Définition d'une représentation intermédiaire basée sur une approche service pour le prototypage virtuel de systèmes sur puce

A. Chureau

INSTITUT POLYTECHNIQUE DE GRENOBLE

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

T H È S E

pour obtenir le grade de

DOCTEUR DE L'Institut polytechnique de Grenoble

Spécialité : Micro et nano-électronique

préparée au laboratoire **TIMA**

dans le cadre de **l'École Doctorale d'Électronique, Électrotechnique, Automatique et Traitement du Signal**

présentée et soutenue publiquement

par

Alexandre CHUREAU

Le 12 novembre 2008

DÉFINITION D'UNE REPRÉSENTATION INTERMÉDIAIRE BASÉE SUR UNE APPROCHE SERVICE POUR LE PROTOTYPAGE VIRTUEL DE SYSTÈMES SUR PUCE

Directeur de thèse : Ahmed Amine JERRAYA
Codirecteur : Frédéric PÉTROT

JURY

M. Tanguy RISSET	, Président
M. Pierre BOULET	, Rapporteur
M. El Mostapha ABOULHAMID	, Rapporteur
M. Ahmed Amine JERRAYA	, Directeur
M. Frédéric PÉTROT	, Codirecteur

À la Famiglia

To do is to be.
To be is to do.
Do be do be do.

Remerciements

Je remercie M. Ahmed Amine Jerraya, ancien responsable de l'équipe SLS au laboratoire TIMA et maintenant responsable des programmes de conception au CEA-LETI, d'avoir accepté de diriger ma thèse et de m'avoir confié un sujet pivot de ses travaux de recherche. Je remercie M. Frédéric Pétrot, professeur à l'Institut polytechnique de Grenoble, pour le temps et la rigueur qu'il a investis dans la co-direction de ma thèse. Je remercie également le professeur Pierre Boulet de l'Université des Sciences et Technologies de Lille et le professeur El Mostapha Aboulhamid de l'Université de Montréal d'avoir évalué et commenté mon travail, ainsi que M. Tanguy Risset, professeur à l'INSA Lyon, d'avoir accepté de présider le jury de ma soutenance.

Un grand merci à mes coéquipiers du SLS pour leur contribution scientifique et technique ainsi que pour les bons moments passés en leur compagnie, avec une pensée particulière pour Patrice Gerin, Hao Shen, Kati Popovici, Lilia Sfaxi, Wassim Youssef ainsi que pour les professeurs Amblard, Rousseau et Zergainoh, sans oublier notre assistante jusqu'en 2007, Sonja Amadou. J'adresse un merci tout spécial au personnel administratif du TIMA pour sa disponibilité et son professionnalisme tout au long de ces trois années de thèse. J'exprime également ma reconnaissance envers les projets européens IST SPRINT et SHAPES qui ont financé cette recherche.

Enfin, je remercie de tout coeur ma famille et mes amis du Québec, de France et de Navarre, toujours présents, sources d'inspiration, de découvertes et d'amour. I love you !

Table des matières

1 Les dispositifs de l'intelligence ambiante **1**

2 L'interface logiciel-matériel au cœur du système **7**
2.1 Introduction 8
2.2 Évolution des modèles utilisés pour la conception des SoC 8
2.2.1 Besoins et évolution de la modélisation 8
2.2.2 Caractéristiques des systèmes sur puce 9
2.2.3 Analyse des activités de conception à optimiser 11
2.3 Représentation de l'interface logiciel - matériel à plusieurs niveaux d'abstraction 13
2.3.1 Les niveaux d'abstraction considérés 13
2.3.2 Caractéristiques de l'interface logiciel-matériel 17
2.3.3 Spécifications liées à la problématique 18
2.4 Conclusion 18

3 Les représentations intermédiaires existantes **21**
3.1 Introduction 22
3.2 Le concept de service dans le design de SoC 22
3.2.1 La notion de service dans l'informatique répartie 22
3.2.2 L'architecture orientée service 25
3.2.3 L'utilisation des services pour la conception de SoC 27
3.3 Représentations intermédiaires 28
3.3.1 Le rôle d'une représentation intermédiaire 28
3.3.2 Représentations formelles 30
3.3.3 Autres représentations 32
3.3.4 Représentation des services 34
3.4 Conclusion 35

4 Le service comme abstraction de l'interface logiciel - matériel **37**
4.1 Introduction 38
4.2 Utilisation des services pour la conception de systèmes hétérogènes monopuce 38
4.3 Proposition d'un modèle de service pour systèmes hétérogènes 39
4.3.1 Les objets structurels 40
4.3.2 Description des ports logiques 44
4.3.3 Définition du service 45
4.3.4 Le modèle de service 46
4.3.5 Services reliés au module hybride 49
4.4 Esquisse d'un flot de conception orienté services 51

4.4.1 Principaux éléments du flot 51
4.4.2 Rôle de la bibliothèque de services 52
4.4.3 Sélection - intégration des services 53
4.5 Conclusion 57

5 Définition d'une représentation intermédiaire basée sur l'approche service 59
5.1 Introduction 60
5.2 Survol de la représentation intermédiaire 60
5.3 Objets de la structure de données 61
5.3.1 Objets structurels 61
5.3.2 Objets reliés aux Services 66
5.3.3 Objets de gestion 68
5.4 Sémantique du modèle Serif 69
5.4.1 Sémantique de composition 70
5.4.2 Sémantique d'exécution : génération de modèles SystemC 71
5.5 Format d'échange et de stockage persistant : SPIRIT+ 73
5.6 API Serif 76
5.7 Contexte méthodologique de Serif 77
5.7.1 Flot de design proposé 77
5.7.2 Environnement logiciel proposé 77
5.8 Conclusion 81

6 Génération de prototypes virtuels et validation de la représentation 83
6.1 Introduction 84
6.2 Première étude de cas : décodeur MJPEG 84
6.2.1 Préparation des spécifications pour l'entrée du flot 85
6.2.2 Modélisation du décodeur au niveau VA 88
6.2.3 Modélisation du décodeur au niveau TA 90
6.2.4 Prototypage virtuel au niveau CABA 93
6.3 Etude de performance des modèles et prototypes 95
6.3.1 Synthèse des résultats obtenus 95
6.3.2 Analyse des performances obtenues 95
6.4 Seconde étude de cas : *Wavefield Synthesis* 96
6.4.1 Capture des spécifications de l'application WFS 97
6.4.2 Analyse et résultats de la seconde étude de cas 97
6.5 Conclusion 98

7 Conclusion et perspectives 101

A Schema des services 105

B Capture du design MJPEG en Serif 107

Bibliographie 111

Acronymes 117

Table des figures

1.1 Exemple de système sur puce : le processeur Diopsis 740 2
1.2 L'écart de productivité entre la capacité d'intégration et les outils de design 3
1.3 L'interface logiciel-matériel aux niveaux système et implémentation 4
1.4 Objectif : capturer un design logiciel-matériel et permettre l'application itérative d'algorithmes 6

2.1 Les points de vue fonctionnel et implémentation d'un même système 10
2.2 Exemple de système hétérogène à plusieurs niveaux d'abstraction 14
2.3 Détails de l'interface logiciel-matériel aux principaux niveaux d'abstraction 16
2.4 Exemples d'interactions à l'interface logiciel - matériel 17
2.5 Les deux tâches à optimiser (exploration et raffinement) et les modèles impliqués 17

3.1 Points d'accès aux services (SAP) entre la couche transport et la couche réseau du modèle OSI 24
3.2 Le modèle de communication basé sur les fonctions de protocole 24
3.3 Exemple d'une architecture orientée service articulée autour d'un bus hétérogène. 26
3.4 Eléments constitutifs d'une représentation interne 29
3.5 Structure du workbench SPI 31
3.6 Connexion de type FIFO illustrant la relation entre service et port en SystemC 34
3.7 Spécification d'un port transactionnel et d'un service requis en IP-XACT . 35

4.1 Connexion entre composants hétérogènes grâce à une spécification de service 40
4.2 Les objets structurels proposés : module-port-net 44
4.3 Définition et instance d'un port logique 45
4.4 Relations possibles entre services et modules 46
4.5 Les principaux éléments d'une description de service 47
4.6 Définition de la visibilité d'un service 49
4.7 Mapping entre les caractéristiques de communication d'un service et les ports d'un module 50
4.8 Bibliothèque d'éléments du niveau TA 54

5.1 Diagramme des classes Serif en notation UML 65
5.2 Définition et instanciation à l'aide des objets structurels 66
5.3 Rôle des objets Serif dans la spécification d'un service requis et fourni . . . 67
5.4 Exemple de composition entre module logiciel et hybride 71
5.5 Exemple de transformation Serif -> SystemC au niveau VA 74

5.6 Proposition de flot de conception basé sur Serif 78
5.7 Interface graphique de l'outil SerifViewer 79

6.1 Flot de conception suivi pour l'étude de cas 85
6.2 Réseau de Kahn des principales tâches de l'application MJPEG 85
6.3 Modèle de l'application MJPEG exprimé en Serif 87
6.4 Modèle de l'architecture multiprocesseur utilisé 88
6.5 Composants de la plateforme exprimés en Serif 89
6.6 Modèle consolidé application - architecture au niveau VA 90
6.7 Application MJPEG et plateforme pour accélération matérielle de l'IDCT . 91
6.8 Modèle de la plateforme au niveau TA 92
6.9 Modèle consolidé application-architecture au niveau TA 93
6.10 Outil de visualisation d'un design Serif : l'application, la plateforme et le modèle consolidé 94
6.11 Flot de conception suivi pour la seconde étude de cas 97
6.12 Outil SerifViewer illustrant le système WFS 99

Liste des tableaux

4.1 Fonctions possibles pour un module logiciel ou matériel 42
4.2 Services normalisés offerts par le composant hybride 50

5.1 Principaux membres de l'objet Module 62
5.2 Principaux membres de l'objet ModuleInterface 62
5.3 Principaux membres de l'objet ModuleContent 63
5.4 Principaux membres des objets reliés au port logique 64
5.5 Objets structurels Serif et classes correspondantes 64
5.6 Objets reliés aux services et classes correspondantes 68
5.7 Objets de gestion et classes correspondantes 69
5.8 Correspondances entre les principaux éléments Serif et IP-XACT 75

6.1 Caractéristiques d'interface de la tâche DEMUX 87
6.2 Plage d'adressage du modèle TA 91
6.3 Mesures de performances des différents modèles et prototypes 95

Chapitre 1

Les dispositifs de l'intelligence ambiante

L'ÉVOLUTION de la microélectronique permet à des objets de plus en plus petits d'accomplir toujours plus de fonctions. Après l'ère des ordinateurs centraux et celle des ordinateurs personnels, nous entrons dans la troisième ère de l'informatique, celle de « l'intelligence ambiante », où de nombreux ordinateurs sont intégrés aux objets qui nous entourent. Certains domaines bénéficient particulièrement de cette nouvelle forme d'informatique, comme le bâtiment intelligent, les systèmes multimédias portatifs et l'automobile. Les constructeurs automobiles estiment que 90% de l'innovation dans leurs véhicules est d'origine électronique et que 75% des fonctions sont contrôlées par du logiciel [1]. Cette *informatique pervasive* [2] a de nombreux impacts sociaux, économiques et éthiques [1, 24, 58].

L'intelligence ambiante repose sur des circuits électroniques à faible coût, de petite taille, économes en énergie et avec de bonnes performances de calcul. Les systèmes sur puce (*System-on-Chip devices* ou SoC), qui intègrent plusieurs fonctions logicielles et matérielles sur un même morceau de silicium, possèdent ces atouts. Leur aspect logiciel leur confère la flexibilité alors que leur aspect matériel leur apporte la performance de calcul et une consommation optimale. Associés à des capteurs-actionneurs et parfois organisés en réseau, les systèmes sur puce offrent toutes les fonctions nécessaires au déploiement des dispositifs intelligents.

Une architecture de système sur puce retient particulièrement notre attention : le système sur puce multiprocesseur. Cette architecture offre la flexibilité du logiciel tout en répondant à des critères stricts de performance et de consommation. On divise généralement ces systèmes en deux grandes catégories : les systèmes homogènes et les systèmes hétérogènes. Dans la première catégorie, le système contient plusieurs instances identiques d'un même processeur, habituellement gérées par un système d'exploitation unique. Dans le cas des systèmes hétérogènes, le système contient différents processeurs spécialisés dans l'exécution de certains types de tâches.

1. Voir http ://www.medeaplus.org/web/downloads/profiles/T124_profile.pdf
2. Termes également utilisés : ubiquitous computing, everyware, ubimedia, objets communicants

La Figure 1.1 illustre le diagramme bloc d'un système de la seconde catégorie, le Diopsis 740 d'Atmel [3]. Les deux processeurs hétérogènes du Diopsis sont typiques des systèmes embarqués haute-performance : un micro-contrôleur (1) s'occupe du contrôle et des applications non-critiques alors qu'un processeur de signal numérique ou *DSP* (2) s'occupe des calculs gourmands de granularité élevée comme les transformées de Fourier et calculs vectoriels. La puce du Diopsis 740 a une surface de 45 mm^2 et consomme à peine un watt dans des conditions nominales de fonctionnement. Le fabricant la destine particulièrement aux applications acoustiques, de communication et de génération d'onde (beamforming). On y retrouve plusieurs autres éléments caractéristiques des systèmes sur puce : des interfaces d'entrée / sortie (3-4), plusieurs éléments mémoire (5) ainsi qu'une infrastructure de communication (6).

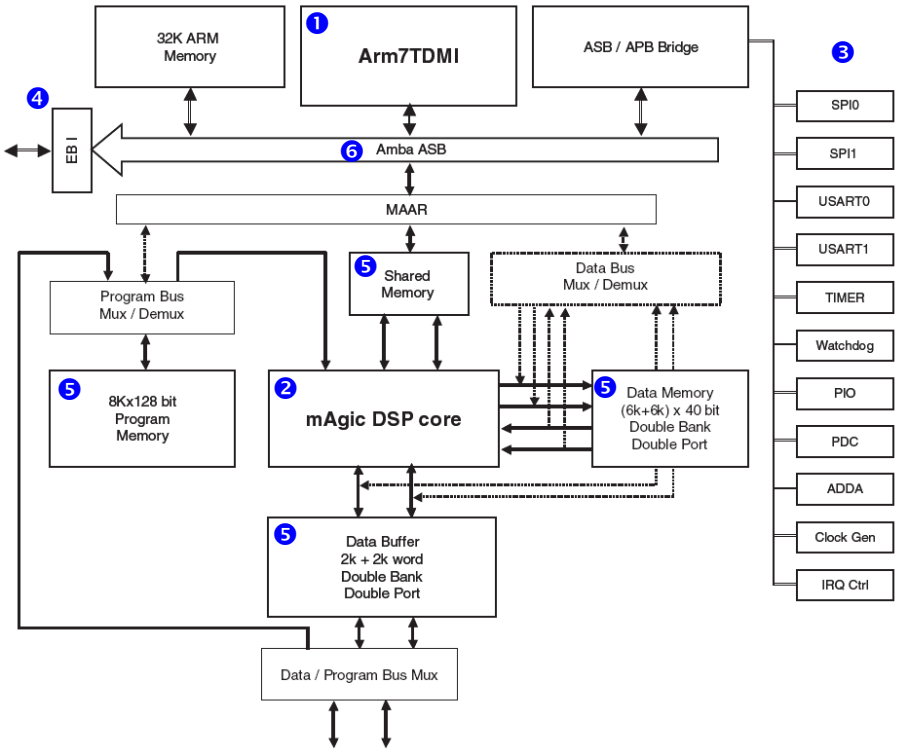

FIGURE 1.1 – Exemple de système sur puce : le processeur Diopsis 740

Le développement de ces systèmes intégrés complexes est soumis à des contraintes de délai de mise en marché et de coût très serrées, tout en ayant à satisfaire un important besoin de différenciation du produit. Les outils de conception et de programmation jouent un rôle déterminant dans l'atteinte de ces objectifs et dans le succès d'un projet. En témoigne l'importance attribuée à la qualité des outils logiciels dans le choix d'une plateforme de conception microélectronique [52]. Or, les outils d'aide à la conception et à la programmation de systèmes sur puce ont connu une évolution plus lente que la densité d'intégration (le nombre de transistor par unité de surface), qui a connu l'évolution de la loi de Moore. Ceci a conduit à l'« écart de productivité du design », représenté à la Figure 1.2 [4], qui traduit la difficulté des

3. Source : www.atmel.com
4. Source : International SEMATECH 1999, citée entre autre dans [34]

outils de conception à tirer profit de toute la puissance de calcul offerte par les transistors.

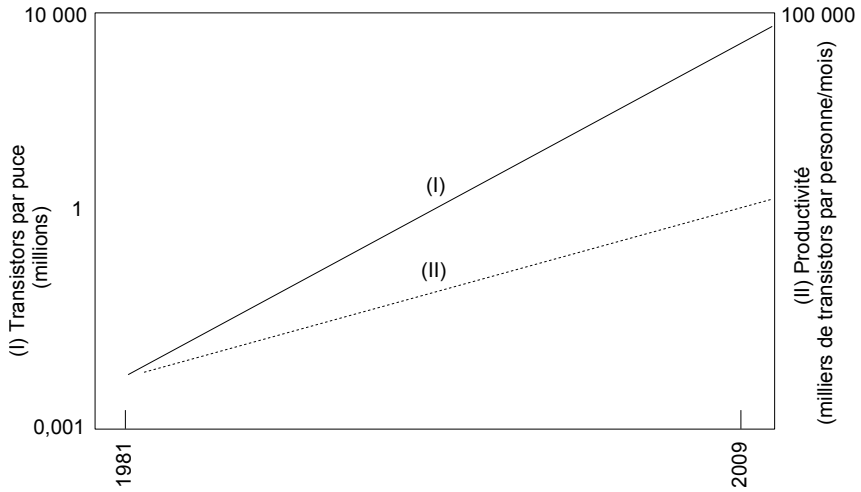

FIGURE 1.2 – L'écart de productivité entre la capacité d'intégration et les outils de design

Il est reconnu qu'une des principales difficultés dans le développement d'outils pour la validation, le raffinement et l'exploration d'architecture est la présence de composants hétérogènes au sein d'un même modèle du système. Des composants hétérogènes sont incapables de communiquer entre eux sans la présence d'adaptateurs. Deux principaux types d'hétérogénéité affectent particulièrement l'exécution des modèles :

– Au sein d'un même outil, on peut trouver des composants décrits à des degrés de détails différents. Un adaptateur d'abstraction suffit normalement à établir la communication. Exemple : un composant matériel décrit au niveau RTL et un composant matériel décrit au niveau transactionnel.
– L'exécution de modèles de composants peut se faire au sein d'outils de simulation différents. Il faut alors disposer d'un adaptateur entre les outils, parfois doublé d'un adaptateur d'abstraction. Exemple : un composant matériel (exécution dans outil de simulation) et un composant logiciel (exécution native ou dans simulateur de processeur).

Dans cette thèse, nous nous concentrons sur la relation entre les composants logiciels et matériels, source d'hétérogénéité du deuxième type qui affecte particulièrement le design de systèmes sur puce haute performance. Plusieurs facteurs de succès contribuent à la multiplication de composants logiciels (et de fait, à la présence de multiples processeurs) au sein des systèmes sur puce : un contrôle précis de la performance et de la consommation d'énergie et une flexibilité accrue. Tout ceci s'obtient au prix d'un effort de design supplémentaire.

La difficulté réside dans le fait que les composants logiciels sont tributaires d'une « tierce-partie », qui prend différentes formes selon le niveau d'abstraction auquel on se trouve : **le processeur**. Le processeur peut être vu comme l'élément qui réalise l'interface entre le logiciel et le matériel. Cette interface prend différentes formes selon l'évolution du design : à haut niveau d'abstraction, le processeur peut être vu comme un simple canal de communication abstrait, alors qu'au niveau de l'implémentation, le processeur est un sous-système matériel complexe. Ces extrêmes sont illustrés à la Figure 1.3.

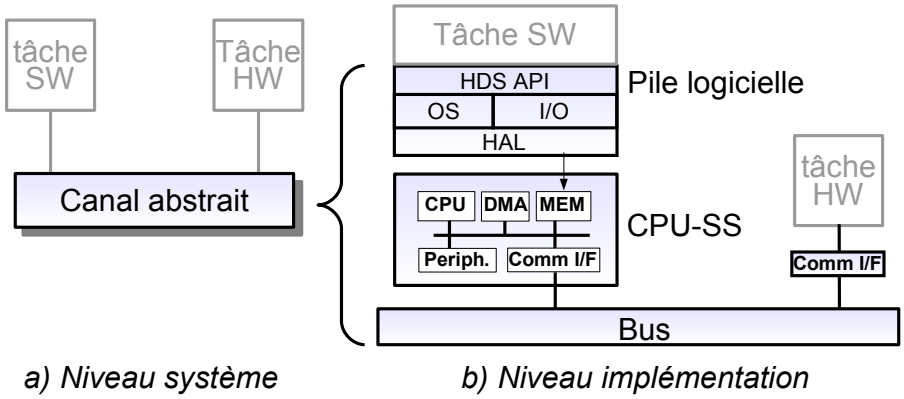

FIGURE 1.3 – L'interface logiciel-matériel aux niveaux système et implémentation

Nous allons tenter, dans cette recherche, de maîtriser les différentes représentation de l'interface logiciel-matériel au fil de l'évolution d'un design, du niveau spécification système à l'implémentation. Ceci devrait permettre le développement d'outils pour raisonner sur cette interface et faciliter certaines tâches de conception.

Pour permettre l'application d'algorithmes sur un design, les outils de conception utilisent une structure de données informatiques qui représente le design. Le développement d'outils pour systèmes sur puce repose donc sur l'utilisation d'une *représentation intermédiaire* des composants de l'interface logiciel-matériel à différents niveaux d'abstraction. Maîtriser l'interface logiciel-matériel revient à proposer un modèle de processeur adéquat pour chaque niveau d'abstraction.

Le besoin de représenter les composants hétérogènes d'un système sur puce dans un même environnement a motivé le développement de plusieurs outils et langages. La plupart de ces outils permettent de capturer et de simuler, au sein d'un même environnement, des composants de contrôle, de flot de données, parfois des composants analogiques ou logiciels. Peu se sont spécialisés dans la représentation de l'interface logiciel-matériel, et aucun, à notre connaissance, ne permet la représentation de cette interface à plusieurs niveaux d'abstraction. Il est important que les modèles de systèmes sur puce soient manipulés à différents niveaux d'abstraction, afin de trouver le meilleur compromis entre le temps de simulation et la précision des modèles. Ceci est valable pour chacune des activités de design et doit prendre chaque fois l'ensemble du système en considération.

Ce n'est donc pas un seul outil, mais une chaîne d'outils spécialisés et interdépendants qui est nécessaire. Les informations sur les interfaces logiciel-matériel doivent pouvoir « circuler » entre les outils, ce qui nécessite un format d'échange assurant la continuité du design. Seule l'information appelée à subir un traitement - analyse, modification ou transformation - est pertinente à représenter et échanger.

La solution développée dans le cadre de cette thèse est une structure de données dont la syntaxe et la sémantique sont adaptées à la représentation et l'assemblage d'éléments d'inter-

faces logiciel-matériel. Cette structure doit permettre à des outils de raisonner sur les caractéristiques fonctionnelles (canaux, protocoles, appels vers bibliothèque) et non-fonctionnelles (mapping tâche-processeur, plages d'adresse, performances) de l'interface. Des liens entre cette structure et les langages courants doivent être définis pour assurer une continuité dans le processus de design, de la capture des spécifications jusqu'à l'implémentation, en passant par la génération de prototype virtuels.

Une approche *service* devrait permettre l'abstraction adéquate des canaux de communication et dépendances entre les composants. Un service décrit un aspect de la relation entre un module et son environnement, que cette relation implique un canal de communication, une dépendance structurelle ou fonctionnelle. Le service est similaire à la notion de *contrat d'interface*, mais avec un aspect amélioré de publication et d'aggrégation : il est possible de rechercher un service dans un répertoire, et éventuellement de faire appel à plusieurs services pour réaliser une fonction complexe.

Dans cette recherche, nous allons explorer la représentation sous forme de service des différents composants de l'interface logiciel-matériel, à plusieurs niveaux d'abstraction. Les modèles obtenus devraient contenir l'information nécessaire pour que des outils de génération d'interfaces puissent intervenir, facilitant notamment l'étape d'exploration de l'architecture. Les niveaux d'abstraction visés vont du niveau *système* (Fig. 1.3(a)) au niveau *cycle accurate / bit accurate* (Fig. 1.3(b)), en passant par deux niveaux intermédiaires, qui permettent d'affiner le ratio précision - performance des modèles : les niveaux *architecture virtuelle* et *transaction accurate*.

Afin de permettre la génération de prototypes virtuels à ces niveaux intermédiaires, nous introduisons le concept de module hybride, qui est une abstraction du processeur. Le module hybride permet l'interconnexion de tâches logicielles et matérielles, en offrant différents services selon le niveau d'abstraction. Les composants logiciels de l'interface, comme le système d'exploitation et les pilotes de bas niveau, ainsi que les composants matériels du sous-système processeur, sont également représentés. Ces trois types d'éléments, logiciels, matériels et hybrides, sont préconçus et paramétrables, et forment des bibliothèques de génération de prototypes virtuels. La représentation intermédiaire devra permettre l'insertion et la configuration de ces composants d'interface par des outils, au sein des modèles d'application.

Un gain en productivité du design devrait être obtenu, nuancé par le temps nécessaire pour construire la représentation intermédiaire et l'effort d'adaptation des designs. Nous validerons ces hypothèse en réalisant deux cas d'études, visant un système multiprocesseur homogène et un système bi-processeur hétérogène.

Les principaux éléments de cette thèse sont résumés à la Figure 1.4. Nous allons développer un métamodèle (la structure de données) qui permettra la construction de modèles de systèmes logiciel-matériel. Ceux-ci pourront être obtenus via l'importation semi-automatique de différents langages de spécification système. Une fois le modèle central obtenu, des outils

de raffinement ou de génération pourront intervenir pour manipuler les interfaces logiciel-matériel. Ce processus itératif permettra de générer des prototypes virtuels exécutables à différents niveaux d'abstraction.

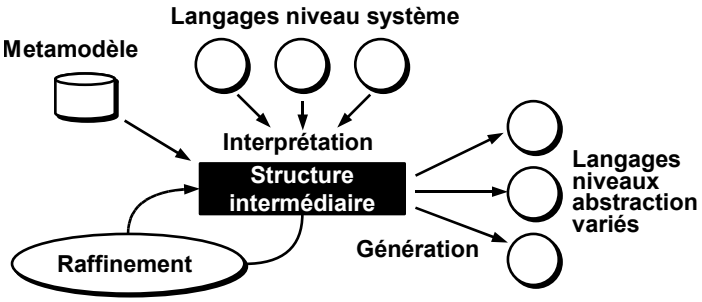

FIGURE 1.4 – Objectif : capturer un design logiciel-matériel et permettre l'application itérative d'algorithmes

Le corps de ce document se divise en cinq chapitres :

Le Chapitre 2 présente la problématique de la thèse, en effectuant un survol des besoins et difficultés de modélisation ainsi que des différents niveaux d'abstraction considérés.

Le Chapitre 3 propose une revue de l'état de l'art dans le domaine de la modélisation de systèmes logiciel-matériel. On y trouvera de nombreuses références aux langages et environnement qui tentent de concilier les aspects logiciel et matériel d'un système sur puce.

Le Chapitre 4 introduit le premier élément de contribution de la thèse : la définition du modèle des services. Ce modèle est dérivé de modèles utilisés dans l'informatique distribuée ainsi que de propositions précédentes de modèle de services pour la conception de système intégrés.

Le Chapitre 5 présente le deuxième élément de contribution : la représentation intermédiaire basée sur le modèle des services. Cette représentation sert au développement de quelques outils, également présentés dans ce chapitre.

Enfin, le Chapitre 6 valide la représentation intermédiaire par le biais d'une étude de cas. Cette étude présente le déploiement d'une application vidéo sur un système multiprocesseur de type SMP, où différentes options de partitionnement logiciel-matériel sont explorées. Une analyse des performances des modèles et du temps de design est effectuée afin d'évaluer l'atteinte des objectifs.

Chapitre 2

L'interface logiciel-matériel au cœur du système

Sommaire

2.1 Introduction 8
2.2 Évolution des modèles utilisés pour la conception des SoC 8
2.2.1 Besoins et évolution de la modélisation 8
2.2.2 Caractéristiques des systèmes sur puce 9
2.2.3 Analyse des activités de conception à optimiser 11
2.3 Représentation de l'interface logiciel - matériel à plusieurs niveaux d'abstraction 13
2.3.1 Les niveaux d'abstraction considérés 13
2.3.2 Caractéristiques de l'interface logiciel-matériel 17
2.3.3 Spécifications liées à la problématique 18
2.4 Conclusion 18

2.1 Introduction

Ce chapitre décrit la problématique abordée par la thèse. Nous y présentons d'abord le rôle de la modélisation dans la conception des systèmes sur puce. Puis nous définissons de façon détaillée l'élément cible de cette recherche, l'interface logiciel-matériel, et les difficultés qu'elle cause en matière de modélisation.

2.2 Évolution des modèles utilisés pour la conception des SoC

2.2.1 Besoins et évolution de la modélisation

Cette thèse porte sur la définition d'un modèle, puis de sa représentation informatique, pour la conception des systèmes sur puce. On sait que la conception d'un système complexe repose toujours sur un modèle, qui permet d'en maîtriser certains aspects. Le terme *modèle* fut d'abord employé dans les Beaux-Arts, au 16e siècle, pour désigner une "figure destinée à être reproduite", sens qu'on lui octroie encore aujourd'hui dans ce domaine. C'est dans les années 1950 qu'apparaît son emploi scientifique, par le biais de la cybernétique [1]. La définition d'un « modèle » tel qu'utilisé dans cette thèse provient de A. Birou [2] :

> Système physique, mathématique ou logique représentant les structures essentielles d'une réalité et capable à son niveau d'en expliquer ou d'en reproduire dynamiquement le fonctionnement.

Cette définition met en évidence les propriétés importantes d'un modèle, comme son *niveau* et la *reproduction dynamique du fonctionnement*, qui seront particulièrement exploitées au courant de cette recherche.

Plusieurs modèles interviennent dans la conception de systèmes électroniques intégrés. Certains modèles ont une origine mathématique et permettent de représenter un aspect du système réel en vue de l'application d'algorithmes de synthèse ou de vérification. Un exemple de tel modèle est le diagramme de Gantt, utilisé pour la synthèse comportementale par l'outil Catapult-C [35]. D'autres modèles ont une origine logicielle et permettent de reproduire le comportement du système réel à différents niveaux d'abstraction, afin d'en valider ou d'en caractériser le fonctionnement. La bibliothèque SystemC [40] fait partie de cette catégorie de modèles. Enfin, des environnements de modélisation reposent sur l'emploi de plusieurs modèles. L'environnement Simulink [49] est un de ceux-là car il utilise à la fois des mo-

1. Voir site du Centre national de ressources textuelles et lexicales, `http://www.cnrtl.fr/lexicographie/modele`
2. A. Birou, "Vocabulaire pratique des sciences sociales", 1966.

dèles mathématiques et logiciels pour reproduire le comportement de systèmes numériques ou analogiques.

Tous les modèles poursuivent le même double objectif : diviser un système complexe et abstraire l'information de la réalité représentée. S'il est naturel de diviser un système pour mieux en maîtriser la complexité, c'est l'abstraction de l'information qui procure un véritable gain en productivité. Un troisième élément, externe, intervient dans le façonnement des modèles : la réutilisation. Diviser, abstraire, réutiliser, ce sont les trois caractéristiques des modèles utilisés pour la conception de systèmes sur puces. Plusieurs modèles interviennent au courant de la vie d'un design et ceux-ci peuvent être liés de différentes façons. Les méthodologies dites "guidées par les modèles" (model-driven), par exemple, reposent sur des règles d'inférence de modèles et constituent un domaine de recherche actif [3].

Les tendances suivantes concernant les modèles utilisés pour la conception de systèmes-sur-puces peuvent être observées ces dernières années :

– Abstraction de plus en plus élevées pour l'exécution
– Synthèse automatique du comportement à partir du logiciel
– Réutilisation de composants pré-conçus
– Degré de parallélisme croissant
– Nombre de tâches réalisées en logiciel croissant

Ces tendances témoignent des principales évolutions des circuits intégrés que sont les systèmes sur puces, qui sont décrits dans la prochaine section.

2.2.2 Caractéristiques des systèmes sur puce

Les systèmes sur puce sont le résultat de la convergence du monde logiciel et du monde matériel en vue d'une flexibilité et d'un coût toujours plus avantageux. D'abord isolés l'un sur un processeur, l'autre sur des circuit dédiés, le logiciel et le matériel cohabitent désormais sur un même morceau de silicium où processeurs, mémoires, blocs IP, entrées, sorties et media de communication sont intimement liés. Lorsqu'un système sur puce contient plusieurs processeurs, on fait référence à un système MPSoC (Multiprocessor System on Chip).

D'un point de vue fonctionnel, le dénominateur commun entre le logiciel et le matériel reste la tâche. Ainsi, les spécifications initiales sont habituellement décrites sous forme de tâches, peu importe qu'il s'agisse éventuellement de tâches logicielles ou matérielles. D'un point de vue architectural cependant, le dénominateur commun entre une tâche logicielle et une tâche matérielle est le processeur.

Ces deux points de vue sont illustrés à la Figure 2.1, en prenant l'exemple d'un téléphone portable. Le fonctionnement du téléphone est décrit à haut niveau d'abstraction à gauche par un ensemble de tâches structurées sous forme de diagramme d'état. À droite, le point de vue architecture matérielle (implémentation) représente un ensemble de composants inter-

connectés qui réalisent les tâches décrites à gauche. Toute la difficulté de conception de ces systèmes consiste à passer du point de vue fonctionnel au point de vue architectural le plus efficacement possible. Cette difficulté se traduit par le « design productivity gap », dont il a été fait mention dans l'introduction.

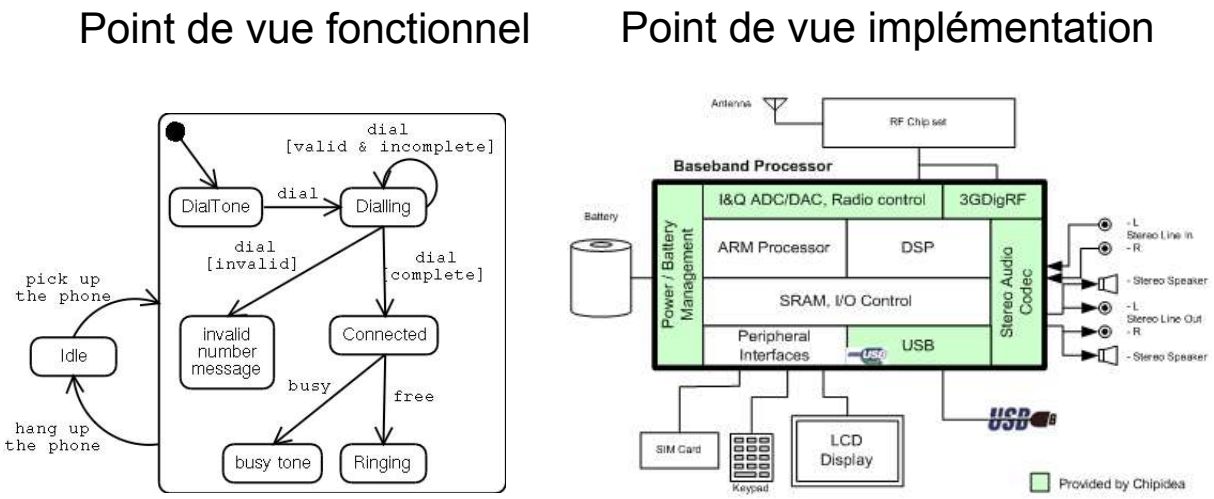

FIGURE 2.1 – Les points de vue fonctionnel et implémentation d'un même système

Les systèmes sur puce contiennent des millions de transistor qui permettent d'intégrer plusieurs fonctions sur un seul morceau de silicium, ce qui se traduit par un meilleur coût unitaire et une consommation moins élevée. En contrepartie, il est très coûteux de refaire les plans d'un tel circuit, sans parler des masques de photogravure eux-mêmes (2M$ à 2,2M$ en CMOS 45nm - EETimes 2008), ce qui arrive lorsqu'une erreur est détectée trop tard. Ainsi, malgré qu'ils soient très complexes, les systèmes sur puce doivent fonctionner pratiquement du premier coup sur silicium, d'où l'utilisation incontournable de modèles et platesformes de prototypage.

Nous tentons de mieux cerner le problème en découpant un système sur puce en quatre catégories de composants, présents à tous les niveaux de raffinement :

– *Les composants logiciels.* Progammables, ils offrent une flexibilité maximale. Le principal inconvénient est la consommation d'énergie du processeur requis pour exécuter le logiciel : de l'ordre de 500 fois plus élevée qu'un composant matériel dédié, pour un processeur généraliste. Leurs caractéristiques font qu'ils sont surtout utilisé pour les fonctions réactives (contrôle).
– *Les composants matériels dédiés.* Utilisés pour l'accélération de certains calculs avec une consommation d'énergie minimale. Ils sont essentiels pour l'implantation des capteurs, actionneurs et autres interfaces physiques. Leur flexibilité minimale et leur performance font qu'ils sont surtout utilisés pour les fonctions tranformatives (chemin de données).
– *Les composants hybrides.* Ils servent d'interface entre les composants logiciels et les composants matériels. Ils ont rôle d'adaptation et de synchronisation des communications, via un mix de contrôle et de transformation de données. Leur fonction précise dépend du niveau d'abstraction considéré, comme nous le verrons plus loin dans ce chapitre. On

trouve un module hybride pour chaque processeur ou par groupe de processeurs SMP de la plateforme.
– *Les composants de communication.* Du plus simple, le fil, au plus évolué, le réseau sur puce, leur fonction est essentiellement de transmettre les informations entre les composants décrits ci-dessus.

Une cinquième catégorie de composants peut venir s'ajouter selon la méthode de design utilisée : les composants de test. En effet, certaines méthodes ou environnement de design peuvent utiliser des composants de test qui se mêlent aux composants du système lui-même. Ces composants ont comme tâche de générer les stimuli (bloc source) ou de recueillir des données pour fins de vérification (bloc moniteur ou bloc récepteur).

