Génération Automatique de Modèles de Simulation pour la Validation de Systèmes Hétérogènes Embarqués

A. Sarmento

UNIVERSITE JOSEPH FOURIER- GRENOBLE I
SCIENCES, TECHNOLOGIE & MEDECINE

T H E S E

pour obtenir le grade de

DOCTEUR DE l'UNIVERSITE JOSEPH FOURIER

Spécialité : Microélectronique

préparée au laboratoire **TIMA** dans le cadre de
l'Ecole Doctorale d'« Electronique, Electrotechnique, Automatique, Télécommunications, Signal »

par

Adriano Augusto DE MORAES SARMENTO

le 28 octobre 2005

Titre :

Génération automatique de modèles de simulation pour la validation de systèmes hétérogènes embarqués

---

Directeur de Thèse : Ahmed Amine Jerraya
Codirecteur : Wander Oliveira Cesario

JURY

M. Frédéric Pétrot	, Président
M. El Mostapha Aboulhamid	, Rapporteur
M. Jean-Paul Calvez	, Rapporteur
M. Ahmed Amine Jerraya	, Directeur
M. Wander Oliveira Cesário	, Codirecteur
M. Philippe Kajfasz	, Examinateur

Remerciements

Cette thèse est dédiée aux nombreuses personnes qui m'ont aidé pendant le long séjour que j'ai passé en France.
D'abord je voudrais remercier Andrea et José pour avoir parcouru ce chemin avec moi. Leur amour, tendresse et patience ont été essentiels pour que je puisse finir cette thèse, surtout pendant les moments plus difficiles. Je vous aime.
Je remercie mes parents Teresinha et Osvaldo pour leur amour et soutien. Grâce à eux j'ai été capable de faire une thèse. Leurs conseils, leurs vies et surtout leur soutien m'ont beaucoup inspiré.
Je remercie également mon frère Carlos et ma belle sœur Verônica pour leur amitié et soutien, ainsi que le cadeau le plus important qu'ils m'ont offert : ma très belle nièce Gabi.
Je tiens à remercier mon oncle José Costa pour m'avoir encouragé pendant toute ma vie et pour m'avait traité comme si j'avais été son fils.
Je n'ai peux pas oublier de remercier ma belle mère Lilia qui nous a toujours aidé et m'a souvent gâté en me préparant la délicieuse mousse aux fruits de la passion. Mes remerciements vont également à mon beau père Teo, ma belle sœur Carol et à mon très beau petit neveu Luca.
J'adresse mes remerciements au couple João et Mariana pour leur amitié et pour tout ce qu'ils ont fait pour que la soutenance puisse avoir lieu. Si je suis arrivé à soutenir ma thèse, c'est grâce à leur gentillesse et patience. Je profite pour m'excuser des nombreux problèmes auxquels ils ont dû faire face pour déposer mon dossier. Je souhaite à tous les deux bon courage et bonne continuation.
Merci à ma chère Sonja ! Elle a toujours été présente pour m'aider et pour soutenir mon moral. Sa gaieté va beaucoup me manquer.
Un grand merci à Patricia Ouhamou et Fred Hunsinger pour les très bons moments qu'on a vécu ensemble dans le bureau 422. Je vous remercie aussi pour avoir corrigé les nombreuses fautes de français de ma thèse. De plus vous m'avez beaucoup appris sur la culture française.
Un merci spécial à ma chère Lobna. On est passé par les mêmes problèmes, surtout la définition de nos sujets de thèse, mais on a bien survécu et on a passé des bons moments. Je suis très content que tu vas bientôt soutenir ta thèse. Je te souhaite bonne chance et bonne continuation
Je remercie également les autres thèsards du groupe SLS : Arnaud, Aimen, Benaoumeur, Ivan, Lorenzo Pieralisi, Marcio, Marius, Wassim, Youssef et Youngchul. Nos discussions m'ont beaucoup aidé à mieux comprendre le monde très complexe des systèmes embarqués. Bonne courage pour la suite.
Un grand remerciement à Damien, Gabriela et Arif pour leur gentillesse lorsque je suis arrivé dans le groupe SLS.
Je ne peux pas oublier de dire merci à tout les « stars » du foot : Aimen, Benaoumeur, Fred, Wassim, Youssef, Ferid, Kamel et Nacer.
Je voudrais remercier Frédéric Rousseau pour m'aider au début de ma thèse lorsqu'on a partagé le même bureau et surtout pour ses conseils tout au long de mon séjour ici qui ont beaucoup enrichi cette thèse.
Un merci spécial à Madame Comtat qui m'a très bien accueilli quand je suis arrivé à Grenoble. Je remercie également mes amis de l'Alliance Française : Jochen, Erika, Lina, John, Roberta et Lorenzo. J'espère qu'on pourra se revoir bientôt quelque part dans le monde. Je remercie énormément Geneviève, Hubert et leur famille pour leur amitié et pour tout ce qu'ils ont fait pour nous. On vous attend à Recife. Mes remerciements vont aussi aux très bons amis Daniel, Oana et Andrei. Il n'y a que de bons moments dans ces trois années qu'on a vécu ensemble dans le même bâtiment Merci aux amis brésiliens Cristiano et Patricia. Bon retour au Brésil et j'espère vous rencontrer bientôt. Enfin un grand merci à Carmen, Fernando et João Pedro et tout les amis qui sont au Brésil.
Je tiens à remercier tous les membres du jury pour leurs remarques qui ont beaucoup contribué pour améliorer cette thèse. Je remercie M. Mostapha Aboulhamid et M. Jean-Paul Calvez de leur gentillesse d'avoir accepté d'être rapporteurs de cette thèse.
Merci à Wander Cesário pour toutes les contributions techniques qu'il a apporté à ce travail. Je te souhaite bonne chance dans ton nouveau poste.
Je voudrais remercier M. Ahmed Amine Jerraya, directeur de recherche au CNRS, de m'avoir donné l'opportunité de faire une thèse ici dans le groupe SLS. Puisse ce travail être à l'hauteur de son encadrement. J'espère que cette thèse apporte quelque chose d'important aux travaux du groupe.

Table des Matières

Liste des Figures

Chapitre 1 : Problématique de la Conception de Systèmes Hétérogènes Embarqués

1.1 Contexte: conception de systèmes hétérogènes embarqués

Cette thèse s'inscrit dans le contexte de la conception des systèmes hétérogènes embarqués monopuces. Ce type de système, est adapté pour implémenter une application spécifique en respectant entre autres des contraints de performance, de consommation d'énergie, et de surface.

Les systèmes embarqués sont, généralement, composés de parties logicielles et matérielles. La partie logicielle concerne le logiciel qui s'exécute sur un processeur, alors que la partie matérielle consiste en un composant matériel qui implémente une fonctionnalité spécifique (ASIC). Dans les années 90, ces systèmes présentaient une architecture relativement simple, généralement un processeur et quelques composants matériels communicant via un unique protocole de communication. Actuellement, ils sont beaucoup plus complexes. Ils sont, généralement, composés de plusieurs processeurs et composants matériels, communicant via des schémas de communication très sophistiqués. De plus, pour respecter les contraintes de l'application, ces systèmes sont implémentés sur une seule puce, raison pour laquelle ils sont souvent appelés SoC (venant de l'anglais System-on-Chip).

Les systèmes embarqués monopuces sont largement utilisés dans différents secteurs d'activité tels que : la télécommunication, l'aérospatiale, les jeux électroniques, l'automobile, etc. La conception de tels systèmes est complexe en raison de l'hétérogénéité des composants et des schémas de communication utilisés. Cette hétérogénéité rend difficile aussi la validation de tels systèmes. En fait, la validation représente, actuellement, 50 à 80% du temps de conception d'un système hétérogène embarqué monopuce.

1.2 Difficultés de la conception de systèmes hétérogènes embarqués

La Figure 1-1 est une représentation conceptuelle d'un système hétérogène embarqué. Ce système peut être composé de plusieurs types de processeurs (microcontrôleurs et DSPs par exemple), plusieurs composants matériels (des mémoires, des IPs, etc.) qui communiquent par divers protocoles de communication (FIFO, handshake, etc.) et utilisent différentes topologies de communication (bus, point à point, multipoint, etc.). En plus, les différentes parties de ces systèmes peuvent être spécifiées en différents langages de spécification et niveaux d'abstraction. Cette diversité de composants et de schémas de communication est nécessaire la plupart du temps pour respecter les contraintes de l'application et le processus de

conception. Cependant, cela va poser deux problèmes majeurs lors de la conception de systèmes hétérogènes embarqués. Le premier est relatif à l'intégration de tous ces composants pour qu'ils puissent travailler ensemble pour implémenter l'application cible. Le deuxième concerne la validation du système, puisque la génération d'un modèle de simulation (dans le cas où nous utilisons la simulation comme méthode de validation) qui utilise des composants hétérogènes est un processus complexe.

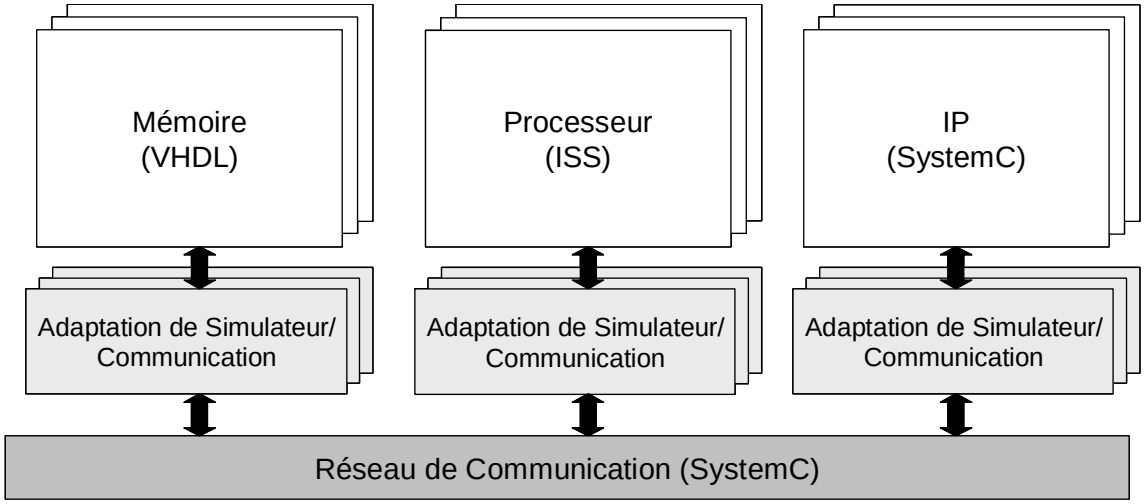


Figure 1-1 Représentation d'un système hétérogène embarqué

1.2.1 Intégration de systèmes hétérogènes embarqués

L'intégration des composants passe par le ciblage du logiciel sur les processeurs utilisés ainsi que l'adaptation de la communication entre tous les composants matériels. Cette intégration concerne aussi les différents modèles de simulation des composants utilisés lors de la validation. La Figure 1-1 montre ce type d'intégration. Les composants qui adaptent la communication peuvent fournir plusieurs fonctionnalités telles que : adaptation de niveau d'abstraction, synchronisation, adaptation de types de donnés, adaptation de protocole de communication, etc. L'implémentation de cette adaptation est complexe car elle met en œuvre plusieurs connaissances et détails pouvant appartenir à des domaines différents. En plus, la spécification du système hétérogène embarqué peut changer plusieurs fois pendant sa conception. Cela veut dire que si l'adaptation n'est pas faite d'une façon flexible et qui facilite la réutilisation des composants, l'effort et le temps d'intégration seront significatifs.

1.2.2 Validation de systèmes hétérogènes embarqués

La validation consomme une grande partie du temps de conception dans le cas des systèmes hétérogènes embarqués. Autrement dit, elle représente le goulot d'étranglement du

processus de conception. La validation doit être faite dans toutes les étapes de la conception, prenant en compte l'hétérogénéité des composants selon l'étape de conception. Initialement, par exemple, les composants peuvent être décrits par différents langages de spécification et à différents niveaux d'abstraction.

Pour effectuer la validation, un modèle de simulation global du système doit être construit. Ce modèle peut utiliser plusieurs simulateurs différents et doit accommoder les différents protocoles de communication, niveaux d'abstraction, langages de spécification et types de données. Dans le cas où plusieurs simulateurs sont mis en œuvre par la simulation globale on parle de cosimulation. Cela implique que des interfaces de cosimulation doivent être implémentées aussi pour adapter les composants et éventuellement les simulateurs. Ces interfaces font partie aussi du modèle de simulation. La construction de tels modèles à chaque étape de conception est un processus long et fastidieux. En conséquence, souvent ce modèle n'est construit que lors des étapes très avancées de la conception, ce qui occasionne des coûts et des délais de conception significatifs.

1.3 Objectif: Génération de modèles de simulation pour la validation de systèmes hétérogènes embarqués

L'objectif principal de ce travail est de réduire le temps de validation de systèmes hétérogènes embarqués par la génération automatique de modèles de simulation de tels systèmes. En réduisant le temps de validation, le temps d'intégration sera également diminué en conséquence. Cette génération est faite partant d'une spécification du système qui reflète les besoins d'adaptation et d'une bibliothèque de composants de base qui sont assemblés pour former le modèle de simulation. La Figure 1-2 présente l'approche proposé. Les modules du système sont spécifiés en abstrayant leurs interfaces de communication. Les interfaces de communication sont décrites en précisant leurs besoins d'adaptation, sans aucun détail sur leurs réalisations. Partant de cette spécification, un modèle de simulation est généré qui intègre tous les modules en adaptant les différents langages de spécification (représenté dans la Figure 1-2 pour l'adaptation de simulateurs), et différents aspects de la communication (représenté dans la Figure 1-2 pour l'adaptation de communication) telles que : les niveaux d'abstraction, les protocoles de communication et les types de données. Les différents types d'adaptation sont réalisés en utilisant une bibliothèque de composants de base. La génération est accomplie à travers l'analyse de la spécification, l'assemblage de composantes de base nécessaires à la réalisation des adaptations et la production de code exécutable.

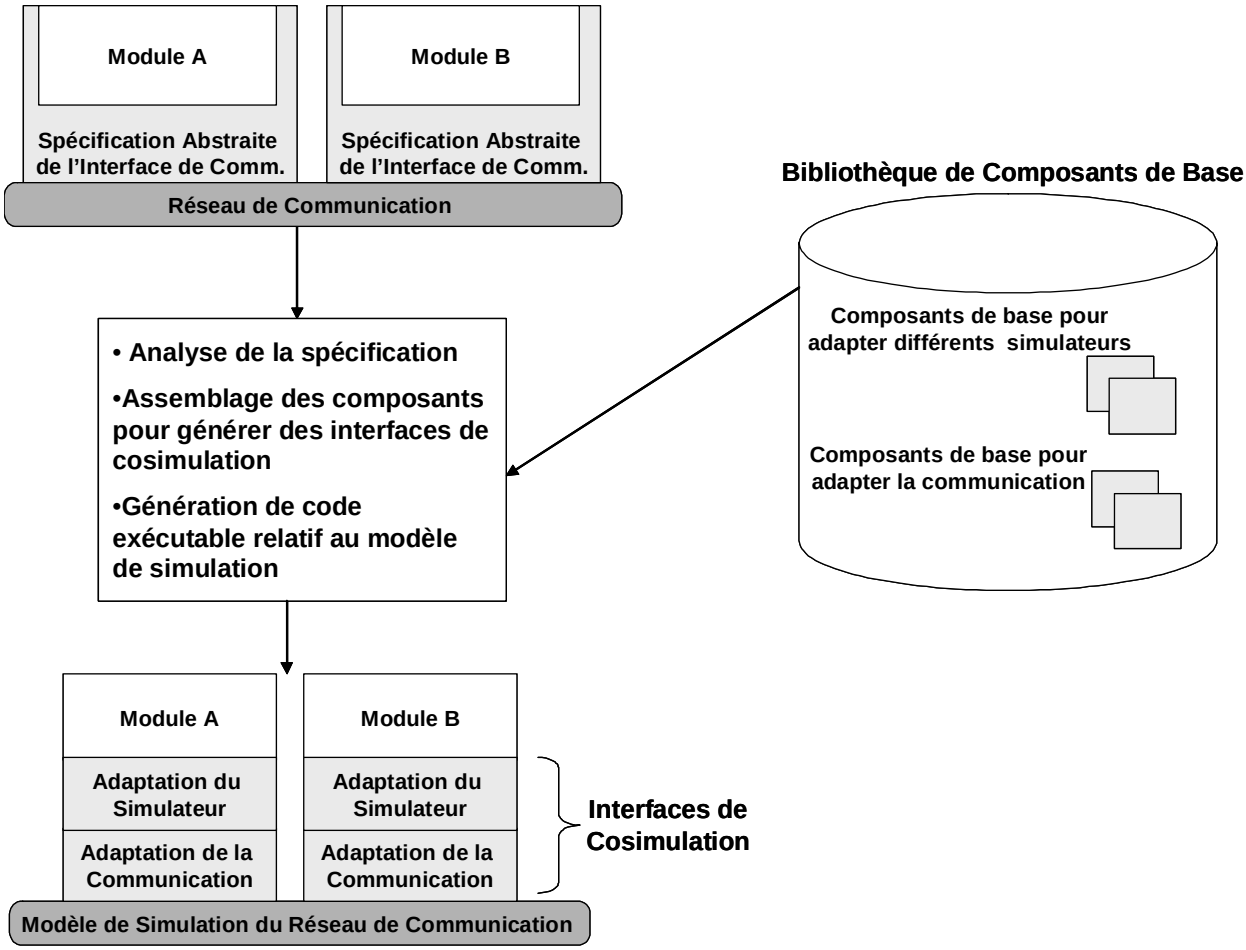


Figure 1-2 Génération de Modèles de Simulation Partant d'une Spécification Abstraite des Interfaces de Communication du Système Hétérogène Embarqué

1.4 Contributions

Ce travail apporte deux contributions : (1) la définition d'un modèle d'adaptateur de communication pour la cosimulation des systèmes hétérogènes embarqués ; (2) la proposition et l'implémentation d'un flot de génération automatique de modèles de simulation pour les systèmes hétérogènes embarqués.

1.4.1 Modèle d'adaptateur de communication pour la cosimulation des systèmes hétérogènes embarqués

Le but d'un adaptateur de communication est de mettre en correspondance des composants ayant différents protocoles de communication et niveaux d'abstraction. L'adaptateur de communication facilite, donc, l'intégration des composants hétérogènes dans la cosimulation. La réalisation de tels adaptateurs manuellement est un travail fastidieux et, selon l'application, complexe. La définition d'un modèle qui puisse représenter ces

adaptateurs en vue de les générer automatiquement devient un point clé pour l'accélération du processus d'intégration de composants hétérogènes. Un tel modèle doit être assez générique et flexible pour permettre l'intégration des composants ayant n'importe quel protocole/niveau d'abstraction, et faciliter la réalisation des adaptateurs performants. En plus, le modèle doit favoriser la réutilisation de composants de base de l'adaptateur pour réduire le temps de conception de celui-là. Les modèles proposés dans la littérature présentent un domaine d'application restreint : ils imposent des restrictions par rapport aux protocoles/niveaux d'abstraction en réduisant ainsi la flexibilité ; ou bien les adaptateurs modélisés possèdent une architecture fixe, ce qui pénalise la flexibilité, la réutilisation et la performance car les composants de base sont trop gros (cela augmente l'effort du concepteur pour les réaliser) et parfois incluent des fonctionnalités qui ne sont pas nécessaires.

Nous proposons dans ce travail, un modèle d'adaptateur de communication basé sur les services. Un adaptateur de communication est représenté par un graphe (appelé graphe de dépendance de services) contenant des composants de base qui peuvent fournir et/ou requérir des services. Ces composants sont liés par la relation de dépendance entre les services requis et fournis par chaque composant. Ce modèle facilite : (1) l'implémentation des adaptateurs flexibles puisque le modèle n'impose aucune contrainte par rapport aux protocoles/niveaux d'abstraction ; (2) la réutilisation des composants de base car ceux-ci sont représentés par les services nécessaires pour l'adaptation ; (3) et implicitement la construction des adaptateurs performants parce que ceux-ci ne contiendront que les services nécessaires pour sa fonction d'adaptation. Ce modèle a été appliqué pour valider deux systèmes hétérogènes – le modem VDSL et l'encodeur MPEG-4, et les résultats ont montré un gain de flexibilité et de réutilisation de composants de base apporté par le modèle. Cela a représenté une réduction importante du temps de validation de ces deux systèmes. Les résultats ne montrent pas le gain de performance, mais cela est implicite parce que les adaptateurs ne contient que les services nécessaires pour leur fonction d'adaptation.

1.4.2 Flot de génération automatique de modèles de simulation pour la validation de systèmes hétérogènes embarqués

La génération automatique des modèles de simulation pour des systèmes hétérogènes accélère le temps de validation. La mécanique de génération doit être capable de générer rapidement des modèles de simulation pour chaque étape du flot de conception des systèmes. En plus, le flot de génération doit être assez flexible pour qu'un changement de langage de spécification

ou de langage de code exécutable n'implique que la modification de quelques parties du flot. Plusieurs travaux portent aujourd'hui sur la génération automatique des modèles de simulation pour la validation des systèmes hétérogènes embarqués. Pourtant, les solutions existantes soit sont bien adaptées pour certaines étapes du flot de conception, soit ne sont pas assez flexibles.

Nous proposons et implémentons un flot de génération automatique de modèles de simulation pour les systèmes hétérogènes embarqués. Partant d'une spécification du système et d'une bibliothèque de composants de base, nous générons le code exécutable qui implémente le modèle de simulation. Ce flot est divisé en trois parties distinctes : (1) l'analyse de l'architecture où nous analysons les besoins d'adaptation entre les composants ; (2) la génération du modèle de simulation dans une représentation intermédiaire ; (3) la génération de code exécutable relatif au modèle de simulation du système. Le flot est flexible, car le changement de langage de spécification initiale ou de langage du code exécutable n'implique que la modification de quelques outils qui font partie du flot. Ce flot a été appliqué pour les deux expérimentations et a généré rapidement des modèles de simulation pour chaque étape du flot de conception. La génération automatique de modèles de simulation proposée dans ce travail a représenté une réduction significative du temps de validation de ces deux systèmes. En plus le flot proposé a été intégré dans le flot de conception des systèmes développé par le groupe SLS, pour améliorer la solution de génération automatique de modèles de simulation préexistant.

1.5 Plan du mémoire

Ce document est structuré comme suit. Le chapitre 2 présente les concepts de base de systèmes hétérogènes embarqués, ainsi qu'une étude sur leur validation. Le chapitre 3 décrit le modèle d'adaptateur de communication pour la cosimulation des systèmes hétérogènes proposé dans ce travail. Le chapitre 4 présente le flot de génération des modèles de simulation. Le chapitre 5 présente les résultats obtenus des expérimentations que nous avons effectué pour valider nos contributions. Le chapitre 6 donne nos conclusions et les perspectives de ce travail.

Chapitre 2 : Validation de systèmes hétérogènes embarqués

2.1 Introduction

Ce chapitre explique la complexité du processus de validation lors de la conception de systèmes. Pour cela, nous commençons par aborder les caractéristiques spécifiques de systèmes hétérogènes embarqués. Nous verrons comment ces caractéristiques rendent difficile la validation et par conséquence la conception de tels systèmes. Ensuite, plusieurs méthodes de validation seront discutées. Enfin, la dernière partie de ce chapitre sera consacrée à la méthode de validation par cosimulation, qui est le sujet auquel nous apportons nos contributions dans ce travail.

2.2 Systèmes hétérogènes embarqués

Les systèmes embarqués sont de plus en plus hétérogènes. Cette hétérogénéité peut avoir plusieurs aspects, comme : types de composant (CPU, Mémoire, ASIC, etc.), protocoles de communication utilisés (FIFO, handshake, etc.), types de données échangées ou même la nature de composants (électroniques, mécanique, optique, etc.). En plus, pour concevoir de tels systèmes, il faut d'abord les spécifier. Ces spécifications peuvent contenir des composants décrits en différents langages ou à différents niveaux d'abstraction, ce qui ajoute un aspect d'hétérogénéité de plus.

Une autre caractéristique très particulière aux systèmes embarqués par rapport à d'autres systèmes électroniques concerne leurs contraintes de temps de mise sur le marché. Dans ce contexte, il faut absolument réduire le temps de conception et de validation de tels systèmes. Dans les sections qui suivent, nous allons examiner de plus près les systèmes embarqués ainsi que les étapes nécessaires pour les concevoir et les valider.

2.2.1 Concepts de base pour la spécification des systèmes

Les concepts de base qui sont présentés ici peuvent s'appliquer aux systèmes hétérogènes électroniques d'une manière générale, et pas seulement à ceux qui sont embarqués. Trois concepts sont fondamentaux pour la spécification des systèmes hétérogènes : le module, l'interface de communication et l'interconnexion. Leur composition forme ce que nous appelons un système. Nous pouvons définir qu'un système est une entité formée par un ensemble de modules interconnectés qui communiquent entre eux via des interfaces de communication comme le montre la Figure 2-1 Par la suite, nous expliquons non seulement ces trois concepts fondamentaux, mais aussi d'autres liés aux systèmes qui seront utilisés tout au long de ce travail.

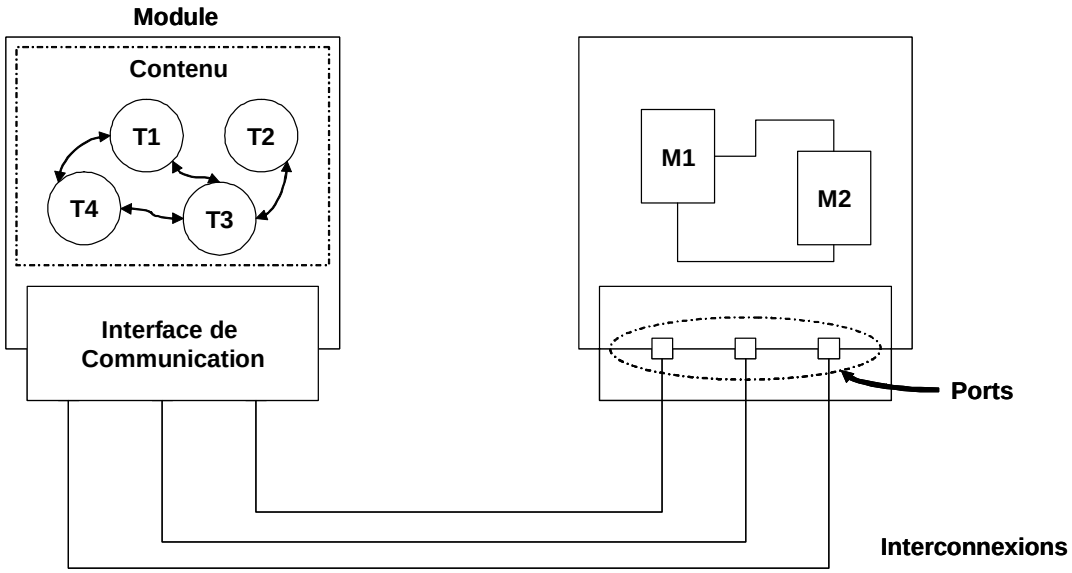


Figure 2-1 Concepts de Base de Systèmes

Module

Un module est une entité qui implémente une ou plusieurs fonctionnalités et qui communique avec l'environnement externe en utilisant une interface de communication. Un module est composé de deux parties distinctes : le contenu et l'interface.

Le contenu représente la fonctionnalité ou le comportement du module. C'est dans le contenu que réside le calcul effectué par le module. Ce contenu peut être représenté par des unités de calcul comme des tâches ou des processus (le module à gauche de la Figure 2-1), ou sinon par d'autres modules qui implémentent le comportement (le module à droite de la Figure 2-1). Si le contenu du module est composé d'autres modules, le module est dit hiérarchique. Ceci étant, un système peut être vu comme un module hiérarchique. La Figure 2-2 présente un exemple de module décrit en SystemC. Le contenu de ce module est implémenté par une tâche (thread) nommé *HS_Write*.

L'interface permet au module de communiquer avec le reste du système. Pour cela, l'interface est connectée au réseau d'interconnexions du système.

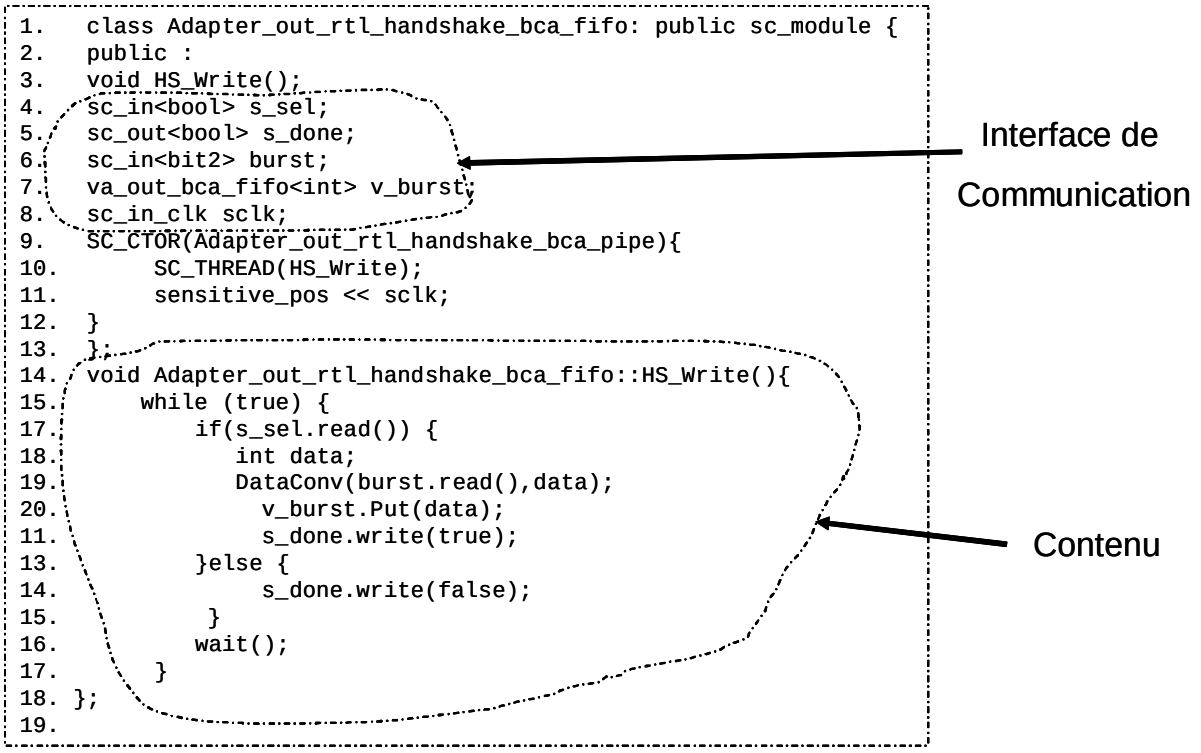


Figure 2-2 Exemple de Module décrit en SystemC

Interface de communication

Une interface de communication est un ensemble de points d'accès permettant à un module de communiquer avec le reste du système.

Selon le type d'implémentation du module (comme nous verrons plus tard), ces points d'accès peuvent être représentés par des fonctions de communication (pour le cas d'implémentation logicielle) ou par des ports physiques (pour le cas d'implémentation matérielle). Comme simplification, dans ce travail nous adoptons le terme « ports » pour designer ces point d'accès indépendamment du type d'implémentation du module.

Nous supposerons que l'interface de chaque module contient un ensemble de ports à travers lesquels les interactions possibles du module avec le reste du système sont spécifiées. Ainsi une interface est constituée de ports auxquels on associe des actions. Quelques exemples de ces actions sont : lire une donnée (*Get ()*), écrire une donnée (*Put (data)*), imprimer un fichier (*Print (file)*), etc. L'interface peut être présentée comme suit : Interface = $\{(P_i, \{A_{ij}, j=1..m\}), i= 1..n\}$, avec P_i désignant le port i, et A_{ij} une action sur ce port [Kri05].

Un port peut être élémentaire ou hiérarchique (c'est-à-dire englobant plusieurs ports ayant une même sémantique comme, par exemple, appartenant au même protocole de communication). Un exemple de port élémentaire est présenté par la Figure 2-2. Nous pouvons voir que le module décrit en SystemC possède cinq ports élémentaires (*s_sel*,

s_done, *burst*, *v_burst* et *sclk*). La Figure 2-3 montre un exemple de port hiérarchique. Dans cet exemple, le module doit envoyer une donnée au reste du système en utilisant le protocole de communication *Handshake*. Le module envoie d'abord une requête (port *Req*), puis la donnée (port *Data*) et enfin il reçoit un acquittement (port *Ack*). L'interface du module peut être spécifiée en utilisant les trois ports nécessaires au protocole ou en les regroupant dans un seul port hiérarchique *P*.

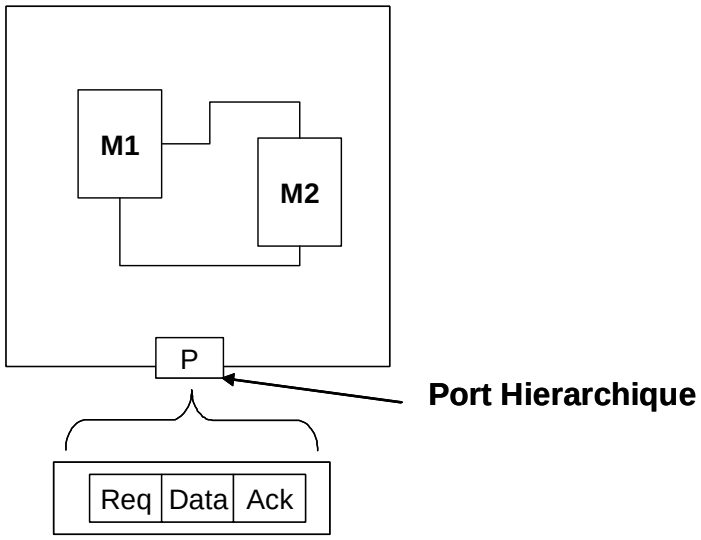


Figure 2-3 Exemple de Port Hiérarchique

API de communication

L'ensemble des actions associés aux ports constitue ce que nous appelons l'API (venant de l'anglais Application Programming Interface) de communication du module. La connaissance de l'API de communication d'un module est essentielle au concepteur pour qu'il sache comment un module peut interagir avec le reste du système. En effet, lors de la sélection par le concepteur d'un module qui va faire partie d'un système, il suffit de connaître la fonctionnalité générale du module et son API de communication. Les détails de l'implémentation et de la fonctionnalité peuvent être ignorés. En connaissant l'API de communication d'un module, le concepteur sait comment ce module peut être accédé par les autres modules du système.

L'API de communication d'un module varie selon le niveau d'abstraction auquel l'interface de celui-ci est décrite. Nous verrons, plus tard, que l'API peut faire abstraction de plusieurs aspects de la communication tels que : type de donnée, adressage, temps, etc. Ainsi, dans un niveau d'abstraction plus élevé, une API peut être composée de primitives comme *Print (file)*, où le type de donnée et l'adressage sont abstraits, tandis que dans un niveau plus

bas, l'API peut être composée par exemple d'une primitive pour lire un octet dans une adresse donnée (*Read (byte, address)*).

Protocole de communication

Un protocole de communication est un ensemble de règles qui définit la façon avec laquelle les données sont échangées sur un canal de communication. Ces règles peuvent englober plusieurs aspects de la communication comme : l'API de communication, le format des données, l'encodage des bits, la synchronisation des horloges, les mécanismes de correction et détection des erreurs de transmission et le routage des données [Sys04]. Handshake (rendez-vous), FIFO (file d'attente) et MPI (passage de message) sont des exemples de protocoles de communication connus.

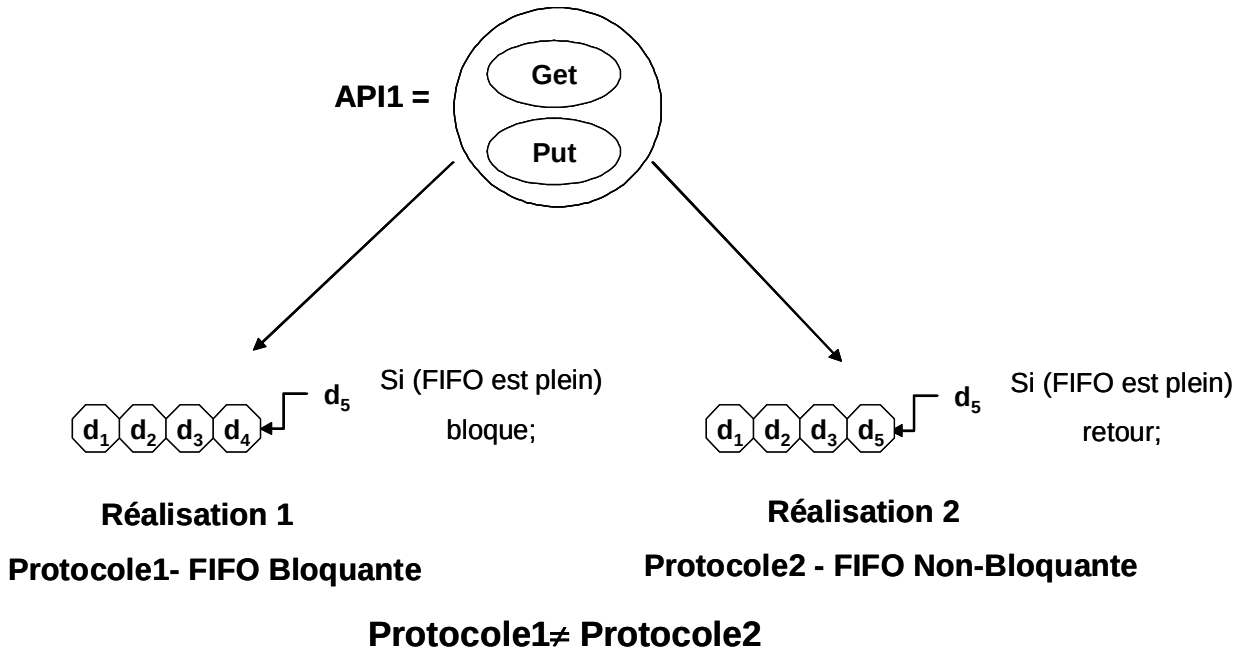


Figure 2-4 Différence entre API et Protocole de Communication

Le terme protocole de communication est souvent confondu avec l'API de communication. Ce dernier correspond aux primitives (actions) définies pour la communication. Alors que le premier est plutôt lié à la réalisation de la communication. La façon avec laquelle une API va être réalisée en utilisant les ressources disponibles définit le protocole de communication [Gra05]. La Figure 2-4 montre la différence entre les deux concepts. L'API possède les primitives *Get* pour lire une donnée et *Put* pour écrire une donnée. Mais cette API peut être réalisée en utilisant deux protocoles différents. Le premier

est une *FIFO* bloquante, où l'écriture d'une donnée bloque l'exécution du module si le tampon est plein ; et le deuxième est une *FIFO* non bloquante, où l'écriture d'une donnée n'implique pas le blocage du module si le tampon est plein.

Interconnexion

L'interconnexion relie des ports entre eux. Elle est chargée de transporter les données entre les modules. Comme le but principal de l'interconnexion est de permettre la communication entre les modules, elle est souvent désignée par le terme canal de communication. Les interconnexions permettent, en fait, la mise en relation de plusieurs modules pour rendre leur fonctionnement interdépendant [Tsi03]. Cette mise en relation se situe au niveau des interfaces de communication.

Les concepts qui définissent une interconnexion sont [Kri05] :

- Le média qui véhicule l'information (par exemple des fils physiques ou réseaux)
- Le type de données transmises (par exemple des données génériques ou données avec représentation fixe),
- Le comportement (la procédure utilisée pour acheminer les données à travers le média).

Selon le niveau d'abstraction et le modèle d'exécution, ces trois concepts peuvent se présenter différemment pour les canaux de communication. Pour des niveaux d'abstraction plus élevés, le canal de communication peut se présenter comme un canal abstrait qui contient des procédures de routage des donnés et d'arbitration et qui transporte des types de données complexes tels que des images. Tandis que pour les niveaux plus bas, le canal de communication peut correspondre à des fils physiques qui transportent des bits.

Modèle d'exécution

Le modèle d'exécution d'un module concerne la réalisation du module. Si le module est réalisé par un programme qui s'exécute sur un processeur, son modèle d'exécution est dit logiciel. Tandis que si le module est réalisé en utilisant un bloc matériel (ASIC), nous disons que le modèle d'exécution de ce module est matériel.

Le terme modèle d'exécution doit être différencié du terme modèle de calcul. Le deuxième consiste en une représentation du comportement (calcul) d'un module, alors que le premier concerne la façon dont ce comportement sera réalisé. Plusieurs travaux concernant les modèles de calcul existent dans la littérature. Quelques exemples de modèles de calcul sont : FSM, CDFG [Ora86], Réseau de Petri [Pet81], Evénement Discret [Buc90], Processus

de Kahn [Kan74], etc. Un modèle de calcul peut avoir plusieurs modèles d'exécution comme le montre la Figure 2-5. Dans ce cas, la fonctionnalité représentée par le modèle de calcul (CDFG) peut être réalisée soit par une fonction logicielle qui s'exécute sur un processeur, soit par un module hiérarchique matériel.

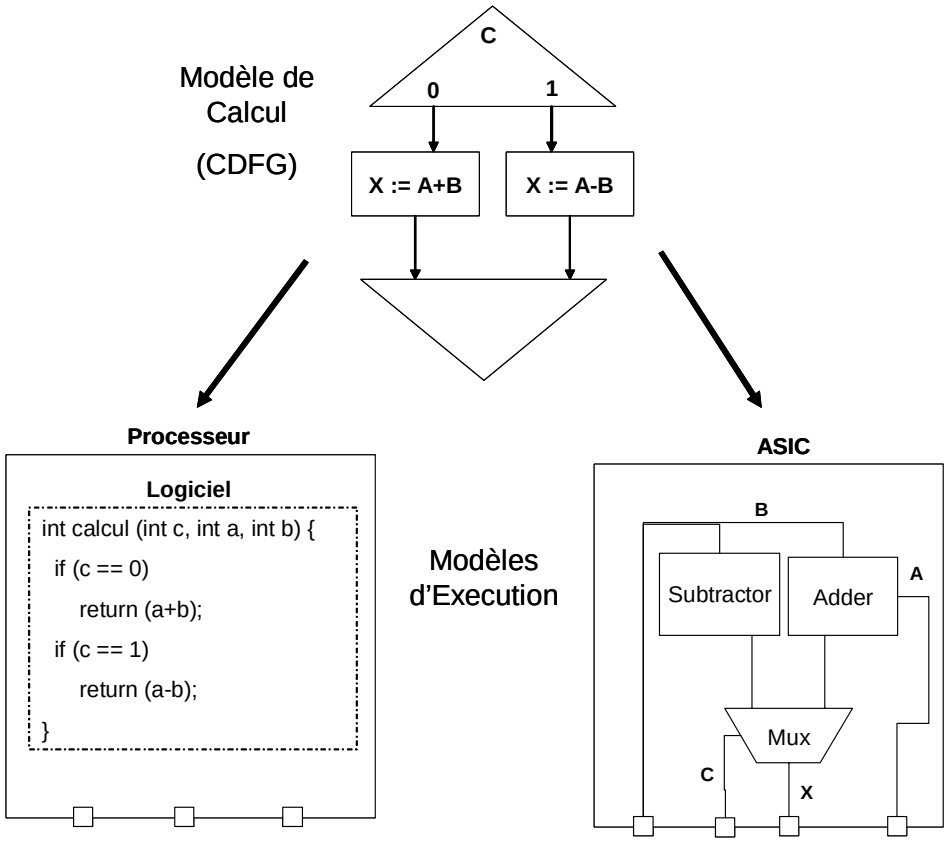


Figure 2-5 Différence entre Modèle d'exécution et Modèle de Calcul

Le modèle d'exécution détermine aussi comment un module communique avec le reste du système. Un module logiciel communique à travers des appels de fonctions logicielles. Alors que pour un module matériel, la communication est faite par moyen des ports physiques. Comme nous pouvons le remarquer, les deux modèles d'exécution utilisent différents concepts pour la communication. Lors de la conception d'un système, le concepteur doit implémenter des interfaces (comme nous le verrons plus tard) pour adapter la communication entre ces deux modèles d'exécution.

2.2.2 Niveaux d'abstraction des modules

La séparation de la partie de calcul de celle de la communication du module permet une certaine indépendance entre ces deux parties. Lors de la conception d'un système

hétérogène embarqué, le calcul et la communication peuvent être raffinés de façon indépendante. Le terme raffinement utilisé ici désigne le processus de transformation d'une entité (de calcul ou de communication) vers un niveau d'abstraction plus proche de l'implémentation finale. Dans ce travail, nous allons traiter les niveaux d'abstraction du calcul et de la communication séparément. Cette section est consacrée à analyser les niveaux d'abstraction de la partie de calcul du module. Comme le modèle d'exécution d'un module peut être logiciel ou matériel, nous traitons les deux cas. Nous verrons que selon le modèle d'exécution d'un module, les niveaux d'abstraction traitent des concepts bien différents.

Niveaux d'abstraction du logiciel

L'exécution du comportement d'un module logiciel se fait par l'interprétation d'un ensemble d'instructions (programme). Cette interprétation est faite par une plateforme (ou un environnement) d'exécution. Selon le niveau d'abstraction du logiciel, cet environnement se présente d'une façon différente. Il peut être représenté soit par un environnement logiciel tel qu'un système d'exploitation ou un *middleware*, soit par une plateforme matérielle comme un processeur. Généralement, un module logiciel s'exécute sur plusieurs couches d'environnement d'exécution où la dernière est toujours le processeur.

