



HAL
open science

Modélisation du logiciel embarqué à différents niveaux d'abstraction en vue de la validation et la synthèse des systèmes monopuces

A. Bouchhima

► **To cite this version:**

A. Bouchhima. Modélisation du logiciel embarqué à différents niveaux d'abstraction en vue de la validation et la synthèse des systèmes monopuces. Micro et nanotechnologies/Microélectronique. Institut National Polytechnique de Grenoble - INPG, 2006. Français. NNT: . tel-00086783

HAL Id: tel-00086783

<https://theses.hal.science/tel-00086783>

Submitted on 19 Jul 2006

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.

Remerciements

Au terme de ce travail, je tiens à exprimer mes remerciements envers toutes les personnes qui ont contribué, de près ou de loin, à l'accomplissement de cette thèse. Je remercie :

M. Ahmed Amine Jerraya, mon directeur de thèse qui m'a offert l'opportunité d'intégrer son équipe, pour ses précieux conseils théoriques et techniques qui m'ont été d'un grand apport dans mon travail;

Mes deux rapporteurs M. Pierre Sens, professeur à l'Université Paris 6 et M. Patrice Quinton professeur d'informatique à l'Université de Rennes 1, pour leurs analyses pertinentes;

M Rached Tourki, professeur à la Faculté de Sciences de Monastir , pour sa disponibilité et ces remarques judicieuses ;

M. Alain Clouard, directeur de recherche à ST Microelectronics pour ses remarques judicieuses portant sur des perspectives industrielles ;

M. Alain Greiner, Professeur à l'Université Pierre et Marie Curie, pour son soutien et pour avoir présidé mon jury.

A ma famille

Table des matières

Chapitre 1

Introduction.....	10
1.1.Contexte : les systèmes multiprocesseurs monopuces.....	11
1.2.Besoins : Conception conjointe des architectures logicielle et matérielle.....	11
1.3.Problème : la discontinuité des modèles de représentation d'architecture.....	12
1.4.Contributions.....	13
1.4.1.Abstraction de l'interface logicielle/matérielle.....	13
1.4.2.Modèle de cosimulation globale	14
1.4.3.Méthodologie et expérimentation.....	15
1.5.Plan du document.....	16

Chapitre 2

Les systèmes multiprocesseurs monopuces.....	17
2.1.Définition d'un système multiprocesseur monopuce (MPSoC).....	18
2.1.1.Système monopuce (SoC).....	18
2.1.1.1.Caractère « hétérogène » des SoC.....	18
2.1.1.2.Caractère « embarqué » des SoC.....	19
2.1.1.3.Caractère « spécifique » des SoC.....	20
2.1.2.Système multiprocesseur monopuce (MPSoC).....	20
2.2.Architectures des systèmes multiprocesseurs monopuces.....	21
2.2.1.Architectures matérielles.....	21
2.2.1.1.Évolution des architectures matérielles des SoC.....	21
1) <i>Les ASIC</i>	21
2) <i>Les architectures monoprocesseurs</i>	21
3) <i>Les architectures multiprocesseurs de première génération</i>	22
4) <i>Les architectures multiprocesseurs de deuxième génération</i>	23
2.2.1.2.Architectures multiprocesseurs monopuces.....	23
1) <i>Architectures hétérogènes et massivement parallèles</i>	23
2) <i>La communication : un goulet d'étranglement</i>	24
3) <i>Réseau de communication sur puce (NoC)</i>	24
2.2.2.Architectures logicielles.....	24
2.2.2.1.Vue globale : Architecture en couches.....	25
1) <i>Complexité et des applications et du matériel</i>	25
2) <i>Architecture en couches</i>	25
3) <i>Risques d'inefficacité de l'architecture en couche</i>	25
2.2.2.2.La couche abstraction du matériel	26
2.2.2.3.La couche système d'exploitation.....	27
2.3.De la spécification fonctionnelle à l'architecture RTL : la discontinuité.....	28
2.3.1.Modèles de représentation.....	29
2.3.1.1.Modèles de spécification.....	29
1) <i>Approche langage pour la spécification</i>	29
2) <i>Aspect exécutable de la spécification</i>	29
3) <i>Universalité de la spécification</i>	30
2.3.1.2.Modèles d'implémentation.....	31
1) <i>sémantique de synthèse d'un langage d'implémentation</i>	31
2) <i>Caractère modulaire d'un modèle d'implémentation</i>	31
2.3.2.Flots de conception des SoC.....	32
2.3.2.1.Flots classiques.....	32
1) <i>Flots séquentiels de conception de circuits spécifiques</i>	32
2) <i>Flots de co-conception de circuits spécifiques</i>	32
3) <i>Flots basés sur la notion de plate-forme</i>	33
2.3.2.2.Approche du groupe SLS : le flot ROSES.....	34
1) <i>Architecture virtuelle : orthogonalité comportement/communication</i>	34
2) <i>Automatisation de la génération</i>	34

3) <i>Limitations actuelles du flot ROSES</i>	35
2.3.3.Approche proposée.....	35
2.4.Conclusion.....	36
Chapitre 3	
Abstraction de l'interface logicielle/matérielle et état de l'art.....	38
3.1.Abstraction de l'interface logicielle/matérielle.....	39
3.1.1.L'entité « interface logicielle/matérielle ».....	39
3.1.1.1.Caractère distribué de l'interface logicielle/matérielle.....	40
3.1.1.2.Position dans le flot ROSES.....	41
3.1.2.Niveaux d'abstraction de l'interface de communication matérielle.....	42
3.1.2.1.Critères de classification.....	42
3.1.2.2.Niveau RTL.....	43
3.1.2.3.Niveau TLM transfert.....	44
3.1.2.4.Niveau TLM transaction.....	44
3.1.2.5.Niveau TLM message.....	45
3.1.3.Niveaux d'abstraction de l'interface de programmation (API)	46
3.1.3.1.Notion d'interface de programmation.....	46
3.1.3.2.Le niveau ISA.....	48
3.1.3.3.Le niveau HAL.....	49
3.1.3.4.Le niveau système d'exploitation OS.....	50
3.1.3.5.Tableau récapitulatif:.....	51
3.2.Validation de l'interface logicielle/matérielle.....	51
3.2.1.simulation multi-niveaux de l'interface logicielle/matérielle.....	52
3.2.2.Approches existants pour la cosimulation logicielle/matérielle.....	54
3.2.2.1.Simulation partielle des parties logicielles et matérielles.....	54
1) <i>Simulation partielle de la partie matérielle</i>	54
2) <i>Simulation partielle de la partie logicielle</i>	54
3.2.2.2.Utilisation d'un modèle de processeur.....	55
1) <i>Abstraction de la micro-architecture du processeur</i>	55
2) <i>Cosimulation avec modèle HDL du processeur</i>	56
3) <i>Cosimulation avec un simulateur du jeux d'instruction</i>	56
3.2.2.3.Approches haut niveau pour la cosimulation.....	57
1) <i>Utilisation d'un modèle de simulation du Système d'exploitation</i>	57
2) <i>L'environnement VCC de Cadence</i>	60
3.2.2.4.Estimation du temps d'exécution du logiciel.....	61
1) <i>Annotation statique</i>	61
2) <i>Estimation dynamique</i>	62
3.3.Conclusion.....	63
Chapitre 4	
Méthodologie multi-niveaux pour la cosimulation de l'interface logicielle/matérielle.....	64
4.1.Méthodologie proposée.....	65
4.1.1.Vue globale des étapes du flot.....	65
4.1.2.Architecture virtuelle.....	67
4.1.3.Prototype virtuel.....	67
4.1.4.Micro-architecture.....	68
4.2.Exécution native du logiciel embarqué dans un simulateur matériel à événements discrets.....	70
4.2.1.Problèmes liés à l'exécution native du logiciel.....	71
4.2.1.1.L'exécution native comme solution naturelle aux niveaux d'abstraction considérés.....	71
4.2.1.2.Exécution native dans SystemC : motivation.....	72
4.2.2.L'environnement SystemC.....	73
4.2.2.1.Eléments structurels d'une description SystemC.....	73
4.2.2.2.Eléments comportementaux d'une description SystemC: le modèle à événements discrets.....	73
4.2.3.Estimation du temps d'exécution du logiciel.....	76
4.2.3.1.Dépendance au flot de contrôle: découpage en blocs de base.....	76
4.2.3.2.Dépendance au compilateur.....	78
4.2.3.3.Dépendance à l'architecture matérielle.....	78

4.2.4.Synchronisation entre le logiciel et le matériel.....	80
4.2.4.1.Relation entre synchronisation et annotation du code logiciel.....	81
4.2.4.2.Auto-synchronisation sur les opérations de contrôle.....	81
4.2.4.3.Points de synchronisation dans un bloc de calcul.....	82
1) <i>Notations et définitions</i>	82
2) <i>Points de synchronisation dans un bloc de calcul</i>	83
4.3.Modèles de simulation de l'interface logicielle/matérielle.....	86
4.3.1.Modèle de l'architecture virtuelle.....	86
4.3.1.1.L'unité des ressources.....	88
1) <i>les éléments de traitement (Processing Element PE) :</i>	88
2) <i>l'espace mémoire partagée :</i>	88
3) <i>les canaux de contrôle :</i>	88
4) <i>les canaux de données :</i>	89
5) <i>les ressources locales spécifiques :</i>	89
4.3.1.2.Le noyau (Kernel).....	89
1) <i>module ordonnanceur : modèle de l'ordonnancement hiérarchique</i>	90
2) <i>module synchronisation</i>	91
3) <i>modules communication inter-tâches</i>	91
4.3.1.3.L'unité d'entrées/sorties.....	91
4.3.1.1.Le méta-modèle.....	92
4.3.2.Modèle du prototype virtuel.....	93
4.3.2.1.Unité d'exécution.....	95
4.3.2.2.Unité de données.....	95
4.3.2.3.Unité de synchronisation.....	96
4.3.2.4.Unité d'accès.....	97
4.3.2.5.Le méta-modèle.....	97
4.3.3.Algorithme de la fonction Synch().....	98
4.4.Conclusion.....	100
Chapitre 5	
Implémentation, outils et applications.....	101
5.1.Implémentation des modèles de simulation sous SystemC.....	102
5.1.1.Structure des bibliothèques de simulation.....	102
5.1.1.1.Conception des bibliothèques de simulation.....	102
5.1.1.2.L'environnement d'exécution.....	104
5.1.2.L'outil d'annotation semi-automatique du code.....	105
5.1.3.L'environnement MP-SIM.....	107
5.2.Exemple d'illustration	110
5.2.1.Spécification fonctionnelle	110
5.2.2.L'étape de partitionnement : l'architecture virtuelle.....	111
5.2.2.1.Premier scénario : utiliser le même code SystemC pour les tâches logicielles.....	111
5.2.2.2.Deuxième scénario : raffinement de l'application pour une API donnée de système d'exploitation... 114	
5.2.3.Raffinement de l'architecture globale : le prototype virtuel.....	115
5.3.Applications.....	118
5.3.1.Application VDSL.....	118
5.3.2.Application encodeur MPEG4.....	120
5.4.Conclusion.....	123
Chapitre 6	
Vers un modèle de raffinement de l'interface logicielle/matérielle.....	125
6.1.Raffinement de l'interface logicielle/matérielle : le modèle à base de composant/service.....	126
6.1.1.Raffinement de l'interface logicielle/matérielle : formulation du problème.....	126
6.1.2.Le modèle à base de composant/service.....	127
6.1.2.1.Pourquoi un modèle à base de composants ?.....	128
1) <i>Modèle générique</i>	128
2) <i>Modèle favorisant la réutilisation</i>	128
3) <i>Modèle se prêtant à l'automatisation</i>	128
4) <i>Limites du modèle à base de composants :</i>	128

6.1.2.2. Notion de composant/service.....	129
1) Définition d'un composant élémentaire.....	129
2) Composant hiérarchique	130
3) Exemples de composants.....	130
6.1.2.3. Graphe de dépendance de services (SDG)	131
1) Définition d'un graphe de dépendance de services.....	132
2) Notion de bibliothèque de composants.....	133
3) La bibliothèque comme graphe maximal.....	133
6.1.2.4. Modèle de l'architecture: résolution du graphe de dépendance.....	134
6.2. Flot multi-niveaux de génération d'architectures MPSoC.....	136
6.2.1. Aperçu global du flot de génération	136
6.2.2. Raffinement de l'architecture virtuelle.....	137
6.2.3. Raffinement du prototype virtuel.....	137
6.3. Conclusion.....	138
Chapitre 7	
Conclusion et perspectives.....	139

Liste des figures

Figure 1.1: Modèles de cosimulation globale.....	15
Figure 2.1: Environnement d'un système sus puce.....	19
Figure 2.2: Architecture monoprocesseur.....	22
Figure 2.3: Architecture multiprocesseur de première génération.....	22
Figure 2.4: Architecture à réseau sur puce.....	24
Figure 2.5: Les différentes couches de la pile logicielle.....	26
Figure 2.6: pseudo parallélisme de trois tâches logicielles sur un seul processeur.....	27
Figure 2.7: modèle générique d'ordonnancement d'une tâche logicielle.....	28
Figure 2.8: Les étapes et modèles d'un flot de conception classique.....	34
Figure 2.9: flot ROSES et notion d'architecture virtuelle.....	35
Figure 2.10: L'approche proposée dans le contexte du flot ROSES.....	36
Figure 3.1: l'interface logicielle/matérielle dans le contexte MPSoC.....	40
Figure 3.2: Position de l'interface logicielle/matérielle dans l'architecture virtuelle.....	41
Figure 3.3: Exemple de signaux formant un port logique.....	43
Figure 3.4: services de communication bloquants au niveau TLM transaction du protocole AHB.....	45
Figure 3.5: services de communication non bloquants au niveau TLM transaction du protocole AHB.....	45
Figure 3.6: Services de communication au niveau TLM message	46
Figure 3.7: notion d'interface de programmation.....	47
Figure 3.8: Niveaux d'abstraction considérés du logiciel embarqué.....	48
Figure 3.9: Niveaux classiques de validation de l'interface logicielle/matérielle.....	53
Figure 3.10: Flot de génération à partir d'un ADL.....	56
Figure 3.11: Principe de la cosimulation à base de ISS.....	57
Figure 3.12: Trace d'exécution d'un système de trois tâches indépendantes (a) sur un processeur réel avec un système d'exploitation (b) dans un modèle de simulation purement fonctionnel.....	58
Figure 3.13: Correspondance fonction/architecture dans VCC.....	60
Figure 3.14 : Les deux techniques d'instrumentation statique.....	62
Figure 3.15 : Principe de l'instrumentation dynamique.....	63
Figure 4.1: Flot générique de l'utilisation de l'approche proposée.....	66
Figure 4.2: Exemple d'architecture virtuelle.....	67
Figure 4.3: Exemple de prototype virtuel.....	68
Figure 4.4: Exemple de micro-architecture.....	69
Figure 4.5: Le processus SystemC comme transformation sur des signaux.....	74
Figure 4.6: représentation d'un signal en SystemC.....	75
Figure 4.7: Exemple de flot de contrôle dans le logiciel et décomposition en blocs de base.....	77
Figure 4.8: résultat de l'annotation temporelle au niveau code source.....	78
Figure 4.9: séquençement des opérations de contrôle et de calcul.....	82
Figure 4.10: Transition d'état dans un bloc de calcul.....	84
Figure 4.11: Principe de l'exécution native d'un bloc de calcul comparée à une exécution interprétée.....	85
Figure 4.12: Cas d'une interruption matérielle.....	85
Figure 4.13: Modèle conceptuel d'un noeud logiciel de l'architecture virtuelle.....	87
Figure 4.14: Modèle de l'ordonnancement hiérarchique.....	91
Figure 4.15: Exemples d'unités d'entrées/sorties.....	92
Figure 4.16: Méta-modèle du noeud logiciel de l'architecture virtuelle.....	93
Figure 4.17: Modèle conceptuel de l'interface logicielle/matérielle dans un prototype virtuel.....	94
Figure 4.18: Méta-modèle d'un noeud logiciel du prototype virtuel.....	98
Figure 4.19: Implémentation de la fonction synch().....	99
Figure 4.20: Modélisation du conflit d'accès sur le bus.....	100
Figure 5.1: diagramme de classe de la bibliothèque de simulation d'OS.....	103
Figure 5.2: Plan mémoire du processus de simulation	105
Figure 5.3: outil d'annotation automatique du code source.....	106
Figure 5.4: exemple de flot de contrôle CFG obtenu pour une « petite » partie de code de la figure	107

Figure 5.5: MP-SIM comme front-end pour SystemC.....	108
Figure 5.6: Aperçu sur l'environnement MP-SIM.....	109
Figure 5.7: Renforcement du pouvoir de détection d'erreurs dans la simulation native.....	109
Figure 5.8: Spécification fonctionnelle de l'exemple.....	110
Figure 5.9: trace de la simulation fonctionnelle.....	111
Figure 5.10: exemple d'utilisation correspondant au premier scénario.....	112
Figure 5.11: exemple d'utilisation correspondant au premier scénario.....	112
Figure 5.12: trace de la simulation avec modèle d'OS.....	113
Figure 5.13: phénomène d'inversion de priorité.....	113
Figure 5.14: exemple d'utilisation correspondant au deuxième scénario.....	114
Figure 5.15: spécification du module SystemC "VXWORKS_MODULE"	115
Figure 5.16: exemple de raffinement du code applicatif pour vxworks.....	115
Figure 5.17: Architecture simplifiée de la plateforme ARM Integrator.....	116
Figure 5.18: Fichier de description de l'architecture locale dans prototype virtuel.....	117
Figure 5.19: Partie du code du pilote de périphérique DRV_FIFO	118
Figure 5.20: Différents niveaux de modélisation de l'application VDSL.....	119
Figure 5.21: Spécification fonctionnelle de l'application DivX.....	120
Figure 5.22: spécification d'un module encodeur.....	121
Figure 5.23: Modèle de simulation au niveau HAL de l'application MPEG4.....	121
Figure 5.24: Modèle de simulation au niveau prototype virtuel de l'application MPEG4.....	122
Figure 6.1 : raffinement de l'interface logicielle/matérielle.....	127
Figure 6.2 : modèle d'un composant élémentaire.....	130
Figure 6.3: Exemple de graphe de dépendance de services.....	133
Figure 6.4: Exemple d'une architecture à base de composant/service.....	134
Figure 6.5: Flot de génération d'architectures MPSoC à différents niveaux d'abstraction.....	136

Liste des tables

Tableau 1: niveaux d'abstraction de l'interface de communication.....	46
Tableau 2: Exemples de services au niveau HAL.....	50
Tableau 3: Tableau récapitulatif des niveaux d'abstraction de l'interface de programmation.....	51
Tableau 4: Comparaison des trois types de modèles : architecture virtuelle, prototype virtuel et micro-architecture	70

Chapitre 1

Introduction

Sommaire

Chapitre 1	
Introduction.....	10
1.1.Contexte : les systèmes multiprocesseurs monopuces.....	11
1.2.Besoins : Conception conjointe des architectures logicielle et matérielle.....	11
1.3.Problème : la discontinuité des modèles de représentation d'architecture.....	12
1.4.Contributions.....	13
1.4.1.Abstraction de l'interface logicielle/matérielle.....	13
1.4.2.Modèle de cosimulation globale	14
1.4.3.Méthodologie et expérimentation.....	15
1.5.Plan du document.....	16

1.1. Contexte : les systèmes multiprocesseurs monopuces

90% des systèmes embarqués monopuces en technologie 130 nm incluent au moins un processeur. Actuellement, on parle de plus en plus de systèmes multiprocesseurs monopuces (MPSoC) pour désigner des systèmes électroniques embarquant plusieurs processeurs sur une même puce de silicium. Cette quête de programmabilité dans les systèmes embarqués est expliquée par la flexibilité qu'apporte la solution logicielle comparée à une solution purement matérielle représentée par les ASIC traditionnels. Cependant, à la différence des systèmes multiprocesseurs à usage général, la conception des systèmes MPSoC est naturellement confrontée à des contraintes strictes de performances et de coûts inhérentes à tout système embarqué. Ainsi, l'exploration et la validation des choix architecturaux, liés à la fois à la conception de la plateforme matérielle et au logiciel embarqué, s'avèrent très importantes afin d'atteindre un compromis performance/coût judicieux. Cette exploration/validation est d'autant plus critique qu'elle se situe dans un contexte étroit de temps de mise sur le marché. Aujourd'hui, le coût d'un tel processus est estimé à plus de 60% du coût total de développement des systèmes monopuces. En particulier, la validation du logiciel embarqué occupe une part importante de ces coûts et constitue désormais un goulet d'étranglement dans le flot de conception des SoC.

En analysant les flots de conception actuels des systèmes monopuces, les causes d'un tel coût de développement peuvent être ramenées à l'intégration tardive des parties logicielle et matérielle d'un système MPSoC. En effet, partant d'une spécification initiale de l'application, les flots actuels séparent la conception du système en deux parties complètement autonomes : une dédiée à l'architecture matérielle et l'autre au logiciel embarqué qui va s'exécuter sur cette architecture. Bien qu'un tel découpage initial s'appuie généralement sur des heuristiques de performance (faisant souvent appel au bon sens du concepteur), la séparation prématurée entre le logiciel et le matériel cache souvent plusieurs choix d'implémentation qui ne peuvent être évalués que pendant la phase finale d'intégration. A ce stade, il faut considérer au moins deux handicaps majeurs. Le premier est lié à la capacité limitée des environnements de validation bas niveaux en terme de vitesse de simulation. Le deuxième concerne la complexité du débogage logiciel dans ces environnements. Il en résulte un long et fastidieux cycle de conception qui rend la validation et l'exploration des choix architecturaux initiaux très coûteuses.

1.2. Besoins : Conception conjointe des architectures logicielle et matérielle

Pour remédier aux problèmes liés à l'intégration tardive des architectures logicielle et matérielle

d'un système MPSoC, un modèle unifié permettant la représentation conjointe à différents niveaux d'abstraction de ces deux types d'architectures est nécessaire. Ce modèle doit faciliter la conception graduelle des architectures MPSoC tout en permettant, à chaque niveau d'abstraction, la validation et l'évaluation des performances qui découlent d'une telle architecture. Précisons qu'il s'agit d'un modèle **d'implémentation** qui représente une architecture mixte logicielle/matérielle. Fort de ce contexte unifié de modélisation, la conception du logiciel embarqué et de l'architecture sous-jacente pourra alors se dérouler en parallèle et d'une manière interactive.

1.3. Problème : la discontinuité des modèles de représentation d'architecture

Les modèles existants de représentation d'architecture se divisent en deux grandes catégories : les modèles globaux qui décrivent l'architecture logicielle/matérielle dans sa totalité et les modèles partiels focalisant sur un seul type d'architecture qu'elle soit matérielle ou logicielle.

Alors que le deuxième type de modèles a connu un essor important en terme d'études visant à définir les différents niveaux d'abstraction relatifs à chaque modèle, les modèles globaux n'ont pas connu le même développement et sont restés à un niveau d'abstraction relativement bas.

Cette discontinuité, traduite par la limite des modèles de représentation globaux d'architecture, se répercute directement au niveau des environnements de simulation et des méthodologies de conception des systèmes MPSoC.

➤ Limites des modèles architecturaux classiques:

Un modèle global d'une architecture logicielle/matérielle est conventionnellement décrit au niveau RTL/ISA. A ce niveau, le logiciel n'est autre qu'une suite d'instructions binaires placée dans une zone mémoire. Le matériel est décrit en utilisant un langage de description de matériel (*HDL*). Ceci inclut l'architecture locale du noeud logiciel (processeur(s), mémoire(s), périphériques, etc.) mais aussi les autres parties du système. A ce niveau, le processeur est considéré comme l'interface ultime entre le logiciel et le matériel. Il fournit d'un côté au programmeur une vision au niveau *ISA* de la machine. De l'autre côté, il interagit avec le reste des composants de l'architecture matérielle via des signaux physiques (bus d'adresse, bus de données, signaux de contrôle, signaux d'interruption, etc.). Cette vision de l'architecture n'est donc valable qu'une fois les deux parties logicielle et matérielle entièrement conçues, c'est à dire vers la fin du cycle de conception.

➤ Limites des environnements de cosimulation existants:

Les environnements de cosimulation conventionnels implémentent le modèle RTL/ISA décrit

précédemment. Ces environnements permettent d'effectuer une validation globale du système lors de l'étape d'intégration. Ceci inclut aussi bien la validation des performances du matériel en présence de la vraie application, mais aussi la validation et le débogage du logiciel dans le contexte du système entier. Vu le niveau d'abstraction employé, la vitesse de simulation reste très réduite et constitue ainsi une barrière empêchant l'exploration et la validation des applications les plus exigeantes.

➤ Absence de méthodologies de raffinement graduel:

Une conséquence directe de la discontinuité des modèles de représentation de l'architecture logicielle/matérielle est l'absence de méthodologies et d'outils permettant une transition non brutale de la spécification initiale à l'architecture finale. Comme nous l'avons mentionné précédemment, cette transition se fait généralement en une seule étape, pendant laquelle, architectures logicielle et matérielle sont raffinées séparément.

1.4. Contributions

Durant cette thèse, nous avons essayé d'apporter des éléments de réponse aux problèmes soulevés dans les paragraphes précédents. Plus précisément, nous nous sommes intéressés au problème de l'abstraction de l'interface logicielle/matérielle. Ce travail d'abstraction nous a permis d'identifier les éléments clés d'une architecture logicielle/matérielle à un niveau d'abstraction donné, et de simplifier la représentation de cette architecture en excluant les éléments jugés non importants à ce niveau.

1.4.1. Abstraction de l'interface logicielle/matérielle

La notion d'interface logicielle/matérielle est un concept clé qui a constitué le fil conducteur à travers l'ensemble des travaux menés dans le cadre de cette thèse. Par interface logicielle/matérielle nous sous-entendons l'entité abstraite qui joue le rôle d'interface entre le logiciel, tel que vu par le programmeur, et le matériel tel que vu par l'architecte. Ainsi définie, l'interface logicielle/matérielle est une entité complexe. Elle est d'autant plus complexe qu'elle se situe dans un contexte MPSoC marqué par un fort degré de complicité et d'interactions entre les parties logicielles et matérielles.

Une telle complexité fait d'ailleurs le quotidien des concepteurs de ces systèmes. Le programmeur du logiciel embarqué est confronté, dès les premiers stades de conception, à des choix qui dépendent de l'architecture matérielle sous-jacente. Le concepteur de l'architecture matérielle, quant à lui, doit tenir compte des caractéristiques de l'application logicielle afin de garantir le niveau de performance souhaité. Malheureusement, cette interdépendance étroite entre les deux parties n'est

traduite, au niveau des modèles de représentation architecturale mis à la disposition des deux types de concepteurs, que tardivement dans le cycle de conception et à un niveau d'abstraction très bas.

L'existence d'un modèle abstrait de l'interface logicielle/matérielle permet de tenir compte de cette complexité à différents niveaux d'abstraction correspondant à différentes étapes d'un flot de conception de systèmes MPSoC. Cette interface constitue ainsi un contexte commun permettant la conception conjointe de l'architecture logicielle/matérielle du système monopuce. Tenir compte de la complexité d'un système donné à différents niveaux implique un travail d'abstraction, donc de modélisation. Ainsi, la première tâche était d'identifier les niveaux d'abstraction pertinents qui correspondent naturellement à des « paliers » dans un flot de conception MPSoC, et de définir, à chaque niveau, le modèle conceptuel de l'interface logicielle/matérielle correspondant. Le modèle doit, d'une part, explicitement refléter les caractéristiques architecturales importantes au niveau d'abstraction considéré et de l'autre part, cacher ou encore faire abstraction des détails qui ne le sont pas à ce niveau.

1.4.2. Modèle de cosimulation globale

Pour être exploitables dans un contexte de validation et d'exploration d'architecture, les modèles conceptuels de l'interface logicielle/matérielle doivent être associés à des modèles de simulation. Ces derniers définissent la sémantique d'exécution de l'interface logicielle/matérielle dans un contexte de cosimulation globale impliquant aussi bien les parties logicielles que matérielles raffinées au niveau d'abstraction considéré. Comme contexte de cosimulation globale, nous considérons l'environnement SystemC qui offre des atouts importants en terme de flexibilité et de performance. Les modèles de simulation proposés de l'interface logicielle/matérielle s'intègrent dans cet environnement, sans avoir besoin de modifier le moteur de simulation interne de celui-ci. Ceci conserve la sémantique habituelle de SystemC et favorise la réutilisation des composants déjà existants.

La figure 1.1 propose une vue globale qui positionne les modèles de simulation proposés dans le cadre de cette thèse par rapport au modèle conventionnel utilisé pour la cosimulation logicielle/matérielle. Le modèle classique (figure 1.1-a) considère que l'architecture du système est complètement raffinée et connue dans ces moindres détails. Ainsi le logiciel embarqué doit être entièrement développé (incluant les couches basses) avant d'être compilé pour le (les) processeur(s) cible(s). L'image binaire obtenue est alors prise en charge par un (des) simulateur(s) de processeurs qui interprètent séquentiellement les instructions et interagissent avec un modèle entièrement raffiné de l'architecture matérielle.

Contrairement à ce modèle de simulation de bas niveau, l'environnement de cosimulation proposé

(figure 1.1-b) permet de représenter l'architecture logicielle/matérielle à différents niveaux d'abstraction. Pour chaque niveau d'abstraction, le logiciel est considéré (raffiné) jusqu'à une API (*Application Programmer Interface*) qui représente la vision avec laquelle le programmeur perçoit « la machine » sous-jacente à ce niveau. Cette API dépend bien évidemment du niveau d'abstraction considéré. Au niveau d'abstraction le plus bas, elle est confondue avec l'ensemble du jeu d'instruction du processeur. Le modèle de simulation de l'interface logicielle/matérielle se réduit alors au simulateur du processeur et on retrouve le cas classique de cosimulation logicielle/matérielle.

Parallèlement à cette abstraction de l'interface logicielle, l'architecture matérielle peut être décrite avec plus ou moins de détails. Des services de communication abstraits généralisent la notion de signaux physiques du modèle conventionnel.

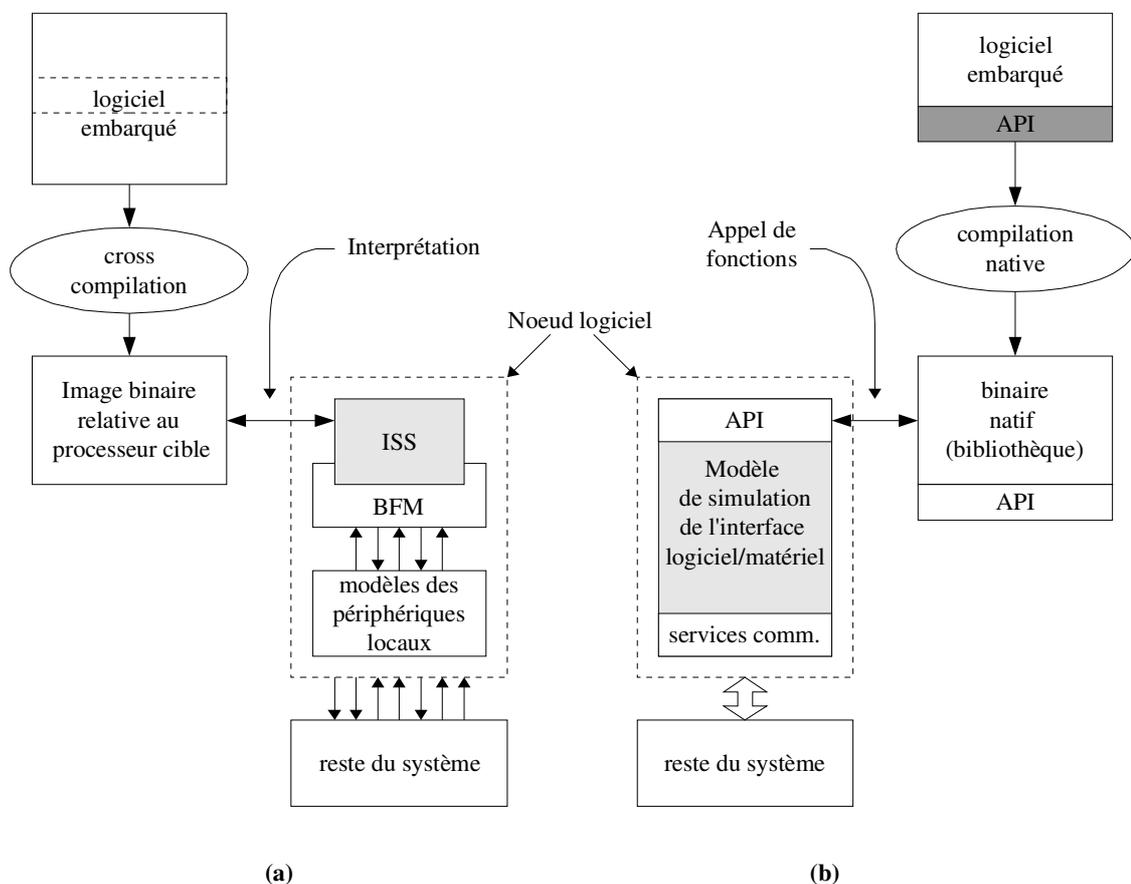


Figure 1.1: Modèles de cosimulation globale

1.4.3. Méthodologie et expérimentation

L'intégration des différents modèles proposés dans un flot de conception automatique de système MPSoC, tel que celui proposé par ROSES [ces03], passe par la définition des étapes à considérer dans le flot ainsi que par la spécification des outils de transformation. Bien que cette thèse n'apporte pas une solution complète à ce sujet, le problème de l'automatisation du flot de conception, en tenant

compte des concepts développés tout au long de la thèse, a été abordé. La méthodologie proposée se base sur un formalisme (en cours de développement au sein de l'équipe) basé sur la notion de composant/service. Selon ce formalisme, un processus de raffinement ou encore de génération d'architecture peut être ramené à la composition d'éléments de base selon un schéma bien déterminé décrivant la dépendance en termes de services fournis et requis par ces composants. L'intérêt majeur de ce formalisme, en plus de l'automatisation de la génération, est le fait qu'il peut s'appliquer aussi bien pour générer des architectures matérielles que logicielles. Il permet ainsi une grande flexibilité au niveau du raffinement de l'interface logicielle/matérielle et favorise l'exploration de l'espace des solutions architecturales envisageables à ce niveau.

1.5. Plan du document

Ce document est organisé en sept chapitres, le premier étant cette introduction. Le deuxième chapitre est dédié à la description du contexte sous-jacent à cette thèse, à savoir les systèmes multiprocesseurs monopuces. Les architectures logicielle et matérielle de ces systèmes sont analysées et les flots classiques de conception sont présentés. Le troisième chapitre introduit la notion d'interface abstraite logicielle/matérielle et se focalise sur l'état de l'art des modèles de représentation et de validation des architectures logicielles/matérielles. Le chapitre 4 présente deux modèles conceptuels de l'interface logicielle/matérielle correspondant à deux niveaux d'abstraction intermédiaires. Une sémantique d'exécution de ces modèles conceptuels est également proposée dans le cadre de l'environnement SystemC. Le chapitre 5 détaille quelques aspects d'implémentation relatifs aux modèles de simulation proposés et présente les résultats expérimentaux obtenus sur deux exemples d'application. Le chapitre 6 aborde le problème de la génération d'architecture et propose une formalisation de ce problème dans une perspective de raffinement de l'interface logicielle/matérielle. Finalement, le chapitre 7 conclut ce document et propose quelques perspectives potentielles à ce travail.

Chapitre 2

Les systèmes multiprocesseurs monopuces

Sommaire

Chapitre 2	
Les systèmes multiprocesseurs monopuces.....	17
2.1.Définition d'un système multiprocesseur monopuce (MPSoC).....	18
2.1.1.Système monopuce (SoC).....	18
2.1.2.Système multiprocesseur monopuce (MPSoC).....	20
2.2.Architectures des systèmes multiprocesseurs monopuces.....	21
2.2.1.Architectures matérielles.....	21
2.2.2.Architectures logicielles.....	24
2.3.De la spécification fonctionnelle à l'architecture RTL : la discontinuité.....	28
2.3.1.Modèles de représentation.....	29
2.3.2.Flots de conception des SoC.....	32
2.3.3.Approche proposée.....	36
2.4.Conclusion.....	37

2.1. Définition d'un système multiprocesseur monopuce (MPSoC)

Cette section a pour but de définir le contexte sous-jacent aux systèmes considérés dans le cadre de cette thèse. Les systèmes multiprocesseurs monopuces sont définis comme un sous ensemble d'une famille plus large : les systèmes monopuces. Ainsi, nous commençons par définir ce qu'est un système monopuce, ces caractéristiques et ces contraintes, avant de s'intéresser plus particulièrement au cas des systèmes multiprocesseurs.

2.1.1. Système monopuce (SoC)

Un système fait référence à un assemblage d'éléments qui se coordonnent pour concourir à un résultat (encyclopédie). En grec « sustêma » signifie ensemble. Ce mot provient du verbe « synistanai » qui veut dire combiner, établir, rassembler.

Les systèmes monopuces (SoC de l'anglais *System on Chip*) sont des systèmes électroniques composés par assemblage d'éléments complexes en eux-mêmes (processeurs, mémoires, DSP, blocs analogiques), le tout intégré sur une seule puce de silicium. La capacité d'intégrer un nombre important de composants sur une même puce est dû au succès qu'a connu (et connaît encore grâce à M. Moore !) la technologie en terme de miniaturisation et de maîtrise du procédé de gravure sub-micronique [Jer04].

Dans cette définition, le caractère « électronique » qui qualifie les SoCs n'est pas strictement précis. Aujourd'hui en effet, les systèmes monopuces peuvent contenir, en plus des parties électroniques, des parties non-électroniques. C'est le cas, par exemple, des systèmes intégrant des éléments micro-mécaniques (MEMS de l'anglais *Micro-Electro-Mechanical Systems*). Ceci dit, l'électron reste incontestablement la vedette des SoCs contemporains.

2.1.1.1. Caractère « hétérogène » des SoC

En plus du fait qu'il peut intégrer des parties non-électroniques, un SoC regroupe généralement – au sein de sa partie électronique – des composants de différentes natures, ce qui en fait un système hétérogène.

Une première classification se fait entre parties analogiques et parties numériques. Un exemple quotidien de tels systèmes, dits hybrides, est le téléphone portable, où la partie radio (analogique) et la partie traitement de signal et gestion de l'interface utilisateur (numérique) cohabitent souvent sur la même puce.

Si l'on s'intéresse à la partie numérique, celle-ci peut elle-même être décomposée selon qu'il s'agit d'un traitement effectué par un programme logiciel tournant sur un (ou plusieurs) processeur(s), ou d'une fonction directement « câblée » en matériel. Par abus de langage, le long de ce manuscrit, nous parlons respectivement de partie logicielle et de partie matérielle.

2.1.1.2. Caractère « embarqué » des SoC

Par définition, un système embarqué est un sous-système intégré (ou encore enfoui) dans un système plus large avec lequel il est interfacé, et pour lequel il réalise des fonctions particulières. Les systèmes monopuces sont aussi des systèmes embarqués (la réciproque ne tient pas forcément) dans le sens où ils sont toujours appelés à évoluer dans le contexte d'un système plus large qui constitue l'environnement du système monopuce. La figure 2.1 représente un exemple d'un tel environnement. Comme l'environnement est généralement physique, l'interaction entre le SoC (numérique) et les éléments de l'environnement (capteurs et actionneurs) se fait via des convertisseurs analogique/numérique (CAN) et vice versa (CNA).

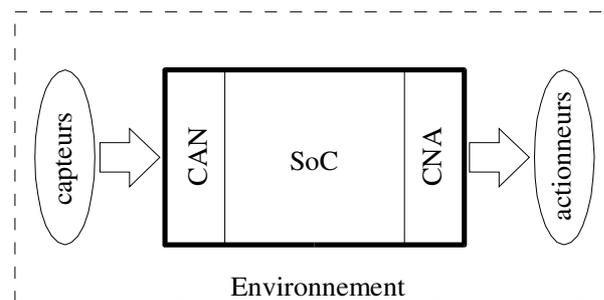


Figure 2.1: Environnement d'un système sur puce

Par ailleurs, un SoC hérite les contraintes de son environnement. Ainsi, dans un environnement réactif où le système doit répondre « correctement » à des événements externes, le SoC doit avoir un caractère « temps réel ». Le degré de tolérance d'un éventuel dysfonctionnement du système (par exemple le dépassement d'une échéance) dépend encore une fois de la caractéristique de l'environnement. Dans un contexte critique (aviation, militaire, etc.), un tel dysfonctionnement est non toléré, on parle alors d'un système temps réel strict (*hard real-time*). Par contre, dans un contexte non critique (applications multimédia pour grand public), les exigences sont moins strictes et les systèmes sont souvent qualifiés de temps réel mous (*soft real-time*).

D'autres caractéristiques de l'environnement peuvent également imposer des contraintes supplémentaires liées par exemple à la consommation d'énergie (système basse consommation) ou à la sécurité (environnements critiques à haut risque).

2.1.1.3. Caractère « spécifique » des SoC

Un système monopuce est, par définition même, différent des systèmes à usages générale tel que les ordinateurs. Par opposition à ces systèmes à usage général, un SoC cible toujours une application particulière bien définie. On ne cherche donc pas qu'un SoC soit polyvalent mais plutôt d'être spécifique. Cette distinction implique des différences radicales au niveau des méthodologies de conception entre un système monopuce est un système à usage général [Jer04].

Notons cependant qu'une telle séparation, si elle est franche dans une grande partie d'applications, l'est beaucoup moins dans certains autres types d'application qui héritent à la fois des caractéristiques des deux « univers », tels que les assistants personnels (PDA), etc.

Il est important également de souligner que même si le caractère spécifique des SoC signifie qu'ils sont « taillés sur mesure », cela n'implique pas forcément qu'ils sont conçus à chaque fois en partant de zéro. En effet, les méthodologies de conception des SoC (cf section 1.3.2) font souvent appel à la réutilisation qui ne contredit pas le caractère spécifique des systèmes en question.

2.1.2. Système multiprocesseur monopuce (MPSoC)

Comme son nom le suggère, un système multiprocesseur monopuce (MPSoC de l'anglais *Multi-Processor System on Chip*) est un SoC à forte composante logicielle. La composante logicielle, qui représente la partie programmable du système, est dédiée à un ou plusieurs processeurs. Ces processeurs peuvent être génériques (GPP pour *general purpose processor*) ou spécifiques comme les processeurs de signaux numériques (DSP pour *digital signal processor*) ou les processeurs dédiés aux réseaux de communication, etc.

La quête de programmabilité dans les MPSoC est motivée essentiellement par deux raisons qui ne sont pas complètement indépendantes. La première concerne la flexibilité des systèmes qu'on cherche à concevoir, la deuxième est liée au coût de ces systèmes.

La flexibilité d'un système mesure la facilité de le faire évoluer pour l'adapter à des nouvelles exigences. Ainsi, il est généralement établi que recompiler le logiciel embarqué après avoir changé ou adapté l'application est beaucoup plus facile que de concevoir un nouveau circuit (cf section 1.2.1.1 sur les ASIC) spécifique aux nouvelles exigences de l'application en question. Un exemple courant d'une telle situation est illustré par le passage d'une certaine norme de codec¹ vidéo à une norme plus récente.

L'argument de coût concerne en premier lieu la conception elle-même. Sans compter le gain de temps et d'effort déjà apporté par flexibilité lors d'une éventuelle adaptation/évolution du système, il

1 Codeur/décodeur

est généralement bien connu que concevoir une certaine fonction en logiciel est moins coûteux que de la réaliser en matériel. Le gain est réalisé à la fois en termes de ressources physiques déployées (partagées dans le cas d'une implémentation logicielle, dédiées dans le cas d'une implémentation matérielle), mais aussi en terme d'effort de conception et de validation.

2.2. Architectures des systèmes multiprocesseurs monopuces

2.2.1. Architectures matérielles

2.2.1.1. Évolution des architectures matérielles des SoC

Dans cette section nous passons en revue les étapes importantes qui ont marqué l'évolution des architectures matérielles des systèmes monopuces.

1) Les ASIC

Le terme ASIC provient de l'anglais *Application-Specific Integrated Circuit* qui signifie circuit intégré spécifique à l'application. Se contentant de cette définition, tous les systèmes monopuces telle que définis plus haut ne sont autres que des ASIC.

Cependant, historiquement, le terme ASIC a désigné des circuits purement matériels qui implémentent des fonctions bien spécifiques sur une même puce [Joh97]. La conception des ASIC doit particulièrement son essor à l'évolution des techniques d'intégration VLSI et à l'utilisation extensive de bibliothèques de composants préconçus à fin d'obtenir des systèmes plus complexes. Pour des applications beaucoup plus complexes, il est coûteux et peu pratique de concevoir le système entier comme un bloc matériel figé. De nos jours, les ASIC désignent des composants plutôt élémentaires réalisant des fonctions bien déterminées (telle que une transformé de fourier rapide) sous forme d'une propriété intellectuelle (IP) qui peut être intégrée dans des systèmes plus larges.

2) Les architectures monoprocesseurs

Pour augmenter la flexibilité des systèmes monopuces et réduire leurs coûts, les premières architectures mettant en oeuvre la solution logicielle ont été des architectures monoprocesseurs (figure 2.1). Ces architectures sont souvent centrées autour d'un processeur à usage général exécutant un programme placé dans une mémoire locale et communicant avec des composants matériels appelés périphériques via un bus système. La partie logicielle représentée par le processeur prendra généralement en charge la partie contrôle de l'application (flot de contrôle). Les autres parties de l'application qui demandent un traitement plus intensif des données (flot de données) seront réalisées

sur des périphériques spécialisés (ASIC) qui jouent le rôle d'accélérateurs matériels.

Les microcontrôleurs conventionnels constituent un exemple typique de ces architectures.

