E C O L E
P O L Y T E C H N I Q U E
D E T U N I S I E

MEMOIRE DE MASTERE SERC

Option: Communication et électronique

Réalisé par

Mouna BAKLOUTI

Etude des modèles haut niveau d'architectures logicielles/matérielles pour la conception et l'évaluation des systèmes multiprocesseurs monopuces

Préparé au: Laboratoire TIMA – Groupe SLS

Jury: M. Adel GHAZEL – Maître de Conférences à SUPCOM (Président)
M. Abderrazek JEMAI – Maître-assistant à l'INSAT (Membre)
M. Adel BENZINA – Maître-assistant à l'EPT (Encadrant)
M. Ahmed Amine JERRAYA – Directeur de Recherche TIMA (Invité)

Année universitaire 2006/2007

A ma mère et mon père,
A mes deux sœurs,
A tous ceux qui me sont chers.

Remerciements

Au terme de ce travail, je tiens tout d'abord à remercier chaleureusement les membres du jury: Mr. Adel GHAZEL, Maître de conférences à SUPCOM, de m'avoir fait l'honneur de présider le jury de mon mémoire de mastère, Mr. Abderrazek JEMAI, Maître-assistant à l'INSAT, d'avoir participé au jury et Mr Adel BENZINA, Maître-assistant à l'Ecole Polytechnique de Tunisie, qui m'a encadré tout au long de ce mémoire.

Je voudrais adresser de même mes sincères remerciements à mes deux autres encadrants: Mr Ahmed Amine JERRAYA, directeur de recherche au CNRS et qui était responsable du groupe SLS du laboratoire TIMA, de m'avoir accueilli dans le groupe et d'avoir encadré ce travail de mastère et Mr Aimen BOUCHHIMA, Docteur en Post-Doc au laboratoire TIMA, de m'avoir guidé et orienté dans mon travail, je les remercie aussi pour leurs conseils pertinents.

J'exprime finalement ma gratitude à tous ceux qui m'ont aidé et soutenu durant ce stage de recherche; et j'exprime ma profonde reconnaissance à tous mes collègues et à tous les membres de l'équipe SLS pour leur active collaboration et pour leur sympathie.

Résumé

Dans les systèmes embarqués, le logiciel prend une place de plus en plus importante, et sa complexité augmente d'autant. Ces systèmes étant très largement utilisés dans les applications récentes, il est important de pouvoir les développer rapidement.
L'accroissement de la complexité de ces systèmes fait de la maîtrise de leurs conceptions un défi à relever par les concepteurs.
En effet, la tendance à améliorer la productivité et réduire le temps de mise sur le marché (*time-to-market*), fait que le niveau transfert de registres (RTL) devient insatisfaisant pour la conception et le flot de vérification.
Pour dépasser ce défi, les nouvelles méthodes de conception sont basées sur des concepts d'abstraction de haut niveau. Une voie pour surmonter la complexité consiste donc à élever le plus possible le niveau d'abstraction des descriptions des systèmes à concevoir, à savoir le niveau de modélisation transactionnel (TLM).

Dans ce travail, nous proposons d'étendre le niveau TLM – vu comme le haut niveau d'abstraction du niveau RTL – pour faciliter la conception et la validation du logiciel embarqué.
Nous visons à présenter les nouveaux concepts du TLM conçu pour le logiciel (***SW TLM***) permettant le raffinement de la communication logicielle.
La méthodologie proposée permet aux concepteurs de décider conjointement à propos de l'architecture logicielle ainsi que matérielle, afin d'assurer une performance maximale dans leurs conceptions. Ainsi, l'hétérogénéité des systèmes multiprocesseurs monopuces serait considérée plus efficacement de point de vue communication.

Le premier chapitre de ce document présentera brièvement notre travail.
Le deuxième chapitre abordera quelques généralités sur la conception des systèmes multiprocesseurs monopuces et présentera le flot de conception et de validation des systèmes hétérogènes monopuces proposé par le groupe SLS sur lequel nous avons travaillé.
Plus spécifiquement, le niveau auquel nous nous sommes intéressés est le niveau Architecture Virtuelle qui sera décrit dans le troisième chapitre. A ce stade, un nouveau flot de conception sera présenté parallèlement à la

description de la nouvelle méthodologie SW TLM. Son implémentation ainsi que ses différents concepts de base seront détaillés dans le chapitre d'après.
Le cinquième chapitre présentera la validation de ces concepts sur l'application MJPEG. Le dernier chapitre conclura et soulignera des perspectives de ce travail.

Mots clés
Systèmes multiprocesseurs monopuces, logiciel embarqué, flot de conception, niveaux d'abstraction, Architecture Virtuelle, niveau de modélisation transactionnel TLM, SW TLM.

Abstract

Embedded software takes an important place in embedded systems and its complexity increases more and more. These systems are largely used in recent applications so it is important to be able to develop them quickly.
With the increasing complexity of embedded systems, mastering their designs is a challenge faced by the designers.
Indeed, the tendency to improve the productivity and to reduce the time to market makes Register Transfer Level (RTL) insufficient for the design and verification flow.
To deal with this challenge, the new design methods are based on high-level abstraction concepts. So, one way to overcome complexity consists at raising the abstraction level when designing embedded systems; the transactional level modeling TLM is emerging.

In this work, we propose to extend TLM approach – initially intended as a higher level abstraction of RTL hardware design – to cope with embedded software (SW) design and validation. We aim at introducing new SW TLM concepts which will enable refinement of communication at the SW side.
The proposed methodology allows system designers to decide about HW and SW communication architecture jointly, so as to ensure maximum performance efficiency for their designs. As such, multiprocessor systems on chip (MPSoC) heterogeneity would be addressed more efficiently from communication viewpoint.

The first chapter will briefly present our work. The second chapter will give a general overview of MPSoC design flow and will present the design and validation flow proposed by the SLS group. More specifically, we concentrate on Virtual Architecture level which will be described in the third chapter. At this stage, a typical design flow involving the VA level will be presented and the new SW TLM methodology will be described. Its implementation as well as its basic concepts will be detailed in the following chapter.
The fifth chapter will present the validation of these concepts on an MJPEG application. The final chapter will conclude and give prospects to this work.

Key words

MPSoC, embedded software design, design flow, abstraction levels, Virtual Architecture, Transaction Level Modeling, SW TLM.

Table des matières

Liste des figures

Liste des tableaux

Chapitre 1
Introduction

Ce chapitre a pour but de situer le travail de ce mémoire. Pour ce faire, il définit dans une première section les systèmes multiprocesseurs monopuces. Dans la deuxième section, la problématique est exposée. Dans la troisième section, sont présentés les objectifs à atteindre ainsi que les solutions apportées par le groupe SLS pour faire face à ces problèmes.
Notre contribution apportée par ce mémoire sera détaillée dans la section suivante. Enfin, le plan du document sera détaillé.

1.1 Contexte: les systèmes multiprocesseurs monopuces

Ce travail s'inscrit dans le domaine de la conception des systèmes embarqués multiprocesseurs monopuces, plus communément appelés *MPSoC*[1].
Les progrès technologiques constants en terme d'intégration sur silicium ont permis de concevoir des systèmes sur puces de plus en plus complexes afin de répondre à une demande forte du marché pour des applications telles que les systèmes multimédia, la téléphonie mobile ou encore les applications de jeux vidéo. On a ainsi vu naître une nouvelle catégorie de systèmes ces dernières années, incluant un ou plusieurs processeurs, des composants dédiés et des modules d'entrée-sortie, le tout sur une seule puce. Ces systèmes sont appelés *Systèmes sur Puce*, ou *System-on-a-Chip* (*SoC*) en anglais. Une grande partie de ces systèmes consistent en l'intégration sur une même puce de plusieurs processeurs, DSP, IP matériel, mémoires, bus partagés, etc. On parle alors de systèmes multiprocesseurs monopuces (*MPSoC*). Ces systèmes sont réalisés en interconnectant des noeuds de calcul avec un réseau de communication.

Pour faire face à la complexité de tels systèmes, des méthodes de conception permettant le découplage de la communication et du calcul ont été proposées [14] [27]. La communication prend d'ailleurs une place de plus en plus importante dans la conception de tels systèmes [16].
La conception des systèmes sur puce doit faire face à de nombreuses contraintes de performance, de consommation et de coût, pour lesquelles

[1] *Multi Processor System on Chip*

actuellement, seules des plateformes spécifiques[2] à chaque application peuvent répondre [1].

De plus, l'augmentation des performances par la fréquence ou par les techniques « classiques » (pipeline, prédiction de branchement,...) ne sont plus significatives en terme de performance et induisent une consommation inacceptable [24]. Une solution pour répondre aux problèmes de performance et de consommation consiste à augmenter le parallélisme dans les systèmes par l'intégration de plusieurs processeurs de type hétérogènes[3] afin de cibler au mieux les applications.

En 2001, les systèmes de prévisions stratégiques du rapport ITRS [12] prévoyaient que 70% des ASIC (*Application Specific Integrated Circuit*) comporteraient au moins un processeur embarqué à partir de l'année 2005. Aujourd hui, les SoC peuvent intégrer de nombreux processeurs et cette tendance est confirmée par le rapport ITRS 2005 [13].

Les figures 1.1(a) et 1.1(b) tirées de [15] montrent que près d'un système sur deux est un système multiprocesseur et que pour une partie importante d'entre eux il s'agit de processeurs hétérogènes.

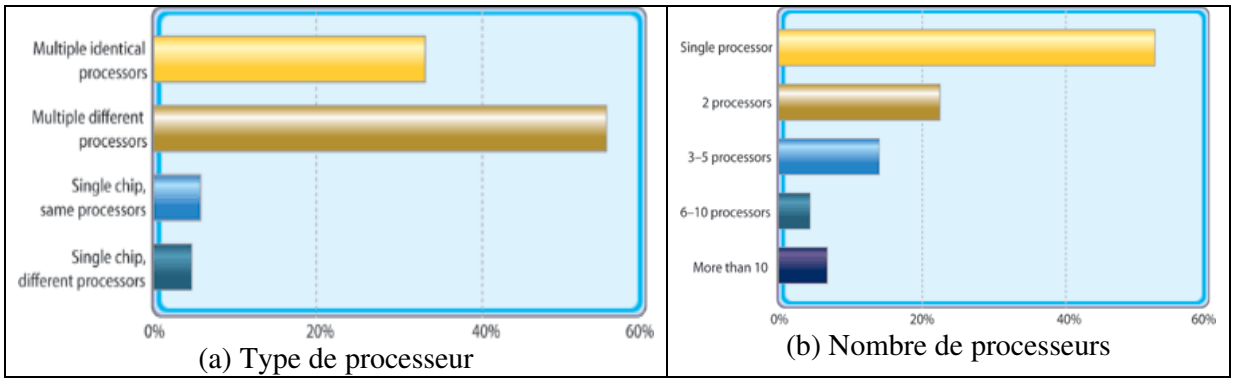


Figure 1.1 - Répartition du nombre et du type de processeurs dans les systèmes actuels

Il est prévu que ces systèmes monopuces soient les principaux vecteurs d'orientation de toute l'industrie des semi-conducteurs. Il est donc crucial de maîtriser la conception de tels systèmes tout en respectant les contraintes de mise sur le marché et les objectifs de qualité.

Le grand défi en ce moment pour les ingénieurs est de réussir à maîtriser la complexité lors de la conception de ces systèmes et d'arriver à une conception

[2] Par rapport aux plateformes à usage générale
[3] Processeurs à usage général, processeurs de traitement de signal, ...

rapide des systèmes monopuces sous de fortes contraintes de qualité et de temps de développement.
Pour dépasser ce défi, les nouvelles méthodes de conception sont basées sur des concepts d'abstraction de haut niveau.

1.2 Problématique: la complexité et les difficultés de la conception des systèmes MPSoC

Habituellement, les flots de conception des systèmes embarqués utilisent principalement le modèle RTL (*Register Transfer Level*) pour modéliser le système au niveau des transferts de registres.
Ainsi, à ce niveau d'abstraction, nous devons modéliser tous les signaux passant entre les différentes entités de simulation. Ce niveau est temporisé au cycle d'horloge près. C'est pourquoi les simulations matérielles effectuées au niveau RTL sont relativement longues, ce qui peut allonger le temps de conception du système.
En plus, un modèle global d'une architecture classique logicielle/matérielle est conventionnellement décrit au niveau RTL/ISA[4]. 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*[5]). Ceci inclut l'architecture locale du nœud logiciel (processeur, mémoire, 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'adresses, bus de données, signaux de contrôle, signaux d'interruptions, 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.
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.
De même, nous notons l'absence de méthodologie et d'outils permettant une transition non brutale de la spécification initiale à l'architecture finale.

[4] *Instruction Set Architecture*
[5] *Hardware Description language*

Pour faire face à ces défis, la modélisation au niveau transactionnel (TLM) a été récemment favorisée pour la conception matérielle parallèlement à une proposition de nouveau flot de conception graduel.
En effet, une solution suggérée est l'augmentation du niveau d'abstraction des modèles pour améliorer la productivité.
Bien que l'ultime objectif du TLM est de permettre le développement tôt du logiciel embarqué ainsi que de paralléliser le développement du matériel et du logiciel dans le cycle de conception des systèmes sur puce, aucune modélisation TLM n'a été définie pour le logiciel.
Dans les applications TLM classiques, le logiciel est considéré au niveau fonctionnel ou bien complètement raffiné et simulé à un très bas niveau d'abstraction sur un simulateur de jeux d'instructions (*ISS: Instruction Set Simulator*) concurremment avec des simulations matérielles, au niveau transaction ou au niveau cycle.
De même, l'évaluation des sous systèmes logiciels embarqués, tôt dans les étapes de conception n'est plus faisable en employant les approches traditionnelles de simulation, précises au niveau cycle. Le problème vient de la vitesse lente de simulation de l'ISS.
Pour une simulation plus rapide, nous avons pensé à un niveau transactionnel pour le logiciel similaire à celui pour le matériel, ceci sera considéré à un haut niveau d'abstraction. D'où les composants aussi bien logiciels que matériels seront modélisés avec un modèle unique afin d'aborder la conception du système dans une seule et même approche cohérente.

1.3 Objectifs et solutions proposées par le groupe TIMA-SLS

Le groupe SLS s'est focalisé sur la conception conjointe du logiciel et du matériel afin de résoudre les problèmes soulevés dans les paragraphes précédents. Cette conception se base sur un raffinement graduel à différents niveaux d'abstraction.
La Figure 1.2 montre les deux niveaux intermédiaires proposés dans le flot de conception des SoC à savoir *Virtual Architecture* (niveau OS du côté logiciel) et *Transaction Accurate* (niveau HAL du côté logiciel).

Les traits continus forts correspondent au cas classiques de co-simulation[6] logicielle/matérielle tels que proposés par les approches conventionnelles. Remarquons que ces approches se basent exclusivement sur le niveau ISA du côté du logiciel. L'utilisation des niveaux TLM pour le matériel est assez récente. Historiquement c'est le niveau RTL qui était utilisé comme niveau de référence pour la co-simulation logicielle/matérielle à côté du niveau ISA.

