UNIVERSITÉ DE MONTRÉAL

GESTION GÉNÉRIQUE ET RAFFINEMENT DE SYSTÈMES ÉLECTRONIQUES À HAUT NIVEAU

NICOLAS LAUG
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2008

Your file Votre référence
ISBN: 978-0-494-48927-7
Our file Notre référence
ISBN: 978-0-494-48927-7

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

GESTION GÉNÉRIQUE ET RAFFINEMENT DE SYSTÈMES ÉLECTRONIQUES À HAUT NIVEAU

présenté par : LAUG Nicolas

en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de :

Mme. NICOLESCU Gabriela, Doct., présidente

M. BOIS Guy, Ph.D., membre et directeur de recherche

M. GAGNON Michel, Ph.D., membre

REMERCIEMENTS

Mes premiers remerciements s'adressent à mon directeur de recherche, M. Guy Bois pour son encadrement de mon travail de maîtrise, et qui a développé mon intérêt pour la conception de systèmes électroniques à haut niveau.

Je tiens également à remercier vivement Mme Gabriela Nicolescu et M. Michel Gagnon pour avoir accepté d'être présidente et membre du jury de cette maîtrise. Le temps qu'ils ont consacré à l'évaluation de ce travail est grandement apprécié.

Je salue aussi les membres du laboratoire CIRCUS, présents et passés, que j'ai rencontrés au cours de ma maîtrise. En particulier, merci à Marc-André pour son aide précieuse sur l'implémentation bas niveau, et à Benoit qui m'a grandement aidé à définir les contraintes des plateformes haut niveau. Merci aussi à Luc et Maxime pour leurs bons conseils. Sans oublier mes collègues de bureau, Fatoumata, Sébastien et Karim, ainsi que les autres membres du CIRCUS, Lam, Michel, Laurent, ...

Merci aussi au personnel de Polytechique pour son aide lors des démarches administratives que j'ai eu à faire.

Merci enfin à ceux qui accepté mon départ de France, et tous ceux qui m'ont fait aimer le Québec.

RÉSUMÉ

La complexité des systèmes sur puce augmente exponentiellement avec le temps. Ajoutée à une forte contrainte sur les temps de mise sur le marché *(time-to-market)*, cette constatation aboutit à un besoin important de productivité. L'émergence récente des techniques de conception niveau système *(ESL)* apporte de nombreux avantages, dont la finalité est de répondre à ce besoin.

Pour réduire le temps de développement, il est également important de favoriser la réutilisation. L'échange de composants ne peut se faire que par un format de données standard. Le format IP-XACT, qui supporte l'ESL depuis sa dernière version 1.4, répond à ce besoin.

Le travail présenté dans ce mémoire de maîtrise est le cœur d'un environnement de conception ESL fondé sur IP-XACT. Il repose sur une bibliothèque de composants et est conçu de manière totalement générique. Autrement dit, le programme est entièrement indépendant des composants qu'il manipule. Il est capable de manipuler des *designs* afin de générer des simulations. Mais il est surtout accompagné, et cela constitue la contribution essentielle du travail, d'un ensemble d'outils génériques. L'utilisation de ces outils nécessite des extensions au format IP-XACT, qui sont proposées dans ce mémoire.

L'utilisation de la plateforme suit la démarche de conception selon l'ESL. Les premières étapes sont consacrées à la spécification de l'application du système à concevoir. Puis, en utilisant des niveaux d'abstraction de plus en plus bas et précis, l'architecture finale est définie. La mise en œuvre réelle du système constitue la dernière étape.

Les outils permettent d'automatiser les tâches de création et de modification des modèles tout au long de ce flot de conception. Ils permettent en outre à l'utilisateur de faire abstraction des détails d'implémentation et d'être assisté lors des étapes de raffinement du système, jusqu'à son implantation sur carte à FPGA.

Le bon fonctionnement de la plateforme et des outils a été validé par différents essais unitaires. Dans ce mémoire, un cas d'utilisation est présenté. Il utilise la plateforme tout le

long du flot de conception d'un système, ce qui valide la fonctionnalité globale.

ABSTRACT

Complexity of system on chips increases exponentially with time. Added to the pressure of time-to-market constraint, that situation leads to a great need of productivity.

Recent rise of electronic system level (ESL) techniques brings various advantages, aiming to answer that need.

To reduce development time, it is also important to foster reuse. Component exchange is possible only with a standard data format. IP-XACT standard, which supports ESL since its last 1.4 version meets that requirement.

The work presented in this report is the core of an ESL design environment based on IP-XACT. It uses a component library and it is design in a totally generic way. In other words, the software program is totally independent from the component it deals with. It is able to handle designs to generate simulations. Moreover, it embeds a set of generic tools, which are the main contribution of this work. Those tools need to add extensions to IP-XACT, which are proposed in this report.

The platform use follows the ESL design flow. The first steps are dedicated to specification of the application of the system to make. Then, using lower and more precise abstraction levels, the final architecture is defined. The last step is the real implementation of the system.

Tools automate model creation and modification tasks, in the ESL design flow. They also let the user make abstraction of implementation details, and be assisted in the refinement steps, down to an FPGA implementation.

Various unit tests have been run to validate the platform and its tools. In this report, a use case is presented. It uses the platform all along the design flow of a system, which validates its global functionality.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES ACRONYMES

BRAM :	Block Random Access Memory
DCM :	Digital Clock Manager
EDA :	Electronic Design Automation
ESL :	Electronic System Level
FCB :	Fabric Coprocessor Bus
FIFO :	First In First Out
FPGA :	Field Programmable Gate Array
FSL :	Fast Simplex Link
HDL :	Hardware Description Language
HTML :	HyperText Markup Language
HTTP :	HyperText Transfer Protocol
IP :	Intellectual Property
ISBN :	International Standard Book Number
ISS :	Instruction Set Simulator
JAR :	Java Archive
JTAG :	Joint Test Action Group
LMB :	Local Memory Bus
LUT :	Look-Up Table
MHS :	Microprocessor Hardware Specification
MoML :	Modeling Markup Language
MPD :	Microprocessor Peripheral Definition
MSS :	Microprocessor Software Specification
MVC :	Modèle, Vue, Contrôleur
OCP :	Open Core Protocol
OPB :	On-chip Peripheral Bus
PAO :	Peripheral Analyze Order

PIC :	Programmable Interrupt Controller
RTL :	Register Transfer Level
SDRAM :	Synchronous Dynamic Random Access Memory
SPIRIT :	Standard Structure for Packaging, Integrating and Re-using IP within Tool-flows
TGI :	Tight Generator Interface
TLM :	Transaction Level Modeling
UART :	Universal Asynchronous Receiver-Transmitter
UCF :	User Constraints File
UML :	Unified Modeling Language
URI :	Uniform Resource Identifier
URL :	Uniform Resource Locator
UTF :	Untimed Functional
VHDL :	VHSIC Hardware Description Language
VHSIC :	Very High Speed Integrated Circuit
VLNV :	Vendor, Library, Name, Version
W3C :	World Wide Web Consortium
XBD :	Xilinx Board Definition
XML :	Extensible Markup Language
XMP :	Xilinx Microprocessor Project
XPS :	Xilinx Platform Studio
XSD :	XML Schema Definition

LISTE DES TABLEAUX

LISTE DES ALGORITHMES

LISTE DES ANNEXES

INTRODUCTION

En 1965, Gordon Moore a énoncé le célèbre principe connu désormais sous le nom de *loi de Moore* [1]. Cette prévision, vérifiée jusqu'à présent, indique que la complexité des circuits électroniques augmente exponentiellement en fonction du temps. La conséquence de cette tendance est l'augmentation constante de l'utilisation de l'électronique, celle-ci ayant des capacités toujours plus importantes à des coûts qui diminuent. À cause de l'évolution rapide des systèmes électroniques, la priorité de l'industrie est de mettre le plus tôt possible les nouveaux produits sur le marché. Cette contrainte se nomme le *time-to-market*, c'est à dire la durée séparant le lancement d'un projet à sa mise sur la marché.

Ainsi, les capacités des circuits augmentent, le temps de développement doit être le plus court possible, et sans augmenter le coût de la conception. La productivité de la conception de systèmes électroniques est donc le premier objectif pour l'industrie. L'amélioration des techniques de conception a permis de l'augmenter considérablement. Cependant, cette augmentation de productivité (en nombre de transistors par rapport à la la quantité de travail) ne suffit pas à compenser la complexification des circuits sur puce. Autrement dit, produire un nouveau système sur puce coûte de plus en plus cher en développement. Ce phénomène est connu sous le terme de fossé de productivité *(productivity gap)*, figure 1[1].

Il existe principalement trois directions pour augmenter la productivité.

La réutilisation est la plus évidente. En effet, tout système sur puce contient des éléments préexistants. Il peut s'agir d'entités de taille relativement petite, comme des additionneurs ou beaucoup plus gros, comme des calculateurs de transformée de Fourier, voire des processeurs. Pour favoriser la réutilisation, les plateformes de conception de systèmes sur puce intègrent généralement une bibliothèque de composants. Il est également possible pour

[1]Principe énoncé lors de l'édition 1999 de l'*International Technology Roadmap for Semiconductors*. Il a été cité de nombreuses fois par la suite, en particulier dans [2].

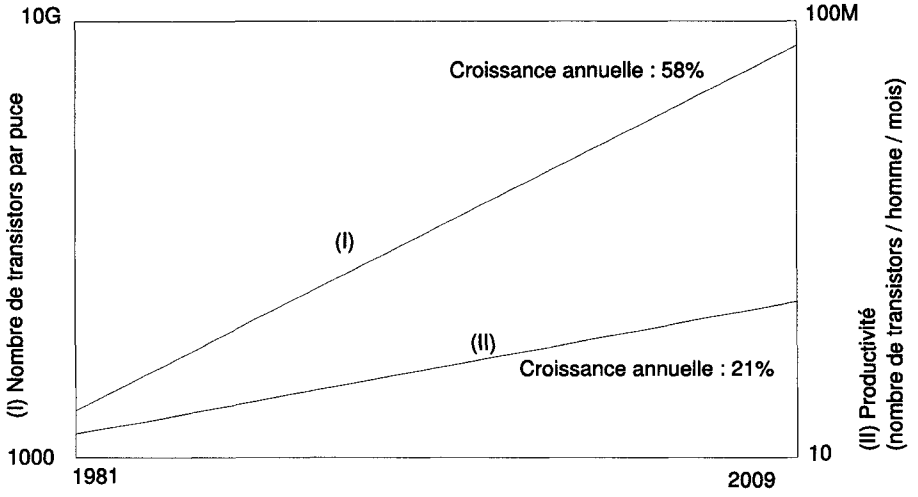


Figure 1: Le fossé de productivité

les concepteurs de systèmes d'acheter spécialement des composants à des fournisseurs. La réutilisation permet évidemment de gagner du temps de développement en évitant de réinventer la roue à chaque fois. Les composants réutilisables apportent aussi fiabilité et performance. D'abord, les développeurs peuvent être spécialisés dans le domaine concerné. De plus, comme le module pourra être vendu un grand nombre de fois, il est possible économiquement de consacrer un travail conséquent à son optimisation : diminution des temps de latence et des ressource utilisées, tests intensifs... Pour des raisons de rentabilité, un tel travail n'aurait peut-être pas été possible pour une utilisation unique.

Une autre voie pour augmenter la productivité est l'utilisation de niveaux d'abstraction plus élevés. La première étape importante fut le passage des descriptions physiques (portes logiques) au niveau des transferts de registres (RTL, *Register Transfer Level*). La simulation d'un système RTL est plus rapide que celle d'une description des portes. Il est donc possible de valider la description RTL avant de passer aux niveaux d'abstraction plus précis. De plus la synthèse de ces descriptions peut être automatisée, ce qui permet de supprimer les erreurs humaines tout en faisant gagner du temps. Enfin, une spécification RTL peut se synthétiser vers plusieurs types de matériel, ce qui favorise donc aussi la réutilisation. S'il est indiqué pour le développement de blocs matériels donnés, le RTL possède néanmoins ses limites.

En effet, il exige que l'architecture du système soit préalablement définie, au moins partiellement. C'est pourquoi des niveaux d'abstraction plus élevés, dits transactionnels, sont apparus. Kai et Gajski [3], et Ghenassia et Clouard [4] détaillent le flot de conception à mettre en œuvre. Ce flot se fait selon une approche *top-down*, c'est à dire partant des hauts niveaux d'abstraction pour aller vers les bas niveaux.

Le développement commence à partir des spécifications du produit. La première étape consiste à diviser l'application en modules fonctionnels communiquant entre eux. Ce haut niveau est nommé selon les cas TL3 *(transaction level 3)* ou UTF *(untimed functional)*. Le temps de calcul n'est pas considéré. Seule la fonctionnalité globale est prise en compte. Puis, par raffinements successifs, la précision des calculs et des communication augmente, jusqu'à arriver à l'implémentation finale. Au fur et à mesure des raffinements, des choix architecturaux sont réalisés. Les plus importants sont le partitionnement logiciel/matériel et l'architecture des communications (par exemple : simple bus, bus multiples, réseau sur puce).

La troisième solution est d'automatiser les tâches qui peuvent l'être tout au long du flot de conception. La synthèse a déjà été évoquée, mais d'autres fonctions peuvent être aussi créées, par exemple pour simplifier l'implémentation des communications inter-modules.

Problématique Ces trois voies (réutilisation, hauts niveaux d'abstraction et automatisation) peuvent être combinées dans une plateforme de développement. Comme nous le verrons dans le chapitre 1, certaines solutions existantes répondent partiellement à ce problème en se focalisant sur une partie du problème. Cependant, il n'existe pas à ce jour de solution connue combinant les trois voies.

La réutilisation, pour qu'elle soit efficace, exige que de nouveaux blocs puissent être ajoutés facilement. Il est donc nécessaire que la plateforme de conception soit générique, c'est à dire qu'elle puisse traiter des nouveaux modules sans modifier le programme lui-même.

Le support des hauts niveaux passe par celui d'un langage approprié. Enfin, des outils automatisant certaines tâches s'ajoutent pour libérer l'utilisateur d'opérations fastidieuses.

Objectif L'objectif du travail de maîtrise présenté dans ce mémoire est de mettre en œuvre le cœur d'une plateforme de conception tirant parti de ces trois axes : réutilisation, hauts niveaux et automatisation. Tout d'abord, elle est conçue de manière à être entièrement générique. Elle est ainsi facilement extensible, puisque le programme en lui-même est totalement indépendant des composants utilisés. Les caractéristiques propres aux composants sont documentées dans des fichiers de données distincts du programme. Pour favoriser la réutilisation, un format de fichier standard est utilisé pour décrire les composants : IP-XACT. De plus, il s'agit d'une plateforme à haut niveau ; bien que facilement extensible à d'autres langages, elle est axée sur SystemC. Enfin, pour que cette plateforme ait un intérêt à l'utilisation, elle intègre différents outils qui facilitent la création des modèles de systèmes.

Contributions Les contributions de ce travail peuvent être vues selon deux aspects. Bien que distincts, ils sont liés et reposent tous deux sur une base logicielle présentée dans le chapitre 3. Le premier aspect est l'ensemble d'outils génériques présenté aux chapitres 4 et 5, en particulier :

- la sélection et l'insertion automatiques de chaînes d'adaptateurs entre un composant fonctionnel et un canal (sec. 4.4),
- la sélection et l'insertion automatiques de lien direct entre deux composants fonctionnels matériels (sec. 4.8.1),
- le raffinement fondé sur une bibliothèque de composants à hauts et bas niveaux (sec. 4.5, chap. 5).

Le second ensemble de contributions est constitué des propositions d'extensions au format IP-XACT. En plus de celles dépendant des outils énumérés ci-dessus, on trouve :

- l'utilisation de nouveaux types de générateurs (sec. 2.4.2),
- la description du logiciel reposant sur des instances de composants (sec. 4.7),
- un type de connexion supplémentaire correspondant aux liens directs abstraits (sec. 4.8.2),
- l'application de la notion d'héritage à des composants matériels afin de définir des équivalences (sec. 4.5.2),
- l'ajout d'informations propres à la disponibilité des composants pour une architecture cible bas niveau (sec. 5.2).

Structure Ce document se divise en sept parties. Le chapitre 1 effectue une revue plus en profondeur des techniques de conception de systèmes sur puce visant à augmenter la productivité. Il introduit ainsi les principaux concepts qui seront utilisés dans la plateforme. Le chapitre 2 a pour objet le format de données IP-XACT, utilisé pour décrire des composants, leurs interfaces et leurs mises en œuvre dans des *designs*. Le chapitre 3 expose la structure générale du logiciel, qui repose grandement sur les paradigmes de IP-XACT. Les outils simplifiant l'utilisation du logiciel se greffent sur cette structure. Les principaux sont détaillés dans le chapitre 4. Le chapitre 5 se concentre sur la génération d'une implémentation basée sur une carte à FPGA. Cela implique des contraintes et descriptions d'objets supplémentaires. Enfin, une utilisation de la plateforme est présentée dans le chapitre 6, et la conclusion se trouve dans le chapitre 7.

Les principes généraux de la plateforme ont fait l'objet d'un article [5], qui a été accepté et qui sera présenté lors de la conférence *IP08*. Cet article se trouve dans l'annexe V.

CHAPITRE 1

REVUE DE LITTÉRATURE

1.1 Hauts niveaux d'abstraction

Comme il l'a été mentionné dans l'introduction, l'utilisation de hauts niveaux d'abstraction est un des moyens pour concevoir rapidement et efficacement des systèmes sur puce. Le premier stade fut l'utilisation du niveau de transfert des registres (RTL), plus élevé que le niveau portes. Plus récemment, l'introduction des niveaux transactionnels (TLM), encore plus hauts, permet de modifier avantageusement le flot de conception.

1.1.1 Première étape : du niveau portes vers le RTL

Face à la complexité croissante des systèmes sur puce, l'utilisation de hauts niveaux d'abstraction se révèle indispensable. La première transition importante a lieu dans les années 1980, avec l'apparition des langages modernes de description du matériel, en particulier Verilog et VHDL. Il est ainsi désormais possible de décrire des systèmes sur puce au niveau RTL *(Register Tranfer Level)*. Le RTL spécifie l'évolution de l'état des registres, sans nécessairement détailler toute l'architecture matérielle implémentant ces fonctions logiques. Le niveau d'abstraction est donc plus élevé qu'une description structurelle des portes logiques, ce qui présente plusieurs avantages. D'abord, pouvoir abstraire les détails de mise en œuvre permet de créer des descriptions plus simplement et rapidement. La simulation est également plus rapide qu'au niveau des portes. On peut donc valider la fonctionnalité d'un module à ce niveau avant de la transformer vers des descriptions plus précises. De plus, des outils de synthèse logique permettent d'automatiser cette opération de raffinement. Le cycle de développement est donc plus rapide, et les

erreurs humaines sont réduites. Enfin, le niveau RTL est suffisamment abstrait pour ne pas être entièrement lié au matériel sur lequel le système sera implanté. La réutilisation est ainsi favorisée.

1.1.2 Du RTL au TLM

Le RTL présente cependant certaines limitations. Tout d'abord, les langages HDL, comme leur nom l'indique, sont dédiés à la description du matériel, et non du logiciel. De plus, le développement de composants RTL exige un travail important en amont, afin de définir précisément les spécifications. C'est pourquoi des niveaux d'abstraction plus élevés sont apparus. Il s'agit des niveaux transactionnels *(TLM, Transaction Level Modeling)*, qui font partie du domaine plus large de l'ESL (modélisation niveau système). Bien que le TLM regroupe en fait tout un ensemble de niveaux, il est possible d'établir des principes communs.

Pour commencer, les modèles transactionnels utilisent un découpage du système en blocs qui communiquent entre eux. Selon les sources, ces blocs sont nommés composants *(components)*, éléments de traitement *(processing elements)* ou modules. Cette division montre que les modèles transactionnels se trouvent à un niveau plus précis que la spécification globale du système.

Le second point réside dans le mode de communication choisi : les *transactions*. Chaque transaction implique deux modules : un *initiateur* et une *cible*. L'initiateur est à l'origine de la requête. La cible la reçoit, et éventuellement y répond. Contrairement au niveau RTL, où les communications reproduisent la réalité physique (un signal porté par un ou plusieurs fils), les communications TLM se font par des appels de fonction. L'initiateur appelle donc une fonction, et cet appel est transmis à la cible. Des canaux intermédiaires peuvent assurer le lien entre les deux modules. La figure 1.1 montre un exemple grandement simplifié d'une communication entre deux modules. Dans ce cas, la communication se fait directement,

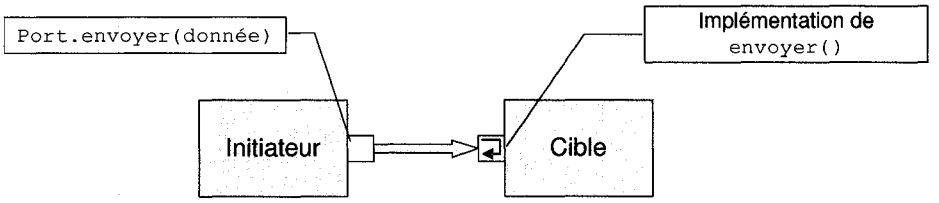
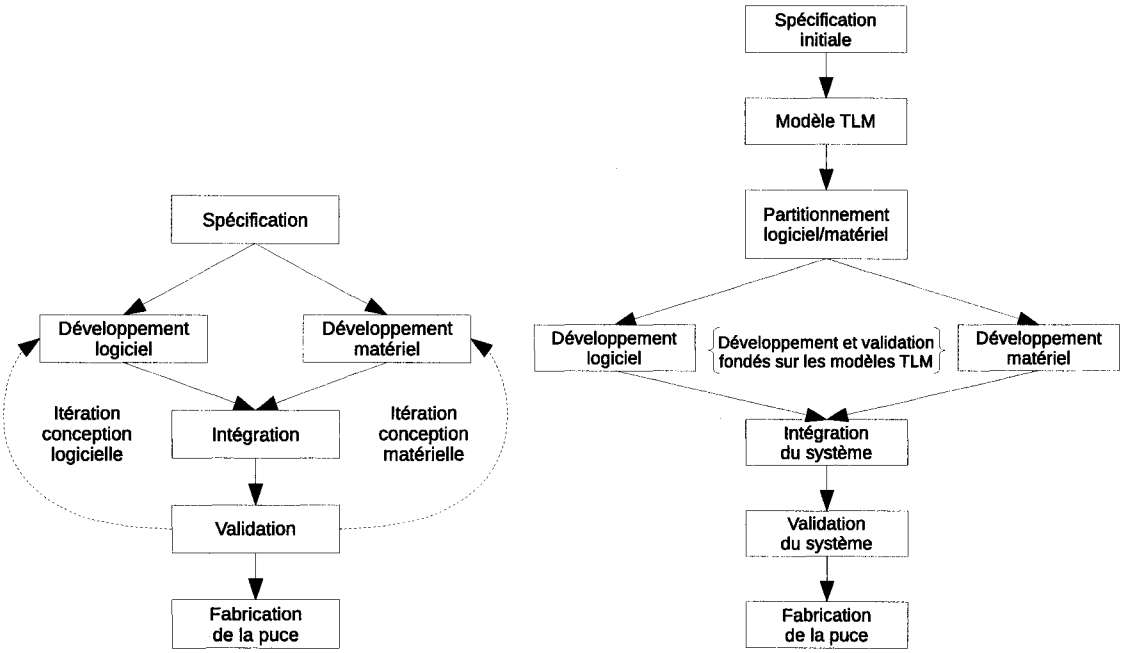


Figure 1.1: Simple communication transactionnelle entre deux modules



(a) Sans TLM (b) Avec TLM

Figure 1.2: Modification du flot de conception en utilisant le TLM

sans canal intermédiaire. On trouve ce type de communication dans le plus haut des niveaux transactionnels.