Chacune de ces catégories emploie différentes représentations selon le niveau d'abstraction du composant. Cette particularité sera analysée plus en détail dans la section sur les représentation de l'interface logiciel-matériel (Section 2.3).

2.2.3 Analyse des activités de conception à optimiser

Il est généralement accepté que l'amélioration de la productivité du design passe, entre autres, par une meilleure intégration des composants logiciels et matériels tôt dans le design. Deux activités de design consommatrices de temps de par leur nature itérative bénéficieraient de cette meilleure intégration : l'exploration d'architecture et le raffinement, intimement liées par le concept de configuration d'architecture.

Activité à optimiser : l'exploration d'architecture

L'exploration d'architecture sert à valider, d'une part, le bon fonctionnement d'une certaine configuration de l'architecture à l'aide d'un modèle exécutable. D'autre part, elle sert à mesurer la qualité d'une configuration en fonction de critères non fonctionnels comme la surface requise, la puissance consommée et la flexibilité. L'exploration d'architecture est divisée en deux grandes étapes. Dans un premier temps, le choix d'une configuration d'architecture doit être fait en fonction de contraintes fonctionnelles et non fonctionnelles. Ceci résulte en une configuration C, faite d'un ensemble de paramètres $P : C = \{P1, P2, P3, ...\}$. Dans un deuxième temps, une co-simulation des composants logiciels et matériels est réalisée via l'exécution de leurs comportements respectifs, dans un but de validation.

La solution qui s'est imposée à ce besoin de co-simulation / co-exécution / co-validation fut d'abord d'utiliser un simulateur de jeu d'instruction (ISS pour Instruction Set Simulator). Cette approche centrée sur le matériel repose sur un outil logiciel qui simule l'exécution d'un programme sur un processeur avec une précision variable (comme pour l'outil Mentor Graphics Seamless [36]). La complexité des systèmes sur puce (potentiellement des centaines de processeurs !) a rendu cette solution obsolète, parce qu'elle impose un niveau de détail

trop fin et est inappropriée pour le niveau système où le type de processeur n'est pas nécessairement fixé. Plusieurs évolutions de la solution ISS ont tenté de résoudre ce problème en élevant le niveau d'abstraction de la co-simulation ou en permettant la communication entre modèles d'exécution indépendants. Les travaux présentés dans [22] et [14] portent notamment sur le couplage de modèles de composants à événements discrets et composants continus, représentatifs des systèmes mixtes logiciel-matériel. Ces approches permettent de valider des spécifications fonctionnelles de systèmes sur puce.

L'approche retenue dans la présente recherche repose sur un modèle de niveau transactionnel (TL pour Transaction Level) du matériel, présentée dans [21]. Selon cette approche, le processeur est modélisé en SystemC et le logiciel est exécuté nativement. Les communications entre le logiciel et le matériel sont interceptées par une couche d'adaptation et transformées en messages sur les ports SystemC du processeur. Il s'agit d'une approche mixte simulation native - SystemC qui a l'avantage d'offrir un bon niveau de détail et des temps de simulation plus court que dans le cas d'un ISS.

Le contexte de co-simulation étant établi, c'est surtout l'obtention d'un modèle exécutable en fonction d'une configuration C qui pose problème. Voici quelques exemple de paramètres P utilisés lors d'une exploration d'architecture :

– L'implantation logicielle ou matérielle d'une tâche
– Le nombre de processeurs
– La configuration des bus (hiérarchie, ponts, mise en réseau)
– Le nombre et la taille des mémoires, leurs ports d'accès
– etc.

De nombreux paramètres dérivés doivent également être ajustés : les plages d'adressage, le protocole et les signaux de l'interface de bus, la configuration du système d'exploitation, etc. Il n'existe pas d'outil ou de méthode générique (indépendante d'une plate-forme et valide de la spécification à l'implantation) qui permette d'assister ce processus. Le problème précis est de **générer des modèles exécutables et observables des composants hybrides qui réalisent l'interconnexion des composants logiciels et matériels à différents niveaux d'abstraction**.

Activité à optimiser : le raffinement

Une autre activité pouvant être optimisée à l'aide d'un outil est le raffinement. Le modèle initial d'une application pour système sur puce est décrit à haut niveau d'abstraction, pour des raisons de temps de simulation rapide et validation fonctionnelle tôt dans le design. Ce modèle doit ensuite être raffiné par des ajouts successifs de détails d'implantation. À cet effet, la configuration C utilisée lors de l'exploration de l'architecture peut être réutilisée. Mais cette fois, ce n'est pas un modèle exécutable qui est visé, mais un modèle raffiné, où les paramètres P deviennent partie intégrante du système. Seront également intégrés au

modèle des composants pré-conçus, soit par synthèse automatique, soit par édition manuelle.

Le problème, dans le cadre du raffinement, est de faciliter le passage vers un niveau d'abstraction plus fin des *composants hybrides*. Car ce sont eux qui subissent les plus importantes transformation lors du raffinement. Comme nous le verrons dans la prochaine section, à haut niveau d'abstraction, l'élément hybride est implicite, alors qu'au niveau implantation, c'est tout un ensemble de composants logiciels et matériels qui le forme.

2.3 Représentation de l'interface logiciel - matériel à plusieurs niveaux d'abstraction

2.3.1 Les niveaux d'abstraction considérés

Les composants qui réalisent la communication entre une tâche logicielle et une tâche matérielle peuvent être matériels (par ex. le processeur, l'interface processeur - bus de communication) ou logiciels (par ex. les pilotes matériel, l'ordonnanceur de tâches). Pour mieux abstraire l'interface logiciel-matériel à haut niveau d'abstraction, nous allons nous servir du composant hybride, introduit à la section 2.2.2. Le composant hybride représente les éléments implicites, ou abstraits, de l'interface logiciel-matériel, il se situe donc au cœur du processus de raffinement de l'interface. Ce qui est implicite dépend du niveau d'abstraction auquel le composant hybride est considéré. Les différents niveaux présentés ici vont permettre l'identification des éléments implicites et explicites.

La Figure 2.2 illustre un système simple aux niveaux clés d'abstraction, du niveau système au niveau cycle. L'application est constituée de deux tâche logicielles qui communiquent entre elles ainsi qu'avec une tâche matérielle. Les différentes étapes de raffinement de l'interface sont illustrées. Le composant hybride, représenté comme un losange, possède à la fois une interface logicielle et une interface matérielle.

Tel qu'illustré à la Figure 2.2a, le niveau d'abstraction le plus élevé considéré est le niveau système. À ce niveau, l'interface logiciel-matériel est complètement abstraite (ou implicite), ce qui permet de représenter la communication entre une tâche logicielle et une tâche matérielle par un canal abstrait de type FIFO ou buffer. Les opérations possibles sur ces canaux sont la lecture et l'écriture avec spécification d'adresse optionnelle. Ce modèle à haut niveau d'abstraction peut être considéré comme une spécification fonctionnelle du système hétérogène, où les interfaces logiciel-matériel sont implicites et réalisées par le langage de spécification utilisé. Des langages ou outils comme Simulink [28], SystemC ou C++ sont utilisés pour construire de tels modèles. On peut également définir dès ce niveau les principales caractéristiques de la plateforme matérielle visée, notamment le nombre de processeurs et de co-processeurs envisagés. Ces information de plateforme peuvent être exprimées à l'aide de langages systèmes comme SystemC ou Simulink, ou de formats spécialisés comme SysML

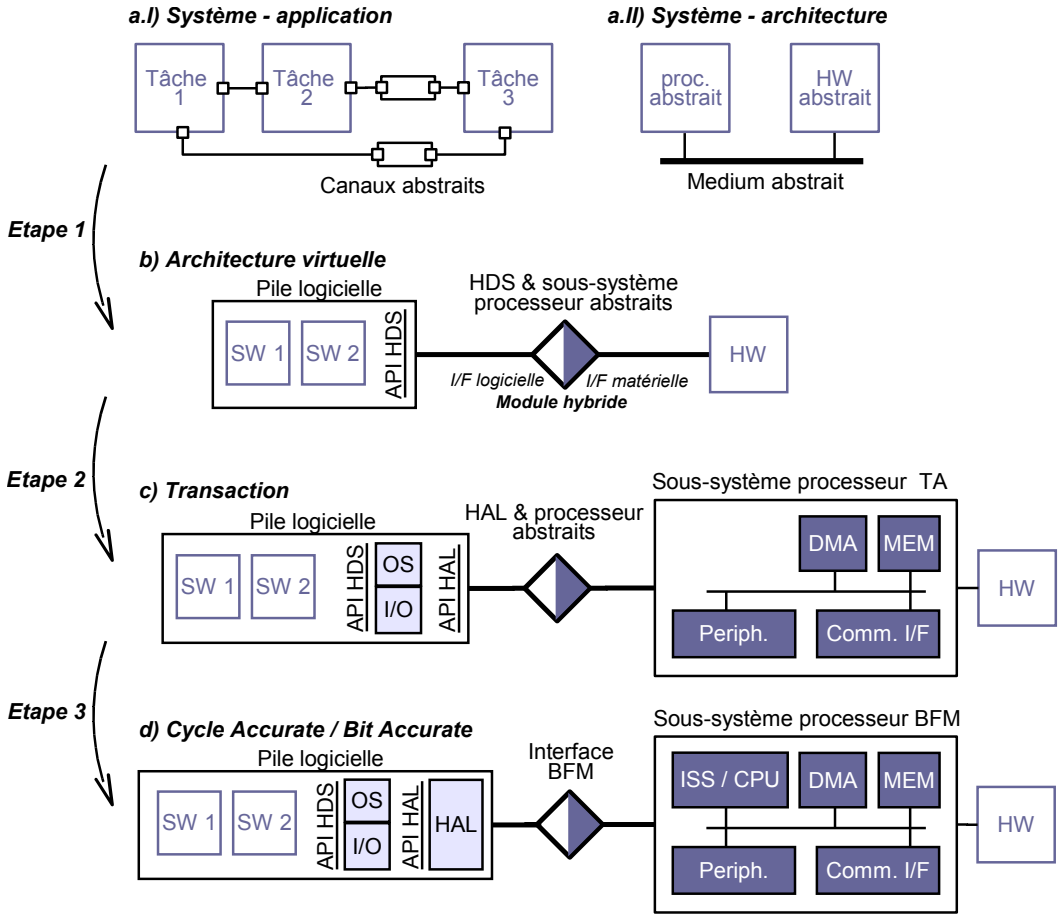

FIGURE 2.2 – Exemple de système hétérogène à plusieurs niveaux d'abstraction

ou DOL [51].

Les deux prochains niveaux d'abstraction sont des niveaux intermédiaires qui segmentent l'écart entre les interface complètement implicites du niveau système et les interfaces explicites de l'implémentation. Ces niveaux ont été présentés dans plusieurs travaux antérieurs, citons [9] pour le niveau transaction, [59] et [41] pour une vision multi-niveaux similaire. Au niveau architecture virtuelle (VA pour Virtual Architecture, Figure 2.2b), les tâches logicielles sont regroupées en sous-systèmes logiciels, qui sont associés aux représentations abstraites de processeurs. Il en résulte un modèle consolidé application - architecture.

À partir de ce niveau, les objets logiciel forment une pile, dont l'interface s'appuie sur l'API HDS (Hardware Dependent Software). Cette API propose des canaux de communication abstraits ainsi que des primitives d'OS (Operating System). Ces canaux fournissent un accès au reste de l'interface logiciel-matériel, représentée par le losange. Au niveau VA, ce losange abstrait complètement l'implantation du HDS et du sous-système processeur, et il communique directement avec le matériel.

Le prochain niveau d'abstraction, le niveau transaction (TA pour Transaction Accurate, Figure 2.2c), possède une architecture plus explicite qui permet des premières estimations de

performance. Les composants matériels du système communiquent en utilisant des transactions plutôt que des canaux abstraits. Un système d'exploitation ainsi que du code d'entrée-sortie spécifique sont ajoutés à la pile logicielle pour implémenter l'API HDS. Les interconnexions entre le sous-système processeur et les tâches matérielles sont rendues explicites par l'utilisation de modèles TA. Les tâches logicielles et matérielles sont synchronisées sur ces transactions via l'interface HAL (Hardware Abstraction Layer). A ce niveau, les implémentations du HAL et du processeur lui-même sont abstraites par le composant hybride, dont le rôle est de coordonner le chargement et l'exécution des tâches et de relayer l'information entre le HDS et les tâches matérielles. Les tâches logicielle sont maintenant sous forme binaire, prêtes à être exécutées de façon native,permettant des simulations plus rapides.

Le niveau Cycle Accurate / Bit Accurate ou CABA (Figure 2.2d) est le dernier niveau important à considérer, où chaque fonction du système doit être associée à une ressource matérielle de la plateforme (processeurs, bus, etc.). L'interface logiciel-matériel est séparée en deux : le sous-système processeur d'une part et le logiciel sous forme binaire d'autre part. Les piles logicielles sont complétées avec l'implémentation de l'API HAL et sont stockées en mémoire, en attente d'être exécutées en tant que code binaire par un simulateur de jeu d'instruction (ISS pour Instruction Set Simulator) ou un modèle matériel précis. La seule partie de l'interface logiciel-matériel qui demeure implicite est l'interface de connexion du processeur, qui peut être abstraite par un Bus Functional Model (BFM). Ainsi, le rôle du composant hybride se limite à la gestion des échanges d'instructions et de données avec le bus. Ce niveau permet des validations temporelles ainsi que des analyses de performances précises, au détriment d'une vitesse de simulation très lente.

Les composants hybrides n'existent pas dans le système implémenté, ce sont des éléments utilisés pour la modélisation seulement. Ils abstraient une unité logique d'exécution, qui peut être un seul processeur dans le cas d'une architecture hétérogène, ou un groupe de processeurs identiques, dans le cas d'une architecture homogène de type SMP.

Trois étapes intermédiaires permettent le raffinement graduel du modèle système, tel qu'illustré à la Figure 2.2 :
– Étape 1 : identifier les interfaces logiciel-matériel du modèle niveau système. Ceci conduira à un modèle partitionné de niveau VA.
– Étape 2 : insérer les composants logiciels indépendants du processeur (OS et entrées-sorties) et l'architecture du sous-système processeur
– Étape 3 : insérer les composants logiciels dépendants du processeur (HAL) et les détails du processeur lui-même.

Il est important qu'à l'issue de chacune de ces étapes les concepteurs disposent d'un modèle exécutable du système. Ce modèle exécutable, appelé prototype virtuel, est un élément clé de l'exploration d'architecture, tel que mentionné dans la Section 2.2.3. Pour mieux comprendre la problématique liée à la génération de ce prototype virtuel, le raffinement de l'interface logiciel-matériel à travers les niveaux d'abstraction est repris à la Figure 2.3.

Que ce soit pour passer d'un niveau d'abstraction à l'autre ou générer un modèle exécutable,

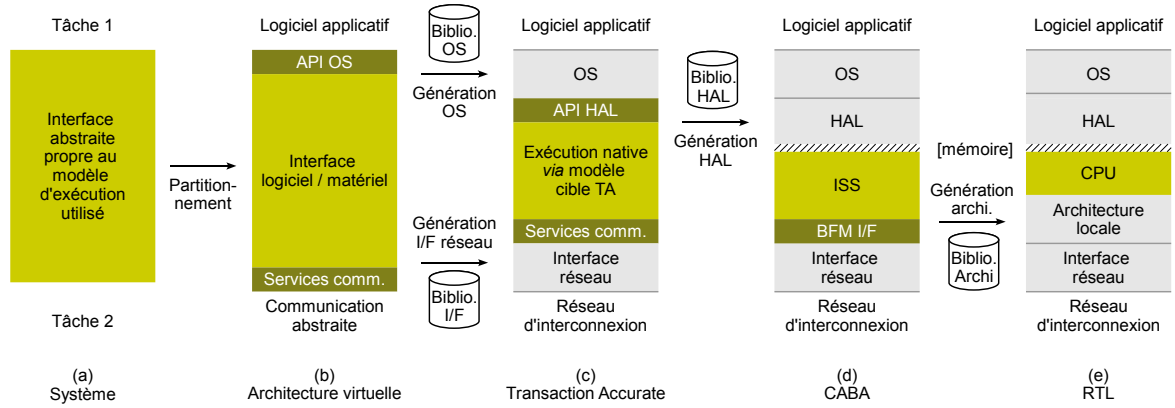

FIGURE 2.3 – Détails de l'interface logiciel-matériel aux principaux niveaux d'abstraction

les concepteurs puisent dans des bibliothèques d'éléments préconçus (OS, I/F, HAL, Archi) et construisent les éléments manquants. Le problème est qu'il n'existe pas d'outils spécialisés dans la manipulation d'interfaces logiciel-matériel, tel que décrit par les trois étapes ci-dessus. La difficulté se situe dans la décomposition de l'interconnexion logiciel-matériel.

Aux niveaux d'abstraction inférieurs, la présence de plusieurs composants logiciels devient une source additionnelle d'hétérogénéité. En effet, au niveau Système, le processeur et le logiciel spécifique (Hardware-Dependent Software, HDS) sont complètement implicites, les tâches sont modélisées comme de simple *threads natifs* [3]. Aux niveaux inférieurs, des détails à la fois du HDS et du processeur cible (détails matériels) sont ajoutés au modèle.

L'introduction de l'élément hybride dans un modèle n'est pas une solution en elle-même, mais elle permet du moins d'isoler le noeud du problème, l'interface logiciel - matériel. L'élément hybride est impliqué dans toute interaction avec une tâche logicielle (i.e. communication logiciel - matériel et logiciel - logiciel). Maîtriser le contenu de cet élément est un facteur clé de l'automatisation du raffinement et de la génération de modèles de systèmes sur puce.

Tel qu'illustré à la Figure 2.4, l'élément hybride présente une interface logicielle et une interface matérielle, ce qui rend possible différents types de communication :
- Du côté logiciel :
  - (API) Communication utilisant une API ne présentant aucune identification de canal
  - (T) Communication impliquant un canal SW/SW
- Du côté matériel, les communications impliquent nécessairement un canal :
  - (T) Canal complexe pour communications de type VA/TA (1 port = plusieurs fonctions)
  - (W) Canal simple pour communication de type R/W

Les outils à développer doivent donc être en mesure d'interpréter correctement le rôle d'un élément hybride au sein d'une interface logiciel / matériel, ce qui définit l'objectif principal du format intermédiaire.

3. Le thread, ou fil d'exécution, est un élément de base du parallélisme logiciel, géré par le système d'exploitation de l'hôte (natif) ou de la plateforme cible.

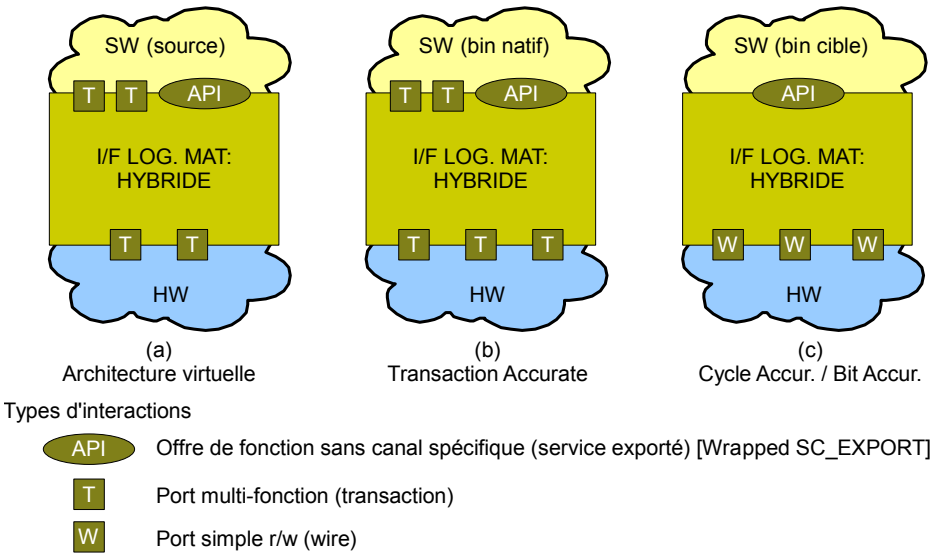

FIGURE 2.4 – Exemples d'interactions à l'interface logiciel - matériel

Nous faisons l'hypothèse que le fait de rendre le raffinement et la génération d'un prototype virtuel orthogonaux facilite le développement d'outils spécialisés et améliore la productivité du design. Cette séparation entre raffinement et génération repose sur un modèle "pivot" sur lequel ces deux dimensions du flot peuvent travailler, l'une nourissant l'autre. La Figure 2.5 résume les deux aspects de la problématique (exploration architecturale et raffinement) et les principaux modèles impliqués.

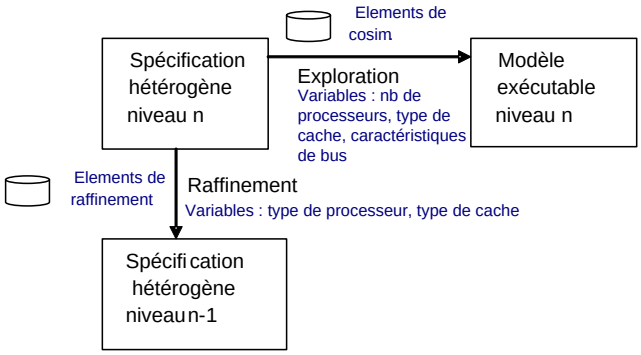

FIGURE 2.5 – Les deux tâches à optimiser (exploration et raffinement) et les modèles impliqués

2.3.2 Caractéristiques de l'interface logiciel-matériel

Les interfaces sont au cœur de nombreuses approches récentes du design de systèmes complexes, qu'il s'agisse de systèmes logiciels, matériels ou mixtes [42]. Le fait de considérer l'interface comme élément de base du design encourage la séparation des communications et de l'implantation, essentielle à la réutilisation.

Une interface, qu'elle soit associée à un composant logiciel, matériel ou hybride, possède les caractéristiques suivantes [6] :

- Une relation qui implique une communication explicite ou une relation structurelle entre deux éléments
  - Communication : canal de communication (de contrôle ou de données) transactionnel ou simple
  - Structurelle : sémantique particulière associée à la composition des interfaces (ex : association processeur - tâche, dépendances logicielles)
- Un protocole de communication, ou d'association (mapping) dans le cas d'une relation structurelle
- Des caractéristiques de plateforme, comme la plage d'adressage d'un composant ou la table de routage d'un routeur.
- Des caractéristiques de performance, comme le débit de traitement de données offert par un composant.
- Des caractéristiques de qualité de service, comme la garantie d'un BER (Bit Error Rate) ou le chiffrement des données.

Il est impératif de tenir compte de ces caractéristiques dans la capture des interfaces logiciel-matériel.

2.3.3 Spécifications liées à la problématique

Pour terminer ce chapitre, quelques spécifications sont identifiées afin de bien cerner le besoin d'une représentation informatique de l'interface logiciel-matériel.

Une première spécification est la capacité de représenter les quatre types de composants d'un système sur puce (logiciels, matériels, hybrides et de communication), du niveau Système au niveau Cycle Accurate. Une deuxième spécification est d'offrir une facilité de traitement et de transformation de la structure par différents outils dans un environnement de design hétérogène. Cette spécification implique l'utilisation d'un format d'échange persistant de l'information. Enfin, une troisième spécification est que la structure doit permettre l'utilisation d'itérateurs afin de modéliser des systèmes sur puce où certains composants peuvent être instanciés des dizaines voire des centaines de fois. Ces spécifications guideront la revue de l'état de l'art, présentée dans le prochain chapitre.

2.4 Conclusion

Ce chapitre a permis de cerner la problématique de la thèse. En résumé, elle se décrit comme suit : définir une représentation intermédiaire permettant (1) de raisonner sur des modèles mixtes logiciel-matériel à différents niveaux d'abstraction et (2) de développer des outils pour automatiser les tâches d'exploration d'architecture et de raffinement.

L'interface logiciel-matériel a été caractérisée à différents niveaux d'abstraction. Nous avons identifié sa composition : des éléments logiciels, matériels, hybride et de communication. Vue de l'extérieur, une interface présente des caractéristiques de communication, de plate-forme et de performance.

L'élément hybride permet l'interconnexion de tâches logicielles et matérielles. Ce composant représente différents sous-systèmes, plus ou moins explicites selon le niveau d'abstraction. Il sera au cœur des transformations du système entre les niveaux d'abstraction et lors de la génération de prototypes virtuels. La représentation intermédiaire doit servir de pivot entre ces deux dimensions et permettre une amélioration sensible de la productivité du design.

Chapitre 3

Les représentations intermédiaires existantes

Sommaire

3.1 Introduction 22
3.2 Le concept de service dans le design de SoC 22
3.2.1 La notion de service dans l'informatique répartie 22
3.2.2 L'architecture orientée service 25
3.2.3 L'utilisation des services pour la conception de SoC 27
3.3 Représentations intermédiaires 28
3.3.1 Le rôle d'une représentation intermédiaire 28
3.3.2 Représentations formelles 30
3.3.3 Autres représentations 32
3.3.4 Représentation des services 34
3.4 Conclusion 35

3.1 Introduction

La problématique a été cernée : il s'agit d'améliorer la productivité du design par le développement d'outils pour la génération et le raffinement de l'interface logiciel - matériel. Le développement d'outil passe d'abord par le choix d'une représentation interne qui permettra de raisonner sur un design hétérogène. Ce chapitre présente d'abord l'état de l'art sur le concept de service utlisé dans les systèmes hétérogènes et systèmes sur puce, suivi d'une revue des représentations internes utilisées pour le design de systèmes sur puce.

3.2 Le concept de service dans le design de SoC

3.2.1 La notion de service dans l'informatique répartie

L'idée principale de cette thèse est l'utilisation de services comme support à la génération d'interfaces logiciel-matériel dans un système sur puce. Le service doit permettre de spécifier les caractéristiques de communication et de performance attendues de chaque interface. Comme nous le verrons dans ce chapitre, différentes versions de cette hypothèse ont été proposées lors de recherches antérieures, sans connaître l'aboutissement d'une méthode de conception pratique et efficace. Pourtant, le concept de service connaît depuis quelques temps une grande popularité dans le milieu de l'informatique répartie, en particulier sous la forme des services Web (Web Services) [8].

Informellement, le service se présente comme la réalisation d'une fonction par un exécutant au bénéfice d'un client. Cette définition trouve adéquatement sa place dans le domaine informatique. Deux programmes, par exemple, peuvent jouer le rôle de l'exécutant et du demandeur. On associe généralement des conditions d'utilisation au service ainsi que des obligations de la part du serveur, parfois sous la forme d'un contrat [27].

Une application classique du concept de service est l'architecture client-serveur qui caractérise la majorité des réseaux informatique. Le terme client-serveur, apparu dans les années 80, désigne un réseau d'ordinateurs où certains jouent le rôle de serveurs et d'autres de clients, les communications s'effectuant sous forme de requêtes ou messages [44]. Cette architecture a supplanté l'architecture centralisée à temps partagé, où les ressources d'un seul ordinateur central (*mainframe*) étaient exploitées par plusieurs terminaux, ceux-ci ne servant qu'à capturer et présenter les données aux utilisateurs. Les raisons du succès de l'architecture client-serveur sont principalement sa flexibilité, sa modularité et sa compatibilité avec les interfaces graphiques (GUI).

Ainsi, le premier service offert est le service d'accès à des données partagées, mis en oeuvre sur un serveur de fichiers. Bientôt, la taille et l'hétérogénéité croissante des réseaux congestionnent le serveur de fichier et provoquent l'apparition de services spécialisés : service ré-

seau, service de messagerie, service d'encryption, habituellement offerts par des serveurs intermédiaires insérées entre le client et le serveur (architecture client-serveur multi-niveaux).

Favorisant la répartition géographique des réseaux, le terme service a été également utilisé pour désigner certaines fonctionnalités liées à la réseautique et aux télécommunications. Le réseau numérique à intégration de services (RNIS), par exemple, est un ensemble de protocoles permettant le transfert de données numériques de type voix, video, données et fax via un medium unique [56]. Les services SMS et MMS, utilisés en téléphonie mobile, illustrent également l'emploi du terme service dans la désignation de fonction de télécommunication. On reconnaît, dans ces exemples, l'utilisation du terme service pour désigner une fonctionnalité normalisée offerte par un système complexe.

Une application courante du concept de service en télécommunication est le critère de « qualité de service » (Quality of Service - QoS en anglais). L'Union internationale des communications définit la qualité de service comme l'« ensemble d'exigences de qualité relatives au comportement collectif d'un ou de plusieurs objets » [53]. On associe souvent la notion de qualité de service au service de transmission d'information, afin de décrire des caractéristiques telles que le débit de transfert, le temps d'attente, la probabilité de défaillance, etc. Plusieurs protocoles permettent la spécification d'un niveau de qualité de service, permettant une meilleure utilisation des ressources en fonction des besoins de l'application.

Représentant essentiellement un échange coordonné d'information entre deux entités, le service a également trouvé une place naturelle dans la description détaillée de protocoles de communication. L'ISO (International Standard Organization) a proposé le modèle OSI (Open Systems Interconnection), un modèle de référence à sept couches utilisé pour la description de protocoles de réseaux hétérogènes [62]. Le modèle OSI utilise abondamment le concept de service comme moyen de *couplage lâche* entre les couches. En théorie, chaque couche de ce modèle fournit des services aux couches supérieures. L'accès à un service se fait par l'intermédiaire des points d'accès aux services (Service Access Points - SAP en anglais), qui permet l'identification unique d'un service offert. Comme l'illustre l'exemple du SAP "NSAP" (Network SAP) offert par la couche réseau à la couche transport (Figure 3.1), un SAP est constitué d'une adresse unique ainsi que d'un protocole d'utilisation du service. L'adresse, constituée d'un identificateur d'entité cible et d'un identificateur de SAP, permet à l'entité transport d'identifier précisément le points d'accès au service voulu.

Adoptant une approche plus près de nos considérations, Zitterbart définit ainsi le service [63] :

> Cette nouvelle caractéristique, absente des systèmes de communication existants, est extrêmement importante pour les futurs sous-systèmes de communication à cause de la variété des applications émergentes. Elles sont aussi variées que la simple transmission d'un message texte ou le traitement complexe d'un contenu multimédia.

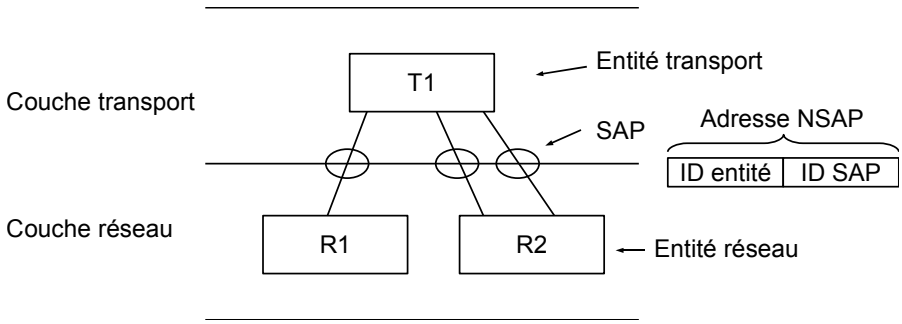

FIGURE 3.1 – Points d'accès aux services (SAP) entre la couche transport et la couche réseau du modèle OSI

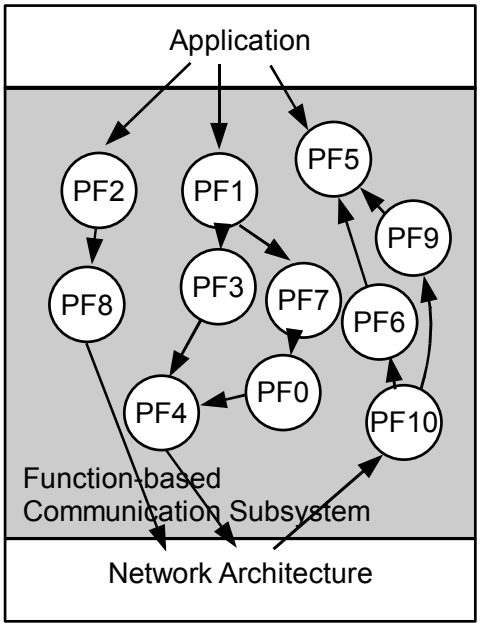

FIGURE 3.2 – Le modèle de communication basé sur les fonctions de protocole

Dans ses travaux, Zitterbart met à l'index le modèle en couche OSI, trop contraignant, et propose un modèle plus modulaire, centré sur la notion de fonction. Motivé par les besoins de flexibilité et de performance des applications communicantes, ce modèle permet un assemblage fin de composants pour un sous-système de communication en fonction des services requis par l'application, court-circuitant ainsi les multiples couches d'adaptation imposées par les modèles de type OSI. Chaque service est réalisé par un ensemble minimal de fonctions (par ex. contrôle de flot, ackowledgement, correction d'erreur, encryption), sélectionnées selon différents critères quantitatifs et qualitatifs. Les dépendances entre fonctions sont stockées dans un base de données et peuvent être illustrées sour la forme d'un graphe (Figure 3.2). Des outils de configuration de session, de protocole et de code effectue le travail d'analyse et d'implantation des machines de protocole insérées entre l'application et le réseau.

3.2.2 L'architecture orientée service

Le paradigme service a joué un rôle de premier plan dans l'évolution des réseaux informatique et, plus généralement, des systèmes hétérogènes. Internet, le réseau des réseaux, est un exemple de sytème hétérogène, où les entités communiquent entre elles grâce à une suite de protocoles : TCP, IP, Telnet, FTP, SMTP, etc. [30]. Ces protocoles sont associés à des applications bien définies, qu'il s'agisse du transfert de données vers un navigateur Web, d'établir une session interactive avec un hôte distant ou de consulter un serveur de messagerie électronique. Or, l'utilisation d'Internet a connu une évolution fulgurante et les multiples interactions possibles par le réseau ont laissé entrevoir un potentiel d'application énorme, sous la dénomination de *Web2.0* [55]. Cette nouvelle génération d'Internet veut que l'on ne se contente plus des services offerts par les protocoles sus-mentionnés, mais que l'on ouvre l'offre de service au niveau application, selon la convention des « services Web ». Cette convention n'est qu'un cas particulier du paradigme plus général de *l'architecture orientée services*.

L'architecture orientée services (en anglais, SOA - Service Oriented Architecture) est une façon de concevoir un logiciel ou un système de façon à fournir un service à une application ou à un autre service, via la publication d'une interface. Ce nouveau paradigme de conception fut proposé comme solution aux interdépendances croissantes entre applications d'entreprises. L'impact de l'approche SOA est tel qu'on la présente souvent comme l'évolution de l'architecture client-serveur [17], et la conception orientée service (Service-Oriented Design, SOD en anglais) comme l'évolution de la conception orientée objet. Pionnier en matière d'architecture orientée service, IBM a placé le concept de service au coeur de sa stratégie d'innovation pour les systèmes d'entreprise. [18] donne la définition suivante du *service* :

> Un service a une interface définie (avec un ensemble de messages que le service reçoit et envoie ainsi qu'un ensemble d'opérations nommées ou verbes), une implémentation de l'interface ainsi que, si déployé, un "binding" vers une adresse réseau documentée. Des exemples de services (...) sont une application basée sur l'envoi de messages (...) ainsi qu'une classe Java.

Reprenant le principe de séparation entre l'interface et le contenu, propre à la conception orientée objet, l'approche SOA fournit les moyens de publier cette interface auprès de clients potentiels et d'en faciliter l'invocation à distance. Les deux clés de voûte d'une architecture SOA de type service Web sont la notation XML, utilisée tant pour la description des services que pour celle des données échangées, ainsi que le protocole HTTP, utilisé pour la transmission des messages XML entre les hôtes. Les communications sont habituellement effectuées via un bus hétérogène, appelé, ESB (Enterprise Service Bus). La Figure 3.3 représente les principaux éléments d'une architecture SOA.

Plutôt que de décrire les détails techniques d'une architecture SOA, nous allons décrire les

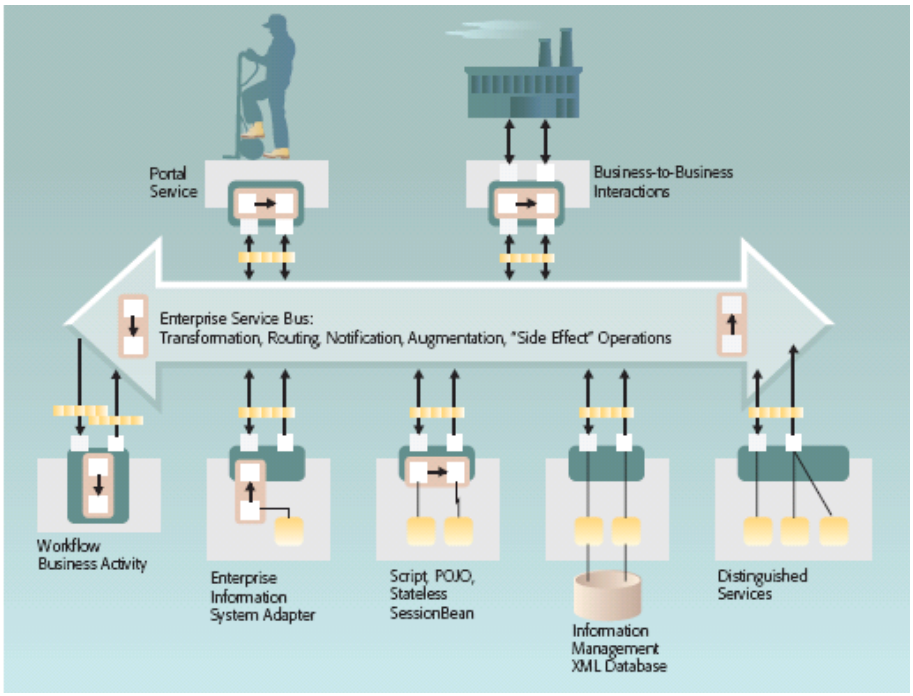

FIGURE 3.3 – Exemple d'une architecture orientée service articulée autour d'un bus hétérogène.

besoins adressés par ce type d'architecture et les solutions qu'elle y apporte. Il est en effet très intéressant d'établir un parallèle entre les besoins-solution liés à l'approche SOA et les besoins-solutions liés à la conception des systèmes monopuce.

