son adresse et une valeur logique pour sa validation. Il est cadencé par une horloge de type `sc_in_clk`. Deux processus de type `SC_METHOD` contrôlent les cotés *CPU* et *RING* du `Ring_Device`. Le processus responsable du contrôle coté *CPU* est sensitif à la valeur positive du signal `io_request` alors que l'autre est sensitif à la valeur positive de l'horloge de cadencement.

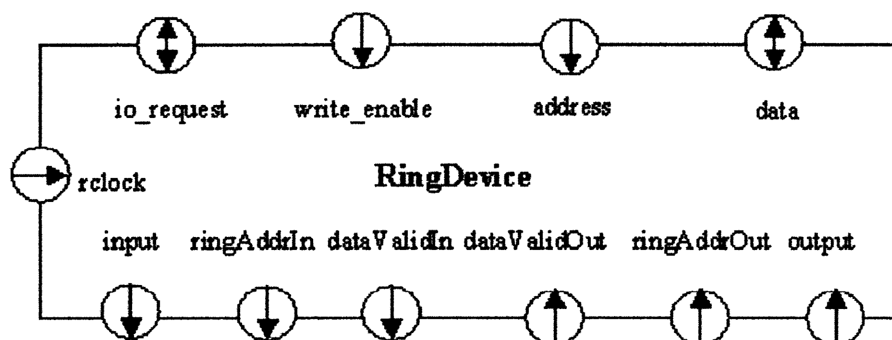


**Figure 22** Le module `RingDevice`

Pour pouvoir connecter les `Ring_Device` entre eux, nous avons développé le module `RingStuff` (Figure 23) qui, grâce à ses trois signaux, permet l'interconnexion des trois ports de sortie d'un `Ring_Device` avec les trois ports d'entrée du `Ring_Device` suivant dans l'anneau. Ces modules `RingStuff` sont alors regroupés et connectés entre eux dans le module `RingManager` (Figure 24). On retrouve tous ces modules dans le diagramme de classes du modèle multiprocesseur illustré par la Figure 25.

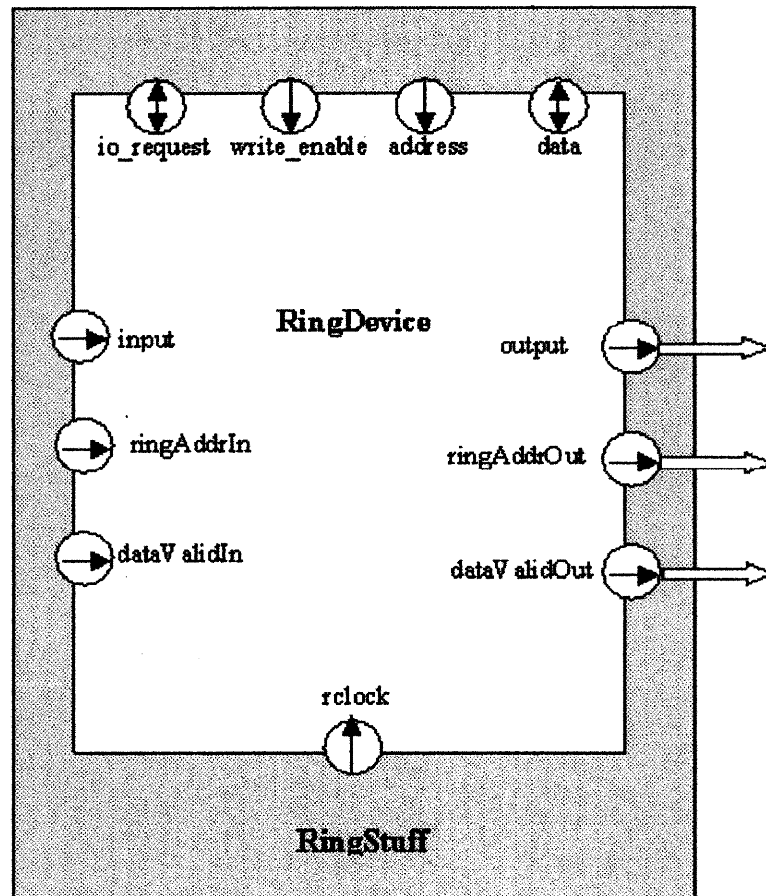


**Figure 23** Le module `RingStuff`

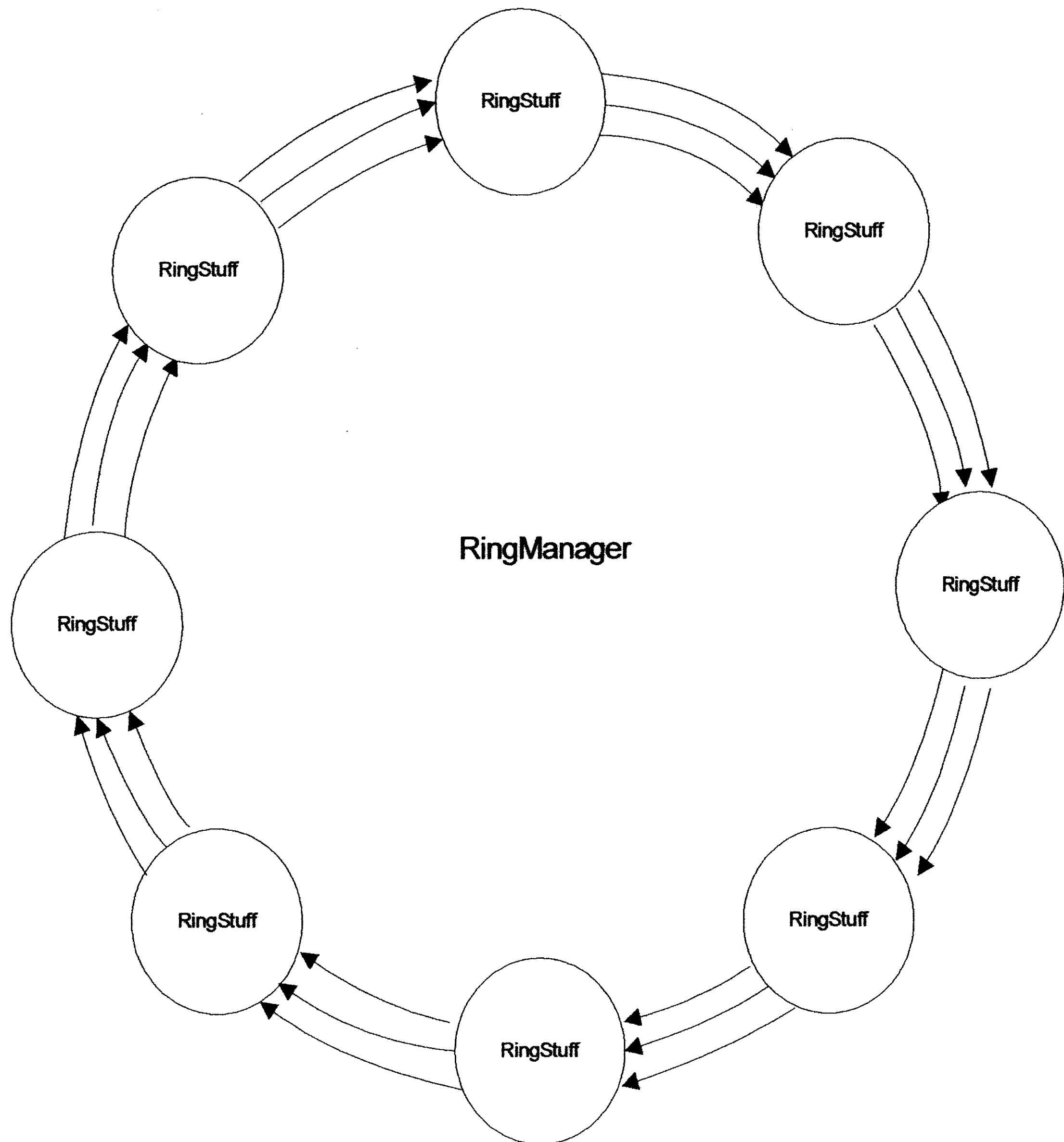


**Figure 24** Le module `RingManager`

Comme exemple de code, voici les déclarations de `RingStuff` et `RingManager` :

```
class RingStuff {
  public:
    RingDevice rd;
    sc_signal<DData> rData;
    sc_signal<DData>   rAddr;
    sc_signal<bool>  dataValid;
    RingStuff(const char *name) : rd(name) { ; }
};
```

```
class RingManager {
      unsigned int numPorts;
      RingStuff **rsp;
  public:
      Addressable_Device *getRingDevice(unsigned int port);
      // Constructeur de la version non distribuée
      RingManager(unsigned int nDevice, sc_clock &clk);
      // Constructeur de la version distribuée
      RingManager(........);
};
```

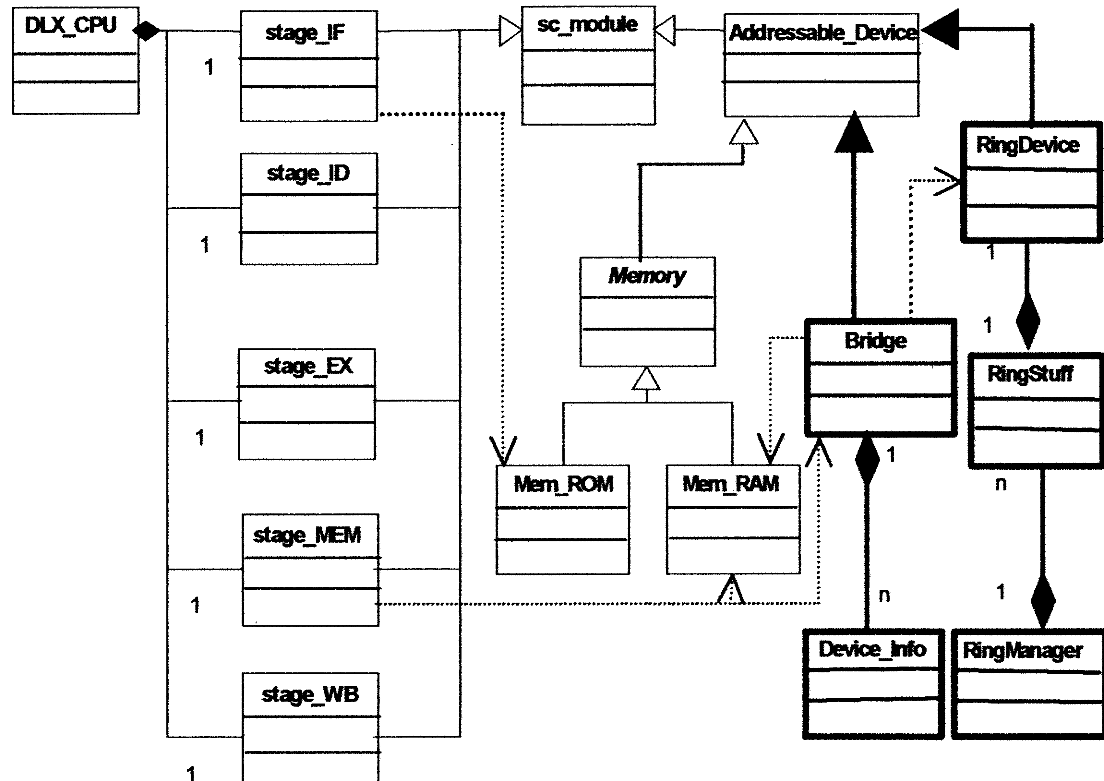


**Figure 25** Diagramme de classe du modèle multiprocesseur

## 2.3 Le modèle multiprocesseur distribué par extension du canal élémentaire de communication *sc_signal*: Canal natif de communication TCP/IP

### 2.3.1 Un exemple : simulation d'un protocole simple de communication

*Simplex* est un protocole de communication utilisé pour transférer des données d'un module vers un autre dans une seule direction. *Simplex* peut détecter les erreurs de

transmission et retransmettre les paquets de données pour compléter avec succès le transfert des données si des erreurs sont détectées [UGSy11].

Le modèle de base consiste en un émetteur couplé avec un réveil (*Timer*), d'un récepteur couplé avec un dispositif d'affichage (*display*) et, en version de simulation non distribuée, d'un module représentant le média de communication (*Channel*) (Figure 26).

