Partitionnement des services de communication en vue de la génération automatique des interfaces logicielles/matérielles

Y. Paviot

► To cite this version:

Y. Paviot. Partitionnement des services de communication en vue de la génération automatique des interfaces logicielles/matérielles. Micro et nanotechnologies/Microélectronique. Institut National Polytechnique de Grenoble - INPG, 2004. Français. NNT : . tel-00007013

HAL Id: tel-00007013
https://theses.hal.science/tel-00007013

Submitted on 1 Oct 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|__/__/__/__/__/__/__/__/__/__|

THESE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Microélectronique

préparée au laboratoire **TIMA** dans le cadre de l'Ecole Doctorale
"Electronique, Electrotechnique, Automatique, Télécommunication, Signal"

Présenté et soutenue publiquement par

Yanick PAVIOT

le 1 Juillet 2004

Partitionnement des services de communication en vue de la génération automatique des interfaces logicielles/matérielles

Directeur de thèse
Ahmed Amine Jerraya
Co-directeur de thèse
Sungjoo Yoo

Jury :

M. Gérard Michel	**Président**
M. Frédéric Pétrot	**Rapporteur**
M. Amara Amara	**Rapporteur**
M. Ahmed Amine Jerraya	**Directeur de thèse**

Remerciements

Je remercie Monsieur Bernard Courtois, directeur de recherche au CNRS et directeur du laboratoire TIMA pour m'avoir accueilli dans son laboratoire.

Je tiens tout particulièrement à remercier M. Ahmed Amine Jerraya, directeur de recherche au CNRS et responsable du groupe SLS, pour m'avoir accepté dans son groupe et encadré tout au long de cette thèse. J'ai particulièrement apprécié sa disponibilité, sa patience et ses conseils m'ayant permis de mener à bien ces travaux.

Je remercie de tout mon cœur Monsieur Sungjoo Yoo, pour tout ce qu'il m'a apporté. J'admire la patience infinie qu'il a su déployer, sa grande capacité d'écoute et tous les bons conseils qui m'ont aidé à surmonter les obstacles.

Je remercie Monsieur Gérard Michel de m'honorer en présidant le jury de ma thèse. Merci à Monsieur Frédéric Pétrot et Monsieur Amara Amara pour avoir bien voulu juger cette thèse en acceptant d'en être les rapporteurs.

Merci à Frédéric Rousseau de m'avoir donné l'occasion d'enseigner et pour sa sympathie naturelle.

Merci à Sonja Amadou pour le réconfort que tu m'as apporté. Tes conseils ont toujours été précieux et je tâcherai de ne pas les oublier.

Je remercie aussi mes partenaires de bureau Arif, Arnaud et Férid pour m'avoir supporté durant cette dernière année. Arif, monsieur sourire, je suis content d'avoir pu te côtoyer. Je te souhaite bonne chance pour la suite : j'espère juste que tu ne seras pas trop dur avec tes futurs employés. Arnaud, désolé de t'avoir fait régulièrement du pied mais c'était pour ton bien : il ne faudrait pas que tu t'endormes, tu sais bien que tu as un énorme potentiel. En tous cas merci d'avoir écouté mes blagues douteuses. Férid, le professionnel du bureau, ta corvette n'est plus très loin, je suis sûr qu'elle sera bientôt à toi, je suis content de t'avoir connu.

Je remercie les anciens du laboratoire qui m'ont accueilli dans leur groupe. Merci aux expatriés Damien, Lovic et Gabriela pour m'avoir transmis leurs connaissances ainsi que leur amitié. Merci aussi à Nacer de m'avoir permis de connaître le groupe et pour tous les conseils que tu m'as donnés.

Je tiens aussi à remercier Wassim, Aimen, Lobna, Fredéric, Ivan et Iuliana, pour leur sympathie.

J'aimerais aussi remercier tous ceux qui ont partagé mes repas, les pauses café ainsi que les soirées qui ont agrémenté mes dernières années. Donc merci au groupe CIS : Dhanistha, Manu, Jérôme, Fraidy, Joao, Estelle, Sophie, Yann, Yannick, Salim et Arnaud. Merci aussi à mon compagnon de virées de moto Greg et sa bande du CMP.

Enfin j'aimerais plus particulièrement remercier celle qui m'a le plus soutenue durant cette thèse. Celle qui m'a donné la force de continuer et d'espérer. Je remercie Emeline pour l'amour qu'elle me donne.

Table des matières

Chapitre 1 Introduction ... 1

1.1 Contexte : Programmation parallèle et communication entre processeurs sur système mono puce ... 2
1.1.1 Les systèmes sur puce : ... 2
1.1.2 La conception des systèmes sur puce multiprocesseurs ... 2
1.1.3 Programmation parallèle et implémentation d'interfaces logicielles/matérielles pour la communication entre processeurs ... 4
1.2 Concepts de base : conception des interfaces logicielles matérielles pour la communication entre processeurs ... 5
1.2.1 Les interfaces logicielles matérielles ... 5
1.2.2 Abstraction des interfaces et modèle de programmation parallèle ... 6
1.2.3 Les différents types de synchronisation entre partie logicielle et partie matérielle ... 8
1.2.4 Etat de l'art sur la conception automatisée des interfaces de communication ... 9
1.3 Problématique ... 10
1.3.1 Objectif : exploration des solutions d'implémentations mixtes logicielles/matérielles de services de communication ... 10
1.3.2 Problèmes et difficultés : ... 11
1.3.3 Conception manuelle des interfaces ... 12
1.3.4 Un espace de solution trop grand pour une exploration manuelle ... 12
1.4 Contribution ... 12
1.4.1 Formalisation des problèmes liés à la réalisation d'interfaces mixtes logicielles/matérielles des services de communication ... 12
1.4.2 Génération automatique des interfaces ... 13
1.5 Plan de thèse ... 14
Chapitre 2 Environnement ROSES pour la conception des interfaces logicielles/matérielles ... 15

2.1 Le flot global ... 16
2.1.1 Domaine d'application : conception de couches de communication logicielles/matérielles pour les systèmes multiprocesseurs mono puce ... 16
2.1.2 Le modèle de représentation COLIF ... 16
2.1.3 Architecture du flot : flux d'entrées et de sorties ... 18
2.2 ASOG : un outil de génération automatique d'OS ... 21
2.2.1 Concept : le ciblage logiciel ... 21
2.2.2 Les mécanismes d'assemblage ... 22
2.2.3 Structure de la bibliothèque ... 22
2.2.4 Le flot de génération automatique d'OS ... 25
2.3 ASAG : génération automatique de la partie matérielle ... 27
2.3.1 Entrées et sorties ... 27
2.3.2 Les bibliothèques ... 27
2.3.3 Les étapes ... 28
2.4 Cosimx : cosimulation ... 30
2.4.1 Utilité de l'outil Cosimx ... 30
2.4.2 Principes de fonctionnement ... 30
2.5 Illustration du flot avec l'application VDSL ... 30
2.5.1 Description d'une application : un framer VDSL ... 30
2.5.2 Spécification de l'application ... 32

2.5.3 La bibliothèque de système d'exploitation pour l'application 37
2.5.4 Résultats 43
2.5.5 Conclusions sur le flot de ciblage logiciel 51
2.5.6 Résultats de la génération de la partie matérielle 52
2.6 Conclusion 53
Chapitre 3 Architecture des interfaces et services de communication 55
3.1 Architecture des interfaces logicielles/matérielles 56
3.1.1 Architecture de la partie logicielle 56
3.1.2 Architecture de la partie matérielle 59
3.2 Introduction à MPI 60
3.2.1 Présentation de MPI 60
3.2.2 Les objets: définitions 61
3.2.3 Les primitives de communications 61
3.2.4 Types de données et primitives de dérivation de données 64
3.2.5 Les primitives de configuration 66
3.2.6 Divers 67
3.3 Sélection d'un sous ensemble de primitives MPI 68
3.3.1 Description détaillée du comportement des primitives de communication point à point 68
3.3.2 Critères de sélection du sous ensemble 73
3.3.3 Sous-ensemble sélectionné 74
3.4 Conclusion 74
Chapitre 4 Propositions pour la génération automatique d'interfaces logicielles/matérielles en vue de l'exploration du partitionnement des services de communication 75
4.1 Partitionnement logiciel/matériel 76
4.1.1 Le principe du partitionnement 76
4.1.2 Critères de choix du partitionnement logiciel/matériel 77
4.1.3 Flot de partitionnement logiciel/matériel des réalisations de services de communication 77
4.2 Modèle de représentation des services de communication 80
4.2.1 Utilité d'un modèle de représentation des services de communication 80
4.2.2 Description du modèle de représentation 81
4.3 Génération automatique 84
4.3.1 Rappels sur la méthode actuelle de génération automatique d'interface logicielle/matérielle dans le flot ROSES 84
4.3.2 Utilisation du modèle de représentation des services de communication pour la génération des interfaces logicielles/matérielles 87
4.3.3 Intégration dans la spécification VADeL des descriptions de modèles de service de communication 87
4.3.4 Génération de la partie logicielle 88
4.3.5 Génération de la partie matérielle 90
4.4 Conclusions 91
Chapitre 5 Expérimentation : réalisations logicielles/matérielles et représentation de services de communication pour la communication MPI 93
5.1 Implémentation 100% logicielle sur une plateforme ARM 94
5.1.1 Objectifs de l'expérience 94
5.1.2 Caractéristiques de l'implémentation 94
5.1.3 Modèle de représentation des primitives MPI_SEND et MPI_RECV 97
5.1.4 Architecture de la partie logicielle 98
5.1.5 Spécification et composants de bibliothèque pour la génération automatique 99
5.1.6 Conclusion 105
5.2 Implémentation mixte logicielle/matérielle 106

5.2.1 Représentation des primitives du sous-ensemble MPI avec le modèle de service de communication 106
5.2.2 Les caractéristiques de l'implémentation 111
5.2.3 La partie logicielle 116
5.2.4 La partie matérielle 118
5.3 Conclusion 119
Chapitre 6 Conclusion et perspectives 121
6.1 Conclusion 121
6.2 Perspectives 122
6.2.1 Développement du flot de génération de la partie logicielle à partir du modèle développé 123
6.2.2 Développement d'un flot de génération de la partie matérielle à partir du modèle commun 123
6.2.3 Estimation de performances 123
6.2.4 Aide à l'insertion d'éléments de bibliothèque d'ASOG 123
6.2.5 Automatisation du choix de partitionnement 123

Liste des figures

Figure 1 : Exemple de système mono puce multi processeurs 2
Figure 2 : Flot de conception des systèmes multi processeurs 3
Figure 3 : Communication entre processeurs dans les systèmes mono puce 5
Figure 4 Les parties logicielles et matérielles d'une interface 6
Figure 5 : Communication par mémoire partagée 7
Figure 6 : Communication par passage de message 7
Figure 7 : Objectif 10
Figure 8 : Les différentes phases pour atteindre l'objectif 11
Figure 9 : Contributions pour la génération automatique des interfaces mixtes 14
Figure 10 : Illustration des modules, ports et nets 17
Figure 11 : Exemple de composant virtuel 18
Figure 12 : Flux d'entrées sorties du flot 19
Figure 13 : Flot simplifié de ROSES 21
Figure 14 : ASOG 26
Figure 15 : ASAG 28
Figure 16 : La partie numérique du modem VDSL 31
Figure 17 : La démonstration VDSL 32
Figure 18 : L'application VDSL 33
Figure 19 : Paramètres de l'application VDSL pour le module M1 34
Figure 20 : Paramètres de l'application VDSL pour le module M2 35
Figure 21 : La bibliothèque de système d'exploitation pour l'application VDSL 40
Figure 22 : Le système d'exploitation généré pour le module M1 45
Figure 23 : Le système d'exploitation généré pour le module M2 46
Figure 24 : Exemple de code généré 50
Figure 25 : La micro-architecture générée pour l'application VDSL 52
Figure 26 : Architecture des interfaces logicielles/matérielles 56
Figure 27 : Concepts de couches logicielles et d'API. 57
Figure 28 : Architecture de l'interface matérielle 60
Figure 29 : Communications collectives 64
Figure 30 : Graphe de transactions et différents scénarios pour MPI_BSEND 68

Figure 31 : Graphe de transactions et différents scénarios pour MPI_RSEND 69
Figure 32 : Graphe de transactions et différents scénarios pour MPI_SSEND 70
Figure 33 : Graphe de transactions et différents scénarios pour MPI_ISSEND 71
Figure 34 : Graphe de transactions et différents scénarios pour MPI_IBSEND 72
Figure 35 : Graphe de transactions et différents scénarios pour MPI_RSSEND 72
Figure 36 : Les primitives MPI sélectionnées 74
Figure 37 : Allocation des unités 76
Figure 38 : flot de partitionnement logiciel/matériel des implémentations de services de communication 79
Figure 39 : Flot de génération des interfaces logicielles/matérielles 80
Figure 40 : Modèle à base d'éléments, services et implémentations 81
Figure 41 : Exemple de représentation à base d'UF et d'USD 82
Figure 42 : Correspondance entre le modèle de Lovic Gauthier et celui proposé 83
Figure 43 : Spécification d'une implémentation de service de communication dans le flot ROSES 85
Figure 44 : Les types de paramètre et leur utilisation par ASOG et ASAG 85
Figure 45 : Influence du paramètre SoftPortType sur la sélection de composants dans ASOG 86
Figure 46 : Influence des paramètres dans la sélection des composants dans ASAG 87
Figure 47 : Nouvelle spécification d'un service de communication1 88
Figure 48 : Sélection de composants à partir d'un modèle de représentation de service de communication 89
Figure 49 : Flot étendu intégrant le modèle de représentation des services de communication 90
Figure 50 : Plateforme ARM 95
Figure 51 : allocations des ressources 96
Figure 52 : Décomposition, transformation et transfert des données 97
Figure 53 : Représentation de MPI_SEND 98
Figure 54 : Représentation de MPI_RECV 98
Figure 55 : Architecture de l'implémentation 100% logicielle de MPI_SEND/MPI_RECV 99
Figure 56: Spécification de l'application OpenDivX 100
Figure 57 : Spécification des services de communication MPI_SEND et MPI_RECV 101
Figure 58 : Relations entre éléments et services utilisés pour la génération de MPI_SEND et MPI_RECV 102
Figure 59: Extrait de description en Lidel des dépendances entre éléments 102
Figure 60 : Extrait de code macro 103
Figure 61: Découpage en unités fonctionnelles et unités de structures données pour MPI_SEND 108
Figure 62 : Découpage en unités fonctionnelles et unités de structures données de MPI_SSEND 109
Figure 63 : Découpage en unités fonctionnelles et unités de structures données de MPI_ISSEND 110
Figure 64 : Découpage en unités fonctionnelles et unités de structures données pour MPI_RECV 110
Figure 65 : Découpage en unités fonctionnelles et unités de structures données de MPI_IRECV 111

Figure 66: Allocation des UF et USD de MPI_SEND et MPI_RECV dans les parties logicielles et matérielles 112
Figure 67: Allocation des UF et USD de MPI_SSEND et MPI_RECV dans les parties logicielles et matérielles 112
Figure 68: Allocation des UF et USD de MPI_ISSEND et MPI_IRECV dans les parties logicielles et matérielles 113
Figure 69 : Communication entre partie logicielle et partie matérielle 114
Figure 70 : Illustration du « multithreading » de la communication avec MPI_SEND 116
Figure 71 : Exécution de deux tâches pour la validation du code indépendant du matériel......117
Figure 72 : Les modèles utilisés pour la validation......118
Figure 73: Architecture de la partie matérielle 119

Liste des tableaux

Tableau 1: Résultat de synthèse de l'interface matérielle 52
Tableau 2 : décomposition en couche de l'architecture logicielle de Pipe_Get 59
Tableau 3 : Primitives de communication point à point bloquantes 62
Tableau 4 : Primitives pour les communications point à point non bloquantes 62
Tableau 5 : Primitives pour les communications multi points 63
Tableau 6 : Les différents types de données 65
Tableau 7 : Les primitives de dérivation 65
Tableau 8 : Primitives de création de groupes 67
Tableau 9: Fichiers du code macro des composants intégrés dans ASOG 103
Tableau 10 : Taille du code généré pour le master et un slave 104
Tableau 11:Temps d'exécution des primitives MPI_SEND et MPI_RECV 105
Tableau 12: Liste des unités fonctionnelles 107
Tableau 13 : Liste des unités de stockage de données 108
Tableau 14 : Taille du code implémentant le sous ensemble MPI 117

Chapitre 1

Introduction

Ce chapitre présente les problèmes spécifiques à la conception des systèmes sur puce multi processeurs. Il précise le contexte de ces travaux de thèse en se focalisant sur les interfaces logicielles/matérielles permettant la communication entre processeurs. Il présente aussi des concepts de base nécessaires à la compréhension du manuscrit. Il explique enfin la problématique liée à la conception des interfaces logicielles/matérielles, quels sont les objectifs à atteindre et quelles contributions apportent ces travaux.

1.1 Contexte : Programmation parallèle et communication entre processeurs sur système mono puce

1.1.1 Les systèmes sur puce :

Un système sur puce, communément appelé SoC pour « System on a Chip », est un ensemble de composants intégrés sur une seule et même puce. L'intégration d'un système complet sur puce permet de réduire les coûts de production, mais aussi la miniaturisation des objets de la vie quotidienne. Les domaines d'utilisation sont vastes : la télécommunication sans fil, l'automobile, l'aéronautique, la médecine, la domotique et le multimédia.
Grâce à l'évolution incessante des techniques de miniaturisation, la diversité et le nombre de composants ne cessent de croître. Ainsi, il est désormais possible d'intégrer plusieurs ASIC, processeurs, DSP, des modules de communications radio fréquence, des convertisseurs analogiques numériques et même des micro systèmes.

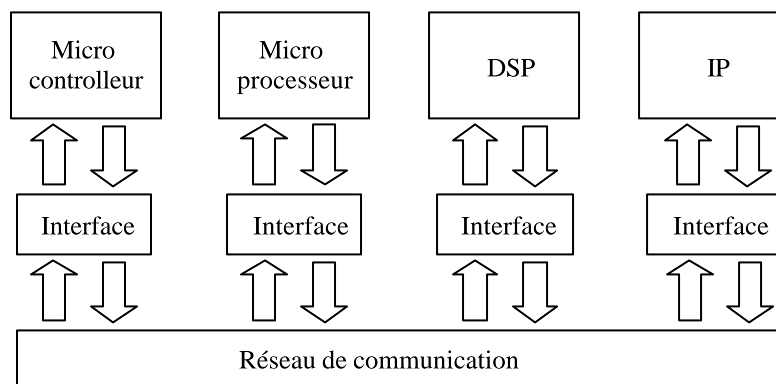

Figure 1 : Exemple de système mono puce multi processeurs

1.1.2 La conception des systèmes sur puce multiprocesseurs

On constate actuellement un véritable décalage entre la capacité d'intégration des transistors et la capacité de conception des systèmes sur puces. En effet, la progression du nombre de transistors par puce suit toujours la loi de Moore. Le rythme observé correspond à une augmentation de 58% par an. Parallèlement l'augmentation de la productivité des équipes de conception serait plutôt de l'ordre de 21%.

La conception des systèmes sur puces est difficile à plusieurs égards. Les systèmes sur puces sont généralement spécifiques à une application ou à un type d'application. Leur conception implique donc une conception spécifique à l'application. Un système sur puce est donc construit selon les contraintes propres à celle-ci. Il faut utiliser des composants spécifiques à certaines fonctionnalités. De plus, la diversité des fonctionnalités ne cesse de s'accroître. L'une des solutions adoptées pour essayer de respecter la loi du « time to market » est de faire appel à la programmation de processeurs. L'avantage du processeur par rapport à un ASIC

est d'être programmable. Ainsi l'ajout de fonctionnalités consiste simplement à programmer des processeurs préalablement validés. La diversité des processeurs (DSP, micro contrôleurs, processeurs complexes, processeurs sur mesure) permet de s'adapter aux types de fonctionnalités à implémenter. Bien sûr, certaines fonctionnalités comme les chaînes d'émission et de réception resteront implémentées par des modules matériels. Mais les systèmes multi processeurs se développent et il faut concevoir des méthodes adaptées à leur conception.

Les systèmes multi processeurs étant complexes, on ne peut les concevoir directement au niveau RTL. Il faut utiliser différentes étapes en partant de spécifications abstraites du système pour aller jusqu'au niveau RTL. La Figure 2 montre les différentes étapes de conception d'un système mono puce multi processeurs. La première étape présente la phase de programmation parallèle des tâches. L'application est décrite par un ensemble de tâches communiquant par des services de communication. La deuxième étape consiste à allouer les tâches dans les processeurs choisis. A la troisième étape, le réseau de communication est choisi. Enfin, à la dernière étape, on ajoute des interfaces logicielles/matérielles pour connecter les processeurs au réseau de communication. Ces interfaces implémentent les services de communication spécifiés à l'étape 1. Les travaux de cette thèse concernent la quatrième étape de ce flot. On part donc des hypothèses que le découpage des tâches, le choix des services de communication, l'allocation des tâches sur les processeurs ainsi que le choix du réseau de communication ont été effectués.

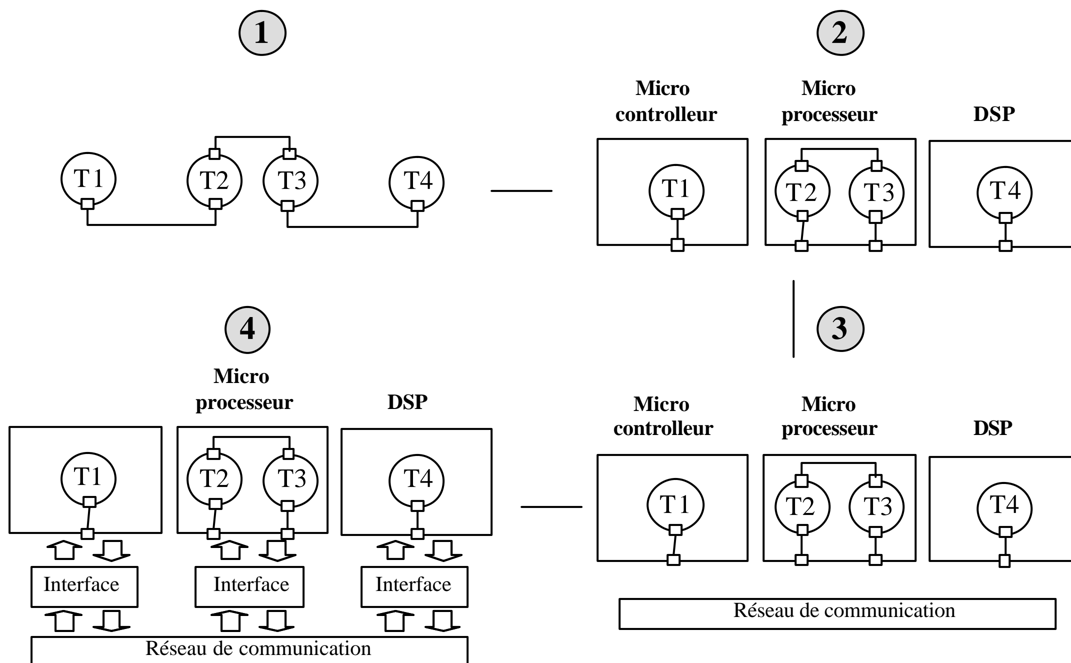

Figure 2 : Flot de conception des systèmes multi processeurs

1.1.3 Programmation parallèle et implémentation d'interfaces logicielles/matérielles pour la communication entre processeurs

Lorsqu'on parle de programmation parallèle, on considère l'exécution de tâches ou processus par un ou plusieurs processeurs. La programmation parallèle consiste en l'exécution « en même temps » de plusieurs tâches. Ces exécutions sont réellement parallèles lorsque chaque tâche possède sa propre ressource d'exécution. Mais lorsqu'on parle de programmation parallèle, on inclus aussi le pseudo parallélisme. C'est l'exécution concurrente des tâches sur une même ressource. Un système d'exploitation utilisant un ordonnanceur permet de partager le processeur entre les différentes tâches selon un algorithme d'ordonnancement. Cette thèse, traite le cas d'ensembles de tâches parallèles communicantes. Comme le montre la Figure 3 le but est d'exécuter ces ensembles sur un système mono puce constitué de processeurs communiquant via un réseau de communication et d'implémenter les services de communication sous forme d'interfaces logicielles/matérielles.

La communication entre composants est devenue un véritable goulot d'étranglement. Les performances, la consommation et les coûts de développement de ces systèmes sont fortement dépendants des choix de protocoles de communication et de leur implémentation. Dans ce manuscrit, nous utilisons plutôt le terme service de communication pour désigner des protocoles de communication complexes. Dans les cas des communications entre processeurs ou entre processeurs et IP, la réalisation d'implémentations mixtes logicielles/matérielles permettent de trouver de meilleurs compromis qu'avec des implémentions entièrement logicielles ou entièrement matérielles. Le logiciel est synonyme de flexibilité et d'évolutivité, mais aussi de forte consommation et de performances limitées. Par exemple, une expérience a été réalisée durant ces travaux, elle consistait à implémenter des services de communication avec un maximum de logiciel. Les performances de cette implémentation sont très décevantes et il est évident que ce type d'implémentations n'est pas utilisable si la performance fait partie des contraintes de l'application (plus de détails au chapitre 5). A l'opposé, le matériel permet d'obtenir de meilleures performances, d'optimiser la consommation, mais manque totalement de flexibilité. En jouant avec les avantages et les inconvénients de chacun, on peut concevoir des implémentations de protocoles de communication satisfaisant les compromis recherchés. L'automatisation de ce type de partitionnement permettrait aux concepteurs de systèmes sur puce complexes d'obtenir une optimisation rapide de leurs designs selon les contraintes imposées.

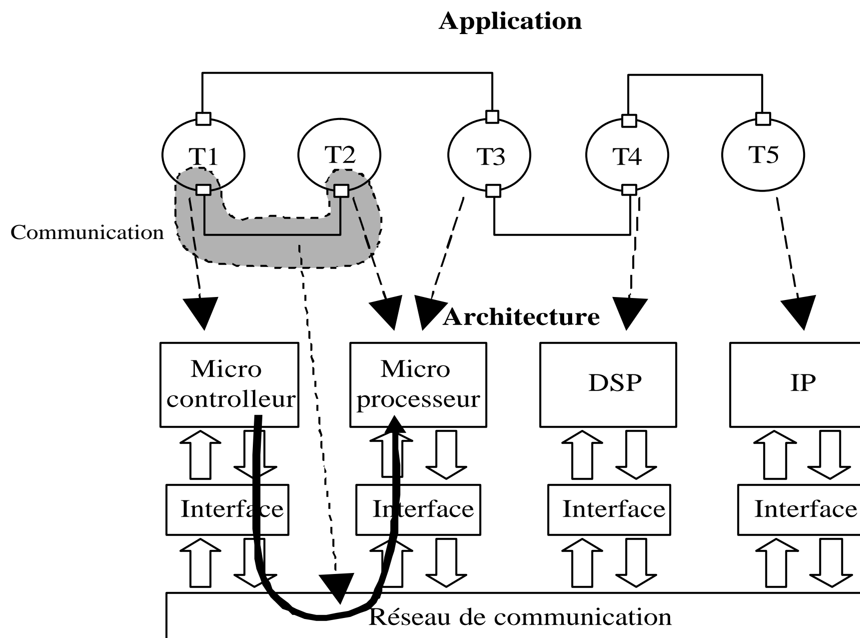

Figure 3 : Communication entre processeurs dans les systèmes mono puce

1.2 Concepts de base : conception des interfaces logicielles matérielles pour la communication entre processeurs

Cette section présente des concepts de base et des termes pouvant être utiles pour la compréhension du reste du manuscrit. Elle présente aussi un état de l'art non exhaustif de la conception des interfaces logicielles/matérielles

1.2.1 Les interfaces logicielles matérielles

Tout au long de ce document, lorsque nous parlons d'interfaces logicielle/matérielle, nous parlons d'adaptateurs de communication entre les tâches logicielles exécutées par un processeur et le réseau de communication. Ces interfaces implémentent des services de communication permettant à des tâches exécutées sur deux processeurs différents de communiquer via un réseau de communication. Leur implémentation est mixte, elle est donc décomposable en deux parties. La Figure 4 montre les deux parties composant une interface logicielle/matérielle. La partie logicielle permet aux tâches (T1, T2, T3) de s'exécuter et de communiquer en utilisant les API fournies par la partie communication et OS. Une fine couche de logiciel permet aux programmeurs de l'OS de s'abstraire du processeur. La partie matérielle comporte une partie dépendante du processeur et une partie dépendante du réseau utilisé.

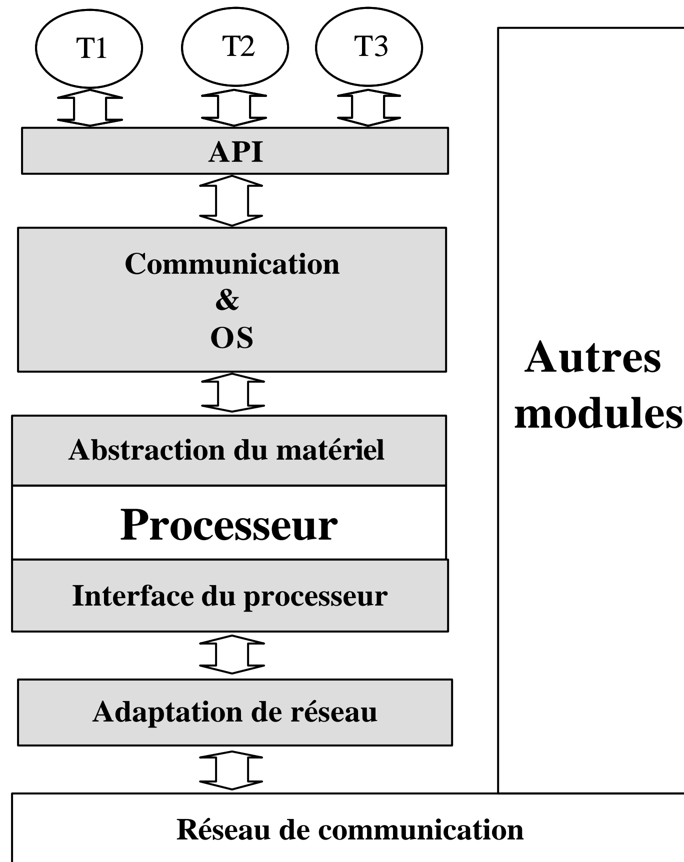

Figure 4 Les parties logicielles et matérielles d'une interface

1.2.2 Abstraction des interfaces et modèle de programmation parallèle

Le programmeur qui développe le code des tâches d'une application voit seulement les services de communication fournis par les interfaces logicielles/matérielles. L'utilisation des ces services revient à utiliser un modèle de programmation parallèle. Il existe deux principaux types de modèles de programmation parallèle. L'un permet la communication par mémoires partagées et l'autre par passage de messages.
Si l'on se place au niveau du concepteur d'interfaces on peut utiliser trois niveaux d'abstraction : le niveau transactionnel, le niveau architecture virtuelle et le niveau micro architecture.

a Les différents types de modèles de programmation parallèle

Communication par mémoire partagée : ex OpenMP [OPEN]

Plusieurs processus parallèles communiquent par accès à une mémoire partagée (Figure 5). L'avantage de ce modèle de programmation est que le programmeur n'a pas besoin de réfléchir à l'allocation des données dans les mémoires locales des différents processeurs. Le principal inconvénient de ce modèle de programmation est qu'il

impose un partage de ressources. L'implémentation d'un arbitrage de des accès à une ressource partagée peut vite coûter cher en temps de communication. Cette méthode implique une dégradation des performances et/ou l'augmentation de la mémoire utilisée pour le stockage des messages.

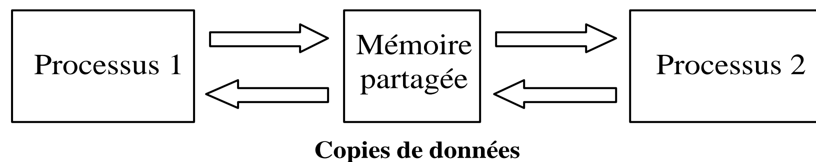

Figure 5 : Communication par mémoire partagée

Communication par passage de message : ex MPI [MPI]

Les données sont transmises entres processus sous forme de messages (Figure 6). Un message est un ensemble de données associées à des informations permettant de les traiter et de les identifier. Le gros avantage du passage de message est qu'il évite les problèmes d'accès à une mémoire commune aux processus. Il permet aussi de faire de la synchronisation entre processus si l'on utilise des primitives d'envoi de messages bloquantes.

Figure 6 : Communication par passage de message

b Les niveaux d'abstractions

Différents niveaux d'abstraction permettent de diminuer les difficultés de conception. L'utilisation de niveaux d'abstraction permet le raffinement progressif d'une spécification abstraite vers une réalisation finale du système. Dans la thèse de Gabriela Nicolescu [Nic 02] trois niveaux principaux peuvent être utilisés pour les interfaces implémentant la communication entre processeurs.

Le niveau message

Les différents modules du système communiquent via un réseau de canaux de communication dits actifs. Ces canaux permettent la synchronisation et peuvent inclure des comportements complexes. Mais les détails de la communication sont englobés par des primitives de communication de haut niveau du type MPI_SEND./MPI_RECEIVE. Tous les détails d'implémentation sont masqués.

Le niveau architecture virtuelle

A ce niveau la communication est assurée par des fils abstraits englobant les protocoles d'entrées/sorties. Un modèle de ce niveau implique le choix d'un protocole de communication et la topologie des interconnexions. Les langages caractérisant le mieux un système à ce niveau d'abstraction sont : CSP[HOA 85], SystemC 1.1 [SYS 00], Cossap[SYN 02] et StateCharts[HAR 87]. Ce modèle correspond au niveau TLM de SystemC.

Le niveau micro-architecture

La communication par des fils et des bus physiques. La granularité temporelle est le cycle d'horloge et les primitives de communication sont du type set/reset sur des ports. Les langages les plus utilisés pour modéliser un système à ce niveau sont SystemC, Verilog [MOO 98] et VHDL [IEE 93]

1.2.3 Les différents types de synchronisation entre partie logicielle et partie matérielle

a Memory mapped IO

Cette technique désigne la communication par écriture et lecture par le processeur dans des registres de la partie matérielle. Pour cela chaque registre est mappé dans une espace mémoire virtuel. Cette méthode implique s'intégrer dans l'interface un décodeur d'adresses matériel spécifique. Ce décodeur doit identifier à quel périphérique correspond chaque adresse avant l'envoi des données dans le bus.

Le processeur et la partie matérielle de l'interface accèdent tous les deux à une mémoire partagée. L'échange d'information se fait donc par « load » et « store ». L'inconvénient principal de cette méthode réside dans la nécessité d'avoir une synchronisation des accès à la mémoire ou au média permettant cet accès. Lorsque l'une des parties accède à la mémoire en lecture ou en écriture, l'autre partie ne doit pas avoir le droit d'accéder à la même zone mémoire.

b Scrutation

La scrutation correspond à une attente active du processeur par lectures successives dans un registre ou une mémoire partagée. Le gros problème de cette technique est que durant le temps de scrutation, le processeur n'exécute aucune autre instruction. Elle n'est donc utilisable que si l'on est certain que l'attente sera courte ou si le résultat attendu empêche l'exécution des autres tâches.

c Interruptions matérielles

L'interruption matérielle est un moyen de synchronisation ne nécessitant pas d'attente active de la part du processeur. La partie matérielle envoi un signal au processeur qui arrête toute exécution, change de mode et se branche sur une routine de traitement d'interruption. L'inconvénient de cette méthode est le coût apporté par l'exécution du traitement de l'interruption et du changement de contexte.

1.2.4 Etat de l'art sur la conception automatisée des interfaces de communication

Les travaux touchant de près ou de loin à l'implémentation mixte logicielle/matérielle des services de communication sont très nombreux et variés. Nous présentons donc seulement quelques travaux marquants, même si aucun n'atteint les objectifs qui ont motivé nos travaux.

a Co-design traditionnel

De nombreux travaux ont été menés sur le co-design dans les années 90 : Chinook [CHO 95], Spec Syn [GAJ 95], Vulcan [GUP 94], Polis [CHI 96], Coware [ROM 96], Cosyma [ERN 93] etc. Tous ont cherché à automatiser le co-design du système complet à partir d'une spécification de haut niveau. Certains avaient pour cible des architectures mono processeurs : Polis, Vulcan, Cosyma et d'autres les architectures distribuées avec plusieurs processeurs (Spec Syn, Coware etc.). Pour toutes ces méthodes, le système subit une décomposition fonctionnelle, puis l'ordonnancement et la synthèse des interfaces. Malheureusement la communication n'est pas l'objet du partitionnement. Il est utilisé pour le traitement du système entier. Le système découpé en fonctionnalités est partitionné en logiciel et en matériel. Ensuite les interfaces permettant la communication entre logiciel et matériel sont conçues manuellement ou synthétisées [CHO 95][GUP 93]. Ces interfaces implémentent des protocoles de communication de bas niveau tels que des interruptions, des écritures dans des registres etc. Synthèse d'interface logicielle/matérielle.

Dans notre cas l'objet du partitionnement concerne seulement les services de communications. Cette focalisation sur la communication est justifiée par la grande influence de la communication entre processeurs sur les performances du système. En effet la programmation parallèle des systèmes multiprocesseurs implique une communication intensive et l'utilisation de services de communication complexes comme ceux présentés au chapitre 4. L'exploration du partitionnement logiciel/matériel des services de communication est donc importante pour permettre la recherche de compromis entre performance, consommation et coût de développement lors de l'implémentation des applications sur systèmes mono puce multi processeurs.

O'Nils [ONI 99a] [ONI 99b] a développé une méthodologie et un outil de conception d'interfaces logicielles matérielles basée sur une modélisation grammaticale du comportement de l'interface. Cette méthode permet d'accélérer la conception d'une interface. Mais elle ne permet pas la génération automatique des interfaces, ni le partitionnement logiciel/matériel du comportement des services de communication.

b Communication entre processeurs

Il y a eu beaucoup de travaux sur l'implémentation matérielle de la communication par des réseaux communication [LAH 01]. Il a cependant eu peu de travaux sur l'implémentation logicielle/matérielle de services de communication. Brunel [BRU 00] présente un ensemble (« template ») d'implémentations mixtes logicielles/matérielles de services pour la communication entre processeurs. Nos travaux diffèrent des ceux-ci

car nous nous focalisons sur le développement d'une méthodologie permettant l'exploration du partitionnement et non de faire un ensemble d'implémentations fixes.

D'autres groupes de recherche ont cherché à résoudre notre problème mais dans le domaine de la communication entre ordinateurs. Dans [NAH 94], les services de communication sont parallélisés pour permettre leur implémentation sur des architectures multi processeurs sous forme d'interfaces de réseaux. Il y a eu aussi quelques solutions académiques et commerciales d'implémentations mixtes de réseaux de communication [DIT 01].

1.3 Problématique

La problématique à l'origine de ce travail est celui de l'exploration des solutions d'implémentations mixtes logicielles/matérielles pour la réalisation des services de communication. Les problèmes empêchant d'atteindre cet objectif sont essentiellement dû à la lenteur de conception des interfaces mixtes et à l'importance de l'espace des solutions à explorer.

1.3.1 Objectif : exploration des solutions d'implémentations mixtes logicielles/matérielles de services de communication

Comme le montre la Figure 7, l'objectif à terme est de partir d'une spécification décrivant les tâches et les services utilisés pour communiquer et d'obtenir automatiquement l'implémentation logicielle/matérielle qui satisfait de façon optimale les contraintes spécifiques à l'application. Cette implémentation sera un ensemble de processeurs connectés à un réseau de communication par des interfaces constituées d'une partie logicielle et d'une partie matérielle. Elle devra permettre d'atteindre le meilleur compromis entre performances, coût et flexibilité.

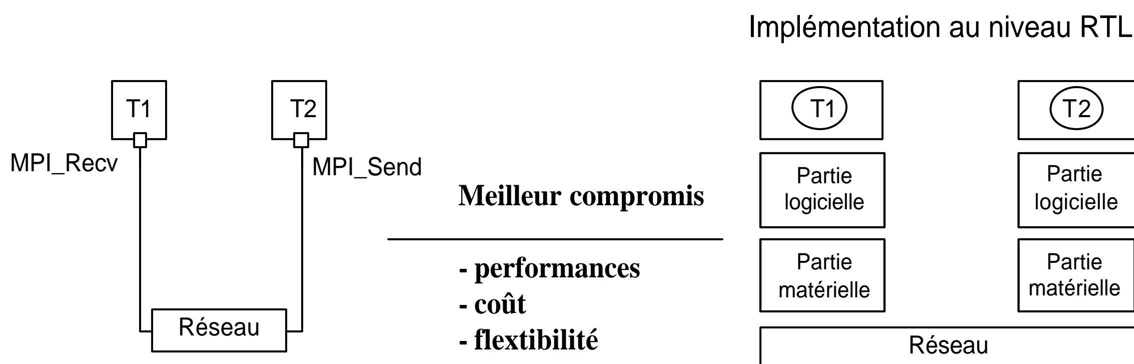

Figure 7 : Objectif

Afin d'atteindre cet objectif, on opte pour une stratégie d'exploration de solutions de partitionnement. A partir d'une spécification de service de communication, il faut pouvoir réaliser rapidement chaque implémentation jusqu'au niveau RTL. La Figure 8 montre les différentes étapes permettant de générer une implémentation mixte d'un service de communication. La première étape est celle du choix du partitionnement. Le comportement associé au service de communication est découpé en deux parties, l'une sera réalisée en logiciel et l'autre en matériel. Le résultat de cette étape sera une spécification du comportement du service de communication partitionné. La deuxième étape consiste, à partir de cette dernière spécification, à générer automatiquement une interface logicielle/matérielle implémentant le service de communication au niveau RTL. Dans le cadre de cette thèse le premier objectif à atteindre a été de permettre la réalisation de l'étape 2. Nous faisons donc l'hypothèse que l'étape 2 sera réalisable plus tard.

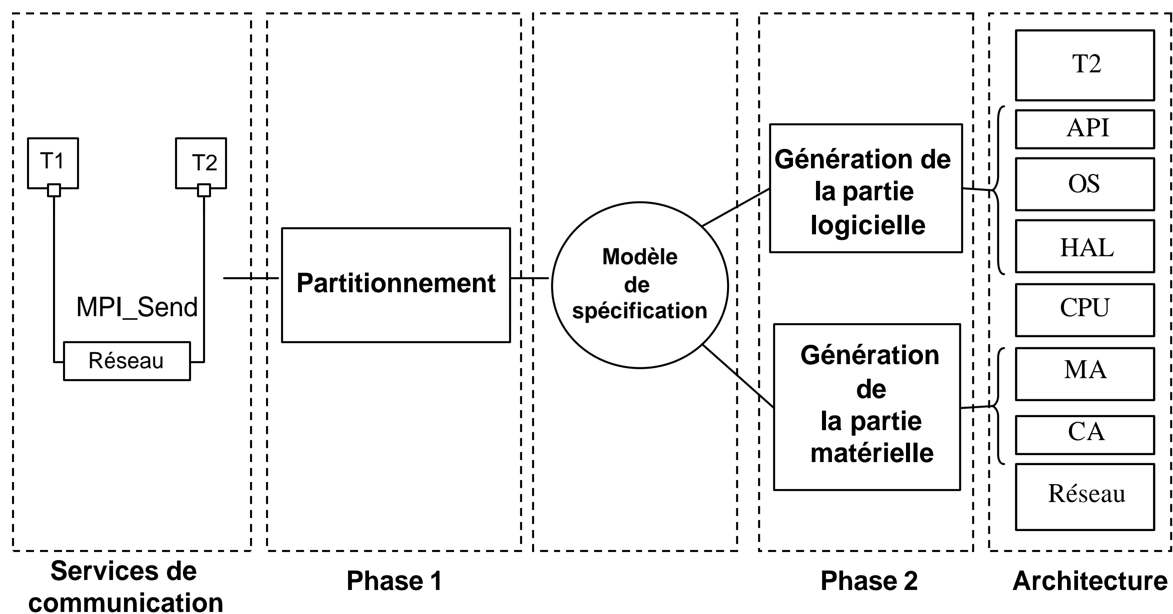

Figure 8 : Les différentes phases pour atteindre l'objectif

1.3.2 Problèmes et difficultés :

Donc si nous revenons à l'objectif à long terme, celui de la génération de l'implémentation optimale d'un service de communication, les difficultés sont essentiellement liées à la lenteur du développement manuel des interfaces logicielles matérielles.

1.3.3 Conception manuelle des interfaces

La conception manuelle des interfaces est complexe car elle demande une double compétence : celle de la conception numérique et celle du logiciel. Cette double compétence est généralement répartie sur deux équipes de développement. La difficulté de dialogue pose donc un premier problème. Pour la cohésion de l'implémentation, il faut être sûr que chaque équipe se comprend. Une erreur de compréhension, peut se voir tardivement dans le flot de conception et impliquer des efforts de recherche de bogues dans les équipes et peut impliquer des modifications coûteuses dans les deux parties. Pour éviter cela il faut qu'une coordination soit assurée par des personnes ayant une vue d'ensemble du système ainsi qu'une bonne connaissance des détails porpres aux implémentations logicielles et matérielles.

Bien souvent la conception simultanée du logiciel et du matériel est impossible. Les programmeurs de la partie logicielle ont en effet besoin de connaître l'architecture cible à programmer. Donc soit le matériel et le logiciel sont développés complètement séquentiellement, soit il faut que l'architecture de la partie matérielle, les services qu'elle doit fournir et les protocoles de communication entre le logiciel et le matériel aient été définis avant de lancer le développement du code logiciel.

Etant donné que les systèmes sur puce sont bien souvent spécifiques à une application, il est difficile d'utiliser des composants ou des architectures standards. La palette de composants utilisés dans les systèmes sur puces est très grande. Que ce soit dans le choix du processeur, des coprocesseurs de communication ou des réseaux de communication, le choix offert rend difficile l'utilisation d'un standard d'interconnexion. Le développement d'interfaces logicielles/matérielles implémentant les services de communication n'est donc pas facilité.

1.3.4 Un espace de solution trop grand pour une exploration manuelle

Comme nous l'avons vu dans la sous-section précédente, la conception manuelle des interfaces logicielles/matérielles est complexe et longue. Cette méthode étant la plus utilisée, il est difficile d'utiliser une méthode itérative de recherche de solution de partitionnement logiciel/matériel. L'espace des solutions est en effet bien trop grand. Par exemple, l'implémentation d'un simple tampon peut être déclinée selon de multiples partitionnements. Plus le comportement associé à un service de communication sera complexe, plus l'espace à explorer sera grand. Il est donc clair que l'exploration du partitionnement logiciel matériel des interfaces pour la communication ne peut être faite en utilisant une méthode manuelle.

1.4 Contribution

1.4.1 Formalisation des problèmes liés à la réalisation d'interfaces mixtes logicielles/matérielles des services de communication

La première contribution a été de comprendre et d'exposer la problématique de l'implémentation logicielle/matérielle des services de communication. Cette étude a permis de connaître les objectifs à atteindre et

d'entrevoir les évolutions à apporter au flot du groupe SLS pour permettre le partitionnement logiciel/matériel des interfaces.

1.4.2 Génération automatique des interfaces

Les contributions apportées pour la génération automatique des interfaces logicielles/matérielles sont présentées à la Figure 9. Cette figure détaille la phase 2 présentée à la Figure 8. On part d'un modèle de représentation des services de communication à partir duquel on génère une partie logicielle et une partie matérielle. Les parties grisées permettent de localiser les contributions. Elles sont présentées par les sous-sections suivantes.

a Modèle commun de représentation des services de communication pour la génération de la partie logicielle et de la partie matérielle.

Un modèle de représentation des services de communication est présenté. Il correspond à la contribution C1 de la Figure 9. Il permet au concepteur de décrire un service de communication à base d'unités fonctionnelles et d'unités de stockage de données. Ce modèle a été développé pour être utilisé par les outils de génération du groupe SLS. Plus de détails sont présentés à la section 4.2.

Le découpage fonctionnel a été testé lors d'expérimentations d'implémentations logicielles/matérielles de primitives de communication par passage de messages MPI. Elles ont permis d'appréhender la correspondance entre unités servant à décrire les services de communication et leurs réalisations en logiciel ou en matériel (voir chapitre 6).

b Extension du générateur de la partie logicielle des interfaces

La contribution notée C2 correspond à l'extension nécessaire au traitement du modèle de services de communication par un outil de génération automatique de la partie logicielle des interfaces. Cet outil, présenté au chapitre 2 a été développé par Lovic Gauthier [GAU 01]. Notre contribution est de proposer une extension de celui ci pour permettre la génération de la partie logicielle à partir du modèle de représentation des services de communication proposé. Plus de détails sont proposés au chapitre 5.

c Développement de composants pour la génération automatique d'interfaces logicielles/matérielles

Des composants ont été développés pour être utilisés par l'outil de génération automatique de la partie logicielle des interfaces. Ils correspondent aux contributions C3 de la Figure 9. Ces composants permettent la génération automatique de primitives MPI (contribution C4). La cible était une plateforme ARM intégrant 2 ARM7, 2 ARM9 un bus AMB AHB et des FPGA. La section 5.1 présente plus en détails ce développement.

D'autres primitives MPI ont été implémentées sous forme d'implémentations mixtes logicielles/matérielles. Seule la partie logicielle a été codée et découpée (contribution C4). Elle n'a pas été intégrée dans l'outil de génération automatique de partie logicielle par manque de temps.

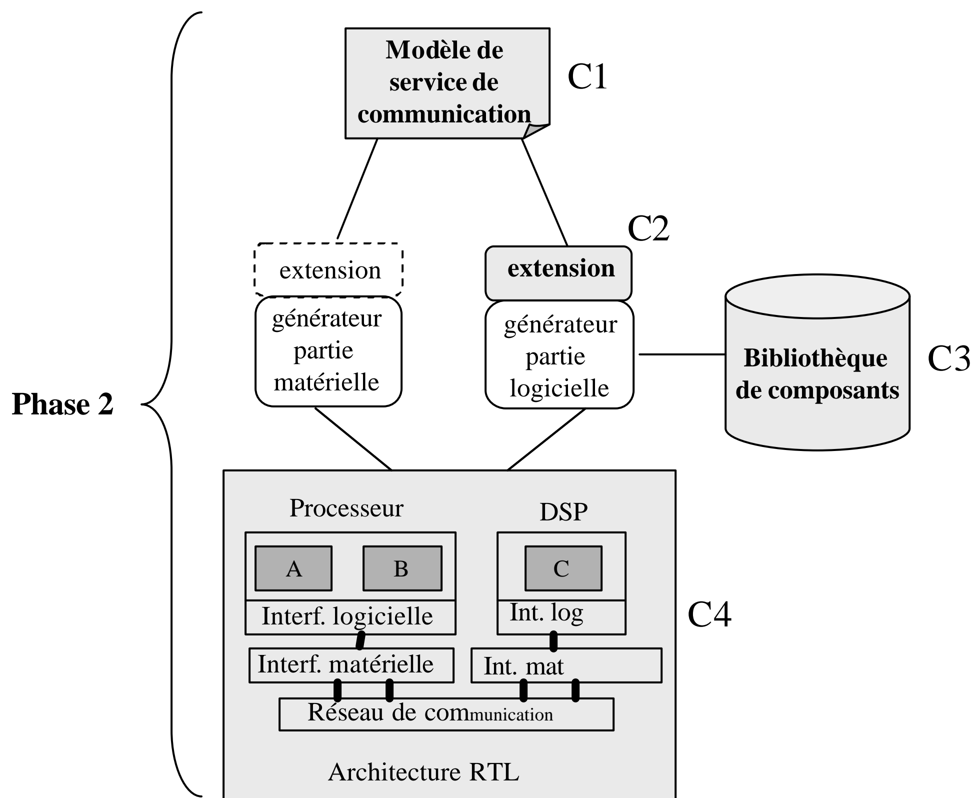

Figure 9 : Contributions pour la génération automatique des interfaces mixtes

1.5 Plan de thèse

Le chapitre 2 présente le flot ROSES développé par le groupe SLS qui constituant le contexte de ce travail. Il explique les concepts utilisés, les différentes étapes et détaille les deux outils pour le raffinement des interfaces. L'utilisation de l'outil ASOG permettant la génération de la partie logicielle des interfaces est illustrée avec un exemple. Le chapitre 3 commence par présenter l'architecture des interfaces logicielles/matérielles. Il présente aussi les primitives MPI et la sélection d'un sous ensemble utilisé lors des expérimentations. Le chapitre 4 présente les contributions de ce travail. Il détaille la problématique du partitionnement logiciel/matériel des interfaces. Il présente aussi une proposition de modèle pour la représentation de services de communication ainsi qu'une évolution du flot de génération automatique des interfaces logicielles/matérielles. Dans le chapitre 5, deux expériences sont présentées. Enfin le chapitre 6 fait un bref résumé des chapitres développés et présente quelques perspectives à ces travaux.