A l'origine de l'architecture SOA se trouve le besoin d'indépendance des plates-formes. Ceci pour permettre, d'une part, la communication entre ordinateurs aux architectures et emplacements géographiques différents, d'autre part, pour permettre une vision cohérente d'un réseau hétérogène. Ainsi, différents système d'exploitation, différentes structures de données, différents langages de programmation, peuvent se trouver unifiés à l'intérieur d'une même offre de service. Les technologies plus anciennes comme CORBA et DCOM poursuivent des objectifs similaires, mais le fait qu'elles soient trop liées à une plate-forme particulière a nui à leur diffusion au delà du PC.

Il va sans dire que les systèmes hétérogènes monopuces ont plusieurs caractéristiques communes avec les systèmes qui bénéficient de l'approche SOA. Ce n'est pas un hasard si l'évolution du design des systèmes sur puce tend à assimiler des principes d'architecture de l'informatique distribuée, comme le démontre si bien l'utilisation récente des réseaux sur puce.

Les solutions proposées par l'approche SOA trouvent donc écho dans la conception de systèmes sur puce, en offrant une réponse aux besoins suivants :

1. Couplage faible : dissociation du processus d'appel d'une fonction grâce à l'utilisation de messages, dont le contenu et la séquence sont décrits en XML (par les normes SOAP - Simple Object Access Protocol et WSDL). L'interconnexion des entités se fait

via un bus hétérogène, le Enterprise Service Bus, un bus multi-protocole, multi-point (selon la nomenclature IBM).

2. Interopérabilité : les entités impliquées dans un service ont des architectures matérielles et logicielles multiples, communiquant entre elles grâce à la description des services et des données en XML.
3. Ouverture des systèmes aux spécialistes application : permet aux personnes impliquées dans la définition de l'application plutôt que dans son implémentation de consulter les services disponibles et de construire eux mêmes des solutions à partir d'éléments préconçus.
4. Répartition : le traitement de l'information peut être réparti entre différentes entités, chacune fournissant un service spécialisé.

Il faut également relever quelques limitations dans l'utilisation d'une approche SOA pour la conception de systèmes sur puce. Le concept de service devient encombrant à très bas niveau d'abstraction, où les concepteurs manipulent directement les signaux d'interruption, les signaux "enable" et "reset" et autres spécificités d'un composant. Il est également difficile d'exprimer, à l'aide de services, la synchronisation et l'interdépendance des nombreuses fonctions réalisées au sein d'un système sur puce. L'utilisation de spécifications complémentaires, comme les diagrammes de séquence UML, s'impose.

Le survol de la vision SOA permet d'apprécier les avantages qu'une telle approche procurerait à la conception des systèmes monopuce. Dans la prochaine section, nous allons présenter différentes applications du concept de service à la conception des systèmes sur puce.

3.2.3 L'utilisation des services pour la conception de SoC

Un des principaux défis de l'automatisation du design des systèmes embarqués et des systèmes sur puce est la synthèse de composants à partir du niveau système. Les travaux de Zitterbart ont illustré la synthèse d'interfaces à partir d'éléments atomiques (les fonctions de protocole), guidée par les besoins de l'application (services requis). Les résultats présentés par Daveau dans [16] ciblent également la génération d'interface, mais cette fois dans le contexte des circuits intégrés. Les interfaces ciblées sont les interfaces créées par le partitionnement logiciel-matériel d'un système hétérogène. Le service, tel que présenté par Daveau, est une fonction de communication offerte par un canal abstrait reliant deux processus. La synthèse des interfaces s'effectue par une sélection d'unités de communication à partir d'une bibliothèque, l'objectif étant d'assurer les fonctions du service de communication à l'aide de protocoles et signaux de bas niveau.

Le service n'occupe pas une place prépondérante dans ce travail et sa sémantique n'est pas détaillée. C'est plutôt dans [19], [23], [37] et [33], travaux menés au sein du groupe TIMA-SLS, que nous trouvons une description détaillée de l'utilisation du service dans la conception des circuits monopuces. Ces travaux complémentaires couvrent respectivement la génération du système d'exploitation, la génération de l'interface de communication externe

du CPU, la génération du sous-système CPU et la génération d'interfaces de cosimulation. Pour Gauthier, un service représente une fonctionnalité de l'OS. Pour Lyonnard, un service est l'abstraction d'une fonctionnalité d'un élément du sous-système CPU. Pour Grasset, le service est similaire au SAP du modèle OSI, c'est l'abstraction d'une fonction de communication.

Des incarnations du concept de service spécifiques à certaines représentations internes seront présentées plus loin dans ce chapitre. Pour le moment, nous voyons que même si certaines utilisations du service s'approchent très près de nos besoins d'abstraction, aucune ne possède les caractéristiques et requis identifiés au Chapitre 2.3.2.

3.3 Représentations intermédiaires

3.3.1 Le rôle d'une représentation intermédiaire

Les outils informatique d'aide à la conception fonctionnent toujours avec une structure de données qui représente un concept du monde réel. Dans le cas des systèmes sur puce, cette structure doit pouvoir représenter deux aspects orthogonaux : l'application et l'architecture. L'application est définie par un ensemble de comportements et représente l'action du système sur son environnement. L'architecture, quant à elle, est l'ensemble de ressources matérielles et logicielles qui permettent d'exécuter l'application.

Dans les systèmes de type ASIC, une architecture est habituellement dédiée à une seule application. On peut dire, dans ce cas, que l'architecture et l'application se confondent. Dans les systèmes sur puce, les extrémités de l'association entre l'application et l'architecture ont une cardinalité {1..*}, ce qui signifie qu'une architecture peut servir à plusieurs applications, et qu'une application peut être exécutée sur plusieurs architectures. Ces deux aspects ont des requis différents pour leur représentation informatique :

– Pour l'application, les outils doivent disposer d'un modèle, formel ou non, permettant d'exprimer *le comportement d'un design*. Un exemple est la machine à états de Harel, utilisée pour la spécification d'algorithmes de contrôle.
– Pour l'architecture, la représentation doit permettre d'exprimer *l'assemblage hiérarchique* de composants matériels et logiciels ainsi que leurs interconnexions. Des modèles formels ou semi-formels peuvent être utilisés, comme dans le cas des langages de description d'architecture.

Qu'il s'agisse de représenter l'application ou l'architecture, ces structures de données sont habituellement associées à un format persistant qui permet la sauvegarde d'un design dans un medium de stockage. Par exemple, le format SCXML (State Chart XML) permet la sauvegarde des machines à états de Harel dans un fichier texte, ce qui permet de sauvegarder l'état actuel d'un design et d'en faciliter l'échange entre les outils. Parfois, une notation graphique

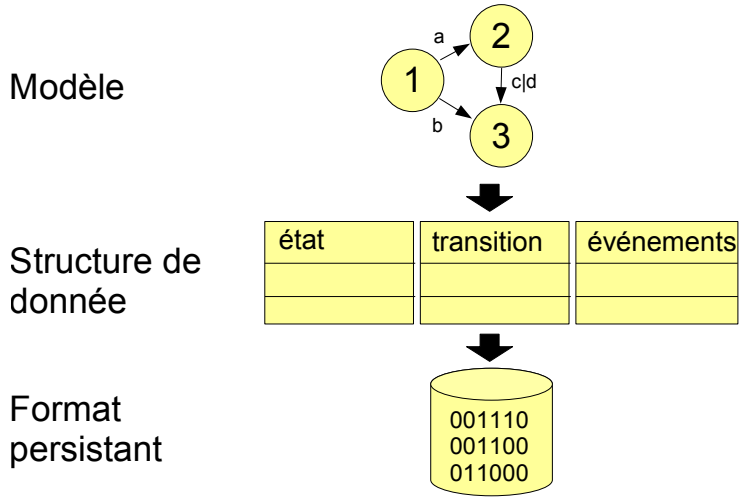

FIGURE 3.4 – Eléments constitutifs d'une représentation interne

peut être associée à la structure de données, comme nous le verrons plus loin. Les différents éléments constitutifs d'une représentation interne (dans ce cas, pour la représentation du comportement) sont représentés à la Figure 3.4.

L'architecture n'est pas entièrement statique, elle contient elle même des éléments de comportement, indépendants de ceux de l'application. Ainsi, l'infrastructure de communication contient un arbitre qui possède son propre comportement. La plupart des éléments des couches HDS et HAL présentées à la section 2.3.1 possèdent leur propre comportement : l'ordonnanceur, les pilotes matériels, le programme de bootstrap (auto-amorçage du processeur), etc. De même, l'application possède sa propre architecture, qui peut correspondre dans un certaine mesure à l'architecture cible. Il apparaît évident que la séparation entre application et architecture est loin d'être claire.

La mise en relation entre une application et une architecture est un processus complexe qui a été au coeur de nombreuses recherche dans le domaine des *méthodes de conception*. Cette problématique est notamment à l'origine du concept de *platform-based design*, où une architecture pré-conçue et une application sont mises en relation par un processus itératif dit *meet-in-the-middle* [54]. Le focus du présent travail n'est pas méthodologique mais plutôt sémantique. Nous nous intéresserons donc, dans ce chapitre, aux représentation qui permettent de manipuler les concepts d'application et d'architecture, et plus particulièrement les composants logiciels, matériels et mixtes. Cette étude est indépendante des méthologies de conception.

Les représentations internes de composants mixtes logiciel / matériel constituent un domaine de recherche très actif. Cette section décrit l'état de l'art sur les représentations internes les plus appropriées pour représenter les éléments d'interfaces mentionnés à la Section 2.3.2.

3.3.2 Représentations formelles

Les représentations décrites dans cette section sont pour la plupart issues de la théorie des graphes. Elles permettent l'utilisation de modèles mathématiques pour la synthèse et la vérification de circuits complexes. Plusieurs outils et méthodes de conception de SoC utilisent ces graphes sous différentes incarnations.

Le réseau de Petri fait partie de ces graphes mis au service du design électronique. Proposé en 1962 par le Prof. Carl-Adam Petri dans sa thèse de doctorat, ce modèle formel, graphique et exécutable est utilisé pour représenter des systèmes à événements discrets. L'environnement de design et de co-simulation proposé par Stoy et Peng [47] utilise des réseaux de Petri comme représentation unifiée de composants logiciels et matériels. Le langage utilisé pour l'implantation de cette représentation est Prolog, pour sa syntaxe proche de la théorie des ensembles. Lors d'une co-simulation, tous les événements qui se produisent au sein des réseaux de Petri à un instant donné sont stockés dans une base de données, ce qui fournit une photo détaillée de l'état du système. Cette approche repose donc sur une interprétation du design sous forme de réseaux de Petri, ainsi que sur l'utilisation d'un simulateur et ordonnanceur d'événements externe.

L'automate à états finis (Finite State Machine) est un modèle formel très utilisé pour les systèmes embarqués en général, et les systèmes sur puce en particulier. Une variation particulièrement répandue est le *Statechart* de Harel [25]. Ce type de diagramme d'état a été intégré au langage UML. Deux implémentations commerciales basée sur les diagrammes de Harel, StateMate (TeleLogic [26, 48]) et StateFlow (The Mathworks [50]), permettent de modéliser les flots de contrôle et de données à l'intérieur d'un système. Les deux outils proposent également la génération de code de prototypage et ainsi que de documentation à partir des diagrammes. Ces outils sont *a priori* destinés à la modélisation à haut niveau ainsi qu'à la génération de logiciel de contrôle, sans attention particulière à la distinction matériel - logiciel.

Une déclinaison intéressante de l'automate d'état est le CFSM (Codesign Finite State Machine [12]), où plusieurs automates d'état communiquent entre eux via l'envoi de message. Les CFSM apportent une solution haut niveau à l'une des principales difficultés des systèmes hétérogènes, à savoir la communication entre logiciel et matériel, sans y apporter de solution au niveau implémentation.

La représentation pour système VLSI (Very Large Scale Integration) proposée dans [31] est un bon exemple des premiers efforts pour unifier l'information de design en une seule structure de données. La représentation proposée permet à la fois la capture du comportement d'un système VLSI, sa structure, ses contraintes physiques ainsi que temporelles. Ce travail avait comme principal objectif de réunir tous ces aspects d'un système dans une structure de donnée homogène.

SPI (System Property Intervals) est un modèle qui permet l'analyse et l'optimisation de

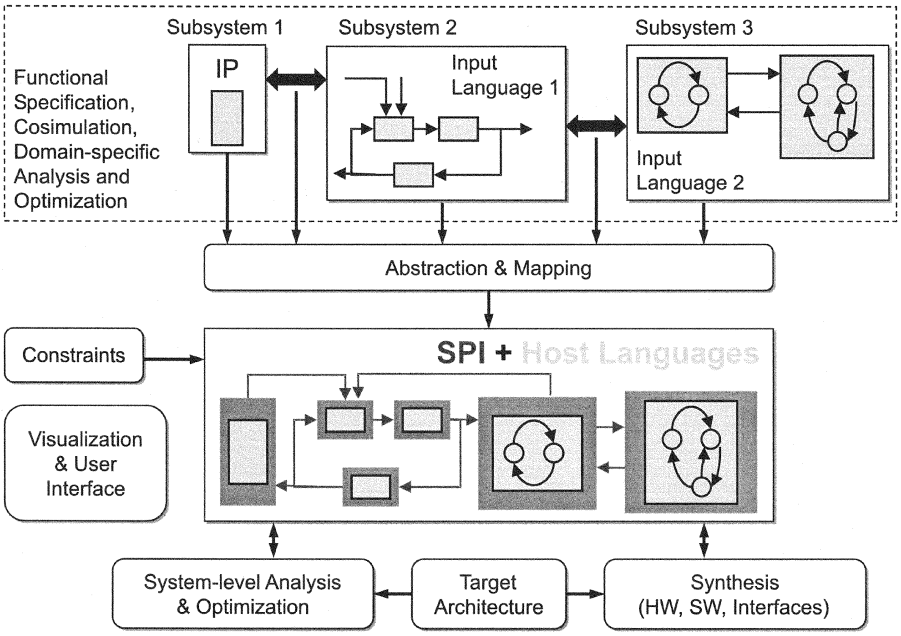

FIGURE 3.5 – Structure du workbench SPI

systèmes hétérogènes, sans frontières de langages [61]. Le modèle, basé sur des processus communicants paramétrables, permet aussi la synthèse de composants de contrôle ou de flot de données. Sa sémantique repose sur les intervalles de comportement (behavioral intervals), qui décrivent les débits d'entrées et de sorties des processus et des communications. Cette description est indépendante de l'implantation du composant en logiciel ou en matériel. Un workbench qui permet l'analyse et la synthèse de système hétérogènes sur la base de SPI a été mis sur pied (illustré à la Fig. 3.5). Le flot SPI prend en entrée des fonctions, décrites dans différents langages (Matlab, VHDL, SDL, etc.). Toutes les informations pertinentes à l'analyse et à la synthèse sont exprimées en SPI : l'utilisation des ressources, les canaux de communications, le timing. SPI n'est pas exécutable, mais fournit toutes les informations au workbench pour une analyse dynamique du système modélisé. En sortie, le workbench propose un ordonnancement des processus (au niveau système) ainsi que des estimations de performance de timing.

Funstate, qui peut être vu comme un raffinement de SPI [61], est une représentation interne qui permet la représentation homogène des flots de contrôle et de données. À nouveau, l'accent est mis sur une distinction fonctionnelle (contrôle vs flot de données) plutôt que sur l'abstraction de l'interface logiciel - matériel. Le processeur ne fait l'objet d'aucune attention particulière.

Le modèle BIP propose une modélisation de composants hétérogènes par la superposition de trois couches : "Behavior", "Interactions" et "Priorities" [5]. BIP permet la représentation des processus synchrones (dataflow et matériel) et asynchrones (threads logiciels) via une sémantique commune. Le composant BIP de base contient un automate d'états et des ports d'entrée / sortie associés aux transitions. Toutes les interactions sont modélisées par

des communication de type rendez-vous ou broadcast. Cette recherche mentionne également l'interaction entre composants décrits à des niveaux d'abstraction différents. Le modèle BIP est construit, exécuté et analysé par des outils externes. Le modèle est conçu pour permettre une validation formelle du comportement système, sans définir toutefois de méthode de synthèse des interfaces ni de lien avec des langages d'entrées courants.

Centré sur la modélisation du comportement, le travail de Camposano [10] utilise une représentation interne basée sur des graphes pour la synthèse à haut niveau. Ce type de synthèse n'est pas adapté pour la représentation de niveau système où co-existent composants logiciels et matériels.

Le projet Metropolis [4] représente un peu la synthèse de tous ces modèles formels. L'objectif de ce projet est de fournir une infrastructure unique pour la modélisation de systèmes hétérogènes et de permettre l'utilisation d'outils de design formels. Au coeur de l'infrastructure se trouve un méta-modèle qui permet la capture de différents modèles de calcul. Les modèles exprimés à l'aide de ce méta-modèle sont ensuite traités par différentes méthodes formelles intégrées dans l'infrastructure. En s'appropriant les informations de design, l'environnement Metropolis est en mesure de fournir sa propre sémantique d'exécution, ce qui rend son ouverture vers d'autres outils, plateformes et langages plus difficile.

3.3.3 Autres représentations

Le UML (Unified Modeling Language), ainsi que le SysML (System Modeling Language), tous deux supportés par le consortium OMG (Object Management Group, [38]), offrent d'intéressantes possibilités pour la représentation de systèmes hétérogènes. Ces langages de modélisation proposent des diagrammes et des tableaux qui permettent la capture les aspects structurels et comportementaux de composants logiciels et matériels. Plusieurs profils UML ont été proposés pour répondre avec précision aux besoins de modélisation des sytèmes temps réel et des SoC, comme le profil MARTE [39]. Certains s'inscrivent dans une approche de design plus générale appelée MDA (Model Driven Architecture), où les modèles UML servent de fil conducteur au flot de design [7]. Malgré le fait que la syntaxe et la sémantique de ces langages soient normalisées par l'OMG, leur exécution et leur stockage persistant ne le sont pas. Ainsi, il existe presque autant de format de stockage d'UML que d'outils sur le marché. Le format XMI (XML Model Interchange), proposé comme format d'échange de modèles entre les outils, n'a pas encore réussi à s'imposer à cet égard à cause de sa trop grande flexibilité [32]. L'interopérabilité entre les outils UML est donc possible mais partielle.

SystemC [40] est une bibliothèque C++ utilisée pour la simulation de systèmes parallèles, standardisée en 2005 par l'IEEE sous le nom de Standard 1666. La bibliothèque fournit les éléments nécessaires pour représenter des composants matériels à différents niveaux d'abstraction. La structure autant que le comportement peuvent être exprimés en objets SystemC, le langage C++ agissant comme format de stockage des informations de design (sous forme

de code source). Comme les autres langages de description matérielle, l'exécution d'un modèle SystemC repose sur un simulateur externe qui gère les processus concurrents. La syntaxe C++ très flexible de SystemC en fait un langage difficile à analyser et transformer, ce qui fait qu'en dehors de la simulation, les outils SystemC sont rares et coûteux à développer et maintenir.

Basé essentiellement sur SystemC, l'environnement de développement SpaceStudio [11] propose une plateforme virtuelle au niveau transactionnel (TLM) pour l'exécution de logiciel. L'objectif premier de cet environnement est de faciliter l'exploration d'architecture, en facilitant la migration des tâches logicielles en matériel, en vue d'obtenir le partitionnement logiciel-matériel optimal. La plateforme est bâtie autour du système d'exploitation μC/OS-II et propose un chemin d'implantation vers les technologies FPGA ou système sur puce.

Le format IP-XACT [43] du consortium SPIRIT est une représentation dont le but est de faciliter les échanges et la réutilisation de blocs IP (Intellectual Property). Le format, basé sur un schéma XML (Extended Markup Language), permet la spécification d'architectures hiérarchiques, tout en proposant une interface normalisée pour des outils de génération et de configuration d'IP. Dédié à la représentation des composants matériel au niveau RTL et au niveau transactions, le format est directement compatible avec les langages VHDL, Verilog et SystemC. La représentation des composants logiciels ainsi que des relations logiciel - matériel n'est pas définie. Par contre, IP-XACT permet d'associer des données utilisateurs à certains éléments du schéma via les *parameters* et les *vendorExtensions*. Il est donc possible d'augmenter la sémantique du format pour représenter le monde logiciel, moyennant les outils adéquats pour l'interpréter ainsi qu'une sémantique IP-XACT de base toujours valide.

Également basé sur un schéma XML, le format DOL (Distributed Object Layer) décrit l'association entre un réseau de processus communicants et une plateforme multiprocesseur [51]. Le principal objectif de DOL est d'utiliser des méthodes formelles pour l'exploration d'architecture et l'analyse de performance, permettant une distribution des tâches et canaux optimale sur en fonction d'une plateforme cible.

Le Prof. Stephan A. Edwards propose, dans ses travaux, le langage SHIM (Software / Hardware Integration Medium), qui apporte une solution à la discontinuité entre les représentations logicielle et matérielle au niveau implantation. SHIM permet de capturer les éléments de bas niveau de l'interface logiciel - matériel, notamment les pilotes de composants matériels. Les communications entre composants hétérogènes sont modélisées à l'aide de canaux point-à-point selon le modèle de communication par rendez-vous. Ces travaux répondent en partie aux requis mais n'offrent pas la flexibilité voulue pour la représentation de l'interface logiciel-matériel à plusieurs niveaux d'abstraction.

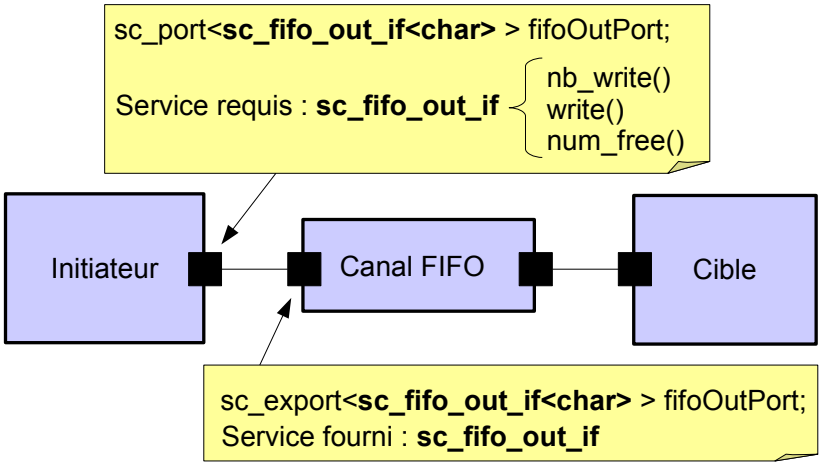

FIGURE 3.6 – Connexion de type FIFO illustrant la relation entre service et port en SystemC

3.3.4 Représentation des services

Quelques unes des représentations citées ci-dessus intègrent la notion de service. C'est le cas par exemple de SystemC, qui définit le service de la même façon qu'UML, c'est à dire sous la forme d'une interface logicielle. L'interface, ou le service, est un ensemble de méthodes associé à un port selon une relation *requis* ou *fourni*. L'exemple ci-dessous illustre la définition d'une interface d'écriture dans un canal FIFO, *sc_fifo_out_if* [40].

```
template <class T>
class sc_fifo_nonblocking_out_if : virtual public sc_interface
{
  public:
    virtual bool nb_write( const T& ) = 0;
    virtual const sc_event& data_read_event() const = 0;
};

template <class T>
class sc_fifo_blocking_out_if : virtual public sc_interface
{
  public:
    virtual void write( const T& ) = 0;
};

template <class T>
class sc_fifo_out_if : public sc_fifo_nonblocking_out_if<T>,
                       public sc_fifo_blocking_out_if<T>
{
  public:
    virtual int num_free() const = 0;
  protected:
    sc_fifo_out_if();
  private:
    // Disabled
    sc_fifo_out_if( const sc_fifo_out_if<T>& );
    sc_fifo_out_if<T>& operator= ( const sc_fifo_out_if<T>& );
};
```

On associe cette interface à un sc_port ou un sc_export, selon que le service est requis ou fourni. On dit que le sc_port requiert une interface et que le sc_export fourni une interface, ou un service. Pour l'objet contenant le port, les méthodes de l'interface sont accessibles via le port, ce qu'illustre la Figure 3.6.

```
<spirit:port>
  <spirit:name>fifoOutPort</spirit:name>
  <spirit:transactional>
    <spirit:service>
      <spirit:initiative>requires</spirit:initiative>
      <spirit:serviceTypeDefs>
        <spirit:serviceTypeDef>
          <spirit:typeName>sc_fifo_out_if</spirit:typeName>
          <spirit:parameters>
            <spirit:parameter spirit:name="char" spirit:resolve="user">char
            </spirit:parameter>
          </spirit:parameters>
        </spirit:serviceTypeDef>
      </spirit:serviceTypeDefs>
    </spirit:service>
  </spirit:transactional>
</spirit:port>
```

FIGURE 3.7 – Spécification d'un port transactionnel et d'un service requis en IP-XACT

Du fait de sa compatibilité avec SystemC, IP-XACT intègre le service exactement de la même façon : en tant qu'interface associée à un port transactionnel, tel que l'illustre l'extrait XML de la Figure 3.7.

Cette vision du concept de service permet de spécifier les opérations possibles via un port ainsi que la dépendance du port par rapport au service. Cependant, elle ne permet pas de spécifier les caractéristiques d'interdépendance entre les ports, ni les caractétistiques de plateforme et de performance, qui sont si importantes dans la conception de systèmes sur puce. C'est un des principaux objectifs de cette thèse de proposer un modèle de service étendu qui englobe ces différents aspects.

3.4 Conclusion

Nous avons présenté, dans ce chapitre, l'état de l'art sur les deux principaux axes de recherche de cette thèse : la modélisation via le concept de service ainsi que les représentations intermédiaires pour le design de systèmes sur puce. La pertinence d'une approche service pour l'abstraction de l'interface logiciel matériel a été démontrée, sans qu'un format pratique pour son application aux design de systèmes sur puce ait été trouvé.

Le manque de représentation unifiée pour l'interface logiciel-matériel à plusieurs niveaux d'abstraction a été mis en évidence. Plusieurs représentations existent mais elles se concentrent pour la plupart autour d'un même niveau d'abstraction ou d'un même type de traitement. D'autres présentent des difficultés à être analysées et éditées par différents outils logiciels.

Une représentation permettant la description de l'interface logiciel-matériel à plusieurs niveaux d'abstraction selon une approche service sera proposée dans la suite de ce document.

Chapitre 4

Le service comme abstraction de l'interface logiciel - matériel

Sommaire

4.1 Introduction 38
4.2 Utilisation des services pour la conception de systèmes hétérogènes monopuce 38
4.3 Proposition d'un modèle de service pour systèmes hétérogènes 39
4.3.1 Les objets structurels 40
4.3.2 Description des ports logiques 44
4.3.3 Définition du service 45
4.3.4 Le modèle de service 46
4.3.5 Services reliés au module hybride 49
4.4 Esquisse d'un flot de conception orienté services 51
4.4.1 Principaux éléments du flot 51
4.4.2 Rôle de la bibliothèque de services 52
4.4.3 Sélection - intégration des services 53
4.5 Conclusion 57

4.1 Introduction

Ce chapitre présente le premier élément de contribution de cette thèse : un modèle des services adapté pour l'abstraction de l'interface logiciel-matériel. L'objectif est de développer un modèle cohérent, facile à manipuler et à raffiner et qui permette le développement d'outils d'analyse et de génération d'interfaces logiciel-matériel.

Rappelons que l'interface logiciel-matériel est constituée de différents composants, plus ou moins implicites selon le niveau d'abstraction considéré (Section 2.3.1) : le HDS (Hardware-Dependent Software, qui contient l'OS et les pilotes matériels), le HAL (Hardware Abstraction Layer), le sous-système processeur et le processeur lui-même. Le modèle proposé doit permettre de raisonner sur l'ensemble de ces composants.

Le modèle des services présenté dans ce chapitre servira de base à la représentation intermédiaire, en complément des spécifications déjà présentées au Chapitre 2.

4.2 Utilisation des services pour la conception de systèmes hétérogènes monopuce

Le chapitre sur l'état de l'art a permis d'apprécier les avantages associés à l'approche orientée service en général. Les outils de manipulation de l'interface logiciel-matériel pourraient bénéficier de cette approche, à condition de bien identifier les processus de conception compatibles. Malgré les similitudes entre les applications traditionnelles de l'approche SOA et les applications destinées à une implantation sur puce (hétérogénéité, parallélisme, distribution), les contraintes de performances imposées à ces dernières exigent une approche orientée service sur mesure.

Le principal avantage du service est qu'il permet d'abstraire et de modéliser les deux aspects suivants des composants d'un système :

- Modélisation des liens inter-composants. Ces liens peuvent être explicite, comme dans le cas des des canaux de communication, ou implicite, comme dans le cas de liens structurels dynamiques (par ex., le lien entre une tâche et le processeur qui l'exécute).
- Modélisation des caractéristiques non fonctionnelles des composants, comme le débit de traitement de l'information, le taux d'erreur, les marges de tension en alimentation, les marges de fréquence, etc.

Nous soutenons l'hypothèse que l'abstraction d'interface via les services faciliterait la composition d'éléments hétérogènes et la génération assistée d'interfaces logiciel-matériel. La notion de service existe à tous les niveaux d'abstraction, constituant un élément liant idéal pour consolider un flot de raffinement et de génération.

En plus d'exister à tous les niveaux d'abstraction, la notion de service est indépendante de toute implémentation. Puisqu'il abstrait une interface, le service peut être réalisé soit en matériel, soit en logiciel, pour autant que ces implémentations présentent les mêmes caractéristiques d'interface. On peut ainsi spécifier une communication à travers une frontière logiciel-matériel grâce à la spécification d'un service unique.

La Figure 4.1 représente des exemples de connexion de modules hétérogènes à l'aide d'une spécification de service. Au niveau système, les communications s'effectuent par canaux abstraits, la cible exacte des modules (logiciel ou matériel) n'étant pas encore connue. Les interfaces requises par les modules initiateur et cible peuvent être exprimées sous forme de services : le module initiateur requiert par exemple un service de type PUT, et le module cible un service de type GET. Cette spécification de service restera valide tout au long du design, car quoi qu'il arrive, ces deux tâches doivent communiquer entre elles.

Le partitionnnement logiciel-matériel vient ensuite préciser la cible de chaque module. Dans cet exemple, trois partitionnements différents sont effectués, résultant en trois différentes architectures virtuelles.

La spécification de service originale doit permettre de « résoudre » la communication entre ces modules désormais hétérogènes. La cible (logiciel ou matériel) des modules en relation ainsi que les caractéristiques de plateforme du service déterminent la composition exacte de l'interface : (Archi 1) un *pipe* POSIX pour une connexion inter-processus, (Archi 2) une mémoire partagée pour une communication inter-processeur, ou (Archi 3) un canal FIFO reliant un composant logiciel à un composant matériel via des adaptateurs. Le service permet donc une *description homogène* de la communication entre *éléments hétérogènes*, description sur laquelle une analyse peut être effectuée en vue de la génération.

En spécifiant différentes caractéristiques de communication et caractéristiques non-fonctionnelles d'un composant, le service permet une meilleure réutilisation des composants. D'une part, il est possible de regrouper des services selon différents critères (type de fonction, plateforme, niveau d'abstraction) afin de créer des **bibliothèques de services**. D'autre part, le service peut décrire certains aspects paramétrables d'un composant, en spécifiant les valeurs possibles. Le service joue donc un rôle de *documentation active* autour de laquelle s'organise la génération d'interface.

4.3 Proposition d'un modèle de service pour systèmes hétérogènes

Le modèle proposé repose sur deux types d'objets : les objets dits structurels et les objets propres aux services. Cette section présente chacun des types ainsi que le modèle de l'objet pivot de l'interface logiciel-matériel, le module hybride.

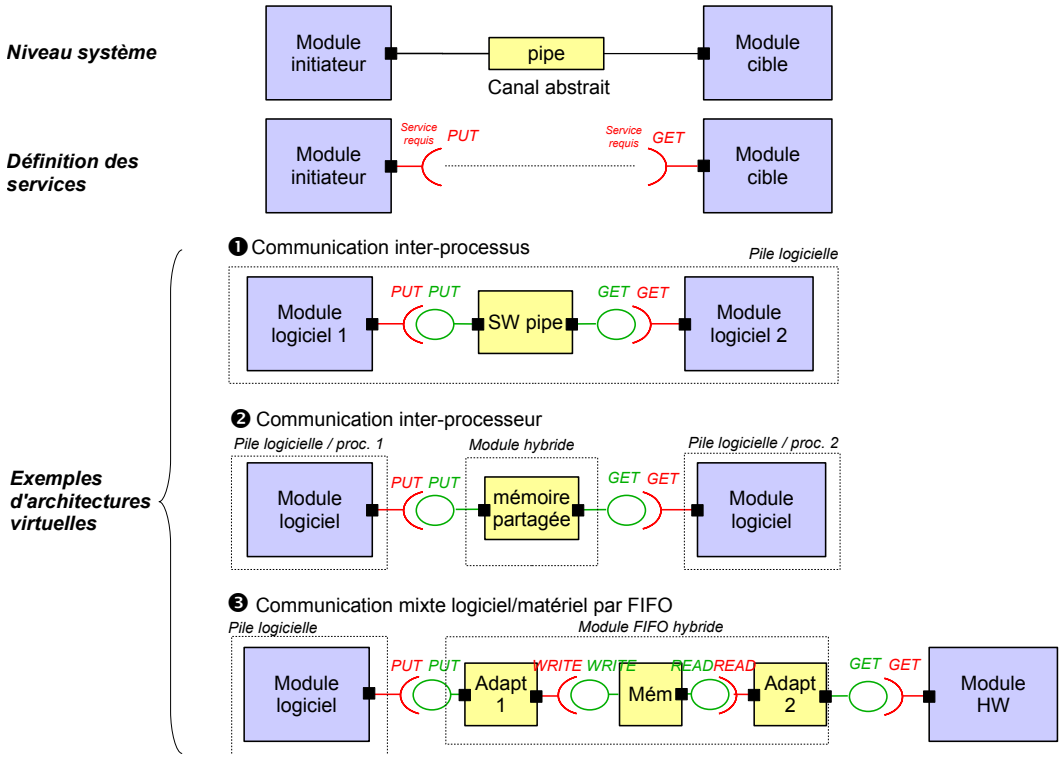

FIGURE 4.1 – Connexion entre composants hétérogènes grâce à une spécification de service

4.3.1 Les objets structurels

Les objets structurels doivents permettre la capture de la structure hiérarchique d'un design ainsi que des interconnexions, que les services viendront ensuite compléter. Ils ne sont donc pas spécifiques à l'approche orientée service et se retrouvent dans plusieurs langages de description de systèmes intégrés.

– Le **module** représente une unité hiérarchique de design. Dans notre modèle, il délimite une unité fonctionnelle, qu'il s'agisse d'un composant logiciel, matériel ou hybride, ayant le potentiel d'offrir ou de requérir un service. Le module sert donc à expliciter l'interface d'une unité fonctionnelle et ses interactions avec l'environnement. La granularité d'un module peut être aussi fine que le justifie sa réutilisation. Un module peut être hiérarchique et contenir un nombre illimité de sous-modules, chacun représentant une instance de module. Ce lien hiérarchique peut être statique (exemple : un OS associé à un processeur) ou dynamique (exemple : une tâche qui migre d'un processeur à un autre dans un système SMP [1]). Un module possède une interface, visible de l'extérieur, et un contenu, invisible de l'extérieur.
– Le **port** représente un point d'échange d'information à travers l'interface d'un module. La notion de port est très importante pour la modélisation des systèmes et se retrouve dans la majorité des langages de conception. Un port contient la signature de la méthode

1. Symmetric Multiprocessor, type d'architecture ou plusieurs processeurs identiques servent à l'exécution d'un ensemble de tâches

offerte ou requise par ce port. Dans le cas d'un port sur un module matériel ou d'un port de type get/set, cette méthode est simplement *read()* ou *write()*. Dans le cas d'un port sur un module logiciel, il peut s'agir d'une méthode avec plusieurs paramètres d'entrées et/ou de sortie. On peut regrouper plusieurs ports en un **port logique** pour définir un protocole d'échange ou une interface complexe. Les ports appartiennent à l'interface d'un module et sont visibles à la fois de l'intérieur et de l'extérieur. Un sens peut être associé à un port afin d'améliorer l'expressivité des schémas et de permettre une validation simple des modèles.

– Le connecteur ou **net** représente une mise en relation de ports. Le connecteur peut relier des ports appartenant à des modules de même niveau hiérarchique ou appartenant à un sous-module et à son conteneur (connexion hiérarchique). Un connecteur fait partie du contenu d'un module, au même titre qu'un sous-module.

Le Module étant l'élément structurel le plus important, il mérite une analyse plus approfondie, présentée dans les paragraphes qui suivent.

Sémantique du module

Le module est une unité fonctionnelle du design qui représente un composant logiciel, matériel ou hybride d'un système sur puce, à plusieurs niveaux d'abstraction. Ces trois types de composants ont des sémantiques différentes selon le niveau considéré.

Nous allons donc définir le concept *de cible de module*, afin de capturer la nature intrinsèque du composant représenté : logiciel (SW), matériel (HW) ou hybride (HYBRID). Cependant, plusieurs traitements liés au processus de design exigent une sémantique plus précise : validation d'architecture, spécification et génération de connexions point-à-point, génération de moniteurs de debug, etc. Afin de mieux caractériser les interfaces des modules SW et HW, il est important d'introduire également le concept de *fonction de module*. Ce deuxième attribut permet de différencier les composants de communication (canal, bus), les processus, les processeurs, les éléments de mémoire, etc. Le Tableau 4.1 présente les différentes fonctions pour les cibles de type SW et HW qui ont été identifiées dans le cadre de cette thèse.

Un module de cible SW et de fonction PROCESS doit pouvoir être associé de façon indépendante à un module de type HYBRID. Ceci exclut les *threads*, qui sont destinés à être exécutés sur un seul processeur (ou du moins un même groupe de processeur SMP). La fonction d'un module hybride est implicite : il s'agit d'établir une communication interne entre son interface logicielle et son interface matérielle.