1.1.3 Les avantages du TLM

Disposer d'un modèle à haut niveau exécutable dès les premières étapes de développement du système est un atout certain. Il est ainsi possible de modifier le lot de conception en en tenant compte [4]. La figure 1.2 montre les conséquences positives du TLM sur

la conception d'un système. Dans les deux cas, la première étape consiste à spécifier la fonctionnalité du système, ou *application*. L'objectif est de parvenir à une *implémentation* qui réalise cette application. L'implémentation finale est souvent divisée en une partie logicielle, exécutée par un processeur, et des blocs matériels qui mettent en œuvre une partie de l'application.

La figure 1.2a représente le cycle de développement classique. La séparation entre le logiciel et le matériel se fait très tôt. Les deux équipes de développement ne disposent donc que des documents de spécification comme base de travail. En outre, il n'est possible de valider le système qu'après intégration. Chaque itération est donc coûteuse en temps et implique à la fois les développeurs du logiciel et du matériel.

En revanche, le TLM apporte une solution à ces problèmes (figure 1.2b). Les modèles transactionnels réalisés avant la séparation permettent d'explorer les possibilités de partitionnement. De plus, le modèle transactionnel des éléments de traitement constitue une référence qui peut être utilisée par les deux parties pour tester et valider leurs composants. Cette validation peut se faire d'au moins deux manières : 1) en comparant les sorties des deux implémentation (transactionnelle et bas niveau) pour des entrées identiques, et 2) en réalisant des simulations transactionnelles du système entier dans lesquelles le composant à tester est intégré (en utilisant des adaptateurs pour permettre les communications).

Pour récapituler, les avantages du transactionnel sont les suivants :
- modélisation du matériel et du logiciel
- exploration architecturale
- rapidité d'exécution des simulations
- création de modèles de référence pour valider les implémentations à bas niveau

De plus, les outils de synthèse comportementale tels que Forte Cynthesizer sont capables de raffiner des modèles transactionnels vers le RTL.

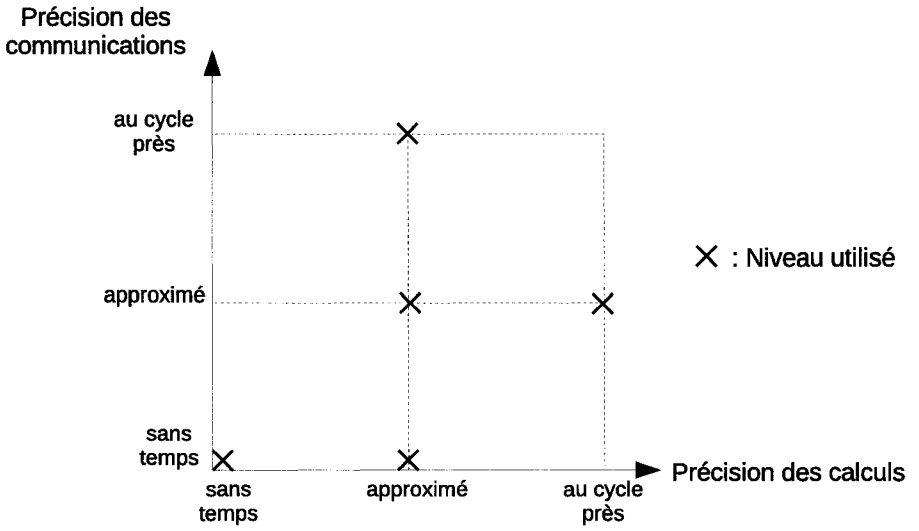


Figure 1.3: Découpage des niveaux transactionnels selon deux axes

1.1.4 Les sous-niveaux transactionnels

Les modèles transactionnels sont conçus selon le principe d'orthogonalité des modèles du calcul et des communications. Il est ainsi possible de modifier la mise en œuvre des communications sans modifier celle des calculs, et réciproquement. À partir de ce principe, Kai et Gajski [3] classent les sous-niveaux du TLM en deux dimensions, tel que représenté dans la figure 1.3. Tous les niveaux possibles ne sont pas présents sur ce graphique. En effet, les auteurs considèrent par exemple qu'il n'est pas pertinent de réaliser un modèle dans lequel les calculs sont modélisés sans notion de temps avec un modèle de communications précis au cycle d'horloge près.

L'orthogonalité communication/calculs implique que les éléments de traitement et les canaux ne partagent pas les mêmes interfaces de communication. Les interfaces des canaux sont en effet nécessairement liées à la granularité des communications dans le modèle choisi. Si les modules étaient directement connectés aux canaux, il devraient donc adapter leurs interfaces au modèle de communication, ce qui s'oppose au principe d'orthogonalité.

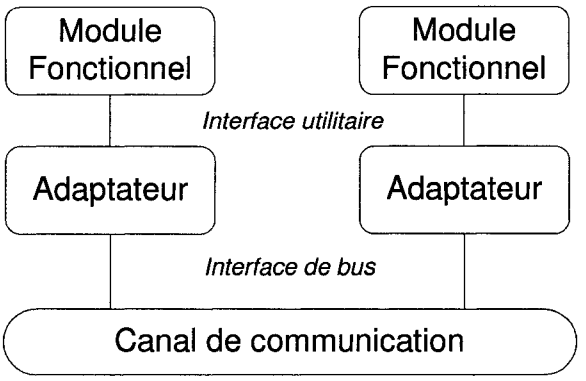


Figure 1.4: Modèle en couches

L'approche retenue est donc la suivante : les éléments de traitement disposent d'une interface simple, tandis que celle des canaux est adaptée à leur modèle. On arrive ainsi à un modèle en couches (figure 1.4), tel que présenté dans [6] et [7]. La première couche est celle des composants fonctionnels, avec une interface simple dont les fonctions sont appelées par le traitement à l'intérieur des modules. Ces composants sont connectés à des adaptateurs. Ceux-ci effectuent le lien entre les communications sur l'interface des modules fonctionnels et celle des canaux, eux aussi connectés aux adaptateurs.

Il existe un très grand nombre de niveaux utilisés, chacun se focalisant sur tel ou tel aspect de la conception du système. On peut néanmoins en dégager trois groupes principaux.

TL3 Aussi appelé PV *(Programmer's View)* ou UTF *(Untimed Functional)*, il n'a pas pour but d'estimer les temps d'exécution. Son rôle est d'établir une spécification fonctionnelle des éléments de calcul, et d'observer leur ordonnancement.

TL2 Ce niveau porte aussi les noms de PVT *(Programmer's View with Time)*, TF *(Timed Functional)* ou AV *(Architect's View)*. Il apporte une dimension temporelle aux communications entre les modules du système. Il offre un bon compromis entre la précision des simulations d'une part, et la rapidité de simulation et de développement d'autre part.

On peut alors explorer différentes architectures, que ce soit pour les communications (architectures multibus, réseaux sur puce, liens directs), ou pour le partitionnement logiciel/matériel, en introduisant des modèles de processeurs.

TL1 Dans ce niveau, les communications et les calculs sont estimés au cycle près. Chaque transaction de ce cycle correspond à une transaction du bus. Ce niveau est très proche du RTL ; la principale différence se situe dans les interfaces de communications : le TL1 est transactionnel, tandis qu'en RTL les interfaces sont un ensemble de ports de signaux. Son avantage par rapport au RTL est dans la rapidité de simulation, pour une précision quasi identique. Il peut aussi être utilisé pour tester un module RTL : on le simule pour cela dans une configuration où les autres modules sont transactionnels. Les sorties du module RTL peuvent également être comparées à son modèle transactionnel (considéré comme modèle de référence). On peut ainsi valider le module, pour une simulation relativement rapide. Cela peut aussi se faire pour les plus hauts niveaux, mais la proximité du TL1 avec le RTL en fait le niveau privilégié pour ce type d'utilisation.

1.2 Réutilisation

Dans la conception de systèmes sur puce, la réutilisation est un domaine complexe. Il faut en effet que les composants puissent communiquer avec le reste du système. Ils doivent en outre être paramétrables pour s'adapter aux différentes situations de mise en œuvre possibles. De nombreuses autres contraintes viennent s'ajouter, telles que les temps d'exécution, le matériel disponible et la protection de la propriété intellectuelle.

Pour pouvoir établir les connexions, il est nécessaire d'adopter des interfaces standard. Il s'agit en général d'interfaces de bus sur puce, comme OPB ou AMBA. Avec cette technique, l'utilisateur commence par choisir un type de bus. Dans le cas où le système possède au moins un processeur, celui-ci contraint fortement le choix du bus. Une fois le

bus choisi, il est possible de sélectionner les composants pouvant s'y connecter.

Cette technique présente un inconvénient majeur : le fournisseur du composant doit choisir son type d'interface lors de la conception. Si un utilisateur a besoin d'un autre type, il faut modifier le composant.

Le standard OCP [8] résout ce problème. Il définit une interface standard, indépendante du bus utilisé. Cette interface est très complète, mais aussi fortement configurable, de manière à éviter d'implémenter des fonctions inutiles. Des adaptateurs pour différents bus courants sont disponibles. Ainsi, tout composant conforme à l'interface OCP peut se connecter à différents bus et réseaux sur puce. L'inconvénient de cette solution se situe dans les ressources supplémentaires utilisées par les adaptateurs.

Au niveau transactionnel, la réutilisation de composants pose également le problème des interfaces. L'organisation qui développe le protocole OCP, l'OCP-IP propose des modèles transactionnel en SystemC des canaux de communication OCP. Ces modèles sont disponibles à trois niveaux d'abstraction différents.

Le standard TLM 2.0 proposé par l'OSCI [9] a pour but d'unifier les interfaces de communication transactionnelles. Il définit à la fois des interfaces proprement dites et la « charge utile » *(payload)* qui y transite. Les interfaces correspondent aux fonctions de communication appelées ou implémentées par les modules. Elles peuvent être bloquantes ou non bloquantes, avec possibilité d'indiquer les temps de communication. La charge utile correspond aux objets qui sont communiqués lors des transactions. La première version de la norme ne définissait pas de charge utile générique. Les utilisateurs devaient donc créer leur propres structures de données, ce qui nuisait fortement à l'interopérabilité et la réutilisation.

Le projet GreenSocs est aussi orienté vers l'interopérabilité. Il propose pour cela différents outils et interfaces à code source ouvert. Parmi ses projets, GreenBus était un environ-

nement définissant des interfaces de communication, à différents niveaux d'abstraction. GreenSocs ayant participé à l'élaboration de la norme OSCI TLM 2.0, ces concepts y ont été intégrés, et le développement de GreenBus est donc arrêté.

En permettant la portabilité des composants entre différentes architectures de communication, ces solutions favorisent donc la réutilisation. Cependant, un outil de gestion s'appuyant uniquement sur une de ces interfaces est fortement limité. En effet il présuppose que tous les composants soient conçus conformément à l'interface. Cela constitue une contrainte importante, et la gestion d'une grande bibliothèque de composants implique nécessairement de devoir manipuler plusieurs types d'interfaces, y compris des interfaces *a priori* inconnues. Pour cela, la plateforme doit être totalement générique vis-à-vis des interfaces et des composants. Il s'agit là du principal atout du travail présenté dans ce mémoire.

La norme IP-XACT est utile pour répondre à ce problème. Elle est dédiée aux métadonnées. Autrement dit, il ne s'agit pas de descriptions exhaustives des composants, mais d'informations qui permettent de les utiliser et des intégrer dans un système. En particulier, les fichiers IP-XACT décrivent les interfaces afin de les connecter. Ce format sera étudié plus longuement dans le chapitre 2.

1.3 Automatisation et assistance de la conception

Pour diminuer le temps de conception, les développeurs peuvent également profiter d'un ensemble d'outils. Ces outils permettent d'automatiser des étapes que la complexité actuelle des systèmes a rendu impossible à réaliser manuellement pour un coût raisonnable. Nous avons abordé les outils de synthèse dans la section 1.1. Il en existe de nombreux autres, que ce soit pour la vérification, le placement, le routage, ou la compilation du logiciel.

Les environnement de développement intégrés facilitent aussi la conception des systèmes sur puce. Tout d'abord, ils fournissent en général un moyen graphique et aisé de visualiser et de modifier les architectures. Ensuite, ils sont capables d'accéder à un ensemble d'outils, tels ceux précédemment abordés. Parmi ces environnements, citons notamment Xilinx Platform Studio (XPS) [10] et Quartus de Altera [11]. Par exemple, XPS donne entre autres accès à :

- la génération de simulations (pour les environnements ModelSim et NCSim)
- une bibliothèque de composants et de cartes
- une vue graphique et modifiable de l'architecture
- un outil de synthèse
- des outils de génération de la mise en œuvre sur FPGA
- un compilateur et un éditeur de liens (gcc et ld)
- un débogueur

Cependant, les outils fournis par ces environnements sont basés sur les bas niveaux, à la différence de la plateforme proposée ici qui se concentre sur les hauts niveaux (sans y être exclusive).

Les outils présentés dans [12, 13] ont les mêmes objectifs que la plateforme : ils permettent à l'utilisateur de créer une implémentation à partir d'une spécification à haut niveau. Ils sont fondés sur la génération de d'adaptateurs et de canaux permettant aux éléments du système de communiquer à partir de primitives simples. Cependant, ils restent dépendants de l'interface de communication *get/set* sur les composants et leur généricité est donc moindre. L'utilisation d'une interface de communication simple du côté des composants est à rapprocher des travaux de SpaceCodesign, présentés dans la section suivante.

1.4 SpaceCodesign

SpaceCodesign [14] est une plateforme de conception à haut niveau. Elle repose sur le langage de spécification SystemC [15].

SystemC Plus qu'un langage à proprement parler, SystemC est en fait un ensemble de classes, fonctions et macros de C++. Il présente plusieurs avantages, qui en ont fait le langage principal pour la modélisation à haut niveau. En particulier, le fait qu'il repose sur C++ permet de disposer des nombreux compilateurs déjà existants. Le code source du moteur de simulation est disponible, ce qui permet d'en créer des variantes répondant à des besoins spécifiques, telle que la co-simulation continu/discret [16]. De plus, SystemC permet de modéliser des composants selon une vue fonctionnelle ou une implémentation logicielle ou matérielle. Enfin, il supporte les ports transactionnels, ce qui est fondamental pour le TLM.

SpaceLib La plateforme SpaceCodesign peut se diviser en deux parties : SpaceLib et SpaceStudio. SpaceLib est un ensemble de composants SystemC qui sont utilisables selon la méthodologie suivante. Tout d'abord l'utilisateur crée la vue applicative de son système. Autrement dit, il conçoit les modules qui constituent la fonctionnalité. Il s'agit des éléments de traitement, que nous avons présentés lors de l'introduction du TLM (section 1.1.2). Ces modules dérivent de SpaceBaseModule, ce qui leur donne accès à des primitives de communication très simples pour envoyer des données ou en demander à d'autres modules.

À partir de ces éléments, il est possible de créer des configurations, en les connectant avec des composants de SpaceLib. Parmi ces composants, on trouve des canaux de communications (bus, liens point à point), des composants d'architecture (mémoires, processeurs), et des adaptateurs. On distingue deux niveaux d'abstraction. Le premier, Elix, permet de valider la fonctionnalité. Il repose sur les bus UTF (sans modélisation du temps)

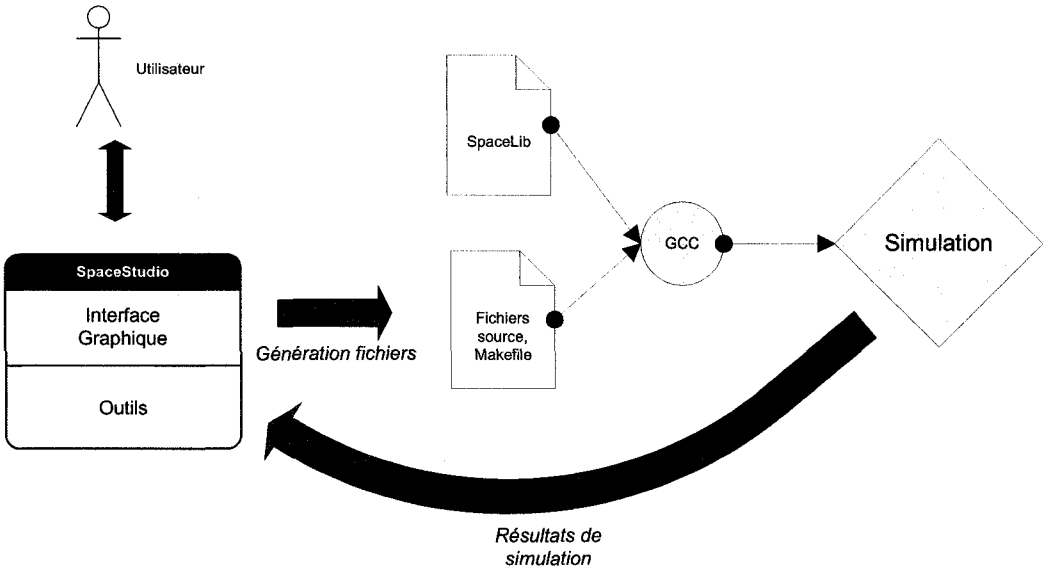


Figure 1.5: Vue globale de SpaceStudio

et TF (modélisation approchée du temps). Ces bus modélisent un bus abstrait, qui n'a pas pour but de représenter une implémentation de bus particulière. Le second niveau, Simtek, est plus bas, il permet d'explorer l'espace des architectures possibles. Les modèles des bus sont plus précis, ils correspondent à des bus réels (OPB, PLB). C'est également à ce niveau que le partitionnement logiciel/matériel est réalisé. En effet, SpaceLib intègre des modèles de processeurs sous forme de simulateurs de jeu d'instructions. De plus, les primitives de communication pour les modules de l'application ont été implémentées sous forme logicielle. Ainsi, il est possible de faire passer un élément de l'application du domaine logiciel au matériel très facilement, sans changer le code du composant.

SpaceStudio SpaceStudio est l'environnement de développement de SpaceCodesign. Fondé sur l'environnement Eclipse, il facilite l'utilisation de SpaceLib. Son fonctionnement global est présenté figure 1.5. D'abord il offre une représentation graphique des configurations de simulation, utilisable pour les visualiser et les modifier. Ensuite, il apporte de nombreux outils, tels que la gestion des identifiants des composants et la génération d'adaptateurs de communication. À partir des configuration créées par l'utilisateur, il

génère des fichiers SystemC qui sont compilés et exécutés pour simuler le système. SpaceStudio peut aussi automatiser le passage d'une configuration Elix vers Simtek. Enfin il intègre Gen-X, qui est capable de générer un projet XPS à partir d'un modèle Simtek, en vue d'une implémentation sur FPGA.

Extensibilité et généricité SpaceStudio est donc orienté entièrement vers une utilisation de SpaceLib. Ce lien lui permet de profiter d'une connaissance complète du fonctionnement des composants de la bibliothèque. Ceux-ci sont donc correctement configurés dans les modèles qu'il crée, sans que l'utilisateur n'ait besoin de beaucoup de connaissances de Space.

Ce point fort induit un inconvénient important : le code de SpaceStudio est fortement lié au contenu de SpaceLib. Le code comporte de nombreux tests pour effectuer des actions spéciales en fonction du type des composants. Le besoin de fichiers de description indépendants du code s'est néanmoins fait sentir. Il existe donc de tels fichiers, qui permettent de définir les ports, les paramètres et des dépendances des composants. Cependant, le contenu de ces fichiers reste très lié à la plateforme Space. En particulier, l'insertion des adaptateurs pour connecter les modules de l'application aux bus est codée en dur. De plus il est conçu en fonction du paradigme « parent-enfant », qui représente les architectures sous forme d'arbres dans lesquels les bus sont les parents des modules qui y sont connectés. Cette conception, propre à SpaceStudio, lui permet d'afficher une vue en arbre facile à comprendre et à visualiser pour les utilisateurs. Cependant, elle n'est pas faite pour pouvoir s'appliquer à n'importe quel type de composant, possédant un nombre quelconque de ports. Pour résumer, les fichiers de données utilisés par SpaceStudio sont utiles pour éviter d'enregistrer des données sous forme de chaînes de caractères dans le code du programme. Mais ils restent profondément liés au paradigmes et fonctionnalités de SpaceStudio et par conséquent sont peu portables à d'autres environnements de développement.