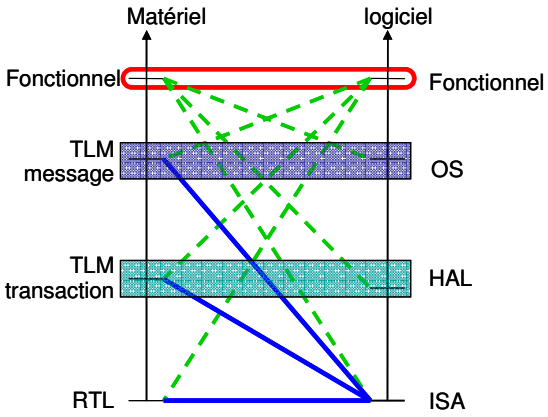


Figure 1.2 – Niveaux de validation de l'interface logicielle/matérielle

L'utilisation d'un modèle à base d'ISS pour la co-simulation logicielle/matérielle 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 co-simulation d'un système logiciel/matériel tôt dans le cycle de conception.

Donc, pour remédier à la discontinuité observée dans les flots classiques de conception et de validation des systèmes MPSoC, le groupe SLS a introduit les concepts de *Virtual Architecture* et de *Transaction Accurate* comme étant des étapes intermédiaires dans le flot de conception permettant la validation, par co-simulation 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.

[6] Simuler conjointement les diverses parties d'un système hétérogène

La Figure 1.3 donne une vision simplifiée du flot de conception proposé par le groupe SLS :

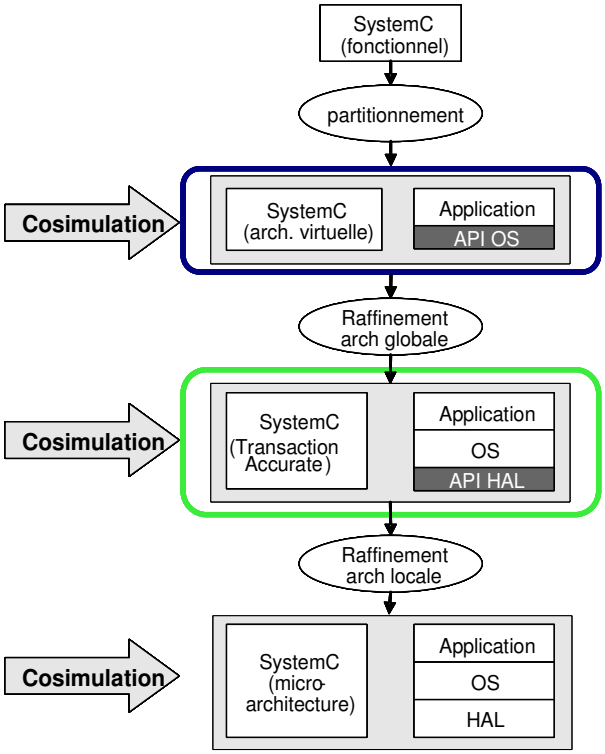


Figure 1.3 – Flot de conception proposé par le groupe SLS

Ce flot débute au niveau fonctionnel après que le partitionnement logiciel/matériel ait été décidé. Il se termine au niveau micro-architecture (RTL), où une étape classique de compilation et de synthèse logique permet d'aboutir à la réalisation finale du système. C'est un flot descendant qui permet de simuler à tous les niveaux et éventuellement de revenir en arrière à chaque étape.

Dans ce flot, l'architecture virtuelle résulte d'une première étape de partitionnement de la spécification fonctionnelle initiale. Le partitionnement sépare les parties qui seront implémentées de façon matérielle de celles qui seront implémentées de façon logicielle.

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

Le groupe SLS s'est intéressé aussi à la génération automatique des interfaces logicielles/matérielles.

1.4 Contributions

Durant ce travail, nous avons contribué à la définition des niveaux de conception intermédiaires notamment *Virtual Architecture* et *Transaction Accurate* proposés par le groupe TIMA-SLS.

Plus précisément, nos travaux au niveau T.A [20] consistaient à prouver l'intérêt et la faisabilité d'une estimation de performance des systèmes en utilisant ce modèle haut niveau de l'architecture logicielle/matérielle.
Nous avons aussi essayé d'adapter le niveau T.A au niveau TLM de ST-Microelectronics.

Nos travaux au niveau V.A qui est un niveau plus abstrait que T.A consistent à valider l'application logicielle sur un modèle de simulation du système d'exploitation et de l'architecture. En effet, nous avons proposé un modèle d'extension pour le niveau V.A.
Nous voulons pour cela étendre le niveau TLM pour supporter non seulement la partie matérielle mais également la conception et la validation du logiciel embarqué.
La contribution attendue est de proposer alors une méthode permettant d'unifier la présentation du logiciel embarqué. 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.
Nous proposons par la suite un modèle transactionnel pour le logiciel embarqué au niveau V.A basé sur TLM OSCI[7]. Les propriétés de ce modèle permettent d'unifier la représentation des composants logiciels et matériels.
L'objectif général à terme étant de développer un environnement complet de simulation d'architectures MPSoC au niveau TLM, basé sur SystemC.

1.5 Plan de ce document

Ce document est organisé en six chapitres dont cette introduction.

[7] *Open SystemC Initiative*

Le second chapitre est dédié à la conception des systèmes MPSoC et pause de manière plus précise la problématique et les objectifs de ce travail.
Le chapitre trois définit le niveau d'abstraction intermédiaire: Architecture Virtuelle et présente alors un flot de conception typique introduisant ce niveau. A ce stade, nous expliquons le raffinement de la communication logicielle et nous définissons la terminologie TLM pour le logiciel « **SW TLM** ».
Le chapitre quatre détaille différents aspects nécessaires pour l'implémentation du SW TLM.
Une application de l'approche est ensuite présentée dans le chapitre cinq. Finalement, le dernier chapitre conclut ce document et propose quelques perspectives potentielles à ce travail.

Chapitre 2
Conception des systèmes multiprocesseurs monopuces

2.1 Introduction

Ce chapitre a pour objet de donner un aperçu de la problématique de la conception des systèmes MPSoC en examinant les flots de conception classiques. Il introduit ensuite le flot de conception proposé par le groupe TIMA SLS.
La première section décrit les architectures logicielles et matérielles des systèmes multiprocesseurs monopuces. Puis, la section suivante introduit les niveaux de modélisation notamment RTL et TLM.

2.2 Architectures logicielles/matérielles des systèmes multiprocesseurs monopuces

Avant de détailler la conception des systèmes multiprocesseurs monopuces, il est nécessaire de dresser un bref aperçu des architectures multiprocesseurs.
En effet, il existe principalement deux types d'organisations pour les architectures des systèmes Multiprocesseurs [8]:

- Mémoire partagée: Dans ce type d'organisation, l'architecture matérielle est en général composée de plusieurs processeurs identiques. L'application *Multithread* repose sur une seule pile logicielle. La communication entre les différents processeurs s'effectue par une mémoire partagée globale.
- Passage de messages: Cette organisation repose sur plusieurs piles logicielles s'exécutant sur des sous systèmes hétérogènes, aussi bien en terme de processeurs, qu'en terme d'entrées/sorties. La communication entre les sous-systèmes est réalisée par passage de messages.

Afin d'intégrer un plus grand nombre de processeurs, les architectures MPSoC hétérogènes combinent généralement ces deux modèles [22]. Les futurs MPSoC hétérogènes seront composés de plusieurs sous-systèmes également hétérogènes, chacun pouvant contenir un nombre important de processeurs identiques exécutant une seule pile logicielle [2].

2.2.1 Architectures matérielles

L'architecture matérielle des systèmes multiprocesseurs monopuces peut être représentée d'une manière générale par un ensemble d'unités d'exécution pouvant être logicielles ou matérielles connectées sur un réseau de communication (Figure 2.1(a)). On parlera de nœud logiciel ou de nœud matériel.

L'architecture matérielle des systèmes MPSoC peut être décomposée en quatre blocs de base: (1) processeur ou sous-système processeur pour exécuter le logiciel, (2) modules mémoires ou unités de stockage de données, (3) sous-système de calcul composé de matériel spécifique et (4) un réseau d'interconnexion. Il y a eu un développement et une sophistication continus de chacun de ces blocs de base, mais c'est surtout leur arrangement qui différencie un système MPSoC d'un autre.

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.

La communication peut être assurée par des réseaux de communication complexes (bus hiérarchiques, bus avec protocole TDMA, connexion point à point, structure en anneau et même des réseaux de communication par paquets). On trouve aussi les réseaux de communication sur puce (NoC de l'anglais Network on Chip) qui constituent une alternative radicale aux bus partagés.

L'architecture matérielle d'un nœud logiciel (Figure 2.1 (b)), appelée sous-système processeur, est composée d'un ou plusieurs processeurs identiques ainsi que des composants périphériques nécessaires pour leur interfaçage ou pour l'accélération de performance.

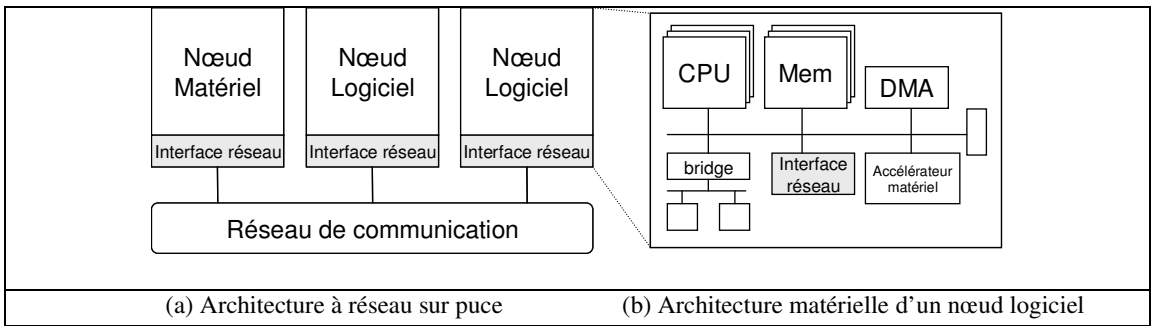


Figure 2.1 - Architectures matérielles

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.

Comme dans la plupart des architectures logicielles actuelles, la pile logicielle utilisée est organisée en couches pour des raisons de standardisation et de réutilisation.

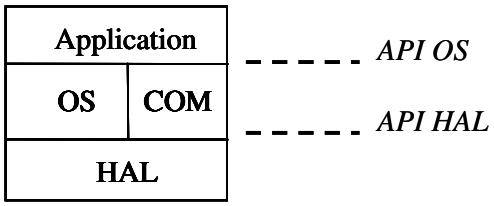


Figure 2.2 – Architecture logicielle

Chaque couche fournit à la couche supérieure une interface de programmation propre (API[8]).

2.2.2.1 La couche applicative

La couche applicative constitue la partie fonctionnelle du logiciel embarqué. En effet, le programme applicatif se compose d'un ensemble de tâches communicantes, réalisant le comportement de l'application tel que décrit dans la spécification fonctionnelle.

Cette couche est utilisée par le concepteur du logiciel pour représenter son application sans se préoccuper de l'architecture matérielle.

Généralement *Multithread*, elle est implémentée sur une API de modèle de programmation parallèle, permettant d'abstraire les détails de l'architecture logicielle et matérielle sous-jacentes. Cette séparation est nécessaire pour le développement du logiciel et du matériel de manière concurrente.

La couche API représente les appels système de haut niveau invoqués par les tâches logicielles. Ainsi, à ce niveau de description, n'existent que des informations liées aux traitements à réaliser.

[8] *Application Programming Interface*

2.2.2.2 La couche Système d'exploitation - Communication

Cette couche permet de gérer localement les ressources disponibles. Ceci inclut par exemple la gestion des tâches, la communication inter tâches et la communication externe, l'ordonnancement (*Scheduling)*, etc.

Dans les systèmes embarqués, le système d'exploitation (SE ou *OS*[9]) est vu comme l'entité logicielle qui permet 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.

L'usage de systèmes d'exploitation est devenu nécessaire dans les systèmes embarqués, du fait de la complexité croissante de ces systèmes, de la présence de fortes contraintes temps réel, de la limitation des ressources disponibles, tant en mémoire qu'en énergie disponible et donc en puissance de calcul, mais également de la pression exercée par le marché sur ces produits. En effet, le temps de développement doit être raisonnable, afin de limiter le temps de mise sur le marché, et ainsi d'assurer le succès du produit. Parmi les exemples des systèmes d'exploitation embarqués nous citons: QNX, eCos, RTLinux, VxWorks, etc.

Brièvement, les fonctions de base d'un système d'exploitation sont:

- gestion de tâches et ordonnancement;
- services d'interruption;
- communication inter tâches et synchronisation;
- gestion de mémoire.

Dans notre travail, nous utilisons un modèle de simulation d'OS fait par l'équipe TIMA-SLS.

Ce modèle se compose de :

API OS: regroupe tous les services représentant l'API du système d'exploitation utilisable par l'application logicielle. Cette famille contient trois sous familles:

- IO (*Input/Output*): regroupe tous les services liés aux communications utilisables par l'application logicielle (par exemple le tube (*pipe*));
- Synchronisation: regroupe tous les services liés aux synchronisations utilisables par l'application logicielle (par exemple les sémaphores);
- Autres services de haut niveau.

[9] *Operating System*

Noyau (*Kernel*): regroupe tous les services concernant le noyau du système d'exploitation. Il contient les sous familles suivantes:

- L'amorce (*Boot*): regroupe tous les services liés au démarrage du système d'exploitation. Cette sous-famille initialise les registres des processeurs, la table des vecteurs d'interruptions, les espaces de piles, l'espace d'adressage, etc. Elle charge le noyau en mémoire;
- Changement de contexte (*Cxt*): regroupe tous les services liés à la gestion des contextes associés aux tâches. Les éléments fournissant ces services sont toujours spécifiques au processeur cible;
- Ordonnanceur (*Scheduler*): regroupe tous les services liés à l'ordonnancement des tâches. Pour cela il utilise un algorithme de gestion, généralement par priorité ou tourniquet[10], et gère l'ordre d'exécution des tâches (par exemple la mise en sommeil ou le réveil d'une tâche);
- Tâche (*Task*): regroupe tous les services liés à la gestion des tâches. En pratique, cette famille de services fait le lien entre les autres sous-familles de la famille Noyau (*Kernel*). Elle décrit la structure de la tâche et contient les tables de tâches.

Synchronisation: regroupe tous les services liés aux mécanismes de synchronisation internes au système d'exploitation. Le partage des ressources sur différentes entités concurrentes (par exemple tâches de l'application) impose une politique de protection qui permet d'assurer la cohérence de l'information contenue dans ces ressources.
Parmi les primitives de synchronisation nous notons : *Wait* pour attendre un signal par exemple et *Notify* pour en générer un, *Block* met la tâche courante dans l'état endormi et la place dans une file d'attente, *Unblock* réveille une des tâches endormies dans une file d'attente, etc.
Pour fonctionner, cet élément a besoin des services du Noyau.

Interrupt: regroupe tous les services liés aux interruptions (par exemple la gestion des fonctions d'interruption, ou des appels système). Ces services ont besoin des primitives de synchronisation.