Le niveau d'abstraction d'un module logiciel concerne l'abstraction des opérations sur les données (calcul) effectuées par ce module. Ce niveau d'abstraction dépend, en effet, de l'ensemble des opérations (API) fournies par l'environnement d'exécution juste au dessous du module logiciel. Prenons comme exemple un environnement d'exécution qui fournit des opérations pour manipuler des matrices. L'API de cet environnement d'exécution peut posséder des opérations comme MultiplierMatrice (Matrice A, Matrice B) pour multiplier deux matrices. Le module logiciel qui s'exécute sur cet environnement est décrit à un niveau d'abstraction très élevé. Un autre exemple est quand l'environnement d'exécution fournit des opérations telles que *LOAD*/ *STORE* de données de/sur la mémoire, le calcul va être décrit dans un niveau d'abstraction très bas. Nous pouvons voir que l'API fournie par le premier environnement fait une abstraction totale de la plateforme matérielle d'implémentation (le processeur), tandis que l'abstraction faite par l'API du deuxième est presque inexistante. Plus les détails du processeur sont cachés, plus élevé est le niveau d'abstraction du calcul d'un module logiciel.

Par la suite, nous présentons les différents niveaux d'abstraction auxquels un module logiciel peut être décrit :

-Niveau Application – Dans ce niveau, l'abstraction de la plateforme matérielle est totale. Les détails d'implémentation tels que le parallélisme et la synchronisation entre les différentes unités de calcul sont implicites à ce niveau. Normalement, le module logiciel est décrit en utilisant un langage de programmation de haut niveau comme C++, Java, ADA, etc. L'API de manipulation des matrices, mentionnée avant, est un exemple du type d'API fournie à ce niveau.

-Niveau OS – À ce niveau, le concepteur prend en compte la présence d'un système d'exploitation pour réaliser le calcul. L'utilisation d'un système d'exploitation rend la synchronisation et la gestion de ressources partagées entre les unités de calcul explicites. Pourtant, l'implémentation du système d'exploitation est toujours abstraite. Des détails tels que la réalisation de l'ordonnancement des unités de calcul ou de la gestion des périphériques sont cachés. Un module logiciel décrit au niveau OS consiste en un ensemble d'appels au système d'exploitation. Un exemple d'API fournie à ce niveau est le *thread* POSIX [But97].

-Niveau HAL – L'API fournie à ce niveau consiste en un ensemble d'opérations qui permettent d'abstraire les accès physiques au matériel (HAL vient de l'anglais Hardware Abstraction Layer). L'implémentation du système d'exploitation est fixée, mais il reste quelques détails de la plateforme matérielle cachés comme le processeur cible, l'architecture mémoire, etc. Un module logiciel décrit au niveau HAL contient des opérations qu'abstraient les fonctions de base d'accès au matériel tel que l'initialisation (*boot*), le changement de contexte, les entrées/sorties de base, la gestion des interruptions et l'accès aux périphériques. L'API HAL de RISC OS est un exemple d'API à ce niveau-là [Ris02].

-Niveau ISA - Le niveau ISA (venant de l'anglais Instruction Set Architecture) correspond à la spécification du logiciel en un langage assembleur relatif à un processeur cible. Le logiciel est décrit en utilisant des opérations tels que *LOAD* et *STORE*. Les accès physiques au matériel sont explicites. Le jeu d'instructions du processeur est l'API fournie à ce niveau.

Niveaux d'abstraction du matériel

Différemment au modèle d'exécution logiciel, l'exécution d'un module matériel ne passe pas par l'interprétation séquentielle d'un ensemble d'instructions. L'exécution du

comportement d'un module matériel correspond vraiment à la réalisation du calcul par une plateforme matérielle.

Pour le modèle d'exécution matériel, nous pouvons distinguer les deux niveaux d'abstraction principaux :

-Niveau Comportemental – À ce niveau, la fonctionnalité du module est décrite, mais les ressources nécessaires pour accomplir sa réalisation sont abstraites. C'est-à-dire, aucune hypothèse n'est faite sur l'implémentation physique. Généralement, le concepteur utilise des langages HDL (venant de l'anglais Hardware Description Language) tels que VHDL, Verilog ou SystemC pour décrire un module à ce niveau. Ces langages offrent au concepteur des opérations de calcul de très haut niveau qui sont comparables à ceux des langages de programmation tels que : C++, Java, etc.

-Niveau RTL – À ce niveau, les détails de synchronisation physiques sont pris en compte, les ressources matérielles sont allouées et assignées aux différentes parties de calcul du module, et l'exécution du comportement du module est réordonnancé. Comme la description d'un module à ce niveau-là peut être très complexe, il existe plusieurs outils qui font la synthèse de haut niveau pour faciliter le passage du niveau comportemental au RTL [Gaj91].

2.2.3 Niveaux d'abstraction de la communication

Ainsi que le comportement du module, la partie de communication peut être décrite à plusieurs niveaux d'abstraction. Les paramètres qui définissent le niveau d'abstraction de la communication sont les suivants :

-Données Transmises – la granularité des données transmises pendant une opération de communication joue un rôle important en ce qui concerne le niveau d'abstraction de la communication. Normalement, dans les niveaux plus élevés, les données transmises consistent en types abstraits et plus complexes comme une image par exemple. Alors que pour les niveaux plus bas comme RTL, la communication est faite par l'envoi/réception des bits.

-Temps – Le modèle de temps utilisé pour l'échange de données influence également le niveau d'abstraction de la communication. Le modèle de temps peut utiliser une horloge logique ou physique. Avec l'horloge logique, le temps n'avance que s'il y a un échange de données entre deux ou plusieurs modules. L'horloge logique nous donne, effectivement, ce que nous appelons l'ordre partiel des événements d'échange de

données. Avec l'horloge physique, nous avons la modélisation au niveau cycle du temps et ainsi nous pouvons obtenir l'ordre total des événements d'échange de données. La communication décrite à des niveaux d'abstraction plus élevés, utilise généralement une horloge logique, tandis que l'horloge physique est employée pour les niveaux plus bas.

-Adressage – Ce critère concerne l'identification du module qui doit recevoir les données. Si la communication entre deux ou plusieurs modules est faite par l'explicitation de l'adresse réelle, nous disons que le niveau d'abstraction de la communication est bas. Comme nous le verrons plus tard, cette adresse peut être logique ou physique. La première est abstraite. Le concepteur ou l'environnement d'exécution assigne un numéro d'adresse quelconque à chaque module. La deuxième correspond à l'adresse réelle du module.

-Contrôle - Ce critère peut englober plusieurs aspects de la communication tels que : la synchronisation, l'arbitrage, le routage de données, etc. Plus le contrôle est explicite, plus le niveau d'abstraction est bas. Un exemple de contrôle explicite est lors de la communication d'une image, l'ordre des cadres de l'image transmise doit être envoyé aussi.

Selon les paramètres mentionnés ci-dessus, nous pouvons classifier la communication en cinq niveaux d'abstraction. Chaque niveau sera défini en fonction du degré d'abstraction de tous ces paramètres comme le montre le Tableau 2-1.

Paramètres **Niveau d'Abstraction**	**Données**	**Temps**	**Adresse**	**Contrôle**
Service	Service et Structure de données (ex : image)	Causalité des événements	_	_
Message	Structure de données	Causalité des événements	Adresse logique (symbolique)	- Synchronisation - Ordonnancement des blocs de données.
Transactionnel (TLM)	Structure de données	Horloge logique-l'ordre des transactions	Adresse logique (symbolique)	- Synchronisation - Ordonnancement des blocs de données. - Arbitration
Transfert (BCA)	Données scalaires (ex : entier ou booléen)	Horloge physique-cycle d'horloge	Adresse physique	- Synchronisation - Ordonnancement des blocs de données. - Arbitration - Routage
RTL	Donnés en représentation fixe (bits)	Horloge physique-cycle d'horloge	Adresse physique	Synchronisation - Ordonnancement des blocs de données. - Arbitration - Routage

Tableau 2-1 Les Degrés d'Abstraction de Données, Temps, Adressage et Contrôle selon les Niveaux d'Abstraction de la Communication : (a) Le Niveau Service, (b) Le Niveau Message, (c) Le Niveau TLM, (d) Le Niveau Transfert, (e) Le Niveau RTL

Le niveau service

L'abstraction totale des données, de temps, d'adressage et de contrôle caractérise ce niveau. La communication est réalisée par une combinaison de requêtes et services. Les données échangées peuvent être complexes comme une image ou un fichier. La notion de temps n'existe pas. Toute la communication est réalisée dans un délai égal à zéro et se base sur le principe de causalité. C'est-à-dire, un module ne recevra une donnée que si celle-ci a

été envoyée par un autre module. L'adressage n'est pas pris en compte non plus, car le canal de communication se chargera de trouver le module qui peut fournir un service requis par un autre et de router les données échangées. Le canal de communication est responsable aussi de l'arbitration et de la synchronisation de la communication. Ceci étant, le contrôle est abstrait à ce niveau-là. Une primitive de communication typique du niveau service est le *Print(file)*. Un exemple d'un canal de communication avec ces caractéristiques est l'ORB (venant de l'anglais Object Request Broker) de CORBA [Cor97].

Le niveau message

Le temps et les données continuent a être abstraites dans le niveau message, par contre l'adressage et le contrôle sont plus explicites qu'avant. Une opération de transmission des données est caractérisée par ce que nous appelons l'échange de messages. Les données transmises peuvent être des structures de données complexes et la notion de temps est similaire à celui du niveau service. Une adresse logique est utilisée pour spécifier le module qui doit recevoir/envoyer les données et certains paramètres de contrôle sont nécessaires pour la communication tels que : la synchronisation (communication bloquante ou non bloquante), le type de donnée, le nombre de blocs de données envoyées, l'adresse du bloc, etc. Par contre, le canal de communication est chargé de décoder l'adresse logique, de router les données et d'arbitrer l'accès au canal. Des exemples de primitives du niveau message sont le *BlockingSend(message, adresse destinataire, type de donnée, numéro du bloc*) et le *BlockingReceive(message, adresse source, type de donnée, numéro du bloc*) similaires aux primitives trouvées dans le standard de communication MPI [Mpi04].

Le niveau transactionnel (TLM)

Le niveau transactionnel que nous allons appeler ici TLM (venant de l'anglais Transaction Level Modeling) est similaire au niveau message, mais le temps et le contrôle sont plus explicites. En fait, [Col04] considère que le terme TLM englobe les niveaux message, transactionnel et transfert. Cependant, dans ce travail, TLM désignera le niveau transactionnel. A ce niveau-là, chaque échange de données est appelé transaction. Les données peuvent être des structures de données, mais une horloge logique est utilisée pour que le concepteur puisse examiner l'ordre partiel des transactions. L'adressage est réglé par des adresses logiques comme pour le niveau message. Le contrôle est plus explicite et peut adresser l'arbitration pour l'accès au canal de communication. Certains aspects de contrôle sont abstraits tel que le routage. Le canal de communication est chargé de les implémenter.

Les primitives de communication typiques de ce niveau sont *Read (transaction, adresse source, numéro du bloc, paramètre d'arbitration)* et *Write (transaction, adresse destinataire, numéro du bloc, paramètre d'arbitration)*. Des exemples de canaux TLM peuvent être trouvés dans [Pas02] [Cop04] [Col04].

Le niveau transfert (BCA)

Le niveau transfert ou BCA (venant de l'anglais Bus Cycle Accurate) n'abstrait que les données transmises. Mais, les types de données à ce niveau sont moins complexes qu'aux autres niveaux cités antérieurement. Les types de données scalaires sont utilisés à ce niveau, tel que le type entier. L'horloge employée est physique, ce qui signifie que la précision de temps est au cycle près. L'adressage est explicite, c'est-à-dire, il correspond à l'adresse physique. Finalement, les canaux de communication ne sont responsables que pour transférer les données. La synchronisation, l'arbitrage et le routage sont explicites. La communication est achevée en utilisant plusieurs canaux de communication, où chacun de ces canaux correspond soit au contrôle, soit à l'adresse, soit aux données. Des exemples de primitive de communication à ce niveau sont *Put(donnée)* et *Get(donnée)*.

Le niveau RTL

Tous les critères de communication sont explicites à ce niveau. Les données transmises sont les bits et l'unité de temps est le cycle d'horloge physique. Le temps de communication est obtenu par le nombre de cycles d'horloge nécessaires pour transférer un bit (ou une chaîne de bits). L'adressage et le contrôle sont explicités par des signaux RTL. Le niveau RTL donne au concepteur une idée bien précise de comment se comporte la communication. Les primitives de communication comportent la lecture et l'écriture (*Read(bit)* et *Write(bit)*) d'un port physique.

2.2.4 Flot de conception de systèmes hétérogènes embarqués

La complexité des systèmes hétérogènes embarqués actuels fait que leur conception est divisée en plusieurs étapes. Généralement, le système commence par être spécifié à un niveau d'abstraction très élevé et passe par des étapes successives de raffinement jusqu'à ce que nous obtenions l'implémentation RTL du système. La Figure 2-6 présente un flot de conception des systèmes embarqués simplifié.

Le flot débute par une spécification fonctionnelle du système. Cette spécification sert à fixer la fonctionnalité requise du système. Elle consiste en plusieurs unités de calcul (tâches ou processus) qui communiquent entre eux en utilisant des primitives de communication au niveau service ou message. Dans cette étape, aucune décision n'est prise en ce qui concerne l'architecture du système.

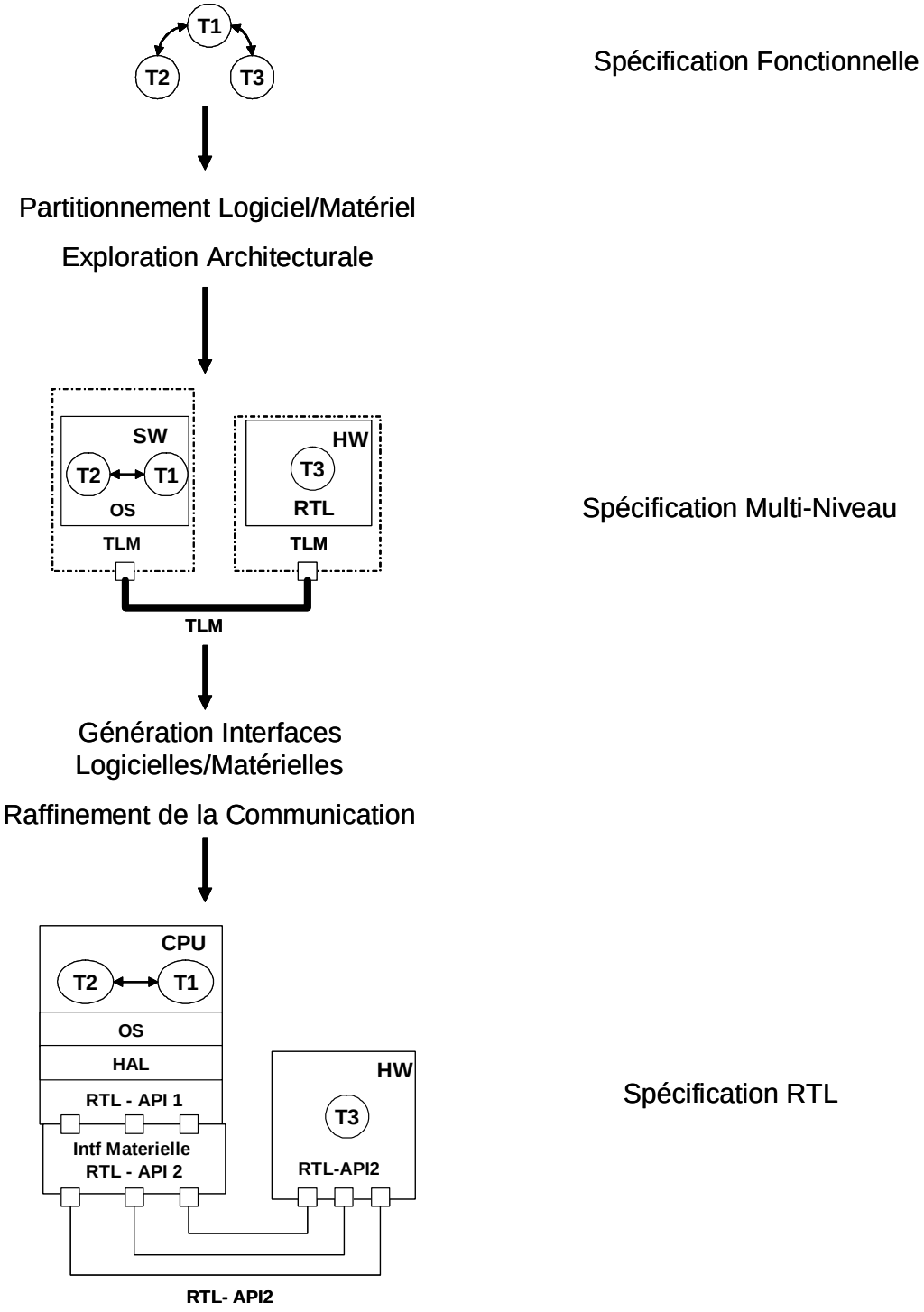


Figure 2-6 Exemple de Flot de Conception de Systèmes Hétérogènes Embarqués

La prochaine étape consiste à partitionner le système et faire une exploration architecturale du système. Le partitionnement du système consiste à décider quelles unités de calcul seront implémentées en logiciel ou matériel. Certaines décisions architecturales peuvent être faites telles que : le module matériel existant (ASIC) qui sera utilisé, la topologie de la communication, etc. Le calcul des modules logiciels est généralement décrit au niveau OS, tandis que celui des modules matériels est décrit soit au niveau comportemental soit au niveau RTL (dans le cas d'un ASIC déjà conçu). Il faut remarquer que les modules logiciels s'exécutent sur des modules matériels ayant une interface de communication à un niveau d'abstraction élevé. La communication entre les modules est faite par des primitives de niveau élevé comme par exemple TLM. Le fait que les modules et la communication soient toujours spécifiés à un niveau d'abstraction élevé facilite l'exploration des différentes solutions architecturales du système de la part du concepteur. Car beaucoup de détails sur l'implémentation du système restent cachés, ce qui signifie que le changement de certaines parties du système n'implique pas un effort significatif de la part du concepteur.

La dernière étape du flot consiste à raffiner le système au niveau RTL. En réalité, cette étape pourrait être divisée en plusieurs sub-étapes. Nous verrons dans les expérimentations présentés par le chapitre 5, que nous avons découpé cette étape en deux phases distinctes : une pour valider le système d'exploitation (OS) générique et l'autre pour valider le HAL spécifique au processeur. Les modules logiciels sont raffinés par moyen de la génération des interfaces logicielles (OS et HAL dans la Figure 2-6). Ces interfaces logicielles sont nécessaires pour permettre que les appels de fonctions utilisés dans des modules logiciels soient traduits en accès à la partie matérielle (les ports physiques par exemple) du processeur. La communication est raffinée au niveau RTL et possède un protocole de communication bien défini. En effet, lors du raffinement de la communication celui-ci peut devenir un module matériel (bus matériel). Les modules matériels peuvent aussi être raffinés, si dans l'étape d'avant ils étaient décrits au niveau comportemental. Finalement, leur partie de communication est adaptée (adaptation de protocole, type de donnée, etc.) à la communication globale du système à travers des interfaces matérielles.

2.2.5 Flot de Validation de Systèmes Hétérogènes Embarqués

En se basant sur le flot de conception présenté dans la section précédente, nous pouvons également définir un flot de validation. Pour chaque étape du flot de conception, nous avons un différent type de validation comme le montre la Figure 2-7.

Au début, nous faisons une validation fonctionnelle du système. Dans cette étape, aucun aspect de performance ou de temps n'est pris en compte. Le but est de valider la fonctionnalité globale du système. Nous testons si pour un ensemble de données d'entrée, si les sorties qui sont produites sont correctes.

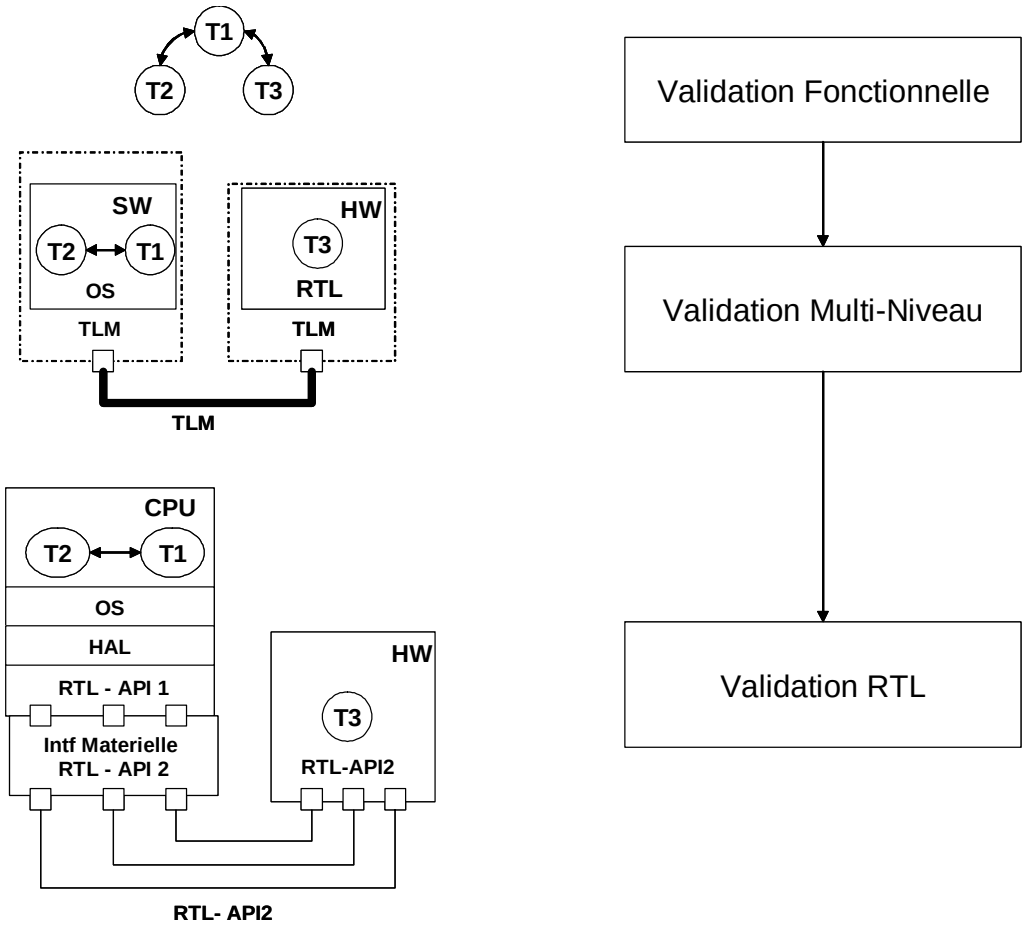


Figure 2-7 Exemple de Flot de Validation de Systèmes Hétérogènes Embarqués

La prochaine étape consiste à valider le partitionnement et les décisions architecturales prises. Dans cette étape, le système est spécifié comme un ensemble de modules qui communiquent en utilisant des primitives de communication à différents niveaux d'abstraction. Ceci étant, nous réalisons une validation multi-niveaux. Dans cette étape nous nous intéressons aussi à valider l'ordre partiel des opérations de communication. La Figure 2-

7 montre que la communication est réalisée au niveau TLM, ce qui veut dire que nous validons l'ordre des transactions. La dernière étape du flot s'agit de valider l'implémentation RTL du système. Nous validons les interfaces logicielles et matérielles, et la communication au niveau RTL. Dans cette étape, chaque contrainte du système et choix d'implémentation doivent être validés. La précision du temps pour chaque opération de communication est au cycle près de l'horloge du système.

Il faut remarquer que la validation d'un système en plusieurs étapes est essentielle pour réduire le temps, l'effort et le coût financier de ce processus. Il est observé que plus la validation intervient tôt dans le flot de conception d'un système, plus une erreur peut être corrigée rapidement [Cal95]. En plus, une erreur détectée au début du flot de conception est moins coûteuse que si elle est découverte à la fin.

2.3 Méthodes de validation de systèmes hétérogènes embarqués

Il existe plusieurs méthodes pour valider un système hétérogène embarqué. Cette partie du chapitre est consacrée à présenter ces différentes méthodes. Les méthodes analysées ici sont : la vérification formelle, le prototypage matériel et la cosimulation. Nous présentons d'abord le critère que nous allons utiliser pour évaluer ces méthodes et ensuite nous détaillons chaque approche. Nous verrons que chaque méthode est plus adaptée à certaines étapes du flot de validation, et que l'idéal serait de les appliquer dans les différentes étapes de validation. En effet, la plupart des projets de conception des systèmes hétérogènes embarqués actuellement combine les diverses méthodes de validation.

2.3.1 Critères de classification des méthodes de validation

Pour que nous puissions analyser la puissance et les faiblesses de chaque méthode de validation, il nous faut fixer certains critères de classification. Dans ce travail, les critères auxquels nous allons nous intéresser sont : (1) le coût pour construire le modèle du système; (2) la flexibilité de la méthode ; (3) la capacité d'utiliser la méthode aux différentes étapes du flot de validation ; (4) la vitesse de validation.

Coût pour construire le modèle

Chaque méthode utilise un modèle du système à valider. Nous verrons que, par exemple, la vérification formelle utilise des notations formelles, tandis que la cosimulation

emploie des modèles de simulation sur un ordinateur. Ce critère correspond au temps et à l'effort nécessaire pour établir le modèle du système qui sera validé.

Flexibilité

La conception d'un système passe par une étape d'exploration architecturale. Il faut que la méthode soit flexible, pour que le concepteur puisse modifier le système, sans que cela implique une augmentation importante de l'effort et du temps pour reconstruire le modèle de validation du système. Cette flexibilité inclut aussi l'adéquation à des systèmes complexes.

Capacité d'utilisation aux différentes étapes de validation

Un système doit être validé à chaque étape de la conception. Ainsi, la méthode doit permettre son utilisation pour chaque étape de validation. Nous verrons que certaines méthodes sont plus adaptées à certaines étapes de la validation.

Précision

Chaque étape de la validation requiert un certain degré de précision en ce qui concernent le temps, la granularité des événements et l'exactitude. La méthode doit fournir la précision adéquate à l'étape de validation.

Vitesse

Une fois que le modèle du système est construit, le temps de validation du système doit être pris en compte. Comme le but de toutes les méthodes est de réduire le temps de validation, la vitesse ne doit pas représenter un goulot d'étranglement du processus de validation.

2.3.2 Vérification formelle

Le principe de base de la vérification formelle est de prouver mathématiquement que la fonctionnalité du système est correcte. Ceci est facilité quand le système lui-même est spécifié en utilisant des notations formelles. Il existe des langages et des modèles de calcul qui facilitent la vérification formelle des systèmes tels que : Occam [May84], Z [Spi89], B [Abr97], LOTOS [Iso89], Réseaux de Petri, etc. D'autres langages peuvent être utilisés comme VHDL et Verilog par exemple, mais généralement le concepteur ne peut utiliser qu'un sous ensemble de ces langages. La vérification formelle peut être utilisée pour le

déboguage de la spécification et pour la vérification de l'implémentation [Sta94]. Le premier vérifie si le système a été spécifié correctement, alors que le deuxième vérifie si le système est bien implémenté. Dans la première catégorie, nous pouvons citer les approches proposées par POLIS [Edw97] et par Chakrabarti et al [Cha02]. Dans la deuxième catégorie, nous trouvons également POLIS, mais aussi Esterel Studio [Est04], SPIN [Bel80], Schubert [Sch03], entre autres. L'utilisation de modèles formels pour décrire les systèmes peut aussi faciliter le processus de raffinement. Barros et al [Bar94] a proposé le partitionnement logiciel/matériel correct à l'aide des méthodes formelles. Dans ce cas, il n'y a pas la nécessité de faire une validation, parce que la procédure elle-même est formellement correcte.

La vérification formelle utilise des techniques différentes pour valider les modules matériels et logiciels. Les premiers sont représentés par des BDD (venant de l'anglais Binary Decision Diagram) et sont validés par des techniques comme « symbolic model checking » [Ker99]. Alors que les modules logiciels sont représentés par des modèles similaires au MSC (venant de l'anglais Message Sequence Charts), et sont validés en utilisant la technique « partial order reduction » [Sdl00]. Lors de la validation, le concepteur doit fournir un ensemble de propriétés du système qu'il veut vérifier. Ces propriétés peuvent être par exemple de savoir si le système va entrer dans un « deadlock » ou s'il aura un « overflow ».

Généralement, la vérification formelle est bien adaptée aux systèmes simples. Pour les systèmes plus complexes, le coût pour construire le modèle devient significatif. Ce coût est relatif à la spécification du système et des propriétés d'une façon formelle. Une fois que le modèle est construit, il existe des outils qui font la vérification automatiquement. La flexibilité est aussi pénalisée parce qu'il est difficile de changer le modèle en cas de changement du système. Même pour les systèmes simples, la vérification formelle restreint un peu le langage de spécification, ce qui réduit la flexibilité de la méthode. Ceci représente un obstacle important pour l'utilisation de la méthode en toutes les étapes du flot de validation. Généralement, la vérification formelle est utilisée seulement dans certaines étapes, notamment pour la validation fonctionnelle et pour valider certaines parties du système au niveau RTL. La précision de la méthode est un peu faible parce qu'il est difficile de mettre des contraintes temporelles dans le modèle. Enfin la vitesse est bonne, une fois que le modèle est construit, le concepteur peut exécuter les outils sur un ordinateur pour vérifier le système.

Notons que, actuellement, la vérification formelle est souvent associée avec d'autres méthodes pour assurer une meilleure validation du système.

2.3.3 Prototypage matériel

Le prototypage matériel est une méthode de validation qui consiste en l'exécution du système sur une plateforme matérielle différente de celle employée pour l'implémentation finale du système [Ros98].

La plateforme matérielle (ou plateforme de prototypage) utilisée dans cette méthode peut être composée d'une grande variété de composants : processeurs, bus, mémoires, blocs matériels reconfigurables (comme des FPGA), etc. Selon la composition de la plateforme, nous pouvons la classifier en deux types [Sas04] :

-plateforme spécifique à l'application – les composants matériels de la plateforme sont spécifiques à l'application. Dans ce type de plateforme, nous avons une transposition directe de tous les modules du système vers les composants de la plateforme. Les interconnexions sont réalisées par des fils ou des liaisons sur un circuit imprimé.

-plateforme reconfigurable - dans ce cas les composants matériels de la plateforme sont re-configurables. Cette plateforme est souvent composée de plusieurs processeurs (qui peuvent aussi être de différents types) et des FPGAs. Ces FPGAs, généralement, implémentent les modules matériels du système ainsi que les interconnexions. Parmi les plateformes existantes de ce type, nous pouvons citer ARM PrimeXsys [Arm01], LOPIOM [Mos96] et une proposée par Ramanathan et al [Ram01].

La Figure 2-8 présente le principe de base de cette méthode pour valider un système. Le concepteur doit adapter la description RTL du système à la plateforme de prototypage. La complexité de cette adaptation dépend, en effet, de la « proximité » de la plateforme par rapport au système. Si le concepteur utilise une plateforme spécifique à l'application, la complexité sera réduite. Le problème d'utiliser une telle plateforme est que celle-ci réduit la possibilité d'exploration architecturale du système. Par contre, si le concepteur emploi une plateforme re-configurable, il aura un gain en terme de flexibilité, mais la complexité de modification du modèle RTL peut être importante. En fait, cette adaptation doit considérer plusieurs aspects de la plateforme et du système comme c'est expliqué en [Sas04]. Il peut être nécessaire de changer une partie des interfaces logicielles par exemple. En plus, le concepteur doit configurer la plateforme et générer le code pour les FPGAs.

L'automatisation du processus de construction des modèles pour la prototypage matériel n'existe pas encore. Sasongko [Sas04] a automatisé certaines parties, mais la plupart

sont toujours à faire manuellement. Ainsi, le coût de construction du modèle peut être significatif. Cela dépend de la compatibilité entre l'application et la plateforme. La flexibilité de la méthode est aussi réduite à cause de cela, car un changement du système peut impliquer un grand effort de la part du concepteur. Cette méthode ne peut pas être appliquée pour toutes les étapes du flot de validation, puisque le système doit être à un niveau d'abstraction très bas (RTL). La précision du prototypage matériel dépend aussi de la « proximité » de la plateforme par rapport au système. Le grand avantage de cette méthode par rapport à d'autres est la vitesse, car le système est exécuté sur des composants matériels, ce qui donne une vitesse très élevée.

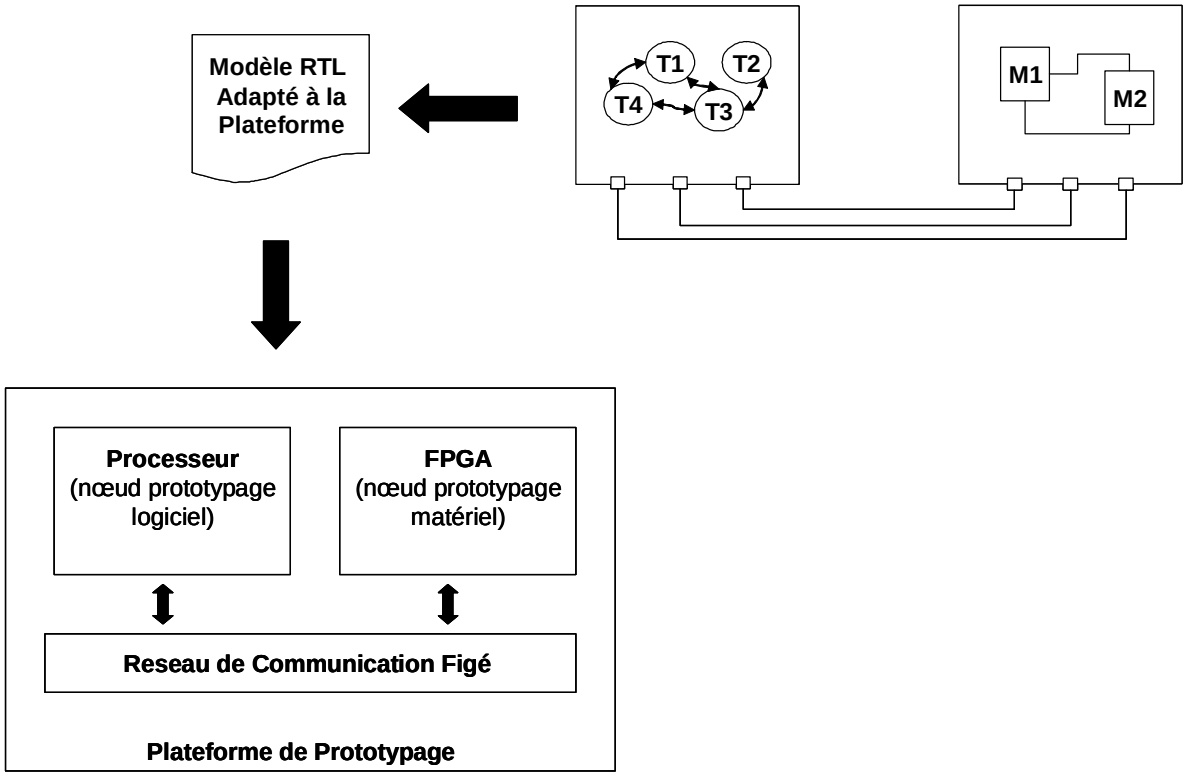


Figure 2-8 Validation par Prototypage Matériel

2.3.4 Cosimulation

La validation par cosimulation consiste en l'utilisation d'un ou de plusieurs simulateurs pour vérifier si le comportement du système est correct. Ces simulateurs sont, en effet, des outils logiciels qui prennent comme entrées un modèle du système et des vecteurs de test (liste des valeurs possibles pour les entrées du système), et exécutent le modèle du système sur un ordinateur. Ce modèle du système est appelé modèle de simulation comme le montre la Figure 2-9. Le principe de base de la méthode est de réduire la probabilité de défaut

du système par l'exposition massive de cas de test. Ces cas de test sont souvent appelés « testbench ».

Le terme cosimulation est souvent employé pour designer la validation du système en utilisant plusieurs simulateurs. Dans ce travail, ce terme sera plus général. Il sera utilisé pour designer la simulation des systèmes ayant des modules et interconnexions hétérogènes. Cette hétérogénéité peut être due à différents aspects tels que : langages de spécification, modèles de calcul, niveaux d'abstraction, APIs de communication, entre autres. Ainsi, le terme cosimulation sera utilisé indépendamment du nombre de simulateurs qui participent à la simulation.

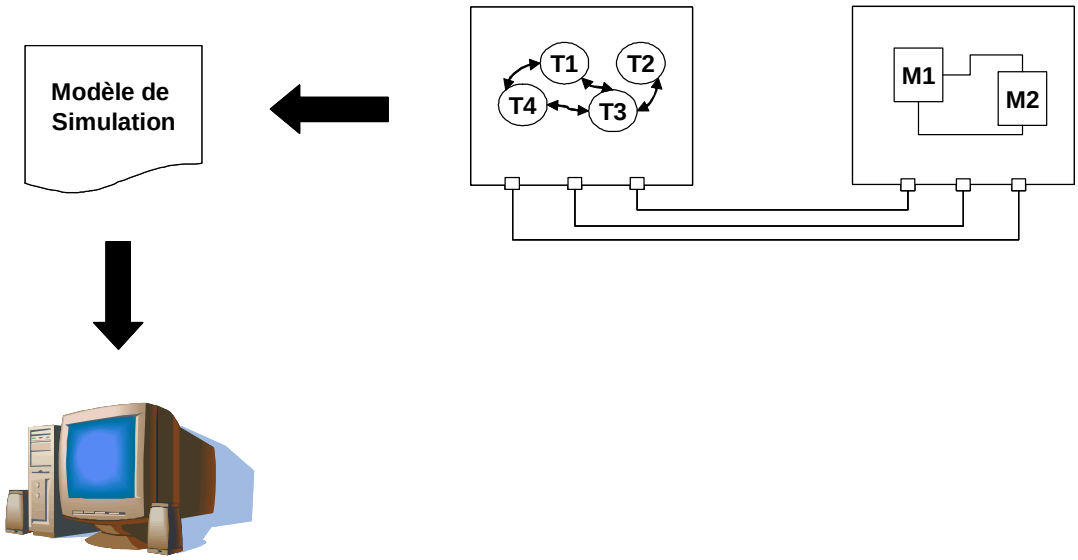


Figure 2-9 Validation Par Cosimulation

La cosimulation est la méthode la plus simple à utiliser. Il existe plusieurs environnements de conception qui fournissent des outils nécessaires pour aider à la création des modèles de simulation à partir des langages de spécification (voir section 2.4). La création du modèle peut être complètement automatisée, ce qui réduit énormément le coût de la méthode. Cette caractéristique est responsable aussi de la grande flexibilité de la cosimulation. Un changement du système n'implique que re-générer le modèle. La flexibilité est avantageuse aussi lorsqu'il y a plusieurs simulateurs pour les différents langages de spécification. La méthode ne pose pas des contraintes significatives par rapport au langage de spécification du système.

La cosimulation peut être utilisée pour des systèmes à différents niveaux d'abstraction, c'est-à-dire qu'elle peut être utilisée tout au long du flot de validation. La précision varie selon le niveau d'abstraction. Pour les niveaux d'abstraction plus élevés, la précision est plus

faible et pour les niveaux plus bas elle est bonne. Autrement dit, la cosimulation fournit la précision convenable à chaque étape de la validation. La vitesse varie aussi selon l'étape de validation. Elle est plus grande lorsque le système est décrit aux niveaux d'abstraction plus élevés et réduite aux niveaux plus bas. En effet, la vitesse de la cosimulation aux niveaux d'abstraction plus bas est le point faible de la méthode. Même si la méthode peut être utilisé dans toutes les étapes de validation, sa vitesse au niveau RTL fait que d'autres méthodes, comme le prototypage matériel par exemple, sont plus adaptées.

2.4 Validation de systèmes hétérogènes embarqués par cosimulation

Dans la section précédente, nous avons vu que parmi les différentes méthodes de validation, la cosimulation est, d'une manière générale, la plus avantageuse. Le but de ce travail est de générer des modèles de simulation automatiquement pour que le processus de validation par cosimulation soit plus rapide. Les problèmes liés à la vitesse de simulation aux niveaux d'abstraction plus bas sortent du cadre de notre travail. Nous essayons, dans ce travail, de réduire le coût de construction du modèle de cosimulation, par une approche de génération automatique. La cosimulation doit prendre en compte divers types d'hétérogénéité. Englober tous ces types d'hétérogénéité n'est pas trivial. Ainsi, nous verrons que plusieurs approches ont essayé de résoudre un sous ensemble du problème d'hétérogénéité. Ce travail se concentre sur les systèmes avec modules et interconnexions à différents niveaux d'abstraction, et avec APIs et protocoles de communication différents. Autrement dit, nous nous intéressons à la cosimulation multi-niveaux. Pour générer des modèles de cosimulation prenant en compte d'autres aspects d'hétérogénéité, nous utilisons des approches déjà proposées.

Dans un premier temps, nous présentons l'état de l'art de la cosimulation, puis nous expliquons un modèle conceptuel de cosimulation pour les systèmes hétérogènes embarqués, ensuite nous évaluons les modèles proposés dans la littérature pour adapter la communication entre modules ayant différents niveaux d'abstraction et API/protocoles, et enfin nous présentons les différents modèles de cosimulation au long d'un flot de validation.

2.4.1 Etat de l'art de la cosimulation

Plusieurs travaux ont été proposés pour faciliter la validation des systèmes hétérogènes embarqués par cosimulation. Dans ce travail, nous classifions les approches selon l'aspect

d'hétérogénéité qu'ils ciblent. Ainsi, ils peuvent être divisés en trois catégories : les approches multi modèles de calcul, les approches multi langages et les approches multi-niveaux.

Cosimulation multi modèles de calcul

Certains travaux focalisent la manière de simuler des systèmes qui possèdent des modules utilisant différents modèles de calcul. Ces travaux sont spécialement utiles pour les systèmes qui contiennent des technologies mixtes (traitement de signal, dispositifs électroniques, mécaniques, optiques, etc.). L'approche plus représentative de cette direction est le projet de recherche Ptolemy [Pto02]. Dans le cadre de ce projet, ils ont fait une étude approfondie sur les modèles de calcul [Lee97] et ont développé un environnement de cosimulation implémenté en Java appelé PtolemyII [Eke01]. Ptolemy II permet la cosimulation des systèmes intégrant plusieurs modèles de calcul comme : machine d'états finis (FSM), événements discret (DE), processus séquentiels communicants, graphe de flot de données (DFG), etc. Chaque modèle de calcul dans PtolemyII est appelé domaine. Cet environnement permet également la cosimulation géographiquement distribuée en utilisant les protocoles de communication fournis par Java RMI et CORBA.

Cosimulation multi langages

Le but de plusieurs travaux est de cosimuler des systèmes ayant des modules décrits en différents langages de spécification. Ces travaux s'intéressent, en effet, à la communication entre les différents simulateurs impliqués dans la cosimulation. La plupart de ces approches exploitent les bibliothèques fournies par les différents simulateurs en vue de la communication avec l'extérieur.

L'outil de cosimulation VCI [Val94] permet la cosimulation des langages C et VHDL à travers de la génération automatique des interfaces de cosimulation (dans ce cas, les adaptateurs de simulateurs qui seront décrits dans la section 2.4.2) à partir d'un fichier de configuration. Ces interfaces synchronisent la communication entre les différents simulateurs en utilisant le mécanisme de communication inter processus (IPC) Unix. En outre, VCI connecte ces interfaces avec le reste du système pour former un modèle de cosimulation du système.

Lemarrec et al [Lem00] ont étendu le travail antérieur par l'intégration de MATLAB/Simulink et SDL/ObjectGeode. Hessel [Hes00] a repris ces travaux pour permettre la cosimulation multi langage géographiquement distribuée. Il réalise la communication entre

les différents simulateurs à travers du mécanisme de « sockets ». Ces travaux ont donné naissance à l'environnement de cosimulation appelé MCI.

Héneault et al [Hen01] propose une approche multi langage, où les modules peuvent être décrits en VHDL, en Verilog et en langages basés sur C et C++. Cette solution utilise l'interface FLI du simulateur ModelSim [Mod01]. Les modules spécifiés dans les langages différents sont exécutés sur un seul simulateur (ModelSim), ce qui permet la réduction du surcoût de synchronisation entre les différents simulateurs.

Enfin, aujourd'hui la plupart des environnements de conception utilisent les approches présentées ici pour permettre la cosimulation multi langage. Les exemples de tels environnements incluent : Mentor Graphics Seamless [Men04], CowareN2C [Cow02], Synopsys SystemStudio [Syn04], etc.

Cosimulation multi-niveaux

Dans ces dernières années, plusieurs travaux sont focalisés sur la cosimulation multi-niveaux. Ils s'intéressent particulièrement à l'adaptation de la communication entre modules qui possèdent des interfaces décrites à différents niveaux d'abstraction et qui utilisent différents APIs et/ou protocoles de communication. Les travaux cités ci-dessous seront discutés en détails dans la section 2.4.4. Dans cette section, nous donnerons juste un aperçu.

Coware N2C [Cow02] est une environnement de simulation et conception des systèmes hétérogènes embarqués. Il permet la cosimulation de modules spécifiés en différents langages (par exemple : VHDL et C) qui doivent être encapsulés par des entités (enveloppes) qui adaptent aussi la communication. Ces entités doivent être décrits dans un langage propre à Coware appelé CowareC (langage basé sur le C). Plusieurs niveaux d'abstraction sont supportés par N2C.