Plusieurs de ces modules possèdent leur propre fil d'exécution, permettant une exécution parallèle. C'est le cas de tous les modules matériels. Pour les modules logiciels, le fil d'exécution doit être explicitement créé. Il est important de permettre la représentation des modules logiciels qui ne disposent pas de leur propre fil d'exécution pour rendre le modèle structurel plus homogène et faciliter l'accès à tous les types de services. Ceci pourra être utile, par exemple, pour représenter une variable globale ou une libraire de l'OS auxquelles plusieurs

TABLE 4.1 – Fonctions possibles pour un module logiciel ou matériel

Fonction	**Description**	**Ex. cible SW**	**Ex. cible HW**
PROCESS	Réalise une transformation de données	Programme "main"	IP spécifique
PROCESSOR	Exécute un processus	Simulateur ou émulateur	Processeur RISC, DSP
CHANNEL	Communication point à point	Pipe, socket	FIFO
BUS	Communication point multi-point	Middleware CORBA, infrastructure SOA	AMBA-AHB
PERIPH	Interaction avec système ou usager externe	Console d'entrée / sortie	Clavier
SYNC	Synchronisation entre processus	Mutex	Spinlock
MEMORY	Stockage de données temporaire ou persistant	Tampon en mémoire	SDRAM
MANAGEMENT	Gestion et exploitation des ressources logicielles ou matérielles	HAL, OS	Mutex
ABSTRACT	Organise le design de façon hiérarchique. Absent de l'implémentation.	Pile logicielle associée à un module hybride	Fichier top d'un design hiérarchique
TEST	Génère des stimuli ou recueille des données pour la validation. Absent de l'implémentation.	Affichage de données sur terminal	Générateur de trafic

modules logiciels ont accès.

L'association tâche logicielle - processeur

Il existe différentes associations entre une tâche logicielle (cible = SW, fonction = Processus) et un processeur (cible = HW, fonction = Processeur). Dans le cas le plus simple, une tâche peut être assignée « à résidence » sur un processeur. Outre le cas trivial des systèmes monoprocesseurs, ce genre d'association caractérise les tâches de gestion des ressources, comme les tâches de HDS ou de HAL (fonction = Gestion). Certaines tâches de transformation (fonction = Processus) peuvent également être associées ainsi à un seul processeur. Ce premier cas pourrait être simplement illustré à l'aide d'une relation hiérarchique entre le processeur, qui deviendrait conteneur, et la tâche, qui deviendrait sous-module.

Dans le cas le plus complexe, on ne connaît pas d'avance quelle tâche sera exécutée sur quel

processeur. C'est le cas de l'architecture SMP, où l'exécution d'une tâche donnée par un processeur est déterminée dynamiquement par le système d'exploitation. La relation hiérarchique n'est pas appropriée pour la capture de ces associations dynamiques. Un moyen plus souple et plus efficace doit être mis en place pour exprimer ce genre de relation. Rappelons que l'objectif final est de générer un prototype exécutable du système à différents niveaux d'abstraction, ce qui implique l'insertion d'un ordonnanceur de type SMP dans le modèle. Cet ordonnanceur doit avoir connaissance, d'une part, des tâches à exécuter et, d'autre part, du groupe de processeurs candidat à leur exécution. On peut donc déterminer à l'avance quel groupe de tâche (cible = SW, fonction = ABSTRACT) est exécuté sur quel groupe de processeurs (cible = HW, fonction = ABSTRACT). Cette information de mapping pourra être capturée à l'aide d'un service « exécution », tel qu'il le sera présenté à la section 4.3.5.

Définition - instanciation du Module

Dans le contexte des systèmes sur puce (ou même des systèmes embarqués complexes), il est fréquent qu'une même spécification de tâche logicielle ou matérielle servent plusieurs fois dans un système. Le degré de reproduction varie en fonction du niveau d'abstraction. Au niveau système, on peut retrouver plusieurs « instances » d'un même processeur, d'une même interface d'entrée / sortie, etc. Au niveau implantation, on peut retrouver plusieurs instances d'un même registre, d'un même compteur, etc. Tout ces modules sont représentés, à plusieurs niveaux d'abstraction, par le même objet de type module.

Il est donc proposé, pour le modèle des objets structurels, d'utiliser un mécanisme de *définition / instanciation*. L'objectif de ce mécanisme est de permettre l'instanciation multiple d'un même objet, afin de faciliter l'organisation du design et d'encourager la réutilisation. Ce mécanisme est explicite dans plusieurs langages de modélisation et de programmation, comme SystemC (définition : SC_MODULE, instanciation : pointeur vers un SC_MODULE), C++ (définition : classe, instanciation : instance de classe) et VHDL (définition : "entity-architecture", instanciation : "component", "generate" et "configuration"). Il est également important que chaque instance puisse être paramétrée individuellement, comme le mécanisme "generic" en VHDL le permet.

Une troisième étape peut suivre celles de définition - instanciation : le déroulement complet de la hiérarchie. Ainsi, lors de la création d'une instance, toute la hiérachie des sous-instances contenues dans cette instance est créée de façon récursive. C'est ce qui correspond à l'étape d'élaboration ou de mise à plat des outils de compilation VHDL. Ce niveau d'information, très détaillé et habituellement utilisé pour la simulation, ne semble pas pertinent à représenter dans le modèle des objets structurels.

Les ports et les connecteurs ne sont pas jugés d'une complexité suffisante pour justifier une instanciation multiple à partir d'une seule définition. Seuls les modules et les ports logiques, les deux objets hiérarchiques du modèle, bénéficieront de ce mécanisme.

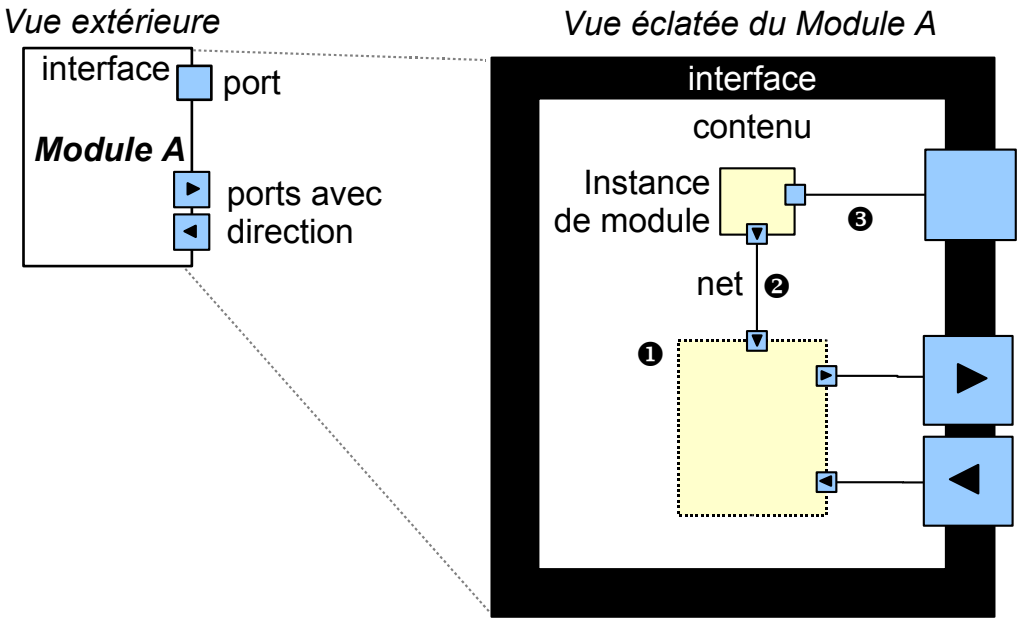

FIGURE 4.2 – Les objets structurels proposés : module-port-net

Les objets structurels mentionnés dans cette section sont illustrés à la Figure 4.2.

La vue extérieure illustre un module dont on ne voit que l'interface, constituée de ses ports. La vue éclatée révèle le contenu d'un module, composé d'instances d'autres modules et de nets. On distingue, dans cette dernière vue, un sous-module ne possédant pas son propre fil d'exécution représenté en trait pointillé (1), des connexions simples entre sous-modules (2) ainsi que des connexions inter-hiérarchie (3). Nous utiliserons cette figure comme convention graphique pour la représentation des modules, ports et nets, dans la suite du document.

4.3.2 Description des ports logiques

Des ports simples peuvent être regroupés en un *port logique* pour définir une interface réutilisable, comme une interface de bus ou une interface logicielle faite de plusieurs fonctions. Il suffit de connecter deux ports logiques afin que tous les sous-ports soient automatiquement connectés. La connexion individuelle de ports appartenant à un port logique n'est pas permise vu de l'extérieur d'un module.

Vu de l'intérieur d'un module, chaque port simple membre du port logique doit pouvoir être manipulé individuellement. On observe donc encore une fois l'application du principe d'encapsulation qui cache les détails d'implémentation au monde extérieur.

Le port logique a trois états. Son premier état est une *définition* qui contient tous les ports liés au port logique. Son deuxième état est une *instance* associée à la définition d'un module. Cette instance contient seulement les ports effectivement utilisés par un module ainsi qu'un

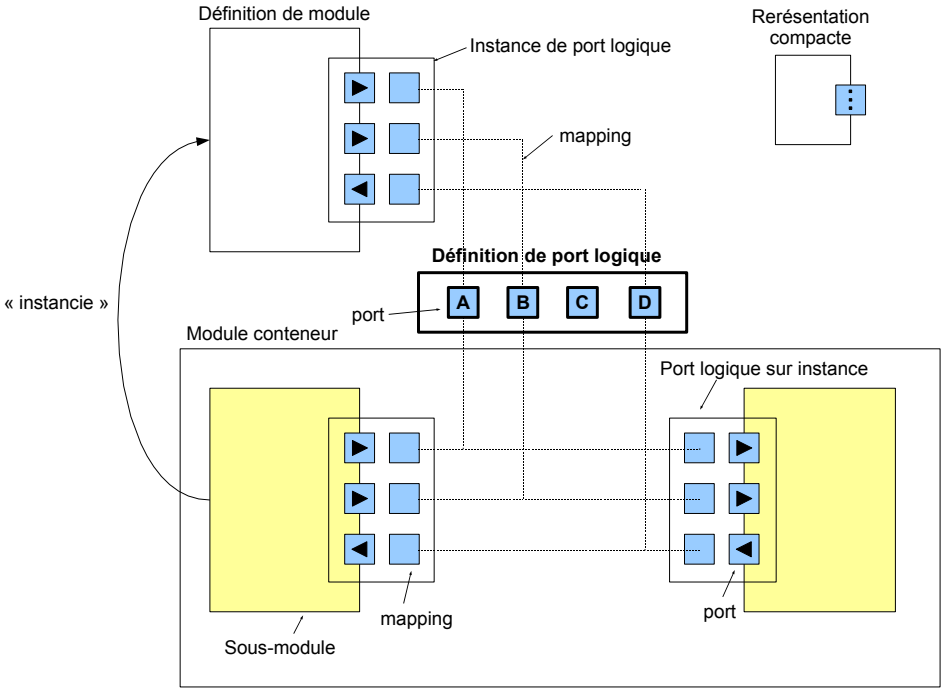

FIGURE 4.3 – Définition et instance d'un port logique

mapping entre les ports présents sur le module et les ports définis dans le port logique. Son troisième état est similiare au deuxième : c'est une instance mais cette fois sur un sous-module. Ainsi, lors de la connexion de ports logiques entre deux sous-modules, les sous-connexions sont réalisées en connectant les sous-ports faisant références au même sous-port de la définition.

La Figure 4.3 illustre la définition d'un port logique, son instanciation sur une définition de module, ainsi que sa deuxième instanciation sur deux sous modules. Le mapping entre les ports des sous-modules et le port de la définition est également illustré. Le port C de la définition n'est pas utilisé dans cet exemple d'interconnexion. Seuls les ports appartenant au module ont une direction, les ports du port logique n'en ont pas. L'interconnexion des deux instances de port logique impliquera l'interconnexion des ports A, B et D.

4.3.3 Définition du service

Il convient de fournir la définition du concept de service qui servira de ligne directrice à l'élaboration du modèle. Un service est l'abstraction d'un aspect externe du module, qu'il s'agisse d'un module logiciel, matériel ou hybride. Un service implique soit un échange d'information entre deux modules, soit un lien structurel dynamique entre deux modules. Un service spécifie également des caractéristiques non-fonctionnelles comme les contraintes de plate-forme et de performance. Le concept de service se situe au centre de notre modèle et doit permettre la génération des interfaces logiciel-matériel. C'est donc un *élément de procédé*, utilisé par des outils spécialisés et indépendant du design lui-même.

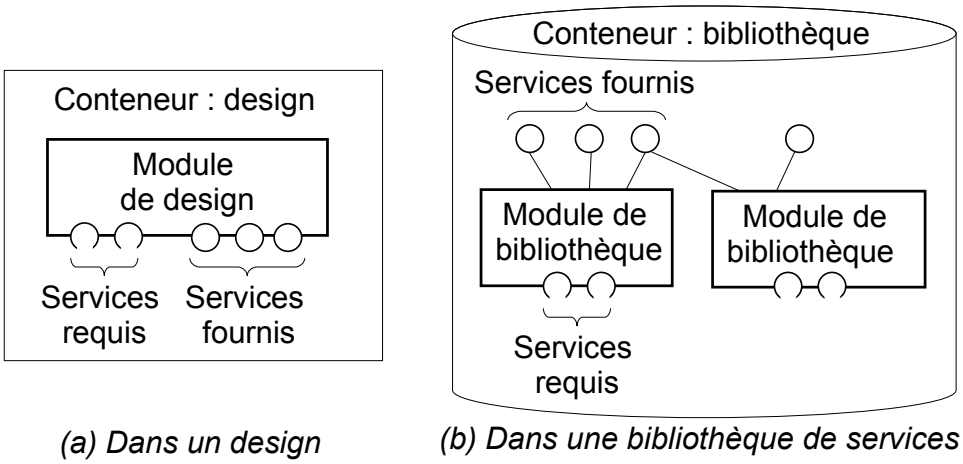

FIGURE 4.4 – Relations possibles entre services et modules

La relation service fourni - service requis est également définie, ce qui permet une interconnexion souple entre les modules hétérogènes. Cette relation sera éventuellement remplacée (ou résolue) par une interconnexion raffinée ou une dépendance fonctionnelle, selon une procédure décrite plus loin. L'objectif de couplage faible entre les modules est donc directement atteint par l'utilisation de spécifications de service requis-fourni.

Enfin, il est proposé qu'un service soit associé à un module d'une façon plus ou moins souple, selon le contexte :
– *Dans un design* (Figure 4.4(a)) : le service appartient à l'interface d'un module, et donc, par extension, au module lui-même puisque son interface est unique. Un service est associé à un seul module (cardinalité 1), tandis qu'un module peut proposer ou requérir plusieurs services (cardinalité 0..*).
– *Dans une bibliothèque* (Figure 4.4(b)) : une bibliothèque organise et publie les services offerts par des composants pré-conçus. Le service fourni appartient à la bibliothèque et est associé à au moins un module par le biais d'un identificateur unique. Un service peut être fourni par plusieurs modules, et un module peut fournir ou requérir plusieurs services. Cependant, un service requis reste associé à un seul module.

4.3.4 Le modèle de service

Un service est spécifié par plusieurs caractéristiques, regroupées en trois niveaux, sur la bases des caractéristiques d'interface présentées à la Section 2.3.2. Ces caractéristiques sont illustrées à la Figure 4.5 :

1. *Les caractéristiques de communication ou de composition*. Pour la communication, ce niveau doit permettre de spécifier les canaux d'entrée et de sortie d'un module, l'ordre dans lequel ils sont utilisés (le protocole) et si le service est actif ou passif (possède son propre thread ou non). Les caractéristiques de communication comportent un identificateur de fonction unique qui reflète la nature du service qui leur est associé. Cet

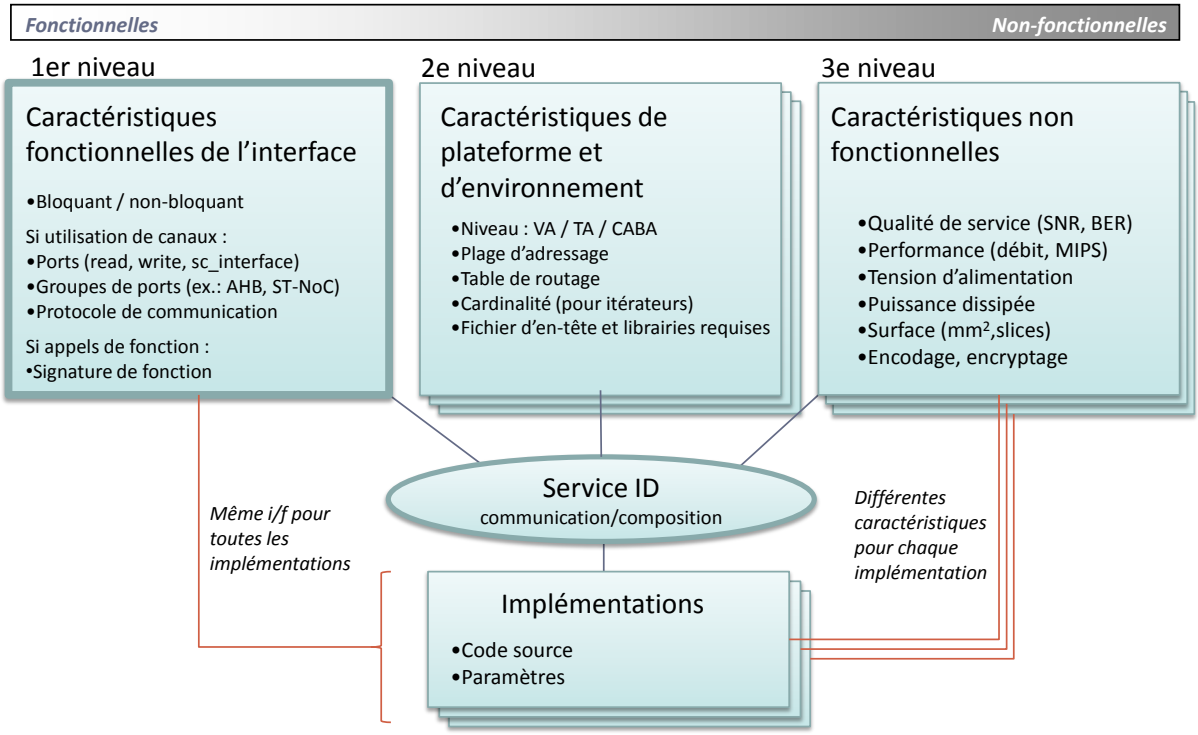

FIGURE 4.5 – Les principaux éléments d'une description de service

identificateur constitue la garantie d'une connexion homogène entre deux éléments complètement hétérogènes (par ex. le service FIFO). Les caractéristiques de composition permettent de spécifier une dépendance fonctionnelle, particulièrement adaptée à l'association d'une tâche et d'un processeur, ou d'une tâche et d'une librairie d'OS.

2. *Les caractéristiques de plateforme*. Ce niveau caractérise les ressources attendues pour la réalisation d'un service. On spécifiera par exemple la dépendance à un système d'exploitation ou à un élément matériel à ce niveau, ainsi que la plage d'adresses d'un module.
3. *Les caractéristiques de performance*. Ce niveau permet de spécifier différents paramètres quantitatifs ou qualitatifs liés au fonctionnement d'un service. On y spécifie par exemple les caractéristiques de latence, de débit, les paramètres de qualité de service, l'aspect sécurité, etc.

Un service contient un identificateur unique (Service ID) permettant de le localiser directement parmi tous les services disponibles dans une bibliothèque, à la manière de l'UUID (*Universally Unique Identifier*) utilisé pour les Services Web ([57]). Dans le cadre de cette recherche, cet identificateur est une concaténation des identificateurs des caractéristiques des trois niveaux.

Les trois niveaux de caractéristiques de la Figure 4.5 représentent un système de priorité dans le choix d'une implémentation. Le premier niveau est mis en évidence par un encadré plus épais car ce sont les caractéristiques de communication qui différencient les services entre

eux. Pour une même interface de communication, il peut y avoir différentes caractéristiques de plateforme. Par exemple, le service FIFO peut être implanté à l'aide d'un tableau en mémoire globale ou à l'aide de communication inter-processus, ce qui est transparent du point de vue des clients de la FIFO. Pour les mêmes caractéristiques de plateforme, il peut y avoir également plusieurs caractéristiques non fonctionnelles. Par exemple, une FIFO matérielle peut être implantée avec une protection contre les erreurs (ECC) ou non.

Le service founi

Le cas le plus simple à décrire est le service fourni. Dans ce cas, le maximum d'information provenant du module doit être utilisé pour spécifier les trois niveaux de caractéristiques. Un identificateur unique est attibué au service et celui-ci peut être inséré dans une bibliothèque. Rien n'empêche un module de fournir plusieurs services différents, ou un service d'être fourni par plusieurs modules différents.

Le service requis

Un service requis diffère d'un service fourni par le degré de précision de ses caractéristiques. Dans le cas le plus flexible, un service requis ne contient que l'identificateur de fonction (par ex. : une FIFO abstraite). Dans le cas le plus contraint, un service requis contient l'identificateur ainsi que les trois niveaux de caractéristiques complètement spécifiés. Dans tous les cas, le service requis est indépendant du service fourni. Un environnement de développement, tel que celui qui sera mis en oeuvre au Chapitre 6, doit permettre la mise en correspondance des services requis et fournis ainsi que les éventuelles adaptations nécessaires.

Visibilité et mise en relation des services

La visibilité d'un service par rapport à son environnement est définie de la même façon que pour les ports : soit un service associé à un module de niveau n, le service peut être référencé par un module de niveau n+1 (module conteneur), n (co-module) ou n-1 (sous-module). Des exemples de modules, services et mises en relation de services (*service mapping*) sont illustrés à la Figure 4.6.

La responsabilité de satisfaire une requête de service incombe à un module de même niveau que le module requérant (niveau n), tel qu'illustré par la relation (1) de la Fig. 4.6. Si un tel module n'existe pas, le service est exporté sur l'interface du module conteneur (niveau $n+1$) pour être satisfaite à l'extérieur (relation (2) dans la même figure). Les services fournis peuvent de même être exportés sur l'interface du module conteneur, tel qu'illustré par la relation (3). Enfin, la relation (4) représente un service ayant deux ports comme caractéristiques de communication. Ceci implique que le fournisseur du service possède exactement les mêmes ports, selon une configuration « miroir » (dont les directions sont opposées). Si

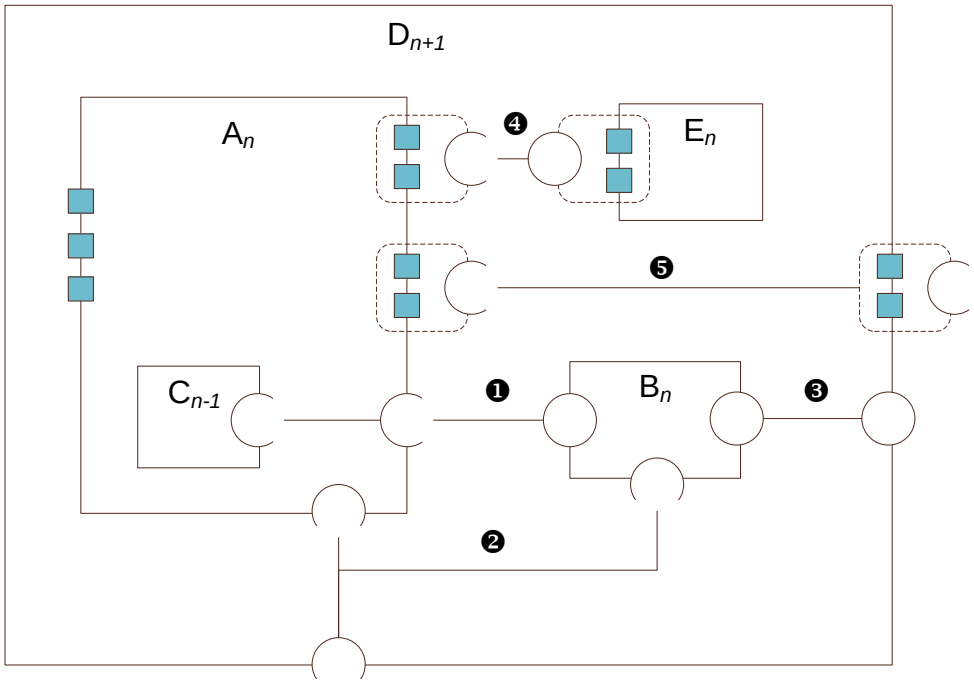

FIGURE 4.6 – Définition de la visibilité d'un service

ce service était exporté, l'interface du module conteneur devrait contenir ces ports, cette fois selon la même configuration (relation (5)). Cette figure représente tous les cas de dépendance de service considérés dans cette recherche.

Les caractéristiques de communication d'un service sont spécifiées avec des ports et ports logiques de la même façon qu'elles sont ajoutées à un module. Lorsqu'un service est associé à un module qui l'offre ou qui le requiert, un mapping est requis entre les ports du service et les ports du module impliqués dans le service. Ce mapping consiste à mettre en correspondance les identificateurs des ports dans le service et dans le module. La relation entre ports, ports logiques et services est illustrée à la Figure 4.7.

4.3.5 Services reliés au module hybride

Le module hybride est utilisé par un concepteur pour représenter une unité d'exécution de logiciel au sein d'un design. C'est le lieu de modification d'un modèle lors de la génération de l'interface logiciel - matériel : selon le niveau d'abstraction visé, différents composants d'interfaces seront insérés dans le modèle. Il offre des services particulièrement importants, notamment le service d'exécution et les services d'API HDS et HAL.

A chaque niveau, le module hybride offre un ensemble de services normalisés (satisfaisant, par exemple, la norme POSIX [29]). Le Tableau 4.2 résume ces différents services. Soulignons que plus le niveau d'abstraction s'affine, plus le nombre de services augmente.

Les caractéristiques de communciation du service IO_ACCESS, disponible à tous les ni-

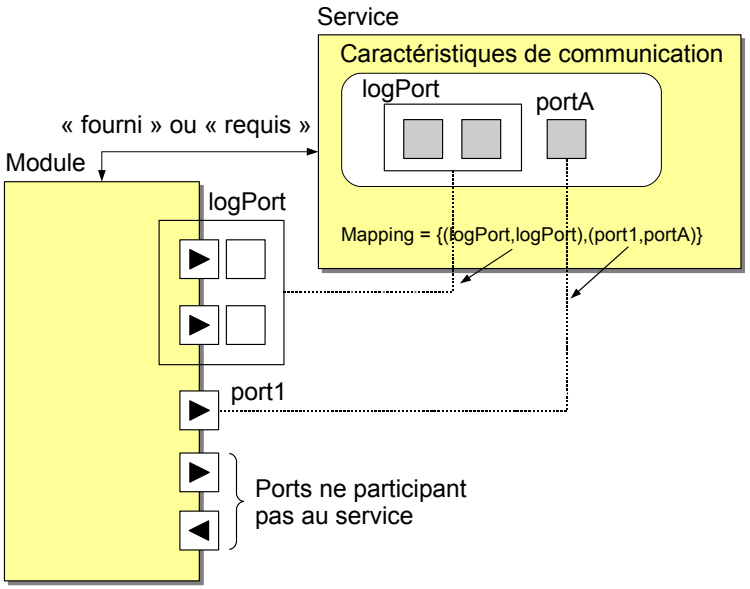

FIGURE 4.7 – Mapping entre les caractéristiques de communication d'un service et les ports d'un module

TABLE 4.2 – Services normalisés offerts par le composant hybride

Niveau	Service	Description
VA, TA, CABA	IO_ACCESS	Accès à la plateforme matérielle
TA, CABA	BOOT	Chargement et exécution d'une tâche
	CONTEXT	Gestion du contexte des threads (init, switch, load...)
	MP	Gestion multiprocesseur
	SPIN	Gestion de la synchronisation (lock, unlock, testlock...)
	IT	Gestion des interruptions (attach, detach, enable...)
CABA	*Pour ARM ISS de SocLib :*	
	RESET_PROC	Réinitialisation des variables
	STEP_PROC	Exécution de la prochaine instruction
	MANAGE_INSTR	Lecture d'instruction (fetch)
	MANAGE_DATA	Get and set données (load-store)

veaux, sont données ici en guise d'exemple :

```
unsigned read(unsigned address);
write(unsigned address, unsigned data);
```

Il faut noter que dès le niveau TA apparaissent des accès en mémoire à partir du logiciel. Il s'agit typiquement du dé-référencement d'un pointeur en C, de la création d'un objet en utilisant « new » en C++ ou de l'accès à une variable globale. Ces accès mémoire, simples en apparence, doivent être explicités au travers de l'interface logiciel-matériel. Les variables globales, qui sont définies à l'extérieur du module qui les utilisent, peuvent être explicitées sous la forme d'un service requis, qui devra être satisfait par le module conteneur. Puisque le développement d'un mécanisme de support des pointeurs nécessite un travail conséquent de recherche et de développement, celui-ci est proposé comme perspective du présent travail.

4.4 Esquisse d'un flot de conception orienté services

4.4.1 Principaux éléments du flot

L'objectif des services est de permettre la génération des interfaces logiciel-matériel à plusieurs niveaux d'abstraction. Cette génération peut se faire selon deux objectifs : obtenir un modèle exécutable du système, ou raffiner le système.

Nous faisons l'hypothèse que les tâches logicielles de l'application changent très peu d'un niveau d'abstraction à l'autre. Ce sont la plateforme matérielle et l'interface logiciel-matériel qui changent. Un flot de conception adapté à cette réalité est présenté ci-dessous :

– Niveau Système. En entrée de ce flot, au niveau système, se trouve la spécification de l'application. La principale activité à ce niveau est d'exécuter le code application et d'effectuer les premières validation et les premiers profilages de tâches.
– Niveau VA. Deux informations permettent d'atteindre ce niveau : (1) l'architecture de la plateforme, avec un module hybride par processeur, et (2) le mapping entre les tâches et la plateforme. Ces données doivent mener à l'obtention du premier modèle consolidé application-architecture. Une opération de résolution de services permet dans un premier temps d'associer les tâches aux modules hybrides. L'élaboration et la génération de l'interface produiront un premier prototype virtuel, où le logiciel est visible au niveau source.
– Niveau TA / CABA. La plateforme est décrite au niveau TA ou CABA. L'opération de résolution de services est réalisée à l'aide d'une bibliothèque TA ou CABA, selon le niveau du prototype virtuel que l'on souhaite obtenir. À ces niveaux, tous les composants logiciels sont compilés sous une forme binaire et exécutés nativement ou à l'aide d'un simulateur de jeu d'instruction.

Il faut souligner que le mapping peut être spécifié de trois façon : à la main, via un script d'exploration d'architecture automatique ou via un fichier de mapping.

4.4.2 Rôle de la bibliothèque de services

Les bibliothèques de services jouent un rôle fondamental dans le développement des outils. Elles permettent de réutiliser des composants préconçus afin de générer des prototypes virtuels à différents niveaux d'abstraction. On peut faire une analogie avec les bibliothèques logicielles spécialisées ou les bibliothèques de cellules de base pour la synthèse matérielle.

L'organisation des services en bibliothèques est donc un élément clé du modèle proposé. Une bibliothèque doit permettre de trouver rapidement le service qui correspond aux attentes d'un module client, selon l'identificateur de service et selon les trois niveaux de caractéristiques (communication, plateforme, performance). Une fois le service identifié, l'implémentation correspondante doit être configurée et intégrée dans le design. Cette implémentation peut consister en un seul module, ou une chaîne de modules si un seul module ne suffit pas. Dans ce cas, la bibliothèque doit spécifier quels modules font partie de la chaîne.

Représentation des trois niveaux de caractéristiques

Différents formats de bibliothèque de services ont été proposés dans les travaux cités précédemment. Gauthier [19] propose une bibliothèque qui contient des services et des éléments qui les fournissent ou requièrent. Chaque élément possède plusieurs implémentations et le choix se fait en fonction des contraintes de plate-forme. La notion de port est absente de la bibliothèque, la sélection d'un service puis de son implémentation s'effectuant sur la base d'identificateurs. Les services et les éléments fournisseurs et requérants sont stockés sous forme d'un graphe de dépendances, où les services sont groupés hiérarchiquement en familles fonctionnelles.

Pour Grasset [23], le graphe de dépendance repose sur la notion de protocole et de SAP. Les implémentations sont choisies en fonction des protocoles qu'ils fournissent, l'objectif étant de trouver une chaîne de modules capable de satisfaire les besoins de communication à ses extrémités.

Le modèle proposé dans la présente recherche propose une consolidation de ces deux approches, à savoir une sélection de service à la fois sur les critères de communication, de plate-forme et de performance. L'objectif est de pouvoir traiter tous types de services, autant logiciel, matériel qu'hybride, afin de prendre en compte l'ensemble de l'interface logiciel-matériel.

La bibliothèque proposée repose sur deux éléments : d'une part, les collections de caractéristiques de services organisées sous forme d'arbres. D'autre part, la liste des spécifications de

services fournis, associant chaque fois un groupe unique de trois caractéristiques de service à un module.

Exemple de bibliothèque de services

Puisque le développement de bibliothèques constitue un thème de recherche à part entière, nous allons nous limiter ici à quelques bibliothèques simples, suffisantes pour fournir un contexte d'expérimentation.

Une bibliothèque de « services essentiels » est définie pour chaque niveau d'abstraction. La bibliothèque du niveau TA est illustrée à la Figure 4.8. Cette bibliothèque minimale contient un composant de communication logiciel (DPN), un système d'exploitation temps réel (RTOS), ainsi que le composant hybride qui abstrait la couche HAL, décliné en version simple et version symmetric multiprocessing (SMP). Ces composants contiennent les informations nécessaires à la génération d'un prototype virtuel de niveau TA.

On note la présence du service *channel_io*, qui représente un canal de communication abstrait. Ce canal permet différents types de communication au niveau TA, que ce soit entre modules logiciels ou matériels (à ce niveau, la différence est conceptuelle). La communication repose sur l'accès à un espace mémoire partagé via les méthodes *channelRead* et *channelWrite*. On remarque aussi le composant hybride CPU_SS qui offre le service de communication *io_access* avec le matériel abstrait, au sein des services de l'API HAL. Le service d'accès est paramétré par la plage d'adressage (memory map), spécifiée manuellement par le concepteur dans le service requis. D'autre part, ce composant requiert un service d'interconnexion aux périphériques (*ta_bus*), contraint par une table d'interconnexion spécifiée par le concepteur dans les *caractéristiques de plateforme* du service. On remarque enfin le service exécution (*exec*), qui permet de spécifier les paramètres d'exécution des tâches par le système d'exploitation.

4.4.3 Sélection - intégration des services

Le but de la résolution de service est d'associer services requis et services fournis avant la génération du prototype virtuel. Ceci revient à insérer les modules qui offrent les services au sein du modèle à partir d'une bibliothèque. C'est à cette étape que l'interface logiciel-matériel est figée, sous forme de modules logiciels, matériels et hybrides.

Afin d'organiser l'information de design et la gestion des services, trois types de modules TOP sont définis :

1. **Le top application**. Ce module représente l'aspect logiciel de l'application. Il est de type SW/GROUP et contient les piles logicielles de type SW/ABSTRACT. Les piles logicielles, fournies par le concepteur, contiennent exclusivement des modules SW. Les interconnexions entre ces piles logicielles doivent être spécifiées à l'aide de ser-

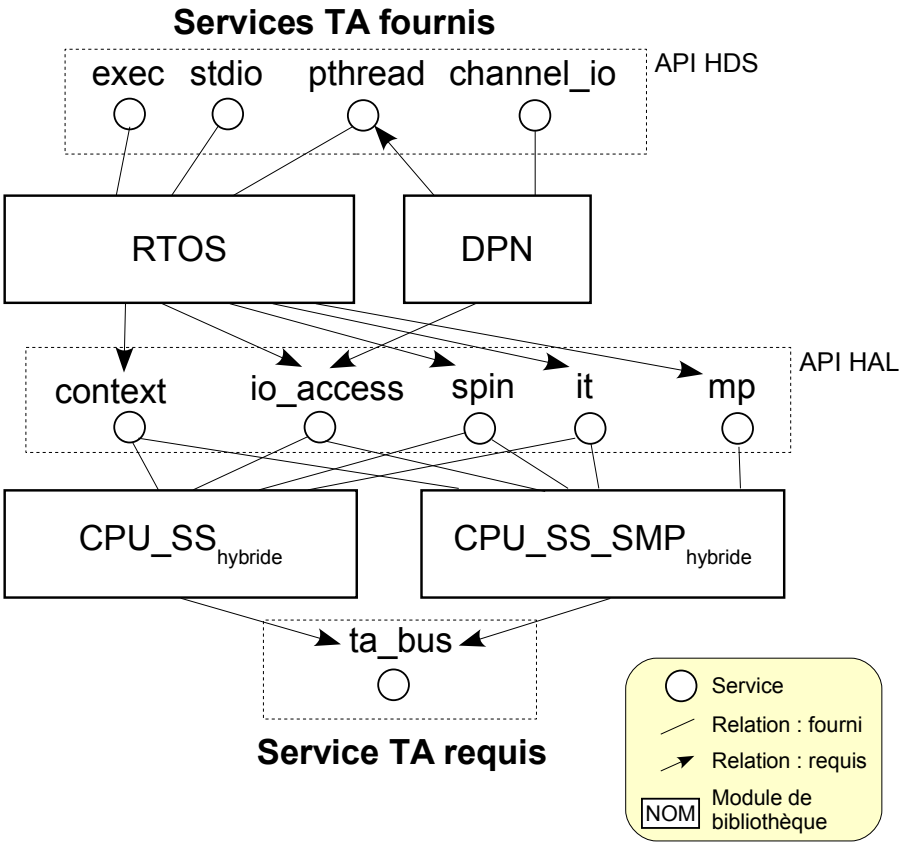

FIGURE 4.8 – Bibliothèque d'éléments du niveau TA

vices. Elles devront passer par la plateforme, seul dénominateur commun entre les piles.

2. **Le top plateforme**. Ce module représente la plateforme cible, qu'elle soit abstraite (niveau VA) ou matérielle (niveau TA ou CABA). Le module plateforme contient des modules HW ainsi qu'un module HYBRID pour chaque processeur de la plateforme. Ce module est également fourni par le concepteur.
3. **Le top consolidé**. Ce module représente le système application-architecture combiné, où tous les services requis ont été associés à un service fourni. Ce module consolidé est généré de façon semi-automatique. Le concepteur (ou un outil qui le fait à sa place) guide la consolidation en indiquant le mapping entre les modules SW/PROCESS et les modules HYBRID (qui représentent les processeurs).