[10] Chaque processus dispose d'un quantum de temps pendant lequel il peut s'exécuter, puis c'est au tour du suivant, en anglais *Round Robin*.

Dans cette même couche nous trouvons la partie COM qui gère principalement la communication de l'application avec le matériel. Ceci est assuré par les pilotes de périphériques.
Un pilote de périphérique fournit un accès aux E/S et sert à les gérer.

2.2.2.3 La couche abstraction du matériel (HAL)

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 re-conception 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*[11]) est introduite pour palier les inconvénients d'une telle dépendance bas niveau de l'architecture matérielle [3].
Cette couche permet l'accès structuré aux ressources; et aussi de cacher les détails bas niveau de l'architecture matérielle. En effet, la couche HAL est une couche logicielle fine qui est supposée fournir une abstraction de l'accès aux ressources matérielles de l'architecture.

2.3 Niveaux de modélisation

L'étude des niveaux d'abstraction de la communication dans les langages de description du matériel n'est pas nouvelle. Dans cette section, nous nous intéressons aux deux niveaux RTL et TLM.

[11] *Hardware Abstraction Layer*

2.3.1 Niveau RTL

Le niveau RTL est probablement le niveau d'abstraction le plus utilisé pour la modélisation des systèmes matériels [23].

Le niveau RTL est clairement identifié. A ce niveau de conception, la communication interne et externe des composants matériels est réalisée par des fils et des bus physiques. Les données intervenant dans une communication sont sous une forme logique, c'est-à-dire qu'elles sont présentées par des vecteurs de bits.

Dans une communication au niveau RTL, le temps est une grandeur réelle, arbitraire et discrète. La granularité de l'unité de temps est le cycle d'horloge et les primitives de communication sont des lectures et écritures sur des ports et l'attente d'un nouveau cycle d'horloge. Ce niveau d'abstraction est supporté dans la majorité des langages HDL[12] en particulier dans VHDL, Verilog et SystemC.

Cependant, ce niveau est de plus en plus considéré comme trop détaillé en approche système. D'une part, il nécessite un travail important pour le décrire complètement, d'autre part son utilisation pour des vérifications par simulation conduit à des temps excessifs.

L'introduction d'un niveau plus abstrait que RTL s'avère nécessaire: il s'agit du niveau TLM.

2.3.2 Niveau TLM

Les modèles transactionnels ont été introduits pour augmenter la productivité des concepteurs de systèmes sur puce. Il s'agit d'un nouveau niveau d'abstraction, plus haut que le niveau RTL qui constitue le point d'entrée des outils de synthèse.

TLM (*Transaction Level Model*) est progressivement adopté par les industriels pour modéliser rapidement et simuler à grande vitesse les architectures de systèmes sur puce (SoC), dans le but notamment de permettre le développement du logiciel embarqué avant de disposer de la description synthétisable complète du matériel, et de procéder à des analyses d'architectures très tôt dans le cycle de conception, ce qui n'était pas possible au niveau RTL, du fait de la lenteur des simulations et de la complexité des modèles manipulés.

[12] *Hardware Description Language*

TLM utilise une approche à base de composants, dans laquelle les blocs matériels sont des modules communicants par le biais des transactions, ou les détails inutiles de communication sont omis. Par suite, il permet d'accélérer la simulation et explorer les alternatives d'implémentation tôt dans le cycle de conception (tel que la topologie du bus, les priorités du bus, l'optimisation de la taille de la mémoire, etc.) [17].
TLM sera plus détaillé dans le chapitre trois.

2.4 Discontinuités du flot classique

La Figure 2.3 représente un flot de conception classique pour les 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.
Ce flot débute par une spécification fonctionnelle de l'application dans des langages haut niveaux, généralement exécutables, tels que SystemC [26] ou Simulink [21].

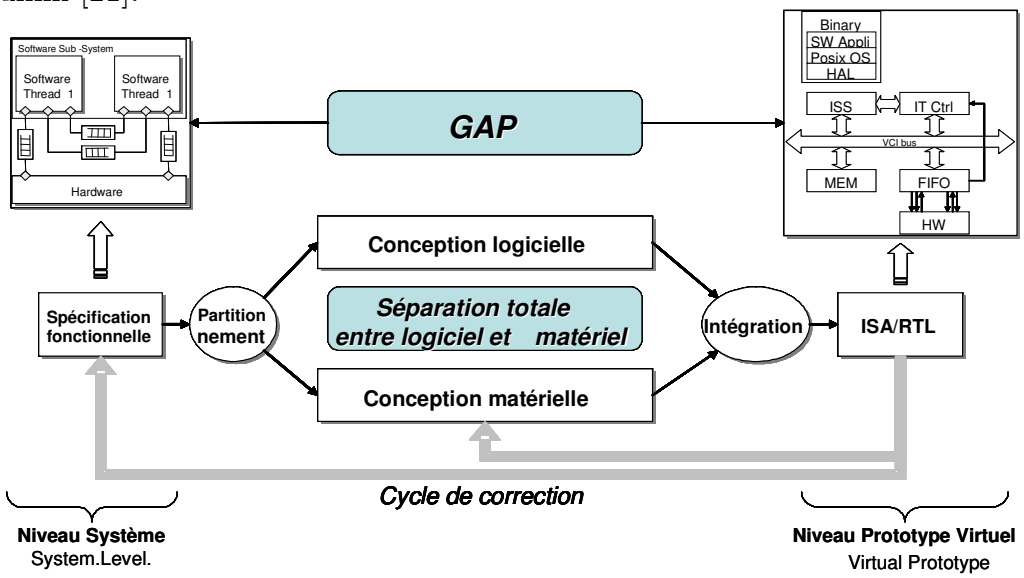


Figure 2.3 – Etapes et modèle d'un flot de conception classique

La seconde étape consiste à répartir les différentes tâches entre le matériel et le logiciel séparant ainsi 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. L'intégration permettra finalement de regrouper à nouveau les deux parties afin d'obtenir un modèle bas niveau où le matériel est décrit en RTL et le logiciel au niveau instruction binaire.

Dans un tel flot il peut être difficile de développer complètement le logiciel sans que le matériel soit défini. C'est pour cela que son développement devait souvent attendre que la partie matérielle soit décrite pour être achevé.

2.5 Nouveau flot de conception proposé dans le groupe TIMA-SLS

L'approche de conception vue précédemment est caractérisée par une discontinuité qui marque le passage de la spécification initiale à l'implémentation finale. Elle constitue comme nous l'avons souligné un handicap dans ce flot classique. 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.
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.

2.5.1 Présentation du flot de conception du groupe SLS

La Figure 2.4 présente la solution apportée par le groupe SLS pour palier aux discontinuités des flots de conception classiques.
Le principe de base de cette approche est de considérer la conception de l'interface logicielle/matérielle comme étant une branche complète du flot. Cette *externalisation* apporte une solution à la discontinuité entre les niveaux d'abstraction sous réserve de la disponibilité de modèles exécutables de l'interface, permettant ainsi de simuler le système complet durant toutes les étapes de la conception conjointe.
La discontinuité dans la conception conjointe du logiciel et du matériel est naturellement solutionnée par cette approche où la conception de l'interface logicielle/matérielle est traitée dans sa globalité.

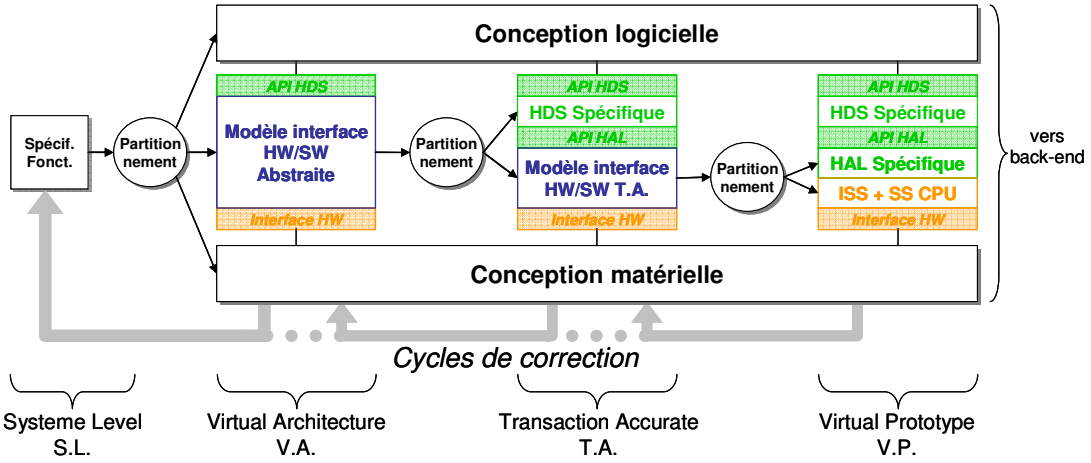


Figure 2.4 - Flot de conception détaillé proposé par le groupe TIMA-SLS

2.5.2 Description des niveaux intermédiaires du flot

En plus des deux niveaux déjà existants dans le flot précédent (*System Level* et *Virtual Prototype*), deux niveaux intermédiaires ont été ajoutés : *Virtual Architecture Level* et *Transaction Accurate Level*. Ils seront décrits brièvement dans les paragraphes suivants.

➲ Niveau *Virtual Architecture*

L'architecture virtuelle résulte d'une première étape de partitionnement de la spécification fonctionnelle initiale. Le partitionnement sépare les parties qui seront implémentées de façon matérielle de celles qui seront implémentées de façon logicielle.

Ce modèle permet d'abstraire tout le logiciel dépendant du matériel, le ou les processeurs et le sous-système processeur. Le modèle de programmation sur lequel peut s'appuyer le concepteur de la partie logicielle est une API de *HDS* (*Hardware Dependent Software*), similaire à celle d'un OS, plus des primitives de communication haut niveau.

A ce stade, l'architecture matérielle est globalement décomposée en sous systèmes, sans pour autant exiger que l'architecture matérielle soit raffinée.

Au niveau de l'architecture virtuelle, le concepteur ne dispose pas d'assez d'informations sur l'architecture matérielle cible.

L'objectif de ce modèle est d'apporter aux concepteurs une première estimation plus ou moins grossière des performances en fonction des choix faits sur le système d'exploitation et les primitives de communication haut niveau (gestion des tâches logicielles par exemple).

➲ Niveau *Transaction Accurate*

Au niveau *Transaction Accurate*, la couche logicielle supérieure du modèle précédent est supposée raffinée. On considère donc que le HDS ne fait plus partie du modèle mais de l'application que l'on veut valider.
Cette étape est caractérisée par la spécification de la nature 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.
Les communications à ce niveau se font avec des adresses spécifiques par des primitives de type *read/write*. La gestion des ressources matérielles comme l'accès aux différents périphériques partagés ou encore aux processeurs sont modélisés.
L'objectif à ce niveau est d'apporter une plus grande précision quant à l'estimation des performances afin de pouvoir valider les décisions de conception prises en amont de manière plus précise.

Dans les deux niveaux *Virtual Architecture* et *Transaction Accurate*, nous utilisons l'exécution native comme mode d'exécution du logiciel embarqué afin de bénéficier de l'avantage d'une simulation rapide à ces niveaux intermédiaires.
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.

➲ Niveau *Virtual Prototype*

A ce niveau, l'architecture logicielle/matérielle est décrite au niveau ISA/RTL.
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 nœud logiciel (processeur(s), mémoire(s), périphériques, etc.) mais aussi les autres parties du système.
A ce niveau, les deux parties logicielle et matérielle sont entièrement conçues.

Ici, nous utilisons un modèle de simulation classique qui considère que l'architecture du système est complètement raffinée et connue dans ses moindres détails. Ainsi, le logiciel embarqué doit être entièrement développé avant d'être compilé par le (les) processeur(s) cible(s). L'image binaire obtenue est alors prise en charge par des simulateurs de processeurs qui interprètent séquentiellement les instructions et interagissent avec un modèle entièrement raffiné de l'architecture matérielle.

2.6 Conclusion

Ce chapitre a été dédié à la description des systèmes qui font l'objet de ce mémoire, 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. L'approche proposée par le groupe TIMA-SLS propose des modèles de représentation intermédiaires permettant un raffinement graduel des systèmes logiciels/matériels.
Notre travail cible les méthodes et techniques de conception de systèmes. Dans ce travail, nous nous sommes concentrés sur un niveau d'abstraction supérieur dans le flot de conception des systèmes monopuces, nommé Architecture Virtuelle, qui sera décrit dans le chapitre suivant.

Chapitre 3
Niveau Architecture Virtuelle

3.1 Introduction

Les systèmes embarqués peuvent inclure plusieurs processeurs, qui exécutent des instructions spécifiques implémentées en logiciel pour des besoins de flexibilité. On estime que dans un futur proche, la complexité du code logiciel sera supérieure à celle de la partie matérielle et demandera par conséquent plusieurs hommes-années de durée de conception. Le logiciel ne pourra donc plus être développé en langage assembleur et une approche de conception à un niveau d'abstraction plus élevé est requise.

Dans ce chapitre, nous décrivons le niveau d'abstraction intermédiaire appelé Architecture Virtuelle sur le quel nous avons travaillé.
Pour aborder la conception du système dans une seule et même approche cohérente, les composants aussi bien logiciels que matériels sont modélisés avec un modèle unique. La méthodologie proposée dans le cadre de ce mémoire pour présenter un modèle de simulation du logiciel embarqué à un haut niveau d'abstraction sera alors exposée. Elle se base sur le niveau TLM.
Nous rappelons dans un premier lieu les principes de base de la méthodologie TLM pour le matériel. Dans un deuxième lieu, nous introduisons les concepts du niveau SW TLM; niveau TLM pour modéliser le logiciel embarqué.

3.2 Définition du niveau Architecture Virtuelle

Le niveau Architecture Virtuelle est appelé encore niveau système d'exploitation.
Le rôle de ce niveau est de palier aux imperfections et limites de 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 » sur l'ensemble des tâches logicielles actives à un instant donné.
Le modèle de simulation à ce niveau peut être vu comme suit (Figure 3.1):

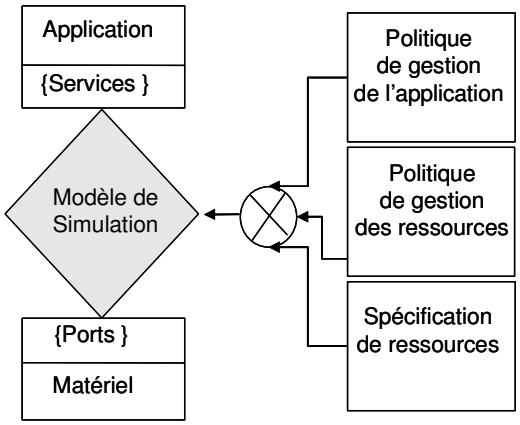


Figure 3.1 – Modèle de simulation au niveau V.A

Ce niveau d'abstraction décrit l'architecture du système mais en ne se préoccupant que des objets fonctionnels pour l'application du système, c'est à dire en excluant complètement tous les détails liés à la réalisation. Le système est réalisé sous la forme de composants pouvant implémenter un ensemble de tâches logicielles ou une fonction matérielle mais sans aucune caractéristique précise pour le type du composant ou de sa structure interne.
La description du système au niveau Architecture Virtuelle est un ensemble de tels composants travaillant concurremment qui communiquent par des canaux de communication abstraits. Ces canaux seront détaillés plus tard.
Les canaux de communication utilisent des primitives transactionnelles définies par la norme TLM [5], pour représenter seulement le transfert ou le processus de synchronisation de données entre les composants sans aucune information sur l'implémentation du protocole de communication.
Le modèle du système au niveau V.A –niveau transactionnel- est toutefois exploitable en utilisant des outils et des méthodes pour l'analyse de performances. Ils explorent les différentes solutions possibles pour le partitionnement des tâches afin de définir une architecture optimale du système.