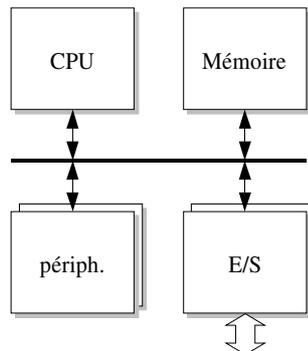


Figure 2.2: Architecture monoprocesseur

3) Les architectures multiprocesseurs de première génération

Les architectures multiprocesseurs de première génération constituent une évolution directe des architectures monoprocesseurs où des traitements de données, qui auparavant été confiés à des accélérateurs matériels dédiés, sont désormais pris en charge par un ou plusieurs processeurs spécialisés (DSP) (figure 2.3). Cette solution constitue un compromis qui exploite à la fois la puissance et la flexibilité des processeurs spécialisés.

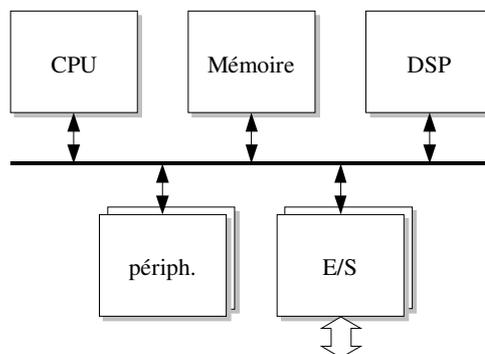


Figure 2.3: Architecture multiprocesseur de première génération

Dans ce type d'architecture, l'application logicielle est statiquement partitionnée entre GPP (parties à dominante contrôle) et DSP (parties à dominante traitement de données). La communication entre les deux parties peut être réalisée de différentes manières, telle que par mémoire partagée (plus courante) ou par canaux de communication point-à-point spécialisés.

Comme exemples d'architectures appartenant à cette catégorie, on peut citer l'architecture OMAP

de texas instrument [Omap] ou encore la plateforme Nomadik de ST [Nomadik] qui ciblent essentiellement le domaine multimédia grand public.

4) *Les architectures multiprocesseurs de deuxième génération*

L'évolution du besoin en terme de puissance de traitement d'une part, et la percée importante de la capacité d'intégration d'une autre part, ont contribué à la mise au point d'architectures hautement parallèles intégrant un nombre important de processeurs et de composants matériels sur une même puce. Vu l'importance de ces architectures dans le cadre de cette thèse, nous dédions la section suivante pour une plus ample description de leurs spécificités.

2.2.1.2. *Architectures multiprocesseurs monopuces*

Dans cette section, nous examinons de plus près les architectures multiprocesseurs monopuces (de deuxième génération) : leurs spécificités, leurs enjeux et les contraintes qui conditionnent leur conception.

1) *Architectures hétérogènes et massivement parallèles*

A la différence des systèmes multiprocesseurs de première génération, ceux de la deuxième génération sont des architectures massivement parallèles. Ce fort degré de parallélisme est une conséquence directe de la complexité croissante des applications embarquées qui désormais exigent des capacités de traitement de plus en plus importantes. En effet, pour palier ces besoins en terme de puissance de traitement, la solution qui consiste à augmenter la fréquence de fonctionnement des processeurs se voit vite atteindre ses limites pour deux raisons. La première est que la technologie est en phase d'atteindre la saturation en terme de délais de propagations [Jer04]. La deuxième est que cette solution pose un sérieux problème en ce qui concerne la consommation d'énergie (qui évolue d'une manière quadratique avec la fréquence [Cha92][Wei94]). Ainsi paralléliser le calcul s'avère être la seule solution pour faire face à ces nouveaux défis.

Comme pour les architectures multiprocesseurs de première génération, ceux de la deuxième génération sont également hétérogènes, dans le sens où la partie logicielle est confiée à différents types de processeurs aussi bien génériques que spécialisés. Cette hétérogénéité permet en fait de bien cibler les spécificités de l'application en question et constitue une caractéristique essentielle qui différencie les systèmes multiprocesseurs monopuces des multiprocesseurs classiques (super-ordinateurs). Ces derniers bénéficient au contraire d'une architecture homogène où les différents noeuds de calcul jouent un rôle équivalent [Int97].

2) *La communication : un goulet d'étranglement*

Augmenter le degré de parallélisme n'est pas sans avoir de conséquences directes sur les architectures des systèmes monopuces. En effet, un problème important auquel font face ces architectures concerne la communication qui désormais constitue un goulet d'étranglement vu la quantité importante d'informations qui doit être échangée entre les différents composants de l'architecture. Les systèmes classiques centrés autour d'un bus simple ne sont plus à même de fournir la bande passante nécessaire pour supporter le trafic généré par les MPSoC. Une première solution à ces problèmes a consisté à remplacer le bus simple par une hiérarchie de bus interconnectés par des ponts (*bridge*). Ce type de solution est implémenté, à titre d'exemple, par la norme AMBA [Arm03] ou encore Sonics [Son03].

Bien qu'ils bénéficient d'une bande passante bien plus importante que les bus simples, les bus hiérarchiques atteignent rapidement la saturation et ne permettent donc pas de supporter un trafic très important [Ben01][Ben02]. Par ailleurs, ce média de communication souffre encore du problème de mise à l'échelle (*scalability*) inhérent à toute architecture à base de bus partagé.

3) *Réseau de communication sur puce (NoC)*

Les réseaux de communication sur puce (NoC de l'anglais *network on chip*) constituent une alternative radicale aux bus partagés [Dal01][Goo02]. Une architecture à base de réseau sur puce (figure 2.4) est formée de plusieurs noeuds (*node* ou *tail* en anglais) qui communiquent par envoi de messages à travers le réseau de communication. Les noeuds peuvent être de différentes natures, en l'occurrence logiciels, matériels, etc. Par ailleurs, un noeud peut être hiérarchique, c'est à dire renfermant lui-même un réseau de communication.

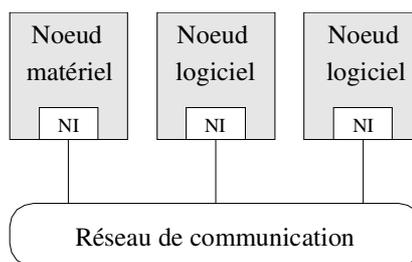


Figure 2.4: Architecture à réseau sur puce

2.2.2. Architectures logicielles

Parallèlement à l'évolution des architectures matérielles des systèmes monopuces, le logiciel embarqué est passé du simple programme séquentiel, souvent développé en langage assembleur, à un système concurrent implémentant un comportement complexe et bénéficiant d'une architecture à part

entière.

2.2.2.1. Vue globale : Architecture en couches

1) Complexité et des applications et du matériel

La complexité des applications embarquées d'une part, et les architectures matérielles de plus en plus sophistiquées d'une autre part, ont largement contribué à l'évolution du logiciel embarqué vers un système complexe bénéficiant d'une architecture à part entière. Par ailleurs, la place désormais importante du logiciel dans les systèmes MPSoC a imposé de nouvelles méthodologies de conception qui visent à réduire le coût de développement d'un tel logiciel (coût estimé à plus de 70% du coût total du système selon des études statistiques récentes [Itr03]). Structurer le logiciel embarqué en couches est donc la solution qui a été retenue pour faire face à ces exigences, une solution qui a visiblement trouvé ses racines dans le domaine de l'informatique classique.

2) Architecture en couches

L'architecture en couches implique un découpage vertical au niveau des fonctionnalités du logiciel embarqué [Tan90]. La figure 2.5 montre un tel découpage. On y distingue quatre couches : la couche applicative qui constitue la partie fonctionnelle du logiciel embarqué, la couche *middelware* ou encore intergicielle qui permet de fournir une vision uniforme des ressources qui sont généralement physiquement distribuées, la couche système d'exploitation qui permet de gérer localement les ressources disponibles et enfin la couche abstraction du matériel qui permet l'accès structuré à ces ressources.

Chaque couche fournit à la couche supérieure une interface de programmation propre (API de l'anglais *application programming interface*).

3) Risques d'inefficacité de l'architecture en couche

Dans le domaine de l'informatique, il est bien connu que multiplier les couches au sein d'un système logiciel entraîne souvent une dégradation des performances de ce système par rapport au même système développé d'une manière monolithique et spécifique au problème traité [Zit93]. Cependant, vu l'importance des avantages qui ressortent d'une architecture en couches (flexibilité, facilité de débogage et de maintenance, développement concurrent, etc.), les concepteurs préfèrent souvent renoncer à une architecture monolithique en dépit d'une éventuelle dégradation au niveau des performances.

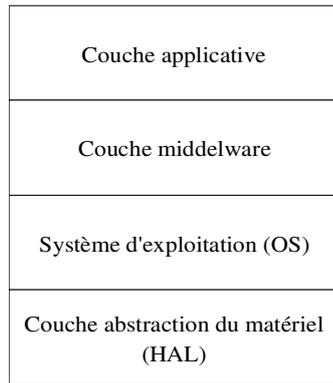


Figure 2.5: Les différentes couches de la pile logicielle

Dans le monde embarqué, le critère de performance joue un rôle critique et la tolérance d'une dégradation à ce niveau n'est généralement pas la bienvenue. Pour minimiser une telle dégradation tout en profitant des atouts d'une architecture structurée en couches (des atouts qui, rappelons-le, sont devenus primordiales pour maîtriser le coût et la complexité des applications récentes), la structure en couche du logiciel embarqué doit assurer une adéquation suffisante avec l'application en question. Ainsi, on évite généralement, pour les SoC, d'imposer une API standard valable pour tout types d'application. Au contraire, chaque domaine d'application peut avoir ses propres API qui reflètent au mieux ses spécificités.

2.2.2.2. La couche abstraction du matériel

Classiquement, le logiciel embarqué est développé à un niveau d'abstraction très bas en utilisant souvent le langage assembleur. Pour ce faire, les programmeurs sont supposés avoir une connaissance très poussée de l'architecture matérielle sous-jacente dans ses moindres détails. D'un point de vue du logiciel, cette dépendance étroite vis-à-vis de l'architecture matérielle présente plusieurs inconvénients: tout d'abord, ceci implique un long cycle séquentiel de conception, puisque les programmeurs sont obligés d'attendre qu'une architecture matérielle complète soit disponible. Cette situation s'aggrave encore plus si des modifications à l'architecture initiale s'avèrent nécessaires, entraînant la reconception d'une majeure partie du logiciel. Ensuite, ceci rend le processus de validation et de débogage du logiciel fastidieux et induit des erreurs à cause de dépendances matérielles subtiles. Enfin, à cause de ces mêmes dépendances, la réutilisation de composants logiciels préconçus se trouve considérablement limitée.

La notion de couche d'abstraction du matériel (HAL de l'anglais *Hardware Abstraction Layer*) est introduite pour palier les inconvénients d'une telle dépendance bas niveau de l'architecture matérielle. Dans le domaine de l'informatique classique, le HAL est déjà un concept bien établi. On peut citer

comme exemples le HAL de *WindowsNT*, l'interface UDI (*Universal Device Interface*) de linux, le SDL (*Simple DirectMedia Layer*) [Sdl], etc. Il s'agit à chaque fois, de couches logicielles assez « épaisses » qui font abstraction du matériel à travers une API standard et figée. Par ailleurs, étant donné qu'elles implémentent déjà plusieurs politiques et choix de conception, ces couches d'abstraction restent étroitement couplées au système d'exploitation. Par opposition, dans un SoC, la couche HAL est une couche logicielle fine qui est supposée fournir uniquement une abstraction de l'accès aux ressources matérielles de l'architecture, sans implémenter aucune politique particulière à ce stade (responsabilité du système d'exploitation) [Yoo04]. Ainsi, la couche HAL peut être considérée comme neutre vis-à-vis du système d'exploitation [Tan95].

2.2.2.3. La couche système d'exploitation

Un système d'exploitation est une entité logicielle qui sert à fournir une abstraction du matériel pour le programmeur de l'application logicielle [Tan92]. Dans les systèmes informatiques classiques, cette abstraction va jusqu'à la virtualisation totale de l'architecture matérielle qui apparaît comme si elle disposait de ressources infinies. Dans le domaine embarqué, une telle abstraction peut s'avérer pénalisante en terme de performance et de coûts [Gau01]. Par exemple, le mécanisme conventionnel de mémoire virtuelle est souvent associé à une inefficacité au niveau de la consommation d'énergie et un comportement non déterministe qui va mal avec les applications à caractère temps réel.

Vu leur caractère spécifique, dans les systèmes embarqués, on fait généralement l'hypothèse d'un concepteur d'application averti qui maîtrise bien son système et qui possède un certain degré de responsabilité vis-à-vis de ce système. Dans ce cadre, le système d'exploitation est vu comme l'entité logicielle qui facilite l'accès au matériel en coopération avec le logiciel applicatif. Son rôle principal est de multiplexer l'accès à des ressources limitées en fournissant une abstraction adéquate de ces ressources, tout en garantissant une certaine qualité de service.

Parmi les abstractions fondamentales d'un système d'exploitation, le parallélisme logiciel consiste à multiplexer une ressource limitée qui est la puissance de calcul ou de traitement sur plusieurs tâches² logicielles. Dans le cas où le nombre de processeurs physiques est inférieur au nombre de tâches (ce qui est fréquemment le cas) le système d'exploitation doit implémenter un pseudo parallélisme logiciel (figure 2.6).

2 Dans cette thèse, le terme tâche est utilisé pour représenter d'une façon générique aussi bien les processus que les threads conventionnels.

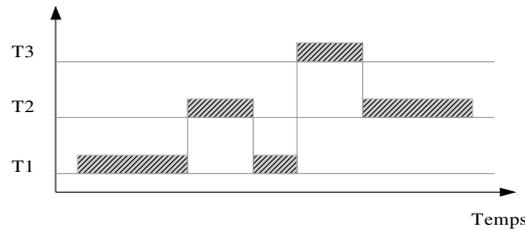


Figure 2.6: *pseudo parallélisme de trois tâches logicielles sur un seul processeur*

Dans la figure 2.6, l'instant précis de commutation entre les tâches et le choix de la prochaine tâche à exécuter constitue la politique d'ordonnancement du système d'exploitation. Une telle décision peut être prise à l'avance durant la phase de conception. Dans ce cas, on parle d'ordonnancement statique et le rôle du système d'exploitation se réduit à garantir le respect de ces décisions au cours de l'exécution, généralement en se référant à des tables statiques qui définissent quelles tâches exécuter à quels moments précis. Ce type d'ordonnancement simple ne reste cependant applicable que pour des systèmes entièrement prédictibles et qui ont généralement un comportement de type flot de données tels que les systèmes dits SDF (*synchronous data flow*) [Lee87][Hal91]. Dans le cas générale, on à recours à un ordonnancement dynamique dans lequel les décisions quant à l'exécution des tâches est prise au cours de l'exécution par l'ordonnanceur du système d'exploitation. La figure 2.7 illustre un modèle générique d'ordonnancement d'une tâche logicielle montrant les différents états qu'une tâche peut avoir tout au long de son cycle de vie.

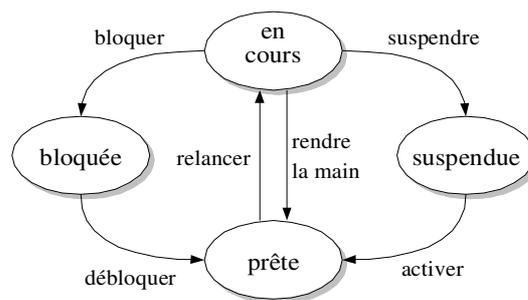


Figure 2.7: *modèle générique d'ordonnancement d'une tâche logicielle*

2.3. De la spécification fonctionnelle à l'architecture RTL : la discontinuité

Pour maîtriser la complexité des systèmes embarqués monopuces, les concepteurs font appel à des flots de conception qui définissent les étapes à suivre et les outils à mettre en oeuvre pour arriver

au système final. Dans ce type de flots, on part généralement d'une spécification initiale du système à concevoir que l'on essaie de raffiner par la suite en suivant les étapes définies par le flot avant d'atteindre une description dite synthétisable, c'est-à-dire qui peut servir comme entrée des outils conventionnels de synthèse physique ou de prototypage³.

Dans le cas des systèmes MPSoC, cette description finale doit contenir deux « ingrédients »: l'architecture matérielle décrite au niveau transfert de registre RTL (*register transfer level*) et l'architecture logicielle avec tous ces composants vus précédemment, sous forme d'images binaires directement «chargeable» sur les différents processeurs cibles. Comparé à la spécification initiale du système qui exprime les fonctionnalités de celui-ci d'une façon purement fonctionnelle et indépendante de l'implémentation, cette description finale est souvent qualifiée de bas niveau [Nic04]. Il en résulte une discontinuité importante (ou encore un gap) entre la spécification et l'implémentation du système. Au niveau des flots de conception classiques, cette discontinuité verticale se traduit également par une autre discontinuité horizontale qui tend à séparer la description du système en deux parties complètement indépendantes, une logicielle et l'autre matérielle qui sont alors conçues et raffinées individuellement d'une manière séparée.

2.3.1. Modèles de représentation

Tout au long du flot de conception, le concepteur manipule différents modèles de représentations qui décrivent le système aux différentes étapes du flot. Le but ultime est d'arriver à une implémentation correcte à partir de la spécification initiale de haut niveau. Le passage d'un niveau de conception à un autre étant généralement assisté par des outils d'aide à la conception, on a besoin de modèles qui expriment les entrées/sorties de chaque étape de conception d'une manière compréhensible par ces outils.

2.3.1.1. Modèles de spécification

Un modèle de spécification doit permettre de décrire fonctionnellement le système que l'on désire concevoir ainsi que les contraintes associées à son fonctionnement. Cette description doit idéalement être indépendante d'une éventuelle implémentation possible de ce système.

1) Approche langage pour la spécification

Une première forme de spécification informelle est constituée par les langages naturels (humains). Il s'agit d'établir une sorte de cahier de charges du système. Cette description informelle ne peut être l'entrée unique d'un flot de conception car elle induit souvent des confusions et ne peut

³ Dans le cadre de cette thèse, un flot de conception SoC s'arrête juste avant l'étape de synthèse physique qui est considérée comme « back-end » pour le flot considéré.

être exploitée par les outils d'aide à la conception.

Ainsi des représentations plus formelles de la spécification s'imposent. Ces représentations se basent généralement sur l'approche langage⁴ qui consiste à décrire le système (ou une partie du système) en utilisant un langage informatique ayant une syntaxe bien définie et univoque. On parle alors de langages de spécification qui peuvent être sous différentes formes syntaxiques. Historiquement, la forme textuelle a été la plus utilisée (SDL, C++). Cependant, récemment, la notation graphique suscite de plus en plus d'intérêt vu son ergonomie et sa simplicité d'utilisation et d'appréhension (UML, Statecharts). Notons que syntaxiquement parlant, ces deux formes de représentation restent équivalentes [Thi98] (dans le sens où on peut passer bijectivement de l'une à l'autre). En pratique, on les utilise souvent d'une manière complémentaire (exemple de Simulink).

2) Aspect exécutable de la spécification

L'aspect exécutable du langage de spécification est souvent considéré comme déterminant à un certain stade de la spécification pour valider le comportement du système. Ceci suppose que le langage de spécification en question possède une sémantique d'exécution (ou encore sémantique opérationnelle) bien définie (cf paragraphe 2.3.1.3 sur les modèles de calcul). D'autres formes de sémantiques du langage de spécification permettent une validation plus systématique à travers la vérification formelle [Win93][Ken99]. Il s'agit généralement de sémantiques plus abstraites mettant l'accent sur certains aspects particuliers du langage (sémantique dénotationnelle, sémantique axiomatique, sémantique algébrique, etc.)[Sch86][Hoa69].

3) Universalité de la spécification

Un débat classique dans la littérature a toujours été de savoir si on a besoin d'un seul langage universel qui convient pour spécifier n'importe quel système, ou faut-il au contraire avoir recours à plusieurs langages de spécifications, chacun étant spécifique à un sous ensemble particulier du système. Les partisans d'un langage universel mettent l'accent sur l'intérêt d'une telle approche comme cadre fédérateur qui facilite le développement, l'échange et la maintenance de la spécification. L'exemple typique d'un tel langage serait UML qui a récemment suscité un intérêt croissant dans la communauté SoC [Dyl05][Ric05][Sha04]. Dans le camp opposé, on évoque le caractère hétérogène des systèmes embarqués pour justifier le besoin de faire appel à plusieurs langages de spécification. On parle alors de langages spécifiques au domaine ou DSL (*Domain Specific Language*) [Van00]. Un exemple de DSL est donné par le langage ALPHA [Dup95] qui est un langage fonctionnel utilisé pour la génération d'architectures systoliques à partir d'une description

4 Le fondement de l'approche langage pour la modélisation des systèmes bénéficie d'un développement théorique important dans la littérature [Van00][Mau04]

basée sur des équations de récurrences [Cac00].

Nous pensons que les deux approches sont complémentaires. En effet, elles correspondent plutôt à deux niveaux d'abstraction différents. Ainsi, un langage de modélisation universel tel que UML peut s'avérer intéressant comme première spécification semi-formelle et non exécutable du système. Le caractère semi-formel est lié au fait que le standard UML ne spécifie pas de sémantique dynamique. De ce fait, le standard se limite à la définition de la syntaxe abstraite du langage augmentée d'une sémantique statique décrivant les contraintes de composition structurelle⁵. Cette absence de sémantique opérationnelle paraît tout à fait logique vue la généralité du modèle UML. En effet, à ce niveau de description, on s'intéresse à l'aspect structurel (décomposition en entités et relations entre ces entités) qui est un aspect générique commun à toute activité de modélisation. Le comportement, c'est à dire la manière avec laquelle le système est supposé évoluer au cours du temps est, quant à lui, un aspect moins générique et dépend du type de système en question. Pour pouvoir définir cette sémantique, il faut que le langage soit plus spécifique au domaine particulier du système (ou du sous-système) en question d'où la nécessité d'un DSL dont la sémantique permet de traduire le comportement dynamique du système. Une approche intéressante serait de considérer un tel DSL comme une spécialisation du langage universel utilisé pour la spécification initiale. Ainsi, le langage universel peut être vu comme un méta-langage qui peut servir pour spécifier tous les DSL qui en dérivent. Cette approche peut alors bénéficier du développement récent dans le domaine de la conception basée sur le formalisme MDA (*Model Driven Architecture*) [Kar03][Szt97].

2.3.1.2. Modèles d'implémentation

Alors qu'un modèle de spécification cherche à décrire ce que fait le système, un modèle d'implémentation traduit plutôt comment est fait ce système [Kri05]. On l'appelle aussi modèle d'exécution ou encore modèle de réalisation. La distinction entre modèle de spécification et modèle d'implémentation est purement conceptuelle. D'ailleurs une séparation totale entre les deux types de modèle n'est que théorique, vu qu'en pratique, la spécification d'un système est souvent influencée par une éventuelle implémentation de celui-ci.

1) sémantique de synthèse d'un langage d'implémentation

Ce qui caractérise un langage d'implémentation est avant tout le fait qu'il possède une sémantique particulière reliée à l'implémentation physique des modèles qu'il décrit, que l'on appelle sémantique de synthèse. Cette sémantique est souvent inférée par l'outil de génération d'architecture associé au langage en question. En plus de cette sémantique de synthèse, les langages d'implémentation possèdent généralement une sémantique opérationnelle pour la simulation. Vu les contraintes

⁵ disponible via OCL (*Object Constraint Language*)

supplémentaires qu'impose la synthèse par rapport à la simulation, la sémantique de synthèse concerne souvent un sous-ensemble restreint du langage. L'exemple type serait le sous-ensemble synthétisable de VHDL.

2) *Caractère modulaire d'un modèle d'implémentation*

Un aspect important partagé par la plupart des langages d'implémentation est le support de la modularité et de la composition. Cet aspect favorise, en effet, la réutilisation de modèles existants au sein d'une description d'architecture. Dans les langages conventionnels de description du matériel, cette modularité a favorisé l'essor de la notion de propriété intellectuelle ou encore IP (*Intellectual Property*). Grâce à cette notion, des composants individuels (IP) peuvent être conçus séparément par différentes équipes pour être intégrés par la suite dans une architecture plus large.

Dans le domaine du logiciel, la composition est une pratique essentielle. Dans sa forme la plus basique, elle se ramène à la notion même de modularité et de programmation structurée [Knuth]. Ainsi, un programme complexe est naturellement « décomposé » en un ensemble de sous-programmes plus simples à concevoir.

2.3.2. Flots de conception des SoC

2.3.2.1. *Flots classiques*

La figure 2.2 présente les différents modèles et étapes d'un flot classique de conception de systèmes embarqués monopuces. Un tel flot est caractérisé par une séparation franche et prématurée entre la conception de la partie matérielle et de la partie logicielle, une séparation qui s'effectue dès lors qu'un partitionnement initial de la spécification fonctionnelle est opéré. Vers la fin du flot de conception (étape d'intégration) les deux parties sont recombinaées dans une même description bas niveau où le matériel est décrit en RTL et le logiciel au niveau instruction binaire. Cette description peut servir pour effectuer une évaluation finale du bon fonctionnement et des performances de l'implémentation obtenue via la cosimulation globale du système [Yoo05].

Selon les hypothèses que l'on prend concernant l'architecture cible et le type de transformations qui aboutissent à cette architecture, on peut classer les différents flots existants en trois catégories.

1) *Flots séquentiels de conception de circuits spécifiques*

Ce type de flot impose très peu de restrictions (idéalement aucune) concernant l'architecture cible. Partant de la spécification initiale de l'application, l'architecture matérielle est conçue spécifiquement à l'application en question. Une fois cette architecture matérielle est figée, le développement du logiciel embarqué peut alors démarrer. Il en résulte un long cycle séquentiel dans

lequel les concepteurs du logiciel sont forcés à chaque fois d'attendre une implémentation finale de l'architecture matérielle. La conception elle-même de chaque partie de l'architecture est souvent faite manuellement sans recours à des outils d'automatisation.

2) *Flots de co-conception de circuits spécifiques*

Pour résoudre le problème de séquentialité dans le premier type de flot, les cycles de conception des parties logicielle et matérielle doivent être concurrents. Les flots de co-conception, qui ont émergé durant les deux dernières décennies, ont apporté une solution à ce problème moyennant des contraintes restrictives sur l'architecture cible [Bar94][Wol03][Qui94]. En effet, en utilisant ce type de flots, les conceptions des parties logicielle et matérielle de l'architecture peuvent être menées en parallèle en se départageant un certain nombre d'hypothèses concernant la nature de l'architecture sous-jacente. De plus, en imposant un formalisme particulier au niveau de la spécification de l'application, il est possible de rendre automatique les étapes de conception des architectures logicielles et matérielles [Ism94]. Malheureusement, vu l'importance des hypothèses et restrictions introduites, ces flots finissent par être trop contraignants pour être utilisables en pratique, avec souvent des architectures cibles simplistes et inadaptées aux applications les plus exigeantes. Aujourd'hui, il est de plus en plus admis que la conception d'architectures aussi complexes que les MPSoC est un art en ce sens qu'il est difficile de la rendre complètement automatique (en partant de zéro).

3) *Flots basés sur la notion de plate-forme*

Récemment, la conception basée sur la notion de plate-forme (*Platform based Design*) a suscité un intérêt croissant dans le domaine de la conception des MPSoC [Vin02][Keu00][Jia02]. Dans ce type d'approche, le problème de la séquentialité de la conception logicielle/matérielle est abordé en déplaçant une grande partie de la conception de l'architecture en amont du flot de conception. Ceci donne lieu à la notion de famille génératrice d'architectures, spécifiques à un domaine d'applications. Une étape de configuration/spécialisation de cette architecture générique est alors nécessaire pour chaque application particulière appartenant à ce domaine. On parle souvent d'étape d'instanciation d'architecture. Bien que le degré de spécificité de l'architecture finale vis-à-vis de l'application reste inférieur aux deux approches précédentes, ce type de flot constitue un compromis intéressant entre performances d'un côté et coûts de l'autre. A ceci, s'ajoute le fait qu'il permet de résoudre un problème de plus en plus important qui est celui du coût de développement du masque (dépassant le million de dollar pour les technologies les plus récentes). En effet, une solution intéressante consiste à faire appel à la capacité de configuration des composants matériels programmables (tels que les FPGA) lors de l'étape de spécialisation ou d'instanciation de l'architecture matérielle.

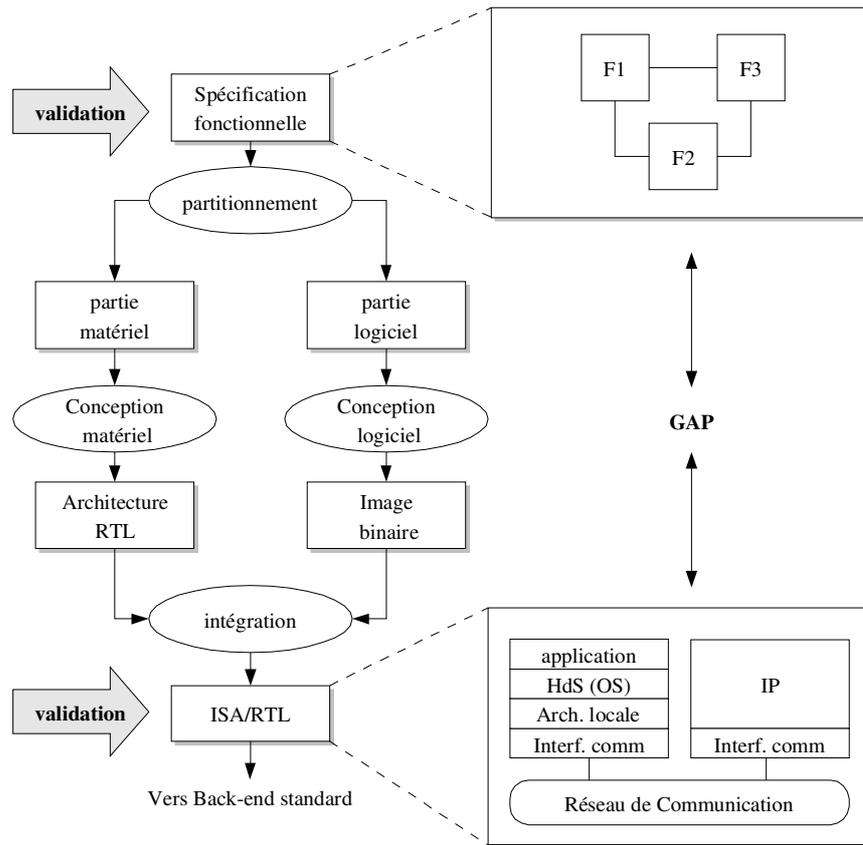


Figure 2.8: Les étapes et modèles d'un flot de conception classique

2.3.2.2. Approche du groupe SLS : le flot ROSES

Le flot ROSES développé au sein du groupe SLS se base sur l'automatisation de la conception des interfaces logicielles et matérielles [Ces04]. Pour cela, la notion d'architecture virtuelle est définie.

1) Architecture virtuelle : orthogonalité comportement/communication

La composition implique qu'un système complexe peut être construit par assemblage de composants élémentaires communiquants les uns avec les autres. Les composants peuvent avoir des origines différentes, décrits à différents niveaux d'abstraction et utilisant différents protocoles de communication. Pour faciliter l'assemblage de ces composants hétérogènes, il faut séparer le comportement et la communication [Nic02].

L'orthogonalité du comportement et de la communication est par ailleurs un concept très favorable au processus de raffinement et de synthèse pour deux raisons. Tout d'abord, il permet de raffiner indépendamment les composants d'une part et la communication de l'autre. Ceci est important dans la mesure où ce raffinement ne s'effectue pas toujours par une seule équipe. Par

ailleurs, séparer le comportement de la communication revient à définir deux interfaces différentes : celle liée au composant et celle liée à la communication. Cette double interface s'avère alors intéressante comme contexte de base pour guider le processus de synthèse et de génération d'adaptateur de communication.

Le flot ROSES met en oeuvre ces concepts à travers la notion d'architecture virtuelle. L'architecture virtuelle permet, en effet, de représenter dans une même description différents composants de calcul et de communication décrits à différents niveaux d'abstraction. Dans cette description, la notion de module virtuel permet d'encapsuler le comportement fournissant ainsi deux interfaces : une interface interne utilisée par le module et une interface externe liée au canal de communication. Selon la nature et le niveau d'abstraction des interfaces interne et externe, un adaptateur est alors utilisé (généré) pour permettre la communication entre les deux interfaces.

2) Automatisation de la génération

Dans le flot ROSES, deux types d'adaptateurs sont utilisés : des adaptateurs fonctionnels pour la simulation et des adaptateurs architecturaux pour la synthèse. Les adaptateurs fonctionnels permettent une simulation comportementale de l'ensemble de la description sans se soucier d'une quelconque implémentation de l'architecture. Ce type d'adaptateurs est utilisé en amont du flot de génération pour valider le comportement fonctionnel du système. Les adaptateurs architecturaux sont générés par les outils de synthèse système (ASOG [Gau01] pour la partie système d'exploitation et ASAG [Lyo01] pour l'architecture locale du sous-système CPU).

3) Limitations actuelles du flot ROSES

Les outils de génération d'architectures logicielle et matérielle partent de la spécification de l'architecture virtuelle et génèrent, pour chaque module virtuel ou encore interface abstraite, l'adaptateur correspondant.

Pour un noeud logiciel, ce processus consiste à générer à la fois la couche système d'exploitation qui va permettre à l'application multi-tâche de s'exécuter, et l'architecture locale du sous-système CPU sous-jacent. Actuellement, cette génération se base sur un « template » d'architecture matérielle fixe qui constitue une sorte de « consensus » partagé par les deux outils de génération ASAG et ASOG.

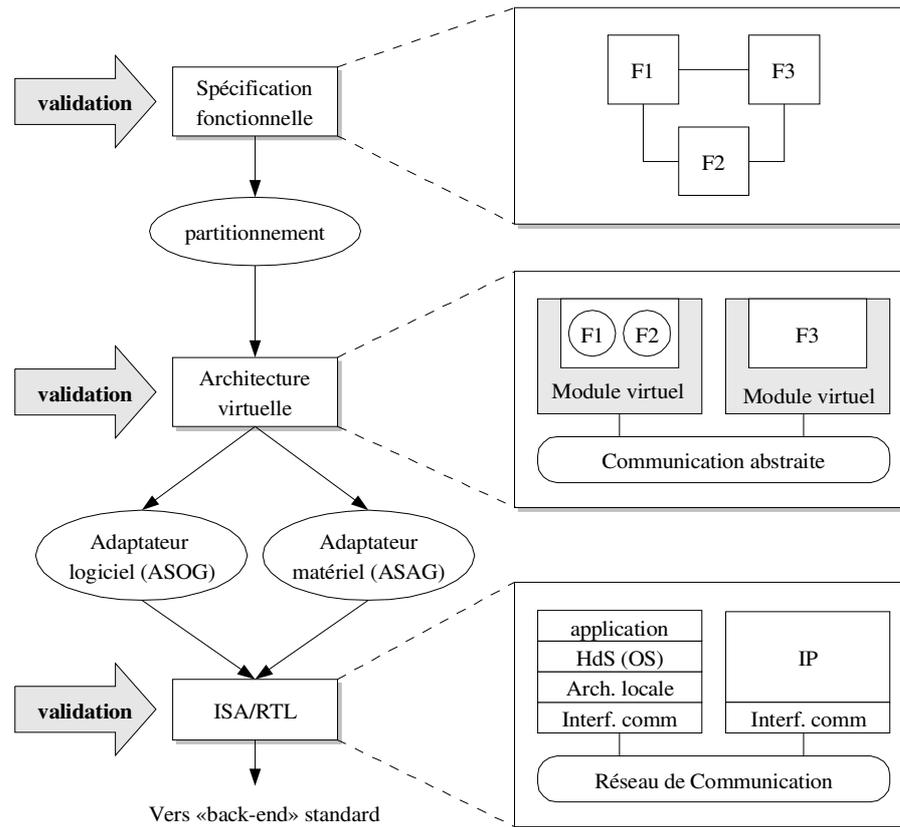


Figure 2.9: flot ROSES et notion d'architecture virtuelle

La simulation fonctionnelle effectuée en amont du flot de génération permet une validation fonctionnelle du comportement du système. Bien qu'elle permet de tester différentes configurations possibles de l'architecture de communication, cette validation ne reflète aucune estimation des performances de l'architecture en question. En ce sens, le flot ROSES souffre des mêmes limitations des flots classiques concernant la discontinuité entre le matériel et le logiciel.

2.3.3. Approche proposée

Les approches de conception classique vues précédemment sont caractérisées par une discontinuité qui marque le passage de la spécification initiale à l'implémentation finale. Cette discontinuité constitue comme nous l'avons souligné un handicap dans les flots classiques.

Pour résoudre ce problème, il est clair qu'il faut introduire, dans le flot de raffinement de l'architecture des étapes supplémentaires permettant l'interaction entre logiciel et matériel. A chaque étape, une exploration de l'architecture logicielle/matérielle doit permettre d'évaluer les performances globales du système à ce niveau d'abstraction et guider les différents choix d'implémentation.

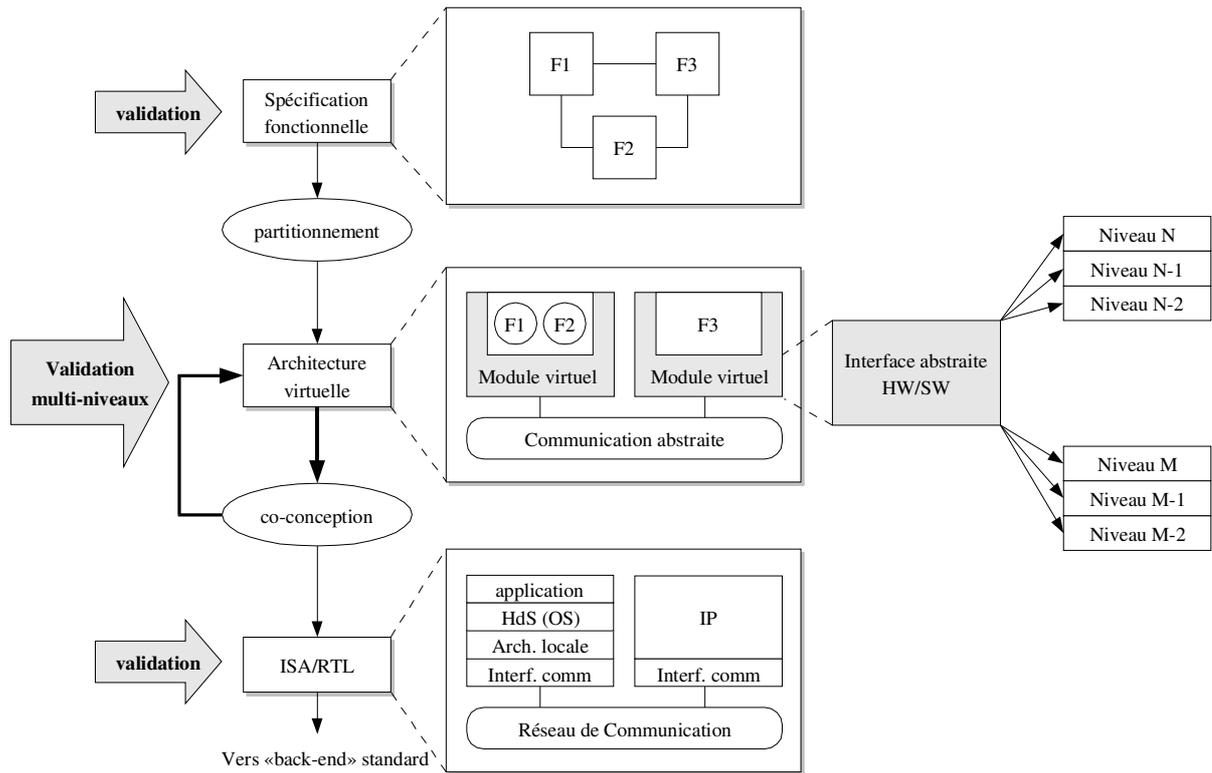


Figure 2.10: L'approche proposée dans le contexte du flot ROSES

Bien que l'approche proposée reste générale, dans le sens où elle n'est pas liée à un flot particulier, elle s'intègre particulièrement bien dans le flot ROSES à travers la notion de module virtuel et d'interface abstraite et peut bénéficier de la génération automatique pour faciliter l'exploration d'architecture.

2.4. Conclusion

Ce chapitre a été dédié à la description des systèmes qui font l'objet de cette thèse, à savoir les systèmes multiprocesseurs monopuces. L'architecture de tels systèmes a été analysée, mettant l'accent sur la complexité aussi bien des parties logicielles que matérielles de ces architectures.

Face à cette complexité, les flots classiques ne semblent pas apporter une solution efficace qui facilite l'exploration et la validation de ces architectures en vue de maîtriser les coûts inhérents à leurs développements. La cause d'une telle défaillance est ramenée à la discontinuité des modèles de représentation architecturaux utilisés au sein de ces flots classiques. L'approche proposée par cette thèse cible spécifiquement cette discontinuité en proposant des modèles de représentation intermédiaires permettant un raffinement graduel des systèmes logiciels/matériels.

Chapitre 3

Abstraction de l'interface logicielle/matérielle et état de l'art

Sommaire

Chapitre 3	
Abstraction de l'interface logicielle/matérielle et état de l'art.....	38
3.1.Abstraction de l'interface logicielle/matérielle.....	39
3.1.1.L'entité « interface logicielle/matérielle ».....	39
3.1.2.Niveaux d'abstraction de l'interface de communication matérielle.....	42
3.1.3.Niveaux d'abstraction de l'interface de programmation (API)	46
3.2.Validation de l'interface logicielle/matérielle.....	51
3.2.1.simulation multi-niveaux de l'interface logicielle/matérielle.....	52
3.2.2.Approches existants pour la cosimulation logicielle/matérielle.....	54
3.3.Conclusion.....	63

Dans la conception des systèmes embarqués classiques, l'interface entre le logiciel et le matériel a toujours été le processeur, considéré comme charnière ultime entre deux mondes différents. Avec la complexité croissante des architectures matérielles qui peuvent embarquer désormais plusieurs de ces processeurs à côté des autres composants, cette vision classique de l'interface logicielle/matérielle qui considère l'architecture à un bas niveau d'abstraction s'avère insuffisante et contraignante vis-à-vis d'une exploration et validation efficace des systèmes MPSoC.

Dans ce chapitre, nous essayons d'examiner le concept d'interface logicielle/matérielle sous un angle plus large. Pour cela, nous nous intéressons dans un premier temps, à définir l'entité interface logicielle/matérielle en la plaçant dans le contexte des systèmes MPSoC. Une étude des niveaux d'abstraction conventionnels, utilisés d'une part pour l'interface de programmation logicielle et de l'autre part pour l'interface de communication matérielle, nous permet alors de classer les méthodes existants de validation de l'interface logicielle/matérielle.

3.1. Abstraction de l'interface logicielle/matérielle

Dans sa Logique (1800), Kant définissait l'abstraction comme « la séparation de tout ce en quoi les représentations données se distinguent ». L'abstraction est donc avant tout une opération de simplification qui consiste à ne garder, d'une entité complexe, que les éléments qui sont significatifs à un niveau donné. Cependant, pour mener à terme un travail d'abstraction encore faut-il bien identifier « l'entité complexe » en question. Alors que le concept « table » aurait comme objet d'abstraction, l'entité physique formée d'un plateau en un matériau donné et reposant sur quatre pieds, le concept « interface logicielle/matérielle » passe d'abord par identifier l'objet d'abstraction. Il est évident à ce niveau que le contexte où on se place, va influencer la signification et la portée de l'interface logicielle/matérielle.

3.1.1. L'entité « interface logicielle/matérielle »

Dans un contexte de systèmes aussi hétérogènes que les MPSoC, l'entité « interface logicielle/matérielle » n'est ni unique ni homogène. En effet, dans un même système MPSoC, on a affaire à plusieurs interfaces logicielles/matérielles qui peuvent avoir des caractéristiques différentes les unes des autres conformément à la nature distribuée et hétérogène du logiciel embarqué. Ce caractère distribué de l'interface logicielle/matérielle est d'ailleurs cohérent avec la notion d'architecture virtuelle et le concept d'enveloppe d'adaptation sur le quel se base cette notion (cf deuxième chapitre).

3.1.1.1. Caractère distribué de l'interface logicielle/matérielle

La figure suggère une vue globale de la notion d'interface logicielle/matérielle dans le contexte considéré des systèmes MPSoC. Conceptuellement, l'interface logicielle/matérielle est vue comme l'entité abstraite qui encapsule le logiciel dans l'environnement d'exécution global qui est l'environnement matériel. Il s'agit, en quelque sorte, de la « machine virtuelle » qui permet d'interpréter les commandes venant du « monde logique » du logiciel et d'interagir correctement avec le « monde physique » du matériel. Selon les niveaux de description avec lesquels les deux mondes sont décrits, le niveau d'abstraction de la machine ou encore de l'interface logicielle/matérielle varie. Notons que la vision proposée par la figure est plutôt une vision matérielle des choses, puisque l'environnement global reste matériel et que, dans cet environnement, c'est le logiciel qui fait l'objet d'encapsulation. Ceci n'est sans doute pas étonnant, vu qu'on reste dans un contexte de systèmes embarqués où la l'aspect performance prime, même si la composante logicielle est fortement présente dans ces systèmes.