Pour qu'une plateforme soit extensible, elle se doit d'être générique. Autrement dit elle doit être conçue de manière à pouvoir supporter tout nouveau type de composant. Les informations propres aux composants figurent alors dans des fichiers de données, distincts du code du logiciel. Ainsi pour ajouter (ou enlever) un élément de la bibliothèque, il suffit de modifier le contenu du dossier des données. Il n'est pas nécessaire de mettre à jour le programme ou de le recompiler pour qu'il intègre les changements.

Tel qu'indiqué précédemment, la plateforme développée dans le cadre de cette maîtrise est conçue selon ce principe de généricité. C'est son principal apport par rapport à Space Studio. Il existe de plus d'autres plateformes génériques supportant le haut niveau, telles que Magillem Platform Assembly [17]. Cependant, elles ne fournissent pas des outils simplifiant la manipulation des modèles tels que ceux présentés dans le chapitre 4.

Le format des fichiers décrivant les composants, leurs interfaces et les *designs*[1] est le standard IP-XACT. Il possède en effet les avantages d'être conçu spécifiquement décrire ces objets, et ne nécessite pas l'ajout de sémantique. De plus, pour favoriser l'interopérabilité, il est important de s'appuyer sur un standard déjà répandu dans l'industrie, ce qui est le cas de IP-XACT.

MoML [18] est un autre format de données, dont le but est de décrire des composants abstraits, paramétrés, et leurs interconnexions. Ces composants peuvent correspondre aussi bien à des processus qu'à des objets logiciels ou des blocs matériels, il est nécessaire de leur ajouter une sémantique pour que la plateforme puisse les manipuler. De nombreuses extensions auraient été nécessaires pour ajouter l'information dont la plateforme a besoin. IP-XACT a donc été choisi comme format pour décrire les composants, interfaces et designs.

[1] Selon la terminologie de IP-XACT, ce terme anglais désigne une configuration d'instances de composants interconnectées (cf. sec. 2.3.2). Afin de conserver le lien avec la norme, il est laissé dans sa version anglaise dans ce mémoire.

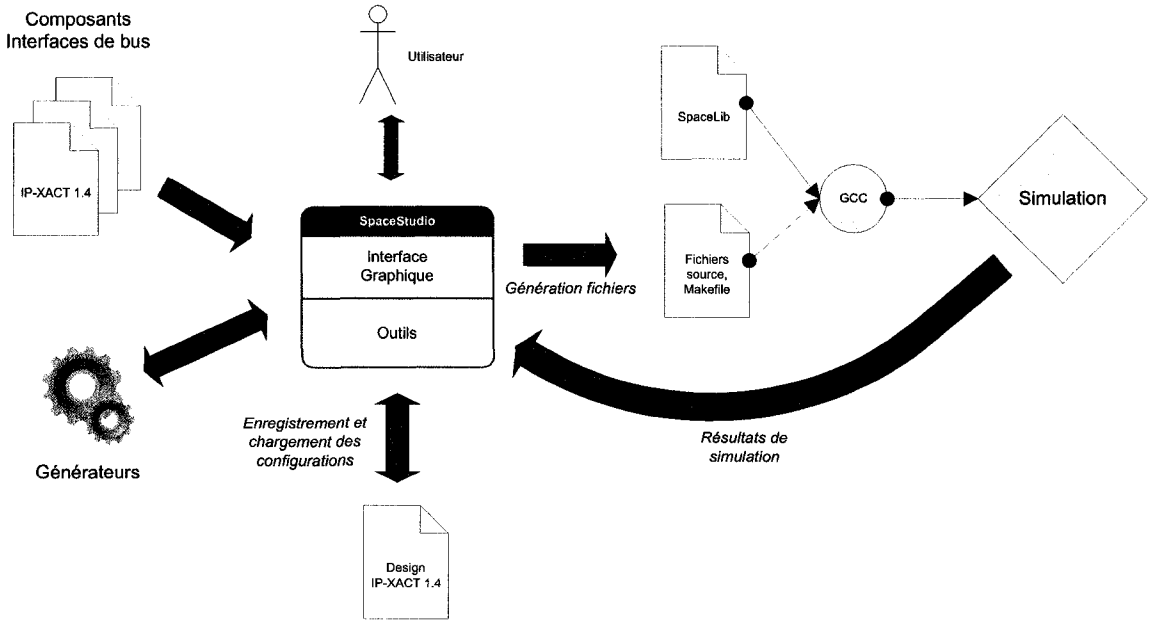


Figure 1.6: Vue globale de la plateforme

Une vue globale de la plateforme est proposée figue 1.6. Le chapitre suivant présente le standard, dont les paradigmes ont fortement influencé la conception du logiciel.

CHAPITRE 2

IP-XACT

2.1 Introduction

Tel que mentionné dans la section 1.3, l'industrie des systèmes sur puce est constituée de nombreux acteurs, jouant des rôles différents :

- Les créateurs d'IP ;
- les fournisseurs d'outils de conception (simulation, vérification, synthèse, placement et routage, etc.) ;
- les créateurs d'environnements de conception, qui offrent une interface utilisateur à ces outils ;
- les concepteurs de systèmes.

Dans le cas typique, ces derniers assemblent des IP pour créer des systèmes. Ils font pour cela appel à des outils de conception, par l'intermédiaire d'un environnement de conception.

Cette multiplicité des acteurs pose un problème important : comment échanger l'information ?

Les outils de simulation et de synthèse (entre autres) nécessitent une description du comportement des éléments à traiter. Il existe pour cela différents langages, comme VHDL et Verilog pour une description matérielle à bas niveau ou SystemC pour des niveaux d'abstraction plus élevés.

Ainsi, chaque composant peut exister sous différents formats simultanément : un modèle à haut niveau en SystemC, un assemblage d'entités VHDL, et une version synthétisée sous

forme de liste de nœuds *(netlist)*. De plus, il est possible de créer des modèles mixtes, qui mélangent plusieurs formats de composants. Ce type de modèle est particulièrement utile pour valider un module à bas niveau, en le simulant avec des modèles à haut niveau. On profite ainsi de la rapidité d'exécution des modèles à haut niveau.

Il est donc nécessaire de disposer d'un format de données regroupant ces informations : identification du composant, interfaces de connexion, paramètres et référence vers des descriptions internes. Ces informations sont appelées *métadonnées*, car elles se situent « au dessus » des descriptions vues précédemment. Autrement dit, elles offrent un point de vue plus global.

La norme IP-XACT [19] répond à ce besoin de métadonnées Elle a pour but de documenter de façon standard les composants, de manière à pouvoir les assembler dans des systèmes. IP-XACT est développé par un consortium d'industriels de l'EDA, le Spirit Consortium. Elle compte parmi ses membres les principaux acteurs du marché, en particulier dans son conseil d'administration (ARM, ST Microelectronics, Synopsys, Texas Instruments, etc.)

La première version de la norme est parue en 2004 et se nommait SPIRIT *(Standard Structure for Packaging, Integrating and Re-using IP within Tool-flows* — structure standard pour l'empaquetage, l'intégration et la réutilisation des IP dans la circulation entre les outils). Depuis la version 1.2, le format se nomme IP-XACT pour éviter la confusion entre le nom de du standard et celui du consortium qui le publie. La version 1.4, sortie en mars 2008 apporte une nouveauté importante : le support des modèles à haut niveau, et en particulier des ports transactionnels.

La documentation des composants repose sur le format XML *(extensible markup language)*. Il est normalisé par le W3C *(World Wide Web Consortium)*. Un document XML contient des balises, qui peuvent elles-mêmes contenir d'autres balises, des attributs et des données (textuelles ou numérales). Il est en outre généraliste et extensible, c'est-à-dire que les utilisateurs peuvent définir leur propres balises pour correspondre à leurs

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
// Description : interrupt.xml
// Author : SPIRIT Schema Working Group
// Version: 1.0
-->
<spirit:busDefinition
        xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4">
    <spirit:vendor>spiritconsortium.org</spirit:vendor>
    <spirit:library>busdef.interrupt</spirit:library>
    <spirit:name>interrupt</spirit:name>
    <spirit:version>1.0</spirit:version>
    <spirit:directConnection>true</spirit:directConnection>
    <spirit:isAddressable>false</spirit:isAddressable>
</spirit:busDefinition>
```

Figure 2.1: Exemple de fichier IP-XACT

besoins de représentation des données. Par conséquent, il est utilisé pour un très grand nombre d'applications (formatage de texte, services web, localisation géographique, etc.). De nombreuses bibliothèques logicielles conformes au standard sont disponibles, ce qui rend l'analyse de fichiers XML simple, robuste et conforme au standard. De plus, bien que XML soit conçu pour que ses fichiers soient analysés par les ordinateurs, il présente l'avantage d'être lisible par l'homme, ce qui facilite son utilisation.

Pour chaque domaine d'utilisation du XML, il est possible de créer des fichiers décrivant la syntaxe des documents. Ces fichiers indiquent notamment comment les balises peuvent se succéder et s'imbriquer, et quelles données elles contiennent. Il existe plusieurs types de ces fichiers de syntaxe. IP-XACT utilise des fichiers XSD *(XML Shema Definition)*, qui sont eux-mêmes des fichiers XML. En disposant d'une description formelle de la syntaxe, il est possible de vérifier de manière automatique qu'un fichier donné respecte bien cette syntaxe ; on parle alors de *validation*. Ainsi, lorsque l'analyse du contenu XML débute réellement, on sait qu'il respecte la syntaxe définie par le XSD.

Parmi les sept types de documents définis par la norme, nous utiliserons les quatre principaux :

busDefinition pour les informations générales sur un type de bus ;

abstractionDefinition pour décrire les mises en œuvre d'une interface à un niveau donné ;

component décrit un type de composant, qui possède notamment des interfaces, des ports et des paramètres ;

design instancie des composants, les paramètre et les connecte.

Chaque objet IP-XACT décrit par ces fichiers possède un identifiant unique nommé VLNV (Vendor, Library, Name, Version).

Les sections suivantes de ce chapitre présentent les paradigmes utilisés par IP-XACT pour décrire les composants et les designs. La section 2.2 présente comment sont spécifiées les connexions, la section 2.3 détaille comment les composants sont décrits, puis leur instanciation dans les fichiers design. Enfin, la section 2.4 offre un aperçu des possibilités laissées par la norme pour étendre sa fonctionnalité de base ; possibilités qui seront largement utilisées dans les chapitres 4 et 5.

2.2 Mode de connexion

2.2.1 Ports

Les ports peuvent être de deux types : *wire* pour les ports transportant un signal, comme les ports physiques ou *transactional* pour les ports transactionnels.

2.2.1.1 Ports transactionnels

Comme indiqué dans la section 1.1.2, les communications transactionnelles sont fondées sur des appels de fonctions inter-modules. En SystemC, avant la version 2.1, la méthode courante était la suivante : Une interface *IF* est définie sous forme de classe abstraite dérivant de *sc_interface*. Un module *B* implémente cette interface, la classe qui le définit dérive pour cela de *IF*. Un autre module *A* requiert cette interface : il possède pour cela un port de type *sc_port<IF>*.

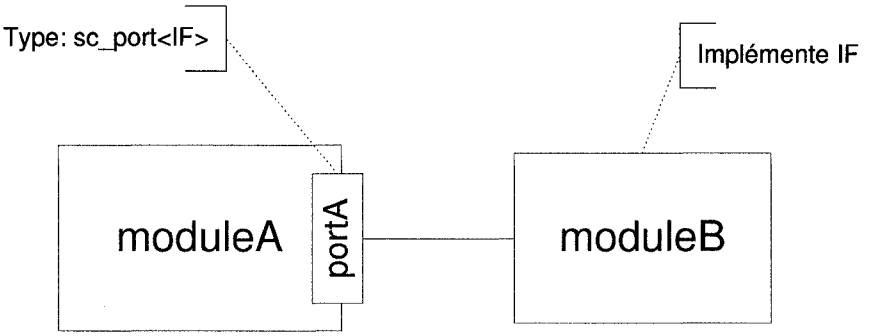


Figure 2.2: Communication port-module

Considérons une configuration possédant une instance de chacun de ces deux modules : *moduleA* et *moduleB* (voir figure 2.2). Pour qu'ils communiquent, le port de *moduleA* est connecté à *moduleB*. Ainsi *moduleA* peut faire des appels de fonctions par l'intermédiaire de son port, qui seront transmis à *moduleB*.

IP-XACT ne gère pas ce genre de connexion « port-à-module ». Il utilise plutôt un type de port apparu dans la version 2.1 de SystemC : *sc_export*. Le principe est de fournir le dual des *sc_port*. Alors que les ports requièrent une interface et doivent être connectés à une implémentation de l'interface, les exports déclarent qu'ils fournissent l'interface et reçoivent les appels de fonctions. On peut facilement faire une analogie, bien qu'imparfaite, avec les ports bas niveau, qui communiquent par signaux. Les *sc_port* correspondraient alors à des ports de sortie, les *sc_export* à des ports d'entrée, et les signaux seraient des appels de fonctions.

La figure 2.3 représente une communication dans laquelle l'interface est implémentée dans le module qui possède l'export. Dans cette situation précise, l'avantage des exports est loin d'être évident. Leur intérêt devient visible dans des cas plus complexes. Dans l'annexe A de [7], on trouve deux de ces situations. Le premier cas est celui où un composant fournit deux implémentations de la même interface (fig. 2.4a). Sans les exports, cela est impossible : on ne pourrait pas distinguer les deux implémentations de l'interface. Pour faire cette distinction, il suffit d'ajouter deux exports au module. Chacun est relié à une

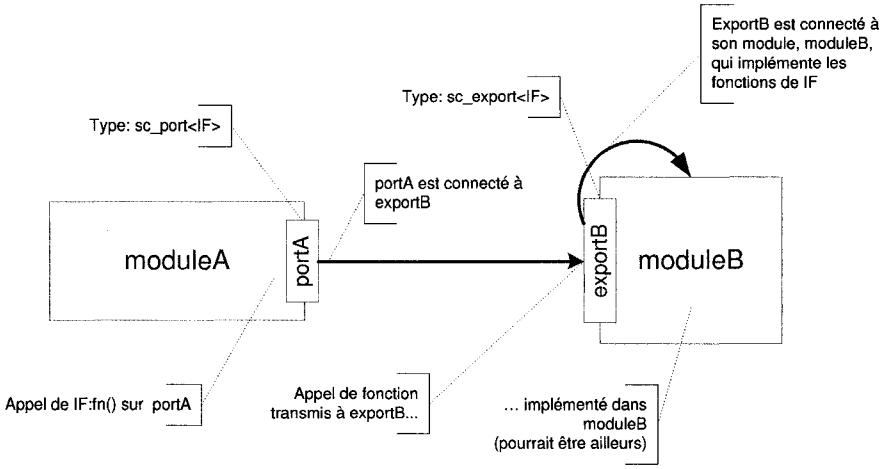


Figure 2.3: Communication port-export

implémentation de l'interface. Ainsi, lors de la connexion au module, avec des liens « port-à-export », on peut choisir l'export, et donc l'implémentation à utiliser. Le second cas est celui des modules hiérarchiques. Les exports permettent de transmettre directement les appels à un sous-module avec une connexion « export-à-export » (fig. 2.4b). Sans cela, il aurait fallu créer pour le super-module une implémentation de l'interface dans laquelle chaque méthode appelle la méthode correspondante dans le sous-module, ce qui est bien plus fastidieux.

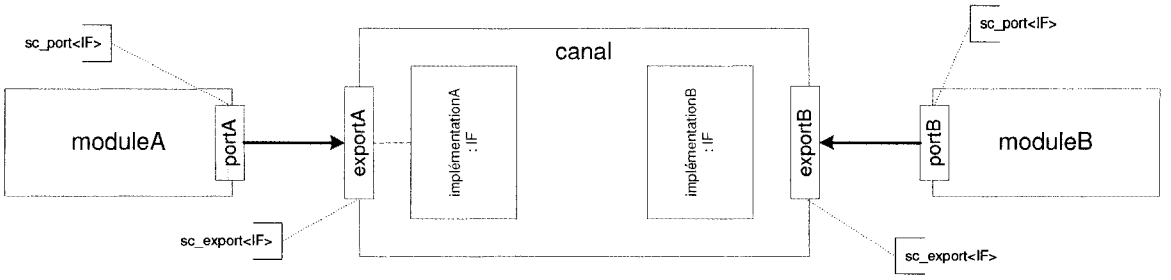
IP-XACT n'étant pas spécifique à SystemC, il utilise un vocabulaire différent pour décrire ces ports. Les ports transactionnels sont définis par deux informations :

l'initiative peut être *provides* (fournit) ou *requires* (requiert). Un *sc_port* requiert un service, un *sc_export* le fournit.

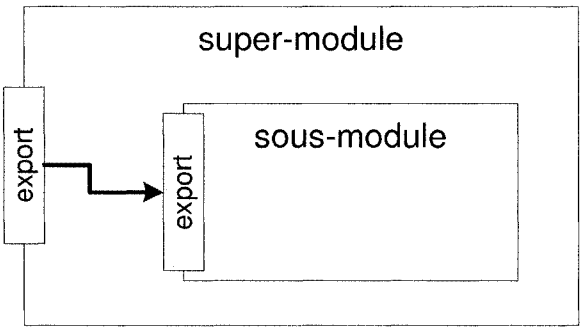
le service est l'interface fournie ou requise par le port.

2.2.1.2 Ports *wire*

Ces ports sont plus simples, cependant il est nécessaire de détailler la manière dont IP-XACT les représente. En effet, la particularité de la norme est que tous les ports de ce type



(a) Exportation de deux implémentation de la même interface



(b) Composants hiérarchiques

Figure 2.4: Cas d'utilisation des exports

ont une largeur unitaire.

Pour les ports de largeur supérieure, on utilise des ports de type *vecteur*. Ils correspondent aux ports dont la déclaration SystemC est du type suivant :