IBM Coral [Ber00] est un environnement qui permet de simuler et synthétiser des systèmes à partir de descriptions du système dit « virtuelles ». Il fait l'adaptation au protocole du bus IBM CoreConnect [Ibm04] des modules utilisant diffèrent protocoles de communication.

ARM MaxSim [Arm04] est un environnement qui permet de modéliser et de simuler des systèmes hétérogènes embarqués. Il permet la cosimulation des modules spécifiés aux niveaux TLM et RTL. Les modules spécifiés au niveau RTL doivent utiliser le protocole AMBA [Arm02].

ST-Microeletronics MultiFlex [Pau04] est un environnement de conception et simulation de systèmes hétérogènes embarqués. Il permet de concevoir un système en

utilisant StepNP [Pau02] comme plateforme d'implémentation. MultiFlex permet la cosimulation des modules spécifies à différents niveaux d'abstraction et utilisant différents protocoles de communication. Les modules au niveau RTL doivent utiliser soit le protocole OCP [Ocp01], soit le protocole du bus STBus.

Hoffman et al. [Hof01] proposent un environnement de simulation permettant la cosimulation des processeurs spécifiés en LISA et des modules matériels décrits en SystemC ou VHDL. Les processeurs et modules matériels peuvent communiquer à différents niveaux d'abstraction, puisque l'environnement fournit une bibliothèque contenant des composants pour adapter la communication et les différents simulateurs. L'environnement connecte ces composants aux modules, en formant ainsi un modèle de cosimulation.

COSY [Bru00] permet la cosimulation multi-niveaux en connectant des composants existants dans une bibliothèque pour adapter la communication.

Abdi et al [Abd03] proposent un environnement permettant de générer l'adaptation d'un protocole de communication propre à l'environnement à plusieurs autres.

Enfin, CosimX [Nic02] génère l'adaptation de la communication permettant la cosimulation multi-niveaux. En outre, CosimX génère l'adaptation de différents simulateurs, ce qui concède un aspect multi langage à l'approche. La génération des interfaces de cosimulation est faite à partir d'une spécification où le système est présenté comme une architecture abstraite (pour la définition d'architecture abstraite voir chapitre 3). En effet, notre travail peut être vue comme une extension de CosimX. Nous proposons un nouveau flot de génération et un nouveau modèle pour adapter la communication.

2.4.2 Modèle conceptuel de cosimulation pour systèmes hétérogènes embarqués

La validation par cosimulation nécessite un modèle de cosimulation du système. Ce modèle de cosimulation doit être assez générique pour représenter n'importe quel système hétérogène. Le modèle de cosimulation que nous allons générer automatiquement suit l'approche classique proposée par Rowson [Row94]. Le modèle est composé de trois parties principales comme le montre la Figure 2-10 : les simulateurs exécutant le comportement des modules, les interfaces de cosimulation et le bus de cosimulation. Dans ce schéma, un module s'exécute sur un simulateur et utilise l'interface de cosimulation pour passer les données au bus de cosimulation qui va les transmettre au reste du système. Ces trois parties représentent le module, son interface de communication et les interconnexions respectivement.

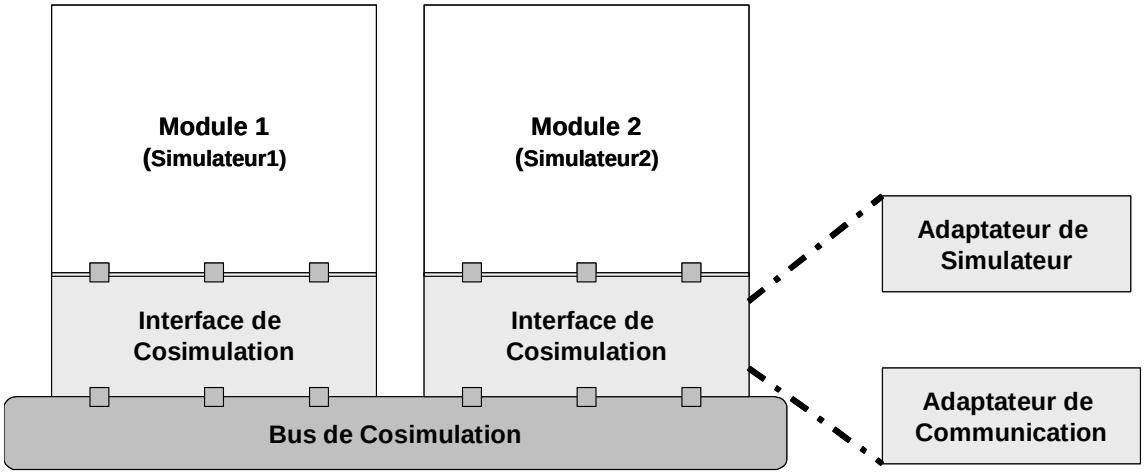


Figure 2-10 Modèle de Cosimulation pour Systèmes Hétérogènes Embarqués

Bus de cosimulation

Le bus de cosimulation est responsable de l'interprétation des interconnexions du système. Il doit coordonner les échanges de données entre les modules. Le bus de cosimulation peut réaliser plusieurs fonctionnalités tels que : la routage de données, la synchronisation entre les modules, l'arbitrage pour l'accès au bus, etc. Comme le bus de cosimulation représente le modèle de simulation des interconnexions, nous allons supposer dans notre approche qu'il peut avoir plusieurs niveaux d'abstraction. En effet, le niveau d'abstraction du bus dicte les fonctionnalités qu'il fournira.

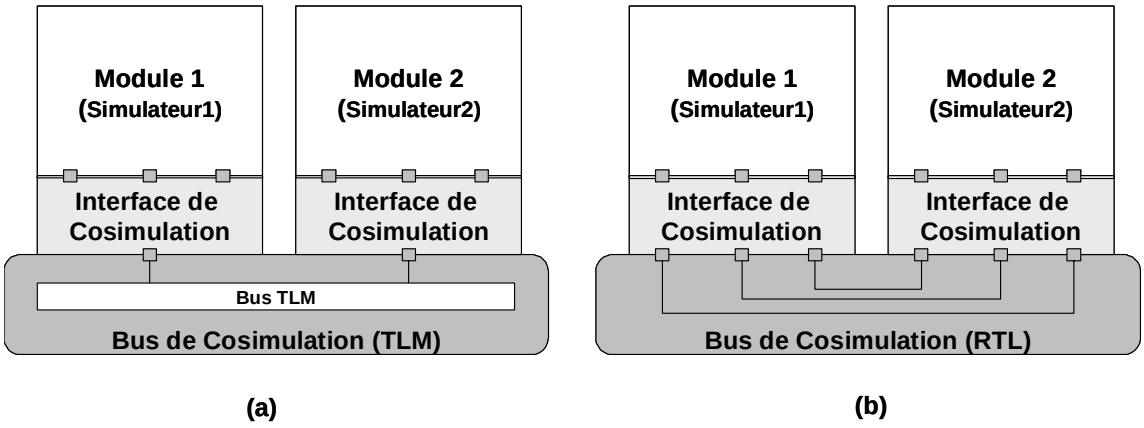


Figure 2-11 (a) Bus de Cosimulation au Niveau Transactionnel (b) Bus de Cosimulation au Niveau RTL

La Figure 2-11 présente des exemples de bus de cosimulation à différents niveaux d'abstraction. Si le bus est au niveau RTL, il est seulement un ensemble de signaux RTL qui

transportent les bits. Tandis que si le bus est au niveau TLM, il va réaliser les mêmes fonctions qu'un canal à ce niveau, c'est-à-dire le routage de données, la synchronisation, etc.

Interfaces de cosimulation

Les interfaces de cosimulation ont deux fonctions bien distinctes : (1) adapter les différentes environnements d'exécution, c'est-à-dire les différents simulateurs ; (2) adapter la communication entre les modules en terme de niveaux d'abstraction, d'API et protocole de communication et de type de donnée. Dans notre approche, les interfaces de cosimulation sont composées de deux parties comme illustre la Figure 2-10 **: l'adaptateur de simulateur** et **l'adaptateur de communication.** Comme nous verrons dans le chapitre 3, une des nos contributions est apporté au niveau de l'adaptateur de communication.

L'avantage de séparer ces deux fonctionnalités est la flexibilité, car un changement de simulateur ou de niveau d'abstraction ne requiert qu'un changement soit dans l'adaptateur de simulateur, soit dans l'adaptateur de communication.

2.4.3 Critères d'évaluation de modèles d'adaptateurs de communication

Dans ce travail, nous nous concentrons sur l'implémentation des adaptateurs de communication, même si nous sommes capables de générer des adaptateurs de simulateur comme le montrera le chapitre 4. Une des contributions de ce travail consiste à définir un modèle (présenté dans le chapitre 3) pour construire des adaptateurs de communication. Mais avant cela, il faut fixer les critères sur lesquels nous allons nous baser pour définir un tel modèle. Ces critères servent pour évaluer non seulement notre approche mais aussi celles proposées par d'autres travaux.

Flexibilité

Un modèle d'adaptateur de communication doit être flexible. La flexibilité dont nous parlons ici concerne :

- l'indépendance du modèle par rapport à des interfaces standard – le modèle doit faciliter l'adaptation d'APIs/protocoles de communication différents, même si les APIs/protocoles ne sont pas standard;
- la facilité de modification de l'adaptateur de communication – l'utilisation du modèle doit apporter une réduction de l'effort de la part du concepteur si celui-ci doit modifier l'adaptateur à cause d'un changement du système ;

L'indépendance des interfaces standard rend le modèle plus général, c'est-à-dire, les adaptateurs de communication implémentés en utilisant ce modèle ont un domaine d'application plus vaste. Cela signifie que le modèle ne doit poser aucune contrainte par rapport aux APIs/protocoles de communication employés par le module ou par l'environnement d'exécution.

La facilité de modification de l'adaptateur de communication permet une exploration d'architecture du système plus significative. Lors de la conception d'un système, il y a une étape d'exploration d'architecture, où le concepteur change certaines parties du système pour analyser les différentes solutions architecturales. Alors, ces changements peuvent impliquer des modifications des adaptateurs de communication. Si le modèle pour implémenter les adaptateurs de communication réduit l'effort de modification de ceux-ci, cela signifie que le modèle facilite l'exploration d'architecture du système.

Réutilisation des composants de base

Un adaptateur de communication est implémenté par un code exécutable. Ce code peut être réparti en plusieurs composants de base qui sont rassemblés lors de la construction de l'adaptateur. Un modèle d'adaptateur de communication doit faciliter la réutilisation des composants de base.

La définition de la granularité des composants de base d'un modèle d'adaptateur de communication est fondamentale. Un composant trop grand risque de n'être pas réutilisable, tandis qu'un trop petit risque d'avoir une applicabilité qui n'est pas importante.

Enfin la réutilisation des composants de base peut aussi influencer l'étape d'exploration architecturale d'un système. Si le modèle ne facilite pas la réutilisation des composants de base, un changement du système implique un effort plus important de la part du concepteur pour ré-implémenter les adaptateurs de communication.

Performance

Un modèle d'adaptateur de communication doit favoriser l'implémentation d'adaptateurs performants. Ceci peut être difficile d'achever parce que cela dépend de la qualité d'implémentation. Toutefois, le modèle peut jouer un rôle important en imposant que seulement les fonctionnalités essentielles pour faire l'adaptation soient incluses dans les adaptateurs.

Encore une fois, la granularité des composants de base du modèle est essentielle. Si le composant est trop grand, nous risquons d'avoir des fonctionnalités qui ne sont pas nécessaires pour faire l'adaptation.

2.4.4 Etat de l'art - modèles d'adaptateurs de communication

Dans la littérature, nous pouvons trouver plusieurs travaux concernant les modèles d'adaptateur de communication. Les modèles proposés dans ces travaux peuvent être divisés en trois catégories : modèles basés sur bus, modèles basés sur protocoles standard et modèles basés sur l'assemblage de composants avec architecture fixe. Par la suite, nous analysons les travaux existants dans chaque catégorie. Les termes protocole et API de communication seront utilisés indifféremment ici, même s'ils n'ont pas exactement le même signification.

Modèles basés sur bus

Ce type de modèle s'appuie sur l'idée que la communication du système se fait en utilisant un protocole fixe d'un bus. Ainsi, l'adaptation de communication doit se faire en convertissant le protocole des modules en celui du bus. Parmi les modèles basés sur bus, nous pouvons citer ceux utilisés par IBM Coral [Ber00] et ARM MaxSim [Arm04].

Le mode de fonctionnement de Coral est le suivant : les modules doivent être décrits en utilisant un nombre limité de protocoles prédéfinis et Coral génère l'adaptation des protocoles des modules à celui du bus IBM CoreConnect [Ibm04]. Pour intégrer un nouveau module, le concepteur a deux choix : (1) récrire le module en utilisant les protocoles connus par Coral ; (2) écrire un adaptateur de communication et le connecter avec le module. Coral contient une bibliothèque avec quelques modules qui utilisent des protocoles prédéfinis pour que le concepteur puisse les utiliser autour du bus CoreConnect.

Avec ARM MaxSim, la communication entre les différents modules d'un système peut se faire de deux façons : soit les modules ont une interface compatible avec le protocole AMBA [Arm02] (utilisé par les bus ARM), soit les modules doivent avoir une interface TLM prédéfinie par MaxSim. Il fournit, également, une bibliothèque avec des modèles de différents types de modules (processeurs, mémoires, buses, périphériques, etc) ayant des interfaces compatibles. L'intégration d'un nouveau module passe par l'adaptation de l'API du module à TLM ou AMBA.

La limitation principale de ces approches est le manque de flexibilité. Les systèmes conçus en utilisant ces approches doivent employer les bus auxquels les adaptateurs de

communication sont prédéfinis. Ceci représente une importante contrainte pour la conception des systèmes, et surtout pour le domaine d'application de telles approches.

Modèles basés sur protocoles standard

L'idée de base de ce type de modèle est très similaire à celui des modèles basés sur bus. Cette fois la communication du système est faite en utilisant un protocole /API de communication standard. L'adaptation de communication se fait, alors, en convertissant les protocoles des modules en un protocole standard.

Parmi les protocoles standard existants, les plus utilisés sont : VCI [Vsi99] (Virtual Component Interface) défini par le consortium VSIA, et OCP [Ocp01] proposé par Sonics. Plusieurs travaux utilisent ces protocoles pour adapter la communication entre les modules d'un système.

ST-Microeletronics MultiFlex [Pau04] permet la spécification du système en utilisant deux types d'API de communication : DSOC (Distributed System Object Component), qui est similaire au concept d'objets distribués de CORBA, et SMP (Symetric Multi-Processing), qui est basé sur le modèle de communication de mémoire partagée. Après la spécification, le concepteur doit choisir les modules matériels ou logiciels qui implémentent le système. Ces modules existent dans une bibliothèque et ils ont une interface OCP. L'environnement génère d'abord l'adaptation des protocoles de haut niveau (DSOC et/ou SMP) à OCP et après l'adaptation du protocole standard OCP à celui du bus de la plateforme (STBus).

Sonics SonicsStudio [Son04] est un environnement qui permet la conception et la simulation des systèmes hétérogènes embarqués. Il permet la communication entre des modules ayant une interface OCP. La communication se fait par le bus Silicon Backplane µNetwork [Son04a]. L'environnement génère l'adaptation d'OCP au protocole utilisé par le bus. Si le concepteur veut intégrer un nouveau module, c'est a lui de faire l'adaptation de l'API du module à OCP.

La limitation de ces approches, encore une fois, est le manque de flexibilité. Ces approches supposent que les composants d'un système ont une interface standard, ce qui n'est pas toujours le cas. Ceci restreint l'utilisation de telles approches.

Modèles basés sur l'assemblage de composants avec architecture fixe

Plusieurs travaux adoptent le modèle basé sur l'assemblage des composants avec architecture fixe pour implémenter les adaptateurs de communication. Le principe de ce

modèle est d'avoir une bibliothèque de composants et de les assembler selon une architecture prédéfinie pour former un adaptateur de communication.

COSY [Bru00] permet l'adaptation des modules utilisant une API de communication basée sur les FIFOs employées par les Processus de Kahn [Kah74] à plusieurs APIs de communication. Cette approche fait l'adaptation en deux phases. D'abord, il fait l'adaptation de la FIFO au protocole standard VCI, et après il fait une adaptation de VCI au protocole de communication du système. COSY fournit une bibliothèque de composants qui font ces adaptations, et il les assemble pour finalement former un adaptateur de communication. Cette approche peut être appliquée dans un domaine d'applications plus grand que celles des modèles basés sur bus ou protocoles standard. Cependant, COSY a deux limitations : (1) restriction de l'API de communication des modules (FIFO) ; (2) adaptation de la FIFO au protocole du système de façon indirecte. Le premier point réduit la flexibilité de l'approche puisqu'il impose une API de communication de départ. Et le deuxième point pénalise la performance de l'adaptateur de communication, une fois que les conversions FIFO-VCI-protocole du système ajoutent un surcoût au processus d'adaptation.

Abdi et al. [Abd03] ont développé la génération d'adaptateurs de communication à partir d'une spécification de système qui utilise un protocole de communication à un niveau d'abstraction élevé. L'API de ce protocole est composée de deux primitives : Send et Receive. L'environnement génère l'adaptation de ce protocole à plusieurs autres. L'adaptateur de communication (qui est appelé Application Layer dans cette approche) est, en fait, un composant monolithique où le concepteur décrit comment va se faire l'adaptation. La description de l'adaptateur est faite en un langage qui ressemble à celui de macro, où le concepteur peut optimiser la fonction d'adaptation. Le manque de flexibilité et la difficulté de réutilisation de ces adaptateurs sont les limitations de cette approche. Le manque de flexibilité est dû au protocole de départ qui est figé, et à la grosse granularité des composants de l'adaptateur (en effet, il n'y a qu'un composant : l'adaptateur lui-même).

CosimX [Nic02] génère des adaptateurs de communication à partir d'une architecture abstraite du système. Il ne pose aucune contrainte aux APIs de communication des modules. La Figure 2-12 présente l'architecture des adaptateurs générés par CosimX. L'adaptation est faite par deux composants : (1) l'adaptateur de module (MA) qui est dépendant du protocole du module ; (2) l'adaptateur de canal (CA) qui est dépendant du protocole du canal de communication. Ces deux composants sont interconnectés par un bus interne (IB). L'adaptation est faite par le schéma suivant : (1) le MA convertit le protocole du module en celui de l'IB ; (2) le CA convertit le protocole de l'IB en celui du canal de communication. Ce

schéma est, en effet, similaire à celui utilisé par COSY, sauf que CosimX ne restreint pas le protocole du module. L'avantage de cette approche est le gain d'une certaine flexibilité. Le changement du protocole du module (ou du canal) n'implique que de changer une partie de l'adaptateur (le MA ou le CA). Pourtant, cette approche pénalise la performance puisqu'elle, comme COSY, fait la conversion du protocole du module en un protocole intermédiaire. Autre point à remarquer, c'est que la granularité des composants de base est encore trop grosse. Si par exemple, le protocole du module change, et l'adaptation requiert presque les mêmes fonctionnalités qu'avant, il faut créer un nouveau MA. Cela réduit la flexibilité et la capacité de réutilisation des composants de cette approche.

Hoffman et al. [Hof01] proposent un adaptateur de communication avec une architecture très semblable à celle de CosimX. L'adaptateur est constitué de deux composants: le bitmapping layer et le protocol layer. Le premier interprète le protocole du module et traduit les données du module en une chaîne de bits. Le deuxième convertit la chaîne de bits en un type de données compatibles avec le protocole du canal et fait l'adaptation de communication nécessaire. Cette approche, en étant similaire à CosimX, en possède les mêmes limitations.

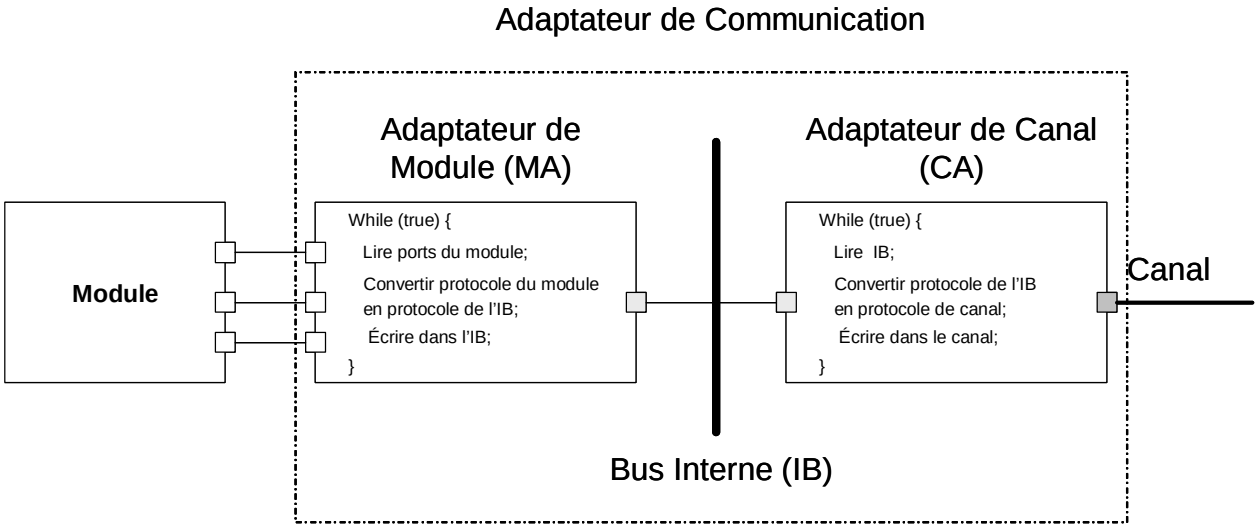


Figure 2-12 Modèle d'Adaptateur de Communication Utilisé Par CosimX

Coware N2C [Cow02] génère des adaptateurs de communication en partant d'une spécification abstraite du système. Chaque module d'un système peut être encapsulé par plusieurs enveloppes abstraites. Ces enveloppes correspondent aux adaptateurs de communication. Lors de la simulation, l'environnement choisit les enveloppes compatibles pour permettre la communication du système. Pour chaque type d'adaptation de

communication, le concepteur doit implémenter une enveloppe. Autrement dit, en cas de changement de protocole d'un module, le concepteur doit créer une nouvelle enveloppe, sauf si celui-ci existe déjà dans la bibliothèque gérée par N2C. Autre point important à remarquer dans cette approche, c'est le modèle de communication utilisé par les enveloppes. Avant, les enveloppes utilisaient le modèle maître-esclave de communication. Chaque enveloppe adaptaient l'interface du module à une interface maître ou (exclusive) esclave. Plus récemment, N2C a adopté une interface moins rigide pour la communication entre enveloppes. Cette nouvelle interface est basée sur le bus AMBA. Une limitation de cette approche est, évidemment, le manque de flexibilité dû à la rigidité des interfaces des enveloppes et à la granularité trop grosse des composants de base (dans ce cas, c'est l'adaptateur de communication entier). Cette granularité réduit également, la réutilisation de l'adaptateur de communication.

2.4.5 Modèles de cosimulation à travers un flot de validation de systèmes hétérogènes embarqués

La Figure 2-13 illustre trois modèles de cosimulation utilisés durant un flot de validation conceptuel. Au début, ce flot suppose que la spécification est composée des unités de calcul homogènes, ce qui signifie que le modèle de cosimulation est composé de modules exécutés sur un seul simulateur et un bus de cosimulation. Aucune interface de cosimulation n'est nécessaire si le système dans cette étape est homogène. Le bus de cosimulation est normalement complexe à cause du niveau d'abstraction qu'il représente (dans la Figure 2-13, il implémente le standard MPI).

Dans la prochaine étape de la validation, le système peut présenter des modules hétérogènes. Ainsi, il faut des adaptateurs de communication pour adapter les différents niveaux d'abstraction, APIs/protocoles de communication et type de données. Dans la Figure 2-13, nous voyons un adaptateur RTL-TLM pour adapter le module matériel RTL au bus de cosimulation qui est au niveau TLM. Des adaptateurs de simulateur peuvent être nécessaires dans le cas où plusieurs simulateurs sont présents à la cosimulation. Le bus de cosimulation est complexe puisqu'il est au niveau TLM.

La dernière étape du flot présente tous les modules raffinés au niveau RTL. Pour les modules logiciels, la validation utilise des simulateurs de processeurs (ISS – Instruction Set Simulator), tandis que pour les modules matériels, les simulateurs matériels sont employés. Cela signifie que des adaptateurs de simulateurs sont nécessaires. Souvent, nous les utilisons

pour adapter les ISS au simulateur matériel. Les adaptateurs d'ISS sont souvent appelés « BFM » (venant de l'anglais Bus Functional Model). Comme tout le système est au niveau RTL, rarement nous avons besoin d'un adaptateur de communication. A ce point-là, les adaptations de communication sont réalisées par les interfaces logicielles /matérielles conçues au long du flot. Le bus de cosimulation est très simple, car il représente les signaux RTL. Ainsi, il ne fait que transporter les bits.

Nous allons voir dans les prochains chapitres comment nous arrivons à générer ces modèles de simulation à partir d'une spécification du système.

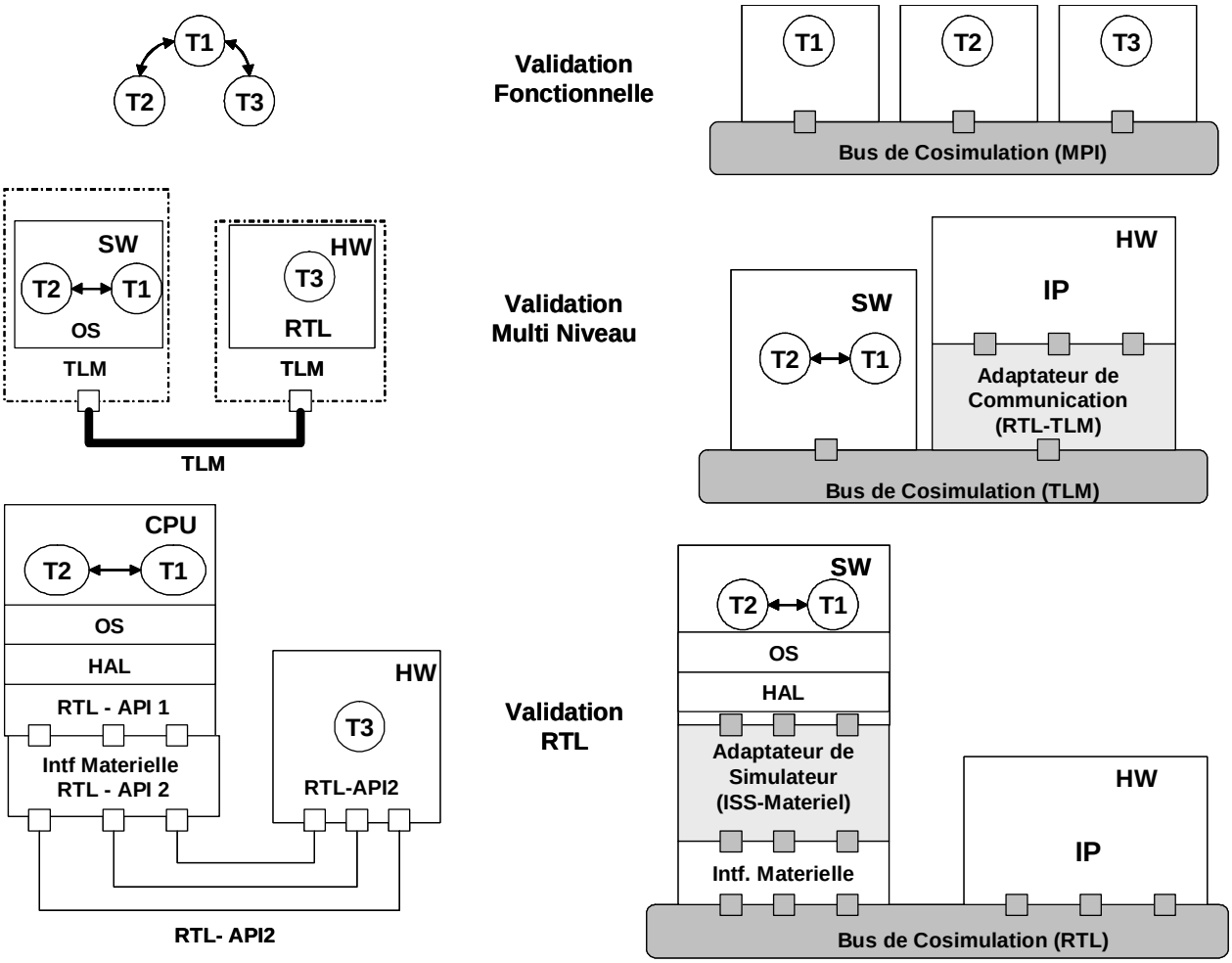


Figure 2-13 Modèles de Simulation à travers d'un Flot de Validation de Systèmes Hétérogènes Embarqués

2.5 Conclusion

Nous avons vu dans ce chapitre les concepts de base des systèmes et de la conception d'un système hétérogène embarqué. Les caractéristiques d'un tel système rendent difficile sa

validation et par conséquent sa conception. Nous avons vu également que la validation doit être faite à toutes les étapes de conception pour réduire le coût financier, le temps et l'effort de la part du concepteur.

Trois méthodes de validation ont été présentées dans ce chapitre. Nous avons montré que la validation par cosimulation est actuellement l'option la plus avantageuse. Cependant, la meilleure approche est encore de combiner toutes les méthodes pour obtenir une qualité plus significative de validation.

Nous avons cité plusieurs travaux qui ciblent la validation par cosimulation. Notre contribution est relative à la cosimulation multi-niveaux. Nous avons fait une étude sur les modèles d'adaptateurs de communication proposés par plusieurs travaux. Les avantages et les faiblesses de chaque approche ont été discutés en prenant en compte les critères d'évaluation que nous avons fixés. Nous allons proposer dans ce travail un modèle d'adaptateur de communication prenant en compte ces critères. Dans le travail que nous proposons, nous allons générer, également, des modèles de simulation automatiquement partant d'une spécification abstraite du système. Ce modèle de simulation est très générique et peut être appliqué à tout le flot de validation.

Chapitre 3 : Modèle d'adaptateur de communication pour la cosimulation de systèmes hétérogènes embarqués

3.1 Introduction

Dans les chapitres précédents, nous avons discuté les difficultés liées à la conception et la validation des systèmes hétérogènes embarqués. Nous avons vu également que la cosimulation est une méthode de validation qui possède certains avantages par rapport à d'autres méthodes. Cette méthode requiert la construction d'un modèle de simulation du système. Ce modèle est composé des différentes parties dont le bus de cosimulation, l'adaptateur de communication et l'adaptateur de simulateur. Cependant, la construction manuelle de chaque partie de ce modèle est un travail long et fastidieux. Le problème consiste, donc, à trouver un moyen de représenter la fonctionnalité de chaque partie du modèle, en vue d'automatiser leur construction. Autrement dit, il nous faut des modèles de représentation pour chaque composant du modèle de simulation. Tandis que plusieurs travaux ont apporté des solutions pour représenter le bus de cosimulation et l'adaptateur de simulateur [Val94] [Lem00] [Hen01], l'adaptateur de communication reste encore un défi. Les modèles d'adaptateur de communication proposés dans de nombreux travaux présentent des limitations qui restreignent leurs utilisations.

Dans ce contexte, ce chapitre présente une des contributions de ce travail. Il propose un modèle d'adaptateur de communication qui surmonte les limitations des autres modèles proposés dans la littérature. Ce modèle est basé sur les services requis pour implémenter la fonctionnalité de l'adaptateur. L'adaptateur de communication sera construit en partant d'une spécification du système cible qui contient la description des services requis pour l'adaptation. Comme nous le verrons, cette spécification représente, ce que nous appelons, l'architecture abstraite du système.

Ce chapitre est organisé comme suit. La section 3.2 explique les concepts d'architecture abstraite, leurs applications dans un flot de conception de systèmes hétérogènes embarqués, ainsi que le lien entre les architectures abstraites et les modèles de simulation. La section 3.3 explique les concepts du modèle basé sur les services, ainsi que les règles de construction des adaptateurs de communication à partir de celui-ci. La section 3.4 montre un exemple d'application de ce modèle d'adaptateur de communication sur un système hétérogène embarqué, nommé WSS (Windows Searching System). La section 3.5 présente les langages utilisés dans ce travail pour décrire des architectures abstraites et les composants du modèle basé sur les services. Enfin, la section 3.6 explique la façon dont le modèle d'adaptateur de communication basé sur les services peut être appliqué dans le flot de conception de systèmes hétérogènes embarqués ROSES. Ce modèle peut être employé

comme format standard pour représenter non seulement les adaptateurs de communication mais aussi les interfaces logicielles/matérielles.

3.2 Utilisation d'architectures abstraites pour la spécification des systèmes hétérogènes embarqués

Lors de la conception d'un système embarqué, le concepteur doit faire face à différents types d'hétérogénéité. Les types d'hétérogénéité peuvent varier selon l'étape de conception. Dans ce contexte, l'approche la plus efficace consiste à séparer le comportement et la communication du module [San02]. Cela facilite la conception parallèle des différents modules et permet que le raffinement du comportement d'un module soit indépendant de celui de la communication du système. Dans ce travail, nous utilisons pour spécifier les systèmes hétérogènes embarqués un modèle d'**architecture abstraite** [Ces02] qui possède les concepts permettant de faire un tel type de séparation. Comme nous le verrons plus tard, les concepts fournis par un tel modèle sont essentiels pour la génération automatique des différentes parties du modèle de simulation du système.

Ainsi, cette section est consacrée à expliquer tout d'abord les concepts de base des architectures abstraites, leurs utilisations dans un flot de conception de systèmes hétérogènes embarqués et finalement leurs liens avec les modèles de simulation .

3.2.1 Concepts de base des architectures abstraites

L'idée principale derrière une architecture abstraite est de faire l'abstraction des interfaces de communication des modules et des interconnexions du système. Autrement dit, nous faisons l'abstraction de certains détails qui concernent la communication. Un système est, ainsi, un ensemble de modules qui communiquent par des interfaces et des interconnexions abstraites. La puissance de ce modèle réside dans le pouvoir d'abstraction de l'hétérogénéité des composants. Il supporte l'hétérogénéité (différents niveaux d'abstraction, différents composants, différents langages) des composants en cachant certains détails d'implémentation et permet de retarder certaines décisions architecturales aux étapes plus avancées du flot de conception [Kri05]. En fait, cette idée n'est pas nouvelle et a été déjà proposée par la communauté logicielle dans plusieurs travaux [Cor03] [Jav04]. Pourtant, ces travaux sont plus adaptés au domaine logiciel. Ils ne prennent pas en compte, par exemple, que la communication entre deux modules peut se faire en utilisant de fils physiques au lieu d'appels de fonctions comme dans le cas de la communication entre deux modules logiciels.

Ainsi, la communauté matérielle, par le moyen de plusieurs travaux [Sys00] [Cow02], a adopté ce modèle mais avec quelques extensions en vue de le rendre plus général pour représenter des systèmes hétérogènes embarqués. Ici, nous utilisons les idées proposées par [Ces02].

La Figure 3-1a présente une architecture abstraite. Elle s'appuie sur deux concepts de base: interface abstraite et environnement d'exécution. Dans notre cas, une architecture abstraite consiste en un ensemble des modules avec des interfaces abstraites qui communiquent en utilisant un environnement d'exécution.

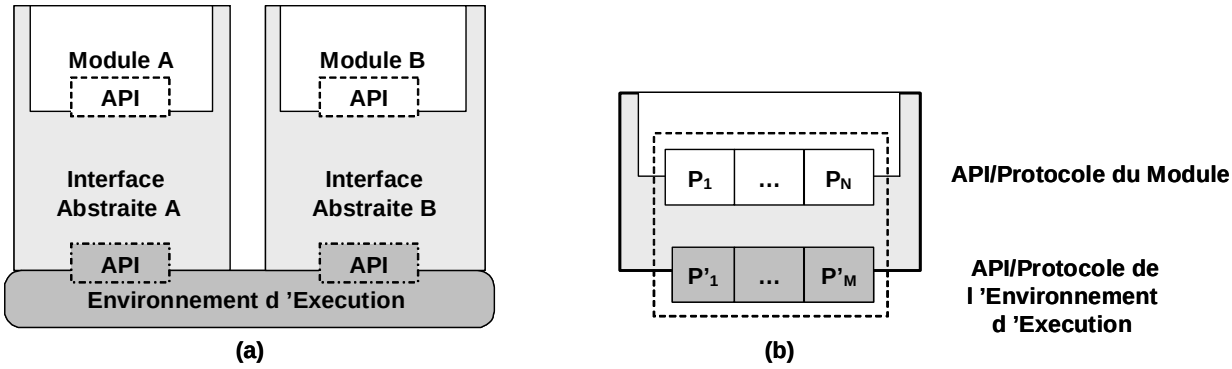


Figure 3-1 (a) Architecture Abstraite (b) Interface Abstraite

Environnement d'exécution

L'environnement d'exécution correspond à une abstraction des interconnexions du système. Autrement dit, il modélise la communication du système. Ainsi, il peut être un bus, un réseau de communication sur puce, une plateforme logicielle (ex. MPICH [Mpi04a], CORBA [Cor03]), ou comme nous le verrons un bus de cosimulation.

Interfaces abstraites

Comme chaque module et l'environnement d'exécution peuvent posséder leurs propres APIs/protocoles de communication, la communication est possible grâce aux interfaces abstraites de chaque module. Ces interfaces adaptent les différents APIs/protocoles de communication.

La Figure 3-1b montre qu'une interface abstraite peut être vue comme un port hiérarchique qui est divisé en deux parties : (1) les ports relatifs à l'interface du module (ports P_1.. .P_N) ; (2) les ports relatifs à l'interface de l'environnement d'exécution (ports P'_1.. .P'_M). Une interface abstraite nous permet de spécifier un ensemble des services requis/fournis par le module/l'environnement d'exécution. Ces services peuvent être relatifs à la synchronisation,

le protocole de communication, la conversion de donnés, les entrées/sorties, l'arbitration, l'allocation de mémoire, etc.

Il faut remarquer qu'en abstrayant les interfaces de communication et les interconnexions, aucune contrainte n'est imposée par rapport à leurs implémentations. Cela veut dire que selon le besoin, les interfaces abstraites et l'environnement d'exécution peuvent être implémentés différemment. Une interface abstraite, par exemple, peut être implémentée par une interface matérielle dans un outil de synthèse matériel, par une interface logicielle dans un outil de synthèse logiciel, ou encore par un adaptateur de communication dans le cas de notre travail. Ainsi, dans la section 3.3, nous proposons un modèle d'adaptateur de communication basé sur les services pour représenter les interfaces abstraites.

3.2.2 Architectures abstraites dans un flot de conception de systèmes hétérogènes

Les architectures abstraites permettent la représentation de différentes étapes du flot de conception. Ainsi, les modèles utilisés dans le flot de conception (présenté par la Figure 2-6) peuvent être représentés comme indiqué par la Figure 3-2.

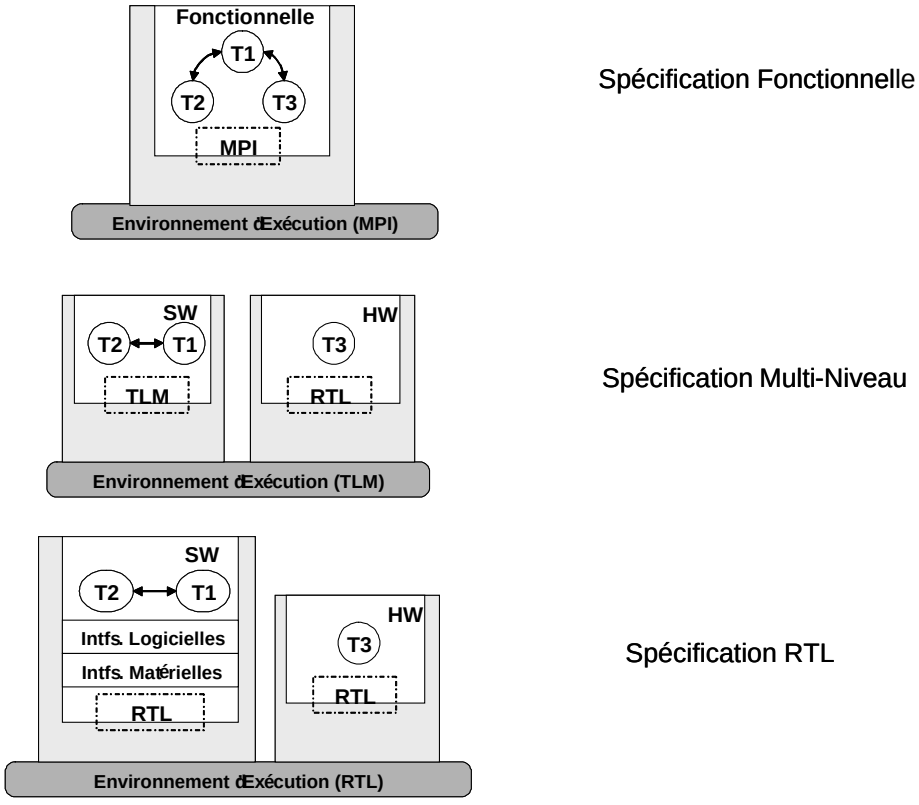


Figure 3-2 Les Architectures Abstraites à Travers un Flot de Conception

La complexité d'implémenter les interfaces abstraites et l'environnement d'exécution varie selon l'étape de conception. Dans le flot de conception simplifié que nous présentons, par exemple, tout au début, il n'y a pas d'adaptation à faire puisque le module et l'environnement d'exécution ont la même API et le même protocole de communication. Par contre, l'implémentation de l'environnement d'exécution comme un canal fonctionnel MPI peut être difficile. Ensuite, après le partitionnement logiciel/matériel, le système devient plus hétérogène. Alors, les interfaces abstraites ont un comportement plus complexe. Dans notre cas, ils peuvent modéliser les interfaces logicielles/matérielles et/ou les adaptateurs de communication et de simulateur (dans le cas de la validation par cosimulation). Dans cette étape, si l'environnement d'exécution reste dans un niveau d'abstraction élevé tel que le TLM, son implémentation peut être toujours complexe. La dernière étape de ce flot présente toutes les interfaces logicielles/matérielles générées et l'environnement d'exécution au niveau RTL. A ce niveau, les interfaces abstraites peuvent modéliser des adaptations de différents simulateurs dans le cas de la validation par cosimulation. L'environnement d'exécution modélise les fils physiques ou bien une IP matérielle implémentant le réseau de communication du système.

3.2.3 La relation entre les architectures abstraites et les modèles de simulation

La Figure 3-3 présente la correspondance entre les deux concepts principaux des architectures abstraites (l'interface abstraite et l'environnement d'exécution) et le modèle de simulation que nous utilisons dans ce travail.

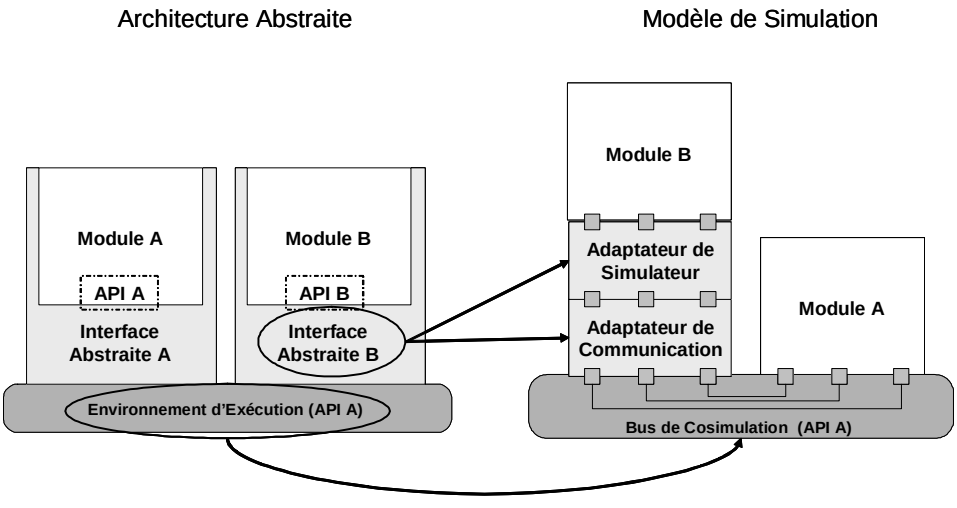


Figure 3-3 Architecture Abstraite Versus Modèle de Simulation

Modèle de simulation pour l'environnement d'exécution – Le bus de cosimulation

L'environnement d'exécution est une abstraction des interconnexions, alors que le bus de cosimulation est le modèle de simulation des interconnexions. Ceci étant, la correspondance entre l'environnement d'exécution et le bus de cosimulation est directe comme le montre la Figure 3-3. Le bus de cosimulation représente, donc, le modèle de simulation de l'environnement d'exécution.

Modèle de simulation pour les interfaces abstraites – Les adaptateurs de communication et de simulateur

Le rôle d'une interface abstraite est d'adapter le module à l'environnement d'exécution. Ceci correspond exactement aux fonctions effectuées par les adaptateurs de communication et de simulateur. L'adaptateur de communication est chargé d'adapter les différents APIs/protocoles de communication, tandis que l'adaptateur de simulateur est chargé d'adapter l'environnement d'exécution d'un module à l'environnement d'exécution du reste du système. Ceci étant, ces adaptateurs représentent le modèle de simulation des interfaces abstraites. Autrement dit, ces adaptateurs correspondent, effectivement, à une réalisation spécifique des interfaces abstraites lors de la validation du système par cosimulation. Il faut remarquer que l'utilisation des adaptateurs de communication pour réaliser les interfaces abstraites dépend, évidement, de l'étape de conception du système.