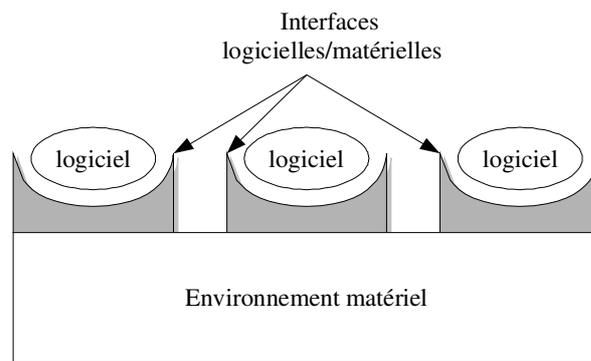


Figure 3.1: l'interface logicielle/matérielle dans le contexte MPSoC

Un autre caractère important souligné par la figure est le caractère distribué de l'interface logicielle/matérielle. Ce caractère est fortement lié à l'aspect distribué et hétérogène du logiciel embarqué lui-même. En effet, pour le concepteur du logiciel embarqué, il s'agit de programmer un ensemble de machines hétérogènes plutôt qu'une super machine homogène. On peut penser malgré tout, à unifier les différentes machines via une interface unique qui permet d'abstraire cette hétérogénéité. Ceci est, d'ailleurs, la solution qui est retenue dans le domaine informatique pour « virtualiser » les ressources hétérogènes et physiquement distribuées d'un système réparti (exemple du NFS *Network File System*). Dans le domaine embarqué, on opte plutôt pour la solution qui consiste à distinguer conceptuellement les différentes machines, et à considérer donc plusieurs interfaces logicielles/matérielles au sein du même MPSoC. L'interaction entre ces différentes interfaces se fait alors via l'environnement matériel. Notons que cette interaction a une signification

différente selon le niveau d'abstraction auquel on se place. Au plus bas niveau considéré, qui consiste à confondre l'interface logicielle/matérielle avec le processeur physique, il s'agit de la façon avec laquelle différents processeurs peuvent communiquer entre eux. Ceci couvre déjà un éventail très large de modes d'interaction possibles. Aux niveaux d'abstraction plus élevés, nous verrons dans les chapitres suivants que les modes d'interaction se trouvent réduits ou encore « canalisés » avec des définitions adéquates de l'interface logicielle/matérielle à chaque niveau. Ceci est, d'ailleurs, cohérent avec l'objectif même de l'abstraction qui est de simplifier pour comprendre et maîtriser la complexité.

3.1.1.2. Position dans le flot ROSES

La notion d'architecture virtuelle (introduite dans le second chapitre) est utilisée au sein du flot ROSES pour rendre compte du principe de l'orthogonalité du comportement et de la communication dans une architecture donnée. Rappelons que selon ce principe, le comportement et la communication peuvent être raffinés indépendamment l'un de l'autre, ce qui rend le processus de raffinement flexible et permet une exploration facile de l'espace des solutions architecturales. L'architecture virtuelle permet, en effet, de représenter dans une même description, différents composants de calcul et de communication décrits à différents niveaux d'abstraction. Dans cette description, la notion de module virtuel permet d'encapsuler le comportement fournissant ainsi deux interfaces : une interface interne liée au module et une interface externe liée au canal de communication. Selon la nature et le niveau d'abstraction des interfaces interne et externe, un adaptateur est alors utilisé (généré) pour permettre la communication entre les deux interfaces (figure 3.2).

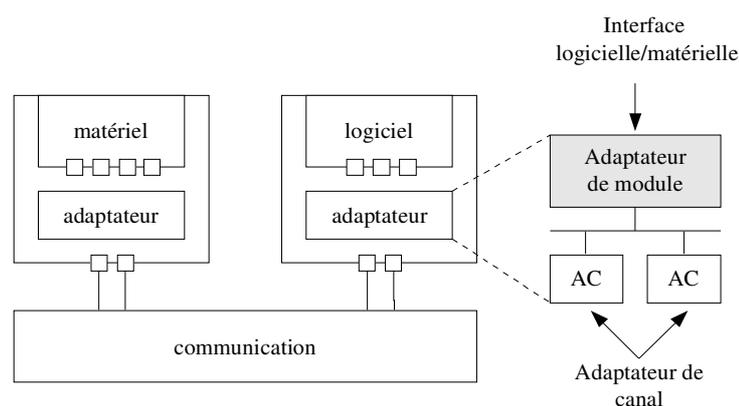


Figure 3.2: Position de l'interface logicielle/matérielle dans l'architecture virtuelle

Dans l'architecture virtuelle, un module virtuel peut correspondre à un noeud logiciel (le module à droite dans la figure) . Dans ce cas, l'interface interne correspond à l'API utilisée par le logiciel,

alors que l'interface externe regroupe l'ensemble des ports utilisés pour communiquer au réseau de communication. Notons qu'en terme de représentation, l'API interne du logiciel est elle aussi représentée sous forme de ports. Ceci permet en effet de regrouper les services de même nature dans un seul port. Il faut, cependant, noter que ces ports n'impliquent pas toujours une sémantique de communication. En effet, on distingue aussi, parmi les ports internes, ceux qui sont destinés à fournir des services locaux (exemple un service *timer*) qui n'engagent pas forcément de la communication avec l'environnement externe.

Conformément au principe de l'orthogonalité du calcul et de la communication, nous avons vu au second chapitre qu'un adaptateur relatif à un module virtuel peut toujours être décomposé en un adaptateur de module et un ou plusieurs adaptateurs de canaux. Cette décomposition reste valable dans le cas particulier où le module virtuel correspond à un noeud logiciel. Dans ce cas, l'adaptateur de module correspond exactement à l'interface logicielle/matérielle que nous avons définie dans le paragraphe précédent. Cette manière de voir l'interface logicielle/matérielle est intéressante car elle permet de mettre l'accent sur deux points : le premier concerne l'aspect distribué de l'interface logicielle/matérielle qui est mis en valeur par l'organisation même de l'architecture virtuelle en modules indépendants. Le deuxième concerne la flexibilité qu'apporte la séparation entre comportement et communication, une flexibilité qui va nous permettre, dans le reste du chapitre, d'étudier séparément les niveaux d'abstraction relatifs à l'API logicielle et ceux relatifs à l'interface de communication avec l'environnement matériel.

3.1.2. Niveaux d'abstraction de l'interface de communication matérielle

L'étude des niveaux d'abstraction de la communication dans les langages de description du matériel n'est pas nouvelle [Gaj03][Bru00]. Dans cette section, nous essayons de formaliser la classification de ces niveaux d'abstraction dans le cadre des hypothèses relatives au contexte qui nous concerne. Ceci nous permettra aussi de fixer la terminologie concernant ces niveaux et de lever certaines confusions liées à leur utilisation.

3.1.2.1. Critères de classification

Dans le contexte où nous nous plaçons, le terme communication désigne spécifiquement la communication matérielle, et non pas la communication dans sa plus vaste signification. Plus particulièrement, nous nous intéressons à des modules matériels concurrents qui échangent de l'information via des canaux (ou encore média) de communication. De plus, comme l'objectif est d'analyser la communication à la frontière de l'interface logicielle/matérielle, nous faisons souvent référence à un type particulier de communication utilisant le bus comme média. Dans ce cas

particulier, nous faisons également l'hypothèse d'un protocole de bus synchrone, c'est à dire cadencé par une horloge. Ceci dit, l'étude des niveaux d'abstraction qui suivra reste assez générale et peut s'appliquer également à d'autres types de médias de communication.

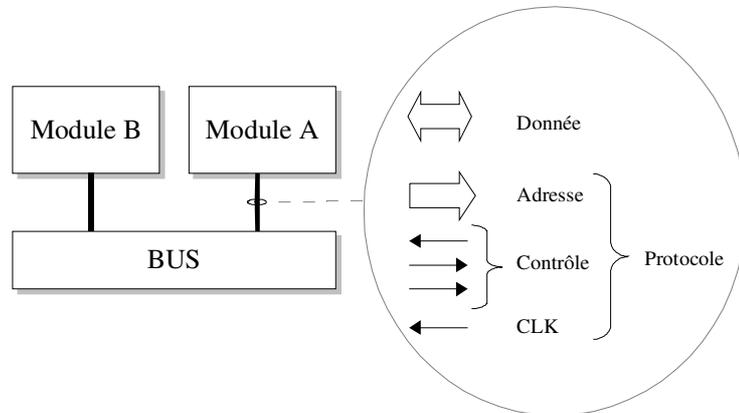


Figure 3.3: Exemple de signaux formant un port logique

Dans ce contexte particulier, une interface de communication est définie comme un ensemble de ports logiques. Un port logique est responsable de la transmission (envoi et/ou réception) d'un type de message via un média donné. Le protocole de communication définit la manière avec laquelle les données sont transmises. La figure 3.3 montre l'exemple particulier d'un protocole de bus.

L'interface de communication peut être modélisée avec un degré de précision variable, ce qui correspond à différents niveaux d'abstraction de cette interface.

3.1.2.2. Niveau RTL

Le niveau RTL (*Register Transfer Level*) est probablement le niveau d'abstraction le plus utilisé pour la modélisation des systèmes matériels [Gup96]. Nous le considérons comme étant le niveau d'abstraction le plus bas dans le cadre de cette thèse, tout en notant que, dans le cas général, il ne l'est pas. En effet, en terme de hiérarchie de niveaux d'abstraction, les niveaux porte (*gate*) et physique lui succèdent.

Au niveau RTL, les données intervenant dans une communication sont sous une forme logique, c'est à dire qu'elles sont représentées par des vecteurs de bits (*std_logic* en VHDL). Une donnée logique est considérée comme une entité cohérente (un seul signal dans la terminologie VHDL) dans le sens où les différents bits du vecteur logique évoluent temporellement ensemble (à la différence du niveau porte).

Dans une communication au niveau RTL, le temps est une grandeur réelle, arbitraire (pouvant avoir n'importe quelle valeur dans \mathbb{R}) et discrète. Ceci permet de modéliser des communications asynchrones et synchrones (comme cas particulier). Pour le cas synchrone (présence d'un signal d'horloge), un modèle RTL permet d'exprimer la phase des différents signaux par rapport à l'horloge.

Ce niveau d'abstraction est supporté dans la majorité des langages HDL en particulier dans VHDL, Verilog et SystemC.

3.1.2.3. Niveau TLM transfert

Le niveau TLM transfert introduit deux abstractions supplémentaires par rapport au niveau RTL. La première concerne le type de données qui devient abstrait et la deuxième concerne le temps qui devient échantillonné c'est à dire aligné sur l'horloge. Une donnée abstraite est représentée par une grandeur mathématique, par exemple un entier naturel. Ainsi un vecteur de 8 bit peut être abstrait par un entier naturel dans l'intervalle $[0,255]$. Cette abstraction introduit évidemment une perte d'information concernant, par exemple, les états haute impédance et indéfini. La deuxième abstraction, n'est valide que pour les systèmes synchrones (existence d'un signal d'horloge). Dans ce cas le signal d'horloge est rendu implicite et est pris comme unité de temps pour exprimer l'évolution temporelle du protocole de communication.

3.1.2.4. Niveau TLM transaction

Au niveau TLM transaction, on abstrait le protocole de communication, c'est à dire le mécanisme assurant la transmission d'une transaction atomique. Cela revient à ignorer les étapes intermédiaires qui aboutissent à cette transmission et ne garder que l'information pertinente (à ce niveau) à savoir la donnée effective (objet de la transaction) ainsi que les instants de début et fin de la transaction. Ainsi, les signaux d'horloge et de contrôle spatiaux ne sont pas représentés à ce niveau.

Pour supporter ce niveau de modélisation, SystemC 2.0 introduit de nouveaux concepts qui ne sont pas disponibles dans d'autres langages (en particulier dans VHDL et SystemC 1.x). Il s'agit de la notion d'interface/canal qui étend la notion classique de signal en permettant de définir des canaux utilisateurs [Ros03][Gro02]. Ces canaux fournissent des services bien définis pouvant être bloquants ou non bloquants. Les services classiques de lecture et d'écriture sur les signaux conventionnels ne sont alors que des cas particuliers de services non bloquants.

Le code de la figure 3.12 illustre un exemple de services de communication bloquants offerts par un modèle transactionnel du bus AHB [Cal03]. Le type du paramètre passé comme premier argument aux deux fonctions est un pointeur sur une zone mémoire qui doit correspondre à la taille d'une transaction atomique sur le bus. Le second argument précise l'adresse physique sur le bus de la

```
AHB_transaction_blocking_send(&data, addr)
```

```
AHB_transaction_blocking_rcv(&data, addr)
```

Figure 3.4: services de communication bloquants au niveau TLM transaction du protocole AHB

donnée (registre) objet de la transaction.

En terme de sémantique, l'appel à ces fonctions dans un processus matériel provoque le blocage du processus jusqu'à ce que l'envoi ou la réception de la donnée est effectivement achevé. En particulier la valeur de la mémoire pointée par *data* est imprévisible durant toute la période de blocage du processus. Cela implique, entre autres, que si un autre processus utilisant la même zone mémoire⁶, accède en lecture ou en écriture pendant le temps de blocage du processus qui a initialement déclenché le transfert, alors le comportement du système est indéterminé. Dans la pratique, on fait appel à des mécanismes de synchronisation qui sont à la charge du concepteur du système (implémentant des sémaphores par exemple) pour résoudre ce type de problème.

```
AHB_transaction_non_blocking_send(&data, addr, notify)
```

```
AHB_transaction_non_blocking_rcv(&data, addr, notify)
```

Figure 3.5: services de communication non bloquants au niveau TLM transaction du protocole AHB

Dans une communication non bloquante (figure 3.11), le processus qui initie le transfert n'est pas forcé à se bloquer jusqu'à l'accomplissement effectif de la communication. Pour cela, un mécanisme supplémentaire au niveau du bus permet d'informer d'une façon asynchrone le processeur de la fin de la transaction. Il faut néanmoins noter qu'un tel mode de communication n'est pas très utilisé au niveau des interfaces conventionnelles de bus. En effet, ceci implique des mesures supplémentaires à prévoir au niveau du logiciel pour assurer la cohérence des données.

3.1.2.5. Niveau TLM message

A ce niveau, le média de communication est abstrait. On parle alors de communication « logique » faisant intervenir différents « agents » échangeant de l'information via un média virtuel. La topologie du média virtuel n'a pas (généralement) de relation avec la topologie du réseau de communication final qui va servir pour l'implémentation réelle. Par analogie au modèle OSI [Tan96] qui établit un standard de communication pour les réseaux informatiques, le niveau TLM message serait équivalent au niveau d'abstraction fourni par la couche « transport » du modèle OSI.

⁶ Par exemple pour une éventuelle communication par DMA

```
fifo_message_send(&data)
fifo_message_rcv(&data)
```

Figure 3.6: Services de communication au niveau TLM message

La figure 3.10 montre deux exemples de primitives de communication au niveau TLM message. Il s'agit de services d'envoi et de réception de messages à travers un canal *fifo* assurant la liaison logique entre un producteur et un consommateur.

Le tableau 1 résume les différentes caractéristiques évoquées plus haut pour chaque niveau d'abstraction de la communication matérielle.

Niveau	Média	Protocole	Donnée	Temps	Abstraction de :
RTL	Explicite	Explicite	Signaux physiques ex : vecteur de 32 bits	discret Asynchrone	Portes logiques/ bascules
TLM transfert	Explicite	Explicite	Signaux logiques ex : entier	Synchrone (aligné à l'horloge)	Signaux/registres déphasages
TLM transaction	Explicite	Implicite	Donnée logique ex : entier	Événementiel opération bloquantes	Protocole de communication (signaux de contrôle et horloge)
TLM Message	Implicite	_	Donnée abstraite ex : structure	Événementiel opération bloquantes	Média

Tableau 1: niveaux d'abstraction de l'interface de communication

3.1.3. Niveaux d'abstraction de l'interface de programmation (API)

3.1.3.1. Notion d'interface de programmation

Par définition, une interface de programmation (en anglais *Application Programming Interface* ou API) spécifie la manière dont un composant informatique peut communiquer avec un autre. Une manière de voir l'interface de programmation est de la confondre avec la vision qu'un programmeur a vis-à-vis de la machine qu'il envisage de programmer. Selon le niveau de détail avec lequel le programmeur voit cette machine, l'interface de programmation varie. Nous parlons de différents niveaux d'abstraction de l'interface de programmation. La figure 3.7 illustre le concept d'interface de programmation. La machine constitue l'intermédiaire entre le programme et son environnement. Elle se charge d'interpréter les « commandes » du programme et d'interagir en conséquence avec les signaux de l'environnement (opérations d'entrées/sorties initiées par la machine ou par l'environnement).

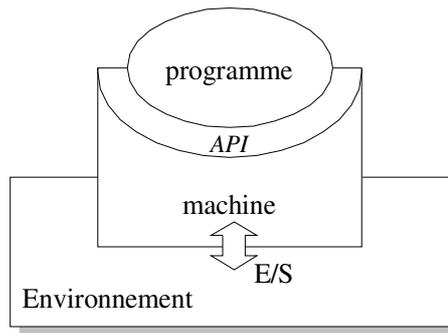


Figure 3.7: notion d'interface de programmation

Étant donné le caractère réactif et concurrent du logiciel embarqué [Jan04], nous définissons le terme programme dans la figure comme étant une suite d'opérations ou encore d'instructions qui peuvent être de deux types différents: calcul et contrôle. Les opérations de calcul permettent d'implémenter un algorithme donné, alors que les opérations de contrôle servent à contrôler l'état de la machine et représentent ainsi l'aspect réactif et concurrent du programme.

L'aspect calcul (ou encore algorithme) fut largement abordé dans le domaine de l'informatique classique à la fois d'un point de vue théorique (machine de Turing, Lambda calculus, etc.) et pratique à travers les langages de programmation. L'aspect contrôle quant à lui, a souvent fait partie de la programmation dite système, une pratique souvent qualifiée de non structurée et de bas niveau.

La figure propose une vue globale des niveaux d'abstraction que nous considérons comme pertinents pour le logiciel embarqué. Pour l'aspect calcul, nous considérons la seule abstraction fournie par les langages de haut niveau HLL (*High Level Language*). Pour l'aspect contrôle, nous considérons les niveaux d'abstraction HAL, OS et COM en se basant sur la structure en couches du logiciel introduite dans le second chapitre. Par abus de langage, nous parlons des niveaux HAL, OS et COM comme niveaux d'abstraction du logiciel entier, sachant que le niveau OS par exemple désigne en réalité le couple (OS, HLL) pour les parties contrôle et calcul respectivement.

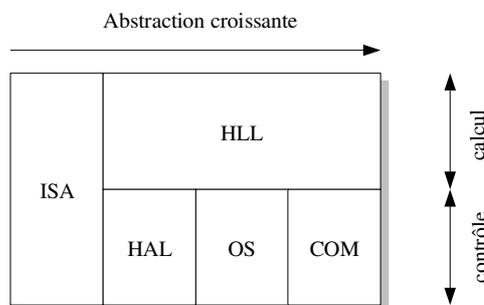


Figure 3.8: Niveaux d'abstraction considérés du logiciel embarqué

A partir du niveau HAL, les opérations de calcul et de contrôle sont explicitement séparées. La partie calcul ou encore algorithme est implémentée par le langage de haut niveau (HLL). La partie contrôle est abstraite via l'API offerte par la machine au niveau d'abstraction considéré. En pratique, cette API est exploitée sous forme d'appels de fonctions à partir du langage de haut niveau.

Dans le cadre de cette thèse, nous considérons C ou C++ comme langage de haut niveau. Dans le cas général néanmoins, d'autres langages peuvent être utilisés fournissant à la fois une abstraction du calcul et du contrôle tels que JAVA (où le parallélisme et la gestion des exceptions par exemple font partie du langage). Cependant, nous estimons que ce type d'approches, qui constitue une sorte d'abstraction universelle, n'est pas la solution de choix dans un domaine fortement contraint par des critères de coût et de performance. En effet, une telle abstraction universelle tend à uniformiser la machine en négligeant les spécificités de celle-ci, des spécificités pouvant être exploitées pour améliorer les performances du système.

Dans la suite, nous examinons d'une manière plus détaillée les particularités de chaque niveau d'abstraction vis-à-vis des concepts développés plus haut.

3.1.3.2. Le niveau ISA

Le niveau ISA (*Instruction Set Architecture*) correspond à la vision la plus basse qu'un programmeur peut avoir vis-à-vis de la machine d'exécution. Ce niveau abstrait l'architecture interne du processeur, en ne gardant que les parties fonctionnelles de celle-ci, comme le banc des registres et les opérations élémentaires fournies par l'unité d'exécution.

L'ensemble des instructions assembleurs formant une ISA donnée peut être vue comme l'API offerte au programmeur à ce niveau. De ce fait, cette API reste très dépendante du processeur utilisé. A ce niveau, les opérations de calcul et de contrôle se confondent bien que l'on puisse, parfois, isoler des instructions qui sont exclusivement dédiées au contrôle ou au calcul (telle que l'instruction de contrôle MSR qui manipule le registre d'état dans un processeur ARM). Dans le cas général néanmoins, la même instruction peut servir les deux types d'opérations (à l'image des instructions de transfert registre/mémoire).

3.1.3.3. Le niveau HAL

Si le niveau ISA est l'abstraction de la vision qu'un programmeur a du processeur, le niveau HAL serait celle que le programmeur a de l'ensemble du sous-système CPU. Par sous-système CPU, nous faisons référence à l'architecture locale d'un noeud logiciel, comprenant, en plus du processeur⁷ lui-même, les autres composants (mémoires, périphériques, bus) nécessaires au fonctionnement du sous-

⁷ Au pluriel dans le cas d'un noeud SMP par exemple

système. Ainsi, le passage du niveau ISA au niveau HAL revient à considérer le sous-système entier comme « machine » au lieu du processeur isolé.

Au niveau HAL, le calcul est encapsulé dans un ensemble de co-routines ou encore *thread*. Chaque co-routine correspond à un chemin d'exécution (flot de contrôle) indépendant [Kah77]. La notion de co-routine est une notion dynamique puisque la suite exacte des instructions logicielles qui seront exécutées ne peut être connue que durant l'exécution à cause des dépendances aux données. Une autre source de changement dynamique du flot de contrôle, qui est indépendant du langage lui-même, est les interruptions matérielles. Au niveau HAL, on doit s'assurer que le mécanisme d'interruption s'effectue dans le contexte d'une autre co-routine et n'interfère donc pas -fonctionnellement- avec la co-routine interrompue. Ainsi, quand la co-routine interrompue reprend le contrôle, elle doit retrouver son contexte initiale tel que définie par l'architecture du processeur (registres du processeur, pile, etc). Dans la pratique, ce mécanisme est généralement assuré par coopération entre le matériel (processeur) et le logiciel (gestionnaire d'interruption). Le changement de contexte peut aussi s'effectuer d'une façon synchrone à partir du code logiciel lui-même (auto changement de contexte). Ce dernier est typiquement implémenté par le système d'exploitation pour gérer le pseudo parallélisme logiciel.

En plus des services de manipulation de contexte, l'API HAL fournit généralement d'autres opérations de contrôle. Le tableau résume quelques services liés au contrôle des interruptions et à la gestion des entrées/sorties. Il faut bien noter que la liste des services présents dans le tableau n'est ni unique ni exhaustive. En particulier, le niveau HAL peut très bien définir d'autres types d'abstraction si besoin est, tels que celle liée à la mémoire (en fournissant le concept de protection de l'espace d'adressage par exemple). Par ailleurs, Il est souvent nécessaire au niveau HAL, de fournir des services spécifiques aux périphériques. Ces services dépendent du périphérique lui-même et ne peuvent pas être listés d'une manière générique. ils concernent généralement la configuration (initialisation) du périphérique en question ou bien la demande d'une action (tel que l'initiation d'un transfert DMA). Les services génériques de lecture/écriture listés dans le tableau permettent d'abstraire les opérations d'entrées/sortie sur des registres.

	<i>Manipulation de contexte</i>	<i>Contrôle des interruptions</i>	<i>E/S génériques</i>
Exemples typiques d'API HAL	- <i>initContext</i> - <i>loadContext</i> - <i>switchContext</i>	- <i>setExceptionHandler</i> - <i>enableInterrupt</i> - <i>disableInterrupt</i>	- <i>read(data,address)</i> - <i>write(data,address)</i>

Tableau 2: Exemples de services au niveau HAL

3.1.3.4. Le niveau système d'exploitation OS

Au niveau OS, les opérations de calcul sont abstraites via le langage de programmation de haut niveau (C/C++), comme pour le niveau HAL. Les opérations de contrôle sont constituées par l'API offerte par le système d'exploitation. Par opposition au niveau HAL où les opérations de contrôle permettent seulement un accès générique aux ressources de l'architecture locale, les opérations de contrôle au niveau OS cachent des politiques de gestion de ces ressources. En effet, le rôle du niveau OS est justement de palier aux imperfections et limites des ressources de l'architecture matérielle en implémentant un certain nombre de politiques pour la gestion de ces ressources limitées. Un exemple de gestionnaire de ressources est l'ordonnanceur qui permet de multiplexer une ressource « rare » -la capacité de traitement- sur l'ensemble des tâches logicielles actives à un instant donné.

La notion de tâche logicielle joue un rôle important à ce niveau puisque elle encapsule les opérations de calcul et les opérations de contrôle. Selon le type du système d'exploitation, une tâche peut être une entité plus ou moins complexe. En particulier, elle peut représenter un processus lourd ou léger (*thread*). Vu que nous avons séparé le concept de co-routine du concept de protection mémoire au niveau HAL, il est facile d'exprimer cette variété de types de tâches en jouant simplement sur la combinaison de co-routines et d'espace d'adressages.

Les services offerts par l'API du système d'exploitation dépendent largement de celui-ci. A ce stade, comme il n'y a pas une API universelle, ces services ne peuvent pas être présentés d'une façon absolue. On peut néanmoins classer certains services dans des catégories tels que les services de communication et de synchronisation inter-tâches, les services de communication externes via les pilotes de périphériques, et les services d'allocation de ressources (mémoire, *timer*, etc.).

3.1.3.5. Tableau récapitulatif:

niveau	langage	ressources	unité d'exécution	E/S (environnement)	Abstraction	
					de	fournit
ISA	Assembleur	physique	Processeur	- Ports - Registres mappés en mémoire - Interruptions	La micro-architecture	Ressources : - de calcul - de stockage
HAL	HLL (C)	logiques	Contexte	Services de communication bas niveau (send/rcv)	implémentation physique	Ressources logiques
OS	HLL (C,C++...)	virtuelles	Tâche	Pilotes	l'imperfection / rareté	Ressources virtuelles (idéales)

Tableau 3: Tableau récapitulatif des niveaux d'abstraction de l'interface de programmation

3.2. Validation de l'interface logicielle/matérielle

Comme le terme validation peut avoir des significations différentes selon le contexte où il est placé, nous commençons par définir et restreindre ce qu'on entend par ce terme dans le contexte de cette thèse. Partant d'une spécification fonctionnelle de l'application et d'une architecture donnée, l'objet de la validation est de s'assurer que le système produit par le processus d'implémentation de l'application sur l'architecture répond bien à des critères fixés à l'avance. Ces critères concernent généralement deux aspects.

- ◆ l'aspect fonctionnel : il s'agit de s'assurer que le système obtenu réalise bien les fonctions initiales de la spécification fonctionnelle. Ceci inclut, pour les systèmes temps réel, le respect des échéances temporelles des différentes tâches. Un résultat « négatif » à ce stade aboutit généralement à une re-considération du processus de raffinement de l'application et/ou une modification/re-conception de l'architecture.
- ◆ l'aspect performance : il s'agit d'évaluer les performances du système vis à vis de certaines grandeurs tels que le débit, la consommation d'énergie, etc. Dans le cas où les performances obtenues sont insuffisantes, on a généralement recours à des « optimisations » au niveau de l'architecture et/ou un partitionnement différent des fonctions de l'application sur les éléments de l'architecture. On parle généralement du processus d'exploration d'architecture.

Cette définition de la validation nécessite quelques clarifications :

- ◆ Dans notre cas, la validation concerne l'étape d'intégration ou de raffinement application/architecture. En ce sens, elle n'inclut pas le test/vérification de l'architecture elle-même ou de l'un de ses composants. Ainsi, nous partons de l'hypothèse que l'architecture (logicielle et matérielle) est « bien conçue ».
- ◆ L'expression « s'assurer que » est générique et donc peu précise. En effet, conventionnellement, il existe deux approches à ce stade : la simulation et la vérification formelle. La simulation consiste à disposer d'un modèle exécutable du système que l'on exécute (sur un ordinateur) avec une instance particulière des données à l'entrée du système. Par observation des résultats générés par l'exécution (trace du système) et en comparant ces résultats par rapport à ceux fournis par la simulation fonctionnelle, on peut décider quant à la validité de l'implémentation (transposition application/architecture). L'inconvénient classique de la simulation est que cette décision reste valable pour une entrée particulière (plus généralement une configuration particulière de l'environnement du système). Pour couvrir tous les cas possibles, il faut répéter la simulation autant de fois qu'il y a de configurations possible de l'environnement. Même pour des systèmes

simples, ceci reste généralement inconcevable vu le grand nombre des valeurs possibles des entrées. Généralement, on se contente de quelques échantillons représentatifs de ces entrées [Kjh03].

La vérification formelle apporte une solution plus systématique est fiable au problème de validation (du moins concernant le premier aspect fonctionnel) [Lem02]. Généralement, il s'agit de transformer et la spécification fonctionnelle initiale et l'implémentation architecturale finale en des représentations mathématiques qui se prêtent à des manipulations formelles visant à prouver certaines propriétés pertinentes du systèmes. Malheureusement, cette approche se heurte, pour des systèmes relativement grands, au problème d'explosion d'état [Dou01]. Ce problème rend le traitement automatique du processus de vérification formelle par une machine ayant des ressources finies (mémoire, vitesse) une tâche pratiquement impossible. Ainsi, la vérification formelle se trouve généralement limitée à des systèmes de petites/moyennes tailles ou à quelques parties critiques d'un système plus large.

Dans le cadre de cette thèse, nous nous intéressons à la simulation pour deux raisons essentielles. La première concerne la taille importante des systèmes MPSoC aux quels nous avons affaire. La deuxième vient du fait que nous sommes également intéressés par l'aspect performance et exploration d'architecture, un aspect qui, à notre connaissance, est uniquement adressé par la simulation.

3.2.1. simulation multi-niveaux de l'interface logicielle/matérielle

Valider un système mixte logiciel/matériel par simulation nécessite des modèles exécutables de la partie logicielle, de la partie matérielle et de l'interface (ou encore la machine) entre ces deux parties. Cette validation peut se faire à différents niveaux d'abstraction correspondant à différentes situations et/ou scénarios de développement. La figure 3.9 reprend les différents niveaux d'abstraction des interfaces logicielle et matérielle introduits dans la première partie du chapitre. Théoriquement, chaque couple associant un niveau d'abstraction du matériel avec un autre du logiciel peut représenter un niveau de validation donné. Dans la figure, seules les possibilités de simulation disponibles en utilisant les méthodes conventionnelles sont représentées (indiquées par des traits qui joignent un niveau d'abstraction donné du matériel avec un autre niveau du logiciel).

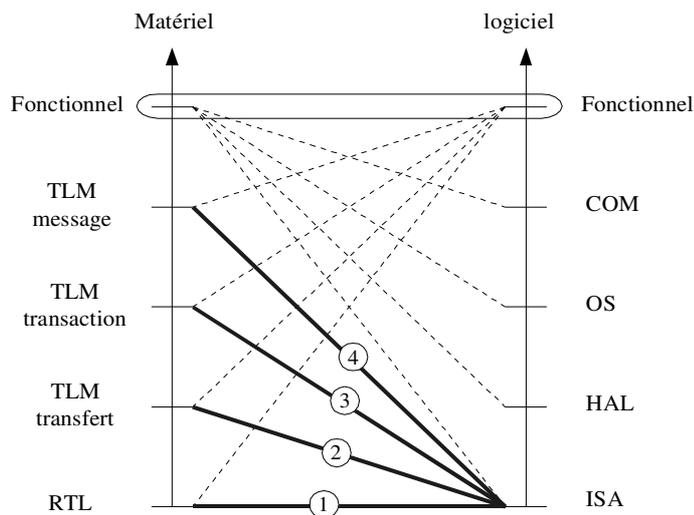


Figure 3.9: Niveaux classiques de validation de l'interface logicielle/matérielle

En toute vigueur, au niveau fonctionnel, la notion de logiciel/matériel ne doit pas exister, car il s'agit d'une notion relative à l'implémentation. Cependant, pour la clarté de la représentation, nous dupliquons le niveau fonctionnel d'une coté comme de l'autre dans la figure. L'environnement Ptolemy [Hyl03] est souvent considéré comme la référence en terme de simulation fonctionnelle combinant différents modèles de calcul [Jan05]. Mis à part ce niveau fonctionnel, tous les autres niveaux sont des niveaux d'implémentation ou encore niveaux architecturaux.

Par définition, la cosimulation logicielle/matérielle implique la mise en correspondance de deux niveaux dont au moins un est architectural. Dans le cas particulier où un niveau architectural est associé à un niveau fonctionnel, on parle de cosimulation partielle (dénotée par des traits fins pointillés dans la figure). Lorsque les deux niveau sont fonctionnels, on retrouve la simulation fonctionnelle. Les traits continus forts correspondent au cas classiques de cosimulation logicielle/matérielle tels que proposés par les approches conventionnels. Remarquons que ces approches se basent exclusivement sur le niveau ISA du coté du logiciel. L'utilisation des niveaux TLM pour le matériel est assez récent. Historiquement c'est le niveau RTL qui fut utilisé comme niveau de référence pour la cosimulation logicielle/matérielle à coté du niveau ISA.

3.2.2. Approches existants pour la cosimulation logicielle/matérielle

Nous abordons l'étude des approches de cosimulation logicielle/matérielle par quelques méthodes de simulation partielle qui se sont focalisées séparément sur le logiciel et le matériel. Ceci nous permettra de mettre en relief les limites de telles approches dans le cadre d'une validation globale du système, ce qui motivera les approches de type globales qui abordent le système dans sa totalité.

3.2.2.1. Simulation partielle des parties logicielles et matérielles

Pendant longtemps, la conception et la validation des SoC était menée indépendamment en ce qui concerne les parties logicielles et matérielles. Chaque partie est développée séparément par des équipes spécialisées et ce n'est que pendant l'étape finale d'intégration que le logiciel est embarqué sur le prototype matériel. A ce moment, le fonctionnement global du système est testé et les performances qui en résultent sont précisément évaluées. Cette intégration tardive à plusieurs inconvénients qui ont été détaillés dans le premier chapitre. Dans ce paragraphe, nous passons en revue les techniques de simulation utilisées par cette approche.

1) Simulation partielle de la partie matérielle

Le développement des environnements de modélisation matérielle ont rendu possible la simulation de larges systèmes. Ces environnements sont basés sur des langages de description du matériel (HDL Hardware Description Languages) tels que VHDL [Iee02] et Verilog [Iee01] permettant la modélisation des systèmes matériels à différents niveaux d'abstraction.

Ainsi l'architecture matérielle d'un SoC est entièrement développée et simulée en utilisant ce type d'environnements. Comme le logiciel embarqué n'est pas disponible, on utilise généralement des générateurs de trafic fonctionnels qui permettent simplement de l'émuler.

2) Simulation partielle de la partie logicielle

Pour valider le logiciel applicatif, les programmeurs ont souvent recours à des émulateurs de systèmes d'exploitation. Il s'agit de programmes natifs (tournant sur la machine hôte) qui émulent le comportement du vrai système d'exploitation et fournissent la même interface de programmation (API) que celui-ci. Les exemples d'émulateurs de systèmes d'exploitation incluent CarbonKernel, un émulateur générique [CarbonK] et VxSim, l'émulateur du système d'exploitation commercial VxWorks [Vxsim]. Cependant, comme l'architecture du SoC final est différente de celle de la machine hôte, la simulation reste purement fonctionnelle et ne fournit aucune indication sur les performances de l'application. Par ailleurs, l'environnement matériel est souvent remplacé par des modèles fonctionnels simplifiés effectuant des opérations d'entrées/sorties standards.

3.2.2.2. Utilisation d'un modèle de processeur

Une solution pour simuler conjointement les parties matérielles et logicielles d'un SoC est d'utiliser un modèle matériel du (des) processeur(s) au sein de l'environnement matériel global. Un modèle de processeur abstrait la microarchitecture du processeur physique en fournissant d'un côté une interface de programmation au niveau ISA, et de l'autre côté l'interface au niveau transfert du bus de communication. Il s'agit d'un modèle classique qui a été largement couvert dans la littérature

[Raj99]. Dans ce paragraphe, on s'intéresse à ce modèle en tant que première représentation de l'interface logicielle/matérielle au plus bas niveau d'abstraction considéré.

1) *Abstraction de la micro-architecture du processeur*

Différents travaux de recherche se sont intéressés à la spécification d'un modèle de processeur, notamment à travers les langages de description d'architecture (ADL pour *Architecture Description Language*) tels que nML [Fau95] et LISA [Ziv96]. Historiquement, ces langages de description d'architecture ont été classés selon différentes catégories : comportementales, structurels ou hybrides. Cette décomposition correspond en fait à plusieurs utilisations différentes du langage en question. La figure 8 montre différents cas de figure de l'utilisation d'un ADL. Ces cas de figure couvrent la génération du simulateur de jeu d'instruction (ISS), la génération d'un compilateur spécifique au processeur et la synthèse de la micro-architecture du processeur en question.

Certains langages tels que LISA peuvent servir à plusieurs types de génération à la fois. Cela signifie que le langage possède plusieurs sémantiques. Cependant, il s'agit souvent de sémantiques qui sont implicitement définies par les outils de génération.

Fondamentalement, un ADL fournit une abstraction de la micro-architecture du processeur. Ceci inclut le chemin de données interne du processeur ainsi que l'implémentation des différentes instructions binaires. Par opposition, il explicite les ressources fonctionnelles d'un point de vue utilisateur (outil) tels que les registres internes et les opérations qui sont implémentées par le jeu d'instruction. Dans le cas où le langage est destiné à générer un simulateur au niveau cycle du processeur, on a aussi besoin de modéliser le pipeline du processeur pour tenir compte du nombre de cycles nécessaires à l'exécution des instructions.

Dans le cadre de cette thèse, nous considérons l'unité d'exécution au niveau ISA comme une boîte noire qui correspond pratiquement au simulateur de jeu d'instruction ISS. Nous faisons néanmoins l'hypothèse que l'ISS a un comportement précis au niveau cycle.

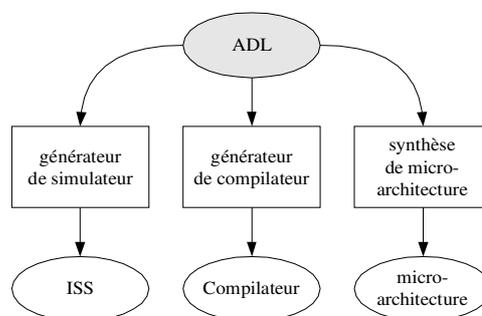


Figure 3.10: Flot de génération à partir d'un ADL

2) *Cosimulation avec modèle HDL du processeur*

Il s'agit généralement d'un modèle RTL, décrivant l'architecture interne du processeur et fournissant une interface fidèle à celle du processeur physique. L'avantage de cette solution est qu'elle permet d'avoir une simulation précise et fidèle au système réel, ce qui éliminera le besoin d'avoir un prototype matériel (souvent cher) qui ne sera peut être pas le bon en fin de compte. Cependant l'approche présente deux limites majeurs. La première est qu'elle intervient tard dans le cycle de conception puisque l'architecture matérielle ainsi que le logiciel doivent être complètement conçu. A ce stade, un nouveau cycle de conception suite à un résultat non satisfaisant est également coûteux. La deuxième est que la vitesse de simulation d'un tel modèle reste très réduite, surtout pour des systèmes multiprocesseurs assez complexes (de l'ordre de quelques dizaines de hertz). Ceci réduit considérablement le pouvoir de validation et rend très difficile l'exploration efficace de l'espace architectural.

3) *Cosimulation avec un simulateur du jeux d'instruction*

Pour remédier au problème de vitesse dans l'approche précédente, le modèle RTL du processeur est remplacé par un modèle fonctionnel appelé simulateur du jeux d'instruction (ISS pour *Instruction Set Simulator*) [Mentor]. Il s'agit généralement d'un programme (C/C++) qui interprète séquentiellement les instructions binaires et simule leurs exécution à la manière du processeur cible, sans pour autant modéliser l'architecture interne de celui-ci comme pour le modèle HDL. Cette abstraction des détails de l'architecture interne du processeur permet d'atteindre des vitesses de simulation beaucoup plus importantes que dans l'approche précédente. Classiquement, l'interaction entre l'ISS et le simulateur HDL s'effectue via des mécanismes de communication inter-processus (IPC) comme illustré dans la figure 3.8. Pour cela, on exploite généralement la capacité du simulateur HDL à communiquer avec l'environnement externe via des interfaces spécifiques (*Foreign Language Interface FLI* pour VHDL et *Program Language Interface PLI* pour Verilog).

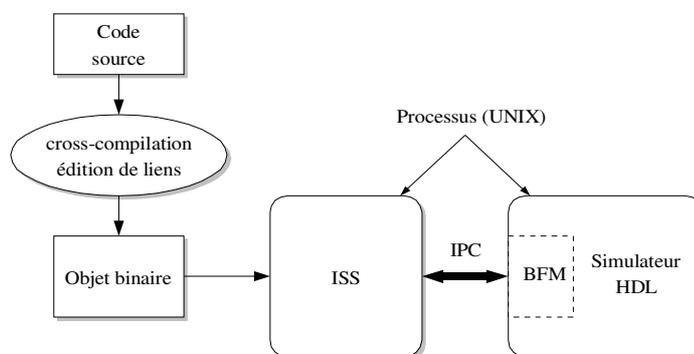


Figure 3.11: Principe de la cosimulation à base de ISS

Récemment, l'émergence d'environnements HDL qui utilisent C/C++ comme langage de modélisation matérielle (SystemC [Systemc], SpecC [Specc]) a rendu possible de s'affranchir du surcoût lié à la communication par IPC [Ben03]. Dans ce type d'environnement, l'ISS peut être compilée comme étant une bibliothèque externe et liée à l'exécutable du simulateur au sein d'un seul processus (UNIX). Ainsi une communication IPC est remplacée par un simple appel de fonction qui coûte beaucoup moins cher en temps d'exécution.

3.2.2.3. *Approches haut niveau pour la cosimulation*

L'utilisation d'un modèle à base de ISS a constitué une percée importante en terme de capacité de cosimulation logicielle/matérielle par rapport aux approches utilisant un modèle HDL du processeur. Cependant, bien qu'elle évite le passage prématuré à un prototype matériel, cette approche présente toujours l'inconvénient d'intervenir tard dans le cycle de conception, c'est à dire une fois l'architecture logicielle et matérielle du système est fixée et complètement développée. Le besoin croissant de pouvoir effectuer la validation et l'exploration des choix architecturaux plus tôt dans le cycle de conception, a récemment poussé vers la mise au point d'approches qualifiées de systèmes -ou encore de haut niveau- permettant la cosimulation d'un système logiciel/matériel tôt dans le cycle de conception.

1) *Utilisation d'un modèle de simulation du Système d'exploitation*

L'utilisation d'un système d'exploitation (temps réel) constitue une étape importante dans la conception et le raffinement du logiciel embarqué. Partant d'un modèle fonctionnel de l'application, l'idée derrière l'utilisation d'un modèle de simulation du système d'exploitation est de pouvoir évaluer les effets liés à l'exécution d'une partie des tâches de l'application à un système d'exploitation [Yoo02]. Ceci signifie qu'on passe d'un modèle représentant un vrai parallélisme (initialement les tâches sont exécutées par des processus « matériels ») à un modèle où le parallélisme n'est que simulé par l'ordonnanceur du système d'exploitation (pseudo-parallélisme). La figure illustre un tel passage où l'on considère un système de trois tâches indépendantes (sans interactions/communications directes entre elles).

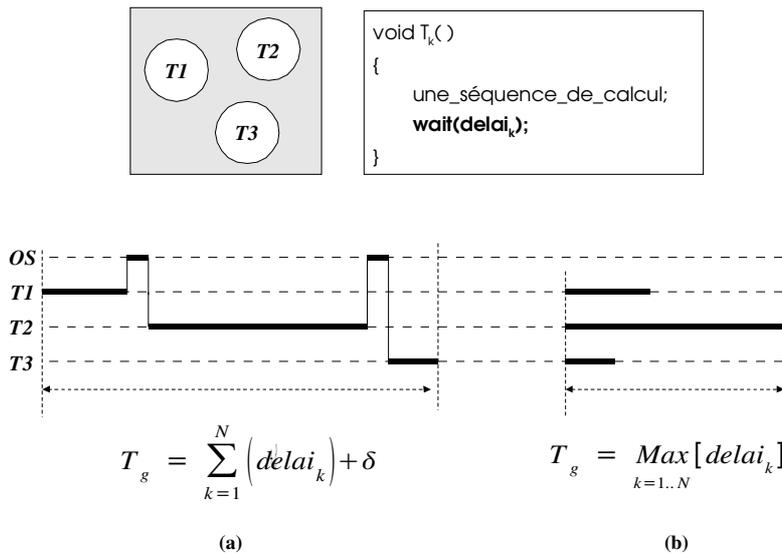


Figure 3.12: Trace d'exécution d'un système de trois tâches indépendantes
 (a) sur un processeur réel avec un système d'exploitation
 (b) dans un modèle de simulation purement fonctionnel

Dans la description fonctionnelle de la figure, chaque tâche est un processus qui exécute - dans un temps nul - une séquence d'opérations de calcul suivie par une instruction *wait(delay)* modélisant le délai associé à ce calcul. En réalité, il s'agit là d'une description où l'on essaie déjà de modéliser les performances du système en introduisant des délais supplémentaires qui ne font pas partie, en principe, de la spécification fonctionnelle initiale du système.