La génération du module consolidé s'effectue en deux étapes : la résolution des services et l'intégration des modules correspondants. Nous appellerons le module qui requiert un service Mr, et le module qui fournit le service, Mf. Nous avons, comme entrée de la méthode, un module Mr qui invoque un service requis, Sr. Par cette spécification, le module déclare que son exécution impliquera un échange d'informations avec un fournisseur de service.

Première étape : résolution (ou mapping) des services

Il s'agit de trouver le service fourni, Sf, qui se rapproche le plus de Sr [2]. On cherchera toujours à résoudre un service à l'aide d'un composant du design en premier, que le service soit requis par un composant de l'application ou de la plateforme. Si aucun composant du design ne peut satisfaire la requête directement, le flot s'orientera vers les bibliothèques de composants.

La sélection peut s'effectuer de deux façons : (1) Sélection directe via l'identificateur unique. Celui-ci est décomposable, rappelons-le, en trois partie : ID de l'interface, ID des caractéristiques plateforme, ID des caractéristiques de performance. (2) Sélection progressive via une analyse *par valeur* des trois niveaux de caractéristiques des services.

Cette deuxième façon de résoudre les services requiert une notation spécialisée pour la capture et l'analyse des données des trois niveaux de caractéristiques, comme celle proposée par le profil UML MARTE. Des algorithmes plus flexibles de recherche et de négociation de services pourraient alors être développés. Une description formelle des protocoles permettrait également l'implantation de techniques de génération d'interface plus souples, telles que présentées dans [16], [15] et [60]. Dans la présente recherche, seule la sélection par identificateur est implantée, en guise de preuve de concept.

Nous faisons l'hypothèse que deux services compatibles ont nécessairement le même identificateur et vice-versa. S'il n'y a pas de service répondant aux trois niveaux de caractéristiques, il faut envisager d'ignorer les requis des 3e, 2e et 1er niveau dans l'ordre. Si le

2. Dans le contexte SOA, cette étape s'appelle aussi« Service Discovery »

compromis n'est pas possible, c'est qu'il faut effectuer une composition de service ou créer un élément de bibliothèque.

Une table d'association des services contient les identificateurs des services requis et fournis compatibles.

La résolution du service exécution a la particularité d'être réalisée de façon légèrement différentes, selon le niveau d'abstraction :

– Au niveau VA, le mapping tâche-processeur est réalisée implicitement par l'architecture. Il n'y a pas de service exécution tel quel, puisque les modules application sont visibles et connectés au même titre que les modules matériels et hybrides. L'association tâche - processeur est donc statique et sert a priori à valider le parallélisme de l'application et le partitionnement logiciel-matériel.
– Au niveau TA et CABA, les tâches logicielles sont sous forme de fichiers binaires, prêts à être chargés et exécutés, nativement ou via un ISS (tel que présenté à la Section 2.3.1). Le mapping est spécifié en reliant une tâche (du module Top Application) à un module hybride (du module Top Plateforme) via le service exécution. Ainsi, le processeur est informé des instances de tâche qu'il doit exécuter.

2e étape : configuration et intégration

Cette deuxième étape consiste à consolider le modèle en vue de la génération du prototype virtuel à un niveau d'abstraction donné. Ceci s'effectue par l'insertion des modules fournisseurs de services provenant de la bibliothèque au sein du design, en se basant sur les informations de la table d'association des services.

Le service exécution est intégré de différentes façons, selon le niveau d'abstraction visé :

– *Niveau VA* : Insertion d'un module HDS_API par module hybride. Configuration du module hybride pour relier l'API HDS aux modules matériels via un premier mapping en mémoire.
– *Niveau TA* : Résolution du service exécution pour chaque tâche logicielle, ce qui se traduit par l'ajout des modules OS, IO, et HAL_API. Configuration de l'algortihme d'ordonnancement des tâches pour chaque module hybride. Configuration du module hybride pour pour relier l'API HAL directement aux modules matériels. Spécification des mappings mémoires de chaque composant matériel.
– *Niveau CABA* : Même chose que pour le niveau TA, avec l'ajout du module logiciel HAL en plus. Configuration du module hybride pour relier les signaux BFM du processeur à la mémoire où est entreposé le code binaire.

Il faut noter qu'une étape intermédiaire de négociation / composition de service peut être insérée entre les deux étapes présentées. La négociation consiste à jouer sur les paramètres d'un module afin de satisfaire la requête. La composition consiste à élaborer un nouveau

service à partir de sous-services existants, en cas d'échec de la négociation. Si malgré tout aucun service ne peut satisfaire la requête, trois solutions sont possibles : soit modifier le design quand c'est possible, soit rendre l'élément de bibliothèque plus paramétrable, soit ajouter un nouvel élément à la bibliothèque.

Ces deux étapes, implémentées au sein d'un outil externe au design, mènent à l'obtention d'un module consolidé, où tous les composants de l'interface logiciel-matériel du niveau d'abstraction visé sont présents. La dernière étape, la génération du prototype lui-même, est dépendante du langage cible. Cette étape sera décrite plus en détail dans le chapitre suivant, qui traite de l'implémentation du modèle des services.

4.5 Conclusion

Dans ce chapitre, nous avons détaillé le premier élément de contribution de cette thèse, à savoir un modèle de services qui permet l'abstraction d'interfaces logiciel-matériel dans les systèmes sur puce. Dans la première partie, nous avons vu les concepts directeurs de notre modèle de service. Ensuite, nous avons présenté les détails du modèle des services, basé sur une description à trois niveaux de caractéristiques.

Le rôle des bibliothèques dans la génération a été présenté. Leur développement constitue l'élément central de la productivité des outils. Pour le moment, des bibliothèques primitives ont été proposées. Elles ne permettent pas, par exemple, d'expliciter les accès mémoires impliqués par l'utilisation de pointeurs.

Le modèle présenté est élémentaire. Son évolution devra permettre l'ajout de nouveaux paramètres aux trois niveaux de contrainte, l'optimisation de la sélection, l'introduction de paramètres de services et paramètres de modules, etc.

Le rôle du chapitre suivant sera de donner corps à ce modèle, par le biais d'une structure de données adaptée.

Chapitre 5

Définition d'une représentation intermédiaire basée sur l'approche service

Sommaire

5.1 Introduction 60
5.2 Survol de la représentation intermédiaire 60
5.3 Objets de la structure de données 61
5.3.1 Objets structurels 61
5.3.2 Objets reliés aux Services 66
5.3.3 Objets de gestion 68
5.4 Sémantique du modèle Serif 69
5.4.1 Sémantique de composition 70
5.4.2 Sémantique d'exécution : génération de modèles SystemC 71
5.5 Format d'échange et de stockage persistant : SPIRIT+ 73
5.6 API Serif 76
5.7 Contexte méthodologique de Serif 77
5.7.1 Flot de design proposé 77
5.7.2 Environnement logiciel proposé 77
5.8 Conclusion 81

5.1 Introduction

Ce chapitre décrit le deuxième élément de contribution de cette thèse, à savoir la représentation intermédiaire développée sur la base du modèle des services. L'objectif de cette représentation est de fournir la structure de données et l'API nécessaires à l'exploitation du modèle des services par différents outils logiciels.

La structure de données développée ainsi que ses règles de composition et d'exécution sont d'abord présentées. Le format d'échange, qui doit permettre l'échange des données entre outils ainsi que leur stockage persistant, est ensuite décrit. Les fonctions de l'interface d'accès à la structure sont ensuite présentées. Le chapitre se termine avec la description d'un flot de conception reposant sur la représentation intermédiaire.

5.2 Survol de la représentation intermédiaire

La représentation intermédiaire repose sur deux principaux éléments :

– La **structure de données** doit permettre de capturer, d'une part, les interfaces logiciel-matériel au cœur d'un design et, d'autre part, les informations de services associées à celles-ci. Les objets qui représentent les interfaces sont les modules, les ports et les nets, désignés comme les *objets structurels* à la Section 4.3.1. À certains de ces objets viendront s'ajouter des spécifications de services requis-fournis.
– Le modèle des services sous-tend une méthodologie où plusieurs outils interviennent, à différents stades du développement. Nous avons donc également besoin d'un **format d'échange** compréhensible par tous les outils, pouvant être sérialisé [1] et sauvegardé sur un support persistant (par exemple, un fichier sur disque). Ce format constitue le deuxième élément de la représentation intermédiaire.

Rappelons les principales informations associées aux flot de conception présenté à la Section 4.4.1 et qui doivent pouvoir être capturées par la représentation intermédiaire :

– La spécification de l'application au niveau système. À ce niveau, seules les principales tâches de l'application sont connues et sont abstraites sous formes de modules, communiquant via des canaux abstraits.
– L'architecture cible, à différents niveaux d'abstraction, constituée des éléments hybrides (un pour chaque processeur ou groupe de processeurs SMP) et matériels sur lesquels s'exécutent les tâches de l'application.
– L'information de mapping entre l'application et l'architecture. L'objectif est d'obtenir un modèle mixte application-architecture spécifique à chaque niveau d'abstraction.

1. Processus d'encodage d'une structure de données sous forme d'une suite d'éléments atomiques

Nous allons définir une représentation homogène capable de capturer ces informations et qui servira à la génération des interfaces logiciel-matériel. Afin d'illustrer l'approche orientée service à la base de cette représentation, nous la baptisons **Serif**, pour *Ser*vice-Based *I*ntermediate *F*ormat.

La structure de données sur laquelle repose la représentation est conçue selon une approche orientée objet, en vue d'une implantation en C++. Les facteurs déterminants dans le choix du C++ comme langage d'implantation sont la familiarité des concepteurs de systèmes intégrés avec les langages C/C++ de même que la compatibilité avec les bibliothèques et outils existants, souvent écrits en C/C++.

Il est nécessaire de bien définir certaines expressions qui sont utilisées dans ce chapitre :

1. Modèle des services : Réfère au modèle présenté au Chapitre 4.
2. Objet Serif : Un objet de la représentation intermédiaire, utilisé pour capturer un concept. Exemple : le Module.
3. Classes Serif : Une classe au sens orienté-objet. Exemple : la classe SerifModule implante l'objet Module.
4. Modèle Serif : Ensemble des objets ou des classes Serif et de leurs relations. Le modèle Serif constitue un métamodèle, c'est à dire un modèle commun utilisé pour la construction de modèles uniques.
5. Description ou Design Serif : Assemblage unique d'objet (de classes) Serif décrivant une application. Il s'agit ici d'un modèle du système en cours de conception, par opposition au métamodèle. Exemple : Description Serif d'une application MJPEG.

5.3 Objets de la structure de données

Les objets de la structure de données peuvent être regroupés en trois catégories : (1) les objets structurels, (2) les objets liés aux services et (3) les objets de gestion. Ces trois catégories permettent la représentation de toutes les informations liées aux interfaces logiciel-matériel.

5.3.1 Objets structurels

Les objets structurels présentés au chapitre précédent, le module, le port et le net, sont représentés tels quels dans la structure de donnée.

Le module Serif

Comme nous l'avons vu, les sytèmes sur puce contiennent plusieurs types de modules : logiciel, matériel, hybride, ou de test. Nous avons été confronté au choix suivant dans l'im-

plantation du modèle : créer une classe spécialisée pour chaque type d'objet ou utiliser une classe générale précisée à l'aide des attributs « cible » et « fonction ». La décision fut prise d'utiliser les attibuts et de ne créer une classe spécifique que lorsque la sémantique de l'objet justifiait la définition de méthodes spécifiques.

Le module Serif possède la structure présentée au Tableau 5.1.

TABLE 5.1 – Principaux membres de l'objet Module

Module		
Membre	Cardinalité	Description
Identificateur	1	Identificateur unique qui permet de créer une référence à ce module lors de son instanciation.
ModuleInterface	1	L'interface unique du module, propriétaire des ports et qui fournit ou requiert les services.
ModuleContent	1..*	Les contenus possibles d'un module, invisibles de l'extérieur. L'un deux est défini comme contenu par défaut.
TargetType	1	Définit la cible du module : {HW | SW | HYBRID | TARGET_TO_SET}
FunctionType	1	Définit le type de fonction. Valeurs possibles : {PROCESSOR | CHANNEL | MEMORY | PROCESS | MANAGEMENT | BUS | ABSTRACT | FUNCTION_TO_SET}

L'interface d'un module est un élément clé du design. Sa structure est détaillée dans le tableau 5.2.

TABLE 5.2 – Principaux membres de l'objet ModuleInterface

ModuleInterface		
Membre	Cardinalité	Description
Ports	0..*	Points de connexion unique avec le module.
LogicalPorts	0..*	Ports logiques ou groupes de ports qui réalisent un protocole.
Services	0..*	Liste des services requis et fournis

Le contenu d'un module, caché de l'extérieur, permet la composition hiérarchique, comme le démontre sa structure détaillée au tableau 5.3.

L'objet ModuleInstance contient simplement une référence à son Module définition ainsi qu'une copie locale des ports et ports logiques (dits « ports sur instance »). Ces copies locales sont nécessaires parce que les interconnexions sont spécifiées à l'aide de références en mémoire et non à l'aide d'identificateur de port (du type "nom_instance.nom_port"). Ceci a comme avantage d'accélérer la performance lors du parcours et de l'analyse d'un design, mais a comme inconvénient que chaque port d'instance doit être un objet unique en mémoire.

TABLE 5.3 – Principaux membres de l'objet ModuleContent

ModuleContent		
Membre	Cardinalité	Description
ModuleInstances	0..*	Liste des sous-modules, qui sont des instances de modules définis antérieurement.
Nets	0..*	Liste des nets qui associent ports et ports logiques entre eux.
ExtBehavior	0..*	Liste des fichiers source qui implémentent le comportement.

La structure ExtBehavior contient une liste des fichiers source du comportement (*sources*), une liste des ports susceptibles de déclencher un appel à ce comportement ainsi (*triggerPortNames*) que le code source d'appel du comportement (*extBehavCallCode*). Ces informations seront utilisées lors de la génération du prototype virtuel.

Il doit être possible de définir un processeur et puis de l'instancier plusieurs fois, tout en attribuant à chacune de ses instances un mapping de tâches spécifique. Pour ce faire, il est nécessaire que chaque instance du processeur possède son propre service exécution, qui sera configuré en fonction du mapping désiré. L'aspect service de la représentation intermédiaire sera présenté à la prochaine section.

Le port logique Serif

Le port logique est le deuxième objet Serif qui bénéficie du mécanisme de définition-instantiation. Ce mécanisme est calqué sur celui de SPIRIT IP-XACT [43], qui permet ainsi la définition d'interface de bus complètes. Le port logique se décline en trois objets distincts qui correspondent aux phases de définition, d'instanciation sur un module, puis d'instanciation sur un sous-module.

Les principaux membres de chacun de ces objets sont présentés dans le Tableau 5.4

Rappelons que vu de l'extérieur d'un module, les sous-ports d'un port logique ne sont pas visibles. L'objet LogicalPortOnInstance ne possède donc aucune information sur les sous-ports.

Objets utilitaires

Quelques objets dits "utilitaires" sont associés aux objets structurels pour en simplifier l'utilisation. C'est le cas de l'*Identificateur unique* attibué à chaque Module et à chaque définition de port logique. Cet identificateur est construit sur le modèle du VLNV (Vendor, Library, Name, Version) utilisé dans IP-XACT, constitué de quatre chaînes de caractères. Le VLNV permet la sérialisation des références entre objets ainsi que la recherche d'un module ou d'un

TABLE 5.4 – Principaux membres des objets reliés au port logique

LogicalPortDefinition		
Membre	Cardinalité	Description
Identificateur	1	Identificateur unique qui permet de créer une référence à ce port logique lors de son instanciation.
Ports	1..*	Liste des ports simples qui constituent le port logique
LogicalPort		
Definition	1	Référence à la définition de ce port logique
Interface	1	Référence vers l'interface de module à laquelle ce port logique appartient
PortMapping	1	Mise en correspondance entre les ports du module et les ports de la définition du port logique.
LogicalPortOnInstance		
Definition	1	Référence vers la définition de ce port logique
Instance	1	Référence vers l'instance de module propriétaire de ce port logique
Nets	0..*	Liste des nets (externes) connectés à ce port logique.

port logique dans un design.

L'objet *Paramètre* permet l'ajout d'une liste de paires *nom-valeur* à chacun des objets Serif. Ces paramètres pourront servir à spécifier des données personnalisées, pour lesquelles aucun objet Serif n'a été créé, et utiles aux outils de manipulation du design.

Le Tableau 5.5 présente une synthèse des objets structurels Serif et les principales classes C++ qui les implémentent.

TABLE 5.5 – Objets structurels Serif et classes correspondantes

Objet du modèle	Classes
Module	SerifModule SerifModuleContent SerifModuleInterface SerifModuleInstance
Port et port logique	SerifPort SerifPortOnInstance SerifLogicalPortDefinition SerifLogicalPort SerifPortOnInstance SerifLogicalPortOnInstance
Net	SerifNet

Les relations entre les classes Serif ainsi que leurs principaux attributs sont illustrés dans le diagramme de classe de la Figure 5.1.

Un exemple d'utilisation des objets structurels est illustré à la Figure 5.2. Cette figure reprend le Module A présenté à la Figure 4.2, mais cette fois avec les classes Serif. Elle met en évidence les deux phases de conception d'un module, à savoir sa définition (à gauche) puis

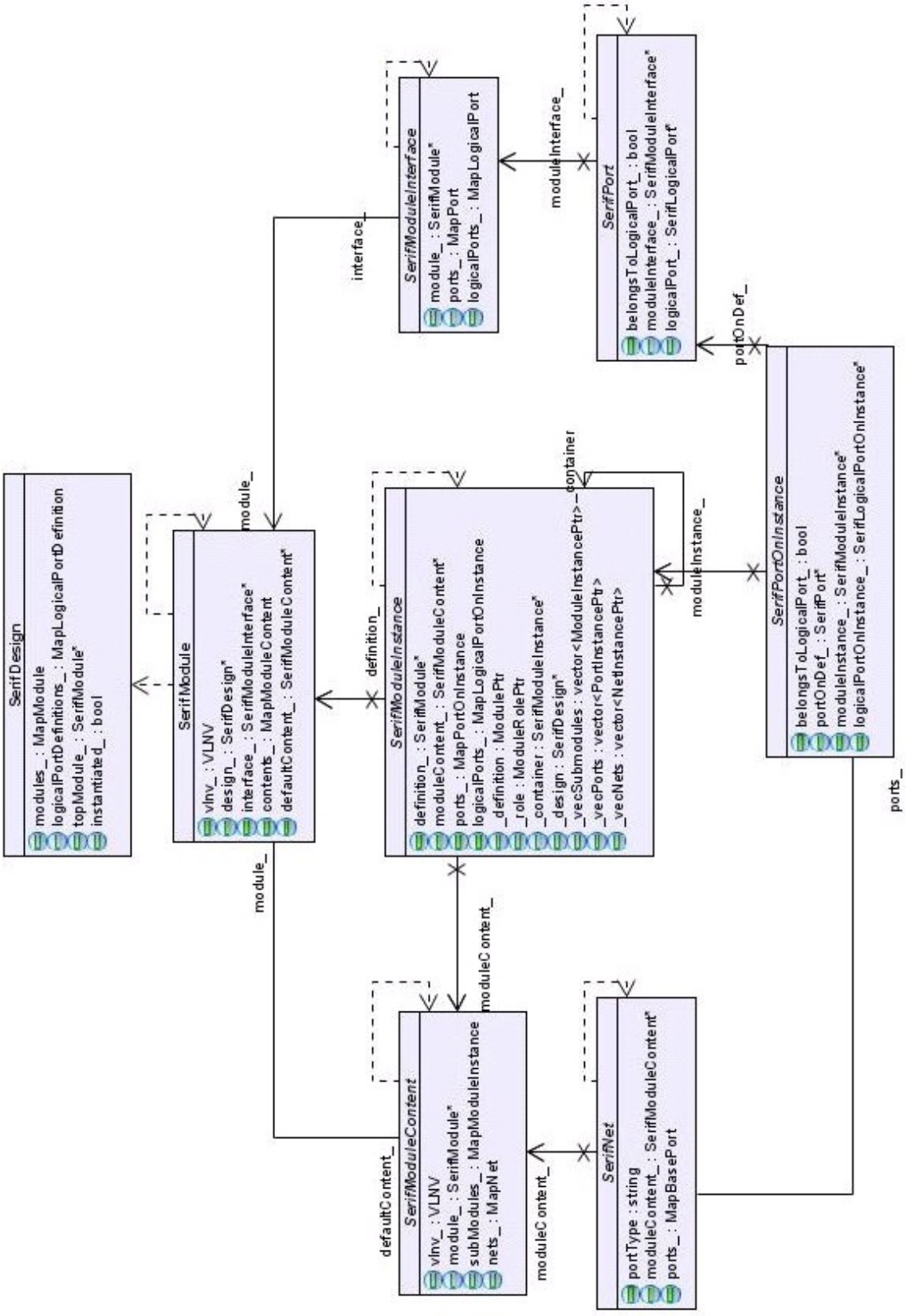

FIGURE 5.1 – Diagramme des classes Serif en notation UML

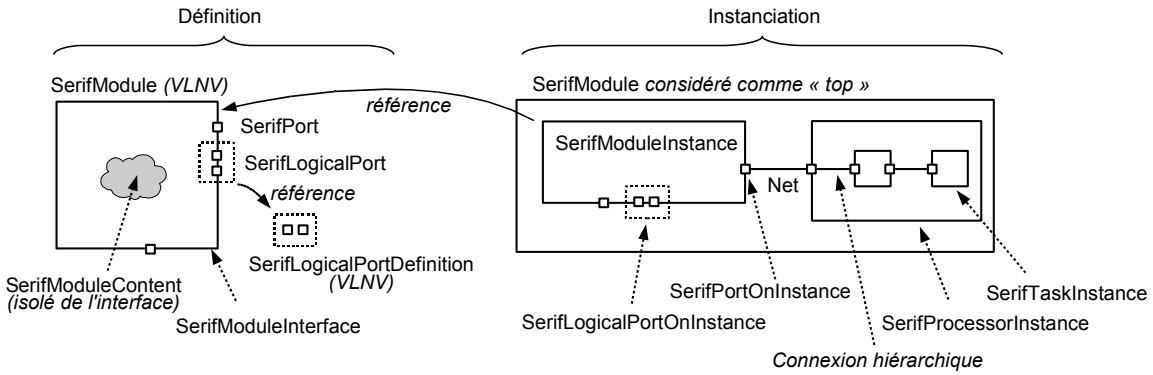

FIGURE 5.2 – Définition et instanciation à l'aide des objets structurels

son instanciation dans un module "top" (à droite). L'instance d'un module contient essentiellement une référence vers la définition ainsi qu'une copie des ports de la définition (ports sur instance, modélisés avec les classes SerifPortOnInstance et SerifLogicalPortOnInstance). Ces ports sur instance contiennent l'information de *connexion externe* d'un module. Pour connaître les *connexions internes* d'un module, il faudra se rapporter à sa définition.

5.3.2 Objets reliés aux Services

Les objets reliés aux services sont au cœur de la contribution de cette thèse. Ils ont comme objectif d'implanter le modèle présenté au Chapitre 4, en association avec les objets structurels Module, Port et Net :

– Les *Caractéristiques de Service* décrivent les trois niveaux de caractéristiques d'un service fourni ou requis : communication, plateforme, performance. Les caractéristiques de communication sont décrites à l'aide de ports et de ports logiques, tandis que celle de plate-forme et de performance sont décrites à l'aide de paramètres pré-définis ou définis par l'utilisateur.
– L'objet *Service* lui-même est un objet qui associe les caractéristiques de service des trois niveaux avec le module qui définissent le service. Le module est identifié à l'aide d'une référence en mémoire ou de son VLNV. Le Service contient également un identificateur unique permettant d'y faire référence.

L'utilisation des objets reliés aux services diffère selon qu'il s'agisse de spécifier un service requis ou fourni. En accord avec les spécification présentés dans la section 4.3.4, les caractéristiques d'un service fourni sont entièrement connues et stockées dans une structure en arbre, afin de faciliter la réutilisation. La spécification de service sert, dans ce cas, à associer un ensemble de caractéristiques à un Module fournisseur de service.

Les caractéristiques d'un service requis, quant à elles, sont simplement spécifiées au sein même du service. Cet objet sert donc de pivot entre client et fournisseur de service. Si les services fournis sont stockés sous forme d'arbre de caractéristiques, la spécification d'un

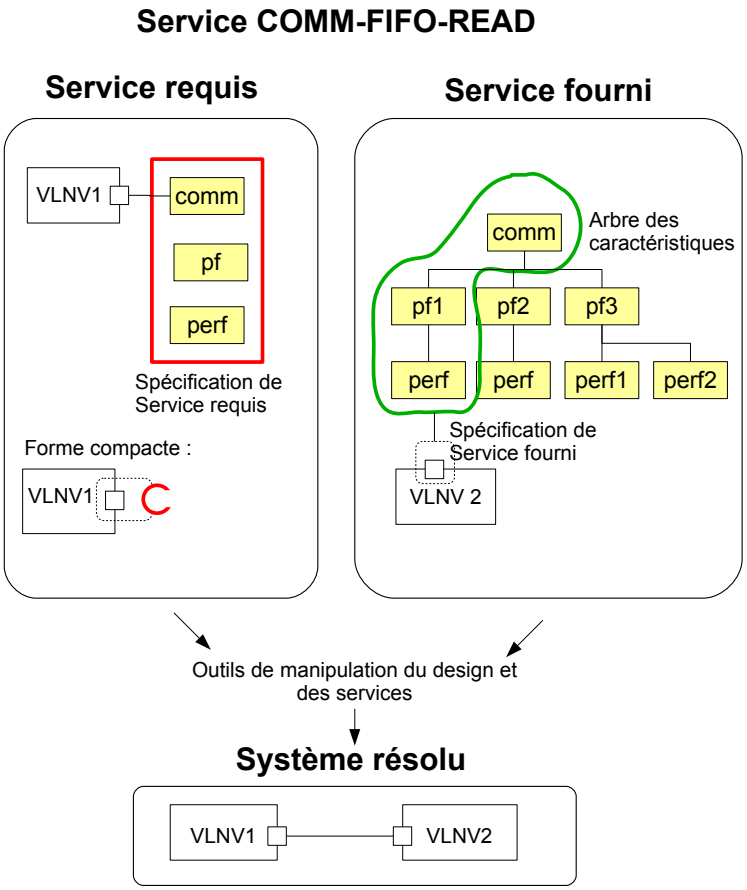

FIGURE 5.3 – Rôle des objets Serif dans la spécification d'un service requis et fourni

Service requis peut être vue comme une branche de cet arbre.

La Figure 5.3 illustre ces concepts par un exemple de service de lecture dans un canal FIFO. À gauche de la figure se trouve la spécification de service requis, avec ses trois caractéristiques. Le port définit implicitement les caractéristiques de communication du service, tandis que les caractéristiques de plate-forme (pf) et de performance (perf) sont spécifiées explicitement. Soulignons la "forme compacte" qui permet une représentation pratique du service requis à même la structure d'un design. La spécification de service fourni apparaît à droite de la figure, englobant les trois niveaux de caractéristiques de service. Rappelons que l'objectif est de résoudre la requête de service et d'insérer le bon élément dans le système, tel que l'illustre le "Système résolu" au bas de la figure.

Chacun de ces objets Serif est implanté à l'aide de différentes classes, tel que présenté dans le Tableau 5.6.

Les caractéristiques fonctionnelles : SerifCommFeatures

Ces caractéristiques décrivent l'interface de communication d'un service. Elles contiennent la liste des ports de communication utilisés par le service. Les ports, qui appartiennent au

TABLE 5.6 – Objets reliés aux services et classes correspondantes

Objet Serif	Classes
Caractéristiques de Service	SerifCommFeatures SerifPlatformFeatures SerifPerfFeatures SerifAddressSpace SerifRegister
Spécification de Service	SerifServiceSpecification

Module, sont toujours reliés par pointeur aux caractéristiques, sauf dans le cas d'un service fourni par une bibliothèque. Dans ce cas, du fait de leur rôle de racine d'arbre de caractéristiques, les caractéristiques de communication sont isolées du module et contiennent elles-mêmes les ports du service.

Les caractéristiques de communication décrivent également la machine à états du protocole associé aux ports. Cette machine à états peut ensuite participer au processus de résolution de service. La structure de données ainsi que le format persistant de cette machine à état n'ont pas été définis dans le cadre de ces travaux.

Les caractéristiques plateforme et non-fonctionnelles : SerifPlatformFeatures et SerifPerfFeatures

Ces caractéristiques décrivent les contraintes plateforme et les aspects non-fonctionnels d'un service. Les caractéristiques de plateforme définissent les plages d'adressage à l'aide des objets SerifAddressMap (maître) et SerifMemoryMap (esclave). Elles spécifient également les dépendances envers des bibliothèques externes, à l'aide de chaînes de caractères.

Quant aux caractéristiques non fonctionnelles SerifPerfFeatures, elles sont simplement spécifiées par une liste de paramètres nom-valeur, délégant aux outils la responsabilité de les interpréter correctement. Il serait intéressant de spécifier ces caractéristiques de service de façon plus formelle à l'aide du profil UML MARTE. Ce profil définit les éléments nécessaires pour capturer les différents attributs temps réel d'un service, que ce soit en terme de fréquence, de temps, de puissance, etc. L'utilisation de ce profil dans Serif permettrait le développement d'outils capable de raisonner sur les caractéristiques non fonctionnelles, ce qui favoriserait la réutilisation des services. La structure de données ainsi que les API permettant la manipulation des données du profil MARTE sont proposées en tant que perspectives de cette recherche.

5.3.3 Objets de gestion

De nombreuses relations existent entre les nombreux objets du modèle Serif, ce qui nécessite une gestion attentive de la mémoire. Par exemple, la liste des ports auxquels est connecté un net est stockée, au sein du net lui-même, sous forme d'une liste de références (c.f. classe

SerifNet de la Figure 5.1). Plusieurs relations sont ainsi réalisées par référence vers l'objet associé, plutôt que d'utiliser un identificateur textuel, ce qui permet un parcours rapide de la structure Serif. On doit s'assurer, dès lors, de la validité de chaque objet référencé en mémoire.

Afin de faciliter la gestion de la mémoire et organiser la création et la manipulation d'une description Serif, deux objets sont nécessaires :

- Le *Gestionnaire de Design* est le propriétaire et gestionnaire d'un design Serif. Cet objet unique facilite l'échange d'un modèle entre outils puisqu'il contient tous les objets d'un design. Il contient :
  - Tous les Module et Ports Logiques stockés à plat, prêts à être référencés lors de leur instanciation dans un design.
  - Trois références vers les modules « top », représentant respectivement le module application, le module plateforme et le module consolidé.
  - Les contraintes globales s'appliquant à un design, comme par exemple les chemins de données possibles définis dans une spécification DOL.
  - La table des associations services requis - fournis, en attente de la consolidation de l'interface logiciel-matériel. L'identification des services dans cette table est construite à partir d'identificateurs d'objets selon convention suivante :

    `{top_design|top_library[::submodule]::service}`

- La *Bibliothèque de Services* sert à organiser les trois niveaux de caractéristiques de service sous forme d'arbre. L'objet bibliothèque de service fournit des fonctions de création, d'analyse et de recherche de service.

Ces objets de gestion sont implantés à l'aide des classes présentées dans le Tableau 5.7.

TABLE 5.7 – Objets de gestion et classes correspondantes

Objet du modèle	Classes
Gestionnaire de Design	SerifDesign SerifPath SerifPathElement MapServiceMapping
Bibliothèque de Services	SerifServiceLibrary SerifCommFeatures

L'objet SerifCommFeatures joue ici un rôle supplémentaire : il sert de racine pour un arbre de caractéristiques de service.

5.4 Sémantique du modèle Serif

La section précédente a présenté les objets Serif et quelques exemples simples de composition. L'objectif de cette section est de décrire les aspects sémantiques des objets Serif. La sémantique de composition, utilisée lors de la description de systèmes hiérarchiques, est présentée en premier, suivie par la sémantique d'exécution.

5.4.1 Sémantique de composition

Les relations entre les objets Serif ont une signification bien précise, qui dépend de la nature des objets mis en relation. L'élément central est bien sûr le module, dont la sémantique définit celle de ses ports, de ses sous-modules et de ses nets internes. Nous nous intéresserons à la sémantique de composition de modules de type logiciel, hybride et matériel, qui représentent l'interface logiciel-matériel.

Les relations entre les objets Serif sont de trois types : la relation hiérarchique, la relation via service requis-fourni et la relation par interconnexion de ports.

La hiérarchie n'est possible qu'entre modules et elle est, a priori, infinie. Les modules matériels sont souvent décrits de façon hiérarchique, dans le but de maîtriser la complexité du design et de faciliter la réutilisation. Le développement logiciel utilise également la notion d'hiérarchie ("aggrégation", en UML), quoique les frontières entre niveaux soient beaucoup plus souples. Puisque avec Serif il doit être possible de représenter une tâche logicielle communiquant avec une tâche matérielle, nous devons définir précisément les liens hiérarchiques impliquant ces deux types d'objet.

En Serif, un Module qui représente une tâche logicielle (C/F = SW/PROCESS) est un élément feuille, c'est à dire qu'il ne peut contenir de sous-module. Le module de groupement logiciel (C/F = SW/ABSTRACT), qui sert à délimiter une pile logicielle, admet un niveau de hiérarchie inférieure. Le module logiciel, qu'il soit PROCESS ou ABSTRACT, doit être lié à un module hybride (cible = HYBRID) via une spécification de service de type « exécution ». Le module hybride, qui réalise le pont entre les tâches logicielles et les tâches matérielles, est à son tour associé à un Module processeur (C/F = HW/PROCESSOR) via des canaux de communication. Une tâche logicielle communique ainsi avec une tâche matérielle via le module hybride.

L'interconnexion explicite de ports et ports logiques constitue la troisième façon de réaliser la composition d'objets Serif. Cette interconnexion permet l'envoi de données via le net, selon un protocole du type put() / get() en matériel, ou un simple appel de fonction en logiciel. Si un comportement particulier doit être attribué au connecteur, par exemple un canal FIFO, il suffit d'insérer un module au milieu du net et de scinder celui-ci en deux.

Ces éléments sont illustrés dans la Figure 5.4. Dans ce diagramme, les tâches T1 et T2, toutes deux instances d'une définition T (par exemple deux instances d'un même programme), sont associées à un module hybride, via un module logiciel abstrait.

Ces relations se traduisent de différentes façons selon le niveau d'abstraction considéré, tel que présenté dans les sections qui suivent.

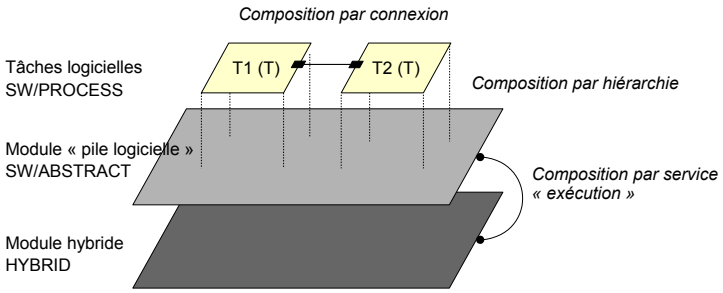

FIGURE 5.4 – Exemple de composition entre module logiciel et hybride

Sémantique du niveau VA

Au niveau VA, les modules logiciels (C/F = SW/PROCESS) offrent normalement une vision "code source", qui permet de jouer sur le parallélisme et le partitionnement et d'identifier les services requis par l'application. Ainsi, chaque module SW/PROCESS représente un point d'entrée dans un programme (en C/C++, la procédure « main »). Du fait de l'absence d'un OS explicite à ce niveau, l'association entre un module logiciel et un module hybride est statique.

Sémantique du niveau TA et CABA

Aux niveaux TA et CABA, le logiciel est compilé et on ne voit qu'un seul exécutable binaire par module SW/PROCESS. L'OS étant explicite, c'est à lui que revient la tâche de charger et d'exécuter les tâches. La relation entre un module logiciel et un module hybride peut alors devenir dynamique.

Ces différentes sémantiques se matérialiseront lors du processus de génération du modèle exécutable, décrit à la prochaine section.

5.4.2 Sémantique d'exécution : génération de modèles SystemC

Serif est une représentation qui n'est pas exécutable. En effet, le comportement des modules est isolé de leur interface et seule une référence vers le fichier où est décrit le comportement existe. Il est néanmoins essentiel d'associer une sémantique d'exécution aux principaux éléments du modèle afin de produire des versions exécutables d'un design. Ces versions exécutables, ou prototypes virtuels, serviront pour la simulation et la validation du design à différents niveaux d'abstraction, selon les principes énoncés dans [37]. Pour ce faire, il faut ré-intégrer interfaces et comportements en un modèle unique et exécutable.

La génération d'un modèle exécutable se fait à partir d'un modèle consolidé, où tous les services sont résolus et les interfaces détaillées (voir l'étape de résolution de services à la Section 4.4.3).

La sémantique d'exécution passe par la génération d'objets exécutables à partir d'objets Serif. Parmi les différentes options possibles, SystemC est celle qui satisfait le mieux les exigences actuelles. SystemC permet de construire des modèles exécutables de systèmes à différents niveaux d'abstraction et il est relativement facile à générer. De plus, plusieurs objets structurels de Serif trouvent une correspondance naturelle avec les objets SystemC.