3.3 Modèle d'adaptateur de communication basé sur les services

Dans cette section, nous proposons un modèle d'adaptateur de communication basé sur les services. Ce modèle d'adaptateur est basé sur l'assemblage de composants mais avec une architecture flexible. Les autres approches d'assemblage de composants que nous avons discutés dans le chapitre 2, proposaient des adaptateurs avec une structure fixe (par exemple le MA et le CA utilisés par CosimX). Alors que avec le modèle proposé ici, l'architecture de l'adaptateur peut varier selon les besoins d'adaptation. Avec ce modèle, nous espérons gagner, en plus de la flexibilité, la capacité de réutilisation de composants de base et peut être améliorer les performances par rapport aux autres approches déjà discutés.

En fait, le modèle basé sur les services n'est pas un concept nouveau. Zitterbart et al [Zit93] a proposé un tel modèle pour implémenter le logiciel embarqué pour des applications de télécommunication. L'outil ASOG du groupe SLS [Gau01] génère des interfaces logicielles en utilisant un modèle basé sur les services. Pourtant, ces travaux sont plus adaptés

pour le domaine logiciel. Ils ne prennent pas en compte certains concepts du domaine matériel (comme des ports par exemple). Ceci étant, le modèle que nous proposons est, en réalité, une extension des modèles de ces deux travaux pour les rendre plus généraux. Le modèle présenté ici sera utilisé dans ce travail pour générer des adaptateurs de communication pour que nous puissions simuler des systèmes hétérogènes embarqués.

Pour mieux expliquer le modèle d'adaptateur de communication basé sur les services que nous proposons, nous présenterons d'abord les composants de base du modèle et ensuite la façon dont ces composants de base sont assemblés pour réaliser un adaptateur de communication.

3.3.1 Composants du modèle d'adaptateur basé sur les services

Les composants de base du modèle d'adaptateur basé sur les services que nous proposons utilisent trois objets fondamentaux : service, élément d'interface et port logique. Un adaptateur est composé par un graphe contenant ces trois types d'objets liés par des relations de dépendance entre les services requis par un élément d'interface/port logique et les services fournis par un autre élément d'interface/port logique.

Service

Un service définit une fonction ou une primitive de communication. L'ensemble de services attachés à l'ensemble de ports d'un module constitue l'API de communication du module. Autrement dit, les services peuvent représenter les actions attachées aux ports (voir section 2.2 du chapitre 2). Un service peut aussi représenter la (les) fonction (s) réalisée(s) par un élément d'interface.

Elément d'interface

Chaque service peut être fourni par un ou plusieurs éléments d'interface. Un élément d'interface (ou simplement élément) est une unité abstraite qui fournit et/ou demande un ou plusieurs services. Les services sont implémentés par des éléments d'interface. Ceux-ci peuvent être des entités logicielles/fonctionnelles (comme des classes C++). Un adaptateur de communication peut être implémenté par un ensemble d'éléments d'interface.

Port logique

Chaque service peut être fourni également par un ou plusieurs ports logiques. Un port logique est une entité qui fournit et/ou demande un ou plusieurs services et qui fait partie de l'interface d'un module ou de l'environnement d'exécution. Il peut regrouper un ou plusieurs ports physiques. Le port logique peut être également vu comme une entité fonctionnelle (comme des ports SystemC) qui représente les ports d'un module à différents niveaux d'abstraction (port au niveau TLM, par exemple). Les actions associées aux ports logiques sont les services fournis par ceux-ci. Ainsi, les services peuvent être aussi implémentés par des ports logiques.

Point d'accès aux services

Un élément d'interface/port logique accède à (requiert) un service fourni par un autre élément d'interface au travers d'une entité qui s'appelle **point d'accès aux services** (SAP). De même, un élément d'interface fournit un service par l'intermédiaire d'un SAP. Ainsi, pour qu'un élément d'interface/port logique puisse requérir un service fourni par un autre élément, il faut une mise en correspondance entre le SAP du premier et celui du dernier. Cette mise en correspondance entre les SAPs représente un **lien** entre les objets (élément d'interface/port logique).

Point d'accès aux ports

Un élément d'interface/port logique accède à (requiert) un service fourni par un port logique à travers d'une entité qui s'appelle **point d'accès aux ports** (PAP). Un port logique fournit, également, un service par moyen d'un PAP. Comme avant, un élément d'interface /port logique peut utiliser un service fourni par un port logique s'il y a une correspondance entre le PAP du premier et celui du dernier. Cette mise en correspondance entre les PAPs représente aussi un **lien** entre les objets.

3.3.2 Composition des adaptateurs de communication

La composition des adaptateurs de communication se fait en utilisant les relations entre les différents composants de base présentés dans la section précédente. Ces relations peuvent être modélisés par un graphe de dépendances de services (GDS). Un adaptateur de communication est constitué des ports relatifs au module et des ports relatifs à l'environnement d'exécution reliés par un GDS.

Graphe de dépendance de services

La Figure 3-4 présente un GDS. Un noeud du graphe de dépendance de services peut être un port logique, un service ou un élément d'interface. Un arc dans le GDS définit la relation de dépendance entre un port logique ou un élément d'interface et un service. Une relation de dépendance existe quand un port logique ou un élément d'interface requiert ou fournit un service particulier au travers d'un SAP ou PAP. Dans l'exemple représenté sur la Figure 3-4, les ports du module ($P_1 \ldots P_N$) et de l'environnement d'exécution ($P'_1 \ldots P'_M$) sont regroupés dans les ports logiques PM et PEE respectivement. Alors, PM demande le service S1 (par le biais d'un SAP), S1 est fourni par l'élément E1 qui requiert le service S5. S5 est fourni par l'élément E2, qui demande le service S6 (par le biais d'un PAP) qui est fourni par le port PEE.

Il faut remarquer qu'un GDS représente la décomposition fonctionnelle de la fonction globale d'adaptation de communication. Les services dans un GDS représentent les fonctions nécessaires pour effectuer une telle adaptation. Tandis que les éléments d'interface et ports logiques représentent les composants de base qui vont implémenter ces services.

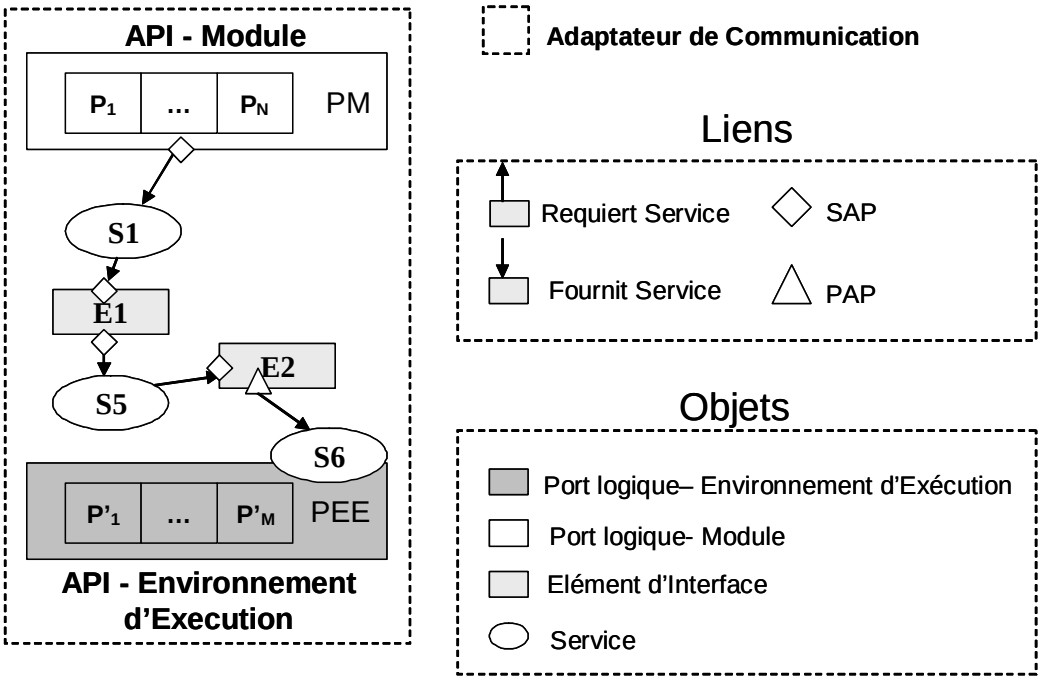


Figure 3-4 Graphe de Dépendance de Services

Ce modèle permet l'utilisation des composants de base fins, car ces composants représentent les fonctionnalités strictement nécessaires pour l'adaptation. La granularité fine

des services facilite l'implémentation d'adaptateurs de communication plus flexibles. Additionner/supprimer un fonctionnalité d'un adaptateur, par exemple, peut impliquer seulement l'addition/suppression d'un service. L'idée d'avoir le service comme composant de base facilite aussi sa réutilisation, puisque une même fonctionnalité peut être nécessaire à plusieurs adaptateurs de communication.

Un autre point à considérer est le fait que l'implémentation finale d'un adaptateur de communication doit fournir seulement les services demandés dans le GDS, même s'il y a des éléments d'interface/ports logiques dans le GDS qui fournissent d'autres services non nécessaires à la fonction d'adaptation. Cela est illustré par la Figure 3-5, où l'élément E1 fournit les services S1 et S4, mais l'adaptateur de communication présenté dans la Figure 3-4 ne contient que l'implémentation du premier service. Le GDS ne présente que les services nécessaires à l'adaptation. S'il y a d'autres services non nécessaires fournis par des éléments d'interface/ports logiques qui font partie d'un GDS, ils n'apparaissent pas dans le GDS. Ainsi, ce modèle permet la génération d'adaptateurs de communication plus performants, car ceux-ci n'implémentent que les services strictement nécessaires pour la fonction d'adaptation. D'autre part, si un élément requiert deux ou plusieurs services, ces services peuvent être exécutés en parallèle dans l'adaptateur. Ceci aide aussi la construction d'adaptateurs plus performants.

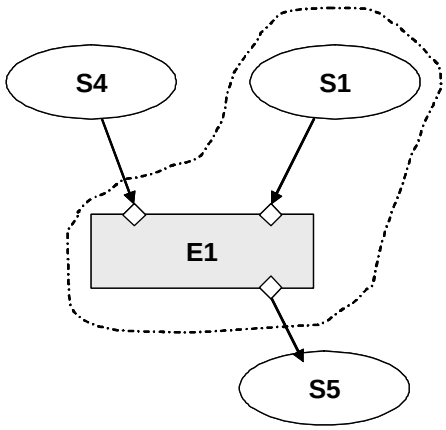


Figure 3-5 Implémentation des Services Strictement Nécessaires Pour l'Adaptation

Enfin, ce modèle nous permettra d'automatiser la construction des GDS représentant les adaptateurs de communication à partir d'une interface abstraite.

La construction des graphes de dépendance de services

La méthode utilisée pour construire un GDS représentant un adaptateur de communication peut influencer la qualité de ceci. La construction d'un GDS pour implémenter un adaptateur de communication peut se faire de différentes façons. Une possibilité est d'avoir une bibliothèque contenant le GDS maximal qui possède tous les éléments d'interface/ports logiques et leurs relations de dépendance. A partir de ce GDS maximal, la sélection des éléments d'interface/ports logiques s'effectue selon un critère prédéfini pour construire un GDS correspondant à l'adaptateur de communication. L'autre possibilité est de disposer d'une bibliothèque avec la description de chaque élément d'interface/port logique, puis de construire le GDS de façon incrémentale en rajoutant à chaque pas un élément d'interface/port logique selon un critère prédéfini. Chaque approche a des avantages et des faiblesses. La première permet d'avoir une meilleure compréhension de tous les composants de base, en facilitant ainsi, l'optimisation globale de l'adaptateur.

Cependant, gérer un GDS maximal est plus complexe. La deuxième approche facilite la gestion des composants de base, par contre la construction d'un GDS optimal peut être plus difficile. Dans ce travail, nous utilisons la première approche. Nous verrons dans le chapitre 4 que nous utilisons un outil pour créer un GDS maximal et après nous faisons la sélection automatique des éléments d'interface/ports logiques pour réaliser l'adaptateur de communication. Cet outil est capable également de traiter des cycles qu'un GDS peut posséder (cela arrive lorsqu'il y a des interdépendances entre les éléments d'interface/ports logiques) avec la technique de coloriage des nœuds du graphe.

La sélection des éléments d'interface/ports logiques qui font partie d'un adaptateur de communication peut avoir un impact important sur la performance globale de cet adaptateur. Effectivement, la seule condition qui doit être respecté lors de la sélection des éléments/ports logiques est que ceux-ci doivent adapter l'API du module à celui de l'environnement d'exécution. Ceci est illustré sur la Figure 3-6. Elle montre deux sélections valides et une non valide. Les deux premiers adaptateurs sont composés d'éléments d'interface différents ({E1, E3} et {E1, E2}), mais les sélections sont valides car elles réalisent l'adaptation de l'API du module à celui de l'environnement d'exécution. La dernière sélection ({E1, E4, P_1}) n'est pas valide parce qu'elle n'arrive pas à faire l'adaptation requise.

Le modèle d'adaptateur de communication basé sur les services laisse au concepteur (ou aux outils de génération des adaptateurs) le choix de la sélection des éléments d'interface/ports logiques qui composent les adaptateurs de communication. Ce choix peut être guidé par des contraintes de plusieurs natures : performance, langage d'implémentation

de l'environnement d'exécution, le type du module (logiciel ou matériel), etc. Le fait du modèle laisser ce choix libre apporte une flexibilité plus importante lors de l'implémentation des adaptateurs de communication, car celui-ci peut être réalisé de plusieurs façons. Cette flexibilité favorise l'exploration d'architecture d'un système.

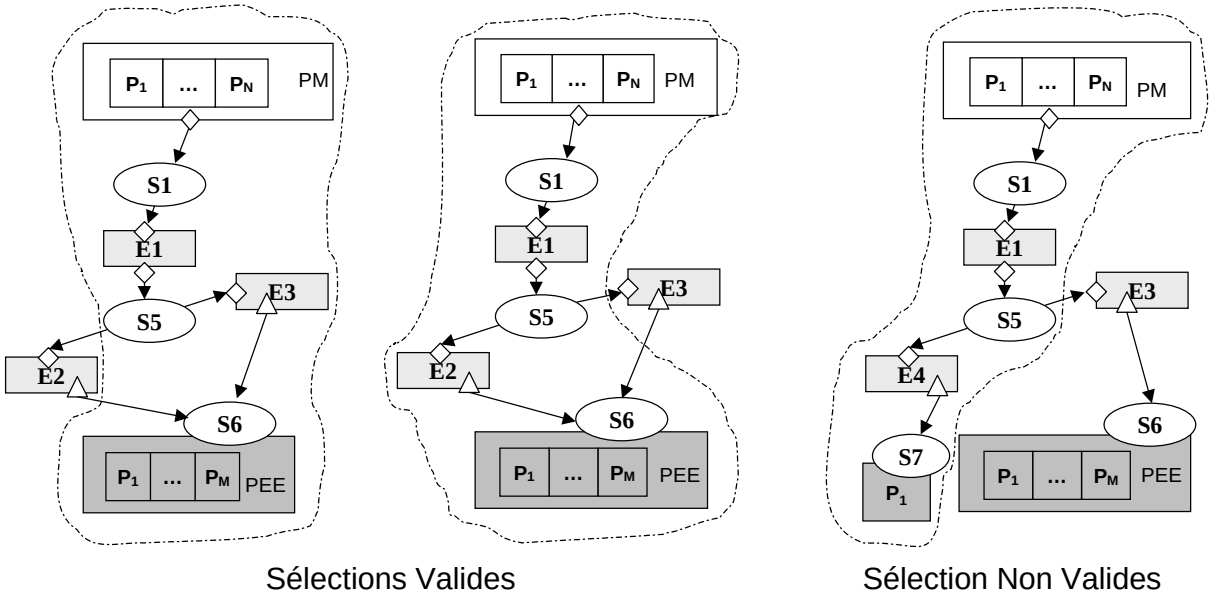


Figure 3-6 Principe de Sélection des Eléments d'interface/Ports Logiques et Services pour Composer un Adaptateur de Communication

3.4 Implémentation d'interfaces abstraites en utilisant le modèle basé sur les services

Dans cette section, nous montrons comment le modèle d'adaptateur de communication peut être appliqué pour la réalisation des interfaces abstraites. Le modèle proposé est utilisé dans un exemple de système hétérogène embarqué, nommé WSS. Cette section est divisé comme suit. Nous commençons par une présentation générale du WSS, ensuite son architecture abstraite est montrée, et enfin les adaptateurs de communication nécessaires pour ce système sont présentés. Des exemples plus significatifs seront présentés dans le chapitre 5 lorsque nous présentons les expérimentations pour valider le modèle d'adaptateur de communication basé sur les services.

3.4.1 Exemple d'application : Le moteur de recherche de séquence de caractères WSS

L'application WSS (Window Searching System) est un moteur de recherche d'une séquence de caractères dans une chaîne de caractères. Le type d'algorithme du WSS peut être utilisé dans diverses applications. Un exemple est le calcul de l'estimation de mouvement pour l'encodage d'une séquence d'images, c'est-à-dire, trouver la position d'une portion de l'image courante (séquence) dans une portion plus grande de l'image précèdente (chaîne) [Jer96].

Dans cet exemple, le WSS est implémenté par un système hétérogène embarqué comme le montre la Figure 3-7. Le système est composé de quatre modules : (1) le Co-Processor qui est chargé de chercher une séquence de caractères dans une chaîne de caractères donnés ; (2) le Sequence Memory qui est une mémoire pour stocker la séquence de caractères ; (3) le String Memory qui est une mémoire pour stocker la chaîne de caractères ; (4) le Top Controller qui est responsable de la coordination des autres modules. Cette architecture permet l'accès aux deux mémoires pendant que le co-processor exécute l'algorithme de recherche. Pour ce système, la Figure 3-7 montre également que, le Top Controller est implémenté comme un module logiciel, alors que les autres ont des implémentations matérielles.

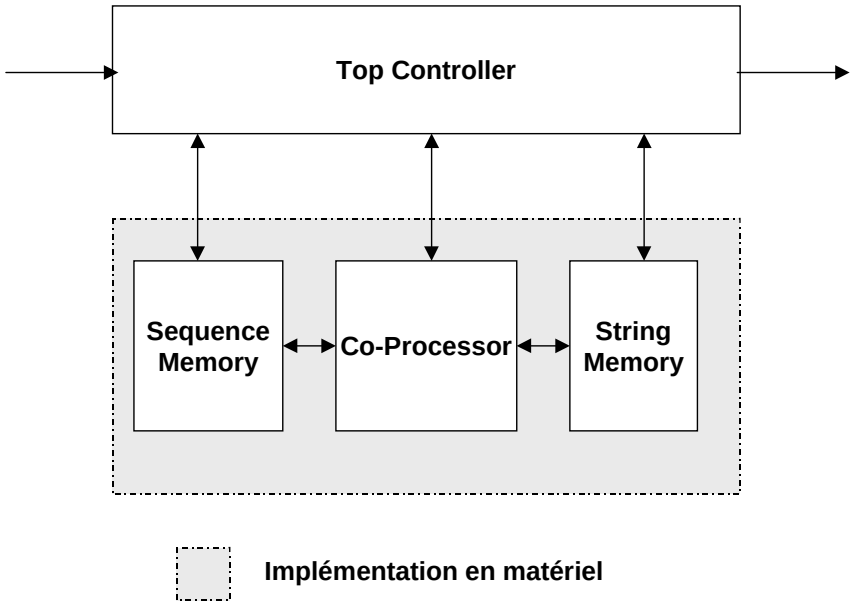
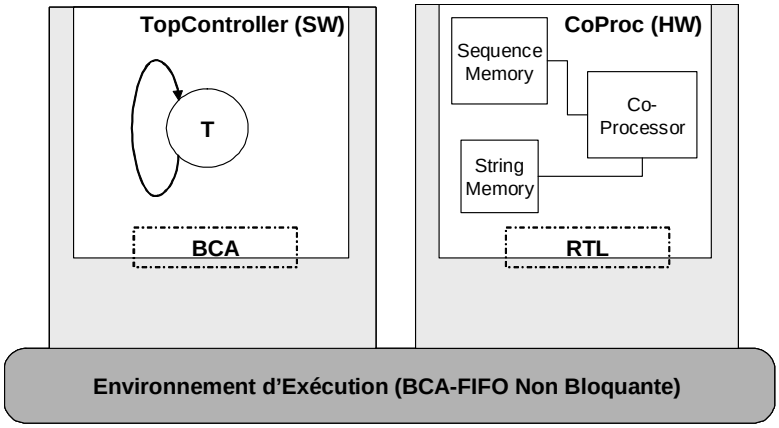


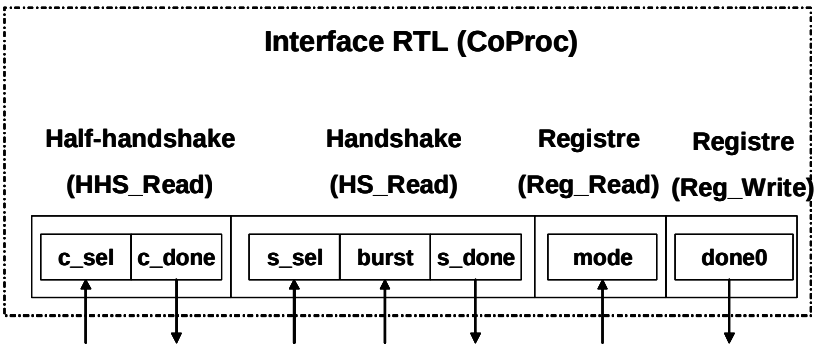
Figure 3-7 WSS -Un Moteur de Recherche de Chaînes de Caractères

3.4.2 L'architecture abstraite du WSS

La Figure 3-8a présente l'architecture abstraite du WSS. Pour cet exemple nous avons décidé de mettre tous les modules matériels dans un seul module hiérarchique (CoProc). Le module matériel communique avec le reste du système au niveau RTL en utilisant des protocoles comme *handshake*, *half-hanshake* et *registre*. La Figure 3-8b montre l'interface RTL du module CoProc. Dans cette figure, nous pouvons également voir les services requis (l'API de communication) de ce module : *HHS_Read* (pour lire la séquence de caractères), *HS_Read* (pour lire la chaîne de caractères), *Reg_Read* (pour lire des signaux de contrôle), *Reg_Write* (pour écrire sur des signaux de contrôle).



(a)



(b)

Figure 3-8 (a) L'Architecture Abstraite du WSS (b) L'Interface RTL du Module CoProc

Le module logiciel (TopController) est composé d'une tache et ses primitives de communications utilisent des *FIFO*s non bloquantes au niveau BCA. L'environnement d'exécution qui interconnecte le TopController et le CoProc utilise également des *FIFO* non

bloquantes au niveau BCA. Ainsi, des interfaces abstraites sont nécessaires pour permettre la communication du CoProc avec le reste du système.

3.4.3 Les adaptateurs de communication du WSS en utilisant le modèle basé sur services

Le Tableau 3-1 présente toutes les adaptations nécessaires pour permettre la communication entre les différents modules. L'API de communication du CoProc est composé de quatre services: une lecture en utilisant le protocole *Half-Handshake* (*HHS_Read*), une lecture en utilisant le protocole *Handshake* (*HS_Read*) et lecture/écriture de registre (*Reg_Read/Reg_Write*). Ces services sont au niveau RTL. L'environnement d'exécution, par contre, est implémenté par une *FIFO* non bloquante qui fournit des services *Get*/*Put* pour lecture/écriture. Ceci étant, il faut implémenter un adaptateur de communication pour chaque service requis par le module CoProc, c'est-à-dire **quatre** adaptateurs de communication. Le Tableau 3-1 montre aussi que des adaptations de type de donnés sont également nécessaires.

Adaptateurs		Half-Handshake/FIFO	Handshake/FIFO	Registre (Read)/ FIFO	Registre(Write)/ FIFO
Module (CoProc)	**Niveau d'Abstraction**	RTL			
	Protocole	Half-Handshake	Handshake	Registre	Registre
	Type de Données	Bit	Bit_vector[8]	bit	bit
	Service Requis	HHS_Read	HS_Read	Reg_Read	Reg_Write
Environnement d'Exécution	**Niveau d'Abstraction**	BCA			
	Protocole	FIFO Non Bloquante			
	Type de Données	Integer			
	Service Fournis	Get/Put			

Tableau 3-1 Les Types d'Adaptateurs Nécessaires pour Permettre la Communication entre les Modules du WSS

Les quatre types d'adaptateur de communication qui réalisent les interfaces abstraites du système sont construits en utilisant le modèle basé sur les services comme le montrent la Figure 3-9 et la Figure 3-10. Les deux premiers adaptateurs permettent la communication du TopController avec les deux mémoires (String Memory et Sequence Memory). Tandis que les adaptateurs présentés dans la Figure 3-10 permettent la communication entre le TopController et le Co-Processor.

Nous pouvons voir dans la Figure 3-9a que le port hiérarchique *PMString* requiert un service *HS_Read*. Celui-ci est fourni par l'élément d'interface *NetworkAccessProvider* qui

requiert cinq types de services : une lecture RTL fournie par le sub-port de *PMString* qui s'appelle *S_DONE* pour savoir si le module est prêt pour lire les donnés (*Read*) ; un service fourni par l'élément *Synchronizer* pour synchroniser l'accès à l'environnement d'exécution (*Wait*); une lecture BCA fournie par le port P relatif à l'environnement d'exécution pour lire les donnés à partir de ce dernier (*Get*); une conversion de données (*DataConv*) fournie par l'élément *Converter* ; deux écritures RTL fournis par les ports *S_SEL* et *BURST* pour écrire les donnés dans le module String Memory (*Write).*

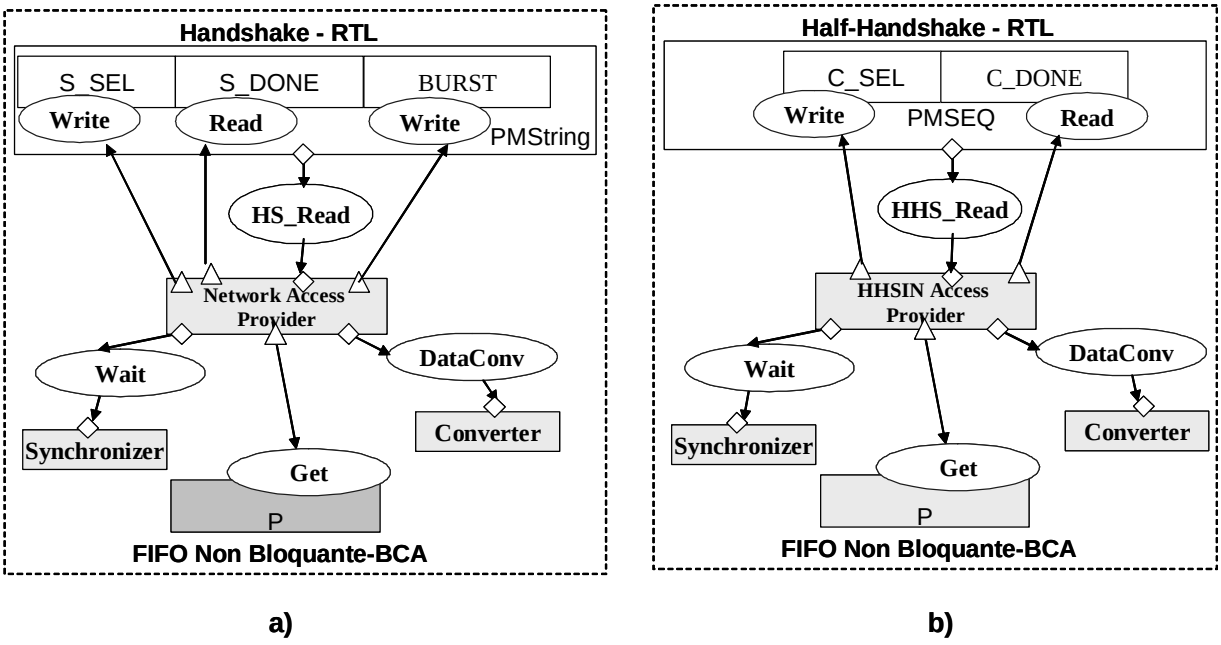


Figure 3-9 (a) Adaptateur pour permettre la Communication Entre le TopController et le StringMemory (b) Adaptateur pour permettre la Communication Entre le TopController et le Sequence Memory

De plus, l'environnement d'exécution utilisait un FIFO bloquante, au lieu d'un non bloquante, il suffirait de changer le port logique, car le service fourni resterait le même (*Get).* Cela signifie que tous les éléments d'interface pourraient être réutilisés pour faire ce nouvel adaptateur de communication. En effet, plusieurs éléments d'interfaces sont réutilisés dans les différents adaptateurs de communication. Tout cela démontre la flexibilité et la facilité de réutilisation de composants de base apportées par le modèle basé sur les services.

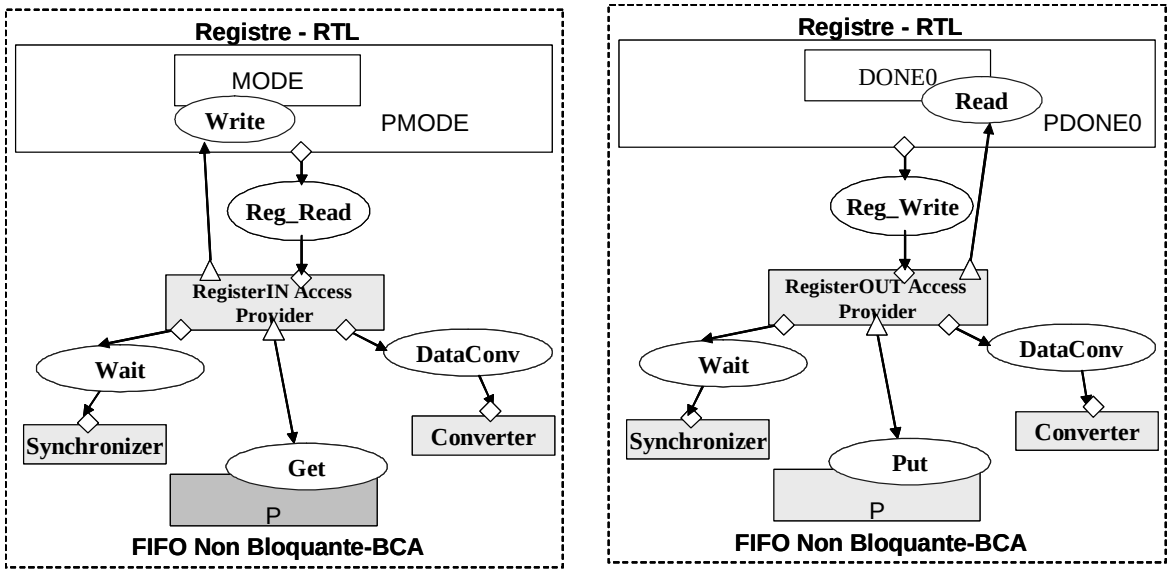


Figure 3-10 Adaptateurs pour Permettre la Communication Entre le TopController et le CoProcessor

3.5 Modélisation de systèmes hétérogènes embarqués

Cette section détaille les langages qui permettent de décrire les systèmes hétérogènes et les GDS. Dans un premier temps, nous parlerons de Colif, qui est un langage de spécification pour les systèmes hétérogènes, puis de Lidel qui est un langage de description pour les GDS. Ce que nous allons présenter est juste un aperçu de ces langages, plus de détails se trouvent en [Ces01] [Gau01]. Pour chaque langage, nous donnerons les concepts de base, les détails d'implémentation et un petit exemple d'utilisation.

3.5.1 Colif : Un langage de spécification pour les systèmes hétérogènes

Pour que nous puissions générer automatiquement des modèles de simulation de systèmes hétérogènes, il nous faut un langage capable de spécifier les architectures abstraites. En plus, ce langage doit avoir des constructions qui permettent de décrire des systèmes hétérogènes en termes de niveaux d'abstractions, protocoles de communication, types de module et interconnexions, etc. Ainsi, nous avons choisi Colif (COdesign Language Intermediate Format) comme langage de spécification de systèmes hétérogènes. En effet, Colif permet de décrire des spécifications multi-langages contenant des modèles de communication de natures différentes et à différents niveaux d'abstraction.

Concepts de base

Colif est un langage de description structurelle de systèmes hétérogènes. Il s'appuie sur trois concepts essentiels : module, port et canal de communication (nommé **net**). Mais

contrairement à la plupart des langages de même type, non seulement le module peut être hiérarchique, mais le canal et le port aussi.

Un port hiérarchique peut être composé de deux types de ports :

- ports internes – ce sont des ports des modules
- ports externes - ce sont des ports qui sont compatibles avec les canaux de communication

Ainsi, un port hiérarchique permet la communication entre deux modules qui utilisent différents langages de spécification, niveaux d'abstraction ou protocoles de communication. Cette abstraction de l'interconnexion entre les modules permet la séparation entre le comportement du module et la communication.

Les différents canaux de communication connectés à un port hiérarchique peuvent être groupés dans des canaux hiérarchiques. Ainsi, une spécification typique de systèmes hétérogènes en Colif est composée de modules ayant des ports hiérarchiques qui sont connectés à des canaux hiérarchiques.

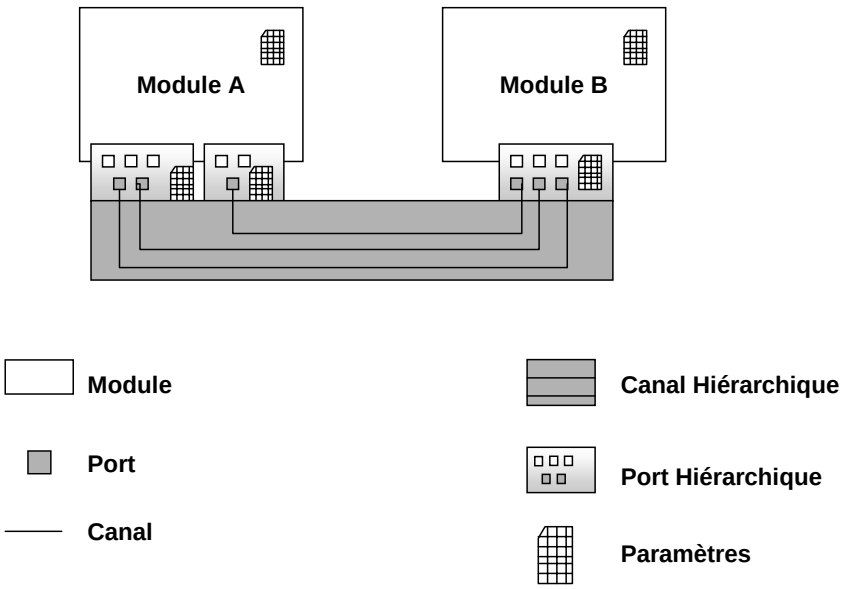


Figure 3-11 Concepts de Base de Colif

Autre aspect important est que Colif permet la spécification de paramètres liés aux modules, ports et canaux. Ces paramètres peuvent contenir diverses informations en vue de la conception ou validation du système. Ils permettent, par exemple, la génération des modèles de simulation. Les paramètres importants sont : les niveaux d'abstraction, les protocoles de communication, les services requis par le module, les services fournis par les canaux et le

langage de spécification des modules. Tous les concepts présentés ici sont illustrés dans la Figure 3-11.

Architectures abstraites en Colif

Les concepts de Colif sont utilisés pour spécifier les architectures abstraites comme le montre la Figure 3-12. La correspondance entre ces concepts et ceux d'architectures abstraites est directe. Un port hiérarchique ou un ensemble de ports hiérarchiques contenant des ports internes et externes représente une interface abstraite. Tandis que l'environnement d'exécution est représenté par des canaux.

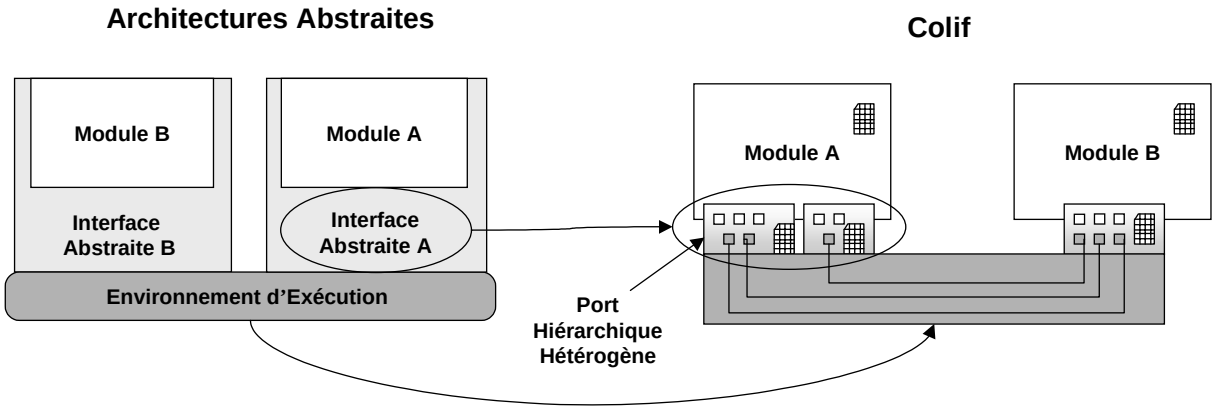


Figure 3-12 Colif et Architectures Abstraites

Modèle d'objets de Colif

L'environnement logiciel de Colif est basé sur les concepts d'objets. Un diagramme de classes simplifié de Colif, en utilisant la notation UML [Uml02], est montré dans la Figure 3-13.

Les trois concepts fondamentaux de Colif – module, port et canal- sont constitués de deux parties : l'interface (nommé ENTITY) et le contenu (nommé CONTENT). L'interface contient un type (TYPE) qui encapsule des propriétés définies par l'utilisateur. Le contenu possède une référence vers un comportement défini et /ou une liste des déclarations (DECL) d'objets.

Pour clarifier un peu, prenons un exemple d'un module spécifié en Colif. Un module a comme ENTITY une liste de déclaration de ports (PORT_DECL) et comme type par exemple SOFTWARE (le module est logiciel). Son contenu est soit une liste de déclaration de modules (dans ce cas il est un module hiérarchique) ou une référence vers un fichier contenant son comportement.

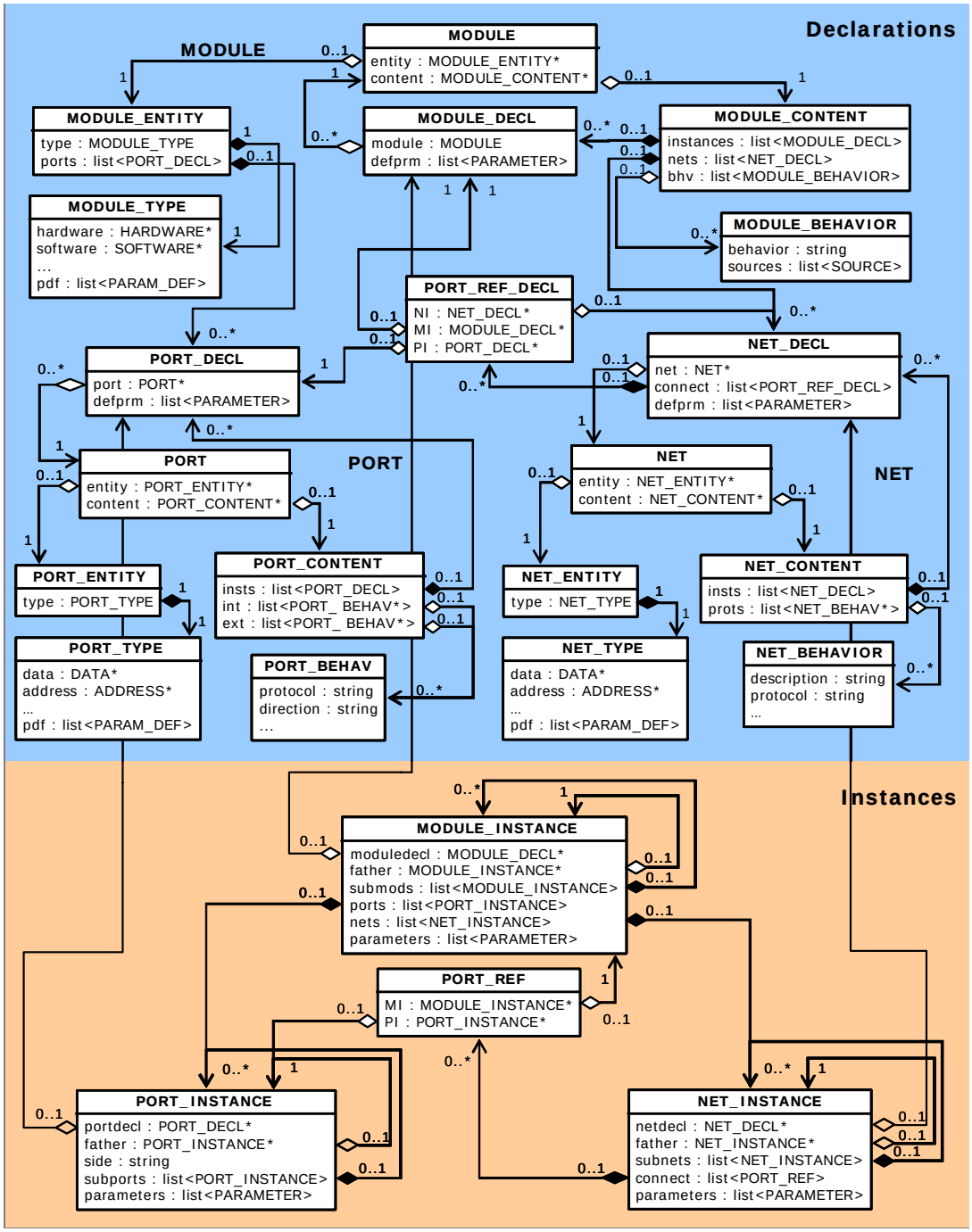


Figure 3-13 Modèle d'Objets de Colif

Nous pouvons remarquer dans ce diagramme de classes, que Colif est divisé en deux parties : une déclarative (en haut du diagramme) et l'autre d'instanciation (en bas du diagramme). La partie déclarative représente des classes définissant un modèle (« template ») réutilisable avec des propriétés génériques (PARAM_DEF) de chaque classe d'objet et aussi des valeurs par défaut de ces propriétés (PARAMETER). La partie d'instanciation représente les classes utilisées pour l'arbre d'instances.

Grâce à cette modélisation de Colif, nous pouvons utiliser une syntaxe uniforme pour spécifier des modules, ports et canaux quelsque soient leurs niveaux d'abstraction, protocoles de communication ou langages de spécification.

Détails d'implémentation

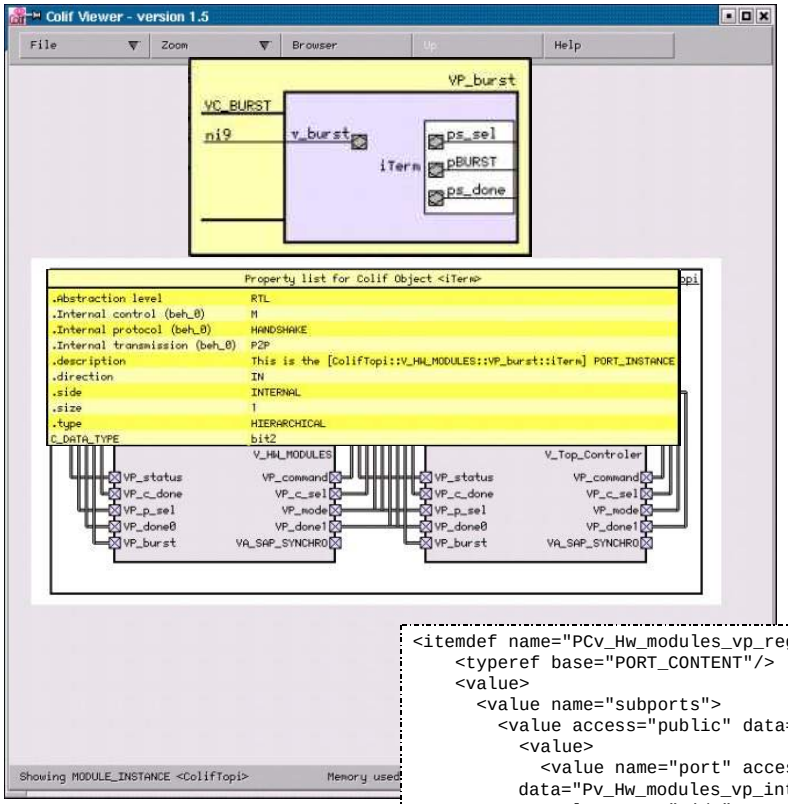
Une spécification Colif consiste en un fichier écrit en XML [XML00] (eXtensible Markup Language) qui suit des règles grammaticales propre à Colif. XML est un langage qui devient, de plus en plus, un standard pour l'échange d'informations entre les différents outils de conception de systèmes.

Une grammaire XML qui s'appelle Middle [Gau01] a été utilisée pour définir le langage Colif (nous verrons plus tard que Lidel utilise aussi cette grammaire). Middle permet de définir des structures de donnés complexes avec une sémantique associée.

Pour les outils qui prennent une spécification Colif en entrée, il y a un analyseur qui génère un arbre d'objets C++ pour accéder aux informations concernant la spécification. Une API C++ est également disponible pour générer des spécifications Colif.