```
sc_in< sc_lv<16> > port;
```

La largeur du port doit être connue et fixe. En particulier les ports *sc_in<int>* ne sont pas autorisés, le type *int* n'ayant pas de largeur fixée en C++ (voir [20], section 2.3.1). De façon plus générale, tous les ports doivent être connus dans la définition des composants. Il n'est donc pas possible de déclarer un port conditionnel en utilisant une macro. Dans ce cas, il faut décrire autant de composants qu'il y a de possibilités.

Les ports peuvent donc porter un vecteur. Par contre, il n'est pas possible de déclarer les vecteurs et tableaux de ports comme *sc_in<bool>[16]* directement. Il faut alors les détailler un à un.

valeurs des bornes	$g_a = 7,\ d_a = 4,$ $g_b = 5,\ d_b = 2$	$g_a = 7,\ d_a = 4,$ $g_b = 2,\ d_b = 5$
connexions des éléments	va[7] - vb[5] va[6] - vb[4] va[5] - vb[3] va[4] - vb[2]	va[7] - vb[2] va[6] - vb[3] va[5] - vb[4] va[4] - vb[5]

Tableau 2.1: Connexion de deux vecteurs *va* et *vb*, avec pour bornes gauche et droite respectivement (g_a, d_a) et (g_b, d_b)

Ces limitations peuvent apparaître contraignantes. Mais il faut garder à l'esprit que le but de la norme est de se situer au dessus des langages de description. Elle se focalise donc sur le plus grand diviseur commun, en évitant les spécificités propres à tel ou tel langage.

Plus précisément, ce n'est pas la taille qui est mentionnée dans les fichiers IP-XACT mais l'intervalle des indices des éléments du vecteur. Cet intervalle est défini par ses bornes gauche et droite. Il n'y a pas de contrainte quant à l'ordre des limites : la borne gauche peut être indifféremment plus grande ou plus petite que la borne droite. L'ordre des bornes sera important lors de la connexion du vecteur de ports avec un autre vecteur de même taille. Lorsqu'on connecte deux vecteurs *va* et *vb* aux bornes (g_a, d_a) et (g_b, d_b), les éléments $va[g_a]$ et $vb[g_b]$ sont connectés, de même que $va[d_a]$ et $vb[d_b]$. Les autres éléments sont reliés en fonction de l'ordre des bornes, comme le montre l'exemple de la table 2.1.

2.2.2 Interfaces

Les interfaces sont un moyen commode de regrouper les ensembles de ports sur un composant. Ainsi, on peut spécifier une connexion de deux groupes de ports en une connexion « interface-à-interface », plutôt que détailler toutes les connexions « port-à-port ». En plus de cette fonction de regroupement, les interfaces permettent de préciser le rôle des ports en y ajoutant un identifiant. Par exemple, c'est de cette façon que l'on distingue un port d'horloge d'un port *reset*, qui sont par ailleurs en tout point identiques.

Cette possibilité sera détaillée dans le chapitre 4, en particulier dans les sections 4.2 et 4.4.

Sur les composants, les interfaces peuvent appartenir à différents modes. Nous nous limiterons aux quatre principaux qui sont : maître, esclave, maître miroir et esclave miroir. Les interfaces miroir sont les « reflets » de leurs équivalents directs. Plus précisément, elle sont identiques, sauf que les directions des ports *wire* sont inversées, de même que les initiatives des ports transactionnels. Typiquement, les interfaces miroir sont présentes sur les canaux de communication, tandis que les modules fonctionnels possèdent des interfaces directes. Dans le cas normal, seules les connexions maître – maître miroir et esclave – esclave miroir sont autorisées.

La description des types d'interfaces que les composants peuvent utiliser se trouve dans deux fichiers distincts. D'abord, chaque fichier *busDefinition* correspond à un type de bus. Seules des informations d'ordre général sur le bus y figurent. En particulier, il y est mentionné si le bus est adressable et si les connexions directes sont autorisées. Dans ce dernier cas, il est possible de connecter directement une interface maître à une interface esclave (sans interface miroir). Par exemple, le type de bus pour les interfaces d'interruptions est non adressable et direct (celui qui émet l'interruption est le maître, et il est connecté directement à celui qui la reçoit — l'esclave). Les bus courants (par exemple OPB, LMB) sont adressables (à chaque transaction est associée une adresse), et non directs (les modules maîtres et esclaves doivent être connectés au bus, et non directement l'un à l'autre).

La liste des ports à utiliser pour implémenter une interface dépend de son niveau d'abstraction. Elle figure donc dans le fichier *abstractionDefinition* correspondant. Chaque fichier de ce type est associé à un type de bus ; on y trouve donc en premier lieu l'identifiant VLNV du fichier busDefinition correspondant. Puis les ports à utiliser pour cette abstraction du bus sont listés. Chaque port est nommé, puis son type (transactionnel ou *wire*) est spécifié. Ensuite, pour les modes maître et esclave, si le port est présent sur ce mode, ses

caractéristiques sont données. Dans le cas d'un port transactionnel, il s'agit de son service et son initiative ; et pour un port *wire*, de sa direction et éventuellement sa largeur.

2.3 Composants et designs

2.3.1 Composants

Chaque modèle de composant est décrit dans un fichier *component*. Les informations essentielles qu'il contient sont ses ports et des interfaces, ainsi que ses paramètres.

Les ports du composant sont appelés ports *physiques*, par opposition aux ports *logiques* présents dans les définitions des abstractions d'interface. Ports physiques et ports logiques sont décrits d'après les mêmes informations : selon les cas, direction et largeur ou service et initiative.

Dans les déclarations d'une interface du composant, on trouve en premier lieu son nom, les identifiants de la définition de bus et d'abstraction, puis son mode. Ensuite la section *portMaps* (facultative) indique les équivalences entre les ports physiques du composant et les ports logiques de la définition de l'abstraction.

Les paramètres possèdent un nom, un identifiant et une valeur par défaut. En SystemC, ils correspondent aux paramètres du constructeur et éventuellement aux paramètres des *templates* génériques C++. En VHDL, les *templates* de IP-XACT correspondent aux *generic* des entités. Il est aussi possible de définir le type de leurs données : entier, flottant, chaîne de caractère ou bien choix parmi une énumération de possibilités.

2.3.2 Designs

Les designs sont les objets IP-XACT dans lesquels les composants sont instanciés. Dans ce type de fichier, on trouve donc après les identifiants du design la liste des instances de composants. Chacune d'entre elles possède un nom, et une référence vers le type de composant. Il s'agit de l'identifiant VLNV présent en tête du fichier *component* correspondant. Elles sont aussi paramétrées, en utilisant les identifiant des paramètres précédemment mentionnés. Il n'est pas obligatoire d'affecter une valeur à tous les paramètres. Dans ce cas, les valeurs par défaut définies dans le fichier *component* sont utilisées.

Puis les connexions entre les instances de composant sont listées. Il en existe deux types principaux : les connexions ad hoc et les interconnexions.

Les connexions ad hoc connectent les ports entre eux. Chaque connexion ad hoc est constituée d'au moins deux références vers des ports d'instances de composants du design. Pour qu'une connexion ad hoc soit possible, les ports concernées doivent être du même type (transactionnel ou *wire*) et posséder respectivement le même service ou la même largeur.

Chaque interconnexion connecte exactement deux interfaces sur deux instances de composants. Les interconnexions possibles connectent des interfaces dont les abstractions sont compatibles, ainsi que leurs modes. Les connexions autorisées sont donc maître – maître miroir et esclave – esclave miroir. Dans le cas où le bus est déclaré direct dans sa définition, les connexions maître – esclave sont également autorisées.

Si les correspondances entre les ports physiques(du composant) et logiques (des définitions d'interfaces) sont déclarées dans la définition du composant, il est possible de connaître l'ensemble de connexions ad hoc correspondant à une interconnexion.

2.3.3 Composants hiérarchiques

Les composants sont habituellement vus comme des boîtes noires, c'est à dire que l'intérieur du composant ne nous préoccupe pas. IP-XACT autorise néanmoins de décrire la structure interne d'un composant. Dans ce cas, le fichier component contient une référence vers le fichier de design en décrivant l'intérieur. Le fichier design correspondant est identique à ceux vus précédemment, sauf que les connexions peuvent faire référence à des interfaces situées sur le super-composant. Elles sont appelées « connexions hiérarchiques ».

2.4 Extensions

2.4.1 Ajout d'informations : les *vendorExtensions*

La syntaxe des fichiers IP-XACT est définie de façon précise, à l'aide du format XSD (voir sec. 2.1). La définition de la syntaxe est définie dans des fichiers de schémas XML. Il n'est donc pas possible d'ajouter des informations non standard à n'importe quel endroit des fichiers de description. Le schéma a néanmoins prévu certaines zones spéciales pour des données supplémentaires. Ces sections, nommées *vendorExtensions*, sont localisées à différents endroits définis précisément. Le contenu de ces balises est laissé entièrement libre, ce qui permet d'y placer tout type de données complémentaires.

Comme nous le verrons dans les chapitres 4 et 5, la plateforme présentée ici utilise beaucoup ces extensions pour ajouter les informations dont elle a besoin. Celles-ci devant obéir à une syntaxe particulière, un fichier XSD définissant cette syntaxe a été créé. Pour bien dissocier ces balises supplémentaires de celles de la norme, le mécanisme des espaces de noms XML [21] a été utilisé. Chaque élément des documents XML appartient ainsi à un espace de nom identifié par une URI (Uniform Ressource Identifier) [22]. L'URI des balises standard IP-XACT est `http://www.spiritconsortium.`

`org/XMLSchema/SPIRIT/1.4`[1], et celui des extensions de la plateforme `http://www.spacecodesign.com/XMLSchema/space/1.0`. Disposer d'un espace de noms propre à notre plateforme permet de distinguer nettement les extensions à traiter de celles qui auraient été ajoutées par des tiers. Le risque de conflit est donc évité : même si des balises créées par plusieurs acteurs portent le même nom, il est toujours possible de les distinguer d'après leur espace de noms.

2.4.2 Générateurs

Les *vendorExtensions* permettent donc d'ajouter des informations aux fichiers de description. Certes, ces données ne sont pas prévues pas la norme, mais le type d'informations qu'elles contiennent doit être défini au moment de la création de la plateforme de conception. Celle-ci les utilise ensuite dans des méthodes qui lui sont propres. Les extensions permettent donc d'ajouter à tout élément des fonctionnalités offertes par la plateforme. Par contre, il peut exister des composants nécessitant des fonctionnalités particulières. Par exemple un module peut avoir besoin de configurer un de ses paramètres en fonctions des propriétés des composants auxquels il est connecté. Ces fonctions sont donc propres au composant. Par conséquent elles ne doivent pas être incluses dans l'environnement de conception ; cela briserait le principe de généricité qui constitue notre principal objectif.

Pour résoudre ce problème, IP-XACT définit les *générateurs*. Ce sont des programmes — habituellement petits — qui accompagnent les fichiers de description dans la bibliothèque. Les descriptions des composants font mention des générateurs. L'environnement en a donc connaissance et peut les appeler au moment de la *génération*. Les générateurs sont alors capables d'analyser le design courant et de le modifier en conséquence.

[1] À titre de remarque, s'il est vrai que cet URI correspond à l'adresse d'une page web (URL), cela n'est pas obligatoire. Le rôle d'une URI est de fournir un identifiant universel unique, au même titre que le numéro ISBN pour les livres ; et non de localiser une ressource sur le Web.

L'interface qui permet aux générateurs d'interagir avec la plateforme est également normalisée. Cela est indispensable pour garantir l'indépendance des générateurs vis-à-vis de l'environnement qui les utilise. Cette interface s'appelle TGI : Tight Generator Interface (interface étroite de générateur). Il s'agit d'un ensemble de fonctions qui sont appelées en suivant le protocole SOAP. SOAP est un autre standard normalisé par le W3C [23]. C'est un protocole pour l'échange de données représentées en XML. Il est surtout connu dans le domaine des services Web, puisqu'il permet de faire des appels de méthodes à distance dans un environnement hétérogène. Il peut s'appuyer sur une variété de protocoles, dans la grande majorité des cas il s'agit du HTTP, habituellement utilisé pour transmettre le contenu de sites web.

Dans notre contexte, les échanges de données se font en local, sur une seule et même machine. Cependant l'utilisation de SOAP est intéressante car elle permet de réaliser ces appels de fonctions tout en garantissant l'indépendance des deux parties. Il s'agit alors en effet de deux programmes s'exécutant dans des espaces d'adressage distincts, qui communiquent par les couches réseau de l'ordinateur.

L'interface TGI est un ensemble de méthodes, de type get/set (obtention et réglage). Pour faire référence à des objets (par exemple une instance de composant), les deux parties utilisent des identifiants uniques, générés par l'environnement de conception. Ces méthodes d'accès et de modification des données consistent en des opérations de base sur le design. Elle n'ont pas accès aux fonctionnalités plus élaborées qui auraient pu être mises en œuvre dans la plateforme. Cette limitation est bien sûr nécessaire pour assurer la portabilité des générateurs : puisque les fonctionnalités supplémentaires sont propres à chaque environnement de conception (et facultatives), il est impossible de les prendre en compte dans l'interface normalisée. Les interactions des générateurs avec la plateforme sont donc limitées. Par conséquent, les générateurs doivent soit se limiter à des fonctions simples, soit implémenter eux-mêmes des outils plus élaborés. On arrive dans ce cas à une architecture telle que représentée figure 2.5. Elle illustre la notion *tight* (étroit) dans

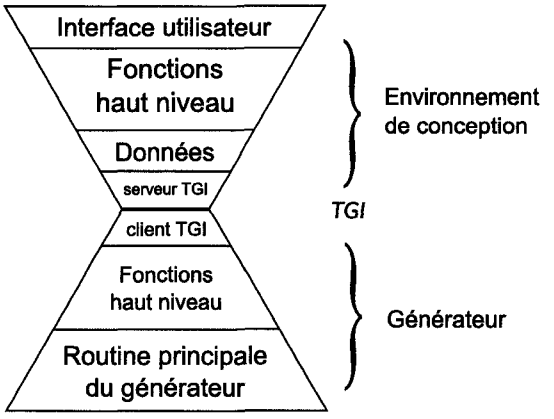


Figure 2.5: Modèle en couches entre l'environnement de conception et un générateur utilisant des fonctions de haut niveau

TGI, et ses conséquences : les fonctions à haut niveau de l'environnement de conception ne peuvent être partagées avec les générateurs. Si besoin, elles doivent donc être implémentées une seconde fois dans les générateurs, et même autant de fois que de générateurs en ayant besoin.

Évidemment, les méthodes se focalisent sur l'utilisation de données standard. Il est certes possible d'accéder aux sections vendorExtensions, mais elles sont peu pratiques à manipuler. Elles doivent en effet être transmises de manière brute, c'est à dire telles qu'elles apparaissent dans le fichier XML. Cela n'est pas approprié pour un traitement efficace de ces données.

Enfin, comme nous le verrons dans le chapitre 4, les générateurs sont appelés pour réaliser des opérations non standard. En particulier, certains sont responsables de créer de nouvelles classes de composant, pour les composants utilisateur. D'autres mettent en œuvre le patron de conception des « écouteurs » [24] pour assurer une mise à jour dynamique du design au gré des actions de l'utilisateur.

Ces trois raisons (données, utilisation et méthodes non standards) font que j'ai choisi d'utiliser, en plus de la TGI, une interface spéciale et plus directe. Cette interface n'utilise

pas le protocole SOAP, mais des appels directs de méthodes Java. Plus précisément, la plateforme (elle même implémentée en Java) charge dynamiquement les classes des générateurs et les exécute. Ces classes sont présentes dans la bibliothèque sous forme de fichiers compilés en code intermédiaire *(bytecode)*, possiblement regroupés dans des archives JAR. Ainsi, la séparation du programme principal et des fonctions propres aux composants est respectée, tout en offrant aux générateurs les mêmes capacités et performances.

Cinq types de générateurs sont utilisés par la plateforme :

userComponent Ils sont chargés de créer les composants utilisateur à partir d'un composant de base. Une grande partie du travail est réalisée par la plateforme, les générateurs effectuent les tâches propres au composant, en particulier la génération de fichiers source.

postInstanciation Ils sont appelés juste après l'ajout d'une instance du composant au design. Leurs modifications sont donc visibles pour l'utilisateur, qui pourra ensuite apporter ses propres modifications. Ils peuvent aussi s'enregistrer comme écouteur pour modifier dynamiquement le design.

preTransform Appelés juste avant un processus de raffinement, ils sont capables de sélectionner quel type de composant doit être utilisé dans le nouveau design.

postTransform Ces générateurs sont appelés après un raffinement. Ils sont en particulier capables de modifier le nouveau design d'après des caractéristiques de l'original.

normal Ils correspondent aux générateurs au sens de la norme. Ils peuvent soit utiliser la TGI, soit l'appel direct de méthodes Java.

Ils peuvent également posséder les propriétés standard pour les générateurs. La *phase* permet de définir l'ordre dans lequel les générateurs sont exécutés, dans le cas où un composant en possède plusieurs. Les générateurs dont la phase n'est pas définie sont exécutés en dernier.

D'autre part, la portée *(scope)* indique si un générateur doit être appelée une fois chaque

instance du composant dans le design considéré, ou une fois pour toutes les instances. Ce paramètre n'est utilisé que pour les types *postInstanciation* et *normal.*

Ce chapitre a offert un aperçu de la manière dont IP-XACT représente les objets utilisés pour la réprésentation des systèmes sur puce. Ces paradigmes se retrouvent dans la structure logicielle de la plateforme, comme on pourra le constater dans le chapitre suivant.

CHAPITRE 3

STRUCTURE LOGICIELLE

Le chapitre précédent a présenté les paradigmes utilisés par IP-XACT pour représenter les composants, designs et interfaces. Le développement de l'environnement de conception s'articule logiquement autour des mêmes concepts. Ce choix permet de faciliter la sauvegarde et la lecture de tels fichiers. Les objets conceptuels décrits par la norme existent sous forme d'objets logiciels lors de l'exécution du programme. Le respect de la norme est ainsi aisé.

Le chapitre présent détaille la structure globale du cœur de la plateforme. Il ne s'agit pas de faire l'inventaire de toutes les classes et leurs relations. La taille conséquente du projet (voir table 3.1) montre que cela serait long et d'un intérêt limité. Nous nous concentrerons donc sur les classes les plus importantes, que ce soit pour leur place dans la structure principale ou pour les réflexions qui ont été nécessaires à leur élaboration.

Pour éviter de se perdre dans les détails d'implémentation inutiles à la compréhension du logiciel, certains détails sont volontairement omis. Par exemple, pour que la taille des classes représentant les composants ne soit pas trop grande, elles possèdent un attribut *portSet* spécialisé dans la gestion de ses ports. Il faut donc passer par cet objet pour manipuler les ports des composants. Comme il n'apporte pas d'information utile à la compréhension du logiciel, nous considérerons dans la suite que les ports sont stockés directement dans les composants, ce qui n'est pas totalement exact. De même, dans les diagrammes de classes UML, les attributs et les méthodes ne seront pas énumérées si ce n'est pas nécessaire.

Ce chapitre permet donc de poser les bases du projet, qui seront utilisées dans les chapitres

Lignes de code (total)	19112
Lignes de code de méthode	10691
Nombre de classes	274
Nombre de méthodes	1651
Nombre d'interfaces	22

Tableau 3.1: Statistiques du projet Java

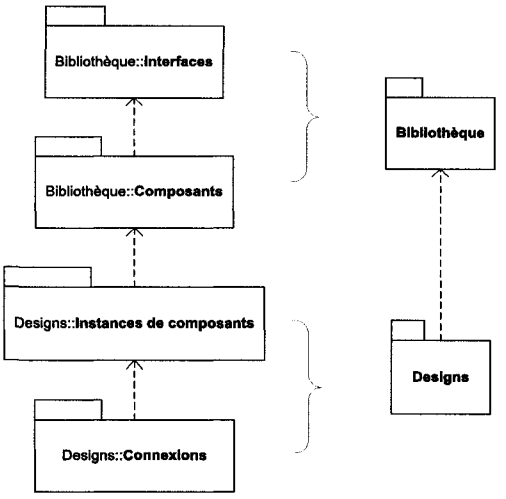


Figure 3.1: Dépendances globales de la plateforme

4 et 5 par les outils de conception.

Le cœur de la plateforme se divise en deux grandes parties distinctes (fig. 3.1). La première correspond à la bibliothèque. Elle contient donc les descriptions des classes de composants, ainsi que les définitions de bus. Dans la seconde partie, on trouve tout ce qui concerne les designs. Les designs utilisent des instances des éléments de la bibliothèque (composants, ports, interfaces). La partie de gestion de la bibliothèque est entièrement indépendante par rapport aux classes décrivant les designs ; c'est à dire qu'il n'y a aucune utilisation des classes relatives au design dans les classes gérant la bibliothèque.

Élément de document IP-XACT	Classe Java correspondante
spirit:busDefinition	BusType
spirit:abstractionDefinition	AbstractionType
spirit:port (dans fichier *abstractionDefinition*)	AbstractionTypePort
spirit:onMaster (dans un port)	LogicalPort
spirit:onSlave (dans un port)	

Tableau 3.2: Équivalences entre éléments IP-XACT et classes Java pour la définition des interfaces

3.1 Gestion de la bibliothèque

La bibliothèque se divise elle-même en deux parties : les définitions d'interfaces de bus et celles des composants. La définition des composants est dépendante de celle des interfaces de bus, mais la spécification des types d'interfaces peut se faire indépendamment des composants pour lesquels elles seront utilisées. Ici aussi, les paquetages Java ont été conçus de manière à refléter cette hiérarchie des dépendances.

3.1.1 Définition des interfaces de bus

Comme il en a été fait mention dans la section 2.2.2, les types d'interfaces sont définis dans deux fichiers distincts : les fichiers *busDefinition* pour les types de bus, et *abstractionDefinition* pour les types d'abstraction. Chaque type d'abstraction correspond à un unique type de bus, et il peut exister plusieurs types d'abstraction pour un type de bus. Chaque type d'abstraction liste ses ports, avec le cas échéant le détail des variantes sur les interfaces maître et esclave. L'implémentation reflète naturellement ces objets et leurs relations : la table 3.2 détaille les équivalences entre les éléments IP-XACT et les classes Java, et leurs relations de composition se trouvent dans la figure 3.2.

La figure 3.2 montre les relations de composition des classes définissant les interfaces de bus. Chaque type de bus possède plusieurs types d'abstraction. Ceux-ci possèdent plusieurs

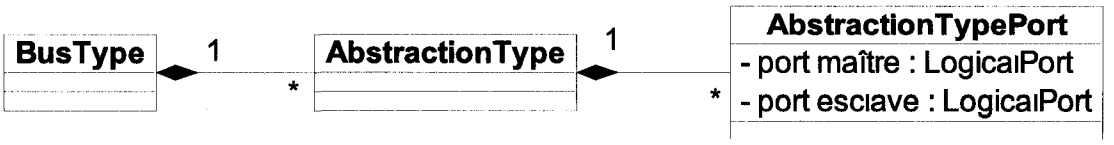