L'idée principale est de générer un SC_MODULE pour chaque module Serif. Pour les modules ABSTRACT, la génération se poursuit au niveau hiérarchique inférieur. Tous les modules dont la fonction n'est pas ABSTRACT sont considérés comme des feuilles, la génération se faisant alors de différente façon selon les attributs target/function. Notons qu'il s'agit d'une utilisation spécifique de SystemC, surtout au niveau du composant hybride, qui offre à la fois une interface logicielle et matérielle. L'algorithme de génération, divisé en deux étapes, est décrit ci-dessous. L'utilisation du tiret (-) dénote un commentaire.

```
- (1) Génération de l'interface logiciel - matériel
Get table d'association des services (contient le mapping tâche-processeur)
Pour chaque association de service "exécution"
   Si niveau de génération = VA
      - A ce niveau, les tâches sont modélisées par des SC_MODULE
      - (1.1) Génération des modules tâches logicielles
      Get module requérant (le module logiciel de type SW/ABSTRACT)
      Get liste des sous modules de type SW/PROCESS
      Pour chaque sous-module
         [A] Procédure de génération d'un module logiciel feuille
      Interconnecter les sous-modules tâches
      - (1.2) Génération du module d'abstraction de l'API HDS
      Générer un SC_MODULE pour le hds_api
      - (1.3) Génération du module d'abstraction du sous-système CPU
      Get module fournisseur (le module HYBRID)
      [B] Procédure de génération d'un module hybride
      - (1.4) Interconnexion du module hybride aux modules tâches
      Connexion des canaux explicites via les ports SystemC
      Connexion des appels de fonction via ports (serv. requis) et exports (fournis)
   Si niveau de génération = TA ou CABA
      - A ces niveaux, les tâches sont sous groupées en fichiers exécutables
      Get nom des binaires application, OS, HAL (dans module SW/ABSTRACT)
      Get module fournisseur (le module HYBRID)
      [B] Procédure de génération d'un module hybride
      - Association du module hybride et des tâches
      Configurer le chargement des binaires dans le module hybride

- (2) Génération des IP matériels et interconnexions
Get top module du modèle consolidé
Get liste des sous modules
Pour chaque sous module HW/ABSTRACT trouvé (module hiérarchique)
   Générer l'en-tête SystemC du module
   Get liste des sous modules
   Déclarer une instance de chaque sous module
   Répéter l'étape (2) avec le sous-module
   Effectuer les interconnexions interne du module HW/ABSTRACT
Pour tous les autres sous module HW trouvé
   Générer l'en-tête .h SystemC du module (ports, nets, sous-modules)
   Configurer la liste de sensibilité du thread ext_behav
   Générer le fichier implantation .cpp (comportement)
   Générer la fonction ext_behav
   Insérer le code extBehavCallCode dans ext_behav

[A] Procédure de génération d'un module logiciel feuille
   - Cette génération a lieu au niveau VA seulement
   Générer le fichier entête .h du SC_MODULE correspondant (ports)
      Créer un sc_port pour chaque canal explicite
      Créer un sc_port pour chaque service requis
         Nommer le sc_port avec le nom du service
```

```
        Associer l'interface du service à celle du port (sc_interface)
   Exporter le comportement du fichier source associé dans le sc_thread principal
   Initialiser la liste de sensibilité du sc_thread principal
   Configurer les adresses dans les tables du composant hybride

[B] Procédure de génération d'un module hybride
   Générer le fichier entête .h du SC_MODULE hybride
   Générer les ports de communication explicites avec le SW ou le HW (sc_port)
   Si au niveau VA
      Générer les ports de services VA fournis au logiciel (sc_export)
   Sinon
      Déclarer les fonctions exportées en fct du niveau d'abstraction (extern "C"")
   Configurer la table de routage des messages
      Associer un sc_port matériel à sc_export - sc_port ou fonction - sc_port)
   Générer le fichier implantation .cpp
```

Pour le port Serif, un objet SC_PORT avec le type et la direction correspondante est créé. Le port logique Serif génère une structure contenant le groupe de ports. Enfin, pour l'objet net Serif, une connexion SC_CHANNEL est réalisée entre deux ports SystemC.

Le concepteur devra intervenir manuellement dans le code généré pour tout élément de design qui n'est pas pris en compte par l'algorithme. Par exemple, la génération d'un composant de communication multi-point peut impliquer une table de routage ou d'adressage, qui établit une correspondance entre ports d'entrée et ports de sortie, ou entre une adresse et un port. Cette table devra être insérée manuellement dans le code du composant.

La Figure 5.5 illustre un exemple d'exportation vers SystemC d'un design décrit en Serif, au niveau VA. On sait qu'à ce niveau l'association entre une tâche et un processeur est statique, et donc bien représentée par des modules SystemC. À partir du niveau TA, chaque objet logiciel devient un fichier binaire exécutable ou une librairie, dont le chargement et l'appel se font par le module hybride.

Le fichier Makefile vient enfin compléter la génération du modèle exécutable SystemC. Sa génération repose sur un canevas, lequel est complété avec les noms de fichiers source des modules ainsi que les librairies spécifiées dans les services de dépendances logicielles.

5.5 Format d'échange et de stockage persistant : SPIRIT+

La structure de données proposée dans les sections précédentes permet à des outils logiciels spécialisés de raisonner sur un design. Cette structure est conçue pour permettre un accès efficace aux données tout en faisant levier sur la conception orientée objet. Sa principale lacune est qu'elle n'existe que dans la mémoire d'un programme. Ceci fait que, d'une part, elle est perdue lorsque le programme se termine et, d'autre part, il est difficile de la transmettre à un autre programme dans le contexte d'une chaîne d'outils. Il est donc essentiel d'associer un format de stockage à Serif qui permette la sérialisation et la sauvegarde des données de design.

Le format tout indiquée pour cette sérialisation est IP-XACT. On y retrouve de fait les

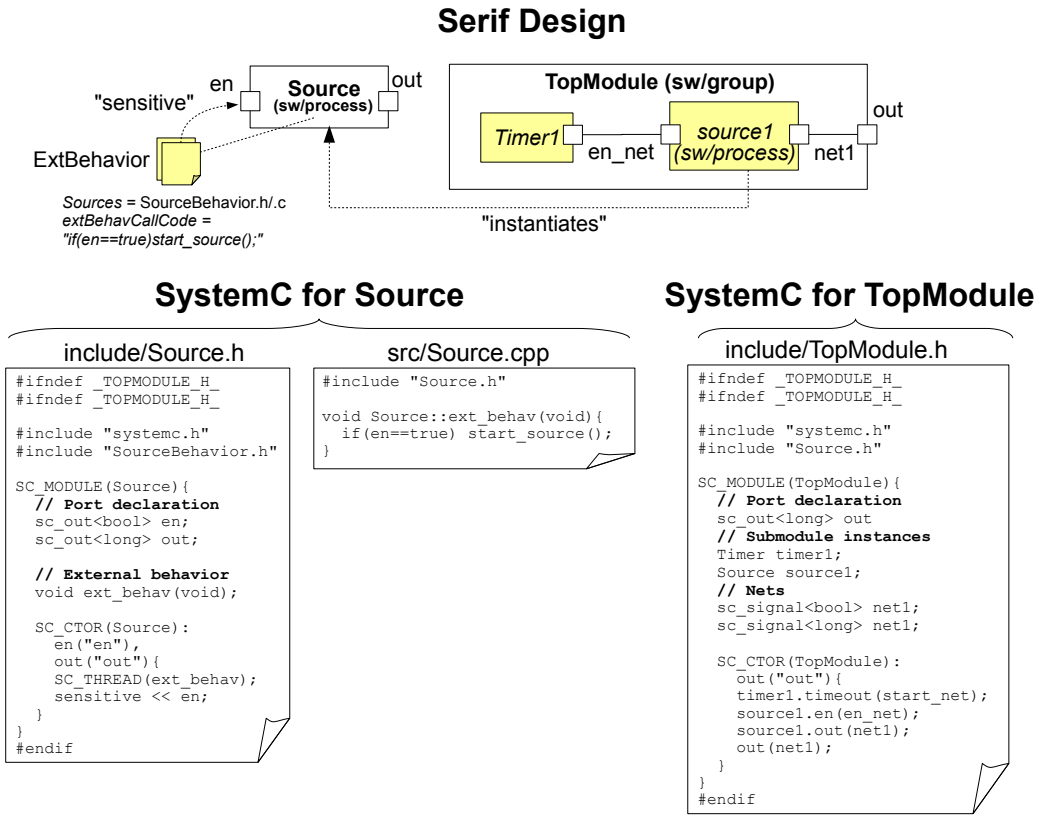

FIGURE 5.5 – Exemple de transformation Serif -> SystemC au niveau VA

concepts de module hiérarchique, de port, de port logique et d'interconnexion. L'utilisation de ce format pour stocker des designs Serif permet permet l'accès à tous l'écosystème d'outils et de librairies compatibles IP-XACT. Ce format est certes très orienté matériel, mais il offre la possibilité de personnaliser les éléments à l'aide de données utilisateurs, ce qui peut être exploité dans le contexte de ce travail.

Il y a quelques objets Serif que le format IP-XACT, dans son état actuel, ne peut pas représenter : le module logiciel et le module hybride (identifiés par leurs attributs cible/fonction), le Service et l'arbre des caractéristiques de services fournis.

Afin de pouvoir stocker les différents types de modules dans une représentation IP-XACT valide, nous allons utiliser deux paramètres IP-XACT, calqués sur les propriété cible/fonction. Tout ce qui a trait aux services sera stocké dans un schéma sur mesure, SerifService.xsd. Les informations de service pourront ainsi être intégrées, lorsque nécessaire, à l'élément *VendorExtensions* du composant associé. Le schéma SerifService est reproduit à l'Annexe A.

Le Tableau 5.8 présente les éléments IP-XACT qui seront utilisés pour stocker les objets Serif.

TABLE 5.8 – Correspondances entre les principaux éléments Serif et IP-XACT

Objets Serif	Élément IP-XACT	Schéma IP-XACT correspondant
SerifModule	componentType	component.xsd
targetType	parameter "TARGET_TYPE" ajouté à un component	commonStructures.xsd et component.xsd
functionType	parameter "FUNCTION_TYPE" ajouté à un component	commonStructures.xsd et component.xsd
SerifModuleInterface	modelType	model.xsd
SerifModuleContent	viewType et design	model.xsd et design.xsd
SerifModuleInstance	componentInstance	subInstances.xsd
SerifPort	port sur un model	port.xsd
SerifLogicalPortDefinition	busDefinition + abstractionDefinition	busDefinition.xsd et abstractionDefinition.xsd
SerifLogicalPort	busInterfaceType ajouté à un component	busInterface.xsd et component.xsd
SerifNet	interconnection et interface	subInstances.xsd
SerifService	serviceType stocké dans vendorExtension d'un component	serifService.xsd, commonStructures.xsd et component.xsd

On remarque que les classes SerifPortOnInstance SerifLogicalPortOnInstance ne trouvent pas de correspondance en IP-XACT. Ces deux classes sont en effet des particularités d'un modèle Serif (permettant l'interconnexion de net par référence plutôt que par ID de port) et n'apportent aucune information utile sur le design.

Notons que dans le cadre de cette thèse, les schémas SPIRIT IP-XACT utilisés sont ceux de la version 1.4 (mars 2008), disponible sur le site Internet du Consortium [2]. Les schémas développés et paramètres utilisés dans le cadre de cette recherche pourront être proposés comme candidats à l'extension ESL d'IP-XACT, dans le cadre du projet européen SPRINT [46], dont le Laboratoire TIMA fait partie.

5.6 API Serif

L'API Serif (Application Programming Interface) est un ensemble de fonctions qui permettent aux outils d'interagir avec la représentation interne. Elle est fournie sous la forme d'une librairie dynamique développée en C++. La plupart des fonctions de l'API sont offertes par les trois objets les plus « structurant » : SerifDesign, SerifServiceLibrary et SerifModule.

Pour utiliser l'API, il suffit d'inclure le fichier *SerifAPI.h* et de lier la librairie dynamique *libserifapi.so*. La documentation complète de l'API est fournie dans le rapport technique [13], lequel est complété par une documentation HTML générée par l'outil Doxygen. Seules les principales fonctions de construction et de parcours d'un design seront présentées ci-dessous.

Fonctions de manipulation structurelle : Ces fonctions permettent la création des modules, ports et interconnexions qui décrivent les composants logiciels, matériels et hybrides.

```
SerifDesign::create_module - Création d'une définition de module Serif
SerifDesign::create_logical_port - Création d'une définition de port logique
SerifModule::add_submodule - Ajout d'un sous module à un module existant
SerifModule::add_port - Ajout d'un port simple
SerifModule::add_logical_port - Ajout d'un port logique
SerifModule::add_net - Ajout d'un net
SerifModule::connect_net - Réalise la connexion d'un net avec deux ports
SerifModule::get_port_list - Récupère la liste des ports (simples et logiques)
```

Fonctions de spécification des services : Ces fonctions servent à créer des services requis et fournis ainsi qu'à définir leurs caractéristiques de communication - plateforme - performance.

```
SerifModule::create_service - Création d'un service requis ou fournis associé au module
SerifModule::create_comm_features - Création du premier niveau de caractéristiques du service
SerifModule::create_platform_features - Création du deuxième niveau de caractéristique
SerifModule::create_perf_features - Création du troisième niveau
SerifServiceLibrary::create_service_root - Création d'une racine d'arbre de caractéristiques
SerifServiceLibrary::find_service - Recherche d'un service dans la bibliothèque
```

Quelques fonctions utiles pour la gestion d'un design sont également proposées. Il est ainsi

2. http ://www.spiritconsortium.org

possible d'interroger l'objet SerifDesign pour recueillir des statistiques sur le design lui-même (espace mémoire occupé, nombre de modules, etc.), ou de définir un canevas de VLNV valide pour l'ensemble des modules.

5.7 Contexte méthodologique de Serif

5.7.1 Flot de design proposé

Cette section décrit un flot de design ainsi qu'un environnement logiciel basé sur la représentation intermédiaire Serif, qui permettra d'en valider certains aspects. Le flot s'articule autour des niveaux d'abstraction présentés à la Figure 2.2.

Ce flot est supporté par des outils liés les uns aux autres par Serif. Ces outils seront présentés à la prochaine section.

5.7.2 Environnement logiciel proposé

Un design Serif est fait pour être manipulé par différents outils logiciels via l'API Serif. Cette section décrit les principaux outils qui ont été réalisés dans le cadre de cette recherche. Développés afin de réaliser le flot de design, ces outils réalisent différentes opérations d'importation, de transformation et d'exportation.

L'outil SerifViewer

L'outil SerifViewer sert de « design cockpit » pour l'ensemble des outils. Il offre une interface graphique permettant au concepteur d'interagir avec un design Serif. Sa fenêtre principale est divisée en deux parties : la moitié gauche propose une série de commandes liées au flot de design, la moitié droite affiche les trois modules *top* d'un design. Le concepteur peut afficher aisément les caractéristiques de chaque objet Serif et manipuler le contenu d'un design. La Figure 5.7 illustre cette interface.

Cet outil a un rôle important dans le flot de design. Puisqu'il possède une vue d'ensemble à la fois sur un design Serif et sur les bibliothèques de services, c'est lui qui réalise les fonctions de consolidation et de construction de l'interface, décrites à la Section 4.4.3 : consolidation puis configuration - intégration. Ces opérations sont démarrées par l'utilisateur à l'aide de boutons correspondants sur l'interface graphique. Un exemple complet de consolidation - configuration sera développé dans le Chapitre suivant.

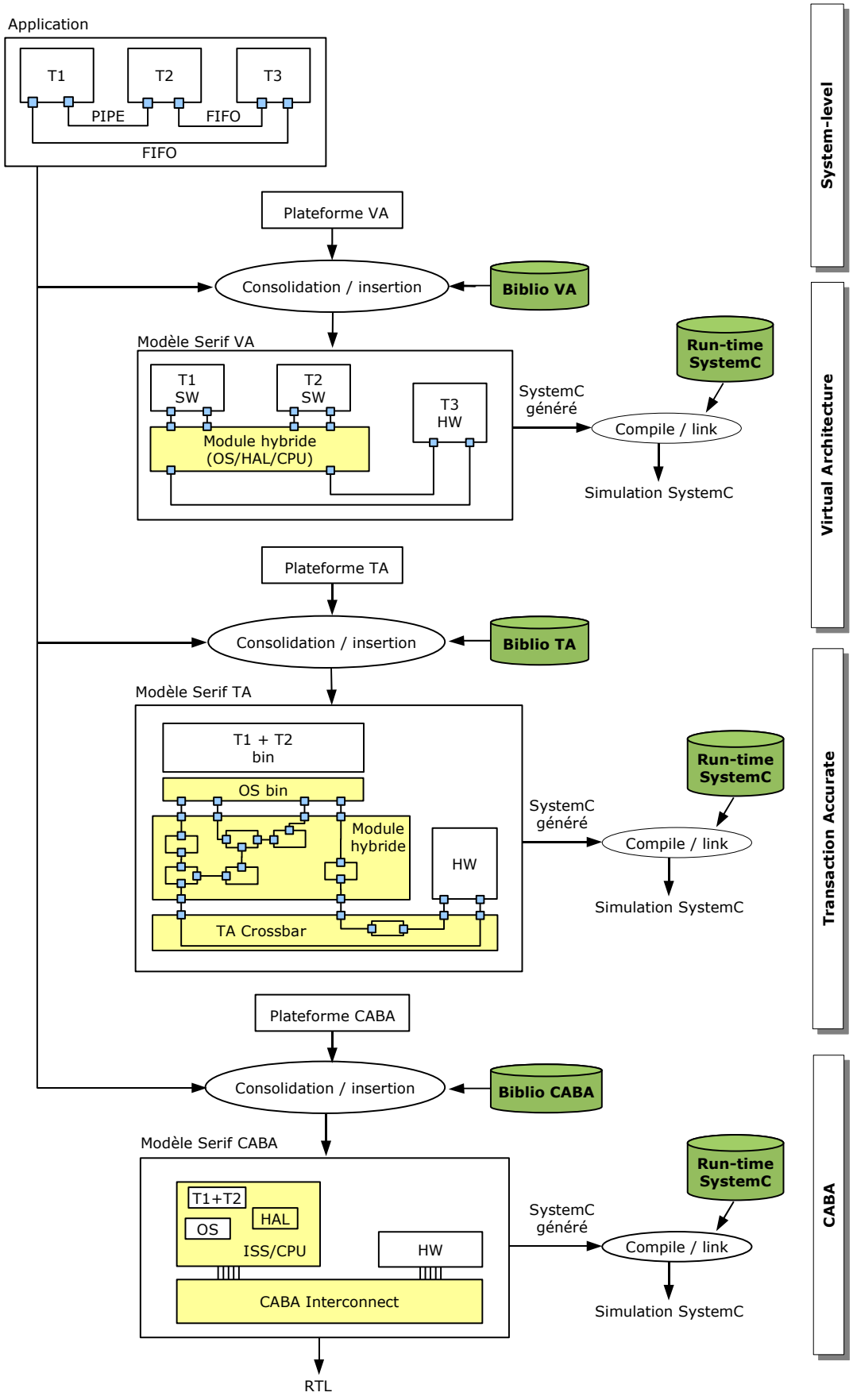

FIGURE 5.6 – Proposition de flot de conception basé sur Serif

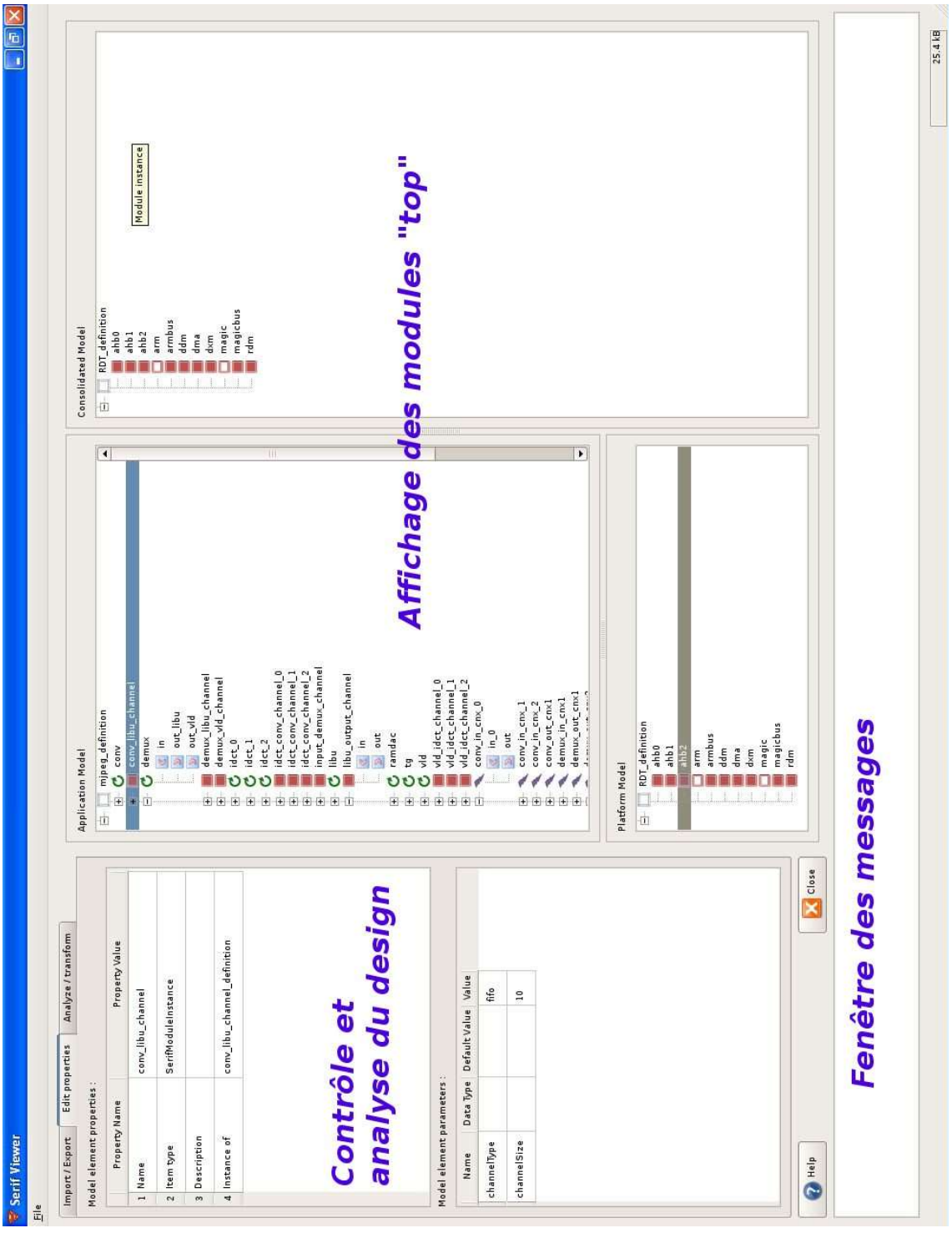

FIGURE 5.7 – Interface graphique de l'outil SerifViewer

L'outil SC2Serif

Cet outil permet la capture d'un design SystemC en Serif, ainsi que l'opération inverse, la génération de code SystemC à partir de Serif.

La capture se fait en utilisant les mécanismes d'élaboration interne de SystemC. Il est donc nécessaire d'effectuer la simulation d'un design pour initier son importation en Serif. L'outil SC2Serif récupère la structure de données interne sc_simcontext et construit un design Serif à l'aide de l'API.

La génération du prototype virtuel en SystemC a été décrite à la section 5.4.2.

L'outil Spirit2Serif

L'outil Spirit2Serif permet l'import-export entre un design Serif et le format SPIRIT IP-XACT. L'outil créé une arborescence de répertoires qui correspond au VLNV de chaque module. Par exemple, un module dont le VLNV est {TIMA, SLS, TIMER, 0.1} sera exporté dans le fichier ./TIMA/SLS/TIMER/0.1/TIMER.xml.

Le schéma d'exportation utilisé contient l'extension SerifService.xsd, stockée dans les VendorExtensions d'un Component. Ceci permet de préserver la compatibilité de la description IP-XACT avec d'autres outils, tel que l'éditeur IP-XACT de Magillem.

Cet outil se base sur les correspondances présentées au Tableau 5.8.

L'outil DOL2Serif

Cet outil permet l'importation d'une description DOL d'un système [51]. Une description DOL est composée de trois fichiers, décrivant respectivement le réseau de processus (process network), l'architecture de la plateforme matérielle, ainsi que le mapping des tâches et des canaux. Le fonctionnement de cet outil se déroule donc en trois étapes :

- Importation du process network. Les objets modules, ports et nets sont utilisés pour capturer chaque processus et leurs interconnexions.
- Importation de la plateforme. Les objets modules, ports et nets sont utilisés cette fois pour représenter les composants matériels de la plateforme.
- Importation du mapping et du scheduling. Les tâches et les processeurs y sont simplement identifiés par leur nom. Cette importation permet de spécifier le service execution des composants logiciels et hybride, en précisant le type de scheduling qui sera utilisé.

À l'heure actuelle, le format de description de la plateforme ne spécifie pas explicitement les interconnexions entre les composants matériels. Les modules Serif de la plateforme sont donc connectés en différé, une fois l'information d'interconnexion extraite de l'application

et du mapping.

À noter que l'opération inverse, l'exportation d'un design Serif vers DOL, n'est pas supportée.

5.8 Conclusion

Dans ce chapitre, nous avons proposé une représentation intermédiaire supportant le modèle des services présenté au Chapitre 4. Une interface de programmation permettant la manipulation du modèle par des outils logiciels a été décrite. Une méthodologie de design et des outils ont également été présentés. Ils seront mis en oeuvre au prochain chapitre pour le développement de deux systèmes sur puce.

Le raffinement assisté constitue un aspect complémentaire intéressant de cette recherche. Il s'agit d'insérer dans le modèle des composants extraits d'une *bibliothèque de raffinement* plutôt que d'une *bibliothèque de prototypage*. Le point de départ du raffinement reste à déterminer, entre les modèles application - plateforme ou le modèle consolidé.

L'implantation des itérateurs ainsi que de fonctions d'analyse et de transformation supplémentaires (par exemple, une fonction *instantiate*, pour la mise à plat de la hiérarchie d'un module) pourra être réalisée au fur et à mesure que les besoins apparaissent.

Chapitre 6

Génération de prototypes virtuels et validation de la représentation

Sommaire

6.1 Introduction 84
6.2 Première étude de cas : décodeur MJPEG 84
6.2.1 Préparation des spécifications pour l'entrée du flot 85
6.2.2 Modélisation du décodeur au niveau VA 88
6.2.3 Modélisation du décodeur au niveau TA 90
6.2.4 Prototypage virtuel au niveau CABA 93
6.3 Etude de performance des modèles et prototypes 95
6.3.1 Synthèse des résultats obtenus 95
6.3.2 Analyse des performances obtenues 95
6.4 Seconde étude de cas : *Wavefield Synthesis* 96
6.4.1 Capture des spécifications de l'application WFS 97
6.4.2 Analyse et résultats de la seconde étude de cas 97
6.5 Conclusion 98

6.1 Introduction

Ce chapitre décrit deux études de cas basées sur l'utilisation d'outils Serif. L'objectif poursuivi était de démontrer (1) que la représentation Serif permet la capture des interfaces d'un système sur puce réel et (2) que la génération de composants d'interfaces logiciel-matériel au sein de prototypes virtuels en SystemC est possible.

La première étude de cas a consisté à générer différents prototypes virtuels d'un décodeur MJPEG aux niveaux d'abstraction intermédiaires VA et TA. Cette étude est structurée selon le flot général de conception présenté à la Section 5.7.1, illustrant ainsi le rôle de Serif comme élément liant entre les outils et niveaux d'abstraction.

Cette étude de cas se termine par la présentation des caractéristiques dynamiques de la représentation intermédiaire et outils liés : occupation mémoire, temps requis pour l'importation depuis le format persistant et nombre de modules d'interface générés.

La seconde étude de cas démontre la généralité de l'environnement Serif, qui a été utilisé pour représenter (1) la vue abstraite d'une application, (2) la vue abstraite d'une architecture de communication et de calcul, ainsi que (3) la manière de déployer l'application sur l'architecture. Pour cette seconde étude de cas, une application de traitement du signal fut utilisée.

6.2 Première étude de cas : décodeur MJPEG

Pour cette étude de cas, nous utilisons un décodeur vidéo MJPEG, application bien maîtrisée par notre groupe de recherche. Ce décodeur, qui offre une grande qualité vidéo au prix d'une bande passante élevée, est destiné à une plateforme matérielle multiprocesseur de type SMP. Le logiciel, divisé en plusieurs *threads*, doit être exécuté par un groupe de processeurs identiques. La distribution des tâches sur les processeurs incombe au système d'exploitation. Une première architecture toute logicielle est considérée, suivie d'une architecture où une tâche particulièrement gourmande en calcul est isolée et accélérée matériellement.

Dans les systèmes sur puces, le goulot d'étranglement en terme de calcul se situe souvent au niveau des processus de traitement numérique du signal (DSP), gourmands en bande passante mémoire. Ces processus sont typiques des applications de transmission et de traitement de contenu multimédia.

Cette étude de cas suit un flot itératif illustré à la Figure 6.1. Les deux principaux outils utilisés sont l'outil de résolution de services (*ServiceResolution*) et l'outil de génération de code SystemC (*Serif2SystemC*). Les designs Serif échangés entre certaines étapes (Application, Plateforme et Consolidé) sont illustrés par les symboles S_X avec $X \in \{A, P, C\}$. On note également la sauvegarde possible de n'importe lequel de ces designs dans le format peristant

IP-XACT+.

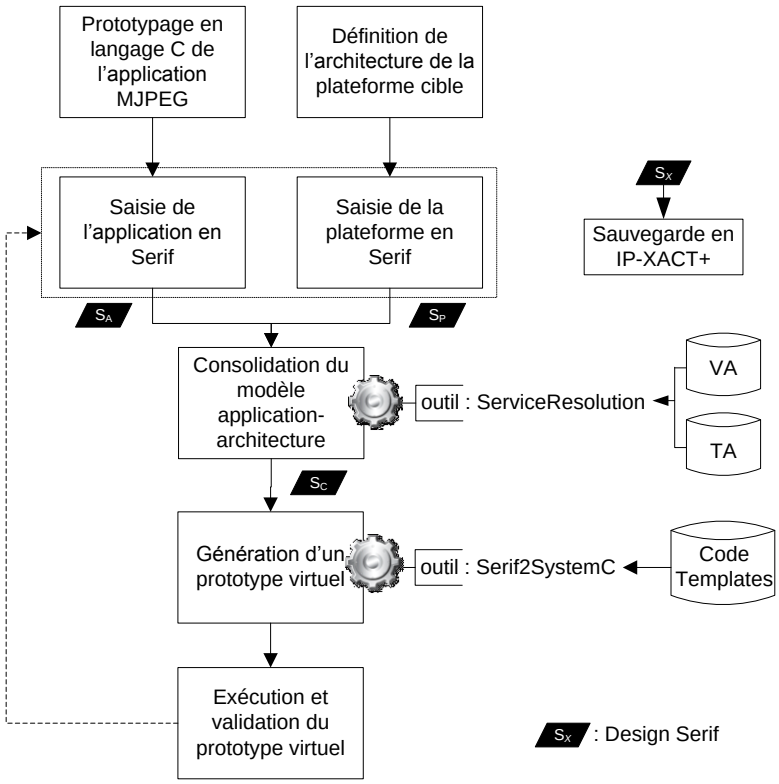

FIGURE 6.1 – Flot de conception suivi pour l'étude de cas

6.2.1 Préparation des spécifications pour l'entrée du flot

Les principales tâches du décodeur sont représentées sous forme d'un réseau de Kahn, de façon indépendante de la plateforme cible, tel qu'illustré à la Figure 6.2. Un prototype de l'application a été réalisé en langage C sur une plateforme Linux, ce qui permet une étape préliminaire de validation-profilage [2]. Dans ce prototype, chaque tâche de l'application est implantée sous forme d'un thread posix.

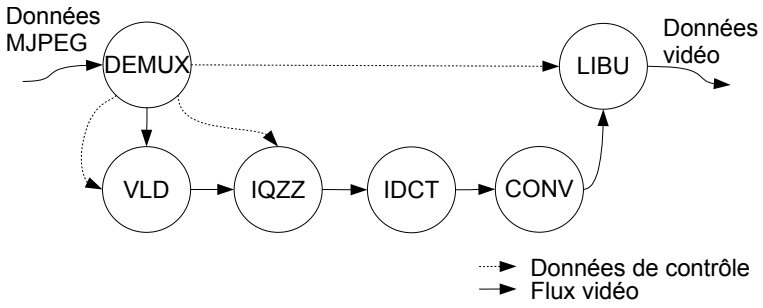

FIGURE 6.2 – Réseau de Kahn des principales tâches de l'application MJPEG

Les différentes tâches sont les suivantes :

– La tâche DEMUX analyse les données vidéo pour en extraire la taille, les tables de Huffman et les tables de quantification. 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.
– La tâche IQZZ effectue la quantification inverse du tampon précédent pour le mettre dans un nouveau tampon. Celui-ci est réorganisé suivant l'ordre zigzag.
– Le bloc IDCT exécute une transformée en cosinus discrète inverse du tampon fournit par IQZZ, et stocke le résultat dans un nouveau tampon.
– La tâche CONV effectue la conversion du format YUV au format RGB.
– La tâche LIBU stocke les blocs issus de CONV dans un dernier 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.

Le prototype est testé à l'aide d'un générateur de données vidéo (TG), qui lit les données d'un fichier, ainsi qu'à l'aide d'un module qui affiche les données vidéo décompressées à l'écran (RAMDAC).

Les tâches, décrites en langage C, communiquent entre elles grâce à des canaux abstraits, lesquels encapsulent un tampon en mémoire globale. Ces canaux offrent des méthodes d'accès *channelRead* et *channelWrite* accessibles aux tâches via des pointeurs. Au niveau système, tous les accès en mémoire liés aux canaux sont gérés implicitement par le système d'exploitation Linux.

La première étape consiste à saisir manuellement la spécification de l'application dans le format intermédiaire en vue de l'obtention d'un modèle de niveau VA. Il s'agit, pour chaque tâche, d'extraire les caractéristiques d'interface et de les représenter en Serif. Cette étape aurait pu se faire via une importation de SystemC vers Serif si la spécification MJPEG avait été développée dans ce langage. Dans la présente étude de cas, un programme en C++ est écrit à la main afin de construire la description du système en utilisant l'API Serif.

Les canaux de communication abstraits sont modélisés à l'aide de ports et les appels vers l'API HDS sont explicités sous forme de services requis. Afin de bénéficier, plus loin dans le flot, des outils de génération, ces canaux et appels de fonction doivent faire appel aux services offerts par les bibliothèques lorsque possible. Le Tableau 6.1 résume les caractéristiques d'interfaces extraites de la tâche DEMUX, en guise d'exemple.

Bien que le service soit présenté dans cette thèse comme moyen de représentation de l'interface logiciel-matériel, il peut être mis à profit au sein même du modèle logiciel ou matériel. Par exemple, les fonctions d'accès au canal utilisées par la tâche DEMUX peuvent être regroupées dans un service "HLchannel", offert par les fichiers HLChannel.h/.c. Ceci permet l'utilisation de composants de bibliothèque dédiées aux communication inter-thread et interprocessus (IPC, Inter-Process Communication), comme le mécanisme de mémoire partagée ou les *pipe*. Il est ainsi possible d'envisager la génération des interfaces logiciel-logiciel ou matériel-matériel.

TABLE 6.1 – Caractéristiques d'interface de la tâche DEMUX

Elément d'interface	**Description**	**Représentation Serif**
Canaux abstraits (Répéter 4X)		
HLchannelInit	Initialisation de canal	Port-signature *int HLchannelInit()*
HLchannelRead	Lecture dans canal	Port-signature *int HLchannelRead()*
HLchannelWrite	Ecriture dans canal	Port-signature *int HLchannelWrite()*
Fonction HDS		
printf (stdio.h)	Affichage de données sur terminal	Service requis "stdio" caractéristique de comm. "printf"

La spécification initiale du décodeur réunit toutes les tâches dans l'espace mémoire d'un processus "main" sous la forme de *pthreads*. C'est également dans cet espace mémoire que sont interconnectées les tâches à l'aide des canaux abstraits. Ce processus fédérateur est modélisé sous la forme d'un module MJPEG_MAIN de cible "SW" avec une fonction "PROCESS". Ce module requiert le service de gestion de thread offert par le fichier "pthread.h" (pour les fonctions pthread_create et pthread_join).

Le modèle Serif résultant, construit manuellement à l'aide de l'API Serif, est illustré à la Figure 6.3. Le code source ayant permis la construction de ce modèle est reproduit à l'Annexe B.

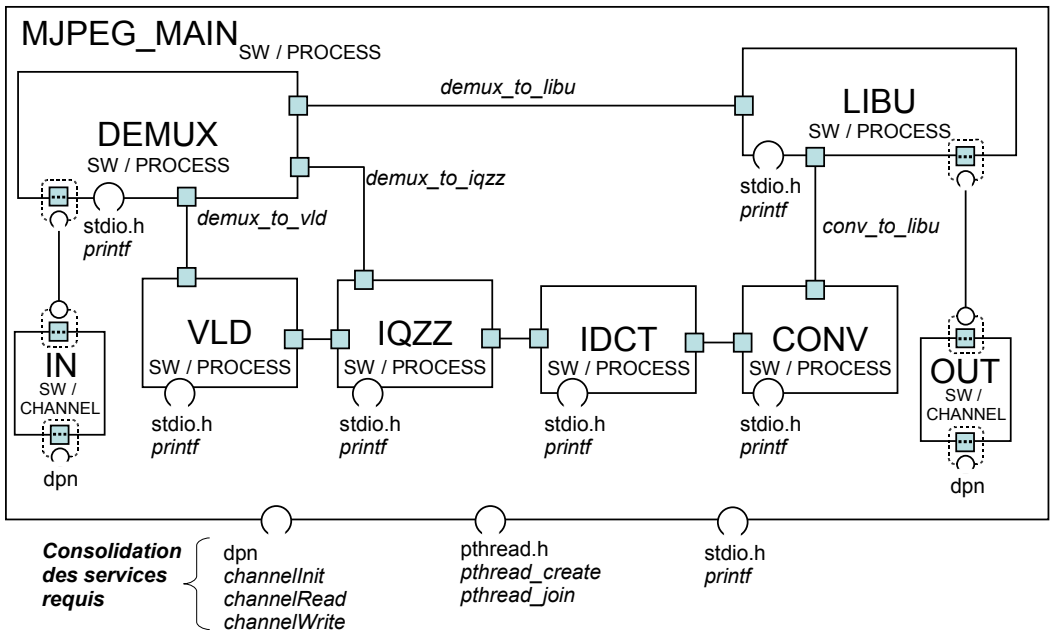

FIGURE 6.3 – Modèle de l'application MJPEG exprimé en Serif

Parmi tous les canaux de communication, seuls ceux qui communiquent avec le matériel (IN et OUT) ont été représentés de façon explicite, car c'est là qu'interviennent les outils d'analyse et de génération. Tous les services requis par les tâches et les canaux ont été consolidés sur l'interface du module MJPEG_MAIN. Par souci de clarté, les liens entre les services

de sous-modules et les services exportés n'ont pas été tracés. On comprend que le service *printf* requis par toutes les tâches ne peut être satisfait par le module MJPEG_MAIN lui-même et doit être relayé à l'interface logiciel-matériel, pour être satisfait par un élément du HDS. Quant aux services *pthread* et *dpn*, ils sont requis par le comportement du module MJPEG_MAIN lui-même.