La Figure 3.2 montre les différents niveaux d'abstraction des deux parties matérielle et logicielle. Les lignes interrompues joignant un niveau d'abstraction donné du matériel avec un autre niveau du logiciel définit des niveaux d'intégration possibles permettant la conception et la simulation des systèmes logiciels/matériels.

En toute rigueur, 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'un côté comme de l'autre dans la figure [3].
Dans ce travail, nous nous intéressons par un nouveau niveau intermédiaire d'intégration Architecture Virtuelle (en anglais *Virtual Architecture V.A*). Il associe *HW TLM* à un niveau équivalent pour le logiciel qu'on appelle *SW TLM*.

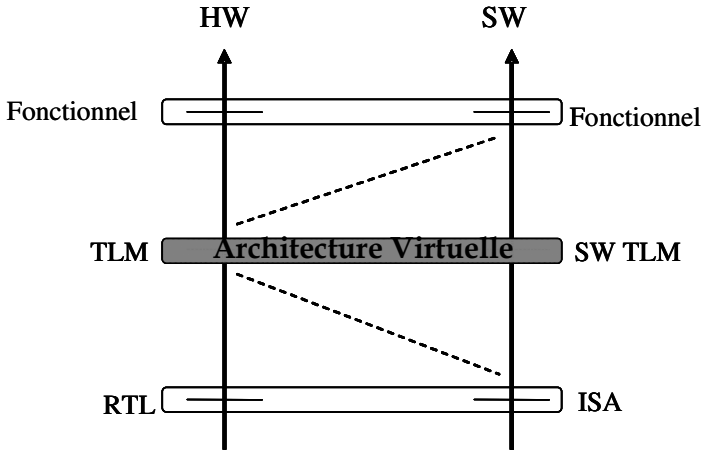


Figure 3.2 – Niveau Architecture Virtuelle

Le niveau SW TLM correspond à une abstraction du niveau bas classique pour le logiciel (ISA). Au niveau SW TLM le logiciel est décrit comme un ensemble d'objets communicants ensemble et qui sont gérés par un environnement d'exécution correspondant au système d'exploitation.
La définition du niveau SW TLM est largement inspirée des recherches récentes sur la conception du logiciel embarqué. En effet, beaucoup de recherches se sont concentrées sur l'abstraction du niveau classique de modélisation RTL utilisé en tant que modèle d'intégration logiciel/matériel. La plupart de ces travaux ont adressé le côté matériel ou le côté logiciel du problème, mais aucun d'eux n'a fourni un modèle flexible et unifié de la plateforme logicielle/matérielle à un niveau d'abstraction plus élevé.
Du côté matériel, le niveau de modélisation transactionnel (TLM) a été identifié en tant que candidat approprié pour l'abstraction du niveau RTL. Dans tous ces travaux, le logiciel est considéré à un bas niveau d'abstraction ou simplement au niveau fonctionnel.

Du côté logiciel, beaucoup de travaux [4] [9] s'étaient concentrés sur la génération automatique des systèmes d'exploitation temps réel et du code pour le logiciel embarqué. Cependant, dans les travaux [4] et [9], l'interaction du modèle de simulation du système d'exploitation avec le matériel n'est pas clairement expliquée. De même dans [27], un raffinement logiciel/matériel a été proposé cependant ce travail se base sur un modèle fixe de l'interface matérielle.

La principale contribution de ce travail est de formaliser ces efforts, en utilisant le niveau TLM pour définir une plateforme de modélisation TLM pour le matériel et le logiciel. Ceci permet le développement d'un modèle unifié pour l'interface logicielle/matérielle pour faire face aux discontinuités de conception entre le matériel et le logiciel et permettre l'exploration rapide et efficace de l'espace des solutions architecturales.

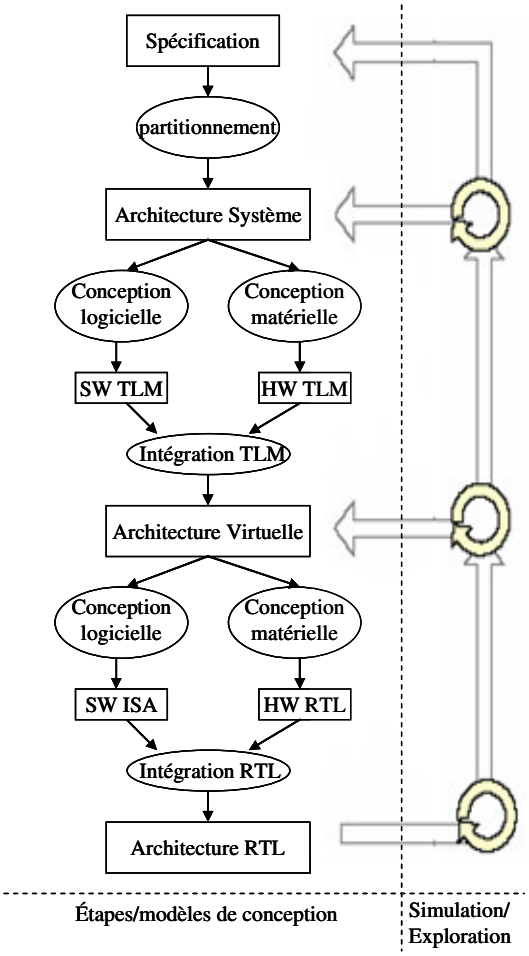


Figure 3.3 – Flot de conception typique comportant le niveau V.A

La Figure 3.3 présente un flot de conception typique comportant le niveau V.A. Comme les flots de conception classiques, le flot proposé part d'une spécification (non exécutable) qui subit une première étape de partitionnement permettant le raffinement du logiciel et du matériel.
Dans la figure, le résultat de cette étape s'appelle «Architecture Système». Ceci correspond à une forme exécutable de la spécification (en utilisant SystemC par exemple) où des annotations sont simplement introduites pour distinguer les parties de l'application qui vont être mappées en matériel ou en logiciel respectivement. Le résultat final du flot est une architecture RTL qui peut servir comme entrée des outils conventionnels de synthèse physique.
Cependant contrairement aux flots de conception classiques, le flot proposé présente une étape intermédiaire de conception basée sur le concept V.A. L'architecture virtuelle résulte de l'intégration des parties logicielles et matérielles raffinées jusqu'au niveau TLM.
Au niveau V.A, le logiciel est modélisé au niveau OS comme un ensemble d'objets SW TLM qui coexistent et interagissent avec le reste des composants HW TLM.
Cette étape intermédiaire du flot permet de:

- palier aux discontinuités des flots de conception classiques et remédier aux problèmes liés à l'intégration tardive des architectures logicielles et matérielles d'un système MPSoC en proposant un niveau intermédiaire pour l'intégration logicielle/matérielle permettant une conception logicielle/matérielle graduelle;
- rompre la longue boucle d'exploration qui sépare classiquement le niveau système du niveau RTL final. Ceci facilite une exploration d'architecture rapide et efficace bénéficiant de la rapidité de simulation du TLM comparé à RTL.

3.3 Vue d'ensemble d'un système au niveau V.A

La Figure 3.4 donne une vue d'ensemble d'un modèle conceptuel de l'interface logicielle/matérielle au niveau d'abstraction intermédiaire V.A. Les parties grises de la figure correspondent aux objets conventionnels du HW TLM.
Dans cette figure, l'exemple de conception est construit autour d'une architecture hiérarchique de bus composée d'un bus système au quel est connecté un bus CPU local via un pont (en anglais *bridge*).

A la différence de la conception TLM conventionnelle, le logiciel n'est ni exécuté sur un simulateur de jeux d'instructions, ni entièrement abstrait au niveau fonctionnel. Au niveau V.A le logiciel est modélisé au niveau OS comme un ensemble d'objets qui co-existent et interagissent avec le reste des composants HW TLM.

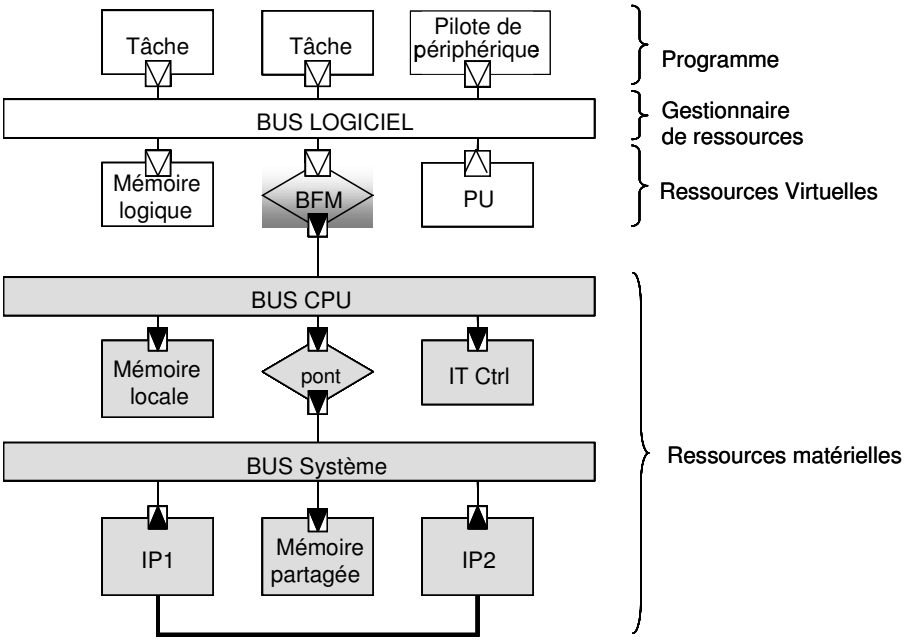


Figure 3.4 – Modèle conceptuel de l'Architecture Virtuelle

Dans une description *SW TLM*, nous identifions principalement trois couches conceptuelles :

- La couche Programme (*Program layer*) qui correspond au logiciel conçu par des programmeurs de logiciel. Ceci se compose des tâches de l'application ainsi que des pilotes de périphériques qui permettent la communication extérieure avec le matériel.

- La couche de gestion des ressources (*resource management layer*) correspond à ce qu'on appelle le bus logiciel (*SW bus*). Cette entité abstrait le vrai système d'exploitation et permet la coordination et l'arbitrage des composants. Cette couche gère de même la communication entre les différents composants logiciels.
  La communication logicielle est abstraite au niveau service. En effet, la communication est représentée comme une combinaison de requêtes et de services. Les différents modules communiquent par des requêtes de services,

via le bus logiciel qui garantit le routage et la synchronisation des connexions établies.
Les services offerts par le bus logiciel dépendent largement du système d'exploitation. On peut 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 des ressources.

• La couche des ressources virtuelles (*virtual resource layer*) qui spécifie, d'un point de vue programmeur, quel type de ressources disponibles dans le sous système logiciel (le nœud logiciel de la Figure 2.1).
Cette couche fournit une abstraction de l'ensemble des ressources disponibles au niveau du nœud logiciel. En effet, l'application dispose d'un certain nombre de tâches qui vont être exécutées sur différentes ressources de calcul (*Processing Elements*), elles ont besoin pour cela d'allouer les ressources de stockage nécessaires à leur exécution. Les mémoires sont utiles pour la mémorisation des données ou des instructions, et sont aussi un passage fréquent pour les communications.
Dans notre cas, nous distinguons deux types de ressources virtuelles : mémoire logique et unité virtuelle de traitement (*Processing Unit*).

Un objet important qui pourrait être qualifié comme un objet TLM hybride logiciel/matériel est le modèle fonctionnel du bus (en anglais *BFM: Bus Functional Model*). Un BFM est un pont spécial qui permet de relier le bus logiciel avec le bus matériel (le bus CPU). Son rôle principal est de transmettre les accès externes du logiciel au matériel. Il est également responsable de transférer les interruptions matérielles venant du côté matériel aux composants appropriés du côté logiciel.

Cette description a nécessité une bonne compréhension du niveau transactionnel d'un point de vue théorique.
Dans les sections suivantes, nous rappelons donc les concepts de base pour le HW TLM ensuite ceux pour le SW TLM sont décrits et leur application pour le raffinement du logiciel est expliquée.

3.4 Concepts de base du TLM pour le matériel

Dans cette section, la méthodologie conventionnelle HW TLM est présentée.
Nous cherchons à donner de même un aperçu sur la structure TLM et nous décrivons brièvement les niveaux TLM.

3.4.1 Méthodologie TLM

TLM est un niveau plus abstrait que RTL. Ceci est dû à la réduction de la quantité de détails que le concepteur doit manipuler facilitant donc la modélisation.
Ce niveau, moins détaillé que le niveau RTL, représente uniquement ce qui se passe au niveau système, en terme d'échange de données et de synchronisation système, sans se soucier de la micro-architecture des blocs.
TLM est décrit et expliqué par beaucoup de travaux [6] [7] [10]. Il est construit comme un niveau élevé d'API qui définit comment les composants matériels communiquent entre eux.
Un modèle TLM se base uniquement sur des appels de fonctions et des transferts de paquets de données. L'idée est de représenter au plus près l'intention du concepteur quant au comportement global du circuit, sans rentrer dans les détails de la description des signaux réalisés au niveau RTL.
L'objectif de ce niveau est de développer du logiciel embarqué et de faire des études d'architectures à un haut niveau d'abstraction. Il permet de même d'accélérer le temps de simulation.
L'API OSCI TLM est construite comme un ensemble d'interfaces qui définissent comment les modules communiquent entre eux.
En effet, l'interface de protocole définit la sémantique pour transférer une transaction entre deux points différents d'un même système tel que *tac_if*, *basic_if*, *synchro_if*, etc.
TLM définit un ensemble d'interfaces génériques et réutilisables (bloquantes/non bloquantes, unidirectionnelles/bidirectionnelles) par une approche en couches tel que *tlm_transport_if<req,resp>*, *tlm_put_if<req>*, *tlm_get_if<req>*, etc. (voir Figure 3.5):

- Couche Utilisateur (*User layer*) :

Dans le jargon TLM, cette interface s'appelle en anglais « *convenience interface* ». Elle se compose typiquement de méthodes qui donnent un sens aux

utilisateurs du protocole en question par exemple *read*, *write*, *burst read* et *burst write*. L'utilisateur va utiliser les ports initiateurs qui fournissent les moyens d'implémenter ces interfaces et définit des modules cibles (*target*) qui héritent de ces interfaces [6]. Les transactions sont envoyées par le module initiateur par le biais de *initiator_port* et sont reçues et traitées par le module target selon l'implémentation de l'utilisateur.