Dans un environnement de simulation de haut niveau tel que SystemC, la trace de simulation du système serait celle illustrée par la figure 2-b. Étant donné la nature matérielle du parallélisme, les tâches seront exécutées simultanément et le temps global à la fin de la simulation coïncidera avec le délai de la tâche la plus lente. La figure 2-a montre un cas de trace d'exécution du même système de tâches, cette fois sur un processeur réel géré par un système d'exploitation. Ceci correspond à une implémentation purement logicielle de la spécification fonctionnelle. Ici, les tâches sont exécutées en pseudo-parallélisme (séquentiellement comme cas particulier) et le temps total d'exécution sera la somme des temps d'exécution de chaque tâche augmenté du surcoût lié au système d'exploitation (commutation, algorithme d'ordonnancement etc). Cet exemple permet de souligner l'importance de l'écart qui peut être observé entre un modèle fonctionnel et un modèle architectural et ce même pour des cas simples. Pour des exemples plus réels, il faut aussi considérer les éventuelles interactions entre tâches ainsi que la communication avec l'environnement matériel externe.

Pour tenir compte de ces effets architecturaux, un premier niveau de modélisation serait de pouvoir évaluer, le plus tôt dans le flot de conception, les conséquences du choix du système

d'exploitation sur le comportement du système. Parallèlement aux travaux menés dans le cadre de cette thèse concernant ce premier niveau de modélisation, différents autres efforts de recherche ont adressé le problème de la modélisation des systèmes d'exploitation dans les environnements de simulation de haut niveau. Parmi ces travaux, nous pouvons citer [Des00], où un environnement propriétaire appelé SoCOS est proposé comme solution permettant la cosimulation haut niveau des systèmes mixtes logiciels/matériels. Cet environnement permet de décrire au plus haut niveau un système comme un ensemble de processus concurrents. Les processus peuvent être de différentes natures (asynchrones, synchrones ou réactifs) et supportent la création dynamique. Partant de ce modèle fonctionnel, le système est raffiné vers un modèle architectural où une partie des processus est implémentée en logiciel. Ce raffinement est accompagné par une étape de génération de code logiciel qui « ré-écrit » les processus logiciels en utilisant une API propriétaire d'un éventuel système d'exploitation. Finalement, le système ainsi obtenu peut être cosimulé en utilisant un modèle de simulation du système d'exploitation qui émule l'API en question et gère aussi la communication et la synchronisation avec le reste du système. Malheureusement, les auteurs ne fournissent pas suffisamment de détails concernant cette dernière partie. Par conséquent, le modèle de simulation utilisé ainsi que les mécanismes de synchronisation ne sont pas connus.

Dans [Gon02][Mad03], les auteurs utilisent une approche différente. L'environnement SystemC, utilisé comme environnement de base pour la modélisation haut niveau, est enrichi par une extension lui permettant de modéliser l'ordonnancement des tâches logicielles. La modélisation du temps d'exécution du logiciel n'est pas effectuée par annotation du code (comme pour le cas de l'exemple de la figure), mais par inclusion d'un paramètre correspondant au pire cas (*worst case execution time* WCET) dans le modèle de la tâche lui-même. La synchronisation avec l'environnement matériel se fait périodiquement sur la base d'une horloge interne, ce qui introduit une erreur systématique par rapport à la sémantique événementielle de SystemC et induit un surcoût important en terme de vitesse de simulation (rapport entre la durée séparant les événements significatifs du système et le pas de synchronisation utilisé).

Une autre approche se basant sur l'environnement SystemC est celle proposée par [Moi04]. Par opposition à l'approche précédente, la modélisation du temps d'exécution du logiciel se fait par annotation directe du code des tâches ce qui permet d'être précis vis à vis du flot de contrôle de celle-ci. Le modèle de l'ordonnanceur est lui-même décrit comme une extension à la bibliothèque de base de SystemC et permet de modéliser différents types d'ordonnancement. Cependant le modèle reste dédié au problème de l'ordonnancement des tâches logicielles et fait abstraction des mécanismes de communication et de synchronisation avec le matériel.

Partant de SpecC comme environnement de base, une approche similaire a été proposée par

[Ger03]. Pour décrire une tâche logicielle, l'approche fait appel à une API générique de système d'exploitation, qui inclut des primitives de communication et de synchronisation inter-tâches. Le modèle supporte aussi l'interaction avec l'environnement matériels via les opérations d'entrées/sorties et les interruptions. Pour modéliser l'ordonnancement des tâches logicielles dans le modèle du système d'exploitation, les auteurs utilisent l'ordonnancement natif des processus « matériels » en SpecC. Ceci passe par la mise en place de mécanismes supplémentaires pour imposer qu'une seule tâche logicielle s'exécute à la fois (dans le cas monoprocesseur). Ces mécanismes introduisent un surcoût considérable au niveau de la simulation et limitent la flexibilité du modèle. Par opposition à ce modèle, nous utilisons le modèle d'ordonnancement hiérarchique qui permet de s'affranchir de ces inconvénients. Ce modèle sera détaillé dans le quatrième chapitre.

2) *L'environnement VCC de Cadence*

VCC [Vcc] est un outil d'exploration d'architecture qui permet de tester différentes configurations du mapping application/architecture tout en restant à un niveau suffisamment haut. La figure montre un exemple de correspondance application/architecture. Les différentes tâches de l'application peuvent être mappées indifféremment sur les composants de l'architecture. La spécification de l'application est basée sur un modèle CFSM (Communicating Finite State Machine). La description de l'architecture utilise, quant à elle, un modèle de haut niveau qui abstrait aussi bien l'architecture matérielle que logicielle. Une fois le mapping fait, une simulation de haut niveau peut être lancée permettant d'avoir une estimation des performances du système. Pour cela, les éléments de l'architecture doivent contenir des modèles de performance appropriés. Par ailleurs le code des tâches qui seront exécutées en logiciel doit contenir des annotations statiques qui indiquent une estimation du temps d'exécution de celui-ci sur le processeur cible.

Malgré l'idée novatrice qu'il y a derrière, VCC n'a pas connu un véritable succès et a été finalement abandonné. Les causes d'un tel échec ont été ramenées à la difficulté de l'utilisation de l'environnement à la fois pour les concepteurs du logiciel et du matériel. Par ailleurs, on a aussi reproché à VCC de ne pas contenir un lien clair à l'implémentation. Nous estimons qu'à l'origine de ces problèmes, il y a le fait que VCC mélange plusieurs niveaux d'abstraction dans une même représentation de l'architecture. Cette confusion au niveau des niveaux d'abstraction entraîne des modèles d'architectures qui sont difficilement compréhensibles d'une part, et pour lesquels il faut gérer un grand nombre de paramètres. En effet, les niveaux d'abstraction étant mélangés, les paramètres relatifs à chaque niveau vont s'additionner au sein d'un même modèle ce qui le rend compliqué et parfois surdimensionné par rapport aux besoins du concepteur, celui-ci ne disposant pas forcément de tous les paramètres au niveau où il se place.

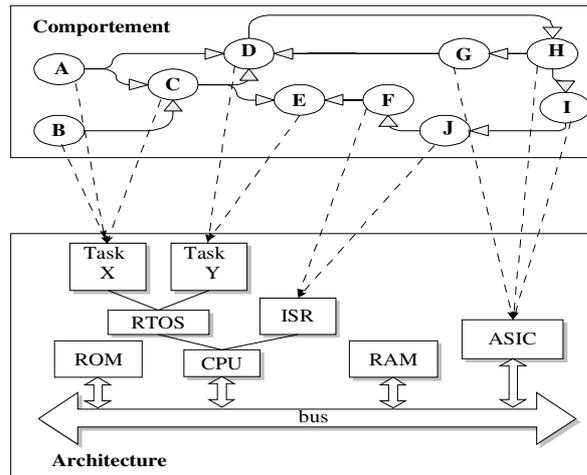


Figure 3.13: Correspondance fonction/architecture dans VCC

3.2.2.4. Estimation du temps d'exécution du logiciel

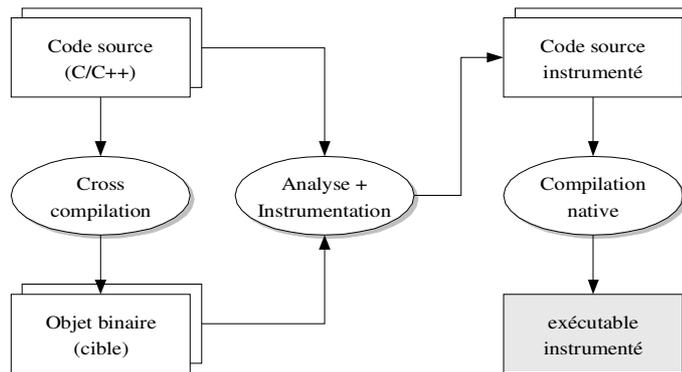
Classiquement [Giu01][Her00], les méthodes d'annotation du code logiciel peuvent être classées en deux catégories : statique et dynamique. Les méthodes statiques se basent sur l'instrumentation du logiciel avant qu'il puisse être exécuté. Cette instrumentation consiste à introduire explicitement et à des endroits précis du code, des instructions (généralement des appels à des fonctions) qui, une fois exécutées, traduisent l'écoulement du « temps logiciel » (par exemple en incrémentant à chaque fois un compteur global). Les méthodes dynamiques, quant à elles, ne nécessitent pas une instrumentation en amont du code logiciel. L'évaluation du temps d'exécution du logiciel se fait dynamiquement au fur et à mesure de l'exécution de celui-ci. Ces méthodes exploitent généralement des aspects particuliers du langage de programmation.

1) Annotation statique

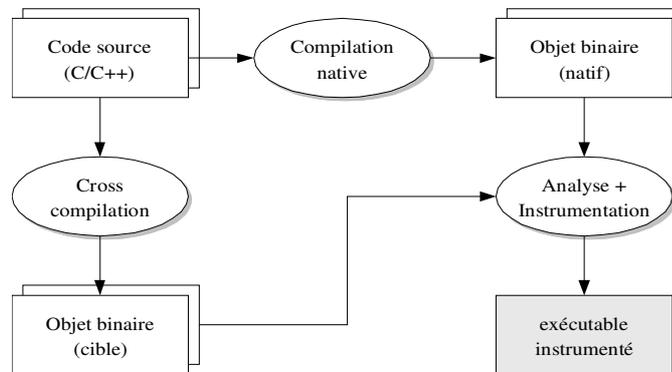
Annoter ou instrumenter statiquement un code logiciel peut être fait à deux niveaux différents : au niveau du code source de celui-ci, ou bien au niveau du code binaire résultant de la phase de compilation [Bra01][Laj99]. La figure 3.1 montre la différence entre ces deux techniques d'instrumentation. Dans les deux cas, le code source est tout d'abord compilé pour le processeur cible (*cross compilation*) avant d'alimenter une des deux entrées de l'outil d'analyse et d'instrumentation. L'autre entrée de l'outil provient du code source lui-même dans le cas d'une instrumentation de type code source (figure 3.1-a) ou de l'objet binaire obtenu par compilation native dans le cas d'une instrumentation du code binaire (figure 3.1-b).

L'outil d'analyse et d'instrumentation procède à l'analyse du flot de contrôle dans la description

logicielle pour déterminer les blocs de base qui feront ensuite l'objet d'une estimation du temps d'exécution. Cette estimation peut se faire de différentes manières, la plus simple étant de consulter une base de données relative au processeur cible et qui contient les délais associés à chaque type d'instruction. Cependant, dans le cas où ces délais ne sont pas connus à l'avance (cas d'un processeur à pipeline complexe), une valeur au pire cas WCET est alors considérée. Une autre méthode consiste à procéder à l'exécution du bout de code logiciel sur un simulateur de processeur pour en déterminer le délai associé.



(a) Instrumentation du code source



(b) Instrumentation du code binaire

Figure 3.14 : Les deux techniques d'instrumentation statique

2) Estimation dynamique

Les méthodes d'estimation dynamique du temps d'exécution du logiciels consistent à déterminer à la volée et au fur et à mesure de l'exécution du code les délais associés à celui-ci. Elles se basent sur la possibilité, dans certains langages de programmation tel que C++, de surcharger des opérateurs standards du langage tels que l'affectation, les opérations arithmétiques et logiques etc. Ces

opérateurs sont alors redéfinis pour contenir, en plus de l'opération elle-même, une fonction d'estimation du délai associée à cette opération, qui une fois exécutée, permet d'incrémenter le temps total d'exécution [Pos04].

Les avantages de cette méthodes résident essentiellement dans sa facilité d'utilisation et la non intrusion vis-à-vis du code initial de l'application (mécanisme totalement transparent pour l'utilisateur). Cependant, elle souffre de plusieurs inconvénients dont la dépendance à une caractéristique particulière et non universelle du langage (C++), mais surtout la difficulté de tenir compte des optimisations introduites par le compilateur natif ce qui en fait une méthode assez approximative. Par ailleurs vue la granularité à la quelle on opère, cette méthode introduit également un ralentissement important de la vitesse de simulation (elle reste quand même plus rapide que le modèle interprété).

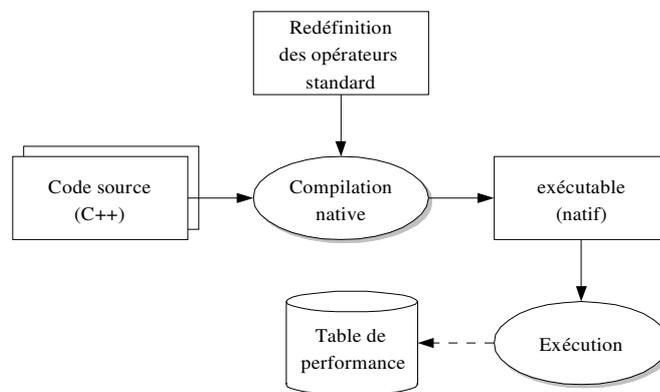


Figure 3.15 : Principe de l'instrumentation dynamique

3.3. Conclusion

Conventionnellement, les parties logicielles et matérielles d'un système MPSoC sont conçues, validées et déployées séparément. Pour cela, dans les deux camps, les concepteurs font appel à des modèles représentant partiellement chaque partie d'architecture à différents niveaux d'abstraction. L'intégration des architectures logicielle et matérielle se fait ainsi à la fin du cycle de conception et fait appel à des modèles de représentation de bas niveaux permettant la validation de l'ensemble du système, mais à un coût élevé.

Récemment, des modèles de représentation d'architectures mixtes logicielles/matérielles ont vu le jour. Ces modèles permettent d'évaluer, aussi tôt dans un flot de conception, certains choix architecturaux en évaluant leurs conséquences sur les performances globales du système.

Cependant, la plupart de ces travaux tendent à se focaliser sur un type particulier d'architecture (le plus souvent la partie logicielle) en faisant des hypothèses simplificatrices (voire simplistes) sur l'autre partie. Ainsi l'intégration des modèles proposés par ces approches dans le cadre du système entier reste ambiguë.

Chapitre 4

Méthodologie multi-niveaux pour la cosimulation de l'interface logicielle/matérielle

Sommaire

Chapitre 4	
Méthodologie multi-niveaux pour la cosimulation de l'interface logicielle/matérielle.....	65
4.1.Méthodologie proposée.....	66
4.1.1.Vue globale des étapes du flot.....	66
4.1.2.Architecture virtuelle.....	68
4.1.3.Prototype virtuel.....	68
4.1.4.Micro-architecture.....	69
4.2.Exécution native du logiciel embarqué dans un simulateur matériel à événements discrets....	71
4.2.1.Problèmes liés à l'exécution native du logiciel.....	72
4.2.2.L'environnement SystemC.....	74
4.2.3.Estimation du temps d'exécution du logiciel.....	77
4.2.4.Synchronisation entre le logiciel et le matériel.....	81
4.3.Modèles de simulation de l'interface logicielle/matérielle.....	87
4.3.1.Modèle de l'architecture virtuelle.....	87
4.3.2.Modèle du prototype virtuel.....	94
4.3.3.Algorithme de la fonction Synch().....	99
4.4.Conclusion.....	101

Dans ce chapitre, nous décrivons la méthodologie proposée dans le cadre de cette thèse pour remédier à la discontinuité observée dans les flots classiques de conception et de validation des systèmes MPSoC. Nous introduisons les concepts d'architecture virtuelle et de prototype virtuel comme étant des étapes intermédiaires dans le flot de conception permettant la validation, par cosimulation globale, des choix architecturaux résultant du raffinement graduel du système. Pour bénéficier de l'avantage d'une simulation rapide à ces niveaux intermédiaires, nous utilisons l'exécution native comme mode d'exécution du logiciel embarqué. Ce mode d'exécution doit, néanmoins, permettre une évaluation suffisamment précise des performances du système dans le contexte d'un simulateur matériel, où la notion de temps joue un rôle important. Pour cela, il faut tenir compte de deux aspects : l'aspect temps d'exécution du logiciel et l'aspect synchronisation entre le « monde du logiciel » et le « monde du matériel ».

Dans la deuxième partie du chapitre, nous détaillons les deux modèles conceptuels servant à décrire l'architecture virtuelle et le prototype virtuel respectivement. Pour chaque modèle, les concepts génériques qui entrent dans la définition du niveau d'abstraction en question sont dégagés et sont distingués des détails d'implémentation, considérés comme non importants à ce niveau. Ces concepts génériques constituent alors les éléments du modèle. Chaque élément du modèle conceptuel est également associé à un modèle de simulation qui définit sa sémantique d'exécution dans l'environnement de simulation SystemC.

4.1. Méthodologie proposée

La méthodologie proposée par cette thèse pour la cosimulation multi-niveaux de l'interface logicielle/matérielle repose sur un flot générique de conception qui se détache des approches classiques par la définition d'étapes intermédiaires entre la spécification initiale et la réalisation finale. Plus précisément, les concepts d'architecture virtuelle et prototype virtuel sont introduits pour assurer une continuité dans les modèles de représentation de l'interface logicielle/matérielle. L'approche proposée reste générique dans le sens où elle est indépendante de l'environnement de conception utilisé et peut s'appliquer aux différents flots existants.

4.1.1. Vue globale des étapes du flot

La figure 4.1 donne un aperçu général sur le flot proposé. Les éléments en trait fort correspondent aux outils et modèles développés dans le cadre de cette thèse.

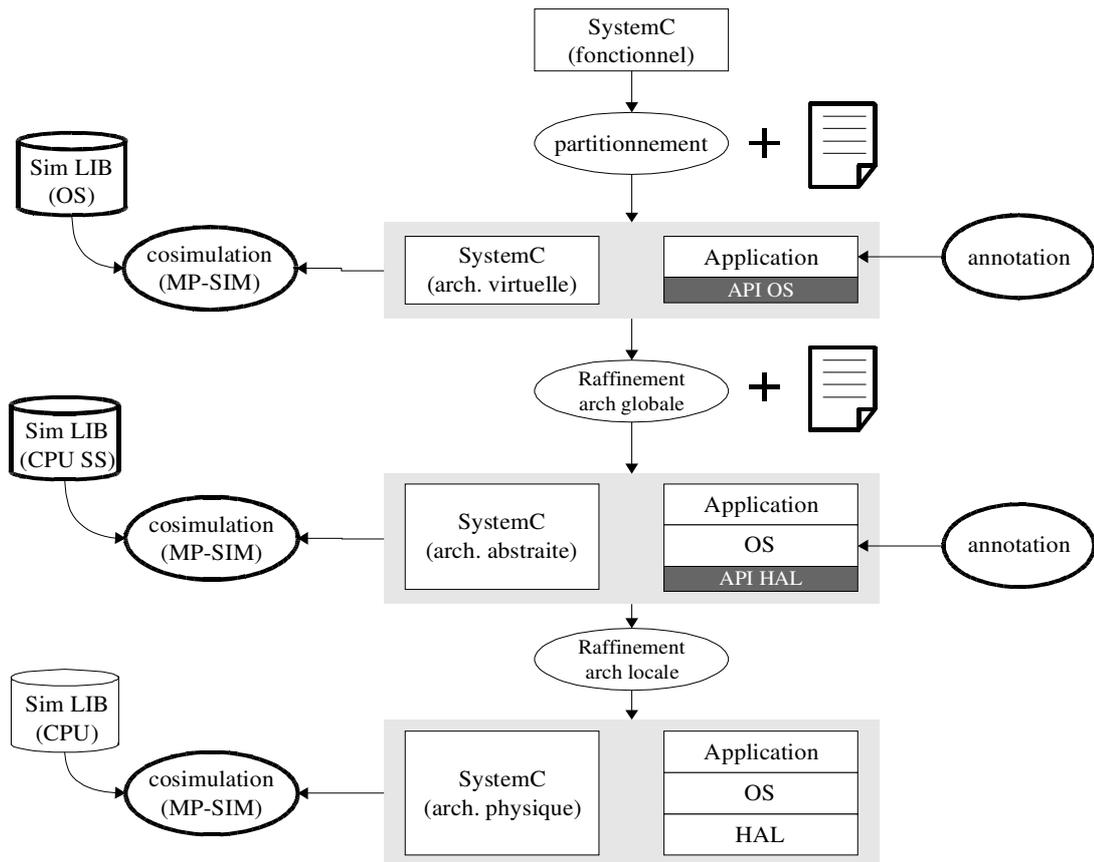


Figure 4.1: Flot générique de l'utilisation de l'approche proposée

L'architecture virtuelle résulte d'une première étape de partitionnement de la spécification fonctionnelle initiale. Le partitionnement sépare les parties qui seront implémentées en matérielle de celles qui seront exécutées comme logiciel. Cette étape peut s'accompagner d'un processus de raffinement qui consiste à ré-écrire la partie logicielle de la spécification sous forme d'un code applicatif qui cible un système d'exploitation particulier.

Notons que le terme raffinement utilisé plus haut veut exactement dire le passage d'une description moins détaillée à une description plus détaillée (de l'architecture). En particulier, cela ne signifie pas forcément qu'on se place dans une optique "top-down", dans le sens où le passage est systématiquement accompagné par la génération d'architecture ou de code. En effet, il est tout à fait envisageable que l'architecture finale soit prédéfinie (à l'image des approches basées sur la notion de plateforme). Dans ce cas, les descriptions manipulées correspondent à différents niveaux d'abstraction de cette architecture.

4.1.2. Architecture virtuelle

L'architecture virtuelle résulte de l'étape de partitionnement de la spécification fonctionnelle et permet une première évaluation de ce partitionnement en faisant l'hypothèse de ressources virtuelles. Ceci correspond à une vision au niveau OS du logiciel applicatif. A ce stade, l'architecture matérielle est globalement décomposée en sous-systèmes, sans pour autant exiger que l'architecture matérielle soit raffinée.

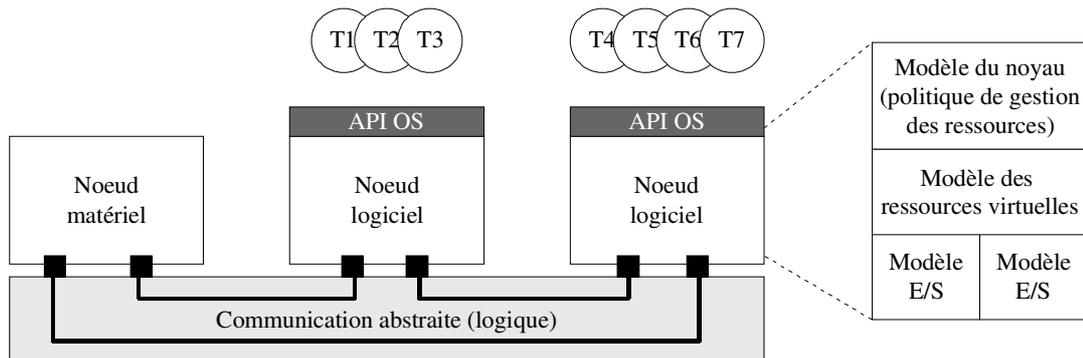


Figure 4.2: Exemple d'architecture virtuelle

Au niveau de l'architecture virtuelle, le concepteur ne dispose pas d'assez d'informations sur l'architecture matérielle cible. Ce modèle est utilisé pour évaluer les conséquences d'un découpage logiciel/matériel donné d'un point de vue ordonnancement des tâches de l'application et valider ainsi certains choix et paramètres architecturaux liés au système d'exploitation à choisir (ou à développer).

4.1.3. Prototype virtuel

La deuxième étape du flot correspond au raffinement de l'architecture virtuelle en utilisant un modèle plus détaillé de l'architecture : le prototype virtuel. Cette étape est caractérisée par la spécification de la nature et du protocole de communication entre les sous-systèmes ainsi que du modèle abstrait de l'architecture locale au niveau de chaque sous-système.

Pour un sous-système logiciel, ceci correspond à une vision au niveau HAL de la machine d'exécution. Le passage au modèle final d'implémentation correspond au raffinement au niveau de chaque sous-système (logiciel) du modèle abstrait de l'architecture en un modèle concret ou encore physique. A ce niveau, le logiciel est également totalement raffiné en disposant de l'implémentation effective de la couche HAL.

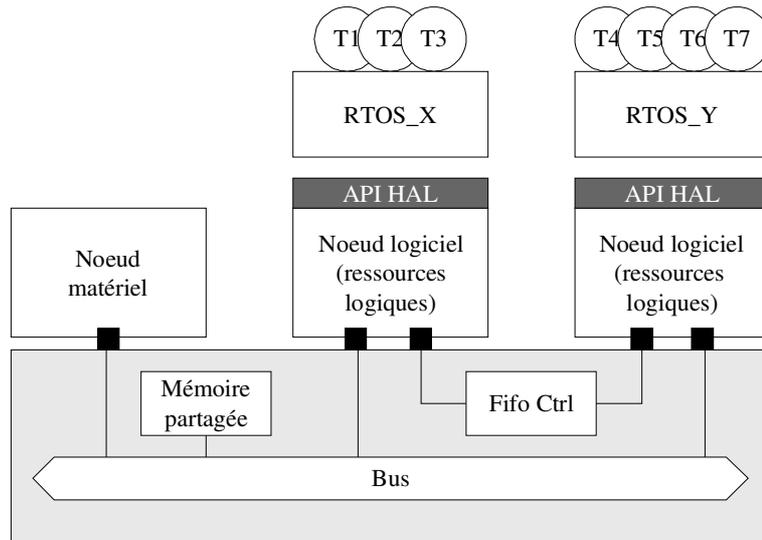


Figure 4.3: Exemple de prototype virtuel

4.1.4. Micro-architecture

La micro-architecture correspond au modèle classique d'intégration logicielle/matérielle. A ce niveau, le logiciel est entièrement développé (comprenant le code de l'application, le système d'exploitation et la couche HAL). L'architecture matérielle est également entièrement conçue. Ceci inclut l'architecture locale de chaque sous-système CPU et l'interface d'adaptation avec le réseau global de communication. La figure 4.4 illustre un exemple de micro-architecture correspondant à un raffinement possible du prototype virtuel du paragraphe précédent.

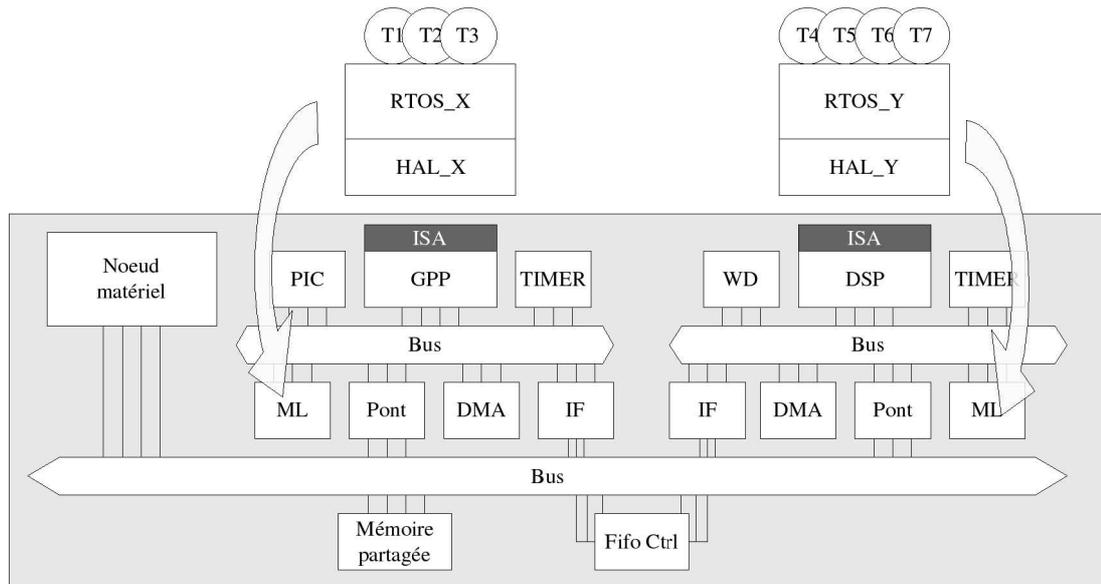


Figure 4.4: Exemple de micro-architecture

Le tableau 4 résume les différentes caractéristiques des trois types de modèles évoqués plus haut. La précision temporelle mesure la fidélité du modèle de simulation à chaque niveau par rapport aux résultats obtenus sur le système réel. Cette mesure concerne généralement des grandeurs qui ont une signification au niveau fonctionnel tels que les signaux d'entrées/sorties de l'application, les évènements marquant l'évolution d'une tâche, etc. La colonne « vitesse de simulation » renseigne sur la rapidité (ou encore la performance) de la simulation en terme de nombre d'instructions logicielles exécutées par seconde. Les chiffres donnés correspondent à des ordres de grandeur typiquement obtenus sur une machine hôte mono-processeur standard (cadencé à 1 Ghz). Ces chiffres dépendent bien évidemment de la complexité du système simulé.

	<i>Interface logicielle</i>	<i>Interface matérielle</i>	<i>Exécution du logiciel embarqué</i>	<i>Précision temporelle</i>	<i>Vitesse de simulation⁸ (Instr/s)</i>	<i>L'interface logicielle/matérielle abstrait</i>
Architecture virtuelle	OS API	TLM Message	Native annotée	Temps-approximatif	100M-1G	-Le système d'exploitation -Le sous-système CPU -L'interface de communication matérielle
Prototype virtuel	HAL API	TLM Transaction	Native annotée	Cycle-approximatif	10M-100M	Le sous-système CPU
Micro-architecture	ISA	-RTL -TLM transfert	Interprétée (ISS)	-Cycle-précis -Cycle-précis	-100 -1K -10K-100K	L'architecture interne du processeur

8 En anticipant sur les résultats du chapitre suivant

	<i>Interface logicielle</i>	<i>Interface matérielle</i>	<i>Exécution du logiciel embarqué</i>	<i>Précision temporelle</i>	<i>Vitesse de simulation (Instr/s)</i>	<i>L'interface logicielle/matérielle abstrait</i>
		-TLM transaction		-Cycle- approximatif	-100K-1M	

Tableau 4: Comparaison des trois types de modèles : architecture virtuelle, prototype virtuel et micro-architecture

En comparant les résultats des différents niveaux d'abstraction, il est clair que nous avons affaire à un compromis précision/performance. Plus la précision souhaitée est élevée, moins la simulation est rapide et inversement. Cette corrélation entre les deux grandeurs est d'ailleurs en cohérence avec le concept même d'abstraction de l'interface logicielle/matérielle. Ainsi, au niveau de l'architecture virtuelle, le concepteur a plus besoin d'évaluer rapidement différents choix de conception, que de procéder à une simulation précise et fidèle de son système. D'ailleurs, dans la majorité des cas, une telle simulation précise n'a pas de sens vue que l'implémentation détaillée du système n'est pas encore connue. Au niveau micro-architecture par contre, une simulation précise est souhaitée pour une validation fine des performances résultants de l'implémentation finale. A ce niveau, la vitesse de simulation devient moins critique vue que la majeure partie du processus d'exploration/validation des solutions architecturales a été effectuée (déplacée) aux niveaux d'abstraction supérieurs.

Pour atteindre une vitesse de simulation élevée au niveau de l'architecture virtuelle et du prototype virtuel, le logiciel embarqué est exécuté en mode « natif » par opposition au mode interprété utilisé au niveau micro-architecture [Bac06]. Cette exécution native doit, cependant, être effectuée dans le contexte d'un simulateur « matériel » où la notion de temps joue un rôle important, à fin de garantir une précision suffisante pour l'estimation de performances.

4.2. Exécution native du logiciel embarqué dans un simulateur matériel à événements discrets

L'exécution native signifie que le logiciel embarqué est compilé pour le processeur de la machine hôte (machine sur laquelle se déroule la simulation) et est exécuté par cette machine. Ceci est à mettre en opposition avec la compilation croisée (*cross compilation*) pour le processeur cible et l'interprétation des instructions binaires via le simulateur du processeur. Dans le cas général, les processeurs cible et hôte sont différents. Ainsi, remplacer une exécution interprétée du logiciel embarqué (fidèle à la réalité) par une exécution native de celui-ci doit être considéré avec précautions. Dans la suite de ce chapitre, nous justifions qu'un tel mode d'exécution du logiciel est non seulement intéressant parce qu'il permet une accélération importante de la simulation, mais aussi

parce qu'il s'avère la solution naturelle au problème de la cosimulation logicielle/matérielle aux niveaux d'abstraction considérés. Pour bénéficier de l'exécution native du logiciel dans le cadre d'une cosimulation temporelle de l'ensemble du système, il faut tenir compte de l'aspect performance lié à l'exécution du logiciel sur la machine cible. La solution retenue passe par l'annotation statique du code logiciel. Cette technique d'annotation permet en fait de découpler le problème de l'estimation de performance en deux parties : une partie statique qui peut être déterminée à la compilation, et une partie dynamique qui profitera, au cours de l'exécution, de connaissances supplémentaires sur l'environnement d'exécution.

4.2.1. Problèmes liés à l'exécution native du logiciel

Si l'exécution native du logiciel embarqué s'avère être en concordance avec la notion d'abstraction de l'interface logicielle/matérielle développée dans le chapitre précédent, son utilisation dans le cadre d'un environnement de simulation à événements discrets tel que SystemC doit être fait avec précautions. SystemC, en tant que tel, ne posant pas de restrictions particulières sur l'utilisation du logiciel (C++) pour décrire le comportement des processus, c'est au concepteur de faire le bon choix. Dans cette section nous analysons les contraintes liées à l'utilisation de l'exécution native du logiciel embarqué dans SystemC.

4.2.1.1. L'exécution native comme solution naturelle aux niveaux d'abstraction considérés

L'utilisation d'un simulateur de processeur (en l'occurrence un ISS) pour exécuter le logiciel dans un environnement de cosimulation suppose que le logiciel soit entièrement développé jusqu'à ses plus basses couches. Ceci inclut la couche HAL et les pilotes des périphériques en plus des autres parties du système d'exploitation. On imagine mal, à ce niveau, faire tourner une application composée de plusieurs tâches parallèles communiquant par des primitives de haut niveau sur un ISS sans disposer du système d'exploitation.

Dans les chapitres précédents, nous avons identifié les niveaux HAL et OS comme niveaux d'abstraction de l'interface de programmation. A ces niveaux d'abstraction, on n'a pas besoin de connaître tous les détails d'implémentation de l'interface logicielle/matérielle. Il est naturel de penser à l'exécution native pour simuler le logiciel dans un environnement de cosimulation tel que SystemC pour au moins trois raisons :

- ◆ aux niveaux d'abstraction considérés de l'interface logicielle/matérielle (HAL et OS), on ne dispose pas (ou on n'est pas censé disposer) d'assez d'informations concernant l'implémentation effective de l'interface logicielle/matérielle (en particulier les couches basses de l'architecture logicielle et les détails de l'architecture locale du sous système CPU) pour utiliser un ISS.

- ◆ au niveaux d'abstraction HAL et OS, le logiciel est entièrement décrit en utilisant un langage de haut niveau (en l'occurrence C/C++) et est supposé être portable, c'est à dire indépendant des détails d'implémentation de la machine sous-jacente. Il peut ainsi être compilé pour la machine hôte sans modification.
- ◆ l'exécution native du logiciel permet une accélération importante (de l'ordre de trois à quatre ordres de grandeur) par rapport à une exécution interprétée par un simulateur de processeur. Cette accélération est d'ailleurs un indicateur supplémentaire qu'on est effectivement à un niveau d'abstraction supérieur.

La solution native possède néanmoins des inconvénients. En effet, une telle approche repose sur l'hypothèse que le code source du logiciel est disponible pour pouvoir le compiler pour le processeur de la machine hôte mais surtout l'instrumenter par des annotations de performance (cf section 4.2.3). Cette hypothèse n'est pas toujours valable, en particulier lorsque certaines parties du logiciel (exemple le système d'exploitation) sont protégées et ne peuvent être déployées que sous forme d'IP binaires pré-compilés pour le processeur cible.

4.2.1.2. Exécution native dans SystemC : motivation

L'un des arguments essentiels que l'on a l'habitude d'avancer en faveur de SystemC est son support intrinsèque de la modélisation du logiciel vu que le langage de description utilisé n'est autre que C++. Cet argument doit toutefois être pris avec précautions. En effet, en tant que tel, SystemC n'inclut aucun support particulier pour tenir compte de l'aspect implémentation ou encore architecture du logiciel embarqué. Il permet simplement d'exprimer le comportement des processus « matériels » en utilisant C++ comme langage de base. Il s'agit donc, à l'origine, d'une description purement fonctionnelle du système qui est différente du logiciel embarqué final.

Ceci dit, vu que SystemC n'impose aucune restriction au niveau de la nature du code C++ utilisé pour spécifier le comportement des processus, rien n'empêche d'utiliser le même code qui va servir pour l'implémentation du logiciel embarqué. Ceci peut même aller jusqu'à inclure des parties du système d'exploitation si ces parties sont « portables ». Cependant, voulant appliquer ce constat, on s'aperçoit très vite qu'on a besoin d'un modèle de simulation de l'interface logicielle/matérielle sous-jacente. De plus, si on veut que l'exécution du code dans le contexte du processus SystemC renseigne sur le temps qu'aurait pris ce même code sur l'architecture matérielle finale, il est nécessaire de modéliser explicitement un tel temps d'exécution. A ce stade, il faut prendre soin de bien gérer les interactions possibles avec l'environnement matériel pour modéliser correctement la performance du système global et éviter des erreurs liés au fonctionnement même de l'ensemble du système.

4.2.2. L'environnement SystemC

SystemC est une bibliothèque de classes C++ offrant un ensemble de fonctionnalités permettant la description et la simulation des systèmes électroniques. S'agissant d'une bibliothèque, SystemC n'est pas un langage à part entière, en ce sens qu'il ne possède pas une syntaxe et un compilateur qui lui sont propres. Ainsi, un modèle SystemC donné peut être compilé avec un compilateur C++ standard. Cette propriété confère à SystemC au moins trois avantages :

- ◆ Son utilisation ne nécessite pas l'apprentissage d'un nouveau langage.
- ◆ Toute la puissance de modélisation logicielle de C/C++ est naturellement transmise à SystemC.
- ◆ Etant donné qu'il s'agit d'un modèle compilé et non pas interprété (comme c'est le cas pour VHDL), la vitesse de simulation est beaucoup plus importante.

De part sa spécification, SystemC renforce la séparation conceptuelle entre structure et comportement d'un système. Dans la suite de la section, nous présentons quelques aspects liés à chacune de ces parties. L'accent sera mis sur les aspects favorisant la conception haut niveau ainsi que sur quelques subtilités liées à la sémantique d'exécution de SystemC.

4.2.2.1. *Éléments structurels d'une description SystemC*

Les modules sont les blocs de base dans une description SystemC. Ils permettent au concepteur de décomposer un système complexe en un ensemble de sous-systèmes. Cependant, un module n'a pas une sémantique d'exécution propre. Il s'agit uniquement d'une structure d'organisation.

Avec la version 2.0 de SystemC, un formalisme unifié est utilisé pour représenter l'aspect communication: il s'agit du formalisme *IMC* (*Interface Method Call*) qui repose sur la notion d'interface/canal. Une interface (*sc_interface*) est une classe abstraite qui sert à déclarer un certain nombre de services de communication. L'implémentation effective de ces services se fait au niveau du canal qui peut alors être vu comme une concrétisation de la classe abstraite *sc_interface*. En SystemC, on distingue deux types de canaux: les canaux primitifs qui ne possèdent pas de comportement actif (processus) et les canaux hiérarchiques qui peuvent incarner des déclarations de processus. Ces derniers se confondent, syntaxiquement parlant, avec les modules standards. Selon le formalisme *IMC*, les ports sont plutôt optionnels et ont surtout un rôle organisationnel.

4.2.2.2. *Éléments comportementaux d'une description SystemC: le modèle à événements discrets*

Le comportement d'un système décrit en SystemC est spécifié par un ensemble de processus. Un processus peut être vu comme une transformation sur des signaux d'entrées qui génère des signaux de sorties (figure 4.5). Comme tout autre environnement basé sur le modèle à événements discrets, la

notion de signal est primordiale. En se basant sur le formalisme et les notations proposés par [Lee98], un signal est défini comme une séquence d'évènements : $s = \{ e_1, e_2, \dots \}$.

Un évènement est un couple formé par une étiquette (*tag*) et une valeur : $e = (t, v) \in T \times V$.

Pour tenir compte des évènements qui peuvent se produire en même temps, l'étiquette est souvent définie, elle-même, comme un couple formé par une date appartenant à l'ensemble des réels et un entier naturel appelé index: $t = (\tau, n) \in R \times N$. Ainsi, pour ordonner chronologiquement les évènements, on utilise l'ordre lexicographique⁹ sur l'ensemble $R \times N$ qui devient alors un ensemble totalement ordonné.

Avec cette terminologie, un signal est dit discret s'il existe une fonction monotone (un morphisme de groupe) entre l'ensemble des valeurs prises par son étiquette et l'ensemble des entiers. Un système S (défini comme un ensemble de signaux) est dit discret si tous ces signaux sont discrets.

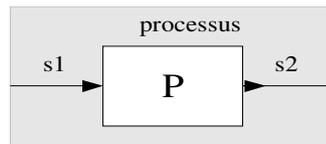


Figure 4.5: Le processus SystemC comme transformation sur des signaux

En SystemC, l'implémentation des signaux ne suit pas ce formalisme général. En particulier la notion de sources simultanées (au sein d'un même évènement) n'est pas considérée. Ceci signifie qu'il peut y avoir une certaine "perte d'informations" si des précautions supplémentaires ne sont pas déployées. Par exemple, si deux processus écrivent en même temps dans un signal donné, une seule action sera retenue (selon l'ordre d'exécution). Même si ce type de situation est interdit par SystemC dans le cas où l'on passe par le biais des ports standards (règles de vérification statiques), ce n'est pas toujours le cas pour la notification dynamique des évènements, où il n'existe pas ce genre de vérification. D'ailleurs, vu les niveaux d'abstraction visés par SystemC, une telle situation est certainement bénéfique d'un point de vue modélisation et il sera complètement néfaste de vouloir l'interdire dans le cas général. La solution, passe à notre avis, par l'extension du modèle d'évènements utilisé par SystemC pour supporter le formalisme général décrit plus haut.

La causalité est garantie en SystemC moyennant la notion de cycle delta¹⁰. Cela signifie qu'une affectation à un signal au sein d'un processus n'est pas considérée immédiatement (à l'instant courant)

⁹ Revient à comparer tout d'abord par rapport aux réels, en cas d'égalité comparer par rapport aux entiers

¹⁰ Tout comme en VHDL

mais plutôt dans "un futur proche". Cet instant futur est obtenu en projetant l'événement dans un temps delta à partir de l'instant courant, delta est une valeur "virtuelle" qui n'a de présence qu'au sein du simulateur et n'est donc pas visible de l'extérieur.

Dans SystemC, un signal peut conceptuellement être représenté comme illustré par la figure 4.6. étant donné que SystemC n'accepte pas les affectations retardées multiples¹¹ (contrairement à VHDL), un signal peut être réduit à un couple formé par une valeur courante et une valeur future associée à une date.

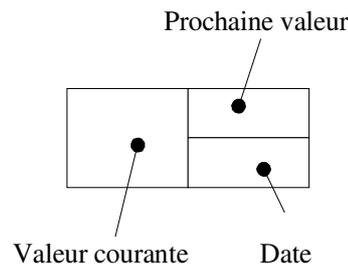


Figure 4.6: représentation d'un signal en SystemC

Pour les signaux réguliers (ceux qui peuvent être accédés normalement par un processus), la date de la prochaine valeur se trouve toujours égale à l'instant courant + delta. Ainsi, la "case" correspondante à cette date est simplement omise. Pour les autres signaux (horloges et événements implicites statiquement associés aux processus pour tenir compte des *wait(time)*), la date de la prochaine valeur doit être explicitement représentée. D'ailleurs, c'est grâce à ce type de signaux que le temps logique de la simulation peut avancer. Un cas particulier de signaux SystemC, appelés événements (*sc_event*), n'ont pas de valeurs spécifiques. C'est le déclenchement de l'événement qui à une signification en tant que tel. D'un point de vue implémentation, tous les autres types de signaux sont, en réalité, obtenus en combinant des événements SystemC avec des structures de données plus ou moins complexes. Ceci permet en particulier d'implémenter des signaux (ou encore des canaux) « sur mesure ».

◆ Le moteur de simulation de SystemC

l'algorithme de simulation peut être décrit de la manière suivante (algorithme simplifié faisant abstraction des détails concernant les types de processus autres que `SC_THREAD`)

- (1) Tous les signaux et événements qui sont actifs à l'instant courant sont mis à jour.
- (2) Tous les processus sensitifs à un événement qui vient de se déclencher sont exécutés jusqu'à

¹¹ La toute dernière version de SystemC (2.1) semble remédier à cette limite.

ce qu'il fassent appel à un *wait*. L'ordre d'exécution des processus est quelconque.

(3) Les étapes 2 et 3 sont répétées jusqu'à ce qu'aucun signal n'est actif à l'instant courant (chaque itération correspond à un nouveau cycle delta).

(4) L'instant courant est avancé pour correspondre au plus proche événement futur. S'il n'y a pas un tel événement, la simulation se termine, sinon on retourne à l'étape 1.