Figure 3.2: Relations de composition dans la définition des interfaces

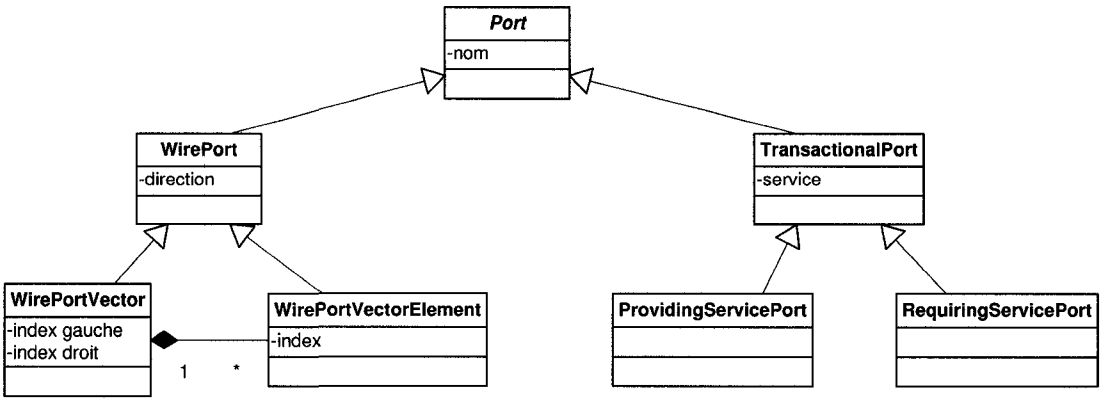


Figure 3.3: Diagramme de classes pour la gestion des ports de composant

ports. Chacun d'entre eux est enregistré sous forme de deux variantes : le port logique maître et le port logique esclave. Les propriétés de ces deux variantes peuvent être très différentes. La seule contrainte est qu'ils doivent être soit tous les deux transactionnels, soit tous les deux *wire*. Pour les modes miroir, les ports n'ont pas besoin d'être détaillés, puisque ce sont simplement les symétriques de leurs équivalents directs.

3.1.2 Composants

Chaque composant possède essentiellement des ports, des interfaces, de paramètres, ainsi que les chemins de ses fichiers source. Cette section détaille la représentation des ports et des interfaces, les classes correspondant aux paramètres et aux fichiers étant relativement simples.

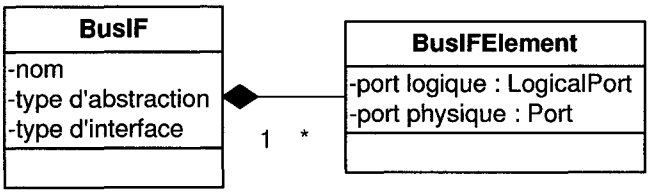


Figure 3.4: Interfaces de bus dans les composants

3.1.2.1 Ports

Les relations d'héritage présentées figure 3.3 correspondent au différents types de ports que l'on peut rencontrer. De même que pour les interfaces, les classes représentant les ports correspondent à la manière dont IP-XACT les décrit (sec. 2.2.1). D'abord, les ports possèdent deux natures : les ports *wire* et transactionnels. Les ports *wire* sont caractérisés par leur direction. Par défaut, ils ont une largeur unitaire. Dans le cas contraire, on définit un vecteur de ports, qui possède comme propriétés les limites gauche et droite des indices de ses éléments.

Les ports transactionnels sont caractérisés par leur service et leur initiative. Une classe par initiative a été créée : les ports qui requièrent un service (en SystemC : les *sc_ports*) et ceux qui en fournissent un (*sc_exports*) ;

3.1.2.2 Interfaces

Pour les interfaces sur les composants[1] aussi, les classes Java respectent les paradigmes IP-XACT. Leur définition est assez simple (figure 3.4). Les objets *BusIF* possèdent comme attributs le type d'abstraction et le type d'interface (maître, maître miroir, esclave, esclave miroir). Ils agrègent de plus un ensemble d'objets de la classe *BusIFElement*. Ceux-ci font la correspondance entre les ports physiques du composant et les ports logiques dans la définition de l'abstraction. Il est donc possible de trouver l'ensemble des connexions ad hoc

[1] à ne pas confondre avec la définition des types d'interfaces section 3.1.1

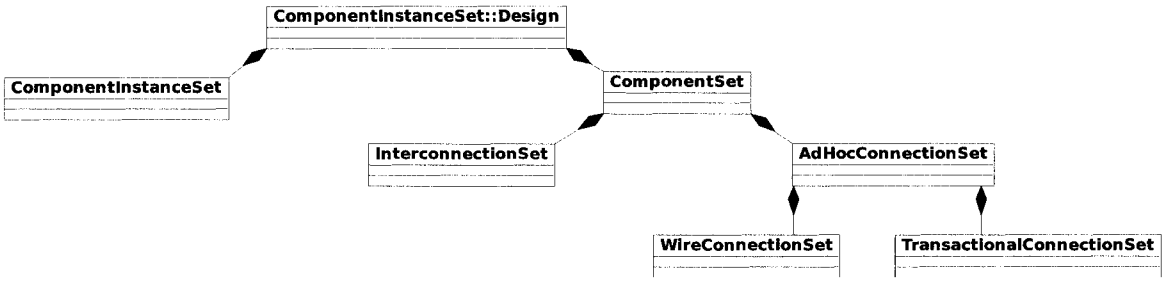


Figure 3.5: Gestionnaires des éléments de design

(port-à-port) équivalentes à une interconnexion (interface-à-interface).

3.2 Designs

Les designs sont les objets IP-XACT dans lesquels les composants sont instanciés. Une fois ces composants instanciés, il est possible de les connecter. Les objets décrivant les designs correspondent donc à deux parties distinctes : d'un côté la gestion des instances de composants, de l'autre les connexions. Comme il l'a été montré figure 3.1 page 39, la définition des connexions dépend des instances de composant, mais pas le contraire. De plus, la gestion des connexions est séparée en sous-parties, chacune étant dédiée à un type particulier. La figure 3.5 détaille les classes utilisées pour gérer les éléments des designs. Les classes qui apparaissent en grisé ne sont pas visibles publiquement. Leurs méthodes sont appelées par l'intermédiaire du *ConnectionSet* qui les contient.

3.2.1 Instances de composants

Les instances de composant d'un même design sont regroupées dans un objet de classe *ComponentInstanceSet*. Cette classe a pour fonction de stocker les instances de composants, mais aussi d'offrir des méthodes permettant de les gérer facilement.

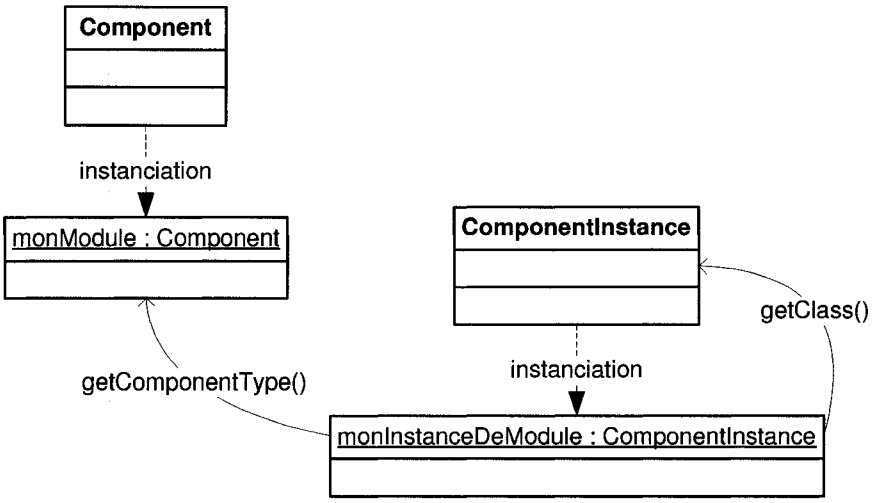


Figure 3.6: « Métaclasse » pour les composants

Classes, objets et métaclasses Il est ici nécessaire de réfléchir sur la signification des classes choisies. Pour commencer, rappelons la définition d'un objet, selon Meyer [25].

Un objet est une instance à l'exécution d'une classe.

Dans le cas présent, nous avons une classe *Component* dont les objets (lors de l'exécution) sont des types de composant. D'autre part, les objets de la classe *ComponentInstance* représentent des instances de composant, présents dans les designs. Ainsi, la classe Component est en quelque sorte une métaclasse, définie ainsi par Meyer [25] :

Une métaclasse est une classe dont les instances sont elles-mêmes des classes.

Les instances de *Component* ne sont pas les classes des objets de *ComponentInstance* au sens strict (et Java) du terme. La classe *Component* n'est donc pas une *vraie* métaclasse. La raison principale est qu'en Java, la classe *Class* est déclarée *final*, c'est à dire qu'elle ne peut pas être surchargée. Il n'est donc pas possible de définir de nouvelles métaclasses en Java.

Un mécanisme parallèle a donc été utilisé : les objet de classe *ComponentInstance* possèdent la méthode *getComponentType()*. Elle donne accès à l'objet *Component* représentant le type de composant correspondant,. Elle est utilisée au lieu de la méthode Java *getClass()*

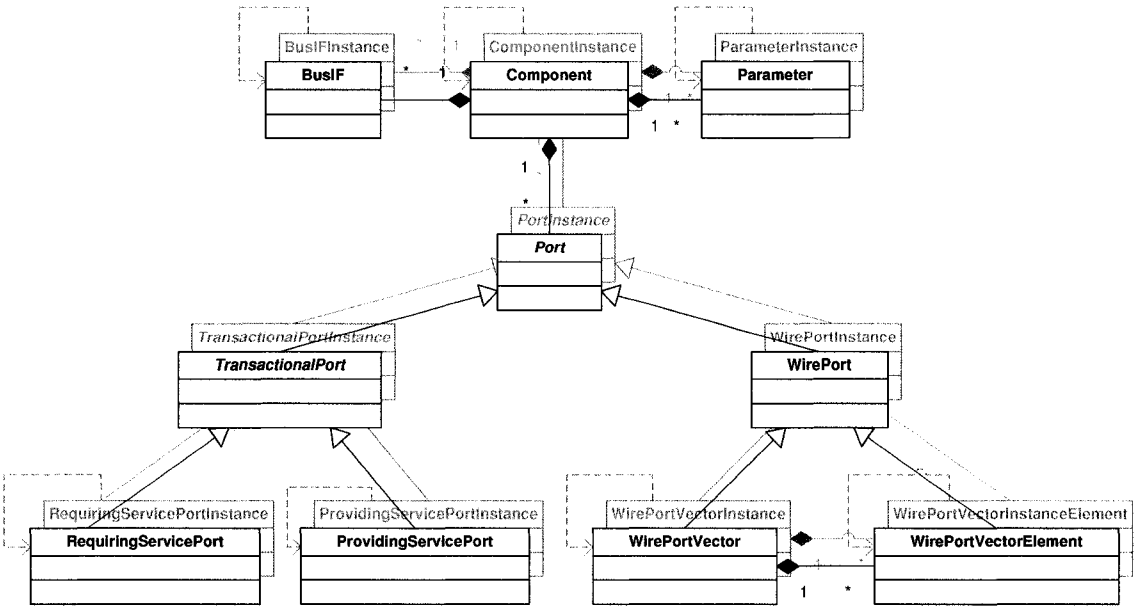


Figure 3.7: Diagrammes de classes de description des instances et types de composants

qui permet d'accéder à la classe de l'instance (voir figure 3.6[2]).

L'utilisation des métaclasses se justifie par les choix de IP-XACT : les designs sont constitués d'instances de composants *(componentInstance)*, chacune d'entre elles faisant référence à son type de composant *(component)* par son identifiant VLNV. Le choix de suivre les paradigmes IP-XACT impose donc l'existence de ces deux classes. De plus, il est utile de faire une référence directe de l'instance vers sa classe. En effet, le programme nécessite souvent d'accéder à cette classe pour en lire des propriétés. Si l'instance ne possédait pas la référence, elle devrait posséder l'identifiant VLNV, puis devrait rechercher l'objet *Component* correspondant dans la table des composants. La référence directe permet donc un gain de mémoire (les objets *ComponentInstance* contiennent une référence au lieu d'un identifiant de quatre chaînes de caractères) et de performance (l'objet *Component* est accédé directement).

[2]Ce diagramme ne respecte pas la norme UML, en particulier pour la représentation des liens. Le mélange de classes et objets n'est pas prévu dans les représentations normalisées UML.

Dualité type de composants / instances de composants La figure 3.7 montre qu'il existe une certaine dualité dans le diagramme de classes. Pour la comprendre, il est nécessaire de définir les deux domaines duaux. D'un côté, on a la description des *types de composants*. Elle a été traitée dans la section 3.1.2. Pour décrire chaque type de composant, on définit aussi ses ports, interfaces et paramètres. De l'autre côté, on a les définitions des *instances de composants*, telles qu'elles apparaissent dans le design. On peut voir dans le diagramme que chaque classe du domaine des instances de composant (en grisé) possède son équivalent dans le domaine des types de composants (en noir). La raison de cette similarité des deux diagrammes de classes est simple : la réalité à modéliser est similaire dans les deux cas. Une fois qu'un modèle satisfaisant a été trouvé pour le domaine des types de composants, il est donc logique d'appliquer le même modèle objet pour le domaine des instances de composant.

De plus, chaque objet qui participe à la description d'une instance de composant a une référence vers l'objet équivalent dans la description du type de composant. C'est le sens des relations de dépendance (flèches en pointillé) sur le diagramme. Par exemple, un port *wire* simple sur une instance de composant (classe *WirePortInstance*) a accès à l'objet de classe *WirePort* correspondant sur la description du type de composant. L'intérêt de cette méthode est que les objets décrivant les instances ont directement accès aux propriétés liées aux types de composant. Toujours pour l'exemple du port, il n'est pas nécessaire de créer des attributs correspondant au nom et à la direction du port. Ceux-ci sont en effet directement accessibles via la référence vers l'objet correspondant au type de port. Il n'y a donc pas de redondance des données, avec ses défauts (risque d'inconsistance, occupation supplémentaire de mémoire). On constate par ailleurs que l'ordre des dépendances est respecté : les classes du domaine des instances de composant dépendent de celles du domaine des types de composant, et le contraire est faux.

3.2.2 Connexions

Si on omet les connexions hiérarchiques, qui sont des cas particuliers, les connexions sont de deux types : les connexions ad hoc connectent au moins deux ports, et les interconnexions relient exactement deux interfaces.

Les connexions ad hoc se divisent elles-mêmes en deux catégories. Il est en effet illégal de mélanger des ports transactionnels et *wire* dans une même connexion ad hoc ; chaque connexion ad hoc appartient donc à l'un ou l'autre de ces deux types.

Ainsi, on a trois types de connexions, chacun d'entre eux étant associé à un type d'élément connectable :

type de connexion	éléments connectés
ad hoc *wire*	instances de ports *wire*
ad hoc transactionnelle	instances de ports transactionnels
interconnexion	instances d'interfaces de bus

Le diagramme des classes représentant les connexions respecte cette classification, tel que présenté à la figure 3.8.

Toutes les connexions d'un même design sont regroupées dans un même objet de type *ConnectionSet*. Afin de limiter sa complexité, il est composé de plusieurs objets, chacun gérant un type de connexion. Ces objets appartiennent chacun à une classe différente, spécialisée pour le type de connexion à traiter. Elles reposent néanmoins toutes sur une table de hachage, dont les clés sont les objets connectés, et les valeurs sont les connexions elles-mêmes. Par exemple pour une connexion ad hoc reliant deux ports portA et portB, on insérera cette connexion dans la table pour les clés portA et portB. Ainsi, il est rapide de retrouver les connexions associées à un objet connectable : cette opération se fait en temps constant.

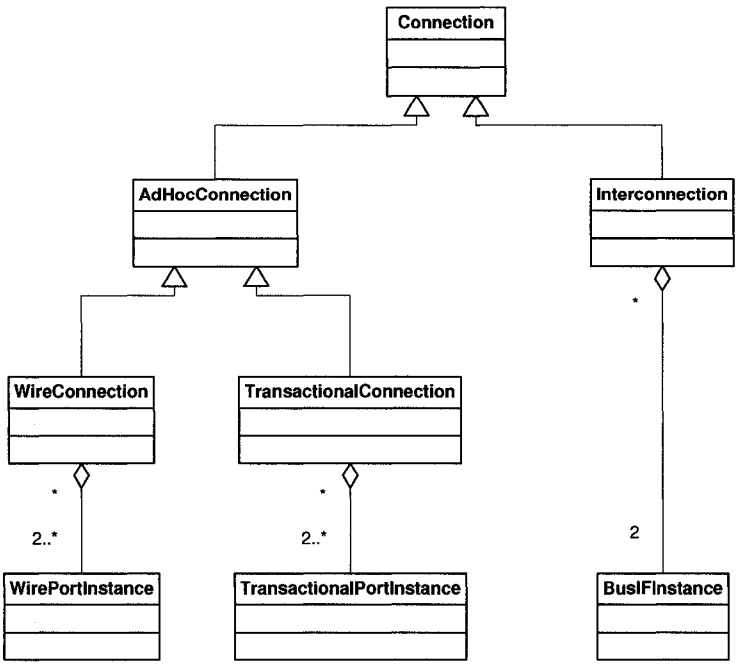


Figure 3.8: Relations d'héritage et de d'agrégation des classes de connexion

Les connexions *wire* exigent d'être manipulées de manière particulière, en raison des vecteurs de ports. Chaque *WireConnection* regroupe plusieurs objets de la classe *WirePortInstance*. Cette classe est abstraite et possède quatre sous-types :

- les ports simples
- les vecteurs entiers
- les éléments de vecteurs
- les parties de vecteurs

Ce dernier type permet de regrouper un intervalle d'éléments de vecteur. IP-XACT autorise cette utilisation des vecteurs de ports, qui permet d'éviter de détailler toutes les connexions élément par élément dans le cas où on ne souhaite connecter qu'une partie du vecteur. La figure 3.9 expose un tel exemple.

Une nouvelle classe d'instance de vecteur a été créée pour gérer la situation. Elle se nomme *SubVectorPortInstance*. Elle possède une référence vers le vecteur dont elle forme une partie, et les bornes son intervalle d'éléments. Notons (g_s, d_s) les bornes du sous-vecteur

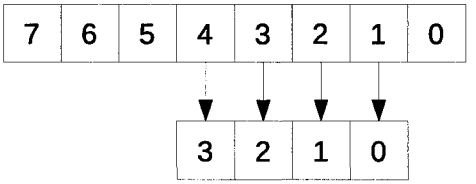


Figure 3.9: Connexion d'une partie de vecteur

s et (g_v, d_v) celles de son vecteur principal v. Alors les bornes de s doivent respecter les contraintes suivantes :

1. $\{g_s, d_s\} \subset \{g_v, \ldots, d_v\}$
2. si $g_v \leq d_v$, alors $g_s \leq d_s$, et si $d_v \leq g_v$, alors $d_s \leq g_s$

De plus, on définit les égalités suivantes :

1. si $g_s = d_s$, alors s est égal à l'élément de v d'indice g_s ;
2. si $g_s = g_v$ et $d_s = d_v$ alors $s = v$;
3. Soit s' de bornes $(g_{s'}, d_{s'})$ un second sous-vecteur de v. Si $g_s = g_{s'}$ et $d_s = d_{s'}$, alors $s = s'$.

Comme les autres gestionnaires de connexions, celui des connexions *wire* doit être capable de trouver rapidement les connexions utilisant une instance de port donnée. Dans le cas présent, la tâche est un peu plus compliquée. D'une part, si une connexion implique un vecteur de ports, il faut pouvoir la retrouver si on cherche les connexions impliquant un élément de ce vecteur. Réciproquement, à partir d'un vecteur, on doit être capable de trouver les connexions n'impliquant que des parties de ce vecteur.

En plus de la table principale stockant les connexions, une seconde table est chargée de référencer les parties de vecteurs impliqués dans les connexions. Pour cela a été créée une interface regroupant toutes les classes d'objets susceptibles de représenter des parties de vecteurs. Cette interface apparaît dans la figure 3.10, ainsi que ses classes filles. On constate qu'elle dispose d'une méthode *getVector*, qui renvoie le vecteur principal.

La figure 3.11 représente les objets manipulés par le gestionnaire de connexions *wire*.

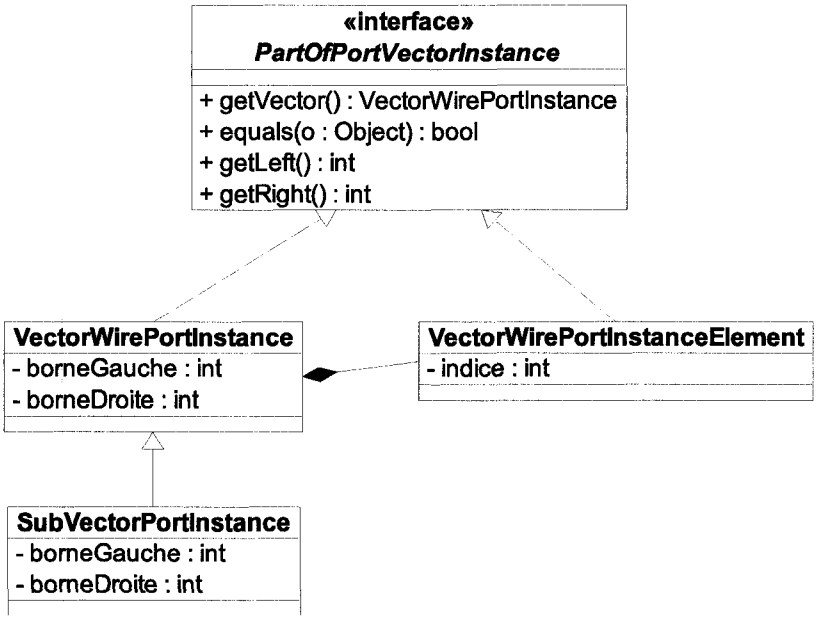


Figure 3.10: Interface *PartOfPortVectorInstance*

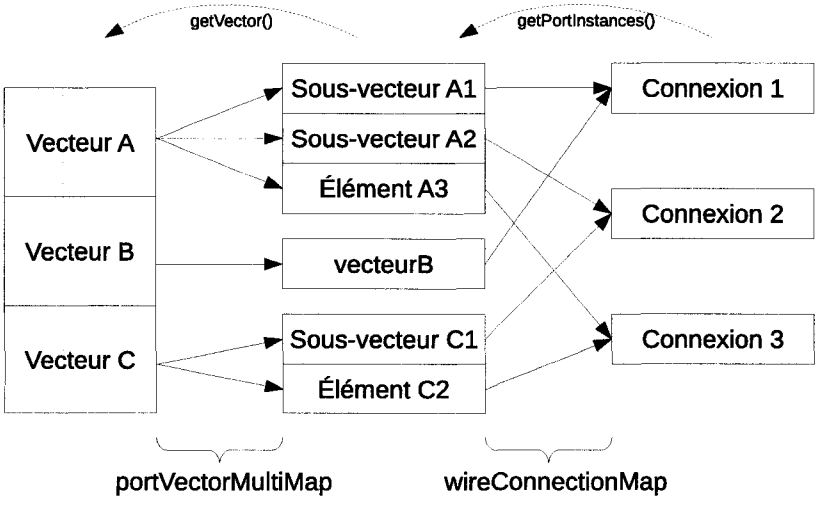


Figure 3.11: Tables de hachage du gestionnaire de connexions *wire*

Plus exactement, il s'agit de tables de hachage multiples : chaque clé peut être associée à plusieurs valeurs. Leur implémentation utilise des tables de hachage simples, dont les valeurs sont des listes chaînées.

Pour accéder aux connexions d'un élément de vecteur, on commence par accéder à son vecteur via la méthode *getVector()*. Avec la première table de hachage, on a accès à toutes les parties de ce vecteur qui sont connectées. À ce stade, on peut filtrer celles qui nous intéressent : certaines d'entre elles peuvent ne pas contenir l'élément étudié. Enfin, la dernière table donne accès aux connexions.

3.3 Projets

La notion de projet n'est pas présente dans IP-XACT. Cependant, elle est utile dans un environnement de conception.

Un projet regroupe un ensemble d'éléments destinés à la création d'un même système. On y trouve donc d'une part un ensemble de composants utilisateur utilisés pour ce système. Il s'agit principalement des composants représentant l'application à implémenter, mais on peut aussi trouver des périphériques d'architecture. D'autre part, un projet contient différents designs du système. Ils diffèrent par leur niveau d'abstraction ou par leur architecture.

Du point de vue de la structure logicielle de la plateforme, la classe représentant les projets contient donc un ensemble de designs et des objets *Component*. Ces derniers sont ajoutés parmi les composants de la bibliothèque lors du chargement du projet, puis retirés lorsque le projet est fermé.

Chaque projet correspond à un répertoire dans le système de fichiers. La structure d'un projet utilisant SystemC comme langage de description est représentée figure 3.12. À

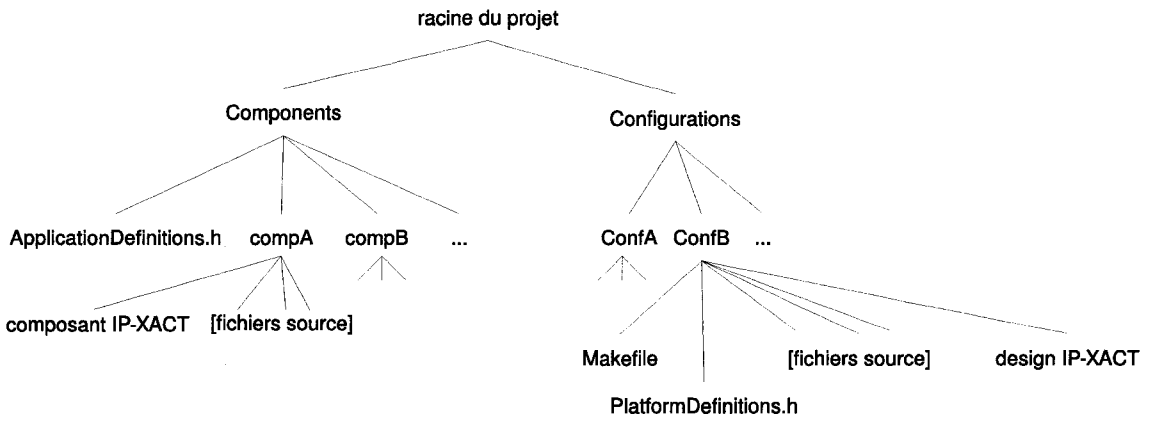


Figure 3.12: Structure de répertoire d'un projet SystemC

la racine du répertoire des composants, on trouve le fichier *ApplicationDefinitions.h.* Ce fichier n'est généré qu'une fois, à la création du projet. L'utilisateur peut alors y ajouter des définitions de constantes qui seront partagées entre les composants de l'application. Ensuite, on trouve dans le répertoire de chaque composant son fichier de définition IP-XACT, et les fichiers SystemC le définissant. Habituellement, les composants ont un fichier d'en-tête *(header)* et un fichier d'implémentation. Lors de la compilation, un fichier objet est ajouté.

De l'autre côté, un dossier est associé à chaque design du projet. Il contient la définition IP-XACT, qui permettra d'enregistrer le design sous une forme persistante. Le fichier *PlatformDefinitions.h* contient des définitions de constantes propres à chaque design. Ces constantes sont définies sous forme de table de hachage à l'intérieur de chaque design. Les clés de la tables sont les noms de variables, auxquels on associe leurs valeurs. Les générateurs peuvent créer de nouvelles paires variable-valeur, qui apparaîtront dans le fichier *PlatformDefinitions.h.* Pour les configurations Space, il s'agit des identifiants des composants.

On trouve aussi le *Makefile*, qui contient les instructions et dépendances de compilation, ainsi que les fichiers source principaux de la plateforme. Dans le cas où le design contient

des modèles de processeurs, il peut également y avoir des sous-répertoires, ce qui permet de séparer les fichiers utilisés pour la génération du logiciel de chaque instance de processeur.

Tous les fichiers que nous venons de voir dans cette section sont générés automatiquement. L'utilisateur ne doit modifier que les fichiers SystemC décrivant les composants, et éventuellement le fichier *ApplicationDefinitions.h*.

CHAPITRE 4

OUTILS DE LA PLATEFORME

4.1 Introduction

Le chapitre précédent a présenté comment élaborer une plateforme de conception haut niveau fondé sur IP-XACT. Les deux premiers objectifs du chapitre 1 sont donc remplis : utilisation des hauts niveaux et généricité. Comme on l'a vu dans la section 1.3, il est nécessaire d'y ajouter des outils pour simplifier la tâche de l'utilisateur. Dans la plupart des cas, ces outils ont besoin d'informations supplémentaires par rapport à celles prévues par la norme IP-XACT. Des sections d'extensions sont donc ajoutées aux fichiers de description. Pour assurer l'interopérabilité, ces extensions ne sont pas indispensables au fonctionnement global de la plateforme. Dans certains cas, les éléments supplémentaires possèdent une valeur par défaut. Par exemple, les composants sont considérés par défaut comme fonctionnels par le générateur d'adaptateurs (section 4.4). Dans d'autres cas, l'absence de balise implique que l'outil ne s'applique pas pour l'élément considéré. Ainsi la création des connexions pour les signaux obligatoires (section 4.2) n'est pas exécutée pour les interfaces ne mentionnant pas cette fonctionnalité.

L'extension qualifiant le type de composant est exploitée par plusieurs des outils décrits dans ce chapitre. La liste complète des types utilisés par la plateforme se trouve à l'annexe II.

4.2 Signaux obligatoires

Les composants utilisés ont en général besoin d'un signal d'horloge et d'un signal de réinitialisation *(reset)* pour fonctionner correctement. Dans certains cas, ces signaux sont transmis parmi d'autres dans une interface de bus. C'est le cas pour les interfaces OPB *(On-chip Peripheral Bus)* par exemple. Un composant connecté à tel bus dispose donc des signaux reset et horloge. D'un autre côté, ces ports spéciaux doivent parfois être connectés spécifiquement. Le bus OPB lui-même doit évidemment être relié à une horloge pour pouvoir fournir ce signal à ses périphériques.

Lorsqu'un composant ayant ce type de port est ajouté à un design, if faudrait que ces connexions puissent se faire sans intervention de la part de l'utilisateur de la plateforme.

Pour déclarer qu'un port doit être connecté automatiquement à un type de signal donné, on ajoute une interface au fichier IP-XACT du composant. Cette interface contient le port à connecter et une balise nommée *mandatorySignalInterface* dans ses extensions. Un exemple est présenté dans l'annexe I.1 p. 115.

L'outil de connexion automatique utilise la notion de *domaine* des composants pour le type d'interface à connecter. Cette notion est courante pour les horloges, mais elle peut s'étendre à tout type de signal. Elle est utile dans les cas où il existe plusieurs sources pour le type de signal considéré. Le but pour l'outil est d'évaluer à laquelle le port doit être connecté. En cas d'erreur, l'utilisateur peut toujours corriger le design.

Afin de définir clairement la notion de domaine telle qu'elle est considérée par l'outil, il est nécessaire de poser la définition préliminaire suivante :

Définition 4.1. Propositions *(1)* et *(2)*

Soit une instance de composant ci ;

Soit un type d'interface de bus $ifType$;

Soit une interface de bus if de type $ifType$ sur une instance de composant dans le même design que ci.

Soit $B_{ci,ifType}$ l'ensemble des interfaces de ci pouvant se connecter légalement à if.

On suppose que toutes les interfaces de $B_{ci,ifType}$ sont connectées à au plus une interface.

On définit alors récursivement les propositions *(1)* et *(2)* suivantes :

(1) Au moins une interface de $B_{ci,ifType}$ est connectée et toutes les interfaces connectées de $B_{ci,ifType}$ le sont avec if.

(2) Aucune interface de $B_{ci,ifType}$ n'est connectée et tous les voisins v de ci respectent soit *(1)* soit *(2)* en remplaçant « ci » par « v ».

On établit alors facilement la propriété 4.1.

Propriété 4.1. *Soit une instance de composant ci ;*
Soit un type d'interface de bus $ifType$.
S'il existe une interface de bus if telle que la proposition (1) ou (2) est vraie, alors elle est unique.

Démonstration Soient ci et $ifType$ tels que définis dans la définition 4.1.

Soient if_1 et if_2 deux interfaces pour lesquelles la proposition *(1)* ou *(2)* est vraie. Montrons que $if_1 = if_2$.

- Si *(1)* est vraie, alors il existe au moins une interface connectée de $B_{ci,ifType}$. Soit if_{ci} l'une d'entre elles.

  if_{ci} est donc connectée à à if_1 et à if_2. Par hypothèse de l'unicité des connexions pour chaque interface de $B_{ci,ifType}$, on a donc $if_1 = if_2$.
- Sinon, *(2)* est vraie. Par définition de *(2)*, tous les voisins de ci respectent *(1)* ou *(2)* avec la même interface if. Donc $if_1 = if_2$.

Dans les deux cas, $if_1 = if_2$. □

On peut alors définir le domaine :

Définition 4.2. Domaine d'une instance de composant pour un type d'interface

Soit un type d'interface $ifType$.

Soit un design d.

On note E_{ci} l'ensemble des instances de composant de d,

et E_{if} l'ensemble des interfaces sur toutes les instances de composants de d.

On définit la fonction suivante :

$$domaine_{ifType} : E_{ci} \rightarrow E_{if} \cup \{0\}$$

$$ci \mapsto \begin{cases} if \in E_{if} \text{ respectant } (1) \text{ ou } (2) \text{ si elle existe,} \\ 0 \text{ sinon.} \end{cases}$$

On dit que ci **appartient au domaine** $domaine_{ifType}(ci)$ pour le type $ifType$.

Cette définition correspond à l'idée que l'ont peut se faire d'un domaine d'horloge :

- Si un composant est connecté à une unique horloge, alors il appartient au domaine de cette horloge.
- S'il est connecté à plusieurs horloges différentes, alors il n'appartient à aucun domaine en particulier.
- S'il n'est connecté à aucune horloge mais que ces voisins sont tous connectés à la même horloge, alors il appartient au domaine de cette horloge.

Algorithme 4.1 : Fonction `domaine` (*instComp, ifType*)