Ces implémentations sont visibles au module target grâce à *sc_export* de SystemC qui est reliée au module target et qui donne accès à ces implémentations.

A ce niveau, la couche protocole est transparente pour l'utilisateur.

- Couche Protocole (*Protocol layer*) :

La couche protocole se compose de [6] :

- classes de requête et de réponse qui encapsulent le protocole. Principalement ceci correspond à la définition d'échange des transactions (l'information à échanger entre l'initiateur et le target: adresse, données, statut, longueur, etc.);
- port initiateur qui hérite de *sc_port* ;
- classe *slave_base* qui implémente les interfaces TLM.

Le module target doit hériter de la classe *slave_base*.

L'utilisateur pourra après utiliser les classes *initiator_port* et *slave_base*. Il faut noter que ce mécanisme de communication est efficace de point de vue temps d'exécution. En effet grâce à la liaison «*port-to-export*» (introduite par SystemC-2.1.v1) l'appel à l'API du protocole résulte à l'exécution de l'implémentation dans le côté du target mais dans le même contexte du *thread*[13] initiateur. Ainsi, la communication entre un maître et un esclave n'implique pas un changement de contexte (*context switch*) (coûteux en temps de simulation).

- Couche Transport (*Transport layer*):

La couche Transport forme la couche de base pour la couche protocole et la couche utilisateur. Elle donne l'accès aux interfaces TLM virtuelles par la liaison SystemC *sc_port* à *sc_export*, à savoir *initiator_port* à *target_port*.

L'interface TLM hérite de la classe *sc_interface* de SystemC. Elle sert comme une base commune pour faciliter l'interopérabilité de divers modèles TLM définis par différentes compagnies.

[13] Processus léger

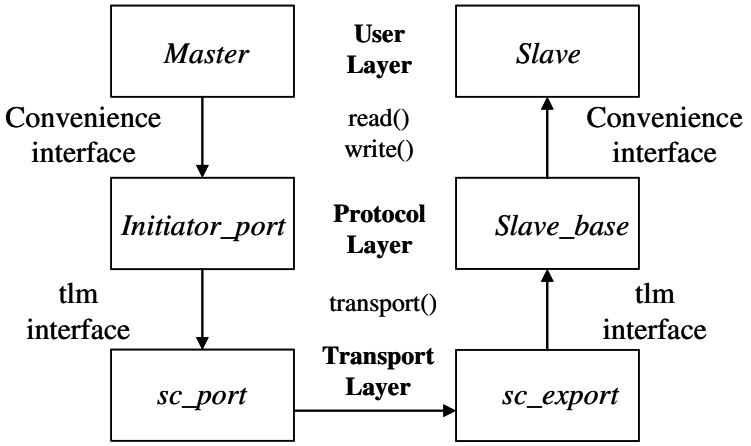


Figure 3.5 – Architecture TLM [6]

La société ST-Microelectronics; qui a adopté le niveau TLM dans la conception; a construit un ensemble de protocoles qui sont conformes à OSCI TLM à savoir les protocoles TAC et Synchro.

TAC acronyme de *Transaction Accurate Communication* est construit sur la base du standard TLM OSCI. Il se base sur l'interface *tlm_transport_if* (interface bloquante bidirectionnelle) qui comporte la requête et la réponse dans un même transfert TAC, *status* fait aussi partie d'une réponse TAC.

Le protocole Synchro est construit sur la base de l'interface *put* de TLM et représente la synchronisation entre plusieurs composants. Il se base sur l'interface *tlm_blocking_put_if* (interface bloquante unidirectionnelle).

3.4.2 Les niveaux TLM PV et PVT

Il existe de nombreuses variantes dans le niveau d'abstraction TLM. Néanmoins, nous pouvons distinguer deux grands types de modèles TLM (voir Figure 3.6) tels que définis dans la littérature.

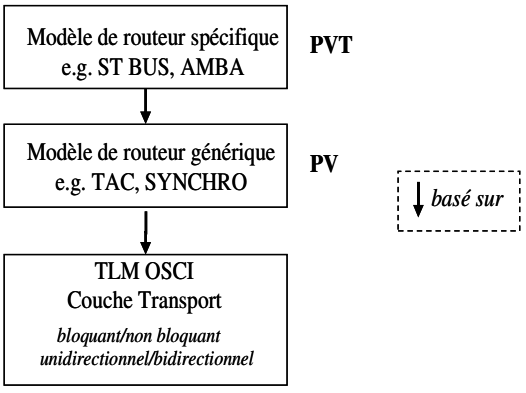


Figure 3.6 – Les couches TLM

Le premier, appelé *« Programmer View »* (PV), est une représentation purement fonctionnelle du circuit sans référence à aucune notion temporelle. A ce niveau, le modèle contient toutes les informations nécessaires (et pas plus) pour que les équipes de développement logiciel puissent travailler, c'est-à-dire faire tourner le logiciel embarqué final, système d'exploitation compris. Le protocole utilisé à ce niveau est générique (tel est l'exemple de TAC) et la synchronisation reflète la dépendance causale entre les différentes unités de calcul et n'est pas basée sur les contraintes de temps.
Le second type, appelé *« Programmer View + Timing »* (PVT), intègre des informations sur les délais (*timing*) qui permettent notamment de travailler sur l'analyse des performances du circuit, sans trop pénaliser les temps de simulation.
Une plateforme au niveau PVT est une plateforme au niveau PV avec son inter connecteur non temporisé, à qui on a ajouté un modèle temporisé du bus qui correspond à un bus spécifique (par exemple *STBus, AMBA*).
Dans une plateforme PVT, on a les différents composants PV individuels (*PV IP*[14]) et leurs modules temporisés correspondants; un ou plusieurs routeurs non temporisés et leurs correspondants temporisés, par exemple *TAC router* et *STbus router* en se référant à la plateforme PVT de ST. Dans une plateforme PVT, l'annotation du temps est effectuée dans le module temporisé de chaque composant et dans le routeur temporisé qui simule les délais de transfert pour chaque transaction.

TLM rapproche l'écart entre les modèles fonctionnels de spécifications et les implémentations RTL par une amélioration progressive de l'infrastructure de communication matérielle.
Du côté logiciel, peu de recherches ont abordé la communication logicielle à un niveau conforme à TLM. Le flot de développement logiciel passe brusquement d'un modèle fonctionnel au niveau d'abstraction le plus bas.
D'où le but des chapitres suivants est d'introduire le nouveau concept TLM destiné pour le développement logiciel. Notre méthodologie raffine le logiciel embarqué au niveau Architecture Virtuelle.

[14] *Intellectual Property*

3.5 Concepts de base du TLM pour le logiciel

Dans cette section, nous présentons l'environnement de conception dans lequel nous avons développé et validé notre méthodologie TLM. Nous allons tout d'abord exposer les concepts de base. Ensuite, nous donnerons une brève présentation de la constitution du niveau SW TLM.

3.5.1 Description des composants SW TLM

La Figure 3.7 illustre les différents composants SW TLM.
Dans une plateforme SW TLM il y a des modules qui requièrent des services (initiateurs en anglais *initiator*), d'autres qui fournissent ces services (cibles en anglais *target*). Ils communiquent en envoyant des requêtes et réponses de part et d'autre. Ces modules sont des composants logiciels qui peuvent être classés comme suit :

- composants de l'application (par exemple tâche applicative);
- ressources abstraites;
- pilotes de périphériques;
- bus logiciel.

Les tâches logicielles peuvent communiquer entre elles ou avec des tâches matérielles. Elles appellent les services du système d'exploitation et les services de communication et peuvent être des modules maîtres ou esclaves.

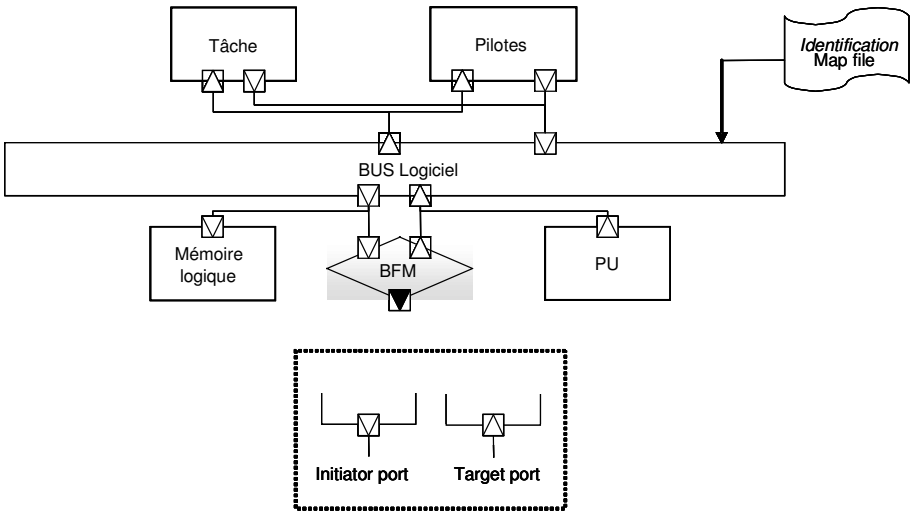


Figure 3.7 – Les composants SW TLM

Les ressources abstraites se composent des mémoires logiques et des unités de traitement (en anglais *Processing units*).
Dans le processus de raffinement de la communication, les zones de stockage matérielles apparaissent dans l'architecture matérielle. Ces zones sont partagées et permettent l'échange de données. Elles pourraient être mappées aux mémoires matérielles locales ou externes.
Au niveau application, le programmeur n'est pas censé savoir le mapping matériel. C'est pourquoi, il adresse habituellement la mémoire logique. Le processus de raffinement est alors responsable de mapper ces adresses logiques une fois que l'architecture est fixe.
Les unités de traitement sont des composants virtuels responsables de l'exécution du logiciel embarqué. Ils appellent le service d'initialisation (*BOOT service*).
Les pilotes de périphériques permettent aux composants de l'application de réagir avec les périphériques matériels. Les pilotes TLM adressent une ou plusieurs mémoires logiques qui connaissent le mapping de la mémoire physique. Ils ont besoin également des services du système d'exploitation pour accéder aux périphériques matériels correspondants.
Le bus logiciel est le conducteur de tout le nœud logiciel. Il pourrait être défini comme étant le chemin logique qui sert des tâches logicielles multiples ou des unités logicielles de calcul et de communication à travers un modèle d'OS. Son rôle principal est:

- assurer l'ordonnancement des tâches et le partage du temps;
- intercepter les transactions logiques et les traiter;
- acheminer ces transactions logiques au *BFM*.

Le bus logiciel a deux mécanismes importants à savoir routage et arbitrage. Le premier est responsable d'acheminer les transactions aux différents modules logiciels, tandis que le deuxième résout les requêtes concurrentes de services.
Dans la section suivante, nous allons décrire la structure du niveau SW TLM.

3.5.2 Structure du SW TLM

Comme dans le HW TLM, nous définissons un mécanisme de passage de transactions dans l'architecture de communication logicielle. Cette architecture serait structurée autour de différentes couches (voir la Figure 3.8):

• Couche Application « *Application layer* »:

Cette couche peut être définie comme la couche de calcul. En effet, elle se compose des tâches de l'application. À ce niveau, nous supposons que le programmeur n'a aucune idée sur l'architecture de communication. Au niveau de cette couche, l'application requiert des services d'OS et des services de communication.

• Couche de pilote « *communication driver layer* »:

Cette couche inclut les pilotes TLM et les mémoires logiques. Ces deux genres d'éléments SW TLM coopèrent – par la communication logique avec la couche du bus logiciel – pour contrôler le processus de communication. En fait, chaque pilote TLM adresserait une ou plusieurs mémoires logiques.

Cette couche est également responsable de répondre aux demandes d'interruptions destinées à la couche application.

• Couche du bus logiciel « *SW BUS layer*»:

Elle est le conducteur de tout le nœud logiciel. Elle abstrait le système d'exploitation.

Généralement un système d'exploitation peut être vu comme un ensemble de couches de services: une couche API, une couche des services de base du système d'exploitation et une couche d'abstraction du matériel.

• Couche du bus matériel « *HW BUS layer*»:

C'est le bus physique, le réseau de communication matérielle. Il peut être un bus CPU local ou un bus système.

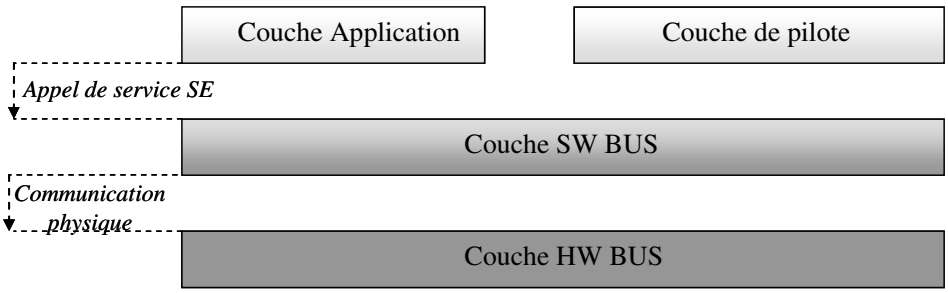


Figure 3.8 – Architecture en couches de la communication logicielle

3.6 Conclusion

Dans ce chapitre, nous avons défini le niveau Architecture Virtuelle, un niveau qui abstrait le système d'exploitation ainsi que l'architecture matérielle.

Puis, dans un deuxième temps, nous avons présenté la méthodologie SW TLM conçue pour unifier la modélisation du logiciel embarqué au niveau V.A.

Chapitre 4
Implémentation (SW TLM)

4.1 Introduction

Cette partie du rapport expose l'implémentation du SW TLM. En premier lieu, elle présente la hiérarchie du SW TLM. En second lieu, elle décrit ses différentes interfaces de base.

4.2 Choix du langage

Le choix de travailler avec un langage de conception au niveau système (*SLDL : System Level Design Language*) (tel que SystemC ou SpecC) est important pour assurer la portabilité du modèle.
La description des fonctionnalités du modèle doit être donc faite dans un langage de description de haut niveau. Ceci permet de valider rapidement le modèle et de profiter d'un environnement de simulation efficace. Pour cela nous avons utilisé la bibliothèque SystemC.
Elle permet de profiter des mécanismes d'héritage ou de polymorphisme du C++ pour décrire des ensembles hiérarchiques [18]. Par ailleurs, cette solution offre la possibilité de simuler conjointement des parties logicielles et matérielles. Ceci se révèle très utile dans notre cas puisque le système complet est composé d'éléments hétérogènes logiciels et matériels.
Plus particulièrement, SystemC est considéré d'un point de vue industriel un standard pour la modélisation TLM et la conception au niveau système et à la co-simulation des systèmes logiciels/matériels.
En bref, SystemC est un langage et un noyau de simulation basés sur C++ qui permet la représentation des composants logiciels et matériels et des communications à différents niveaux d'abstraction. Il permet la modélisation et la simulation de systèmes logiciels/matériels globalement synchrones, ou asynchrones avec un modèle à événements. Il convient pour l'augmentation de la complexité de conception des systèmes, en fournissant un modèle exécutable tôt dans le cycle de conception.