4.2.3. Estimation du temps d'exécution du logiciel

Dans le cas général, le temps d'exécution d'un programme donné *Prog* est donné dans le cas général par la formule :

$$T_{Prog} = \frac{\sum_{inst\ exécutées} CPI(inst)}{f_{clk}} \approx \frac{N_{inst} \times CPI}{f_{clk}}$$

où $CPI(inst)$ est le nombre de cycles nécessaires pour chaque instruction *inst* effectivement exécutée entre le point d'entrée et le point de sortie du programme. Cette formule, simple en apparence, cache en réalité plusieurs sources de dépendances [Bju01].

4.2.3.1. Dépendance au flot de contrôle: découpage en blocs de base

Un premier niveau de dépendance est lié au logiciel lui même. Le temps que prend l'exécution d'un bout de code dépend des instructions effectivement exécutées. Ceci est directement lié au flot de contrôle CFG (*Control Flow Graph*) sous-jacent au code en question. La figure 4.7 montre un exemple de flot de contrôle associé à un code source. L'exécution de certaines parties du code dépend des valeurs des variables (*x* et *y*) contrôlant les différents tests effectués. Étant donné que ces valeurs ne sont pas connues à l'avance mais déterminées dynamiquement au cours de l'exécution, il est théoriquement impossible de connaître a priori et d'une manière précise le temps associé à l'exécution du code de la figure. Dans notre cas, une estimation de type « pire cas » ou encore WCET (*Worst Case Execution Time*) n'est pas intéressante¹². En effet, comme les annotations que nous envisageons d'insérer dans le code source vont être elles aussi exécutées dynamiquement en même temps que le code original, une solution directe serait de « suivre » le flot de contrôle, en ce sens que les annotations seront appliquées aux différentes branches de ce flot.

¹² Elle l'est pour les gens qui s'appuient sur l'analyse statique du code en vu de déterminer par exemple un ordonnancement des tâches.

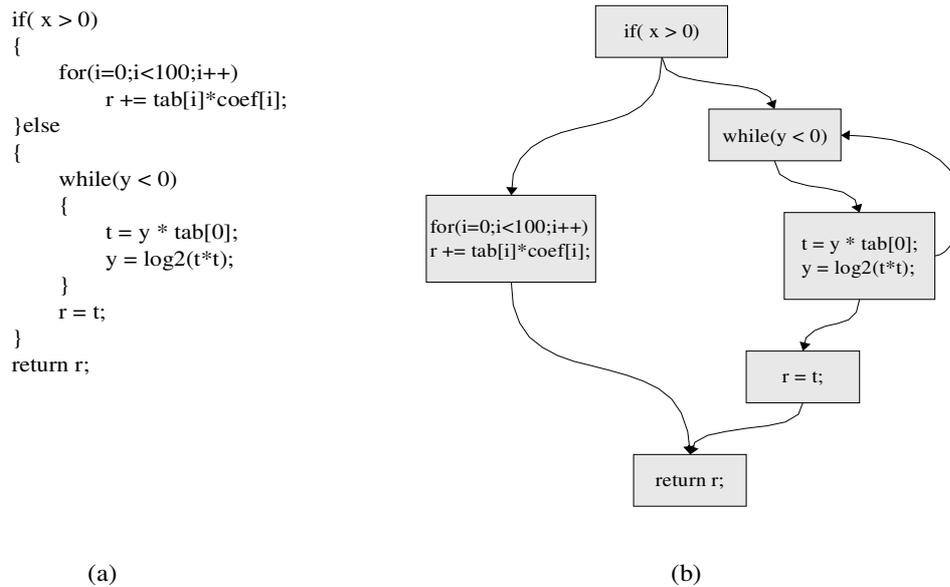


Figure 4.7: Exemple de flot de contrôle dans le logiciel et décomposition en blocs de base

Dans la même figure, la partie (b) montre la décomposition en blocs de base du code original en suivant le flot de contrôle de celui-ci. Notons qu'il s'agit d'une décomposition macroscopique, dans le sens où seuls les branchements comportant des décisions dynamiques sont considérés. Ainsi, dans l'exemple, la boucle *for* n'a pas été décomposée bien qu'elle contient un branchement qui dépend de la valeur de la variable compteur *i*. Ceci vient du fait que l'on peut regrouper les différents « micro blocs » dans un seul « macro bloc » sans perdre de précision au niveau de l'estimation du temps d'exécution, vu que le nombre d'itérations est fixe.

Le découpage en blocs de base étant effectué, l'étape suivante sera l'annotation du code. Pour cela, nous avons adopté l'approche d'annotation au niveau code source. Le résultat de l'application d'une telle étape sur le code de l'exemple précédent est illustré par la figure 4.8. Dans notre cas, la fonction d'annotation s'appelle *consume*. Cette fonction est insérée dans le code source avant¹³ chaque bloc de base. L'argument de la fonction correspond à l'estimation du temps d'exécution du bloc de base correspondant. Notons que cette valeur est maintenant indépendante du flot de contrôle.

¹³ La position de l'annotation par rapport au bloc de base est arbitraire pour peu qu'elle soit maintenue partout

```

consume(d1);
if( x > 0)
{
    consume(d2);
    for(i=0;i<100;i++)
        r += tab[i]*coef[i];
}else
{
    consume(d3)
    while(y < 0)
    {
        consume(d4);
        t = y * tab[0];
        y = log2(t*t);
    }
    consume(d5)
    r = t;
}
consume(d6)
return r;

```

Figure 4.8: résultat de l'annotation temporelle au niveau code source

4.2.3.2. Dépendance au compilateur

Le temps d'exécution d'un bloc de base correspond à la somme des temps d'exécution des instructions binaires qui le composent. Le nombre et le type de ces instructions dépendent étroitement du compilateur et des optimisations utilisés. Par exemple, le temps d'exécution d'un bloc de base peut varier d'une manière significative selon le type d'optimisation. Par ailleurs, le compilateur peut générer du code binaire qui n'a pas d'équivalence directe au niveau du code source. L'exemple type serait l'épilogue et le prologue générés au niveau de chaque fonction pour gérer correctement le passage des arguments et assurer la cohérence de la pile. Ce code dépend encore une fois du compilateur utilisé.

En utilisant l'approche basée sur l'analyse statique du code binaire (cf chapitre 3), ce type de dépendances se trouve résolu. En effet, selon cette approche, l'analyse se fait au niveau du code binaire généré par compilation croisée partielle du code source. Les éventuelles optimisations ainsi que le code spécifique au compilateur sont ainsi pris en compte par l'outil d'analyse.

4.2.3.3. Dépendance à l'architecture matérielle

Le troisième type de dépendance concerne l'architecture matérielle qui va prendre en charge l'exécution du code. A ce niveau, les sources d'incertitudes sont multiples :

- ◆ Incertitudes dues au processeur:

Etant le premier sur la chaîne d'exécution, le processeur constitue la première source de dépendances architecturales (mais pas forcément la plus importante d'un point de vue temps

d'exécution). Pour des processeurs ayant une architecture interne simple, le nombre de cycles nécessaires à l'exécution d'une instruction est prédictible et peut être déterminé à l'avance¹⁴ (en consultant la documentation spécifique au processeur). L'exemple type serait le processeur ARM7 qui possède une architecture relativement simple avec un pipeline à trois étages. Pour les processeurs ayant une architecture plus complexe (pipeline plus profond, prédiction de branchements, etc.) le nombre exact de cycles ne peut être déterminé à l'avance puisqu'il dépend de phénomènes dynamiques et imprévisibles. A ce niveau, deux solutions peuvent être utilisées. La première (et la plus simple) est d'approcher ce temps par une valeur moyenne (ou bien au pire cas selon le contexte) au détriment de la précision de l'estimation. La deuxième consiste à utiliser un modèle de simulation fonctionnelle de la source de dépendance dynamique (par exemple un modèle fonctionnel du pipeline). Un tel modèle n'a pas pour objectif la simulation précise du comportement de la source d'incertitude, mais plutôt de rendre compte de son influence sur le temps d'exécution en se servant au maximum d'informations disponibles via l'analyse statique. Dans le cadre de cette thèse, nous nous sommes limités à la première solution pour les processeurs ayant ce type de dépendances.

- ◆ Incertitudes dues aux interruptions matérielles

Ce type d'incertitude est probablement celui qui a la contribution la plus importante sur la variation du temps d'exécution d'un bloc de base. En effet, si statiquement un bloc de base est vu comme une « entité atomique » ayant un seul point d'entrée et un seul point de sortie, cette hypothèse n'est pas toujours respectée au cours de l'exécution, notamment à cause des interruptions matérielles¹⁵. Lorsqu'une telle interruption surgit pendant l'exécution d'un bloc de base, le contrôle est inconditionnellement passé à la routine de gestion des interruptions. Les détails de ce mécanisme dépendent étroitement du processeur et de l'architecture du sous-système CPU correspondant. Du point de vue du bloc de base, ce qui importe c'est que l'exécution a été interrompue involontairement et ne peut reprendre qu'ultérieurement dans un temps futur imprévisible. Ainsi, les effets d'un tel mécanisme d'interruption s'avèrent être de premier ordre sur la qualité de l'estimation de la performance du système global. Dans la suite du chapitre, nous détaillerons notre approche concernant ce problème.

- ◆ Incertitudes dues à la hiérarchie mémoire

Le temps nécessaire pour l'exécution d'un bloc de base peut étroitement dépendre de la hiérarchie de mémoire sous-jacente. Par hiérarchie mémoire, on fait référence à l'organisation de la mémoire au sein de l'architecture. Ceci inclut notamment les mécanismes de cache utilisés pour améliorer

14 En faisant l'hypothèse que tout est parfait par ailleurs

15 Les interruptions logicielles quant à eux sont considérées comme des branchements classiques et sont prises en compte au niveau de l'analyse statique du code

statistiquement les performances du système (cache de données et/ou cache d'instructions, TLB, etc.). Comme pour les processeurs ayant une architecture interne complexe, ces mécanismes introduisent de l'indéterminisme au niveau du temps d'exécution du logiciel. Dans un contexte temps-réel, la solution est simplement de les éviter (d'ailleurs, augmenter statistiquement les performances dans un contexte temps-réel pure n'a pas de sens). Même dans un contexte plus général des systèmes embarqués, ces mécanismes ne sont généralement pas les bienvenus à cause des coûts supplémentaires qu'ils engendrent (notamment au niveau consommation). Dans le cadre de cette thèse, nous faisons l'hypothèse simplificatrice d'une architecture dépourvue de ces mécanismes. Dans le cas général, toutefois, on peut penser à des méthodes d'estimation hybrides combinant à la fois la méthode d'annotation statique avec des modèles fonctionnels de cache.

- ◆ Incertitudes dues à l'aspect communication

Cet aspect concerne la communication entre le processeur et le bus système. Ceci inclut le chargement des instructions et les transferts registres/mémoire ou mémoire/registres. Dans un système idéal, cette communication ne doit pas ajouter de latences supplémentaires, autres que celles liées au pipeline du processeur. Ceci revient généralement à supposer que les différents transferts processeur/mémoire se font en un seul cycle; ce qui signifie que la mémoire (ou le périphérique) est parfaite (pas de cycles d'attente introduits) et que le bus n'introduit aucun délai supplémentaire (à cause d'une éventuelle congestion par exemple). Malheureusement, ce genre d'hypothèse est rarement vérifié en pratique: les bus sont souvent sujets à des conflits d'accès, surtout dans un cadre multiprocesseur et la mémoire est loin d'être parfaite dans le cas général, encore moins s'il s'agit d'un périphérique. Ces aspects étant clairement liés à l'architecture locale du sous-système CPU, un modèle de simulation de ce dernier doit permettre de tenir compte de leurs effets sur la performance globale du système. Nous détaillerons notre approche pour tenir compte de ce problème dans le reste de ce chapitre.

4.2.4. Synchronisation entre le logiciel et le matériel

La synchronisation entre le logiciel et le matériel signifie l'ordonnancement relatif des événements liés à l'architecture matérielle et des actions issues du logiciel. Bien évidemment, il s'agit d'un problème "artificiel" qui concerne la simulation de haut niveau. Dans la réalité physique des choses, une telle synchronisation s'effectue naturellement à chaque coup d'horloge du processeur. Lorsqu'on passe à des niveaux d'abstraction plus élevés, cette notion d'exécution cyclique cadencée par une horloge matérielle disparaît. Le logiciel et le matériel deviennent beaucoup plus découplés et peuvent désormais « évoluer » indépendamment l'un de l'autre tant qu'il n'y a pas besoin qu'une partie accède à "l'état" de la partie opposée. Dans le cas contraire, il faut s'assurer que l'ordre des opérations

est correct, sous peine de compromettre l'état global du système et se retrouver ainsi avec des résultats erronés. Dans l'approche utilisant l'exécution native du logiciel, étant donné que c'est l'environnement matériel qui se comporte comme maître (dans le sens où c'est lui qui est responsable de faire évoluer le temps global du système), la synchronisation revient à déterminer, dans le code du logiciel, les points où il est nécessaire de s'aligner par rapport au « temps matériel » avant d'exécuter l'opération suivante.

4.2.4.1. Relation entre synchronisation et annotation du code logiciel

Nous avons présenté, précédemment, la méthode utilisée pour instrumenter le code logiciel par des annotations temporelles. Ces annotations ne sont autres que des appels à la fonction *consume()* qui sont exécutés en même temps que le logiciel embarqué. A chaque appel à la fonction *consume()*, le temps local du noeud logiciel est incrémenté. Cependant, cela ne signifie pas forcément que nous avons une correspondance bijective entre les annotations d'une part et les endroits où la synchronisation temporelle doit être effectuée de l'autre part. En effet, il sera inutile, et surtout coûteux, d'effectuer la synchronisation à chaque exécution d'un bloc de base. Ceci est dû au fait que les blocs de base sont simplement liés à l'aspect flot de contrôle dans le logiciel, chose qui est totalement indépendante de l'endroit où la synchronisation entre le temps global du simulateur matériel et le temps local associé à un noeud logiciel doit s'effectuer. Le problème lié à la détermination des points de synchronisation est d'ailleurs un problème commun à plusieurs domaines de recherche, dès lors que l'aspect distribué est mis en jeu. Nous pouvons citer comme exemples le domaine de la simulation distribuée des systèmes à événements discrets [Cha79] et le domaine de la conception des architectures distribués mettant en oeuvre des modèles relaxés de mémoires partagée [Ara01].

Dans la suite de cette section, nous nous proposons d'étudier les situations qui risquent de compromettre la validité de l'exécution native du logiciel et pour lesquelles la synchronisation temporelle entre le logiciel et le matériel doit être effectuée.

4.2.4.2. Auto-synchronisation sur les opérations de contrôle

Dans le troisième chapitre, nous avons classé le logiciel embarqué selon deux types d'opérations: les opérations de calcul et les opérations de contrôle. La figure 4.9 reprend cette décomposition en mettant l'accent sur la séquentialité temporelle de ces actions durant l'exécution. Une opération de contrôle implique une interaction potentielle avec la machine d'exécution sous-jacente puisqu'elle correspond, en fait, à l'utilisation d'un service offert par cette machine. Il est donc naturel de considérer cette opération comme un point de synchronisation étant donnée qu'elle est susceptible de

modifier l'état de la machine faisant partie du « monde matériel ». Dans l'implémentation des modèles de simulation de l'interface logicielle/matérielle (section 4.3), la synchronisation est systématiquement effectuée chaque fois qu'un service lié à la machine est invoqué par le logiciel. Ainsi, il n'est pas nécessaire de procéder explicitement à une telle synchronisation au niveau du code du logiciel embarqué.

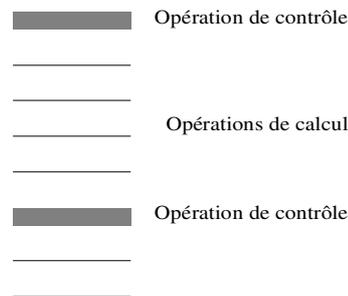


Figure 4.9: séquençement des opérations de contrôle et de calcul

Pour la partie calcul qui, dans le cas d'une exécution native, ne fait pas appel aux services sous-jacents de la machine, la question qui s'impose est de savoir si on a besoin d'effectuer des synchronisations et si oui, à quels endroits.

4.2.4.3. Points de synchronisation dans un bloc de calcul

1) Notations et définitions

◆ État, action et transition

Dans la suite de la section, nous notons par S l'espace des états possibles de la machine d'exécution. Une action atomique est une fonction $a : S \rightarrow S$ qui, à un état initial s_i , associe un état final s_f tel que $s_f = a(s_i)$. Nous parlons également d'une transition d'états déclenchée par l'action a et notée: $S_i \rightarrow S_j$

Une action dans le cas général est définie comme une suite non vide (éventuellement réduite à un seul élément) d'actions atomiques. L'ensemble de toutes les actions est noté $A \subset [S \rightarrow S]$. Au niveau ISA, cet ensemble correspond au jeu d'instructions du processeur. Aux autres niveaux d'abstraction (HAL et OS), il correspond à l'API offerte par la machine abstraite.

◆ Programme et *thread* d'exécution

Avec ces notations, un programme P est une fonction d'un ensemble fini d'entiers vers l'ensemble des actions, $P : \{1..m\} \rightarrow A$. $P(1)$ correspond alors à l'entrée du programme. L'exécution

d'un programme donne lieu à la notion de *thread*. Un *thread* est défini comme un état initial et une séquence (finie ou infinie) d'actions:

$$thread : \left\{ \begin{array}{l} s_0 \in S \\ t : \{1..n\} \rightarrow A \end{array} \right\}$$

Lorsque $n = \infty$, le *thread* est dit infini (*non terminating*). Dans le cas contraire, il est fini. Le séquencement des opérations dans un *thread* est contrôlé au cours de l'exécution par le pointeur d'opération (un niveau ISA c'est le PC) qui trace le flot de contrôle associé au programme. Formellement, ce pointeur est défini comme une fonction de l'espace des états vers l'ensemble des entiers : $c : S \rightarrow IN$ où $c(S) = 0$ est utilisé pour dénoter l'action "STOP".

Ainsi, un *thread* décrit la sémantique opérationnelle d'exécution d'un programme comme une suite de transitions dans l'espace des états $S_0 \rightarrow S_1 \rightarrow \dots \rightarrow S_n \rightarrow S_{n+1} \rightarrow \dots$ avec:

$$\begin{aligned} s_0 &\in S \\ s_1 &= P(c(s_0))(s_0) = t(1)(s_0) \\ s_2 &= P(c(s_1))(s_1) = t(2)(s_1) \\ &\cdot \\ &\cdot \\ &\cdot \\ s_n &= P(c(s_{n-1}))(s_{n-1}) = t(n)(s_{n-1}) \\ &\cdot \\ &\cdot \end{aligned}$$

2) *Points de synchronisation dans un bloc de calcul*

Pour déterminer les points de synchronisation qui devraient apparaître dans un bloc de calcul, nous nous proposons de comparer le modèle d'exécution native à l'exécution interprétée du code logiciel.

Pour cela, considérons un bloc de calcul (délimité par deux opérations de contrôle) tel que celui illustré par la figure 4.10. L'exécution interprétée des instructions formant ce bloc donne lieu à une succession de transitions d'état $\{S_0^\phi, S_1^\phi, \dots, S_N^\phi\}$. S^ϕ représente l'état de la machine à un instant donné. Ceci inclut la valeur de tout les registres internes du processeur, mais aussi la valeur des différentes cases mémoires, y compris les registres des périphériques.

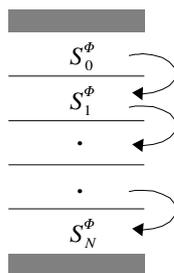


Figure 4.10: Transition d'état dans un bloc de calcul

Vue que nous avons décomposé les instructions en deux parties, calcul et contrôle, nous pouvons faire l'hypothèse que l'état S^ϕ peut lui-même être décomposé en deux sous-ensembles, un qui sera vu comme le domaine principal d'action des opérations de contrôle, l'autre celui des opérations de calcul:

$$S^\phi = S_{\text{contrôle}}^\phi \cup S_{\text{calcul}}^\phi$$

S_{calcul}^ϕ correspond à l'ensemble des registres généraux du processeur ainsi que l'espace mémoire réservé aux données locales et globales de l'application. S_{calcul}^ϕ , quant à lui, correspond à ce qu'on appelle couramment l'état système. Ceci inclut les registres d'état du processeur et les zones mémoires contenant des données de contrôle (registres des périphériques, TLB, etc.).

La figure 4.11 schématise une vue équivalente introduisant l'exécution native du même bloc de calcul. Dans l'exécution native du logiciel, l'ensemble des actions constituant le bloc de calcul peut être modélisé comme étant une seule action atomique qui s'exécute "d'un seul coup" en un temps égal à zéro. Cette exécution fait basculer l'état du système de l'état initial S_0^θ à l'état final S_N^θ , où S^θ peut être vu comme l'état abstrait équivalent à l'état S^ϕ la machine réelle.

Pour prendre en compte le temps réellement pris par l'exécution de cette « super action », une transition identitaire I_d est rajoutée. Au temps t_0+N , nous espérons ainsi aboutir à un système dont l'état est équivalent à celui obtenu par l'exécution interprétée.

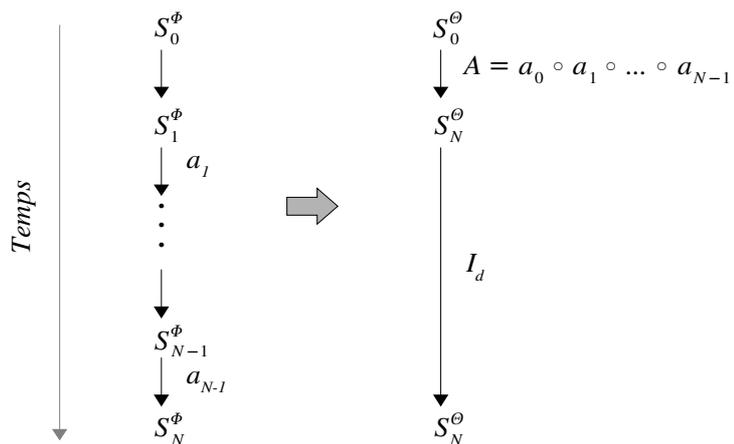


Figure 4.11: Principe de l'exécution native d'un bloc de calcul comparée à une exécution interprétée

Ce raisonnement reste, cependant, valable en l'absence d'interruptions matérielles. Dans le cas contraire, des précautions supplémentaires doivent être prises en considération avant de pouvoir tirer des conclusions semblables. Pour cela, plaçons nous donc dans le cas où une interruption matérielle surgit pendant l'exécution d'un bloc de calcul (figure 4.12).

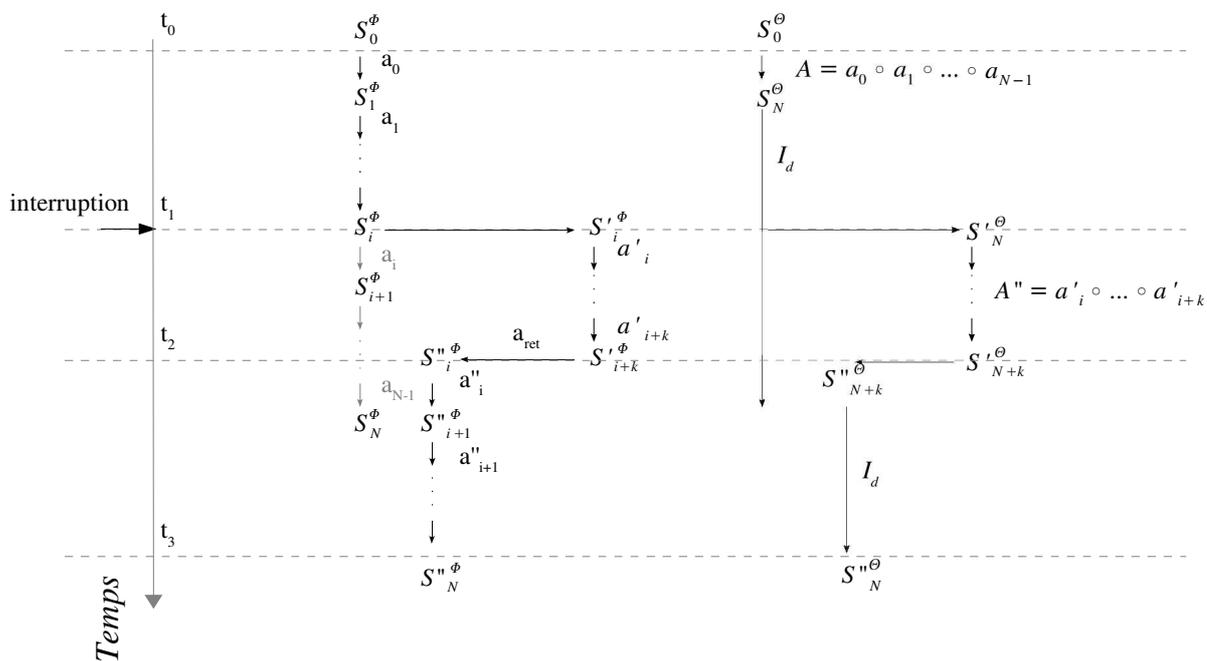


Figure 4.12: Cas d'une interruption matérielle

Dans ce cas nous avons:

$$\begin{aligned}
S''_N^\Phi &= a''_i \circ a''_{i+1} \circ \dots \circ a''_{N-1} (S''_i^\Phi) \\
&= a_i \circ a_{i+1} \circ \dots \circ a_{N-1} (S''_i^\Phi)
\end{aligned}$$

or

$$\begin{aligned}
S''_i^\Phi &= a'_i \circ a'_{i+1} \circ \dots \circ a'_{i+k} \circ a_{iret} (S''_i^\Phi) \\
&= A'' (S''_i^\Phi) \\
&= A'' (I(S_i^\Phi)) \\
&= A'' (I(a_0 \circ \dots \circ a_i(S_0^\Phi)))
\end{aligned}$$

d'où

$$S''_N^\Phi = a_i \circ a_{i+1} \circ \dots \circ a_{N-1} \circ A'' (I(a_0 \circ \dots \circ a_i(S_0^\Phi)))$$

Il est facile de démontrer qu'une condition suffisante pour que l'état S''_N^Θ soit équivalent à S''_N^Φ est que les domaines d'actions des opérations $a_0 \dots a_{N-1}$ sont disjoints. Ceci signifie qu'il n'y a pas de mémoire partagée qui soit accessible au milieu de la séquence d'opérations en question. Noter qu'il s'agit bien d'une condition faible (suffisante mais non nécessaire).

4.3. Modèles de simulation de l'interface logicielle/matérielle

Dans cette section, nous définissons deux modèles conceptuels de l'interface logicielle/matérielle (machine), relatifs à l'architecture virtuelle [Bou04][Yoo03] et au prototype virtuel [Bou05] décrits précédemment. Nous distinguons, à chaque fois, les éléments qui rentrent dans la définition du modèle, des éléments qui seront abstraits à ce niveau. Le modèle est alors décrit par la spécification des relations qui régissent ses éléments. Ceci constitue en fait le méta-modèle associé au modèle en question. nous utiliserons une notation basée sur le formalisme UML pour décrire un tel méta-modèle. L'autre approche serait de définir un langage spécifique avec une syntaxe (textuelle) propre. Nous avons opté pour la première solution qui présente l'avantage d'être plus facilement appréhendable, surtout à ce premier stade de définition des concepts. Cette solution présente aussi l'avantage de se prêter naturellement au formalisme MDA (*Model Driven Architecture*) qui peut être une perspective intéressante aux travaux de cette thèse. La sémantique d'exécution associée à chaque élément du modèle, dans le contexte d'un environnement global de cosimulation (SystemC), est également présentée.

4.3.1. Modèle de l'architecture virtuelle

La figure 4.13 présente le modèle conceptuel de l'architecture virtuelle. Vue globalement, le modèle proposé s'apparente à un modèle client/serveur avec un gestionnaire central de ressources

virtuelles (par opposition aux ressources physiques de l'architecture matérielle réelle) qui fait office de serveur et de modules logiques indépendants faisant office de clients. Les modules logiques implémentent les différentes politiques du système d'exploitation. Par exemple, le module ordonnanceur implémente la politique d'ordonnancement des tâches de l'application. Le module « allocation de mémoire » définit quant à lui la politique de gestion de la mémoire dynamique.

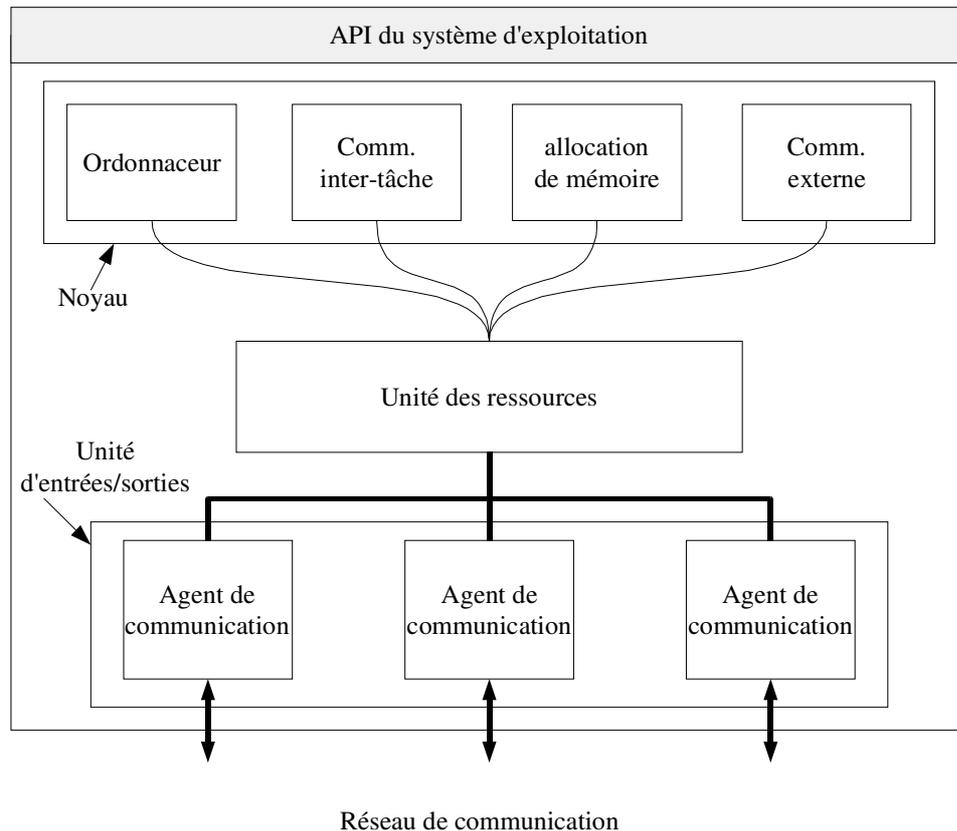


Figure 4.13: Modèle conceptuel d'un nœud logiciel de l'architecture virtuelle

Le serveur des ressources contrôle l'accès aux ressources du système. Parmi ces ressources, nous pouvons citer les ressources de calcul (processeurs), les ressources de stockage (mémoire), les ressources de synchronisation et de communication.

L'interface de programmation offerte par le modèle via l'API du système d'exploitation est l'union de tous les services offerts par les modules logiques. Cette manière de considérer le modèle permet une flexibilité intéressante quant à l'ajout de services spécifiques. L'interface de communication est constituée par les unités d'entrées/sorties. Cette décomposition modulaire selon l'architecture client/serveur est fortement inspirée des travaux récents concernant la conception des systèmes d'exploitation basés sur la notion de composants et ayant une architecture dite « nanokernel ». Cependant, nous appliquons le concept pour la conception du modèle de simulation et non pas au

système d'exploitation final.

4.3.1.1. L'unité des ressources

Cette unité fournit une abstraction de l'ensemble des ressources disponibles au niveau du noeud logiciel. Ces ressources seront partagées à la fois par les tâches de l'application, mais aussi par les "agents de communication" qui matérialisent l'interaction avec l'environnement (matériel) externe. Ces ressources sont constituées de plusieurs types d'éléments.

1) les éléments de traitement (*Processing Element PE*) :

Les éléments de traitement sont responsables de l'exécution du logiciel embarqué. L'unité des ressources spécifie le nombre et les caractéristiques (fréquence, etc.) de chacun de ces éléments. Les PE n'offrent pas une API explicite. Leur utilisation passe impérativement par le biais du noyau (plus particulièrement par l'élément ordonnanceur) qui définit la politique adéquate permettant leur exploitation.

Dans SystemC, un élément de traitement est associé à un processus (`SC_THREAD`). Le paramètre « fréquence » associé à chaque PE intervient lors de l'appel à une fonction `consume()` dans un code logiciel, pour déterminer le temps qu'aurait pris le bout de code sur le processeur cible. Notons que ce paramètre peut très bien varier au cours du temps pour émuler un processeur supportant l'ajustement dynamique de la fréquence dans le cadre d'une politique de gestion d'énergie.

2) l'espace mémoire partagée :

L'espace mémoire partagée, tout comme les éléments de traitement, est une ressource limitée qui fait l'objet d'allocation et d'accès concurrents par les différents éléments du modèle, en l'occurrence par les agents de communication. Cette ressource spécifie la taille et l'adresse de base de l'espace mémoire partagée. Notons que la valeur de l'adresse de base n'est que symbolique à ce niveau.

3) les canaux de contrôle :

Un canal de contrôle représente une requête d'interruption qui provient d'un agent de communication ou d'une ressource locale spécifique (exemple *Timer*) afin d'effectuer un traitement spécifique (via un élément de traitement). Au niveau architecture virtuelle, les détails concernant le mécanisme spécifique utilisé pour la gestion des interruptions et du déploiement effectif de la routine d'interruption sont abstraits. Ce qui nous intéresse à ce niveau est la routine d'interruption elle-même ainsi que la spécification des priorités relatives des différents canaux de contrôle. Ceci permet d'implémenter différentes politiques de gestion d'interruptions au niveau du noyau.

4) les canaux de données :

Un canal de données est utilisé pour effectuer un transfert de données entre un (ou plusieurs) agents de communication et l'espace mémoire partagée décrit plus haut, sans passer par les ressources de calcul. Différents canaux de données correspondent à des accès parallèles à l'espace mémoire partagée faisant abstraction des mécanismes d'implémentation pouvant mettre en oeuvre des transferts DMA, des multiplexages d'accès sur le bus ou des périphériques multi-ports. Dans le cas où plusieurs agents de communication partagent un seul canal de données, une politique d'allocation et d'arbitrage est nécessaire pour multiplexer la ressource partagée.

5) les ressources locales spécifiques :

En plus des ressources de calcul et de communication, l'architecture virtuelle peut inclure des ressources locales fournissant des services spécifiques à l'application. L'exemple type d'une telle ressource est le *Timer*.

4.3.1.2. Le noyau (Kernel)

Le noyau incarne la partie responsable d'implémenter la politique de gestion des ressources partagées du sous-système. Il contrôle ainsi les demandes d'accès à ces ressources venant à la fois de l'application (tâches) et des agents de communication. La gestion des ressources inclut principalement deux aspects :

◆ L'aspect multiplexage

Les ressources disponibles étant généralement limitées, une politique de multiplexage de ces ressources est nécessaire pour subvenir aux besoins de l'application et des agents de communication. L'exemple type est celui de l'ordonnanceur qui implémente un multiplexage temporel des ressources de calcul sur l'ensemble des tâches logicielles de l'application. Les canaux de communication (données et contrôle) peuvent aussi faire l'objet d'arbitrage visant à multiplexer ces ressources limitées sur l'ensemble des agents de communication.

◆ L'aspect synchronisation

Le partage des ressources sur différentes entités concurrentes (tâches de l'application, agents de communication, etc.) impose une politique de protection qui permet d'assurer la cohérence de l'information contenue dans ces ressources. Les services offerts par le système d'exploitation étant eux-même soumis à des accès concurrents, il convient de choisir la politique adéquate pour garantir leur utilisation correcte. A ce stade, différentes approches peuvent être utilisées pour protéger les données partagées du système d'exploitation. Le choix d'une approche particulière parmi d'autres repose généralement sur des critères de complexité (coût) et de performance. Par

exemple, une solution simple serait de considérer le système d'exploitation en entier comme une ressource partagée atomique et d'empêcher l'accès simultané à cette ressource (cas des noyaux dits non-préemptifs). Cette solution souffre cependant d'un problème de performance étant donnée qu'elle limite les possibilités de parallélisme au sein du système d'exploitation. Pour remédier à cette limite, des politiques de protection plus complexes peuvent être utilisées, mettant en oeuvre une granularité plus fine au niveau de la protection des données critiques du système d'exploitation.

Bien qu'il soit difficile d'établir une liste fixe et exhaustive des fonctionnalités pouvant être implémentées par un noyau de système d'exploitation dans le cas général, ces fonctionnalités peuvent néanmoins être regroupées dans des sous-familles correspondant à différents modules logiques :

1) module ordonnanceur : modèle de l'ordonnancement hiérarchique

Le module ordonnanceur est responsable de gérer les ressources de calcul dans un sous-système logiciel. Il fournit à l'ensemble des tâches logicielles « l'illusion » d'un pseudo-parallélisme. Le concept lui-même est générique. En pratique différents types d'ordonnanceurs peuvent être implémentés (parfois dans le même système d'exploitation). Dans le cas où les ressources de calcul sont multiples (cas d'un noeud SMP par exemple), l'ordonnanceur peut profiter de ce parallélisme matériel pour distribuer efficacement les différentes tâches sur les éléments de traitement disponibles. Cette distribution n'est pas forcément statique dans le cas général, puisqu'une politique adéquate de migration de tâches permet une meilleure utilisation des ressources de calcul [Haj99].

Une première approche pour modéliser l'ordonnancement logiciel au sein du modèle de simulation est d'exploiter l'ordonnanceur natif de SystemC à travers la notion de processus offerte par celui-ci. Cependant, cette approche se heurte à deux difficultés. La première vient du fait qu'en utilisant l'ordonnanceur matériel, il sera très difficile de simuler le pseudo parallélisme logiciel, c'est à dire, l'exécution séquentielle des tâches logicielles. En effet, dans un tel ordonnanceur matériel, les processus sont évalués simultanément, d'une façon parallèle et dans un ordre quelconque. Pour simuler, malgré tout l'exécution série des tâches logicielles, on peut faire appel à des mécanismes de synchronisation complexes qui assurent qu'un seul processus exécute à la fois. Cependant cette approche entraîne non seulement la modification du code de l'application, mais aussi un surcoût important au niveau de la performance de simulation. La deuxième difficulté est due au fait que l'ordonnanceur matériel ne fournit pas naturellement le moyen de choisir entre différentes politiques d'ordonnancement et n'implémente pas de mécanismes particuliers basés sur la notion de priorité.

Nous avons plutôt opté pour une autre approche qui consiste à implémenter l'ordonnanceur logiciel d'une façon complètement découplé de l'ordonnanceur matériels, d'où l'idée d'un

ordonnanceur hiérarchique (figure 4.14).

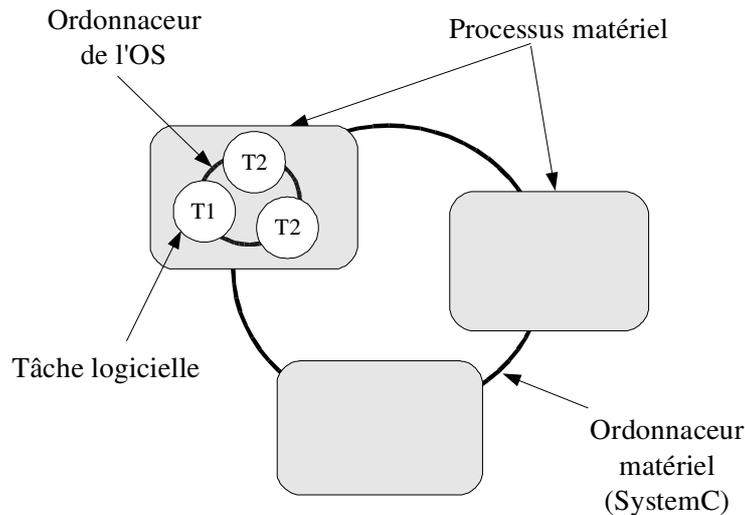


Figure 4.14: Modèle de l'ordonnancement hiérarchique

2) *module synchronisation*

Ce module fournit les services nécessaires pour protéger les données partagées au sein du système d'exploitation. Il est utilisé par les autres modules (communication inter-tâches, agents de communication, ordonnanceur, etc.).

3) *modules communication inter-tâches*

Ces modules sont responsables de fournir les services de communication inter-tâches. Selon le type du système d'exploitation, différents types de communication peuvent être implémentés. Généralement, on distingue la communication par mémoire partagée de la communication par envoi de messages.

4.3.1.3. *L'unité d'entrées/sorties*

L'unité d'entrées/sorties est composée d'un ou plusieurs agents de communication. Conceptuellement, le modèle d'un agent de communication peut être décomposé en deux sous-entités. Une unité d'adaptation au mécanisme de communication et de gestion des interruptions utilisée au sein du système d'exploitation qu'on appelle modèle du pilote et une autre unité d'adaptation au protocole du canal de communication externe qu'on appelle interface fonctionnelle de communication. La synchronisation interne entre les deux sous-unités se fait par le biais d'évènements SystemC.

La figure 4.15 montre deux exemples d'agents de communication correspondant à deux niveaux

d'abstraction différents. Dans le premier cas de figure (partie à gauche), le service fourni au logiciel est un service de lecture (*Get*) d'un entier. La communication avec le matériel est assurée par un canal *fifo* bloquant décrit au niveau TLM message. Dans le deuxième cas, le service fourni au logiciel est un service d'écriture (*Put*) d'un entier et la communication matérielle se fait à travers un protocole *hand-shake* au niveau transfert.

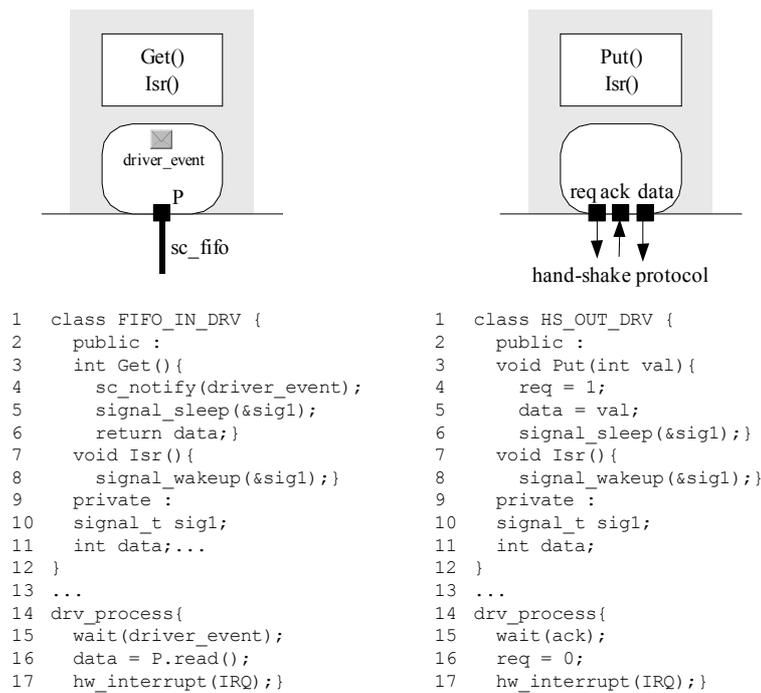


Figure 4.15: Exemples d'unités d'entrées/sorties

Notons que dans les deux cas, l'interface de communication fonctionnelle incarne un processus SystemC dont le rôle est de découpler le processus « parent » (exécutant le code du logiciel) du reste de l'environnement matériel. En effet, il est clair que l'appel au service bloquant de la *fifo* par exemple ne peut être directement fait à partir du code logiciel, sous peine de bloquer tout le processus « parent » et donc toutes les autres tâches logicielles qui s'exécutent au sein de ce processus, ce qui n'est pas le comportement souhaité.

4.3.1.1. Le méta-modèle

Les différents éléments du modèle abstrait de l'architecture virtuelle décrits précédemment ainsi que leurs relations peuvent être exprimés d'une façon plus formelle en utilisant le méta-modèle de la figure 4.16. La zone en gris dans la figure correspond aux éléments génériques du modèle. Les autres

parties correspondent aux éléments spécifiques qui dépendent d'un type particulier d'architecture virtuelle. Ces éléments sont obtenus par extension des éléments génériques de base.