```
1 E_domaines ← ensemble des instances d'interfaces de bus connectées à celles de type
  ifType sur instComp ;
2 si taille (E_domaines) = 1 alors
3 |  retourner E_domaines.premier();
4 sinon
5 |  retourner domaineSurVoisins (instComp, ifType) ;
```

L'algorithme 4.1 détaille la fonction *domaine*, utilisée pour évaluer le domaine d'un composant. Elle observe d'abord si le composant est déjà connecté à une interface du type

Algorithme 4.2 : Fonction `domaineSurVoisins` (*instComp, ifType*)

```
1 E_domaines ← ∅;
2 pour chaque Instance de composant voisin connectée à instComp faire
3 |  E_domaines ← E_domaines ∪ {domaine (voisin, ifType)};
4 si taille (E_domaines) = 1 alors
5 |  retourner E_domaines.premier();
6 sinon
7 |  retourner null ;
```

voulu. Normalement ce n'est pas le cas. Les voisins sont alors analysés avec la fonction *domaineSurVoisins*. Comme elle fait elle-même appel à *domaine*, le processus est récursif. Une liste d'instances de composants est maintenue parallèlement à son exécution pour éviter que les fonctions ne soient appelées infiniment sur les mêmes composants, sans analyser les autres voisins. Dans un souci de simplification, cette liste n'apparaît pas dans le détail des méthodes.

Maintenant que la manière d'évaluer le domaine des instances de composants est établie, la création de la méthode de connexion automatique peut se faire relativement simplement. Elle se trouve dans l'algorithme 4.3. Si le domaine de l'instance de composant est défini, alors on l'utilise pour la connexion. Dans le cas contraire, on connecte le composant d'après les modules existants, en en instanciant un nouveau si besoin.

L'utilisation des interfaces faite ici montre que leur intérêt n'est pas uniquement d'agréger des ports pour les connecter ensemble. Comme chaque interface d'un composant est associée à un identifiant IP-XACT VLNV *(Vendor, Library, Name, Version)*, il est possible d'utiliser cet identifiant comme moyen de connaître la fonction des ports. Dans cette section, cette possibilité est utile par exemple pour différencier un signal d'horloge d'un signal *reset*. Les ports seuls n'auraient pas permis de faire cette distinction : dans les deux cas il s'agit d'un port d'entrée *wire* de largeur 1 bit.

Algorithme 4.3 : Connexion automatique des signaux obligatoires

Entrées : Interface de bus sur une instance de composant instBusIF à connecter

```
1  début
2      IFàConnecter ← domaine (instBusIF.instComp (), instBusIF.typeIF ());
3      si IFàConnecter == null alors
4          E_sources ← instances de composant du design pouvant se connecter à
           instBusIF ;
5          si taille (E_sources) = 0 alors
6              E'_sources ← composants possédant l'interface recherchée dans la
               bibliothèque;
7              compChoisi ← choix de l'utilisateur parmi E'_sources;
8              ci ← nouvelle instance de compChoisi ;
9              IFàConnecter ← interface de ci connectable à instBusIF ;
10         sinon si taille (E_sources) = 1 alors
11             IFàConnecter ← E_sources.premier();
12         sinon
13             IFàConnecter ← choix de l'utilisateur parmi E_sources;
14     connecter instBusIF et IFàConnecter ;
15 fin
```

4.3 Importation de paramètres

Un outil assez proche de la connexion automatique des signaux requis est l'importation de valeurs de paramètres.

Ce besoin a été rencontré par exemple pour des modules ayant en paramètre la fréquence d'horloge. Cela permet d'augmenter la vitesse de la simulation en diminuant le nombre d'appels à l'ordonnanceur SystemC. Sans cela, les arguments des appels à la fonction *wait()* indiquent un nombre de cycles, ce qui implique une opération à chaque événement sur le signal d'horloge. En indiquant la période d'horloge, il est possible d'appeler *wait()* avec une durée comme argument. Le moteur de simulation peut alors évaluer directement à quel moment de simulation le thread doit être réactivé.

Notons p_i un paramètre sur un type de composant c_i devant importer sa valeur d'un

paramètre « original » p_o appartenant au composant c_o. Pour spécifier le besoin d'importation, la section XML décrivant p_i possède dans ses extensions les identifiants de c_o et p_o. Un exemple est présenté dans l'annexe I.2.1 page 115.

Lors de l'instanciation de c_i, l'outil commence par observer si le composant c_o requis a déjà une instance dans le design. Dans le cas négatif, une nouvelle instance est ajoutée. Notons à présent c_i^i et c_o^i les instances des composants c_i et c_o. Les paramètres p_i et p_o sur c_i et c_o sont notés p_i^i et p_o^i. Une fois la dépendance définie, p_i^i est mis automatiquement à jour quand p_o^i est modifié. L'implémentation est fondée sur le motif de conception de l'écouteur (ou observateur), présenté par exemple dans [24]. p_i^i s'enregistre comme observateur auprès de p_o^i. Lorsque la valeur de p_o^i est modifiée, les observateurs en sont notifiés. Ils peuvent alors effectuer leur mise à jour.

Lors de l'enregistrement du design, il est nécessaire de sauvegarder ces dépendances dans le fichier IP-XACT généré. Pour cela, pour chaque instance de composant, on ajoute si nécessaire une liste de correspondances entre ses paramètres dépendants et les instances de composants du design qui définissent la valeur (voir annexe I.2.2 p. 116).

4.4 Génération d'adaptateurs

Comme nous l'avons vu dans la section 1.1.4, les niveaux transactionnels doivent être conçus selon le principe d'orthogonalité des calculs et des communications. Cela aboutit nécessairement à un modèle en couches, représenté à la figure 1.4 p. 11. Ce modèle repose sur des adaptateurs qui sont insérés entre les éléments de traitement et les canaux. Ils n'apportent pas directement de fonctionnalité au système ; leur rôle est de rendre compatibles deux interfaces qui ne le sont pas. Il est donc indispensable d'offrir aux utilisateurs de l'environnement de conception un outil leur permettant de faire abstraction de ces adaptateurs. Autrement dit, ils doivent être en mesure de considérer qu'un

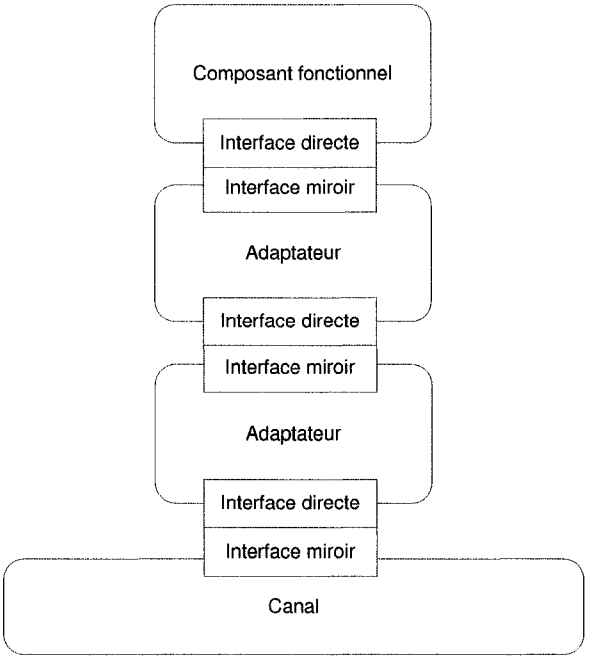


Figure 4.1: Exemple de chaîne de deux adaptateurs

module est directement connecté à un canal de communication, sans se préoccuper de l'implémentation de cette connexion.

Pour cela, j'ai développé une méthode permettant de construire une chaîne d'adaptateurs à partir des descriptions présentes dans la bibliothèque. Elle repose sur une classification des composants en trois catégories : les composants fonctionnels *(functional)*, les adaptateurs *(adapters)* et les canaux *(channel)*. Le type du composant est mentionnée dans sa section *vendorExtensions*, et le type par défaut est *functional*.

Il est requis que chaque adaptateur possède au moins deux types d'interface : direct et miroir. Comme représenté figure 4.1, les interfaces miroir sont orientées du côté du composant fonctionnel et les interfaces directes se connectent du côté du canal.

L'algorithme repose sur une représentation des possibilités de connexion sous forme de forêt dans laquelle les nœuds sont des adaptateurs. L'arbre est construit au fur et à mesure de l'exécution de l'algorithme, et il repose sur les règles suivantes :

1. Les nœuds racine peuvent se connecter au composant fonctionnel.
2. Un nœud F est fils du nœud P si et seulement si P possède une interface directe qui peut se connecter à une interface miroir de F.

Le but est donc de trouver un nœud-adaptateur qui puisse se connecter au canal visé. Le chemin de la racine à ce nœud forme alors une liste d'adaptateurs dont le premier se connecte au composant fonctionnel (règle 1), le dernier au canal, et chaque élément peut se connecter à ses deux voisins (règle 2). Cette liste constitue donc une solution.

Cette représentation sous forme de forêt se justifie par la nature du problème : il s'agit d'établir une liste (d'adaptateurs) dans laquelle les possibilités de chaque élément dépendent du choix de l'élément précédent (les adaptateurs doivent pouvoir se connecter). Le premier élément est choisi parmi un ensemble de possibilités : ce sont les racines des arbres. Ensuite, le choix de chaque élément détermine l'ensemble des possibilités des éléments suivants ; ce sont ses fils.

Algorithme 4.4 : Principe de la recherche d'adaptateurs