Chapitre 2 Environnement ROSES pour la conception des interfaces logicielles/matérielles

L'objectif de ce chapitre est de présenter les outils du flot ROSES. Les outils du flot ont été étudiés et utilisés. Leur présentation doit permettre de comprendre le cadre de développement et de recherche utilisé dans ces travaux de thèse. Un premier paragraphe présente le flot global. Les trois suivants présentent les outils de génération automatique d'OS, d'interfaces matérielles et de modèles pour la validation des systèmes par co-simulation. Le dernier paragraphe présente l'étude de l'outil de génération d'OS via le développement d'une application VDSL.

2.1 Le flot global

2.1.1 Domaine d'application : conception de couches de communication logicielles/matérielles pour les systèmes multiprocesseurs mono puce

Le flot SLS, nommé ROSES, permet la conception de systèmes mono puce à partir de composants hétérogènes. Il permet l'utilisation de plusieurs processeurs mais aussi de blocs matériels existants. Il résout les problèmes liés à la conception des systèmes complexes par la génération automatique d'interfaces de communication. La conception des parties logicielles et matérielles est conjointe. Il permet en plus une validation à plusieurs niveaux du flot par simulations. Il propose surtout la réalisation automatique d'un système à partir d'une simple spécification de haut niveau accompagnée d'une table d'allocation.

Dans un premier temps, par simplification, le flot permet uniquement de générer un OS par processeur, c'est à dire que chaque processeur possède son propre OS pour gérer la communication et la synchronisation de ses tâches. Il n'est pas possible d'obtenir un OS global gérant l'ensemble des processeurs. Cette hypothèse n'est pas très contraignante dans le cas des systèmes sur puce dédiés. En effet, la plupart des processeurs sont dédiés à des fonctions particulières et conçus pour agir de manière indépendante.

Le principe le plus important dans ce flot est la conception d'un système complet par assemblages d'éléments. Que ce soit pour composer les parties logicielles des interfaces ou les parties matérielles, mais aussi pour le développement de modèles de simulation, la technique reste la même : elle consiste en un assemblage d'éléments de bibliothèque. On retrouve cette technique à différents niveaux : le système à concevoir sera un assemblage de composants (processeurs, IP) liés par des interfaces de communication. Ces interfaces seront le résultat d'assemblages de parties logicielles et matérielles qui auront été obtenues par assemblage d'éléments de bibliothèque.

2.1.2 Le modèle de représentation COLIF

Le flot utilise un modèle de description commun à tous les outils nommé COLIF. C'est un modèle de description de systèmes hétérogènes permettant de décrire des modules logiciels et matériels. Avec COLIF, la description de la structure du système est indépendante de celle du comportement. Cela permet de modéliser les systèmes à partir de composants hétérogènes par leur niveau d'abstraction, leur langage utilisé ou encore leur type d'implémentation (ex : logicielle ou matérielle).

a Une description structurelle modulaire

La description COLIF d'un système est un ensemble d'objets interconnectés de trois types : les modules, les ports et les nets [CES 01b]. Un objet, quel que soit son type, est composé de deux parties : une interface (entity) et un contenu (content). L' «entity » permet la connexion avec les autres objets. Le «content » fournit soit une référence à un comportement, soit des instances d'autres objets. Cette possibilité d'instancier des éléments au

sein d’un objet permet le développement de modules, ports et nets hiérarchiques : de cette façon, dans un même module, on peut instancier une structure complète de modules, ports et nets.

Le module représente un composant matériel ou logiciel. Son «entity » donne son type et les ports par lesquels ils communiquent. Son « content » peut être caché (black box), contenir des références à un comportement (ex : fichier C), ou un sous-système de modules, ports et nets.

Le port représente un point de communication pour un module. Son «content » est composé de deux parties : l’une est en relation avec l’intérieur du module et l’autre avec l’extérieur. Ces deux parties peuvent être cachées, contenir des références à un comportement ou des instanciations de ports.

Le net représente une connexion entre plusieurs ports. De la même façon, un net peut contenir des références à un comportement ou contenir d’autres nets.

Dans la figure (Figure 10) deux modules A et B communiquent entre eux via des ports (petits carrés) et un net. Ils contiennent tous les deux un sous-ensemble de modules, ports et nets. Dans A nous retrouvons deux modules représentant des tâches. Dans B les modules représentent des blocs existants dont on connaît uniquement les entrées et la façon de communiquer.

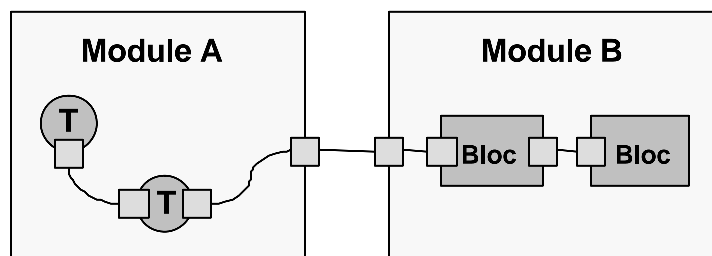

Figure 10 : Illustration des modules, ports et nets

b Les différents niveaux d’abstraction

COLIF permet pour l’instant de distinguer plusieurs niveaux d’abstraction. Dans ce travail, seulement deux niveaux sont utilisés :

Le niveau macro-architecture est le niveau spécifique à la communication par des fils abstraits, englobant des protocoles de niveau pilote (driver). Un modèle de ce niveau implique par conséquent le choix d’un protocole de communication ainsi que la topologie des interconnexions.

Le niveau RTL ou micro-architecture : la communication est réalisée par des fils et des bus physiques. La granularité de l’unité de temps devient le cycle d’horloge et les primitives de communication sont «set/reset » sur des ports et l’attente d’un nouveau cycle d’horloge.

c Composants virtuels et ports hiérarchiques

Pour permettre la connexion de composants de niveaux d'abstraction différents, COLIF utilise le concept de composants virtuels.

Un composant virtuel possède deux interfaces de communication : une interne et une externe. L'interface interne est adaptée au niveau d'abstraction du module enveloppé. L'interface externe est adaptée au niveau d'abstraction des canaux connectés au module virtuel. Pour communiquer, l'interface interne possède des ports internes et l'interface externe des ports externes (voir Figure 11). Pour envelopper ces ports dans un même protocole de communication, on les rassemble dans un port hiérarchique appelé port virtuel. Un port virtuel modélise un protocole. De la même façon deux ports virtuels sont connectés par des canaux virtuels : ce sont des canaux abstraits modélisant les protocoles utilisés.

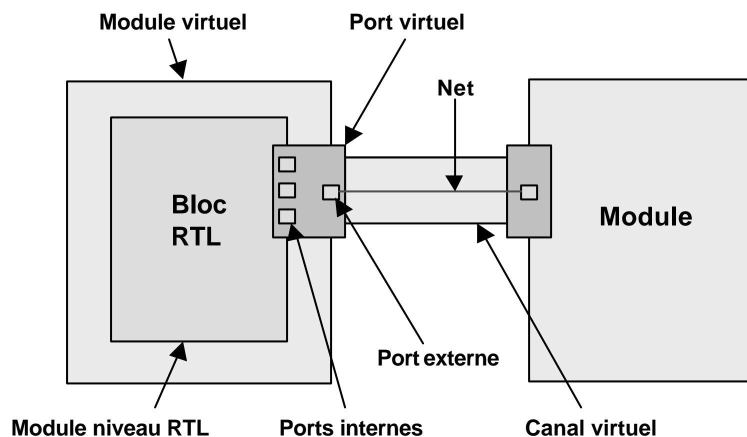

Figure 11 : Exemple de composant virtuel

2.1.3 Architecture du flot : flux d'entrées et de sorties

Le flot peut être vu comme une boite noire (Figure 12), un seul outil auquel on fournit en entrée une spécification de haut niveau pour récupérer un système complet constitué de parties matérielles synthétisables, de parties logicielles compilables sur leurs processeurs respectifs, d'une micro-architecture mais aussi d'un modèle de simulation fonctionnelle de haut niveau ainsi qu'un modèle de cosimulation au niveau RTL. Les bibliothèques peuvent être vues comme des entrées puisqu'elles sont extensibles : on peut ajouter des éléments spécifiques à une application.

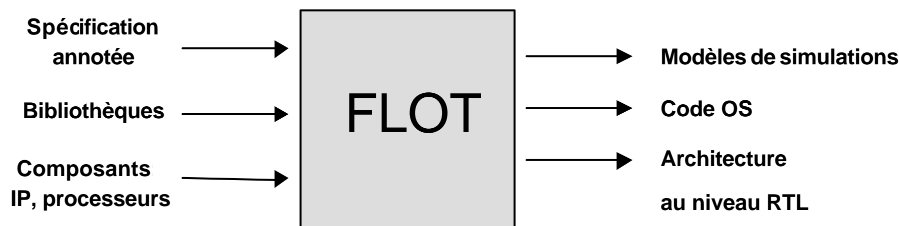

Figure 12 : Flux d'entrées sorties du flot

a Les entrées du flot : spécification annotée et bibliothèques

En entrée du flot, le concepteur fournit une spécification de haut niveau annotée de paramètres et éventuellement de nouveaux éléments de bibliothèque. Il prend aussi en entrée des composants : IP, processeurs.

Une description de haut niveau simple en C++ :

Le langage de spécification développé par le groupe SLS est une extension de SystemC [SYS 01] nommé VADeL pour Virtual Architecture Description Language pour faire référence au concept de modules virtuels décrit précédemment.

VADel est un langage construit à partir de C++ : il est donc orienté objet. Le C++, avec son concept de classes d'objets permet le développement aisé de nouvelles structures de données. En fait VADeL n'est pas une extension directe de C++ : la base était en effet SystemC, un langage de description de matériel pour la simulation. SystemC utilisait déjà les concepts de modules, nets et ports. L'extension propre à VADeL donne la possibilité de décrire les modules, nets et ports virtuels, pour la spécification de systèmes hétérogènes. Ce langage permet donc de décrire tous les objets du modèle COLIF : il donne donc au concepteur la possibilité de faire une spécification multi niveau d'un système complet incluant les parties logicielles et matérielles.

Des annotations pour le raffinement et la table d'allocation :

Entre une spécification de haut niveau et son implémentation, une foule d'informations est ajoutée. Par exemple, dès que l'on utilise un processeur et des IP, il devient nécessaire de concevoir une table d'allocation des ressources : il faut donner des adresses aux ressources matérielles mais aussi aux ressources logicielles. Il faut indiquer certaines caractéristiques de ces ressources. Il faut aussi pouvoir préciser les services dont une tâche a besoin. Toutes ces informations sont ajoutées à la spécification, par le concepteur, grâce à un système d'annotation de paramètres.

Bibliothèques :

Il existe une bibliothèque VADeL pour la spécification, des bibliothèques d'éléments logiciels, une autre d'éléments matériels ainsi que des bibliothèques pour la simulation. Elles constituent des entrées au sens où il est possible de les modifier et de les étendre. Cependant, avec des bibliothèques suffisamment importantes, il ne devrait pas être nécessaire au concepteur de s'en occuper.

b Les sorties du flot : interfaces logiciels/matériels et modèles de simulation

A la sortie du flot, on retrouve des fichiers contenant le code exécutable de la partie logicielle, le code synthétisable de la partie matérielle ainsi que les fichiers de simulation.

La partie logicielle :

Pour chaque processeur, on obtient le code des tâches ainsi que l'OS spécifique. Le code des tâches est en C++ et celui des OS générés est en C. Ces codes sont compilables sur leurs processeurs respectifs.

Les interfaces matérielles :

Chaque élément de ces interfaces est décrit en VHDL ou en SystemC synthétisable.

Une micro-architecture COLIF :

C'est une description en COLIF de la structure du système après la génération des interfaces. La micro architecture correspond à une description structurelle au niveau RTL.

Les modèles pour la simulation :

Pour permettre une validation de la spécification au niveau macro-architecture, un modèle exécutable de la spécification est généré [NIC 01b]. Ce modèle est écrit en SystemC. Il permet par exemple au concepteur de mesurer les effets de choix de protocoles de communication sur le fonctionnement de son système.
Un autre modèle de simulation permet la validation du système au niveau micro-architecture. Nous obtenons en fait un ensemble de fichiers SystemC permettant une cosimulation du système : il est alors possible de faire simuler en parallèle les composants matériels décrits en VHDL sur un simulateur VHDL, les parties logicielles sur des ISS et les IP sur des simulateurs appropriés à leurs langages de description.

c Algorithme du flot

La Figure 13 montre la partie du flot permettant le raffinement d'un système décrit au niveau macro-architecture vers une description au niveau RTL.

La première étape consiste à traduire la spécification annotée en COLIF. A partir de ce modèle, vient une étape de génération d'interfaces matérielles. Un autre outil permet la génération des interfaces de communication logicielles. Il existe aussi un outil de génération d'enveloppes de simulation utilisable à partir du modèle de macro-architecture mais aussi de la micro-architecture.

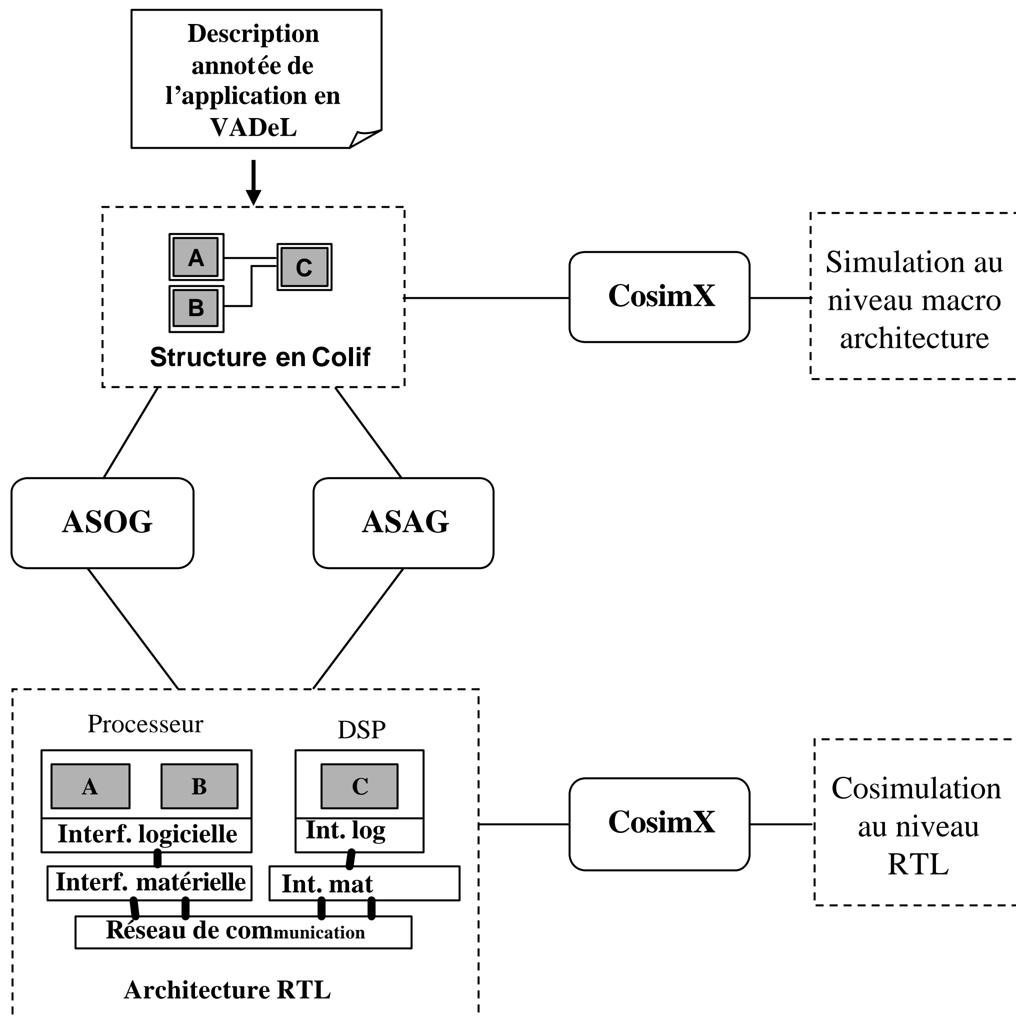

Figure 13 : Flot simplifié de ROSES

2.2 ASOG : un outil de génération automatique d'OS

Cet outil a été développé par Lovic Gauthier dans le cadre de sa thèse [GAU 01]. Ce paragraphe présente les concepts de base, les mécanismes, la structure et la composition de la bibliothèque. Enfin le flot de génération automatique d'OS sera décrit en détails avec la présentation des entrées et sortie ainsi que les principales étapes de la génération.

2.2.1 Concept : le ciblage logiciel

Le ciblage logiciel consiste à produire une description exécutable (pour une architecture donnée) d'une application logicielle, à partir d'une ou plusieurs descriptions de haut niveau de cette dernière, ne contenant pas

de détails d'implémentation. L'outil de génération automatique d'OS permet un ciblage logiciel puisqu'il fourni la couche logicielle permettant l'exécution de l'application sur l'architecture cible.

2.2.2 Les mécanismes d'assemblage

La génération du code du système d'exploitation consiste en l'assemblage et en la paramètrisation de morceaux de code d'une bibliothèque. Les morceaux de code sont encapsulés dans des macros. Leur expansion avec des paramètres adéquats permet la génération du code du système d'exploitation. Seuls les éléments nécessaires à l'application sont sélectionnés pour être expansés.

2.2.3 Structure de la bibliothèque

La bibliothèque est composée de deux parties : un ensemble de fichiers en langage macro et un fichier de description des dépendances entre les éléments.

a Les objets et la description de leurs dépendances

Le langage de description de dépendance

Le langage utilisé a été développé au sein du groupe. C'est un langage de description structurelle nommé LiDeL (pour « Library Description Langage »). Il est composé d'un ensemble de structures de donnée manipulées par des API.

Les concepts de base pour la description

La description est basée sur trois concepts fondamentaux donnant chacun une vue différente du système d'exploitation : l'élément, le service et l'implémentation.

L'**élément** représente une partie de l'OS, c'est une brique de base. Cependant cette notion ne correspond pas directement à un bout de code généré. Ce terme désigne un type de composant non spécialisé (implémentation) au sens ou il n'est pas forcement dédié à une architecture particulière.

Le **service** représente une fonctionnalité du système. C'est une notion abstraite qui permet de diviser et de structurer le comportement fonctionnel de l'OS. Les services sont fournis par des éléments, mais un élément peut aussi requérir un service.

L'**implémentation** représente une réalisation particulière d'un comportement. Un élément peut posséder plusieurs implémentations. Les implémentations désignent des objets concrets : à chaque implémentation correspond une portion de code générique de l'OS.

Les objets

La description de la bibliothèque est composée d'un ensemble d'objets. Ils correspondent à différents types de structures de données différenciées par le type d'information qu'elles apportent. Les trois concepts présentés précédemment, constituent des objets de la bibliothèque, mais ils ne sont pas seuls. On peux classer ces objets en

trois grands types : les objets symboliques, les objets de type « brique de base » et les objets d'informations pour le ciblage.

Les objets symboliques :

- Les **services** : ils désignent des fonctionnalités, ils sont représentés par des chaînes de caractères.
- La **fonction** : elle décrit la fonction d'une source pouvant être appelée par le système d'exploitation généré ou par l'application. Sa description contient les informations suivantes :
  - le nom de la fonction, simple chaîne de caractères
  - le type de la fonction : ce type peut être une fonction appelée par un appel de fonction classique, fonction système (appelée par une trappe système) et fonction d'interruption (appelée suite à une interruption)

Les objets de type « brique de base » :

- L'**élément** : il représente une partie du système d'exploitation. Un élément contient les informations suivantes :
  - un nom unique permettant de l'identifier
  - une liste de services fournis par l'élément
  - une liste de services requis par l'élément
  - un arbre d'implémentations de l'élément
- L'**implémentation** : elle représente une réalisation possible d'un élément. Une implémentation contient les informations suivantes :
  - une liste de sources composant l'implémentation
  - une liste de processeurs avec lesquels l'implémentation est compatible
  - une liste de médias qui indiquent des choix d'implémentation
  - une liste d'implémentations filles, qui donnent une structure d'arbre aux implémentations d'un élément

Les objets d'informations pour le ciblage :

- Le **code source** : il contient les informations suivantes :
  - un langage, indiquant dans quel langage de programmation le code généré correspondant sera écrit
  - une liste de fichiers de macro-code source (liste de chaînes de caractères)
  - une liste de fichiers de macro-entête (liste de chaînes de caractères)
  - une liste de fichiers de macro à utiliser pour générer les éléments qui ont besoin d'un service fourni par l'élément correspondant à la source
  - une liste de fonctions, décrivant les fonctions d'interface fournies par la source : ces fonctions peuvent être des appels système ou des fonctions d'interruption (ISR)

- Le **média** : il symbolise un choix d'implémentation pour l'architecture matérielle ou logicielle. Par exemple, une communication entièrement logicielle utilisera le média Soft, tandis qu'une communication par registre matériel utilisera le média Register. Dans la bibliothèque, les médias sont des chaînes de caractères.

- Le **processeur** : il représente un processeur ou une famille de processeurs sur lesquels pourra s'exécuter le système d'exploitation généré. Il contient les informations suivantes :
  - le nom du processeur, simple chaîne de caractères
  - une liste de compilateurs
  - une liste d'éditeurs de liens
  - une liste de convertisseurs
  - une liste de processeurs fils utilisés dans le cas des familles hiérarchiques de processeurs

- Le **langage** : il représente un langage ou une famille de langages de programmation avec lesquels peut être écrite une source. Il contient les informations suivantes :
  - le nom du langage, simple chaîne de caractères
  - une liste de compilateurs compilant le langage
  - Une liste de langages fils utilisés dans le cas des familles hiérarchiques de langages

- Le **compilateur** : il représente un compilateur pouvant compiler des sources dans un langage pour un ou plusieurs processeurs. Sa description contient les informations suivantes :
  - le nom du compilateur, simple chaîne de caractères
  - une liste de langages supportés
  - une liste de processeurs supportés
  - diverses informations sur les options à utiliser pour effectuer la compilation

- L'**éditeur de liens :** il donne les informations permettant de lier des programmes compilés pour un ou plusieurs processeurs. Sa description contient les informations suivantes :
  - le nom de l'éditeur de liens, simple chaîne de caractères
  - une liste de processeurs supportés
  - diverses informations sur les options à utiliser pour effectuer l'édition de liens

- Le **convertisseur** : c'est en fait un outils d'aide au ciblage. C'est un programme pouvant convertir un programme d'un format binaire en un autre. Sa description contient les informations suivantes :
  - le nom du convertisseur, simple chaîne de caractères
  - une liste de processeurs supportés
  - des informations sur les formats supportés en entrée et en sortie
  - diverses informations sur les options à utiliser pour effectuer l'édition de liens

b Les éléments de macro

Les éléments de macro sont des fichiers écrits en langage de macro. Une fois assemblés et expansés, ils forment le code de l'OS.

Le langage utilisé est appelé Rive, c'est un langage développé par Lovic Gauthier. Il offre à peu près les même pouvoir d'expression que le langage m4 [M4]. Il a été préféré à ce dernier car plus facile à écrire. Ce langage s'accompagne d'un outil du même nom qui prend en entrée des fichiers texte écrits en Rive, des paramètres mais aussi des macros. Le fichier de sortie est un fichier texte. Ce type de langage et d'outils permet d'écrire n'importe quel type de code puisqu'il ce contente d'éditer du texte. L'avantage de la descriptions d'élément en macro est d'avoir des élément génériques : selon les paramètres utilisés, le code expansé ne sera pas le même. Un exemple d'expansion est visible en annexe.

2.2.4 Le flot de génération automatique d'OS

a Les entrées et sorties

En entrée :

Le flot prend en entrée une description du système en COLIF, le code des tâches, la description des dépendances de la bibliothèque en LiDeL (un langage développé en interne) et l'ensemble des éléments en langage de macro. La description COLIF contient les besoins de services pour l'application et tous les paramètres pour l'implémentation.

En sortie :

Le flot produit en sortie le code du système d'exploitation adapté à l'architecture et à l'application. Ce code est constitué de plusieurs fichiers pouvant être de divers langages comme le C le C++ ou les langages d'assemblage. Il produit aussi les fichiers de compilation : ce sont les fichiers qui permettent l'automatisation de la compilation. Ces fichiers sont dans le format Makefile, compréhensible par le programme d'automatisation de commandes make [MAK]. Enfin le flot produit en sortie des comptes rendus sur les opérations effectuées : rappel des actions effectuées durant le ciblage et erreurs survenues durant le ciblage.

b Les étapes

L'ensemble des étapes est présenté par la Figure 14. Les actions exécutées lors de ces étapes sont exposées dans la suite du paragraphe.

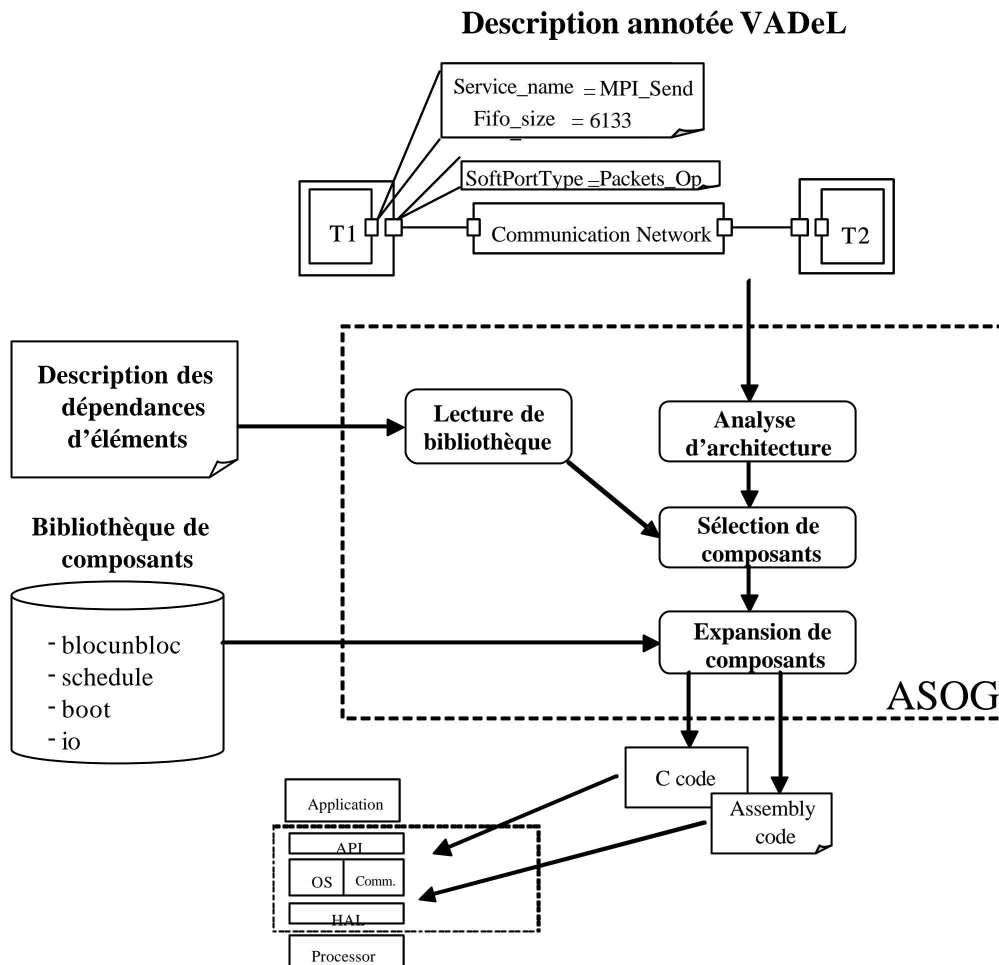

Figure 14 : ASOG

Lecture de bibliothèque :

- Création de tables pour chaque type d'objets composant la bibliothèque (éléments, services etc.)
- Parcours des éléments et création d'un arbre de dépendances entre les différents objets.

Analyse d'architecture :

- Recherche les processeurs.
- Pour chaque processeur :
    - Extrait les paramètres du processeur.
    - Recherche les tâches et ports.
- Extrait et regroupe les services et paramètres des ports et des tâches en unités de générations.

Sélection de composants :

- Pour chaque unité de génération :
  - Construit le graphe éléments/services à partir des services de base.
  - Élimine les éléments/services invalides pour l'architecture.
  - Produit la liste des éléments nécessaires et suffisants.

Expansion de composants :

- Pour chaque élément :
  - Rassemble tous les paramètres le concernant.
  - Appelle l'expanseur de macros pour les sources de l'élément avec les paramètres.

Génération de Makefiles :

- Pour chaque source de l'OS :
  - Génère les règles de compilation.
- Pour chaque source des tâches.
  - Génère les règles de compilation.
  - Génère les règles d'édition de liens.

2.3 ASAG : génération automatique de la partie matérielle

2.3.1 Entrées et sorties

Cet outil prend en entrée, comme ASOG, la description structurelle de l'application en COLIF ainsi que les paramètres annotés. Il utilise aussi deux bibliothèques. L'une contient des structures génériques d'interfaces et l'autre des fichiers en langage macro décrivant le comportement des composants assemblés.

En sortie, on obtient une description COLIF au niveau RTL de l'architecture de l'interface et de celle de l'architecture locale du processeur. On obtient aussi le code synthétisable VHDL ou systemC de l'architecture.

2.3.2 Les bibliothèques

a La bibliothèque de structures :

Cette bibliothèque contient deux types de composants. Des composants propres au processeur utilisé, ce sont des architectures génériques locales au processeur. Les autres composants sont des architectures d'adaptateurs de communication. Ces composants sont décrits en COLIF.

b La bibliothèque de comportements :

Cette bibliothèque contient des fichiers écrits en langage macro décrivant le comportement des composants de l'interface. Il existe plusieurs version d'un même composant selon si il doit être expansé en VHDL ou en systemC.

2.3.3 Les étapes

La Figure 15 montre le flot de conception de génération des interfaces matérielles. Il est constitué de 5 étapes dont le comportement est détaillé ci-dessous.

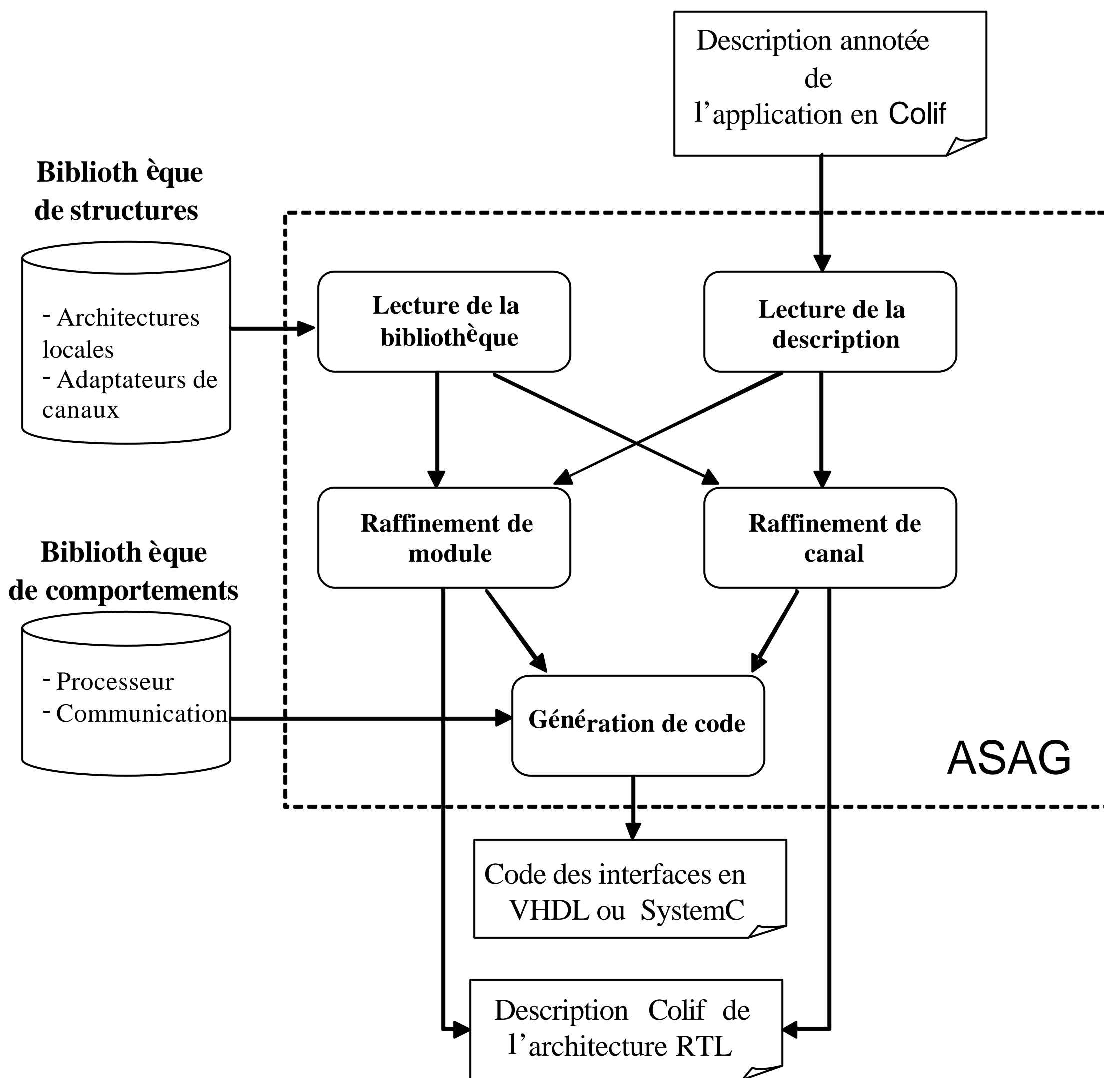

Figure 15 : ASAG

a Analyse de la description de l'architecture

L'outil prend en entrée une description de l'architecture logicielle et matérielle en format COLIF sous forme d'un fichier XML. Cette étape consiste à récupérer les paramètres de configuration qui annotent la spécification d'entrée.

b Lecture de la bibliothèque d'architectures internes

Cette étape consiste à charger la bibliothèque de structures. Cette lecture permet de connaître la disponibilité des ressources architecturales nécessaires pour la réalisation de l'architecture cible. La bibliothèque contient différents composants de l'architecture cible. La structure de chaque composant de cette bibliothèque est décrite en COLIF. A chaque structure est associé un lien vers un composant de la bibliothèque de comportement.

c Raffinement d'un nœud

Un nœud de calcul est décrit comme un module virtuel dont l'interface est composée de ports virtuels hiérarchiques regroupant deux types de ports : des ports internes spécifiques à la fonctionnalité du nœud de calcul et des ports externes spécifiques aux canaux de communication. Le raffinement d'un nœud de calcul consiste à :

- Choix et configuration d'une architecture interne à partir de la bibliothèque d'architecture en COLIF en utilisant les paramètres d'annotation attachés au module virtuel.
- Définition de la structure de l'adaptateur de canal virtuel de l'architecture interne. En effet, selon les paramètres d'annotation attachés aux ports internes et aux ports externes, on fixe le nombre et l'interface des adaptateurs de canaux (CA) qui vont implémenter l'adaptateur de canal virtuel. Le choix d'un CA dépend des paramètres liés au protocole de communication et type de bus interne. Un CA peut être dupliqué selon un paramètre appelé « prolifération » qui peut annoter un module, un port ou un canal.

d Raffinement de canal virtuel

Le raffinement d'un canal virtuel est composé de deux parties :

- Une analyse de tous les ports liés au canal permet de donner une liste d'attributs caractérisant la spécificité du canal à sélectionner à dans la bibliothèque.
- Une partie sélection utilise la liste précédente pour choisir un modèle de canal parmi les différents modèles disponibles dans la bibliothèque. Selon le paramètre « proliferation », le canal comme son adaptateur peuvent être également répliqués.

e Génération du code :

A chaque instance d'élément de l'architecture interne (sauf le processeur et les mémoires locales), correspond un modèle d'implémentation générique décrit en langage macro. Une implémentation générique n'est ni

simulable ni synthétisable car elle n'est pas encore configurée. Le type de données est abstrait, le nombre, la taille et la direction des ports sont encore génériques. Pour une application donnée, on utilise le paramètre de lien vers les sources du comportement et d'autres paramètres de configuration (type de données, taille du port, etc.) pour configurer le code générique sélectionné à partir de la bibliothèque comportementale. Cette expansion du code génère le code final de l'adaptateur de module.

2.4 Cosimx : cosimulation

2.4.1 Utilité de l'outil Cosimx

CosimX a été développé par Gabriela Nicolescu [NIC 02]. Il est utilisé pour valider les systèmes à plusieurs niveaux d'abstraction. Il permet de rendre exécutable la description VADeL du système. L'exécution du système avant génération des interfaces permet de valider le comportement des tâches, celui IP ainsi que le choix des services de communication.
CosimX offre aussi la possibilité de tester le fonctionnement du système après le raffinement partiel ou complet du système.

2.4.2 Principes de fonctionnement

Cet outil utilise aussi des bibliothèques et prend en entrée une spécification COLIF du système. Les ports et les canaux non raffinés sont remplacés par des implémentations en systemC au niveau fonctionnel des services de communication.

Lorsque des parties du système ont été raffinées, CosimX utilise un environnement de cosimulation pour exécuter les parties raffinées sur des simulateurs (ISS pour le logiciel, simulateurs VHDL etc.). Cet environnement est composé d'adaptateurs de simulateur et d'un bus de cosimulation en systemC

2.5 Illustration du flot avec l'application VDSL

Cette section a pour but de donner au lecteur une vision plus précise de flot ROSES via la réalisation d'une application VDSL. L'accent est surtout mis sur l'utilisation de l'outil ASOG puisque c'est l'outil qui a été le plus étudié dans cette thèse. La première partie correspond au chapitre 5 [PAV 03] de l'ouvrage publié par l'auteur dans [GAU 03]: il présente l'application VDSL et l'expérimentation de l'outil ASOG aussi qualifié d'outil de ciblage automatique du logiciel pour les systèmes multi processeurs mono puce. La partie matérielle des interfaces est détaillée dans la thèse [LYO 03].

2.5.1 Description d'une application : un framer VDSL

Le VDSL (Very-high-data-rate DSL) est une technique de communication utilisant les lignes téléphoniques. Elle fait partie des techniques xDSL (Digital Subscriber Line). Cette technique est encore au stade du prototype, et de nombreuses entreprises proposent leur propre version du protocole VDSL.

La version VDSL utilisée dans ce chapitre vient de [MES 00]. Cette version utilise le codage DMT (Discret Multi-Tone) qui découpe la bande de fréquence initiale en sous-bandes de transmission simultanée. Le Zipper découpe la bande de fréquence en 2048 sous-bandes qui peuvent être allouées dynamiquement, et par logiciel, à différents utilisateurs. Un prototype de modem utilisant cette technique a été développé. C'est une partie de ce modem que nous allons étudier ici.

a Le sous-ensemble de l'application VDSL utilisé pour la démonstration

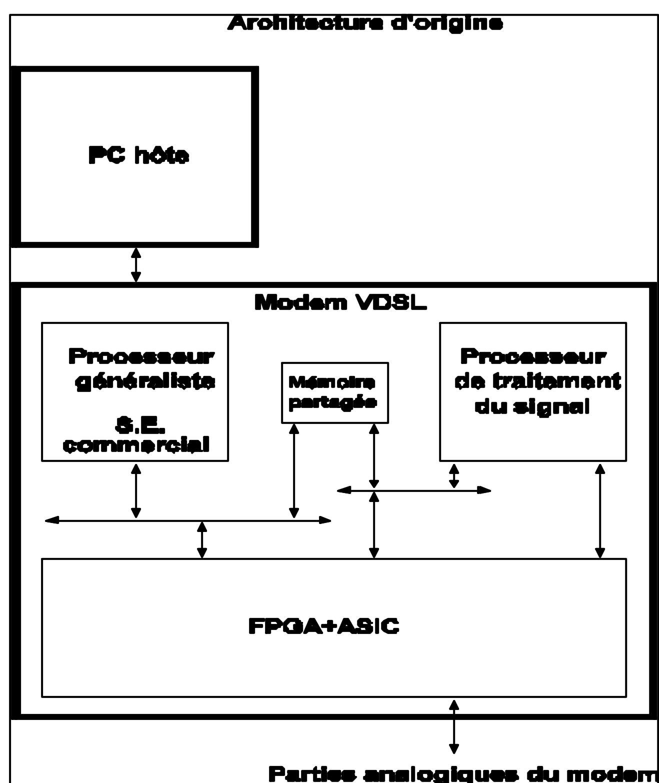

Figure 16 : La partie numérique du modem VDSL

L'architecture du modem est constituée d'ASIC et de FPGA pour le traitement du signal fixe, d'un processeur spécifique pour le traitement du signal configurable, et d'un processeur généraliste pour le contrôle du modem et l'interface avec le PC hôte. Pour gérer les nombreuses tâches du processeur de contrôle, un système d'exploitation commercial avait été choisi. L'architecture de la partie numérique de ce modem est présentée dans la Figure 16.

Cette section utilise un sous-ensemble de l'architecture VDSL avec modification de l'architecture de la partie contrôle. Il a donc été proposé de montrer l'utilisation du flot de conception pour remplacer une partie des ASIC par un autre processeur, et pour générer les systèmes d'exploitation. La Figure 17 illustre les changements

attendus sur l'architecture du modem. La démonstration se limite à la partie grisée de la figure, c'est-à-dire la partie «framer» du VDSL.

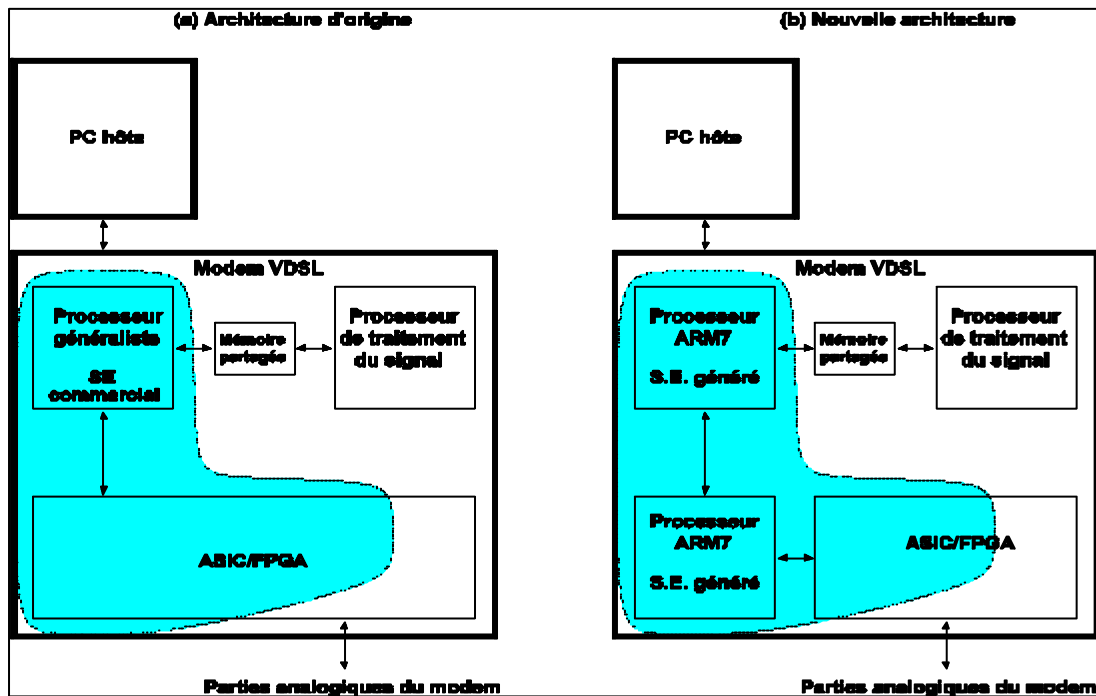

Figure 17 : La démonstration VDSL

Cette démonstration servira à analyser la qualité du système d'exploitation généré. La méthode de ciblage doit aussi permettre une séparation plus claire entre le système d'exploitation et l'application logicielle. En effet les systèmes d'exploitation embarqués classiques proposent rarement toutes les fonctionnalités nécessaires pour une application donnée. Ces fonctionnalités sont donc ajoutées sous la forme de tâches supplémentaires ou de routines d'interruptions, ce qui rend le code de l'application confus et potentiellement instable. La souplesse de la bibliothèque du système d'exploitation doit permettre d'ajouter facilement de nouvelles fonctionnalités aux systèmes d'exploitation sans avoir à les transférer dans le code de l'application.

Enfin il s'agira aussi d'analyser la flexibilité de l'approche. On s'intéressera plus particulièrement à

- la possibilité d'utiliser plusieurs fois un même composant logiciel dans des contextes différents. Dans de nombreux outils, cette source d'ambiguïté n'est pas levée, ce qui interdit l'utilisation multiple d'un même composant.
- la possibilité de gérer des communications multipoints.
- la possibilité de réduire au maximum la taille de la bibliothèque proportionnellement au nombre de possibilités qu'elle offre.

2.5.2 Spécification de l'application

a Construction de la spécification

La spécification VDSL disponible comprenait la description de l'architecture, une description informelle des différentes tâches logicielles, et la spécification en C++ du framer.

Cet ensemble de spécifications n'était pas directement utilisable dans le flot. Nous les avons donc traduites en une spécification SystemC compatible avec notre flot. Le framer a été découpé en deux parties : une partie devant être remplacée par un processeur ARM7, et l'autre conservée en tant qu'ASIC. Le processeur d'origine a lui-même été remplacé par un processeur ARM7. Enfin, la partie logicielle d'origine a été redéfinie en ne conservant que les parties contrôlant le framer.

b La spécification que nous avons utilisée dans le flot

La Figure 18 donne l'architecture globale d'application que nous avons étudiée. Elle est constituée de trois modules : M1 est le processeur de contrôle et d'interface entre le framer et le reste du modem, c'est lui qui remplace le processeur d'origine. M2 est le processeur de configuration du flot de données du framer, il remplace une partie des ASIC du circuit. Enfin, M3 est l'ASIC du flot de données du framer.

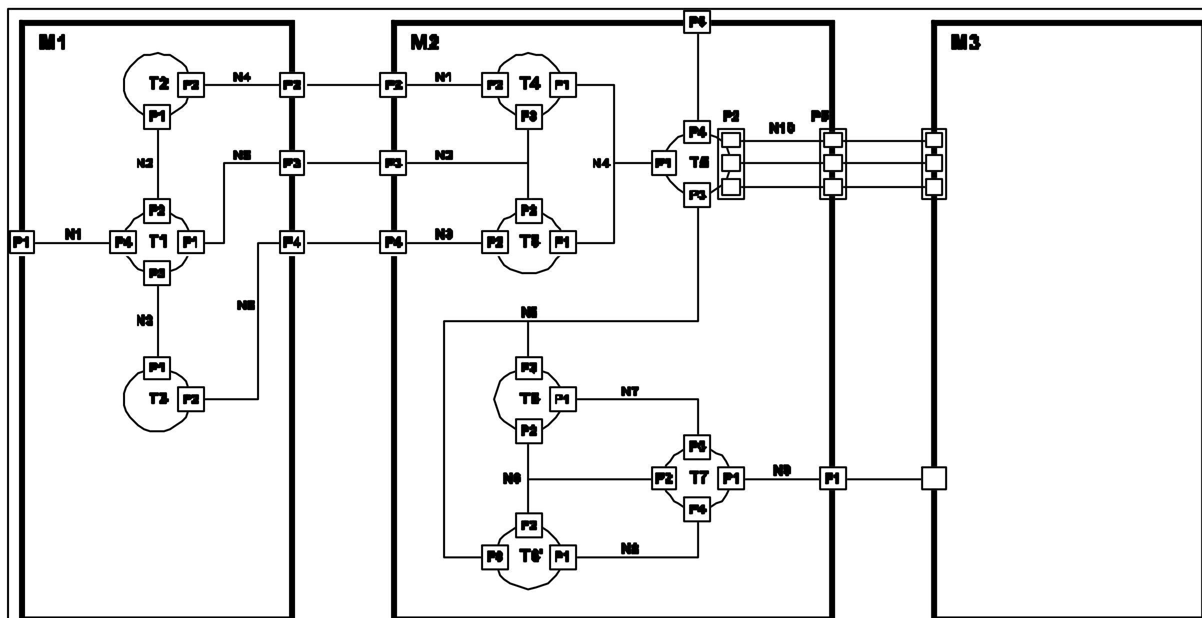

Figure 18 : L'application VDSL

Objets			Paramètres
M1			Processeur : ARM7 Allocation : IT=interrupt:0:256 Allocation : MEM=char:0:65536
		P1	Média : interrupt Interruption : 0:(IT)
		P2	Média : LockedRegister Interruption :1:(IT) type donnée : long int Adresse donnée : 0xF00000 Adresse état : 0xF00004
		P3	Média : LockedRegister Interruption : 2: (IT) type donnée : long int Adresse donnée : 0xF00008 Adresse état : 0xF0000C
		P4	Média : LockedRegister Interruption : 3:(IT) type donnée : long int Adresse donnée : 0xF00010 Adresse état : 0xF00014
		N2	Interruption : (1):IT
		N3	Interruption : (1):IT
	T1		Priorité : 0 Sources : M1/T1.cpp
		P1	Service : API/IO/Pipe/Get
		P2	Service : API/Synchronization/Signal/Notify
		P3	Service : API/Synchronization/Signal/Notify
		P4	Service : API/Synchronization/Signal/Wait
	T2		Priorité : 0 Sources : M1/T2.cpp
		P1	API/Synchronization/Signal/Wait
		P2	API/IO/Pipe/Put
	T3		Priorité : 0 Sources : M1/T3.cpp
		P1	API/Synchronization/Signal/Wait
		P2	API/IO/Pipe/Put

Figure 19 : Paramètres de l'application VDSL pour le module M1

Objets		Paramètres
M2		Processeur : ARM7 Allocation : IT=interrupt:0:256 Allocation : MEM=char:0:65536
	P1	Média : Buffer Interruption : 0:(IT) type donnée : long int Taille : 64 Addresse donnée : 0xF00000 Addresse état : 0xF00004
	P2	Média : LockedRegister Interruption : 1:(IT) type donnée : long int Addresse donnée : 0xF00008 Addresse état : 0xF0000C
	P3	Média : LockedRegister Interruption : 2:(IT) type donnée : long int Addresse donnée : 0xF00010 Addresse état : 0xF00014
	P4	Média : LockedRegister Interruption: 3:(IT) type donnée : long int Addresse donnée : 0xF00018 Addresse état : 0xF0001C
	P5	Média : Register type donnée : long int Addresse donnée : 0xF00020
	P6	Média : LockedRegister type donnée : long int Addresse donnée : 0xF00024
	N4	type donnée : long int Taille : 16 Addresse donnée : (64):MEM
	N5	type donnée : long int Taille : 1 Addresse donnée : (4):MEM
	N6	type donnée : long int Taille : 64 Addresse donnée : (256):MEM Interruption : (1):IT
	N7	Interruption : (1):IT
	N8	Interruption : (1):IT
T4		Priorité : 0 Sources : M2/T4.cpp
	P1	Service : API/IO/Pipe/Put
	P2	Service : API/IO/Pipe/Get
	P3	Service : API/IO/Pipe/Put
T5		Priorité : 0 Sources : M2/T5.cpp
	P1	Service : API/IO/Pipe/Get
	P2	Service : API/IO/Over/Put
	P3	Service : API/IO/Memory/Attach
	P4	Service : API/Synchronization/Timer/SetVal Service : API/Synchronization/Timer/Wait
T6, T6'		Priorité : 0 Sources : M2/T6.cpp
	P1	Service : API/Synchronization/Signal/Wait
	P2	Service : API/IO/Memory/Attach
	P3	Service : API/Synchronization/Semaphore/P Service : API/IO/Memory/Attach
T7		Priorité : 0 Sources : M2/T7.cpp
	P1	Service : API/IO/Pipe/Put
	P2	Service : API/IO/Memory/Attach Service : API/Synchronization/Semaphore/V
	P3	Service : API/Synchronization/Signal/Notif y
	P4	Service : API/Synchronization/Signal/Notif y
T8		Priorité : 0 Sources : M2/T8.cpp
	P1	Service : API/IO/Pipe/Put
	P2	Service : API/IO/Pipe/Put
	P3	Service : API/IO/Pipe/Get

Figure 20 : Paramètres de l'application VDSL pour le module M2

Les deux figures précédentes précisent les paramètres des différents objets de la spécification concernant la génération de systèmes d'exploitation :

- Au niveau des processeurs nous retrouvons leur type et la définition des ressources locales. Par exemple dans la Figure 19, le module M1 est un processeur de type ARM7, dont les ressources en mémoire sont de 65536 octets, et dont les ressources en interruption (virtuelles) sont de 256 numéros.
- Au niveau des tâches nous retrouvons leur priorité et un lien vers leur code source. Par exemple dans la Figure 19, la tâche T1 a la priorité 0 et a pour source le fichier M1/T1.cpp. Dans cette figure, toutes les tâches ont la priorité 0, ce qui revient à dire qu'il n'y a pas de priorité.
- Au niveau des ports des tâches nous retrouvons les services de haut niveau utilisés par les tâches. Par exemple dans la Figure 19, le port P1 de la tâche T1 requiert le service API/IO/Pipe/Get.
- Au niveau des ports des processeurs nous retrouvons les média utilisés et les paramètres d'allocations associés (adresses, types des données, etc...). Par exemple dans la Figure 19, le port P1 du module M1 utilise le média LockedRegister aux adresses 0xF00000 et 0xF00004, avec l'interruption numéro 1.
- Au niveau des canaux, nous retrouvons les paramètres d'allocations si nécessaires. Dans la Figure 19 le canal N2 utilise l'interruption numéro 1 pour la synchronisation du tube entre T2 et T3.

c Les protocoles utilisés

Au niveau des tâches

Les tâches communiquent et se synchronisent en utilisant divers protocoles que nous allons décrire.