Dans le cas où notre objectif serait de simuler le protocole de communication et non le média, nous constatons déjà qu'une version de simulation distribuée nous évite de recourir à une modélisation inutile du média de communication et nous permet ainsi de simuler en conditions réelles (média physique).

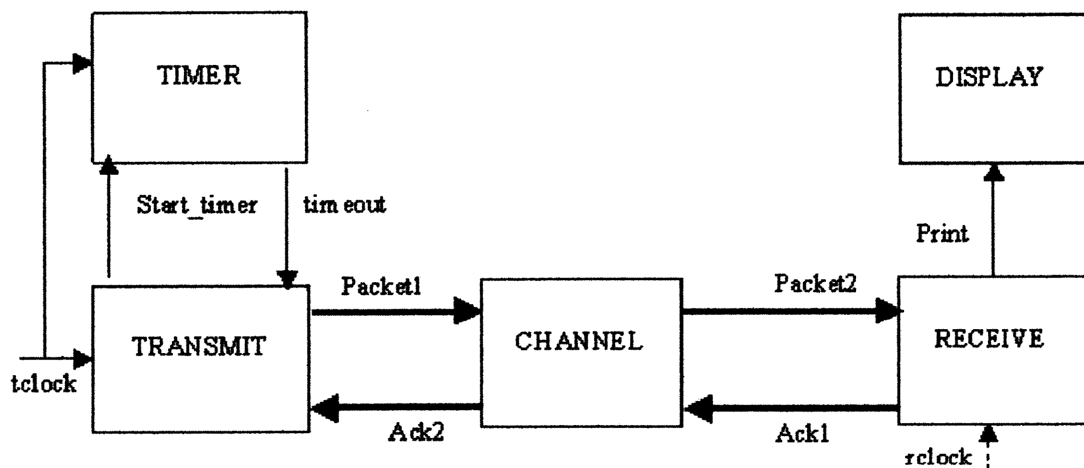


**Figure 26** *Simplex* : Exemple d'architecture non distribuée

En version non distribuée, les cinq modules (`TRANSMIT`, `TIMER`, `CHANNEL`, `RECEIVE` et `DISPLAY`), les sept signaux (`Packet1`, `Packet2`, `Ack1`, `Ack2`, `Start_Timer`, `Timeout` et `Print`) ainsi que les deux horloges (`tclock` et `rclock`) sont tous implémentés sur une seule et même machine.

L'émetteur envoie donc les données au canal de communication qui les transmet au récepteur. Le canal peut introduire des erreurs dans les données pour modéliser le taux d'erreur actuel du canal physique.

Le récepteur reçoit les données du canal de communication et les analyse pour détecter d'éventuelles erreurs. Si les données sont correctes, le récepteur génère un acquittement et l'envoie vers le canal qui le transmettra à l'émetteur. Pour simplifier les choses on considère que le canal retransmet sans erreur l'acquittement. Une fois que l'émetteur reçoit l'acquittement du paquet qui vient juste d'être transmis, il envoie le paquet suivant. Dans le cas où un acquittement tarderait à arriver, le réveil (`TIMER`) déclenche un `timeout` qui causera la retransmission du paquet non encore acquitté. Ce processus continue jusqu'à l'émission complète des données. Voici un exemple de code du programme principal (pour un code détaillé voir [UGSy11]) :

```
#include "packet.h"
#include "timer.h"
#include "transmit.h"
#include "receiver.h"
#include "display.h"
#include "channel.h"
int sc_main(int argc, char* argv[]) {
      // packet_type est un type défini par l'utilisateur
      // représentant l'entité transmise (le message) sur le
      // canal.
      // Déclaration des différents signaux
      sc_signal<packet_type> Packet1, Packet2, Ack1, Ack2;
      sc_signal<int> Print;
      sc_signal<bool> Timeout, Start;
      // Déclaration de l'horloge de transmission de période
      // 20 unités de temps
      sc_clock tclock("clock", 20);
      // Déclaration de l'horloge de réception de période
      // 15 unités de temps
      sc_clock     rclock("rclk", 15);
      TRANSMIT t1("transmit"); // Déclaration du module émetteur
      t1.tpackout(Packet1); // Connexion du signal Packet1 en sortie
      t1.tpackin(Ack2); // Connexion du signal Ack2 en entrée
      t1.timeout(Timeout); // Connexion du signal Timeout en entrée
      t1.start_timer(Start); // Connexion du signal Start en sortie
      t1.clock(tclock); // Connexion de l'horloge d'émission tclock

      CHANNEL c1("channel"); // Déclaration du module CHANNEL
      c1.tpackin(Packet1); // Connexion du signal Packet1 en entrée
      c1.rpackout(Packet2); // Connexion du signal Packet2 en sortie
      c1.rpackin(Ack1); // Connexion du signal Ack1en entrée
      c1.tpackout(Ack2); // Connexion du signal Ack2en sortie

      RECEIVER r1("receiver"); // Déclaration du module récepteur
      r1.rpackin(Packet2); // Connexion du signal Packet2 en entrée
      r1.rpackout(Ack1); // Connexion du signal Ack1en sortie
      r1.dout(Print); // Connexion du signal Print en sortie
      r1.clock(rclock); // Connexion de l'horloge de réception
```

```
    DISPLAY d1("display"); // Déclaration du module DISPLAY
    d1 << Print; // Autre façon de connecter un signal en entrée

    TIMER tm1("timer"); // Déclaration du module TIMER
    // Façon élégante et rapide pour connecter plusieurs signaux
    // en entrée !
    tm1 << Start << Timeout << tclock.signal();

    sc_start(50); // On simule pour 50 unités de temps !
    return(0);
}
```

L'exécution de ce programme nous donne la sortie illustrée par la Figure 27.

```
bash - Konsole <3>
Fichier Sessions Configuration Aide
Transmit: sending packet: (1804289383...1...0)
Transmit: start_timer
Channel: Received packet seq no. = 1
Receiver: got packet: (0...0...0)
Timer: timer start detected
Receiver: got packet: (1804289383...1...0)
Receiver: sending ack for packet 1
Display: Data Value Received, Data = 1804289383
Transmit: receiving Ack for packet 1
Transmit: sending packet: (1714636915...2...0)
Transmit: start_timer
Channel: Received packet seq no. = 2
Timer: timer start detected
Receiver: got packet: (1714636915...2...0)
Receiver: sending ack for packet 2
Display: Data Value Received, Data = 1714636915
Transmit: receiving Ack for packet 2
Transmit: sending packet: (719885386...3...0)
Transmit: start_timer
Channel: Received packet seq no. = 3
Timer: timer start detected
Receiver: got packet: (719885386...3...0)
Receiver: sending ack for packet 3
Display: Data Value Received, Data = 719885386
Transmit: receiving Ack for packet 3
Transmit: sending packet: (1189641421...4...0)
Transmit: start_timer
Channel: Received packet seq no. = 4
Timer: timer start detected

Appuyez sur Entrée pour continuer !
Nouveau   bash
```

**Figure 27** Sortie du programme de simulation non distribuée

### 2.3.2 Extension du canal de communication : *Simplex* version distribuée (Figure 28)

Un exemple d'architecture d'une simulation distribuée serait d'avoir l'émetteur (TRANSMIT) et le réveil (TIMER) implantés sur une machine alors que le récepteur (RECEIVER) et l'afficheur (DISPLAY) le seraient sur une autre, la modélisation du canal de communication (CHANNEL) devenant dans ce cas inutile (Figure 29):

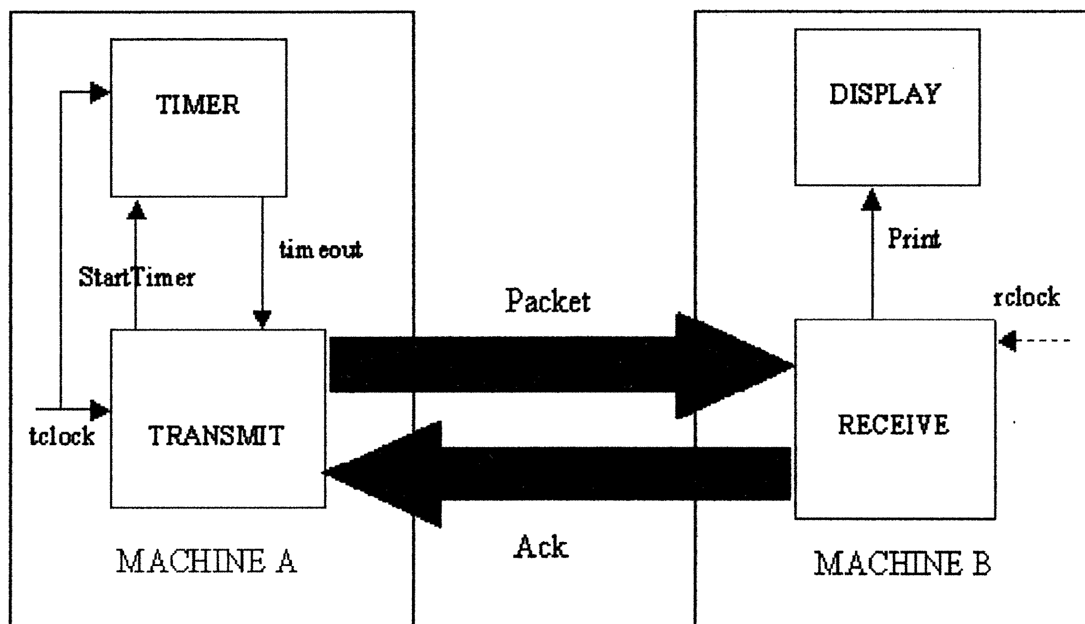


**Figure 28** *Simplex* : Exemple de simulation distribuée

`tcp_client_packet1`, `tcp_server_packet2`, `tcp_client_ack1` et `tcp_server_ack2` sont des canaux *SystemC* (`sc_channel`) fournissant des services *TCP/IP* pour le transfert de données.

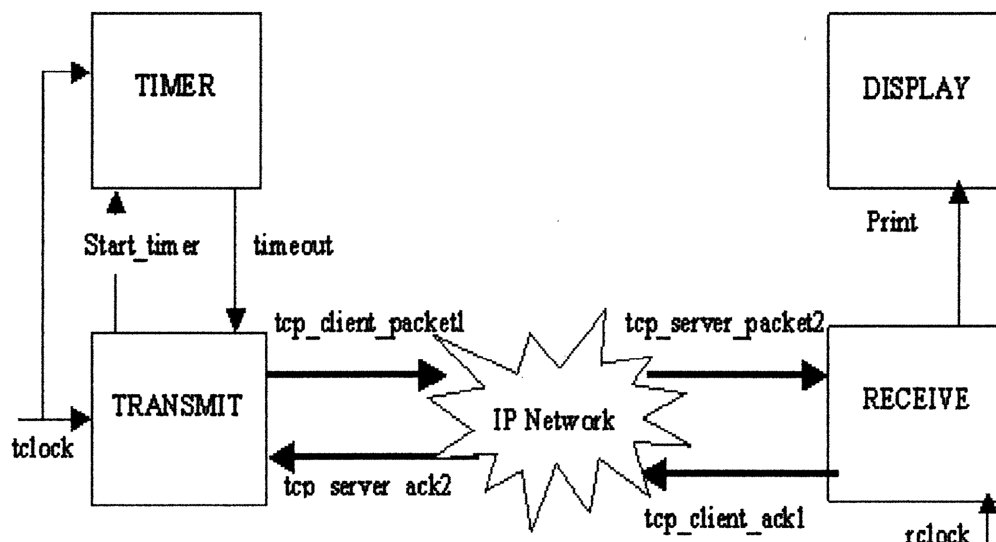


**Figure 29** Exemple d'architecture distribuée avec canal serveur et canal client

Un client *TCP/IP* peut être vu comme un port fournissant une interface du type « *write only* » permettant de transférer des données vers un autre module distant. Un exemple d'architecture serait un port *SystemC* (`sc_port`) fournissant l'interface désirée, associé à un canal *SystemC* (`sc_channel`) qui encapsulerait tout le code nécessaire à la fourniture du service *TCP/IP* (dans ce cas, service « *send* » uniquement) (Figure 30).