Un modèle SystemC est composé des éléments suivants :

- Modules: un module (*sc_module*) est l'élément de base SystemC qui permet d'encapsuler une description matérielle. Les modules communiquent avec d'autres modules à travers des ports. D'une manière générale, un module peut contenir un ou plusieurs processus implémentant le comportement de celui-ci.
- Processus: les processus sont utilisés pour décrire le comportement d'un composant. Ils s'exécutent de manière concurrente dans l'environnement SystemC.
- Ports: un port (*sc_port*) est le moyen utilisé en SystemC pour permettre à un module d'accéder à l'environnement extérieur. Les ports représentent les points d'entrées/sorties des modules.
- Interfaces: une interface (*sc_interface*) permet de déclarer une méthode qui sera implémentée par un canal (ou un module à partir de SystemC 2.1) et qui sera accessible via un port.
- Canaux: en général, les canaux SystemC (*sc_channel*) sont utilisés pour implémenter le comportement d'une fonction déclarée par une interface.
- Export: disponible à partir de la version 2.1 de SystemC, un export (*sc_export*) permet de rendre accessible une interface implémentée par un module.

4.3 Implémentation du TLM pour le logiciel (SW TLM)

4.3.1 Hiérarchie du SW TLM

Comme HW TLM, SW TLM se divise en différentes couches comme le montre la Figure 4.1:

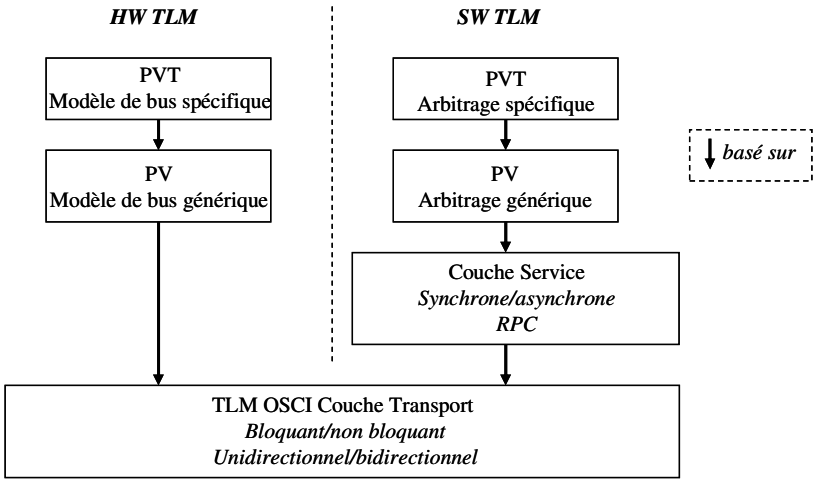


Figure 4.1 – Les couches du SW TLM

Au niveau service, les interfaces des modules sont composées de ports d'accès au bus logiciel. Ces ports fournissent des services de type synchrone ou asynchrone et les opérations sur les ports sont des requêtes et des services. Les tâches élémentaires sont des processus qui interagissent avec l'environnement via des requêtes et des services.

La couche de service (*Service layer*) est construite comme un ensemble d'interfaces qui définissent comment les modules communiquent. En effet, l'interface protocole définit la sémantique de transfert d'un service entre deux modules différents.

Les interfaces SW TLM (synchrones/asynchrones) spécifient les services de communication et sont basées sur la couche transport du TLM OSCI.

Les transactions synchrones se font séquentiellement, chaque transaction devant être terminée avant que la prochaine ne s'exécute.

En mode asynchrone la main est immédiatement rendue à l'initiateur, ainsi les appels normalement bloquants sont traités parallèlement.

Toutes les interfaces héritent du *sc_interface*. Quand un service passe de l'initiateur au target il est appelé « service_requis » et quand il passe du target à l'initiateur il est appelé « service_fourni ».

SW TLM PV est construit en se basant sur SW TLM plus particulièrement sur la couche service. Au niveau PV, nous n'avons aucune vraie notion sur le temps et l'arbitrage des transactions est générique. En effet, l'arbitrage du bus est le mécanisme qui alloue le contrôle du bus aux dispositifs qui le demandent, évitant tout conflit.

PVT est un modèle qui inclut le niveau PV et une spécification de l'arbitrage. PVT ajoute des informations de temps sur chaque traitement ou transfert de données. Pour le bus logiciel le timing doit tenir compte du nombre de transferts ainsi que l'arbitrage entre les différents composants logiciels.

Après avoir mis en évidence la hiérarchie du SW TLM, nous introduisons ses différentes interfaces de base ci après.

4.3.2 Les interfaces de base

SW TLM est construit en se basant sur les interfaces *tlm_blocking_put_if* et *tlm_blocking_get_if* du TLM OSCI.

Put et *Get* sont utilisés pour assurer les transferts de données mais dans notre cas nous les adoptons pour supporter les requêtes et réponses de services.

SW TLM définit la couche Service qui implémente le protocole *RPC* (*Remote Procedure Call*).

Le modèle RPC est un modèle de communication par invocation à distance se basant sur l'appel des services distants. Nous pouvons trouver deux types de RPC: synchrone et asynchrone.

- Modèle RPC synchrone: dans ce modèle l'initiateur est bloqué en attente d'une réponse du target. Ce modèle est facile à comprendre. De plus, il permet la détection des erreurs facilement, d'autant plus qu'il n'est pas nécessaire de stocker l'information.
- Modèle RPC asynchrone: dans ce modèle l'initiateur n'est pas bloqué, mais il existe un test continu sur la réponse du target.

L'interface *sw_tlm_service_if* est implémentée comme le montre l'extrait de code ci-dessous :

```
//bidirectional blocking interfaces
template<typename SERVICE_CALL, typename RSP>
class sw_tlm_service_if :public virtual sc_interface
 {
   public:
     virtual RSP service (const SERVICE_CALL&) = 0;
 };
```

La classe *sat_service_call* décrit l'information envoyée par le module initiateur au module target. Cette classe définit la première partie du protocole *sw_tlm_sat*. Elle est utilisée comme un paramètre SERVICE_CALL pour la classe *sw_tlm_service_if*.

De la même manière, la classe *sat_response* décrit l'information retournée par le module target, comportant *sat_status*, au module initiateur. Cette classe définit la seconde partie du protocole *sw_tlm_sat*. Elle est utilisée comme un paramètre RSP pour la classe *sw_tlm_service_if*.
SW TLM PV est basé sur la couche Service. Il implémente le protocole *SAT* (*Service Accurate Transaction*).
L'interface du protocole *SAT*, *sat_if*, est définie avec une méthode virtuelle *CALL* et *sat_status* comme étant la valeur de retour de la fonction de l'interface du protocole *SAT*.

```
#ifndef _SAT_IF_H_
#define _SAT_IF_H_

/*--------------------
 * Includes
 *--------------------*/
#include “sat_protocol.h”

…

//-----------------------------------------------------------------
/// Class sat_if: sw_tlm_sat protocol layer convenience function definition
//-----------------------------------------------------------------
template<typename ID, typename DATA>
class sat_if {

   public:
     //-----------------------------------
     /* \brief Call access convenience API (implemented in
      *  sat_initiator_port and sat slaves).
      */

     virtual sat_status CALL (const ID& id,
                              DATA& data,
                              sat_error_reason& error_reason,
                              const unsigned int service_id = NO_SERVICE,
                              …
                              ) = 0;
     …
   };
}

#endif /* _SAT_IF_H_ */
```

La classe *sat_error_reason* est un message de caractères encapsulant la raison de l'erreur (du target à l'initiateur) en cas d'échec d'une requête.
Le paramètre *service_id* représente le service appelé, il est défini dans le fichier « sat_protocol.h » comme suit :

```
/** \defgroup service_id_values Predefined service_id values
*@{
**/
```

```
static const unsigned int NO_SERVICE = 0xffffffff;
static const unsigned int REGISTER_TASK = 0x1;
static const unsigned int MUTEX_INIT = 0x2;
static const unsigned int MUTEX_LOCK = 0x3;
static const unsigned int MUTEX_UNLOCK = 0x4;
…
/* @ */
```

La classe *sat_status* est le statut d'une transaction *SAT*. Elle définit le statut d'une requête d'un initiateur avec le protocole *sw_tlm_sat*. La valeur de statut est fixée par le target (esclave ou routeur) et utilisée par les initiateurs en cas de besoin.
La classe sat_*initiator_port* modélise le port initiateur construit en se basant sur l'interface *sw_tlm_service_if* en se basant de même sur le protocole *SAT*.
L'initiateur appelle un service du target et reçoit une valeur de retour pour indiquer si le service a été fourni ou non. Par exemple, la tâche lance un appel de service « *CALL* » comme suit :

```
void task()
{
  …
  status=initiator_port.CALL (id, data, error_reason, SERVICE, params);
  …
}
```

« params » sont des variables qui dépendent du type de service appelé par l'initiateur. Les services peuvent être lire (*read*), écrire (*write*) ou les autres services du système d'exploitation.
Deux exemples d'extraits de code applicatif sont montrés ci après:

```
void threadDemux()
{…
  sat_status status;
  sat_error_reason error_reason;
  unsigned long size = 1;
  int data;
  …
  status=initiator_port.CALL (os_id, data, error_reason, PIPE_WRITE,
                              dv_cmd, cmd, size);
  …
}
```

```
void application_init()
{
  …
  sat_status status;
  sat_error_reason error_reason;
  int length = 64;
  int size = 256;
  int irq = 2;
  int prio = 20;
  char t = '1';
  …
  status=initiator_port.CALL (os_id, length, error_reason, PIPE_INIT,
                              dv_data, base_dv_data);
  …

  status=initiator_port.CALL (base_qz_data, size, error_reason, FIFO_INIT,
                              qz_data, irq);
  …
  status=initiator_port.CALL (os_id, prio, error_reason, REGISTER_TASK,
                              threadDemux, t);
  …
}
```

A la couche basse « *SW bus layer* », le bus logiciel est implémenté comme un « *sw_ router* » qui est responsable de relier les différentes requêtes de services provenant de différents initiateurs vers les targets correspondants.
On peut distinguer deux situations :

- Si le service est un service d'OS (par exemple communication inter tâches), il sera directement fourni par le bus logiciel lui même;
- Sinon (par exemple service de communication avec l'extérieur), le bus logiciel détermine le target approprié fournissant le service demandé (exemple: pilote de périphérique) à l'initiateur appelant.

Le bus logiciel hérite de la classe *rtos_ base* [27] comme le montre le code ci-après. Par conséquent, son instanciation dans le nœud logiciel donne accès au modèle de simulation du système d'exploitation.

```
template <typename ID, typename DATA>
class sw_router :
  public rtos_base,
  public sc_module,
  public virtual sw_tlm_service_if<
                      sat_service_call<ID, DATA>,
                      sat_response<DATA>
                                  >,
  public sw_tlm_router_base<ID, sw_tlm_service_if<
                                    sat_service_call<ID, DATA>,
                                    sat_response<DATA>
                                                >,
                            0>
```

Il est à noter que la classe de base *rtos_base* fournit des fonctionnalités suffisantes et génériques nécessaires pour construire des modèles de simulation de systèmes d'exploitation. Chaque implémentation d'un modèle d'OS héritera alors ces fonctionnalités de base afin de construire ses spécificités.
Le bus logiciel hérite également du *sw_tlm_router_base* pour acheminer les services vers les différents composants logiciels.
La figure 4.2 montre la hiérarchie de classe de notre bus logiciel en utilisant une notation basée sur le formalisme UML.

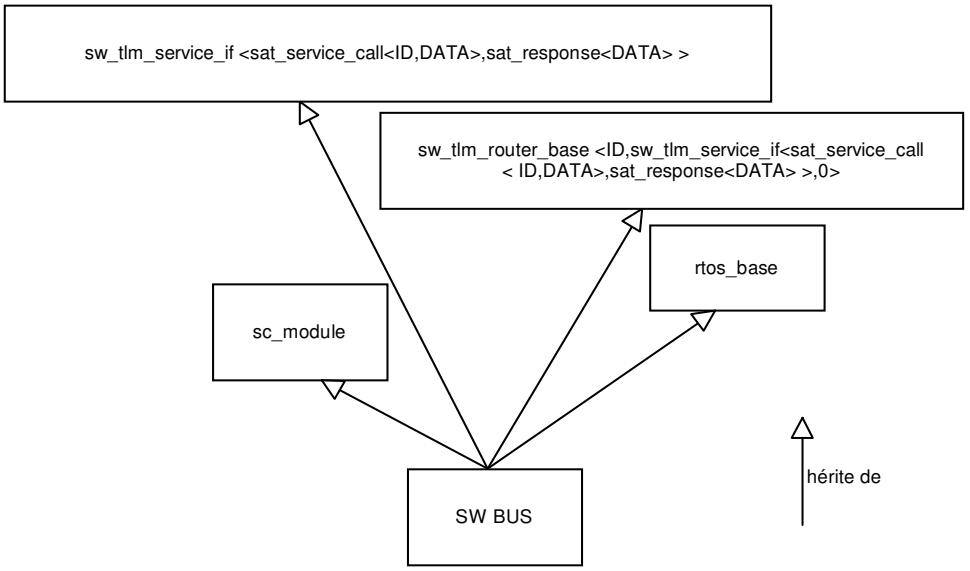


Figure 4.2 – Hiérarchie de classe du bus logiciel

Le bus logiciel est relié au bus matériel par l'intermédiaire du BFM comme suit :

- un port initiateur (*initiator_port*) qui représente le port CPU. Ce port initie des transactions SAT au bus matériel (bus CPU local ou bus système);
- un *target_port* qui représente le port d'interruption. Ce port sert à servir les interruptions provenant de l'extérieur du nœud logiciel.

Ainsi, le rôle du BFM est de traduire des transactions logiques aux transactions physiques et de les amener à la couche physique du bus matériel (*physical HW BUS layer*).
Le bus logiciel se sert d'un fichier d'identificateurs de services (*Identification map file*) permettant de rendre compte du composant logiciel fournisseur du service.

Une transaction initiée par un pilote TLM par exemple résulte à un accès *READ/WRITE* selon l'identificateur de service demandé à travers le port target correspondant de la mémoire logique (identifié grâce au *map file*).
Toutes ces méthodes et classes forment la base du SW TLM. Sur la base de ce simple mécanisme de services nous pouvons établir des modèles logiciels et des routeurs génériques.
Les interfaces du SW TLM sont facilement comprises et efficaces.
Les utilisateurs peuvent concevoir leurs propres composants logiciels mettant en application quelques ou toutes ces interfaces, ou ils peuvent les implémenter directement dans le target en utilisant *sc_export*. La fonction service en particulier sera souvent directement implémentée dans le target.

4.4 Conclusion

Ce chapitre a décrit la structure et les interfaces du SW TLM. Ce niveau est facilement compris et utilisable. Cependant, pour pouvoir tester l'efficacité de ce modèle, nous l'avons appliqué sur un exemple de système multiprocesseur à savoir l'application MJPEG.