Exemple : Visualisation du WSS spécifié en Colif

La Figure 3-14 montre l'architecture abstraite du WSS en Colif. Nous utilisons dans cet exemple un outil du groupe SLS pour visualiser des descriptions Colif. Nous pouvons voir un port hiérarchique (*VP_burst*), ainsi que les paramètres qui lui sont associés dans la partie gauche de la figure. Ce port a trois ports internes et un autre externe. Le trois ports internes requièrent un service de lecture (*HS*_Read) en utilisant le protocole *handshake* au niveau RTL. Le service fourni par le canal (ou port externe) est le *Put* qui utilise le protocole FIFO au niveau BCA. La partie droite de la figure présente la description Colif en XML de ce port hiérarchique.




```
<itemdef name="PCv_Hw_modules_vp_reg9_VP_burst">
    <typeref base="PORT_CONTENT"/>
    <value>
      <value name="subports">
        <value access="public" data="iTerm">
          <value>
            <value name="port" access="ref"
          data="Pv_Hw_modules_vp_interne_iTerm"/>
            <value name="side" access="value" data="!"/>
            <value name="size" access="value" data="1"/>
            <value name="parameters"/>
          </value>
        </value>
        <value access="public" data="v_burst">
          <value>
            <value name="port" access="ref" data="Port3"/>
            <value name="side" access="value" data="!EXTERNAL"/>
            <value name="size" access="value" data="2"/>
            <value name="parameters">

              <value>
                <value name="name" access="value" data="!C_DATA_TYPE"/>
                <value name="value" access="value" data="!bit2"/>
              </value>
              <value>
                <value name="name" access="value" data="!DATA_BIT_WIDTH"/>
                <value name="value" access="value" data="!2"/>
              </value>
            </value>
          </value>
        </value>
      </value>
```

Figure 3-14 Visualisation du WSS en Colif

3.5.2 Lidel : Un langage de spécification pour les graphes de dépendance de services

Lidel est un langage qui permet la description des dépendances entre éléments/ports logiques et services. Lidel permet non seulement d'exprimer ce type de dépendance, mais aussi d'ajouter des paramètres concernant l'implémentation de chaque composant. Ces paramètres définissent les éléments d'interfaces/ports logiques devant être utilisés dans les adaptateurs de communication.

Concepts de base

Lidel s'appuie sur trois concepts de base : **élément**, **service** et **implémentation**. Avec ces concepts, nous pouvons décrire des GDS en Lidel.

L'élément et le service ont été définis dans la section 3.3.1. Il faut remarquer que dans ce travail nous avons défini les ports logiques comme des éléments Lidel. **L'implémentation** est une réalisation particulière d'un élément. Elle possède ce que nous appelons un domaine de validité : elle peut être compatible avec une architecture matérielle (notamment le processeur), et incompatible avec d'autres. Cette compatibilité peut aussi être associée au langage utilisé pour réaliser l'implémentation. Par exemple, si la réalisation d'un adaptateur de communication nécessite un élément avec une implémentation en C++, et si la seule implémentation disponible est en Java, alors cette implémentation ne sera pas valide.

Plusieurs paramètres sont liés à une implémentation, tels que : le langage de implémentation, le processeur avec lequel elle est compatible, le type d'implémentation (élément d'interface ou port logique), etc.

Les relations entre les trois concepts sont comme suit. Un élément peut fournir et/ou requérir des services (comme expliqué dans le chapitre 3), et il peut posséder une ou plusieurs implémentations.

Détails d'implémentation

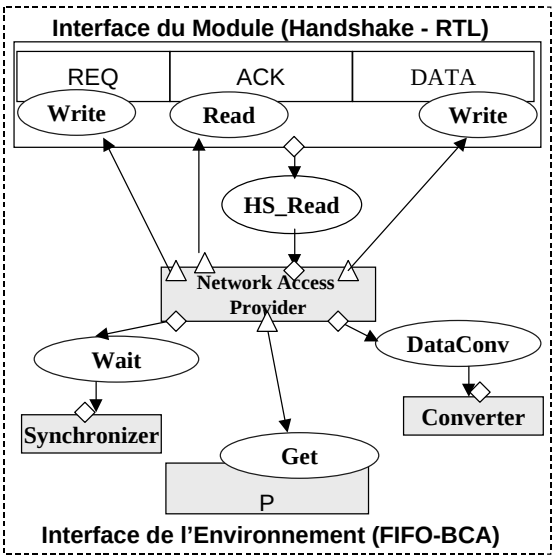
De façon similaire à Colif, une description Lidel consiste en un fichier XML. Sa sémantique est aussi définie en utilisant le métalangage Middle.

L'environnement logiciel de Lidel est basé sur les concepts d'objets. Ainsi, il existe un analyseur pour générer l'arbre d'objets. Cet analyseur est utilisé par les outils qui adoptent Lidel comme langage de description de GDS, pour accéder aux informations contenues dans tel sort de description. Pour générer une description Lidel, la solution la plus efficace est d'utiliser une API C++ fournie par l'environnement logiciel.

Exemple : Description d'un élément d'interface du WSS

La Figure 3-15 présente un exemple d'un élément d'interface du WSS décrit avec Lidel. Dans cet exemple, l'élément *NetworkAccessProvider* est décrit en utilisant l'API C++. Nous pouvons voir que les services fournis et requis par *NetworkAccessProvider* sont spécifiés dans la première partie de la description (lignes 2-8), tandis qu'une implémentation est définie dans la deuxième partie (lignes 9-12).

a)



b)

```
1.  Element NetworkAccessProvider;
2.  NetworkAccessProvider.description("NW Provider");
3.  NetworkAccessProvider.append2provided("HS_Read");
4.  NetworkAccessProvider.append2required("Wait");
5.  NetworkAccessProvider.append2required("DataConv");
6.  NetworkAccessProvider.append2required("Read");
7.  NetworkAccessProvider.append2required("Write");
8.  NetworkAccessProvider.append2required("Get");

9.  Implementation NPImp;
10. NPImp.language("C++");
11. NPImp.append2source("NetworkAcc.cpp");
12. NetworkAccessProvider.append2Implementation(NPImp);
```

Figure 3-15 (a) Adaptateur de Communication du WSS (b) Description d'un Elément d'Interface en Lidel

3.6 Intégration du modèle basé sur les services au flot de conception de systèmes hétérogènes embarqués ROSES

Dans cette dernière partie du chapitre nous discutons la manière dont le modèle basé sur les services peut être intégré au flot de conception de systèmes hétérogènes embarqués qui est nommé ROSES et est développé par le groupe SLS. Les buts de cette intégration sont : (1) unifier les processus de génération des interfaces logicielles/matérielles et des adaptateurs de communication ; (2) apporter une flexibilité plus grande à la réalisation des interfaces logicielles/matérielles et adaptateurs de communication. Nous donnerons d'abord une vue globale de ROSES en présentant toutes les étapes du flot et les outils associés. Ensuite nous décrirons l'intégration du modèle basé sur les services dans l'environnement ROSES.

3.6.1 Présentation du flot ROSES

Le flot ROSES aide la conception des systèmes hétérogènes embarqués en automatisant la génération des interfaces logicielles/matérielles ainsi que les modèles de simulation. Ce flot débute avec la spécification du système à un niveau d'abstraction élevé, mais après que le partionnement logiciel/matériel ait été décidé. La spécification est raffinée de niveaux d'abstraction en niveaux d'abstraction jusqu'à l'implémentation finale du système. Comme sortie, l'utilisateur obtient le code des interfaces logicielles/matérielles du système.

ROSES permet aussi la validation de chaque étape du raffinement en générant un modèle de simulation du système spécifique à l'étape.

.

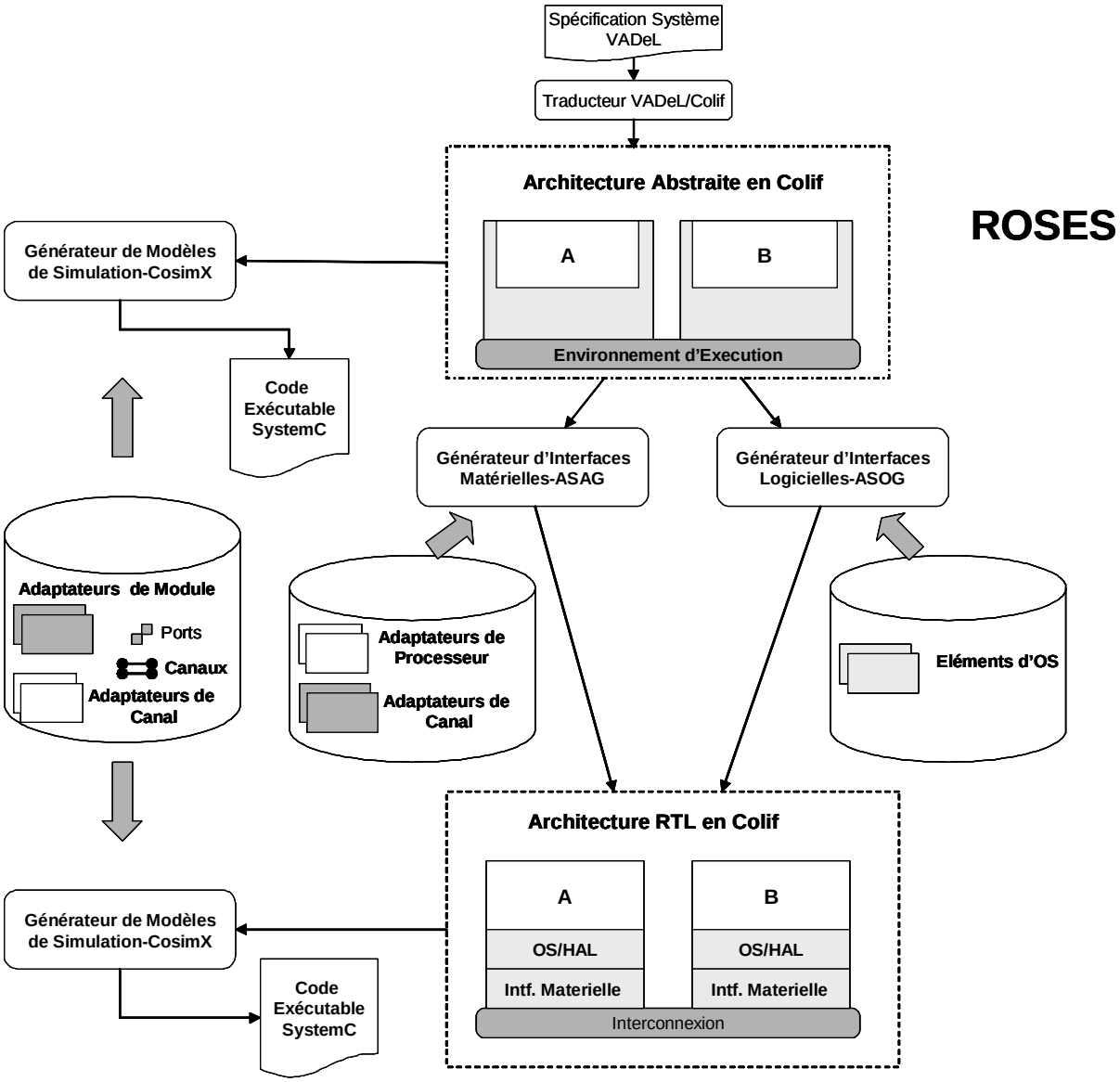


Figure 3-16 Flot de Conception de Systèmes Hétérogènes Embarqués –ROSES

La Figure 3-16 présente en détail le flot ROSES. D'abord le système hétérogène embarqué est spécifié dans un niveau d'abstraction plus élevé que le RTL en utilisant une extension de SystemC, appelé VADeL (venant de l'anglais Virtual Architecture Description Language). VADeL permet de décrire un système comme une architecture abstraite. Cette spécification est, alors, traduite en Colif par un traducteur VADel/Colif. Ensuite, cette spécification haut niveau peut être validée en générant un modèle de simulation en SystemC ou sinon procéder aux étapes de génération des interfaces logicielles/matérielles. En fait

l'architecture abstraite en Colif sert comme entrée pour trois outils : (1) le générateur de modèles de simulation en SystemC – CosimX ; (2) le générateur d'interfaces matérielles – ASAG ; (3) le générateur d'interfaces logicielles – ASOG. Les sorties de chaque outil sont réalisées par l'assemblage des composantes de base existants dans les bibliothèques propres à chaque outil. En effet, le principe de base de ROSES est la conception d'un système complet par l'assemblage des composantes de base. Ainsi, ce principe est appliqué pour la génération des modèles de simulation, pour la génération des interfaces logicielles, et pour la génération des interfaces matérielles.

En sortie du flot, nous obtenons l'architecture RTL du système en Colif, le modèle de simulation de celui-ci, ainsi que les implémentations des interfaces logicielles/matérielles.

3.6.2 Les outils de ROSES

Par la suite, nous allons examiner les trois outils principaux du flot ROSES : ASAG, ASOG et CosimX.

Générateur d'interfaces matérielles – ASAG

L'outil ASAG est chargé de générer des interfaces matérielles ainsi que la représentation Colif de l'architecture du système au niveau RTL. Ces interfaces permettent la communication des processeurs [Lyo03] et des mémoires [Gha03] avec le réseau de communication.

ASAG est basé sur le principe d'assemblage de composants de base existants dans une bibliothèque pour réaliser les interfaces matérielles. La bibliothèque comporte deux composants principaux : des adaptateurs de processeur et des adaptateurs de canal. Ces deux composants interconnectés par un bus interne propre à l'outil forment une interface matérielle. L'adaptateur de processeur traduit le protocole natif du bus du protocole en un protocole propre au bus interne. Et chaque adaptateur de canal effectue la traduction du protocole du bus interne en celui du réseau de communication. L'architecture de ces interfaces permet une certaine flexibilité, car un changement du processeur par un autre pendant le processus de conception du système n'implique qu'un changement d'une partie de l'interface (adaptateur de processeur). Ce modèle d'interface est très similaire à celui qui est présenté dans la figure 2-12 du chapitre 2.

Toutes les informations sur la topologie du système, les paramètres de raffinement, les protocoles de communication et les types de modules sont extraits de la représentation Colif

de l'architecture abstraite. La sortie de cet outil est l'implémentation des interfaces matérielles en VHDL synthétisable ou en SystemC et l'architecture RTL du système en Colif.

Générateur d'interfaces logicielles – ASOG

L'outil ASOG effectue la génération des différentes couches des interfaces logicielles (HAL et OS). Ces interfaces permettent que le code logiciel du système s'exécute sur un processeur cible.

ASOG utilise aussi une bibliothèque de composants de base pour générer les interfaces logicielles, mais différemment d'ASAG (et aussi de CosimX), il utilise un modèle d'interface basé sur les services. En fait, ce modèle avec quelques extensions est celui que nous utilisons pour construire les adaptateurs de communication pour la cosimulation.

Le flot de génération des interfaces logicielles est présenté dans la partie droite de la Figure 3-16. Il prend comme entrées la représentation Colif de l'architecture abstraite et la description des GDS entre les éléments existants dans la bibliothèque en Lidel. Au début, ASOG lit les informations des GDS. Ensuite, il extraite des paramètres de l'architecture abstraite concernant les services requis par le code de l'application, l'implémentation matérielle de ce service, l'espace d'adressage, entre d'autres. Il sélectionne ensuite les éléments qui font partie de l'interface en prenant en compte les paramètres lus dans l'étape précédente et les GDS.

Enfin, ASOG génère le code de l'interface en faisant quelques optimisations pour n'implémenter que les services requis par le code de l'application.

Générateur de modèles de simulation –CosimX

L'outil CosimX génère des modèles de simulation à plusieurs niveaux d'abstraction en partant d'une représentation en Colif de l'architecture du système à concevoir.

Ainsi que les autres outils du flot ROSES, CosimX construit les modèles de simulation en assemblant des composants de base existants dans une bibliothèque. Le modèle utilisé pour générer des interfaces de communication adopte le même principe que celui d'ASAG. Les avantages et désavantages d'un tel modèle ont été discutés dans le chapitre 2. Ceci étant, la bibliothèque comporte des adaptateurs de module et des adaptateurs de canal décrits en SystemC. Evidemment, la bibliothèque contient aussi des ports, des canaux et des adaptateurs de simulateur pour réaliser le reste du modèle de simulation.

CosimX utilise SystemC comme l'environnement d'exécution et permet la cosimulation entre plusieurs simulateurs. La sortie de l'outil est composée de plusieurs fichiers SystemC et d'un Makefile pour compiler ces fichiers.

3.6.3 Les problèmes du flot ROSES

Chaque outil du flot ROSES possède son propre modèle pour représenter les interfaces (ou adaptateurs de communication). Ceci pose deux problèmes majeurs au flot ROSES :

- **Les méthodes d'assemblage de composants de base de chaque outil sont différentes** – La méthode d'assemblage de composants pour implémenter des interfaces/adaptateurs dépend directement du modèle utilisé pour les représenter. La diversité de modèles d'interface/adaptateur fait que ROSES possède un outil d'assemblage pour chaque modèle. Le nombre élevé d'outils existants rend la maintenance de ROSES plus difficile.
- **Les interfaces ne peuvent pas contenir des composants de natures différentes –** Lors de l'étape d'exploration architecturale du système, le concepteur analyse les différentes solutions pour implémenter les interfaces logicielles/matérielles. Ces solutions peuvent varier en fonction du nombre de fonctionnalités réalisées en logiciel ou en matériel (par exemple, l'implémentation de la gestion de tâches peut être soit en logiciel, soit en matériel). Le fait que ROSES utilise des modèles différents pour représenter les composants logiciels et les composants matériels fait qu'une interface ne peut pas être représentée avec ces deux types de composants en même temps. Ceci pénalise la capacité du flot pour faciliter l'exploration architecturale.

3.6.4 L'utilisation des graphes de dépendance de services pour modéliser les interfaces logicielles/matérielles

Le concept proposé de port logique comme une entité capable de fournir/requérir des services est une extension importante au modèle basé sur les services utilisé par ASOG. Ce concept nous permet de représenter deux types de composants très différents : logiciels et matériels. En effet, la généralisation du concept d'élément d'interface avec la notion de port logique permet la représentation des interfaces hétérogènes logicielles/matérielles/fonctionnelles en utilisant un modèle unifié : le graphe de dépendance de services. Ainsi, une interface logicielle/matérielle peut être modélisée par un ensemble de composants logiciels et matériels, tel que l'adaptateur de communication pour la cosimulation proposé dans ce travail.

L'adoption du GDS comme format intermédiaire pour représenter les interfaces logicielles/matérielles/fonctionnelles se présente comme une solution aux problèmes du flot

ROSES. Il peut simplifier le processus d'assemblage des composants de base pour les différents outils (ASAG, ASOG et CosimX). Nous pouvons même envisager l'utilisation d'un seul outil pour assembler les composants. Cet outil peut être utilisé par ASAG, ASOG et CosimX. L'autre avantage de l'adoption des GDS est la modélisation des interfaces hétérogènes qui peuvent contenir des composants logicielles, matérielles et fonctionnelles. Cela permet une exploration d'architecture plus efficace de la part du concepteur en utilisant ROSES.

3.7 Conclusion

Ce chapitre a présenté une des contributions de ce travail qui est un modèle basé sur les services pour représenter des adaptateurs de communication. Les adaptateurs de communication sont composés d'un ensemble d'éléments d'interface et/ou ports logiques liés par la relation de dépendance des services requis/fournis de chaque composant (ce que nous appelons graphe de dépendance de services ou GDS). Nous avons vu que ce modèle peut apporter de nombreux avantages pour la construction de tels adaptateurs, notamment la flexibilité et la facilité de réutilisation des composants de base. Le modèle peut également faciliter l'implémentation d'adaptateurs plus performants puisque seul les services nécessaires pour la fonction globale d'adaptation sont présents.

Ce modèle peut être appliqué pour construire des adaptateurs de communication en partant d'une spécification sous forme d'architecture abstraite du système. Les adaptateurs de communication correspondent à la réalisation des services requis qui sont décrits dans les interfaces abstraites de la spécification. Nous avons appliqué le modèle proposé, à titre d'exemple, pour représenter les adaptateurs de communication du système hétérogène embarqué WSS. L'exemple nous a démontré les avantages que ce modèle peut nous apporter pour implémenter ce type d'adaptateur.

Ce chapitre a présenté également le flot de conception de systèmes hétérogènes embarqués appelé ROSES, qui a été développé par le groupe SLS. Notre travail utilise les langages Colif et Lidel pour spécifier les architectures abstraites et les GDS respectivement.

Nous avons discuté les problèmes du flot ROSES, notamment le manque d'un format unifié pour représenter les différents types d'interface, ce qui rend difficile la gestion des outils du flot et réduit la capacité d'exploration architecturale fournie par celui-ci. Ainsi, nous avons proposé l'utilisation du GDS comme format unifié pour représenter des interfaces hétérogènes logicielle/matérielle/fonctionnelle.

Chapitre 4 : Génération automatique de modèles de simulation pour systèmes hétérogènes embarqués

4.1 Introduction

Le but principal de ce travail est d'accélérer la validation de systèmes hétérogènes embarqués. Ceci étant, il faut trouver un moyen de construire rapidement des modèles de simulation, sachant que pour chaque étape du processus de conception de systèmes hétérogènes, un modèle doit être créé. Dans ce chapitre nous proposons un flot de génération automatique de modèles de simulation pour systèmes hétérogènes embarqués. Ce flot va nous permettre de générer un modèle de simulation pour chaque étape de la conception de systèmes hétérogènes embarqués en partant d'une architecture abstraite du système. Nous verrons que la génération des adaptateurs de communication se fait en utilisant le modèle basé sur les services, expliqué dans le chapitre précédent.

Ce chapitre est organisé en cinq parties. La section 4.2 donne une vue globale du flot proposé, en décrivant de manière générale ses composants, ainsi que l'enchaînement des étapes nécessaires pour la génération automatique de modèles de simulation. La section 4.3 concerne la réalisation des composants existants dans la bibliothèque de cosimulation. La bibliothèque de cosimulation possède les composants de base nécessaires à la construction des modèles de simulation. La section 4.4 détail l'implémentation de chaque outil du flot développé dans ce travail. La section 4.5 présente les fichiers générés à la fin du flot. Enfin, la section 4.6 propose l'intégration du flot de génération automatique de modèles de simulation dans le flot ROSES.

4.2 Flot de génération de modèles de simulation pour systèmes hétérogènes embarqués

Cette section est consacrée à donner une vue globale du flot de génération automatique de modèles de simulation pour systèmes hétérogènes embarqués. La stratégie principale de cette génération est basée sur le principe de sélection, de configuration et d'assemblage de composants de base existants dans une bibliothèque. Ce flot, qui est illustré dans la Figure 4-1, prend comme entrée une spécification de l'architecture abstraite du système et sort le code exécutable du modèle de simulation, après l'interaction de plusieurs outils développés dans ce travail. Dans cette section, nous verrons d'abord les principaux composants du flot, et ensuite l'enchaînement des étapes de celui-ci.

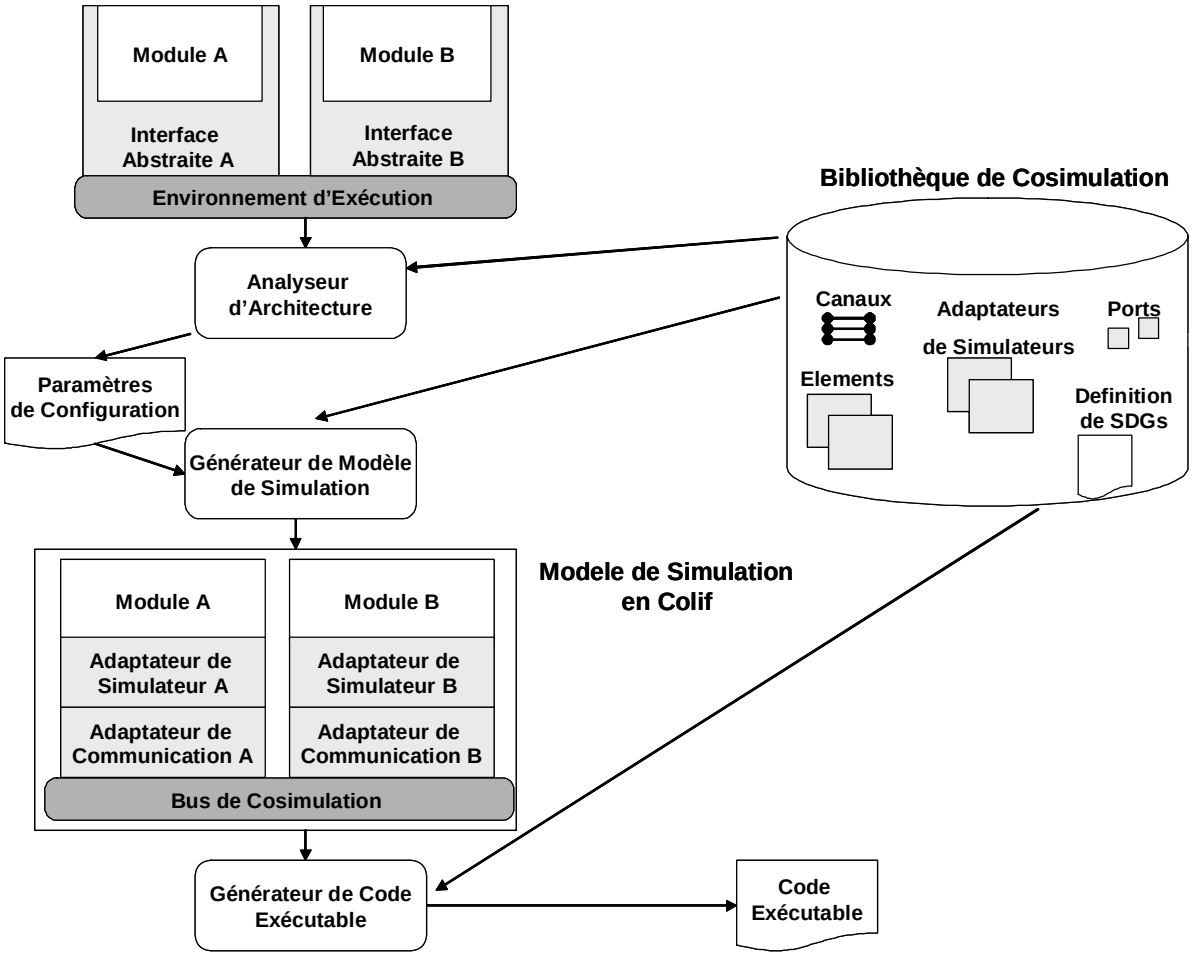


Figure 4-1 Flot de Génération de Modèles de Simulation pour Systèmes Hétérogènes Embarqués

4.2.1 Composants du flot de génération de modèles de simulation

Le flot proposé est constitué de quatre composants principaux : trois outils et une bibliothèque de composants de base pour les modèles de simulation. Dans cette section, nous ne présentons que la description générale de chaque composant du flot. Les détails d'implémentation des composants se trouvent dans les sections 4.3et 4.4.

Analyseur d'architecture

L'Analyseur d'Architecture est un outil qui prend comme entrées la spécification du système hétérogène et les descriptions des GDS entre les éléments d'interface/ports logiques existants. Il analyse, ainsi, s'il y a des adaptations à faire pour permettre la communication entre les modules et l'environnement d'exécution. L'Analyseur d'Architecture est responsable aussi d'interpréter les GDS d'entrée et d'extraire les informations importantes pour la construction des adaptateurs de communication.

Bibliothèque de cosimulation

La Bibliothèque de Cosimulation contient tous les composants de base pour créer les modèles de simulation. Ces composants de base consistent en : éléments d'interface, ports logiques, adaptateurs de simulateur et canaux de communication (ce dernier, pour implémenter le bus de cosimulation). Cette bibliothèque possède aussi la description des GDS.

Générateur de modèle de simulation

Le Générateur de Modèle de Simulation est un outil qui sélectionne, configure et assemble les différents composants de base existants dans la Bibliothèque de Cosimulation pour créer un modèle de simulation du système. Un des rôles principaux de cet outil est de sélectionner les éléments d'interface/ports logiques qui implémentent les adaptateurs de communication. Cet outil est responsable, également, d'autres tâches dont générer les adaptateurs de simulateurs et connecter tous ces composants pour former un modèle de simulation. Le modèle de simulation généré par cet outil n'est pas dans un premier temps exécutable. Il est décrit en Colif (expliqué dans la section 3.5).

Générateur de code exécutable

Le Générateur de Code Exécutable est un outil qui sort le code exécutable du modèle de simulation du système. Il assemble les implémentations des composants de la Bibliothèque de Cosimulation qui sont requis par le modèle de simulation et génère un code exécutable compatible avec l'environnement d'exécution.

4.2.2 Enchaînement des étapes du flot de génération de modèles de simulation

Le flot de génération automatique de modèles de simulation commence par la spécification du système hétérogène sous forme d'architecture abstraite, et la description des GDS entre les éléments d'interface/ports logiques existants dans la Bibliothèque de Cosimulation. La spécification est annotée avec diverses paramètres tels que : niveaux d'abstraction des modules et ports, protocoles de communication, services requis par les modules et fournis par l'environnement d'exécution, langages de description des modules, type de module, etc. La description des GDS donne non seulement des informations sur les dépendances entre les éléments d'interface/ports logiques, mais aussi des informations concernant ces composants

telles que : le répertoire où se trouve les éléments d'interface/ports logiques, le langage d'implémentation, le type de composant (port logique ou élément d'interface), etc.

La prochaine étape du flot est l'analyse d'architecture. L'Analyseur d'Architecture prend en entrée la spécification du système et la description des GDS, vérifie quelles adaptations sont nécessaires et génère les paramètres pour la génération du modèle de simulation. Typiquement, cet outil prend en compte des informations comme les niveaux d'abstraction, les protocoles de communication et les services requis et fournis, pour analyser si une adaptation est nécessaire.

Les paramètres générés pendant l'analyse d'architecture sont les entrées de la prochaine étape du flot, qui est la génération du modèle de simulation. Cette étape consiste à générer les adaptateurs de simulateur et de communication, et à les connecter avec le reste du système. Comme nous le verrons plus tard, la génération des adaptateurs de communication se fait en utilisant le modèle basé sur les services. La sortie de cette étape est un modèle de simulation en Colif.

La dernière étape du flot est la génération de code exécutable du modèle de simulation décrit en Colif. Il instancie les composants de la Bibliothèque de Cosimulation requis par le modèle de simulation et génère des fichiers de code compatible avec l'environnement d'exécution ainsi qu'un Makefile pour les compiler.

Nous avons découpé ce flot en trois étapes distinctes (l'analyse d'architecture, la génération de modèles de simulation et la génération de code exécutable) pour assurer une meilleure flexibilité. En faisant ceci, le flot n'est pas si dépendent du langage de spécification d'entrée ou de l'environnement d'exécution. Ainsi, si nous changeons l'environnement d'exécution, il ne faut pas changer tous les outils, seulement le Générateur de Code Exécutable. Dans ce cas, il faut aussi ajouter des composants à la Bibliothèque de Cosimulation compatibles avec le nouvel environnement d'exécution

4.3 La bibliothèque de cosimulation

Cette section détaillera la Bibliothèque de Cosimulation. La Bibliothèque de Cosimulation contient des composants de base de natures diverses. En assemblant ces composants, nous pouvons construire les différentes parties du modèle de simulation. Essentiellement, il y a quatre types de composants dans la bibliothèque : (1) les adaptateurs de simulateurs ; (2) les canaux de communication pour réaliser le bus de cosimulation ; (3) les éléments d'interface ; (4) les ports logiques.

Par la suite, nous allons examiner chacun de ces composants. Nous utilisons SystemC [Sys00] comme l'environnement d'exécution dans ce travail, ainsi la plupart de ces composants sont implémentés en C, C++ ou SystemC.

4.3.1 La réalisation des adaptateurs de simulateur

L'Adaptateur de Simulateur apporte l'aspect multi langage à la cosimulation. Ici, nous utilisons une approche classique pour implémenter ces adaptateurs. La Figure 4-2 présente l'architecture d'un adaptateur de simulation. Il est composé de deux parties principales:

- FLI (Foreign Language Interface) – c'est un composant décrit en utilisant le langage du simulateur étranger à l'environnement d'exécution. Ce langage étranger doit fournir des bibliothèques qui permettent des appels de fonctions décrites en C. Des exemples de telles bibliothèques sont : CLI pour VSS/VHDL [Syn00], S functions pour Simulink/MATLAB [Mat00], etc ;
- Adaptateur de l'Environnement d'exécution – c'est un composant décrit en utilisant le même langage de l'environnement d'exécution. Dans notre cas, il est implémenté en SystemC.

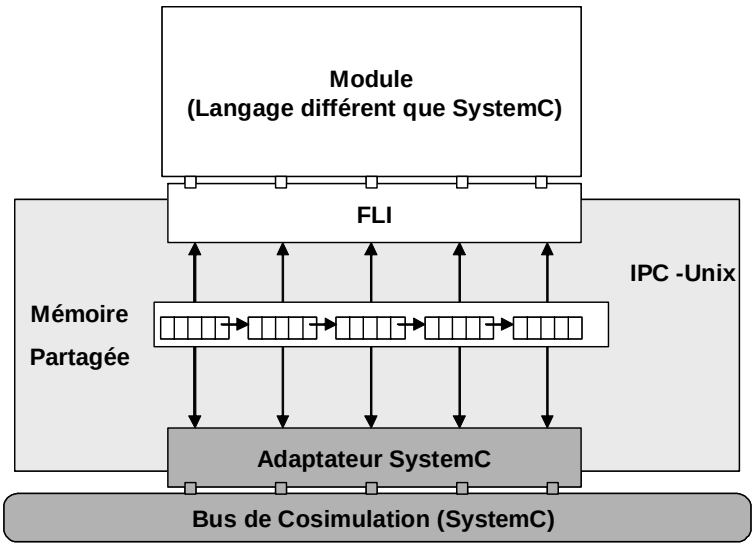


Figure 4-2 Architecture des Adaptateurs de Simulateur

Dans notre réalisation, ces deux composants communiquent entre eux en utilisant les mécanismes de communication interprocessus fournis par UNIX, nommés IPC (venant de l'anglais Inter Process Communication). Chaque simulateur participant à la simulation est en fait un processus fils d'un processus SystemC avec lequel il communique par des IPC. Les

données échangées entre les simulateurs sont stockées dans des mémoires partagées et nous utilisons des sémaphores pour contrôler leurs accès.

La Bibliothèque de Cosimulation contient, actuellement, les adaptateurs de simulateurs pour l'ISS du processeur ARM7 et ARM9 [Arm99], et pour adapter la simulation native en UNIX à l'environnement d'exécution. Les adaptateurs pour VSS/VHDL et Simulink/MATLAB ne sont pas dans la bibliothèque, parce qu'ils peuvent être générés de manière automatique. Nous avons choisi de mettre les adaptateurs de simulateurs de processeurs (ex : ISS) dans la bibliothèque parce que les processeurs sont des composants standard qui sont employés souvent en systèmes hétérogènes. Tandis que pour les IPs (ils ne sont pas toujours des composants standard), c'est plus pratique de les générer automatiquement.

4.3.2 La réalisation des bus de cosimulation

Nous avons implémenté plusieurs canaux de communication pour réaliser des bus de cosimulation à différents niveaux d'abstraction. Ces canaux ont été implémentés en utilisant SystemC 2.0 [Sys00] [Sys00a]. En effet, cette version de SystemC introduit quelques concepts intéressants (notamment le concept d'interface de canal) qui facilitent la tâche d'implémenter des canaux de communication plus complexes (avec synchronisation, arbitrage, etc.).

Actuellement, la Bibliothèque de Cosimulation contient des bus de cosimulation aux niveaux d'abstraction suivants:

- RTL – réalisé par les signaux fournis par SystemC (sc_signal) ;

- BCA – réalisé par des canaux que nous avons implémentés. Ils sont un peu plus complexes que les signaux RTL, et peuvent implémenter quelques protocoles comme par exemple des FIFOs bloquantes et non bloquantes ;

- TLM – réalisé par un canal (ou plutôt un bus) qui fait le routage de données et, selon le besoin, peut faire l'arbitrage d'accès.

- Message – réalisé par des canaux qui implémentent le standard de communication MPI. Nous disposons de deux implémentations de canaux MPI : une point-à-point et l'autre multipoint.

4.3.3 La réalisation des éléments d'interface

Les composants d'un GDS peuvent être classifiés en éléments d'interfaces et ports logiques. Nous allons parler maintenant des éléments d'interface.

Les éléments d'interface, existant dans la bibliothèque, sont implémentés en deux langages : C++ et Rive.

Rive est un langage de macro, employé ici pour nous aider à configurer l'élément que nous voulons générer. En réalité, nous pourrions utiliser d'autres langages de macro, comme m4 [Gnu04] par exemple, mais nous lui avons préféré Rive pour sa facilité d'intégration avec d'autres outils, et pour sa possibilité de gérer aussi bien les itérations que les recursivités.

La Figure 4-3 illustre l'implémentation d'un élément appartenant à un adaptateur de communication du WSS. Les expressions non traités par l'expandeur de macro Rive sont entre guillemets (″ ″). Rive offre des expressions de contrôles conditionnels comme IF-ELSE et des fonctions prédéfinies comme ISDEFINED (pour savoir si un paramètre a été déjà défini). Nous pouvons aussi définir des paramètres d'expansion comme par exemple PORT_OUT_CTRL_REQ_RTL (ligne 7). Les valeurs de ces paramètres seront passées au moment de la génération du code exécutable. Cette génération de macro fait que les éléments soient générés avec les services strictement nécessaires à l'adaptation, même si l'élément peut fournir d'autres services.

Nous pouvons voir également dans la Figure 4-3 les services requis par cet élément (lignes 7, 10, 11, 12, 14, 16, 20). Il faut remarquer une contrainte importante dans l'implémentation d'un élément d'interface: lorsqu'un service requis par l'élément est fourni par un port logique (un PAP), il faut le signaler explicitement comme le montre la ligne 10 par exemple. Lors de l'expansion de la macro, les noms réels des ports sont passés comme des paramètres pour que les éléments puissent utiliser leurs services. Pour accéder aux services fournis par un autre élément d'interface, il suffit de faire un appel à la fonction relative au service requis comme le montre la ligne 11.

Un autre point important à remarquer, c'est qu'un adaptateur de communication est implémenté comme un module SystemC. Ce module SystemC possède une partie déclarative où nous devons déclarer les ports, les fonctions et le constructeur, et une partie concernant le comportement. Les outils développés dans ce travail, génèrent toute la partie déclarative et ajoutent le code des éléments pour la partie comportementale. Autrement dit, l'utilisateur ne doit que fournir les éléments qui implémentent le comportement.

```
1. @{IF (ISDEFINED {NetworkAccessProvider_HS_Read}) DO

2. "void" CLASSNAME "::HS_Read()
3.   {
4.       bool data_read = true;
5.       while (true) {
6.          if (data_read) {
7.               "PORT_OUT_CTRL_REQ_RTL".write(true);
8.               "DATATYPE_CHANNEL "dataCh;
9.               "DATATYPE_MODULE "dataMod;
10.              dataCh = "PORT_IN_DATA_BCA".Get();
11.              DataConv(dataCh,dataMod);
12.              " PORT_OUT_DATA_RTL ".write(dataMod);
13.         }
14.         if ("PORT_IN_CTRL_ACK_RTL".read()) {
15.              data_read = true;
16.              "PORT_OUT_CTRL_REQ_RTL".write(false);
17.         } else {
18.              data_read = false;
19.         }
20.         wait();
21.      }
22. }"
23.  ENDIF
24.}@
```

Figure 4-3 Réalisation d'un Elément d'Interface en Utilisant le Langage de Macro Rive

4.3.4 La réalisation des ports logiques

De même que les canaux de communication, les ports logiques sont également implémentés en SystemC 2.0. Les niveaux d'abstraction de ces ports sont, évidemment, les mêmes que ceux des canaux, pour permettre la interconnexion entre les modules. Les services fournis par les ports sont effectivement implémentés par les canaux de communication qui leur sont connectés.

4.4 Détails d'implémentation des outils développés pour la génération automatiques de modèles de simulation

Cette section est consacrée aux trois outils que nous avons développé pour générer des modèles de simulation. Nous aborderons, ici, les détails d'implémentation de chacun de ces outils. De manière générale, nous verrons les informations d'entrée qui sont prises en compte et le mode de fonctionnement de chaque outil.

4.4.1 Analyseur d'architecture

L'Analyseur d'Architecture peut être divisé en deux parties : la première extraire les informations importantes de l'architecture abstraite du système en Colif ; la deuxième extraire les informations pertinentes des GDS décrits en Lidel.

La première partie de l'Analyseur d'Architecture examine chaque composant de l'architecture abstraite, c'est-à-dire, le module, le port et le canal. A partir des informations obtenues de chaque composant, l'outil analyse si une adaptation est nécessaire et quel type en est.

Les informations extraites d'un module sont :

- type de implémentation du module (logicielle ou matérielle)
- langage de spécification
- niveau d'abstraction

Ces informations déterminent le besoin d'une adaptation de simulateur. Par exemple, si le module possède une implémentation logicielle et son niveau d'abstraction est RTL, nous aurons besoin d'une adaptation car ce module va être simulé par un ISS.

Les informations extraites d'un port sont :

-si le port est hiérarchique avec des ports internes et externes

-niveaux d'abstraction des ports internes et externes

-protocoles de communication des ports internes et externes

-services requis par les ports internes

-services fournis par les ports externes

-type de données des ports internes et externes

En regardant ces informations, l'analyseur sait si une adaptation de communication est nécessaire. Par exemple, si le niveau d'abstraction d'un port interne est différent de celui d'un port externe, l'adaptation doit se faire. De plus, en analysant des ports hiérarchiques l'outil n'a plus besoin d'analyser les mêmes informations contenues dans les canaux de communication puisque les ports externes les fournissent déjà.

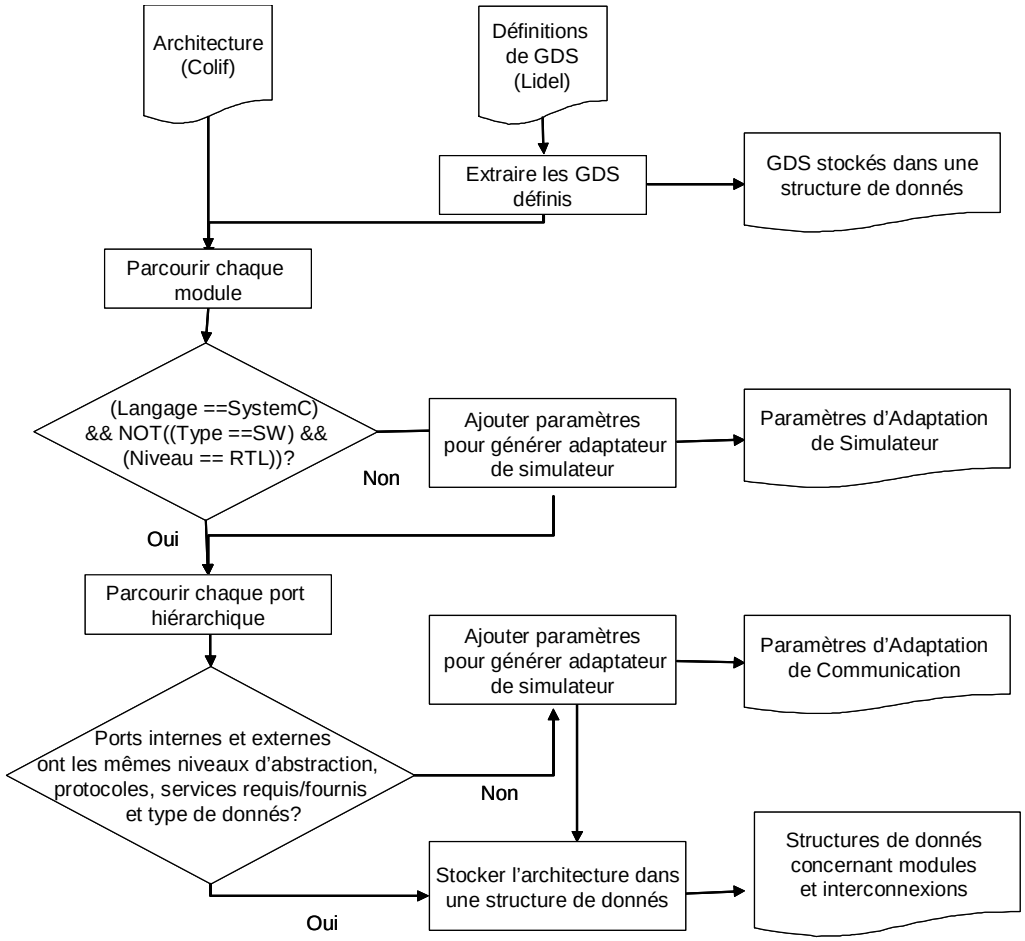


Figure 4-4 Flot d'Analyse d'Architecture

La deuxième partie de l'Analyseur d'Architecture stocke les informations concernant les GDS décrit en Lidel dans des structures de données en mémoire. Les informations stockées sont typiquement : les noms des éléments/ports logiques, les services requis et fournis par chacun, le langage d'implémentation, une référence vers le code source, entre autres. Cette partie d'outil construit le(s) GDS maximal(aux) des éléments d'interface/ports logiques existants dans la Bibliothèque de Cosimulation. Ce GDS maximal sera utilisé par le Générateur de Modèles de Simulation pour construire les adaptateurs de communication.