Le premier est le protocole libre (Over) : il permet d'écrire ou lire directement dans une variable globale au système. Il est surtout utile pour accéder directement aux registres d'un périphérique quelconque. Dans le cas de l'application, il est utilisé pour accéder aux registres de configuration du flot de données. Dans la Figure 18, ce protocole est utilisé pour le port P2 de la tâche T5.

Le deuxième est le tube (Pipe) : il permet à un nombre quelconque de tâches de s'échanger des données de manière sûre en FIFO. Lorsqu'une tâche essaie d'écrire dans un tube, elle est bloquée jusqu'à ce que la ressource concernée puisse accepter cette écriture. Il en est de même en lecture : une tâche est bloquée tant qu'il n'y a aucune donnée dans la ressource. Dans la Figure 18, les ports de tâche des connexions N4, N5 et N6 du module M1 sont du type Pipe.

Le troisième est la mémoire (Memory) : il s'agit de la définition d'une zone mémoire accessible par une tâche. Pour pouvoir y accéder, la tâche doit «attacher» cette zone de mémoire, ce qui lui permet ensuite d'y accéder en tableau ou pointeur, comme n'importe quelle autre variable. Cette mémoire peut être partagée entre plusieurs tâches, mais aucune garantie n'est apportée quant à la cohérence du contenu d'une mémoire partagée : si certains accès sont critiques, il convient d'utiliser en plus un autre protocole tel que les sémaphores pour en garantir l'accès exclusif. Dans la Figure 18, les ports de tâche des connexions N6 et N5 du module M2 sont du type Memory.

Le quatrième est le signal (Signal) : une tâche peut attendre ou émettre un signal. Lorsqu'une tâche est en attente d'un signal, elle est bloquée jusqu'à ce que le signal soit émis par une autre tâche. L'émission d'un signal n'est

pas bloquante et il est perdu si aucune tâche ne l'attend. Dans la Figure 18 les ports de tâches des connexions N1, N2 et N3 du module M1 sont du type Signal.
Le cinquième est le sémaphore (Semaphore) : il s'agit du protocole de synchronisation décrit par Dijkstra [DIJ 67]. Dans la Figure 18, les ports de tâche de la connexion N6 sont aussi du type Semaphore en plus d'être du type Memory.
Enfin le dernier est le temporisateur (Timer) : il s'agit d'un protocole qui permet à une tâche d'être bloquée pendant un temps programmé à l'avance. Le port P4 de la tâche T5 est du type Timer.

Au niveau des processeurs

Les processeurs communiquent entre eux et avec le flot de données par l'intermédiaire de plusieurs protocoles. Chacun est réalisé par un contrôleur de communication qui dispose de sa propre interface avec le logiciel. Ces sont ces interfaces avec le logiciel qui nous intéressent dans la génération des systèmes d'exploitation, puisque ce sont elles qui permettent de choisir les pilotes de périphériques.
La première interface est celle de type registre (Register) : elle permet d'accéder simplement à un registre par le biais de son adresse sur le bus du processeur. C'est l'interface qui est utilisée pour accéder aux registres de configuration du flot de données. Dans la Figure 18, le port P5 du module M2 est du type Register.
La deuxième est celle de type registre à verrou (LockedRegister) : cette interface est composée d'un registre de données dans lequel le logiciel peut lire ou écrire des données, un registre d'état qui indique s'il est possible d'accéder en lecture ou en écriture au registre de données, et d'une interruption qui indique un changement d'état. Cette interface convient bien à des communications matérielles de type FIFO par exemple. Dans la Figure 18, les ports P2, P3 et P4 des modules M1 et M2 sont du type LockedRegister.
La troisième est celle de type tampon (Buffer) : cette interface est composée d'un registre de données et d'une interruption qui indique quand le tampon doit être rempli. Lorsque cette interruption intervient, le logiciel doit envoyer un nombre fixe de données par l'intermédiaire du registre. Dans la Figure 18, le port P1 du module M2 est du type Buffer.

La dernière est celle de type interruption (Interrupt) : il s'agit tout simplement d'une interruption qui est utilisée pour activer une action logicielle quelconque. Dans la Figure 18, le port P1 du module M1 est du type Interrupt.

2.5.3 La bibliothèque de système d'exploitation pour l'application

a Les choix effectués pour la construction de la bibliothèque

La bibliothèque offre de nombreuses libertés quant à la définition des éléments et des services, de nombreuses orientations possibles pour définir la bibliothèque.
Remarque : les orientations pour la définition de la bibliothèque n'ont aucune influence sur l'outil. En effet c'est au niveau des dépendances et du code des macros qu'elles prennent effet.

Les langages utilisés pour le code des éléments : le langage C et les langages d'assemblage

Il est naturel de nos jours d'utiliser des langages de programmation de haut niveau, et notamment les langages orientés objet, pour décrire le logiciel. Cependant, les langages de haut niveau ne permettent pas de décrire certaines parties des systèmes d'exploitation telles que les changements de contexte, ou le traitement des

interruptions. Une partie du code d'un système d'exploitation doit donc impérativement être décrite en langage d'assemblage.
Il reste cependant une grande partie du système d'exploitation qui peut être décrite dans un langage de haut niveau. Pour cette partie, nous avons utilisé le langage C qui est couramment utilisé dans ce domaine.
La spécification d'entrée du flot est décrite en SystemC qui est un langage basé sur le langage C++. Nous avons pourtant écarté ce langage car il pose des problèmes à l'édition de liens comme nous le verrons dans la section « Compilation du système d'exploitation généré » page 47. De plus, même si les compilateurs ont fait beaucoup de progrès, un programme écrit en C++ est souvent moins efficace qu'un programme équivalent écrit en C. Enfin l'approche orientée objet simplifie le travail d'un programmeur d'application de haut niveau en cachant l'implémentation des données et du code dans des structures abstraites (les objets). Mais cet avantage peut devenir un inconvénient lorsqu'il s'agit de décrire du logiciel de très bas niveau, comme les pilotes de périphériques où l'implémentation doit être parfaitement connue pour pouvoir interfacer correctement avec les périphériques.
C'est donc le langage C et le langage d'assemblage qui ont été choisis pour décrire le code des éléments de la bibliothèque. Il reste à déterminer sous quelles proportions nous allons utiliser ces langages.
Pour cette première bibliothèque, nous avons utilisé le moins possible le langage d'assemblage : son utilisation a été réduite à certaines parties de l'initialisation du système, au changement de contexte et à l'entrée dans les traitement des interruptions. Le langage C étant indépendant du processeur (contrairement aux langages d'assemblages), ce choix limite le nombre d'éléments à modifier, ce qui permet de faciliter l'ajout d'un nouveau processeur. Cependant, cela ne permet pas de réaliser un système d'exploitation optimal en taille et en performance. En effet, malgré les progrès effectués en compilation, certaines parties restent plus efficaces si elles sont écrites directement en langage d'assemblage.
Il est prévu, au fur et à mesure de l'extension de la bibliothèque, d'ajouter aux éléments actuellement écrits en C des implémentations en langage d'assemblage. Cela permettra d'avoir le choix entre un code C général ou un code en langage d'assemblage spécifique à certains processeurs.
Remarque : il est possible d'écrire des parties de code en langage d'assemblage à l'intérieur d'un programme en langage C (grâce à la directive asm). Cette possibilité n'a cependant pas été utilisée car, pour permettre la compilation du programme complet, le code assembleur est souvent modifié par le compilateur. Or certaines parties critiques, telles que le changement de contexte, ne doivent pas être changées pour pouvoir fonctionner correctement.

Découpage en éléments

Il est préférable de découper toute fonctionnalité générale en au moins deux éléments (un de haut niveau, et un de bas niveau) ce qui permet de décliner cette fonctionnalité sur divers média à moindres frais.
Nous avons poussé cette recommandation au maximum dans la bibliothèque de telle sorte qu'aucune fonctionnalité n'est complètement définie par un élément. Les avantages de ce choix sont que la taille de la bibliothèque est très réduite comparée à ses possibilités et que l'extension de la bibliothèque est peu coûteuse : il suffit souvent d'ajouter un service d'API de haut niveau pour qu'il soit directement décliné sur divers média. Et inversement, le simple ajout d'un pilote est suffisant pour permettre à de nombreux services d'API d'accéder à de nouveaux périphériques. L'inconvénient est que les relations entre les divers éléments de la bibliothèque

deviennent très complexes comme en témoigne la Figure 21; de plus, il devient nécessaire d'ajouter des éléments ne servant que de lien entre plusieurs fonctionnalités comme c'est le cas pour interrupt.

L'interface entre les tâches de l'application et le système d'exploitation

Nous parlons ici des fonctions ou appels système fournis par le système d'exploitation, et non pas des méthodes qui encapsulent ces dernières pour adapter le code des tâches sans avoir à les modifier.

Appels système ou appels de procédure

Le système d'exploitation peut offrir l'accès à ses services par l'intermédiaire de simples appels de procédure ou par des appels système. Ces derniers provoquent des interruptions logicielles qui font basculer le processeur en mode système, ce qui permet notamment de masquer les interruptions lors des traitements critiques.
Dans la bibliothèque, toute l'interface avec le système d'exploitation est réalisée par des appels système. Cela permet une séparation claire entre l'application et le système d'exploitation ; par contre, les appels système ajoutent des délais importants dans les réponses de l'application.

Masquage des interruptions

Dans un système d'exploitation, lors d'une section critique, il est conseillé de masquer le moins d'interruptions possibles et le moins longtemps possible.
En pratique, il n'est pas aisé de déterminer pour chaque cas quelles sont les interruptions à masquer. Pour simplifier la définition de la bibliothèque, les interruptions sont systématiquement masquées lors d'un appel système.

Généralité de l'interface entre l'application et le système d'exploitation

Comme nous l'avons vu dans la section « La spécification que nous avons utilisée dans le flot » page 33, les systèmes d'exploitation des deux processeurs doivent générer l'accès à de très nombreuses ressources. De plus, les points d'accès au système d'exploitation (matérialisés par les ports des tâches) sont eux aussi très nombreux. L'ensemble des appels système constitue l'interface du système d'exploitation. Elle permet d'accéder sans ambiguïté aux ressources.
La méthode la plus simple pour le faire est de générer un appel système pour chaque service de chaque port de tâche. Cette solution est aisément intégrable dans la bibliothèque, mais elle apporte beaucoup de redondance dans le code et accroît d'autant sa taille. La solution opposée consiste à avoir un appel système par service de haut niveau, et de passer en paramètre les identificateurs permettant de désigner la ressource et le port concerné. Avec cette dernière solution, le code des appels système devient compliqué, surtout lorsqu'un même service de haut niveau utilise des ressources de types différents (comme c'est le cas pour le tube dans l'application VDSL). Cette complexité de l'appel système peut fortement réduire ses performances.
Un compromis entre les deux précédentes approches a été préféré lors de la définition de la bibliothèque : un appel système est généré par service de haut niveau couplé à un type de ressource. Ainsi le code d'un appel système reste simple et rapide, sans qu'il y ait de redondance puisque le code est différent suivant le type de ressource.

b Contenu de la bibliothèque

La bibliothèque contient des éléments pour générer un système d'exploitation complet pouvant faire tourner l'application VDSL. Les processeurs supportés sont le processeur ARM7, le processeur 68000 et le pseudoprocesseur UNIX.

Relations entre les éléments et les services

Les relations entre les éléments et les services de la bibliothèque sont données dans la Figure 21. Cette figure est complexe. Nous allons décrire rapidement les éléments qui la composent au cours des sections suivantes.

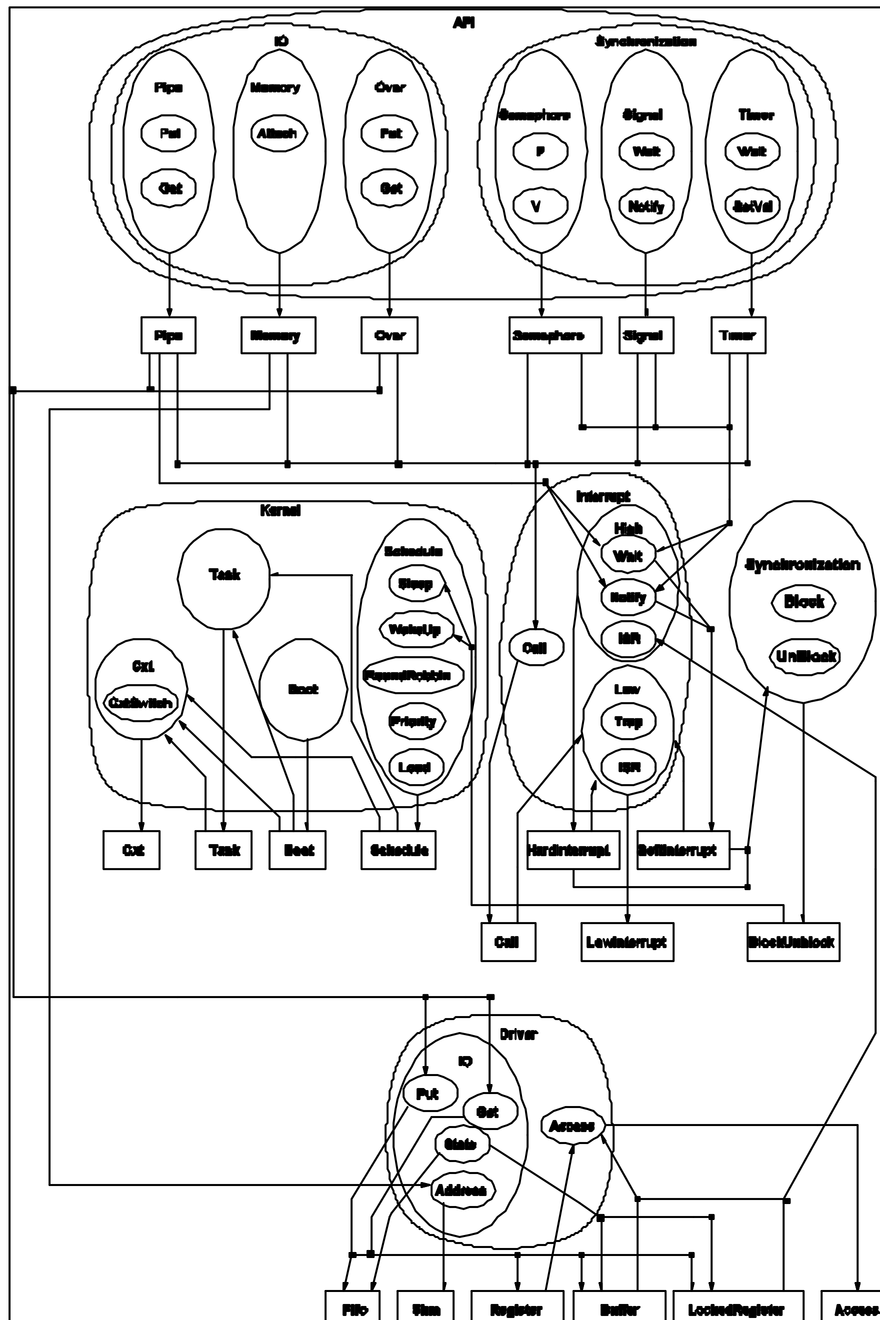

Figure 21 : La bibliothèque de système d'exploitation pour l'application VDSL

Les éléments fournissant les services d'API

Ces éléments permettent la génération des appels système utilisables par l'application. Pour fonctionner ils doivent être assemblés avec des éléments fournissant les services de Driver (qui permettent d'accéder aux ressources de bas niveau) et des éléments fournissant les services de Interrupt (qui permettent la gestion des événements).

L'élément «Pipe»

Il fournit les services d'API réalisant une communication par tube. Cet élément permet la génération de deux types d'appels système : Put et Get pour écrire et lire des données atomiques dans le tube.

L'éléme nt «Memory»

Il fournit le service d'API permettant «d'attacher» une zone mémoire à une tâche.

L'élément «Over»

Il fournit les services d'API permettant d'accéder directement à une ressource telle qu'un registre. Cet élément permet la génération de deux types d'appels système : Put et Get pour écrire ou lire directement dans la ressource.

L'élément «Semaphore»

Il fournit les services d'API réalisant des synchronisations par sémaphores. Cet élément permet la génération de deux types d'appels système : P et V pour effectuer les opérations de même nom sur un sémaphore.

L'élément «Signal»

Il fournit les services d'API réalisant des synchronisations par signaux. Cet élément permet la génération de deux types d'appels système : Wait pour attendre un signal et Notify pour en générer un.

Les éléments fournissant les services de «Kernel»

Ces éléments permettent la génération du noyau du système d'exploitation, c'est-à-dire l'ensemble des fonctionnalités permettant de gérer les tâches. Ces éléments ont de nombreuses dépendances entre eux, mais ils ne dépendent d'aucune autre famille de services.

L'élément «Cxt»

Il fournit la fonction de changement de contexte qui permet de passer de l'exécution d'une tâche à l'exécution d'une autre tâche. Son code dépend entièrement du processeur cible, c'est pourquoi il dispose d'une implémentation différente par processeur.

L'élément «Task»

Il fournit le squelette de la structure de données qui décrit une tâche. La structure de données finale est obtenue en complétant ce squelette avec les informations propres au processeur (élément Cxt) et à l'algorithme d'ordonnancement (élément Schedule).

L’élément «Boot»

Il fournit le code d’initialisation du processeur et du système d’exploitation. Une partie de cet élément dépend du processeur. Il dispose donc d’un arbre d’implémentations partant d’un code général pour l’initialisation du système d’exploitation, et allant à un code spécifique au processeur pour son initialisation.

L’élément «Schedule»

Il fournit l’algorithme d’ordonnancement des tâches. C’est un élément très flexible qui permet la construction de divers algorithmes d’ordonnancement. Il peut fournir l’algorithme du tourniquet avec la possibilité de pondérer différemment l’utilisation du processeur pour chaque tâche. Il peut aussi fournir un algorithme à base de priorités (statiques ou dynamiques). Enfin ces algorithmes peuvent être combinés.

Les éléments fournissant les services de «Interrupt»

Ces éléments permettent de générer le gestionnaire d’interruptions pour le système d’exploitation. Pour s’abstraire des spécificités du processeur vis à vis des interruptions, un système d’interruptions virtuelles a été mis en place : il fournit un nombre arbitraire d’interruptions virtuelles qui encapsulent les interruptions réelles (matérielles ou logicielles). Ces éléments ont besoin des primitives de synchronisation de base du système fournies par l’élément BlockUnBlock.

L’élément «Call»

Il encapsule les interruptions logicielles du type «appel système».

L’élément «LowInterrupt»

Il encapsule les interruptions matérielles : à chaque fois que l’une d’entre elles apparaît, la routine d’interruption générée par cet élément va lire le numéro de l’interruption virtuelle souhaitée sur le bus donnée du processeur.

Les éléments «HardInterrupt» et «SoftInterrupt»

Ils fournissent l’interface de haut niveau pour ces interruptions virtuelles : Wait et Notify permettent d’en attendre ou d’en générer une. HardInterrupt permet aussi la déclaration de routines d’interruption virtuelles.

L’élément fournissant les services de «Synchronization»

Cet élément fournit deux primitives de base et des files d’attente permettant de construire toutes les synchronisations dans le système. La primitive Block met la tâche courante dans l’état endormi et la place dans une file d’attente. La primitive UnBlock réveille une des tâches endormies dans une file d’attente. Pour fonctionner, cet élément a besoin des services de Kernel permettant d’endormir et de réveiller des tâches.

Les éléments fournissant les services de «Driver» (pilotes)

Ces éléments permettent de générer des pilotes de périphériques. Ce sont eux qui font l’interface entre le logiciel et le matériel. Ils peuvent utiliser des routines d’interruption, et donc requérir les éléments fournissant les services de Interrupt.

L’élément «Fifo»

Il réalise un tampon logiciel fonctionnant en FIFO. Il ne gère aucun périphérique ; cependant sa fonctionnalité et son niveau étant identiques à ceux des pilotes de périphériques il a été placé dans cette catégorie.

L’élément «Shm»

Il réalise une zone de mémoire locale partagée entre plusieurs tâches. C’est un élément très simple, qui fournit simplement la plage d’adresses qui représente la mémoire partagée.

L’élément «Register»

Il permet l’accès à des registres. Il contient des tables d’adresses de registres, et des macros permettant d’y accéder. Cet élément est indépendant du processeur quel que soit son mode d’accès aux périphériques. En effet il utilise le service Access qui est chargé de générer le code permettant ce type d’accès.

Les éléments «LockedRegister» et «WaitRegister»

Ils permettent l’accès aux périphériques de type LokedRegister et Buffer (voir la section 5.1.2.3.). Ils contiennent des tables d’adresses de registres de données et état, ainsi que des routines d’interruption appelées au moment d’un changement d’état. Comme l’élément Register, ces éléments utilisent le service Access.

L’élément «Access»

Il fournit les macros permettant d’accéder à un périphérique d’un processeur. Si le processeur place ses entrées/sorties en mémoire, cet accès se réduit à l’utilisation d’un pointeur. Sinon, il faut utiliser des instructions particulières du processeur.

2.5.4 Résultats

a Evaluation du flot de génération de système d’exploitation

La spécification d’entrée

La démonstration a mis en avant l’intérêt de ne pas décrire la spécification dans le langage Colif : en effet c’est un langage qui a été prévu pour être aisément analysé et modifié par les outils du flot. Il est fastidieux de décrire une spécification dans ce langage. Au contraire SystemC, avec sa syntaxe provenant du C++, est accessible pour un homme, surtout si ce dernier a une bonne expérience en programmation C ou C++.
Les premières spécifications ont été écrites directement en Colif, puis, pour l’application VDSL, un traducteur SystemC vers Colif a été développé. Le gain en productivité est important : une spécification telle que celle de l’application VDSL fait plus de 1 Mo en Colif, tandis que l’équivalent en SystemC ne fait plus que 23 Ko.
Un inconvénient commun entre SystemC et Colif est la détection des erreurs d’architecture (mauvais placements ou interconnexions de modules, ports ou nets). Pour détecter ces erreurs, un outil graphique de visualisation d’architecture Colif a donc été développé dans le cadre de la démonstration.
Nous avons indiqué dans à la page 33 les paramètres concernant la génération de systèmes d’exploitation. Il y en a beaucoup d’autres pour la génération des interfaces matérielles et pour la génération des modèles de

simulation. Bien que le flot permette de s'affranchir de nombreux détails d'implémentation, il reste beaucoup trop de paramètres à définir manuellement. La démonstration a mis en relief ce problème puisque une grande partie des erreurs de spécification étaient des erreurs au niveau des paramètres. Ces erreurs ne sont souvent détectées qu'au cours de la génération.
Ce problème au niveau des paramètres montre que l'étape de synthèse de la communication est nécessaire pour que le flot soit complet.

La bibliothèque de système d'exploitation

La bibliothèque a été complétée pour satisfaire l'application VDSL. C'est pourquoi elle n'a manqué d'aucun élément pour permettre la génération des systèmes d'exploitation de la démonstration.
Bien que la bibliothèque ait été spécialement prévue pour une application particulière, la plupart des éléments de la bibliothèque sont suffisamment généraux pour qu'elle puisse être utilisée pour d'autres applications.
De plus, les dépendances indirectes et l'utilisation d'un langage de macros complet ont permis de factoriser fortement la bibliothèque. C'est ainsi qu'avec 21 éléments, la bibliothèque permet de générer 37 fonctionnalités différentes (29 appels système et 8 algorithmes d'ordonnancement différents) indépendamment du processeur cible. En outre, le code de chaque élément dépasse rarement la centaine de lignes. Enfin ces diverses fonctionnalités peuvent toutes cohabiter dans un même système d'exploitation (il est même possible d'avoir plusieurs ordonnanceurs de tâches).
Pour le moment la bibliothèque ne contient que peu de pilotes de périphériques ainsi que de peu d'API. Cela signifie que le rapport entre nombre d'éléments et nombre de fonctionnalités possibles sera bien meilleur à mesure que la bibliothèque sera complétée.

La Figure 22 et la Figure 17 représentent les éléments sélectionnés et assemblés à partir de la bibliothèque pour les deux processeurs. Dans ces figures, les rectangles tangents représentent un assemblage d'éléments, tandis que les formes arrondies représentent des liens par appels des fonctions C. Chaque fonction correspond au service indiqué dans la forme. Le haut de la figure correspond à l'interface avec l'application tandis que le bas correspond au noyau. Ces figures montrent par exemple qu'un élément d'API tel que Pipe est dupliqué pour être assemblé avec chaque pilote de périphérique utilisé pour ce type de protocole.

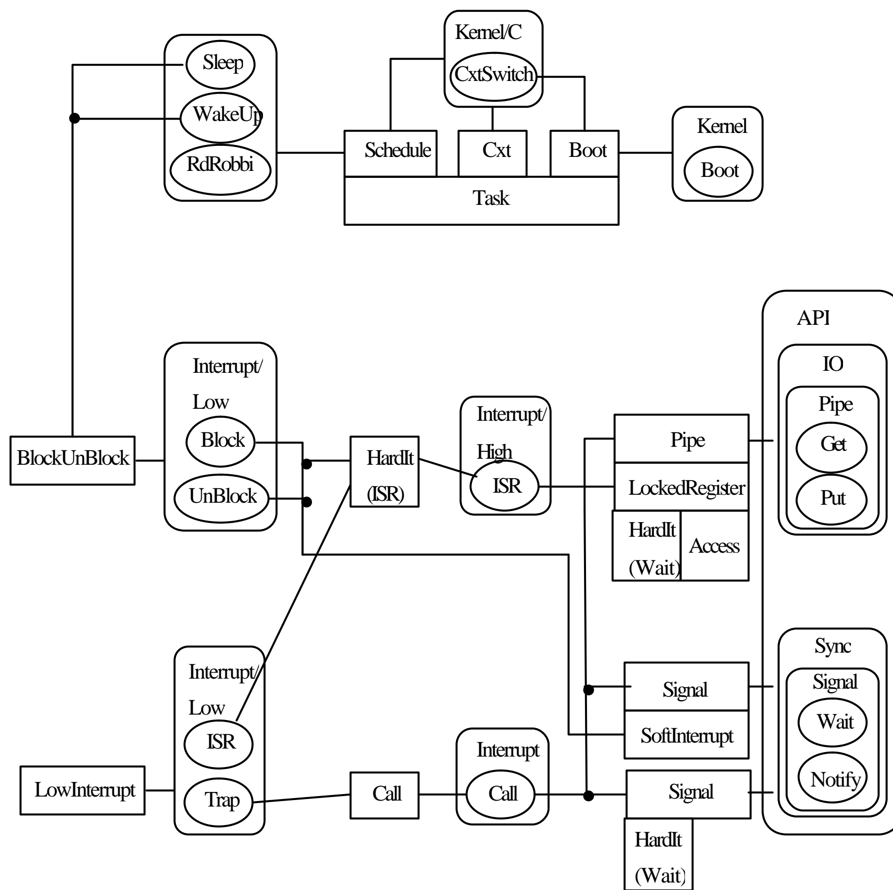

Figure 22 : Le système d'exploitation généré pour le module M1

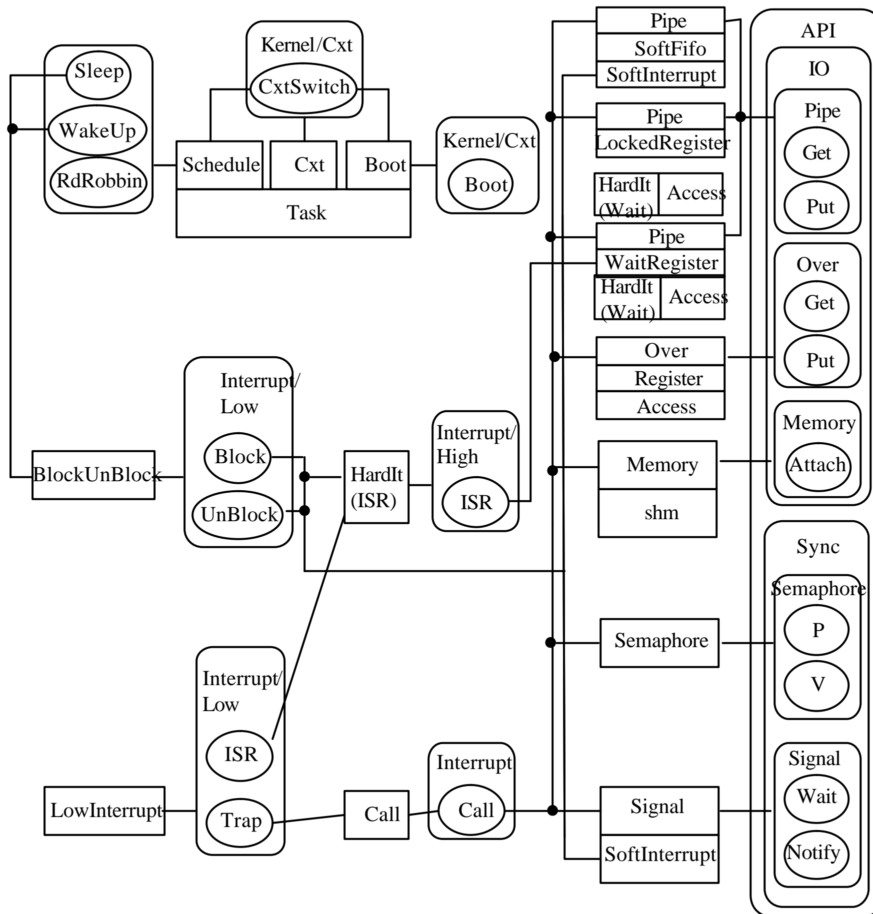

Figure 23 : Le système d'exploitation généré pour le module M2

L'extension de la bibliothèque peut se faire suivant deux axes : l'ajout de fonctionnalités ou l'extension de sa compatibilité avec d'autres architectures.

Pour le premier axe, l'essentiel du travail consiste maintenant à ajouter des éléments d'API. Une fois que l'algorithme d'un tel élément est connu, son ajout dans la bibliothèque est très rapide : il suffit de déterminer quels sont les services de bas niveau dont il a besoin en s'inspirant des éléments d'API existants, puis d'encapsuler son code dans une macro qui permettra son assemblage et son adaptation lors de la génération. Les macros qui encapsulent le code des éléments d'API suivent toutes le même modèle, l'intégration du code d'un nouvel élément d'API est donc très simple. Une fois que l'utilisateur a une bonne expérience de la bibliothèque et du langage de macro, l'intégration d'un élément d'API prend de quelques minutes à quelques heures suivant la complexité de l'élément.

Pour le second axe, si le matériel à ajouter est un périphérique, il suffit d'ajouter l'élément de Driver correspondant. Le travail est alors identique à celui de l'ajout d'un élément d'API. Si le matériel est un processeur, il faut repérer tous les éléments dépendants du processeur, et leur ajouter les implémentations qui les rendront compatibles avec ce dernier. Ce travail peut être fastidieux si la bibliothèque est importante : et un outil permettant d'effectuer automatiquement ce type de recherche serait le bienvenu.

Pour le moment, seul le test après génération permet de valider le système d'exploitation, aussi bien du point de vue des fonctionnalités que du point de vue temporel. Il serait intéressant de pouvoir être sûr avant le test que le système généré soit correct et qu'il puisse respecter les contraintes temporelles.
Pour avoir à l'avance une estimation des propriétés temporelles du système d'exploitation, un flot d'estimation de performance basé sur les résultats obtenus avec des systèmes d'exploitation déjà générés va être étudié au cours des prochains travaux. Le but est d'ajouter dans la bibliothèque des annotations sur les propriétés temporelles de chaque élément suivant les paramètres de génération. Les possibilités d'extension du langage de macro permettront d'inclure les algorithmes d'estimation directement dans le code des macros sans avoir à modifier l'outil de génération. Ainsi, en même temps que le code du système d'exploitation sera généré, des estimations temporelles sur ce dernier seront produites.
S'assurer de la validité fonctionnelle du système d'exploitation généré est un problème complexe à résoudre. L'idéal serait de le prouver formellement. Dans un premier temps, il faudra définir une classification pour les éléments et les services. Il faudra ensuite étudier la stabilité des propriétés d'un élément par composition (c'est-à-dire assemblage avec un autre élément) et adaptation.

L'outil de génération

Une fois la spécification et la bibliothèque définies, l'outil génère automatiquement les systèmes d'exploitation spécifiques à l'architecture et à l'application logicielle, ainsi que les fichiers permettant leur compilation.
Pour la démonstration, il manquait cependant la génération des fichiers d'adaptation du code des tâches de l'application. Ces fichiers ont donc été écrits à la main. Cela a permis de constater que leur génération automatique est identique à la génération de n'importe quelle autre partie du système d'exploitation.
La génération des deux systèmes d'exploitation pour l'application VDSL ne prend que quelques dizaines de secondes. Au cours de cette génération, de nombreux messages indiquent l'évolution des opérations, ainsi que les problèmes rencontrés. Le compte rendu est suffisamment précis : dans le cadre de la démonstration, toutes les erreurs de la spécification (aussi bien au niveau de l'architecture, qu'au niveau des paramètres) ont été détectées au cours de la génération :
- Les erreurs au niveau de l'architecture sont détectées au moment de son analyse.
- Les erreurs au niveau des paramètres sont détectées à la génération du code par les algorithmes de traitement des paramètres inclus dans les macros.

Compilation du système d'exploitation généré

La compilation des systèmes d'exploitation générés conjointement avec les tâches de l'application a permis de découvrir des difficultés qui n'avaient pas été prévues initialement.
La première de cette difficulté est que le compilateur utilisé pour le processeur ARM7 n'était pas totalement compatible avec la norme ANSI. Ce problème a été surmonté en modifiant quelques macros pour passer outre. Cela met en avant le problème du langage : il est important que le langage choisi soit vraiment portable sur tout processeur pour être utilisé. Dans le cas du langage C, la norme ANSI permet d'assurer cette portabilité.
La deuxième difficulté est intervenue à l'édition de liens entre le système d'exploitation et les tâches. Nous rappelons que les tâches étaient décrites en C++. C'est un langage orienté objet, qui permet notamment de

surcharger les méthodes (fonctions). A la compilation cette surcharge est traitée en renommant les méthodes pour que chaque méthode ait un nom unique. La difficulté est qu'il n'y a pas de convention pour ce renommage : suivant les compilateurs ces méthodes sont renommées différemment. Dès lors, il devient difficile de faire référence à une tâche décrite en C++ à partir d'un autre langage ou d'un autre compilateur. Pour résoudre ce problème, un élément spécial de la bibliothèque ayant pour but de calculer le véritable nom de chaque tâche (c'est-à-dire après renommage) a été défini.
Remarque : la solution qui consisterait à décrire toutes les tâches et tous les éléments du système d'exploitation dans le même langage est inapplicable (en outre, elle manque totalement de souplesse) : par exemple, certaines parties comme le changement de contexte doivent être décrites en langage d'assemblage.

Validation du résultat

Nous avons vu que le système d'exploitation généré ne peut être validé que par test. Cependant, comme il est spécifique à l'application et à l'architecture, sa validation complète ne peut être effectuée que conjointement avec toute l'application.
Le système de cosimulation est utilisé pour valider l'ensemble de l'application. Pour ce faire les systèmes générés sont compilés avec les tâches et exécutés sur des ISS (Instruction Set Simulator) des processeurs ARM7. Ces simulateurs sont intégrés dans le banc de cosimulation. Cette cosimulation peut prendre beaucoup de temps car il y a de nombreux éléments à simuler.
Une cosimulation plus rapide peut être obtenue en élevant le niveau d'abstraction des modèles simulés, et en exécutant nativement le code des tâches et du système d'exploitation (c'est-à-dire directement sur la station).
Pour exécuter nativement le code des tâches et du système d'exploitation tout en conservant l'environnement logiciel de la station (système UNIX), le pseudoprocesseur UNIX a été défini et intégré dans la bibliothèque de génération de systèmes d'exploitation. Il s'agit en fait d'un processus classique, qui simule les interruptions du processeur par des signaux, les ports du processeur par des tubes, et les changements de contexte entre les tâches grâce aux instructions sigsetjmp et siglongjmp. Ces fonctions permettent respectivement de sauvegarder et de restaurer l'état du processus. Ainsi, tout le code est exécuté nativement, ce qui permet d'obtenir de bonnes performances de simulation tout en utilisant les algorithmes du système d'exploitation générés.

b Résultats concernant les systèmes d'exploitation générés

Le code généré

Deux systèmes d'exploitation ont été générés pour les deux processeurs.
Pour le premier processeur (le module M1), 968 lignes de code C ont été générées, ainsi que 281 lignes de code en langage d'assemblage. Après compilation, le système d'exploitation utilise 3829 octets pour le code et 500 octets pour les données.
Pour le deuxième processeur (le module M2), 1872 lignes de code C ont été générées, ainsi que 281 lignes de code en langage d'assemblage. Après compilation, le système d'exploitation utilise 6684 octets pour le code et 1020 octets pour les données.

Comparé à des systèmes d'exploitation classiques, ou même à des systèmes d'exploitation configurables, le résultat est très bon. En effet, en moyenne la taille minimale d'un système d'exploitation embarquée est de 4k octets ; eCos [ECO] permet d'obtenir de meilleurs résultats puisque sa taille minimale est d'un peu moins de 1k

octets. Cependant, nous parlons ici de taille minimale, les systèmes d'exploitation générés pour l'application fournissent déjà de nombreuses fonctionnalités.

Le code généré peut être de qualité équivalente à celle d'un code écrit à la main. La Figure 24 en donne un échantillon (il s'agit d'une partie du code de l'ordonnanceur). Comme le montre la figure, il est possible d'avoir un code aéré et disposant de commentaires. En effet, la génération à partir de macros permet de donner facilement n'importe quel style dans l'écriture du code et aussi de générer les commentaires.

```
/*####################################################################*/
/*##      The scheduler : contain all the scheduling functions                          ##*/
/*####################################################################*/
 /* Services Headers */
#include "Kernel/Cxt/unixcxt.h"
#include "Kernel/Task/task.h"
#include "Kernel/IT/High/ithigh.h"
#include "schedule.h"

/*####################################################################*/
/*##                        The tasks states                                             ##*/
/*####################################################################*/
#define READY     1
#define SLEEP  2
#define RUNNING 3
/*####################################################################*/
/*##                        Global variables                                             ##*/
/*####################################################################*/
int cur_tid,new_tid;/* current and new task id */
/*####################################################################*/
/*##    Scheduling functions with :                                                      ##*/
/*##                                                                                     ##*/
/*##    - round-robbin                                                                   ##*/
/*####################################################################*/
/* Main scheduling function */
void __sched_schedule(void)
{
    /* Update running task id */
    int old_tid=cur_tid;
    cur_tid=new_tid;
    /* Context switching call */
    __cxt_switch(__task_tasks[old_tid].cxt,__task_tasks[cur_tid].cxt);
}

/* Round-robbin function */
void __sched_roundrobbin(void)
{
    cur_tid=__task_tasks[cur_tid].sched.next;
}
```

Figure 24 : Exemple de code généré

Un avantage de l'utilisation du langage de macro complet par rapport à celle du préprocesseur du langage C, est que le code accessible par l'utilisateur n'est pas rendu confus par les directives de précompilation nécessaires à l'adaptation de code. Dans le cas de l'utilisation du préprocesseur, plus la flexibilité est grande, plus il y aura de directives, et plus le code sera confus.

Qualité du système d'exploitation généré

Pour qu'un système d'exploitation puisse supporter des applications temps-réel, une propriété importante est qu'il soit déterministe. Pour favoriser ce déterminisme, le code des systèmes d'exploitation a été écrit de telle manière qu'il n'y ait pas de boucle dépendante des données. Dès lors, les temps d'exécution de chaque partie des systèmes générés sont bornés de bornes connues.
Les performances d'un système d'exploitation sont en général estimées à partir du temps de changement de contexte et du temps de latence pour une interruption. D'autres temps peuvent être intéressants comme la latence d'un appel système, ou le temps de retour du système d'exploitation vers une tâche. Nous allons donner ces valeurs pour le processeur ARM7 cadencé à 25MHz.
Pour les systèmes d'exploitation générés, le changement de contexte prend 36 cycles, c'est-à-dire 1,44µs. La latence pour un interruption matérielle est de 59 cycles c'est à dire 2,36µs. Il faut ajouter à ce temps de latence celui mis par le processeur lui-même pour traiter l'interruption qui est de 4 à 28 cycles. La latence pour un appel système est de 50 cycles c'est-à-dire 2,00µs. Enfin le temps de retour du système d'exploitation à une tâche est de 26 cycles c'est-à-dire 1,04µs.
Ces résultats sont bons comparés à ceux des systèmes d'exploitation commerciaux pour des processeurs équivalents à l'ARM7 : en effet leur latence pour les interruptions (caractéristique la plus souvent donnée) ne descend jamais au-dessous de 2µs quel que soit le processeur, et tourne en moyenne autour de 5µs.

2.5.5 Conclusions sur le flot de ciblage logiciel

Le flot de ciblage a été appliqué avec succès sur une application VDSL. Cette application non triviale a en fait été traitée dans le flot global décrit au chapitre 2. Le ciblage est la phase ultime pour ce qui concerne le logiciel.
Pour appliquer le flot, une bibliothèque de système d'exploitation a été développée. Cette dernière fournit déjà une grande variété de communication. Elle ne possède que peu d'éléments vraiment spécifiques au processeur, ce qui veut dire qu'elle est aisément portable vers d'autres processeurs que l'ARM7 utilisés pour la démonstration. Un simulateur de processeur effectuant une exécution native a par exemple été intégré dans la bibliothèque pour faciliter son déboguage ainsi que celui de l'application.
Après utilisation, le flot s'avère rapide, et il simplifie le travail du concepteur en faisant abstraction des détails concernant le système d'exploitation. Cependant, une étape de synthèse et d'allocation préalable serait nécessaire pour s'abstraire complètement des détails de l'architecture tels que les adresses mémoire. Les systèmes d'exploitation générés sont quant à eux de très petite taille comparés aux systèmes d'exploitation embarqués commerciaux. Ils présentent aussi de bonnes performances. Un autre point positif est que les sources des systèmes d'exploitation générés sont aussi commentées et claires que si elles avaient été écrites directement à la main (c'est-à-dire sans génération, mais aussi sans les directives de configurations qui se trouvent dans le code des systèmes d'exploitation configurables). Nous pouvons remarquer à ce sujet que la qualité du système d'exploitation généré dépend fortement de la qualité de la bibliothèque.

2.5.6 Résultats de la génération de la partie matérielle

Cette section présente les résultats de la génération de la partie matérielle des interfaces générées pour l'application VDSL. Ils sont extraits de la thèse de Damien Lyonnard [LYO 03].

La Figure 25 montre l'architecture générée pour l'application. On observe les deux processeurs M1 et M2 communiquant avec M0 et M3 par des adaptateurs de canaux. A l'intérieur des modules M1 et M2, on peut voir l'architecture locale à chaque processeur avec leur ROM et leur RAM, leur décodeur d'adresse et un timer.

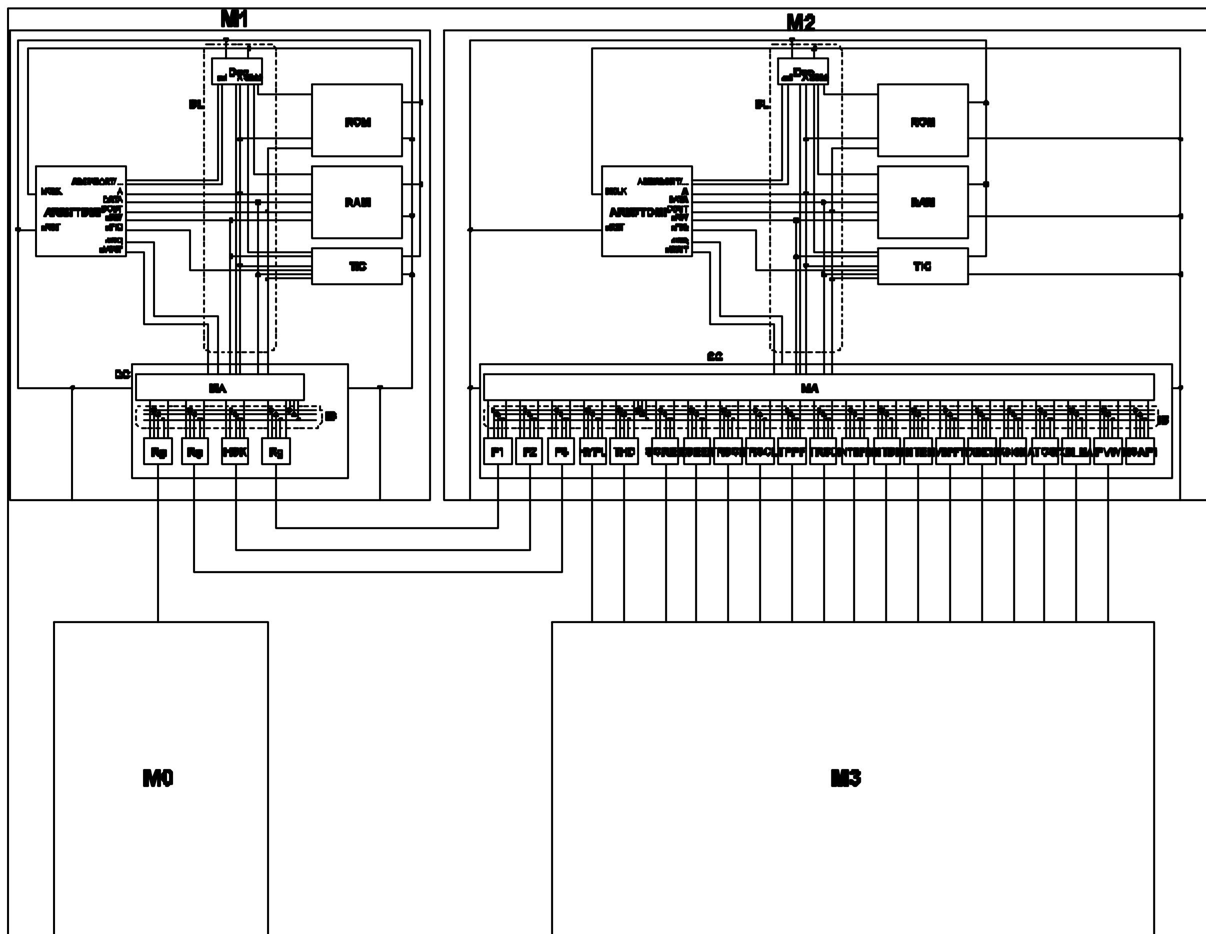

Figure 25 : La micro-architecture générée pour l'application VDSL

Le tableau suivant présente les résultats de synthèse du code VHDL des interfaces générées.

Module	Nombre de portes logiques équivalentes	Surface (µm2)	Période d'horloge minimale (ns)
M1	3284	8168	5,95
M2	3795	5100	6,16

Tableau 1: Résultat de synthèse de l'interface matérielle

2.6 Conclusion

Ce chapitre a mis en évidence les capacités du flot de conception ROSES. Le système de spécification permet de décrire des systèmes hétérogènes en utilisant des modules, des ports, des nets et le concept de modules virtuels. Ces spécifications sont traitées par deux outils permettant la génération des parties logicielles et matérielles des interfaces liant les processeurs au réseau de communication. Leur système de génération à base d'assemblage de composants génériques de bibliothèque permet la génération d'interfaces sur mesures. La réalisation de l'application VDSL a permis de montrer les capacités de ces outils. Le flot actuel est une bonne base pour le développement d'un flot d'exploration des solutions d'implémentations mixtes logicielles/matérielles des services de communication.

Chapitre 3 Architecture des interfaces et services de communication

---

Ce chapitre présente dans un premier temps, l'architecture des interfaces logicielles/matérielles. Cette architecture est la cible dans laquelle sont allouées les fonctionnalités composant les services de communication. La deuxième section, présente les services de communication offerts par le standard de communication par passage de message MPI. Ces services sont des exemples de services de communication complexes permettant la programmation parallèle. Une première partie fait une introduction à MPI. Les deux suivantes présentent la sélection d'un sous ensemble de MPI utilisé lors des expériences.