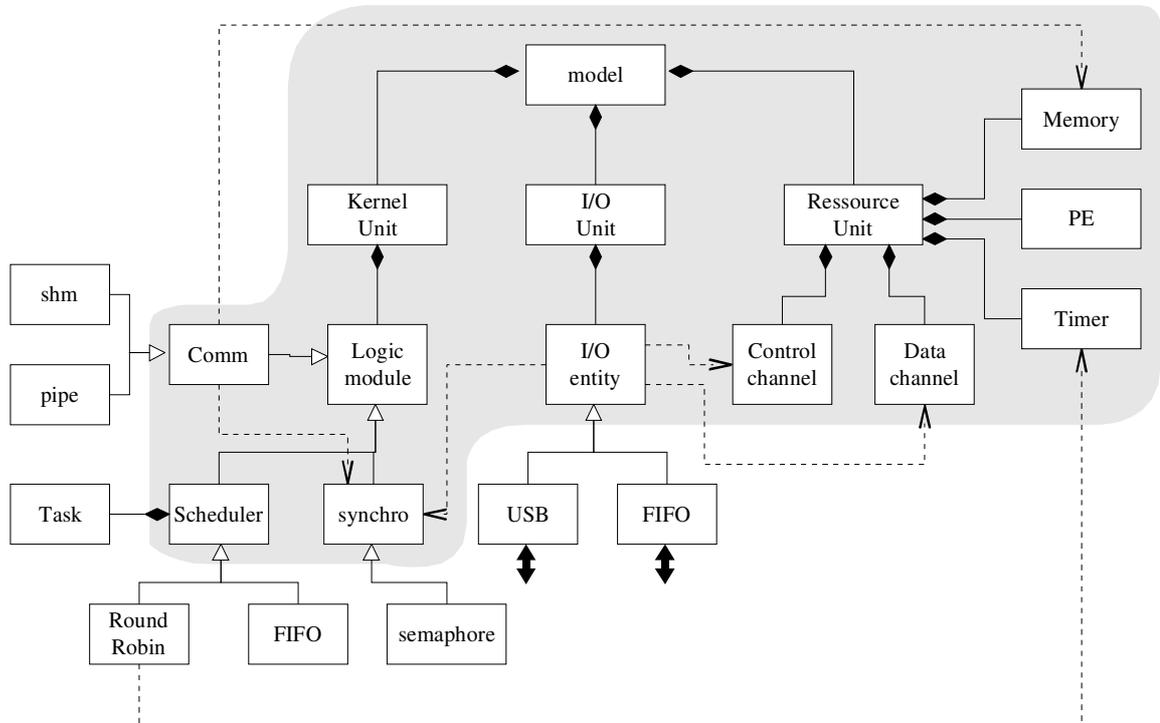


Figure 4.16: Méta-modèle du noeud logiciel de l'architecture virtuelle

4.3.2. Modèle du prototype virtuel

Dans le modèle du prototype virtuel, l'architecture matérielle du sous-système CPU est abstraite via un modèle fonctionnel équivalent qui maintient les caractéristiques pertinentes de celle-ci sans pour autant contenir les détails bas niveau d'implémentation. Ce modèle de l'interface logicielle/matérielle permet d'exécuter l'application logicielle, décrite au niveau HAL, dans un environnement matériel temporel permettant ainsi une évaluation rapide et assez précise de la performance du système et une validation des choix architecturaux qui peuvent être introduits à ce niveau.

Au plus bas niveau d'abstraction auquel nous nous intéressons, un sous-système CPU est vu comme un ensemble de composants matériels communicants via des fils physiques (bus, signaux de contrôle, signaux d'interruptions, etc.). Cette « vue matérielle » est naturelle tant que le logiciel est considéré au niveau ISA. Cependant, elle devient non adéquate dès lors qu'on veut élever le niveau d'abstraction du logiciel. Au niveau HAL, le sous-système entier doit être considéré comme une

entité fonctionnelle qui offre les services requis par l'application. Pour décrire cette entité fonctionnelle, il faut disposer d'un modèle qui permet d'abstraire les détails d'implémentation bas niveau et ne conserver que les aspects fonctionnels de l'architecture du sous-système, tout en permettant une évaluation assez précise de la performance de celle-ci. Pour être utile, ce modèle doit, par ailleurs, permettre de modéliser des architectures réelles et non pas se limiter à des cas simplistes. En particulier, le modèle doit pouvoir exprimer des architectures à fort degré de parallélisme correspondant à un traitement intensif de l'information, mettant en oeuvre des transferts de données multiples et faisant appel à des mécanismes de synchronisation sophistiqués.

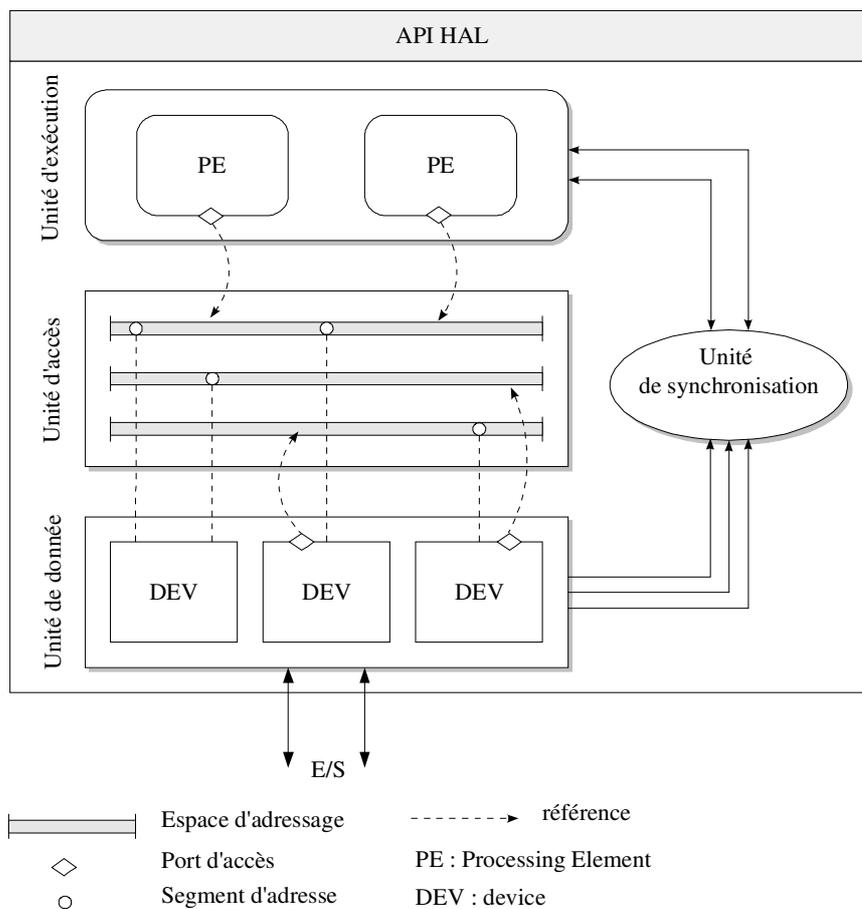


Figure 4.17: Modèle conceptuel de l'interface logicielle/matérielle dans un prototype virtuel

La figure 4.17 illustre le modèle proposé de l'interface logicielle/matérielle dans un prototype virtuel [Bou05]. Il est important de souligner que les différents éléments de la figure ne correspondent pas à des composants matériels physiques mais plutôt à des entités fonctionnelles telle que vues par le programmeur. Par ailleurs, les flèches dans la figure ne sont pas des signaux physiques mais plutôt des relations logiques entre différentes entités fonctionnelles.

Vue globalement, le modèle s'apparente à un modèle classique de Von Neumann, avec une unité

d'exécution et une unité de données. Cependant, contrairement à ce modèle basique de calcul (utilisé par le compilateur), le modèle proposé inclut d'autres aspects systèmes liés à la synchronisation (gestion des interruptions) et aux transferts de données de/vers les entrées/sorties. Par ailleurs, le modèle supporte le parallélisme au niveau de l'unité d'exécution et permet de modéliser la coordination des transferts de données entre les différentes entités.

4.3.2.1. Unité d'exécution

L'unité d'exécution (EU) abstrait le calcul dans un sous-système CPU. Elle est composée d'un ou plusieurs éléments de calcul (*processing element PE*). Un élément de calcul implémente un chemin d'exécution indépendant (*thread*). Cette décomposition du calcul correspond en fait à une vision au niveau tâche du parallélisme matériel (*task level parallelism TLP*) et est explicitement visible pour le programmeur. Ceci inclut, par exemple, aussi bien les architectures homogènes et symétriques de type SMP (*Symmetrical Multi-Processing [Sas03]*) ou SMT (*Simultaneous Multi-Tasking [Sas03]*), que des architectures plus hétérogènes, connus sous le nom générique CMP (*Chip Multi-Processing [Spr05]*). Par conséquent, cette même décomposition ne concerne pas les formes de parallélisme au niveau instruction (*instruction level parallelism ILP*) exhibées à titre d'exemple par les architectures super-scalaires ou de type VLIW. En ce qui nous concerne, nous considérons que ce dernier type de parallélisme est caché par le compilateur et n'apparaît donc pas explicitement dans le modèle. Ceci est, d'ailleurs, valable pour toutes formes d'extension du jeu d'instructions du processeur moyennant des co-processeurs de calcul spécialisés (par exemple une unité de calcul flottant).

Concernant l'API offerte par l'unité d'exécution, celle-ci doit normalement inclure un service de *boot* qui devrait garantir un comportement consistant au moment de l'initialisation du système (surtout dans un contexte multiprocesseurs). Un service d'identification est aussi nécessaire pour différencier les éléments de calcul disponibles ainsi que des services de synchronisation inter-PE (en utilisant le *spin lock* par exemple). Notons que les opérations atomiques utilisées pour implémenter certaines opérations de synchronisation seront fournies par l'unité d'accès.

L'API offerte par un PE doit, par ailleurs, fournir des services de manipulation et de contrôle du chemin d'exécution (*thread*) sous-jacent. Ces services sont nécessaires pour implémenter l'aspect multi-tâches en logiciel au niveau du système d'exploitation. Il s'agit généralement des services d'initialisation et de commutation de contextes.

4.3.2.2. Unité de données

L'unité de données est un conteneur générique pour des éléments de données plus basiques qui sont les éléments périphériques (*device DEV*). Un élément périphérique est une abstraction d'un

périphérique physique qui détient une information pertinente d'un point de vue application. Ceci exclut tout autre périphérique qui ne remplit pas un rôle fonctionnel du point de vue programmeur de l'application (par exemple les contrôleurs et arbitres de bus, contrôleur d'interruptions, etc.).

Un élément périphérique peut être de deux types différents : passif ou actif. Un élément passif correspond à une simple collection d'emplacements mémoire idempotents¹⁶ (exemples RAM, ROM, etc.). Un élément périphérique actif est composé, quant à lui, d'un certain nombre d'emplacements mémoires et d'un comportement sous-jacent susceptible de modifier le contenu de cette mémoire violant ainsi le caractère idempotent de celle-ci.

Les services fournis par l'unité de données doivent être considérés sur la base de chaque élément périphérique. Généralement, un élément passif fournit uniquement des services de lecture/écriture de/vers des adresses spécifiques dans son espace d'adressage. Par opposition, un élément actif peut fournir des services de plus haut niveau qui implémentent des fonctionnalités plus ou moins complexes telles que initialiser un *timer*, programmer un transfert sur un canal DMA, etc. D'un point de vue implémentation, ces services correspondent finalement à une série d'opérations de lecture/écriture sur les registres du périphérique en question. Il s'agit donc d'une vision fonctionnelle de plus haut niveau du périphérique qui abstrait cette série d'opérations par le service qu'elle implémente, ce qui est cohérent avec le concept HAL permettant de se passer des détails d'implémentation.

4.3.2.3. Unité de synchronisation

Dans notre modèle, nous considérons que le processus de synchronisation par interruption est un mécanisme global dont la visibilité traverse les différents autres éléments du modèle. Physiquement, l'implémentation de la gestion des interruptions est généralement distribuée sur plusieurs composants matériels, partant du processeur lui-même et incluant d'autres périphériques spécialisés comme les contrôleurs d'interruptions. Cette implémentation cache souvent un comportement complexe, surtout dans un environnement multiprocesseur où la distribution des sources d'interruption sur les processeurs disponibles fait souvent appel à des algorithmes sophistiqués. Dans le modèle considéré, tout le processus de gestion des interruptions est abstrait par l'unité de synchronisation. Les requêtes d'interruption sont caractérisées par des identifiants globaux et ont des priorités associées qui déterminent leurs précédence dynamique.

L'unité de synchronisation doit fournir un service permettant d'associer une routine de gestion d'interruption (*Interrupt Service Routine ISR*) à une source d'interruption déterminée. Elle doit aussi fournir les services adéquats de contrôle et de gestion du comportement dynamique du gestionnaire

16 Qui préservent la même valeur entre deux accès successifs en lecture

des interruptions tels que l'activation/désactivation des sources d'interruption et/ou la modification de leurs priorités respectives.

4.3.2.4. Unité d'accès

L'unité d'accès correspond à l'abstraction d'une caractéristique clé dans les systèmes MPSoC à savoir la communication ou encore le transfert de données. Cette unité est constituée de deux entités : la collection d'espaces d'adressage et l'ensemble de ports d'accès.

La collection d'espaces d'adressage est un ensemble d'entités indépendantes appelées espaces d'adressage. Par définition, un espace d'adressage abstrait un domaine physiquement adressable qui peut être un bus ou une hiérarchie de segments de bus inter-connectés. A chaque espace d'adressage est associé un ensemble d'éléments périphériques. Cette association constitue le plan des adresses (*address map*) de l'espace d'adressage en question. Notons qu'un élément périphérique peut être « mappé » à plus qu'un espace d'adressage. Dans ce cas, il peut être accédé indépendamment et simultanément depuis plusieurs espaces d'adressages, comme c'est le cas par exemple d'une mémoire à double ports. Dans le modèle considéré, un élément périphérique peut aussi avoir une correspondance virtuelle avec un espace d'adressage (*virtual map*), du moment où la valeur des adresses ou des intervalles d'adresses associés au périphérique n'est pas importante en elle même au niveau d'abstraction considéré (HAL).

Les ports d'accès sont les seuls endroits depuis lesquels un espace d'adressage peut être accédé. Ces ports sont souvent associés aux éléments de calcul (PE) mais peuvent aussi être associés aux entités périphériques qui sont en mesure d'initier des transferts de données (cas d'un périphérique supportant les opérations DMA).

En terme de services offerts, l'entité espace d'adressage doit fournir des opérations élémentaires de lecture/écriture. D'autres services de synchronisation et de résolution de conflits moyennant, par exemple, la sérialisation des accès (*barriers*) ou les opérations atomiques sur le bus peuvent aussi être nécessaires. Concernant les ports d'accès, ceux-ci sont généralement transparents pour le programmeur. Cependant, dans le cas où ils cachent un comportement complexe, comme la translation dynamique des adresses (en utilisant une MMU, par exemple) et/ou l'utilisation d'une hiérarchie de mémoire via un système de caches, des services additionnels peuvent être requis comme l'invalidation du cache et/ou les opérations associées à la table de translation d'adresses.

4.3.2.5. Le méta-modèle

Les différents éléments du modèle abstrait du sous-système CPU décrits précédemment ainsi que leurs relations peuvent être exprimés d'une façon plus formelle dans le cadre d'un langage spécifique

au domaine (DSL) utilisant le formalisme basé sur la notion de méta-modèle. La figure 4.19 dépeint la syntaxe abstraite d'un tel langage. Le méta-modèle se base sur une notation qui s'apparente à UML, où les différents éléments du modèle sont décrits par un diagramme de classe. A chaque service HAL offert par un élément, correspond une fonction membre dans la classe associée. Dans la figure, la zone en gris correspond au "noyau" du méta-modèle. Il s'agit des éléments de base communs à tout type d'architecture locale. Des extensions spécifiques peuvent alors être obtenues en spécialisant ces éléments de base.

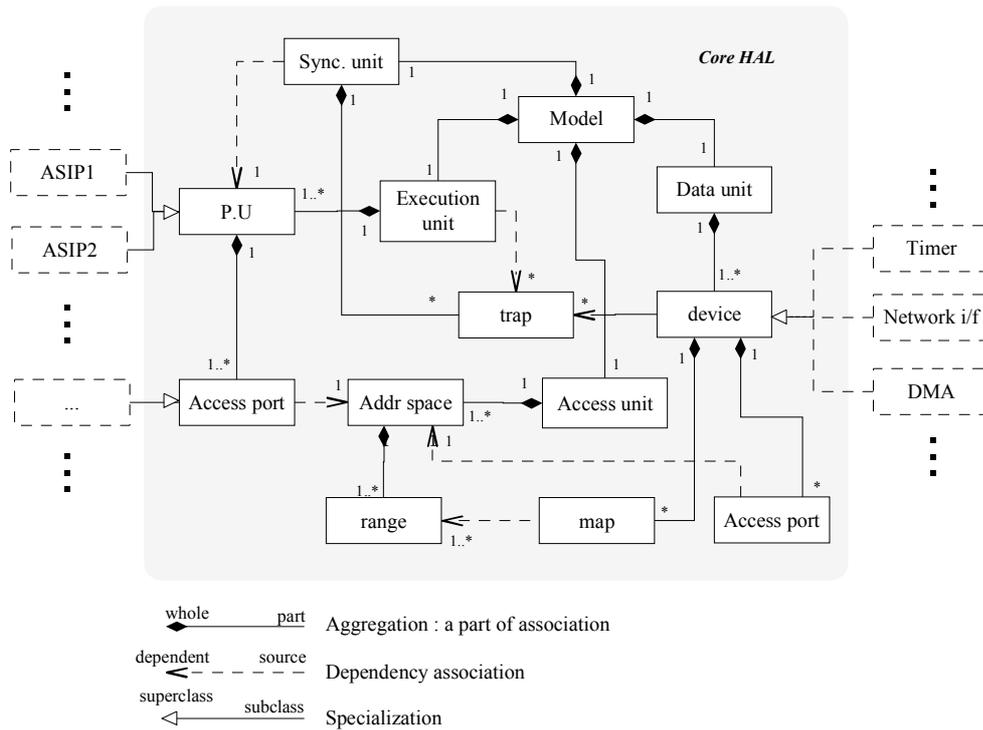


Figure 4.18: Méta-modèle d'un noeud logiciel du prototype virtuel

4.3.3. Algorithme de la fonction *Synch()*

La figure 4.19 montre le principe de base utilisé pour gérer la synchronisation temporelle avec le matériel et la prise en considération des interruptions au niveau de la fonction d'annotation *consume*.

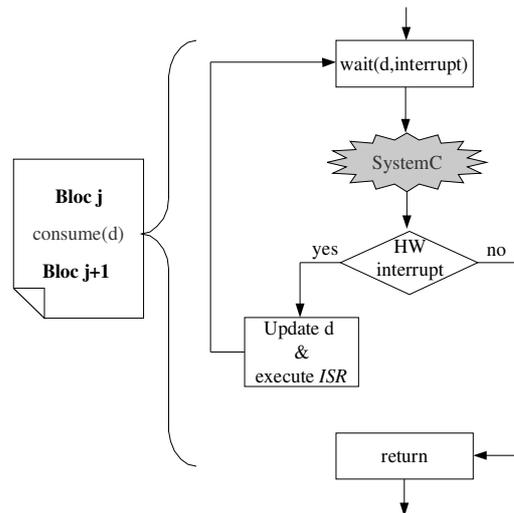


Figure 4.19: Implémentation de la fonction *synch()*

Lorsqu'elle est appelée à l'intérieur du code logiciel, la fonction *synch()* commence par invoquer la fonction *wait(delay, event)* de SystemC. Cette dernière permet de bloquer le processus courant pendant la durée maximale *delay* tant que l'événement *event* n'a pas été déclenché en cours de route. L'appel à cette fonction rend donc la main à l'ordonnaceur SystemC qui va pouvoir mettre à jour l'horloge globale. Lorsque la fonction *wait()* retourne (en fait ceci correspond en réalité à un changement de contexte dans SystemC) deux situations sont envisageables : ou bien le délai initial a été complètement écoulé, au quel cas on sort de la fonction *synch()*. Sinon, c'est qu'il y a eu une interruption matérielle (à travers l'événement *event*) en cours de route. Dans ce cas, la valeur de la variable *delay* est mise à jour pour correspondre au délai restant (s'il n' y a pas eu d'interruption). La routine de gestion d'interruption (ISR) est invoquée et finalement l'algorithme reprend avec l'appel de la fonction *wait* avec la nouvelle valeur du délai.

Il est important de noter que l'ISR peut très bien (ce qui est souvent le cas) transférer le contrôle à l'ordonnaceur du système d'exploitation qui va lui même choisir une autre tâche pour être exécutée. Dans ce cas, la tâche initiale se trouve bloquée au niveau de la fonction *consume* et l'algorithme implémenté par cette fonction est alors « gelé » jusqu'à ce que le contrôle soit transféré de nouveau à la tâche en question.

Dans le cas où le modèle tient compte des conflits d'accès sur le bus système (au niveau du prototype virtuel), la fonction *synch* appelée dans le contexte d'un élément de traitement, permet de modéliser les délais supplémentaires qui peuvent résulter d'un tel accès concurrent. La figure 4.20 illustre une telle situation entre un élément de traitement (processeur) et un périphérique maître (un DMA par exemple) qui accèdent occasionnellement (pendant les périodes dénotées par *shared* dans

la figure) à un même espace d'adressage.

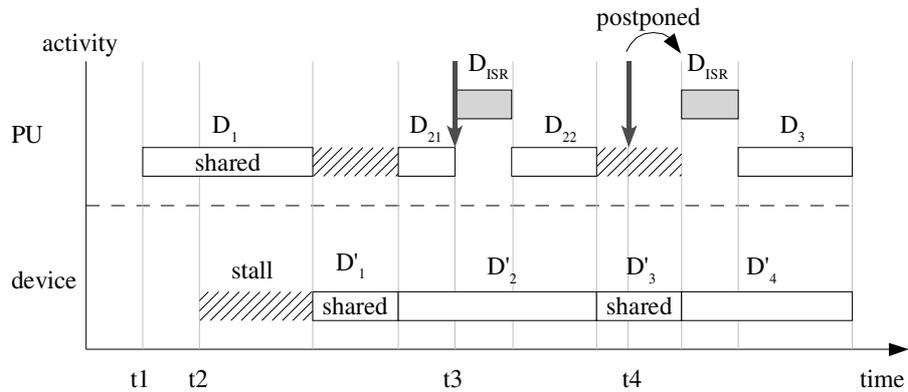


Figure 4.20: Modélisation du conflit d'accès sur le bus

4.4. Conclusion

Dans ce chapitre le concept d'interface logicielle/matérielle a été présenté comme élément clé dans un modèle unifié de représentation pour les architectures des systèmes logiciels/matériels. Des modèles conceptuels de cette interface ont été élaborés à fin de permettre la représentation à différents niveaux d'abstraction des architectures logiciels/matériels.

Chapitre 5

Implémentation, outils et applications

Sommaire

Chapitre 5

Implémentation, outils et applications.....	102
5.1. Implémentation des modèles de simulation sous SystemC.....	103
5.1.1. Structure des bibliothèques de simulation.....	103
5.1.2. L'outil d'annotation semi-automatique du code.....	106
5.1.3. L'environnement MP-SIM.....	108
5.2. Exemple d'illustration	111
5.2.1. Spécification fonctionnelle	111
5.2.2. L'étape de partitionnement : l'architecture virtuelle.....	112
5.2.3. Raffinement de l'architecture globale : le prototype virtuel.....	116
5.3. Applications.....	119
5.3.1. Application VDSL.....	119
5.3.2. Application encodeur MPEG4.....	121
5.4. Conclusion.....	125

Dans ce chapitre, nous présentons, dans une première partie, quelques aspects liés à l'implémentation des différents modèles de simulation décrits dans le chapitre précédent. Le développement réalisé a pour objectif de mettre au point un environnement de simulation et d'exploration qui facilite l'intégration et l'exploitation de ces différents modèles de simulation. Pour cela, un certain nombre de bibliothèques et d'outils d'aide à la conception ont été développés.

Dans la deuxième partie du chapitre, nous décrivons deux exemples d'application qui ont mis en jeu différents modèles de simulations de l'interface logicielle/matérielle, à différents niveaux d'abstraction. Les résultats expérimentaux obtenus nous ont permis alors d'analyser l'intérêt de l'approche proposée, notamment en termes de vitesse et de précision de la simulation

5.1. Implémentation des modèles de simulation sous SystemC

Les différents modèles de simulation de l'interface logicielle/matérielle sont développés et organisés sous forme de bibliothèques de composants (classes C++). Ces bibliothèques permettent étendre l'environnement de base SystemC tout en restant découplées de celui-ci.

Deux outils ont été également développés dans le cadre des travaux menés dans cette thèse, pour faciliter l'exploitation efficace de l'approche proposée. Le premier est un outil d'annotation semi-automatique permettant d'instrumenter un ensemble de fichier sources en introduisant des annotations temporelles au niveau des blocs de base. Le deuxième est une interface utilisateur permettant la simulation, le débogage et l'évaluation des performances à différents niveaux d'abstraction.

5.1.1. Structure des bibliothèques de simulation

5.1.1.1. Conception des bibliothèques de simulation

Au niveau architecture virtuelle, nous avons été amenés à considérer deux cas correspondant à deux scénarios différents d'utilisation du modèle de simulation:

- ◆ le premier scénario correspond à la situation où, partant d'un modèle purement fonctionnel, le concepteur envisage un premier découpage logiciel/matériel et désire valider son choix en évaluant, par exemple, différents politiques d'ordonnancement des tâches logicielles. Dans ce cas, la description fonctionnelle initiale du système doit rester inchangée. Des indications supplémentaires, au plus haut niveau de la hiérarchie SystemC (`sc_main`), permettent alors de distinguer les parties logicielles et matérielles.
- ◆ le deuxième scénario correspond, en réalité, au raffinement du premier scénario, où le concepteur décide de porter la partie logicielle de son application en utilisant une API spécifique de système

d'exploitation. Cet étape implique généralement des transformations dans le code initial des tâches.

Nous avons utilisé une seule bibliothèque pour couvrir les besoins des deux scénarios. La seule différence se situe au niveau du degré d'intrusion par rapport à la bibliothèque de base SystemC. En effet, alors que le deuxième scénario peut être satisfait sans aucune modification de SystemC en utilisant l'approche développée dans le chapitre précédent, le premier scénario est plus contraignant d'un point de vue degré de liberté et nécessite d'intervenir au niveau du code interne de la bibliothèque SystemC. Ces modifications sont toutefois minimales et ne concernent pas le moteur de simulation et sont donc sans effet sur la sémantique d'exécution originale.

Les modèles de simulation de l'interface logicielle/matérielle ont été développés et organisés en bibliothèques C++ qui viennent étendre la bibliothèque de base SystemC. A chaque niveau d'abstraction, la bibliothèque bénéficie d'une structure hiérarchique de classes C++ lui conférant un aspect modulaire et flexible.

La figure 5.2 illustre la structure hiérarchique de la bibliothèque de simulation d'OS. Seules les parties ombrées font partie de la bibliothèque proprement dite. Les autres éléments font partie de la bibliothèque de base SystemC et sont représentés à fin d'explicitier les relations qui existent entre les deux bibliothèques.

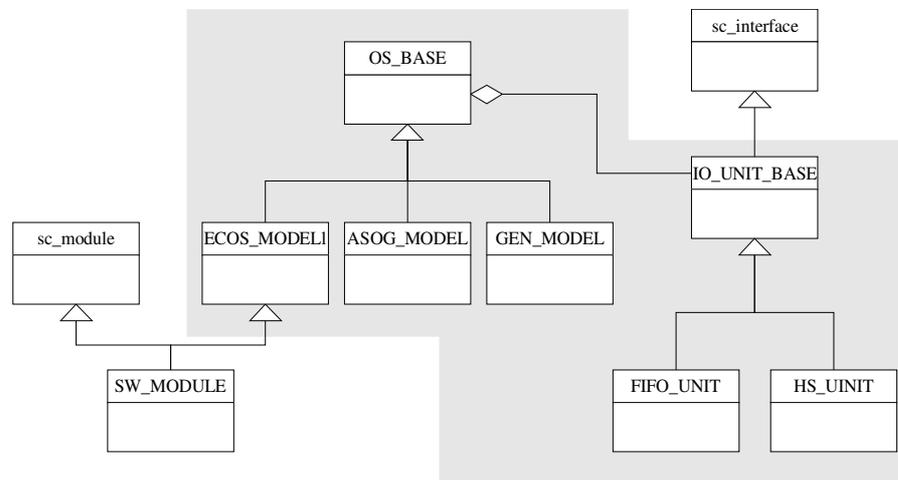


Figure 5.1: diagramme de classe de la bibliothèque de simulation d'OS

La classe de base *OS_BASE* occupe le plus haut niveau de la hiérarchie. Cette classe correspond, en fait, au serveur de ressource dans le modèle conceptuel de l'architecture virtuelle (chapitre 4). Elle incarne les fonctionnalités de base d'un modèle de simulation d'OS, permettant ainsi à différents modèles de systèmes d'exploitation d'être implémentés par simple dérivation. Plus spécifiquement, *OS_BASE* implémente les mécanismes de base pour gérer l'aspect multi-tâche. Cependant, elle ne

spécifie aucune politique concrète d'ordonnancement et se limite à la déclaration du prototype virtuel de la fonction d'ordonnancement. Chaque classes dérivée doit ainsi implémenter la politique d'ordonnancement spécifique au modèle d'OS en question. La figure laisse distinguer trois modèles d'OS déjà développés au cours de cette thèse. Le premier implémente une API générique et sert notamment pour dans le cas du premier scénario. Le deuxième et le troisième modèle implémentent des API réelles et correspondent respectivement au système d'exploitation développé au sein du groupe dans le cadre de l'outil ASOG et à eCos, un système d'exploitation du domaine du libre [Ecos]. Les agents de communication sont représentés, dans la figure, par la classe de base *IO_UNIT_BASE* qui est liée à la classe *OS_BASE* par une relation d'agrégation (dans un modèle d'OS, on peut évidemment avoir plusieurs agents de communication). Comme pour le cas de *OS_BASE*, la classe de base *IO_UNIT_BASE* n'implémente pas un agent de communication spécifique, mais représente plutôt des fonctionnalités génériques servant à développer des modèles spécialisés par simple dérivation. La classe *IO_UNIT_BASE* hérite, elle-même, de la classe *sc_interface* qui, elle, fait partie de la bibliothèque SystemC. Cette forme d'héritage permet de connecter un agent de communication aux canaux SystemC standards (du moment où ces canaux implémentent les mêmes services déclarés par l'interface de l'agent de communication).

Dans la figure précédente, la classe *SW_MODULE* représente la partie utilisateur et ne fait pas partie de la bibliothèque de simulation proprement dit. Elle est déclarée, par l'utilisateur, dans le cadre du deuxième scénario, pour décrire l'architecture virtuelle d'un noeud logiciel, comme étant un module SystemC conventionnel qui hérite en plus d'une classe correspondant à un modèle particulier de système d'exploitation.

Le développement de la bibliothèque de simulation relative au prototype virtuel a suivi la même démarche « orientée objet ». En effet, la structure en classe de la bibliothèque correspond parfaitement au méta-modèle du prototype virtuel décrit dans le chapitre précédent.

5.1.1.2. L'environnement d'exécution

Dans les deux cas (architecture virtuelle et prototype virtuel), la partie correspondant au logiciel embarqué est compilée séparément comme étant une bibliothèque indépendante de l'environnement SystemC. Ceci a un double objectif. Le premier est de faciliter l'utilisation d'un langage autre que C++ pour le logiciel embarqué (en l'occurrence C). Le deuxième est de permettre l'utilisation du même code à la fois pour la simulation et l'implémentation finale, sans aucune interférence liée à l'aspect simulation. Dans cette même perspective, l'outil d'annotation, ayant pour objectif d'instrumenter le code source pour la simulation, a été conçu de manière à ne pas modifier ce code (en générant une copie représentant le code instrumenté).

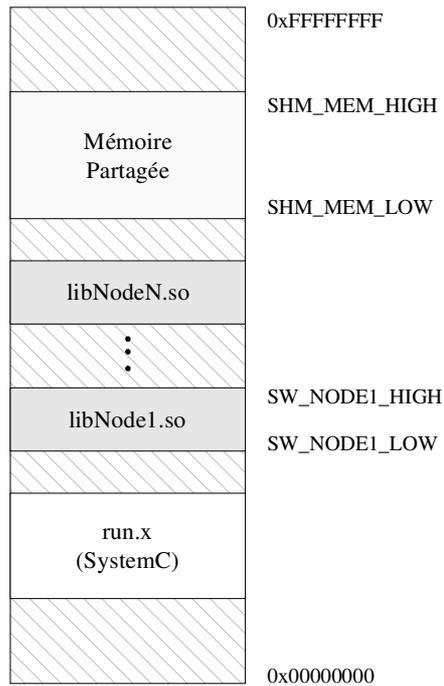


Figure 5.2: Plan mémoire du processus de simulation

5.1.2. L'outil d'annotation semi-automatique du code

L'outil d'annotation automatique de code se base sur l'approche décrite au chapitre 2 en faisant les hypothèses détaillés dans la section 4.2.2.3 du chapitre 4. La figure montre l'architecture de l'outil en question. Il s'agit en effet d'un ensemble d'outils qui coopèrent afin de réaliser l'objectif final qui est l'annotation du code source par les estimations de performance. Plus précisément, on distingue principalement trois outils : un analyseur de code binaire, un estimateur de performance et un instrumenteur de code source.

Pour développer l'analyseur binaire au format ELF, nous nous sommes basés sur un outil existant du domaine du libre, appelé DIABLO [Van05] que nous avons adapté pour répondre à nos besoins. Cet outil est à l'origine prévu pour l'optimisation globale du code binaire après l'étape d'édition des liens. Pour ce faire, il se base sur une phase d'analyse statique des blocs de base du code binaire afin de déterminer le flot de contrôle du programme (*CFG*). Nous avons donc récupéré cette partie et nous l'avons adaptée pour s'interfacer avec les autres outils du flot d'annotation.

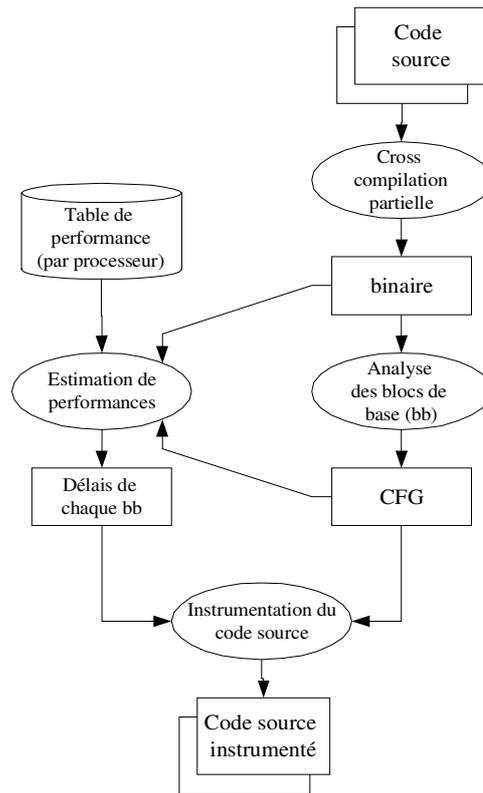


Figure 5.3: outil d'annotation automatique du code source

L'intérêt majeur de cet outil est qu'il permet de rendre automatique une tâche qui est très fastidieuse à faire manuellement, surtout vue la taille du code logiciel auquel on a affaire. A titre d'illustration, la figure 5.4 donne un aperçu sur un exemple de flot de contrôle obtenu pour une partie relativement petite du code d'un pilote de périphérique.

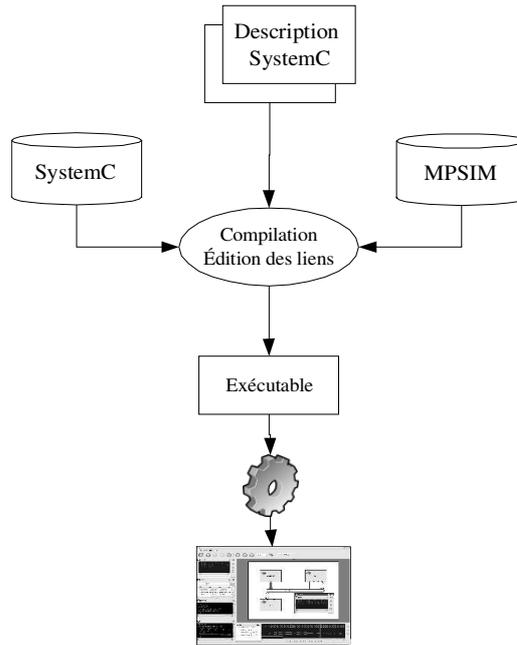


Figure 5.5: MP-SIM comme front-end pour SystemC

Durant la phase d'initialisation, la description SystemC du système est dynamiquement traversée et une vue graphique de la structure hiérarchique (modules) du système est affichée. Notons, à ce niveau, que l'outil n'impose pas de restrictions quant aux types des composants SystemC utilisés dans la description originale. En particulier, l'environnement MP-SIM peut représenter des systèmes décrits à différents niveaux d'abstraction.

L'explorateur graphique de la structure du système (figure 5.6) permet une navigation rapide dans cette structure tout en ayant la possibilité d'effectuer différentes actions de visualisation (monitoring) et/ou de débogage associés à des éléments spécifiquement sélectionnés de la structure. En particulier, les canaux de communication (signaux standards, bus, canaux utilisateurs, etc.) peuvent être marqués pour une visualisation interactive au cours de la session de simulation/débogage. Les processeurs (ou plus généralement les unités d'exécution aux différents niveaux d'abstraction) peuvent être associés à des débogueurs en ligne (actuellement l'outil supporte GDB). Dans le cas d'un système multiprocesseur, le déroulement de la session de débogage est contrôlé par un processeur maître (sélectionné par l'utilisateur).

Finalement, l'environnement est conçu d'une façon modulaire pour faciliter son extension en utilisant des greffons (*plugins*) qui peuvent implémenter, par exemple, différents types d'indicateurs de performance (statistiques sur l'utilisation du bus, taux des défauts de cache, etc.)

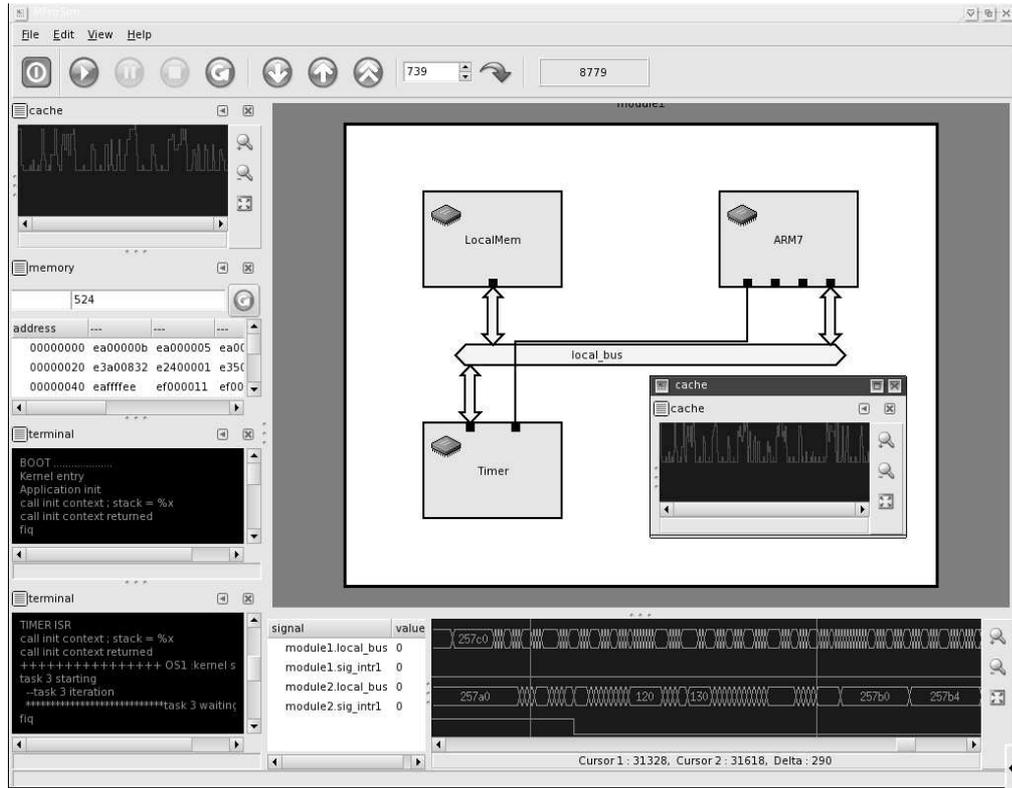


Figure 5.6: Aperçu sur l'environnement MP-SIM

La figure 5.7 montre la technique utilisée pour renforcer le pouvoir de détection des bugs dans le code logiciel en mode natif. Il s'agit d'isoler, dans l'espace d'adressage du processus de simulation, les parties relatives au logiciel embarqué du reste du simulateur.

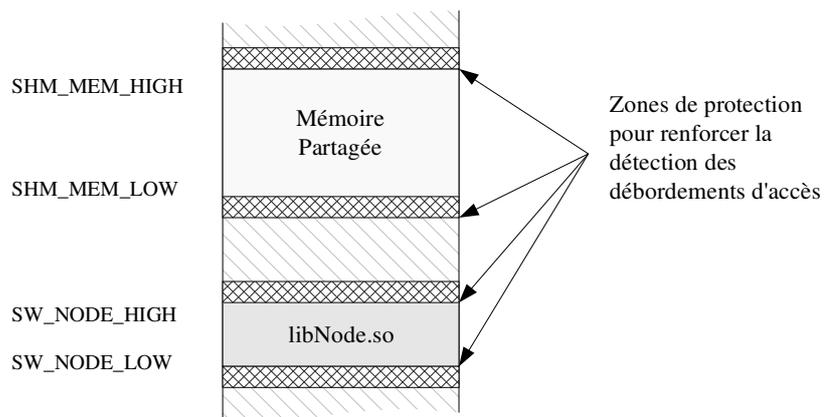


Figure 5.7: Renforcement du pouvoir de détection d'erreurs dans la simulation native

Cette isolation concerne aussi bien l'espace occupé par la bibliothèque dynamique correspondant au noeud logiciel en question, que la mémoire partagée utilisée au sein du modèle du système. Bien que cette technique ne garantie pas une isolation complète du logiciel embarqué, elle permet, néanmoins, d'éliminer un nombre important de bugs, très tôt dans le cycle de conception. Ceci constitue un avantage très important d'autant plus qu'on se situe dans un contexte multiprocesseur caractérisé par la difficulté et la complexité du débogage bas niveau[You04].

5.2. Exemple d'illustration

Dans cette section, nous illustrons la méthodologie de validation multi-niveaux de l'interface logicielle/matérielle à travers un exemple académique.

5.2.1. Spécification fonctionnelle

l'exemple est constitué de cinq tâches concurrentes : quatre tâches qui réalisent une chaîne de production / échange des données (T5 -> T4 -> T1 -> T2) et une tâche (T3) qui, à tout moment, doit être réactive à un événement externe. Lorsqu'elle est activée, la tâche T3 utilise une ressource qui est partagée avec la tâche T2 et protégée par un mécanisme d'exclusion mutuelle (*mutex*).

Les tâches T1, T2, T4 et T5 communiquent à travers des FIFO de tailles finies.

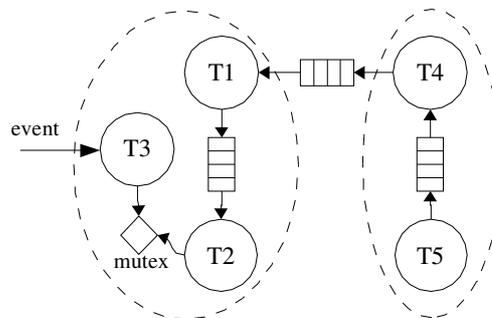


Figure 5.8: Spécification fonctionnelle de l'exemple

Une décision est prise de regrouper les tâches T1, T2 et T3 dans une unité logique (module) et les tâches T4 et T5 dans une autre unité. A ce stade, aucun raffinement architectural n'a été fait (le regroupement des tâches en module peut être considéré comme purement logique et pas forcément un partitionnement logiciel/matériel). La simulation de l'ensemble dans l'environnement SystemC permet de valider la spécification. Les FIFO sont réalisées par des primitives SystemC de haut niveau de type *sc_fifo*.

Remarque : dans ROSES, le concept de module virtuel et port hiérarchique permet de combiner différents niveaux d'abstraction de la communication et de générer automatiquement les adaptateurs nécessaires en cas de non adaptation. Dans cet exemple, on n'utilisera pas cette caractéristique.

La figure suivante représente un diagramme temporel montrant l'évolution des états des différentes tâches de l'exemple (r = *running*, b = *blocked*) et du signal *clock* utilisé comme déclencheur de l'évènement *event*.

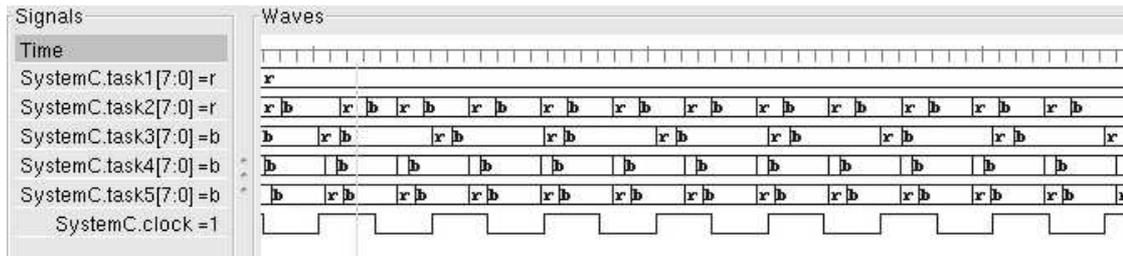


Figure 5.9: trace de la simulation fonctionnelle

Remarquons la régularité des différentes formes d'onde qui reflète l'aspect « idéal » d'une simulation purement fonctionnelle. Remarquons aussi le fait que la tâche T3 se déclenche toujours d'une manière instantanée à l'occurrence d'un front montant du signal *clock* (matérialisant l'évènement externe) ce qui est le comportement souhaité.

5.2.2. L'étape de partitionnement : l'architecture virtuelle

Dans une deuxième phase, une décision architecturale a été prise concernant la réalisation des différents modules du système qui seront alors en logiciel.

A ce niveau, l'utilisation d'un modèle abstrait de (RT)OS s'avère intéressant pour évaluer d'une manière plus précise les performances (effet de l'ordonnancement) mais aussi de valider le bon comportement temporel de l'application (respect des échéances des différentes tâches, etc.), un comportement qui peut être très variable selon le modèle d'OS utilisé.

Dans notre cas, on a utilisé le même type de modèle pour les deux modules du système (un par module). Ce modèle implémente un ordonnanceur préemptif basé sur les priorités des tâches.

5.2.2.1. Premier scénario : utiliser le même code SystemC pour les tâches logicielles

Au niveau spécification, aucun changement par rapport à la spécification fonctionnelle n'est nécessaire. Il suffit d'indiquer quel module logique est attaché à quel modèle d'OS abstrait comme illustré par la figure 5.10.