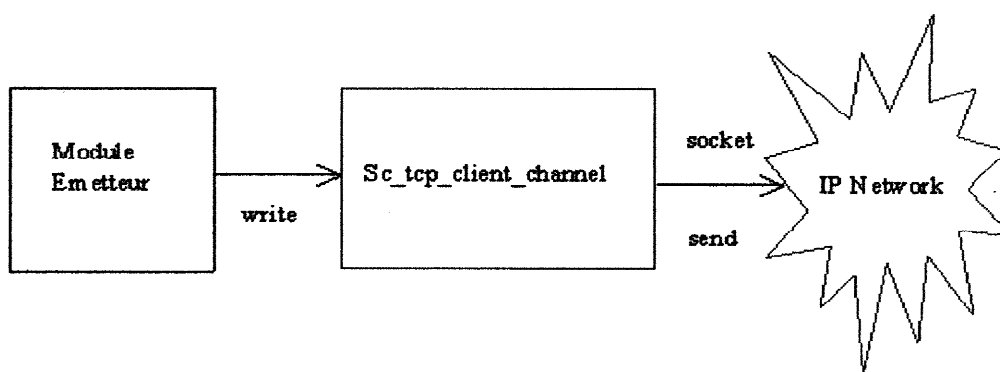


**Figure 30** Exemple d'architecture d'un canal client *TCP/IP*

Parmi les architectures possibles d'un serveur *SystemC TCP/IP*, l'une d'elles, simple, serait celle de la Figure 31.

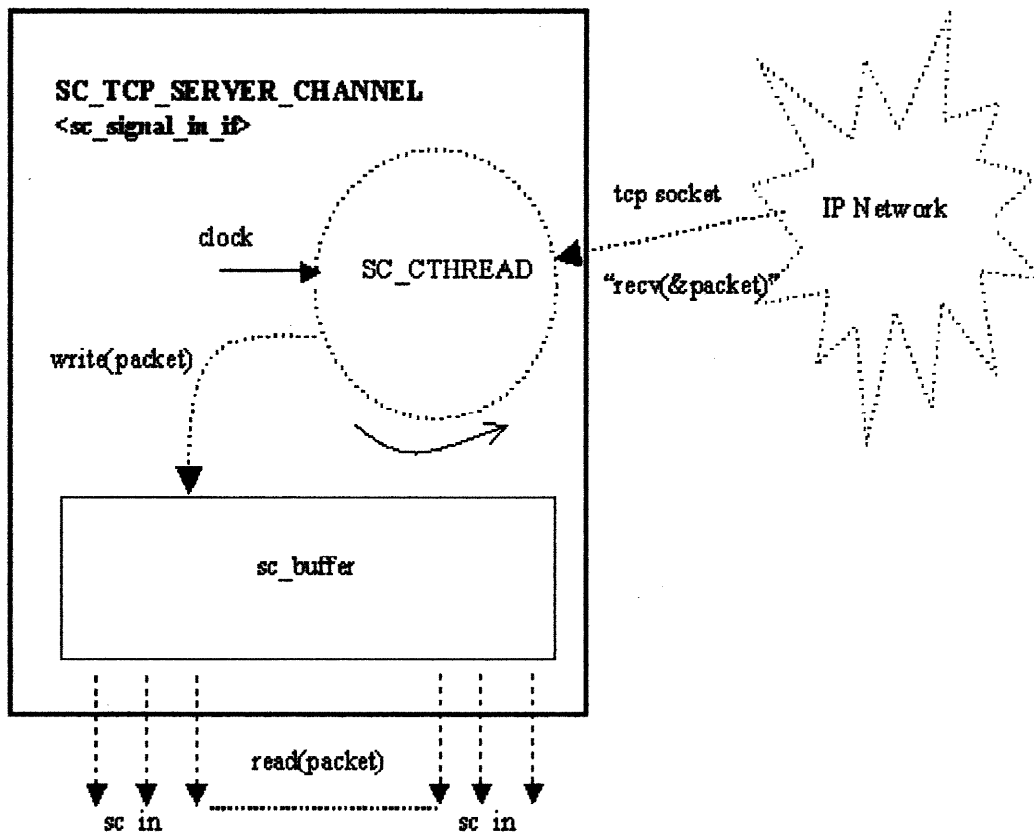


**Figure 31** Exemple d'architecture d'un canal serveur *TCP/IP*

Ce canal est de type « composé » (*composite channels*). Il est composé de deux membres privés (*private attributes*), l'un de type `sc_buffer` (canal interne) fournissant l'interface `sc_signal_in_if` permettant à des ports externes de s'y connecter afin de lire la donnée qui vient juste d'y être « écrite » par l'autre membre qui est de type `SC_CTHREAD` ( processus interne) et qui « écoute », à chaque top de l'horloge `clock`, le socket *TCP/IP* connectée sur un module *SystemC* distant. Cette horloge est accessible de l'extérieur du canal (*public attribute*) et peut donc être initialisée par le programmeur.

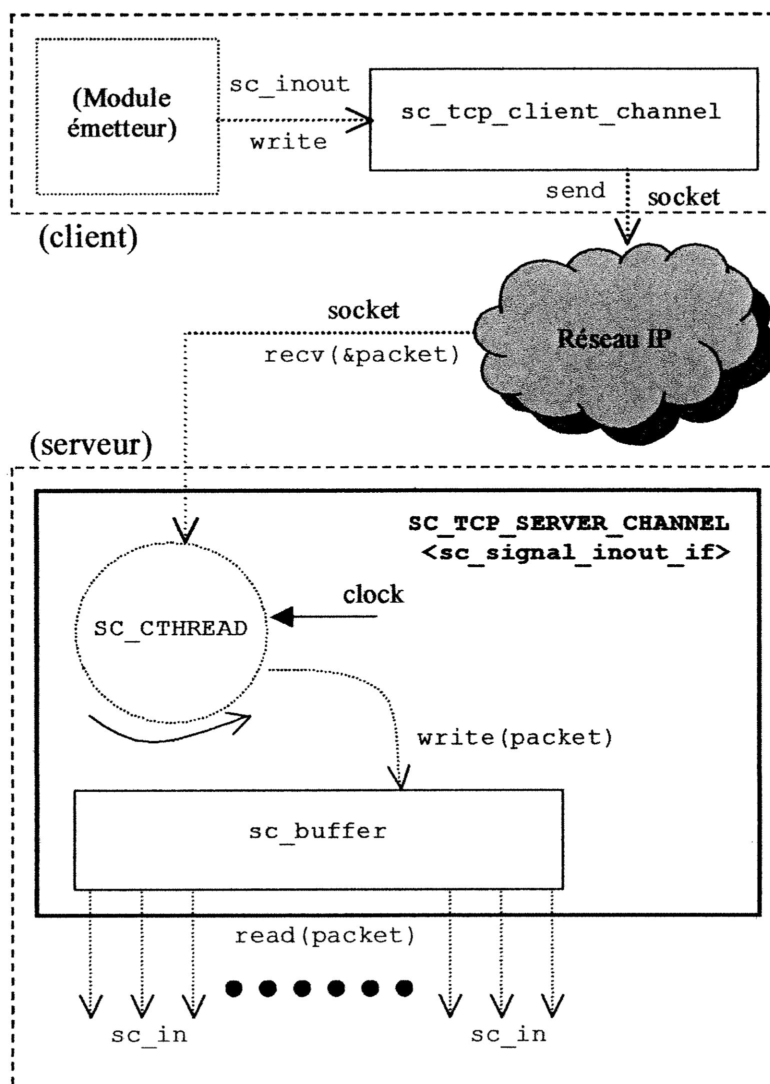


**Figure 32** Extension *TCP/IP* du canal primaire de communication

Dans le cas d'une modélisation d'un module émettant et recevant des données vers et en provenance d'un même autre module distant (comme c'est le cas des modules *RECEIVE* et *TRANSMIT*), il est préférable d'utiliser un seul socket pour l'émission et la réception des données sur le réseau. Dans ce cas, on peut utiliser un canal client/serveur comme celui illustré par la Figure 33. L'architecture globale du système est alors équivalente à celle de la Figure 34.

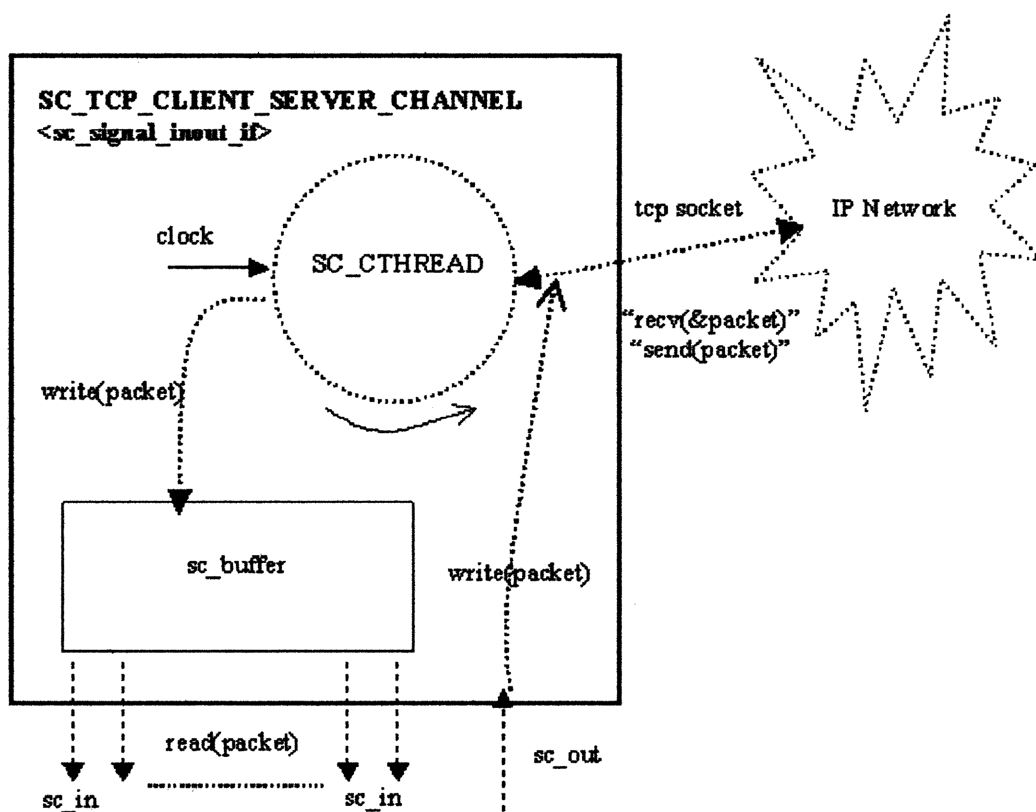


**Figure 33** Exemple d'architecture d'un canal client/serveur *TCP/IP*

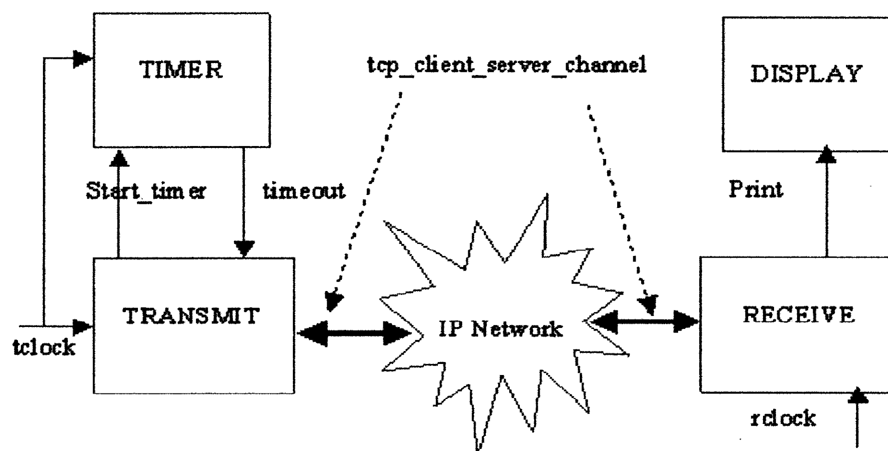


**Figure 34** *Simplex* : Architecture distribuée avec canal *TCP/IP* client/serveur

Les programmes principaux d'une telle version distribuée avec canal client/serveur seraient les suivants:

**Sur la machine A (émetteur), initiatrice ( client ) de la connexion *TCP/IP*:**

```
int main(int argc, char *argv[ ] ) {

      // Déclaration de l'horloge du canal TCP/IP
      sc_clock channel_clk("channel_clock", 15);

      // Déclaration de l'horloge du module émetteur
      sc_clock tclk("clock",20);

      // Déclaration du canal TCP/IP avec :
      // port local = 7000
      // adresse IP locale = "172.16.243.2"
      // port distant = 5000, adresse ip distante = 172.16.243.1
      // Entre autres, grâce au dernier argument de valeur false le
      // constructeur du canal va initier la connection TCP/IP
      sc_tcp_client_server_channel<packet_type>
      IP_CLIENT_SERVER_CHANNEL ( "IPClientServerChannel", 7000,
      5000, "172.16.243.1", false);

      // Connection de l'horloge
      IP_CLIENT_SERVER_CHANNEL.clock(channel_clk);

      // Déclaration des signaux utilisés par le côté émetteur
      sc_signal<bool> TIMEOUT, START;

      // Déclaration du module TIMER
      timer tm1("timer");

      // Connexion des signaux et de l'horloge d'émission au TIMER
      tm1 << START << TIMEOUT << tclk.signal();

      // Déclaration du module émetteur
      transmit t1("transmit");

      // Connexion de l'horloge d'émission
      t1.clock(tclk);

      // Connexion des canaux TCP/IP et des signaux locaux
      t1.tpackout(IP_CLIENT_SERVER_CHANNEL);
      t1.tpackin(IP_CLIENT_SERVER_CHANNEL);
      t1.timeout(TIMEOUT);
      t1.start_timer(START);

      sc_start(50); // On y va pour 50 unités de temps !

      return 0;
}
```

Sur la machine B (récepteur) :

```
int main(int argc, char *argv[])
{
	// Entre autres, le constructeur du canal initiera une écoute
	// sur un socket
	// Port local = 5000, adresse IP locale = "172.16.243.1"
	// Port distant = 7000, adresse IP distante = "172.16.243.2"
	sc_tcp_client_server_channel<packet_type>
	IP_CLIENT_SERVER_CHANNEL ("IPClientServerChannel", 5000, 7000,
	"172.16.243.2");

	// Déclaration et connexion de l'horloge du canal
	sc_clock channel_clk ("channel_clock", 15);
	IP_CLIENT_SERVER_CHANNEL.clock(channel_clk);

	// Déclaration du signal local de visualisation
	sc_signal<int> Print;

	// Déclaration de l'horloge de réception
	sc_clock rclock("clock", 15);

	// Déclaration du module récepteur
	receiver r1("receiver");

	// Connexion de l'horloge de réception
	r1.clock(rclock);

	// Connexion du canal TCP/IP
	r1.rpackin(IP_CLIENT_SERVER_CHANNEL);
	r1.rpackout(IP_CLIENT_SERVER_CHANNEL);

	// Connexion du signal sortant de visualisation
	r1.dout(Print);

	// Déclaration du module de visualisation
	display d1("display");

	// Connexion du signal entrant de visualisation
	d1.din(PRINT);

	sc_start(50); // On y va pour 50 unités de temps !

	return 0;
}
```

On constate alors que le code principal reste pratiquement le même par rapport à la version non distribuée ; il a seulement été « partagé » entre les deux machines. Les modules qui communiquent à distance sont « connectés » par l'intermédiaire du canal *TCP/IP* client/serveur.

```
bash - Konsole <2>
Fichier Sessions Configuration Aide
connect on: 172.16.243.1
Transmit: sending packet: (1804289383...1...0)
Transmit: start_timer
Timer: timer start detected
Transmit: received Ack for packet 1
Transmit: sending packet: (846930886...2...0)
Transmit: start_timer
Timer: timer start detected
Transmit: received Ack for packet 2
Transmit: sending packet: (1681692777...3...0)
Transmit: start_timer
Timer: timer start detected
Transmit: received Ack for packet 3
Transmit: sending packet: (1714636915...4...0)
Transmit: start_timer
Timer: timer start detected

Appuyez sur Entrée pour continuer !
Nouveau  bash
```

**Figure 35** *Simplex* : Exemple de sortie du programme « émetteur »

```
bash - Konsole
Fichier Sessions Configuration Aide
Attente de connexion sur: 0.0.0.0:5000
connexion de 172.16.243.1
Receiver: got packet: (1804289383...1...0)
Receiver: send ack for packet 1
Display: Data Value Received, Data = 1804289383
Receiver: got packet: (846930886...2...0)
Receiver: send ack for packet 2
Display: Data Value Received, Data = 846930886
Receiver: got packet: (1681692777...3...0)
Receiver: send ack for packet 3
Display: Data Value Received, Data = 1681692777

Appuyez sur Entrée pour continuer !
Nouveau  bash
```

**Figure 36** *Simplex* : Exemple de sortie du programme « récepteur »

Cet exemple, par sa simplicité, a surtout servi à présenter quelques exemples d'architectures de modélisation d'un canal *SystemC* utilisant le protocole *TCP/IP* pour une communication distante inter modules.

Pour que ce type de simulation distribuée soit « faisable », il faut disposer d'un modèle à simuler dont les éléments puissent être distribués, de façon équilibrée, sur un processeur réseau. Dans tous les cas, il faudrait que parmi les connexions inter modules du modèle, il y en ait au moins une au niveau de laquelle le traitement global puisse être partagé. Dans ce cas, ces connexions seraient utilisées pour distribuer le modèle sur deux ou plusieurs machines distantes qui se partageraient donc la simulation, ce qui logiquement se traduirait par une augmentation de la performance. On a alors tout intérêt à choisir des voies de communication réseau très performantes pour éliminer les problèmes de désynchronisation entre les différents noyaux de simulation et éviter ainsi d'être obligé d'implémenter des protocoles de synchronisation.

# CHAPITRE 3 : EXPÉRIMENTATION, PRÉSENTATION ET ANALYSE DES RÉSULTATS

## 3.1 Utilisation d'une machine Linux à 450 MHz [CAPP02]

Le modèle fut d'abord construit en utilisant *SystemC 1.2.1*, puis fût porté vers la nouvelle version 2.0. Les deux versions de *SystemC* furent évaluées. Les étages du pipeline furent d'abord modélisés en utilisant des SC_METHOD et par la suite, des SC_THREAD. La simulation du modèle monoprocesseur *DLX* fut faite pendant 249 026 cycles (Tableau II).

**Tableau II** Performance du modèle *DLX* monoprocesseur

| Version de SystemC | Temps (sec) | Fréquence (kHz) |
|---|---|---|
| 1.2.1 (SC_METHOD) | 3,29 | 75,69 |
| 1.2.1 (SC_THREAD) | 4,59 | 54,25 |
| 2.0 (SC_METHOD) | 6,17 | 40,36 |
| 2.0 (SC_ THREAD) | 8,32 | 29,93 |

Le modèle le plus rapide s'est exécuté à une fréquence très respectable de 76 kHz. Ce même modèle n'a malheureusement pas pu aller au-delà de 40 kHz avec *SystemC 2.0*, et ce, sur la même machine à 450 MHz (Figure 37).

Différents modèles multiprocesseur non distribués (l'anneau est, dans sa totalité, implémenté sur une seule machine ) de 2 à 128 *DLX* furent ensuite simulés.

Le Tableau III et le Tableau IV montrent les résultats obtenus, où n indique le nombre de processeurs *DLX*. Comme précédemment, les étages du pipeline furent d'abord modélisés par des SC_METHOD puis par des SC_THREAD.

**Tableau III** Temps de simulation du multiprocesseur non distribué

| *n* | SystemC 1.2.1 | | SystemC 2.0 | |
|---|---|---|---|---|
| | SC_METHOD | SC_THREAD | SC_METHOD | SC_THREAD |
| 2 | 8,76 | 12,86 | 18,06 | 23,96 |
| 4 | 19,56 | 28,35 | 36,62 | 46,31 |
| 8 | 46,07 | 65,24 | 74,43 | 95,92 |
| 16 | 105,98 | 144,10 | 153,15 | 204,77 |
| 32 | 230,01 | 327,43 | 343,04 | 484,84 |
| 40 | 296,80 | 432,08 | 465,36 | 656,94 |
| 64 | 524,72 | 804,61 | 868,12 | 1239,99 |
| 96 | 875,36 | 1403,87 | 1504,14 | 2085,94 |
| 128 | 1235,98 | 2010,82 | 2063,81 | 2796,77 |

**Tableau IV** Fréquence combinée du multiprocesseur non distribué

| n | SystemC 1.2.1 | | SystemC 2.0 | |
|---|---|---|---|---|
| | SC_METHOD (kHz) | SC_THREAD (kHz) | SC_METHOD (kHz) | SC_THREAD (kHz) |
| 2 | 73,09 | 49,78 | 35,44 | 26,71 |
| 4 | 65,43 | 45,15 | 34,95 | 27,64 |
| 8 | 55,57 | 39,24 | 34,39 | 26,69 |
| 16 | 48,31 | 35,53 | 33,43 | 25,00 |
| 32 | 44,52 | 31,27 | 29,85 | 21,12 |
| 40 | 43,13 | 29,62 | 27,51 | 19,48 |
| 64 | 39,03 | 25,45 | 23,59 | 16,52 |
| 96 | 35,09 | 21,88 | 20,42 | 14,73 |
| 128 | 33,14 | 20,37 | 19,85 | 14,65 |

L'usage des *SC_THREAD* résulta en un accroissement d'environ 48 % du temps d'exécution dans le cas de la version 1.2 et d'environ 36 % dans le cas de la version 2.0. L'usage de *SystemC 2.0* au lieu de *SystemC 1.2.1* résulta en un accroissement d'environ 60 % du temps d'exécution (Tableau III).

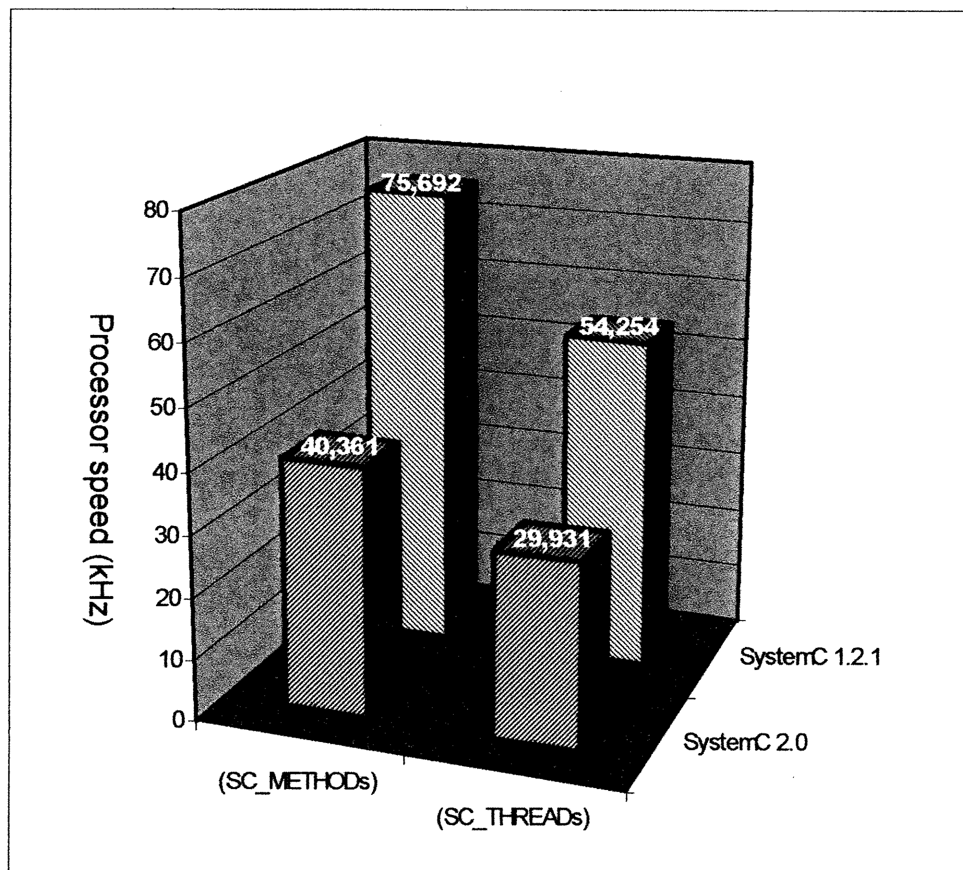


**Figure 37** `SystemC` 1.2 vs. 2.0 : Performance d'un *DLX* monoprocesseur

Les résultats obtenus (Figure 38 et Figure 39 (temps de simulation en fonction du nombre de *DLX*)) indiquent donc que les nouvelles fonctionnalités introduites par *SystemC 2.0* entraînent une baisse relativement importante de la performance. L'écart de performance introduit par l'utilisation des `SC_THREAD` au lieu des `SC_METHOD` est moins grand même s'il reste relativement important. Ceci nous amène à penser que pour faire en sorte que l'introduction de ces nouvelles fonctionnalités, dont l'intérêt pour la conception ne fait aucun doute, ne se fasse pas au détriment de la performance de la simulation, nous avons tout intérêt à utiliser des techniques de simulation qui apportent un gain de performance. La réponse est alors toute trouvée : la simulation distribuée.