Chapitre 5
Etude de cas: Application du SW TLM sur l'application MJPEG

5.1 Introduction

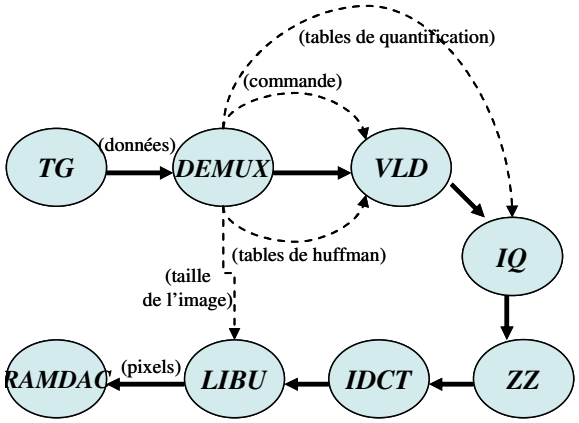
Ce chapitre présente l'étude de cas réalisée pour illustrer l'utilisation du SW TLM défini tout au long de ce travail.
La structure de ce chapitre se décompose en trois sections. La première section décrit l'application sur laquelle a été appliqué SW TLM à savoir l'application MJPEG. Dans la deuxième section, nous détaillons l'utilisation du SW TLM pour la modélisation du logiciel embarqué au niveau V.A. Enfin la dernière section évalue les résultats obtenus.
Nous cherchons à montrer que le modèle adopté permet une modélisation rapide et qu'il offre de la flexibilité.

5.2 Description de l'application MJPEG

5.2.1 L'application MJPEG

La Figure 5.1 montre un graphe de tâches de l'application MJPEG qui est un décodeur d'images vidéo JPEG[15]. C'est en fait une application logicielle multi-thread réalisant le décodage d'un flux d'images JPEG.
Cette application est modélisée comme un ensemble de tâches parallèles communiquant à travers des canaux point à point de type FIFO.
Les arcs en gras représentent le flux de décompression, et les arcs en pointillés représentent les paramètres de configuration qui sont des variables globales de la configuration initiale [11].

[15] *Joint Photographic Experts Group*



TG: Traffic Generator
DEMUX: Demuxer
VLD: Variable Length Decoding
IDCT: Inverse Discrete Cosine Transform
IQ: Inverse Quantization
ZZ: ZigZag scan
LIBU: LINE BUILDER
RAMDAC: Random Access Memory Digital-to-Analog Converter

Figure 5.1 – Graphe de tâches de l'application MJPEG

Le périphérique d'entrée est un générateur de trafic noté TG, et le périphérique de sortie est un convertisseur vidéo noté RAMDAC.
Cette application lit un flux d'images JPEG 64x64 et produit un flux de pixels dans l'ordre des lignes pour affichage.
Le décodage d'une image nécessite 6 étapes [11]:
- DEMUX analyse le flux fournit par le périphérique d'entrée pour en extraire la taille de l'image, les tables de Huffman et les tables de quantification. Ces informations sont usuellement stockées dans des variables globales. Les données de l'image compressée sont lues par paquet et rangées dans un tampon.
- Le décodeur de Huffman, VLD, décompresse ce tampon et met le résultat dans un deuxième tampon.
- ZZ réorganise le tampon suivant l'ordre zigzag et produit son résultat dans un troisième tampon.
- IQ effectue la quantification inverse du tampon précédent pour le mettre dans un nouveau tampon.
- IDCT exécute une transformé discrète inverse en cosinus du tampon fournit par IQ dans un nouveau tampon.
- LIBU stocke les blocs (8x8) issus de IQ dans un sixième tampon dont la largeur en bit correspond à la taille de l'image. Une fois les lignes disponibles, elles sont émises vers le périphérique de sortie.

5.2.2 Partitionnement logiciel/matériel

Nous avons procédé à un partitionnement logiciel/matériel de l'application MJPEG. Elle sera alors constituée de deux noeuds logiciels et deux nœuds matériels à savoir le générateur de trafic TG, qui écrit le flux MJPEG compressé en mémoire, et le coprocesseur RAMDAC qui lit les images décompressées en mémoire et les envoie vers le terminal VIDEO. Les deux nœuds logiciels comportent arbitrairement chacun trois tâches logicielles.

Modélisé au niveau Architecture Virtuelle, le système se présente comme suit:

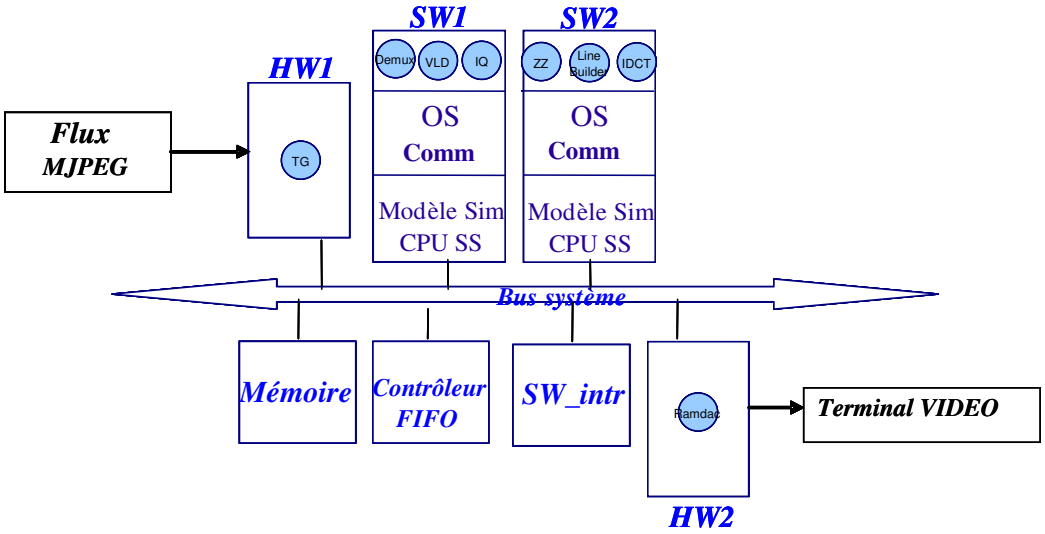


Figure 5.2 – Partitionnement logiciel/matériel de l'application MJPEG

Avec la méthodologie SW TLM, nous avons procédé à la co-simulation logicielle/matérielle de l'application MJPEG.

Nous limitons notre étude au niveau PV.

Les objectifs à atteindre, lors de l'utilisation de la méthodologie SW TLM, sont multiples:

- utiliser la méthodologie dans le but de montrer sa validité;
- montrer la flexibilité et la rapidité de la méthodologie;
- tester le bon fonctionnement des divers composants.

5.3 Architecture de l'application MJPEG au niveau V.A

Dans notre modèle, nous disposons de deux blocs matériels : TG et RAMDAC, deux modules logiciels mappés à deux processeurs ARM7 (chaque nœud logiciel se compose de trois tâches), une mémoire, un contrôleur de FIFO et un module « *Interrupt SW* » qui gère les interruptions entre les modules logiciels.

Ces composants sont connectés par l'intermédiaire du bus système (voir Figure 5.2).

En utilisant la présentation TLM et en détaillant l'architecture du sous système CPU, le modèle sera présenté par la Figure 5.3.
Dans notre cas, nous avons seulement besoin de pilotes de FIFO pour la communication. Le pilote FIFO est un composant esclave/maître. Dans ce cas, il fournit une simple API à l'application plus spécifiquement les services *Read* et *Write* :

- *w_fifo_drv* : contrôle l'accès d'écriture dans une FIFO;
- *r_fifo_drv* : utilisé quand l'application logicielle procède à un accès en lecture.

Pour synchroniser les tâches logicielles des différents nœuds pour des accès en lecture et écriture bloquants, un contrôleur de FIFO intervient pour débloquer la tâche bloquée. Par suite, si une tâche logicielle est bloquée dans un accès parce que la FIFO est vide ou pleine, elle doit attendre une interruption matérielle pour pouvoir accéder à la FIFO. Cette interruption est capturée par le port *interrupt_port*.
D'une part, *w_fifo_drv* doit envoyer une interruption quand la FIFO ou il va écrire est initialement pleine. Cela va réveiller les lectures bloquées. Pour assurer ceci, *w_fifo_drv* écrit dans une adresse particulière du contrôleur de FIFO.
En outre, les pilotes de FIFO tiennent des informations sur l'état de FIFO modélisée comme une FIFO circulaire, à savoir :

- *read_index*: pointeur de lecture;
- *write_index*: pointeur d'écriture;
- *full*: indique si la FIFO est pleine;
- *buffer*: pointe à la base de la FIFO.

En fait, ces informations sont présentées comme étant des adresses dans l'espace d'adressage de la mémoire logique.

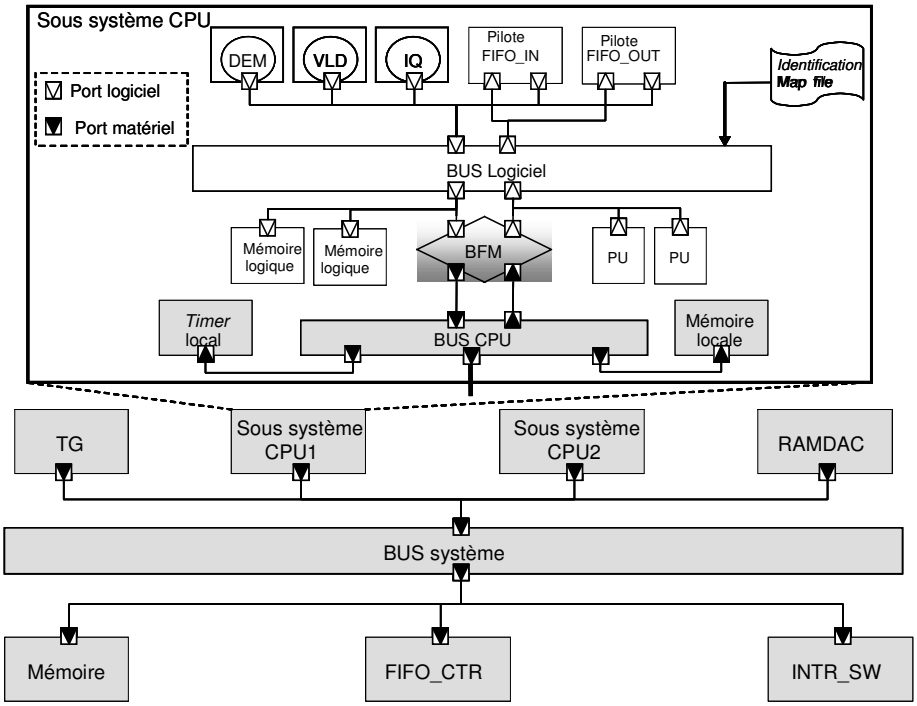


Figure 5.3 – Modèle MJPEG au niveau Architecture Virtuelle

Le code suivant illustre une partie de l'implémentation d'un accès en écriture par *w_ fifo_ drv*:

```
sat_status FIFO_DRIVER::w_fifo_drv(drv_fifo_t * drv, void *buf, unsigned
long size)
{
sat_status status;
sat_error_reason error_reason;
int full, data, temp_wi;

…
//waiting if fifo full
status= initiator_port.CALL(drv->full, full, error_reason, READ);

if (full) {
           …
          status= initiator_port.CALL(os_id, data, error_reason,
                                      SIGNAL_SLEEP, drv->sig_r);
          …
          }

//prepare and send write transaction
status=initiator_port.CALL(drv->wi, temp_wi, error_reason, READ);
status=initiator_port.CALL(drv->buffer+temp_wi, data, error_reason,
                           WRITE);
…
```

Le routage des services entre les différents composants est facilité par le fichier d'identificateurs de services (*Identification map file*) qui contient en fait l'identificateur du module fournisseur de service.
Un aperçu de ce fichier est présenté ci-dessous :

```
;-----------------------------|-------------|---------
; slave_name.port_name     |Services id|      Size
;-----------------------------|-------------|---------

TOP.MEMORY.target_port        0x10000000        0x4000

TOP.FIFO_DRIVER.target_port   0x10010000        Ox4

TOP.BFM.target_port           0x10010010        Ox4

…
```

Au niveau « top level » nous trouvons l'instanciation des divers composants logiciels et nous définissons leur connexion:

```
#ifndef _TOP_H_
#define _TOP_H_


/*--------------------------------
 * Includes
 *-------------------------------*/
  #include "systemc.h"
  …
  #include "sw_router.h"
  #include "sat_memory.h"
  #include "drv_fifo.h"
  …
{
   …

  //---------------------------------
  // Components
  //---------------------------------

  //Channel
   sw_router<int,int> * SW_ROUTER;

  //Memory
  sat_memory<int,int> * LOGICAL_MEMORY ;

  // Fifo Driver
  fifo_drv * FIFO_DRIVER;

  //BFM
  sw_hw_bridge<int> * BFM;

   …
  // Memory instantiation
  LOGICAL_MEMORY = new sat_memory<int,int>("LOGICAL_MEMORY",0x4000);
```

```
 //Channel instantiation
  SW_ROUTER = new sw_router<int,int>("SW_ROUTER");

 //Binding

SW_ROUTER->initiator_port(FIFO_DRIVER->target_port);
SW_ROUTER->initiator_port(BFM->target_port);
 …
FIFO_DRIVER->initiator_port(SW_ROUTER->target_port);
   …
} ;

#endif  /* _TOP_H_  */
```

5.4 Analyse expérimentale

Pour valider notre méthodologie, nous l'avons appliqué sur l'exemple MJPEG. Dans cette section, nous allons alors analyser les résultats que nous avons obtenus. Ces résultats nous permettent d'effectuer des analyses non seulement quantitatives, mais aussi qualitatives. Nous démontrons la validité de notre méthodologie. Ensuite nous présentons ses avantages. Les difficultés ainsi quelques limitations seront développées en dernier.

Réduction du temps de la phase de modélisation

Nous avons appliqué notre méthodologie à cet exemple. L'effort de modélisation de la spécification a été faible. L'écriture d'une telle spécification est facile et rapide.

Cette méthodologie nous a offert un format clair pour bien décrire le logiciel.

Par contre, sa conception est une tâche laborieuse et a nécessité un temps considérable (environ 5 semaines). Cette difficulté est notamment due à l'étendue des connaissances que requiert la conception: bonne connaissance du niveau TLM, du flot de conception des systèmes multiprocesseurs monopuces, de la structure et des fonctionnalités des systèmes d'exploitation embarqués, des protocoles de communication, etc.

Validation

Cette expérimentation nous a permis de valider les concepts et la méthodologie que nous avons proposés et qui ont requis un travail important. La méthodologie a été appliquée avec succès sur l'application MJPEG.