3.1 Architecture des interfaces logicielles/matérielles

L'interface logicielle matérielle comprend une partie logicielle exécutée sur le processeur et une partie matérielle connectant le processeur au réseau de communication. L'architecture présentée Figure 26 montre un ensemble constitué d'un processeur entouré de deux couches : la couche HAL (Hardware Abstraction Layer) et la couche AM (Adaptateur de module). Ces couches permettent l'abstraction des détails du processeur pour la partie logicielle et la partie matérielle. Cet ensemble constitue un processeur générique. Les services de communication sont implémentés dans la couche communication de la partie logicielle et dans les adaptateurs de canaux de la partie matérielle. Ces parties de l'interface contiendront donc les implémentations des fonctionnalités constituant le service de communication implémenté. Ceci ne signifie pas que les couche HAL et AM sont fixes. Si le partitionnement choisi du service de communication implique la réalisation de nouveaux protocoles de communication entre partie logicielle et partie matérielle, la couche HAL et la couche AM peuvent être modifiées. Par exemple, dans le cas d'une implémentation avec un processeur ARM7, un composant de la couche de communication de la partie logicielle peut avoir besoin d'être réactif aux sollicitations de la partie matérielle. On peut alors avoir besoin d'utiliser des interruptions de type « FIQ » (Fast Interrupt Request). Cela implique l'ajout d'une sous routine de traitement des FIQ dans la couche HAL et l'ajout d'un contrôleur de FIQ dans le AM.

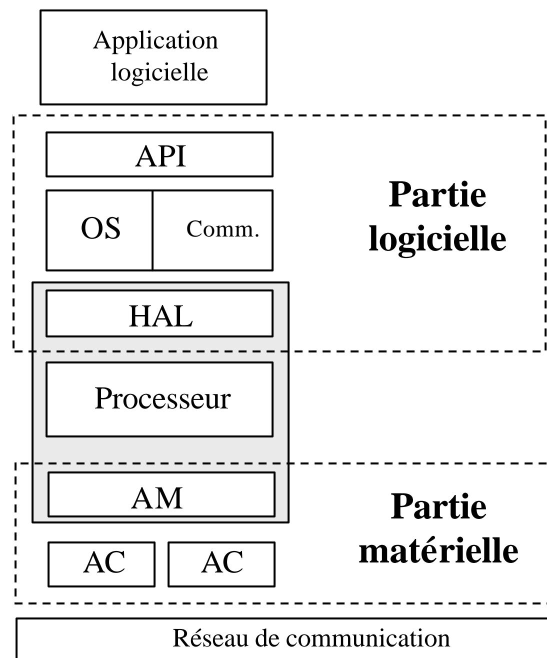

Figure 26 : Architecture des interfaces logicielles/matérielles

3.1.1 Architecture de la partie logicielle

La partie logicielle est découpée en plusieurs couches. Chaque couche fournie des API (« Application Program Instructions »). Comme le montre la Figure 27, une couche est l'implémentation de fonctionnalités sous forme de fonctions, de code macro ou de programmes exécutés en tant que processus indépendants. Les API de

la couche 2 sont utilisées par la couche logicielle s'exécutant au dessus de celle-ci. Les API sont généralement des prototypes de fonction, des noms de macro ou des appels système. Chaque appel d'une API entraîne l'exécution d'une partie du code contenue par la couche logicielle fournissant l'API. La programmation au dessus d'une couche revient à enrichir le jeu d'instructions fourni par le processeur.

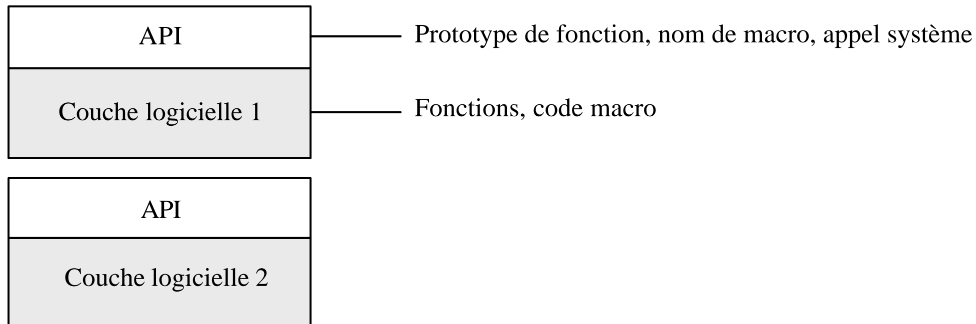

Figure 27 : Concept de couches logicielles et d'API.

La partie logicielle de l'interface est constituée de trois couches principales présentées à la Figure 26 :
- La couche COM : c'est un ensemble des fonctions implémentant la partie logicielle du comportement du service de communication.
- La couche OS : c'est un ensemble de fonctions pour l'ordonnancement des tâches, la gestion des interruptions et les entrées et sorties. Le contenu de cette couche est détaillé dans le chapitre suivant.
- La couche HAL : elle regroupe les fonctions permettant l'accès au matériel.

a La couche HAL

La couche HAL est utilisée pour offrir aux couches qui utilisent ses API, une abstraction des caractéristiques de l'architecture cible. Son utilité est donc de rendre portable tout code utilisant ses API.
Elle regroupe tout code dépendant du processeur et des périphériques associés. Il existe deux types de dépendances : une dépendance liée à l'utilisation d'un langage spécifique à l'architecture ou une dépendance liée à l'utilisation de fonctionnalités spécifiques à l'architecture. La dépendance au langage spécifique concerne tout code écrit en assembleur ou utilisant des instructions C spécifiques au processeur (par exemple __swi pour déclarer une interruption logicielle avec un processeur ARM7). On peut découper cette couche en un ensemble non exhaustif de catégories :
- Initialisation du processeur : initialisation des mode du processeur, initialisation des zones mémoires, mapping des plan d'allocation mémoire, branchement sur le boot.
- Manipulation des masques d'interruption : permet de contrôler les masques d'interruption d'un processeur (exemples d'API : disable_all_IRQ, enable_FIQ).
- Sous routines de traitement des interruptions : traitement des interruptions logicielles et des interruptions matérielles (exemples d'API : __SWI_Handler, __IRQ_Handler).
- Changement de contexte : la sauvegarde du contexte courant et le chargement du contexte de la tâche élue (exemple d'API : cxt_switch).

- Pilotes de périphériques : fonctions de manipulation de registres pour communiquer avec un périphérique, implémentation de protocoles spécifiques à un périphérique (exemples d'API : get_FIFO_status, copy_data).
- Code critique : certaines parties du code très utilisées peuvent être amenées à être développées en assembleur pour plus de performance. Ce code fera alors partie de la couche HAL.

b La couche OS

On peut découper le contenu de l'OS en quatre catégories de composants. Ces composants sont des fonctions ou des structures de données.

Les différentes catégories :

- Gestion de tâches : Cette catégorie inclut tout ce qui concerne l'ordonnancement, la gestion de l'état des tâches et leur synchronisation (exemples d'API : sched_schedule, sched_sleep, bloc_bloc).
- Fonctions de traitement d'interruptions : Cette catégorie comprend les fonctions permettant le traitement des interruptions logicielles (exemple d'API : trap_trap).
- Boot : initialisation de structures de données, lancement de la première tâche.
- IPC : communication entre tâches.

c La couche de communication

Cette couche comprend le code indépendant du matériel et implémentant la partie logicielle du comportement des services de communication. Elle peut utiliser des services de la couche OS mais aussi celle de la couche HAL pour communiquer avec la partie matérielle de l'interface. La couche de communication peut contenir des implémentations de services de communication simples comme l'envoi de données en utilisant une FIFO gardée ou implémenter des services de communication plus complexes. Si on prend l'exemple des services MPI, cette couche peut contenir l'implémentation de primitives de communication point à point fournissant des API du type MPI_SEND ou MPI_RECV. Elle peut aussi contenir des fonctions de gestion de type de données MPI, des fonctions de gestion de « communicators » ou encore des fonctions pour les communications collectives entre processus.

d Illustration de la structure en couches par un exemple simple

Le tableau suivant présente un petit exemple de partie logicielle générée par ASOG. Cette implémentation correspond à la réalisation de la partie logicielle d'un service de communication « get » sur une FIFO gardée. La partie matérielle est une simple FIFO envoyant une interruption à chaque changement d'état. Le tableau 1 présente les fonctions et les structures de données utilisées. Ces composants sont classés dans leurs catégories et couches respectives.

La couche des API contient une fonction : « Port.get » est la méthode utilisée par la tâche utilisant le service « get ».

La couche OS contient des fonctions de gestion des tâches, une fonction permettant d'effectuer un appel système et une fonction de traitement des interruptions. Les fonctions « __BlocBloc » et « __BlocUnBloc » permettent de bloquer ou de débloquer des tâches en attente de données. «__Sched_Sleep », « __Sched_WakeUp » permettent d'endormir et de réveiller une tâche. « _Sched_Schedule » est chargée d'élire la prochaine tâche à exécuter. « __trap_trap » est une fonction permettant d'effectuer un appel système. Enfin « __Interrupt » est une fonction de traitement permettant de vectoriser les fonctions traitant les interruptions matérielles.

La couche de communication contient une simple fonction implémentant le comportement du service de communication : cette fonction teste si la FIFO n'est pas vide. Elle demande la lecture d'une donnée dans la FIFO si celle ci n'est pas vid,e sinon elle ordonne le blocage de la tâche.

La couche HAL contient deux fonctions permettant de dialoguer avec la partie matérielle (« GuardedRegister_Get », « GuardedRegister_OK2_Get »). « __CxtSwitch » est une fonction écrite en assembleur, elle permet d'effectuer le changement de contexte. Enfin « __IRQ_Handler » et « _SWI_Handler » sont les sous routines de traitement d'interruptions matérielles et logicielles ».

Couche	Catégorie	Fonction	Structure de données
API		Port.get	
OS	Gestion de tâches	__BlocBloc __BlocUnBloc __Sched_Sleep __Sched_WakeUp __Sched_Schedule	bloc_blocs task_tasks
	Appels système	__trap_trap	
	Fonctions de traitement des interruptions	__Interrupt	calls
COM		Pipe_Get_GardedRegister	
HAL		GuardedRegister_Get GuardedRegister_OK2_Get __CxtSwitch __SWI_Handler __IRQ_Handler	

Tableau 2 : décomposition en couche de l'architecture logicielle de Pipe_Get

3.1.2 Architecture de la partie matérielle

Comme le montre la Figure 28, l'architecture de la partie matérielle est constituée de trois types d'éléments : un adaptateur de processeur aussi appelé MA pour « Module Adapter » connecté à un bus interne lui-même connecté avec des adaptateurs de canaux aussi appelés CA pour « Channel Adapter ».

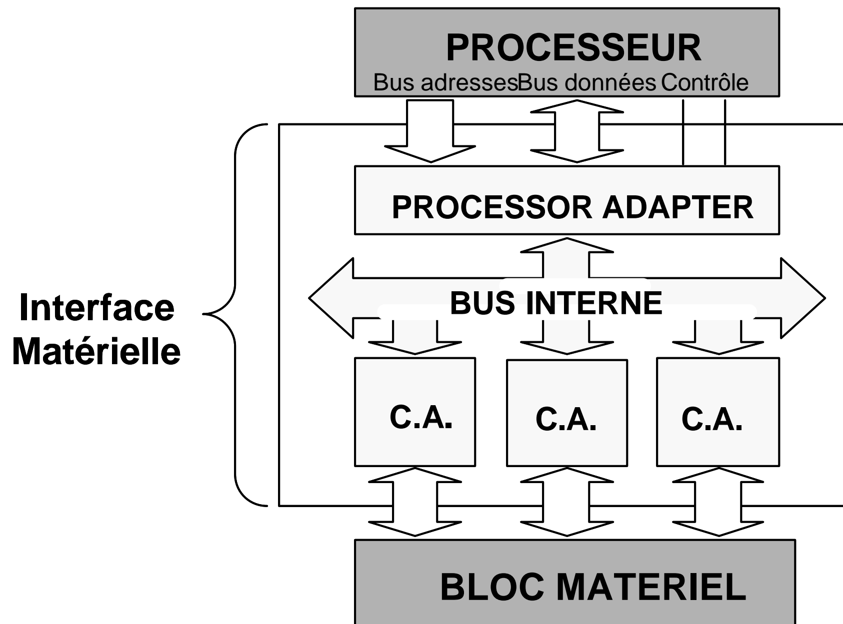

Figure 28 : Architecture de l'interface matérielle

a La couche d'adaptation du processeur

Cette couche permet l'adaptation du processeur au reste de l'interface matérielle. Elle a un rôle d'interconnexion entre le bus du processeur et le bus interne de l'interface. Elle contient aussi une partie dédiée à la gestion des requêtes d'interruption. Les requêtes émises par les adaptateurs de canaux sont stockées et arbitrées pour palier au nombre limité d'entrées de requêtes d'interruptions acceptées par les processeurs. Enfin cette couche a aussi un rôle de décodeur d'adresse permettant de diriger les données vers les adaptateurs de canaux.

b Les adaptateurs de canaux de communication

Les adaptateurs de canaux sont spécifiques aux services de communication implémentés. Ils implémentent la partie matérielle du comportement des services de communication. Ce type de composant est indépendant du processeur utilisé mais dépend par contre fortement du réseau de communication.

c Le bus interne

Le bus interne est une partie fixe de l'interface matérielle. Il permet de connecter n'importe quel CA avec n'importe quel MA.

3.2 Introduction à MPI

3.2.1 Présentation de MPI

La spécification de MPI a été développée pour permettre la programmation parallèle en utilisant une bibliothèque standard de communication. Avant son développement, de nombreuses bibliothèques ont été développées avec chacune des syntaxes différentes pour des sémantiques bien souvent similaires. MPI permet de rassembler les utilisateurs de bibliothèques de passage de messages autour d'un même standard.

Dans notre cas, nous utilisons MPI pour plusieurs raisons. Premièrement, ce standard a été conçu pour la communication entre processus possédant chacun leur propre mémoire. Hors les systèmes sur puce à base de processeurs hétérogènes contiennent des éléments correspondant à ce modèle : chaque tâche est une unité d'exécution possédant sa propre mémoire. Deuxièmement, le standard MPI est un standard de bibliothèque pour la communication par passage de message. MPI est donc prédisposé à être intégré dans des outils de génération automatique d'interfaces de communication, comme ASOG et ASAG, utilisant des bibliothèques de composants. Enfin, MPI semble de plus en plus étudié et utilisé, il permet donc d'illustrer les concepts utilisés dans nos méthodes de conception à l'aide de protocoles de services de communication relativement connus.

MPI signifie Message Passing Interface. C'est un standard de bibliothèque de communication développé pour permettre la programmation parallèle à l'aide de processus coopératifs. Le standard MPI existe en deux versions : MPI 1 et MPI 2. MPI 1 permet le communication par passage de message traditionnel alors que MPI 2 permet en plus l'accès à des mémoires distantes, la parallèlisation des IO et les processus dynamiques. Dans notre cas, nous nous limitons à l'étude de MPI 1.

MPI regroupe 125 fonctions permettant la communication et la synchronisation de processus. Les données sont envoyées sous forme de messages. Les processus sont coopératifs, c'est-à-dire que lorsque deux processus demandent explicitement la communication, l'un demande l'envoi du message et l'autre demande explicitement sa réception.

Les primitives de MPI peuvent être classées en 3 types : les primitives de communication, les primitives de dérivation de données et les primitives de configuration.

3.2.2 Les objets: définitions

MPI utilise plusieurs types d'objets qu'il semble utile de définir :

- Processus : c'est l'exécution d'un programme utilisant une zone mémoire propre
- Groupes : c'est un ensemble de processus ordonnés.
- Contexte : c'est un contexte de communication. Ce n'est pas un objet MPI mais il est utilisé pour définir des « communicators ». Il représente les modes de communication utilisés.
- « Communicators » : c'est l'union d'un groupe et d'un contexte. Un « communicator » représente donc un ensemble de processus numérotés auxquels on attache des modes de communications.

3.2.3 Les primitives de communications

Il existe deux types de primitives de communication : les primitives pour la communication point à point et les primitives pour la communication multi points.

a Communication Point à Point :

Les primitives de communication point à point sont elles aussi classables en deux sous ensembles : celui des communications bloquantes et celui des communications non bloquantes.

Les communications bloquantes comprennent 4 primitives pour l'envoi de données et 1 pour la réception des données (tableau ci-dessous).

Envoi de messages	Réception de messages
MPI_SEND : Send standard MPI_BSEND : Send bufferisé MPI_RSEND : « ready Send » MPI_SSEND : Send synchrone	MPI_RECV : Receive standard

Tableau 3 : Primitives de communication point à point bloquantes

MPI_RECV est utilisable pour réceptionner les messages envoyés par n'importe quelle primitive de type Send bloquant.

Les communications non bloquantes comprennent aussi 4 primitives pour l'envoi de données et 1 pour la réception de données (tableau ci-dessous). Quatre autres primitives permettent de gérer les communications pendantes.

Envoi de messages	Réception de messages	Synchronisation explicite et renseignement
MPI_ISEND : Send standard non bloquant MPI_IBSEND : Send bufferisé non bloquant MPI_IRSEND : « ready Send » non bloquant MPI_ISSEND : « Send » synchrone non bloquant	MPI_IRECV : Receive standard non bloquant	MPI_WAIT : Attend la complétion d'une communication MPI_TEST : Test l'état d'une communication MPI_PROBE : Permet de savoir s'il y a des communications pendantes MPI_CANCEL : Permet de désactiver une communication

Tableau 4 : Primitives pour les communications point à point non bloquantes

b Communication multi points :

La communication multi points au sein d'un groupe permet d'établir des communications collectives telles que de la diffusion de messages, de la dispersion, mais aussi du réassemblage et de la concentration de messages. Une primitive de synchronisation explicite permet de faire des points de synchronisations dans un groupe de processus communicants. Les principales primitives existantes sont présentées dans le tableau ci-dessous.

Envoi et réception de messages	Synchronisation explicite
MPI_BCAST : « Broadcast » MPI_SCATTER : « Splitting and Dealing » MPI_REDUCE : « Reassembling » MPI_GATHER : « concentrating »	MPI_BARRIER : permet de synchroniser un groupe de processus

Tableau 5 : Primitives pour les communications multi points

La Figure 29 montre les différents types de communications multi points permises par la bibliothèque MPI. Dans les tableaux présentés, chaque colonne représente un processus et chaque ligne correspond à une donnée parmi la suite de données à envoyer. Les grosses flèches indiquent le sens d'envoi des messages. Les dessins de la colonne de droite présentent les topologies de connexion propres à chaque type d'opération. Les «S» indiquent les « Senders » et les « R » les « Receivers ». Le premier type d'opération est le « broadcast » : chaque donnée envoyée par un processus est distribuée sur les autres processus. Ensuite l'opération « scatter » : chaque donnée envoyée par un processus est distribuée vers un processus différent. Avec l'opération « gather », un processus reçoit une suite de données dont chaque élément est envoyé par un processus différent. « allgather » ressemble à « gather » avec plusieurs processus « Senders ». Enfin « alltoall » ressemble à un « scatter » avec plusieurs processus « senders ».

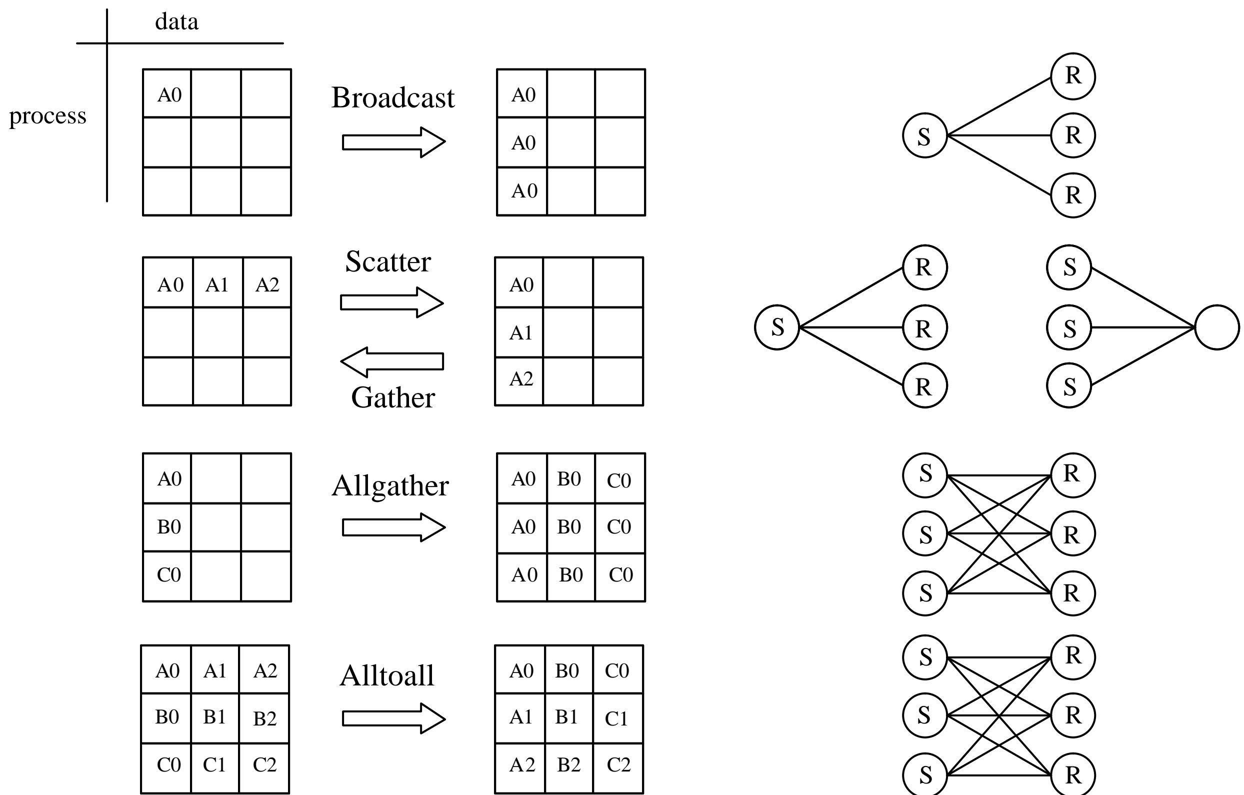

Figure 29 : Communications collectives

3.2.4 Types de données et primitives de dérivation de données

MPI contient plusieurs types de données prédéfinies. Mais il est aussi possible de construire ses propres structures de données constituées de compositions de types prédéfinis. Cette construction est effectuée en utilisant des primitives de dérivation de données. L'intérêt de la construction de structures de données réside dans la possibilité d'effectuer l'envoi de messages hétérogènes. Par exemples, on peut avoir besoin d'envoyer un message constitué d'un entier représentant la taille d'un liste, puis une série de données de type « float » constituant cette liste.

a Les types de données prédéfinis :

Les types prédéfinis sont attachés à un langage de programmation. La spécification MPI précise les types de données associés au langage C et au fortran. Dans notre cas seuls les types relatifs au C nous intéressent : le tableau ci-dessous présente les types prédéfinis pour le langage C.

MPI_CHAR MPI_SHORT MPI_INT MPI_LONG	MPI_UNSIGNED_CHAR MPI_UNSIGNED_SHORT MPI_UNSIGNED MPI_UNSIGNED_LONG	MPI_FLOAT MPI_DOUBLE MPI_LONG_DOUBLE MPI_BYTE MPI_PACKED

Tableau 6 : Les différents types de données

b Les primitives de dérivations :

A partir des types de données prédéfinis, il est possible de créer des types plus complexes à l'aide de primitives de construction de types. Le Tableau 7 présente les principales primitives ainsi que leur comportement associé.

Les primitives de construction de types	Comportement
MPI_TYPE_CONTIGUS	Crée un type constitué de blocs uniformes de types de données prédéfinis. En entrée, le nombre de blocs, le type de chaque bloc. Retourne le nouveau type.
MPI_TYPE_VECTOR	Crée un type constitué de blocs de tailles uniformes. Chaque bloc est constitué d'un nombre donné d'éléments d'un type prédéfini. En entrée, le nombre de blocs, le nombre d'éléments dans chaque bloc, l'espacement entre chaque bloc (l'unité est la taille d'un élément), le type des éléments. Retourne le nouveau type.
MPI_TYPE_HVECTOR	Comme MPI_TYPE_VECTOR, seul l'unité utilisée pour donner l'espacement entre chaque blocs change : elle est donnée en bytes.
MPI_TYPE_INDEXED	Crée un type constitué de blocs de tailles hétérogènes. Chaque bloc peut être constitué d'un nombre différent d'éléments d'un type prédéfini. En entrée, le nombre de blocs, un tableau du nombre d'élément pour chaque bloc, un tableau de la position du bloc à partir de l'adresse de base, le type des éléments. Retourne le nouveau type
MPI_TYPE_STRUCT	C'est une évolution de MPI_TYPE_INDEXED. Les blocs peuvent contenir une concaténation d'éléments d'un type différent. En entrée, au lieu de donner un seul type d'élément, on donne un tableau de types d'éléments.

Tableau 7 : Les primitives de dérivation

c Les primitives permettant d'extraire des informations à partir d'un type :

Il est possible d'extraire des informations telles que la taille du type (avec les primitives MPI_TYPE_EXTENT, MPI_TYPE_SIZE), l'adresse de début d'une zone d'adresse (MPI_ADDRESS),

l'adresse de la borne inférieure et l'adresse de la borne supérieure d'un type de données dérivé (MPI_TYPE_LB, MPI_TYPE_UB).

3.2.5 Les primitives de configuration

Ce paragraphe regroupe les primitives permettant la création, la manipulation et la destruction d'objets (processus, groupes, « communicators »).

a Configuration topologique des processus :

L'ensemble des processus et les liens virtuels qui leur permettent de communiquer forme une topologie virtuelle. Elle peut être représentée par un graphe constitué de nœuds, les processus, et d'arcs, les liens virtuels. La topologie virtuelle est un attribut de « communicator ».

- Création : La primitive MPI_GRAPH_CREATE permet de créer un graphe à partir d'un « communicator » et d'informations telles que le nombre de nœuds, le nombre de liens etc. Lorsque le graphe est régulier, il peut être remplacé par une représentation cartésienne de la topologie. La primitive MPI_CART_CRATE permet de créer une représentation cartésienne de la topologie à partir du nombre de dimensions, du nombre de processus dans chaque dimension etc.
- Test : Il est possible de tester le type de représentation topologique associée à un « communicator » avec la primitive MPI_TOP_TEST.
- Extraction d'informations : Un certain nombre de primitives permettent d'extraire des informations sur une topologie virtuelle. Par exemple MPI_GRAPHDIMS_GET permet d'extraire le nombre de nœuds et d'arcs.

b Configuration des « communicators » :

- Création : Tout nouveau « communicator » est créé à partir d'un ancien. Il existe un « communicator » par défaut (MPI_COM_WORLD) comprenant tous les processus du système. A partir de ce « communicator » par défaut, on peut créer tous les « communicators » en utilisant les primitives MPI_COMM_DUP et MPI_COMM_CREATE.
- Extraction d'informations : Un certain nombre de primitives permettent d'extraire des informations sur un « communicator ». MPI_COM_SIZE donne la taille du groupe associé au « communicator », MPI_COMM_RANK donne le rang du processeur dans le groupe associé au « communicator », MPI_COMPARE identifie si deux « communicators » ont le même groupe et le même contexte de communication etc.
- Destruction : La primitive MPI_GROUP_FREE permet de détruire un groupe.

c Configuration des groupes :

- Création : Il existe deux façons de créer un groupe. Soit on part d'un groupe existant, soit on l'extrait d'un « communicator ». Le tableau ci-dessous présente brièvement les primitives de création :

Création à partir d'un communicator	Création à partir d'un ou de plusieurs groupes
MPI_COMM_GROUP	MPI_GROUP_UNION : union de deux groupes MPI_GROUP_INTERSECTION : intersection entre deux groupes MPI_GROUP_INCL : inclusion de processus dans un groupe MPI_GROUP_DIFFERENCE : différence entre deux groupes MPI_GROUP_EXCL : un groupe duquel on exclut des processus

Tableau 8 : Primitives de création de groupes

- Extractions d'informations : Un certain nombre de primitives permettent d'extraire des informations d'un groupe. MPI_GROUP_SIZE donne la taille du groupe, MPI_GROUP_RANK donne le rang du processeur dans le groupe etc.
- Destruction : La primitive MPI_GROUP_FREE permet de détruire un groupe.

3.2.6 Divers

a Notions d'« intra-communicators » et d' « inter-communicators »:

Les «communicators » considérés jusqu'à maintenant était constitués d'un groupe et d'un contexte de communication. Il permettait de faire des communications collectives au sein d'un groupe ainsi que des communications point à point entre processus appartenant au groupe. Ce type de « communicators » peut être qualifié d' « intra-communicator ». Cependant un autre type de « communicator » existe, il permet de faciliter la programmation d'applications de type serveur-client. Ce type de «communicator» est qualifié d' «inter-communicator». Il modélise la communication entre groupes de processus. Il est possible de tester si un « communicator » est un « inter-communicator » avec la primitive MPI_COMM_TEST.

b Gestion des erreurs:

La gestion des erreurs est fortement dépendante de l'implémentation. La spécification laisse beaucoup de liberté sur ce sujet. En tous cas, les communications effectuées en utilisant une primitive MPI avec les bons paramètres et les bonnes conditions d'utilisation sont considérées comme sûes. En fait, l'implémentation doit assurer la sûreté de la communication (bonne copie des données etc.). Par contre, il est bien sûr possible d'avoir des erreurs d'arguments, de tag, d'utilisation de primitives dans un mauvais contexte. Dans ce genre de situation, la spécification MPI contient un ensemble de primitives permettant de créer et de gérer des «handlers » d'erreurs. Deux « handlers » d'erreurs ont été prédéfinis. MPI_ERRORS_ARE_FATAL doit arrêter l'exécution de tous les processus. MPI_ERRORS_RETURN doit simplement retourner un code d'erreur.

c Ouverture et fermeture d'un session MPI:

On ouvre une session MPI en utilisant MPI_INIT. On ferme cette session en utilisant MPI_FINALIZE.

3.3 Sélection d'un sous ensemble de primitives MPI

3.3.1 Description détaillée du comportement des primitives de communication point à point

Cette sous-section présente un zoom sur le comportement des primitives de communication point à point. Ces primitives sont pour nous les plus intéressantes puisqu'elles présentent le meilleur rapport entre coût de développement et intérêt pour notre projet.

a Les communications bloquantes

Une communication est appelée communication bloquante lorsque la tâche exécutant cette primitive de communication est bloquée tant que le message n'a pas été copié « ailleurs », dans le cas d'un « Send », ou copier dans le tampon de destination, dans le cas d'un « Receive ».

MPI_BSEND :

MPI_BSEND est permet d'envoyer des messages en utilisant un buffer. Tant qu'il y a suffisamment de place dans le tampon de communication, le MPI_BSEND peut être appelé indépendamment de l'appel d'un MPI_RECV. Si la place dans le tampon de communication est remplie et si le message n'est pas totalement copié, une erreur doit être générée.
La Figure 30 montre une tentative d'interprétation de ce comportement en montrant les échanges implicites et explicites : les échanges implicites sont des flèches en pointillés alors que les échanges explicite sont en trait plein. Les rectangles modélisent l'exécution d'une primitive.

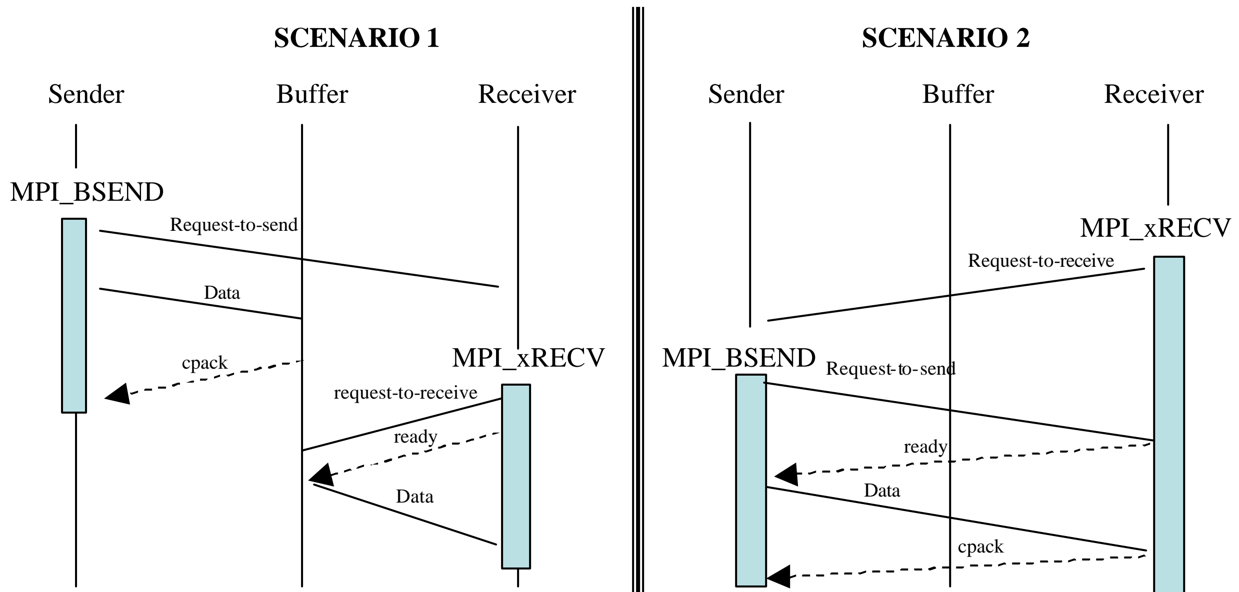

Figure 30 : Graphe de transactions et différents scénarios pour MPI_BSEND

La syntaxe d'appel de la primitive est la suivante :
MPI_BSEND(buf, count, datatype, dest, tag, comm)

- Buf : adresse du tampon source
- Count : nombre d'éléments constituant le message

- Datatype : type des éléments
- Dest : identificateur du processus destinataire
- Tag : identificateur de message
- Comm : communicator (groupe de processus communicants)

Ces paramètres sont communs à tous les autres Send bloquants.

MPI_RSEND :

MPI_RSEND ne peut être appelé que si le « Receiver » est prêt. Si ce n'est pas le cas, une erreur est générée. Si le receiver est prêt, les données sont envoyées directement vers le « Receiver ».

La Figure 31 montre une tentative d'interprétation de ce comportement en montrant les échanges implicites et explicites.

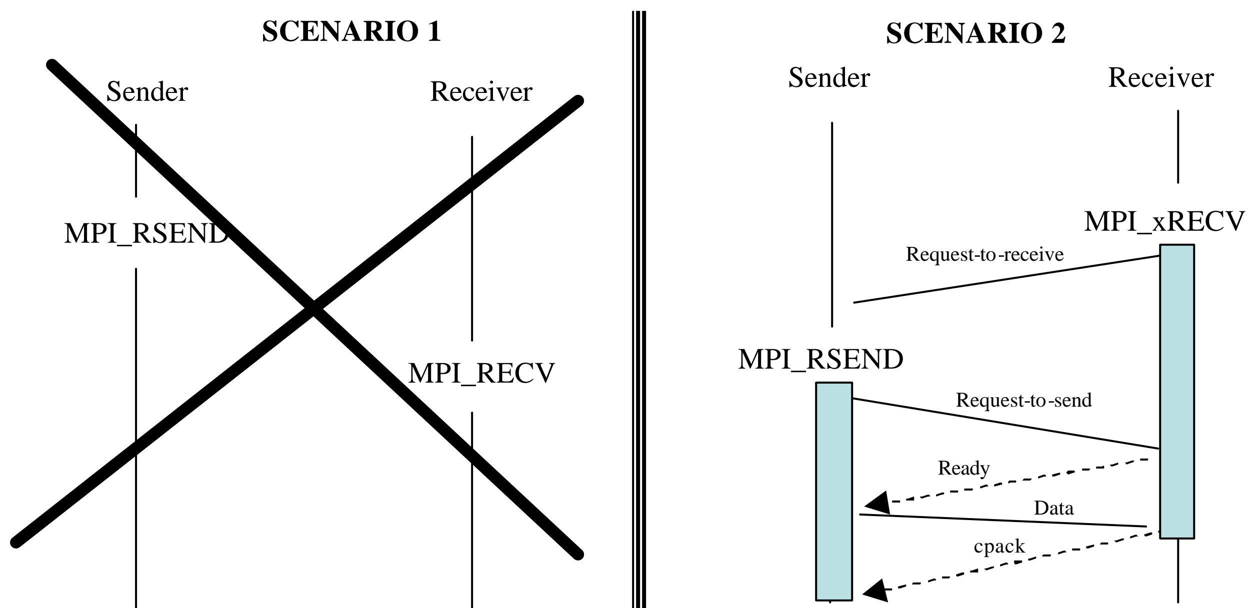

Figure 31 : Graphe de transactions et différents scénarios pour MPI_RSEND

La syntaxe d'appel de la primitive est la suivante :

MPI_RSEND(buf, count, datatype, dest, tag, comm)

MPI_SSEND :

MPI_SSEND a le même comportement qu'un MPI_SEND standard sans tampon. L'appel du « Send » peut être fait indépendamment de l'appel « Receive » du « Receiver ». Si le « Receiver » n'est pas prêt, le processus « Sender » est bloqué. Si le « Receiver » est prêt, les données sont envoyées au « Receiver ».

La Figure 32 montre une tentative d'interprétation de ce comportement en montrant les échanges implicites et explicites.

La syntaxe d'appel de la primitive est la suivante :

MPI_SSEND(buf, count, datatype, dest, tag, comm)

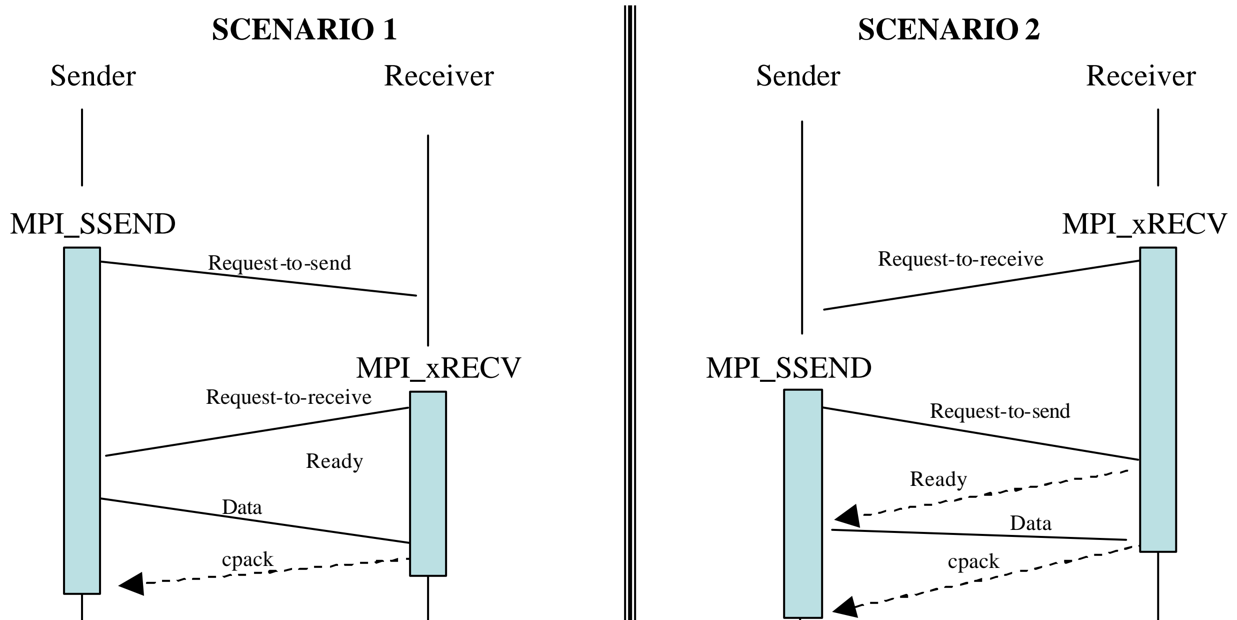

Figure 32 : Graphe de transactions et différents scénarios pour MPI_SSEND

MPI_SEND standard :

Le « Send » standard peut être décliné en deux versions : il existe une version avec tampon et une version sans tampon. Un « Send » standard consiste simplement à copier le message en dehors du tampon source. Pour la version avec tampon, le message sera copier dans un tampon de communication. Le MPI_SEND standard avec tampon est donc indépendant de l'appel d'un MPI_RECV. Si le tampon n'est pas suffisant pour stocker le message, la tâche est bloquée en attente d'une libération du tampon. Pour la version sans tampon, le message est envoyé directement au processus destinataire. Les données ne sont copiées que si le « Receiver » est prêt. Si le « Receiver » n'est pas prêt, le processus du « Sender » est bloqué. En fait la version sans tampon équivaut à un MPI_SSEND et la version avec tampon correspond à un MPI_BSEND tolérant (pas d'erreurs lorsque le tampon est plein). Le comportement des deux versions du Send standard peut donc être représenté par la Figure 32 et la Figure 30.

La syntaxe d'appel est la suivante :

MPI_SEND(buf, count, datatype, dest, tag, comm)

MPI_RECV :

MPI_RECV est utilisé pour recevoir les messages envoyés par MPI_SEND, MPI_BSEND, MPI_RSEND, MPI_SSEND. L'appel MPI_RECV peut être fait indépendamment de l'appel du « Send ». Si le Sender n'est pas prêt à envoyer le message, le processus Receiver est bloqué. Si le « Sender » est prêt où si le message a été envoyé dans un tampon de communication ou si les données sont copiées dans le tampon destination.

La syntaxe d'appel de la primitive est la suivante :

MPI_RECV(buf, count, datatype, dest, tag, comm)

- Buf : adresse du tampon destination
- Count : nombre d'éléments constituant le message à recevoir
- Datatype : type des éléments

- Source : identificateur du processus émetteur
- Tag : identificateur de message
- Comm : communicator
- Ierror : statut d'erreur

b Les communications non bloquantes

Une primitive de communication est dite non bloquante si la procédure peut retourner avant que la copie des donnée est été complètement effectuée et notamment avant que le tampon (source ou destination) spécifié lors de l'appel ne soit réutilisable par le processus. Chaque appel de primitive de communication MPI non bloquante est qualifié d'initiation de communication. Une fois initiée, une communication peut être complétée lorsque les données sont complètement copiées ou rester pendante. Toutes les primitives de communication non bloquante reprennent le nom des versions bloquantes avec un I en plus signifiant « Immediate ».

MPI_ISSEND :

La primitive MPI_ISSEND, comme toutes primitives non bloquantes, retourne lorsque une requête de communication à été émise. Le «Message Send Manager» reçoit une requête pendante initiée par l'appel MPI_ISSEND. Cette requête de communication pourra être complétée si et seulement si une requête de « Receive » a été initiée et elle se fera lorsque le « Sender » saura que la copie des données va être faite. La Figure 33 illustre le comportement de MPI_ISSEND. MRM signifie « Message Receive Manager » et MSM signifie « Message Send Manager ».

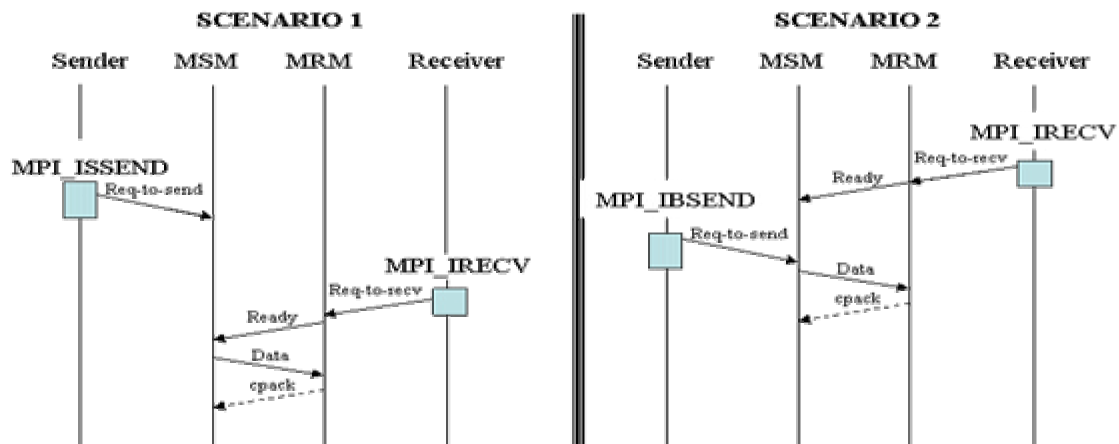

Figure 33 : Graphe de transactions et différents scénarios pour MPI_ISSEND

La syntaxe d'appel de la primitive est la suivante :

MPI_ISSEND(buf, count, datatype, dest, tag, comm, request)

- Buf : adresse du tampon source
- Count : nombre d'éléments constituant le message
- Datatype : type des éléments
- Dest : identificateur du processus destinataire
- Tag : identificateur de message
- Comm : communicator
- Request : requête de communication

Toutes le autres primitives pour l'envoi d'un « Send » non bloquant utilisent les mêmes paramètres.

MPI_IBSEND :

La Figure 34 illustre le comportement MPI_IBSEND. Comme pour la version bloquante de cette primitive, une erreur est générée lorsque le tampon est plein. La requête de communication sera complétée lorsque la requête de « Receive » correspondante aura été reçue et que le « Sender » saura que la copie des données va être faite.

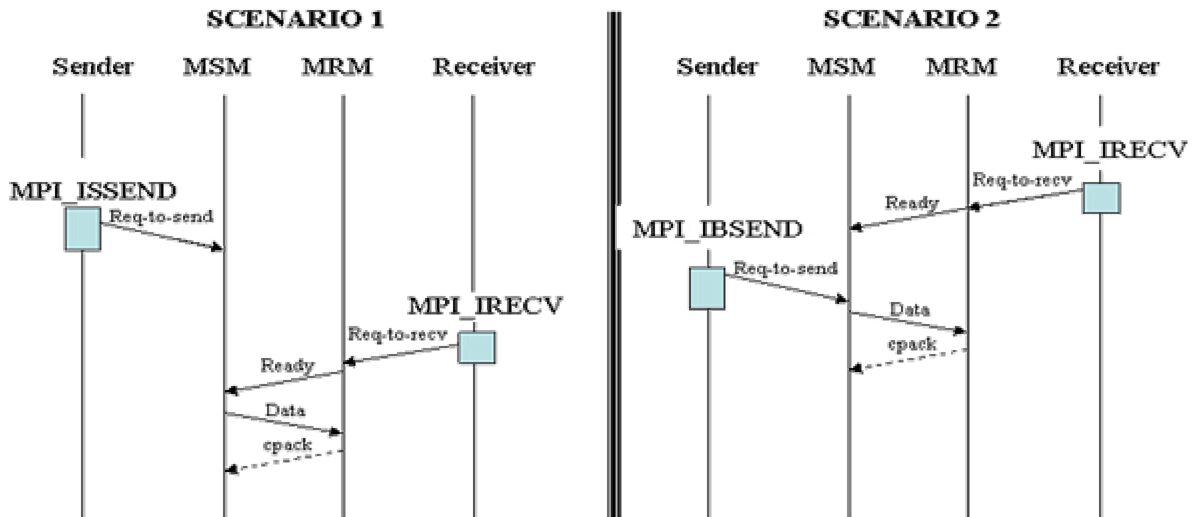

Figure 34 : Graphe de transactions et différents scénarios pour MPI_IBSEND

La syntaxe d'appel de la primitive est la suivante :
MPI_IBSEND(buf, count, datatype, dest, tag, comm, request)

MPI_IRSEND :

La Figure 35 illustre le comportement MPI_IRSEND. Comme pour la version bloquante, un appel « Send » ne peut être fait avant l'appel du « Receive » correspondant, sinon une erreur est générée. La complétion de la requête de communication initiée par MPI_IRSEND se fera lorsque le « Sender » saura que la copie des données va être faite.

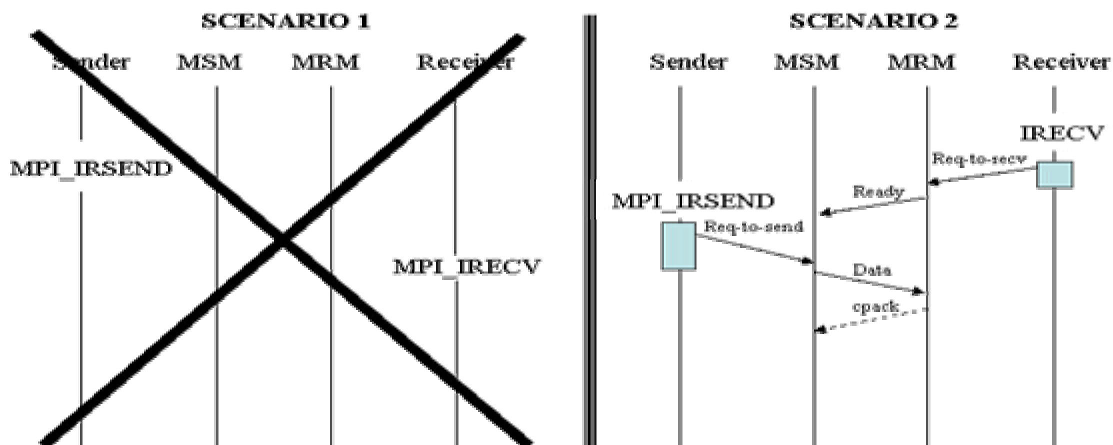

Figure 35 : Graphe de transactions et différents scénarios pour MPI_RSSEND

La syntaxe d'appel de la primitive est la suivante :
MPI_IRSEND(buf, count, datatype, dest, tag, comm, request)

MPI_ISEND :

C'est la version non bloquante du MPI_SEND standard. Il y a donc une version avec tampon et une version sans tampon. La première s'apparente à un MPI_IBSEND ne générant pas d'erreur lorsque le tampon est plein. La deuxième se comporte comme un MPI_ISSEND. Le comportement de ces deux versions peut donc être illustré par les Figure 34 et Figure 33.

MPI_IRECV :

On peut aisément retrouver le comportement de cette primitive en étudiant les trois figures précédentes.
Le prototype et les paramètres utilisés sont les suivant :
MPI_IRECV(buf, count, datatype, dest, tag, comm, request)

- Buf : adresse du tampon de destination
- Count : nombre d'éléments constituant le message à recevoir
- Datatype : type des éléments
- Source : identificateur du processus émetteur
- Tag : identificateur de message
- Comm : communicator (groupe de processus communicants)
- Request : requête de communication
- Ierror : statut d'erreur

MPI_TEST :

Cette primitive permet de tester l'état d'une requête de communication. Il est aussi possible de tester plusieurs communications en même temps.

MPI_WAIT :

Cette primitive sert à attendre la complétion d'une requête de communication. Elle retourne lorsque la requête de communication a été complétée. Il est aussi possible d'attendre la complétion de plusieurs requêtes.

MPI_PROBE :

Cette primitive permet de savoir s'il y a des requêtes de communication pendantes.

MPI_CANCEL :

Permet d'annuler une requête de communication pendante.

3.3.2 Critères de sélection du sous ensemble

a - Rôle d'illustration:

Pour permettre d'illustrer les avantages de l'utilisation de bibliothèques d'éléments pour la réutilisation, il faut que le nombre de primitives à implémenter soit suffisant et que ces primitives utilisent des composants communs. Cela nous permettra d'estimer le gain en temps de développement apporté par la méthode.
Les primitives choisies doivent être intéressantes pour le partitionnement logiciel pour permettre d'illustrer les possibilités de partitionnement logiciel/ matériels et les bénéfices qui en découlent.

b - Utilisation :

L'un des objectifs évoqués dans ce projet est d'offrir un jeu de primitives de communication suffisamment important pour permettre le développement ultérieur d'applications utilisant le standard MPI.

c - Temps d'implémentation :

Le temps de développement est limité, il ne faut pas que l'implémentation du sous-ensemble prenne plus de 4 mois.