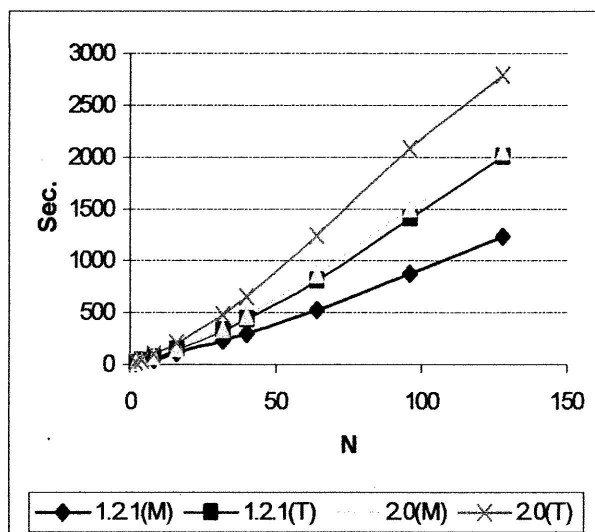


**Figure 38** Courbes des temps de simulation du multiprocesseur non distribué

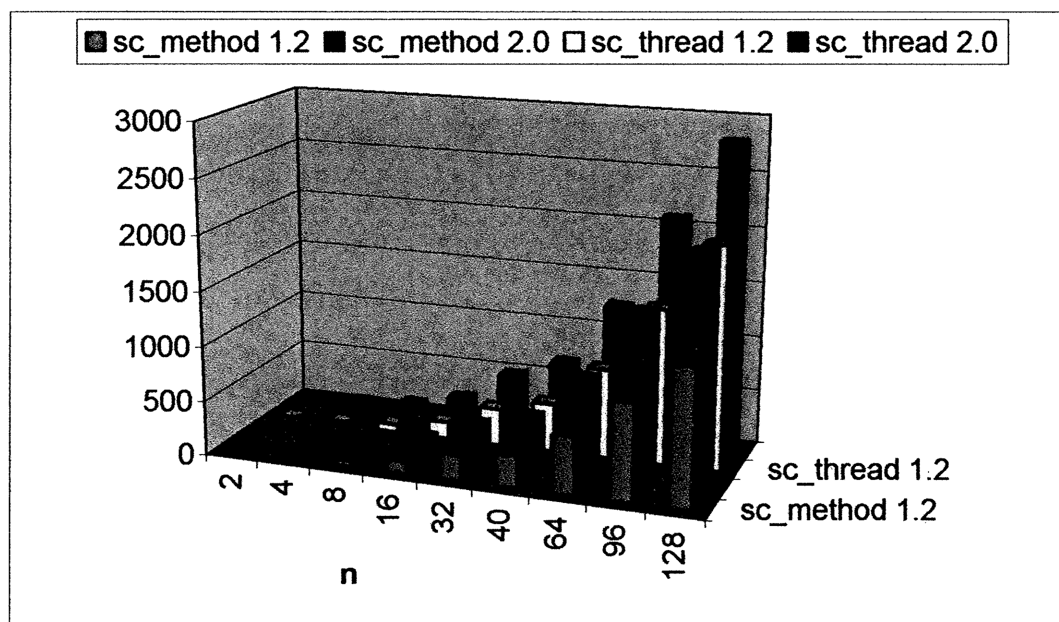


**Figure 39** Chute de performance introduite par la version 2.0

### 3.2 Simulation distribuée : Utilisation d'un processeur réseau de 32 machines Linux à 1,2 GHz connectées à travers un commutateur Ethernet 1 Gbit

Les expérimentations précédentes ont été refaites en utilisant un processeur réseau constitué d'un maximum de 32 nœuds. Sur chaque nœud on exécuta alors un noyau de simulation *SystemC* dans le but de simuler globalement un anneau de 2 à 128 processeurs *DLX*. Chaque noyau communique avec son proche voisin par le biais de trois sockets *TCP/IP*. Les processeurs *DLX* qui se trouvent sur le même nœud communiquent eux par canaux élémentaires (*sc_signal*). La durée de la simulation fût de 100 000 cycles.

Le Tableau V donne les temps de simulation du modèle multiprocesseur en fonction du nombre de *DLX* ( `n` ), du nombre de nœuds du processeur réseau (2, 4, 8, 16 et 32) et du type de processus *SystemC* utilisé (`M` pour `SC_METHOD` et `T` pour `SC_THREAD`). La dernière colonne de la table donne les temps de simulation du modèle multiprocesseur non distribué (simulation de l'anneau sur une seule machine Linux à 1.2 GHz).

**Tableau V** Temps de simulation du modèle multiprocesseur distribué

| | 32 | | 16 | | 8 | | 4 | | 2 | | 1 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| n | M | T | M | T | M | T | M | T | M | T | M | T |
| 128 | 6 | 7 | 10 | 13 | 24 | 44 | 96 | 150 | 267 | 348 | 576 | 717 |
| 64 | 3 | 4 | 5 | 6 | 10 | 13 | 24 | 45 | 95 | 150 | 254 | 333 |
| 32 | 2 | 2 | 3 | 4 | 5 | 6 | 10 | 13 | 23 | 42 | 85 | 142 |
| 16 | | | 2 | 2 | 3 | 4 | 5 | 6 | 10 | 13 | 23 | 40 |
| 8 | | | | | 2 | 2 | 3 | 3 | 5 | 6 | 9 | 10 |
| 4 | | | | | | | 1 | 1 | 3 | 3 | 4 | 4 |
| 2 | | | | | | | | | 1 | 1 | 1 | 2 |

Avant d'analyser la performance, on peut déjà faire une première constatation, même si son intérêt est moindre : Si on considère la Figure 40 construite à partir du Tableau V et donnant la différence des temps de la simulation entre les versions `SC_METHOD` et `SC_THREAD`, on constate que quand la complexité du modèle croît (quand `n` croît), cet écart, qui tend à croître rapidement, est fortement aplani et ce,

jusqu'à, pratiquement, son annulation, par une augmentation de la distribution (augmentation du nombre de nœuds du processeur réseau).

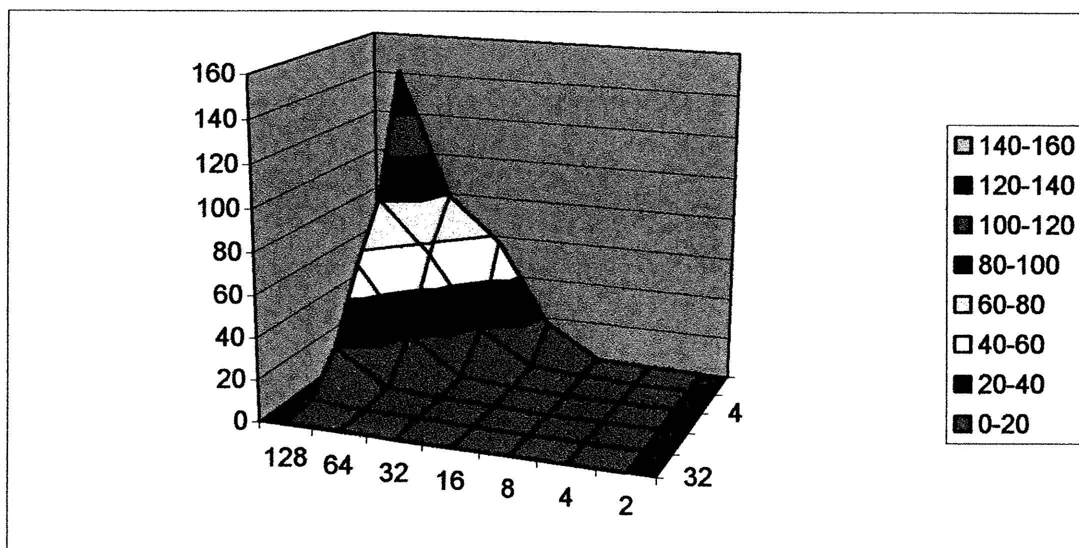


**Figure 40** Différence des temps de simulation : `SC_THREAD` et `SC_METHOD`

Analysons maintenant l'effet de la distribution sur la performance.

Considérons, par exemple, la simulation avec les `SC_THREAD` (Tableau VI). On voit alors qu'on peut simuler 100 000 cycles d'un anneau de 128 *DLX* en seulement 7,18 secondes sur un processeur réseau de 32 nœuds en comparaison aux 717 secondes obtenues sur un seul nœud (*speedup* de 100) !

**Tableau VI** Temps de simulation des `SC_THREAD` du modèle distribué

| | # nœuds du proc. réseau (résultat en sec. pour 100 000 cycles) | | | | | |
|---|---|---|---|---|---|---|
| # DLX | 32 | 16 | 8 | 4 | 2 | 1 |
| 128 | 7,18 | 13,00 | 44,04 | 149,25 | 348,16 | 717,00 |
| 64 | 3,78 | 6,48 | 13,00 | 44,83 | 150,33 | 333,00 |
| 32 | 2,06 | 3,56 | 6,00 | 12,66 | 42,33 | 142,00 |
| 16 | | 2,00 | 3,91 | 6,00 | 13,17 | 40,00 |
| 8 | | | 2,00 | 3,25 | 6,00 | 10,00 |
| 4 | | | | 1,50 | 3,00 | 4,00 |
| 2 | | | | | 1,00 | 2,00 |

La Figure 41 illustre graphiquement ces résultats et le Tableau VII montre qu'on peut obtenir une fréquence « combinée » de simulation de 1,78 MHz en utilisant un processeur réseau de 32 machines. Cette fréquence « combinée » prend en considération le travail réalisé par tous les *DLX* de l'anneau.

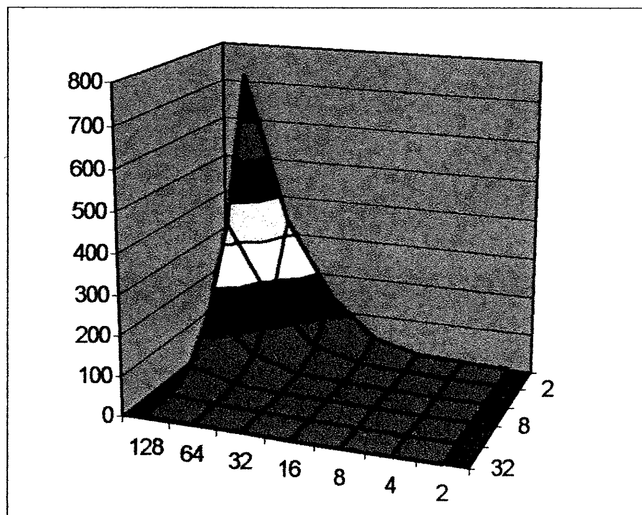


**Figure 41** Temps de simulation des `SC_THREAD` du modèle distribué

**Tableau VII** Fréquence « combinée » du modèle multiprocesseur distribué

| | Nb d'UCT dans l'anneau (résultat en kHz) | | | | | |
|---|---|---|---|---|---|---|
| #DLXs | 32 | 16 | 8 | 4 | 2 | 1 |
| 128 | 1783 | 985 | 291 | 86 | 37 | 18 |
| 64 | 1693 | 988 | 492 | 143 | 43 | 19 |
| 32 | 1553 | 899 | 533 | 253 | 76 | 23 |
| 16 | | 800 | 409 | 267 | 121 | 40 |
| 8 | | | 400 | 246 | 133 | 80 |
| 4 | | | | 267 | 133 | 100 |
| 2 | | | | | 200 | 100 |

Nous obtenons ainsi une *accélération superlinéaire* due au fait de la répartition « quasi-naturelle » du modèle sur le processeur réseau (Figure 42).