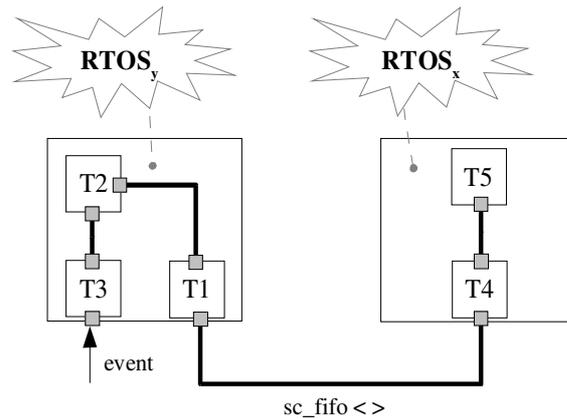


Figure 5.10: exemple d'utilisation correspondant au premier scénario

Au niveau de la description SystemC (figure 5.11), ceci revient, au plus haut niveau de l'hierarchie, à déclarer les entités représentant les différents modèles de simulation des systèmes d'exploitation (lignes 6 et 7) et à spécifier quel processus SystemC sera géré par quel modèle de système d'exploitation (lignes 8 à 12). Dans l'exemple de la figure, cette spécification comprend également la priorité qui sera associée à chaque tâche.

```

1: int sc_main()
2: {
3:     module1_type module1("module1");
4:     module2_type module2("module1");
5:     ...
6:     sc_rtos rtos1("RTOS1","gen_rtos.conf");
7:     sc_rtos rtos2("RTOS2","gen_rtos.conf");
8:     rtos1.attach_task("module1.P1",5);
9:     rtos1.attach_task("module1.P2",3);
10:    rtos1.attach_task("module1.P3",10);
11:    rtos2.attach_task("module2.P4",5);
12:    rtos2.attach_task("module2.P5",3);
13:    ...
14:    sc_start(-1);
15: }

```

Figure 5.11: exemple d'utilisation correspondant au premier scénario

La figure 5.12 représente un diagramme temporel montrant l'évolution des états des différentes tâches logicielles (r = *running*, b = *blocked*, w = *waiting* = *ready*).

Par rapport à la forme d'onde obtenue précédemment, la présente forme d'onde est nettement moins régulière avec l'apparition de l'état « waiting » qui dénote une tâche qui est prête à être exécutée, mais mise en file d'attente par l'ordonnaceur de l'OS parce que une tâche de plus forte priorité est en train de s'exécuter. La plus forte priorité a été donnée à la tâche T3 pour assurer au

maximum sa réactivité à l'évènement externe (vue comme une interruption matérielle).

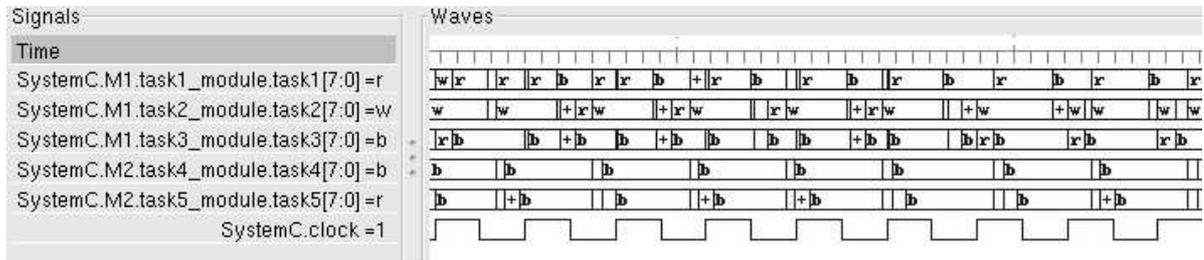


Figure 5.12: trace de la simulation avec modèle d'OS

Un autre phénomène qui peut avoir des conséquences graves (à l'image de la mission Pathfinder sur mars en 1997) est celui de l'inversion de priorité. Ce phénomène se traduit par le retardement d'exécution d'une tâche de forte priorité (H) par une tâche de moins forte (moyenne) priorité (M). Ceci est possible quand la tâche (H) se trouve bloquée par une tâche de faible priorité (L) (parce que celle-ci possède une ressource partagée entre les deux) et que la tâche (M) vient interrompre l'exécution de (L) « à la régulière » parce que l'ordonnanceur est préemptif.

La figure 5.13 montre un cas d'inversion de priorité révélé par les tâches T3 (H), T1(M) et T2(L).

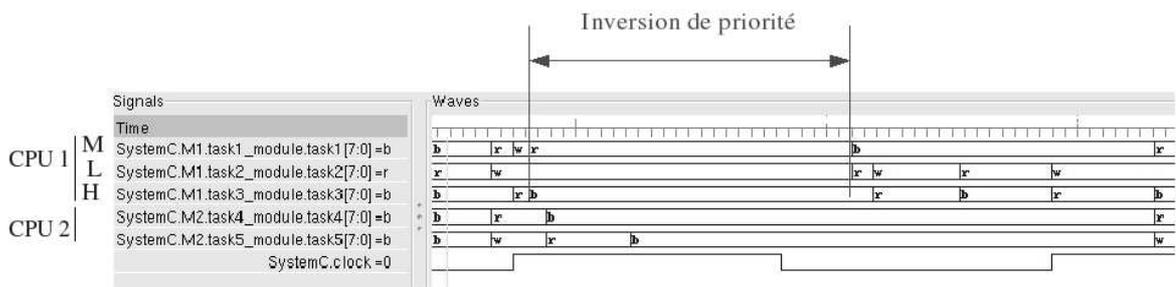


Figure 5.13: phénomène d'inversion de priorité

Le niveau « OS abstrait » permet de mettre en relief l'effet de l'ordonnancement des tâches logicielles sur le système entier. Cependant, aucune autre information sur l'architecture n'est incluse. Ainsi, la communication et les mécanismes de synchronisation (interruptions) sont considérés idéaux comme pour le cas de la simulation fonctionnelle. Ces informations ne peuvent être obtenues qu'après raffinement de l'architecture de la communication et du calcul.

5.2.2.2. Deuxième scénario : raffinement de l'application pour une API donnée de système d'exploitation

L'étape suivante consiste à raffiner l'application en utilisant une API spécifique du système d'exploitation [Tan03]. Ceci correspond au choix d'un système d'exploitation particulier. Bien qu'une API générique puisse être utilisée à ce niveau, il est plus logique d'utiliser une API réelle pour éviter un double effort de raffinement (finalement, le choix d'un OS réel est inévitable). Notons que dans certains cas, ce raffinement peut être automatisé comme c'est le cas de l'outil ASOG utilisé dans le flot ROSES. Dans ce cas précis, le même code initial SystemC est utilisé avec un système d'exploitation propriétaire spécifiquement développé à cet effet. Dans le cas général, l'étape de raffinement de l'application se fait d'une façon manuelle à cause de certains aspects non fonctionnels qui empêchent d'avoir une relation univoque (bijective) entre le code SystemC initial et le code applicatif final (performance, protection mémoire, etc.).

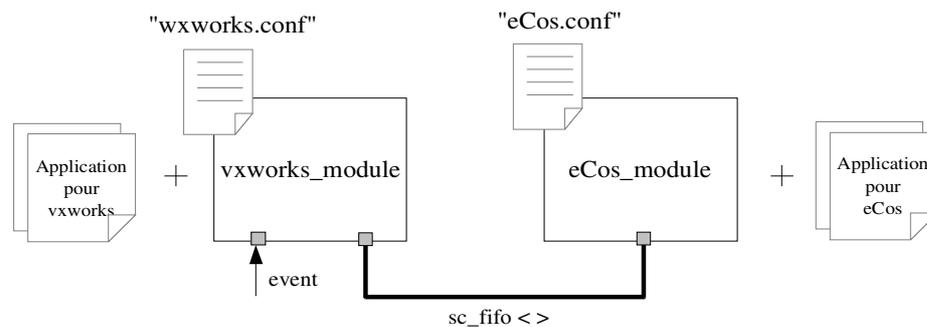


Figure 5.14: exemple d'utilisation correspondant au deuxième scénario

La figure 5.14 montre un raffinement possible de l'exemple précédent, où deux systèmes d'exploitation différents (vxworks et eCos) sont utilisés pour le raffinement des deux noeuds logiciels. Notons que les processus SystemC à l'intérieur de chaque module disparaissent et laissent place au code applicatif raffiné pour le système d'exploitation en question.

Un exemple de spécification de module "logiciel" (*vxworks_module*) est donné par la figure 5.15.

```

1: class VXWORKS_MODULE : public sc_module, RTOS_MODEL
2: {
3: public :
4: ...
5: //SystemC user ports declaration
6: ...
7: VXWORKS_MODULE(sc_module_name name, char * conf_file) :
8:   sc_module(name),RTOS_MODEL(conf_file)
9: {
10:   SC_THREAD(SW_process)
11:   //other initializations

```

```

12: }
13: ...
14: void SW_process()
15: {
16:   rtos_model.kernel();
17: }
18: ...
19: };

```

Figure 5.15: spécification du module SystemC "VXWORKS_MODULE"

le passage du fichier de description du modèle du système d'exploitation utilisé (en l'occurrence *vxworks*) est fait au niveau du constructeur de la classe *VXWORKS_MODULE* (lignes 7 et 8). Cette classe hérite doublement de la classe de base *sc_module* de SystemC et de la classe générique *RTOS_MODEL*. La spécialisation du modèle se fait dynamiquement via le fichier de configuration.

```

1: #include "vxWorks.h"
2:
3: SEM_ID syncSem; /* ID of sync semaphore */
4:
5: init ( int someIntNum )
6: {
7:   intConnect (INUM_TO_IVEC (someIntNum), eventInterruptSvcRout, 0);
8:   syncSem = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
9:   taskSpawn ("sample", 100, 0, 20000, task1, 0,0,0,0,0,0,0,0);
10:  consume(N1);
11: }
12:
13: task1 (void)
14: {
15:   ...
16:   semTake (syncSem, WAIT_FOREVER); /* wait for event to occur */
17:   ...
18:   consume(N2);
19: }
20:
21: eventInterruptSvcRout (void)
22: {
23:   consume(N3);
24:   semGive (syncSem); /* let task 1 process event */
25:   ...
26: }

```

Figure 5.16: exemple de raffinement du code applicatif pour *vxworks*

5.2.3. Raffinement de l'architecture globale : le prototype virtuel

A ce niveau d'abstraction, le système est vu comme un ensemble de noeuds de calcul et de stockage inter-connectés par un réseau de communication. Ce niveau est d'une importance capitale dans la conception d'une architecture SoC parce qu'il permet d'explorer un aspect fondamental dans ces architectures qui est la communication. Cette importance est d'autant plus ressentie aujourd'hui avec les systèmes à base de NoC mettant en oeuvre des mécanismes de communication sophistiqués.

Plusieurs travaux récents ont souligné l'importance du niveau TLM comme niveau adéquat pour modéliser l'architecture de la communication à un haut niveau d'abstraction. Dans cet exemple, il est retenu comme moyen de modéliser le réseau de communication à ce stade de la conception.

Ce raffinement de la communication doit aussi être accompagné d'un raffinement des noeuds de calcul, où des informations plus précises sur leur comportement temporel est nécessaire pour aboutir à des résultats significatifs et en cohérence avec le niveau TLM. Dans le cas de cet exemple, supposons que l'on se place dans la situation où l'architecture matérielle cible existe déjà. Supposons aussi qu'on a choisi la plate-forme « ARM Integrator » comme architecture matérielle candidate.

La plate-forme «ARM integrator » est représentée dans la figure 5.17. Seuls deux des quatre modules ARM sont illustrés dans la figure. Pour le but de l'exemple aussi, quelques modifications ont été introduites sur l'architecture locales de chaque module processeur (notamment l'ajout d'un *TIMER* et d'un contrôleur d'interruption local).

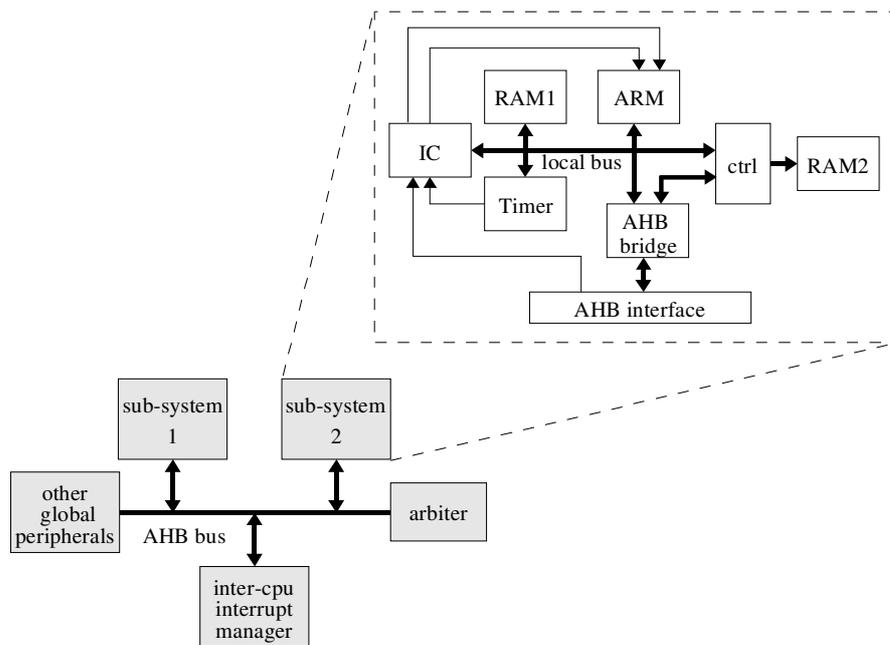


Figure 5.17: Architecture simplifiée de la plateforme ARM Integrator

La figure 5.18 représente le fichier de description utilisé pour configurer le modèle abstrait de chaque sous-système CPU dans le modèle du prototype virtuel. Il s'agit d'une représentation textuelle spécifique du méta-modèle générique décrit dans le chapitre précédent. Notons aux lignes 39 et 47 la déclaration de deux espaces d'adressage correspondant au bus système et au bus auxiliaire relatif à la mémoire RAM2.

```

1: RESOURCES(LocalArch)
2: {
3:     ALIAS ARM_MODULE_BASE = 0x80000000
4:     EXEC_UNIT {
5:         ARM7_CORE core {
6:             PARAM Frequency(60)
7:         }
8:     }
9:     SYNCH_UNIT {
10:        INTR_OUT out
11:        INTR_IN fiq1(255)
12:        INTR_IN irq1(1)
13:        INTR_IN irq2(2)
14:        INTR_IN irq3(2)
15:        BIND (fiq1,out1)
16:    }
17:    DATA_UNIT {
18:        DEVICE ram1:SRAM {
19:            REGS 0x0003FFFF
20:        }
21:        DEVICE ram2:SDRAM {
22:            REGS 0x0FFFFFFF
23:        }
24:        DEVICE timer:TIMER {
25:            REGS virtual
26:            INTERRUPT irq1
27:        }
28:        DEVICE ahb:AHB_IF {
29:            REGS 0xEFFFFFFF
30:            INTERRUPT irq1
31:            INTERRUPT irq2
32:            INTERRUPT irq3
33:        }
34:    }
35:    ACCESS_UNIT {
36:        ALIAS RAM1_BASE = 0x00000000
37:        ALIAS RAM2_BASE = 0x00040000
38:        ALIAS AHB_BASE = 0x10000000
39:        SYSTEM ADS bus1 {
40:            ADDRESS 32
41:            DATA_WIDTH 32
42:            ACCESS core
43:            MAP ram1(RAM1_BASE)
44:            MAP ahb(AHB_BASE)
45:            REMAP bus2(RAM2_BASE)
46:        }
47:        ADS bus2 {
48:            ADDRESS 32
49:            DATA_WIDTH 32
50:            ACCESS ahb
51:            MAP ram2(ARM_MODULE_BASE)
52:        }
53:    }
54: }

```

Figure 5.18: Fichier de description de l'architecture locale dans prototype virtuel

La figure 5.16 montre un exemple de code raffiné du système d'exploitation correspondant à la fonction de lecture des données d'un pilote de périphérique. Notons à ce niveau (lignes 7, 12, 21...) l'utilisation des services de bas niveau fournis par l'API HAL.

```

1: int drv_fifo_read(drv_fifo_t * drv)
2: {
3:     consume(17); // prologue
4:     int data;
5:     int temp_ri = READ(drv->state);
6:     consume(24);
7:     if((READ(drv->state + 4) == temp_ri)&&(! READ(drv->state + 8)))
8:     {
9:         printf("PROCESSOR 1 : drv fifo sleep\n");
10:        signal_sleep(&drv->sig_w);
11:        printf("PROCESSOR 1 : drv fifo awaked\n");
12:    }
13:    data = READ(drv->buffer + temp_ri);
14:    temp_ri ++;
15:    consume(7);
16:    if(temp_ri == drv->size)
17:    {
18:        temp_ri = 0;
19:        consume(8);
20:    }
21:    WRITE(temp_ri,drv->state);
22:    WRITE(0,drv->state + 8);
23:    //send an interrupt to the "other" processor
24:    WRITE(0x00000010,0x14000050);
25:
26:    consume(12); //epilogue
27:    return data;
28: }

```

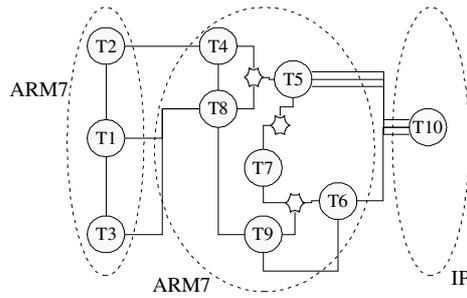
Figure 5.19: Partie du code du pilote de périphérique DRV_FIFO

5.3. Applications

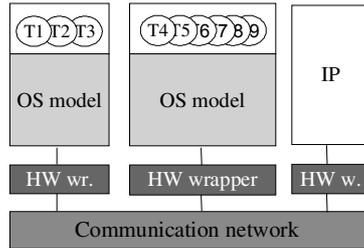
Dans cette section, nous présentons quelques résultats expérimentaux correspondant à deux applications différentes : un modem VDSL et un encodeur vidéo MPEG4. Dans la première application (VDSL), le modèle de simulation d'OS est utilisé et est comparé à un modèle fonctionnel et à un autre précis au niveau cycle de la même application. Dans la deuxième application (MPEG4), il s'agit d'appliquer le modèle abstrait de l'architecture locale du sous-système CPU. Les résultats obtenus avec ce modèle sont comparés à un modèle précis au niveau cycle.

5.3.1. Application VDSL

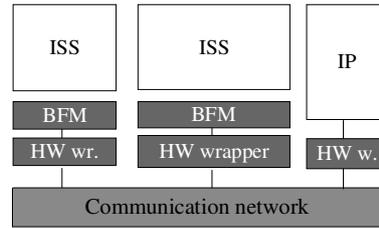
L'application VDSL est en réalité un sous-système d'un système plus large implémentant un modem se basant sur le standard VDSL. Une description détaillée du système peut être trouvée dans [Ces04]. Le sous-système qui nous intéresse est composé de deux processeurs ARM7 et d'un bloc matériel déployé sous forme d'IP. Le premier et le deuxième noeud logiciel se chargent de l'exécution de trois et de cinq tâches logicielles respectivement.



(a) Modèle fonctionnel



(b) Modèle de l'architecture virtuelle



(c) Modèle ISA/RTL

Figure 5.20: Différents niveaux de modélisation de l'application VDSL

La simulation du système est effectuée à trois niveaux d'abstraction différents : fonctionnel, architecture virtuelle, et micro-architecture

La table montre les résultats obtenus pour les trois niveaux d'abstraction. La précision du modèle est définie par rapport au temps requis par une tâche logicielle pour accomplir un traitement donné. Nous considérons le temps mis par un modèle de simulation précis au niveau cycle comme référence. La précision moyenne d'un modèle donné est calculée comme suit :

$$précision \equiv \left(1 - \frac{\sum_{all\ tasks} |Tca - Te|}{\sum_{all\ tasks} Tca}\right) \times 100$$

Où Tca est le temps référence requis par une tâche dans le modèle précis au niveau cycle et Te est le temps mis par cette même tâche au niveau d'abstraction considéré.

Les résultats obtenus (tableau) montrent que la simulation au niveau architecture virtuelle permet une accélération considérable par rapport à une simulation au niveau cycle utilisant un ISS (plus de quatre ordre de grandeur) et s'approche ainsi de la vitesse de simulation d'un modèle fonctionnel. Le surcoût en terme de vitesse de simulation par rapport à un modèle fonctionnel est maintenu faible (moins de 15%) grâce notamment à l'efficacité de l'implémentation de l'ordonnanceur hiérarchique.

	Temps de simulation	précision
Niveau fonctionnel	2,2 s	19%
Niveau architecture virtuelle	2,5 s	86%
Niveau micro-architecture	32562 s	100%

Table : résultats comparatifs de la vitesse et la précision de la simulation aux différents niveaux d'abstraction considérés

En terme de précision, le modèle fonctionnel fournit, comme prévu, les résultats les moins bons. En effet, à ce niveau, ni la séquentialité des tâches logicielles (pseudo-parallélisme) ni le surcoût infligé par le système d'exploitation n'est considéré. En utilisant la simulation au niveau architecture virtuelle, ces aspects sont désormais pris en charge par le modèle et la précision de la simulation du système entier est considérablement améliorée, sans pour autant atteindre la précision absolue d'un modèle cycle à base de ISS. Ceci est dû notamment aux erreurs introduites au niveau de l'estimation du temps d'exécution du logiciel et au niveau de la modélisation de l'interface de communication.

5.3.2. Application encodeur MPEG4

La figure 5.1 montre la spécification fonctionnelle de l'application encodeur temps réel MPEG4 [Bon06]. Le module d'entrée (*In*) reçoit un flux de données correspondant à la vidéo non compressée. A ce niveau, chaque trame vidéo est décomposée en quatre sous-trames qui sont envoyées séparément à quatre modules d'encodage fonctionnant en parallèle (*Enc*). Les données traitées sont ensuite acheminées vers un encodeur à taille variable (*VLC*) qui est également responsable de la reconstitution de la trame entière avant de l'envoyer vers le module de sortie.

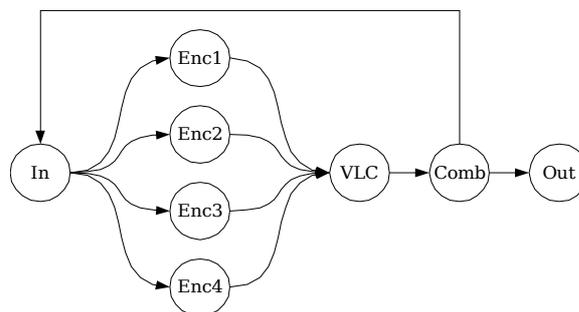


Figure 5.21: Spécification fonctionnelle de l'application DivX

Chaque module d'encodage est composé de quatre tâches comme illustré par la figure 5.3. La tâche d'entrée est responsable de la gestion du tampon d'entrée et de la mise en forme de la sous-

trame pour être traitée par la tâche d'encodage (*divx_encoder*). La tâche de sortie est responsable de l'envoi des données encodées dès qu'un macro bloc a été traité (ce n'est pas nécessaire d'attendre une sous-trame complète). Une tâche de gestion des erreurs contrôle le bon déroulement des différentes tâches (surcharge des tampons, etc.).

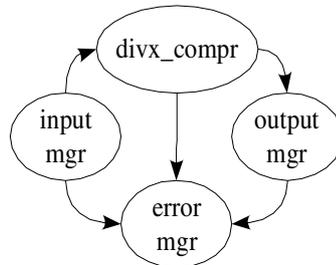


Figure 5.22: spécification d'un module encodeur

L'architecture cible est construite autour d'un réseau point-à-point à base de transferts DMA. Chaque module encodeur est implémenté sur un noeud logiciel correspondant à un sous-système CPU à base de processeur ARM7. Le module VLC est implémenté, quant à lui, sur un sous-système à base de ARM9. Les modules d'entrées et de sorties sont confiés à des blocs matériels.

La figure 5.8 montre le modèle de simulation au niveau HAL de l'application MPEG4. A ce niveau, les modules d'entrées et de sorties sont disponibles comme étant des IP matériels décrits en RTL. Le réseau de communication est modélisé au niveau TLM transaction. L'OS utilisée est obtenu par génération automatique en utilisant l'outil ASOG. Le modèle de simulation du sous-système CPU abstrait est choisi de telle façon qu'il fournit exactement la même API HAL que celle utilisée par (la partie supérieure) du système d'exploitation.

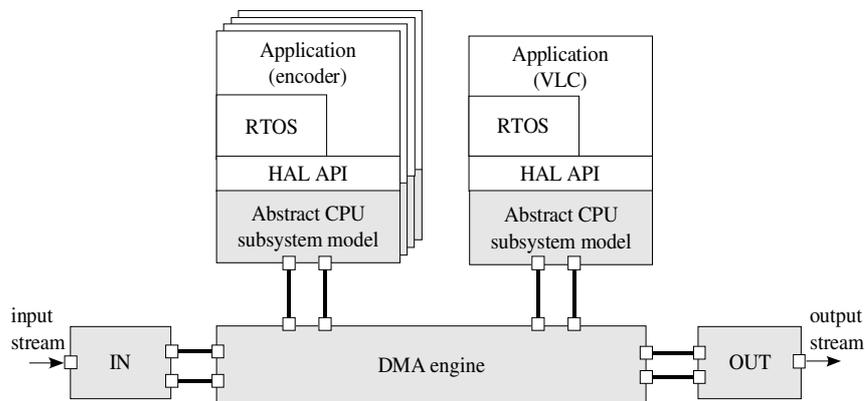


Figure 5.23: Modèle de simulation au niveau HAL de l'application MPEG4

Nous avons expérimenté deux variantes de modèles de sous-systèmes CPU. Dans la première (figure 5.9-a), la mémoire tampon utilisée pour l'entrée vidéo est prise comme partagée sur la mémoire locale. Ainsi, le contrôleur de communication se trouve dans l'obligation de passer par le bus système pour effectuer des transferts DMA vers cet espace tampon. Les conflits d'accès sur le bus système sont résolus par un arbitre implémentant une politique d'ordonnancement de type "tourniquet".

Dans la deuxième configuration (figure 5.9-b), une mémoire vidéo double-bancs dédiée est utilisée. Ceci permet le parallélisme calcul/communication. En particulier, les transferts DMA depuis/vers la mémoire vidéo peuvent s'effectuer en parallèle avec le traitement des données au niveau du processeur.

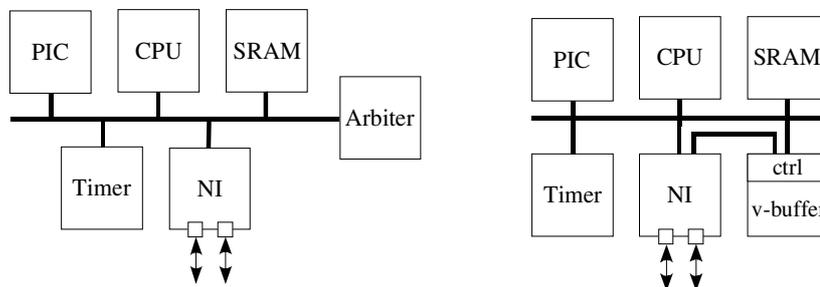


Figure 5.24: Modèle de simulation au niveau prototype virtuel de l'application MPEG4

Pour comparer les deux configurations d'architectures, nous avons effectué dans un premier temps la cosimulation au niveau prototype virtuel avec les modèles de simulation correspondant des sous-systèmes CPU abstraits. Dans un deuxième temps, nous avons effectué une simulation au niveau cycle après avoir raffiné les modèles abstraits des sous-systèmes CPU. Ceci correspond à l'utilisation de "vrais" composants dans l'architecture locale et la vraie couche HAL associée. Les modèles de processeurs sont constitués par des ISS précis au niveau cycle (*armulator*).

Les résultats obtenus sont résumés dans le tableau 2. Ceci correspond à une séquence vidéo d'une seconde. Par ailleurs les différents processeurs ARM sont cadencés à 60MHz.

		Temps simulé		erreur	Temps de simulation		Accélération
		Micro-architecture	prototype virtuel		Micro-architecture	prototype virtuel	
Confl	Enc	1.337s	1.419s	6%	8h	22s	x1300
	VLC	0.840s	0.933s	11%			

		Temps simulé		erreur	Temps de simulation		Accélération
		Micro-architecture	prototype virtuel		Micro-architecture	prototype virtuel	
Conf2	Enc	0.918s	0.951s	4%	8h	16s	x1800
	VLC	0.622s	0.682s	10%			

Tableau 2 : Résultats expérimentaux de l'application MPEG4

Les parties supérieure et inférieure du tableau correspondent respectivement à la première et deuxième configuration de l'architecture locale.

La première colonne montre le temps consommé par les différents CPU pour traiter une seconde de la séquence vidéo. Ce temps a été mesuré à la fois pour les niveaux prototype virtuel et micro-architecture.

La deuxième colonne calcule l'erreur introduite par le niveau HAL par rapport au niveau ISA pris comme référence. Les résultats obtenus montrent que :

- ◆ La première configuration de sous-système CPU ne permet pas de respecter la contrainte temps réel.
- ◆ L'erreur introduite par la simulation au niveau HAL est positive dans tous les cas. Ceci est dû au fait que les estimations de performances sont effectuées en considérant le pire cas vis-à-vis des incertitudes introduites par le pipeline du processeur ARM9.
- ◆ L'erreur correspondant au module VLC est relativement plus importante (utilisation de ARM9 comme processeur).

Les erreurs sont légèrement plus importantes dans le cas de la première variante d'architecture (sans mémoire double-bancs dédiée). Ceci est dû à l'imprécision supplémentaire introduite par le modèle de simulation simplifié de l'arbitre de bus.

Les deux dernières colonnes concernent la vitesse de simulation. Comparé à une simulation précis au cycle près au niveau ISA, il est clair qu'une simulation au niveau HAL permet d'atteindre une accélération importante (de l'ordre de trois ordres de grandeurs). La vitesse est légèrement inférieure dans le cas de la première configuration d'architecture à cause du surcoût introduit par l'ordonnancement des transactions sur le bus.

5.4. Conclusion

Ce chapitre a mis l'accent sur l'aspect utilisation de l'approche proposée dans le cadre d'un flot de conception partant d'une spécification initiale et arrivant à une implémentation finale. Ces différentes étapes ont été illustrées par sur un exemple académique mettant en relief les différents modèles utilisés et le rôle du concepteur à chaque étape du flot. L'outil d'annotation automatique du code logiciel et l'environnement de débogage et de simulation multi-niveaux MP-SIM ont été également présentés comme outils facilitant la tâche du concepteur.

La deuxième partie du chapitre a été consacrée à la présentation de quelques résultats expérimentaux obtenus sur deux exemples d'application. Ces résultats ont permis de montrer les avantages de l'utilisation des modèles de simulation proposés tant au niveau vitesse de simulation que précision temporelle.

Chapitre 6

Vers un modèle de raffinement de l'interface logicielle/matérielle

Sommaire

Chapitre 6	
Vers un modèle de raffinement de l'interface logicielle/matérielle.....	126
6.1.Raffinement de l'interface logicielle/matérielle : le modèle à base de composant/service.....	127
6.1.1.Raffinement de l'interface logicielle/matérielle : formulation du problème.....	127
6.1.2.Le modèle à base de composant/service.....	128
6.2.Flôt multi-niveaux de génération d'architectures MPSoC.....	137
6.2.1.Aperçu global du flôt de génération	137
6.2.2.Raffinement de l'architecture virtuelle.....	138
6.2.3.Raffinement du prototype virtuel.....	138
6.3.Conclusion.....	139

Dans les chapitres précédents, nous avons décrit deux modèles conceptuels de l'interface logicielle/matérielle correspondants à deux niveaux d'abstraction intermédiaires dans un flot de conception MPSoC. Ces modèles, associés à une sémantique d'exécution dans un environnement de simulation à événements discrets, permettent la validation et l'exploration rapide des architectures logicielles et matérielles avant même d'arriver à une implémentation complète du système. Le passage d'un niveau d'abstraction à un autre signifie que l'on passe d'un modèle abstrait de l'interface logicielle/matérielle à un modèle moins abstrait de celle-ci. Ceci peut être vu comme un processus de raffinement de l'interface logicielle/matérielle. Un tel processus de raffinement implique la génération d'architectures "réelles" aussi bien coté matériel que logiciel.

Dans ce chapitre, nous essayons d'aborder le problème de synthèse et de génération d'architecture sous cet angle de vue. Pour cela, nous commençons par formuler le problème de raffinement de l'interface logicielle/matérielle, le but étant d'identifier clairement les enjeux et les difficultés qui sont liés à ce processus. En vue de l'automatisation de la génération d'architecture, nous partons d'une approche de composition utilisée au sein du groupe basée sur la notion d'élément/service.

6.1. Raffinement de l'interface logicielle/matérielle : le modèle à base de composant/service

6.1.1. Raffinement de l'interface logicielle/matérielle : formulation du problème

La figure 6.1 illustre le processus générique de raffinement de l'interface logicielle/matérielle. Dans la figure, le raffinement s'effectue lors du passage du modèle abstrait de l'interface logicielle/matérielle (partie gauche de la figure) à un modèle moins abstrait de cette même interface (partie droite). En pratique, ceci correspond par exemple au passage du modèle virtuel d'architecture à un modèle d'architecture locale du sous-système CPU. Lors de ce passage, la génération d'architecture concerne aussi bien la couche système d'exploitation que les interfaces de communication entre le sous-système CPU et le réseau de communication. L'autre situation correspond au passage d'un modèle abstrait de l'architecture locale à un modèle concret de celle-ci. Dans ce cas, le processus de génération concerne l'architecture locale elle-même mais aussi la couche HAL qui permet d'accéder aux fonctionnalités de cette architecture.

Dans tous les cas, le processus de raffinement est composé de deux étapes importantes. La première est le choix de la "plate-forme" cible autour de laquelle l'interface abstraite de départ va être raffinée. Dans la figure, cette plate-forme est représentée par le modèle "moins abstrait" de l'interface logicielle/matérielle (partie droite de la figure). Remarquons que ce choix nécessite l'intervention du

concepteur et est loin d'être automatique.

La deuxième étape consiste à générer les architectures logicielle et matérielle résultant du choix de la plate-forme cible. Cette génération peut en effet être vue comme un processus d'adaptation entre des interfaces (ou encore des API) différentes. Le premier type d'interface est lié au modèle abstrait de départ (I_1^{SW} et I_1^{HW} pour les parties logicielle et matérielle respectivement). Le deuxième est lié au modèle d'arrivée de l'interface logicielle/matérielle -qu'on a appelé plate-forme- (I_2^{SW} et I_2^{HW}). Il s'agit ainsi de générer des "couches" d'adaptation entre I_1^{SW} et I_2^{SW} d'une part et I_1^{HW} et I_2^{HW} de l'autre.

Dans la suite du chapitre, nous essayons d'étudier la possibilité d'une génération automatique de ces couches d'adaptation architecturales à partir d'un modèle à base de composant/service.

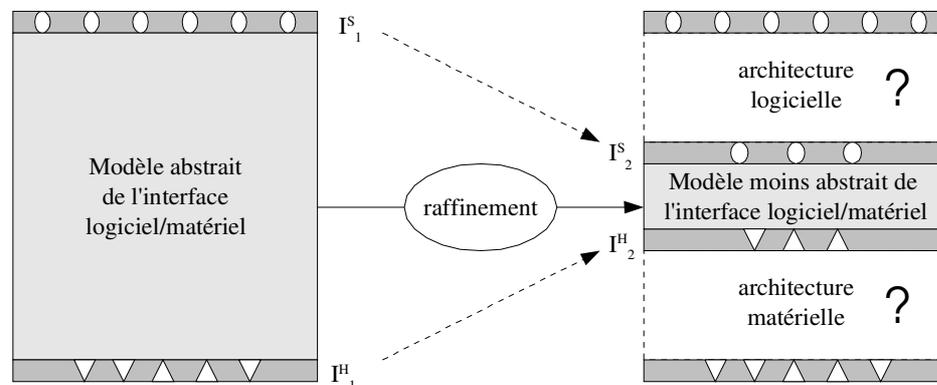


Figure 6.1 : raffinement de l'interface logicielle/matérielle

6.1.2. Le modèle à base de composant/service

L'idée derrière un modèle à base de composants et de services est de pouvoir décrire un système complexe comme étant un assemblage de composants élémentaires présentant des inter-dépendances statiques. Outre ses avantages classiques en termes de réutilisation et de conception structurée, un tel modèle présente l'avantage d'être générique dans le sens où il peut s'appliquer à différents domaines et à différents niveaux d'abstraction. Il constitue ainsi un cadre unificateur pour décrire un système aussi hétérogène que celui de l'interface logicielle/matérielle. Par ailleurs, le modèle se prête bien à l'automatisation moyennant la définition d'une structure particulière que nous appelons bibliothèque et que nous définissons formellement grâce à la notion de graphe de dépendance de services.

6.1.2.1. Pourquoi un modèle à base de composants ?

1) Modèle générique

Le concept de composant n'est certainement pas nouveau. Les gens du matériel ont toujours eu recours à la composition pour structurer leurs systèmes en entités plus élémentaires qu'ils utilisent et réutilisent à différents niveaux. Dans le domaine de l'informatique, ce concept rejoint l'idée même de programmation modulaire et peut être sous différents aspects allant de la simple procédure aux objets plus complexes tels que les acteurs. Aussi, la composition est-elle inhérente au processus même de conception. Cette généralité nous semble un atout essentiel pour permettre une description unifiée de l'architecture de l'interface logicielle/matérielle [Bou05], qui est --par nature même-- un système fortement hétérogène.

2) Modèle favorisant la réutilisation

Un modèle à base de composants se prête naturellement à la réutilisation via la notion d'interface. En effet, ce modèle favorise ce qu'on appelle la conception par contrat (*design by contract*) où différentes équipes peuvent travailler séparément sur des composants qui font partie d'un seul système, du moment que l'interface de ces composants est bien définie. Ceci permet de réduire considérablement le temps de conception global et permet la réutilisation des mêmes composants dans d'autres applications, une réutilisation d'autant plus significative que la granularité des composants est fine.

3) Modèle se prêtant à l'automatisation

Par automatisation, nous désignons le processus automatique de composition qui permet l'assemblage d'un système complexe, soumis à un certain nombre de contraintes, étant donné un ensemble de composants disponibles. Le modèle à base de composant/service se prête bien à une telle composition automatique en utilisant le concept de graphe de dépendance de services (cf section 6.1.2.3).

Dans ses travaux de thèse [Gau03], L. Gauthier utilise cette mécanique de composition pour automatiser la génération de la couche système d'exploitation d'une manière spécifique à l'application cible. La qualité du système généré est directement liée à la granularité des composants, une granularité qui peut être choisie arbitrairement fine.

4) Limites du modèle à base de composants :

Les limites d'un modèle à base de composants peuvent être résumées essentiellement en deux points. Le premier concerne la qualité du système obtenu par rapport à un modèle dit « monolithique » dans lequel, le système est entièrement construit à partir de zéro pour « coller »

parfaitement au problème en question. Intuitivement, par rapport à ce modèle, le modèle à base de composants introduit une certaine « inefficacité » traduite par le sur-dimensionnement du système obtenu, un phénomène essentiellement lié à la redondance et/ou la non-utilisation de certaines fonctions à travers un ensemble de composants. Pour remédier à cet inconvénient, il convient de choisir une granularité assez fine des composants pour mieux s'adapter aux besoins du problème à résoudre. Par ailleurs, regrouper les composants selon leur adéquation aux domaines traités -conformément à la notion de plateforme- permet aussi de mieux cibler l'application en question.

La seconde limitation concerne la nature des dépendances entre composants au sein du modèle. En effet, dans notre cas, nous exigeons que ces dépendances soient statiques, c'est à dire, connues à l'avance durant la phase de conception. Ceci exclut une dépendance de type dynamique, où un composant aurait besoin d'un service fourni par un autre composant d'une manière imprévue au cours du fonctionnement du système. Si cette condition peut paraître assez naturelle pour le matériel¹⁷, elle l'est beaucoup moins pour le logiciel dans le cas général. En effet, dans les systèmes informatiques dits « ouverts », une telle situation est assez courante. Heureusement, dans le cas des systèmes embarqués, on a plutôt affaire à des systèmes majoritairement statiques où l'application et l'architecture sont connues à l'avance. Dans ces systèmes le recours à des solutions dynamiques -- quand la solution statique suffit-- induit des pénalités importantes en termes de performances.

6.1.2.2. *Notion de composant/service*

1) *Définition d'un composant élémentaire*

Un composant élémentaire est une entité atomique réalisant une certaine tâche. D'un point de vue système, un composant est entièrement défini par son *interface* qui est un ensemble de *références* vers des *services*. Un service est une fonctionnalité élémentaire qu'un composant peut fournir ou demander dans le cadre de l'accomplissement de sa tâche. Il est important de noter que la notion de service est indépendante de celle de composant. En particulier, chaque service est identifié d'une manière unique dans le contexte global considéré (que nous désignerons par bibliothèque). Dans ce contexte, l'ensemble fini des services est noté S , alors que l'ensemble fini des composants est noté C .

Une référence sur un service s de S est un élément de $S_f \cup S_r$ avec $S_f \subseteq (S \times C)$ et $S_r \subseteq (C \times S)$. Par abus de langage, une référence $s_f = (s, c) \in S_f$ est appelée service fourni par le composant c alors qu'une référence $s_r = (c, s) \in S_r$ est appelée service requis par le composant c .

Pour un composant c , nous notons par :

- $I_f(c) = \{s_f^1, s_f^2, \dots, s_f^n\}$ où $s_f^i = (s^i, c) \in S_f \quad \forall i=1..n$ l'ensemble des services fournis par

¹⁷ Si on exclut le cas du matériel dynamiquement reconfigurable

le composant c ou encore l'interface fournie par le composant c .

- $I_r(c) = \{s_r^1, s_r^2, \dots, s_r^m\}$ où $s_r^i = (c, s^i) \in S_r \quad \forall i=1..m$ l'ensemble des services requis par le composant c ou encore l'interface requise par le composant c .

- $I(c) = I_f(c) \cup I_r(c)$ l'interface du composant c .

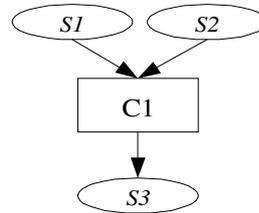


Figure 6.2 : modèle d'un composant élémentaire

La figure 6.2 illustre graphiquement la notion de composant/service. Ici, le composant c_1 fournit les services s_1 et s_2 et requière le service s_3 . Notons que schématiquement, un service fourni est dénoté par une flèche rentrante vers le composant alors qu'un service requis est dénoté par une flèche sortante du composant.

2) Composant hiérarchique

Un composant hiérarchique est un composant non élémentaire, c'est à dire, composé de plusieurs autres composants. En ce sens, il constitue en lui-même un sous-système et a donc déjà fait l'objet d'une étape de conception (cf la section 6.1.2.4). D'un point de vue système, ce qui nous intéresse, c'est son interface qui est dans ce cas définie à partir des interfaces des sous-composants qui le constituent.

$$c_H = \{c^1, c^2, \dots, c^k\} \text{ où } c^i \in C \quad \forall i=1..k$$

$$I_f(c_H) \subset \left(\bigcup_{i=1..k} I_f(c^i) \right)$$

$$I_r(c_H) \subset \left(\bigcup_{i=1..k} I_r(c^i) \right)$$

3) Exemples de composants

Dans ce paragraphe, nous introduisons quelques exemples concrets de composants permettant d'illustrer les concepts abstraits développés ci-dessus. Plus particulièrement, deux types de composants seront décrits appartenant à deux domaines différents: matériel et logiciel.

- ◆ Exemple de composant matériel : DMA

Un DMA est un composant matériel permettant le transfert direct entre périphérique et mémoire sans passer par le processeur. Pour ce faire, il dispose généralement de deux ports d'accès maître et esclave sur le bus système ainsi que d'un port d'interruption pour alerter le processeur à la fin d'un transfert. Les services fournis par le DMA peuvent être ramenés (au niveau transactionnel) aux services de lecture/écriture sur le port esclave. Ces services seront typiquement utilisés par le processeur pour configurer les transferts DMA.

$$I_f(DMA) = \{bus_slave_read, bus_slave_write\}$$

Les services requis par le composant DMA sont d'une part les services de lecture/écriture requis par le port maître pour effectuer le transfert des données, et de l'autre part le service d'interruption qui permet de déclencher le traitement nécessaire à la fin d'un transfert.

$$I_r(DMA) = \{bus_master_read, bus_master_write, interrupt_trigger\}$$

◆ Exemple de composant logiciel : *Mutex*

En logiciel, un *mutex* est une primitive de synchronisation utilisée pour coordonner deux processus concurrents en effectuant une exclusion mutuelle avant l'accès à une ressource partagée. Il existe différentes implémentations d'un *mutex*. Par exemple, un premier type d'implémentation utilise des opérations atomiques du genre « *test-and-set* » alors qu'un deuxième type peut compter sur le masquage des interruptions.

Dans les deux cas, les services fournis peuvent être les mêmes, à savoir, les services d'initialisation, de réservation et de libération du *mutex*.

$$I_f(Mutex_{1,2}) = \{mutex_init, mutex_lock, mutex_unlock\}$$

Les services requis, quant à eux, diffèrent selon le type de composant et peuvent inclure soit des services d'accès atomiques à la mémoire, soit des services de contrôle des interruptions (à côté des autres services liés à l'ordonnancement des tâches)

$$I_r(Mutex_1) = \{atomic_test_and_set, schedule\}$$

$$I_r(Mutex_2) = \{enable_interrupt, disable_interrupt, schedule\}$$

6.1.2.3. Graphe de dépendance de services (SDG)

La notion de graphe de dépendance de services (SDG de *Service Dependency Graph* en anglais) est introduite pour rendre compte des relations entre composants appartenant à un contexte de conception donné.