La Figure 4-4 illustre le mode de fonctionnement de l'Analyseur d'Architecture. L'outil prend l'architecture abstraite et les définitions des GDS en entrée. Il extraire les GDS (construit les GDS maximaux) et les stocke dans des structures de données en mémoire. L'outil commence à analyser l'architecture abstraite, en parcourant d'abord les modules. S'il trouve qu'une adaptation de simulateur est nécessaire, il ajoute les paramètres pertinents pour la génération de cet adaptateur dans une structure de données en mémoire qui va être accédée, après, par le Générateur de Modèle de Simulation. Ces paramètres sont : une référence vers le

module, le langage de spécification du module et le type de module. Puis, l'Analyseur d'Architecture parcourt chaque port pour évaluer le besoin d'une adaptation de communication. S'il y a en un, l'outil ajoute les paramètres pour la génération de ces adaptateurs. Les paramètres ajoutés sont : une référence vers le port concerné, les niveaux d'abstraction, les protocoles, les services requis/fournis et les types de données trouvées. Après tout cela, l'Analyseur d'Architecture garde l'architecture abstraite dans des structures de données qui sont accédées par le Générateur de Modèle de Simulation.

4.4.2 Générateur de modèle de simulation

Le Générateur de Modèle de Simulation est un outil qui est chargé de plusieurs tâches: générer des adaptateurs de simulateur, générer des adaptateurs de communication, et connecter tous ces composants avec le reste du système pour créer un modèle de simulation. Il est un outil complexe qui peut être vu comme un ensemble d'outils dans un seul. Par la suite, nous allons décortiquer cet outil pour mieux comprendre comme il arrive à accomplir chaque tâche.

L'architecture du générateur de modèle de simulation

La Figure 4-5 présente l'architecture du Générateur de Modèle de Simulation. Il est composé de trois parties principales : la première génère les adaptateurs de simulateur (Générateur d'Adaptateur de Simulateur) ; la deuxième génère les adaptateurs de communication (Générateur d'Adaptateur de Communication) ; et la troisième génère un modèle de simulation en Colif avec les adaptateurs connectés au reste du système par des canaux choisis par cette partie d'outil (Générateur de Colif).

Les entrées pour les générateurs d'adaptateurs sont données par l'Analyseur d'Architecture. Tandis que les entrées pour le Générateur de Colif viennent non seulement de l'Analyseur d'Architecture, mais aussi des générateurs des adaptateurs.

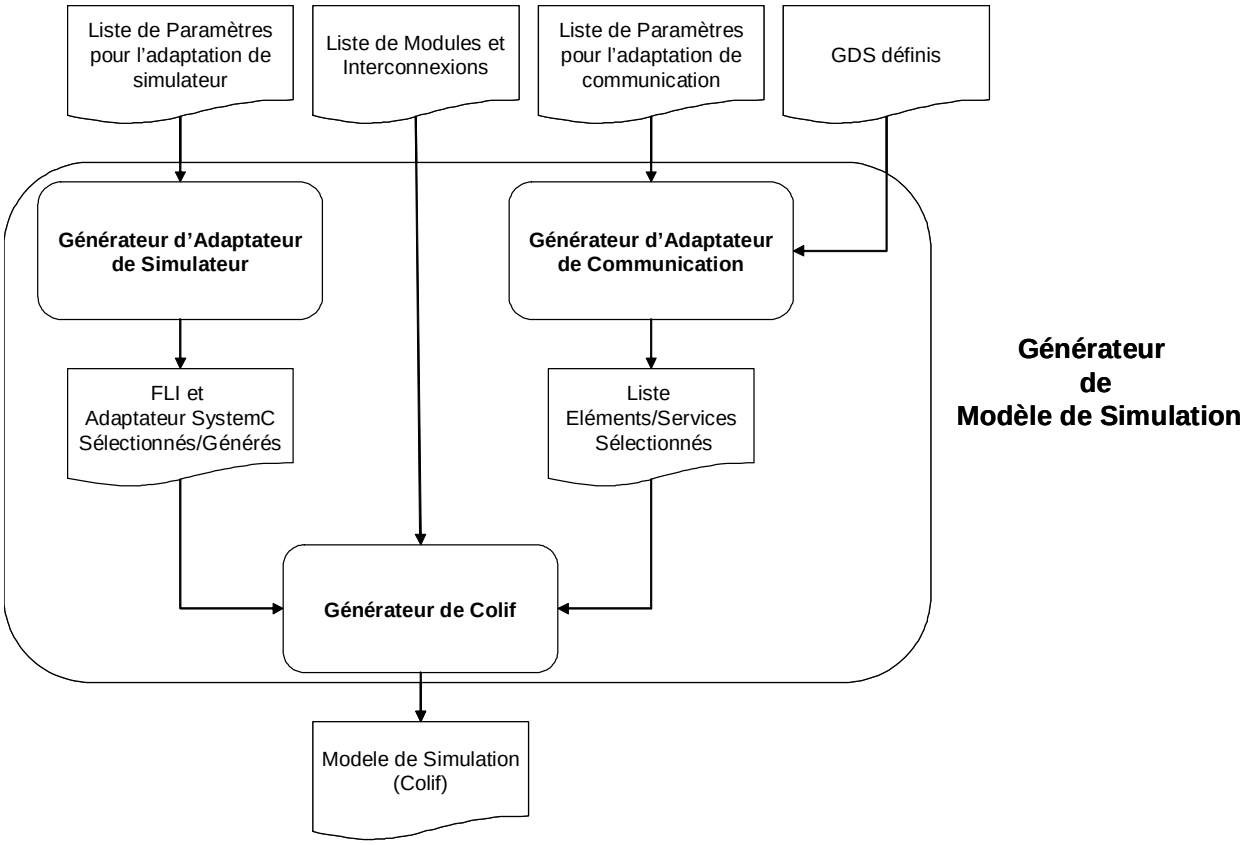


Figure 4-5 L'Architecture du Générateur de Modèle de Simulation

Générateur d'adaptateur de simulateur

La Figure 4-6 montre le mode de fonctionnement du Générateur d'Adaptateur de Simulateur. Il prend la liste de paramètres pour l'adaptation de simulateur, et inspecte d'abord si l'implémentation du module est logicielle au niveau RTL (un processeur). Dans ce cas, il choisit un adaptateur pour ce module (un BFM) parmi ceux qui sont dans la Bibliothèque de Cosimulation. S'il n'existe pas un BFM pour ce type de processeur, le Générateur d'Adaptateur de Simulateur rapporte l'erreur à l'utilisateur. Dans le cas où le module n'est pas un processeur, le Générateur d'Adaptateur de Simulateur génère automatiquement le FLI et l'Adaptateur SystemC. Enfin, tous les composants nécessaires pour l'adaptation (le BFM, ou le FLI et adaptateur SystemC) sont regroupés en formant un adaptateur de simulateur. Les sorties sont : une référence vers le module qui a besoin d'une adaptation et les composants qui font partie de l'adaptateur (le FLI et l'adaptateur SystemC s'ils étaient crées ou une référence vers le BFM utilisé).

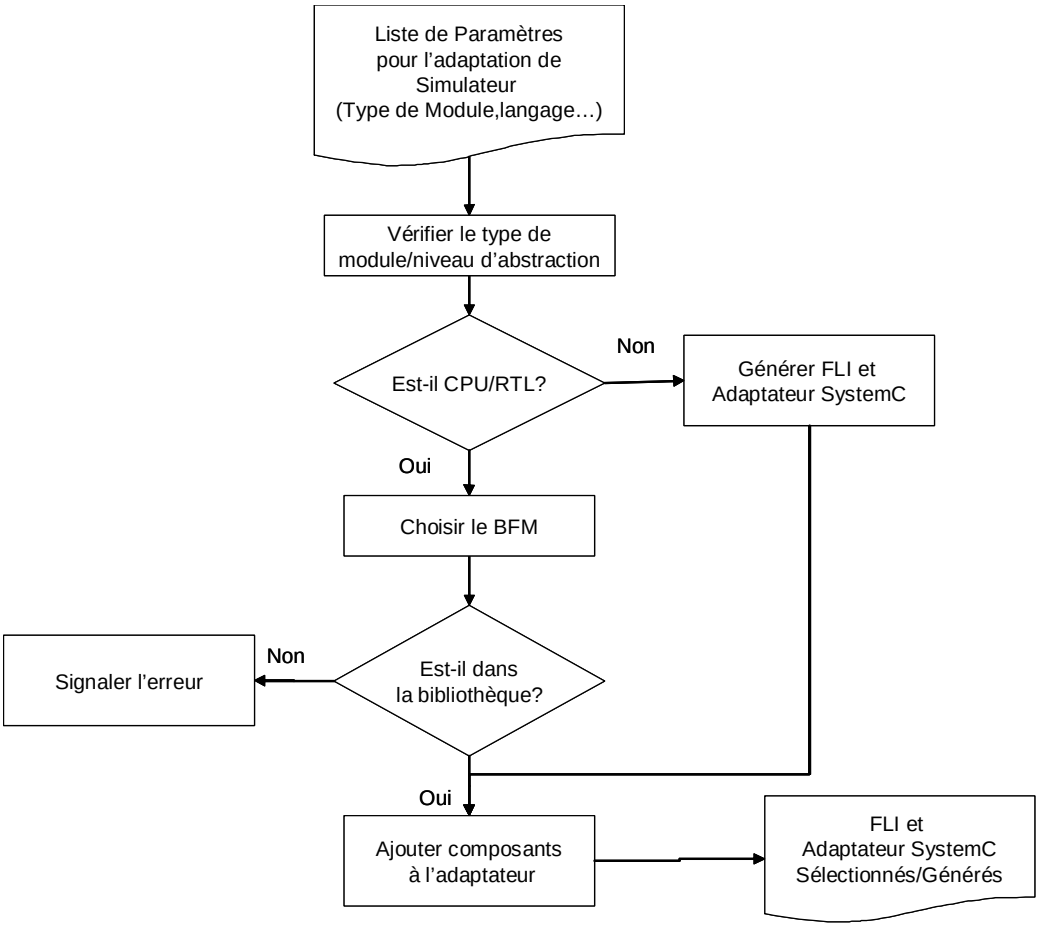


Figure 4-6 Flot de Génération d'Adaptateurs de Simulateur

Générateur d'adaptateur de communication

Le Générateur d'Adaptateur de Communication construit ce type d'adaptateur en utilisant le modèle basé sur les services. Il prend les GDS des éléments d'interface/ports logiques existants dans la Bibliothèque de Cosimulation et une liste de paramètres, et sélectionne une liste d'éléments/ports logiques qui implémentent l'adaptateur de communication. Cette liste consiste en paramètres tels que : les services requis et fournis, les protocoles des modules/environnement d'exécution, les niveaux d'abstraction des modules/environnement d'exécution, etc. Ces paramètres et le langage de l'environnement d'exécution sont décisifs pour la sélection des éléments/ports logiques qui composent l'adaptateur de communication. En effet, lors de la génération des adaptateurs, l'outil ne sélectionne que les éléments qui peuvent fournir l'adaptation et qui contiennent des implémentations en utilisant des langages compatibles avec celui de l'environnement d'exécution.

L'algorithme utilisé par le Générateur d'Adaptateur de Communication parcourt récursivement le GDS d'entrée en partant de ses racines. Pour chaque racine l'algorithme recherche un élément d'interface/port logique qui fournit les services requis et qui respecte les

paramètres d'implémentation (langage d'implémentation, protocole de communication, niveau d'abstraction). Si l'algorithme trouve un, il supprime tous les services et éléments d'interface/port logiques partant de cette racine, conduisant à des services non nécessaires (ou qui ne sont pas d'accord avec les paramètres d'implémentation). Une technique de coloriage des nœuds de graphe est employée pour sortir des cycles et éliminer les éléments d'interface/ports logiques et les services qui ne sont pas valides (soit parce qu'ils ne sont pas nécessaires à l'adaptation, soit parce qu'ils ne respectent pas les contraintes d'implémentation).

La Figure 4-7 illustre le mécanisme de sélection d'éléments employé par le Générateur d'Adaptateur de Communication. Un service *HS_Write* est requis par un module et l'environnement d'exécution fournit le service *Put*. Dans le GDS présenté, les éléments *NetworkAccessProvider* et *NWProvider* fournissent le service *HS_Write*. Malgré cela, seulement le premier élément est sélectionné par le Générateur d'Adaptateur de Communication, car il est le seul à requérir le service *Put* fourni par l'environnement d'exécution.

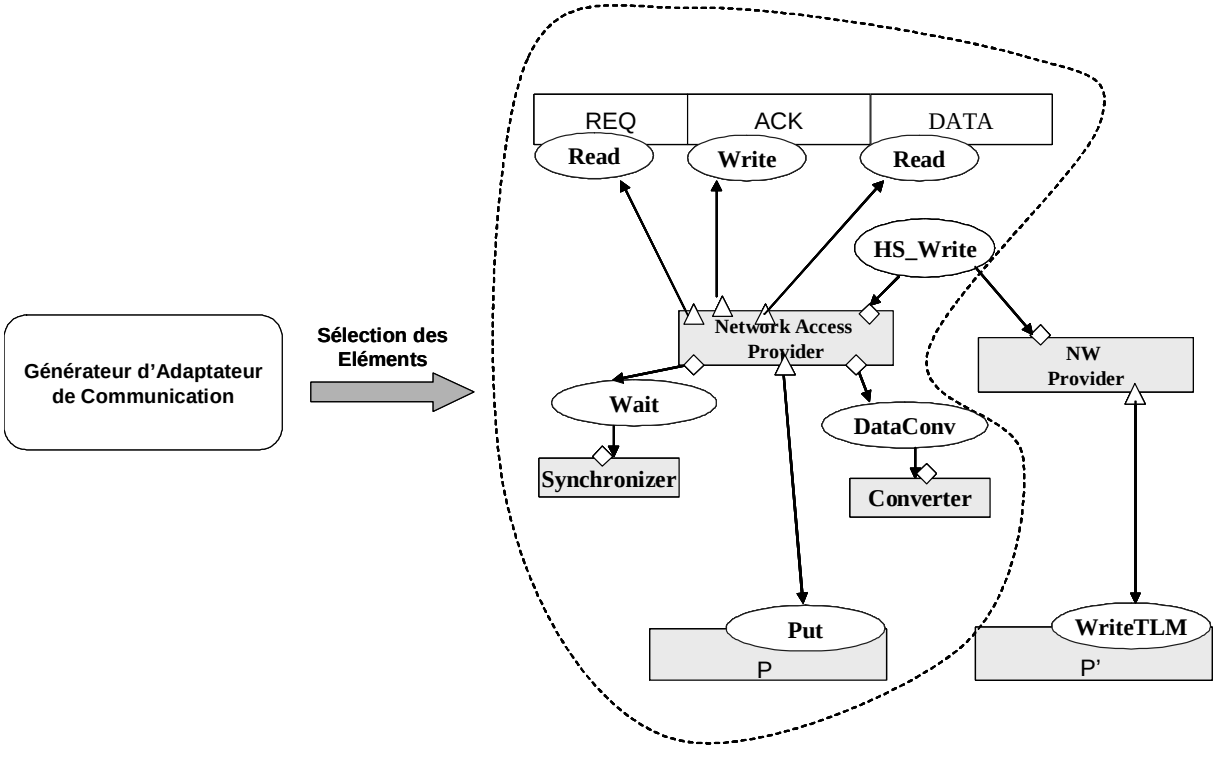


Figure 4-7 Mécanisme de Sélection des Eléments d'Interface

Les sorties du Générateur d'Adaptateur de Communication sont : une référence vers le port hiérarchique qui a besoin d'une adaptation, une liste avec les noms des éléments d'interface/ports logiques sélectionnés, les noms des services requis par chaque élément

d'interface/port logique et les noms des fichiers qui contiennent le code source de chaque composant.

Génération des adaptateurs de communication à l'aide de l'outil ASOG

Dans ce travail nous avons utilisé certaines parties de l'outil ASOG du flot ROSES pour générer les adaptateurs de communication. Les parties que nous avons intégrées dans notre flot de génération de modèles de simulation sont ceux qui concernent la construction des graphes maximaux à partir d'une description Lidel et la résolution de cycles dans un GDS.

L'outil ASOG possède une architecture constituée de cinq composants : l'Extracteur des GDS, l'Analyseur d'Architecture, le Sélectionneur de Code, le Générateur de Code et le Générateur de Makefile. Les détails de chaque composant peuvent être trouvés dans [Gau01]. Nous avons utilisés dans ce travail quelques fonctions de l'Extracteur des GDS et du Sélectionneur de Code. Le premier a été utilisé par notre Analyseur d'Architecture pour construire des GDS maximaux partant d'une description en Lidel. Le deuxième a été employé par notre Générateur de Modèle de Simulation pour éliminer les cycles dans un GDS.

Même si l'intégration d'un outil dans un autre n'est pas toujours évidente, cette intégration nous a permis de réduire le temps d'implémentation des outils du flot de génération de modèles de simulation. Cette réduction de temps est due, évidement, à la réutilisation des fonctions de grande complexité implémentées par ASOG.

Générateur de Colif

La troisième partie du Générateur de Modèle de Simulation est le Générateur de Colif. Il reçoit des informations de l'architecture du système et des adaptateurs de simulateur et de communication générés. Les informations de l'architecture sont: les modules qui font partie du système, comment les modules sont interconnectés et quels types de canaux les interconnecte. Alors que pour les adaptateurs, les informations sont : une référence vers le module ou le port hiérarchique (cela dépend du type d'adaptateur) et le contenu de l'adaptateur.

Le Générateur de Colif génère les objets Colif relatifs à ces adaptateurs (des modules Colif), choisit les canaux qui les connectent avec le reste du système, crée ces canaux en Colif et finalement sort un fichier Colif avec tous les composants (modules et adaptateurs) interconnectés.

Il faut remarquer que pendant la génération de Colif, cet outil enlève tous les ports et canaux hiérarchiques. Il les remplace pour des ports simples (non-hiérarchiques) connectés aux adaptateurs de communication (si une adaptation est nécessaire) par des canaux simples.

4.4.3 Générateur de code exécutable

Le Générateur de Code Exécutable prend comme entrée le fichier Colif avec le modèle de simulation et sort quelques fichiers de code exécutable, pour simuler le système en utilisant l'environnement SystemC. Le flot de génération de code exécutable est présenté dans la Figure 4-8

Dans un premier moment, cet outil recherche dans la Bibliothèque de Cosimulation toutes les implémentations des composants spécifiés par le modèle de simulation en Colif. Il effectue une recherche lexique de quatre types de composants existants dans la bibliothèque : les ports logiques, les canaux, les BFM et les éléments d'interface. Cette recherche lexique peut varier selon le type de composant. Chaque type de composant utilise un schéma différent de nomenclature comme suit :

-port logique – « va_*direction_niveau_protocol* »

-canal – « va_ch_*niveau_protocol* »

-BFM – « *processeur*_bfm »

-élément d'interface – « *nom* »

Les mots en italique représentent les paramètres qui sont extraits du modèle de simulation en Colif. Donc, si nous prenons comme exemple un port d'entrée au niveau d'abstraction BCA et qui implémente le protocole FIFO, ce port doit être nommé : va_in_bca_fifo.

La prochaine étape de la génération de code exécutable consiste à générer les adaptateurs de simulateurs (sauf pour les BFM). Il peut le faire pour les simulateurs VSS/VHDL et Simulink/MATLAB. Il génère un module SystemC pour l'adaptateur de l'environnement d'exécution et un FLI pour l'autre simulateur (cela peut être, par exemple, un module VHDL utilisant la bibliothèque CLI).

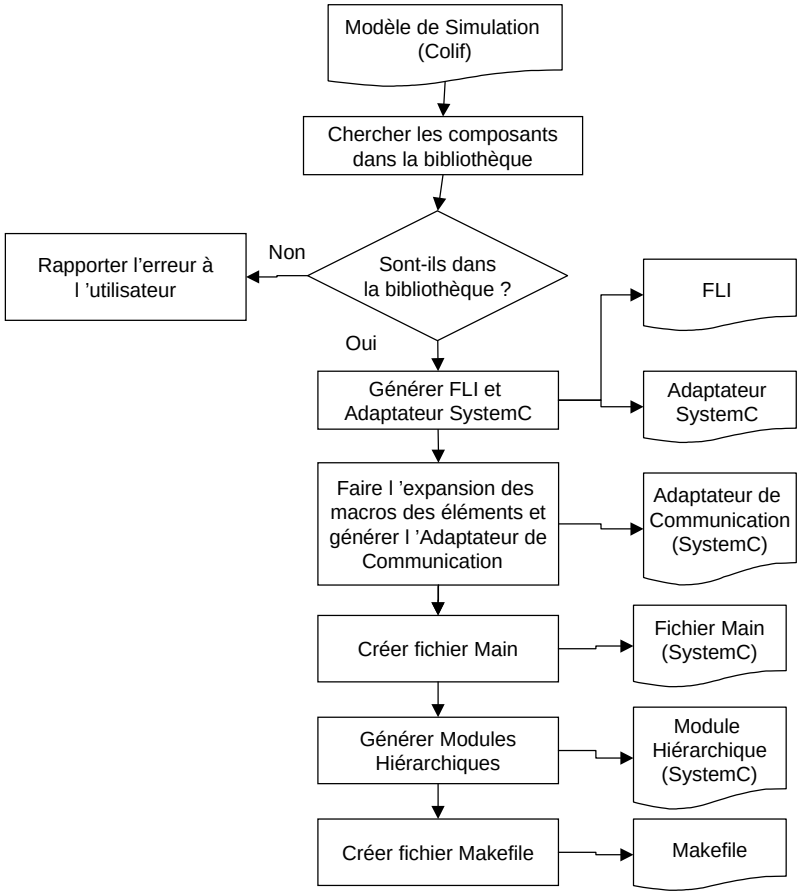


Figure 4-8 Flot de Génération de Code Exécutable

Ensuite le Générateur de Code Exécutable fait l'expansion des fichiers de macros qui implémentent les éléments d'interface. Alors, l'outil fait appel au programme d'expansion de macros en passant les paramètres suivants :

-les noms des éléments d'interface;

-les noms des ports logiques ;

-les services que chaque élément d'interface doit fournir ;

-les types de données manipulées ;

- le nom de l'adaptateur de communication auquel les éléments font partie.

Dans cette étape, l'outil crée, aussi, un module SystemC pour chaque adaptateur de communication. Ces adaptateurs font appel aux services fournis par les éléments générés.

La prochaine étape est la génération des modules SystemC pour chaque module hiérarchique du modèle de simulation. L'exception à cela, est pour les modules hiérarchiques ayant un paramètre appelé *BLACKBOX*. Normalement, ce paramètre est utilisé pour les IPs auxquels leurs contenus sont protégés.

Après cette étape, l'outil génère un fichier *main* qui interconnecte tous les composants du modèle et qui fait appel à la librairie SystemC.

Finalement, le Générateur de Code Exécutable crée un fichier *Makefile* pour la compilation du code généré.

4.5 Sorties du flot

Dans cette section, nous présentons tous les sorties qui sont disponibles à l'utilisateur à la fin du flot de génération automatique de modèles de simulation. D'abord nous allons faire un petit rappel des fichiers générés pendant le flot et puis nous montrerons quelques exemples de ces fichiers.

4.5.1 Les fichiers générés par les outils du flot

Les sorties du flot sont générées en deux moments distincts : lors de la génération du modèle de simulation en Colif et pendant la génération de code exécutable.

Le Générateur de Modèle de Simulation sort un fichier Colif représentant le modèle de simulation du système. Ce modèle sert non seulement comme entrée pour le Générateur de Code Exécutable, mais aussi pour faciliter la compréhension du modèle généré, puisqu'il peut être visualisé par l'utilisateur.

Le Générateur de Code Exécutable sort plusieurs fichiers de code exécutable dont la plupart en SystemC. Il génère un fichier SystemC pour chaque adaptateur de communication, un fichier SystemC pour chaque module hiérarchique (sauf pour les IPs, voir 4.4.3), un fichier SystemC pour chaque adaptateur SystemC, un fichier décrit en utilisant un langage étranger pour chaque FLI, un fichier SystemC pour le *main* et finalement un Makefile. Tous ces fichiers sont nécessaires pour la production de code exécutable.

4.5.2 Exemple : Un modèle de simulation en Colif et un adaptateur de communication générés pour le WSS

La Figure 4-9 montre un exemple d'un fichier sorti par le flot de génération automatique de modèles de simulation pour le WSS. Il s'agit du modèle de simulation en Colif. Nous pouvons regarder dans cette figure que plusieurs adaptateurs ont été générés permettant la communication entre les deux modules initiaux. Effectivement, à l'exception des modules qui sont marqués par l'ovale, tous les autres sont des adaptateurs de communication. Notons qu'il n'existe plus de modules avec des ports hiérarchiques. Ces ports hiérarchiques ont été remplacés par des adaptateurs de communication.

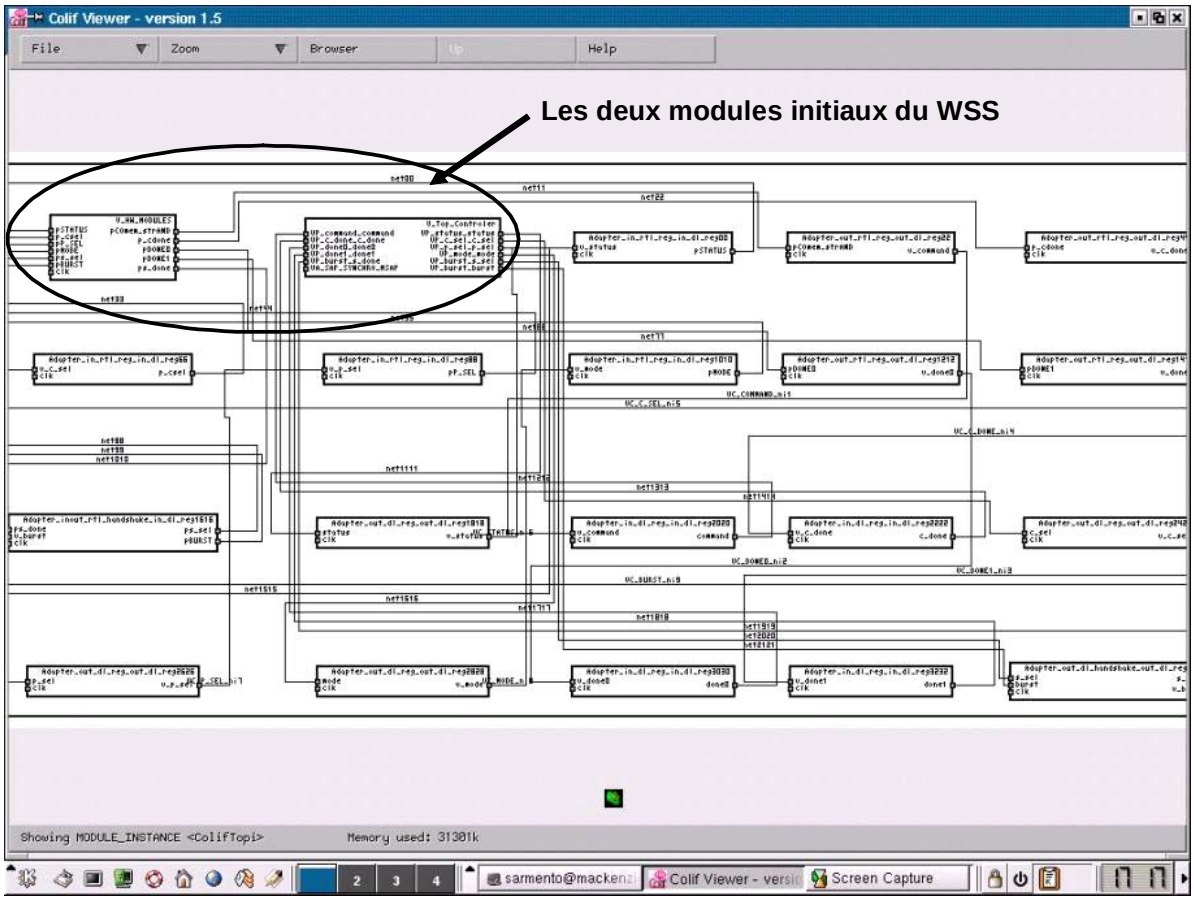


Figure 4-9 Modèle de Simulation du WSS en Colif

Le prochain exemple concerne le code exécutable d'un adaptateur de communication généré. Il est implémenté en SystemC et est présenté dans la Figure 4-10. Le code montré ici correspond à l'implémentation de l'adaptateur présenté dans la Figure 3-9a. Cet adaptateur contient une partie déclarative et une autre relative au comportement. La première est constituée du nom de l'adaptateur, et des services et des ports utilisés (lignes 1- 14). La deuxième concerne l'implémentation des services nécessaires pour l'adaptation (lignes 15-37).

```
1. class Adapter_in_rtl_handshake_bca_fifo: public sc_module {
2.   public :
3.   void HS_Read();
4.   void DataConv (int dataBef, bit2 &dataAft);
5.   sc_out<bool> s_sel;
6.   sc_in<bool> s_done;
7.   sc_out<bit2> burst;
8.   va_in_bca_fifo<int> v_burst;
9.   sc_in_clk sclk;
10.  SC_CTOR(Adapter_in_rtl_handshake_bca_fifo){
11.       SC_THREAD(HS_Read);
12.       sensitive_pos << sclk;
13.  }
14. };
15. void Adapter_in_rtl_handshake_bca_fifo::HS_Read(){
16.       bool data_read = true;
17.       while (true) {
18.         if (data_read) {
19.                 s_sel.write(true);
20.                 int dataCh;
21.                 bit2 dataMod;
22.                 dataCh = v_burst.Get();
23.                 DataConv(dataCh,dataMod);
24.                 burst.write(dataMod);
25.          }
26.          if (s_done.read()) {
27.                 data_read = true;
28.                 s_sel.write(false);
29.          } else {
30.                 data_read = false;
31.          }
32.          wait();
33.     }
34. }"
35. void Adapter_in_rtl_handshake_bca_fifo:: DataConv (int dataBef, bit2 &dataAft){
36.       dataAft = dataBef;
37. };
```

Figure 4-10 Code SystemC d'un Adaptateur de Communication du WSS

4.6 Intégration du flot de génération automatique de modèles de simulation à ROSES

Dans cette partie du chapitre nous discutons comment le flot de génération automatique de modèles de simulation proposé peut être intégré au flot ROSES. En effet, nous proposons de remplacer l'outil CosimX par le flot proposé dans ce chapitre. Nous détaillerons d'abord le flot utilisé par CosimX pour générer des modèles de simulation, et ensuite nous proposerons l'intégration de notre flot de génération au flot ROSES.

4.6.1 Le flot utilisé par CosimX pour générer des modèles de simulation

Le flot global de CosimX est présenté dans la Figure 4-11. Nous pouvons voir qu'un seul outil est chargé de lire la spécification, de faire son analyse et de générer le modèle de simulation en SystemC du système. Le manque de plusieurs étapes et outils pour générer ces modèles de simulation pénalise la flexibilité du flot proposé par CosimX. Le découpage du

flot en étapes d'analyse d'architecture, de génération de modèles de simulation et de génération de code exécutable est essentiel pour qu'un flot de tel type ne soit pas si dépendent du langage de spécification d'entrée ou de l'environnement d'exécution. Si nous adoptons, par exemple, un environnement d'exécution différent de SystemC, il faut faire plusieurs modifications dans CosimX. Dans CosimX, la génération de modèles de simulation est faite directement de la représentation Colif d'entrée en SystemC, sans passer d'abord par un format intermédiaire. Ce qui rend difficile l'adaptation de l'outil à différents environnements d'exécution.

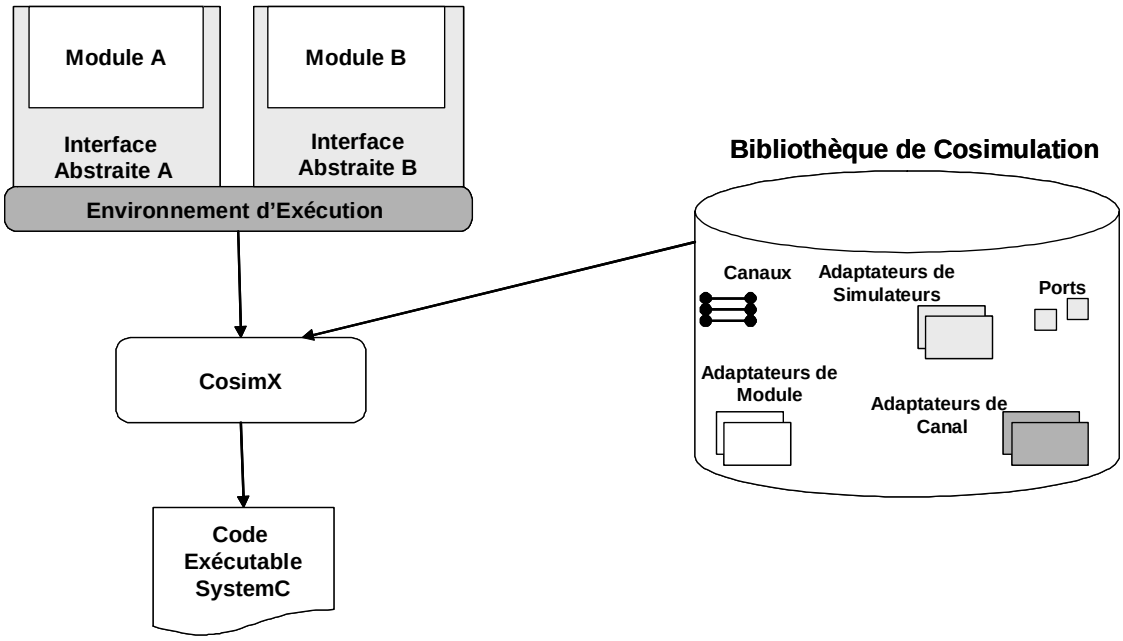


Figure 4-11 Flot de Génération de Modèles de Simulation par CosimX

4.6.2 Le nouveau flot de génération automatique des modèles de simulation dans ROSES

En raison des limitations présentés par CosimX, nous proposons d'intégrer notre flot de génération automatique de modèles de simulation au flot ROSES, pour que l'utilisateur puisse générer des adaptateurs de communication plus flexibles et pour faciliter l'intégration de différents langages de spécification et environnements d'exécution dans ROSES. Cette intégration est illustrée par la Figure 4-12.

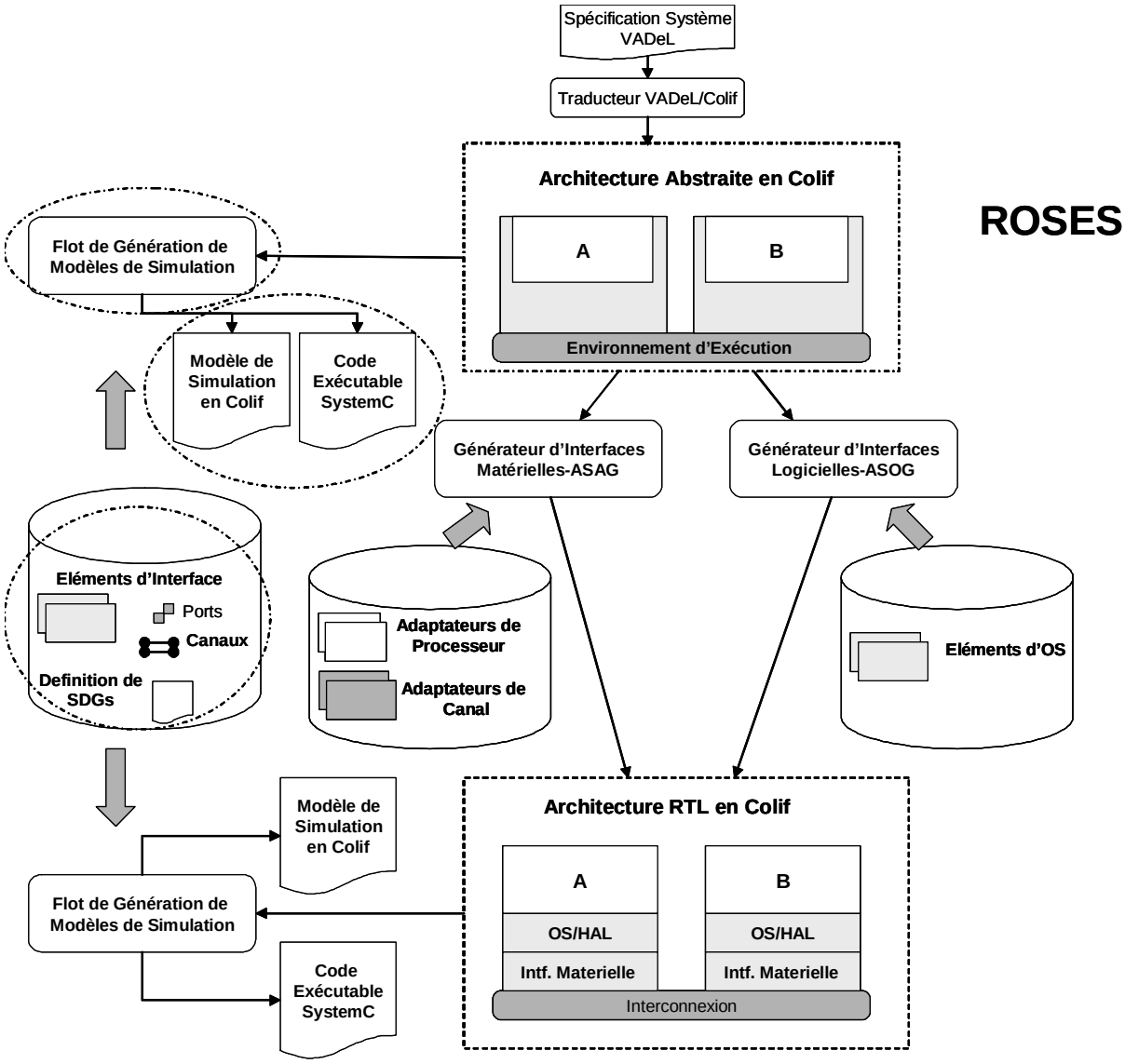


Figure 4-12 Nouvelle Façon de Génération de Modèles de Simulation dans ROSES

Dans la Figure 4-12, nous pouvons remarquer que CosimX est remplacé par le flot de génération de modèles de simulation que nous proposons, et que la Bibliothèque de Cosimulation ne contient plus des adaptateurs de module et de canal, mais des éléments d'interfaces pour composer les adaptateurs de communication. Il y a également une sortie en plus qui est le modèle de simulation en Colif.

Le coût d'adopter cette nouvelle façon de générer des modèles de simulation, dans un premier moment, est de remplacer les adaptateurs de module et de canal par des éléments d'interface ainsi que les décrire en Lidel. D'ailleurs, pour que les adaptateurs de communication implémentent seulement les services strictement nécessaires, il faut que l'utilisateur apprenne un langage de macro comme l'explique la section 4.4.

4.7 Conclusion

Dans ce chapitre nous avons proposé un flot de génération automatique de modèles de simulation pour systèmes hétérogènes embarqués. Ce flot nous permet de générer un modèle de simulation pour chaque étape de la conception de systèmes hétérogènes embarqués en partant d'une architecture abstraite du système. Les modèles de simulation sont générés par l'assemblage de composants de base existants dans une bibliothèque. La principale caractéristique de ce flot est sa flexibilité. En découpant le flot en plusieurs étapes, il n'est pas si dépendant du langage de spécification d'entrée ou de l'environnement d'exécution. Nous avons vu, aussi, que la génération des adaptateurs de communication se fait en utilisant le modèle basé sur les services.

Nous avons présenté, également, les problèmes liés à la méthode de génération de modèles de simulation utilisée actuellement dans le flot ROSES. Nous avons proposé une intégration de notre flot de génération automatique de modèles de simulation dans le flot ROSES. Les avantages de cette intégration sont de permettre la génération des adaptateurs de communication plus flexibles et de faciliter l'intégration des différents langages de spécification et environnements d'exécution dans le flot ROSES.

Chapitre 5 : Résultats expérimentaux

5.1 Introduction

Ce chapitre est consacré à l'application de notre flot de génération automatique de modèles de simulation pour valider deux systèmes hétérogènes embarqués : un modem VDSL et un encodeur MPEG-4. Les modèles de simulation générés contiennent des adaptateurs de communication qui sont générés en utilisant le modèle basé sur les services. Donc les expérimentations présentées ici, servent pour évaluer non seulement le flot, mais aussi la modélisation basée sur les services des adaptateurs de communication.

Ce chapitre est divisé comme suit. La première partie présente le flot de validation utilisé pour les deux études de cas. Ensuite, nous présentons les résultats expérimentaux concernant la validation du modem VDSL. La troisième partie présente les résultats obtenus par la validation de l'encodeur MPEG-4. Et la dernière partie est consacrée à l'évaluation du modèle basé sur les services pour implémenter des adaptateurs de communication et aussi du flot de génération automatique de modèles de simulation, par rapport aux résultats expérimentaux.

De manière générale, pour chaque système embarqué abordé dans ce chapitre, nous faisons une présentation générale, ensuite l'architecture abstraite est présentée, puis les modèles de simulation pour chaque étape du flot de validation sont montrés et enfin nous faisons une analyse des résultats obtenus.

5.2 Le flot de validation utilisé dans les expérimentations

Dans cette première partie du chapitre, nous présentons le flot de validation utilisé pour réaliser les expérimentations développées tout au long de ce chapitre. Ce flot est illustré par la Figure 5-1. La partie gauche de la figure montre les étapes de conception du système requis, alors que la partie droite présente les types de validation réalisés pour chaque étape de conception.

Le flot de conception débute par la spécification sous forme d'architecture abstraite du système. Le système est composé de modules à différents niveaux d'abstraction qui communiquent par un réseau de communication qui peut être aussi à un niveau différent d'abstraction par rapport aux modules. Dans les deux études de cas réalisés, les IPs matériels étaient au niveau RTL, tandis que les modules logiciels étaient à un niveau plus élevé. Dans ce cas, nous réalisons une validation multi-niveaux pour vérifier si les échanges de données et de contrôle entre les modules sont corrects.

L'étape suivante consiste à raffiner les modules et la communication. Donc, nous avons utilisé l'outil ASOG pour générer des interfaces logicielles pour les modules logicielles et l'outil ASAG pour raffiner le réseau de communication et générer les interfaces matérielles adaptant ces modules au réseau. Les interfaces logicielles générées sont composées de deux parties : une qui est indépendante du processeur (OS) et l'autre spécifique au processeur (HAL). Dans cette étape, le HAL généré est spécifique à la machine hôte. Autrement dit, les interfaces logicielles générées servent pour l'exécution native du code. Ainsi, nous avons réalisé une validation RTL avec exécution native du code logiciel, pour valider les interfaces matérielles et la partie générique de l'interface logicielle (OS). L'avantage de réaliser un tel type de validation est le gain de vitesse par rapport à une simulation en utilisant un ISS.

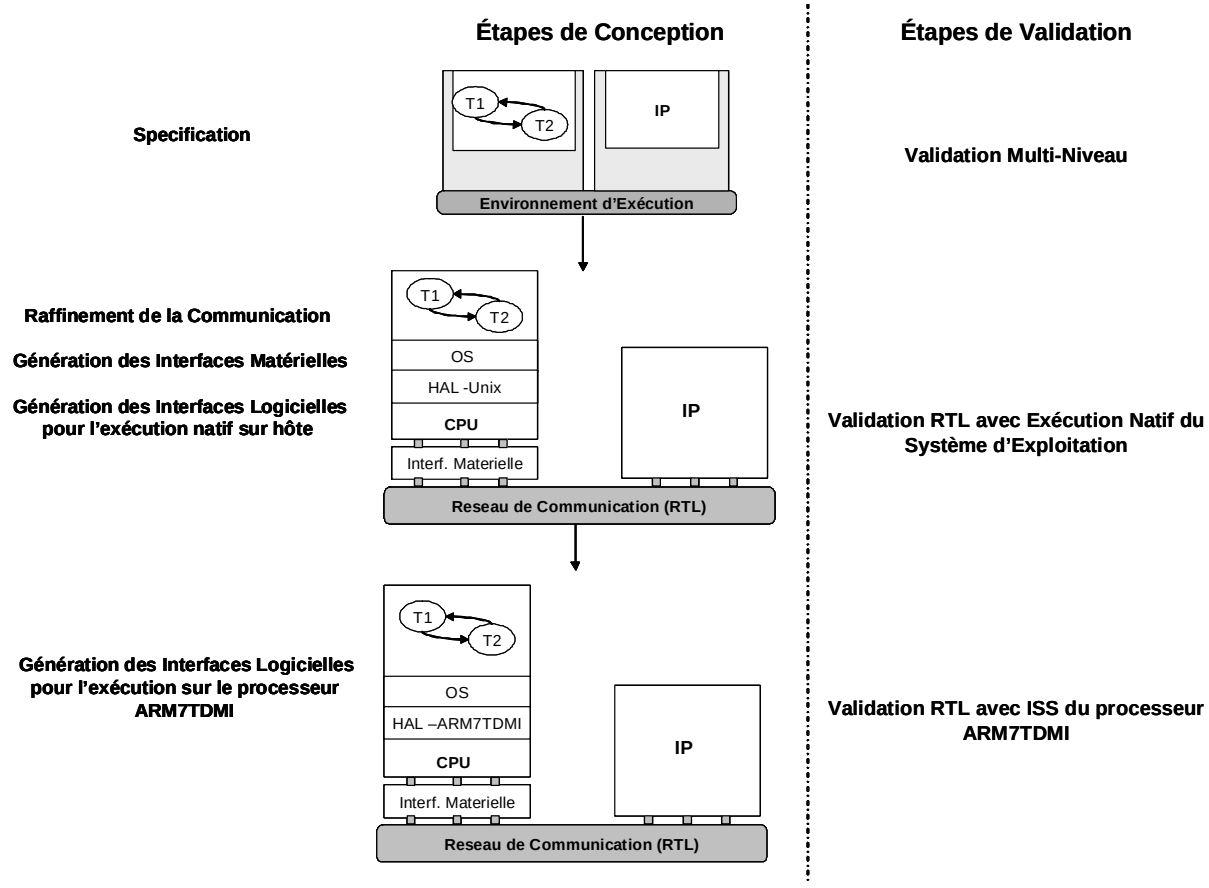


Figure 5-1 Flot de Validation Utilisé Pour les Expérimentations

La dernière étape de ce flot est la génération des interfaces logicielles qui permettent l'exécution du code de l'application sur le processeur réel du système. Encore une fois, nous avons utilisé ASOG pour générer le HAL spécifique au processeur réel du système (dans

notre cas, le processeur choisi a été l'ARM7TDMI). Ainsi, nous procédons à une validation RTL avec l'ISS du processeur pour valider l'implémentation finale du système.