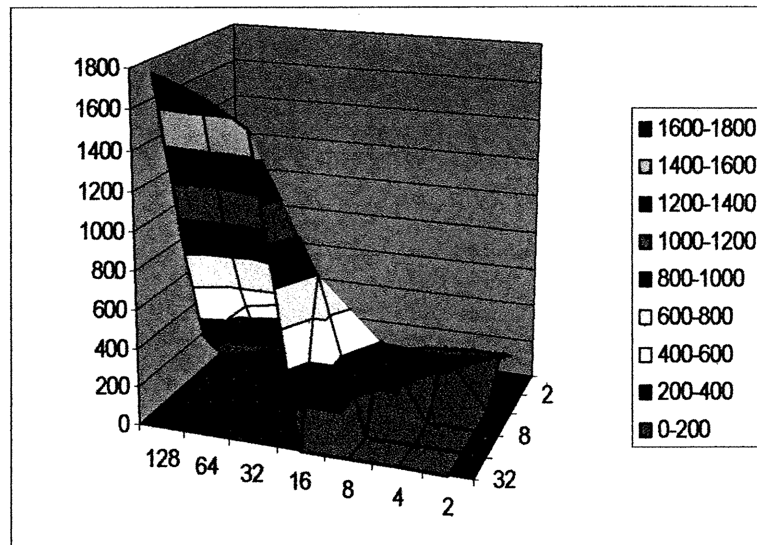


**Figure 42** Accélération *superlinéaire* (kHz) obtenue par la distribution

Dans le cadre de notre expérimentation, ces résultats mettent clairement en évidence la capacité de la simulation distribuée à améliorer significativement la performance de la simulation de modèles complexes.

**Canal natif vs MPI**

Les temps de simulation obtenus avec la version *MPI* du programme de test sont fournis par le Tableau VIII. Ces résultats sont confrontés avec leurs homologues du Tableau V par la Figure 43 (sur la figure, les versions du programme avec canal natif sont préfixées avec `SYS` et celles correspondant à *MPI* le sont avec `MPI`. Ensuite, vient soit la lettre `M` pour `SC_METHOD` soit la lettre `T` pour `SC_THREAD`. Finalement, on trouve le nombre de nœuds utilisés par le processeur réseau. Par exemple : `SYS_M16` veut dire: version *SystemC*, donc canal natif utilisé, modélisation avec des `SC_METHOD` et simulation sur un processeur réseau de 16 nœuds. Par contre, `MPI_T16` veut dire : version *MPI* du canal, modélisation avec des `SC_THREAD` et simulation sur un processeur réseau de 16 nœuds ).

On constate, que l'overhead introduit par l'utilisation de *MPI*, par rapport à la version distribuée « native », augmente avec la distribution du modèle sur le processeur réseau. On passe d'un overhead pratiquement nul avec 128 *DLX* distribués

sur un processeur réseau de 2 nœuds à un overhead de plus de 400 % avec les mêmes 128 *DLX* mais, cette fois-ci, distribués sur un processeur réseau de 16 nœuds !

Il est ainsi clair que dans le cadre de notre expérimentation la version avec canal natif est bien plus performante que celle utilisant la librairie *MPI.*

**Tableau VIII** Temps de simulation avec *MPI*

| n | 16 | | 8 | | 4 | | 2 | |
|---|---|---|---|---|---|---|---|---|
| | M | T | M | T | M | T | M | T |
| 128 | 57 | 71 | 77 | 147 | 168 | 288 | 255 | 352 |
| 64 | 32 | 37 | 41 | 54 | 44 | 85 | 88 | 147 |
| 32 | 23 | 25 | 26 | 30 | 24 | 26 | 23 | 42 |
| 16 | 13 | 15 | 21 | 23 | 16 | 20 | 12 | 17 |
| 8 | | | 18 | 18 | 14 | 18 | 9 | 11 |
| 4 | | | | | 12 | 12 | 9 | 11 |
| 2 | | | | | | | 9 | 9 |

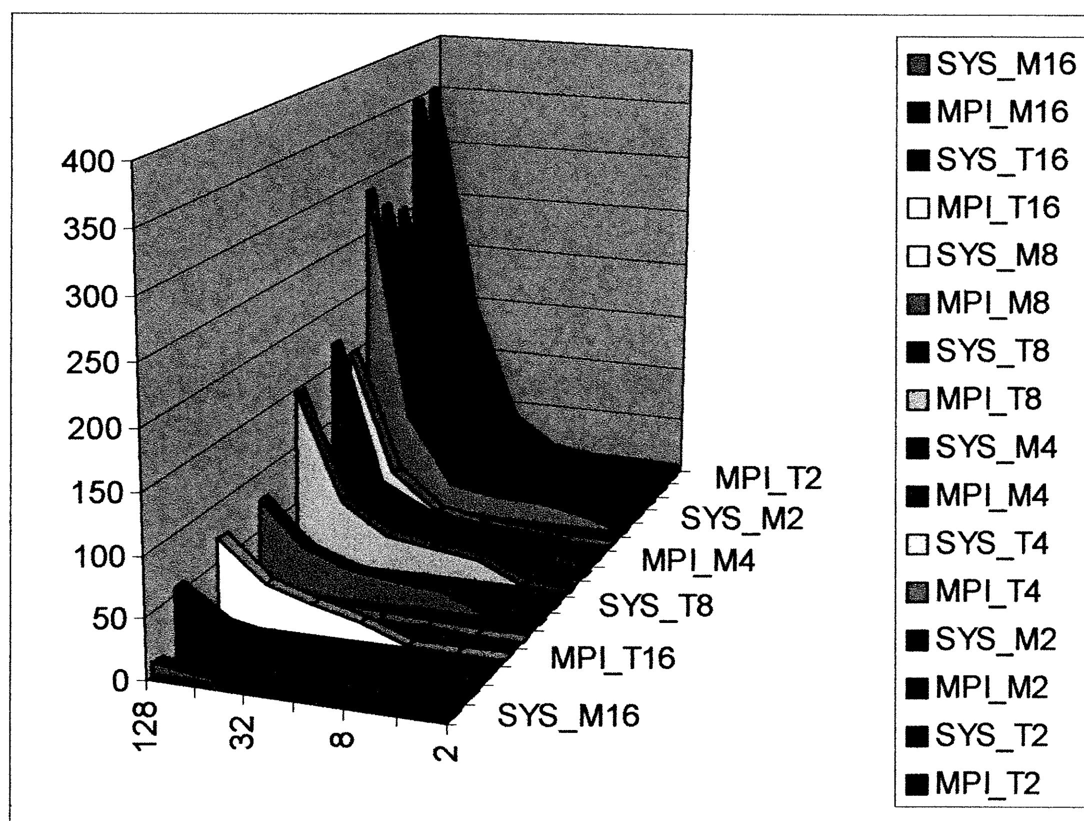


**Figure 43** Canal natif vs *MPI*

# DISCUSSION GÉNÉRALE ET CONCLUSION

Nous avons montré, tout au long du chapitre précédent, que *SystemC* est capable de soutenir la simulation d'un modèle de plus en plus complexe et qu'il demeure donc efficace pour la modélisation de systèmes dont la complexité est comparable, par exemple, à celle d'un système multiprocesseur. Par contre, nous avons vu que la version 2.0 de *SystemC*, malgré l'introduction de nouvelles nécessaires et efficaces fonctionnalités de conception, cause une baisse relativement importante de la performance de la simulation. Heureusement, cette nouvelle version met à notre disposition de puissants outils de conception (canaux, interfaces et évènements) qui nous ont permis, dans le cadre de notre expérimentation, de répondre à cette baisse de performance par le développement d'un nouveau canal de communication utilisant le protocole *TCP/IP*, canal que nous avons ensuite utilisé pour effectuer une simulation distribuée de notre modèle multiprocesseur. Cette distribution de la simulation, réalisée par une distribution des éléments du modèle à simuler, a eu comme résultat une *accélération superlinéaire* de la simulation due au fait de la répartition « quasi-naturelle » du modèle sur le processeur réseau, situation idéale qui nous a évité d'avoir recours à des algorithmes de synchronisation (ce qui aurait eu pour effet d'introduire un overhead) et nous a aussi évité de faire face à un déséquilibre de charge impliquant des cycles inactifs nuisant à la performance. La synchronisation « naturelle » des éléments du modèle à simuler fut possible grâce à l'existence d'un faible échange de données et d'une latence négligeable de la communication sur le réseau. La tâche effectuée par chaque noyau de simulation s'est trouvée alors diminuée, ce qui s'est logiquement traduit par un accroissement superlinéaire de la performance.

Nous avons aussi montré que pour communiquer à distance, les modules *SystemC* ont tout intérêt à utiliser des canaux natifs au lieu d'une implémentation de ce type de communication par des librairies logicielles externes comme *MPI*. En effet, quand la distribution du modèle est grande, l'overhead introduit par ces librairies de code, par rapport à la version distribuée « native », devient important.

*SystemC* en tant que standard largement utilisé doit nécessairement, un jour ou l'autre, introduire des fonctionnalités de simulation distribuée et donc aura besoin de canaux de communication distante utilisant divers protocoles réseau. Nous avons proposé quelques exemples d'architecture d'un tel canal utilisant le protocole *TCP/IP* et l'avons implémenté et testé. Le canal que nous avons développé peut être enrichi par un ajout de fonctionnalités. Ceci est possible grâce à l'héritage objet. Par exemple, nous pouvons imaginer un canal utilisant un adressage en interne pour cibler n modules destinataires sur la même machine cible, au lieu d'un seul comme dans le cas de notre expérimentation, permettant ainsi une importante économie de ressources en n'utilisant qu'un seul socket au lieu des *n* sockets autrement nécéssaires (3 sockets utilisés dans notre cas). Le paquet transmis doit alors, par exemple, prévoir un champ d'adresse pour identifier de façon unique le module destinataire. Nous avons, lors de notre expérimentation du protocole *Simplex*, réalisé le même genre d'économie de ressource en utilisant un canal client-serveur avec un seul socket pour l'émission et la réception de données au lieu de deux canaux séparés client et serveur utilisant chacun un socket. On peut aussi vouloir proposer à l'utilisateur, de manière transparente, un choix entre plusieurs protocoles de communication en implémentant et en encapsulant toutes les routines d'émission/réception de ces différents protocoles à l'intérieur du canal. Enfin, ce type de canal peut être utilisé pour y implémenter les algorithmes de simulation parallèle vus au chapitre 1, en plus de ceux qu'on pourrait implémenter dans le noyau du moteur de simulation, et ce, dans le but d'implémenter un moteur de simulation distribuée pour une éventuelle future version distribuée de SystemC.

# BIBLIOGRAPHIE


[AB89] Y. Aahlad et J.C. Browne, "Balanced sequencing protocols", *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 58-63, SCS Simulation Series, 1989.

[AR90] R. Ayani and H. Rajaei, "Parallel simulation of a generalized cube multistage interconnection network", *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 60-63, Jan. 1990.

[AS92] D. K. Arvind and C. R. Smart, "Hierarchical parallel distributed simulation in Composite ELSA", *Proceedings of the 6th Workshop on Parallel and Distributed Simulation* (PADS'92), ed. M. Abrams and P. F. Reynolds, Jr., pp. 147-156, Newport Beach, USA, Jan. 1992.

[Ashe96] P.J. Ashenden, *The designer's guide to VHDL*, Morgan Kaufmann Publishers, Inc, USA, 688 p., 1996.

[Awe85] B. Awerbuch, "Complexity of network synchronization", *Journal of the ACM*, Vol. 32, 4, October 1985, pp. 804-823.

[BA91] B. Berkman and R. Ayani, "Parallel simulation of multistage interconnection networks on an SIMD computer", *Proceedings of the SCS Multiconference on Advances in parallel and distributed simulation*, pp. 133-140, SCS Simulation Series, Jan. 1991.