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. En effet, après utilisation le niveau s'avère rapide et il simplifie le travail du concepteur en faisant abstraction du logiciel et du matériel.
Le tableau ci-après résume les résultats de simulation des trois niveaux d'abstraction : niveau fonctionnel, niveau V.A et niveau RTL:

Tableau 5.1 – Résultats comparatifs de la simulation aux différents niveaux d'abstraction

Niveau d'abstraction	**Temps d'exécution**	**Temps de simulation**	**Précision**	**Vitesse**
Fonctionnel	--	$<$ 1 ms	0%	~ 10^6
Architecture Virtuelle	0.90 s	20 s	77%	~ 1260
RTL	0.73s	~ 7 h	100%	--

Ces résultats correspondent à la simulation de 25 images. Nous avons utilisé des processeurs ARM7TDMI cadencés à 40 Mhz.
La deuxième colonne du tableau montre le temps d'exécution qui représente le temps « SystemC » consommé par les différents CPU afin de traiter une seconde de séquences vidéo. Le temps de simulation correspond au temps utilisé par la machine hôte pour achever la simulation. Les deux dernières colonnes sont reliées à la précision ainsi qu'à la vitesse de simulation.
Les résultats obtenus montrent que la simulation au niveau Architecture Virtuelle permet une accélération considérable par rapport à une simulation au niveau RTL (plus de 3 ordres de grandeur). De même, la précision de la simulation du système entier est considérablement améliorée sans pour autant atteindre la précision absolue d'un modèle cycle à base d'*ISS* (~20% d'erreur). L'erreur introduite au niveau V.A est due notamment à l'inexactitude de l'estimation du temps d'exécution du logiciel et au niveau de la modélisation de l'interface logicielle/matérielle sans oublier que nous nous plaçons à un niveau d'abstraction élevé ce qui nous mène à tolérer une certaine erreur. D'où le niveau VA permet d'atteindre une précision comparable à celle obtenue au niveau cycle précis.
En utilisant cette méthode nous pouvons aussi démarrer très en amont le développement des logiciels embarqués sur un modèle représentatif du circuit final, tout en bénéficiant d'une vitesse de simulation importante plus rapide qu'au niveau RTL.
Donc les résultats obtenus montrent une vitesse de simulation considérable comparée avec une simulation classique basée sur ISS et une précision

raisonnable qui sont des critères clés pour une exploration d'architectures à un niveau d'abstraction élevé.
Un autre avantage de cette méthode est la simplicité de l'utilisation du modèle de simulation ce qui rend l'exploration des différents choix architecturaux facile tôt dans le cycle de conception. En effet, pour modéliser le logiciel embarqué un utilisateur pourra facilement manipuler le niveau SW TLM.
Pour implémenter un module *master* il doit avoir un port initiateur. Pour rendre un module utilisateur un *sat slave*, il doit hériter de *sat_slave_base* et implémenter alors l'interface SAT.
Les appels de services ont lieu quand un module *master* appelle l'une des méthodes de l'interface SAT à travers son port initiateur. Selon la méthode appelée, le port initiateur crée une requête et la transfère au port target en utilisant l'interface *service* de la couche Service. La partie *slave_base* du target décode la requête et appelle alors la méthode *sat_if* appropriée.
Pour le routage entre les composants maître et esclave nous utilisons un bus logiciel *« sw_router »*. Il conduit les transactions au target correspondant en suivant le service demandé.

Limitations

Comme toute nouvelle approche, il y a toujours quelques difficultés et quelques limitations. Nous allons en citer celles qui nous semblent les plus importantes.
En effet, le bus logiciel abstrait un système d'exploitation générique et donc au cas ou l'application nécessite un système d'exploitation spécifique nous devons intervenir dans l'implémentation du *sw_router* afin de supporter tous les services et les particularités de l'OS requis par l'application. Par ailleurs, les fonctionnalités requises pour les systèmes d'exploitations embarqués sont d'une grande variété, notamment pour les communications. Il est donc nécessaire que ceux derniers puissent supporter cette variété, et ils doivent donc disposer de très nombreuses parties spécifiques. C'est un obstacle à l'idée de standardisation générale des systèmes d'exploitation embarqués: en effet, à moins d'avoir un jeu de fonctionnalités disproportionné capable de fournir des fonctions optimales pour chaque cas, il est souvent nécessaire d'ajouter des fonctions spécifiques au système pour qu'il puisse fonctionner avec une architecture particulière [19].
De même chaque pilote TLM représente un élément qui fournit des services et requiert des services fournis par d'autres éléments. Pour cette raison, l'ajout

d'un nouveau pilote nécessite la définition de ses relations avec les autres éléments déjà existants. La description doit aussi définir les services fournis et requis par l'élément ajouté, les paramètres d'appel de chaque méthode du pilote et les liens vers les sources d'implémentation.
Toutes ces difficultés sont superficielles. Ainsi, l'expérimentation a confirmé la faisabilité de l'approche proposée.

5.5 Conclusion

Nous avons développé et illustré les possibilités d'une nouvelle méthodologie pour modéliser le logiciel embarqué en se basant sur TLM.
L'expérimentation de la méthode a montré tout d'abord sa faisabilité. Ensuite, elle a permis de mettre en évidence la simplicité de développement du logiciel embarqué à un haut niveau d'abstraction. Cette méthodologie est alors efficace et permet de fournir une présentation unifiée de tout le système.
Le chapitre suivant conclut ce document en donnant un bilan du travail effectué et les perspectives envisageables au terme de cette recherche.

Chapitre 6
Conclusion

Les systèmes embarqués sont présents dans des applications de plus en plus nombreuses. Récemment la demande pour ces systèmes et le nombre des fonctionnalités souhaitées s'est fortement accrue tandis que les délais de conception requis diminuent. Des architectures multiprocesseurs hétérogènes semblent devenir la clé pour que les systèmes embarqués puissent supporter cette complexité. En parallèle, l'intégration a fait de grands progrès. Cependant, les concepteurs n'arrivent plus à concevoir de tels circuits dans des délais raisonnables: ils manquent de méthodologies et d'outils; par ailleurs la vérification de ces systèmes devient de plus en plus complexe.
Aussi est-il important de fournir les méthodologies et les outils qui faciliteront et accéléreront la conception des systèmes monopuces. Pour ce faire, un flot de conception descendant est proposé par le groupe TIMA-SLS. De même, le besoin d'une méthodologie de conception basée sur une approche plus abstraite pour la conception des systèmes MPSoC est bien ressenti par le monde industriel et celui de la recherche. Dans cette optique, l'utilisation d'un modèle de représentation unifié est requise d'où une méthodologie pour la modélisation du logiciel embarqué est proposée.

Dans ce document, nous avons présenté les systèmes multiprocesseurs monopuces ainsi que les défis de conception de ces systèmes. A la lumière de ces défis, nous avons entrepris une étude des solutions proposées pour leur faire face.
Nous avons présenté après les architectures logicielles et matérielles des systèmes multiprocesseurs monopuces. Les niveaux de modélisation RTL ainsi que TLM ont été décrit. Ensuite, les flots de conception classiques ont été étudiés. Nous comprenons alors dans quelle mesure ceux-ci ne répondent pas aux besoins des futurs systèmes embarqués. Une nouvelle approche de conception plus appropriée, élaborée au groupe SLS, a été donc présentée. Cette approche se base sur un raffinement de l'architecture logicielle/matérielle.
Le niveau d'abstraction intermédiaire dans ce flot de conception à savoir le niveau Architecture Virtuelle a été ensuite décrit tout en présentant un

nouveau niveau de modélisation pour le logiciel embarqué qui est SW TLM ainsi que ses différents concepts de base. Nous avons exposé finalement l'application des concepts proposés sur l'application MJPEG. Cette expérience a montré l'intérêt d'une telle approche d'un point de vue pratique.

Ainsi, ce travail inaugure un axe de recherche important.
En effet, notre approche de la modélisation du logiciel embarqué offre de nouvelles perspectives et repousse encore les limites des flots de conception classiques.
Grâce à la méthodologie SW TLM, le matériel ainsi que le logiciel sont conçus parallèlement au niveau TLM permettant l'accélération de la simulation et l'exploration d'architectures tôt dans le cycle de conception.
Une perspective envisageable en prolongement direct de ce mémoire concerne la définition du SW TLM au niveau PVT. Un futur travail serait aussi de développer un outil automatique de génération de code pour les pilotes de communication en se basant sur les concepts du SW TLM. L'automatisation est une perspective très importante pour pleinement exploiter la méthodologie proposée et réduire le temps total de conception d'un système MPSoC.

Sans doute, les objectifs importants de conception sont de fixer les demandes de performance, de pouvoir comparer différents alternatives et de choisir celle qui respecte le mieux ces demandes. Il est naturel d'associer une phase d'évaluation des performances avec chaque étape de conception pour choisir la réalisation optimale.
Une perspective de ce travail serait alors l'exploration de l'architecture du bus logiciel, permettant d'offrir une bonne efficacité pour la réalisation du système d'exploitation embarqué, facteur critique dans les MPSoC actuels.

Glossaire

- **API** : Application Programming Interface, ensemble de routines standard destinées à faciliter au programmeur le développement d'applications.
- **ASIC** : Application Specific Integrated Circuit, circuit intégré développé spécifiquement pour une application.
- **BFM** : Bus Functional Model, interface pour la simulation permettant de transformer les accès mémoire fonctionnels en des accès mémoires cycle-près.
- **CPU** : Central Processor Unit, partie principale d'un système, réservée aux traitements.
- **FIFO** : First In First Out, classe de protocole de communication qui assure que les premières données envoyées sont les premières données reçues.
- **HAL** : Hardware Abstraction Layer, la couche basse de l'organisation du logiciel fournissant les pilotes et les contrôleurs pour la gestion de la communication.
- **IP** : Intellectual Property, élément (logiciel ou matériel) dont le fonctionnement est connu et documenté, mais dont la structure interne est inconnue.
- **IPC** : Inter-Process Communication (communication interprocessus), ensemble de fonctions de communications inter-processus. Les IPC fournissent des services de mémoire partagée, sémaphores et messagerie.
- **ISA** : Instruction Set Architecture, niveau d'abstraction pour le logiciel simulant l'architecture du jeu des instructions, avec la précision du cycle d'horloge.
- **ISS** : Instruction Set Simulator, outil qui s'exécute sur la machine hôte et qui émule la fonctionnalité d'un processeur.
- **MPSoC** : Multi Processor System on Chip, système monopuce – circuit intégrant sur une même puce différents composants fonctionnels (mémoires, processeurs, etc.).

- **RTL** : Register Transfer Level, niveau d'abstraction pour la spécification des systèmes.

- **SoC** : System on Chip, système monopuce, circuit intégrant sur une même puce différents composants fonctionnels (mémoires, processeurs, etc.).

Références

[1] A. A. Jerraya « *Long Terme Trends for Embedded System Design* » CEPA 2 Workshop – Digital Platforms for Defence, Bruxel, Belgique, Mars 15-16, 2005

[2] A. A. Jerraya. « *Programming Models and Hw-Sw Interfaces Abstraction for Multiprocessor SoC* ». DAC, Juillet 2006.

[3] A. Bouchhima: *Modélisation du logiciel embarqué à différents niveaux d'abstraction en vue de la validation et la synthèse des systèmes monopuces*. Rapport de thèse, TIMA, Mai 2006.

[4] A. Gerstlauer, H. Yu et D. Gajski. *RTOS Modeling for System Level Design*. Proc. Of Design, Automation & Test in Europe, Mars 2003.

[5] A. Haverinen, M. Leclercq, N. Weyrich, et D. Wingard, « *SystemC™ based SoC Communication Modeling for the OCP™ Protocol* », OSCI Technical Paper, Octobre dans www.systemc.org, 2002.

[6] A. Rose, S. Swan, J. Pierce, JM. Fermendez, "*Transaction Level Modeling in SystemC*", Disponible sur le site: Open SystemC Initiative: http://www.systemc.org, consulté le 11/10/06.

[7] B. Vanthournout. *Transactional level as the new design and verification abstraction above RTL*. Coware Inc, Leuven, Belgium, 2003.

[8] D. Culler, J.P. Singh, et A.Gupta. « *Parallel Computer Architecture: A Hardware/Software Approach* ». The Morgan Kaufmann series in Computer Architecture and Design, Août 1998.

[9] D. Desmet, D. Verkest et H. De Man. *Operating System based Software Generation for Systems-on-Chip*. Proc. Design Automation Conference, Juin 2000.

[10] F. Ghenassia. *Transaction Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer, Novembre 2005.

[11] I. Augé, F. Pétrot, R. Buchmann, F. Donnet, P. Gomez, et E. Faure. « *Disydent: un environnement pour la conception de systèmes numériques synchrones* ». Premier congrès international de Signaux Circuits et Systèmes (SCS'04), pp. 72-77, Monastir, Tunisie, Mars 2004.

[12] ITRS. « *International Technology Roadmap for Semiconductors: Design* ». 2001. Disponible sur le site http:// www.itrs.net/ Links /2001ITRS /Design.pdf, consulté le 20/12/06.

[13] ITRS. « *International Technology Roadmap for Semiconductors: System Drivers* ». 2005. Disponible sur le site http:// www.itrs.net/ Links/ 2005ITRS/ SysDrivers2005.pdf, consulté le 20/12/06.

[14] J. A. Rowson, A. S.-Vincentelli : *Interface-Based Design*. DAC 1997.

[15] J. Turley. « *Survey says: software tools more important than chips* ». Embedded Systems Design Journal, novembre 2005.

[16] L. Benini, G. De Micheli: *Networks on Chips: A New SoC Paradigm*. IEEE Computer, vol. 35, Janvier 2002.

[17] L. Cai et D. Gajski. *Transaction Level Modeling in System Level Design*. CEC Technical Report 03-10, Mars 28, 2003.

[18] L. Charest, E. M. Aboulhamid, et A. Tsikhanovish. *Designing with SystemC: Multiparadigm modeling and simulation performance evaluation*. Dans International HDL Conference, San Jose, USA, Mars 2002.

[19] L. Gauthier: *Génération de système d'exploitation pour le ciblage de logiciel multitâche sur des architectures multiprocesseurs hétérogènes dans le cadre des systèmes embarqués spécifiques*. Rapport de thèse, TIMA, Décembre 2001.

[20] M. Baklouti: *Performance estimation based on a high level abstraction model of MPSoC Hardware/Software architecture*. Mémoire de mastère, TIMA et EPT, Juin 2006.

[21] MathWorks. «The Mathworks – *Simulink – Simulation and Model-Based Design*», disponible sur le site http://www.mathworks.com/ products/ simulink/, consulté le 17/02/07.

[22] P. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, O. Benny, B. Lavigueur, D. Lo, G. Beltrame, V. Gagne, G. Nicolescu. « *Parallel Programming Models for a Multi-Processor SoC Platform Applied to Networking and Multimedia* ». IEEE Transactions on Very Large Scale Integration (VLSI) Journal, 2006.

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

[24] Samuel K. MOORE. « *Winner multimedia monster* » IEEE Spectrum journal, pages 18–21, Janvier 2006.

[25] S. Wang, S. Malik, et R. A. Bergamaschi, *Modeling and Integration of Peripheral Devices in Embedded Systems*. Dans les actes de Design Automation and Test in Europe (DATE 03). Mars 2003.

[26] SystemC 2.1, disponible sur le site http://www.systemc.org/ consulté le 22/10/06.

[27] W. Cesario and all: *Component-Based Design Approach for Multicore SoCs*. DAC 2002.