5.3 Un modem VDSL

Dans cette section, nous présentons la première expérimentation que nous avons conduit pour ce travail. Elle consiste à valider un modem VDSL tout au long d'un flot de conception.

5.3.1 Présentation du modem VDSL

VDSL (venant de l'anglais Very high bit rate Digital Subscriber Line) est une technologie de communication qui sert pour le transport numérique de l'information sur une simple ligne téléphonique de paire torsadée [Vdsl04]. Elle fait partie des gammes de technologies connues comme xDSL, dont font partie aussi l'ADSL, le HDSL et le SDSL. Ce qui distingue VDSL des autres xDSL sont ses débits plus élevés et aussi sa capacité d'adapter ce débit selon l'état (longueur, bruit, etc.) de la ligne téléphonique. Avec cette technologie, l'utilisateur peut téléphoner et se connecter à l'Internet en même temps en utilisant une seule prise téléphonique. Sans recâblage ni changement de poste téléphonique, l'utilisation d'un modem spécifique est suffisante pour profiter de cette technologie.

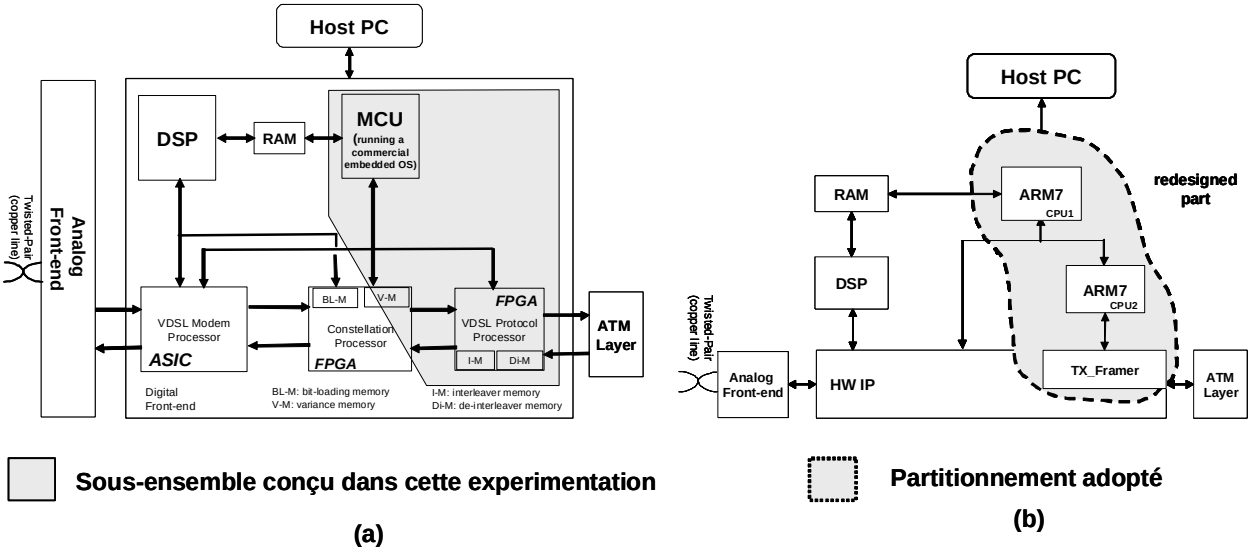


Figure 5-2 a) Modem VDSL b) Partitionnement Adopté pour Concevoir le VDSL

La Figure 5-2a présente l'architecture du modem VDSL qui a été utilisé pour cette expérimentation [Mes00]. Elle est composée de trois blocs matériels, un DSP et un

microcontrôleur. Le trois blocs matériels et le DSP sont en charge de la réception, l'émission, le codage et le recodage des signaux. Le microcontrôleur est responsable de plusieurs taches telles que : mesurer le débit maximal supporté par la ligne, interfacer le PC hôte au modem, et aussi contrôler, configurer et synchroniser la chaîne de transmission en fonction de l'état de la ligne. Pour cette expérimentation, nous n'avons utilisé que la partie hachurée de la Figure 5-2a. La fonctionnalité du sous-ensemble hachuré a été partitionné en deux processeurs ARM7TDMI et un pipeline de blocs matériels qui réalisent la chaîne de transmission. Ce partitionnement a été fait afin de rendre plus flexible la chaîne de traitement réalisé par l'assemblage de blocs matériels. Ainsi le deuxième processeur configure dynamiquement le chemin de donnés du bloc matériel. La Figure 5-2b montre ce partitionnement qui nous a été suggéré par l'équipe qui a réalisé le modem VDSL [Mes00].

5.3.2 L'architecture abstraite du modem VDSL

L'architecture abstraite du modem VDSL est présentée par la Figure 5-3. Elle est composée d'un module matériel (HW) et deux modules logiciels (SW1 et SW2). Le module matériel est représenté par une IP qui communique au niveau RTL en utilisant des protocoles comme *handshake*, *enable-handshake* et *registre*. Alors que les deux modules logiciels sont composés de plusieurs taches et de primitives de communications au niveau BCA. L'environnement d'exécution connectant SW1 et HW à SW2 utilise des *FIFO* au niveau BCA. En concernant les langages de spécification, les différents modules ont été spécifiés en SystemC.

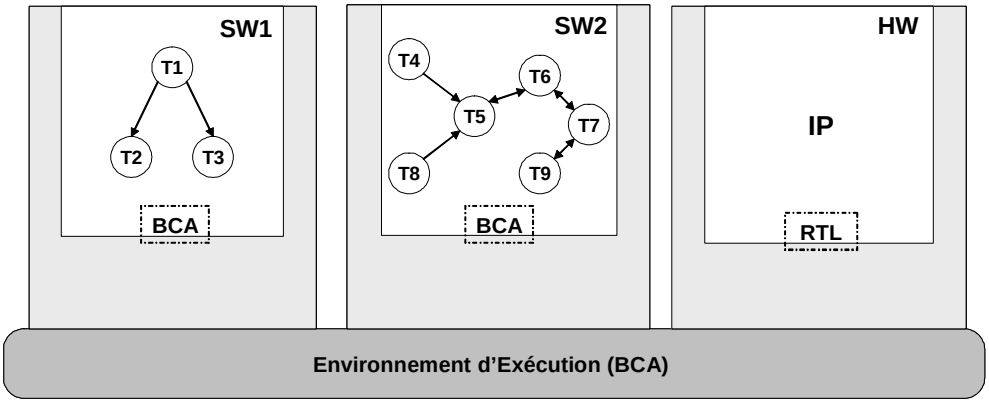


Figure 5-3 L'Architecture Abstraite du Modem VDSL

Des interfaces abstraites sont nécessaires pour permettre la communication entre HW et SW2 parce que ceux-ci sont spécifiés à différents niveaux d'abstraction et utilisent

différents protocoles de communication. En effet, même s'il n'y a pas de communication entre deux modules (comme par exemple entre SW1 et HW), ces interfaces abstraites sont nécessaires puisqu'elles représentent l'abstraction des interfaces logicielles/matérielles qui seront générées lors du raffinement des modules et de la communication.

5.3.3 Validation multi-niveaux du modem VDSL

Pour que le module HW puisse communiquer au module SW2 en utilisant l'environnement d'exécution, il faut des adaptations. Tandis que l'environnement d'exécution est implémenté par un *FIFO* bloquante fournissant des services *Get*/*Put* pour lecture/écriture, le module HW requiert des services de communication au niveau RTL comme le montre le Tableau 5-1. Quatre services de communication sont requis par le module HW : une écriture façon *Enable-Handshake* (*EHS_Write*), une lecture façon *Handshake* (*HS_Read*) et lecture/écriture de registre (*Reg_Read/Reg_Write*). Nous pouvons remarquer que des adaptations de type de données sont également nécessaires. Par contre, comme l'environnement d'exécution et le module HW utilisent SystemC, aucune adaptation en termes de langage de spécification n'est nécessaire. Donc, pour que nous puissions faire une validation multi-niveau, il ne suffit que de générer des adaptateurs de communication permettant la communication entre le module HW et l'environnement d'exécution.

Adaptateurs		Enable-Handshake/FIFO Bloquante	Handshake/FIFO Bloquante	Registre(Read)/ FIFO Bloquante	Registre(Write)/ FIFO Bloquante
Module (HW)	**Niveau d'Abstraction**	RTL			
	Protocole	Enable-Handshake	Handshake	Registre	Registre
	Type de Données	Bit_vector[2]	Bit_vector[8]	Bit_vector[n]	Bit_vector[n]
	Service Requis	EHS_Write	HS_Read	Reg_Read	Reg_Write
Environnement d'Exécution	**Niveau d'Abstraction**	BCA			
	Protocole	FIFO Bloquante			
	Type de Données	Integer			
	Services Fournis	Get/Put			

Tableau 5-1 Les Types d'Adaptateurs Nécessaires pour la Validation Multi-Niveaux du VDSL

Le modèle de simulation multi-niveaux

La Figure 5-4a illustre le modèle de simulation obtenu par notre flot de génération automatique. Le seul type d'adaptateur présent dans ce modèle est celui de communication. Le module HW est connecté à plusieurs adaptateurs de communication et le bus de cosimulation est composé des canaux *FIFO* bloquantes. La Figure 5-4b présente deux exemples d'adaptateurs de communication générés : un adaptant le service *EHS_Write* au service *Put*, et l'autre qui adapte le service *HS_Read* au service *Get*. Nous pouvons remarquer que le premier est très similaire à celui que nous avons implémenté pour le WSS (voir chapitre 3). La seule différence est que pour le WSS, l'environnement d'exécution utilisait des *FIFO* non bloquantes pour fournir le service *Put*. Donc cet adaptateur de communication du VDSL est implémenté par les mêmes éléments d'interface que celui du WSS, mais avec un port logique différent.

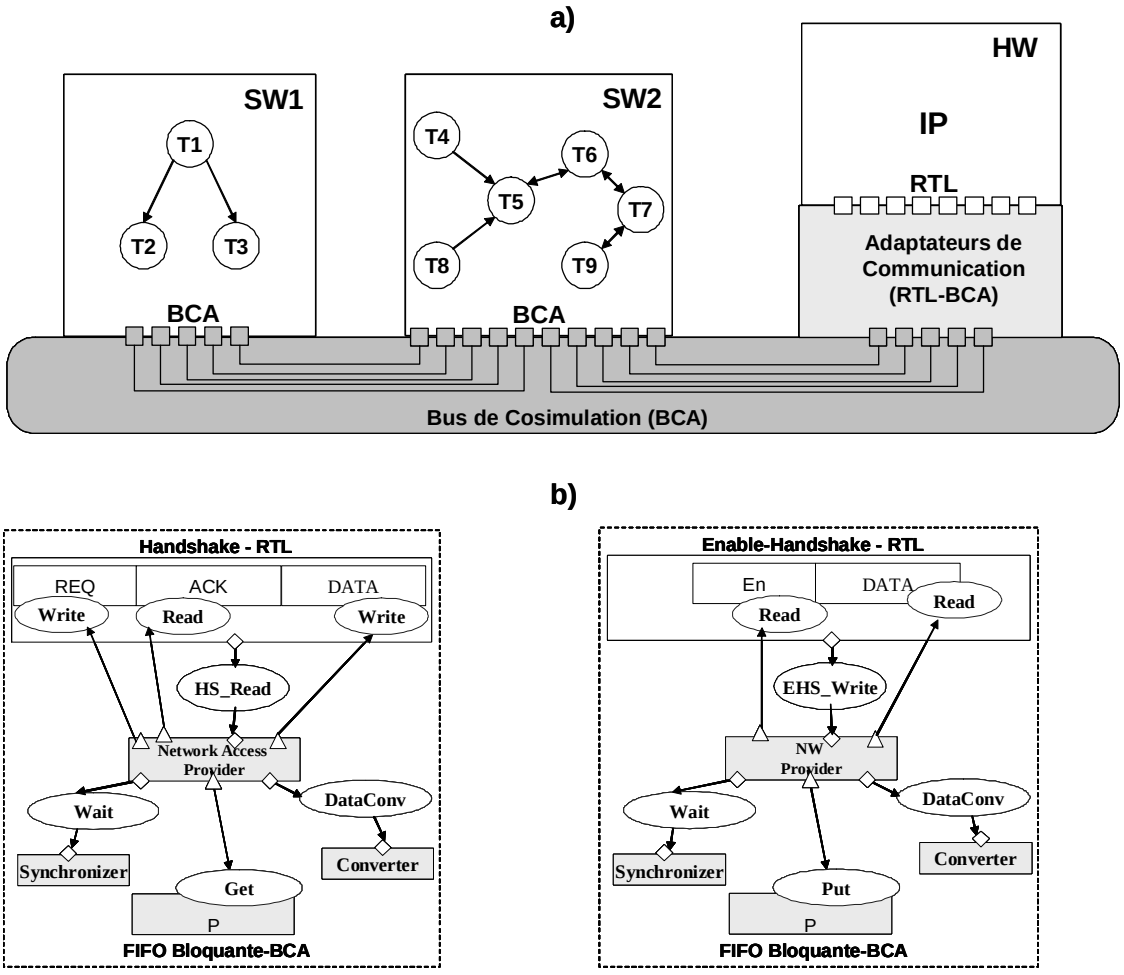


Figure 5-4 a) Modèle de Simulation Multi-Niveau du VDSL b) Exemples des Adaptateurs de Communication Générés

Le deuxième adaptateur requiert un service *EHS_Write* qui est fourni par l'élément d'interface *NW_Provider*. Cet élément a été ajouté à la Bibliothèque de Cosimulation pour cette expérimentation. Les autres éléments sont les mêmes utilisés par le premier adaptateur.

Les deux autres adaptateurs de communication générés dans cette étape de validation sont pour adapter les services *Reg_Read /Reg_Write* aux services *Get/Put.* Encore une fois, ces adaptateurs sont presque identiques à ceux générés pour le WSS qui adaptent les services *Reg_Read /Reg_Write* aux *Get/Put* non bloquantes.

Résultats de la validation multi-niveaux

Pour effectuer la validation multi-niveaux du VDSL, seulement un élément d'interface a été ajouté comme le montre le Tableau 5-2. La plupart des éléments d'interface ont été implémentés pour l'exemple du WSS. En concernant les ports logiques utilisés pour implémenter les adaptateurs de communication, nous avons réutilisé ceux qui étaient présents dans la Bibliothèque de Cosimulation de CosimX. Ceci démontre la flexibilité et la facilité de réutilisation apportée par le modèle basé sur les services. Le Tableau 5-2 montre que seulement 11 lignes de code ont du être ajoutées manuellement. Les 11 lignes ajoutées par l'utilisateur représentent moins de 10% du total de lignes requis pour tous les adaptateurs de communication.

No. d'Eléments Utilisés (Eléments et Ports Logiques)	No. d'Eléments ajoutés à la bibliothèque	No. de lignes total de code des adaptateurs	No. de lignes de code ajouté manuellement
6	1	125	11

Tableau 5-2 Eléments Nécessaires pour les Adaptateurs du VDSL

Le Tableau 5-3 présente d'autres résultats de la génération du modèle de simulation multi-niveaux. Il montre le nombre de lignes de code ajouté par l'utilisateur et celui-ci généré automatiquement. Le Tableau 5-3 montre également le temps estimé pour écrire ces lignes de code. Ici, nous considérons que pour écrire 20 lignes de code correct, il faut 1 personne/jour. Ainsi, avec la génération automatique du modèle de simulation, l'utilisateur n'a à écrire que 176 lignes de code au lieu des 460 lignes nécessaires pour le modèle de simulation. Cela représente une réduction par un facteur à peu près de 3 fois du temps et de l'effort d'implémentation. Le temps de génération de ce modèle en utilisant une machine Linux 2.0 GHz est d'environ 1 minute.

	No. de lignes de code	Temps Estimé de Codage (jours)
Architecture Abstraite	150	7,5
Description des GDS	15	0,8
Elément Ajouté	11	0,5
Modèle de Simulation Généré	460	23

Tableau 5-3 Résultats de la Génération du Modèle de Simulation Multi-Niveaux du VDSL

5.3.4 Validation RTL du modem VDSL avec l'exécution native du système d'exploitation

La validation RTL du VDSL avec l'exécution native du système d'exploitation a été effectuée après le raffinement de la communication, la génération des interfaces matérielles et la génération des interfaces logicielles spécifiques à la machine hôte. A ce moment-là, tous les modules étaient au niveau RTL et les adaptations de protocole et type de données étaient faites par les interfaces matérielles. Ainsi, ce type de validation ne nécessite aucun adaptateur de communication. Par contre, les modules logiciels en SystemC ont été remplacés par le code de l'application en C qui s'exécute sur un système d'exploitation spécifique à la machine hôte (ce qui s'appelle l'exécution native du code). Ce qui veut dire que les modules logiciels utilisent un environnement d'exécution (UNIX) différent de celui du reste du système (SystemC). Ainsi, il faut adapter les différents environnements d'exécution (ou autrement dit, simulateurs) pour permettre la communication entre tous les modules.

Le Modèle de simulation RTL- exécution native du système d'exploitation

La Figure 5-5 présente le modèle de simulation généré pour cette étape de validation. Ce modèle a été généré à partir de l'architecture RTL en Colif fournie par ASAG. Dans ce modèle, les modules logiciels sont connectés aux interfaces matérielles par des adaptateurs de simulateurs. Chaque module logiciel est en réalité le code C de l'application et le système d'exploitation (OS + HAL-Unix) compilés pour la machine hôte. Les adaptateurs de simulateur permettent l'échange de données entre le code C qui s'exécute sur la machine hôte et les interfaces matérielles simulées par l'environnement SystemC. Les interfaces matérielles et le module HW sont connectés par le bus de cosimulation composé des signaux RTL.

Ces adaptateurs de simulateur ont été implémentés par le FLI spécifique à l'environnement d'exécution UNIX de la machine hôte et l'adaptateur de l'environnement d'exécution SystemC existants dans la Bibliothèque de Cosimulation de CosimX.

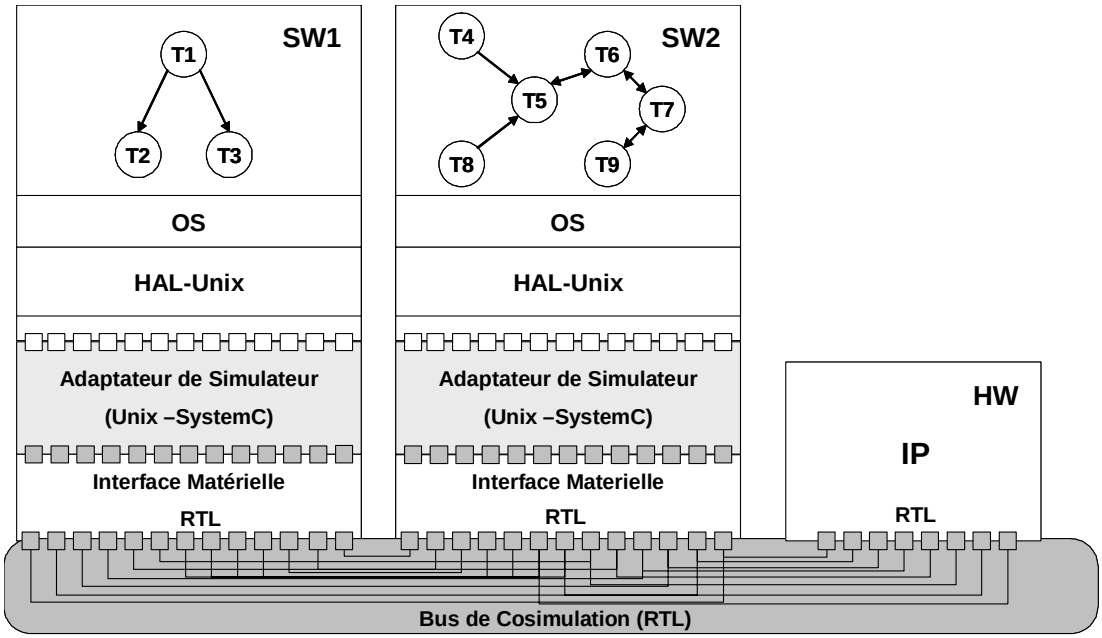


Figure 5-5 Modèle de Simulation du VDSL - Exécution Native du Système d'Exploitation

Résultats de la validation RTL- exécution native du système d'exploitation

Dans cette partie de l'expérimentation aucun composant n'a été ajouté à la Bibliothèque de Cosimulation, une fois que nous avons utilisé des composants existants dans celle de CosimX. Le Tableau 5-4 montre les résultats de cette intégration. Pour générer des interfaces logicielles/matérielles, le concepteur doit ajouter des paramètres de configuration de telles interfaces à l'architecture abstraite de départ. Ceci étant, le total de lignes de code apportée par ces nouveaux paramètres est 200. Apres l'application d'ASAG et d'ASOG à cette architecture abstraite, nous avons générés un modèle de simulation de 1447 lignes de code. Ceci représente une réduction du temps et de l'effort d'implémentation de ce modèle par un facteur à peu près de 7. Finalement, l'outil a pris environ 2 minutes pour générer ce modèle RTL.

.

	No. de lignes de code	**Temps Estimé de Codage (jour)**
Paramètres ajoutés à l'Architecture Abstraite pour générer les interfaces logicielles/matérielles	200	10
Modèle de Simulation Généré	1447	72

Tableau 5-4 Résultats de la Génération du Modèle de Simulation RTL du VDSL

5.3.5 Validation RTL du modem VDSL avec ISS

Apres la génération des interfaces logiciels spécifiques au processeur ARM7TDMI, nous avons effectué la validation RTL avec l'ISS spécifique au processeur.

Le modèle de simulation RTL-ISS

Le modèle de simulation généré dans cette étape est très semblable à celui de l'étape précédente, comme le montre la Figure 5-6. La différence est que le code logiciel (code de l'application + système d'exploitation spécifique au ARM) de chaque processeur s'exécute sur un ISS ARM. Nous avons, alors, utilisé un adaptateur de simulateur spécifique à l'ISS. En effet, la seule partie différente de cet adaptateur par rapport à celui de l'étape précédente, est le FLI. L'Adaptateur d'Environnement SystemC reste le même. Encore une fois, ces deux composants étaient déjà présents dans la Bibliothèque de Cosimulation. Ainsi, ils ont été connectés automatiquement au modèle de simulation RTL.

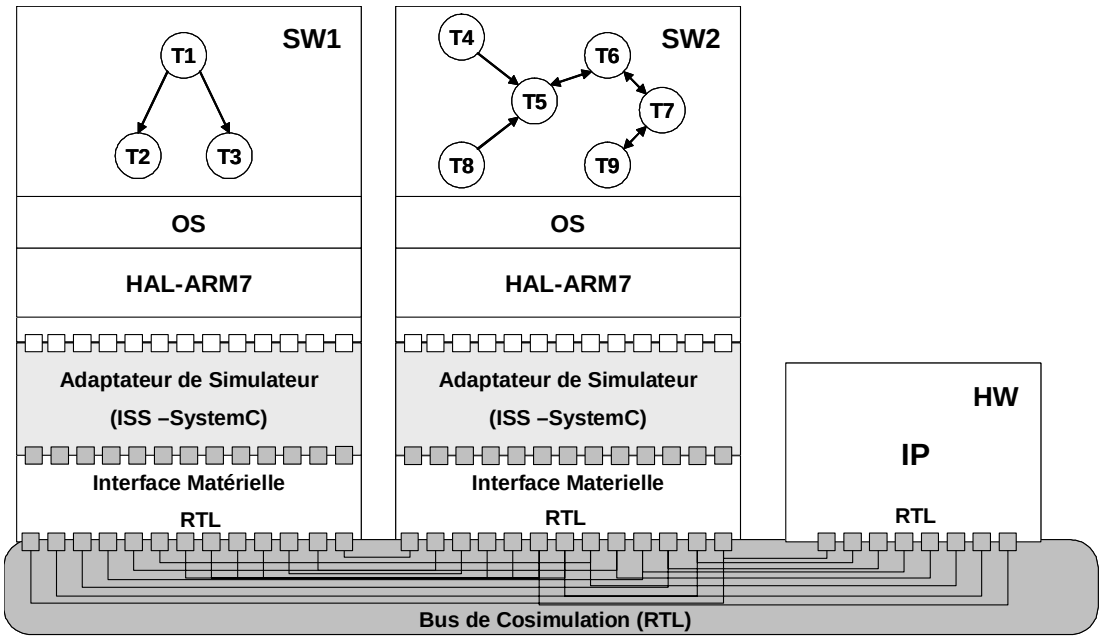


Figure 5-6 Modèle de Simulation du VDSL avec ISS -ARM7

Résultats de la validation RTL-ISS

Concernant les résultats de la génération du modèle de simulation, ils ont été les même qui ceux obtenus dans l'étape précédente. Par rapport à la vitesse de simulation, évidement la simulation avec ISS est plus lente que celle avec exécution native du code logiciel. La perte de vitesse se situe de 2 ordres de grandeur.

5.3.6 Evaluation des résultats

Cette expérimentation nous a permis de valider certains aspects du travail que nous proposons :

- Le flot de génération automatique de modèles de simulation sert pour valider des systèmes hétérogènes embarqués à plusieurs étapes de conception.
- Ce flot représente un facteur de réduction de temps et d'effort importante pour la validation de systèmes hétérogènes embarqués.
- Les avantages du modèle d'adaptateur de communication basé sur les services : gain en flexibilité et en réutilisation de composants de base (ports logiques et éléments d'interface).

5.4 Un encodeur MPEG-4

Dans cette partie du chapitre, nous présentons la deuxième expérimentation que nous avons effectuée pour ce travail. Elle consiste à valider un encodeur MPEG-4 tout au long d'un flot de conception.

5.4.1 Présentation de l'encodeur MPEG-4

MPEG-4 est un standard utilisé pour coder des informations audio-visuelles dans un format numérique compressé. Il fait partie de la famille de standards développé par le groupe MPEG (venant de l'anglais Moving Picture Expert Group), où font partie de même les standards MPEG-1 et MPEG-2. Un avantage important du format MPEG-4 par rapport à d'autres formats est la taille réduite des fichiers produits pour une même qualité d'image/son. Ceci est dû aux techniques avancées de compression de données utilisées par MPEG-4 [Mpe04].

L'algorithme de codification MPEG-4 demande un grand nombre de calculs, notamment l'estimation de mouvement (en anglais motion estimation), la compensation de mouvement (motion compensation), la transformation en cosinus discrète (DCT), la quantification et enfin la compression des données [Mpe04a] [Tou00].

Dans cette expérimentation, nous avons pris un encodeur MPEG-4 qui a été implémenté comme un système hétérogène embarqué. Le group qui a conçu ce système a décidé de partitionner l'encodage MPEG-4 en deux blocs [You04], comme le montre la Figure 5-7a. Le premier bloc est responsable pour la partie d'encodage et le deuxième est chargé de la compression. En plus, pour gagner en vitesse de calcul, les concepteurs ont

décidé de paralléliser les calculs d'encodage sur quatre processeurs ARM7 (voir Figure 5-7b). Chacun réalise le même type de calcul et est responsable de coder une partie de l'image (processeurs VPROC dans la Figure 5-7b). Pour la même raison, les concepteurs ont utilisé deux processeurs ARM7 pour accélérer la compression de l'image (VLC). Le système possède aussi trois blocs matériels : un pour lire l'image d'entrée (INPUT) ; l'autre pour regrouper les parties d'image après l'encodage et la compression (Combiner) ; et le troisième pour recevoir/transmettre les parties des images des/aux processeurs (Memory Server).

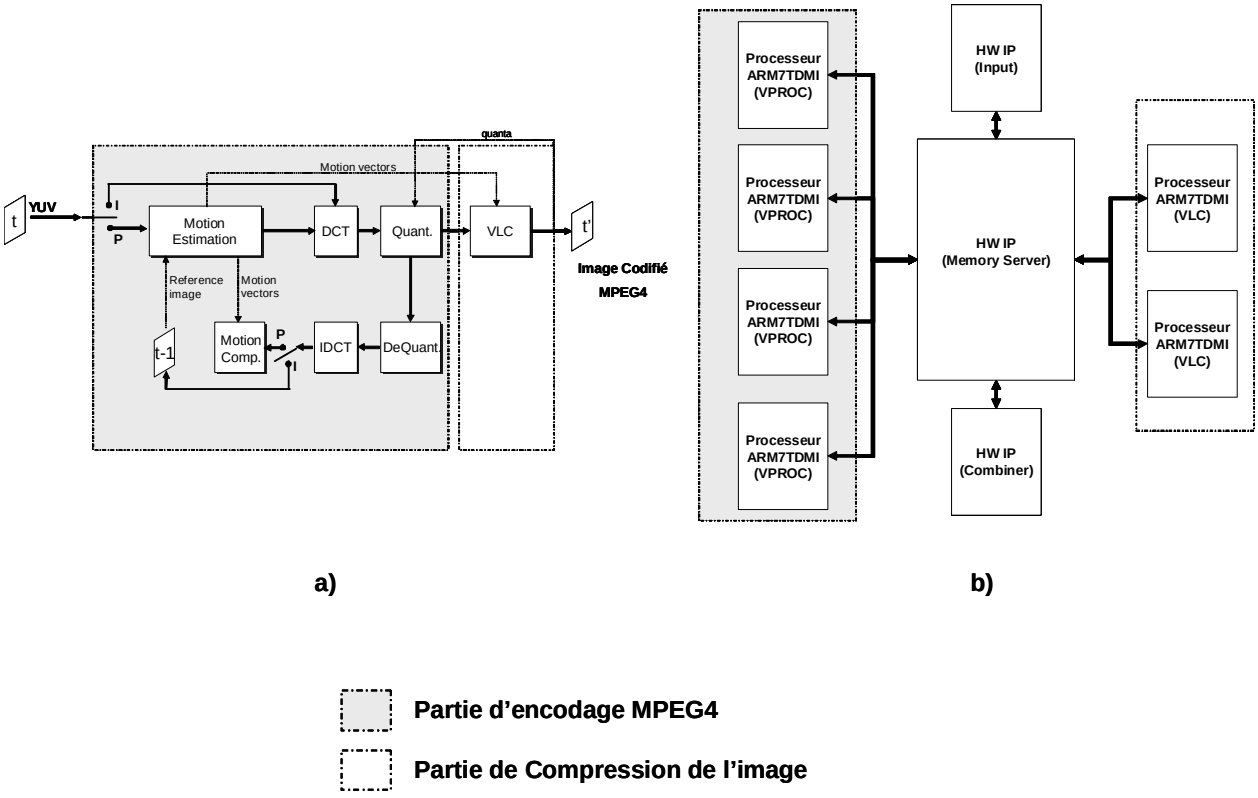


Figure 5-7 a) L'Encodeur MPEG-4 b) Partitionnement de l'Application

5.4.2 L'architecture abstraite de l'encodeur MPEG-4

La Figure 5-8 présente l'architecture abstraite initiale de l'encodeur MPEG-4. Afin de rendre plus simple cette architecture, nous avons regroupé tous les modules matériels dans un seul. Ces modules matériels sont décrits au niveau RTL et le Memory Server communique avec tous les modules logiciels. Les modules logiciels (4 VPROCs et 2 VLCs) utilisent des services de communication au niveau TLM, qui sont fournis par l'environnement d'exécution.

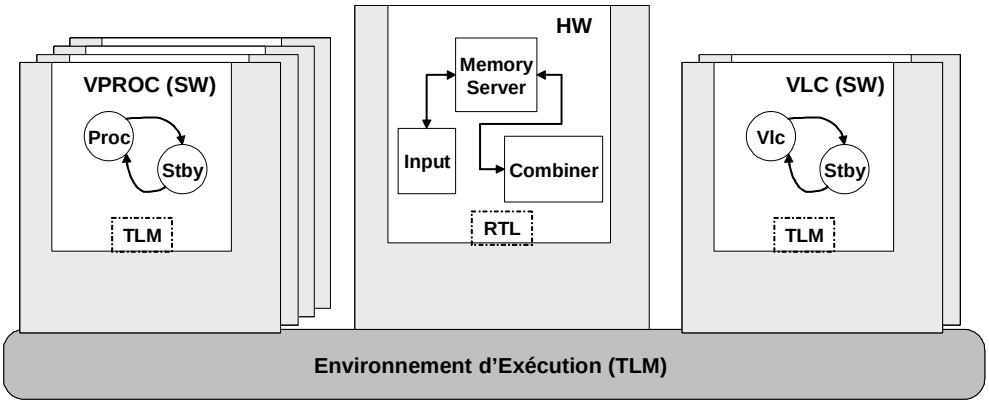


Figure 5-8 L'Architecture Abstraite de l'Encodeur MPEG-4

Ces services de communication TLM peuvent être classés en deux types : (1) services de transfert de données et (2) services de contrôle. Le premier type sert pour les transferts de données entre les VPROCs ou VLCs et le Memory Server, tandis que le deuxième type est utilisé pour synchroniser ces transferts. Le module Memory Server nécessite, donc, d'une interface abstraite pour qu'il puisse utiliser les services de communication fournis par l'environnement d'exécution. Enfin, en ce qui concerne les langages de spécification des modules, tous ont été spécifiés en SystemC.

5.4.3 Validation multi-niveaux de l'encodeur MPEG-4

La validation multi-niveaux de l'encodeur MPEG-4 a été effectuée en générant les adaptateurs de communication permettant la communication entre le MemoryServer et les modules logiciels.

Le Tableau 5-5 montre les différents types d'adaptation nécessaires pour cette étape de validation. L'environnement d'exécution fournit les services *ReadDataTLM/WriteDataTLM* pour le transfert de données et les services *ReadEventTLM/WriteEventTLM* pour la synchronisation de ces transferts. Pourtant, il y a deux implémentations différentes de *ReadDataTLM/WriteDataTLM*. Une possède deux mémoires tampon pour stoker les données (protocole MBMC ou Multiple Bank Memory Controller) et l'autre possède seulement une mémoire tampon (protocole SBMC ou Single Bank Memory Controller). Le Memory Server utilise le premier pour communiquer avec les modules VPROCs et le deuxième pour les VLCs.

Trois services de communication sont requis par le Memory Server : une écriture/lecture utilisant un protocole propre à ce module (*MemoryBusIO),* une lecture du nombre d'événements produits par les modules logiciels (*Get_Count*) et finalement une

écriture d'un événement (*Notify*). Encore une fois, nous pouvons remarquer que des adaptations de type de données sont également nécessaires.

Adaptateurs		MemoryServer Bus/MBMC	MemoryServer Bus/SBMC	EventCounter/ Event	Event(RTL)/ Event(TLM)
Module (Memory Server)	**Niveau d'Abstraction**	RTL			
	Protocole	MemoryServer Bus	MemoryServer Bus	EventCounter	Event
	Type de Données	Bit_vector[32]	Bit_vector[32]	Bit_vector[2]	bit
	Service Requis	MemoryBusIO	MemoryBusIO	Get_Count	Notify
Environnement d'Exécution	**Niveau d'Abstraction**	TLM			
	Protocole	MBMC	SBMC	Event	
	Type de Données	Long Integer	Long Integer	Integer	
	Services Fournis	ReadDataTLM/ WriteDataTLM	ReadDataTLM/ WriteDataTLM	ReadEventTLM/ WriteEventTLM	

Tableau 5-5 Les Types d'Adaptateurs Nécessaires pour la Validation Multi-Niveau de l'Encodeur MPEG-4

Le modèle de simulation multi-niveaux

La Figure 5-9 illustre le modèle de simulation obtenu pour la validation multi-niveaux. Dans ce modèle, plusieurs adaptateurs de communication sont connectés au module Memory Server et permettent, ainsi, les transferts de données et la synchronisation entre lui et les modules logiciels. Le bus de cosimulation est composé de plusieurs canaux TLM qui implémentent les services de communication TLM spécifiés dans l'architecture abstraite.

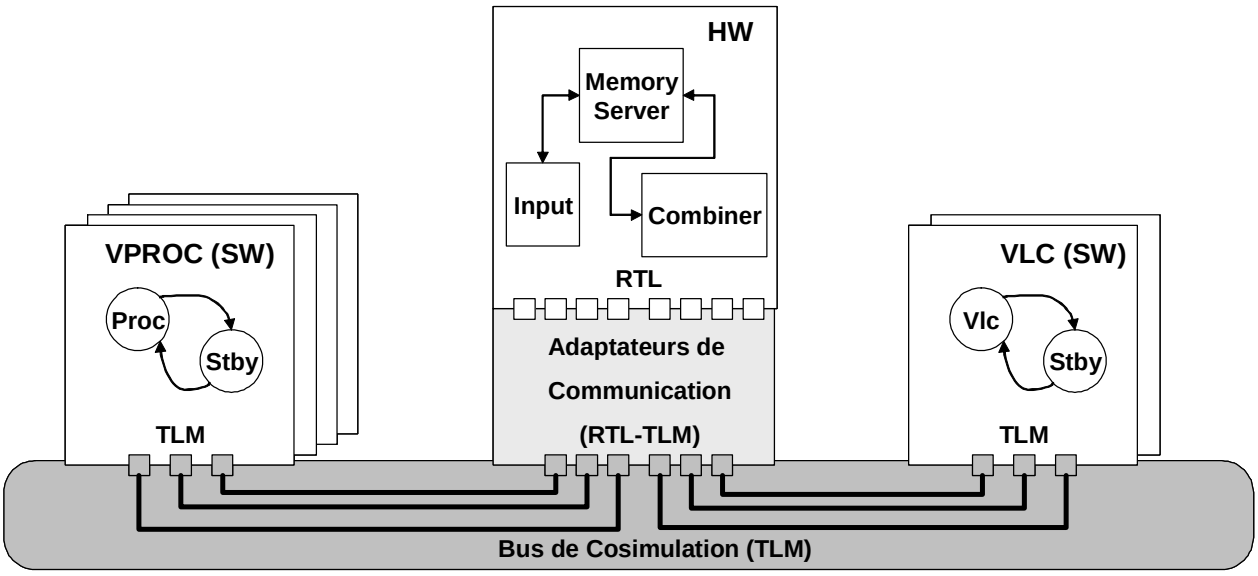


Figure 5-9 Modèle de Simulation Multi-Niveaux de l'Encodeur MPEG-4

La Figure 5-10 présente le GDS représentant les adaptateurs de communication nécessaires pour le transfert de données entre le Memory Server et les VPROCs, et entre le Memory Server et les VLCs. Concernant le transfert de données, l'interface du Memory Server est composée de cinq ports : (1) *EN* informe si le Memory Server est prêt pour commencer un transfert de données; (2) RW concerne le type de transfert - lecture ou écriture ; (3) ADR donne l'adresse où le Memory Server va lire ou écrire les données ; (4) DIN est le port de réception des données ; (5) DOUT est le port d'envoi des données. Les données sont transférées en utilisant les services *ReadDataTLM/WriteDataTLM* qui assemblent les différents éléments d'interface et ports logiques comme le montre la Figure 5-10. Ces adaptateurs sont plus complexes que ceux générés pour le VDSL, parce qu'il faut aussi un décodage d'adresse. Notons, aussi, qu'ils sont presque identiques, à l'exception des ports qui fournissent les services *ReadDataTLM/WriteDataTLM* de chacun. Le premier adaptateur contient un port logique MBMC, alors que l'autre possède un port SBMC. Pour la génération de ces adaptateurs, nous avons ajouté trois éléments d'interface (*MemoryBusIOProvider*, *TLMNetworkProvider* et *MSB_MBMC*) et deux ports logiques (Port MBMC et Port SBMC).

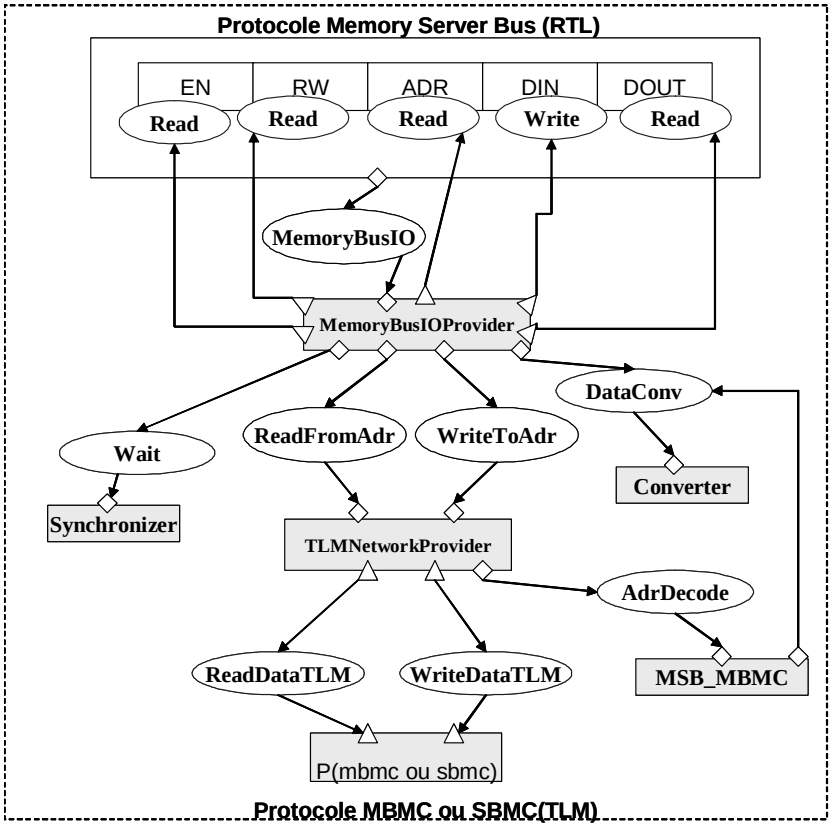


Figure 5-10 Adaptateur de Communication pour Permettre le Transfert des Données entre Memory Server, VPROC et VLC

La Figure 5-11 présente les adaptateurs nécessaires pour la synchronisation des transferts de données entre les différents modules. L'adaptateur à gauche permet au Memory Server de savoir si les processeurs ont fini leurs calculs pour qu'il puisse initier un transfert de données. Cet adaptateur, alors, compte le nombre d'événements envoyés par les processeurs. L'adaptateur à droite permet au Memory Server d'envoyer un événement informant aux processeurs qu'ils peuvent commencer l'exécution de leur calculs. Nous pouvons remarquer que ces deux adaptateurs utilisent le même port pour accéder aux services ReadEventTLM/WriteEventTLM. Nous avons ajouté deux éléments d'interface pour générer ces adaptateurs (*NWEventCounter* et *NWEventHandler*). En ce qui concerne le port TLM Event, nous l'avons pris de la bibliothèque de CosimX.

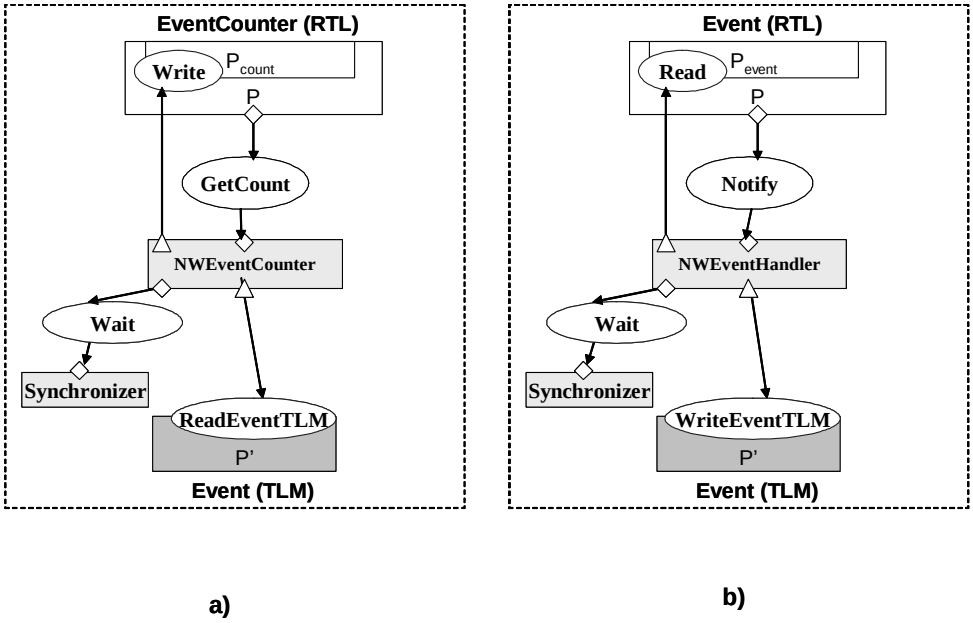


Figure 5-11 Adaptateurs de Communication pour Permettre la Synchronisation de Transfert de Données entre Memory Server, VPROC et VLC

Résultats de la validation multi-niveaux

Pour faire la validation multi-niveaux de l'encodeur MPEG-4, nous avons ajouté 7 composants (5 éléments d'interface et 2 ports logiques) comme le montre le Tableau 5-6. Le deuxième adaptateur utilise tous les éléments d'interface du premier, ainsi seul un port logique a été ajouté pour l'implémenter. La flexibilité et la facilité de réutilisation du modèle basé sur les services sont prouvées encore une fois. Le changement du protocole de l'environnement d'exécution n'implique que de remplacer une partie de l'adaptateur. Le Tableau 5-6 montre, également, que 101 lignes de code ont du être ajoutées manuellement

pour implémenter les éléments constituant les adaptateurs requis, tandis que le total de lignes de code des adaptateurs a été 238. Ceci représente moins de 50% du total de lignes de l'adaptateur de communication.