[BCG+96] R. Bagrodia, Y. -A. Chen, M. Gerla, B. Kwan, J. Martin, P. Palnati and S. Walton, "Parallel simulation of a high-speed wormhole routing network", *Proceedings of the 10th Workshop on Parallel and Distributed Simulation* (PADS'96), pp. 47-56, Philadelphia, USA, May 1996.

[BCJS95] R. Bagrodia, Y. -A. Chen, V. Jha and N. Sonpar, "Parallel gate-level circuit simulation on shared memory architectures", *Proceedings of the 9th Workshop on Parallel and Distributed Simulation* (PADS'95), pp. 170-174, Lake Placid, USA, June 1995.

[BEA98] C. Beaumont, "Simulation distribuée : de l'application vers un support système", Thèse de Doctorat, Université de Rennes 1, institut de formation supérieure en informatique et communication, France 1998.

[BMB94] Bailey, M., Briner, J., Chamberlain, R.D. "Parallel Logic Simulation of VLSI Systems", *Computing Surveys*, Vol. 26, No. 3, pp. 255-294, September 1994.

[BPH85] F. Brglez, P. Pownall and R. Hum, "Accelerated ATPG Fault Grading via Testability Analysis", In *IEEE International Symposium on Circuits and Systems* (ISCAS'85), pp. 695-698, June 1985.

[BRE77] Bryant, R.E. "Simulation of Packet Communication Architecture Computer Systems", Tech Report MIT/LCS/TR-188, Laboratory for Computer Science, MIT, Cambridge, MA, 1977.

[Bri91] J. V. Briner, "Fast parallel simulation of digital systems", In *Advances in parallel and distributed simulation*, pp. 71-77, SCS Simulation Series, Jan. 1991.

[Bry79] R. E. Bryant, "Simulation on a distributed system", *Proceedings of the 1st International Conference on Distributed Computing Systems* (ICDCS), pp. 544-552, Huntsville, USA, October 1979.

[CAPP02] L. Charest, E. M. Aboulhamid, C. Pilkington, and P. Paulin, "SystemC Performance Evaluation Using A Pipelined DLX Multiprocessor", *Proceedings of Design and Test in Europe Designer's Forum*, Paris, March 4-8, 2002.

[CCH+99] Henry Chang, Larry Cooke, Merril Hunt, Grant Martin, Andrew McNelly, Lee Tood, *Surviving the SOC Revolution*, Kluwer Academic Publishers, Boston, 335p., 1999.

[ChaBro83] A. Chandrak and J. Browne, "Vectorization of discrete event simulation", *Proceedings of the 1983 International Conference on Parallel Processing*, pp. 359-361, August 1983.

[ChMis79] K. M. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs", *IEEE Transactions on Software Engineering*, Vol. 5, No. 5, September 1979, pp. 440-452.

[ChMis81] K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations", Communications of the ACM, Vol. 24, 11, pp. 198-206, April 1981.

[ChMis82] K. M. Chandy and J. Misra, "Termination detection of diffusing computations in communicating sequential processes", ACM, Transactions on Programming Languages and Systems, Vol. 4, 1, pp. 37-43, Jan. 1982.

[CL85] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems", ACM Transactions on Computer Sytems, Vol. 3, 1, pp. 63-75, Jan. 1985.

[Con89] A.I. Concepcion, "A hierarchical computer architecture for distributed simulation", *IEEE Transactions on Computers*, Vol. 38, 2, pp. 311-319, Feb. 1989.

[CRAB01] L. Charest, M. Reid, E. M. Aboulhamid, G. Bois, "A Methodology for Interfacing Open Source SystemC with a Third Party Software", *Proceedings of Design Automation and Test in Europe Conference & Exhibition*, Munich, Germany, pp. 16-20, 13-16 March, 2001.

[CT96] J. G. Cleary and J. -J. Tsai, "Conservative parallel simulation of ATM networks", *Proceedings of the 10th Workshop on Parallel and Distributed Simulation* (PADS'96), pp. 30-38, Philadelphia, USA, May 1996.

[CYR01] G. Cyr, "Interface configurable pour un processeur ARM basée sur le protocole VCI", Mémoire de maîtrise, École polytechnique de Montréal, université de Montréal, Montréal, Québec, février 2001.

[DHN94] P.M. Dickens, P. Heidelberger and D. M. Nicol (D. M.), "Parallelized direct execution simulation of message-passing parallel programs", Rapport technique, Hampton, USA, ICASE, NASA Langley Research Center, 1994.

[DR91] P. M. Dickens, and P. F. Reynolds Jr., "A performance model for parallel simulation", *Proceedings of the 1991 Winter Simulation Conference*, pp. 618-626, Phoenix, USA, December 1991.

[EDP+89] M. Ebling, M. DiLorento, M. Presley, F. Wieland and D. R. Jefferson, "An ant foraging model implemented on the Time Warp Operating System", *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 21-26, Tampa, USA, March 1989.

[Fid88] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering", *Proceedings of the 11th Australian Computer Science Conference*, pp. 55-66, Université du Queensland, Australie, Feb. 1988.

[FSSy20] "Functional Specification for SystemC 2.0", www.systemc.org, Jan. 2001.

[Fuj90] Fujimoto, R.M., "Parallel Discrete Event Simulation", Communications of the ACM, Vol. 33, No. 10, pp 30-53, October 1990.

[FW95] J. Fleishmann and P. A. Wilsey, "Comparative study of periodic state saving techniques in Time Warp simulators", *Proceedings of the 9th Workshop on Parallel and Distributed Simulation* (PADS'95), pp. 50-58, Lake Placid, USA, June 1995.

[Gaf88] A. Gafni, "Rollback mechanisms for optimistic distributed simulation systems", *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 61-67, San Diego, USA, July 1988.

[GDG00] Gajski, Zhu, Domer, Gerstlauer, Zhao, *SpecC Design Methodology*, Kluwer Academic Publishers, Boston, 313p., 2000.

[GGN93] B. Gaujal, A. Greeberg and D. M. Nicol, "A sweep algorithm for massively parallel simulation of circuit-switched networks", Journal of parallel and distributed computing, Vol. 18, 4, pp. 484-500, August 1993.

[GR00] J. Gerlach and W. Rosenstiel, "System level design using the SystemC modeling platform", In *Workshop on System Design Automation*, pp. 185–189, Rathen, Germany, Mar. 2000.

[Gun94] M. A. Gunter, "Understanding superlinear speedup", *Proceedings of the 8th Workshop on Parallel and Distributed Simulation* (PADS'94), éd. D.K. Arvind, R. Bagrodia, J.Y. –B. Lin, pp. 81-87, Edinburgh, Ecosse, May 1994.

[Gup95] R. K. Gupta, *Co-Synthesis of Hardware and Software forDigital Embedded Systems*, Kluwer Academic Publishers, Boston, U.S.A, 1995.

[GZR+00] D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, Norwell, U.S.A, 2000.

[HBD+89] P. Hontalas, B. Beckman, M. DiLorento, L. Blume, P. Reiher, K. Sturdevant, L. Van Warren, J. Wedel, F. Wieland et D. R. Jefferson, "Performance of the colliding pucks simulation on the Time Warp Operating System", *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 3-7, Tampa, USA, March 1989.

[HePat95] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Second Ed: Morgan Kaufmann Publishers, 1995.

[JBW+87] D. R. Jefferson, B.Beckman, F. Weiland, L. Blume, M. DiLorento, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, J. Weder, H. Younger et S. Bellenot , "The Time Warp Operating System", Operating System Review, ACM Press, Vol. 21, 5, pp. 77-93, 1987.

[Jef85] D. R. Jefferson, "Virtual time", ACM Transactions on Programming Languages and Systems, Vol. 7, 3, pp. 404-425, July 1985.

[JRW89] Jefferson D. R., P. L. Reiher and F. Wieland, "Limitation of optimism in the Time Warp Operating System", *Proceedings of the 1989 Winter Simulation Conference*, Ed. E. A. McNair, K. J. Musselman et P. Heidelberger, pp. 765-769, Washington D.C., USA, December 1989.

[Lam78] L. Lamport, "Time, clocks, and the ordering of events in a distributed system", Communications of the ACM, Vol. 21, 7, pp. 558-565, July 1978.

[Lin90] Y. -B. Lin, "Understanding the limits of optimistic and conservative parallel simulation", Thèse de doctorat, University of Washington, USA, Août 1990.

[LSW89] B. D. Lubachevsky, A. Schwartz et A. Weiss, "Rollback sometimes works. . . if filtered", *Proceedings of the 1989 Winter Simulation Conference*, éd. by E. A. McNair, K. J. Musselman and P. Heidelberger, pp.630-639, Washington D.C., USA, December 1989.

[LTG97] S. Liao, S. Tjiang, and R. Gupta, "An efficient implementation of reactivity for modeling hardware in the scenic design environment", In *Design Automation Conference*, pp. 70–75, Anaheim, CA, June 1997.

[Lub88] B. D. Lubachevsky, "Bounded lag distributed discrete event simulation", *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 183-193, San Diego, USA, feb. 1988.

[Lub89] B. D. Lubachevsky, "Efficient distributed event-driven simulation using multiple-loop networks", Communications of the ACM, Vol. 32, 1, pp. 111-123, Jan. 1989.

[Mar91] J. Marline, "Noyau de simulation distribuée sur Transputer", Brest, France, Rapport de mastère GtII, Telecom Bretagne, avril 1991.

[Mat88] F. Mattern, "Virtual time and global states of distributed systems", *Proceedings of the Workshop on Parallel and Distributed Algorithms*, ed. by M. Cosnard and others, pp. 215-226, Bonas, France, October 1988.

[Misra86] Misra, J. "Distributed Discrete-Event Simulation", Computing Surveys, Vol. 18, No. 1, March 1986.

[Moore65] Gordon E. Moore, *Cramming more components onto integrated circuits*, Electronics, Volume 38, Number 8, April 19, 1965.

[MPI] "MPI – The Message Passing Interface Standard", http://www.mcs.anl.gov/mpi

[MRR90] B. C. Merrifield, S. B. Richardson et J. B. G. Roberts, "Quantitative studies of discrete event simulation of road traffic", *Proceedings of the*

*SCS Multiconference on Distributed Simulation*, pp. 188-193, SCS Simulation Series, 1990.

[NiFu94] D. M. Nicol and R. M. Fujimoto, "Parallel simulation today", Annals of Operation Research, Vol. 53, pp. 249-285, 1994.

[OSCI] Open SystemC Initiative (OSCI), www.systemc.org

[PLH88] B. R Preiss, W. Loucks and V. C. Hamaker, "A unified modeling methodology for performance evaluation of distributed discrete event simulation mechanisms", *Proceedings of the 1988 Winter Simulation Conference*, ed. by M. Abrams, P. Haigh et J. Comfort, pp. 315-324, San Diego, USA, December 1988.

[Ppr01] Preeti Ranjan Panda, Synopsys Inc., "SystemC - A modeling platform supporting multiple design abstractions", *ISSS'01,* Montreal, Quebec, Canada, October 1-3, 2001.

[PWM79] J. K. Peacock, J.W. Wong et E.G. Manning, "Distributed simulation using a network of processors", Computer Networks, Vol. 3, 1, pp. 44-56, Feb. 1979.

[PWPDS] *Proceedings of the Workshops on Parallel and Distributed Simulation*, IEEE Computer Society.

[RaMu91] H. Rakotoarisoa et P. Mussi, "PARSEVAL : PARallélisation sur réseaux de transputers de Simulations pour l' EVALuation de performances", Rapport technique 131, INRIA, septembre 1991.

[RAT93] H. Rajaei, R. Ayani and L.-E. Thorelli, "The Local Time Warp approach to parallel simulation", *Proceedings of the 7th Workshop on Parallel and Distributed Simulation* (PADS'93), ed. by R. Bagrodia and D. R. Jefferson, pp.119-126, San Diego, USA, May 1993.

[RW89] R. Righter and J.C. Walrand, "Distributed simulation of discrete event systems", *Proceedings of the IEEE*, Vol. 77, 1, pp. 99-113, Jan. 1989.

[Sam85] B. Samadi, "Distributed simulation, algorithms and performance analysis", Thèse de doctorat, University of California, Los Angeles, USA, 1985.

[SBW88] L. Sokol, D. Briscoe and A. Wieland, "MTW: strategy for scheduling discrete event simulation events for concurrent execution", *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 34-43, San Diego, USA, Feb. 1988.