```
   Données : bibliothèque d'adaptateurs, maxAdapt, composants compDépart et
             compArrivée
   Résultat : chaîne d'adaptateurs à utiliser, ou null si elle n'est pas trouvée
 1 début
 2     nœuds racine ← adaptateurs connectables à compDépart ;
 3     nbAdapt ← 1;
 4     tant que nbAdapt ⩽ maxAdapt faire
 5         pour chaque feuille ∈ feuilles de la forêt faire
 6             si feuille peut être connecté à compArrivée alors
 7                 retourner chemin de racine à feuille ;
 8         pour chaque feuille ∈ feuilles de la forêt faire
 9             feuille.enfants ← adaptateurs connectables à feuille;
10         nbAdapt++;
11     retourner null
12 fin
```

L'algorithme 4.4 présente une version simplifiée de la méthode. Il s'agit avant tout d'une

recherche arborescente. Sa principale particularité est qu'elle traite une forêt dont les nœuds sont construits au fur et à mesure de la recherche. Les nœuds racines de la forêt sont initialisées à la ligne 2, puis l'arbre est construit niveau par niveau. Pour chaque niveau, on commence par chercher s'il ne contient pas une solution (ligne 6). Dans le cas négatif, on construit le niveau suivant dans la boucle à la ligne 5.

Dans son implémentation réelle, cet algorithme a été amélioré de manière à réduire son temps d'exécution. Cela en consiste en l'élagage *(pruning)* d'une partie de la forêt lorsqu'on sait qu'une branche ne pourra pas aboutir à une solution. Cela survient à la ligne 9 : s'il n'y a pas d'adaptateurs connectables à *feuille*, il est possible de la supprimer. Si le père de *feuille* n'avait pas d'autre enfant, il peut être aussi supprimé, et ainsi de suite. Cela accélère l'exécution de l'algorithme : l'évaluation des connexions possibles (ligne 6) et la recherche d'adaptateurs (ligne 9) sont économisées pour les éléments supprimés.

Cet algorithme est totalement générique : lui-même ainsi que les méthodes qu'il invoque ne sont pas dépendantes de certains types particuliers d'adaptateurs. Toute l'information nécessaire se trouve dans la description des interfaces qui provient des fichiers IP-XACT décrivant les composants. De plus, il permet de fixer arbitrairement le nombre maximal d'adaptateurs (*maxAdapt*).

Deux adaptateurs sont en général suffisants. Ce cas survient dans Space pour connecter un composant utilisateur à un modèle de bus comme OPB. Le premier adaptateur, dit *module adapter*, se connecte sur le composant utilisateur. Il effectue la conversion entre les requêtes sur l'interface de communication simple de Space et une interface de bus commune. Le second adaptateur permet de relier cette interface commune au modèle de bus. Un adaptateur supplémentaire peut être nécessaire dans le cas de simulations mixtes RTL/TLM [26]. Aucune situation où il est nécessaire d'utiliser plus de trois adaptateurs n'a été rencontrée.

Tel qu'il est présenté, l'algorithme 4.4 est destiné à des connexions de type « composant-

à-composant ». On remarque qu'en modifiant les tests de connexion (lignes 6 et 9), il est possible de l'utiliser pour des connexions « composant-à-interface » ou « interface-à-interface ».

4.5 Automatisation du raffinement

4.5.1 Introduction

Le flot de conception se fait selon une approche *top-down* (de haut en bas). Cela signifie que les premières étapes modélisent le système à un haut niveau d'abstraction, puis les modèles sont de plus en plus précis. Le passage d'un modèle donné à un autre plus précis se nomme *raffinement*.

Chaque étape de raffinement impose généralement des choix architecturaux, en particulier concernant les communications et le partitionnement logiciel-matériel. Ces choix sont laissés à l'utilisateur. Il est néanmoins possible de lui proposer un modèle utilisable comme point de départ pour la réalisation du nouveau modèle. Le principe de base est de recréer une architecture semblable à celle de départ, mais utilisant des composants à plus bas niveau.

Les étapes sont les suivantes :

1. appel des générateurs *preTransform* (section 4.5.5.1),
2. sélection des composants équivalents (section 4.5.2),
3. liste des possibilités de designs et choix de l'utilisateur (section 4.5.3),
4. création du nouveau design à partir de ce choix et des propriétés du design original (section 4.5.4),
5. appel des générateurs *postTransform* (section 4.5.5.2),
6. appel des générateurs *postInstanciation*.

4.5.2 Spécification des équivalences de composants

Le principe de base de la création du modèle raffiné est le remplacement de chaque instance de composant à haut niveau (adaptateurs exceptés) par un composant équivalent, et correspondant au niveau visé. Il faut donc définir un moyen d'exprimer l'équivalence de composants. Pour cela, la notion d'héritage, telle qu'utilisée dans les langages de programmation orientés objets a été utilisée. Elle offre la possibilité de grouper facilement les composants : ceux qui héritent d'un même composant père sont considérés comme appartenant au même groupe. Il est également possible de définir aisément des sous-groupes en faisant dériver des composants d'un composant possédant lui-même un père (cf. fig. 4.2.) Tout comme dans les langages orientés objet comme Java ou C++, il est possible de définir des composants virtuels (ou abstraits). Ils ne sont pas instanciables ; la création d'un *componentInstance* à partir d'un tel composant engendrera une erreur. Leur rôle est uniquement de grouper les composants qui en dérivent, et d'en définir les caractéristiques communes. Celles-ci sont de deux types :

1. Les paramètres : pour chaque paramètre du composant père, il existe un paramètre possédant le même identifiant sur le fils.
2. Les interfaces : chaque interface du père est associée à une interface équivalente sur le fils. L'association se situe dans les extensions de l'interface fille : le nom de l'interface équivalente sur le composant père y est mentionné.

Il apparaît que dans un cas, l'équivalence est définie uniquement par le nom, dans l'autre cas il est nécessaire d'ajouter spécifiquement une extension. Cela est dû à la différence de signification des identifiants. En effet, pour les paramètres, l'identifiant est une chaîne de caractères distincte du nom. Elle est destinée uniquement au fonctionnement interne. On peut donc l'utiliser librement, sans que cela ait de conséquence sur les propriétés visibles du composant. En revanche, le nom de l'interface peut appartenir aux caractéristiques extérieures du composant. C'est notamment le cas pour les fichiers MHS *(Microprocessor Hardware Specification)* des projets de Xilinx Platform Studio (voir annexe III). Les

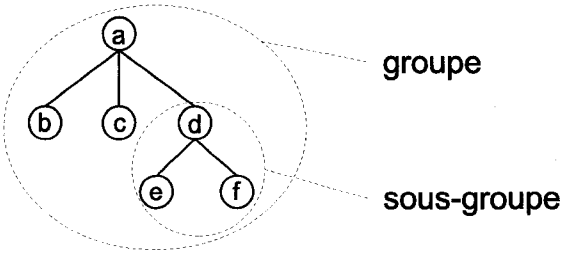


Figure 4.2: Définition de groupes et sous-groupes par relations d'héritage

connexions peuvent y être spécifiées de manière « interface-à-interface ». Le nom des interfaces est fixé dans la définition du composant, et elles ne peuvent être modifiées.

Les contraintes sur les propriétés des modules fils ne sont pas absolues : une interface ou un paramètre manquant n'empêchera pas l'outil de raffinement de fonctionner. Les paramètres manquants seront simplement ignorés lors de l'importation des valeurs, et les connexions risquent de se faire sur les mauvaises interfaces du composant. Il est toujours possible pour l'utilisateur de rectifier ces erreurs après l'exécution de l'outil.

En outre, il est parfois utile de grouper des composants par une relation d'héritage, sans qu'ils ne soient considérés comme équivalents. En particulier, les composants utilisateur (cf. sec. 4.6) héritent de leur composant de base. Cela permet à un processeur d'indiquer qu'il supporte tous les composants utilisateur créés à partir de ce composant de base (cf. sec. 4.7). Cependant, les composants utilisateur ne sont pas équivalents entre eux. Une balise *isEquiv* peut être ajoutée à la définition d'une relation d'héritage pour indiquer si elle définit une équivalence. Sa valeur par défaut est « vrai ». Dans les cas où elle est fausse, la méthode analysant les équivalences ne tiendra pas compte de la relation.

Algorithme 4.5 : Principe de la méthode de raffinement

```
   Données : design original d_orig
   Résultat : solutions de designs raffinés possibles
 1 début
 2 |  solutions ← ∅;
 3 |  conn ← trouvConnAbstraites (d_orig);
 4 |  pour chaque composant non-adaptateur c_orig de d_orig faire
 5 |  |  C_equiv[c_orig] ← ensemble des composant équivalents à c_orig convenant pour le
   |  |  niveau d'abstraction visé;
 6 |  pour chaque configuration possible config utilisant les composants de C_equiv
   |  faire
 7 |  |  si toutes les connexions de conn peuvent être réalisées alors
 8 |  |  |  solutions.ajouter (config);
 9 |  retourner solutions;
10 fin
```

4.5.3 Exploration des choix de design

L'algorithme 4.5 présente la méthode de l'automatisation du raffinement. La ligne 3 établit les connexions abstraites[1] du design original d_{orig}. Ce type de connexion fait abstraction des adaptateurs. Autrement dit, un composant fonctionnel est considéré comme étant directement connecté à un canal, même si dans la réalité, des adaptateurs font le lien entre les deux. Ainsi, elles se présentent sous forme de paires d'interfaces, qui ne sont pas forcément compatibles. Elles ne figurent pas telles quelles dans le design, c'est la méthode *trouvConnAbstraites()* qui les établit d'après ses interconnexions. Son implémentation se trouve dans l'algorithme 4.6.

Puis l'algorithme de raffinement établit (ligne 5) la liste des composants équivalents à ceux présents dans d_{orig}. Ils sont trouvés d'après les relations d'héritage entre modules comme expliqué dans la section précédente. Seuls les composants applicables pour le niveau visé sont sélectionnés. Les niveaux pour lesquels un composant peut être utilisé sont déclarés

[1] à ne pas confondre avec les liens abstraits des connexions point à point définies section 4.8.2

Algorithme 4.6 : Fonction `trouvConnAbstraites` (*componentInstance*)

Données : instance de composant c^i,
Les ensembles d'instances de composants E_a^i (adaptateurs) et E_v^i (voisins) sont initialisés lors du premier appel : $E_a^i \leftarrow \emptyset$; et $E_v^i \leftarrow \emptyset$;

```
1 début
2     pour chaque voisin v^i connecté à c^i faire
3         si v^i est une instance d'adaptateur et v^i ∉ E_a^i alors
4             ajouter v^i à E_a^i;
5             trouvConnAbstraites (v^i);
6         sinon
7             ajouter v^i à E_v^i;
8 fin
```

dans ses extensions. Par défaut, si un composant ne mentionne pas son ou ses niveaux, il sera considéré comme utilisable pour tout niveau. Si un composant du design original c_{orig} est autorisé pour le niveau visé, alors il sera simplement recopié dans le nouveau design : dans ce cas $C_{equiv}[c_{orig}] = \{c_{orig}\}$.

Enfin, la méthode explore les possibilités de design (ligne 6). Chaque instance de composant de d_{orig} est remplacée par des instances des composants de $C_{equiv}[c_{orig}]$. Si toutes les connexions peuvent être réalisées, la configuration est enregistrée. À la fin, l'utilisateur obtient une série de configurations possibles ; il peut alors faire son choix.

Pour accélérer l'exécution de cet outil, plusieurs améliorations lui ont été apportées. D'abord, à la ligne 6, lorsqu'on évalue si la réalisation d'une interface abstraite est possible, le résultat est enregistré pour qu'il puisse être réutilisé. On évite ainsi d'exécuter la méthode de recherche d'adaptateurs pour chaque connexion abstraite dans chaque design à tester.

Deuxièmement, des classes spéciales, plus légères, sont utilisées pour représenter les configurations testées. Tous les attributs des objets utilisés habituellement ne sont en effet pas indispensables ici. Par exemple, il n'est pas utile d'enregistrer d'information sur les connexions : on sait qu'elles réalisent celles de conn. De plus, il n'est pas nécessaire de

créer une collection pour les paramètres des instances de composant. Le « vrai » design sera créé par la suite, lorsque l'utilisateur aura fait son choix parmi les solutions.

La troisième accélération de la méthode se situe dans la manière d'explorer les possibilités de designs. Chaque instance de composant c_o^i du design original est associée à une liste de composants $\mathcal{L}_{equiv}[c_o^i]$. Celle-ci contient les possibilités de composants pour remplacer c_o^i dans le nouveau design. Le principe de l'accélération est de mettre à jour les voisins lors du choix d'un composant dans $\mathcal{L}_{equiv}$. La fonction utilisée se trouve dans l'algorithme 4.7. Elle est récursive : si elle modifie un composant, elle s'appelle elle-même pour notifier ses voisins (ligne 7). Grâce à cette méthode, lors du test des connexions ligne 7 de l'algorithme 4.5, on évalue en premier celles qui sont impossibles.

Algorithme 4.7 : Fonction `répercuterSurVoisins` (*componentInstance*)

Données : instance de composant c_o^i dont la liste d'équivalences $\mathcal{L}_{equiv}[c_o^i]$ a été modifiée

```
1 début
2     pour chaque voisin v_o^i de c_o^i faire
3         pour chaque composant équivalent v_e ∈ L_equiv[v_o^i] faire
4             si v_e ne peut pas se connecter à tous les composants équivalents de
              L_equiv[c_o^i] alors
5                 enlever v_e de L_equiv[v_o^i];
6         si L_equiv[v_o^i] a été modifiée alors
7             répercuterSurVoisins (v_o^i);
8 fin
```

4.5.4 Création du design raffiné

L'exécution de l'algorithme 4.5 renvoie une liste de configurations possibles. Chaque configuration contient simplement les équivalences entres les instances de composant du design original (adaptateurs exceptés) et des composants utilisables dans le niveau visé. L'utilisateur fait alors son choix. Le design correspondant à ce choix de l'utilisateur est alors créé. Les instances de composant sont ajoutées, et les valeurs des paramètres importées.

Pour l'importation de paramètre, on commence par déterminer l'ancêtre commun c_a (dans les relations d'héritage) entre le composant original c_o et le nouveau c_n. Il est défini comme étant l'unique composant c_a tel que :

1. c_o et c_n dérivent de c_a,
2. et il n'existe pas de composant c'_a dérivant de c_a tel que c_o et c_n dérivent de c'_a.

On remarque que si c_n est un descendant de c_o, alors $c_a = c_o$, et inversement.

Dans l'exemple de diagramme d'héritage figure 4.2 page 66, en prenant $c_o = e$ et $c_n = f$, l'ancêtre commun c_a est d. Le nœud a respecte la règle 1, mais pas la 2, car d dérive de a et respecte la règle 1.

Une fois que l'ancêtre commun a été déterminé, il est possible d'importer ses paramètres. La méthode est simple, son détail se trouve dans l'algorithme 4.8. On constate que la ligne 8 vérifie l'existence du paramètre à importer sur le composant original et sur le nouveau. S'il manque sur un des deux composants, il est simplement ignoré.

Algorithme 4.8 : Importation de paramètres lors du raffinement

```
1  début
2      pour chaque instance de composant i_o du design original faire
3          i_n ← instance de composant remplaçant i_o dans le nouveau design;
4          c_a ← ancêtre commun de des types de composants de i_o et i_n;
5          pour chaque paramètre p_a de c_a faire
6              p_o^i ← paramètre correspondant à p_a sur i_a;
7              p_n^i ← paramètre correspondant à p_a sur i_n;
8              si p_o^i et p_n^i existent alors
9                  valeur(p_n^i) ← valeur(p_o^i);
10 fin
```

Pour recréer les connexions, le principe est relativement proche ; il se trouve dans l'algorithme 4.9. L'étape de la ligne 5, qui recherche les interfaces équivalentes, n'a pas été détaillée. Son fonctionnement est identique à celui des lignes 4 à 7 de l'algorithme 4.8, en remplaçant « paramètre » par « interface ».

Les méthodes *connecter*, appelées aux lignes 7, 9, 11 et 13, réalisent les connexions en insérant si besoin des adaptateurs. Selon les cas, il s'agit de connexions « interface-à-interface » (ligne 7), « interface-à-composant » (lignes 9 et 11), ou « composant-à-composant » (ligne 13). Dans ces deux derniers cas, l'outil pourra utiliser n'importe quelle interface des composants qui convient pour réaliser la connexion.

Algorithme 4.9 : Recréation des interconnexions lors du raffinement

```
1  début
2      pour chaque connexion abstraite {if_o^1, if_o^2} du design original faire
3          i_o^k ← instance de composant de if_o^k, pour k ∈ {1,2};
4          i_n^k ← instance équivalente à i_o^k dans le nouveau design, pour k ∈ {1,2};
5          if_n^k ← interface de i_n^k équivalente à instance équivalente à if_o^k, pour
           k ∈ {1,2};
6          si if_n^1 et if_n^2 existent alors
7              connecter (if_n^1, if_n^2);
8          sinon si if_n^1 existe et if_n^2 n'existe pas alors
9              connecter (if_n^1, i_n^2);
10         sinon si if_n^2 existe et if_n^1 n'existe pas alors
11             connecter (if_n^2, i_n^1);
12         sinon
13             connecter (i_n^1, i_n^2);
14 fin
```

Les outils habituels pour la création d'instances de composants sont également appelés : connexion automatique pour les signaux obligatoires, importation de paramètres, générateurs *postTransform*.

4.5.5 Support des cas particuliers : utilisation des générateurs

4.5.5.1 Choix du successeur : générateurs *preTransform*

Il peut arriver que le choix du successeur de certains composants doive obéir à des règles particulières. Pour cela, un type de générateur a été créé : *preTransform*. Comme son nom

l'indique, les générateurs de ce type sont appelés juste avant l'exécution du raffinement du design. La signature de leur méthode principale est la suivante :

```
public Component run(ComponentInstance ci, AbstractProject
    proj) throws Exception;
```

Le premier argument est l'instance de composant dont on veut sélectionner le type du successeur. Il possède une référence du design auquel il appartient. On peut donc accéder à toutes les propriétés de celui-ci, en particulier ses instances de composants et ses connexions.
Le second argument est le projet courant de la plateforme. Il regroupe un ensemble de designs, ainsi que des composants utilisateurs éventuellement instanciés dans ces designs. Il est aussi associé à un répertoire du système de fichiers. Donner le projet en argument permet donc au générateur de créer de nouveaux fichiers dans son répertoire, et d'y ajouter des composants utilisateur.
L'objet retourné par la méthode est le type de composant à utiliser pour le remplaçant de *ci*.

Ce type de transformateur est utilisé pour faire migrer les composants utilisateur dans la génération de plateforme FPGA (section 5.4.1 p. 91).

4.5.5.2 Modification du design après transformation : générateurs *postTransform*

Il peut arriver que la création du nouveau design en fonction du design original tel que mentionné dans la section 4.5.4 soit incomplète. En effet, certains composants au comportement particulier peuvent nécessiter de modifier le nouveau design après les opérations automatiques de la plateforme. Un autre type de générateur a été défini pour ces cas : *postTransform*. Leur méthode principale est définie comme ceci :