Adaptateurs de Communication	No. de Composants Utilisés (Eléments et Ports)	No. de Composants ajoutés à la bibliothèque	No. de lignes de code des adaptateurs	No. de lignes de code ajouté manuellement
MemoryServerBus – MBMC	6	4	78	62
MemoryServerBus – SBMC	6	1	78	19
EventCounter- Event	3	1	44	13
Event (RTL)- Event(TLM)	3	1	38	7
Total	18	7	238	101

Tableau 5-6 Eléments Nécessaires pour les Adaptateurs de l'Encodeur MPEG-4

Le Tableau 5-7 présente les résultats de la génération du modèle de simulation multi-niveaux. Il montre le nombre de lignes de code ajouté par le concepteur et celui généré automatiquement. La génération automatique du modèle de simulation apporte une réduction du temps et de l'effort d'implémentation d'à peu près 50%. Le modèle de simulation a 1197 lignes de code, alors que le total de lignes de code écrites par le concepteur est 651.
Le temps de génération de ce modèle est environ 1 minute comme pour le VDSL.

	No. de lignes de code	Temps Estimé de Codage (jours)
Architecture Abstraite	450	22,5
Description des GDS	100	5
Eléments + Ports Ajoutés	101	5
Modèle de Simulation Généré	1197	60

Tableau 5-7 Résultats de la Génération du Modèle de Simulation Multi-Niveaux de l'Encodeur MPEG-4

5.4.4 Validation RTL de l'encodeur MPEG-4 avec l'exécution native du système d'exploitation

Ainsi comme l'expérimentation avec le modem VDSL, pour effectuer la validation RTL de l'encodeur MPEG-4 avec l'exécution native du système d'exploitation aucun adaptateur de communication n'a été nécessaire. Nous n'avons généré que les adaptateurs de simulateur.

Le modèle de simulation RTL- Exécution native du système d'exploitation

La Figure 5-12 présente le modèle de simulation généré automatiquement dans cette étape de validation. Comme pour le VDSL, chaque module logiciel est connecté à un adaptateur de simulateur. La seule différence est le nombre de modules logiciels qui participent à la cosimulation, qui dans ce cas est six.

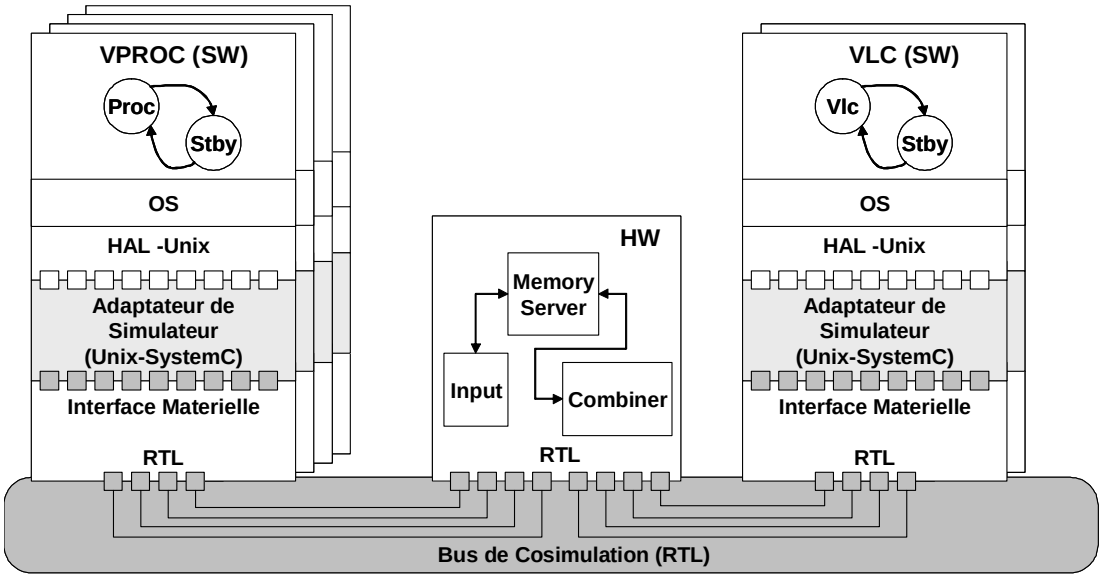


Figure 5-12 Modèle de Simulation de l'Encodeur MPEG-4 –Exécution Native du Système d'Exploitation

Résultats de la validation RTL- exécution native du système d'exploitation

Les résultats de la génération automatique du modèle de simulation RTL avec l'exécution native du système d'exploitation sont présentés dans le Tableau 5-8. Les paramètres ajoutés à l'architecture abstraite de départ pour la génération des interfaces logicielles/matérielles par ASOG et ASAG font un total de 400 lignes de code. Avec la génération automatique du modèle de simulation RTL, nous avons réduit le temps d'implémentation par un facteur de 5. Le temps de génération reste presque le même que pour le VDSL, d'environ 2 minutes.

	No. de lignes de code	Temps Estimé de Codage (jour)
Paramètres ajoutés à l'Architecture Abstraite pour générer les interfaces logicielles/matérielles	400	20
Modèle de Simulation Généré	2078	103

Tableau 5-8 Résultats de la Génération du Modèle de Simulation RTL de l'Encodeur MPEG-4

5.4.5 Validation RTL de l'encodeur MPEG-4 avec ISS

La validation RTL de l'encodeur MPEG-4 avec ISS a été effectuée après la génération des interfaces logicielles spécifiques au processeur ARM7TDMI.

Le modèle de simulation RTL-ISS

La Figure 5-13 montre le modèle de simulation RTL en utilisant l'ISS ARM7. Nous avons remplacé automatiquement les six adaptateurs de simulateur spécifiques à l'environnement UNIX par ceux spécifique à l'ISS ARM7.

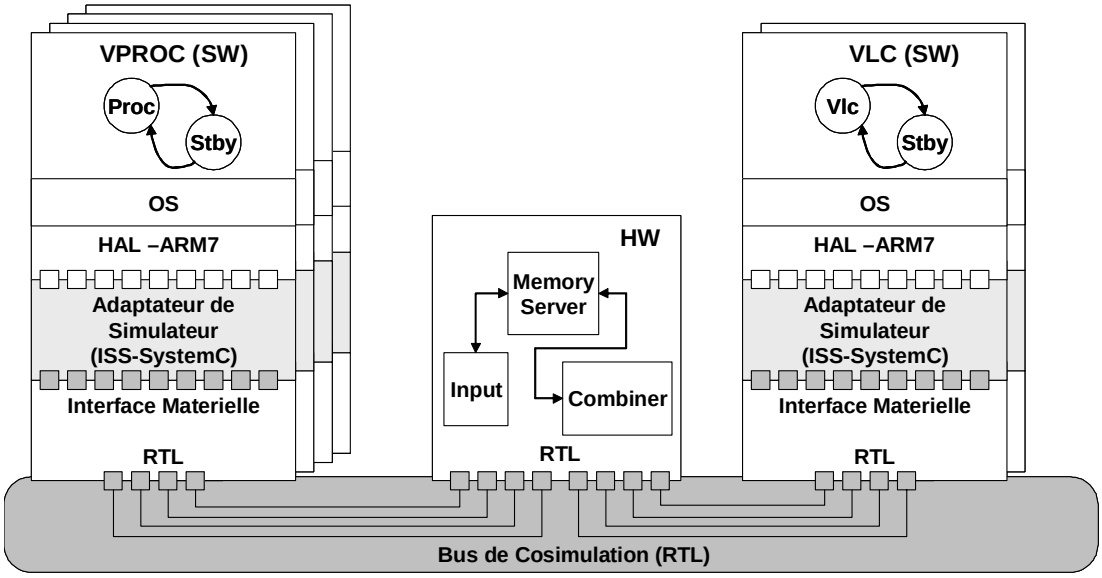


Figure 5-13 Modèle de Simulation de l'Encodeur MPEG-4 avec ISS -ARM7

Résultats de la validation RTL-ISS

En ce qui concernent les résultats de la génération du modèle de simulation, ils ont été les mêmes obtenus dans l'étape précédente. Mais la présence des nombreux ISSs a ralenti le temps de simulation énormément. Juste pour donner une idée de ce ralentissement, le temps pour encoder 25 images (frames) en utilisant l'exécution native du code logiciel est de 10 minutes, tandis que cela dure 25 heures avec l'ISS.

5.4.6 Evaluation des résultats

La validation de l'encodeur MPEG-4 nous a permis de renforcer les conclusions que nous y sommes arrivés dans l'expérimentation du VDSL. En plus, l'encodeur MPEG-4 nous a permis de générer des modèles de simulation en partant d'une spécification à un niveau plus élevé (en utilisant TLM). Les adaptateurs de communication ont été plus complexes, ce qui a prouvé l'efficacité de les générer en utilisant le modèle basé sur les services.

5.5 Evaluation de la génération automatique de modèles de simulation dans les cas présentés

Les expérimentations nous ont permis de faire une analyse quantitative sur les deux contributions apportées par ce travail : le modèle d'adaptateur de communication basé sur le services et le flot de génération automatique de modèles de simulation. Cette dernière partie du chapitre est consacrée à l'évaluation des avantages et des limitations de ces deux sujets.

5.5.1 Les avantages du modèle d'adaptateur de communication basé sur les services par rapport à d'autres approches

Les résultats obtenus par la validation multi-niveaux du modem VDSL et de l'encodeur MPEG-4 ont mis en évidence la plupart des avantages que le modèle d'adaptateur de communication basé sur les services apporte. Ces avantages sont discutés par la suite.

Flexibilité

Les expérimentations ont démontré la flexibilité du modèle que nous proposons par rapport à d'autres approches. Différemment des modèles basés sur bus ou protocoles standard, le modèle basé sur les services n'impose aucune restriction par rapport aux protocoles de communication utilisés par les différents modules et l'environnement d'exécution. Les modules et l'environnement d'exécution des deux systèmes validés utilisaient plusieurs protocoles de communication.

L'autre point à remarquer, c'est la facilité de changer l'adaptateur en cas de changement du protocole utilisé par le module et/ou l'environnement d'exécution. Ceci a été le cas des adaptateurs de communication de l'encodeur MPEG-4 illustrés par la Figure 5-10, où il a été suffisant de changer juste un port logique. L'apporte de cette flexibilité par le modèle basé sur les services est essentielle pour faciliter l'étape d'exploration d'architecture lors de la conception de systèmes. Pour d'autres modèles basés sur l'assemblage des composants, notamment celui utilisé par CosimX, l'effort pour changer un adaptateur est plus important, dû à la grosse granularité de leurs composants de base. Comme comparaison, si le concepteur utilise CosimX dans le même cas, il doit changer l'adaptateur de canal, ce qui correspond à plusieurs éléments d'interface (le port logique inclus).

Enfin dans le modèle proposé, un service peut être fourni par plusieurs éléments qui par ailleurs peut contenir plusieurs implémentations. Ce qui augmente la flexibilité du modèle, une fois que le concepteur peut choisir l'implémentation que lui convient le plus.

Réutilisation des composants de base

Les résultats des expérimentations attestent aussi que le modèle basé sur les services facilite la réutilisation des composants de base. La fine granularité de ce modèle est le point clé pour cela. Dans les adaptateurs de communication présentés, plusieurs éléments ont été réutilisés comme par exemple le convertisseur de type de donnés (*DataConv*), l'élément de synchronisation, les ports, etc. Les autres modèles d'adaptateur de communication (basés sur bus, protocole standard ou assemblage de composants) possèdent des composants de base trop gros où plusieurs fonctionnalités y sont imbriquées. Cela réduit la réutilisation de ces composants.

Performance

Les performances des adaptateurs de communication n'ont pas été quantifiées dans les expérimentations. Pourtant, l'utilisation du modèle basé sur les services implique des adaptateurs de communication avec des performances plus élevés, car l'adaptateur ne contient que les services essentiels pour la fonction d'adaptation. Par contre, la grosse granularité des composants de base des autres modèles d'adaptateur de communication pénalise la performance, en incluant dans leurs composants des services qui ne sont pas nécessaires.

5.5.2 Les limitations du modèle d'adaptateur de communication basé sur services

La grande flexibilité apportée par le modèle d'adaptateur de communication sur les services peut poser des problèmes en ce qui concerne la définition des services et éléments d'interface. Il n'existe pas de règle stricte pour les définir, c'est-à-dire, il faut avoir une sorte de savoir-faire pour définir la bonne taille du service/élément pour qu'il soit réutilisable et pas très grande. Cette sorte de limitation est moins fréquente dans les autres modèles d'adaptateur de communication, parce qu'ils ont une structure plus « linéaire ».

L'autre limitation du modèle proposé est que la performance globale de l'adaptateur de communication n'est pas assurée. Cela se produit à cause de la caractéristique « bottom-up » de ce modèle pour composer des interfaces. Le modèle facilite des optimisations de

chaque élément, mais pas de l'adaptateur entier, autrement dit nous ne pouvons que fournir une optimisation locale.

5.5.3 Les avantages globaux du flot de génération automatique de modèles de simulation

En réalisant les deux expérimentations nous avons pu constater les avantages globaux du flot de génération automatique de modèles de simulation proposé dans ce travail.

Réduction du temps de validation

Même si la plupart du temps de validation d'un système est consacré au déboguage, la génération de modèles de simulation représente une partie importante du temps de ce processus. Avec le flot proposé, le concepteur est capable de générer en quelques minutes ces modèles à partir d'une architecture abstraite et d'une description des GDS des éléments/services. Le temps de génération de tels modèles (qui incluent le temps consacré à la spécification) en utilisant notre flot a été réduit en moyenne par un facteur de 5.

Validation multi-niveaux

Nous avons réussi à faire des validations multi-niveaux en utilisant le flot proposé. Les deux systèmes validés dans les expérimentations étaient composés de modules décrits en différents niveaux d'abstraction qui communiquaient en utilisant différents protocoles. En étant capable de réaliser une telle validation, une partie significative des erreurs des systèmes peut être découverte aux étapes initiales de la conception. Ceci représente, aussi, une réduction importante du temps de validation.

Validation à plusieurs étapes de la conception

Le flot de génération automatique de modèles de simulation peut être appliqué à différentes étapes de validation, comme les expérimentations l'ont montré. Nous pouvons générer des modèles de simulation en partant d'une spécification de haut niveau, comme cela a été le cas pour l'encodeur MPEG-4 (en utilisant TLM), ou à partir d'une spécification RTL. Cela permet au concepteur une validation de meilleure qualité, une fois qu'il peut valider chaque étape du processus de conception. En plus, le flot sort des modèles de simulation dont la vitesse et la précision sont variables. Autrement dit, le flot est capable de générer des

modèles de simulation plus rapides mais moins précis au début du processus de conception, ou des modèles moins rapides mais plus précis lorsque le système est au niveau RTL.

5.5.4 Les limitations globaux du flot de génération automatique de modèles de simulation

Une limitation de ce flot est, évidement, qu'il s'appuie sur une bibliothèque des composants de base. Le concepteur, donc, doit ajouter divers composants de nature différente (ports logiques, éléments d'interface, adaptateurs de simulateur et canaux) pour que les outils du flot puissent générer les modèles de simulation. La complexité de chaque composant peut varier selon son type. Si, par exemple, l'implémentation d'un élément d'interface peut être facile, cela n'est toujours pas le cas pour un adaptateur de simulateur. En raison de cela, selon le système, le concepteur peut passer une partie significative du temps de conception en implémentant des composants pour la bibliothèque.

L'autre limitation importante de ce flot est la diversité de langages de spécification (ou de programmation) qu'un concepteur doit apprendre pour que les outils du flot puissent générer des modèles de simulation. Pour spécifier l'architecture abstraite et les GDS, il faut apprendre Colif (ou VADeL) et Lidel respectivement. En ce qui concerne l'implémentation des éléments d'interface, le concepteur nécessite des connaissances des langages de macro (Rive dans notre cas). Et finalement, pour tous les composants de base, il faut connaître au moins une langage de programmation.

5.5.5 Les avantages et limitations du flot de génération automatique de modèles de simulation par rapport à CosimX

Dans la section précédente, nous avons évalué d'une manière générale les avantages et les limitations de notre flot dans les expérimentations que nous avons conduit. Dans cette section nous nous concentrons à analyser les avantages ainsi que les limitations de notre flot par rapport à CosimX.

Les entrées pour la génération

En utilisant CosimX, le concepteur ne doit fournir que l'architecture abstraite du système. Pour notre flot, il doit fournir en plus une description des GDS. Même si cette description ne représente pas un effort important par rapport à celui de spécifier une architecture abstraite, cela reste une limitation du flot en comparaison à CosimX.

La bibliothèque de cosimulation

Au niveau de la Bibliothèque de Cosimulation, la seule différence concerne les composants qui implémentent les adaptateurs de communication. Les adaptateurs de module et canal de CosimX sont implémentés comme des modules SystemC. Le concepteur doit, donc, écrire pour chaque type d'adaptateur (de module et de canal) un module SystemC qui contient le comportement de l'adaptateur, ses ports et ses signaux. En plus, il doit écrire des fonctions qui connectent ces adaptateurs au reste du système. Par exemple, ce type de fonction pour l'adaptateur de module prendre comme paramètres les ports du module dont il va se connecter.

Toute cette démarche n'est pas nécessaire dans le flot que nous proposons. Même si nous implémentons aussi les adaptateurs de communication comme des modules SystemC, le concepteur n'a à écrire que les parties (éléments d'interface) qui composent le comportement de ces adaptateurs. Lors de la génération, les éléments sont rassemblés et les outils du flot s'occupent de générer un module SystemC (avec tous ses ports et signaux) pour chaque adaptateur de communication. Ainsi, l'effort requis de la part du concepteur pour ajouter les composants de base des adaptateurs de communication dans notre approche est normalement moins important que celui de CosimX.

La flexibilité du processus de génération

Nous avons discuté la flexibilité de notre flot par rapport à CosimX dans le chapitre 4. C'est vrai que dans les expérimentations que nous avons réalisé, cette flexibilité n'est pas évidente puisque nous avons utilisé les mêmes langages d'entrée et de sortie (Colif et SystemC respectivement) que CosimX. Pourtant, lors de l'implémentation des outils de notre flot, nous avons fait une première version de l'outil de génération de modèle de simulation en Colif (voir chapitre 4) qui a employé le modèle d'adaptateur de communication de CosimX. Nous avons réussi à appliquer le générateur de code exécutable tel quel il était aux modèles de simulation en Colif générés. Ce qui permet de constater le gain de flexibilité de notre approche en comparaison à CosimX. Encore une fois, le découpage du processus de génération en trois étapes distinctes dans notre flot est essentiel pour ce gain de flexibilité.

Le temps de génération

Le temps de génération des modèles de simulation de notre flot est plus grand que celui de CosimX. Cela s'explique par le fait que notre flot génère d'abord un modèle de

simulation en Colif et ensuite le code exécutable de ce modèle. La partie d'analyse des fichiers Colif prend la plupart du temps de génération. Ceci étant, dans notre flot cette analyse est faite deux fois (pour le générateur de modèle de simulation et pour le générateur de code exécutable), alors que CosimX ne le fait qu'une fois. Cependant, ce temps de génération additionnel représente une partie négligeable du processus de validation (en effet quelques minutes), comme nous l'avons vu dans les expérimentations. D'ailleurs, cela est le prix payé par le gain de flexibilité de notre flot.

Les sorties de la génération

La taille du code exécutable généré par notre flot est équivalente à celui de CosimX. Par contre, nous générons en plus un modèle de simulation en Colif. Cela facilite la compréhension du modèle de simulation généré, une fois que nous pouvons le visualiser.

5.6 Conclusion

Ce chapitre a présenté l'application du flot de génération automatique de modèles de simulation proposé par ce travail pour valider deux systèmes hétérogènes embarqués : un modem VDSL et un encodeur MPEG-4. Les modèles de simulation générés contenaient des adaptateurs de communication qui étaient implémentés en utilisant le modèle basé sur services. Les expérimentations présentées ici, avaient comme but d'évaluer le flot proposé et la modélisation basée sur les services des adaptateurs de communication.

Nous avons démontré par des expérimentations que le modèle d'adaptateur de communication basé sur les services peut faciliter : l'exploration d'architectures en étant flexible, la réutilisation de composants de base et l'implémentation d'adaptateurs plus performants.

Les expérimentations nous ont aidé à prouver que les buts globaux de ce flot sont atteints. Ce flot est capable de réduire le temps de validation des systèmes, fournir un modèle de simulation multi-niveaux, et générer des modèles de simulation tout au long d'un flot de validation en variant la vitesse et la précision de simulation. Nous avons vu en outre que ce flot peut apporter quelques avantages par rapport à CosimX, notamment une flexibilité plus grande dans le processus de génération des modèles de simulation, une réduction d'effort de la part du concepteur pour ajouter des composants de base à la Bibliothèque de Cosimulation et une meilleure compréhension du modèle généré par moyen d'un fichier Colif de celui-ci.

Chapitre 6 : Conclusion et perspectives

Conclusion

La pression de qualité et de mise sur le marché de systèmes embarqués monopuces fait que la validation de tels systèmes devient le point clé du processus de conception. La validation représente plus de la moitié du temps de conception. Cependant chaque jour la validation devient plus difficile car les systèmes sont de plus en plus hétérogènes. Cette hétérogénéité touche plusieurs aspects du système, comme les niveaux d'abstraction, les APIs et les protocoles de communication, les langages de spécification, entre autres. Les points clés pour réduire les temps de validation sont : maîtriser l'intégration des composants hétérogènes à travers de l'adaptation de la communication et générer automatiquement le modèle de simulation du système.

Le chapitre 2 a montré la diversité des aspects de communication relatifs aux systèmes hétérogènes embarqués. Nous avons vu que cette diversité rend difficile la validation de tels systèmes. Le chapitre 2 a analysé les différentes méthodes de validation existantes. Nous avons vu que la cosimulation, malgré ses limitations, reste encore l'option la plus avantageuse pour la plupart des étapes de la validation. L'étude sur l'état de l'art de la cosimulation et des adaptateurs de communication nous a fait constater que tandis que plusieurs approches donnent des solutions satisfaisantes pour la problématique de l'adaptation de différents langages de spécification (plus précisément de différents simulateurs), l'adaptation de la communication reste encore un défi.

Le chapitre 3 a proposé un modèle d'adaptateur de communication basé sur les services. Ce modèle essaie de résoudre les problèmes existants dans les autres approches, notamment le manque de flexibilité, la difficulté de réutiliser des composants de base et le surcoût relatif à la performance. Un adaptateur de communication est représenté par un graphe (appelé graphe de dépendance de services ou GDS) contenant des composants de base qui peuvent fournir et/ou requérir des services. Ces composants sont liés par la relation de dépendance entre les services requis et fournis par chaque composant. Ce modèle fait que les adaptateurs de communication n'implémentent que les services requis pour l'adaptation et facilite l'exploration architecturale du système grâce à sa flexibilité et sa facilité de réutilisation des

composants de base. Nous avons discuté aussi les problèmes du flot ROSES, notamment le manque d'un format unifié pour représenter les différents types d'interface, ce qui rend difficile la gestion des outils du flot et réduit la capacité d'exploration architecturale fournie par le flot. Ainsi, nous avons proposé l'utilisation du GDS comme le format unifié pour représenter des interfaces hétérogènes logicielle/matérielle/fonctionnelle.

La génération des adaptateurs de communication est montrée dans le chapitre 4. Ce chapitre présente également la génération automatique de tout le modèle de simulation d'un système hétérogène embarqué. Cette génération est essentielle pour accomplir le but principal de ce travail qui est l'accélération de la validation de systèmes hétérogènes embarqués. Partant d'une spécification en forme d'architecture abstraite du système et d'une bibliothèque de composants de base, nous arrivons à générer le modèle de simulation du système. Le flot de génération est divisé en plusieurs parties, ce qui le rend plus flexible à des changements de langage de spécification d'entrée ou de langage du code exécutable de sortie. Nous avons vu que cette génération peut être intégré au flot ROSES, et qu'il représente une évolution par rapport à l'outil de génération de modèles de simulation déjà existant (CosimX).

En effet, cette évolution a été montrée dans le chapitre 5. Le modèle d'adaptateur de communication et le flot de génération automatique de modèles de simulation ont été appliqués avec succès dans la conception et la validation de deux systèmes hétérogènes embarqués : le modem VDSL et l'encodeur MPEG-4. La flexibilité et la facilité de réutilisation des composants de base du modèle basé sur les services ont été prouvées. La plupart des adaptateurs générés automatiquement utilisaient des composants déjà existants dans la bibliothèque, et des changements de protocoles ne représentaient pas un grand impact sur le temps de validation et l'effort de la part du concepteur. Le probable gain de performance du modèle n'était pas vérifié, mais implicitement le modèle facilite cela parce qu'il n'y a dans l'adaptateur que les services requis pour la fonction globale d'adaptation. Les expérimentations nous ont également servies pour prouver que le flot de génération automatique de modèles de simulation accélère la validation, en générant un modèle à chaque étape de la validation en quelques minutes. Le nombre de lignes de code écrites par le concepteur pour générer les modèles de simulation a été plus petit en comparaison de celui qui doit être écrit en utilisant CosimX. Ce qui démontre une évolution du flot proposé par rapport à CosimX.

Perspectives

Le modèle d'adaptateur basé sur les services que nous avons proposé est une contribution importante pour adapter l'hétérogénéité des systèmes embarqués. Cependant, la définition plus formelle des concepts de services, d'éléments d'interface et de ports logiques est très importante pour l'adoption de ce modèle. La définition de la bonne taille d'un élément d'interface reste encore liée au savoir faire du concepteur. Un élément trop gros peut contenir des fonctionnalités qui ne sont pas nécessaires. Une fois que la bonne taille de l'élément d'interface est définie, un grand pas sera fait vers l'implémentation performante des adaptateurs de communication. Dans les travaux futures, cette étude doit être faite, ainsi qu'une analyse de performance de ces adaptateurs pour que nous puissions évaluer si ce modèle facilite effectivement la construction des adaptateurs plus performants comme nous le pensons.

En ce qui concerne le flot de génération automatique de modèles de simulation, il reste des ajustements à faire sur les outils du flot. Notamment, la partie de sélection des éléments du générateur de modèle de simulation. Le nombre de paramètres qui dirigent la sélection des éléments est encore réduit. Des paramètres qui concernent la performance de l'adaptateur doivent être ajoutés. Un exemple de tel paramètre est le temps estimé pour exécuter un service. A partir de la définition des nouveaux paramètres, nous pourrons modifier le mécanisme de sélection des éléments basé sur une métrique de performance.

Ce travail a démontré que l'extension du modèle de composition d'éléments d'interfaces basé sur le GDS est une façon efficace pour traiter les problèmes liés à l'hétérogénéité des systèmes embarqués. La généralisation du concept d'élément d'interface avec la notion de port logique (introduit dans ce travail) permet la représentation des interfaces hétérogènes logicielles/matérielles/fonctionnelles en utilisant un modèle unifié : le graphe de dépendance de services.

Bibliographie

[Abd03] S. Abdi, D. Shin, D.Gajski « Automatic Communication Refinement for System Level Design », Proc. Design Automation Conference (DAC'03), June 2003, Anaheim, USA

[Abr97] J. R. Abrial, « The B Book. Assigning Programs to Meaning », Cambridge University Press, 1997

[Arm99] ARM Limited, « ARM7TDMI Technical Reference Manual », available at http://www.arm.com/pdfs/DDI0029G_7TDMI_R3_trm.pdf

[Arm01] ARM Limited, « ARM PrimeXsys », available at http://www.arm.com/community/primexsys.html

[Arm02] ARM Limited, « AMBA Rev. 2.0 Specification », available at http://www.arm.com/arm/AMBA

[Arm04] ARM Limited, « MaxSim Designer », available at http://www.arm.com/products/DevTools/MaxSimDesigner.html

[Bar94] E. Barros, W. Rosenstiel, « A Clustering Approach to Suppport Hardware/Software Partitioning », Computer Aided Software/Hardware Enginnering, 1994

[Bel80] Bell Labs. « Spin – Formal Vérification » available at http://spinroot.com/spin/whatispin.html

[Ber00] R. A. Bergamaschi, W. R. Lee, « Designing Systems-on-Chip Using Cores » Proc. Design Automation Conference (DAC'00), June 2000, Los Angeles, USA

[Bru00] J-Y. Brunel, W. M. Kruijtzer, H. J. H. N. Kenter, F. Pétrot, L. Pasquier, E. A. de Kock, W. J. M. Smits, "COSY Communication IP's," Proc. Design Automation Conference (DAC '00), June 2000, Los Angeles, USA.

[Buc90] J. Buck, S. Ha, E. A. Lee, D. G. Masserschimitt, « Ptolemy : a framework for simulating and prototyping heterogeneous systems » International Journal of Computer Simulation, special issue on Simulation Software Development, January, 1990

[But97] D. R. Butenhof, « Programming with Posix Thread », Published by Addison Wesley Professional, May, 1997.

[Cad02] Cadence Design Systems, "FPGA Design with Cadence SPW" available at http://www.cadence.com/datasheets/3829_FPGAdesSPWDSfnl.pdf

[Cal95] J. P. Calvez, D. Heller, O. Pasquier, « System performance modeling and analysis with VHDL : benefits and limitations », In. Proc. of VHDL forum for CAD in Europe Conference, 1995.

[Ces01] W.O. Cesario, G. Nicolescu, L. Gauthier, D. Lyonnard, A. A. Jerraya, « Colif : a Multilevel Design Representation for Application-Specific Multiprocessor System-on-Chip Design », 12th IEEE International Workshop on Rapid System Prototyping, June 25-27, 2001, Las Vegas, USA.

[Ces02] W. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, « Component-Based Design Approach for Multicore SoCs », Proc. Design Automation Conference (DAC'02), June 2002, New Orleans, USA

[Cha02] A. Chakrabarti, P. Dasgupta, P. P Chakrabarti, A. Banerjee, « Formal Vérification of Module Interfaces against Real Time Specifications », Proc. Design Automation Conference (DAC'02), June 2002, New Orleans, USA

[Col04] J. Colgan, P. Hardee, « Advancing Transaction Level Modeling (TLM): Linking the OSCI and OCP-IP Worlds at Transaction Level » available at http://www.opensystems-publishing.com/whitepapers/colgan_and_hardee

[Cop04] M. Coppola, S. Curaba, M. D. Grammatikakis, R. Locatelli, G. Maruccia, F. Papariello, « OCCN: a NoC modeling framework for design exploration », Journal of Systems Architecture: the EUROMICRO Journal, Volume 50, Special issue: Networks on chip, Issue 2-3, February, 2004

[Cor97] J. M. Geib, C. Gransart, P. Merle, « Corba, des concepts à la pratique », Masson Editeur, Paris, 1997

[Cor00] « Deployment & Configuration of Component-based Distributed Applications Specification », OMG final adopted specification, pct/03-07-08, 2003

[Cow02] CoWare Inc., « CoWare N2C System Designer », available at http://www.coware.com

[Edw97] S. Edwards, L. Lavagno, E. A. Lee, A. Sangiovanni-Vincentelli, « Design of Embedded Systems : Formal Models, Validation and Synthesis », Proc. of the IEEE, Vol. 85, No. 3, March, 1997.

[Eke01] J. Eker, C. Fong, J. Janneck, J. Liu, « Design and Simulation of Heterogeneous Control Systems using Ptolemy II », Proc. IFAC Conference on New Technologies for Computer Control (NTCC'01), Hong Kong, China, November 2001

[Est04]Esterel Technologies, « Esterel Studio – Overview » available at http://www.esterel-technologies.com/products/esterel-studio/overview.html

[Gaj91] D. Gajski, « Essential issues and possible solutions in high-level synthesis », in High-Level VLSI Synthesis, Kluwer Academic Publishers, Boston, 1991

[Gau01] L. Gauthier, « Génération de Système d'Exploitation pour le Ciblage de Logiciel Multitâche sur des Architectures Multiprocesseurs Hétérogènes dans le Cadre des Systèmes Embarqués Spécifiques », Thèse de doctorat, INPG, Laboratoire TIMA, Décembre 2001.

[Gha03] F. Gharsali, « Conception des Interfaces Logiciel-Matériel pour l'Intégration des Mémoires Globales dans les Systèmes Monopuces », Thèse de doctorat, INPG, Laboratoire TIMA, Juillet 2003.

[Gnu04] « GNU m4 » available at http://www.gnu.org/software/m4

[Gra05] A. Grasset, F. Rousseau, A. A. Jerraya « Automatic Generation of Component Wrappers by Composition of Hardware Library Elements Starting from Communication Service Spécification », Proc. Rapid System Prototyping (RSP'05), June 2005, Montreal, Canada

[Hen01] Y. Héneault, G. Bois, E. M. Aboulhamid, « A Fast Hardware Co-Spécification and Co-Simulation Methodology Integrated in a H/S Co-Design Platform », Proc. International Conference on Microeletronics, October 2001, Rabat, Morocco

[Hes00] F. Hessel, « Conception des Systèmes Hétérogènes Multilangages », Thèse de doctorat, UJF, Laboratoire TIMA, Juin 2000

[Hof01] A. Hoffmann, T. Kogel, H. Meyr , « A Framework for Fast Hardware/Software Co-simulation », *Proc. Design Automation and Test in Europe* (DATE'01), March 2001, Munich, Germany

[Iso89] ISO, IS 8807, « LOTOS a formal description technique based on temporal ordering of observational behavior », February, 1989

[Ibm04] IBM Corp., « IBM CoreConnect Bus Architecture », available at http://www3.ibm.com/chips/products/coreconnect/index.html

[Jav04] « Enterprise JavaBeans Specification, Version 2.1 », available at http://java.sun.com/products/ejb

[Jer96] A. A. Jerraya, P. Kission, M. Rahmouni, « Behavioral Synthesis and Component Reuse with VHDL », Kluwer Academic Publisher, 1996

[Kan74] G. Kahn, “The semantics of a simple language for parallel programming,” in *Information Processing*, J. L. Rosenfeld, Ed. North-Holland Publishing Co., 1974.

[Ker99] C. Kern and M. Greenstreet, « Formal Verification in Hardware Design: A Survey », ACM Transactions on Design Automation of E. Systems, Vol. 4, April 1999, pp. 123-193

[Kri05] L. Kriaa, « Modélisation et Validation des systèmes hétérogènes embarqués : Modèle d'exécution », Thèse de doctorat, UJF, Laboratoire TIMA, Septembre 2005

[Lee97] E. A. Lee, A. Sangiovanni-Vincentelli, « A Denotational Framework for Comparing Models of Computation », ERL Memorandum UCB/ERL-M97/11 University of California, Berkeley, January 1997

[Lem00] Ph. Lemarrec, « Cosimulation multi-niveaux dans un flot de conception multi-langage », Thèse de doctorat, INPG, Laboratoire TIMA, Juin 2000

[Lyo03] D. Lyonnard, « Approche d’Assemblage Systématique d’Eléments d’Interface pour la Génération d’Architecture Multiprocesseur », Thèse de doctorat, INPG, Laboratoire TIMA, Avril 2003.

[Mat00] “MATLAB : The Language of Technical Computing” available at http://www.mathworks.com

[May84] D. May, R. Taylor, « Occam : An Overview », *Microprocessors and Microsystems*, Vol. 8, No. 2, March, 1984

[Men04] Mentor Graphics, Inc. “Seamless CVE 5.0” available at http://www.mentor.com/seamless

[Mes00] D.J.G. Mestdagh, M.R.Isaksson, P.Odling, « Zipper VDSL : A Solution for Robust Duplex Communication over Telephone Lines », IEEE Communication Magazine, pp. 90-96, May 2000.

[Mod01] ModelSim, available at http://www.model.com

[Mos96] E. Mosanya, M. Goeke, J. Linder, J. Y. Perrier, F. Rampogna, E. Sanchez, « Platform for Codesign and cosynthesis based on FPGA », 12th Rapid System Prototyping (RSP96), Thessaloniki, 1996

[Mpe04] « The Reference Website for MPEG » available at http://www.mpeg.org

[Mpe04a] « Open DIVX » available at http://www.projectmayo.com

[Mpi04] « The Message Passing Interface Standard » available at http://www-unix.mcs.anl.gov/mpi/

[Mpi04a] « MPICH - A Portable Implementation of MPI », available at http://www-unix.mcs.anl.gov/mpi/mpich/

[Sdl00] SDL Forum Society « MSC », available at http://www.sdl-forum.org/MSC

[Nic02] G. Nicolescu, « Spécification et Validation des Systèmes Hétérogènes Embarqués », Thèse de doctorat, INPG, Laboratoire TIMA, Novembre 2002.

[Ocp01] « OpenCore Protocol » available at http://www.ocpip.org

[Ora86] A. Orailoglu, D. D. Gajski, « Flow Graph représentation », Proc. Design Automation Conference (DAC'86), June 1986, Las Vegas, USA

[Pas02] S. Pasricha, « Transaction Level Modeling of SoC with SystemC 2.0", Synopsys User Group Conference (SNUG 2002), May 2002 , Bangalore, India,

[Pau02] P. G. Paulin, C. Pilkington, E. Bensoudane, "StepNP: A System-Level Exploration Platform for Network Processors", *IEEE Design & Test of Computers*, vol. 19, no.6, Nov. 2002.

[Pau04] P. G. Paulin, C. Pilkington, E. Bensoudane M. Langevin, G. Nicolescu, « Parallel Programming Models for a Multi-Processor SoC Platform Applied to High-Speed Traffic Management », Proc. of International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS 04), September 2004, Stockholm, Sweden

[Pet81] J. L. Peterson, « Petri Net Theory and the Modeling of Systems » Prentice Hall, 1981, Englewood Cliffs, New Jersey, USA

[Pto02] The Ptolemy project, available at http://ptolemy.berkeley.edu

[Ram01] A. Ramanathan, R. Teisser, D. McLaughin, « Acquisition of Sensing Data on a Reconfigurable Platform », International Geosciences and Remote Sensing Symposium, Sidney, 2001

[Ris02] « Hardware abstraction Layer », available at http://www.iyonix.com/32bit/HAL.shtml, December, 2002

[Ros98] W. Rosenstiel, « Rapid Prototyping, Emulation and Hardware-Software Co-debbuging », in System-Level Synthesis, pp. 219-262, Kluwer Academic Publisher, 1998

[San02] A. Sangiovanni-Vincentelli, « Defining Platform Based Design », EE Design, March 2002

[Sas04] A. Sasongko, « Prototypage basé sur une plateforme reconfigurable pour la vérification des systèmes monopuces », Thèse de doctorat, UJF, Laboratoire TIMA, Octobre, 2004

[Sch03] T. Schubert, « High Level Formal Verification of Next Generation Microprocessors », Proc. Design Automation Conference (DAC'03), June 2003, Anaheim, USA

[Son04] Sonics Inc. « Sonics Studio » available at http://www.sonicsinc.com/sonics/products/sonicsstudio

[Son04a] Sonics Inc. « SiliconBackplane MicroNetwork » available at http://www.sonicsinc.com/sonics/products/siliconbackplaneIII

[Spi89] J. M. Spivey, « An introduction to z formal specifications », Software Enginnering Journal, January, 1989

[Sta94]J. Staunstrup, « A Formal Approach to Hardware Design », Kluwer Academic Publisher, 1994

[Syn00] Synopsys, Inc. « VHDL System Simulator (VSS) », available at http://www.synopsys.com

[Syn04] Synopsys, Inc. "Co-Centric System Studio" available at http://www.synopsys.com/products/cocentric_studio/cocentric_studio.htm

[Sys00] « SystemC 2.0 Language Reference Manual » available at http://www.systemc.org

[Sys00a] « SystemC 2.0 : Functional Specification » available at http://www.systemc.org

[Sys04] « System Architecture 4th Edition » available at http://averia.mgt.unm.edu/sa4e_student/

[Tou00] A. M. Tourapis, O. C. A Ming, L. Liou, G. Shen, I. Ahmad "Optimizing the MPEG4 encoder – Advanced diamond zonal search", IEEE Int. Symp. Circuits and Systems, 2000

[Tsi03]A. Tsikhanovich, E. M. Aboulhamid, G. Bois, « Object-Oriented Techniques in Hardware modeling using SystemC », NEWCAS 2003, June 2003, Montreal, Canada

[Uml02] « UML –Unified Model Language » available at http://www.rational.com/uml

[Val94] C. Valderrama, F. Nacabal, P. Paulin, A. A. Jerraya, « Automatic VHDL-C Interface Generation of Distributed Cosimulation : Application to Large Design Examples », Design Automation for Embedded Systems vol. 3, no. 2/3, p. 199-217, April 1994

[Vdsl04] « CCM – Encyclopédie Informatique Libre – ADSL » disponible au site http://www.commentcamarche.net/technologies/adsl

[Vsi99] VSIA Alliance, On-Chip Bus Development Working Group, Virtual Component Interface Specification version 1.0, pp. 1-19, Septembre 1999.

[Xml00] « XML 1.0 Spécification » 2d ed., W3C Recommendation, October 2000, available at http://www.w3c.org/XML

[You04] M. W. Youssef, S. Yoo, A. Sasongko, Y. Paviot, A. A. Jerraya « Debugging HW/SW Interface for MPSoC : Video Encoder System Design Case Study », Proc. Design Automation Conference (DAC'04), June 2004, San Diego, USA

[Zit93] M. Zitterbart, « A Model for Flexible High performance Communication Subsystems », IEEE Journal on selected areas in communication, VOL. 11, NO, 4, MAY 1993.

Publications

1. A.M. FOUILLIART, E. VAUMORIN, R. NOUACER, A. SARMENTO, L. KRIAA, P. KAJFASZ, P. COUSSY, F. BLANC, "SystemC'Mantic: A high level Modeling and Co-design Framework For Reconfigurable Real Time Systems", Forum on specification and Design Languages FDL'05, Lausanne, Switzerland, Sept.27-30, 2005.

2. A. SARMENTO, L. KRIAA, A. GRASSET, W. YOUSSEF, A. BOUCHHIMA, F. ROUSSEAU, W. CESARIO, A.A. JERRAYA, "Service Dependency Graph, an Efficient Model for Hardware/Software Interfaces Modeling and Generation for SoC Design", International Conference on Hardware - Software Codesign and System Synthesis CODES-ISSS 2005, New York Metropolitan Area, USA, Sept. 18-21, 2005.

3. A. SARMENTO, W. CESARIO, A.A. JERRAYA, "Automatic Building of Exécutable Models from Abstract SoC Architectures", 15th IEEE International Workshop on Rapid System Prototyping (RSP 2004), Geneva , Switzerland , June 2004.

RESUME

La pression pour la qualité et de mise sur le marché de systèmes embarqués monopuces fait que la validation de tels systèmes devient le point clé du processus de conception. La validation répond pour plus de la moitié du temps de conception. Mais à chaque jour la validation devient plus difficile car les systèmes sont de plus en plus hétérogènes. Cette hétérogénéité touche plusieurs aspects du système, comme les niveaux d'abstraction, les APIs et protocoles de communication, les langages de spécification, entre autres. Les points clés pour réduire le temps de validation sont : (1) maîtriser l'intégration des composants hétérogènes à travers de l'adaptation de la communication, (2) et générer automatiquement le modèle de simulation du système.

Ainsi, les contributions apportées par ce travail pour accélérer le temps de validation sont: (1) la proposition d'un modèle d'adaptateur de communication basé sur les services pour la cosimulation des systèmes hétérogènes embarqués ; (2) la proposition et l'implémentation d'un flot de génération automatique de modèles de simulation pour les systèmes hétérogènes embarqués.

Les approches proposées ont été validées sur deux systèmes hétérogènes embarqués : un modem VDSL et un encodeur MPEG-4.

TITRE EN ANGLAIS

Automatic Génération of Simulation Models for the Validation of Heterogeneous Systems-on-Chip

ABSTRACT

As quality and time-to-market constraints of systems-on-chip increase, validation becomes the key point of the design process. Validation represents more than 50% of design time. The ever-increasing heterogeneity of systems-on-chip makes validation harder. This heterogeneity concerns different aspects of the system such as: abstraction levels, API's and communication protocols, specification languages, etc. The key points to reduce validation time are : (1) mastering heterogeneous components integration by adapting their communication (2) and generating simulation models automatically.

The main contributions of this work for accelerating validation time are : (1) the definition of a service-based model for communication adapters in order to enable the cosimulation of heterogeneous systems-on-chip ; (2) the definition and implementation of an automatic generation flow of simulation models for heterogeneous systems-on-chip.

The proposed approach was validated on two systems-on-chip : a VDSL modem and a MPEG-4 encoder.

SPECIALITE Microélectronique

MOTS CLES

Systèmes hétérogènes monopuces, validation, cosimulation, modèle de simulation, adaptateur de communication, modèle basé sur les services.

INTITULE ET ADRESSE DU LABORATOIRE

Laboratoire TIMA, Techniques de l'Informatique et de la Microélectronique pour l'Architecture des ordinateurs

46, avenue Felix Viallet, 38031 Grenoble

ISBN 2-84813-073-3 Broché ISBNE 2-84813-074-1 Format electronique