[SG91] L. Soulé et A. Gupta, "An evaluation of the Chandy-Misra-Bryant algorithm for digital logic simulation", ACM Transactions on Modeling and Computer Simulation, Vol. 1, 4, pp. 308-347, 1991.

[SK92] S. Shen et L. Kleinrock, "The virtual time data-parallel machine", *Proceedings of the 4th Symposium on the frontiers of massively parallel computation*, pp. 46-53, National Aeronautics and Space Administration (NASA), IEEE Computer Society Press, 1992.

[SS01] S. Swan, Cadence Systems, Inc, "An introduction to system level modeling in SystemC 2.0", OSCI, www.systemc.org, Technical papers, May 2001.

[Ste92] J. Steinman, "SPEEDES: A unified approach to parallel simulation", *Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS'92)*, ed. by M. Abrams and P. F. Reynolds, Jr., pp. 75-83, Newport Beach, USA, Jan. 1992.

[TK88] P. Tinker et M. Katz, "Parallel execution os sequential Scheme with Paratran", *Proceedings of the ACM Conference on Lisp and Functional Programming*, pp. 28-39, Snowbird, USA, July 1988.

[TuCai93] S.J. Turner and W. Cai, " 'The Logical Clocks'approach to the visualization of parallel programs", In *Performances Measurement and Visualization of Parallel Systems*, ed. by Haring (G.) and Kotsis (G.), pp. 45-66, North Holland, 1993.

[TX92] S. J. Turner et M. Q. Xu, "Performance evaluation of the Bounded Time Warp algorithm", *Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS'92)*, ed. by M. Abrams and P. F. Reynolds, Jr., pp. 117-128, Newport Beach, USA, Jan. 1992.

[TZ91] G. S. Thomas and J. Zahorjan, "Parallel simulation of performance petri nets: Extending the domain of parallel simulation", *Proceedings*

*of the 1991 Winter Simulation Conference*, pp. 564-573, Phoenix, USA, December 1991.

[UGSy11] "SystemC 1.1 User's guide", available at www.systemc.org.

[Ver93] E. Sternheim, R. Singh, Y. Trivedi, R. Madhaven and W. Stapleton, *Digital Design and Synthesis with Verilog HDL*, published by Automata Publishing Co., Cupertino, CA, 1993.

[VRL86] K. Venkatesh, T. Radhakrishnan and H. F. Li, "Discrete event simulation in a distributed system", In IEEE COMPSAC. IEEE Computer Society Press, pp. 123-129, 1986.

[VSB99] S. Vernalde, P. Schaumont, I. Bolsens, "An Object Oriented Programming Approach for Hardware Design", *Proceedings of IEEE Computer Society Workshop on VLSI*, Orlando, April 1999.

[WHF+89] F. Wieland, L. Hawley, A. Feinberg, M. DiLorento, L. Blume, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot et D. R. Jefferson, "Distributed combat simulation and time warp: The model and its performance", *Proceedings of the SCS Multiconference on Distributed Simulation*, pp.14-20, Tampa, USA, March 1989.

[WT94] K. R. Wood and S. J. Turner, "A generalized carrier-null method for conservative parallel simulation", *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS'94)*, ed. by D. K. Arvind, R. Bagrodia and J. Y. -B. Lin, pp. 50-57, Edinburgh, Ecosse, July 1994.

# ANNEXE : L'ORDONNANCEUR SYSTEMC 2.0

```
// -----------------------------------------------------------------------------------------------
// Ceci est du pseudo code pour l'Ordonnanceur SystemC
//
// Note:  Les termes thread et process sont, ici, utilisés de manière interchangeable.
// -----------------------------------------------------------------------------------------------

// Type sc_time will actually be a class (so that it can have various
// constructors), but it basically works like an unsigned long long.

typedef unsigned long long sc_time;

const sc_time SC_ZERO_TIME = 0;

class sc_scheduler {

        public:

        sc_scheduler(): _sc_now( 0 ), delta_count( 0 ) {}
        sc_time sc_now(){ return _sc_now; }
        void execute();

        private:

        friend class sc_event;
        friend class sc_thread;

        // Thread Sets:
        runnable: The set of threads that are ready to run ;
        not_runnable: The set of threads that are not ready to run ;

        // Note: Every thread is either in the runnable or not_runnable sets.
        // The thread sets never contain any duplicate entries.
        // Notification Sets:
        delayed_notifications: The set of events that have pending
        notifications for the next delta step ;
        timed_notifications: A sorted list of future times. For each unique future time, there is a set of
        events with pending notifications at that future time ;

        // Note: At any point, an event can have at most one pending
        // notification. SystemC always follows the "earliest wins"
        // rule for notifications.
        sc_time _sc_now;

        // In order to be able to efficiently compute signal.event(),
        // which is true iff a signal value changed in the previous delta cycle,
        // the scheduler maintains a delta cycle counter. This counter is never
        // reset.
        unsigned long long delta_count;
};
```

```
class sc_event {

        public:

        sc_event(): notify_type( NONE ), when_to_notify( 0 ) {}

        notify() {                              // an immediate notification
                trigger();
        }

        notify( sc_time t ) {           // a timed notification

                if( notify_type == DELAYED ) return;
                if( t == SC_ZERO_TIME ) {
                        delayed_notify();
                        return;
                }
                if( notify_type == TIMED ) {
                        if( when_to_notify <= sc_now() + t ) return;
                        remove this event from its current notification set ;
                }
                notify_type = TIMED;
                when_to_notify = sc_now() + t;
                add this to timed_notifications at time when_to_notify ;
        }

        ~sc_event() {
                for t in waiting_dynamic_threads t.remove_dynamic_event( *this );
                for t in waiting_static_threads t.remove_static_event(*this);
                if( notify_type != NONE ) remove this event from its current notification set ;
        }

        private:

        friend class sc_thread;
        waiting_dynamic_threads: A set of references to threads dynamically waiting on this event ;
        waiting_static_threads: A set of references to threads statically waiting on this event ;

        enum notify_t { NONE, DELAYED, TIMED };
        notify_t notify_type;
        sc_time when_to_notify;

        delayed_notify() {                      // a delayed notification (one delta cycle)
                if( notify_type == DELAYED ) return;
                if( notify_type == TIMED ) remove this event from its current notification set ;
                notify_type = DELAYED;
                add this event to the delayed_notifications set ;
        }


        trigger() {
                if( notify_type != NONE ) remove this event from its current notification set ;
                for t in waiting_static_threads t.trigger( this, false );
                for t in waiting_dynamic_threads t.trigger( this, true );
                reset();
        }
```

```
        reset() {
                notify_type = NONE;
                when_to_notify = 0;
                clear waiting_dynamic_threads set ;
        }

        add_dynamic_thread( sc_thread& t ) {
                add t to waiting_dynamic_threads ;
        }

        remove_dynamic_thread( sc_thread& t ) {
                remove t from waiting_dynamic threads ;
        }

        add_static_thread( sc_thread& t ) {
                add t to waiting_static_threads ;
        }

        remove_static_thread( sc_thread& t ) {
                remove t from waiting_static_threads ;
        }
};


class sc_thread { // base class for a thread or process

        public:

        wait( e_col: A reference to a collection of event references ){
                if( is_sc_method ) {
                        cerr << "Can't call wait() from an SC_METHOD\n";
                        return;
                }
                add each event in e_col into dynamic_events ;
                for e in dynamic_events e.add_dynamic_thread( this );
                set "and_sensitive" to "true" if collection e_col is an AND collection ;
                waiting_dynamically = true;
                save the context (ie. stack) of this thread ;
                suspend execution in the current context and resume execution in the previous
                context (the scheduler's context) ;
        }

        wait( sc_time t ){
                sc_event e;
                e.notify( t );
                wait( e ); // calls wait( e_col ) via a conversion
        }

        wait( sc_time t, e_col: A reference to a collection of event references){
                sc_event e;
                e.notify( t );
                if( e_col is not an "AND" collection ) {
                        add e to e_col ;
                        wait( e_col );
                        remove e from e_col ;
                }
```

```
        else {
                and_timeout_event = &e;
                wait( e_col );
        }
}

wait() {                        // ie. wait on static sensitivity
        if( is_sc_method ) {
                cerr << "Can't call wait() from an SC_METHOD\n";
                return;
        }
        and_sensitive = false
        waiting_dynamically = false;
        save the context (ie. stack) of this thread ;
        suspend execution in the current context and resume execution
        in the previous context (the scheduler's context) ;
}

sc_thread( bool sc_method_flag ){
        is_sc_method = sc_method_flag;
        waiting_dynamically = false;
        and_sensitive = false;
        if( ! is_sc_method ) create a new context (ie execution stack) for this thread ;
}

~sc_thread(){
        unwait();
        for e in static_events remove_from_static_sensitivity( e );
        remove "this" from runnable or not_runnable ;
        if( ! is_sc_method ) delete the context (ie execution stack) for this thread ;
}

add_to_static_sensitivity( sc_event& e ){
        if( e already in static_events ) return;
        add e to static_events ;
        e.add_static_thread( this );
}

remove_from_static_sensitivity( sc_event& e ){
        remove e from static_events ;
        e.remove_static_thread( this );
}

private:

friend class sc_event;
friend class scheduler;
dynamic_events: A set of references to events that this thread is dynamically sensitive to ;
static_events: A set of references to events that this thread is statically sensitive to ;
bool and_sensitive; // true iff currently waiting on "AND" event collection
sc_event* and_timeout_event;
bool is_sc_method;          // true iff this thread is an SC_METHOD rather than SC_THREAD
bool waiting_dynamically;           // true iff this thread is currently
```

```
        // waiting on dynamic_events
        unwait(){
                for e in dynamic_events e.remove_dynamic_thread( this );
                clear dynamic_events ;
                and_sensitive = false;
                and_timeout_event = 0;
        }

        resume(){
                if( is_sc_method ) call the SC_METHOD "start method" ;
                else {
                        suspend execution of the current thread (i.e. the scheduler's context) and
                        resume execution of this thread in its previously saved context ;
                        // (We automatically return to this point when this thread next
                        // waits, and we will be executing in the scheduler's context
                        // again.)
                }
        }

        trigger( sc_event& e, bool dynamic ){
                if( dynamic != waiting_dynamically ) return;
                if( is_sc_method ) {
                        move "this" from not_runnable to runnable ;
                        return;
                }
                if( ! and_sensitive ) {
                        move "this" from not_runnable to runnable ;
                        unwait();
                }
                else {
                        if( &e == and_timeout_event ) {
                                move "this" from not_runnable to runnable ;
                                unwait();
                                return;
                        }
                        remove e from dynamic_events ;
                        if( dynamic_events is empty ) {
                                move "this" from not_runnable to runnable ;
                                unwait();
                        }
                }
        }
};

void scheduler::execute(){
        // Initialization Phase:
        // Note that all threads except SC_CTHREADS are run once at
        // initialization in an unspecified order.

        for t in ( all threads except SC_CTHREADS ) t.resume();

        while( true ) {
                // Evaluate Phase:
                while( runnable is not empty ) {
                        select a thread t in runnable and move it to not_runnable ;
                        t.resume();
```

```
                    }
                    // Update Phase:
                    execute any pending calls to update() resulting from
                    request_update() calls ;
                    delta_count ++ ;
                    // Process delayed notifications
                    if( delayed_notifications is not empty ) {
                            for e in delayed_notifications e.trigger();
                            continue;
                    }
                    // Process timed notifications
                    if( timed_notifications is empty ) break;
                    _sc_now = the earliest time in timed_notifications ;
                    for e in ( set of events to be notified at earliest time in timed_notifications )
                    e.trigger();
                    delete the earliest time in timed_notifications and the set of events to be notified at
                    that time ;
          }
}
```

*Notes:*

L'ordre dans lequel l'ordonnanceur choisit les *threads* à exécuter dans les phases d'évaluation et de mise à jour est non spécifié et est dépendant de l'implémentation choisie. Cependant, quand la même conception est simulée de multiples fois en employant le même stimulus et la même version du simulateur, l'ordre des *threads* entre les différentes simulations ne varie pas.

*SystemC* fournit des options de ligne de commande du simulateur qui permettent à l'utilisateur de randomiser l'ordre des *threads* durant la phase d'évaluation. Cette fonctionnalité est utile pour la détection de défauts de conception résultant d'une synchronisation inadéquate dans les spécifications de conception.