3.3.3 Sous-ensemble sélectionné

Le sous-ensemble sélectionné est constitué de 3 primitives de communication point à point bloquantes. Les communications multi points sont écartées car elles nécessitent le développement parallèle des primitives de gestion de groupe et de communications. Les communications point à point non bloquantes sont intéressantes car elles permettent de bien utiliser le parallélisme entre communication et calcul. De plus elles offrent un peu plus de complexité que les versions bloquantes. Deux de ces primitives ont donc été retenues.

	Bloquantes	Non bloquantes
Primitives sélectionnées	- MPI_SEND - MPI_SSEND - MPI_RECV	- MPI_ISSEND - MPI_IRECV - MPI_WAIT

Figure 36 : Les primitives MPI sélectionnées

3.4 Conclusion

Ce chapitre a permis de présenter l'architecture en couche des interfaces logicielles/matérielles. Il a aussi permis de présenter le modèle de programmation parallèle MPI. Une introduction a été faite pour montrer les possibilités offertes par le standard. Il nous aussi permis d'illustrer ce que pouvait être un service de communication complexe. La dernière partie du chapitre s'est focalisée sur les primitives de communication point à point dont quelques unes ont été sélectionnées pour l'expériences présentée à la section 5.2.

Chapitre 4 Propositions pour la génération automatique d'interfaces logicielles/matérielles en vue de l'exploration du partitionnement des services de communication

Comme nous l'avons expliqué dans la présentation de la problématique au chapitre 1, nous partons d'une spécification de services de communication et nous voulons obtenir une implémentation sous forme d'interfaces logicielles/matérielles. Pour cela, nous utilisons un modèle de représentation du service de communication que nous découpons en une partie logicielle et une partie matérielle. Ensuite une phase de génération automatique permet de passer du modèle du service partitionné à l'implémentation au niveau RTL. Ce chapitre présente les éléments nécessaires à l'élaboration d'un flot permettant l'implémentation logicielle/matérielle des services de communication en vue de l'exploration des solutions de partitionnement. Pour cela, il présente en premier lieu une définition plus précise de la problématique du partitionnement logiciel/matériel avec une proposition de flot. Ensuite, il présente une proposition de modèle de représentation des services de communication permettant l'utilisation des outils ASOG et ASG pour la génération des interfaces logicielle/matérielles. Enfin, il présente l'étape de génération automatique en proposant une méthode pour la génération de la partie logicielle et en soulignant les besoins spécifiques à la génération de la partie matérielle.

4.1 Partitionnement logiciel/matériel

Cette section présente le partitionnement logiciel/matériel, en explique le principe ainsi que les critères qui guident le choix du partitionnement. Elle présente aussi le flot de partitionnement logiciel/matériel des réalisations de services de communication.

4.1.1 Le principe du partitionnement

Le principe du partitionnement logiciel/matériel est simple. On part d'un service de communication caractérisé par un comportement. Ce comportement doit être alloué dans une architecture d'interface logicielle matérielle en utilisant les avantages et inconvénients du logiciel et du matériel pour tenter de respecter les contraintes de conception imposées. Il y a donc une phase de choix d'allocation dicté par les contraintes de conception et une phase d'allocation (Figure 37). A gauche, cette figure présente un découpage de service de communication MPI_SEND et MPI_RECV en unités fonctionnelles et unités de stockage de données. Ce modèle sera détaillé dans la section 4.2. Chaque unité est allouée soit dans la partie logicielle de l'interface, soit dans la partie matérielle.

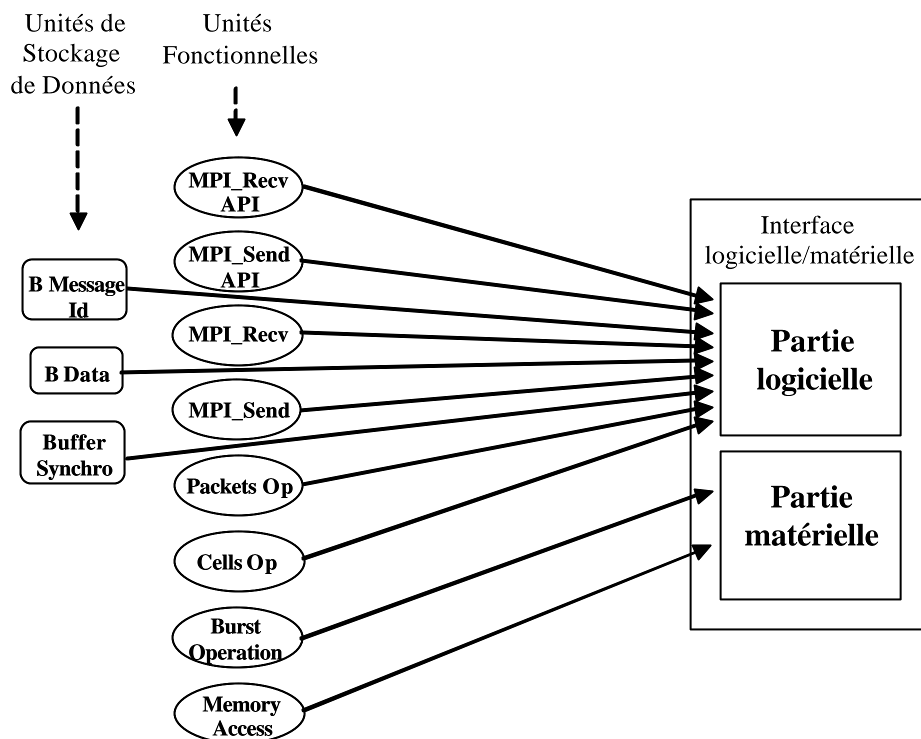

Figure 37 : Allocation des unités

4.1.2 Critères de choix du partitionnement logiciel/matériel

a Les avantages des réalisations logicielles

Le principal avantage des réalisations logicielles réside dans leur flexibilité. Cette flexibilité est multiple. Elle permet de faire évoluer un produit même après commercialisation. Le client peut en effet par exemple charger des patches permettant l'acquisition de nouvelles fonctionnalités ou d'adapter le produit à de nouvelles normes. Il est aussi possible de corriger des erreurs de programmation très tard dans le flot de conception, soit juste avant la commercialisation voir même après celle-ci. Enfin, l'ajout de nouveaux modules logiciels est beaucoup plus simple que pour l'ajout de modules matériels.

Même si le débuguage du logiciel sur un système prend une part importante du temps de fabrication d'un système sur puce, le simple fait de ne pas avoir à fabriquer des masques comme pour le matériel rend le développement du logiciel moins coûteux. De plus, les erreurs de conception coûtent toujours moins cher qu'avec la conception du matériel toujours à cause des moyens mis en oeuvre lors de la production de la partie matérielle.

b Les avantages des réalisations matérielles

Contrairement à un processeur. Un ASIC dédié ne comporte que les composants nécessaires à la réalisation des fonctionnalités. Le flot de donné est spécifique à l'implémentation et non pas standard comme celui d'un bus. Il n'y a pas de contrainte de cadencement calqué sur le traitement d'instruction le plus long comme sur un processeur. La liberté d'implémentation d'un ASIC étant plus grande que celle d'un processeur, l'optimisation d'un ASIC permet généralement de dépasser les performances de l'implémentation des mêmes fonctionnalités en logiciel. Pour les mêmes raisons, la consommation d'une implémentation logicielle est plus importante qu'une implémentation matérielle. La surface est aussi un critère plaidant pour l'implémentation matérielle.

c Influence des interactions entre logiciel et matériel sur la performance des communications

Lorsqu'un comportement de communication implique un enchaînement séquentiel d'actions, il faut que l'implémentation logicielle/matérielle respecte un ordre d'exécution. Il est donc nécessaire de synchroniser la partie logicielle et matérielle. Selon le partitionnement des fonctionnalités, le nombre de synchronisations et le type de communications entre logiciel et matériel peuvent grandement varier. Certains partitionnements ne sont donc pas intéressants si le temps de synchronisation et de communication entre partie logicielle et partie matérielle devient trop important.

4.1.3 Flot de partitionnement logiciel/matériel des réalisations de services de communication

a Les méthodes usuelles de partitionnement des interfaces logicielles/matérielles de communication

La méthode classique de partitionnement :

Généralement, le chef de projet chargé du développement d'une application à base de processeurs et d'ASIC, choisit dans un premier temps le partitionnement logiciel matériel du système complet. Il découpe le comportement de son application, choisit les composants qu'il veut utiliser et fait l'allocation des différentes

fonctionnalités sur les différents composants. Pour cela, il peut utiliser des outils d'aide au partitionnement ou tout simplement utiliser son expérience et celle de l'équipe de développement. Ensuite, les développeurs logiciel programment l'architecture choisie ou conçue. Dans certain cas, si les performances des communications entre processeurs deviennent critiques, il peut être amené à envisager la conception d'accélérateurs de communication avant que l'équipe du logiciel développe sa partie. En tous cas, même si le problème du partitionnement peut être appréhendé dans son ensemble avant le développement de l'architecture, l'ordre de développement des interfaces logicielles/matérielles est toujours le même. On développe le logiciel après avoir développé le matériel.

La migration de fonctionnalité vers le matériel :

Une autre forme de partitionnement logiciel/matériel des interfaces de communication est celle de la migration de fonctionnalités. On part d'une implémentation existante et on veut réaliser une nouvelle implémentation respectant des contraintes de performances, flexibilité, ou de consommation différentes. Il est alors intéressant d'extraire certaines fonctionnalités pour les implémenter différemment. Par exemple, pour obtenir de meilleures performances, on peut extraire une fonctionnalité implémentée en logiciel pour l'implémenter en matériel. Ou alors, pour obtenir plus de flexibilité, certaines fonctionnalités réalisées en matériel peuvent migrer vers une réalisation logicielle.

b Vers l'automatisation du partitionnement

Voici une proposition de flot permettant l'automatisation du partitionnement logiciel/matériel automatique des implémentations de services de communications. La Figure 38 montre un exemple de flot d'outils nécessaires pour automatiser le partitionnement des implémentations de services de communication. On part d'une spécification de service de communication. Le premier outil permet d'effectuer un premier choix d'allocation de fonctionnalités en logiciel ou en matériel. Pour cela cet outil doit utiliser deux types d'informations : des règles de sélection et des informations de base sur les implémentations possibles de chaque unité fonctionnelle. Par exemple, pour les informations de base, les unités pourraient être annotées selon un système de notation simplifiée permettant des études comparatives en terme de surface, performance, consommation, flexibilité selon le style d'implémentation. Les règles de sélection utiliseraient les critères exposés à la sous-section 4.1.2.

L'outil d'exploration est un outil simple attribuant aléatoirement une architecture générique pour l'implémentation de l'interface logicielle/matérielle. Cet outil sera plus efficace si le concepteur peut intervenir dans le choix du modèle d'architecture.

L'outil de génération d'implémentations génère une implémentation à partir d'un modèle de service de communication partitionné et d'un modèle générique d'interface.

Les outils « caractérisation », « estimation de consommation » et « estimation de performance » seront les sens du flot, il permettront de juger les implémentations générées.

Enfin le dernier outil est un outil de contrôle. Il doit arrêter l'exploration lorsque les contraintes sont atteintes. Il doit aussi contrôler le retour dans la boucle 1 ou 2 si les contraintes ne sont pas atteintes. La boucle 1 permet de refaire un choix de partitionnement alors que la boucle 2 permet de faire une exploration architecturale.

Cet exemple de flot contient plusieurs phases de choix : avec le choix de partitionnement des fonctionnalités, le choix d'architecture, le choix de continuer ou non l'exploration. Ces phases sont sûrement les plus difficiles à automatiser. On peut alors se demander lesquels de ces choix sont les plus urgents à automatiser, lesquels offrent le meilleur rapport entre rapidité et qualité, entre coût d'élaboration de l'automatisation et coût d'utilisation de du personnel.

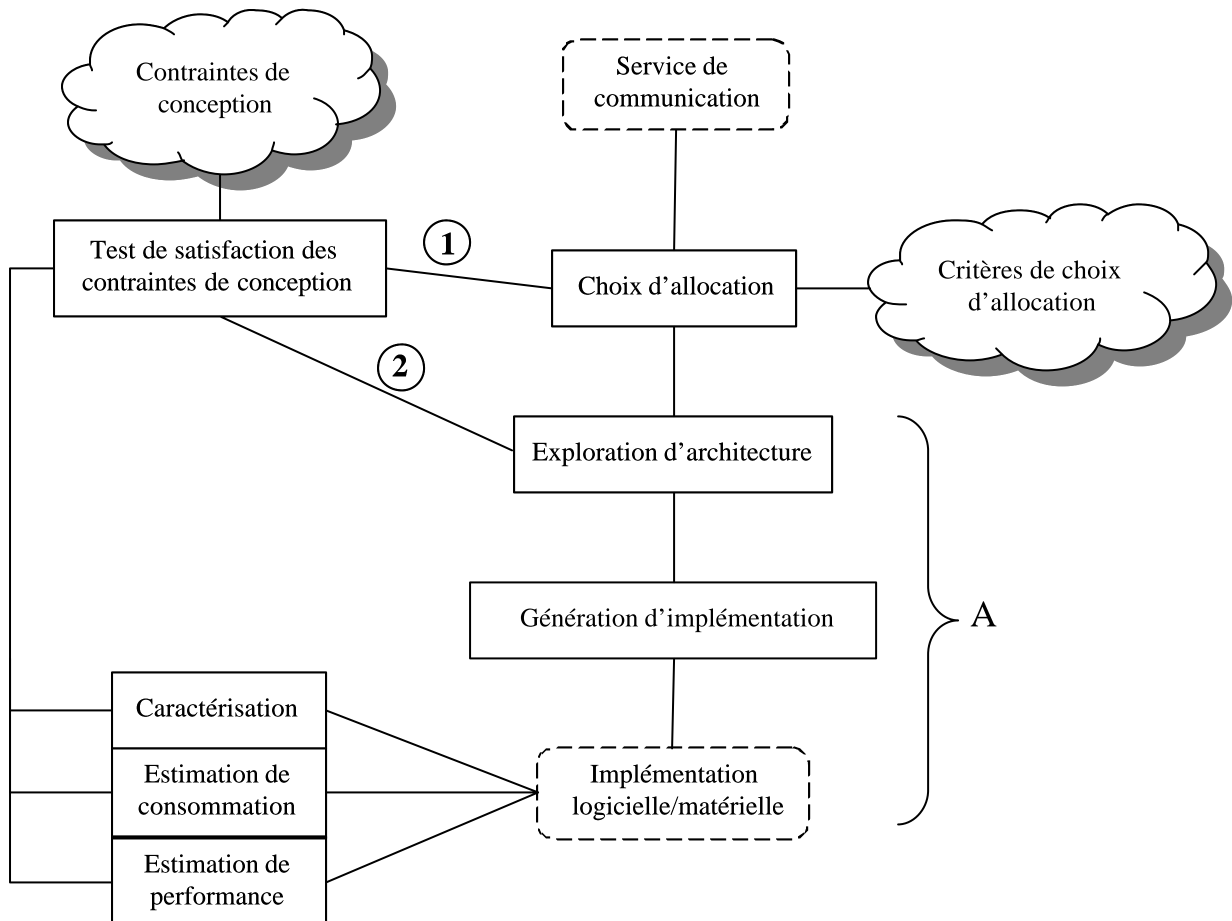

Figure 38 : flot de partitionnement logiciel/matériel des implémentations de services de communication

Notre travail porte sur la partie A du flot présenté à la Figure 38. La Figure 39 montre une proposition de réalisation de cette partie du flot. Cette proposition consiste en l'utilisation d'un modèle pour représenter les services de communication et en l'utilisation des outils ASOG et ASAG pour générer les interfaces logicielles/matérielles. Les sections suivantes présentent le modèle de représentation et le nouveau flot de génération automatique d'interfaces logicielles/matérielles.

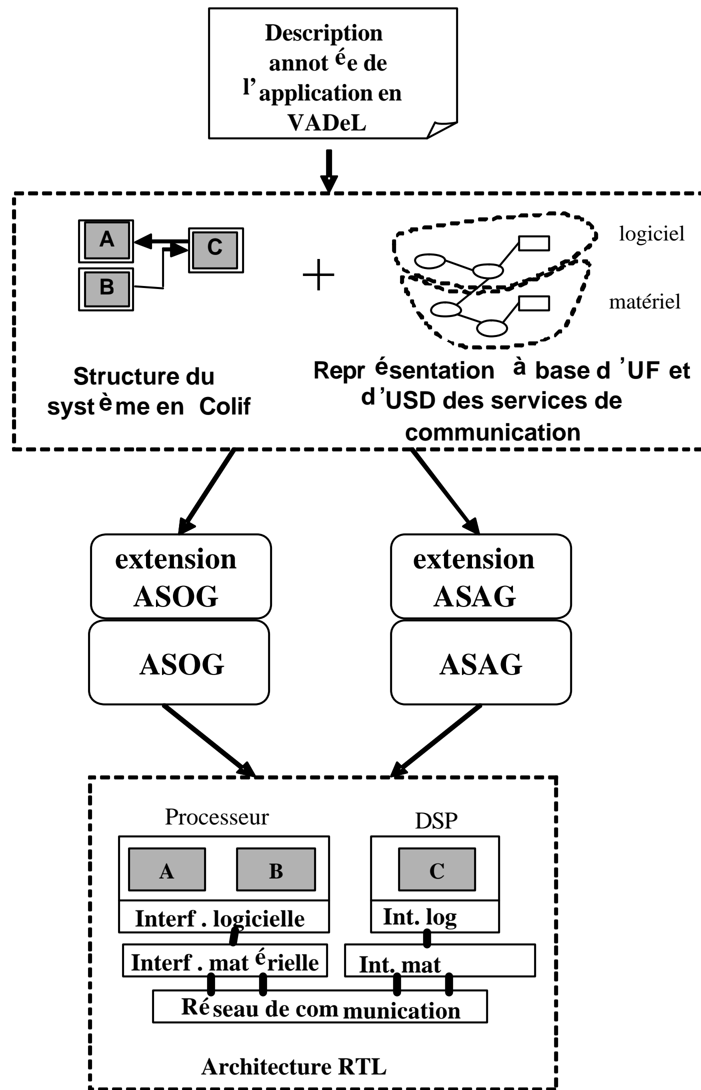

Figure 39 : Flot de génération des interfaces logicielles/matérielles

4.2 Modèle de représentation des services de communication

Cette section présente une proposition de modèle de représentation des services de communication. Nous présentons l'utilité d'un tel modèle puis sa description.

4.2.1 Utilité d'un modèle de représentation des services de communication

a Modèle commun pour les outils de génération automatique

Le modèle de représentation des services de communication doit pouvoir être interprété et traité par les deux outils de génération automatique.

b Description des fonctionnalités du service de communication

Il doit permettre au concepteur d'applications de comprendre ce qui ce cache derrière le service de communication spécifié. Il faut qu'il puisse évaluer la complexité des fonctionnalités mises en œuvre. Les outils

de génération doivent générer des implémentations ayant le comportement spécifié. Le modèle de représentation sert de modèle aux outils de génération.

c Abstraction des détails propres à l'implémentation logicielle ou matérielle

Le modèle doit être indépendant du style d'implémentation. Il ne doit donc pas montrer des détails spécifiques à l'implémentation en logiciel ou en matériel. Tout détail impliquant un type d'implémentation réduirait l'espace des solutions de partitionnement et donc d'implémentation.

4.2.2 Description du modèle de représentation

Pour le modèle proposé, nous nous sommes inspirés du modèle de représentation des éléments de bibliothèque d'ASOG développé par Lovic Gauthier. Nous rappelons, dans un premier temps, les principes de ce modèle, puis celui proposé pour représenter les services de communication.

a Modèle à base de services et éléments de Lovic Gauthier

Utilisation du modèle à base d'éléments et de services :

Le modèle développé par Lovic Gauthier [GAU 02] a été créé pour permettre l'assemblage de morceaux de code générique pour la génération automatique d'OS ciblé. Ce modèle servait donc de modèle d'entrée pour un interpréteur de bibliothèque et était spécifié par un développeur de bibliothèque.

Détails du modèle :

Ce modèle est composé d'éléments, de services et d'implémentations. L'élément représente un ensemble de fonctionnalités pouvant être implémentées de différentes façons. Un nom d'élément représente un ensemble de fonctionnalités.

Une partie de ces fonctionnalités est exprimée sous forme de services. Un service fourni est une fonctionnalité offerte par un élément alors qu'un service requis par un élément est une fonctionnalité demandée.

La Figure 40 présente les différents objets composant le modèle. Une flèche provenant d'un service et pointant vers un élément indique que ce service est fourni par l'élément désigné. Une flèche provenant d'un élément et pointant vers un service indique que l'élément a besoin du service désigné. Les flèches en pointillés illustrent la possibilité d'avoir plusieurs implémentations d'un même élément selon l'architecture cible choisie.

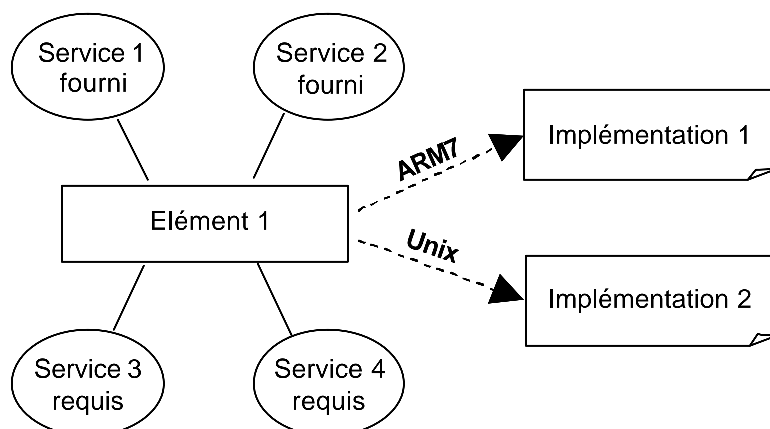

Figure 40 : Modèle à base d'éléments, services et implémentations

b Evolution proposée : Description à base d'Unités Fonctionnelles et d'Unités de Stockage de Données

Description du modèle :

Le modèle est composé d'Unités Fonctionnelles (UF) et d'Unités de Stockage de Données.
Une UF représente un ensemble de fonctionnalités du comportement du service de communication. Une USD représente une entité de mémorisation nécessaire au comportement du service de communication. Par exe mple dans le cas d'un MPI_BSEND un tampon est explicitement spécifié dans la spécification de la primitive. Le modèle de représentation de cette primitive devra donc contenir au moins une USD représentant ce tampon. Une UF accède à une autre UF pour utilis er l'un de ses services. Une UF accède à une USD pour lire ou écrire des données.

La Figure 41 montre un extrait du modèle de représentation de la primitive MPI_SEND avec tampon. Les ovales représentent des Unités Fonctionnelles et les rectangles des Unités de Stockage de Données. Une flèche représente l'accès à une UF ou un USD par une UF. L'USD « status table » sert à stocker le statut (l'état) des messages. L'UF « message status controller », écrit et lit dans l'USD pour stocker le statut d'un nouveau message ou consulter celui d'un message existant. «message management » est l'unité fonctionnelle qui contrôle les messages : elle utilise « address decoder » pour connaître la destination du message selon le numéro de processus du destinataire. Elle utilise « buffer contoller » pour stocker les identificateurs de messages non envoyés. Elle utilise aussi les services de « requ2send signalization » pour indiquer au processus récepteur que l'envoyeur veut envoyer un message. Enfin, l'unité « message management » utilise l'unité « message sender » pour donner l'ordre d'envoyer un message ou le stockage ses données dans un tampon mais aussi l'envoi d'un message stocké dans le tampon.

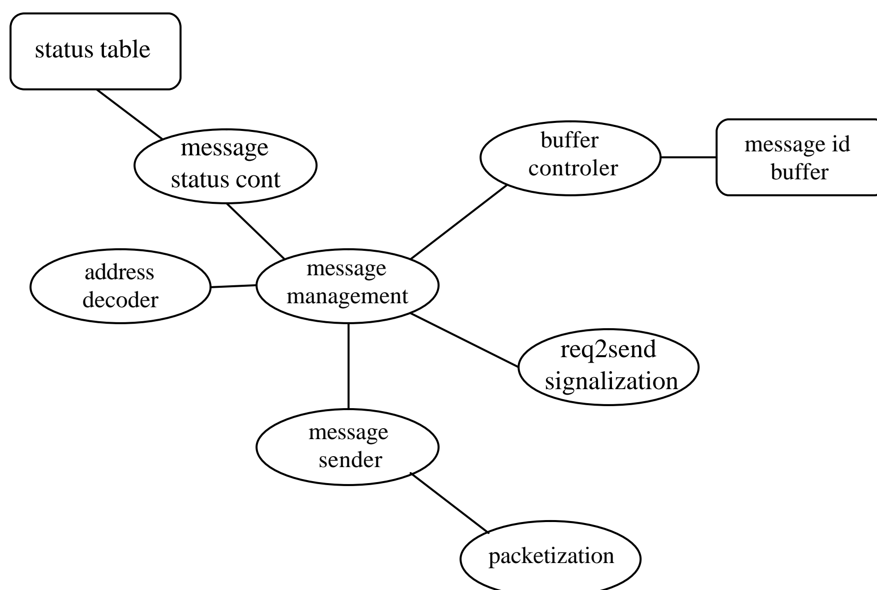

Figure 41 : Exemple de représentation à base d'UF et d'USD

Comparaison avec le modèle de Gauthier :

Le modèle de représentation est plus tourné vers de l'utilisateur. Il doit permettre au concepteur d'applications de comprendre ce qu'il spécifie. La spécification étant explicite les relations entre les unités n'ont pas besoin d'être spécifiées. Il n'y a donc pas de représentation des services fournis et requis. Ce modèle est un outil de description interprétable par les outils de génération automatique. Le niveau d'abstraction n'est pas non plus le même. Le modèle de représentation à base d'UF et d'USD ne montre pas de détails d'implémentation propres au logiciel ou au matériel. Ainsi, une UF pourra correspondre, après interprétation par ASOG, à plusieurs éléments et services. La Figure 42 illustre cette différence. Si l'unité fonctionnelle « message management » est destinée à être implémentée en logicielle, l'outil de génération automatique d'interface logicielle sélectionnera les éléments appelés « message management », « message sender driver » et « HAL_IO » pour l'implémentation.
Les deux types de modèles sont complémentaires au sein du flot puisqu'ils ne sont pas utilisés pour atteindre les mêmes objectifs.

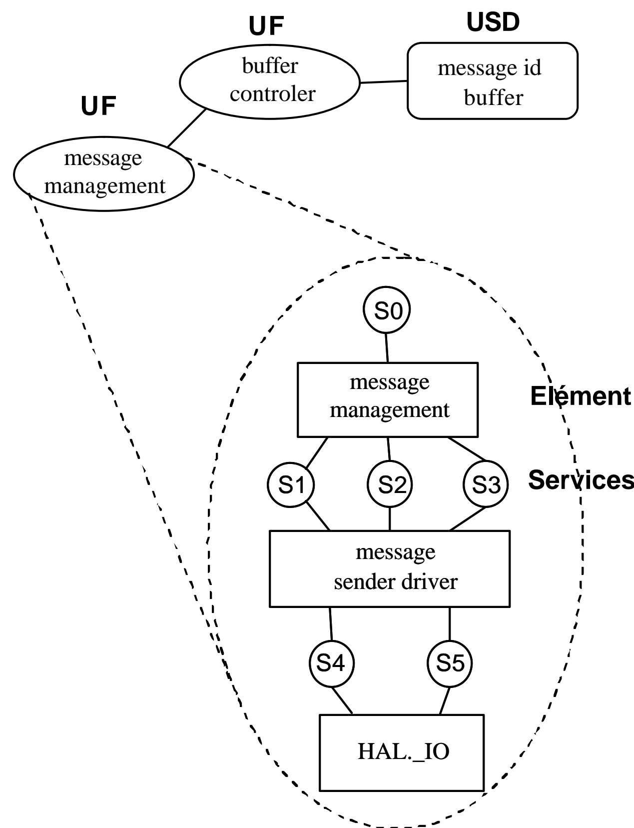

Figure 42 : Correspondance entre le modèle de Lovic Gauthier et celui proposé

c Les critères de choix de granularité des unités

La représentation des services de communication par des unités fonctionnelles et des unités de stockage de données est un outil de spécification. Elle doit permettre l'exploration du partitionnement logiciel/matériel du comportement associé au service de communication. Les unités étant les briques de base que l'on cherchera à allouer à l'architecture logicielle ou matérielle, leur granularité influe directement sur la taille de l'espace des

solutions d'implémentations. Plus la granularité est fine, plus les possibilités de partitionnement seront nombreuses. On serait donc tenté d'adopter la granularité la plus fine possible. Seulement, pour chaque partitionnement, il faut que les outils de génération puissent générer l'implémentation correspondante. Les capacités d'implémentation des outils seront donc un premier frein à la finesse de granularité des unités. L'autre frein est la celui du bon sens. Certains ensembles de fonctionnalités ne peuvent être scindées en plusieurs unités, notamment si leur scission implique trop de synchronisations et de communications entre la partie implémentée en logiciel et la partie implémentée en matériel. Enfin certaines fonctionnalités méritent d'être isolées si on espère pouvoir les extraire pour obtenir un comportement différent ou la réutiliser pour modéliser un autre service de communication.

Aux vues de ces critères, on imagine facilement que la granularité ne sera pas homogène. On peut aussi comprendre que le découpage ne sera pas trivial puisqu'il nécessitera de connaître les implémentations possibles de chaque unité.

4.3 Génération automatique

4.3.1 Rappels sur la méthode actuelle de génération automatique d'interface logicielle/matérielle dans le flot ROSES

Ce paragraphe explique comment les services de communication sont actuellement spécifiés pour les outils ASOG et ASAG. Il rappelle aussi comment est effectué la sélection des composants pour la génération de la partie logicielle et de la partie matérielle des interfaces.

a Spécification

Comme indiqué dans le chapitre 2, la spécification du système est faite en VADeL à base de modules, ports et nets. Cette description est ensuite traduite en COLIF, le format intermédiaire servant d'entrée aux outils de génération. Pour être plus précis sur ce qui nous intéresse, la spécification des services de communication, celle-ci se fait en utilisant 3 ports. Le port de la tâche qui utilise le service, un port interne du module représentant le processeur ainsi qu'un port externe.

Cet ensemble de ports contient les paramètres nécessaires à la spécification d'un service de communication et de son implémentation. Comme le montre la Figure 43 le paramètre « SoftService », annoté au niveau du port de tâche, permet d'indiquer le nom du service de communication demandé. Sur le port interne du module, le paramètre « SoftPortType » de donner une indication sur le type d'implémentation utilisée. Dans cet exemple, ce paramètre prend la valeur « GuardedRegister ». Ceci indique que le tampon utilisé pour stocker les données des messages est gardé, c'est-à-dire que l'écriture dans le tampon sera bloquée lorsque celui-ci sera plein. Le port interne contient aussi des paramètres utilisés lors du raffinement (adresses de données, numéro d'interruption etc.). Sur le port externe on spécifie le protocole de communication utilisé pour communiquer avec le média de communication (réseau, bus ou connection point à point) avec le paramètre «HardPortType ». Dans cet exemple, la valeur « HNDSHK » indique une synchronisation par poignée de main sur une connexion point à point. Le port externe contient aussi des paramètres nécessaires au raffinement de l'interface.

Le Figure 44 montre quels paramètres sont utilisés par chaque outil. ASOG utilise le paramètre « SoftService » pour connaître le service utilisé, « SoftPortType » pour connaître une partie de l'implémentation

demandée et des paramètres de raffinement. ASAG utilise aussi le paramètre « SoftPortType » et des paramètres de raffinement.

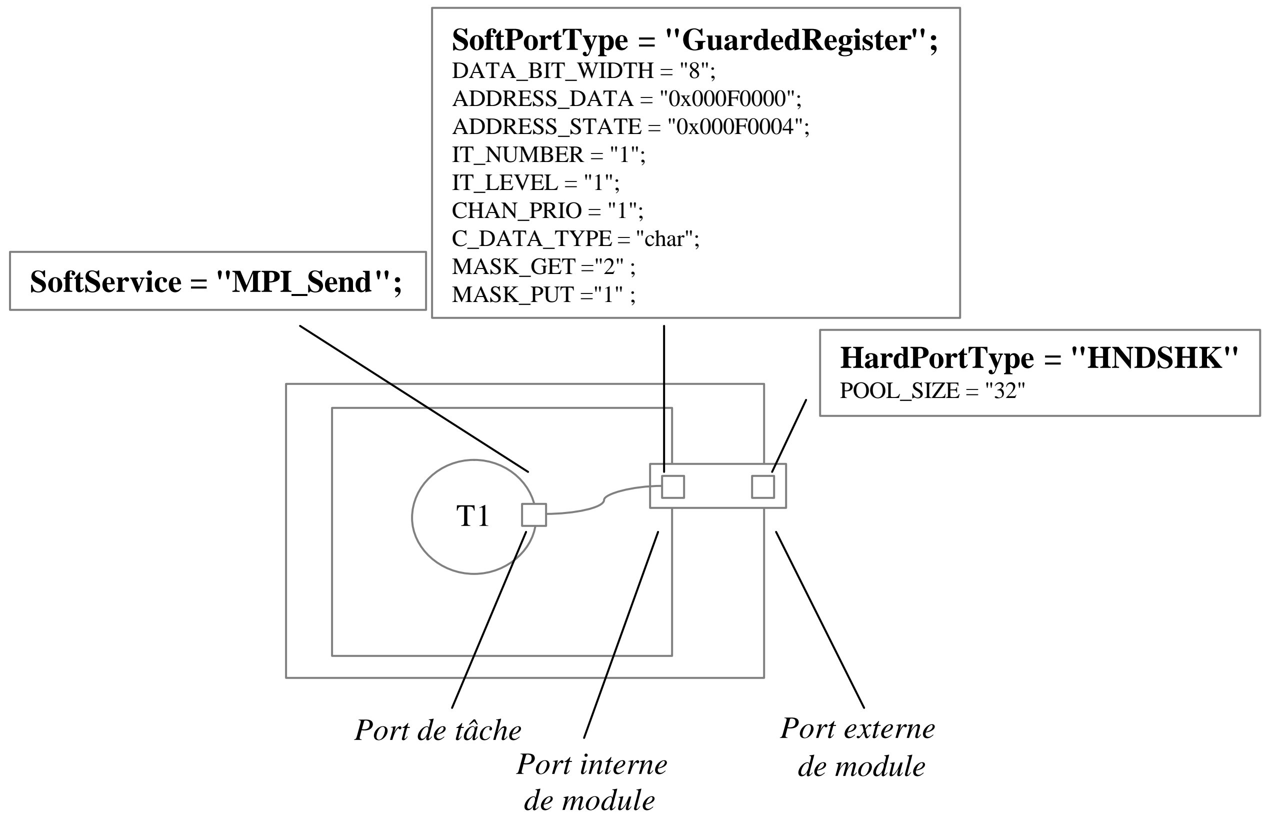

Figure 43 : Spécification d'une implémentation de service de communication dans le flot ROSES

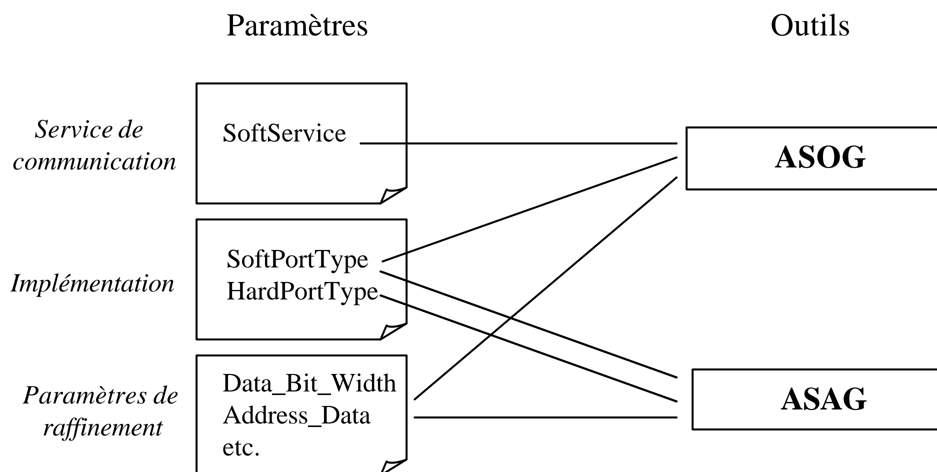

Figure 44 : Les types de paramètre et leur utilisation par ASOG et ASAG

b Sélection de composants

Partie logicielle

La sélection des composants se fait en deux étapes. La première étape se fait lors de la description des dépendances entre éléments et services de la bibliothèque. Ensuite le module d'ASOG chargé de sélectionner les composants crée un arbre d'éléments et de services à partir du service demandé par le paramètre « SoftService ». A ce stade là, certains services peuvent être fournis par plusieurs éléments. Le choix entre ces différents éléments est effectué selon le type de communication demandé (communication entre tâche/communication avec l'extérieur) mais aussi selon le paramètre « SoftPortType ». Ce dernier paramètre contraint la sélection d'éléments.

La Figure 45 montre l'exemple du service « Put » pouvant être fourni par « GuardedRegister » ou « Register ». Le paramètre « SoftPortType » permet de choisir directement l'élément « GuardedRegister ».

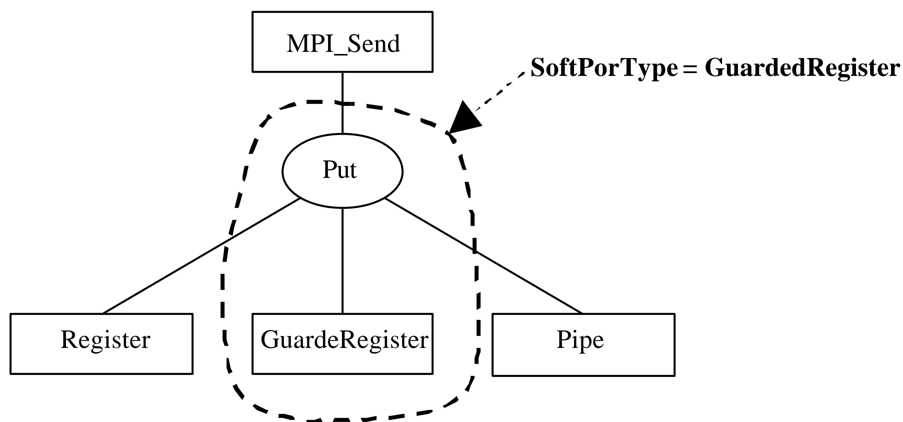

Figure 45 : Influence du paramètre SoftPortType sur la sélection de composants dans ASOG

Partie matérielle

ASAG ne manipule pas d'éléments ni de services pour décrire les dépendances entre composants. Les 3 types de composants sélectionnés sont l'adaptateur de module, le bus interne et les adaptateurs de canaux. L'adaptateur de module est directement sélectionné selon le nom du processeur choisi. Le bus interne est quant à lui toujours le même. Enfin les adaptateurs de canaux sont sélectionnés en fonction d'une combinaison de deux paramètres (voir Figure 44). Le « SoftPortType » indique le type de périphérique auquel a accès le logiciel tandis que le « HardPortType » indique le protocole de communication entre l'adaptateur de canaux et le réseau de communication.

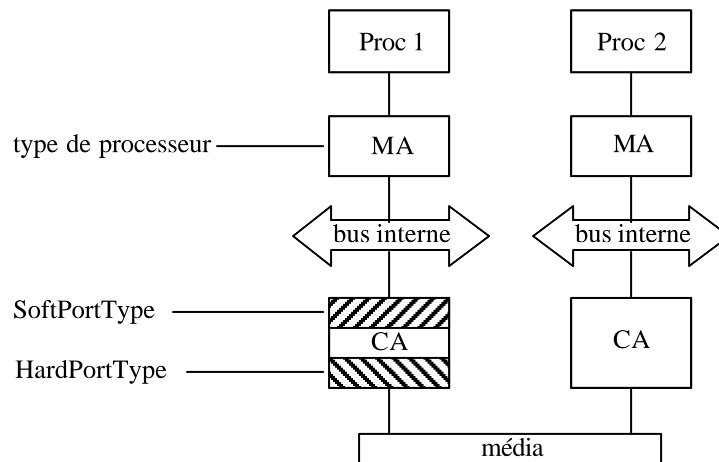

Figure 46 : Influence des paramètres dans la sélection des composants dans ASAG

4.3.2 Utilisation du modèle de représentation des services de communication pour la génération des interfaces logicielles/matérielles

Le but est d'intégrer le modèle de représentation de services de communication (présenté au chapitre 4) dans le flot de génération d'interfaces logicielles/matérielles. Cela passe par une modification de la spécification actuelle et une modification des mécanismes de sélection des composants. La première sous-section présente une proposition de spécification VADeL du service de communication. Ensuite, la sous-section suivante présente la technique de génération de la partie logicielle et enfin la dernière présente les problèmes spécifiques à la génération de la partie matérielle.

4.3.3 Intégration dans la spécification VADeL des descriptions de modèles de service de communication

Comme le montre la Figure 47, on conserve le paramètre « SoftService » indiquant le nom du service de communication spécifié. Le port interne est désormais dédié à la spécification de la partie logicielle du service de communication. Il contient un nom de fichier dans le quel est décrit une partie de modèle de représentation du comportement du service de communication. La description est faite en COLIF. Chaque UF ou USD est représentée par un module. Les modules sont qualifiés par leur type d'unité (UF ou USD) et leur nom. Chaque module peut recevoir des paramètres nécessaires au raffinement. Dans un premier temps, les modules ne sont pas liés. Aucune notion d'interdépendance n'est spécifiée à ce niveau. Le port externe contient le nom du fichier spécifiant la partie matérielle du comportement du service de communication. La description est effectuée de la même façon que pour la partie logicielle.

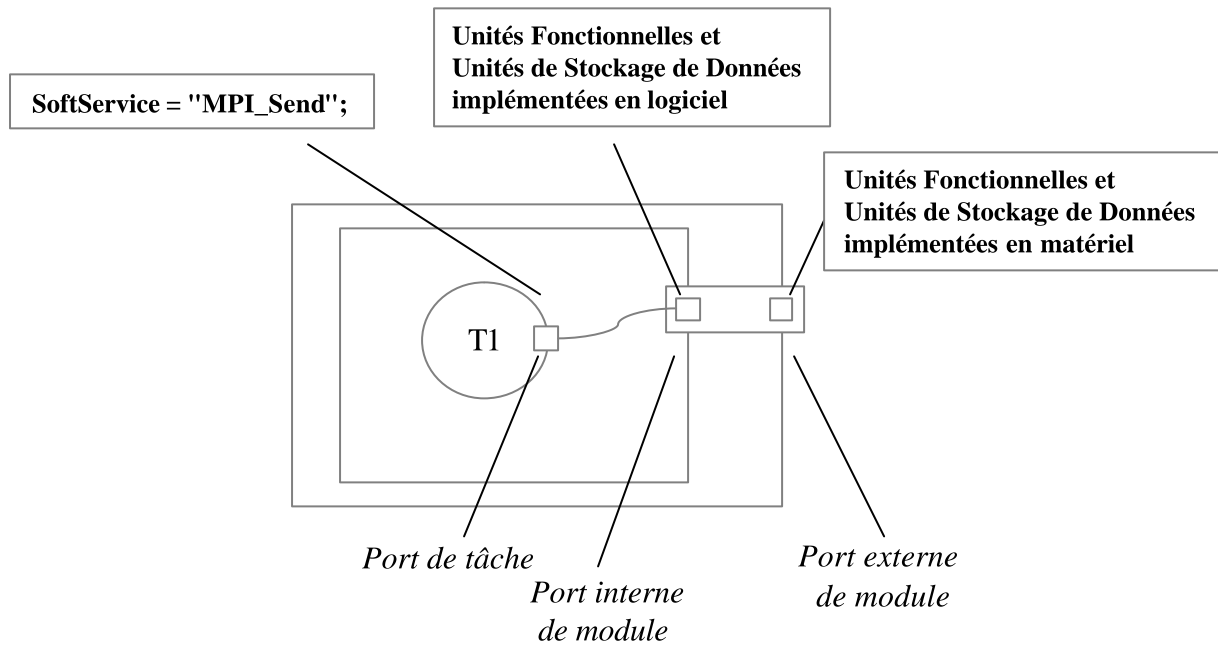

Figure 47 : Nouvelle spécification d'un service de communication1

4.3.4 Génération de la partie logicielle

a Méthode de sélection des composants nécessaires à l'implémentation

L'idée est de conserver la même méthode de sélection d'un arbre d'éléments et de services à partir du service de communication spécifié avec le paramètre « SoftService ». Ce paramètre permet toujours de sélectionner le nom du service de communication. Ensuite, chaque Unité Fonctionnelle ou Unité de Stockage de Données joue le même rôle en désignant un élément comme on le faisait auparavant avec le paramètre « SoftPortType ». La Figure 48, explique le mécanisme de sélection de composants adopté. A gauche, on part d'une représentation de service de communication à base d'unités fonctionnelles et d'unités de stockage de données. Au centre, le dessin représente une description des dépendances entre éléments et services d'une bibliothèque d'ASOG. Le paramètre « SoftService » permet de sélectionner le service s1. Ensuite, chaque UF ou USD pointe sur un élément nécessaire à son implémentation en logiciel. Ainsi l'élément E1 est sélectionné directement par UF1, E2 par UF2, E4 par UF3 et enfin E4 est directement sélectionné par USD1. La deuxième étape de sélection est guidée par les dépendances décrites en Lidel dans la bibliothèque d'ASOG : E1 ayant besoin du service S3, l'élément E3 est sélectionné par ASOG. De même pour E9 qui est lié par E6 qui a besoin de s7. Finalement, les éléments sélectionnés pour l'implémentation du service de communication spécifié sont E1, E2, E3, E4, E6 et E9.

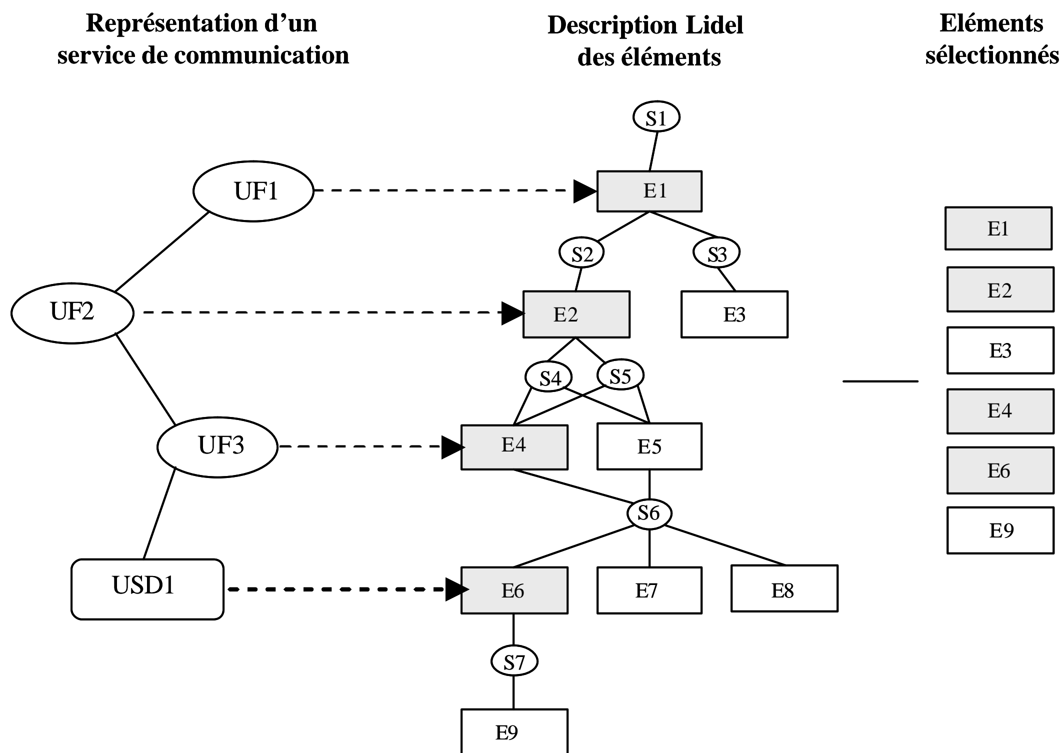

Figure 48 : Sélection de composants à partir d'un modèle de représentation de service de communication

b Extension d'ASOG permettant la lecture du modèle d'interface de communication.

L'extension consiste simplement à ajouter un module permettant la lecture des fichiers COLIF contenant les descriptions de modèles de service de communication. Il parcourt chaque module et crée un tableau de noms d'unités fonctionnelles. Les paramètres sont aussi récoltés pour chaque module et stockés pour être traités par ASOG.

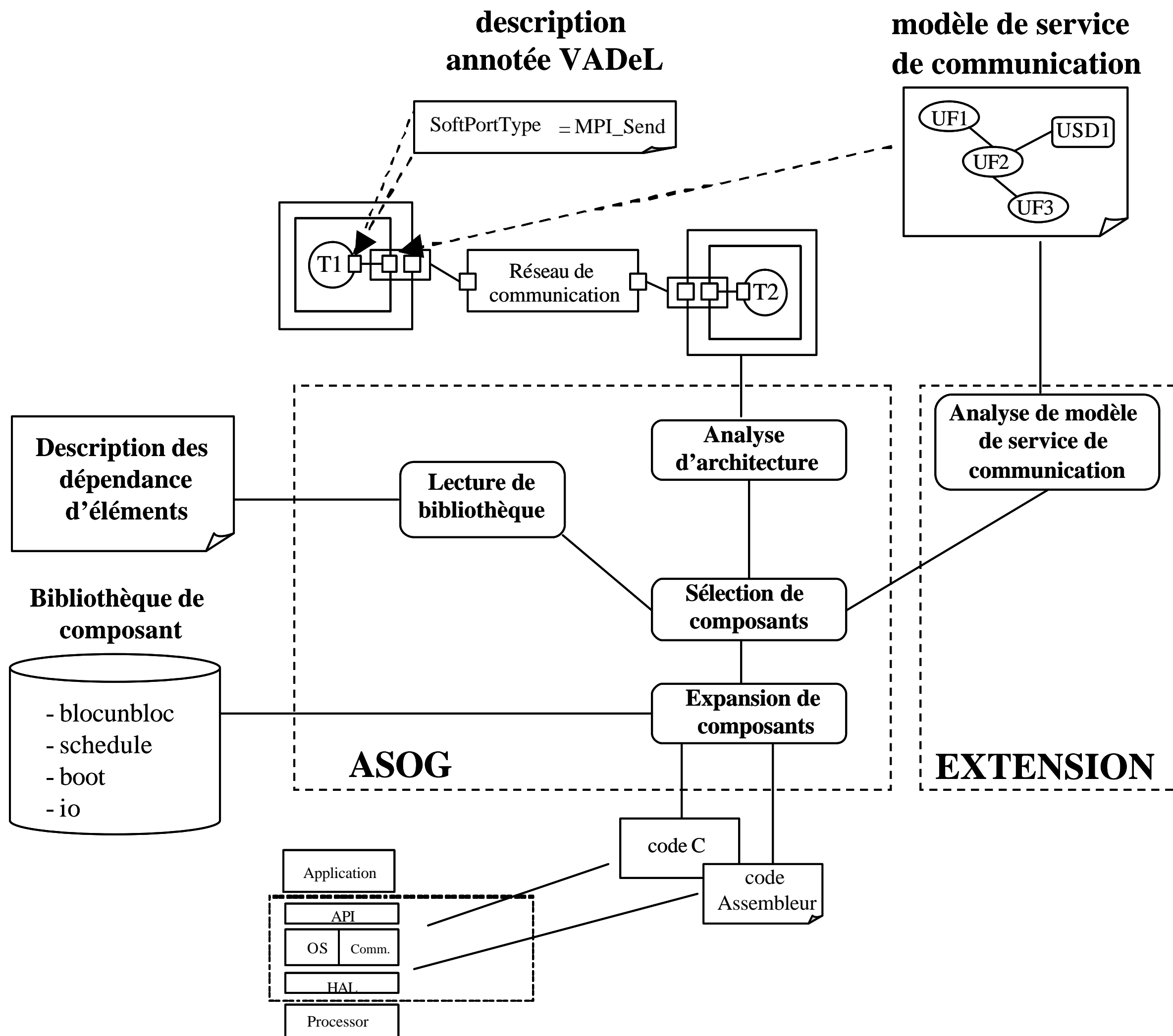

Figure 49 : Flot étendu intégrant le modèle de représentation des services de communication

4.3.5 Génération de la partie matérielle

a Problèmes et difficultés propres à la génération de la partie matérielle de l'implémentation

Le gros problème avec la génération de la partie matérielle est dû à l'influence de l'architecture sur les caractéristiques du circuit. Un même ensemble de fonctionnalités peut être implémenté de multiples façons. Par exemple, un simple FIFO peut être simplement implémentée par des registres à décalage ou par un DMA avec une mémoire et une partie contrôle. Selon l'architecture choisie, la surface, la consommation ainsi que les performances (temps de latence, bande passante) varient significativement.