Ce modèle illustre clairement les liens de communications et dépendances de service entre les différents composants de l'application. Chaque module possédant son propre fil d'exécution représente une cible potentielle pour une exportation en matériel. La prochaine étape propose précisément de réaliser le modèle VA du système, qui doit intégrer les éléments de l'architecture et les informations de partitionnement logiciel - matériel.

Côté plateforme, une architecture multiprocesseur a été développée au sein du groupe SLS dans le cadre des travaux présentés dans [20]. Cette architecture est représentée de façon schématique à la Figure 6.4. On y trouve plusieurs unités d'exécution de type SMP (EU), un composant mémoire, des composants de génération de trafic et d'affichage d'image (TG et RAMDAC), une mémoire pour les sémaphores (SEMRAM), un terminal pour interaction avec l'usager (TTY) ainsi qu'un réseau de communication (Generic_NoC). Nous disposons des modèles de niveau TA et CABA de cette plateforme, développés en SystemC.

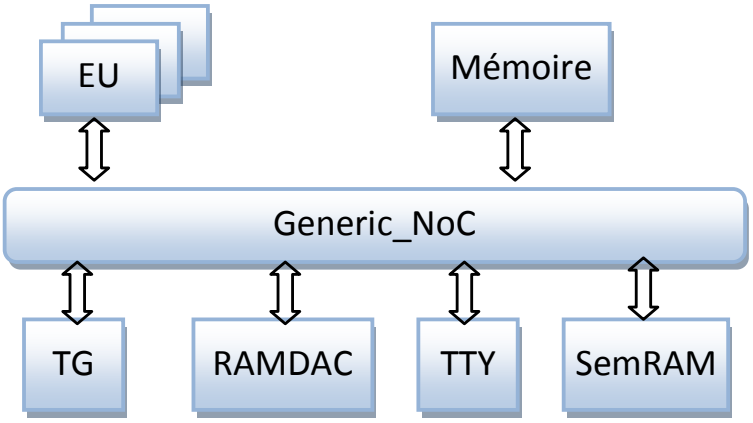

FIGURE 6.4 – Modèle de l'architecture multiprocesseur utilisé

6.2.2 Modélisation du décodeur au niveau VA

Le modèle de niveau VA est un modèle mixte application-architecture. Pour l'instant, nous ne disposons que du modèle Serif de l'application, construit manuellement à partir du réseau de Kahn. Il faut donc introduire les informations en provenance de la plateforme cible : le nombre de processeurs, les principaux modules matériels et les canaux de communication abstraits.

Les informations contenues dans le modèle de la plateforme doivent pouvoir être exprimées au niveau VA afin d'effectuer l'intégration application-architecture et de générer un premier

prototype virtuel. Rappelons qu'au niveau VA l'interface logiciel-matériel est *complètement implicite*, n'exposant que l'API HDS d'une part et des canaux de communication abstraits vers les composants matériel d'autre part.

Le modèle Serif illustré à la Figure 6.5 est capturé à l'aide d'un code source écrit à la main, reproduit en deuxième partie de l'Annexe B. Le modèle contient les deux modules matériels utilisés pour la génération de trafic (TG) et l'affichage de l'image (RAMDAC), ainsi que le module hybride ABSTRACT_CPU_SS représentant le groupe de processeur SMP. Les services que devra offrir ce module sont illustrés sur son interface : le service d'exécution *exec*, le service de communication *dpn* ainsi que les services d'HDS *pthread* et *stdio*. On assigne à ce dernier une plage d'adresse temporaire, au sein des caractéristiques de plateforme du service *dpn* : 0x10001C00 pour l'accès en lecture au composant TG, 0x20001800 pour l'accès en écriture au composant RAMDAC. Les deux premiers octets de l'adresse déterminent le composant cible. Les deux derniers octets de chaque adresse, 0x1C00 et 0x1800, déterminent l'accès aux registres de lecture / écriture du canal de communication, tel que défini par le concepteur du canal.

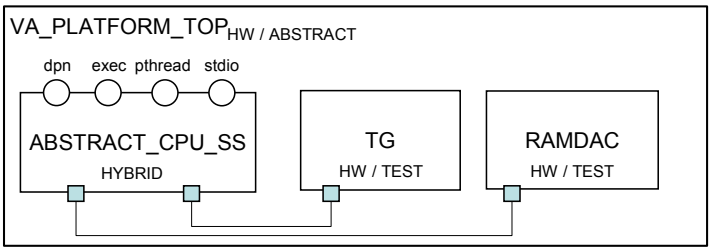

FIGURE 6.5 – Composants de la plateforme exprimés en Serif

De son côté, l'application MJPEG est encapsulée dans un module abstrait SW_STACK_1 qui délimite la pile logicielle à associer à un processeur. Ce module requiert le service exécution, qui devra être satisfait par un élément hybride de la plateforme.

La mise en place des composants de l'interface logiciel-matériel s'effectue automatiquement au sein du modèle consolidé à l'aide de l'outil *ServiceResolution*, qui effectue les étapes de résolution et d'intégration (Section 4.4.3).

Dans le cas du système MJPEG, les résolutions de services suivantes sont effectuées :
– Le service *exec* requis par le module SW_STACK est associé au service *exec* fourni par le module hybride ABSTRACT_CPU _SS.
– Le service *dpn* est résolu entre le module MJPEG_MAIN et le module ABSTRACT_CPU _SS par l'ajout des canaux de communication *dpn* dans le module hybride.
– Les services *pthread* et *stdio* requis par le module MJPEG_MAIN sont résolus par le système d'exploitation hôte. Le nom des fichiers entête et librairies qui fournissent ces services sont stockés dans les caractéristiques de plateforme des services fournis par le module hybride.

Les liens entre services requis-fournis sont insérés dans la table de résolution de services du module TOP consolidé. Le module résultant est illustré à la Figure 6.6.

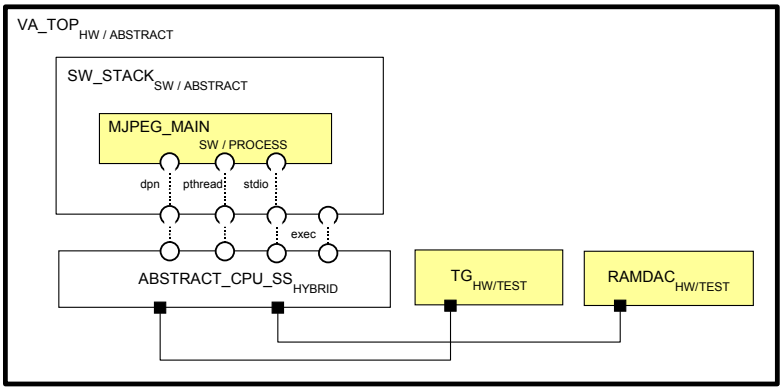

FIGURE 6.6 – Modèle consolidé application - architecture au niveau VA

Toutes les informations d'interface étant en place, ce modèle VA est prêt pour la génération d'un prototype virtuel en SystemC, selon les étapes décrites à la Section 5.4.2.

Le prototype résultant est exécuté pour une première validation. Sa vitesse d'exécution observée (temps de traitement d'une image) est plus lente que le même modèle décrit en logiciel pur, mais permet de bénéficier de l'environnement de simulation SystemC pour l'analyse et le debug.

Afin d'accélérer le temps de traitement d'une image, nous allons réaliser la tâche IDCT, qui accapare près de 40% du temps de traitement, en matériel. Ceci implique que les canaux utilisés pour la connexion aux modules IQZZ et CONV soient redirigés vers la plateforme matérielle. Deux canaux sont ainsi ajoutés au modèle : TO_IDCT et FROM_IDCT. La plateforme, elle, est équipée d'un module matériel IDCT, qui possède la même interface d'accès que les deux autres modules matériels. Ces modifications sont illustrées à la Figure 6.7.

Une nouvelle étape de consolidation - génération est effectuée, ce qui requiert au préalable quelques ajustements au niveau de la plage d'adressage et des accès à la plateforme, pour tenir compte du nouveau composant matériel IDCT. Ceci se fait au prix d'un temps de simulation encore plus long dû à la complexité de cette configuration. Cependant, le processeur étant libéré du traitement IDCT, on peut s'attendre à un gain de performance appréciable. Les analyses de performances du niveau TA permettront de comparer les performances de ces deux configurations.

6.2.3 Modélisation du décodeur au niveau TA

La génération d'un prototype virtuel au niveau Transaction Accurate (TA) permet une validation et un profilage plus précis du système. Il est possible de réutiliser les spécifications de l'application en l'associant cette fois à une plateforme modélisée au niveau transactionnel. Ainsi, l'application de la Figure 6.3 est combinée à la plateforme TA de la Figure 6.8.

À ce niveau d'abstraction, le système d'exploitation est explicite. Un module OS_PROCESS

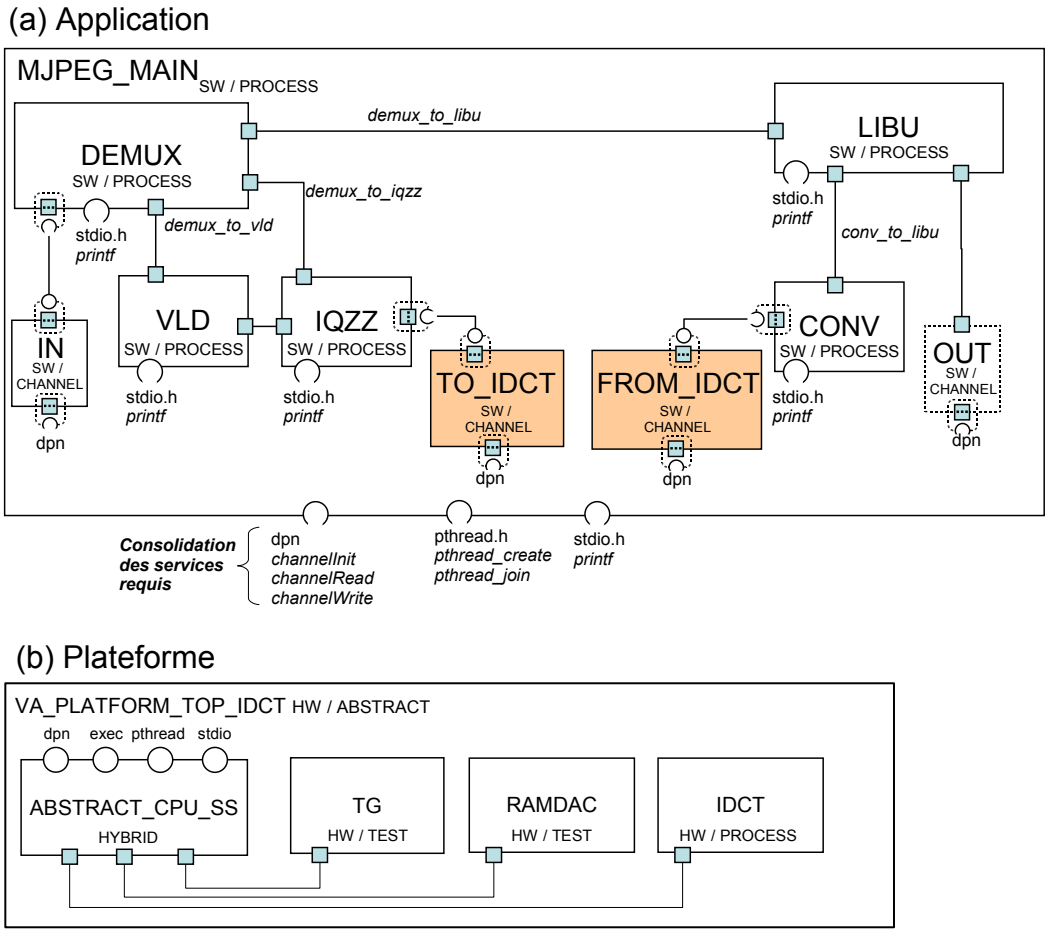

FIGURE 6.7 – Application MJPEG et plateforme pour accélération matérielle de l'IDCT

est donc ajouté à la pile logicielle pour le représenter. L'association entre la tâche MJPEG et l'OS est spécifiée avec le mécanisme de mapping de service *exec*. Un deuxième mapping de ce genre est également requis entre l'OS et le module hybride de la plateforme TA.

La plage d'adressage associée au service *dpn* doit maintenant tenir compte de l'ensemble des communications avec le matériel, de même que des contraintes imposées par l'OS lui-même. Les détails de cette nouvelle plage d'adressage apparaissent dans le Tableau 6.2.

On remarque la présence des services normalisés pour le niveau TA, offerts par le composant hybride : *exec*, *io_access*, *context*, *mp*, etc.

TABLE 6.2 – Plage d'adressage du modèle TA

Base Address	Width (bytes)	Target Port
0xA2001C00	4	channel-tg
0xA3001800	4	channel-ramdac
0xA4001800	4	channel-to-idct
0xA5001C00	4	channel-from-idct

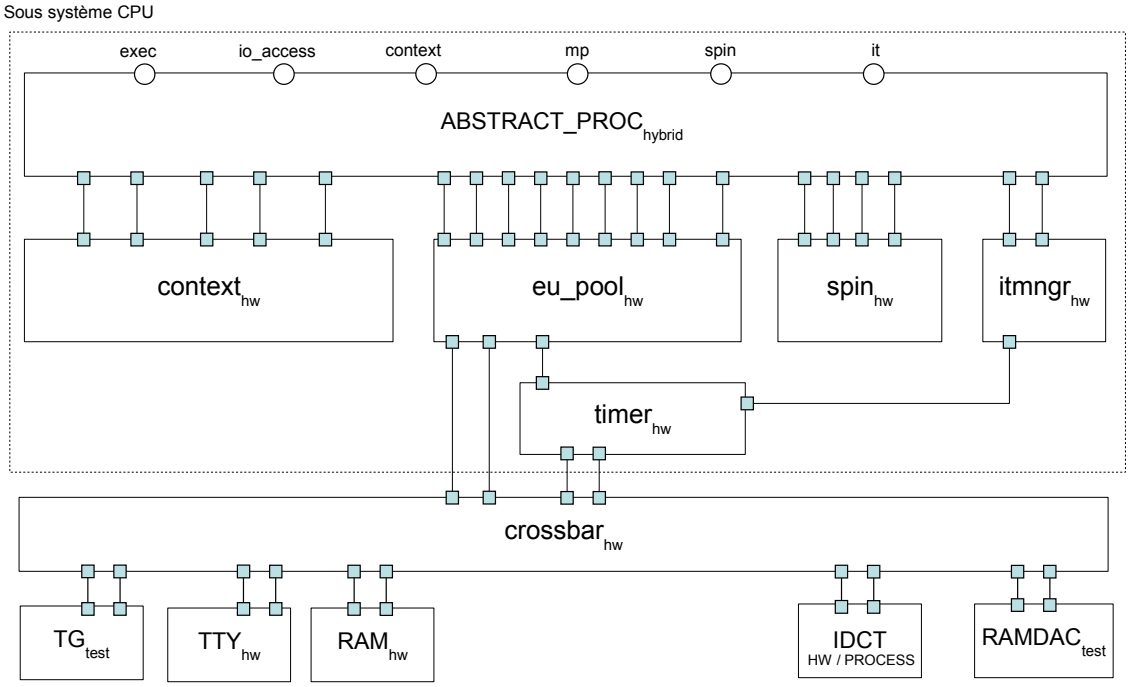

FIGURE 6.8 – Modèle de la plateforme au niveau TA

Cette fois, les étapes de résolution - intégration sont effectuées à l'aide de la bibliothèque d'éléments du niveau TA. Les services exécution sont résolus en priorité, suivi des services HAL et d'entrée-sortie (io_access). Ces deux étapes résultent en un modèle Serif consolidé, illustré à la Figure 6.9.

Ce modèle consolidé sert de base à la génération SystemC des prototypes virtuels. Deux prototypes sont générés : le premier où toutes les tâches du décodeur MJPEG sont en logiciel, et le second où la tâche IDCT est réalisée en matériel. La différence entre les deux prototypes se résume à l'utilisation de deux exécutables MJPEG_MAIN différents. La même plateforme matérielle peut être utilisée dans le deux cas, sachant que dans le cas « tout logiciel » le module matériel IDCT sera superflu.

Il faut souligner le rôle d'un autre outil dans la manipulation du modèle, le *SerifViewer*, qui permet de visualiser les différents modules top et leur sous-modules. La Figure 6.10 illustre une capture d'écran de l'outil Serif avec les composants d'application et d'architecture. Cet outil illustre bien le genre d'opérations qui sont facilitées par l'utilisation d'une représentation intermédiaire.

La performance estimée du système lorsque toutes les tâches sont en logiciel est de 120 ms pour le traitement d'une image, ce qui donne un débit d'à peine 8 images / seconde, ce qui est bien insuffisant. L'implantation en matériel de l'IDCT permet de réduire le temps de traitement d'une image à 40 ms, soit environ 25 images / seconde. Ceci signifie que le décodage MJPEG a peu de chance de pouvoir fonctionner en parallèle avec d'autres fonctions sur un même processeur, notamment les fonctions d'interface avec l'utilisateur.

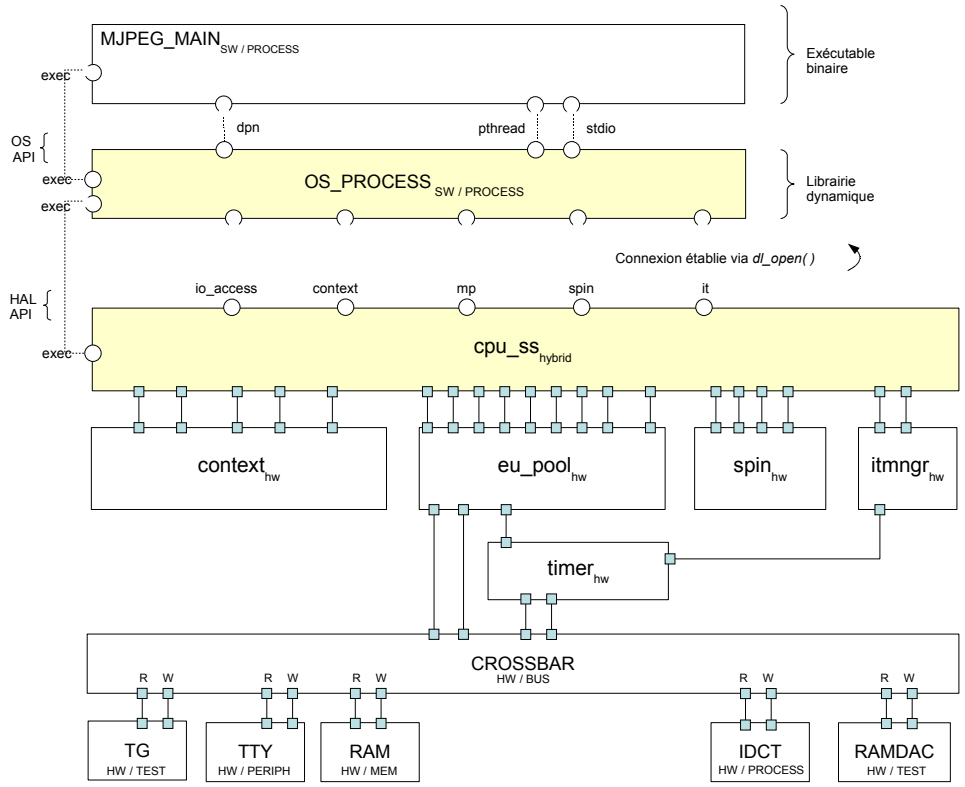

FIGURE 6.9 – Modèle consolidé application-architecture au niveau TA

Tel que démontré, l'exploration d'architecture se trouve facilitée par l'utilisation de la représentation Serif. Les changements à apporter au modèle se limitent au remapping manuel des services *exec* et à la reconfiguration des plages d'adressage. La génération automatique des protoypes virtuels en SystemC soulage le concepteur d'une bonne part du travail de ré-écriture.

6.2.4 Prototypage virtuel au niveau CABA

Le prototype virtuel de l'application MJPEG au niveau CABA repose sur les composants de la bibliothèque SoCLib [1]. Un ISS du processeur ARM est utilisé afin de compléter le modèle et d'obtenir une bonne précision pour les analyses de performance.

Les interfaces entre composants matériels sont basées sur la norme VCI, tel que le prescrit la librairie SoCLib. La génération d'un prototype virtuel SystemC se fait en insérant trois modèles de processeur ARM (pour réaliser l'architecture SMP), les interfaces VCI corres-

1. Le projet SoCLib est une plateforme de modélisation et de simulation pour systèmes MPSoC développée en SystemC. Voir `www.soclib.fr`

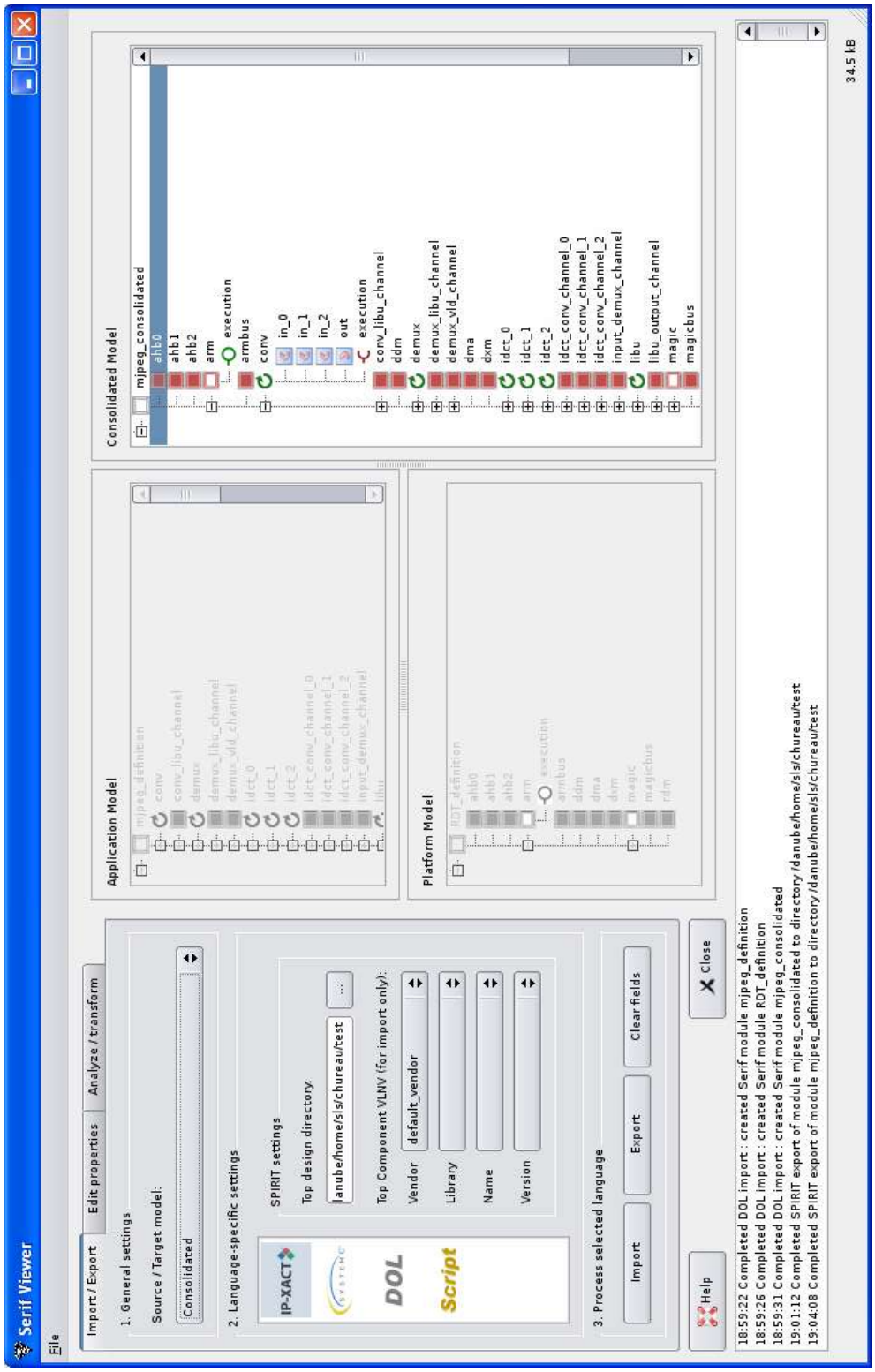

FIGURE 6.10 – Outil de visualisation d'un design Serif : l'application, la plateforme et le modèle consolidé

pondantes et en effectuant la synchronisation des accès mémoire.

L'estimation de performance est alors la plus précise, au prix d'un temps de simulation plus long. La génération du prototype virtuel à ce niveau nécessite un temps de construction des bibliothèques que ne permettent pas les contraintes de cette thèse et est laissé comme perspective.

6.3 Etude de performance des modèles et prototypes

6.3.1 Synthèse des résultats obtenus

La performance des prototypes virtuels a été mesurée selon plusieurs critères, qui permettent d'en apprécier l'utilité et les limites. Ces métriques sont présentées dans le Tableau 6.3.

TABLE 6.3 – Mesures de performances des différents modèles et prototypes

Métrique	**MJPEG**				
Niveau d'abstraction	SL[1]	VA[2]	VA[3]	TA	CABA
Nb de module SW	6	6	5	5	4
Nb de module HW	3	4	5	6	11
Nb de module I/F	s.o.[4]	1	1	2	s.o.
Perf. observée (s)	s.o.	0,3	0,5	1,5	4
Occup. mémoire (kB)	s.o.	8,2	9,4	21,1	s.o.
Temps importation IP-XACT[5] (ms)	s.o.	841 (appl) 409 (pf)	865 (appl) 490 (pf)	765 (appl) 945 (pf)	s.o.
Temps génération SystemC (ms)	s.o.	72	80	90	130

[1] Modèle de référence, en C
[2] IDCT SW
[3] IDCT HW
[4] Sans objet
[5] Temps de chargement de l'application (appl) et de la plateforme (pf)

6.3.2 Analyse des performances obtenues

À partir de ces résultats, nous observons que les modèles Serif sont constitués d'un nombre relativement faible de modules. En effet, bien que tous les modules représentant une application ou une plateforme puissent être représentés, seul les modules faisant appel aux services sont traités lors de la génération. Ceci est important dans la mesure où un système peut être constitué de dizaines de processeurs, sur lesquels peuvent être exécutés des centaines de tâches. On peut donc anticiper un bon contrôle sur la complexité des modèles, qui croît moins rapidement que celle des applications à représenter.

Le gain en temps de conception reste un élément difficile à évaluer. Il est entendu que plus le nombre d'architectures à explorer est élevé, plus le bénéfice d'utilisation d'un format comme Serif est appréciable. Il faut cependant relativiser ce bénéfice en prenant en considération le temps de développement des bibliothèques. Dans la présente étude de cas, des bibliothèques simples ont été développées dans une optique de preuve de concept. La représentation en Serif de plateformes complexes comme SoCLib (environ 30 IP à modéliser, chacun à deux niveaux, TA et CABA) ou SystemC TLM exigerait certainement un effort d'une dizaine de jour voire quelques semaines. Le bénéfice d'une bibliothèque reste lié à sa flexibilité et à sa réutilisation, ce qui, dans le contexte des services, semble être acquis.

Enfin, on constate que le format a été en mesure de représenter différentes configuration de l'application MJPEG, à différents niveaux d'abstraction. Ceci est significatif dans la mesure où les mêmes outils ont pu être réutilisés, que ce soit pour la génération de code SystemC ou la visualisation du design. Cette généralité du format a de nouveau été mise en évidence dans la seconde étude de cas.

6.4 Seconde étude de cas : *Wavefield Synthesis*

Dans cette seconde étude de cas, une application de traitement du signal (*Wavefield Synthesis* ou WFS) est exécutée sur une plateforme modulaire haute performance. Cette plateforme, basée sur un processeur Diopsis 940 d'Atmel, est utilisée dans le cadre du projet européen SHAPES [45]. L'application WFS permet de créer un environnement sonore virtuel, constitué de plusieurs source artificielles.

L'objectif est de répartir les tâches de l'application WFS sur les deux processeurs du Diopsis, le DSP et le microcontrôleur. Contrairement à l'application précédente, aucune tâche ne sera implantée en matériel, toute l'application est réalisée en logiciel. Une fois la répartition des tâches déterminée, les composants HDS et HAL de l'interface logiciel-matériel sont générés sur mesure afin d'obtenir un prototype exécutable du système. Le niveau CABA est le seul niveau visé.

Cette deuxième étude de cas suit un flot de conception différent, illustré à la Figure 6.11 [2]. Ce flot repose sur deux outils spécialisés qui permettent de générer le code de l'interface ainsi que des fichiers annexes comme le « Makefile ». Nous attirons l'attention sur l'outil AGES, qui utilise Serif comme représentation intermédiaire pour faciliter l'analyse et la manipulation des tâches et des processeurs du système.

2. *Extraite de* X. Guérin et A. Chureau, "Software Stack Generation for Heterogeneous MPSoCs starting from a High-Level Application Model", présentation à CASTNESS'08, Rome, 2008.

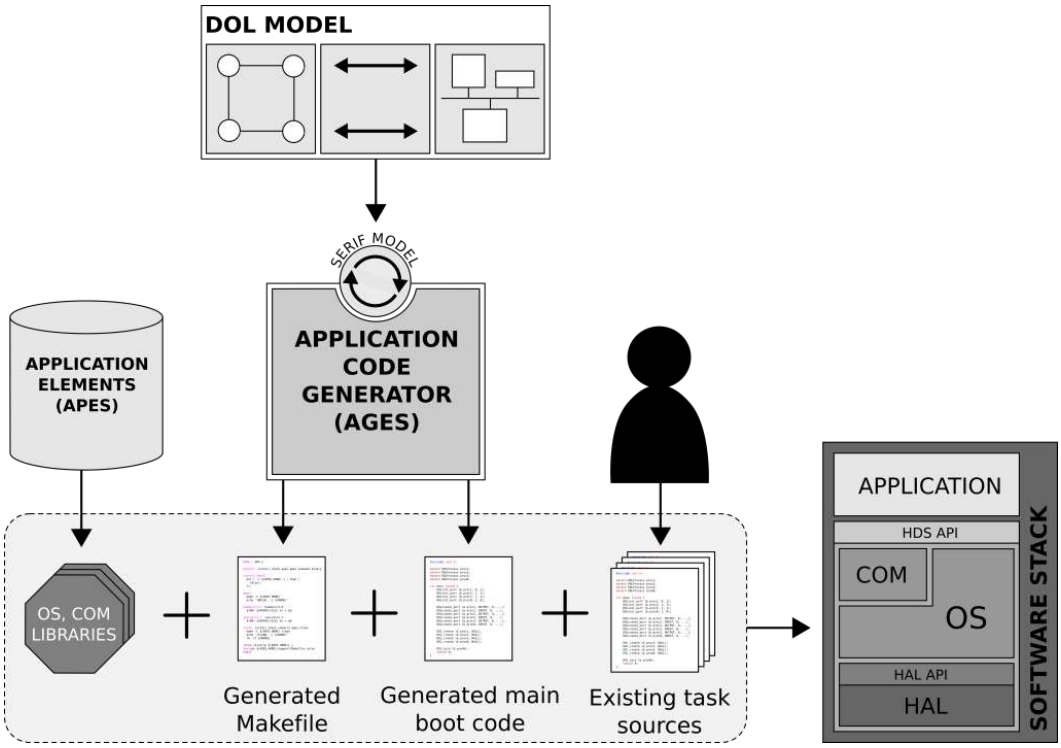

FIGURE 6.11 – Flot de conception suivi pour la seconde étude de cas

6.4.1 Capture des spécifications de l'application WFS

Le schéma DOL est utilisé pour représenter les spécifications du système. Selon ce schéma, un système est décrit par trois fichiers XML : le fichier *process network*, qui représente les tâches de calcul et de communication sous forme d'un graphe ; le fichier *architecture*, qui décrit les ressources matérielles disponibles ; le fichier *mapping*, qui associe les tâches de l'application aux ressources de la plateforme. Les fichiers sources du comportement des tâches, écrits en langage C, sont fournis séparément.

L'outil *dol2serif*, présenté à la Section 5.7.2, permet d'interpréter chacun de ces fichiers. Les modules top « application » et « platform » sont respectivement créés à partir des fichiers *process network* et *architecture*. Le fichier *mapping* est utilisé pour créer directement le modèle consolidé application-architecture.

6.4.2 Analyse et résultats de la seconde étude de cas

Le format DOL contient des informations qui n'avaient d'abord pas été prévues dans le format Serif. De nouveaux objets ont donc été ajoutés à la représentation intermédiaire, afin de capturer les spécifications DOL dans leur intégralité.

Les objets SerifPath et SerifPathElement ont été ajoutés afin de capturer les chemins de communication privilégiés utilisés entre certaines tâches. Une liste d'objets SerifPath a été

ajoutée comme attribut de l'objet SerifDesign, seul objet à posséder une vue d'ensemble de tous les éléments de ces chemins. D'autres informations DOL ont été capturées à l'aide de paires nom-valeur de type SerifParameter. C'est le cas des spécifications du type de mémoire ainsi que des propriétés channelType et channelSize, associées aux modules de type SW-CHANNEL.

Le mapping tâche-processeur se fait directement à l'aide des informations fournies par le fichier mapping de DOL, sans passer par les services *exec*. Puisque le prototype est généré au niveau CABA, le mapping consiste simplement à spécifier les tâches qui seront chargées par chacun des processeurs. Les fonctions de l'API Serif utilisées à cette fin sont SerifProcessorInstance : :add_task(), SerifProcessorInstance : :set_task_config() et SerifTaskInstance : :set_processor(). L'outil AGES accède à ces informations à l'aide des fonctions SerifProcessorInstance : :get_task_list().

Bien que ce soit principalement un outil spécialisé qui ait bénéficié de la représentation intermédiaire, d'autres outils basés sur Serif ont été mis à contribution. C'est le cas notamment du SerifViewer, utilisé pour visualiser les spécifications système. La Figure 6.12 illustre les trois top du système WFS (application, plateforme et consolidé), avec les propriétés du canal *fifo_conv_ls* affichées dans la moitié gauche de la fenêtre.

6.5 Conclusion

Le format intermédiaire Serif a été mis en pratique dans ce chapitre par le biais d'une étude de cas portant sur le développement d'un décodeur MJPEG. Les différents composants du système ont été modélisés à l'aide de Serif, tant pour l'application que pour la plateforme matérielle. Trois niveaux d'abstraction ont été explorés : le niveau système, qui a servi de niveau d'entrée, le niveau VA et le niveau TA.

Serif a permis de capturer les éléments d'interfaces des composants logiciels et matériels à ces différents niveaux. Les outils de résolution de services, de génération de code SystemC et de visualisation d'un design ont été également présentés.

Il est difficile, dans ce contexte expérimental, d'évaluer l'effort de construction des bibliothèques. Ce délai a un impact direct sur le bénéfice offert par l'utilisation de Serif et il doit être minimisé.

L'automatisation du processus d'exploration d'architecture apporterait une réduction supplémentaire du temps de conception. D'autres outils spécialisés, comme pour la vérification ou l'analyse de performance d'un système multiprocesseurs, peuvent également s'appuyer sur cette représentation intermédiaire pour leur mise en oeuvre.

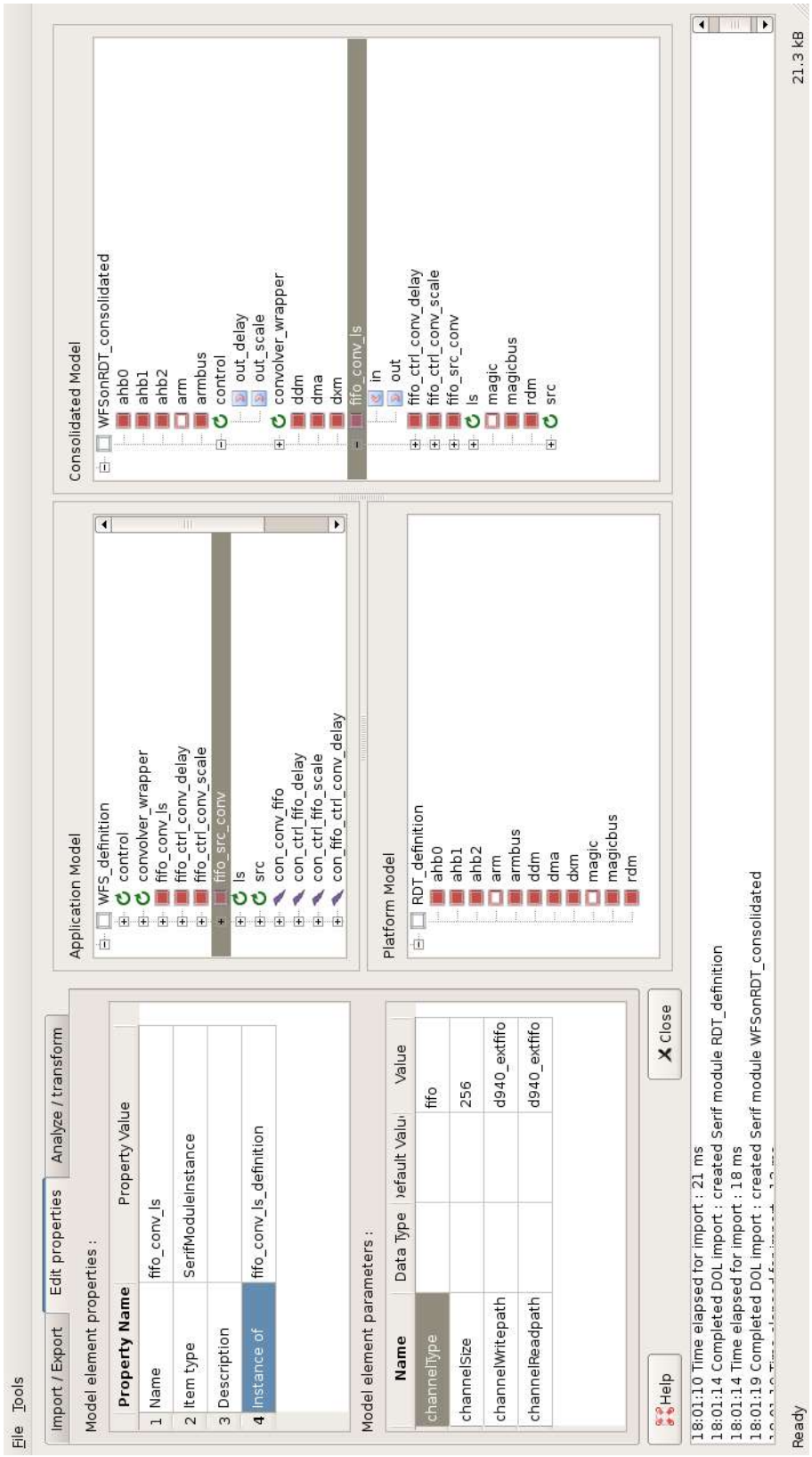

FIGURE 6.12 – Outil SerifViewer illustrant le système WFS

Chapitre 7

Conclusion et perspectives

Cette recherche avait comme objectif de développer une représentation intermédiaire qui permette de raisonner sur les interfaces logiciel-matériel. Ces interfaces, candidates à la génération automatique lors de l'exploration d'architecture, sont légion dans les sytèmes sur puce. Il convenait d'abord de bien cerner la nature de ces interfaces aux différents niveaux d'abstraction utilisés dans le design.