1) Définition d'un graphe de dépendance de services

Formellement, un graphe de dépendance de services SDG est définie comme suit :

$SDG = \langle C, S, S_r, S_f \rangle$ où

C : ensemble fini de composants

S : ensemble fini de services

$S_r \subseteq (C \times S)$: ensemble d'arcs correspondant aux services requis

$S_f \subseteq (S \times C)$: ensemble d'arcs correspondant aux services fournis

Par ailleurs, on note par :

$D = (S_r \cup S_f)$ l'ensemble de toutes les dépendences

$C_f(s) = \{c \in C \text{ tel que } \exists s_f \in S_f \wedge s_f = (s, c)\}$ l'ensemble des composants offrant le service s

$C_r(s) = \{c \in C \text{ tel que } \exists s_r \in S_r \wedge s_r = (c, s)\}$ l'ensemble des composants requérant le service s

$S_f(c) = \{s \in S \text{ tel que } \exists s_f \in S_f \wedge s_f = (s, c)\}$ l'ensemble des services fournis par le composant c

$S_r(c) = \{s \in S \text{ tel que } \exists s_r \in S_r \wedge s_r = (c, s)\}$ l'ensemble des services requis par le composant c

Dans la suite, on fera aussi l'hypothèse que chaque composant offre au moins un service :

$$\forall c \in C, \text{card}(S_f(c)) > 0$$

- Définition : service résolu

un service $s \in S$ est dit résolu si et seulement $\text{card}(C_f(s)) \leq 1$. Dans le cas contraire le service est dit non résolu.

- Définition : graphe résolu

Un graphe $SDG = \langle C, S, S_r, S_f \rangle$ est dit résolu si et seulement si tous ces services sont résolus.

Étant donnée un graphe $SDG = \langle C, S, S_r, S_f \rangle$, on a les définitions suivantes :

- Définition : service requis par le graphe

Un service $s \in S$ est un service requis par le graphe si et seulement si $\text{card}(C_r(s)) = 0$

Autrement dit, le service n'est fourni par aucun composant du graphe.

- Définition : service fourni par le graphe

Un service $s \in S$ est un service fourni par le graphe si et seulement si $\text{card}(S_r(c)) = 0$

Autrement dit, le service n'est requis par aucun composant du graphe.

- L'interface requise $I_R(SDG)$ est l'ensemble des services requis par le graphe.

- L'interface fournie $I_F(SDG)$ est l'ensemble des services fournis par le graphe.

La figure 6.3 montre un exemple de SDG. Notons que le graphe accepte des dépendances cycliques, comme c'est le cas entre les composants c1 et c2. Par ailleurs, le service s3 n'étant pas fourni par aucun composant dans le graphe, il représente une dépendance externe.

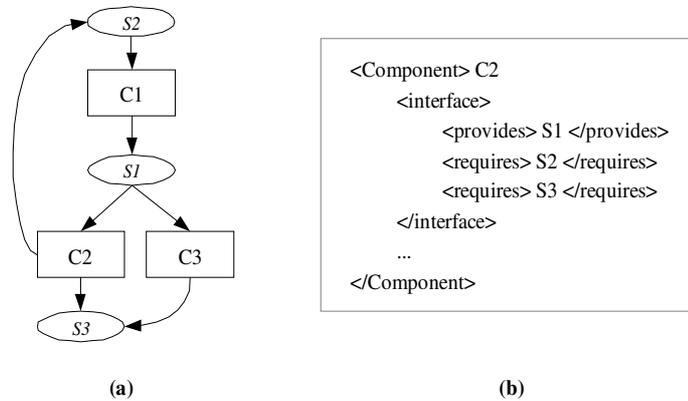


Figure 6.3: Exemple de graphe de dépendance de services

Dans cet exemple, le même service s1 est fourni par deux composants différents (c2 et c3) et est donc non résolu. Lors de l'étape d'instanciation du système (cf section suivante), il faudra choisir un parmi ces deux composants. On parle alors de la résolution des dépendances multiples. Cette résolution peut être simplement réalisée connaissant les autres services effectivement disponibles. Dans le cas où la donnée de ces contraintes ne suffit pas à résoudre la dépendance multiple, un choix de conception est alors nécessaire. Ce choix peut être fait explicitement par le concepteur, ou via des algorithmes/heuristiques en introduisant des paramètres non fonctionnels (par exemple basés sur le coût, la consommation, etc.).

2) Notion de bibliothèque de composants

Le graphe de dépendance de services constitue un formalisme adéquat pour exprimer les relations statiques entre composants. Généralement, on regroupe les composants --selon le domaine auquel ils appartiennent-- dans des bibliothèques. Dans notre terminologie, une bibliothèque est simplement un SDG spécifique à un domaine particulier auquel s'associe un ensemble de mécanismes et d'outils permettant l'exploitation et la maintenance de la bibliothèque.

3) La bibliothèque comme graphe maximal

L'intérêt d'une bibliothèque de composants est bien entendu de pouvoir l'utiliser et la réutiliser pour la conception de plusieurs systèmes. Ces systèmes peuvent être constitués d'un nombre variable de composants. Ainsi, une bibliothèque peut être vue comme l'ensemble maximal de tous les

composants qui peuvent intervenir dans la conception d'un système donné, ce qui ne correspond pas, rappelons-le, à une instance maximale puisqu'il s'agit d'un graphe non résolu.

Durant le processus d'instanciation (ou encore de conception du système) seule une partie du graphe maximale sera utilisée pour subvenir aux besoins du système en question. L'architecture ainsi obtenue sera alors spécifique au problème considéré, ce qui est d'autant plus vrai que la granularité des composants de la bibliothèque est plus fine.

6.1.2.4. *Modèle de l'architecture: résolution du graphe de dépendance*

Dans notre modèle, une architecture est définie en spécifiant les services requis et fournis au niveau des deux interfaces délimitant l'architecture. En utilisant la notion de bibliothèque définie plus haut, le but est de trouver les composants adéquats de cette bibliothèque qui permettent de satisfaire exactement les services requis tout en utilisant les services fournis. Ceci revient à trouver un sous graphe **résolu** du SDG maximal de la bibliothèque en question qui répond à ces critères.

La figure 6.4 montre un exemple d'un tel graphe avec $\{S_1, S_2, S_3, S_4\}$ comme services requis (par l'application) et $\{S'_1, S'_2, S'_3\}$ comme services fournis (par la plate-forme). Le résolution du graphe de dépendance d'une bibliothèque hypothétique génère dans le cas de la figure, l'architecture composée des éléments $\{E_1, E_2, E_3, E_4, E_5, E_6\}$. Ces composants mettent en jeu également des services internes qui sont $\{S^i_1, S^i_2, S^i_3, S^i_4\}$.

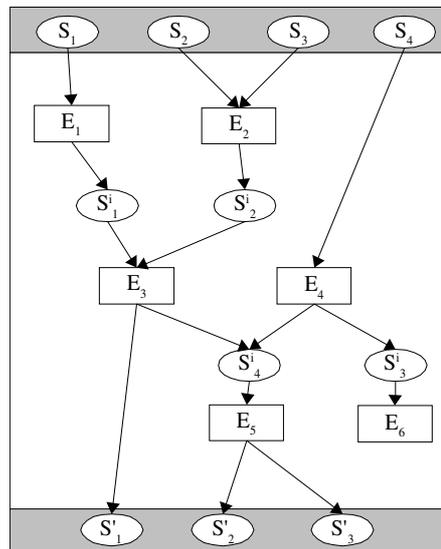


Figure 6.4: Exemple d'une architecture à base de composant/service

Étant donnée une bibliothèque définie par un graphe de dépendance de services (*SDG*) et étant

donnés deux sous-ensembles $i_R \subset I_R(SDG)$ et $i_F \subset I_F(SDG)$, on note par :

SDG^* la fermeture transitive de SDG , $SDG^* = \langle C, S, T \rangle$

et $T(s) = \{c \in C \text{ tel que } (s, c) \in T\} \cup \{s' \in S \text{ tel que } (s, s') \in T\}$

La résolution du graphe SDG par rapport aux deux sous-ensembles i_R et i_F est un sous-graphe $ARCH$ vérifiant :

$ARCH = \langle C', S', S_r', S_f' \rangle$ où

(1) $C' \subseteq (C \cap T(i_F))$, $S' \subseteq (S \cap T(i_F))$, $S_r' \subseteq S_r$, $S_f' \subseteq S_f$

(2) $\forall s \in S'$, $\text{card}(C_f(s)) \leq 1$

(3) $i_F = I_F(ARCH)$

(4) $i_R \supseteq I_R(ARCH)$

(5) Il n'existe pas $ARCH'$ vérifiant (1) (2) (3) et (4) tel que $ARCH \subset ARCH'$

Autrement dit, $ARCH$ est un sous-graphe résolu du graphe SDG , maximal dans SDG^* , dont les interfaces requise et fournie correspondent respectivement aux deux sous-ensembles i_R et i_F .

◆ Existence de $ARCH$:

Il est facile de démontrer qu'une condition nécessaire et suffisante pour l'existence de $ARCH$ est

que $(T(i_F) \cap I_R(SDG)) \subseteq i_R$ où $T(i_F) = \bigcup_{s \in i_F} T(s)$

En fait, supposons $(T(i_F) \cap I_R(SDG)) \subseteq i_R$. Dans ce cas en prenant $ARCH$ un sous graphe résolu maximal de $T(i_F)$, on vérifie les conditions (3) et (4).

Par ailleurs, supposons qu'il existe $ARCH$ vérifiant les conditions (1) à (5). Il est facile de démontrer que $(T(i_F) \cap I_R(SDG)) \subseteq I_R(ARCH)$. Comme $I_R(ARCH) \subseteq i_R$ (condition (5)), il vient alors que $(T(i_F) \cap I_R(SDG)) \subseteq i_R$.

Ainsi l'existence de l'architecture permettant l'adaptation entre les deux interfaces n'est pas toujours garantie. La condition établie plus haut signifie que les deux interfaces doivent être "compatibles" ou encore cohérentes pour assurer cette existence.

◆ Unicité de $ARCH$:

Dans le cas général $ARCH$ n'est pas unique. Pour s'en apercevoir, il suffit de considérer un même service qui est fourni par deux composants distincts pouvant être sélectionné indifféremment (en supposant par exemple que ces deux composants ne dépendent pas eux mêmes d'autres services qui peuvent imposer la sélection de l'un ou de l'autre).

6.2. Flot multi-niveaux de génération d'architectures MPSoC

Cette section donne un aperçu sur ce que serait un flot de génération multi-niveaux d'architectures en se basant sur l'approche élément/service décrite plus haut. Les différents niveaux concernés par la génération sont ceux utilisés dans le quatrième chapitre pour la validation et l'exploration multi-niveaux de l'interface logicielle/matérielle.

6.2.1. Aperçu global du flot de génération

La figure 6.5 montre les différentes étapes du flot de génération proposé. Dans ce flot, l'architecture virtuelle est prise comme point de départ pour la génération, et non pas la spécification fonctionnelle. Cela signifie, entre autre, que nous nous intéressons pas ici au problème de l'automatisation du partitionnement de la transposition application/architecture.

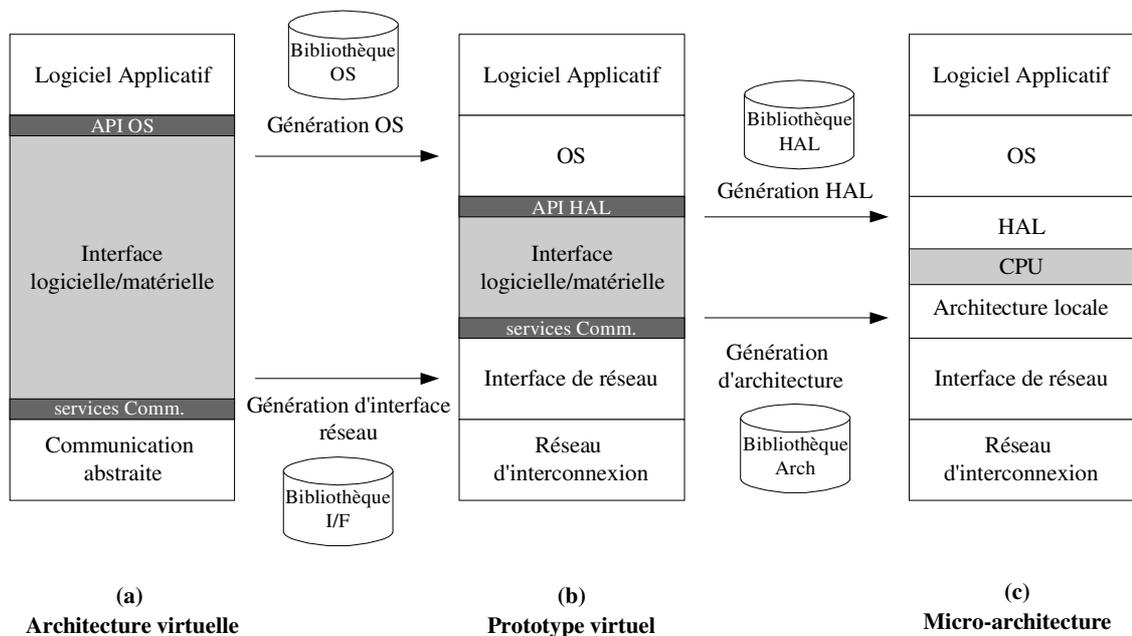


Figure 6.5: Flot de génération d'architectures MPSoC à différents niveaux d'abstraction

Le passage d'un niveau de modélisation à un autre est toujours accompagné de deux types de processus complémentaires :

- ◆ Le processus de raffinement.

Ce processus signifie le passage d'un modèle abstrait du système à un modèle moins abstrait. Dans le cas général, ce passage nécessite la mise en oeuvre d'un certain nombre de décisions architecturales impliquant l'intervention du concepteur (non objet à l'automatisation)

- ◆ Le processus de génération d'architecture.

Il s'agit d'une conséquence du raffinement du système qui peut être vue comme l'étape d'adaptation entre le modèle initial du système et le modèle raffiné. Si les deux modèles sont correctement spécifiés, on peut espérer une automatisation de ce processus.

L'intérêt d'un flot de génération qui combine différents niveaux d'abstraction avec la génération automatique d'architecture est de faciliter la tâche du concepteur en décomposant le processus de raffinement. A chaque niveau d'abstraction, le concepteur peut en effet explorer rapidement différents paramètres liés au processus de raffinement. Cette exploration, qui implique souvent plusieurs itérations entre l'étape de départ et l'étape d'arrivée, est rendue efficace en réduisant le gap entre les deux étapes de conception et en automatisant la génération de l'adaptation entre les modèles manipulés à chaque étape.

6.2.2. Raffinement de l'architecture virtuelle

Cette étape désigne le passage du modèle d'architecture virtuelle vers le modèle du prototype virtuel. Le raffinement concerne particulièrement l'interface logicielle/matérielle et la communication globale qui passe d'un niveau abstrait (logique) à un niveau architectural (physique). Ce passage doit s'accompagner par:

- ◆ la génération d'une couche d'adaptation logicielle entre les services requis par le logiciel applicatif (partie de l'API OS) et les services fournis par l'interface logicielle/matérielle du prototype virtuel (partie de l'API HAL). Cette couche correspond en fait au système d'exploitation, qui sera alors spécifique à l'application et à l'architecture cible.
- ◆ la génération d'une couche d'adaptation matérielle rapprochant les services de communication requis par le modèle de l'interface logicielle/matérielle et ceux fournis par le réseau de communication global.

6.2.3. Raffinement du prototype virtuel

Cette étape correspond au passage du modèle d'architecture virtuelle vers le modèle du prototype virtuel. Le raffinement concerne l'interface logicielle/matérielle qui passe d'un modèle abstrait de sous système CPU à une architecture locale détaillée centrée autour d'un (ou plusieurs) processeur(s) cible(s) . Ce passage doit s'accompagner par:

- ◆ la génération d'une couche d'adaptation logicielle entre les services requis par le système d'exploitation (sous-ensemble de l'API HAL) et les services fournis par le processeur (ISA). Il

s'agit d'une couche HAL spécifique à l'architecture et au système d'exploitation généré.

- ◆ la génération d'une architecture locale permettant d'adapter l'interface physique du processeur aux services de communication locale.

6.3. Conclusion

Dans ce chapitre, le problème de génération d'architecture a été abordé sous la même perspective développé tout au long de cette thèse qui est la notion d'interface logicielle/matérielle . Ce problème fut ramené au processus de raffinement de l'interface logicielle/matérielle, un processus qui peut avantageusement bénéficier de l'automatisation moyennant une approche de composition.

Bien que la solution proposée reste incomplète, une première formalisation de cette approche a été néanmoins proposée, suggérant plusieurs pistes pour des développements ultérieurs.

Chapitre 7

Conclusion et perspectives

Pour remédier aux problèmes liés à l'intégration tardive des architectures logicielle et matérielle d'un système MPSoC, un modèle unifié permettant la représentation conjointe à différents niveaux d'abstraction de ces deux types d'architectures a été présenté le long de cette thèse. Ce modèle a comme objectif ultime de remédier à la discontinuité qui caractérise les flots et les modèles de représentation existants. Ceci passe par (1) une validation graduelle des architectures MPSoC tout en permettant l'évaluation, à chaque niveau d'abstraction, des performances qui en découlent, (2) la conception conjointe et parallèle du logiciel embarqué et de l'architecture matérielle sous-jacente.

Nous nous sommes intéressés d'abord au problème de l'abstraction de l'interface logicielle/matérielle. Ce travail d'abstraction nous a permis d'identifier les éléments clés dans une architecture logicielle/matérielle à un niveau d'abstraction donné, et de simplifier la représentation de cette architecture en excluant les éléments jugés non importants à ce niveau. Ainsi, la première tâche était d'identifier les niveaux d'abstraction pertinents qui correspondent naturellement à des « paliers » dans un flot de conception MPSoC, et de définir, pour chaque niveau, le modèle conceptuel de l'interface logicielle/matérielle correspondant. Un tel modèle permet d'une part de refléter explicitement les caractéristiques architecturales importantes au niveau d'abstraction considéré et de l'autre part, cacher ou encore faire abstraction des détails qui ne le sont pas à ce niveau.

Pour être exploitables dans un contexte de validation et d'exploration d'architecture, les modèles conceptuels de l'interface logicielle/matérielle sont associés à des modèles de simulation. Ces derniers définissent la sémantique d'exécution de l'interface logicielle/matérielle dans un contexte de cosimulation globale impliquant aussi bien les parties logicielles que matérielles raffinées au niveau d'abstraction considéré. Comme contexte de cosimulation globale, nous considérons l'environnement SystemC qui offre des atouts importants en terme de flexibilité et de performance. Les modèles de simulation proposés de l'interface logicielle/matérielle s'intègrent dans cet environnement, sans avoir besoin de modifier le moteur de simulation interne de celui-ci. Ceci conserve la sémantique habituelle de SystemC et favorise la réutilisation des composants déjà existants.

La méthodologie proposée a été appuyée par une démarche expérimentale visant à démontrer l'intérêt d'une telle approche d'un point de vue pratique. Ainsi, la dernière partie du document a été réservée à l'analyse de quelques résultats expérimentaux obtenus sur des exemples réels d'applications. Plus particulièrement, nous nous sommes intéressés aux problèmes de précision et de performance (vitesse de simulation) résultant de l'utilisation des modèles proposés de l'interface logicielle/matérielle à chaque niveau d'abstraction. Il faut mentionner toutefois que ces expérimentations ont été entreprises au fur et à mesure de l'avancement des travaux, et sont donc présentées comme preuve de faisabilité et non pas comme études de cas exhaustifs.

Nous avons également abordé le problème de l'intégration des différents modèles proposés dans un flot de conception automatique de systèmes MPSoC, tel que celui proposé par ROSES. Ce problème a été placé dans une perspective unifiée de raffinement de l'interface logicielle/matérielle. Une formalisation de ce raffinement se basant sur une approche de composition faisant appel à la notion d'élément/service a été élaborée. Selon ce formalisme, un processus de raffinement ou encore de génération d'architecture, peut être ramené à la composition d'éléments de base, fournis par une bibliothèque, selon un schéma bien déterminé décrivant les dépendances en termes de services fournis et requis. L'intérêt majeur de ce formalisme, en plus de l'automatisation de la génération, est le fait qu'il peut s'appliquer aussi bien pour générer des architectures matérielles que logicielles. Il permet ainsi une flexibilité importante au niveau du raffinement de l'interface logicielle/matérielle et favorise l'exploration de l'espace des solutions architecturales envisageables à ce niveau.

Cependant, cette formalisation de la mécanique de génération d'architectures reste incomplète et ne constitue qu'un premier apport à ce sujet. En particulier, rien n'est dit sur la manière avec laquelle les éléments de la bibliothèque sont conçus. Le choix et la nature de ces éléments est en fait d'une importance capitale sur la qualité et la faisabilité même du processus de génération d'architecture. Ceci est d'autant plus subtile qu'on désire se placer dans un contexte où on peut bénéficier d'un degré de flexibilité important concernant le découpage des fonctionnalités de l'interface logicielle/matérielle.

Pour relever ces défis, un modèle de représentation des composants de la bibliothèque est nécessaire. Ce modèle permettra de relier le processus de raffinement (composition) au modèle conceptuel initial de l'interface logicielle/matérielle et de rendre ainsi compte d'un quelconque découpage logiciel/matériel au sein de cette interface. Par ailleurs, pour être complet, ce modèle doit tenir compte de l'aspect performance en intégrant la notion de qualité de service. Ceci permet de garantir une certaine qualité de service au niveau de l'architecture générée par rapport aux contraintes de performances exprimées dans le modèle initial de l'interface logicielle/matérielle.

Une solution intéressante serait de considérer un tel modèle de raffinement comme une manière de définir une sémantique de synthèse pour les modèles conceptuels de l'interface logicielle/matérielle proposée par cette thèse. Ainsi, en plus de la sémantique d'exécution que nous avons décrit, le modèle abstrait de l'interface logicielle/matérielle se voit associé une sémantique de synthèse permettant d'inférer une sémantique appropriée aux éléments de la bibliothèque de génération.

Bibliographie :

- [Ara01] L. Arantes, D. Poitrenaud, P. Sens, B. Folliot, "The Barrier-Lock Clock: A Scalable Synchronisation-oriented logical clock" *Parallel Processing Letters*, Vol. 11(1):65-76, 2001
- [Arm03] ARM. "AMBA AXI Protocol Specification", Juin 2003.
- [Bac06] I. Bacivarov, A. Bouchhima, S. Yoo, A.A. Jerraya, "ChronoSym – a New Approach for Fast and Accurate SoC Cosimulation". Dans *International Journal of Embedded Systems*, Inderscience Publishers. 2006.
- [Bar94] 4. Barros, E. and A. Sampaio. "Towards Provably Correct Hardware/Software Partitioning Using Occam". Dans *Third International Workshop on Hardware/Software Codesign*. Grenoble, France : IEEE Computer Society Press, pp210-217. 1994.
- [Ben01] L. Benini and G. De Micheli. "Powering networks on chips". Dans les actes de *ISSS'01*, 2001.
- [Ben02] L. Benini and G. De Micheli. "Networks on chips: A new SoC paradigm". Dans *IEEE Computer*, 2002.
- [Ben03] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, M. Poncino. "SystemC cosimulation and emulation of multiprocessor SoC designs". Dans *IEEE Computer*, 36/4, pp53-59, Avril 2003.
- [Bju01] P. Bjur us, A. Jantsch, "Performance analysis with confidence intervals for embedded software processes", Dans les actes de *the International Symposium on System Synthesis, Montr al, Canada*. pp45-50. Septembre 2001.
- [Bon06] M. Bonaciu, A. Bouchhima, M.W. Youssef, X. Chen, W. Cesario, A.A. Jerraya, "High-Level Architecture Exploration for MPEG4 Encoder with Custom Parameters". Dans les actes de *11th Asia and South Pacific Design Automation Conference ASP-DAC'06, Yokohama City, Japan*, Janvier 2006.
- [Bou04] A. Bouchhima, S. Yoo, A.A. Jerraya, "Fast and Accurate Timed Execution of High Level Embedded Software Using HW/SW Interface Simulation Model", Dans les actes de *9th Asia and South Pacific Design Automation Conference ASP-DAC'04, Yokohama, Japan*, Janvier 2004.
- [Bou05'] A. Bouchhima, X. Chen, F. Petrot, W. Cesario, A.A. Jerraya, "A Unified HW/SW Interface Model to Remove Discontinuities between HW and SW Design", Dans les actes de *EMSOFT 2005, Jersey City NJ, USA*, Septembre 2005.
- [Bou05] A. Bouchhima, I. Bacivarov, W. Youssef, M. Bonaciu, A.A. Jerraya, "Using Abstract CPU Subsystem Simulation Model for High Level HW/SW Architecture Exploration". Dans les actes de

10th Asia and South Pacific Design Automation Conference ASP-DAC'05, Shanghai, China. Janvier 2005.

[Bra01] C. Brandolese, W. Fornaciari, F. Salice, D. Cciuto, "Source-level execution time estimation of C programs". Dans les actes de *the International Symposium on HW/SW CoDesign, Copenhagen, Denmark, IEEE, Los Alamitos, CA.* pp98-103. Avril 2001.

[Bru00] J-Y. Brunel, W.M. Kruijtzter, H.J.H.N. Kenter, F. Pétrot, L. Pasquier, E.A. de Kock, W.J.M. Smits, "COSY Communication IP's," Dans les actes de *Design Automation Conference DAC'00*, Juin 2000.

[Cac00] D. Cachera, P. Quinton, S. Rajopadhye, T. Risset, "Proving Properties of Multidimensional Recurrences with Application to Regular Parallel Algorithms" *Rapport de Recherche Irisa, No1362*, Novembre 2000.

[Cal03] M. Caldari, M. Conti, M. Coppola, S. Curaba, L. Pieralisi, C. Turchetti "Transaction-Level Models for AMBA Bus Architecture Using SystemC 2.0". Dans les actes de *Design, Automation & Test in Europe DATE'03*. 2003.

[CarbonK] CarbonKernel. Disponible sur <http://savannah.nongnu.org/projects/carbonkernel/>

[Ces02] W. Cesario, et. al., "Component-Based Design Approach for Multicore SoCs", Dans les actes de *Design Automation Conference DAC'02*, 2002.

[Ces04] W. Cesario, Y. Paviot, L. Gauthier, D. Lyonnard, G. Nicolescu, S. Yoo, A.A. Jerraya, "Object-based Hardware/Software Component Interconnection Model for Interface Design in System-on-a-chip Circuits". Dans *The Journal of Systems and Software, SI: Rapid System Prototyping, Ed. by L.M. Wills, F. Kordon and Luqi, 70/3 pp. 229-244, Elsevier Science*, 2004.

[Cha79] K. Chandy, J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Transactions on Software Engineering*, pp. 440--452, Septembre 1979.

[Cha92] A. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-Power CMOS Digital Design," Dans *IEEE Journal of Solid-State Circuits*, 27/4, pp. 473-- 483, Avril 1992.

[Clo02] A. Clouard, "Functional and Timed Transactional-Level SoC Models in SystemC", *5th European SystemC Users Group Meeting*, Mars 2002

[Dal01] W. J. Dally and B. Towles. "Route packets, not wires: On-chip interconnection networks". Dans les actes de *Design Automation Conference DAC'01*, 2001.

[Des00] D. Desmet, D. Verkest and H. De man. "Operating system based SW generation for system-on-chip". Dans les actes de *Design Automation Conference DAC'00*, Juin 2000.

[Dou01] D.A. Stuart, M. Brockmeyer, A. K. Mok, F. Jahanian. "Simulation Verification: Biting at the State Explosion Problem," Dans *IEEE Transactions on Software Engineering*, 27/7, pp. 599-617 Juillet 2001.

[Dup95] F. Dupont de Dinechin, P. Quinton, T. Risset, "Structuration of the Alpha Langage"

International Conference on Massively Parallel Programming Models, Berlin, Octobre 1995.

[Dyl05] Dylan McGrath. "Unified Modeling Language gaining traction for SoC design", Dans *EE Times*. Avril 2005.

[Ecos] eCos. Disponible sur <http://sources.redhat.com/ecos/>

[Fau95] A. Fauth and J. Van Praet and M. Freericks. "Describing Instruction Set Processors Using nML". Dans les actes de *the European Design and Test Conference (ED&TC)*, Mars 1995.

[Gaj03] D. Gajski et al, "Transaction based design : Another Buzzword or the solution to a Design Problem". Dans les actes de *Design, Automation & Test in Europe, Munich, Germany*, Mars 2003.

[Gau01] L. Gauthier, S. Yoo, A.A. Jerraya, "Automatic Targeting of Embedded Systems Software with Application Specific Operating Systems Generation". Dans les actes de *5th International Workshop on Software and Compilers for Embedded Systems SCOPES, St. Goar, Germany*. Mars 2001.

[Lyo01] D. Lyonnard, S. Yoo, A. Baghdadi, A. A. Jerraya, "Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip". Dans les actes de *Design Automation Conference DAC'01, Las Vegas, USA*. Juin 2001.

[Ger03] A. Gerstlauer, H. Yu, D.D. Gajski, "RTOS Modeling for System-Level Design," Dans les actes de *Design, Automation & Test in Europe, Munich, Germany*, Mars 2003.

[Giu01] P. Giusto, G. Martin, E. Harcourt, "Reliable estimation of execution time of embedded software". Dans les actes de *Design, Automation and Test Conference, Munich, Germany*, Mars 2001.

[Gon02] M. Gonzales and J Madsen. "Abstract rtos modelling in systemc". Dans les actes de *NORCHIP'02*, Novembre 2002.

[Goo02] K. Goossens et al. "Networks on silicon: Combining besteffort and guaranteed services". Dans les actes de *Design, Automation & Test in Europe DATE'02*, 2002.

[Gro02] T. Grötke, S. Liao, G. Martin, S. Swan. "System design with SystemC". Par *Kluwer, Dordrecht, The Netherlands*, 2002.

[Gup96] R.K. Gupta, D. Gajski, R. Allen, Y. Trivedi. "Opportunities and pitfalls in HDL-based system design". Dans les actes de *ICCD 1996*.

[Haj99] Y. HajMahmoud, P. Sens, B. Folliot, "Quantifying the Performance Improvement of Migration Mechanism in Load Distributing Systems" *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, Juillet 1999

[Hal91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. "The synchronous data flow programming language LUSTRE". Dans les actes de *IEEE*, 79(9):1305–1320, Septembre 1991.

[Her00] A. Hergenhan, W. Rosenstiel. "Static timing analysis of embedded software on advanced processor architectures". Dans les actes de *the Design, Automation and Test Conference DATE'00*,

Munich, Germany, Mars 2000.

[Hoa69] C.A.R. Hoare. "An axiomatic basis for computer programming". Dans *Communications of the ACM*, 12/10, pp.576-583, Octobre 1969.

[Hyl03] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, and H. Zheng. "Ptolemy II - Heterogeneous Concurrent Modeling and Design in Java, volume 2". Par *Department of Electrical Engineering and Computer Sciences University of California at Berkeley, Memorandum UCB/ERL M03/29, version 3.0*. 2003.

[Iee01] IEEE. "IEEE Standard for Verilog Hardware Description Language". *IEEE*, 2001.

[Iee02] IEEE. "IEEE Standard VHDL Language Reference Manual". *IEEE*, 2002.

[Int97] Intel Corporation, "MultiProcessor Specification Version 1.4". Mai 1997.

[Ism94] T.B. Ismail, M. Abid, and A. Jerraya. "COSMOS: A CoDesign Approach for Communicating Systems", Dans les actes de *Third International Workshop on Hardware/Software Codesign*. Grenoble, France: *IEEE Computer Society Press*, 17-24. 1994.

[Itr03] ITRS Technology Working Group. "International Technology Roadmap for Semiconductors - Design", 2003 edition, 2003.

[Jan04] A. Jantsch. "Models of embedded computation. In Embedded Systems". *CRC Press*, 2004.

[Jan05] A. Jantsch, I. Sander "Models of Computation and Languages for Embedded System Design" 2005.

[Jer04'] A.A. Jerraya and W.Wolf, editors. "Multiprocessor System-on-Chips". *Morgan Kaufmann Publishers Inc.*, Octobre 2004.

[Jer04] A.A. Jerraya, "Long Term Trends for Embedded System Design", *EUROMICRO Symposium on Digital System Design (DSD 2004)*, Rennes, France, Septembre. 2004.

[Jia02] J. Xu and W. Wolf, "Platform-Based Design and the First Generation Dilemma", Dans les actes de *9th IEEE/DATC Electronic Design Processes Workshop (EDP)*, Avril 2002.

[Joh97] M. John S. Smithy. "Application-Specific Integrated Circuits", *Addison-Wesley*, 1ere édition juin 1997.

[Kah77] G. Kahn and D. B. MacQueen. "Coroutines and networks of parallel processes". Dans les actes de *IFIP '77 North- Holland*, 1977.

[Kar03] G. Karsai, J. Sztipanovits, A. Ledeczi, T. Bapty. "Model-Integrated Development of Embedded Software", Dans les actes de *IEEE*, Vol. 91, Number 1, pp. 145-164. Janvier, 2003.

[Ken99] C. Kern and M. R. Greenstreet. "Formal verification in hardware design: A survey". Dans *ACM Transactions on Design Automation of Electronic Systems*, 4/2, Avril 1999.

[Keu00] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey and A. S. Vincentelli, "System Level Design: Orthogonalization of Concerns and Platform-Based Design". Dans *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19/12, Decembre 2000.

- [Kjh03] N. KJha, S. Gupta. "Testing of Digital Systems - Technology" – 2003.
- [Kri05] L. Kriaa "Modélisation et validation des systèmes hétérogènes : définition d'un modèle d'exécution", *document de thèse*, Novembre 2005
- [Laj99] M. Lajolo, M. Lazarescu, A. Sangiovanni-Vincentelli. "A Compilation-based Software Estimation Scheme for Hardware/Software Co-simulation", Dans les actes de *CODES'99*, 1999.
- [Lee87] E. A. Lee and D. G. Messerschmitt. "Static scheduling of synchronous data flow programs for digital signal processing". Dans *IEEE Transactions on Computers*, C-36/1:pp.24–35, Janvier 1987.
- [Lee98] E. A. Lee and A. Sangiovanni-Vincentelli, "A Framework for Comparing Models of Computation," Dans *IEEE Transaction Computer-Aided Design*, 17/12, Decembre 1998.
- [Lem02] J.-F. Lemire, S. Regimbal, S. Bois, Y. Savaria, and E. M. Aboulhamid. "A survey on current functional verification practice". *Technical report*, *Ecole Polytechnique de Montréal, Université de Montréal*, 2002.
- [Mad03] J. Madsen, S. Mahadevan, K. Virk, M. Gonzalez. "Network-on-Chip Modeling for System-Level Multiprocessor Simulation," Dans les actes de *24th IEEE International Real Time Systems Symposium (RTSS'03)* pp.256, 2003.
- [Mau04] S. Mauw, W. Wiersma, T. Willemse. "Language-driven system design". Dans *International Journal of Software Engineering and Knowledge Engineering (2004)*.
- [Mentor] Mentor Seamless CVE, Disponible sur <http://www.mentor.com>
- [Moi04] R. Le Moigne, O. Pasquier, J-P. Calvez. "A Generic RTOS Model for Real-time Systems Simulation with SystemC," *Design, Automation and Test in Europe Conference and Exhibition Designers' Forum DATE'04*, pp. 30082, 2004.
- [Nic02] G. Nicolescu, K. Svarstad, W. Cesario, L. Gauthier, D. Lyonnard, S. Yoo, P. Coste, A.A. Jerraya, "Desiderata pour la spécification et la conception des systèmes électroniques ". Dans *Technique et Science Informatiques*, Mars 2002.
- [Nic04] A. A. Jerraya, G. Nicolescu, "La spécification et la validation des systèmes monopuces", *HERMES Science Publications*, 2004.
- [Nomadik] Nomadik multimedia processor, Disponible sur <http://www.st.com/stonline/prodpres/dedicate/proc/proc.htm>
- [Omap] OMAP platform. Disponible sur <http://focus.ti.com/omap/docs/omaphomepage.tsp>
- [Pos04] H. Posadas, F. Herrera, P. Sánchez, E. Villar F. Blasco. "System-level performance analysis in SystemC". Dans les actes de *the Design, Automation and Test Conference, Paris, France*, Fevrier 200.
- [Qui94] P. Quinton, "Towards a multi-formalism framework for architectural synthesis: the ASAR" *Codes/CASHE'94*, pages 25-32, *Grenoble, France*, Septembre 1994.

- [Raj99] V. Rajesh and R. Moona. "Processor modeling for hardware software codesign". Dans les actes de *International Conference on VLSI Design, Goa, India*, Janvier 1999.
- [Ric05] E. Riccobene, P. Scandurra, A. Rosti, S. Bocchio. "A SoC Design Methodology Involving a UML 2.0 Profile for SystemC," Dans les actes de *Design, Automation and Test in Europe (DATE'05) Volume 2*, pp. 704-709, 2005.
- [Ros03] W. Rosenstiel, "Modeling of Systems-on-Chip in SystemC", Dans les actes de *3rd Int'l Seminar on Application-Specific Multi-Processor SoC, Chamonix, France*, Juillet 2003.
- [Sas03] R. Sasanka, S.V. Adve, Y.-K. Chen, and E. Debes. "Comparing the Energy Efficiency of CMP and SMT architectures for Multimedia Workloads". *Technical Report UIUCDCS-R-2003-2325, University of Illinois at Urbana-Champaign*, Mars 2003.
- [Sch86] D.A. Schmidt. "Denotational Semantics: A Methodology for Language Development". *Allyn and Bacon, Inc., Newton, MA*, 1986.
- [Sdl] Simple DirectMedia Layer. Disponible sur <http://www.libsdl.org/index.php>
- [Sha04] A. Shankar Basu, M. Lajolo, M. Prevostini. "UML in an Electronic System Level Design Methodology". Dans les actes de *UML-SOC'04 - International Workshop on UML for SoC Design*, pp.47-52, *San Diego, CA (U.S.A)*, Juin 2004.
- [Son03] Sonics Inc. "Product Brief: SiliconBackplane III MiroNetwork IP". <http://www.sonicsinc.com/sonics/products/siliconbackplaneIII/productinfo/docs/siliconbackplaneIII.pdf>, Mai 5th, 2003.
- [Specc] SpecC. Disponible sur <http://www.specc.org/>
- [Spr05] L. Spracklen and S. Abraham. "Chip Multithreading: Opportunities and Challenges". Dans les actes de *11th Int'l Symp. On High-Performance Computer Architecture (HPCA), San Francisco, USA*, Fevrier 2005.
- [Systemc] SystemC. Disponible sur <http://www.systemc.org/>
- [Szt97] J. Sztipanovits. "Model-integrated computing environments and computer-based systems". *ECBSpp 480*. 1997.
- [Tan03] T. Tan and A. Raghunathan. "SW Architectural Transformation : A new approach to Low Energy Embedded SW". Dans les actes de *Design, Automation & Test in Europe, Munich, Germany*, Mars 2003.
- [Tan90] A. Tanenbaum. "Structured Computer Organization". *Prentice Hall Inc.*, 1990.
- [Tan92] A. Tanenbaum. "Modern Operating Systems". *Prentice Hall Inc.*, 1992.
- [Tan95] S. M. Tan, et. al., "Virtual Hardware for Operating System Development", Technical rep., UIUC, Septembre. 1995. disponible sur <http://choices.cs.uiuc.edu/uChoices/Papers/uChoices/vchoices/vchoices.pdf>
- [Tan96] A.S. Tanenbaum. "Computer Networks". *Prentice Hall*, 1996.

- [Thi98] S. Thibault. "Domain-Specific Languages: Conception, Implementation and Application". *Thèse de doctorat, IRISA/Université de Rennes 1*, 1998.
- [Van00] A. van Deursen, P. Klint, and J. Visser. "Domain-specific languages: An annotated bibliography". *SIGPLAN Notices*, 35/6, pp.26-36, 2000.
- [Van05] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, K. De Bosschere, "DIABLO: a reliable, retargetable and extensible link-time rewriting framework", Dans les actes de *2005 IEEE International Symposium On Signal Processing And Information Technology*. IEEE. 2005. pp. 7-12
- [Vcc] "Virtual Component Codesign", *Cadence Design Systems Inc.* Disponible sur <http://www.cadence.com/products/vcc.html>
- [Vin02] A. Sangiovanni-Vincentelli, "Platform-based Design", *EEDesign de EETimes*, Février 2002.
- [Vxsim] VxSim. Disponible sur www.windriver.com/products/vxsim/
- [Wei94] M. Weiser, B. Welch, A. Demers, and S. Shenker. "Scheduling for reduced cpu energy". Dans les actes de *USENIX Symposium on OSDI*, 1994.
- [Win93] G. Winskel. "The Formal Semantics of Programming Languages". *MIT Press*, 1993.
- [Wol03] W. Wolf, "A Decade of Hardware/Software Codesign," Dans les actes de *IEEE Computer*, vol. 36, pp. 38- 43, Avril 2003.
- [Yoo02] S. Yoo, G. Nicolescu, L. Gauthier, and A.A. Jerraya. "Automatic generation of fast timed simulation models for operating systems in SoC design". Dans les actes de *the Design, Automation and Test Conference, Paris, France*, Mars 2002.
- [Yoo03] S. Yoo, I. Bacivarov, A. Bouchhima, Y. Paviot, A.A. Jerraya. "Building fast and accurate SW simulation models based on hardware abstraction layer and simulation environment abstraction layer". Dans les actes de *Design, Automation and Test Conference, Munich, Germany*, Mars 2003.
- [Yoo04] S. Yoo, M.W. Youssef, A. Bouchhima, A.A. Jerraya, M. Diaz-Nava, "Multi Processor SoC Design Methodology Using a Concept of Two-Layer Hardware-Dependent Software", Dans les actes de *Design, Automation & Test in Europe DATE'04, Paris, France*, Février 2004.
- [Yoo05] S. Yoo, A.A. Jerraya, "Hardware/Software Cosimulation from Interface Perspective", *IEE Proceedings Computers & Digital Techniques*, 2005.
- [You04] M.W. Youssef, S. Yoo, A. Sasongko, Y. Paviot, A.A. Jerraya, "Debugging HW/SW Interface for MPSoC: Video Encoder System Design Case Study", Dans les actes de *Design Automation Conference, DAC'04, San Diego, USA*, Juin 2004.
- [Zit93] M.Zitterbart, "A Model for Flexible High performance Communication Subsystems", Dans *IEEE Journal on selected areas in communication*, 11/4, Mai 1993.
- [Ziv96] V. Zivojnovic, S. Pees, and H. Meyr, "LISA---Machine description language and generic machine model for HW/SW co-design," Présenté dans *IEEE VLSI Signal Processing Workshop*, 1996.

7.1.1. RESUME

L'exploration et la validation des choix architecturaux liés à la fois à la conception de la plate-forme matérielle et au logiciel embarqué s'avèrent très importantes afin d'atteindre un compromis performance/coût judicieux. Cette exploration/validation est d'autant plus critique qu'elle se situe dans un contexte étroit de temps de mise sur le marché. Aujourd'hui, le coût d'un tel processus est estimé à plus de 70% du coût total de développement des systèmes mono puces. L'analyse des flots de conception classiques montre que les causes d'un tel coût de développement peuvent être ramenées, en grande partie, à l'intégration tardive des parties logicielles et matérielles d'un système MPSoC. Les travaux de cette thèse s'intéressent à ce problème d'intégration tardive en proposant un modèle unifié permettant la représentation conjointe à différents niveaux d'abstraction des architectures logiciel/matériel. Ce modèle doit faciliter la conception graduelle de ces architectures tout en permettant la validation et l'évaluation, à chaque niveau d'abstraction, des performances qui en découlent. Les contributions apportées par cette thèse sont (1) la définition d'un modèle de représentation unifié et à différents niveaux d'abstraction des architectures logicielles/matérielles des systèmes MPSoC basé sur le concept d'interface abstraite logiciel/matériel, (2) la spécification d'une sémantique d'exécution de ce modèle dans le cadre d'un environnement de cosimulation globale basé sur SystemC et (3) la proposition d'une méthodologie de raffinement automatique de ces interfaces abstraites exploitant une technologie de composition à base de graphe de dépendance de services.

7.1.2. MOTS-CLES

Interface logiciel/matériel, Abstraction, Validation.

7.1.3. TITLE

Modeling embedded software at different abstraction levels for validation and synthesis of System-on-chip

7.1.4. ABSTRACT

Exploring and validating architectural choices related both to hardware platform design and embedded software is a key enabler to reach a convenient performance/cost tradeoff. By analyzing classic design flows, it turns out that the major source behind such development cost is due to the late integration of hardware and software parts of a multiprocessor system-on-chip (MPSoC) system. In this thesis, we address this problem of late integration by proposing a unified model allowing the joint representation, at different abstraction levels, of the hardware/software architecture. This model is aimed at easing the gradual SoC design while allowing the validation and the evaluation of the resulted performance at each abstraction level. The contributions of this thesis are (1) the definition of a unified representation model of hardware/software architectures at different abstraction levels, (2) the specification of an execution semantic of this unified model in the context of a global cosimulation environment based on SystemC and (3) a methodology for the automatic refinement of these abstract interfaces relying on a composition technology based on the service dependency graph.

7.1.5. INTITULE ET ADRESSE DU LABORATOIRE

Laboratoire TIMA, 46 avenue Félix Viallet, 38031 Grenoble Cedex, France.

ISBN : 2-84813-089-X (version brochée)

ISBNE : 2-84813-089-X (version électronique)