Le second problème est lié à la méthode d'assemblage. La granularité des composants assemblés influence l'architecture des interfaces générées et donc les caractéristiques des réalisations. Pour mieux illustrer les problèmes de relations entre architecture et granularité, prenons un exemple. Si on considère la granularité actuellement utilisée dans ASAG, l'unité de base est un Adaptateur de canaux. Cet adaptateur peut être optimisé

pour correspondre aux attentes de performances et de consommation. L'architecture interne du AC est libre de toute contrainte. Il pourrait par exemple être implémenté sous forme d'un flot de données, d'une FSM de contrôle et d'un DMA. Par contre si la granularité utilisée avait été dix fois plus petite, comment aurions nous fait pour obtenir le même résultat ? La FSM aurait été découpée en plusieurs FSM synchronisées ? Si on utilise une méthode d'assemblage simple se contentant de connecter des modules, chaque connexion entre module contraint l'architecture en obligeant a utiliser des interfaces de communication entre modules.

b Desiderata d'un flot de génération automatique de la partie matérielle de l'implémentation

En tous cas, le concepteur doit pouvoir diriger le choix de l'architecture utilisée pour l'implémentation de la partie matérielle de l'interface. Si on revient sur la méthode utilisée par ASAG, on note l'utilisation d'une architecture générique constituée d'un module d'adaptation du processeur, d'un bus interne et d'un ensemble d'adaptateurs de canaux implémentant les services de communication. La seule partie changée lors d'une exploration sera le nombre d'adaptateurs de canaux et leur composition. Ce type d'architecture ne pourra pas convenir à toutes les implémentations : l'architecture générique choisie est trop rigide et contraignante. Dans le cahier des charges d'un flot de génération de la partie matérielle des interfaces, il faudrait que le concepteur puisse choisir entre plusieurs architectures génériques. Dans la sous-section précédente, on évoquait le problème de granularité pouvant contraindre l'architecture. Pour répondre à ce problème il faudrait, pour chaque architecture générique, déterminer les parties découpables et la granularité utilisable. Le plus difficile sera de trouver les critères guidant ce découpage.

4.4 Conclusions

Ce chapitre a permis de présenter la problématique du partitionnement logiciel/matériel des services de communication ainsi qu'un flot pour l'exploration du partitionnement logiciel/matériel des services de communication. L'exploration du partitionnement ne peut être réalisée que par l'utilisation d'un flot itératif. Pour résumer, ce flot est constitué d'une partie générant une proposition d'implémentation et d'une partie permettant un retour d'informations sur l'implémentation générée. Tant que les contraintes spécifiées pour la conception de l'application ne sont pas atteintes, le flot repart dans une boucle d'itération. L'utilisation d'un tel flot ne peut être possible que si les interfaces logicielles/matérielles sont générées automatiquement.

Un modèle de représentation des services de communication a été présenté. Ce modèle a été développé dans l'objectif de servir de modèle d'entrée aux outils du flot ROSES pour permettre l'exploration du partitionnement logiciel/matériel des interfaces pour l'implémentation des services de communication.

La contribution principale est la proposition d'un flot de génération automatique de la partie logicielle des interfaces en partant de l'outil ASOG développé par Lovic Gauthier. Une extension de l'outil a été proposée permettant l'utilisation du modèle de représentation des services de communication. Les algorithmes de sélection de composants utilisés par ASOG sont le moteur du flot de génération automatique. La contribution consiste juste à changer le moyen de spécification. Au lieu de spécifier un service de communication et son implémentation par deux paramètres, on spécifie les unités fonctionnelles et les unités de stockage de données composant le service de communication. Cette solution est plus adaptée à l'exploration du partitionnement des services de communication mais elle permet aussi de spécifier des services de communication plus complexes que ceux auparavant utilisés avec ASOG.

Les problèmes spécifiques à la génération de la partie matérielle des interfaces ont été étudiés et présentés. La composante architecturale de la partie matérielle influe bien plus sur les performances que pour la partie logicielle. Il faut laisser au concepteur la capacité de diriger le choix de l'architecture de la partie matérielle.

Chapitre 5 Expérimentation : réalisations logicielles/matérielles et représentation de services de communication pour la communication MPI

Ce chapitre présente les deux expériences qui ont permis d'élaborer les solutions proposées dans cette thèse. La première expérience est une implémentation 100% logicielle de primitives MPI sur une plateforme ARM. Elle a permis la découverte de l'idée d'un modèle de représentation des services de communication à base d'unités fonctionnelles et d'unités de stockage de données. Elle a aussi permis de montrer les limitations en terme de performance d'une implémentation 100% logicielle. La seconde expérience est l'implémentation mixte d'un sous ensemble de primitives MPI sélectionné au chapitre 3. Elle a permis de tester le modèle de représentation des services de communication présenté au chapitre 4, mais aussi de concevoir le flot de génération automatique de la partie logicielle.

5.1 Implémentation 100% logicielle sur une plateforme ARM

5.1.1 Objectifs de l'expérience

L'objectif initial était de développer plusieurs implémentations mixtes de primitives de communication MPI. Le but était de pouvoir comparer les variations de performance selon le partitionnement choisi. Le deuxième objectif était d'utiliser les outils de génération automatique ASOG et ASAG. A ces objectifs est venu s'ajouter la participation à un projet de plateforme de prototypage flexible permettant l'utilisation de plusieurs types de réseaux pour la communication entre plusieurs processeurs. Ce projet, le projet Archiflex, a été développé en collaboration avec ST Microelectronics et le LETI.

L'équipe SLS devait fournir une implémentation de primitives MPI pour permettre la communication entre processeurs. Cette expérience correspond donc à l'implémentation de primitives MPI en logiciel sur une plateforme ARM et à l'intégration de composants dans la bibliothèque d'ASOG pour la génération automatique.

5.1.2 Caractéristiques de l'implémentation

a Les primitives implémentées

Les primitives choisies sont MPI_SEND, MPI_RECV et MPI_INIT. Les deux premières primitives sont les primitives de base de la communication MPI. Elles permettent en effet d'obtenir la synchronisation des tâches par leur qualité de communication bloquantes, mais aussi de limiter les blocages de communication grâce à l'utilisation de tampons de communication. La primitive MPI_INIT a été développée pour permettre l'initialisation des ressources de communication.

b Architecture cible

Description de la plateforme

La Figure 50 présente l'architecture globale de la plateforme ARM. Elle est composée de deux processeurs ARM7 et de deux processeurs ARM9. Le média de communication utilisé pour connecter les processeurs est un bus AMBA AHB. Chaque processeur est connecté au bus AMBA par un FPGA. Chaque FPGA contient une interface permettant l'utilisation du bus AMBA, mais aussi un contrôleur mémoire arbitrant l'accès à la mémoire locale du processeur. Grâce à ce contrôleur, il est possible d'accéder à chaque mémoire locale depuis le processeur mais aussi depuis le bus AMBA. Ainsi chaque processeur peut accéder aux mémoires locales des autres processeurs. Ces interfaces permettent aussi d'envoyer un signal d'interruption à tous les processeurs. Des registres permettent de masquer les processeurs auxquels on ne veut pas envoyer d'interruptions. La plateforme possède aussi un FPGA libre connecté au bus AMBA. Il n'a pas été utilisé dans cette expérience.

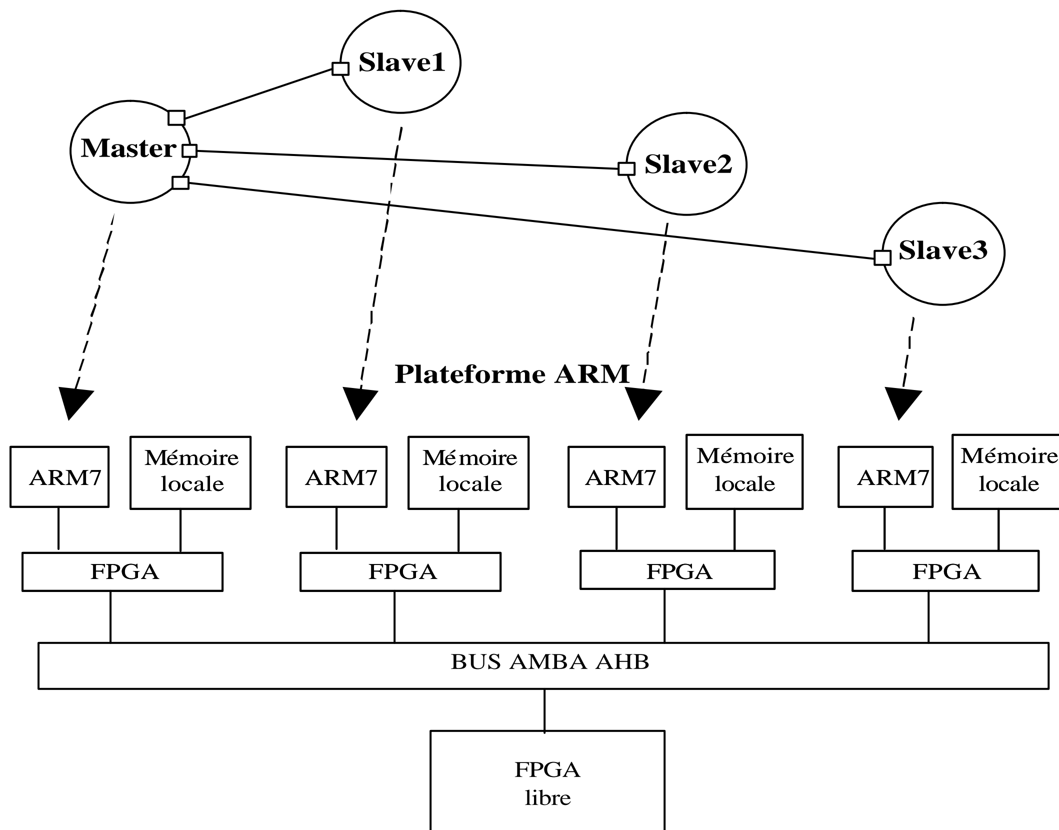

Figure 50 : Plateforme ARM

Les contraintes imposées par la plateforme

Le seul FPGA disponible n'étant pas placé entre le média de communication (le bus AMBA), et les processeurs, il n'était pas possible de l'utiliser pour implémenter la partie matérielle de l'interface. Nous n'avions donc pas la possibilité de faire une implémentation mixte logicielle/matérielle des primitives de communication selon l'architecture présentée au chapitre 3. Les implémentations ont donc été faites entièrement en logiciel en exploitant le matériel déjà présent dans la plateforme. Les fonctionnalités de la plateforme utilisées sont la possibilité pour un processeur d'écrire dans la mémoire d'un autre processeur et la possibilité pour un processeur de générer une interruption matérielle chez d'autres processeurs.

Les choix d'implémentation

Allocation mémoire

Pour chaque couple Sender/Receiver :

- 1 FIFO pour le TAG
- 1 FIFO pour les données
- 2 registres indiquant le nombre d'éléments contenus dans la FIFO
- 2 registres implémentant des sémaphores

- 2 registres pour l'identificateur d'interruption
- 2 registres contenant l'adresse de Tête
- 2 registres contenant l'adresse de Queue

La Figure 51 montre le choix d'allocation des ressources de communication. Les FIFO et les registres servant à les contrôler sont implémentés dans la mémoire locale du processeur qui envoie les messages.

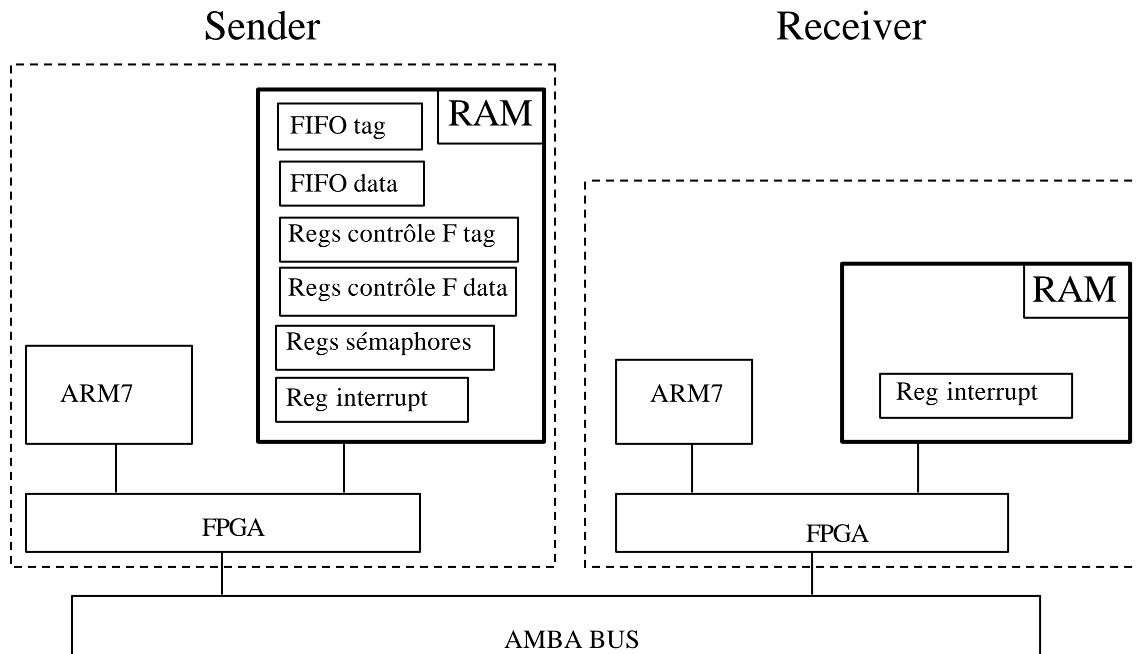

Figure 51 : allocations des ressources

Echange de données

La Figure 52 présente le processus d'envoi d'un message. Un message correspond à un ensemble d'éléments de type « datatype ». Si le type de données utilisé est par exemple MPI_long_long, un élément fait 64 bits. Le message complet fait donc count*64 bits. Ensuite ces éléments sont lus dans le tampon source et découpés en « cells » de 32 bits pour être stockées dans le tampon de communication. Enfin, lorsque le « receiver » est prêt à recevoir le message, les « cells » sont envoyées en mode burst via le bus AMBA. Chaque burst correspond à l'envoi d'un ensemble de « cells » appelé paquet.

Les autres types d'échanges sont des accès mémoire effectués par le processeur « sender » et le processeur « receiver » pour contrôler les FIFOs, pour écrire dans les registres permettant l'identification des interruptions ou encore pour accéder aux sémaphores.

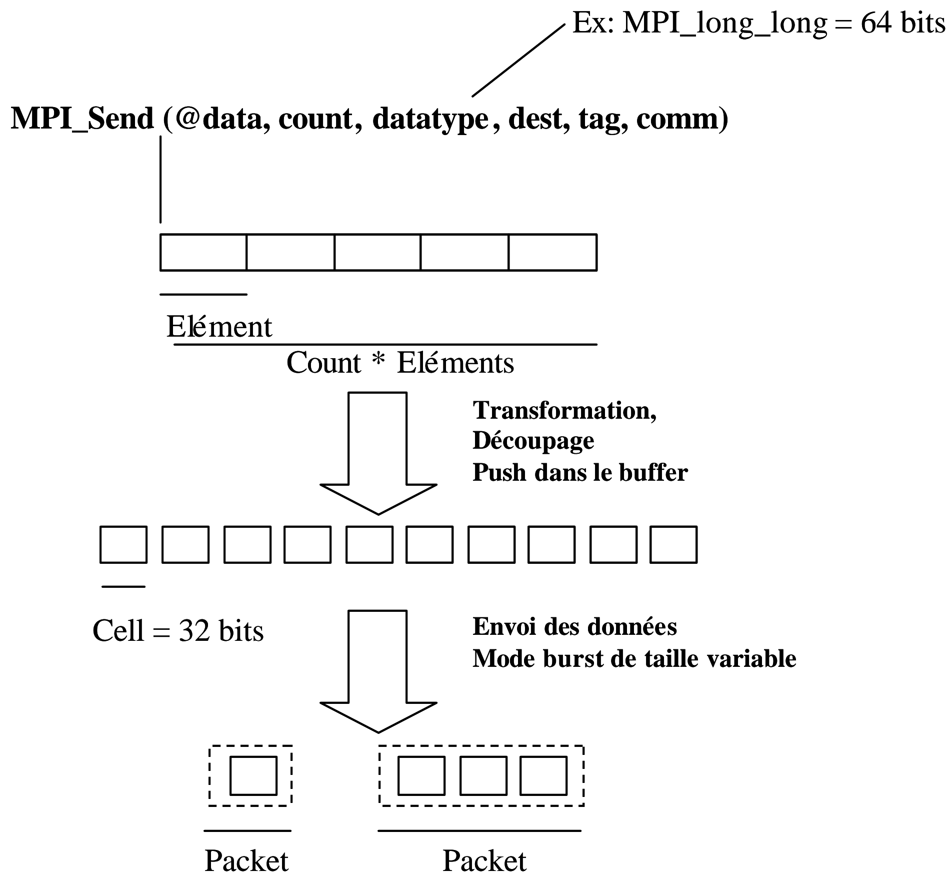

Figure 52 : Décomposition, transformation et transfert des données

5.1.3 Modèles de représentation des primitives MPI_SEND et MPI_RECV

Les modèles de représentation des services de communication présentés Figure 53 et Figure 54 ont été définis après l'implémentation des primitives. Les unités fonctionnelles « bus access » et « Memory access » font partie des fonctionnalités offertes par la plateforme. La première permet l'accès au bus et l'envoi de bursts de données. La seconde permet à un processeur d'accéder à la mémoire d'un autre processeur. Les autres unités sont issues de l'implémentation logicielle. « MPI_Send API » fournie l'API de communication aux tâches. « MPI_Send » et « MPI_Recv » rassemblent le plus gros des services de communication. « cells operation » et « packets operation » permettent respectivement l'envoi de « cells » et l'assemblage de paquets. Enfin pour les unités de stockage de données, l'une permet le stockage des étiquettes « tag » de messages et l'autre les données.

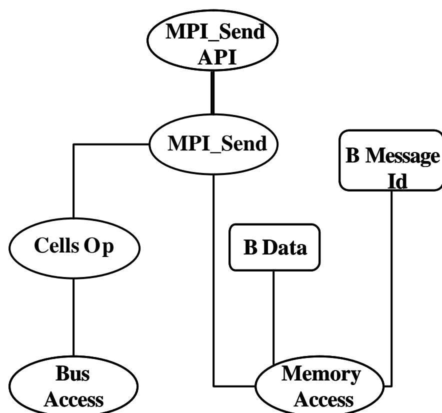

Figure 53 : Représentation de MPI_SEND

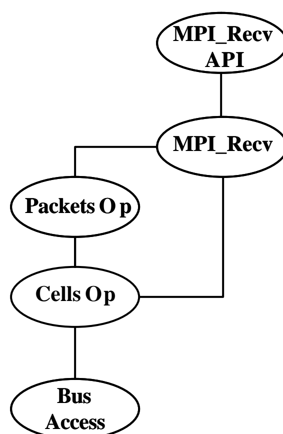

Figure 54 : Représentation de MPI_RECV

5.1.4 Architecture de la partie logicielle

Le Figure 55 représente l'architecture de l'implémentation de MPISEND et MPI_RECV. Elle représente deux processeurs. MPI_SEND est implémentée sur le processeur 1 et MPI_RECV est implémentée sur le processeur 2. Les ovales représentent des unités fonctionnelles et les rectangles aux bords arrondis des unités de stockage de données. L'intérêt de cette figure est de montrer l'allocation des unités dans l'architecture logicielle est les relations entre les parties OS, HAL et communication. La partie montrant le processeur 1 est détaillée complètement alors que la partie du processeur 2, pour limiter la complexité du dessin, on ne présente pas l'OS est les piles utilisées. Si on regarde plus en détails la partie représentant le processeur 1, on s'aperçoit que l'OS utilise un ensemble de trois piles par tâches. L'une est dédiée aux sauvegardes de registres dans le mode « user », l'autre aux sauvegardes en mode « system » et la troisième pour les sauvegardes en mode « IRQ ».

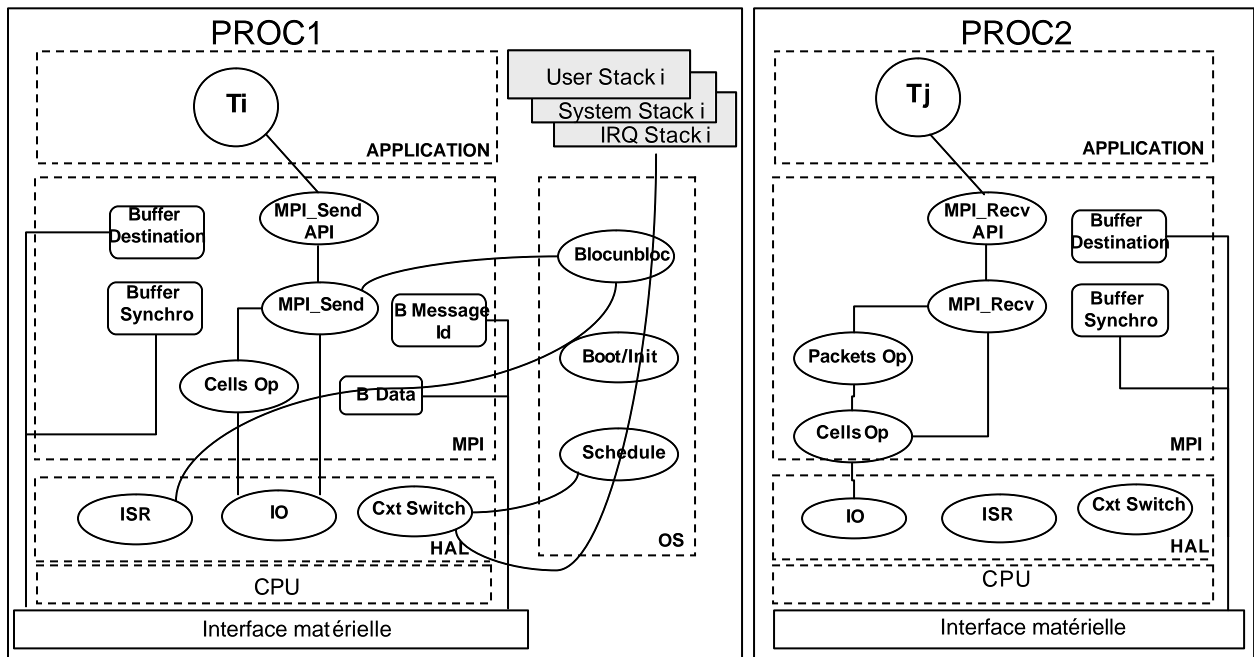

Figure 55 : Architecture de l'implémentation 100% logicielle de MPI_SEND/MPI_RECV

5.1.5 Spécification et composants de bibliothèque pour la génération automatique

a La spécification

La spécification présentée est celle de l'application utilisée dans le projet Archiflex. Cette application est une implémentation d'un encodeur OpenDivX sur quatre processeurs. L'encodeur OpenDivX est une application qui a pour but de coder un fichier vidéo en produisant un fichier compressé. L'objectif étant de réduire la taille du fichier vidéo initial.

Le principe de compression utilisé par l'encodeur est basé sur le fait que dans une séquence vidéo, il existe deux sortes de redondances : la redondance spatiale et la redondance temporelle. Sans entrer dans les détails des algorithmes de compression utilisés, on peut dire que certaines images (images I) sont codées entièrement avec des codages classiques alors que d'autres (images P) sont calculées par estimation de mouvement. Le code de l'encodeur a été découpé en quatre modules : un maître et trois esclaves. Le maître lit des images à partir du fichier à encoder. S'il s'agit d'une image I, il effectue le codage classique. Il envoie à chaque esclave un tiers de l'image lorsque celle-ci est une image P. Chaque esclave calcule alors les vecteurs de mouvement ainsi que la compensation de mouvement d'un tiers de l'image. Puis il envoie les résultats au maître qui effectue encore dessus des traitements avant de les stocker dans le fichier cible.

Comme le montre la Figure 56, chaque tâche est allouée à un processeur. La tache maître est décrite comme un module VADeL. Elle possède trois ports utilisant les services MPI_SEND et MPI_RECV. Le processeur exécutant cette tâche est modélisé par un module VADeL comportant les trois ports pour la communication MPI.

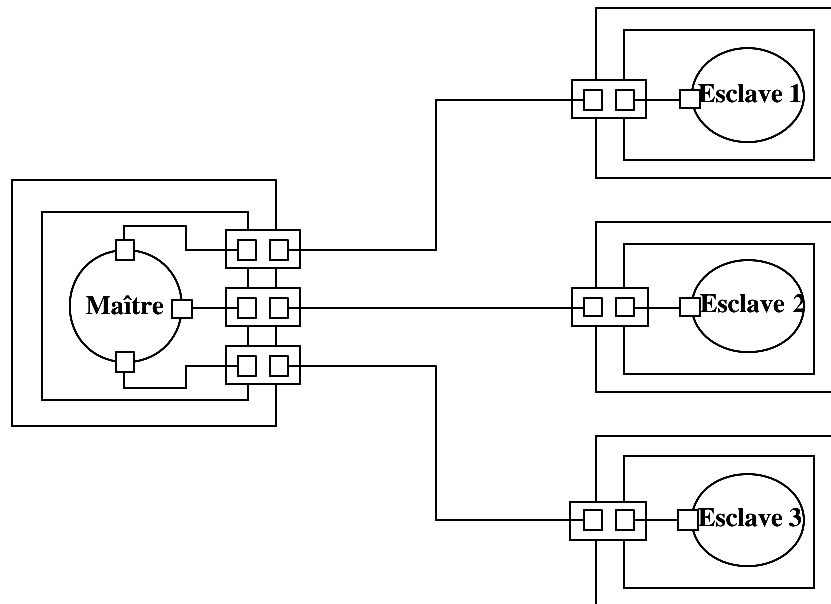

Figure 56: Spécification de l'application OpenDivX

La méthode de spécification reste la même que celle utilisée pour l'application VDSL présentée au chapitre 2. Avec cette expérience seule la partie logicielle est générée. La Figure 57 reprend la spécification de la tâche maître en se focalisant sur les paramètres nécessaires à la génération automatique de l'implémentation logicielle de MPI_SEND et MPI_RECV. Le paramètre SoftService prend les valeurs « MPI_Std_Send » et « MPI_Std_Recv » pour indiquer à ASOG que l'on veut deux services de communication MPI. Ensuite, le paramètre SoftPortType précise une partie du code que l'on pourrait qualifier de «driver ». Les autres paramètres sont des paramètres de raffinement

- R_FIFOTagAddress : adresse de la FIFO stockant les étiquettes (« tag ») des messages.
- R_AddressHeadFIFOTagAddress, R_AddressTailFIFOTagAddress, R_NbElementFIFOTagAddress, R_NbElementMaxFIFOTag : ces adresses servent au contrôle de la FIFO d'étiquettes. La FIFO étant circulaire, on a besoin de connaître les adresses des pointeurs de tête et de queue, l'adresse ou est stockée le nombre d'éléments contenus ainsi que le nombre maximum d'éléments pouvant être contenus.
- R_FIFODataAddress : adresse de la FIFO stockant les données de message.
- R_AddressHeadFIFODataAddress, R_AddressTailFIFODataAddress, R_NbElementFIFODataAddress, R_NbElementMaxFIFOData : adresses servant au contrôle de la FIFO.
- R_FIFOTagAddressSem, R_FIFODataAddressSem : adresses de zones mémoire servant pour l'implémentation d'un sémaphore permettant d'éviter les accès simultanés aux zones mémoires contenant le nombre d'éléments contenus dans les FIFO.
- Tous les paramètres commençant par R_ sont dédiés aux ressources permettant l'implémentation de MPI_RECV. Les mêmes paramètres existent en version S_ pour l'implémentation de MPI_SEND.
- ProcessorId: numéro d'identification du processeur.
- RegExtItNumberAddress : adresse du registre où les numéros d'interruption sont récupérés.

Le code complet de la spécification de l'application OpenDivX est disponible dans la partie annexe du manuscrit.

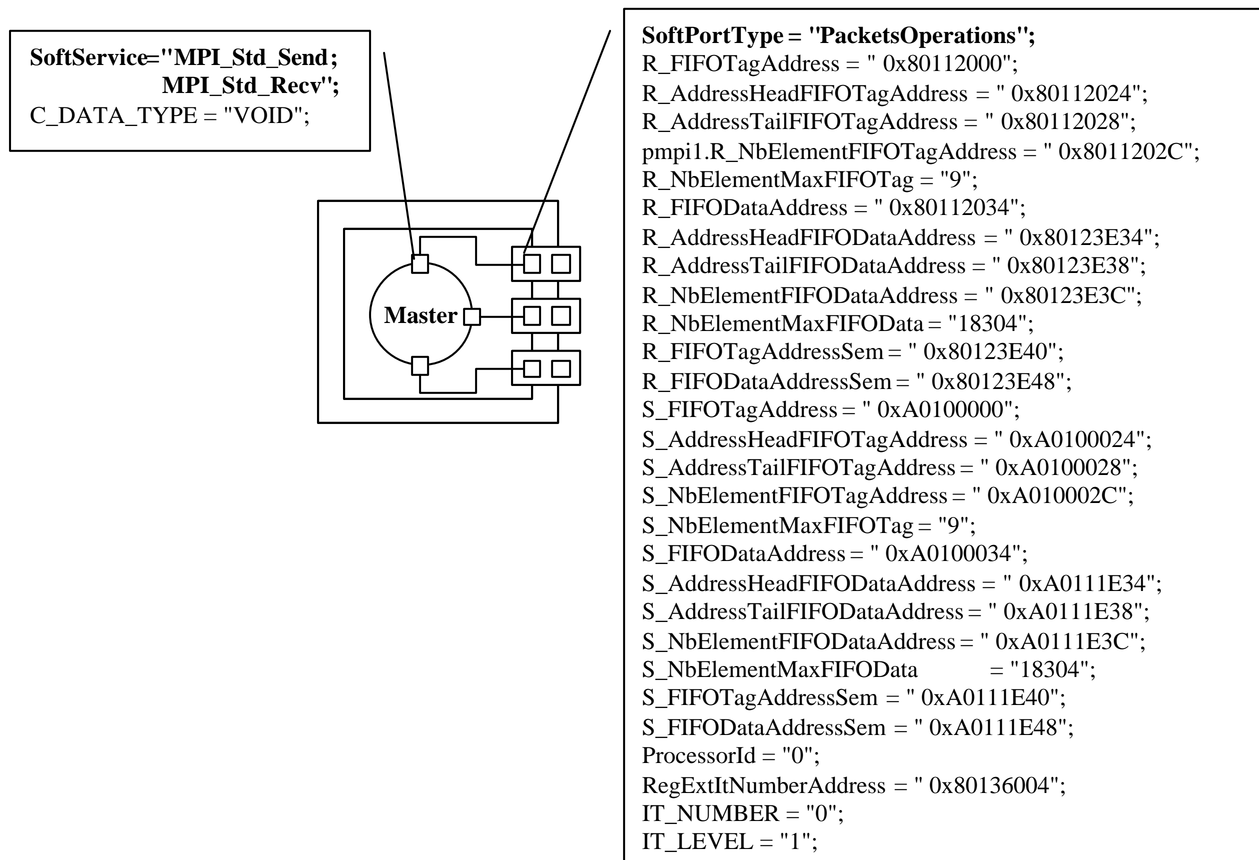

Figure 57 : Spécification des services de communication MPI_SEND et MPI_RECV

b Intégration de composants de bibliothèque dans ASOG

L'intégration de l'implémentation dans ASOG se fait en deux étapes. L'une consiste à décrire les dépendances entre les composants à intégrer dans la bibliothèque. La seconde consiste à écrire le code macro correspondant à chaque élément.

Description des dépendances entre éléments

Les éléments créés sont présentés dans la Figure 58. Chaque élément désigné par un rectangle est accompagné des services qu'il fourni et de ceux dont il a besoin. Les services sont désignés par des ovales. Lorsqu'une flèche part d'un élément et pointe sur un service, elle indique que ce service est requis par l'élément. Lorsque la flèche est en sens inverse, cela signifie que le service et fourni par cet élément. Dans cette figure, seuls les éléments et les services nécessaires à l'implémentation des primitives sont présentés. On ne montre pas les éléments appartenant à l'OS, même si certains ont dû être modifiés (comme le traitement des interruptions).

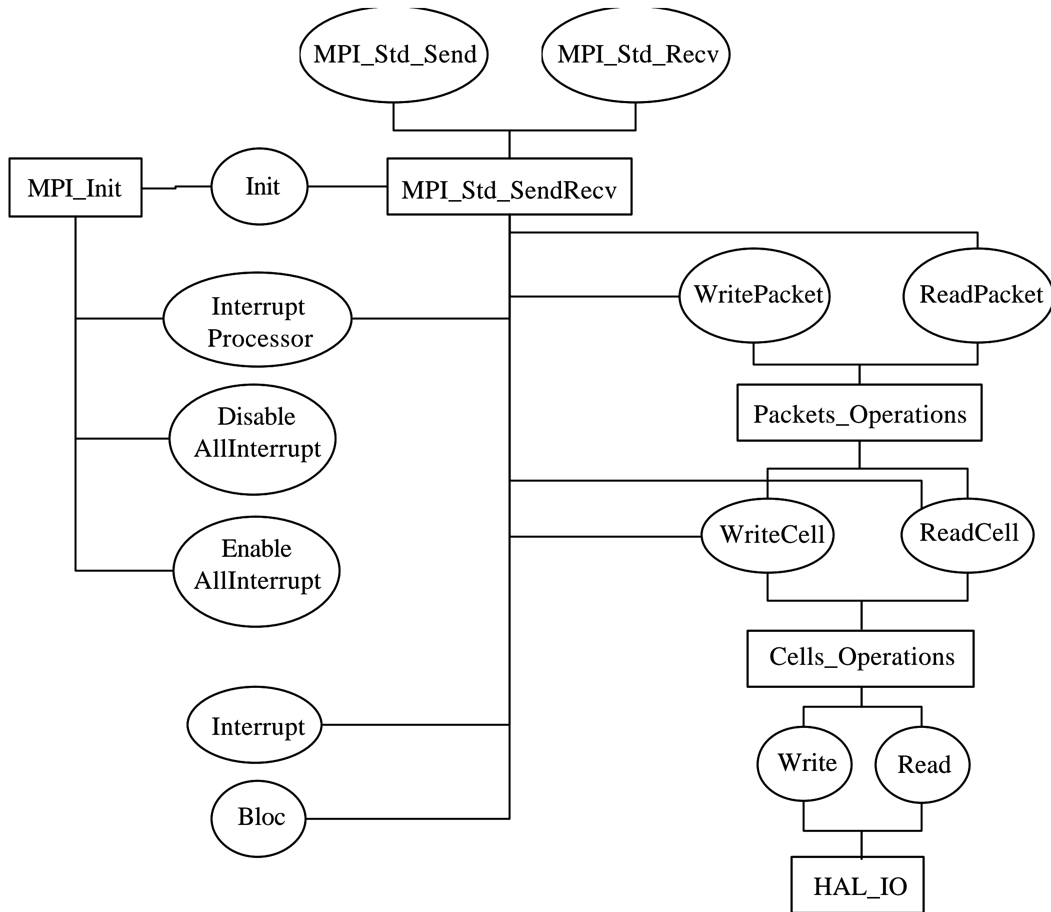

Figure 58 : Relations entre éléments et services utilisés pour la génération de MPI_SEND et MPI_RECV

Cette description est écrite en Lidel comme l'illustre la Figure 59. Elle présente la syntaxe utilisée pour spécifier les services fournis et les services requis par les éléments. La description ne se limite pas à ça. On décrit aussi les dépendances entre les éléments et le code macro, les paramètres de raffinement utilisés par chaque élément et les dépendances avec les architectures cibles. Par exemple dans cette expérience, nous avons dû ajouter une cible supplémentaire spécifique à la plateforme. Pour tout code dépendant d'une architecture, il faut décrire quelle est l'architecture cible

```
MPI_Std_SendRecv.append2provided("HighIO/MPI/ MPI_Std_Send");
MPI_Std_SendRecv.append2provided("HighIO/MPI/ MPI_Std_Recv");
MPI_Std_SendRecv.append2required("Kernel/Bloc");
MPI_Std_SendRecv.append2required("LowIO/WritePacket")
Packets_Operations.append2provided("LowIO/WritePacket");
Packets_Operations.append2required("LowIO/WriteCell");
Cells_Operations.append2provided("LowIO/WriteCell");
Cells_Operations.append2required("HAL/IO/Write");
HAL_IOSource.append2codefiles("HAL/IO/HAL_IO");
```

Figure 59: Extrait de description en Lidel des dépendances entre éléments

Les composants en code macro :

La Figure 60 montre un bout de code macro écrit pour l'intégration de MPI_RECV. On peut noter l'utilisation de guillemets qui permet délimiter texte devant être généré tel quel. Par exemple, « MPI_Recv_channel_type MPI_Recv_channel [» sera généré tel quel dans le code C. On peut aussi noter

l'utilisation de boucles conditionnelles permettant de générer un tableau de taille ajustable au nombre de canaux utilisés.

```
@# MPI_Std_Recv_channel declaration
@DEFINE declare_MPI_Std_Recv_Channel ="
@MPI_Recv_channel_type MPI_Recv_channel["nb_chan"]= {{"IT_NB[0]",0}"
@  IF (nb_ressources >=3) DO
@    FOR i FROM 1 TO (nb_ressources-2) DO",\
@                                        {"IT_NB[i]","i"},\"
@    ENDFOR"
@                 {"IT_NB[nb_chan-1]","nb_chan-1"}};"
@  ENDIF
```

Figure 60 : Extrait de code macro

Le code macro macro utilisé pour générer l'implémentation des services de communication MPI_SEND, MPI_RECV et MPI_INIT fait à peu près 1300 lignes. Le tableau suivant présente les fichie rs correspondants au code macro ajouté dans la bibliothèque d'ASOG. Le code du fichier MPI_Send_Ressources.h.gen est présenté en annexe.

Eléments ajoutés	Fichiers de code macro	Taille du code en lignes
MPI_Std_SendRecv	MPI_Recv_Ressources.h.gen MPI_Send_Ressources.h.gen MPI_Std_Send.c.gen MPI_Std_Send.h.gen MPI_Std_Recv.c.gen MPI_Std_Recv.h.gen MPI_type_definitions.h.gen	1064
MPI_Init	MPI_Init.c.gen MPI_Init.h.gen	146
Packets_Operations	Packets_Operations.c.gen Packets_Operations.h.gen	48
Cells Operations	Celles_Operations.c.gen Celles_Operations.h.gen	38
HAL_IO	HAL_IO.c.gen HAL_IO.h.gen	15

Tableau 9: Fichiers du code macro des composants intégrés dans ASOG

c Résultats

Le code généré

Le tableau suivant montre la taille du code généré pour le processeur exécutant la tâche maître et un processeur implémentant une tâche esclave. Tous les chiffres présentés sont exprimés en octets. Pour chaque

couche de la partie logicielle, trois types d'informations sont donnés. Le premier type donne la taille du code, le deuxième la taille des données en lecture/écriture et le dernier la taille des données en lecture seule. Ce tableau permet d'observer les différences de taille de code observées entre deux tâches n'utilisant pas le même nombre de port MPI_SEND/MPI_RECV. La tâche maître utilise trois ports alors que les tâches esclaves n'en utilisent qu'une. On peut noter que le code de la couche HAL ne varie pas. Ceci est tout à fait normal puisque la cible ne change pas. Pour la couche OS, une légère variation est observée. Cela correspond à l'ajout de deux files d'attente de tâches bloquées. Elles sont utilisées pour les blocages sur les deux ports supplémentaires utilisés par la tâche « master ». Le code généré à parti du fichier MPI_Send_Ressources.h.gen est visible en annexe.

		Maître	Esclave
MPI	Taille de code	923	829
	Données en RW	3028	2772
	Données en RO	436	148
OS	Taille de code	443	439
	Données en RW	712	712
	Données en RO	88	48
HAL	Taille de code	1005	1005
	Données en RW	1884	1884
	Données en RO	84	84

Tableau 10 : Taille du code généré pour le master et un slave

Les performances

Le Tableau 11 contient des mesures réalisées sur le temps d'exécution des primitives MPI_SEND et MPI_RECV. Elles ont été réalisées en utilisant des compteurs intégrés dans la plateforme ARM. Le processeur utilisé est un ARM9 cadencé à 120 MHZ. Le tableau présente les temps mesurés pour plusieurs tailles de messages et plusieurs tailles de tampon de communication. Avec ces mesures, on observe que dès qu'un message à une taille supérieure au tampon de communication, le temps d'exécution augmente très rapidement. Ceci est dû au temps ajouté par le blocage de la tâche et son déblocage, ainsi que l'augmentation du nombre d'échanges pour le contrôle des FIFO.

	Taille du message en octets	MPI_Send				MPI_Recv			
Taille du buffer en octets		*64*	*256*	*1024*	*4096*	*64*	*56*	*1024*	*4096*
Temps d'exécution en micro secondes	*16*	113	113	114	113	194	193	193	193
	64	123	124	122	123	298	275	276	276
	256	3288	152	158	161	950	628	606	606
	1024	15287	5060	308	306	3535	2269	1950	1928
	4096	63218	24097	12250	897	13874	8817	7559	7243

Tableau 11:Temps d'exécution des primitives MPI_SEND et MPI_RECV

Les performances observées sont plutôt mauvaises puisque l'on obtient une bande passante moyenne de 0,18MB/s. Cette lenteur de communication est explicable par plusieurs raisons :

- Le contrôle des FIFO par deux processeurs est trop coûteux en échanges de données et interruptions.
- L'implémentation de l'arbitrage des interruptions en logiciel prend beaucoup de temps.
- Après vérification l'utilisation des bursts n'était pas effective : le burst incrémentale ne fonctionne pas sur la plateforme.

5.1.6 Conclusion

L'extraction des fonctionnalités et leur découpage n'ont pas été très concluants. Le découpage en unités fonctionnelles n'a pas été homogène. Certaines unités fonctionnelles comme « MPI_Send » contiennent trop de fonctionnalités et d'autre comme «cells operation » trop peu. Certaines unités fonctionnelles font directement allusion au style d'implémentation comme « memory access » et « bus access ». La démarche consistant à extraire un modèle de représentation à partir d'une implémentation logicielle n'est pas facile. La connaissance de l'implémentation et de l'architecture existante influence le découpage en unités fonctionnelles, il est difficile de s'en abstraire. Une expérience ou la description du modèle de service de communication et la conception de l'interface est conjointe doit être menée.

L'implémentation 100% logicielle est trop lente. Cela montre seulement qu'une implémentation avec autant de logiciel ne permet pas d'obtenir de bonnes performances. Si par exemple, les FIFOs avaient été implémentées en matériel, une bonne partie des accès mémoires et interruptions auraient été évitées. Cette conclusion est peut être un peu caricaturale puisque cette implémentation est extrême. Mais elle montre que la recherche d'un bon partitionnement logiciel matériel est nécessaire pour atteindre les contraintes imposées lors de la réalisation d'une application.

5.2 Implémentation mixte logicielle/matérielle

Cette section présente la deuxième expérience. L'ensemble de primitives implémentées correspond à celui qui a été sélectionné au chapitre 3. La réalisation de ces primitives est, cette fois ci, réellement mixte avec des parties logicielles et matérielles équilibrées. Cette expérience a permis de faire en parallèle la conception de l'interface et la représentation des services avec le modèle présenté au chapitre 4. On présente en premier lieu la représentation des primitives à base d'unités fonctionnelles et de stockage de données, puis les caractéristiques de ces réalisations.

5.2.1 Représentation des primitives du sous-ensemble MPI avec le modèle de service de communication

a Objectif

L'objectif est d'arriver à représenter le comportement de services de communication à base d'ensembles d'unités fonctionnelles et d'unités de structures de données. Ces unités doivent pouvoir être implémentées en logiciel ou en matériel. La granularité de découpage doit être suffisamment fine pour permettre plusieurs partitionnement et permettre la customisation. Mais elle ne doit pas être trop petite pour ne pas contraindre l'implémentation.

b Les différentes unités utilisées

Ce paragraphe rassemble l'ensemble des unités utilisées sans donner de détails de comportement.

Liste des unités fonctionnelles:

- ***Address Decoder*** : correspondance entre adresse où le message doit être envoyé et numéro de processeur
- ***Message Management*** : gestion de message
- ***Message Sender*** : permet d'envoyer un message
- ***Message Receiver*** : permet de recevoir un message
- ***Request initiator*** : permet d'initier une communication non bloquante
- ***Request management*** : permet de gérer les communications non bloquantes pendantes
- ***Buffer Controler*** : gestion de tampon
- ***Request to Send Signalization*** : envoi de signal indiquant l'intention d'envoyer un message
- ***Request to Receive Signalization*** : envoi de signal indiquant l'intention de recevoir un message
- ***Request to Send Signal Treatment*** : traitement des requêtes d'envoi de message
- ***Request to Receive Signal Treatment*** : traitement des requêtes de réception de message
- ***Ack Signal Receive Treatment*** : traitement des signaux d'acknowledgment
- ***Ack Signal Send Treatment*** : traitement des signaux d'acknowledgment
- ***Ready Signal Treatment***: traitement du signal ready
- ***Ready Signalization***: envoi de signal signifiant que le receiver est prêt
- ***Message Status Controller***: contrôle un tampon contenant le statut de chaque message
- ***Request Signal Treatment*** : traitement des requêtes
- ***Packetization*** : découpage du message en paquets
- ***Reassembly*** : assemblage des paquets en messages

- ***Media access 1*** : permet l'envoi de données à travers le média

- ***Media access 2*** : permet la réception de donnée depuis le média

Listes des unités de structures de données :

- ***Message Id Buffer*** : tampon d'identificateur de message

- ***Status Table*** : tampon de stockage d'états de communication.

- ***Communication Buffer*** : tampon de communication

Le tableau ci-dessous présente les unités dont a besoin chaque primitive. Les unités en gris dans le tableau peuvent être différentes selon les primitives qui les utilisent.

Unités Fonctionnelles	SEND	SSEND	ISSEND	IRECV	RECV
Address Decoder	×	×			×
Message Management	×	×	×	×	×
Message Sender	×	×	×		
Message Receiver				×	×
Request Initiator			×	×	
Request Management			×	×	
Buffer Controler	×				
Request to Send Signalization	×	×	×		
Request to Receive Signalization					
Request to Send Signal Treatment				×	×
Request to Receive Signal Treatment	×	×	×		
Ack Signal Send Treatment	×	×	×		
Ack Signal Receive Treatment				×	×
Ready Signal Treatment:	×	×	×		
Ready Signalization				×	×
Message Status Controller	×	×	×	×	×
Packetization	×	×	×		
Reassembly				×	×
Media access 1	×	×	×	×	×
Media access 2	×	×	×	×	×

Tableau 12: Liste des unités fonctionnelles

Unités de stockage de données	SEND	SSEND	ISSEND	IRECV	RECV
Message Id Buffer	×				
Status Table	×	×	×	×	×
Communication Buffer	×				

Tableau 13 : Liste des unités de stockage de données

c MPI_SEND standard avec tampon

La Figure 61 montre la représentation du service de communication MPI_SEND à base d'unités fonctionnelles et d'unités de stockage de données. Les flèches indiquent les dépendances entre chaque unité. Le rectangle dessiné avec des tirets ne fait pas partie du service de communication. C'est le tampon source utilisé par les tâches. Il a été représenté sur ce dessin pour la bonne compréhension du comportement de la communication. On observe que les UF « buffer controler » et « packetization » ont besoin d'y accéder pour lire les données à envoyer ou à copier. La représentation de MPI_SEND est différente de celle présentée lors de la première expérience car l'implémentation conçue à partir de la spécification du standard MPI est plus complexe. Le comportement de cette version de MPI_SEND permet notamment de faire du « multithreading » de la communication (plus de détails sont donnés avec le paragraphe 5.2.2.c).

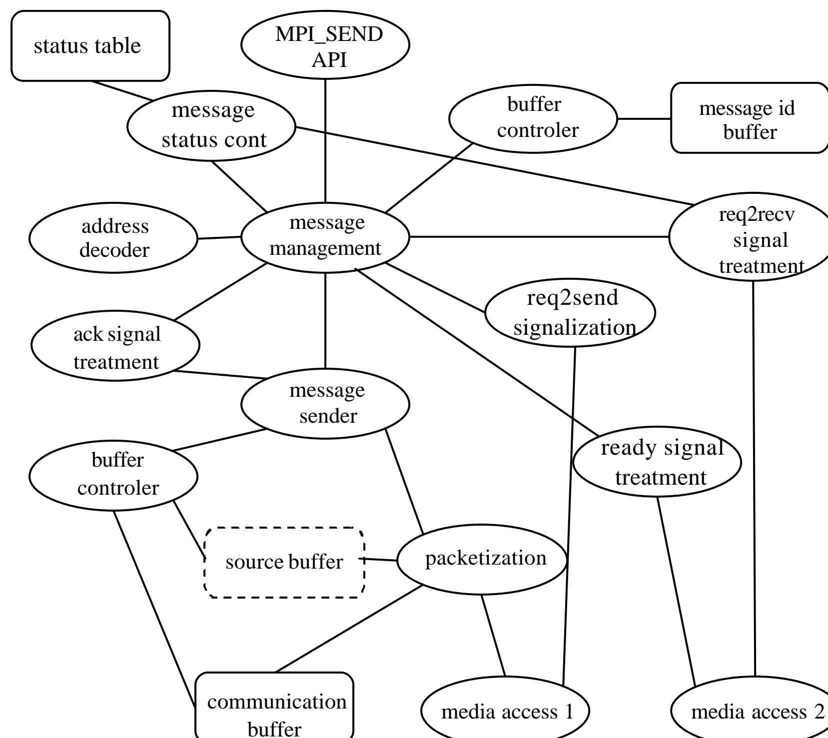

Figure 61: Découpage en unités fonctionnelles et unités de structures données pour MPI_SEND

d MPI_SSEND

MPI_SSEND appelé « send » synchrone ressemble à un MPI_SEND sans tampon. Comme on peut le voir sur la Figure 62, les USD « message id buffer » et « communication buffer » ainsi que les UF « buffer contrôler » ont été enlevées.