La composition de l'interface logiciel-matériel varie selon qu'on la considère à haut niveau d'abstraction ou plus près de l'implémentation. Au niveau spécification, l'interface se résume à de simples canaux de communication abstraits entre les tâches. Au niveau de l'implémentation, l'interface est constituée de différents composants logiciels et matériels. Dans cette recherche, nous avons également considéré deux niveaux d'abstraction intermédiaires, le niveau architecture virtuelle et le niveau transaction, qui permettent un raffinement graduel du système, au prix d'un temps de simulation croissant.

Il était donc nécessaire de représenter trois principaux types de composants, présents dans l'interface logiciel-matériel à tous les niveaux d'abstraction : les composants logiciels, les composants matériels et les composants hybrides. Ces derniers jouent un rôle majeur dans l'abstraction de l'interface, en permettant à des tâches logicielles de communiquer avec des tâches matérielles. Nous avons choisi de représenter ces trois types de composants à l'aide d'un objet module, et leurs interconnexions à l'aide de ports et de nets, un modèle compatible avec la plupart des langages de conception actuels.

La génération des interfaces repose sur l'analyse des besoins de communication et dépendances entre les composants logiciels et matériels. Afin d'exprimer ces caractéristiques, nous avons utilisé le modèle des services, reconnu dans plusieurs domaines de l'informatique comme un moyen approprié d'intégrer des composants hétérogènes. Ainsi, les composants de part et d'autre de l'interface logiciel-matériel peuvent requérir ou fournir des services, caractérisés par leurs canaux de communication, leur contraintes de plateforme et leur niveau de performance. C'est le rôle des outils de trouver, dans une bibliothèque de service, les composants appropriés pour réaliser l'interface logiciel-matériel. Dans le cadre de cette

thèse, la résolution de service se fait par correspondance d'identificateur. La résolution de service basée sur la valeur des paramètres eux-mêmes exigerait une notation spécialisée, comme celle proposée par le profil MARTE. L'intégration d'outils supportant à la fois Serif et MARTE représente une perspective très intéressante.

Les bibliothèques, spécifiques à chaque niveau d'abstraction, contiennent typiquement des composants de HDS (système d'exploitation et pilotes d'entrée-sortie), de HAL (pilote de matériel de bas niveau), des composants de sous-système processeur (timer, DMA, sem-RAM), ainsi qu'un élément hybride. Le composant hybride abstrait plus ou moins de fonctions selon le niveau d'abstraction ciblé et possède une interface logicielle et une interface matérielle, toute deux configurables.

Ce modèle des services a été traduit sous forme d'une structure de données, manipulable via une interface de programmation et pouvant être sauvegardée dans un format persistant. Ces éléments composent ensemble une représentation intermédiaire basée sur une approche service, ou *Service-Based Intermediate Format* (Serif). Cette représentation intermédiaire est développée en C++ et repose sur une adaptation du format SPIRIT IP-XACT pour la sérialisation. Des outils d'import-export ont été développés afin de capturer et générer des designs en SystemC, SPIRIT ou DOL. Des outils de manipulation du design ont été développé afin de résoudre les services requis à l'aide de composants de bibliothèque. Ceci a rendu possible la production de modèles Serif consolidés, où l'interface logiciel-matériel est réalisée à différents niveaux d'abstraction. Ces modèles consolidés, destinés à la génération de prototypes virtuels, sont construits de façon semi-automatique, accélérant l'exploration d'architecture.

Une étude de cas a été présentée afin de valider la possibilité de construire des outils de génération de prototypes virtuels de systèmes-sur-puce. L'étude de cas a consisté à implanter une application de décodage MJPEG sur une plateforme personnalisée, aux niveaux d'abstraction VA et TA. L'exploration de deux différentes architectures a été réalisée, l'une adoptant une architecture tout-logiciel, l'autre effectuant l'accélération d'une tâche en matériel. Il a été démontré que le format Serif permet la capture de spécifications d'interfaces logiciel-matériel à plusieurs niveaux d'abstraction, ainsi que la manipulation de ces interfaces par des outils spécialisés.

Cette recherche expose également quelques limites à l'utilisation d'une représentation intermédiaire. Par exemple, l'effort de développement des bibliothèques conditionne la productivité des outils. Pour contrer cela, il faudrait affiner la résolution des services et permettre la synthèse automatique d'interfaces à partir de descriptions de protocole. Néanmoins, le coût relatif de construction des bibliothèque diminue avec le nombre d'architectures différentes générées.

On remarque également la complexité de la manipulation du code source des applications. Hormis une intervention manuelle coûteuse, il est très difficile d'expliciter toutes les dépendances implicites d'un logiciel. Le *parsing* de code source (C, C++, Java) est un processus complexe et sujet à l'interprétation. Le simple déréférencement d'un pointeur, par exemple,

devrait être explicité lorsque des modèles à bas niveau d'abstraction doivent être générés. Ainsi, la représentation intermédiaire est avantageuse dans la mesure où les concepteurs ont utilisé des éléments de communication « normalisés », i.e. qui peuvent être interprétés par les outils.

Les outils développés dans le cadre de cette recherche avaient principalement pour but de guider le développement du format. Nous voyons donc comme perspectives le développement d'outils plus performants, faisant levier sur la facilité d'accès et de traitement d'information que procure la représentation intermédiaire. Le besoin d'outils de raffinement est notamment souligné, ainsi que des outils d'analyse de performance et de vérification.

Annexe A

Schema des services

Schéma utilisé pour le stockage des services dans un élément component/vendorExtension.

Listing A.1 – serifService.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
// Description : service.xsd
// Author:       Alexandre Chureau
// Version:      $Revision$
// Date:         $Date$
// Tag:          $Name$
-->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="/danube/home/sls/chureau/workspace/spirit_xsd_cpp/sls"
           xmlns:spiritsls="/danube/home/sls/chureau/workspace/spirit_xsd_cpp/sls"
           elementFormDefault="qualified"
           attributeFormDefault="qualified">

  <xs:complexType name="service_t">

    <xs:annotation>
      <xs:documentation>
        Specification of a Service associated to the component
      </xs:documentation>
    </xs:annotation>

    <xs:sequence>
        <xs:element name="serviceName" type="xs:string">
            <xs:annotation>
                <xs:documentation>
                Specification of a Service associated to the component
                </xs:documentation>
            </xs:annotation>
        </xs:element>

        <xs:element name="serviceType" type="xs:string">
            <xs:annotation>
                <xs:documentation>
                    Type of service : Communication or Logistics
                </xs:documentation>
            </xs:annotation>
        </xs:element>


        <xs:element name="serviceDirection" type="xs:string">
                <xs:annotation>
                        <xs:documentation>
                        Relation of the service to the component : Provided or Required
                        </xs:documentation>
                </xs:annotation>
        </xs:element>



    </xs:sequence>
```

```
52     </xs:complexType>
53
54     <xs:element name="service" type="spiritsls:service_t"/>
55
56  </xs:schema>
```

Annexe B

Capture du design MJPEG en Serif

Code source C++ utilisant l'API Serif pour la capture de l'application MJPEG et de la plateforme associée.

```
/** @file
 * Case study program : creation of a VA model from a SL model
 *
 * Copyright (c) 2007 System-Level Synthesis Group at TIMA
 *
 * $Id$
 *
 * Creation Date  : November 2007
 * Contributors   : Alexandre Chureau
 */

#include <iostream>
#include <sstream> // For int to string conversion

#include "SerifAPI.h"
#include "SpiritTranslator.h"

#define BASE_DIR_SOURCE string("/danube/home/sls/chureau/workspace/mjpeg_models/mjpeg-1-SL"
    )
using namespace std;

/*
 * This program will create a Serif Model of the MJPEG application used by Patrice Gerin &
    co.
 * The original source files are located in chureau/workspace/mjpeg_models/mjpeg-1-SL
 * The role of the program is to create Modules, Ports & Nets that represent the main tasks
 * and channels of the application. The modules are then further characterized to determine
 * if they are SW, HW or ABSTRACT. Original source files are attached to the Serif elements
    .
 * The resulting model is called the Virtual Architecture model (VA) and is exported to
 * SPIRIT for reuse by other tools.
 */

int main(void){

        ostringstream strOut; // Used to build the name of some modules

        //==============================
        // Build the application model
        //==============================

        cout << "Creation␣of␣a␣VA␣MJPEG␣Application␣Model" << endl;

        SerifDesign theDesign("VADesign","MJPEG␣Application␣at␣VA␣level␣(partial)");

        // Create the task definition modules
```

```
    SerifModule& demuxModule  = theDesign.create_module("demux","",SW,PROCESS);
    SerifModule& vldModule    = theDesign.create_module("vld","",SW,PROCESS);
    SerifModule& iqzzModule   = theDesign.create_module("iqzz","",SW,PROCESS);
    SerifModule& idctModule   = theDesign.create_module("idct","",SW,PROCESS);
    SerifModule& libuModule   = theDesign.create_module("libu","",SW,PROCESS);
    SerifModule& convModule   = theDesign.create_module("conv","",SW,PROCESS);

    // Used to group the processes into a 1-processor program
    SerifModule& mjpegModule  = theDesign.create_module("main_mjpeg","Main␣program␣of␣
        MJPEG",SW,PROCESS);

    SerifModule& swStackModule = theDesign.create_module("sw_stack_1","Software␣stack␣
        for␣SW␣organization",SW,ABSTRACT);

    // Add ports to definitions and other information
    mjpegModule.add_port("in_from_tg","unsigned␣char",IN);
    mjpegModule.add_port("out_to_ramdac","unsigned␣char",OUT);
    mjpegModule.add_source("cSource",BASE_DIR_SOURCE+"/src/mjpeg.c");
    mjpegModule.add_source("cSource",BASE_DIR_SOURCE+"/include/mjpeg.h");

    // DEMUX
    demuxModule.add_port("in","unsigned␣char",IN);
    demuxModule.add_port("to_vld","unsigned␣char",OUT);
    demuxModule.add_port("to_libu","unsigned␣char",OUT);
    demuxModule.add_port("to_iqzz","unsigned␣char",OUT);
    demuxModule.add_source("cSource",BASE_DIR_SOURCE+"/src/demux.c");

    // VLD
    vldModule.add_port("from_demux","unsigned␣char", IN);
    vldModule.add_port("to_iqzz","unsigned␣char", OUT);
    vldModule.add_source("cSource",BASE_DIR_SOURCE+"/src/vld.c");

    // IQZZ
    iqzzModule.add_port("from_demux","unsigned␣char",IN);
    iqzzModule.add_port("from_vld","unsigned␣char",IN);
    iqzzModule.add_port("to_idct","unsigned␣char",OUT);
    iqzzModule.add_source("cSource",BASE_DIR_SOURCE+"/src/iqzz.c");

    // IDCT
    idctModule.add_port("from_iqzz","unsigned␣char",IN);
    idctModule.add_port("to_conv","unsigned␣char",OUT);
    idctModule.add_source("cSource",BASE_DIR_SOURCE+"/src/idct.c");

    // CONV
    convModule.add_port("from_idct","unsigned␣char",IN);
    convModule.add_port("to_libu","unsigned␣char",OUT);
    convModule.add_source("cSource",BASE_DIR_SOURCE+"/src/conv.c");

    // LIBU
    libuModule.add_port("from_demux","unsigned␣char",IN);
    libuModule.add_port("from_conv","unsigned␣char",IN);
    libuModule.add_port("to_output","unsigned␣char",OUT);
    libuModule.add_source("cSource",BASE_DIR_SOURCE+"/src/libu.c");

    // Group the task modules on the hybrid process module
    mjpegModule.add_submodule("demux",demuxModule);
    mjpegModule.add_submodule("vld",vldModule);
    mjpegModule.add_submodule("iqzz",iqzzModule);
    mjpegModule.add_submodule("idct",idctModule);
    mjpegModule.add_submodule("conv",convModule);
    mjpegModule.add_submodule("libu",libuModule);

    // Add the nets and connect the modules
    mjpegModule.add_net("in").add_net("out");
mjpegModule.add_net("demux_to_vld").add_net("demux_to_iqzz");
    mjpegModule.add_net("demux_to_libu").add_net("vld_to_iqzz");
    mjpegModule.add_net("iqzz_to_idct");
    mjpegModule.add_net("idct_to_conv");
    mjpegModule.add_net("conv_to_libu");

    // Connect the nets in the mjpeg defintion top module
    mjpegModule.connect_net("in", "in_from_tg").connect_net("in","demux.in");
```

```
mjpegModule.connect_net("out", "libu.to_output").connect_net("out","out_to_ramdac");
    mjpegModule.connect_net("demux_to_vld","demux.to_vld").connect_net("demux_to_vld","
        vld.from_demux");
    mjpegModule.connect_net("demux_to_libu","demux.to_libu").connect_net("demux_to_libu
        ","libu.from_demux");
    mjpegModule.connect_net("demux_to_iqzz","demux.to_iqzz").connect_net("demux_to_iqzz
        ","iqzz.from_demux");
    mjpegModule.connect_net("vld_to_iqzz","vld.to_iqzz").connect_net("vld_to_iqzz","
        iqzz.from_vld");
    mjpegModule.connect_net("iqzz_to_idct","iqzz.to_idct").connect_net("iqzz_to_idct","
        idct.from_iqzz");
    mjpegModule.connect_net("idct_to_conv","idct.to_conv").connect_net("idct_to_conv","
        conv.from_idct");
    mjpegModule.connect_net("conv_to_libu","conv.to_libu").connect_net("conv_to_libu","
        libu.from_conv");

    mjpegModule.create_service("channel", "To␣communicate␣with␣the␣platform", REQUIRED,
         LOGISTICS);

    swStackModule.add_submodule("mjpeg_main",mjpegModule);
    swStackModule.create_service("execution","Request␣an␣execution␣service",REQUIRED,
        LOGISTICS);

    cout << "Display␣of␣top␣simulation␣module␣:␣" << endl;
    cout << mjpegModule << endl;

    //==============================
    // Build the platform model
    //==============================
    SerifModule& tgModule     = theDesign.create_module("tg","",TEST,PROCESS);
    SerifModule& ramdacModule = theDesign.create_module("ramdac","",TEST,PROCESS);
    SerifModule& cpuModule    = theDesign.create_module("cpu_ss_hds","",HYBRID);

    // The top module that contains the platform description
    SerifModule& topPlatformModule = theDesign.create_module("va_platform_top","Top␣
        module␣of␣plaform",TEST,ABSTRACT);

    // Add ports to definitions
    // TG
    tgModule.add_port("out","unsigned␣char",OUT);
    tgModule.add_source("cSource",BASE_DIR_SOURCE+"/src/tg.c");
    // RAMDAC
    ramdacModule.add_port("in","unsigned␣char",IN);
    ramdacModule.add_source("cSource",BASE_DIR_SOURCE+"/src/ramdac.c");
    // CPU SS HDS
    cpuModule.add_port("to_ramdac","unsigned␣char",OUT);
    cpuModule.add_port("from_tg","unsigned␣char",IN);
    // Add the execution service
    cpuModule.create_service("execution","The␣execution␣service",PROVIDED,LOGISTICS);

    // Add module instances to platform top
    topPlatformModule.add_submodule("tg",tgModule);
    topPlatformModule.add_submodule("ramdac",ramdacModule);
    topPlatformModule.add_submodule("cpu_ss_hds",cpuModule);

    // Perform the interconnections
    topPlatformModule.add_net("tg_to_proc").add_net("proc_to_ramdac");
    topPlatformModule.connect_net("tg_to_proc", "tg.out").connect_net("tg_to_proc","
        cpu_ss_hds.from_tg");
    topPlatformModule.connect_net("proc_to_ramdac", "cpu_ss_hds.to_ramdac").connect_net
        ("proc_to_ramdac","ramdac.in");

    // Export the design information to SPIRIT
    cout << "End␣of␣construction,␣will␣now␣export␣everything␣to␣SPIRIT␣IP-XACT" << endl
        ;
    string exampleDir = string(getenv("HOME")) + "/workspace/mjpeg_models/mjpeg-1-SL/
        export2va/xml";

    SpiritTranslator spiritTranslator(theDesign);
    spiritTranslator.serif2spirit(swStackModule, exampleDir);
    spiritTranslator.serif2spirit(topPlatformModule, exampleDir);
```

```
        cout << "End␣of␣SPIRIT␣Export" << endl;
}
```

Bibliographie

[1] AARTS, Emile H. L., RABAEY, Jan M. et WEBER, Werner. *Ambient Intelligence*. Springer, 2005.

[2] AUGÉ, Ivan, PÉTROT, Frédéric, BUCHMANN, Richard, DONNET, Francois, GOMEZ, Pascal et FAURE, Etienne. *Disydent : un environnement pour la conception de systèmes numériques synchrones*. Dans *Premier congrès international de Signaux Circuits et Systèmes (SCS'2004)*. Monastir, Tunisie, mars 2004.

[3] BABAU, Jean Philippe, CHAMPEAU, Joël et GÉRARD, Sébastien (rédacteurs). *From MDD concepts to experiments and illustrations (Models driven engineering for distributed real-time embedded systems)*. Lavoisier, 2006.

[4] BALARIN, Felice, WATANABE, Yosinori, HSIEH, Harry, LAVAGNO, Luciano, PASSERONE, Claudio et SANGIOVANNI-VINCENTELLI, Alberto. *Metropolis : an integrated electronic system design environment*. Computer, 36(4) :45–52, 2003, ISSN 0018-9162.

[5] BASU, Ananda, BOZGA, Marius et SIFAKIS, Joseph. *Modeling Heterogeneous Real-time Components in BIP*. Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on, pages 3–12, 2006.

[6] BEUGNARD, Antoine, JÉZÉQUEL, Jean Marc, PLOUZEAU, Noël et WATKINS, Damien. *Making components contract aware*. IEEE Software, pages 38–45, juin 1999.

[7] BOULET, Pierre, CUCCURU, Arnaud, DEKEYSER, Jean Luc, DUMOULIN, Cédric, MARQUET, Philippe, SAMYN, Mickaël, SIMONE, Robert de, SIEGEL, Gunther et SAUNIER., Thierry. *MDA for SoC design : UML to SystemC experiment*. Dans *International Workshop on UML for SoC Design*, 2004. `http://jerry.c-lab.de/uml-soc/uml-soc04/boulet.pdf`.

[8] BROWN, Alan, JOHNSTON, Simon et KELLY, Kevin. *Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications*. Rational Software Corp., 2002. `http://www-128.ibm.com/developerworks/rational/library/content/03July/2000/2169/2169.pdf`.

[9] CAI, Luka et GAJSKI, Daniel. *Transaction Level Modeling : an Overview*. Dans *Proc. of CODES+ISSS'03*, pages 19–24. ACM, octobre 2003.

[10] CAMPOSANO, Raul et TABET, Raja M.. *Design representation for the synthesis of behavioral VHDL models*. Dans DARRINGER, J. A. et RAMMIG, F. J. (rédacteurs) : *GHDL, Proceedings of the Ninth IFIP Symposium*, pages 49–58, 1989.

[11] CHEVALIER, Jérôme, BENNY, Olivier, RONDONNEAU, Mathieu, BOIS, Guy, ABOULHAMID, El Mostapha et BOYER, François Raymond. *SPACE : A Hardware/Software SystemC modeling platform including an RTOS*. Dans *Proc. Forum on Design Languages (FDL03)*, pages 704–715, Frankfurt, septembre 2003.

[12] CHIODO, M., GIUSTO, P., JURECSKA, A., HSIEH, H.C., SANGIOVANNI-VINCENTELLI, A. et LAVAGNO, L.. *Hardware-software codesign of embedded systems*. Micro, IEEE, 14(4) :26–36, Aug 1994, ISSN 0272-1732.

[13] CHUREAU, Alexandre. *Serif User Guide*. rapport technique, Grenoble INP - Laboratoire TIMA, 2008.

[14] CHUREAU, Alexandre, SAVARIA, Yvon et ABOULHAMID, El Mostapha. *The Role of Model-Level Transactors and UML in Functional Prototyping of Systems-on-Chip : A Software-Radio Application*. Dans *Proceedings of Design, Automation and Test in Europe, 2005*, pages 698–703, mars 2005.

[15] CYR, Geneviève, BOIS, Guy et ABOULHAMID, El Mostapha. *Generation of processor interface for SoC using standard communication protocol*. Computers and Digital Techniques, IEE Proceedings, 151(5) :367–376, 2004, ISSN 1350-2387.

[16] DAVEAU, J. M., MARCHIORO, G.F., BEN-ISMAIL, T. et JERRAYA, A.A.. *Protocol selection and interface generation for HW-SW codesign*. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 5(1) :136–144, 1997, ISSN 1063-8210.

[17] ENDREI, Mark, ANG, Jenny, ARSANJANI, Ali, CHUA, Sook, COMTE, Philippe, KROGDAHL, Pal, LUO, Min et NEWLING, Tony. *Patterns : Service-Oriented Architecture and Web Services*. rapport technique, IBM, 2004.

[18] FERGUSON, D. F. et STOCKTON, M. L.. *Service-oriented architecture : Programming model and product architecture*. IBM SYSTEMS JOURNAL, VOL 44(4) :753–780, 2005. `http://www.research.ibm.com/journal/sj/444/ferguson.pdf`.

[19] GAUTHIER, Ludovic. *"Génération de système d'exploitation pour le ciblage de logiciel multitâche sur des architectures multiprocesseurs hétérogènes dans le cadre des systèmes embarqués spécifiques"*. Thèse de doctorat, Institut national polytechnique de Grenoble, 2001.

[20] GERIN, Patrice, GUÉRIN, Xavier et PÉTROT, Frederic. *Efficient Implementation of Native Software Simulation for MPSoC*. Dans *Design, Automation and Test in Europe, 2008. DATE '08*, pages 676–681, mars 2008.

[21] GERIN, Patrice, SHEN, Hao, CHUREAU, Alexandre, BOUCHHIMA, Aimen et JERRAYA, Ahmed. *Flexible and Executable Hardware/Software Interface Modeling For Multiprocessor SoC Design Using SystemC*. Dans *Proceedings of the 12th Asia and South Pacific Design Automation Conference*, pages 390–395, 2007.

[22] GHEORGHE, L., BOUCHHIMA, F., NICOLESCU, G. et BOUCHENEB, H.. *Semantics for Model-Based Validation of Continuous/Discrete Systems*. Design, Automation and Test in Europe, DATE, pages 498–503, mars 2008.

[23] GRASSET, A.. *Synthèse des interfaces de communication dans la conception des systèmes monopuces : de la spécification à la génération automatique*. Thèse de doctorat, Institut national polytechnique de Grenoble, 2005.

[24] GREENFIELD, Adam. *Everyware : La révolution de l'ubimédia*. FYP éditions, 2007.

[25] HAREL, David. *Statecharts : A Visual Formalism for Complex Systems*. Science of Computer Programming, 8(3) :231–274, juin 1987.

[26] HAREL, David et NAAMAD, Amnon. *The STATEMATE semantics of statecharts*. ACM Trans. Softw. Eng. Methodol., 5(4) :293–333, 1996, ISSN 1049-331X.

[27] HELM, R., HOLLAND, I. M. et GANGOPADHYAY, D.. *Contracts : Specifying Behavioral Compositions in Object-Oriented Systems*. Dans *Proc.of the OOPSLA/ECOOP-90*, pages 169–180, Ottawa, Canada, 1990.

[28] HUANG, Kai, HAN, Sang il, POPOVICI, Katalin, BRISOLARA, Lisane, GUÉRIN, Xavier, LI, Lei, YAN, Xiaolang, CHAE, Soo Ik, CARRO, Luigi et JERRAYA, Ahmed Amine. *Simulink-Based MPSoC Design Flow : Case Study of Motion-JPEG and H.264*. Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE, pages 39–42, juin 2007, ISSN 0738-100X.

[29] IEEE STANDARDS ASSOCIATION. *IEEE POSIXő Certification Authority*. [En ligne], 2006. `http://standards.ieee.org/regauth/posix/`.

[30] Internet Engineering Task Force. *RFC-1122 : Requirements for Internet Hosts – Communication Layers*, 1989. `http://tools.ietf.org/html/rfc1122`.

[31] KNAPP, D. W. et PARKER, A. C.. *A Unified Representation for Design Information*. Dans *Proceedings of the 7th International Symposium on Computer Hardware Description Languages and their Applications*, pages 337–353, 1985.

[32] LUNDELL, Björn, LINGS, Brian, PERSSON, Anna et MATTSSON, Anders. *UML Model Interchange in Heterogeneous Tool Environments : An Analysis of Adoptions of XMI 2*. Dans *MoDELS*, pages 619–630, 2006.

[33] LYONNARD, Damien. *Approche d'assemblage systématique d'éléments d'interface pour la génération d'architecture multiprocesseur*. Thèse de doctorat, Institut national polytechnique de Grenoble, 2003.

[34] MAGARSHACK, P.. *Improving SoC design quality through a reproducible design flow*. Design & Test of Computers, IEEE, 19(1) :76–83, 2002.

[35] MENTOR GRAPHICS CORPORATION. *Catapult-C Synthesis*. Logiciel, 2005. Wilsonville, Oregon.

[36] MENTOR GRAPHICS CORPORATION. *Seamless*. Logiciel, 2005. `http://www.mentor.com/seamless`, Wilsonville, Oregon.

[37] NICOLESCU, Gabriela. *Spécification et validation des systèmes hétérogènes embarqués*. Thèse de doctorat, Institut national polytechnique de Grenoble, 2002.

[38] OBJECT MANAGEMENT GROUP. *Page d'accueil*. `http://www.omg.org`.

[39] OBJECT MANAGEMENT GROUP. *UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE)*, 2007. `http://www.omg.org/cgi-bin/doc?ptc/2007-08-04`, version 1.0Beta1.

[40] OSCI. *Draft Standard SystemC Language Reference Manual*, 2005.

[41] POPOVICI, K., GUÉRIN, X., BRISOLARA, L. et JERRAYA, A.. *Mixed Hardware Software Multilevel Modeling and Simulation for Multithreaded Heterogeneous MP-SoC*. VLSI Design, Automation and Test, 2007. VLSI-DAT 2007. International Symposium on, pages 1–4, avril 2007.

[42] ROWSON, James A. et SANGIOVANNI-VINCENTELLI, Alberto. *Interface-based Design*. Dans *Proceedings of the 34th Design Automation Conference*, pages 178–183. ACM Press, 1997, ISBN 0-89791-920-3.

[43] SCHEMA WORKING GROUP. *IP-XACT v1.4 : A specification for XML metadata and tool interfaces*. SPIRIT Consortium, mars 2008. `http://www.spiritconsortium.com`.

[44] SCHUSSEL, George. *Client/Server : Past, Present and Future*. [En ligne ; page disponible le 21-août-2007], 1997. `http://www.dciexpo.com/geos/dbsejava.htm`.

[45] SHAPES PROJECT - SCALABLE SOFTWARE HARDWARE COMPUTING ARCHITECTURE PLATFORM FOR EMBEDDED SYSTEMS. *Page d'accueil*. [En ligne], 2008. `http://shapes.atmelroma.it/twiki/bin/view/ShapesPublic`.

[46] SPRINT PROJECT. *Open SoC Design Platform for Reuse and Integration of IPs*, 2007. `http://www.sprint-project.net`.

[47] STOY, E. et PENG, Z.. *Hardware / software co-simulation using a unified design representation*. Dans *Proc. of the 6th Swedish Workshop on Computer System Architecture*, pages 7–9, 1995.

[48] TELELOGIC. *Statemate*. Logiciel, 2008. `http://www.telelogic.fr/products/statemate/index.cfm`.

[49] THE MATHWORKS. *Simulink 7.0*. Logiciel, 2005. Natick, Mass.

[50] THE MATHWORKS. *Stateflow Documentation Homepage*. Wold Wide Web, 2008. `http://www.mathworks.com/access/helpdesk/help/toolbox/stateflow/index.html`.

[51] THIELE, Lothar, BACIVAROV, Iuliana, HAID, Wolfgang et HUANG, Kai. *Mapping Applications to Tiled Multiprocessor Embedded Systems*. Application of Concurrency to System Design, 2007. ACSD 2007. Seventh International Conference on, pages 29–40, juillet 2007, ISSN 1550-4808.

[52] TURLEY, J.. *"Survey says : software tools more important than chips"*. Embedded Systems Design Journal, novembre 2005. http://www.embedded.com/showArticle.jhtml?articleID=160700620.

[53] Union internationale des télécommunications. *Technologies de l'information - traitement réparti ouvert - modèle de référence : fondements*, 1997.

[54] VINCENTELLI, Alberto Sangiovanni. *Defining platform-based design*. EEDesign of EETimes, février 2002. http://www.gigascale.org/pubs/141.html.

[55] WIKIPÉDIA. *Web 2.0 — Wikipédia, l'encyclopédie libre*. [En ligne ; page disponible le 22-août-2007], 2007. http://fr.wikipedia.org/w/index.php?title=Web_2.0&oldid=19915767.

[56] WIKIPEDIA. *Integrated Services Digital Network — Wikipedia, The Free Encyclopedia*. [En ligne ; page disponible le 21-août-2007], 2007. http://en.wikipedia.org/w/index.php?title=Integrated_Services_Digital_Network&oldid=151996949.

[57] WIKIPEDIA. *Universally Unique Identifier — Wikipedia, The Free Encyclopedia*. [En ligne ; page disponible le 2-septembre-2007], 2007. http://en.wikipedia.org/w/index.php?title=Universally_Unique_Identifier&oldid=152966312.

[58] WIKIPEDIA. *Ubiquitous computing — Wikipedia, The Free Encyclopedia*. [En ligne ; page disponible le 1-juin-2008], 2008. http://en.wikipedia.org/w/index.php?title=Ubiquitous_computing&oldid=215846362.

[59] WOLF, P. van der, KOCK, E. de, HENRIKSSON, T., KRUIJTZER, W. et ESSINK, G.. *Design and programming of embedded multiprocessors : an interface-centric approach*. Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004. International Conference on, pages 206–217, septembre 2004.

[60] YANG, Guang, CHEN, Xi, BALARIN, Felice, HSIEH, Harry et SANGIOVANNI-VINCENTELLI, Alberto. *Communication and co-simulation infrastructure for heterogeneous system integration*. Dans *Proceedings of DATE*, 2006.

[61] ZIEGENBEIN, D., RICHTER, K., ERNST, R., THIELE, L. et TEICH, J.. *SPI - a system model for heterogeneously specified embedded systems*. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 10(4) :379–389, août 2002, ISSN 1063-8210.

[62] ZIMMERMANN, H.. *OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection*. Communications, IEEE Transactions on [legacy, pre - 1988], 28(4) :425–432, 1980, ISSN 0096-2244.

[63] ZITTERBART, M. et AL. *A Model for Flexible High-Performance Communication Subsystems*. IEEE Journal on selected areas in communication, 11 :507–518, mai 1993.

Acronymes

A

API Application Programming Interface
Interface de programmation qui définit la manière dont un composant informatique peut accéder à un autre.

ASIC Application-Specific Integrated Circuit
Circuit électronique dédié à une seule application.

C

CABA Cycle accurate / Bit Accurate
Niveau d'abstraction très près de l'implémentation où les communications peuvent être observées à une granularité de l'ordre du cycle.

CORBA Common Object Request Broker Architecture
Architecture logicielle qui permet à des objets écrits dans des langages différents et déployés sur des processeurs différents de communiquer.

E

ECC Error-Correction Coding
Encodage des données transmises ou stockées qui permet la détection et la correction d'erreurs.

F

FIFO First In First Out
Type de canal où les éléments sont lus dans l'ordre d'écriture (premier écrit premier lu).

G

GSM Global System for Mobile Communication
Une norme utilisée pour les communications mobiles.

GUI Graphical User Interface
Interface utilisateur graphique, popularisée par les systèmes Mac et Windows.

I

ISO International Standard Organization
Organisme qui a proposé le modèle de base OSI.

ISS Instruction Set Simulator Logiciel qui simule l'exécution de code binaire d'un processeur embarqué, avec un degré variable de précision sur les accès matériels.

M

MEMS Micro Electro Mechanical System
Microsystème électromécanique.

MPSoC Multiprocessor System-on-Chip
Système multiprocesseur monopuce.

O

OSI Open Systems Interconnection
Modèle à sept couches pour la représentation de protocoles réseau, décrit dans la norme ISO 7498.

P

PC Personal Computer
Désigne l'ordinateur de bureau conventionnel.

Q

QoS Quality of Service
Qualité de service.

R

RTL Register-Transfer Level
Description d'un système intégré en terme de transfert de registre, c'est à dire très près de l'implémentation finale.

S

SAP Service Access Point
Une connexion logique représentant un point d'accès à un service fourni par une couche du modèle de référence OSI.

SDG Service Dependency Graph
Graphe de dépendance de service, utilisé pour représenter les relations entre les composants de bibliothèque et les services qu'ils fournissent et ceux qu'ils requièrent.

SDL Specification and Description Language
Langage de modélisation utilisé pour la description de protocoles de télécommunication et plus généralement d'applications temps réel.

SLS System-Level Synthesis
Synthèse haut-niveau pour systèmes microélectroniques. Nom d'un groupe de recherche au sein du TIMA qui effectue de la recherche sur ce type de synthèse.

SMP Symmetric Multiprocessor
Architecture où plusieurs processeurs partagent un même espace mémoire et où les tâches peuvent migrer afin d'équilibrer la charge.

SOA ou SOD Service Oriented Architecture / Design.
Une façon de concevoir des systèmes répartis, basée sur l'offre et la demande de services décrits en format XML.

SOAP Simple Object Access Protocol
Un protocole de transmission de message utilisé dans le cadre des services Web et décrit en XML.

SoC System-on-Chip
Système électronique complet intégré sur une puce unique.

SPI System Property Interval
Représentation intermédiaire formelle basée sur des processus communicants.

T

TA Transaction Accurate
Niveau d'abstraction où les communications sont abstraites sous forme de transactions.

TIMA Techniques de l'informatique et de la microélectronique pour l'architecture des systèmes intégrés
Laboratoire de recherche de l'Université de Grenoble, spécialisé en micro-électronique.

V

VA Virtual Architecture
: Niveau d'abstraction « architecture virtuelle » où le modèle de l'application définit implicitement l'architecture du système.

VLSI Very Large Scale Integration
: Désigne la catégorie de circuits intégrés à très grande échelle, ce qui représente la majorité des circuits intégrés aujourd'hui.

W

WSDL Web Service Description Language
: Format XML pour la description d'interfaces de services Web..

X

XML Extended Markup Language
: Format texte normalisé pour la description d'informations diverses.

Résumé

Les architectures multiprocesseurs de systèmes sur puce permettent de réaliser un nombre croissant de fonctions en logiciel, ce qui multiplie le nombre d'interfaces entre le logiciel et le matériel. Cette interface est représentée de différentes façons au sein des modèles, selon leur niveau d'abstraction : à haut niveau, un canal abstrait est utilisé ; plus près de l'implémentation, plusieurs composants d'adaptation et de communication composent l'interface. La conception assistée des systèmes multiprocesseurs repose donc sur la maîtrise de l'interface logiciel-matériel à plusieurs niveaux d'abstraction. Dans cette thèse, le concept de service est utilisé pour abstraire les caractéristiques de communication et de performance des interfaces. Une structure de données permet de capturer ces caractéristiques et de développer des outils d'analyse et de génération d'interfaces. Une étude de cas illustre l'exploration d'architecture par la génération de prototypes virtuels en SystemC.

Title

Definition of a Service-Based Intermediate Representation for Virtual Prototyping of Systems-on-Chip

Abstract

Multiprocessor system-on-chip architectures allow implementing more functions in software, which increases the number of interfaces between software and hardware. These interfaces have different representations within the models, depending on the level of abstraction : at high level, an abstract channel is used, whereas closer to the implementation, many adaptation and communication components are required. Mastering the software-hardware interface at many levels of abstraction is thus mandatory to master design complexity. In this thesis, the concept of service is used to abstract the communication and performance features of interfaces. A data structure allows capturing these features and developing tools for interface analysis and generation. A case study illustrates architecture exploration through the generation of SystemC virtual prototypes.

Mots clés

Conception de systèmes-sur-puce, systèmes multiprocesseurs, interface logiciel-matériel, architecture orientée service, prototypage virtuel

Keywords

System-on-chip design, multiprocessor systems, hardware-software interface, service-oriented architecture, virtual prototyping

Intitulé du laboratoire

Laboratoire TIMA
Techniques de l'informatique et de la microélectronique pour l'architecture des systèmes intégrés
46, avenue Félix Viallet
38031 Grenoble Cedex
France

ISBN 978-2-84813-128-3