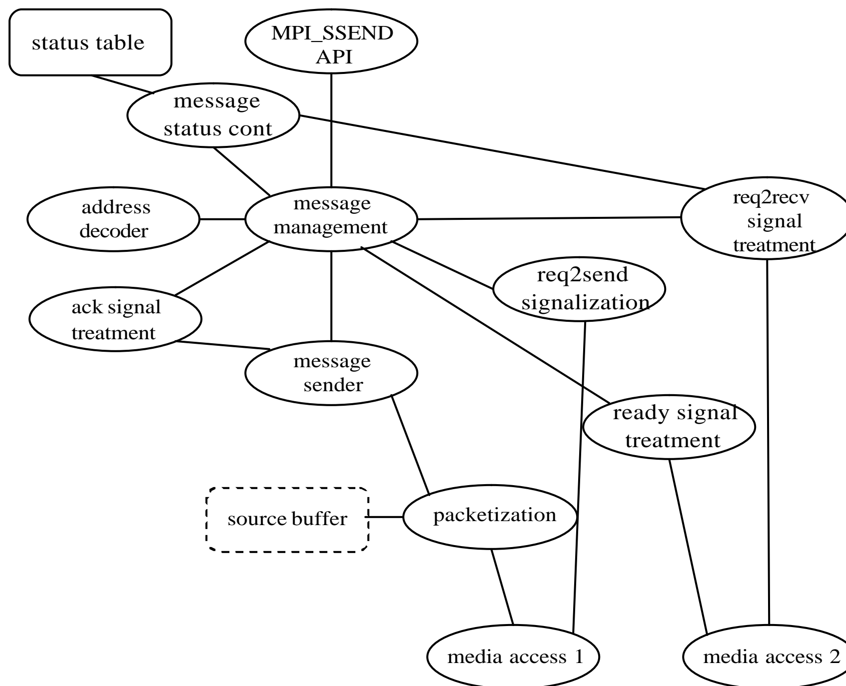

Figure 62 : Découpage en unités fonctionnelles et unités de structures données de MPI_SSEND

e MPI_ISSEND

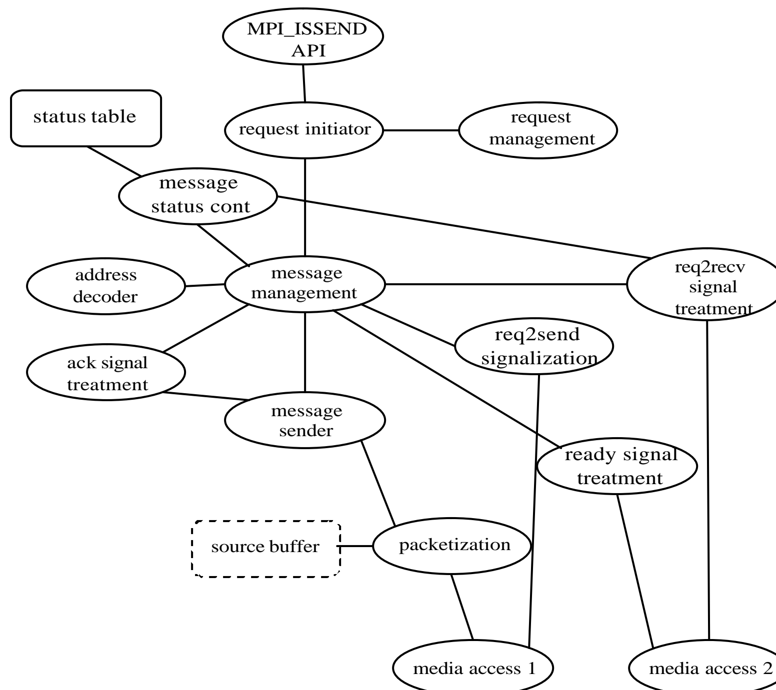

Figure 63 : Découpage en unités fonctionnelles et unités de structures données de MPI_ISSEND

f MPI_RECV

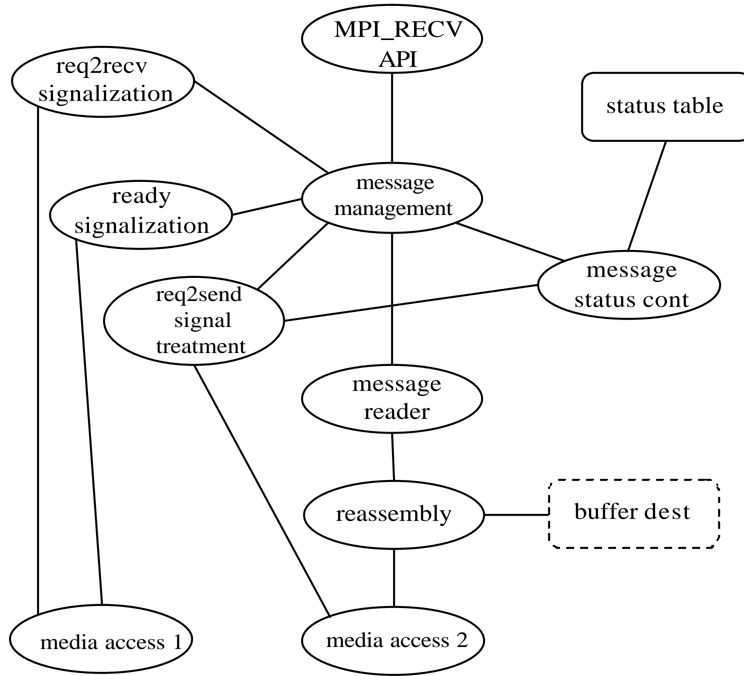

Figure 64 : Découpage en unités fonctionnelles et unités de structures données pour MPI_RECV

g MPI_IRECV

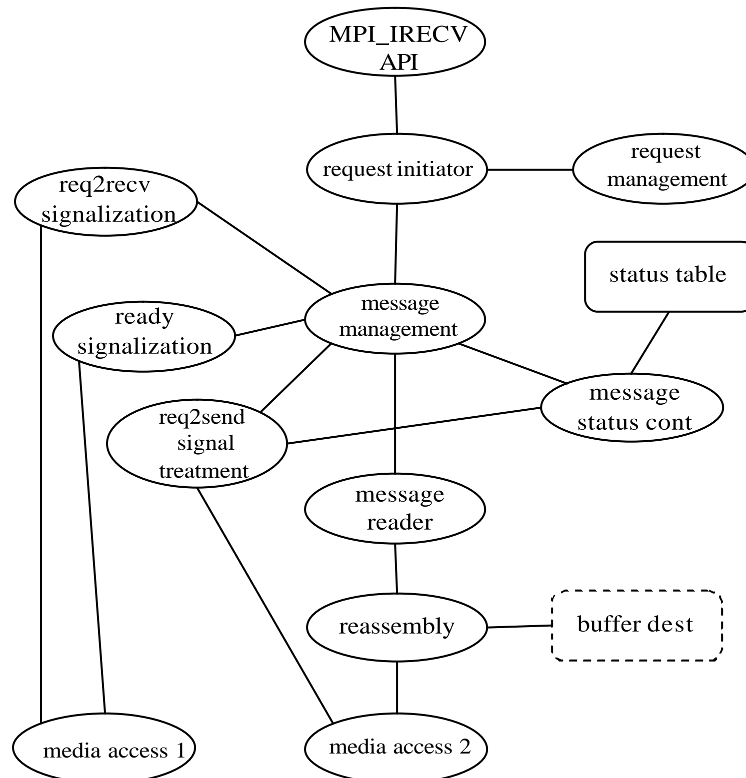

Figure 65 : Découpage en unités fonctionnelles et unités de structures données de MPI_IRECV

5.2.2 Les caractéristiques de l'implémentation

a Allocation des UF et USD dans les parties logicielles et parties matérielles

Les trois figures suivantes présentent le choix de partitionnement effectué pour chaque primitive de communication. Tout ce qui est contrôle de la synchronisation des messages est implémenté en logiciel. Ainsi les UF d'émission et de traitement des requêtes servant à synchroniser les tâches et les UF « message management » sont implémentées en logiciel. La partie matérielle contient l'USD « communication buffer » pour MPI_SEND et tout ce qui est chargement, découpage, transfert et réassemblage des données. Ce choix de partitionnement a été fait pour obtenir une répartition équilibrée des unités dans chaque partie de l'interface.

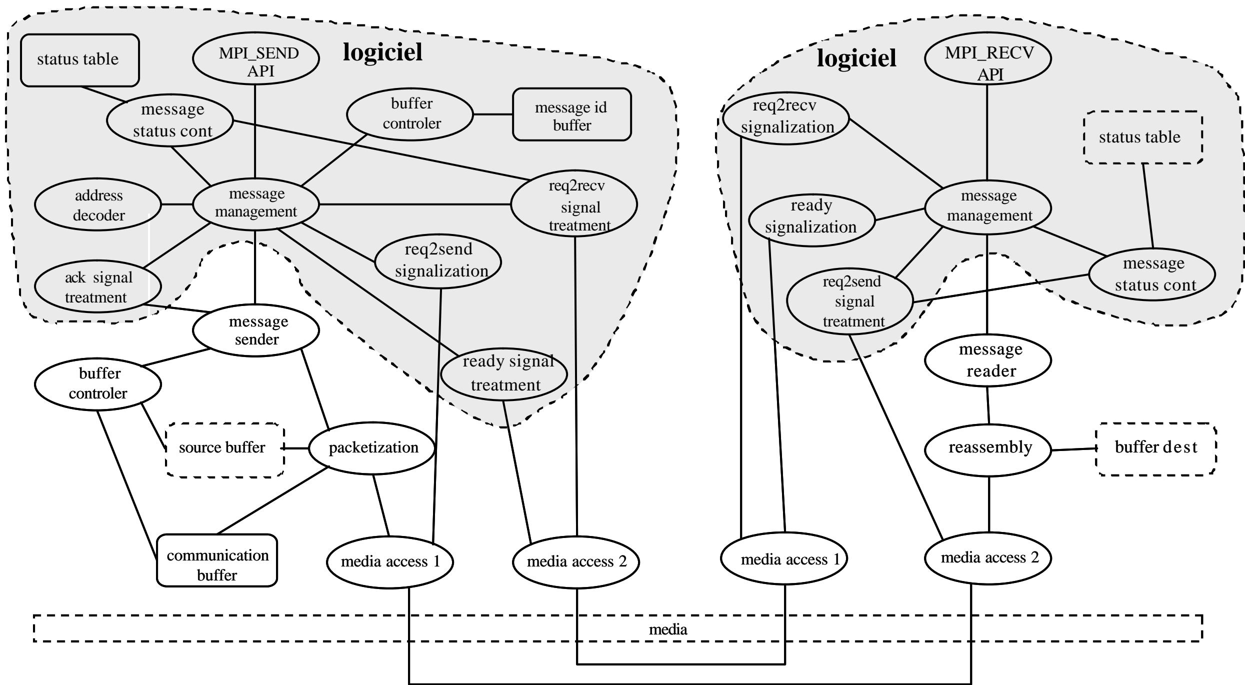

Figure 66: Allocation des UF et USD de MPI_SEND et MPI_RECV dans les parties logicielles et matérielles

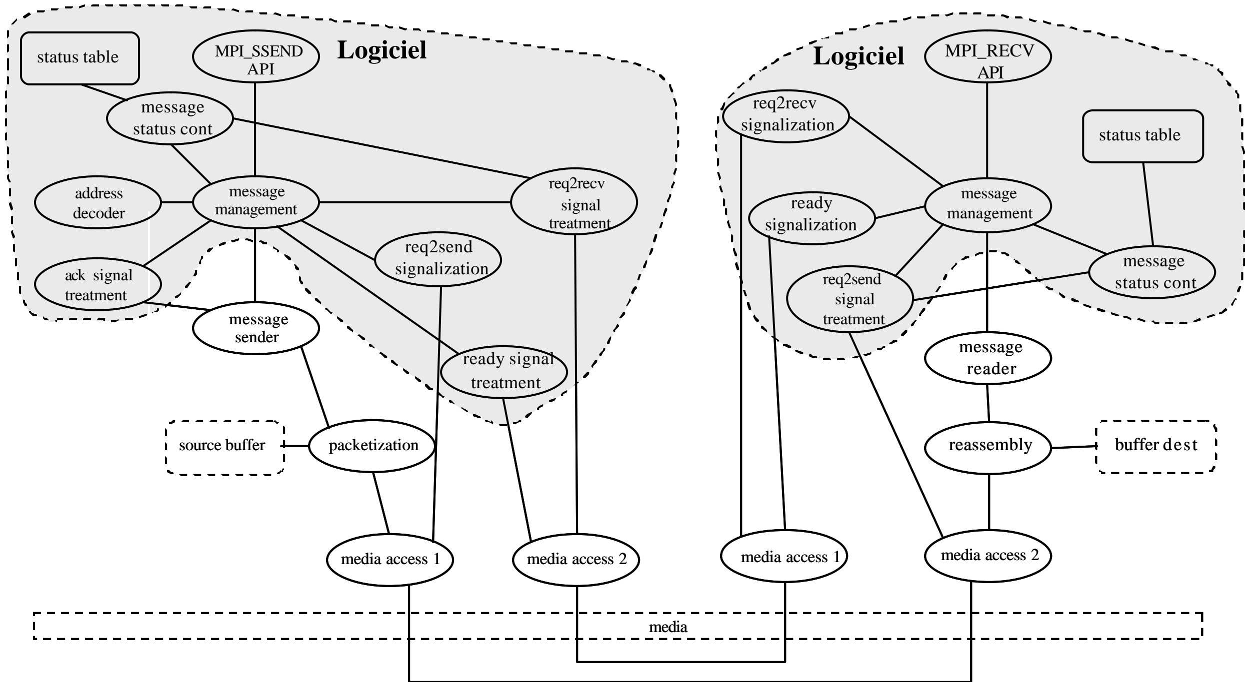

Figure 67: Allocation des UF et USD de MPI_SSEND et MPI_RECV dans les parties logicielles et matérielles

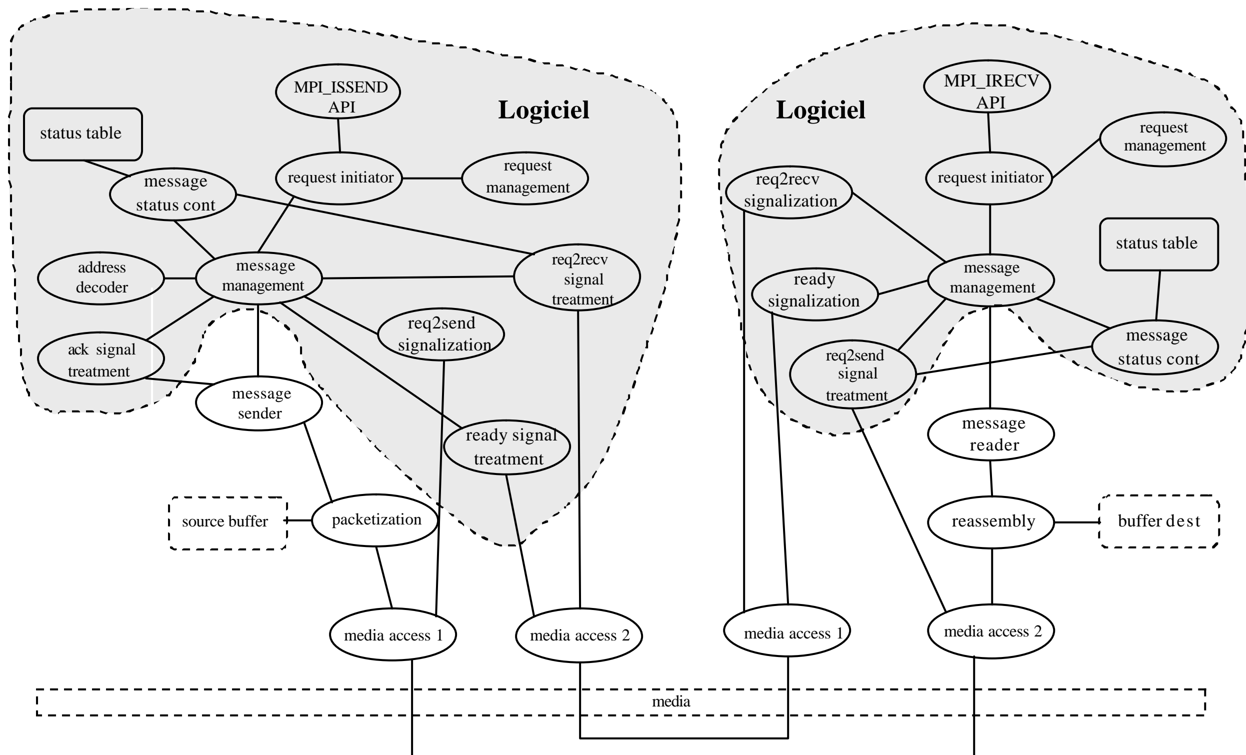

Figure 68: Allocation des UF et USD de MPI_ISSEND et MPI_IRECV dans les parties logicielles et matérielles

b Communication entre partie logicielle et partie matérielle

La Figure 69 montre les relations entre unités fonctionnelles qui nécessiteront l'implémentation d'une communication entre la partie logicielle et la partie matérielle. Cette figure ne présente que le cas des primitives MPI_SEND et MPI_RECV puisque les autres primitives conservent les mêmes caractéristiques de communication entre le logiciel et le matériel.

Seul le premier cas de communication est présenté en détails. Il concerne la communication entre « message management » et « message sender » (flèche 1). On utilise des écritures et lectures dans des registres et des interruptions. Voici, les différentes interactions possibles entre ces deux unités fonctionnelles :

- Pour envoyer un message précédemment copier dans le tampon :
  - Ecriture de l'identificateur de canal à l'adresse ADDR_SM
  - Lecture à l'adresse ADDR_DISPO_MS pour savoir si le matériel est disponible. Tant que celui-ci n'est pas libre, on scrute cette adresse.
  - Indique que le matériel est utilisé en écrivant la valeur à l'adresse ADDR_SM.
  - Ecriture des informations nécessaires à l'envoi du message à l'adresse ADDR_SM : on écrit l'adresse de destination, le tag du message, le nombre d'éléments, le type de données et l'adresse source.
- Pour envoyer un message précédemment copié dans le tampon :
  - Ecriture de l'identificateur de canal à l'adresse ADDR_SBM
  - Lecture à l'adresse ADDR_DISPO_MS pour savoir si le matériel est disponible. Tant que celui-ci n'est pas libre, on scrute cette adresse.
  - Indique que le matériel est utilisé en écrivant la valeur à l'adresse ADDR_SBM.

- Ecriture à l'adresse ADDR_SBM des informations nécessaires à l'envoi du message copié: on écrit l'adresse de destination, le tag du message, le nombre d'éléments et le type de données.

- Pour copier un message dans le tampon :
  - Ecriture de l'identificateur de canal à l'adresse ADDR_CMTB
  - Lecture à l'adresse ADDR_DISPO_MS pour savoir si le matériel est disponible. Tant que celui-ci n'est pas libre, on scrute cette adresse.
  - Indique que le matériel est utilisé en écrivant la valeur à l'adresse ADDR_CMTB.
  - Ecriture à l'adresse ADDR_CMTB des informations suivantes : le tag du message, le nombre d'éléments et le type de données.
  - Lecture à l'adresse OK_BUFFER pour tester si le tampon a suffisamment de place pour contenir le message. Si ce n'est pas le cas, on bloque la tâche et le processeur est alloué à une autre tâche. Le déblocage se fera par une interruption.
  - Ecriture de l'adresse source à ADDR_CMTB.

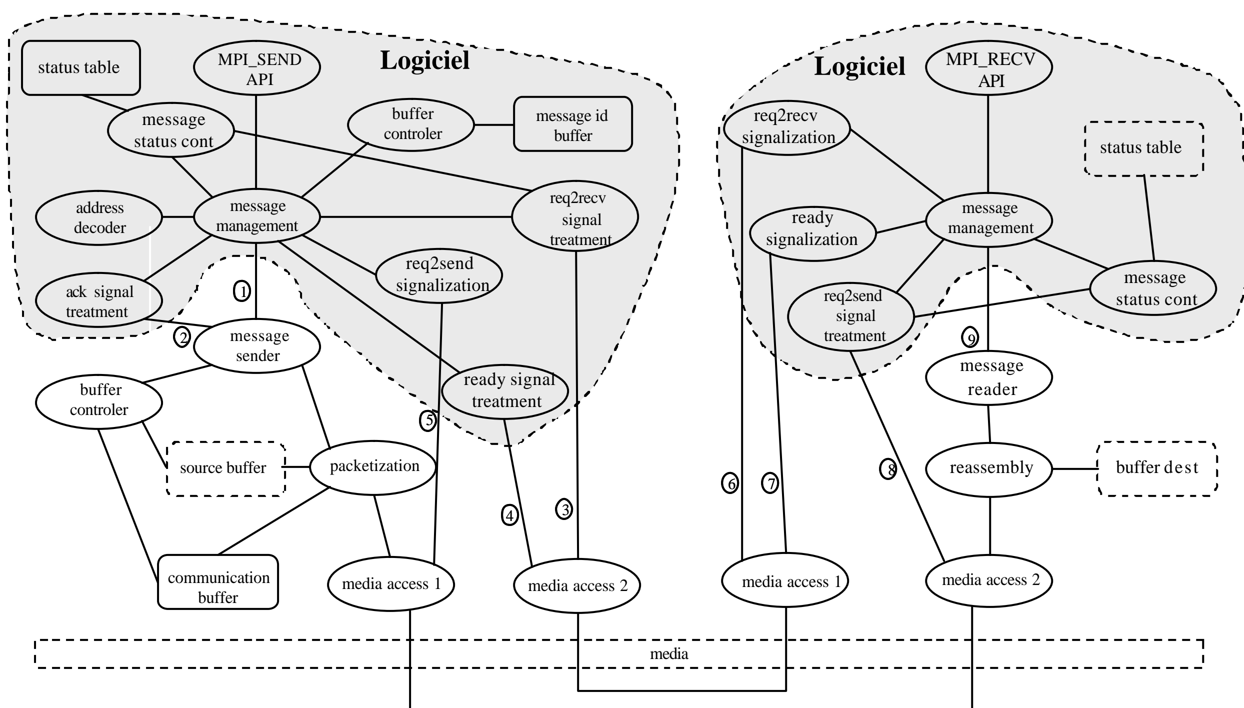

Figure 69 : Communication entre partie logicielle et partie matérielle

c Multithreading

L'un des objectifs de ces implémentations était de rendre possible l'envoi de plusieurs messages en même temps. Par exemple, si une tâche commence à envoyer un gros message et que son temps d'utilisation du processeur est dépassé, une autre tâche est exécutée. Avec le « multithreading » de la communication, cette tâche peut envoyer ou recevoir un message même si la communication initiée par la tâche précédente n'est pas finie. L'utilité du « multithreading » de la communication est de favoriser la réactivité de la communication

(diminution de la latence) au dépit de la bande passante. Pour atteindre cet objectif, on peut utiliser plusieurs méthodes.

La première consiste à multiplier toutes les ressources de communication. C'est-à-dire, pour la partie matérielle, d'avoir des blocs matériels dédiés à chaque canal de communication. Pour la partie logicielle, le nombre de variable est multiplié par le nombre de canaux et chaque communication est exécutée dans un processus séparé.

La deuxième méthode consiste à limiter la multiplication de ressources en essayant de partager les ressources et sauvegarder les états de communication avant chaque changement de tâche. C'est cette méthode qui a été choisie pour ces implémentations. Le but était de favoriser le « multithreading » lorsque celui-ci n'est pas trop coûteux en ressources. Nous avons donc choisi :

- de rendre la partie allouée au matérielle critique. Cette partie n'est pas interruptible, elle ne peux pas être utilisée tant que sont traitement n'est pas terminé.
- d'exécuter les communications bloquantes dans les tâches appelant ces communications
- d'exécuter les communications non bloquantes dans une tâche dédiée à la communication. Il y a donc une tâche de communication par canal de communication non bloquante
- de multiplier certaines variables nécessaires à la communication
- de créer des zones critiques avec masquage des interruptions lors d'accès à des ressources partagées. Ces ressources partagées peuvent être des variables pour la communication entre tâches, des variables de stockage de l'état de la communication, ou des ressources matérielles (par exemple accès aux registres permettant la communication entre le logiciel et le matériel).

La Figure 70 montre la stratégie employée pour l'implémentation des communications bloquantes. Elle montre le cas de MPI_SEND ou la partie logicielle de la communication est implémentée dans les tâches de calcul utilisant la communication. Toute la partie matérielle est implémentée dans un seul processus.

La stratégie employée pour l'implémentation des communications non bloquantes est légèrement différente. Pour la partie logicielle, la plus grande partie de la communication est exécutée par une tâche dédiée à la communication. Seule la partie initiation des communications est conservée dans la tâche appelant la communication. La partie matérielle est implémentée de la même façon que pour les communications bloquantes.

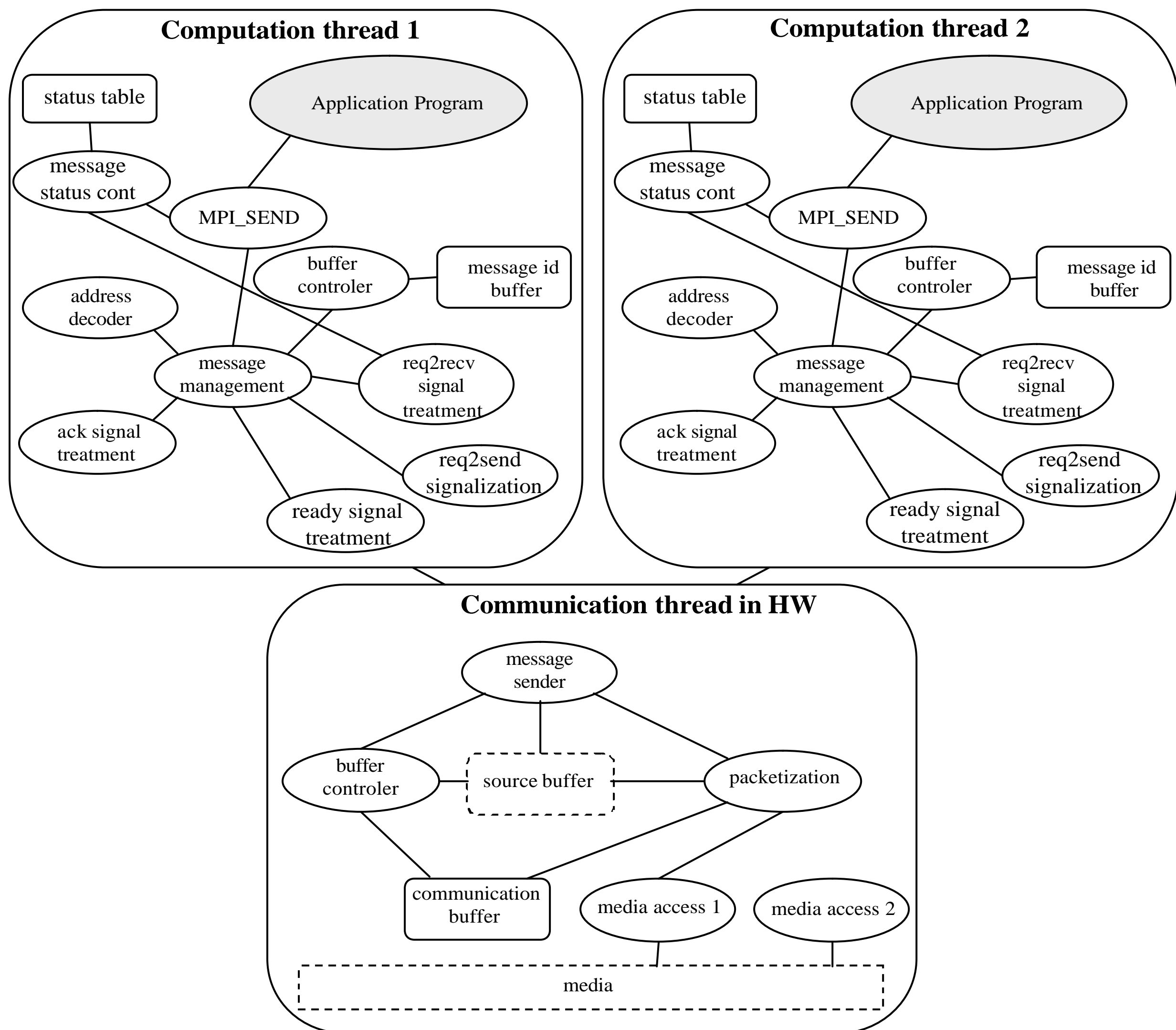

Figure 70 : Illustration du « multithreading » de la communication avec MPI_SEND

5.2.3 La partie logicielle

a Le code développé

Le code a été développé en C pour les parties indépendantes du processeur et en assembleur ARM pour certaines parties de la couche HAL. La taille du code réalisé implémentant les services de communication sélectionnés, la partie OS nécessaire à l'exécution de deux tâches et la couche HAL sont présentées dans le tableau suivant.

	Nombre de ligne de code	Nombre de fichiers
COM	1713	35
OS	558	10
HAL	847	7

Tableau 14 : Taille du code implémentant le sous ensemble MPI

b La méthode de validation utilisée

Le but était de valider le code développé sans avoir développé la partie matérielle. Il fallait valider le maximum de code en un minimum de temps. Etant donné que la partie HAL était une évolution de celle développée lors de la première expérience, nous avons décidé de valider la partie du code indépendante du processeur. Pour cela, il a fallu modéliser les parties dépendantes du matériel. Nous avons utilisé un modèle de couche HAL développé par Aimen Bouchiman du groupe SLS . Ce modèle écrit en C++ implémente les services fournis par la couche HAL et est connectable à des modules SystemC. Pour simuler la partie matérielle, nous avons développé un module SystemC ayant le comportement de communication présenté au paragraphe 5.2.2.b. La figure suivante présente l'application utilisée pour valider le code. L'objectif était de tester la possibilité d'envoyer plusieurs messages en même temps. La tâche T1 utilise les services MPI_SEND, MPI_RECV et MP_ISSEND. La tâche T2 utilise les services MPI_ISSEND et MPI_IRECV. Ces services sont fournis par les couches de communication et d'OS implémentées. Le code implémenté utilise les services fournis par l'environnement de simulation modélisant la couche HAL.

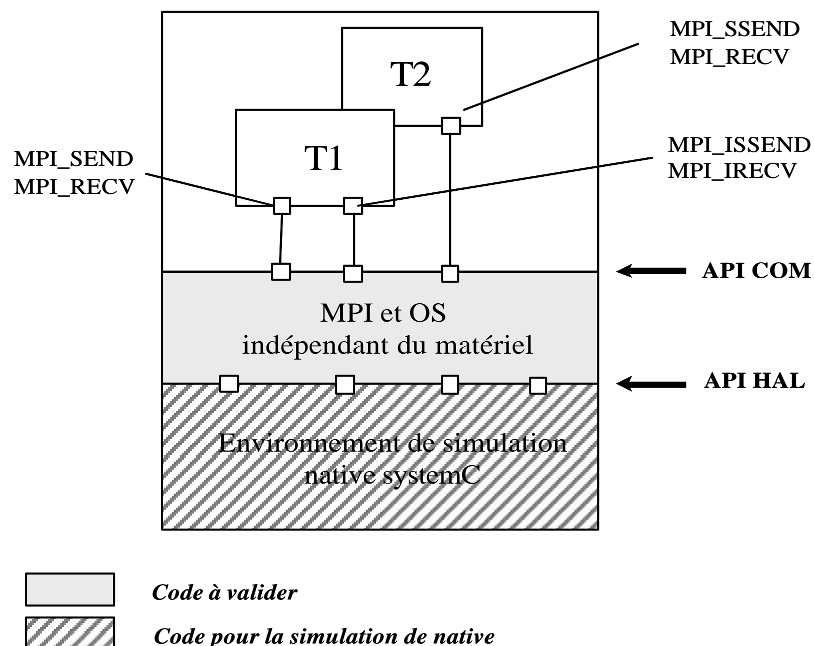

Figure 71 : Exécution de deux tâches pour la validation du code indépendant du matériel

La figure 72 présente l'environnement de simulation développé. Le modèle de couche HAL en SystemC est connecté à un bus TLM permettant le transfert de données avec le modèle d'interface matérielle. Des canaux

permettent au modèle matériel (testbench) de générer des interruptions matérielles. Ce module modélisant le comportement de l'interface matérielle est une machine d'états réagissant aux sollicitations de la couche HAL. Ce module peut aussi lire un scénario contenu dans un tableau. Dans chaque case, on fixe l'état que l'on veut imposer au module. On peut ainsi fixer des séquences d'états et donc créer plusieurs scénarios de communication. Un troisième module implémente un timer communiquant via un canal FIQ. Ce «timer» permet de fixer le quanta de temps alloué à chaque tâche.

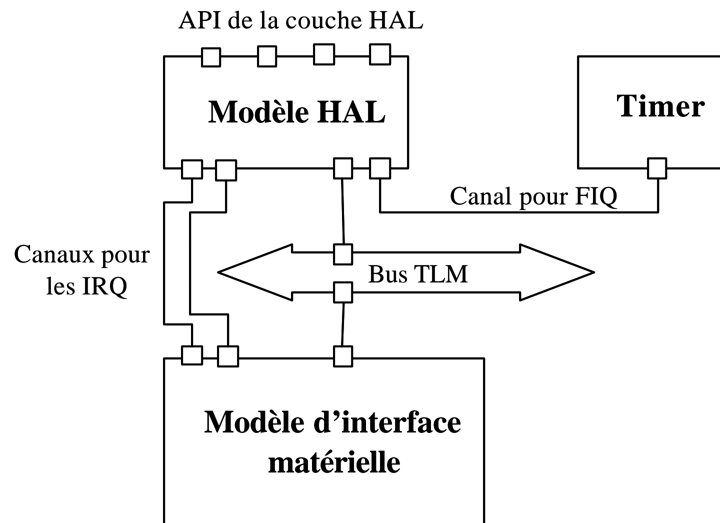

Figure 72 : Les modèles utilisés pour la validation

Cet environnement de validation a permis le débuguage du code avec une débugueur classique. Il nous a permis notamment de corriger des erreurs d'algorithme ainsi que des erreurs dans les protocoles de communication entre l'interface logicielle et l'interface matérielle. Le « multithreading » des communications a pu être vérifié.

5.2.4 La partie matérielle

La partie matérielle n'a pas été implémentée mais son architecture et son comportement ont été définis avec Arnaud Grasset qui prépare le flot de génération automatique de la partie matérielle des interfaces. Avec cette implémentation matérielle nous avons voulu obtenir un traitement séquentiel rapide de la communication. Lorsqu'une demande d'envoi de message a été demandée par le logiciel, la partie matérielle ne s'interrompt pas tant que le message n'a pas été envoyé. La Figure 73 montre l'architecture choisie. Pour favoriser la performance, deux DMA sont utilisés pour les copies de données dans le tampon de communication et pour le transfert de données vers le tampon des destinataires des messages. Cette interface étant conçue pour communiquer via un bus AMBA AHB, elle contient une interface AHB maître et une interface AHB esclave.

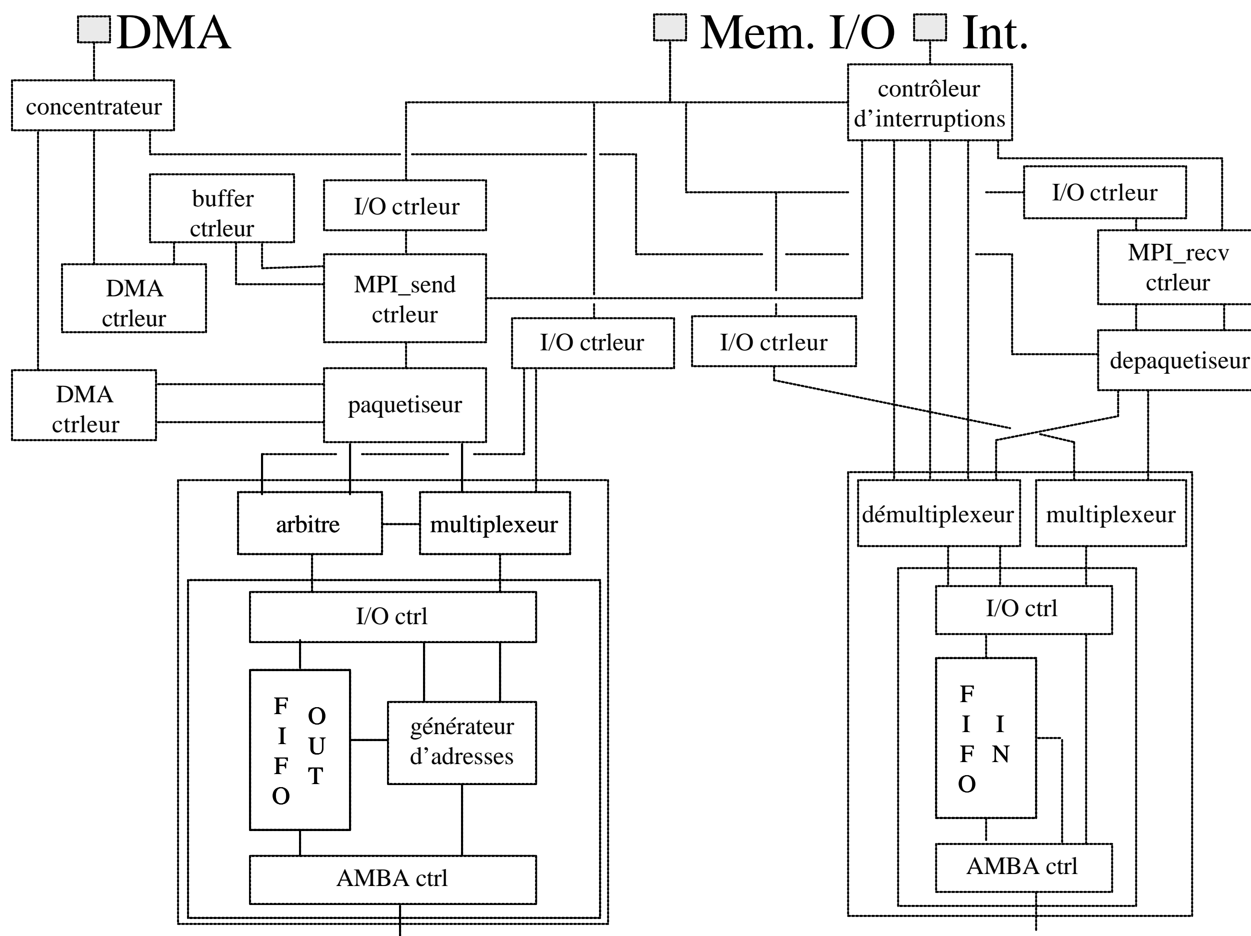

Figure 73: Architecture de la partie matérielle

5.3 Conclusion

Ces deux expériences on permis de tester deux types d'implémentations logicielles/matérielles de primitives MPI. La première expérience à montrer la nécessité d'utiliser des implémentations mixtes. Un implémentation 100% logicielle est en effet très peu performante. Une implémentation 100% matérielle aurait très certainement donné de bonnes performances mais sans offrir aucune flexibilité.

Ces expériences ont aussi permis d'expérimenter le découpage des services de communication et leur représentation avec le modèle proposé. Après un découpage trop simpliste lors de la première expérience, les représentations faites pour la deuxième expérience semblent satisfaisantes. Les unités extraites sont indépendantes du style d'implémentation et on peut aisément imaginer d'autre partitionnement que ceux utilisés pour la réalisation proposée. Par contre, ces expériences ont aussi montré qu'il est difficile de faire un bon choix de découpage en unités fonctionnelles et unités de stockage de données. Il faut toujours penser aux possibilités d'implémentation en logiciel ou en matériel, aux possibilités de customisation et à ne pas contraindre l'implémentation.

Chapitre 6 Conclusion et perspectives

6.1 Conclusion

Les technologies actuelles permettent l'intégration de systèmes de plus en plus complexes sur une seule puce. L'augmentation de la complexité induit un accroissement du temps de conception alors que paradoxalement, la concurrence économique impose des temps de mise sur le marché de plus en plus courts. Pour tenter de limiter ce décalage et d'accroître la productivité, l'industrie fait de plus en plus appel à la réutilisation de composants logiciels et matériels pour concevoir leurs systèmes. La réutilisation, pose des problèmes de communication entre composants.

Dans cette thèse, nous nous sommes focalisés sur l'implémentation des services de communication pour la communication entre processeurs via des réseaux de communication. Ces communications sont implémentées sous forme d'interfaces mixtes logicielles/matérielles. Les implémentations mixtes logicielles/matérielles permettent aux concepteurs de trouver des compromis entre performance et coût de réalisation, alors que l'implémentation logicielle ou matérielle seule ne le permettrait pas. Cependant, leur réalisation est longue et complexe car bien souvent effectuée manuellement. Les connaissances multidisciplinaires nécessaires et les interactions entre les équipes de développement logiciel et celles développant le matériel ne simplifient pas l'implémentation de la communication. De plus, les avantages et inconvénients liés à chaque type d'implémentation offrent un large espace de solutions d'implémentation. Il faut donc trouver des solutions pour diminuer les temps de conception des interfaces mixtes logicielles/matérielles et permettre l'exploration des différentes implémentations possibles.

La contribution de cette thèse consiste en une formalisation des problèmes liés à l'implémentation mixte logicielle/matérielle des services de communication et une proposition de solution de génération automatique. L'objectif était d'utiliser les outils ASOG et ASAG du groupe SLS pour générer respectivement la partie logicielle et la partie matérielle des implémentations. Pour cela, un modèle commun de représentation des fonctionnalités composant les services de communication a été proposé. Ce modèle doit servir de description d'entrée pour les outils de génération et permettre le partitionnement des services de communication. Une autre contribution de cette thèse est le développement de composants pour la génération automatique d'implémentations de primitives MPI, qui est l'un des modèles de programmation parallèle permettant la communication par passage de message. La première réalisation a été l'implémentation 100% logicielle des primitives MPI_SEND et MPI_RECV pour la communication entre processeurs sur une plateforme ARM. La seconde réalisation, a été l'implémentation mixte logicielle/matérielle d'un sous ensemble plus étendu de

primitives MPI. La complexité de ces implémentations a permis d'expérimenter le découpage des fonctionnalités selon le modèle proposé.

Le **chapitre 1** a introduit le contexte de cette thèse en présentant les systèmes sur puce et les problèmes associés à leur conception. Il a aussi permis d'expliquer les objectifs à long terme et à court terme des ces travaux. Les contributions de cette thèse ont été présentées.

Le **chapitre 2** nous a permis de détailler le flot de conception ROSES et de comprendre le fonctionnement des outils de génération des parties logicielles et matérielles des interfaces. Une illustration avec l'application VDSL a permis au lecteur de connaître plus en détails l'utilisation du logiciel ASOG. L'utilité de ce chapitre a été principalement de fixer le cadre de développement utilisé lors de la réalisation de ces travaux de recherche.

Avec le **chapitre 3**, la première section a permis de présenter l'architecture logicielle/matérielle des interfaces. Ensuite, la seconde section a présenté MPI. Le choix d'un modèle de programmation comme MPI illustre bien la complexité que peut atteindre un modèle communication. La dernière section, nous a présenté le sous ensemble de primitives MPI sélectionné pour nos expériences.

Le **chapitre 4** a précisé la problématique associée au partitionnement ainsi qu'une proposition de modèle de représentation des services de communication. Ce modèle a été développé pour être utilisé dans le cadre d'un flot d'exploration du partitionnement logiciel/matériel des implémentations de services de communication. Mais aussi pour permettre la sélection de fonctionnalité pour obtenir l'implémentation de services de communication sur mesure. Enfin une méthode pour la génération automatique des interfaces logicielle/matérielles a été proposée.

Le **chapitre 5** nous a présenté les deux principales expériences réalisées lors de la thèse. La première a permis d'obtenir la génération d'une implémentation 100% logicielle de primitives MPI sur une plateforme ARM. L'utilité de cette implémentation a été d'implémenter des protocoles de communication dans l'outil ASOG. Il a aussi permis de souligner le manque de performance d'une 100% logicielle. Enfin, cela a été l'occasion de la naissance de l'idée d'un modèle de représentation des services de communication. La deuxième expérience a été l'occasion de concevoir des implémentations mixtes d'un sous ensemble de primitives MPI. Elle a permis de tester la réalisation conjointe de représentations de primitives de communication à l'aide du modèle développé et l'implémentation de la partie logicielle. Pour être complète, cette expérience mériterait d'être complétée par l'implémentation de la partie matérielle et l'intégration des composants dans les outils de génération automatique.

6.2 Perspectives

Les perspectives présentées sont classées par ordre décroissant d'« urgence » de développement pour atteindre l'objectif à long terme d'exploration automatique d'implémentations mixtes logicielles/matérielles des services de communication.

6.2.1 Développement du flot de génération de la partie logicielle à partir du modèle développé

Il faudrait intégrer les implémentations des primitives MPI de la deuxième expérience dans ASOG et développer l'extension permettant de lire le modèle de représentation du comportement des services de communication. Cela permettrait de vérifier la faisabilité de cette méthode et ainsi de déterminer s'il est utile de chercher à étendre ASAG pour l'utilisation de ce modèle.

6.2.2 Développement d'un flot de génération de la partie matérielle à partir du modèle commun

Cette perspective doit impérativement être investie si l'on veut pouvoir faire de l'exploration automatique d'implémentations mixtes logicielle/matérielles des services de communication. Elle n'est cependant pas simple à réaliser à cause de la forte influence du choix d'architecture sur les performances des implémentations matérielles. Il faudra sûrement trouver un moyen de diriger le choix de l'architecture en offrant des choix de motifs d'implémentation ou d'architectures génériques.

6.2.3 Estimation de performances

Il sera nécessaire d'obtenir un moyen de « juger » les implémentations générées. Il faut donc développer un moyen de caractériser l'intégralité de l'interface logicielle/matérielle. Actuellement, plusieurs travaux du groupe se focalisent sur l'estimation des performances du logiciel [BAC 02] [BOU 04], mais il faudrait considérer le problème dans son ensemble pour pouvoir extraire des temps de latence et des débits de communication de l'interface complète.

6.2.4 Aide à l'insertion d'éléments de bibliothèque d'ASOG

Les expériences réalisées ont permis de « goûter » à la difficulté d'insertion d'éléments dans ASOG. Ces opérations nécessitent une bonne connaissance du logiciel, la connaissance d'un langage de description des dépendances entre éléments de bibliothèque, la connaissance d'un langage de macro en plus du langage d'implémentation de la partie logicielle, ainsi que la connaissance de l'architecture cible. De plus la visibilité du code déjà intégré dans la bibliothèque est très mauvaise. Il faut impérativement générer le code pour connaître à quoi correspond chaque composant. Il faudrait donc développer des méthodes permettant une insertion plus aisée des éléments de bibliothèque. On pourrait peut être ajouter à chaque éléments des informations sur son comportement, sur les variables globales et les paramètres des fonctions utilisées. Une interface graphique permettant la description des dépendances entre éléments serait aussi bienvenue.

6.2.5 Automatisation du choix de partitionnement

La dernière étape est celle de l'automatisation du choix de partitionnement. Ce problème se résume à une recherche d'optimum. Les travaux dans ce domaine ne manquent pas [GUP 93][ERN 93], et il serait intéressant de faire une synthèse des méthodes existantes.

ANNEXE

Description VADeL de l'application OpenDivX :

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <vadel.h>
#include "../appli_types.h"
#include "../Master/Vmaster.h"
#include "../Slave1/Vslave1.h"
#include "../Slave2/Vslave2.h"
#include "../Slave3/Vslave3.h"

/***************************************************************************
********
      definition of the virtual channels
****************************************************************************
*******/

VA_CHANNEL(vc1)
{
 va_ch_mac_MPI_P2P     mpi_channel;

  VA_CCTOR(vc1)
   {
    VA_CEND
   };
};

VA_CHANNEL(vc2)
{
 va_ch_mac_MPI_P2P     mpi_channel;

  VA_CCTOR(vc2)
   {
    VA_CEND
   };
};

VA_CHANNEL(vc3)
{
 va_ch_mac_MPI_P2P     mpi_channel;

  VA_CCTOR(vc3)
   {
    VA_CEND
   };
};


/***************************************************************************
********
           main program
****************************************************************************
*******/
```

```
int sc_main(int argc, char *argv[])
{
  /*---------------------------- VADeL initialization  --------------------
------------*/
  va_init();

  /*----------------------- instanciate virtual channels  -----------------
----------*/
    vc1 VC1("VirtualChannelMPI1");
    vc2 VC2("VirtualChannelMPI2");
    vc3 VC3("VirtualChannelMPI3");


  /*-------------- instanciate virtuals modules & sc_modules. -------------
-*/
  vmaster vmaster("VMaster");
  vslave1 vslave1("VSlave1");
  vslave2 vslave2("VSlave2");
  vslave3 vslave3("VSlave3");


  /*********************************************************************
**/
  /*---------------------------------- Parameters -----------------------
----------*/
  /*********************************************************************
**/

  /*-----------------------Parameters for Virtual Nets---------------------
------*/
  VC1.level = "DL";
    VC2.level = "DL";
    VC3.level = "DL";

  /*------------------- Parameters for Virtual Ports ----------------------
-------*/
    vmaster.VPmpi1.level = "DL";
    vmaster.VPmpi2.level = "DL";
    vmaster.VPmpi3.level = "DL";

  vslave1.VPmpi1.level = "DL";
    vslave2.VPmpi1.level = "DL";
    vslave3.VPmpi1.level = "DL";

  vmaster.VPmpi1.C_DATA_TYPE = "VOID";
    vmaster.VPmpi2.C_DATA_TYPE = "VOID";
    vmaster.VPmpi3.C_DATA_TYPE = "VOID";

    vslave1.VPmpi1.C_DATA_TYPE = "VOID";
    vslave2.VPmpi1.C_DATA_TYPE = "VOID";
    vslave3.VPmpi1.C_DATA_TYPE = "VOID";

/*
  vmaster.VPmpi1.SystemCDataType = "VOID";
  vmaster.VPmpi2.SystemCDataType = "VOID";
  vmaster.VPmpi3.SystemCDataType = "VOID";
  vslave1.VPmpi1.SystemCDataType = "VOID";
  vslave2.VPmpi1.SystemCDataType = "VOID";
```

```
  vslave3.VPmpi1.SystemCDataType = "VOID";
*/

  /*------------------- Parameters for Virtuals Modules -------------------
-------*/

  /* Virtual Master */
  vmaster.CPUName      = "CpuARMIntegrator";
    vmaster.level       = "DL";
  vmaster.ItNumberAddress   = "0x00136000";
  vmaster.ItTypeAddress     = "0x00136040";
  vmaster.RamBase      = "0x00000000";
  vmaster.RamLimit     = "0x08000000";
  vmaster.FileBase     = "0x06000000";
  vmaster.IrqStack     = "0x06000000";
  vmaster.SvcStack     = "0x05FFFC00";
  vmaster.HeapBase     = "0x00300000";
  vmaster.UsrStack     = "0x04300000";

  //vmaster.ADDRESS_IT    = "0x0000FA00";

  /* Virtual  Slave1 */
  vslave1.CPUName      = "CpuARMIntegrator";
    vslave1.level       = "DL";
  vslave1.ItNumberAddress   = "0x00116000";
  vslave1.ItTypeAddress     = "0x00136040";
  vslave1.RamBase      = "0x00000000";
  vslave1.RamLimit     = "0x08000000";
  vslave1.FileBase     = "0x06000000";
  vslave1.IrqStack     = "0x06000000";
  vslave1.SvcStack     = "0x05FFFC00";
  vslave1.HeapBase     = "0x00300000";
  vslave1.UsrStack     = "0x04300000";

  //vslave1.ADDRESS_IT    = "0x0000FA00";

  /* Virtual  Slave2 */
    vslave2.CPUName      = "CpuARMIntegrator";
    vslave2.level       = "DL";
  vslave2.ItNumberAddress   = "0x00116000";
  vslave2.ItTypeAddress     = "0x00136040";
  vslave2.RamBase      = "0x00000000";
  vslave2.RamLimit     = "0x08000000";
  vslave2.FileBase     = "0x06000000";
  vslave2.IrqStack     = "0x06000000";
  vslave2.SvcStack     = "0x05FFFC00";
  vslave2.HeapBase     = "0x00300000";
  vslave2.UsrStack     = "0x04300000";

  //vslave2.ADDRESS_IT    = "0x0000FA00";

    /* Virtual  Salve3 */
    vslave3.CPUName      = "CpuARMIntegrator";
    vslave3.level       = "DL";
  vslave3.ItNumberAddress   = "0x00116000";
  vslave3.ItTypeAddress     = "0x00136040";
  vslave3.RamBase      = "0x00000000";
  vslave3.RamLimit     = "0x08000000";
  vslave3.FileBase     = "0x06000000";
  vslave3.IrqStack     = "0x06000000";
```

```
  vslave3.SvcStack    = "0x05FFFC00";
  vslave3.HeapBase    = "0x00300000";
  vslave3.UsrStack    = "0x04300000";

  //vslave3.ADDRESS_IT    = "0x0000FA00";

  /*---------------------------- Parameters for  Modules ------------------
-----------*/

  vmaster.Mmaster->setColifParam("module_level","DL");
    vslave1.Mslave1->setColifParam("module_level","DL");
    vslave2.Mslave2->setColifParam("module_level","DL");
    vslave3.Mslave3->setColifParam("module_level","DL");

  /*-------------------------- Parameters for Modules Ports ---------------
-------*/

  /* Internal Ports  ----------------------------------  */

  /* @ Master Module */
  /* Master Port pmpi1 for communication with Slave1*/
  vmaster.Mmaster->pmpi1.C_DATA_TYPE      = "VOID";
    vmaster.Mmaster->pmpi1.SoftPortType       = "PacketsOperations";

  vmaster.Mmaster->pmpi1.R_FIFOTagAddress       = " 0x90100000";//size
9x32b
  vmaster.Mmaster->pmpi1.R_AddressHeadFIFOTagAddress    = " 0x90100024";
  vmaster.Mmaster->pmpi1.R_AddressTailFIFOTagAddress    = " 0x90100028";
  vmaster.Mmaster->pmpi1.R_NbElementFIFOTagAddress    = " 0x9010002C";
  vmaster.Mmaster->pmpi1.R_NbElementMaxFIFOTag    = "9";//old
value0x90100030
  vmaster.Mmaster->pmpi1.R_FIFODataAddress      = " 0x90100034";//size
9107x32b __ big for slave
  vmaster.Mmaster->pmpi1.R_AddressHeadFIFODataAddress     = " 0x90111E34";
  vmaster.Mmaster->pmpi1.R_AddressTailFIFODataAddress     = " 0x90111E38";
  vmaster.Mmaster->pmpi1.R_NbElementFIFODataAddress     = " 0x90111E3C";
  vmaster.Mmaster->pmpi1.R_NbElementMaxFIFOData     = "18304";//0x90111E40
  vmaster.Mmaster->pmpi1.R_FIFOTagAddressSem      = " 0x90111E40";
  vmaster.Mmaster->pmpi1.R_FIFODataAddressSem       = " 0x90111E48";

  vmaster.Mmaster->pmpi1.S_FIFOTagAddress       = " 0x80100000";//size
9x32b
  vmaster.Mmaster->pmpi1.S_AddressHeadFIFOTagAddress    = " 0x80100024";
  vmaster.Mmaster->pmpi1.S_AddressTailFIFOTagAddress    = " 0x80100028";
  vmaster.Mmaster->pmpi1.S_NbElementFIFOTagAddress    = " 0x8010002C";
  vmaster.Mmaster->pmpi1.S_NbElementMaxFIFOTag    = "9";//0x00100030
  vmaster.Mmaster->pmpi1.S_FIFODataAddress      = " 0x80100034";//size
9107x32b
  vmaster.Mmaster->pmpi1.S_AddressHeadFIFODataAddress     = " 0x80111E34";
  vmaster.Mmaster->pmpi1.S_AddressTailFIFODataAddress     = " 0x80111E38";
  vmaster.Mmaster->pmpi1.S_NbElementFIFODataAddress     = " 0x80111E3C";
  vmaster.Mmaster->pmpi1.S_NbElementMaxFIFOData     = "18304";//old_value
0x00111E40
  vmaster.Mmaster->pmpi1.S_FIFOTagAddressSem      = " 0x80111E40";
  vmaster.Mmaster->pmpi1.S_FIFODataAddressSem       = " 0x80111E48";

  vmaster.Mmaster->pmpi1.ProcessorId      = "1"; //Slave1 id
  vmaster.Mmaster->pmpi1.RegExtItNumberAddress      = " 0x90116000";
  vmaster.Mmaster->pmpi1.IT_NUMBER      = "0" ;
  vmaster.Mmaster->pmpi1.IT_LEVEL       = "1" ;
```

```
  /* Master Port pmpi2 */
  vmaster.Mmaster->pmpi2.C_DATA_TYPE      = "VOID";
  vmaster.Mmaster->pmpi2.SoftPortType       = "PacketsOperations";

  vmaster.Mmaster->pmpi2.R_FIFOTagAddress       = " 0xA0100000";//size
9x32b
  vmaster.Mmaster->pmpi2.R_AddressHeadFIFOTagAddress    = " 0xA0100024";
  vmaster.Mmaster->pmpi2.R_AddressTailFIFOTagAddress    = " 0xA0100028";
  vmaster.Mmaster->pmpi2.R_NbElementFIFOTagAddress    = " 0xA010002C";
  vmaster.Mmaster->pmpi2.R_NbElementMaxFIFOTag    = "9";//0xA0100030
  vmaster.Mmaster->pmpi2.R_FIFODataAddress      = " 0xA0100034";//size
9107x32b __ big for slave
  vmaster.Mmaster->pmpi2.R_AddressHeadFIFODataAddress     = " 0xA0111E34";
  vmaster.Mmaster->pmpi2.R_AddressTailFIFODataAddress     = " 0xA0111E38";
  vmaster.Mmaster->pmpi2.R_NbElementFIFODataAddress     = " 0xA0111E3C";
  vmaster.Mmaster->pmpi2.R_NbElementMaxFIFOData     = "18304"; //0xA0111E40
  vmaster.Mmaster->pmpi2.R_FIFOTagAddressSem      = " 0xA0111E40";
  vmaster.Mmaster->pmpi2.R_FIFODataAddressSem       = " 0xA0111E48";

  vmaster.Mmaster->pmpi2.S_FIFOTagAddress       = " 0x80112000";//size
9x32b
  vmaster.Mmaster->pmpi2.S_AddressHeadFIFOTagAddress    = " 0x80112024";
  vmaster.Mmaster->pmpi2.S_AddressTailFIFOTagAddress    = " 0x80112028";
  vmaster.Mmaster->pmpi2.S_NbElementFIFOTagAddress    = " 0x8011202C";
  vmaster.Mmaster->pmpi2.S_NbElementMaxFIFOTag    = "9"; //0x00112030
  vmaster.Mmaster->pmpi2.S_FIFODataAddress      = " 0x80112034";
  vmaster.Mmaster->pmpi2.S_AddressHeadFIFODataAddress     = "
0x80123E34";//size 9107x32b
  vmaster.Mmaster->pmpi2.S_AddressTailFIFODataAddress     = " 0x80123E38";
  vmaster.Mmaster->pmpi2.S_NbElementFIFODataAddress     = " 0x80123E3C";
  vmaster.Mmaster->pmpi2.S_NbElementMaxFIFOData     = "18304"; //0x00113140
  vmaster.Mmaster->pmpi2.S_FIFOTagAddressSem      = " 0x80123E40";
  vmaster.Mmaster->pmpi2.S_FIFODataAddressSem       = " 0x80123E48";


  vmaster.Mmaster->pmpi2.ProcessorId      = "2"; //Slave2 id
  vmaster.Mmaster->pmpi2.RegExtItNumberAddress      = " 0xA0116000";
  vmaster.Mmaster->pmpi2.IT_NUMBER      = "1" ;
  vmaster.Mmaster->pmpi2.IT_LEVEL       = "1" ;

  /* Master Port pmpi3 */
    vmaster.Mmaster->pmpi3.C_DATA_TYPE      = "VOID";
  vmaster.Mmaster->pmpi3.SoftPortType       = "PacketsOperations";

  vmaster.Mmaster->pmpi3.R_FIFOTagAddress       = " 0xB0100000";//size
9x32b
  vmaster.Mmaster->pmpi3.R_AddressHeadFIFOTagAddress    = " 0xB0100024";
  vmaster.Mmaster->pmpi3.R_AddressTailFIFOTagAddress    = " 0xB0100028";
  vmaster.Mmaster->pmpi3.R_NbElementFIFOTagAddress    = " 0xB010002C";
  vmaster.Mmaster->pmpi3.R_NbElementMaxFIFOTag    = "9";  //0xB0100030
  vmaster.Mmaster->pmpi3.R_FIFODataAddress      = " 0xB0100034";//size
9107x32b __ big for slave
  vmaster.Mmaster->pmpi3.R_AddressHeadFIFODataAddress     = " 0xB0111E34";
  vmaster.Mmaster->pmpi3.R_AddressTailFIFODataAddress     = " 0xB0111E38";
  vmaster.Mmaster->pmpi3.R_NbElementFIFODataAddress     = " 0xB0111E3C";
  vmaster.Mmaster->pmpi3.R_NbElementMaxFIFOData     = "18304";//0xB0111E40
  vmaster.Mmaster->pmpi3.R_FIFOTagAddressSem      = " 0xB0111E40";
  vmaster.Mmaster->pmpi3.R_FIFODataAddressSem       = " 0xB0111E48";
```

```
  vmaster.Mmaster->pmpi3.S_FIFOTagAddress       = " 0x80124000";//size
9x32b
  vmaster.Mmaster->pmpi3.S_AddressHeadFIFOTagAddress    = " 0x80124024";
  vmaster.Mmaster->pmpi3.S_AddressTailFIFOTagAddress    = " 0x80124028";
  vmaster.Mmaster->pmpi3.S_NbElementFIFOTagAddress    = " 0x8012402C";
  vmaster.Mmaster->pmpi3.S_NbElementMaxFIFOTag    = "9";//0x00114030
  vmaster.Mmaster->pmpi3.S_FIFODataAddress      = " 0x80124034";//size
9107x32b
  vmaster.Mmaster->pmpi3.S_AddressHeadFIFODataAddress     = " 0x80135E34";
  vmaster.Mmaster->pmpi3.S_AddressTailFIFODataAddress     = " 0x80135E38";
  vmaster.Mmaster->pmpi3.S_NbElementFIFODataAddress     = " 0x80135E3C";
  vmaster.Mmaster->pmpi3.S_NbElementMaxFIFOData     = "18304";
  vmaster.Mmaster->pmpi3.S_FIFOTagAddressSem      = " 0x80135E40";
  vmaster.Mmaster->pmpi3.S_FIFODataAddressSem       = " 0x80135E48";

  vmaster.Mmaster->pmpi3.ProcessorId      = "3";//Slave3 id
  vmaster.Mmaster->pmpi3.RegExtItNumberAddress      = " 0xB0116000";
  vmaster.Mmaster->pmpi3.IT_NUMBER      = "2" ;
  vmaster.Mmaster->pmpi3.IT_LEVEL       = "1" ;


  vmaster.Mmaster->MSAP2.SoftPortType       = "GuardedRegister";


  /* @ Slave1 Module */
  vslave1.Mslave1->pmpi1.C_DATA_TYPE      = "VOID";
  vslave1.Mslave1->pmpi1.SoftPortType       = "PacketsOperations";

  vslave1.Mslave1->pmpi1.R_FIFOTagAddress       = " 0x80100000";//size
9x32b
  vslave1.Mslave1->pmpi1.R_AddressHeadFIFOTagAddress    = " 0x80100024";
  vslave1.Mslave1->pmpi1.R_AddressTailFIFOTagAddress    = " 0x80100028";
  vslave1.Mslave1->pmpi1.R_NbElementFIFOTagAddress    = " 0x8010002C";
  vslave1.Mslave1->pmpi1.R_NbElementMaxFIFOTag    = "9";//0x80100030
  vslave1.Mslave1->pmpi1.R_FIFODataAddress      = " 0x80100034";//size
9107x32b
  vslave1.Mslave1->pmpi1.R_AddressHeadFIFODataAddress     = " 0x80111E34";
  vslave1.Mslave1->pmpi1.R_AddressTailFIFODataAddress     = " 0x80111E38";
  vslave1.Mslave1->pmpi1.R_NbElementFIFODataAddress     = " 0x80111E3C";
  vslave1.Mslave1->pmpi1.R_NbElementMaxFIFOData     = "18304";
  vslave1.Mslave1->pmpi1.R_FIFOTagAddressSem      = " 0x80111E40";
  vslave1.Mslave1->pmpi1.R_FIFODataAddressSem       = " 0x80111E48";

  vslave1.Mslave1->pmpi1.S_FIFOTagAddress       = " 0x90100000";//size
9x32b
  vslave1.Mslave1->pmpi1.S_AddressHeadFIFOTagAddress    = " 0x90100024";
  vslave1.Mslave1->pmpi1.S_AddressTailFIFOTagAddress    = " 0x90100028";
  vslave1.Mslave1->pmpi1.S_NbElementFIFOTagAddress    = " 0x9010002C";
  vslave1.Mslave1->pmpi1.S_NbElementMaxFIFOTag    = "9";//0x00100030
  vslave1.Mslave1->pmpi1.S_FIFODataAddress      = " 0x90100034";//size
9107x32b __ big for slave
  vslave1.Mslave1->pmpi1.S_AddressHeadFIFODataAddress     = " 0x90111E34";
  vslave1.Mslave1->pmpi1.S_AddressTailFIFODataAddress     = " 0x90111E38";
  vslave1.Mslave1->pmpi1.S_NbElementFIFODataAddress     = " 0x90111E3C";
  vslave1.Mslave1->pmpi1.S_NbElementMaxFIFOData     = "18304";//0x00111E40
  vslave1.Mslave1->pmpi1.S_FIFOTagAddressSem      = " 0x90111E40";
  vslave1.Mslave1->pmpi1.S_FIFODataAddressSem       = " 0x90111E48";

  vslave1.Mslave1->pmpi1.ProcessorId      = "0"; //Master id
  vslave1.Mslave1->pmpi1.RegExtItNumberAddress      = " 0x80136004";
```

```
  vslave1.Mslave1->pmpi1.IT_NUMBER      = "0" ;
  vslave1.Mslave1->pmpi1.IT_LEVEL       = "1" ;

  vslave1.Mslave1->MSAP2.SoftPortType     = "GuardedRegister";

  /* @ Slave2 Module */
  vslave2.Mslave2->pmpi1.C_DATA_TYPE      = "VOID";
  vslave2.Mslave2->pmpi1.SoftPortType       = "PacketsOperations";

  vslave2.Mslave2->pmpi1.R_FIFOTagAddress       = " 0x80112000";//size
9x32b
  vslave2.Mslave2->pmpi1.R_AddressHeadFIFOTagAddress    = " 0x80112024";
  vslave2.Mslave2->pmpi1.R_AddressTailFIFOTagAddress    = " 0x80112028";
  vslave2.Mslave2->pmpi1.R_NbElementFIFOTagAddress    = " 0x8011202C";
  vslave2.Mslave2->pmpi1.R_NbElementMaxFIFOTag    = "9";//0x80112030
  vslave2.Mslave2->pmpi1.R_FIFODataAddress      = " 0x80112034";//size
9107x32b
  vslave2.Mslave2->pmpi1.R_AddressHeadFIFODataAddress     = " 0x80123E34";
  vslave2.Mslave2->pmpi1.R_AddressTailFIFODataAddress     = " 0x80123E38";
  vslave2.Mslave2->pmpi1.R_NbElementFIFODataAddress     = " 0x80123E3C";
  vslave2.Mslave2->pmpi1.R_NbElementMaxFIFOData     = "18304";//0x80113140
  vslave2.Mslave2->pmpi1.R_FIFOTagAddressSem      = " 0x80123E40";
  vslave2.Mslave2->pmpi1.R_FIFODataAddressSem       = " 0x80123E48";

  vslave2.Mslave2->pmpi1.S_FIFOTagAddress       = " 0xA0100000";//size
9x32b
  vslave2.Mslave2->pmpi1.S_AddressHeadFIFOTagAddress    = " 0xA0100024";
  vslave2.Mslave2->pmpi1.S_AddressTailFIFOTagAddress    = " 0xA0100028";
  vslave2.Mslave2->pmpi1.S_NbElementFIFOTagAddress    = " 0xA010002C";
  vslave2.Mslave2->pmpi1.S_NbElementMaxFIFOTag    = "9";//0x00100030
  vslave2.Mslave2->pmpi1.S_FIFODataAddress      = " 0xA0100034";//size
9107x32b __ big for slave
  vslave2.Mslave2->pmpi1.S_AddressHeadFIFODataAddress     = " 0xA0111E34";
  vslave2.Mslave2->pmpi1.S_AddressTailFIFODataAddress     = " 0xA0111E38";
  vslave2.Mslave2->pmpi1.S_NbElementFIFODataAddress     = " 0xA0111E3C";
  vslave2.Mslave2->pmpi1.S_NbElementMaxFIFOData     = "18304";//0x00111E40
  vslave2.Mslave2->pmpi1.S_FIFOTagAddressSem      = " 0xA0111E40";
  vslave2.Mslave2->pmpi1.S_FIFODataAddressSem       = " 0xA0111E48";

  vslave2.Mslave2->pmpi1.ProcessorId      = "0";//Master id
  vslave2.Mslave2->pmpi1.RegExtItNumberAddress      = " 0x80136008";
  vslave2.Mslave2->pmpi1.IT_NUMBER      = "1" ;
  vslave2.Mslave2->pmpi1.IT_LEVEL       = "1" ;


  vslave2.Mslave2->MSAP2.SoftPortType       = "GuardedRegister";

  /* @ Slave3 Module */
    vslave3.Mslave3->pmpi1.C_DATA_TYPE      = "VOID";
  vslave3.Mslave3->pmpi1.SoftPortType       = "PacketsOperations";

  vslave3.Mslave3->pmpi1.R_FIFOTagAddress       = " 0x80124000";//size
9x32b
  vslave3.Mslave3->pmpi1.R_AddressHeadFIFOTagAddress    = " 0x80124024";
  vslave3.Mslave3->pmpi1.R_AddressTailFIFOTagAddress    = " 0x80124028";
  vslave3.Mslave3->pmpi1.R_NbElementFIFOTagAddress    = " 0x8012402C";
  vslave3.Mslave3->pmpi1.R_NbElementMaxFIFOTag    = "9";//0x80114030
  vslave3.Mslave3->pmpi1.R_FIFODataAddress      = " 0x80124034";//size
9107x32b
  vslave3.Mslave3->pmpi1.R_AddressHeadFIFODataAddress     = " 0x80135E34";
```

```
  vslave3.Mslave3->pmpi1.R_AddressTailFIFODataAddress     = " 0x80135E38";
  vslave3.Mslave3->pmpi1.R_NbElementFIFODataAddress     = " 0x80135E3C";
  vslave3.Mslave3->pmpi1.R_NbElementMaxFIFOData     = "18304";//0x80115E40
  vslave3.Mslave3->pmpi1.R_FIFOTagAddressSem      = " 0x80135E40";
  vslave3.Mslave3->pmpi1.R_FIFODataAddressSem       = " 0x80135E48";

  vslave3.Mslave3->pmpi1.S_FIFOTagAddress       = " 0xB0100000";//size
9x32b
  vslave3.Mslave3->pmpi1.S_AddressHeadFIFOTagAddress    = " 0xB0100024";
  vslave3.Mslave3->pmpi1.S_AddressTailFIFOTagAddress    = " 0xB0100028";
  vslave3.Mslave3->pmpi1.S_NbElementFIFOTagAddress    = " 0xB010002C";
  vslave3.Mslave3->pmpi1.S_NbElementMaxFIFOTag    = "9";//0x00100030
  vslave3.Mslave3->pmpi1.S_FIFODataAddress      = " 0xB0100034";//size
9107x32b __ big for slave
  vslave3.Mslave3->pmpi1.S_AddressHeadFIFODataAddress     = " 0xB0111E34";
  vslave3.Mslave3->pmpi1.S_AddressTailFIFODataAddress     = " 0xB0111E38";
  vslave3.Mslave3->pmpi1.S_NbElementFIFODataAddress     = " 0xB0111E3C";
  vslave3.Mslave3->pmpi1.S_NbElementMaxFIFOData     = "18304";//0x00111E40
  vslave3.Mslave3->pmpi1.S_FIFOTagAddressSem      = " 0xB0111E40";
  vslave3.Mslave3->pmpi1.S_FIFODataAddressSem       = " 0xB0111E48";

  vslave3.Mslave3->pmpi1.ProcessorId      = "0"; //Master id
  vslave3.Mslave3->pmpi1.RegExtItNumberAddress      = " 0x8013600C";
  vslave3.Mslave3->pmpi1.IT_NUMBER      = "2" ;
  vslave3.Mslave3->pmpi1.IT_LEVEL       = "1" ;

  vslave3.Mslave3->MSAP2.SoftPortType       = "GuardedRegister";


  /* External Ports ---------------------------------- */
  /* @ Master Module */
  vmaster.VPmpi1.POOL_SIZE = "32";
    vmaster.VPmpi1.HardPortType = "HNDSHK";

  vmaster.VPmpi2.POOL_SIZE = "32";
    vmaster.VPmpi2.HardPortType = "HNDSHK";

  vmaster.VPmpi3.POOL_SIZE = "32";
    vmaster.VPmpi3.HardPortType = "HNDSHK";

  /* @ Slave1 Module */
  vslave1.VPmpi1.POOL_SIZE = "32";
    vslave1.VPmpi1.HardPortType = "HNDSHK";

  /* @ Slave2 Module */
  vslave2.VPmpi1.POOL_SIZE = "32";
    vslave2.VPmpi1.HardPortType = "HNDSHK";

  /* @ Slave3 Module */
  vslave3.VPmpi1.POOL_SIZE = "32";
    vslave3.VPmpi1.HardPortType = "HNDSHK";

  /*---------------------------------- Parameters for Task Port -----------
--------------------*/
  /* @ Master Module */
  vmaster.Mmaster->Master-
>Pmpi1.SoftService="HighIO/MPI/Pt2Pt/MPI_Std_Send;HighIO/MPI/Pt2Pt/MPI_Std_
Recv";
```

```
  vmaster.Mmaster->Master-
>Pmpi2.SoftService="HighIO/MPI/Pt2Pt/MPI_Std_Send;HighIO/MPI/Pt2Pt/MPI_Std_
Recv";
  vmaster.Mmaster->Master-
>Pmpi3.SoftService="HighIO/MPI/Pt2Pt/MPI_Std_Send;HighIO/MPI/Pt2Pt/MPI_Std_
Recv";

  vmaster.Mmaster->Master->Pmpi1.C_DATA_TYPE = "VOID";
  vmaster.Mmaster->Master->Pmpi2.C_DATA_TYPE = "VOID";
  vmaster.Mmaster->Master->Pmpi3.C_DATA_TYPE = "VOID";

  //vmaster.Mmaster->Master->TSAP2.SoftService="  ";

  /* @ Slave1 Module */
  vslave1.Mslave1->Slave1-
>Pmpi1.SoftService="HighIO/MPI/Pt2Pt/MPI_Std_Send;HighIO/MPI/Pt2Pt/MPI_Std_
Recv";
    vslave1.Mslave1->Slave1->Pmpi1.C_DATA_TYPE = "VOID";

  //vslave1.Mslave1->Slave1->TSAP2.SoftService="  ";


  /* @ Slave2 Module */
  vslave2.Mslave2->Slave2-
>Pmpi1.SoftService="HighIO/MPI/Pt2Pt/MPI_Std_Send;HighIO/MPI/Pt2Pt/MPI_Std_
Recv";
    vslave2.Mslave2->Slave2->Pmpi1.C_DATA_TYPE = "VOID";

  //vslave2.Mslave2->Slave2->TSAP2.SoftService="  ";

  /* @ Slave3 Module */
  vslave3.Mslave3->Slave3-
>Pmpi1.SoftService="HighIO/MPI/Pt2Pt/MPI_Std_Send;HighIO/MPI/Pt2Pt/MPI_Std_
Recv";
    vslave3.Mslave3->Slave3->Pmpi1.C_DATA_TYPE = "VOID";

  //vslave3.Mslave3->Slave3->TSAP2.SoftService="  ";

  /*---------------------------------- Parameters for Tasks --------------
-------------------------*/
  /* @ Master Module */
  vmaster.Mmaster->Stby0->setColifParam("TaskPriority","0");
    vmaster.Mmaster->Stby0->setColifParam("Source", "SystemC",
"Master/stby0.cpp", "Master/stby0.h");
    vmaster.Mmaster->Stby0->setColifParam("module_level", "DL");
    vmaster.Mmaster->Stby0->setColifParam("module_type", "software");

  vmaster.Mmaster->Stby0->setColifParam("UserStackAddress"," 0x05D00000");
  vmaster.Mmaster->Stby0->setColifParam("SvcStackAddress", " 0x05E00000");

    vmaster.Mmaster->Master->setColifParam("TaskPriority","1");
    vmaster.Mmaster->Master->setColifParam("Source", "SystemC",
"Master/task_master.cpp", "Master/task_master.h");
    vmaster.Mmaster->Master->setColifParam("module_level", "DL");
    vmaster.Mmaster->Master->setColifParam("module_type", "software");

  vmaster.Mmaster->Master->setColifParam("UserStackAddress"," 0x05C00000");
  vmaster.Mmaster->Master->setColifParam("SvcStackAddress"," 0x05F00000");

  /* @ Slave1 Module */
```

```
  vslave1.Mslave1->Stby1->setColifParam("TaskPriority","0");
    vslave1.Mslave1->Stby1->setColifParam("Source", "SystemC",
"Slave1/stby1.cpp", "Slave1/stby1.h");
    vslave1.Mslave1->Stby1->setColifParam("module_level", "DL");
    vslave1.Mslave1->Stby1->setColifParam("module_type", "software");

    vslave1.Mslave1->Stby1->setColifParam("UserStackAddress","
0x05D00000");
    vslave1.Mslave1->Stby1->setColifParam("SvcStackAddress", "
0x05E00000");

    vslave1.Mslave1->Slave1->setColifParam("TaskPriority","1");
    vslave1.Mslave1->Slave1->setColifParam("Source", "SystemC",
"Slave1/task_slave1.cpp", "Slave1/task_slave1.h");
    vslave1.Mslave1->Slave1->setColifParam("module_level", "DL");
    vslave1.Mslave1->Slave1->setColifParam("module_type", "software");

    vslave1.Mslave1->Slave1->setColifParam("UserStackAddress","
0x05C00000");
    vslave1.Mslave1->Slave1->setColifParam("SvcStackAddress","
0x05F00000");

  /* @ Slave2 Module */
    vslave2.Mslave2->Stby2->setColifParam("TaskPriority","0");
    vslave2.Mslave2->Stby2->setColifParam("Source", "SystemC",
"Slave2/stby2.cpp", "Slave2/stby2.h");
    vslave2.Mslave2->Stby2->setColifParam("module_level", "DL");
    vslave2.Mslave2->Stby2->setColifParam("module_type", "software");

    vslave2.Mslave2->Stby2->setColifParam("UserStackAddress","
0x05D00000");
    vslave2.Mslave2->Stby2->setColifParam("SvcStackAddress", "
0x05E00000");

    vslave2.Mslave2->Slave2->setColifParam("TaskPriority","1");
    vslave2.Mslave2->Slave2->setColifParam("Source", "SystemC",
"Slave2/task_slave2.cpp", "Slave2/task_slave2.h");
    vslave2.Mslave2->Slave2->setColifParam("module_level", "DL");
    vslave2.Mslave2->Slave2->setColifParam("module_type", "software");

    vslave2.Mslave2->Slave2->setColifParam("UserStackAddress","
0x05C00000");
    vslave2.Mslave2->Slave2->setColifParam("SvcStackAddress","
0x05F00000");

  /* @ Slave3 Module */
    vslave3.Mslave3->Stby3->setColifParam("TaskPriority","0");
    vslave3.Mslave3->Stby3->setColifParam("Source", "SystemC",
"Slave3/stby3.cpp", "Slave3/stby3.h");
    vslave3.Mslave3->Stby3->setColifParam("module_level", "DL");
    vslave3.Mslave3->Stby3->setColifParam("module_type", "software");

    vslave3.Mslave3->Stby3->setColifParam("UserStackAddress","
0x05D00000");
    vslave3.Mslave3->Stby3->setColifParam("SvcStackAddress", "
0x05E00000");

    vslave3.Mslave3->Slave3->setColifParam("TaskPriority","1");
    vslave3.Mslave3->Slave3->setColifParam("Source", "SystemC",
"Slave3/task_slave3.cpp","Slave3/task_slave3.h");
```

```
    vslave3.Mslave3->Slave3->setColifParam("module_level", "DL");
    vslave3.Mslave3->Slave3->setColifParam("module_type", "software");

    vslave3.Mslave3->Slave3->setColifParam("UserStackAddress","
0x05C00000");
    vslave3.Mslave3->Slave3->setColifParam("SvcStackAddress","
0x05F00000");

  /*********************************************************************
**/
  /*-------------------------------- Connecting Ports --------------------
----------*/
  /*********************************************************************
**/

    /* Highest hierarchical level */
    (*vmaster.VP1)(VC1);
    (*vmaster.VP2)(VC2);
    (*vmaster.VP3)(VC3);

    (*vslave1.VP1)(VC1);
    (*vslave2.VP1)(VC2);
    (*vslave3.VP1)(VC3);

  /* Internal level */
  vmaster.VPmpi1(VC1.mpi_channel);
  vmaster.VPmpi2(VC2.mpi_channel);
  vmaster.VPmpi3(VC3.mpi_channel);

  vslave1.VPmpi1(VC1.mpi_channel);
  vslave2.VPmpi1(VC2.mpi_channel);
  vslave3.VPmpi1(VC3.mpi_channel);

  /*********************************************************************
**/
  /*-------------------------------- Translate on COLIF ------------------
--------*/
  /*********************************************************************
**/

  sc_start(0);

    return EXIT_SUCCESS;
};
```

Fichier de code macro MPI_Send_Ressources.c.gen :

```
@
@########################################################################
@### Paramètres :
@###  - Arch_S_FIFOTagAddress :
@###  - Arch_S_AddressHeadFIFOTagAddress :
@###  - Arch_S_AddressTailFIFOTagAddress :
@###  - Arch_S_NbElementFIFOTagAddress :
@###  - Arch_S_NbElementMaxFIFOTag :
@###  - Arch_S_FIFODataAddress :
@###  - Arch_S_AddressHeadFIFODataAddress :
```

```
@###  - Arch_S_AddressTailFIFODataAddress :
@###  - Arch_S_NbElementFIFODataAddress :
@###  - Arch_S_NbElementMaxFIFOData :
@###  - Arch_ProcessorId :
@###  - Arch_RegExtItNumberAddress :
@###  - Arch_S_FIFOTagAddressSem
@###  - Arch_S_FIFODataAddressSem
@########################################################################
@#Quelques macros initiales
@
@
@# Macro pour savoir si une chaîne est dans un tableau
@DEFINE IS_NOT_IN={str,tab}
@ DEFINE ok=1 ENDDEFINE
@ FOR i FROM 0 TO (SIZEOF tab)-1 DO
@   IF (str==tab[i]) DO
@     REDEFINE ok=0 ENDDEFINE
@   ENDIF
@ ENDFOR
@ ok
@ENDDEFINE
@
@# Macro qui retourne la position d'une chaîne dans un tableau
@DEFINE GET_POS={str,tab}
@ DEFINE res=0 ENDDEFINE
@ FOR i FROM 0 TO (SIZEOF tab)-1 DO
@   IF (str==tab[i]) DO
@     REDEFINE res=i ENDDEFINE
@   ENDIF
@ ENDFOR
@ res
@ENDDEFINE
@
@######### Recuperations des paramètres et calculs #########
@
@# Table des numero d'IT
@DEFINE IT_NUMBER=[[Arch_IT_NUMBER[0]]] ENDDEFINE
@
@
@# Calcul du nombre de ressources MPI_Send
@
@DEFINE nb_ressources= SIZEOF Arch_S_FIFODataAddress ENDDEFINE
@

typedef struct
{
  volatile long int* volatile FIFO_Tag;                  //FIFO Tag
  volatile long int* volatile Address_Head_FIFO_Tag;    //Address of Head
of FIFO Tag
  volatile long int* volatile Address_Tail_FIFO_Tag;    //Address ofTail of
FIFO Tag
  volatile long int* volatile Nb_Element_FIFO_Tag;      //Number of
elements in FIFO Tag
  long int Nb_Element_Max_FIFO_Tag;                     //Size of FIFO Tag
  volatile long int* volatile FIFO_Data;                //FIFO Data
  volatile long int* volatile Address_Head_FIFO_Data;   //Address of Head
of FIFO Data
  volatile long int* volatile Address_Tail_FIFO_Data;   //Address of Tail
of FIFO Data
```

```
  volatile long int* volatile Nb_Element_FIFO_Data;     //Number of
elements in FIFO Data
  long int Nb_Element_Max_FIFO_Data;                    //Size of FIFO Data
  long int  Address_reg_IT_Number;                      //Address of reg IT
identifier
  volatile long int* FIFOTagAdSemS;          //Sender Semaphore  which
protect de FIFO Tag
  volatile long int* FIFOTagAdSemR;          //Receiver Semaphore  which
protect de FIFO Tag
  volatile long int* FIFODataAdSemS;         //Sender Semaphore  which
protect de FIFO Data
  volatile long int* FIFODataAdSemR;         //Receiver Semaphore  which
protect de FIFO Data
  short Processor_id;                                   //Receiver
processor indentifier
} MPI_Send_Ressources_type;


#ifndef __MPI_INIT_C

extern MPI_Send_Ressources_type MPI_Send_Ressources[];

#else
@
@DEFINE decl_MPI_Send_Ressources=
@ "MPI_Send_Ressources_type MPI_Send_Ressources["nb_ressources"] = \
@ {"
@ IF (nb_ressources==1) DO
@   FOR i FROM 0 TO (nb_ressources-1) DO
@   "{(long int*)"Arch_S_FIFOTagAddress[i]",\
@   (long int*)"Arch_S_AddressHeadFIFOTagAddress[i]",\
@   (long int*)"Arch_S_AddressTailFIFOTagAddress[i]",\
@   (long int*)"Arch_S_NbElementFIFOTagAddress[i]",\
@   "Arch_S_NbElementMaxFIFOTag[i]",\
@   (long int*)"Arch_S_FIFODataAddress[i]",\
@   (long int*)"Arch_S_AddressHeadFIFODataAddress[i]",\
@   (long int*)"Arch_S_AddressTailFIFODataAddress[i]",\
@   (long int*)"Arch_S_NbElementFIFODataAddress[i]",\
@   "Arch_S_NbElementMaxFIFOData[i]",\
@   "Arch_RegExtItNumberAddress[i]",\
@   (long int*)"Arch_S_FIFOTagAddressSem[i]",\
@   (long int*) ("Arch_S_FIFOTagAddressSem[i]"+0x4),\
@   (long int*)"Arch_S_FIFODataAddressSem[i]",\
@   (long int*) ("Arch_S_FIFODataAddressSem[i]"+0x4),\
@   "Arch_ProcessorId[i]"}};
@   "
@   ENDFOR
@ ELSE
@   FOR i FROM 0 TO (nb_ressources-1) DO
@   "{(long int*)"Arch_S_FIFOTagAddress[i]",\
@   (long int*)"Arch_S_AddressHeadFIFOTagAddress[i]",\
@   (long int*)"Arch_S_AddressTailFIFOTagAddress[i]",\
@   (long int*)"Arch_S_NbElementFIFOTagAddress[i]",\
@   "Arch_S_NbElementMaxFIFOTag[i]",\
@   (long int*)"Arch_S_FIFODataAddress[i]",\
@   (long int*)"Arch_S_AddressHeadFIFODataAddress[i]",\
@   (long int*)"Arch_S_AddressTailFIFODataAddress[i]",\
@   (long int*)"Arch_S_NbElementFIFODataAddress[i]",\
@   "Arch_S_NbElementMaxFIFOData[i]",\
@   "Arch_RegExtItNumberAddress[i]",\
```

```
@   (long int*)"Arch_S_FIFOTagAddressSem[i]",\
@   (long int*) ("Arch_S_FIFOTagAddressSem[i]"+0x4),\
@   (long int*)"Arch_S_FIFODataAddressSem[i]",\
@   (long int*) ("Arch_S_FIFODataAddressSem[i]"+0x4),\
@   "
@     IF (i<(nb_ressources-1)) DO
@   Arch_ProcessorId[i]"},\
@   "
@     ELSE
@   Arch_ProcessorId[i]"}};
@   "
@     ENDIF
@   ENDFOR
@ ENDIF
@ENDDEFINE
 @decl_MPI_Send_Ressources@

#endif
```

Fichier généré à partir de MPI_SendRessources.c.gen :

```
typedef struct
{
  volatile long int* volatile FIFO_Tag;                 //FIFO Tag
  volatile long int* volatile Address_Head_FIFO_Tag;    //Address of Head
of FIFO Tag
  volatile long int* volatile Address_Tail_FIFO_Tag;    //Address ofTail of
FIFO Tag
  volatile long int* volatile Nb_Element_FIFO_Tag;      //Number of
elements in FIFO Tag
  long int Nb_Element_Max_FIFO_Tag;                     //Size of FIFO Tag
  volatile long int* volatile FIFO_Data;                //FIFO Data
  volatile long int* volatile Address_Head_FIFO_Data;   //Address of Head
of FIFO Data
  volatile long int* volatile Address_Tail_FIFO_Data;   //Address of Tail
of FIFO Data
  volatile long int* volatile Nb_Element_FIFO_Data;     //Number of
elements in FIFO Data
  long int Nb_Element_Max_FIFO_Data;                    //Size of FIFO Data
  long int  Address_reg_IT_Number;                      //Address of reg IT
identifier
  volatile long int* FIFOTagAdSemS;           //Sender Semaphore  which
protect de FIFO Tag
  volatile long int* FIFOTagAdSemR;           //Receiver Semaphore  which
protect de FIFO Tag
  volatile long int* FIFODataAdSemS;          //Sender Semaphore  which
protect de FIFO Data
  volatile long int* FIFODataAdSemR;          //Receiver Semaphore  which
protect de FIFO Data
  short Processor_id;                                   //Receiver
processor indentifier
} MPI_Send_Ressources_type;


#ifndef __MPI_INIT_C

extern MPI_Send_Ressources_type MPI_Send_Ressources[];

#else
```

```
MPI_Send_Ressources_type MPI_Send_Ressources[3] = \
 {{(long int*) 0x80124000,\
  (long int*) 0x80124024,\
  (long int*) 0x80124028,\
  (long int*) 0x8012402C,\
  9,\
   (long int*) 0x80124034,\
   (long int*) 0x80135E34,\
  (long int*) 0x80135E38,\
  (long int*) 0x80135E3C,\
  18304,\
   0xB0116000,\
  (long int*) 0x80135E40,\
  (long int*) ( 0x80135E40+0x4),\
  (long int*) 0x80135E48,\
  (long int*) ( 0x80135E48+0x4),\
   3},\
   {(long int*) 0x80112000,\
  (long int*) 0x80112024,\
  (long int*) 0x80112028,\
  (long int*) 0x8011202C,\
  9,\
   (long int*) 0x80112034,\
   (long int*) 0x80123E34,\
  (long int*) 0x80123E38,\
  (long int*) 0x80123E3C,\
  18304,\
   0xA0116000,\
  (long int*) 0x80123E40,\
  (long int*) ( 0x80123E40+0x4),\
  (long int*) 0x80123E48,\
  (long int*) ( 0x80123E48+0x4),\
   2},\
   {(long int*) 0x80100000,\
  (long int*) 0x80100024,\
  (long int*) 0x80100028,\
  (long int*) 0x8010002C,\
  9,\
   (long int*) 0x80100034,\
   (long int*) 0x80111E34,\
  (long int*) 0x80111E38,\
  (long int*) 0x80111E3C,\
  18304,\
   0x90116000,\
  (long int*) 0x80111E40,\
  (long int*) ( 0x80111E40+0x4),\
  (long int*) 0x80111E48,\
  (long int*) ( 0x80111E48+0x4),\
   1}};

#endif
```

BIBLIOGRAPHIE

[BAC 02] BACIVAROV I., YOO S., JERRAYA A. A., Timed HW-SW cosimulation using native execution of OS and application SW, IEEE International High Level Design Validation and Test Workshop (HLDVT '02), Cannes, France, 27-29 October 2002 , IEEE, 2002

[BOU 04] BOUCHIMA A., YOO S., JERRAYA A. A., Fast and accurate timed execution of high level embedded software using HW/SW interface simulation model, Asia South Pacific Design Automation Conference (ASP-DAC 2004),Yokohama, Japan, January 27-30 , 2004

[BRU 00] J-Y Brunel, .et. al., COSY Communication IP's, Proc. DAC, 2000.

[CES 01b] CESARIO W., NICOLESCU G., GAUTHIER L., LYONNARD D., and JERRAYA A. Colif: a Multilevel Design Representation for Application-Specific Multiprocessor System-on-Chip Design. In Proc. Twelveth IEEE International Workshop on Rapid System Prototyping, Jun 2001.

[CHO 95] CHOU P.H., ORTEGA R.B., BORIELLO G., The Chinook Hardware/Software Co-Synthesis System, Proceedings of the 8th International Symposium on System Synthesis, pp. 22-27, September 1995

[CHI 96] CHIODO M., ENGELS D., GIUSTO P., HSIEH H., JURECSKA A., LAVAGNO L., SUZUKI K., SANGIOVANNI-VINCENTELLI A., A Case Study in Computer Aided Codesign of Embedded Controllers, Design Automation for Embedded Systems, Vol. 1, No? 1-1, pp. 51-67, January 1996

[DIJ 67] DIJKSTRA E. W. Cooperating sequential processes. Programming Languages, pages 43-112, 1967.

[DIT 01] Zubin Dittia, Integrated Hardware/Software Design of a High Performance Network Interface, Ph. D. thesis, Washington Univ. May 2001.

[ECO] Red Hat. eCos. available at http: //sources.redhat.com/ecos/

[ERN 93] ERNST R., HENKEL J., BENNER T., Hardware/Software Co-Synthesis for Microcontrollers, IEEE Design & Test of Computers, Vol. 10 No. 4, pp. 64-75, December 1993

[GAJ 95] GAJSKI D., VAHID F., Specification and Design of Embedded Hardware/Software Systems, IEEE Design Test of Computers, pp. 53-67, Spring 1995

[GAU 01] GAUTHIER L., Génération de système d'exploitation pour le ciblage de logiciel multitâche sur des architectures multiprocesseurs hétérogènes dans le cadre des systèmes embarqués spécifiques, Thèse de Doctorat INPG, Spécialité Microélectronique, laboratoire TIMA, Mai 2001

[GAU 03] GAUTHIER L., JERRAYA A.A., PAVIOT Y., Conception des Logiciels Embarqués pour les Systèmes Monopuces, Hermès, Traité EGEM, 2003.

[GUP 93] GUPTA R.K., DE MICHELLI G., Hardware/Software Cosynthesis for Digital Systems, IEEE Design & Test of Computer, Vol. 10 No. 4, pp. 29-41, December 1993

[HAR 87] HAREL, Statecharts: A Visual Formalism for Complex Systems, Science of Computer Programming, 1987, 8, p. 231-274

[HOA 85] HOARE C.A.R., Communicating Sequential Processes, 1985, Prentice Hall

[ITRS] http://public.itrs.net/Files/2003ITRS/Home2003.htm

[IEE 93] Institute of Electrical and Electronically Engineers, IEEE Standard VHDL Language Reference Manual, 1993, STD 1076-1993. IEEE

[LAH 01] K. Lahiri, .et .al, System-Level Performance Analysis for Designing On-Chip Communication Architectures", IEEE Trans. on CAD, vol. 20, no.6, pp.768-783, June 2001

[LYO 01] LYONNARD D., YOO S., BAGHDADI A., and JERRAYA A. A. Application-Specific Architecture Generation for Heterogeneous Multiprocessor System-on-Chip. In Proc. Design Automation Conf., pages 518-523, June 2001.

[MES 00] MESTDAGH D.J.G., ISAKSSON M.R., ODLING P., Zipper VDSL: A Solution for Robust Duplex Communication over Telephone Lines, IEEE Communication Magazine, pp. 90-96, May 2000.

[MOO 98] MOORBY R.P.R, DONALD E. THOMAS, The Verilog Hardware Description Language, May 1998, Hardcover

[MPI] http://www.mpi -forum.org/docs/docs.html

[M4] GNU implementation of the UNIX macro processor

[NAH 94] E. Nahum, *.et .al*, Performance Issues in Parallelized Network Protocols, Proc. of the Operating Systems Design and Implementation, pp 125–137, 1994.

[NIC 01] NICOLESCU G., YOO S., JERRAYA A., Mixed-level cosimulation for fine gradual refinement of communication in SoC design, Design Automation and Testing in Europe (DATE 2001), Munich, Germany, March 13-16, 2001

[NIC 01] NICOLESCU G., YOO S., and JERRAYA A. A. Mixed-Level Cosimulation for Fine Gradual Refinement of Communication in Soc Design. In Proc. Design Automation and Test in Europe, pages 754-759, March 2001.

[NIC 02] NICOLESCU G., Spécification et validation des systèmes hétérogènes embarqués, Thèse de doctorat, INPG, laboratoire TIMA, 2002.

[ONI 99a] O'NILS M., Specification, Synthesis and Validation of Hardware/software Interfaces, Phd thesis, Department of Electronics, Electronic System Design, Royal Institute of Technology, Electrum 229, Isafjordsgata 22-26, S-164 40 Kista, Sweden, 1999

[ONI 99b] O'NILS M., JANTSCH A., Operating System Sensitive Device Driver Synthesis from Implementation Independent Protocol Specification, In Proceeding Design Automation and Test in Europe, March 1999

[OPEN] http://www.openmp.org/

[PAV 03] PAVIOT Y., GAUTHIER L., JERRAYA A. A., "Application du flot de ciblage logiciel", chapitre 5 du livre Object Conception des logiciels embarqués pour les systèmes monopuces, Traité EGEM, série Electronique et micro-électronique, janvier 2003

[ROM 96] ROMPAEY K.V., VERKEST D., BOLSENS I., DE MAN H., CoWare, A Design Evironnement for Heterogeneous Hardware/Software Systems, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 252-257, September 1996

[SYS 00] Synopsys Inc., SystemC, avaible at http://www.systemc.org/

[SYN 02] Synopsys Eaglei, available on line at http://www.synopsys.com/products/hwsw/eagle_ds.html

PUBLICATIONS

PAVIOT Y., GAUTHIER L., JERRAYA A. A., Application du flot de ciblage logiciel, chapitre 5 du livre "Object Conception des logiciels embarqués pour les systèmes monopuces, Traité EGEM, série Electronique et micro-électronique, janvier 2003

YOUSSEF M. W., YOO S., SASONGKO A., PAVIOT Y., JERRAYA A. A., Debugging HW/SW Interface for MPSoC: Video Encoder System Design Case Study, 41th Design Automation Conference (DAC'02), San Diego USA, june 7-11, 2004

CESARIO W., PAVIOT Y., GAUTHIER L., LYONNARD D., NICOLESCU G., YOO S., JERRAYA A. A, Object-based hardware/software component interconnection model for interface design in system-on-a-chip circuits, Journal of Systems and Software, Vol. 70, No.3 , 229-244 , 2004

BAGHDADI A., CESARIO W., GAUTHIER L., LYONNARD D., NICOLESCU G., PAVIOT Y., YOO S, Application-specific multiprocessor systems-on-chip, Microelectronics Journal, Vol.33, No.11, November , 2003

YOO S., BACIVAROV I., BOUCHIMA A., PAVIOT Y., JERRAYA A. A., Building fast and accurate SW simulation models based on hardware abstraction layer and simulation environment abstraction layer, Design, Automation and Test in Europe (DATE'03), Munich, Germany, March 3-7 , 2003

CESARIO W., BAGHDADI A., GAUTHIER L., LYONNARD D., NICOLESCU G., PAVIOT Y., YOO S.,

JERRAYA A. A., DIAZ-NAVA M., Component-based design approach for multicore SoCs, 39th Design Automation Conference (DAC'02), New Orleans, USA, June 10-14, 2002 , 2002

CESARIO W., PAVIOT Y., BAGHDADI A., GAUTHIER L., LYONNARD D., NICOLESCU G., YOO S.,

JERRAYA A. A., DIAZNAVA M., HW/SW interfaces design of a VDSL modem using automatic refinement of a virtual architecture specification into a multiprocessor SoC: a case study, Design, Automation and Test in Europe (DATE'02), Paris, France, March 4-8, 2002 , 2002

CESARIO W., GAUTHIER L., LYONNARD D., NICOLESCU G., PAVIOT Y., YOO S., JERRAYA A. A.,

DIAZ-NAVA M., Multiprocessor SoC platforms: a component-based design approach, IEEE Design & Test of Computers, Vol.19, No.6, November-December , 2002

JERRAYA A.A., BAGHDADI A., CESARIO W., GAUTHIER L., LYONNARD D., NICOLESCU G., PA VIOT Y.,

YOO S., Application-specific multiprocessor systems-on-chip, The Tenth Workshop on Synthesis And System Integration of Mixed Technologies (SASIMI'01), Nara, Japan, October 18-19, 2001 , 2001

RESUME

Les technologies actuelles permettent l'intégration de systèmes de plus en plus complexes sur une seule puce. L'augmentation de la complexité induit un accroissement du temps de conception alors que paradoxalement, le concurrence économique impose des temps de mise sur le marché de plus en plus courts. Pour tenter de limiter ce décalage et d'accroître la productivité, l'industrie fait de plus en plus appel à la réutilisation de composants logiciels et matériels pour concevoir leurs systèmes.

L'un des problèmes de ce type de conception est celui de la réalisation de la communication entre composants. Cette thèse traite de la communication entre processeurs réalisée par des interfaces mixtes logicielles/matérielles. Leur difficulté de conception et l'impact du choix de partitionnement entre parties logicielles et matérielles nécessitent le développement de méthodes de génération automatique d'interfaces logicielles/matérielles pour l'exploration du partitionnement des services de communication.

La contribution de cette thèse consiste en une formalisation des problèmes liés à l'implémentation mixte logicielle/matérielle des services de communication et une proposition de flot pour la génération automatique d'interfaces. Des expériences de réalisations de primitives MPI ont permis d'appréhender le problème et de proposer un flot de génération automatique.

MOTS CLES
Systèmes mono puce, multi processeurs, interfaces logicielles/matérielles, partitionnement, génération automatique, conception à base de composants.

ABSTRACT

Current technologies enable the integration of more and more complex systems on a single chip. The increase in complexity leads to an increase of design time whereas economical competition demands to ever shorter time to market. In order to try to fill this gap and to increase productivity, reuse of HW and SW components to design systems is used more and more by industrial firms.

One problem of these design methodologies is about communication between components. The topic of this thesis is hardware/software interface implementation for communication between processors. Interface design difficulties and impact of hardware/software partitioning require automatic interfaces generation in order to enable exploration of partitioning solutions.

The contributions of this thesis are formalization of problems with regard to communication services implementation by hardware/software interface, and a suggestion of an automatic interfaces generation flow. Two experiences with MPI implementations led to understanding of the partitioning problematics and to an automatic generation flow.

KEYWORDS
System-on-Chip, multiprocessor, hardware/software interface, partitioning, automatic generation, component-based design.

ISBN (paperback) 2-84813-034-2 ; ISBN (electronic frmat) 2-84813- 035-0