```
public void run(ComponentInstance originalCompInst,
    ComponentInstance newCompInst) throws Exception;
```

Ses deux arguments sont l'instance de composant du design original et l'instance équivalente dans le nouveau design. De même que pour les générateurs *preTransform*, les objets *ComponentInstance* passés en argument donnent accès aux designs qui les contiennent. Ces générateurs sont donc capables d'analyser le design original et de modifier le nouveau en conséquence. Leurs capacités sont donc plus grandes que celles des générateurs normaux ou *postInstantiation* qui n'ont accès qu'aux composants du nouveau design.

À titre d'exemple, dans la génération de plateforme FPGA, un générateur de ce type est utilisé la gestion des adresses de composants Space. En effet, les composants transactionnels de la bibliothèque Space possèdent des identifiants uniques. À bas niveau, ces identifiants sont transformés en adresses. Cette opération se fait grâce à un générateur *postTransform* qui lit les identifiants Space dans le niveau supérieur, calcule les adresses correspondantes, et paramètre les composants bas niveau en conséquence.

4.5.6 Composants hiérarchiques

Selon IP-XACT, les composants hiérarchiques sont des composants comme les autres, excepté qu'ils possèdent la référence d'un design décrivant leur intérieur (section 2.3.3 p. 32). Cette représentation fait une forte séparation entre d'une part le design principal, dans lequel l'instance de composant hiérarchique (ou super-composant) est vue comme une boîte noire, et d'autre part son design intérieur. À cause de certains usages particuliers que fait la plateforme avec les composants hiérarchiques, les limites que posent cette séparation doivent être dépassées. Ces usages sont les suivants :

dépendances de composants Certains composants en requièrent d'autres pour fonctionner correctement. Pour cela, on utilise un composant hiérarchique qui regroupe

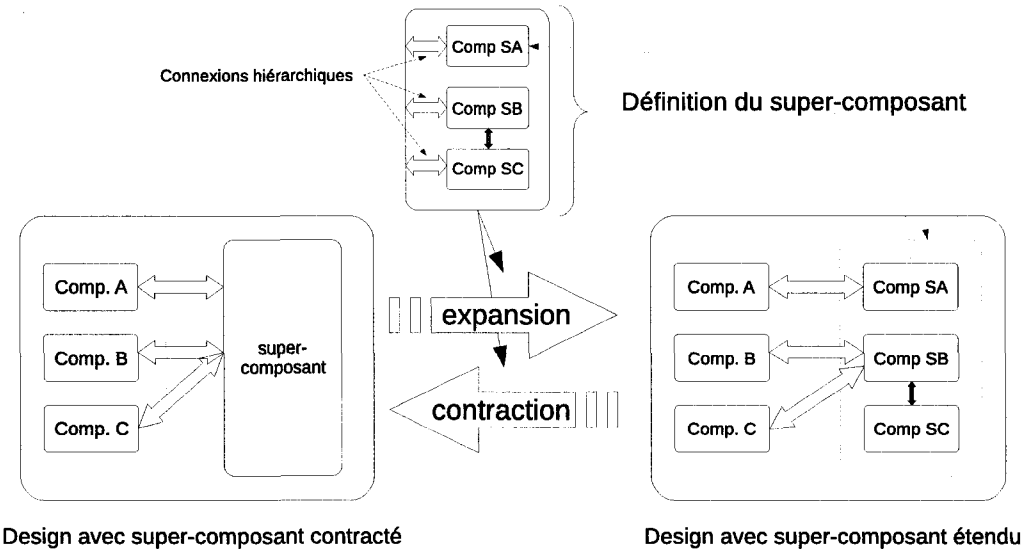


Figure 4.3: Expansion et contraction de composants hiérarchiques

le composant principal et ses dépendances. Une fois le composant hiérarchique instancié, il est étendu pour faire apparaître ses sous-modules. C'est notamment le cas pour les processeurs, qui requièrent un *timer*, un contrôleur d'interruptions, etc.

éléments d'une carte Dans la génération de prototype sur carte, un composant hiérarchique représente la carte avec les périphériques qui y sont implantés (cf. section 5.2.2). Ce composant est d'abord étendu, ce qui fait apparaître les sous-composants. Au moment de la génération des fichiers, il doit apparaître comme un tout, sur lequel les modules du FPGA sont connectés. Il est donc contracté.

Les deux méthodes *expand()* (expansion) et *contract()* (contraction) sont donc ajoutées sur les instances de composant hiérarchiques. Leurs actions sont schématisées dans la figure 4.3. Lors de l'expansion, les interconnexions avec le super-composant sont remplacées par des interconnexions avec les sous-composants, en tenant compte de la définition des connexions hiérarchiques. La contraction fait l'opération inverse : toutes les interconnexions impliquant des sous-composants sont remplacées par les connexions équivalentes avec le super-composant.

La contraction implique d'être en mesure de faire de faire le lien entre les instances dans le design principal et l'intérieur de l'instance du super-composant. Pour cela, chaque sous-composant possède deux références :

1. l'instance de composant dont il fait partie,
2. l'instance de sous-composant correspondante, telle qu'elle apparaît dans la définition du design interne du super-composant.

Il est alors possible de réaliser la contraction. Grâce à la première référence, les sous-composants appartenant à la super-instance à contracter sont identifiés. La seconde référence donne leur rôle dans le super-composant, et les connexions de remplacement peuvent être créées.

Dans le cas où le design est enregistré en configuration étendue, ces informations doivent être sauvegardées pour rester capable d'opérer la contraction des super-composants lorsqu'il sera rechargé. Un exemple d'extensions décrivant une telle situation est présentée dans l'annexe I.5 page 122.

4.6 Composants utilisateur

Pour concevoir un système, l'utilisateur a besoin de créer ses propres composants. Dans la plupart des cas, ils correspondent aux éléments de traitement définissant la fonctionnalité de l'application. Éventuellement, ils peuvent aussi être des composants de l'architecture, comme des modules d'entrée-sortie. Une méthode permet à l'utilisateur de créer facilement de nouveaux composants. Elle se base sur des modèles, qui sont modifiés après avoir été copiés.

La première étape est de définir les composants servant de modèle. Bien que cela ne soit pas obligatoire, ils sont généralement virtuels, c'est-à-dire non instanciables. Afin de les identifier, ils possèdent une balise *userComponentBase*, contenant une variable booléenne.

Par défaut, sa valeur est « faux ». Un générateur de type *userComponent* est aussi associé à ces composants. Son rôle principal est de générer des squelettes de fichiers source. Il est aussi capable de modifier le document IP-XACT du nouveau composant.

Le processus de création d'un composant utilisateur se déroule comme suit.

1. L'utilisateur choisit le composant de base.
2. Le document XML décrivant le composant de base est copié.
3. Le nom du nouveau composant est mis à jour dans son identifiant VLNV
4. Les extensions sont mises à jour. En particulier, il est indiqué qu'il hérite du composant de base, sans en être équivalent. En revanche, il ne peut pas servir lui-même de base comme composant utilisateur. La balise *userComponentBase* est donc ôtée.
5. Le générateur de type *userComponent* du composant de base est exécuté.

Après cela, le composant est enregistré dans le projet. Il ne fait donc pas partie de la bibliothèque principale. Les fichiers source générés peuvent ensuite être modifiés par l'utilisateur pour qu'ils décrivent l'application.

4.7 Exploration du partitionnement logiciel-matériel

Le partitionnement est le principal défi de la conception d'un système sur puce. Cette étape consiste à assigner des composants aux éléments fonctionnels. En particulier, le partitionnement logiciel-matériel associe ces éléments soit à une implémentation matérielle, soit à une forme logicielle. Le principal avantage du matériel est qu'il permet d'apporter du parallélisme ; dans ce cas le temps d'exécution est plus faible qu'une implémentation logicielle. En revanche, le logiciel est avantageux en termes de flexibilité et de coût de développement.

En raison de l'importance du partitionnement dans le flot de conception (cf. sec. 1.1.3

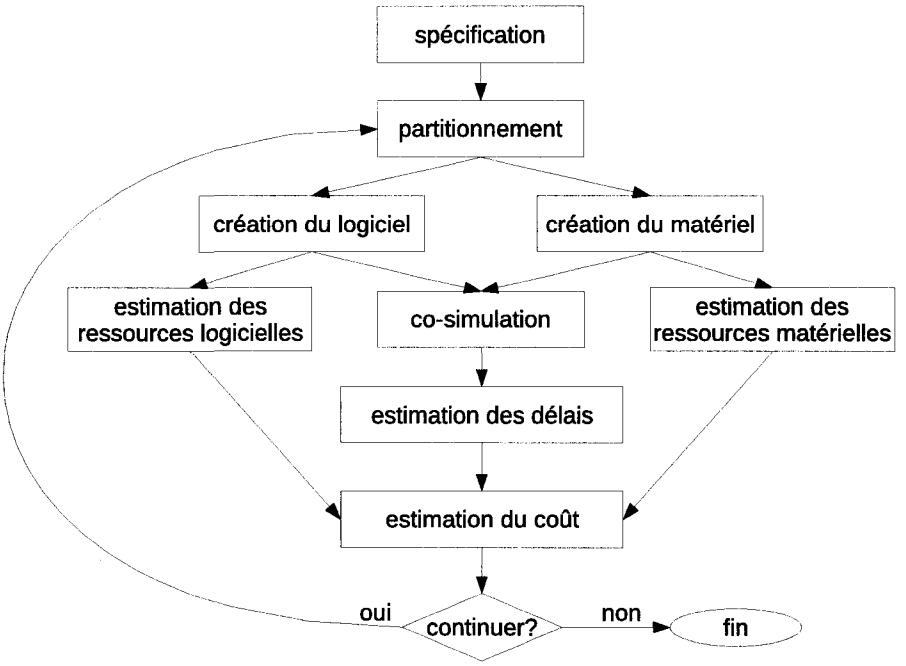


Figure 4.4: Principe général des méthodes de partitionnement

p. 8), de nombreuses méthodes ont été proposées. Les plus importantes sont regroupées et analysées dans [27]. Le principe global du partitionnement est présenté à la figure 4.4. On constate que quelle que soit la méthode choisie (manuelle ou automatique), il faut être en mesure d'estimer le coût des éléments, qu'ils soient exécutés en logiciel ou non. Le coût est généralement calculé à partir du temps d'exécution et des ressources matérielles utilisées par le système. D'autres facteurs peuvent parfois s'ajouter, en particulier la consommation de puissance, mais il ne seront pas considérés ici.

Pour évaluer le temps d'exécution d'un module implémenté en matériel, la méthode la plus simple consiste à ajouter des annotation temporelles (*wait()* en SystemC) à son code. Bien entendu, les résultats obtenus dans ce cas ne sont que des estimations et leur précision dépend de celle des annotations. Cependant, la simplicité de cette méthode et sa rapidité d'exécution la rendent intéressante. Pour obtenir un modèle précis au cycle près, il est possible de synthétiser un module fonctionnel vers le RTL avec des outils comme Forte Cynthecizer [28]. En outre, la synthèse permet d'estimer les ressources matérielles requises

par le module.

Pour estimer le logiciel, la tâche est plus facile, car le modèle transactionnel d'un composant est proche de son implémentation logicielle. Les ressources matérielles utilisées par un processeur sont connues à l'avance, il n'y a donc pas d'opération particulière à effectuer pour les évaluer. Pour estimer le temps d'exécution, il faut être en mesure d'exécuter le composant sur un modèle de processeur. Une méthode possible [14, 29] consiste à implémenter les interfaces de communication des composants sous forme logicielle. Ainsi, il est très aisé de déplacer un composant d'un modèle logiciel à un modèle matériel. Il suffit d'utiliser l'implémentation des interfaces correspondant au partitionnement choisi.

La plateforme supporte ce type de fonctionnalité, en utilisant les extensions IP-XACT. Le type de composant *processor* est défini pour cela. C'est un sous-type de composant fonctionnel. En plus de cette mention, le fichier *component* de chaque processeur contient une liste de systèmes d'exploitation. Ce sont les systèmes d'exploitation pour lesquels le générateur du processeur est capable de créer un exécutable. Pour chacun d'entre eux, les types de composants qui peuvent être implémentés en logiciel sont listés. Les modules dérivés[2] seront également acceptés par le générateur. Un extrait des extensions d'un processeur (le MicroBlaze) est proposé dans l'annexe I.3.1 p. 117.

Toutes les informations sont maintenant disponibles pour créer un design avec des parties logicielles. Considérons un design contenant au moins une instance de processeur. Lorsque l'utilisateur choisit qu'une instance de composant est logicielle, elle est d'abord supprimée du design. Pour cela, on l'enlève de la collection *ComponentInstanceSet* du design, et ses connexions sont supprimées du *ConnectionSet* (à l'exception des liens directs, voir section 4.8.2). Puis il est ajouté à une collection interne à l'instance de processeur, qui regroupe tous les modules applicatifs qu'il exécute. Ainsi, l'ensemble *ComponentInstanceSet* ne contient

[2]d'après la définition de l'héritage de la section 4.5.2

que les instances de modules matérielles du design, et pour chaque instance de processeur, la plateforme est en mesure d'énumérer toutes les instances de composants qu'elle exécute.

Lors de la génération de la simulation, seules les instances matérielles sont considérées par la plateforme. En effet, la génération du logiciel dépend évidemment du modèle de processeur utilisé. La généricité de la plateforme impose de placer cette fonctionnalité à l'extérieur : les générateurs seront donc utilisés.

Ces générateurs sont exécutés lors de la dernière étape avec la création des fichiers de simulation. Leur type est donc *normal*. Ils interrogent la plateforme pour connaître la liste des instances de composants. À partir de ces informations, ils peuvent créer les fichiers principaux du programme, ainsi que les scripts de compilation.

L'enregistrement dans le fichier *design* suit ce paradigme : la section *componentInstances* ne contient que les instances de composant matérielles, et les instances logicielles figurent dans les extensions du processeur qui les exécute. Un exemple de déclaration d'une instance de processeur se trouve Annexe I.3.2 page 118.

De cette manière, il est possible d'instancier plusieurs processeurs, chacun d'entre eux exécutant ses propres composants d'application. Cette structure est donc conçue pour pouvoir s'appliquer aux systèmes sur puce multiprocesseurs (MPSoCs), qu'ils soient homogènes ou hétérogènes. Sa principale limitation est qu'elle vise les architectures dans lesquelles chaque processeur possède sa propre application, exécutée par son propre système d'exploitation. Elle ne s'applique donc pas en l'état aux systèmes d'exploitation distribués comme QNX [30], dans lesquels il est possible d'exécuter une même instance du système d'exploitation sur plusieurs processeurs.

4.8 Liens directs

Les liens directs sont des canaux unidirectionnels dédiés aux communications entre deux modules. Ils permettent de diminuer les temps de latence pour ces communications, tout en désengorgeant les bus de communication. Cela est particulièrement important pour les applications de type « flot de données ». Dans de telles applications les bus peuvent constituer un goulet d'étranglement, en raison du fort trafic qui y transite, avec éventuellement des demandes simultanées d'accès.

Cette section propose une manière générique de gérer ces composants, que ce soit pour implémenter des liens purement matériels, ou impliquant des éléments logiciels. La conception de cet outil a été motivée par les travaux de maîtrise de Sylvain Goyette [31], qui étudient leur intérêt et proposent des implémentations. Celles-ci concernent à la fois les liens matériel-matériel, matériel-logiciel ou logiciel-logiciel, et sont basés sur la plateforme Space et les processeurs MicroBlaze et PowerPC.

L'exécution des méthodes présentées dans cette section a lieu lorsque l'utilisateur demande une connexion « composant fonctionnel à composant fonctionnel », ceux-ci étant possiblement implémentées sous forme logicielle. La génération d'adaptateurs simples (section 4.4), elle, est exécutée lors d'une connexion « composant fonctionnel à canal ».

Pour les cas de connexions « canal à canal », un outil d'insertion de pont à été conçu. Il cherche à connecter les deux canaux en utilisant un composant dont le type est *bridge*. L'algorithme étant très simple (il n'utilise pas d'adaptateurs), il ne sera pas détaillé.

La section 4.8.1 aborde les liens purement matériels, tandis que la section 4.8.2 traite des connexions impliquant au moins un module logiciel.

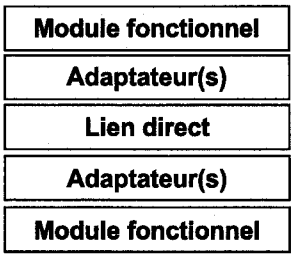


Figure 4.5: Structure générée pour réaliser des liens directs matériels

Figure 4.6: Classes utilisées pour la génération de liens directs

4.8.1 Liens matériel-matériel

Pour pouvoir fonctionner, la génération automatique de liens directs matériel-matériel a besoin de distinguer les liens directs des autres composants. Pour cela, un nouveau type de composant nommé *directLink* est défini. Il s'agit d'un sous-type de *channel*, les liens directs sont donc aussi considérés comme des canaux.

Comme pour tout autre canal, il peut être nécessaire d'insérer des adaptateurs entre les liens directs et les composants qu'ils relient. Le résultat que l'on souhaite générer est donc tel que présenté dans la figure 4.5. Pour chacun des deux groupes d'adaptateurs, leur nombre est compris entre 0 et *maxAdapt*, où *maxAdapt* est un nombre fixé par l'utilisateur.

L'algorithme 4.10 présente le principe de la recherche de liens directs. Il utilise des structures de recherche, instanciées à la ligne 5. Elles sont représentées à la figure 4.6. Chacune d'entre elles est associée à un type de lien direct. Leur but est de rechercher des solutions pour connecter les deux composants fonctionnels à ce canal. Elles utilisent pour cela deux objets exécutant la méthode de recherche d'adaptateurs de la section 4.4, un pour chaque composant fonctionnel à relier au canal. Leur particularité est qu'elles possèdent

Algorithme 4.10 : Principe de la génération de liens directs

Données : bibliothèque de liens directs, maxAdapt, composants compDépart et compArrivée
Résultat : structures de recherches de liens directs résolues

```
1  début
2      nbAdapt ← 0;
3      solutions ← ∅;
4      pour chaque lien direct dl de la bibliothèque faire
5          F_dl ← nouvelle structure de recherche pour dl ;
6      tant que solutions = ∅ & nbAdapt ⩽ maxAdapt faire
7          pour chaque lien direct dl de la bibliothèque faire
8              F_dl.rechercherNouveauNiveau ();
9              si F_dl.estRésolu () alors
10                 solutions.ajouter (F_dl) ;
11         nbAdapt ++;
12     retourner solutions;
13 fin
```

une méthode *rechercherNouveauNiveau()*, qui effectue une seule itération de la boucle principale de l'algorithme 4.4 (ligne 4). Le premier appel de cette fonction cherchera à réaliser la connexion sans adaptateur, avec un adaptateur pour le second appel, et ainsi de suite. Ainsi il est possible de rechercher sur les différentes possibilités de canal en parallèle, c'est-à-dire en augmentant la longueur des chaînes d'adaptateurs d'un seul niveau à la fois. Une structure de recherche est considérée comme résolue (ligne 5 de l'algorithme 4.10) lorsqu'une solution connectant les deux composants fonctionnels au lien direct a été trouvée.

Comme le nombre d'adaptateurs utilisés augmente progressivement, les solutions trouvées sont minimales. Dans le cas où plusieurs solutions sont trouvées, elles sont toutes retournées (ligne 12), et l'utilisateur pourra choisir celle qui lui convient le mieux.

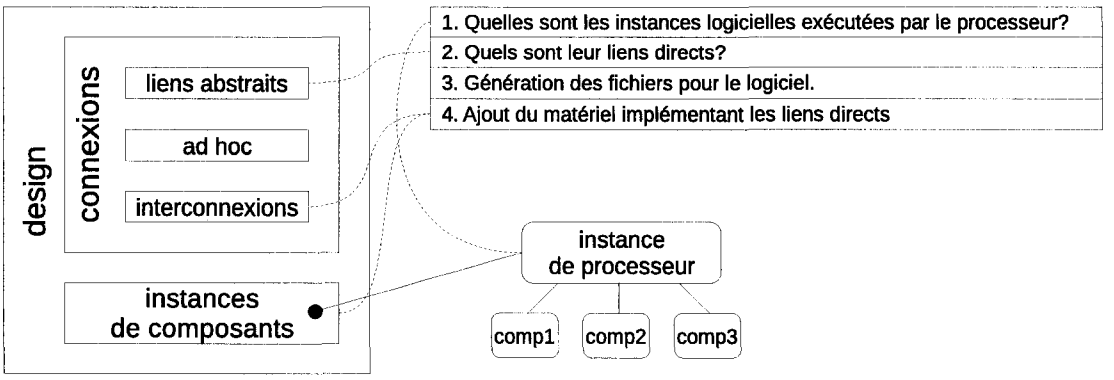


Figure 4.7: Utilisation des générateurs dans les liens directs avec logiciel

4.8.2 Liens matériel-logiciel et logiciel-logiciel

Tout comme les liens directs entre deux composants matériels, il est possible d'utiliser des canaux dédiés lorsqu'au moins un des deux composants à connecter est implémenté sous forme logicielle. Il peut s'agir de liens matériel-logiciel ou logiciel-logiciel, dans le cas où les deux composants à relier sont exécutés par des processeurs différents. Un lien point à point au sein d'un même processeur n'a pas grand intérêt. La communication peut alors se faire sous forme purement logicielle. Dans les cas qui nous intéressent, le lien utilise les ports du processeur destinées à l'accélération matérielle : liens FSL *(Fast Simplex Link)* pour le MicroBlaze et bus FCB *(Fabric Coprocessor Bus)* pour le PowerPC. Le logiciel exécuté par le processeur doit en outre être modifié pour prendre en compte ces communications. Pour résumer, l'implémentation de ces liens dépend du type de processeur utilisé, que ce soit pour la création du logiciel ou du matériel. Elle est donc confiée aux générateurs de chaque processeur, qui interrogent la plateforme pour connaître les liens directs impliquant les instances de composants qu'il exécute (figure 4.7).

Il est donc nécessaire de recenser les liens directs qui doivent être implémentés. Pour les représenter, une nouvelle classe de connexion, non standard a été créée : *DirectConnection*. Ces liens sont particuliers pour deux raisons. D'une part, ils ne référencent pas des ports ou interfaces appartenant à des instances de composant, mais ces instances elles-mêmes.

D'autre part, ils ont la possibilité de connecter des instances de composant qui ne figurent pas directement dans la structure *ComponentInstanceSet*, dans le cas où il s'agit d'instances logicielles. Pour cela, ces liens sont appelés *liens abstraits*.

Ils sont représentés par la classe *DirectConnection*, qui possède deux attributs :

1. une paire d'instances de composants (logiciels ou matériels),
2. une instance de canal.

Cette dernière permet d'identifier l'instance de type *directLink* utilisée dans le lien direct. Lorsque le lien n'a pas encore été implémenté, cet attribut est nul. Sa valeur est affectée soit après l'exécution de l'algorithme 4.10, dans le cas de lien matériel-matériel, soit par un générateur de processeur.

Cette information est ensuite utilisée lorsqu'un module est déplacé, et que cela nécessite la suppression du lien direct, dans les deux cas suivants :

1. Le module est mis en œuvre sous forme matérielle, et il est transformé en une forme logicielle.
2. Il est déplacé sous forme logicielle dans un autre processeur, qu'il provienne d'une implémentation logicielle dans un autre processeur ou matérielle.

Alors l'instance de canal est supprimée du design, ainsi que les éventuels adaptateurs la reliant aux composants fonctionnels. L'attribut correspondant devient nul. Puis la nouvelle réalisation de la connexion directe est générée.

Les objets *DirectConnection* sont enregistrés dans les extensions des fichiers de design. Un exemple se trouve dans l'annexe I.4 p. 122.

Grâce aux différents outils présentés dans ce chapitre, les modèles du système peuvent être créés et modifiés aisément, jusqu'à une description architecturale correspondant à l'implémentation. Il est alors possible de générer cette implémentation à partir de ce dernier modèle. C'est l'objet du chapitre suivant.

CHAPITRE 5

VERS L'IMPLÉMENTATION MATÉRIELLE

5.1 Introduction

Ce chapitre est consacré à la création d'une implémentation d'un système sur une carte à FPGA à partir d'un modèle transactionnel. SpaceStudio possède cette capacité ; l'outil qui la fournit se nomme Gen-X. Il génère les fichiers principaux d'un projet de Xilinx Platform Studio (XPS) en fonction d'une configuration d'un système Space au niveau Simtek (TL2). Ensuite, XPS peut être exécuté. Il charge les fichiers générés, et peut alors réaliser des simulations bas niveau, synthétiser le système, et programmer un FPGA pour qu'il l'exécute. Le travail présenté dans ce chapitre reproduit la même fonctionnalité, avec la différence majeure d'être totalement générique. La généricité rend l'outil extensible : il suffit d'ajouter des fichiers de description de composants dans le répertoire de la bibliothèque pour qu'ils puissent être supportés. De plus, le format de fichier utilisé étant ici aussi le standard IP-XACT, l'interopérabilité est encouragée.

Pour résumer, l'opération que l'on cherche à réaliser ici est la création d'une architecture bas niveau d'un système à partir d'un modèle transactionnel à plus haut niveau. Il s'agit donc avant tout d'un raffinement. Les instances de composant du design original sont remplacées par des composants équivalents plus précis, et l'architecture des connexions est conservée. Cependant, les contraintes spécifiques au bas niveau font qu'il n'est pas possible d'appliquer tel quel l'outil de raffinement vu dans la section 4.5.

La section 5.2 présente les différentes contraintes rencontrées dans la génération de l'architecture bas niveau, et leurs conséquences sur l'utilisation des fichiers IP-XACT. La section 5.3 détaille la méthode utilisée par l'outil de génération bas niveau. Le lecteur

pourra également se référer à l'annexe III pour un détail des fichiers utilisés par XPS, ainsi qu'à l'annexe I.6 pour des exemples de déclarations IP-XACT.

5.2 Contraintes et spécifications dans les fichiers IP-XACT

5.2.1 Blocs IP

Un IP *(Intellectual Property)* est un composant tel qu'il figure sur le FPGA. Il possède donc un fichier IP-XACT *component*. Son contenu est similaire à celui des fichiers *component* vus précédemment, sauf qu'il possède des extensions spécifiques aux IP.

Modèles de FPGA Il peut arriver qu'un composant soit disponible sous forme compilée *(netlist)* uniquement. Cela permet au fournisseur du module de protéger sa propriété intellectuelle en ne dévoilant pas le code en HDL. La contrainte est que cette forme compilée n'est utilisable que pour un type de FPGA. Bien sûr, il est possible que plusieurs versions soient disponibles, pour autant de types de FPGA. La liste des modèles de FPGA pour lesquels un composant est disponible se trouve dans les extensions du composant. Dans le cas où aucun FPGA n'est spécifié, on considère que les sources en HDL sont disponibles et que l'IP peut donc être implantée sur tout type de FPGA.

Version de XPS Certains composants peuvent n'être disponibles que sur certaines les versions de XPS. De même qu'au paragraphe précédent, ces versions sont listées dans les extensions, et si cette information n'apparaît pas, on considère le composant acceptable pour toute version de XPS.

Déclaration MSS Il est également possible d'indiquer la déclaration du pilote à utiliser dans les extensions du composant. Cela permet de générer les fichiers MSS. Ces informa-

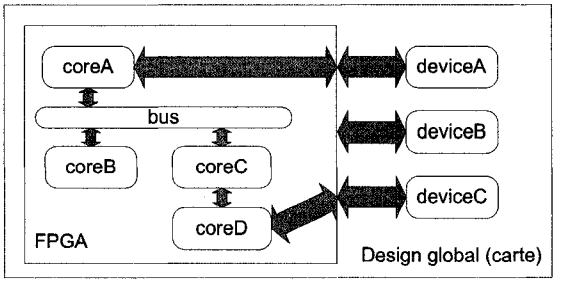


Figure 5.1: Représentation habituelle des cartes

tions sont uniquement constituées du nom du pilote et de sa version (cf. annexe I.6.1) ;

Les processeurs ont une déclaration MSS particulière, puisqu'en plus de déclarer leur pilote, ils indiquent aussi le nom de leur compilateur et archiveur.

5.2.2 Description des cartes

Considérons une carte avec un FPGA, sur lequel est exécutée une application. Une manière logique de représenter cette situation est de considérer la carte comme un design, constituée de plusieurs instances de composants. Ces instances correspondent à différents modules présents physiquement sur la carte. Parmi eux on trouve un FPGA, qui peut être vu comme un composant hiérarchique, dont les sous-modules sont les blocs qui y sont assemblés. Les connexions hiérarchiques permettent aux sous-modules de communiquer avec les instances de composants de la carte. Ce modèle est représenté figure 5.1. Il s'agit du modèle le plus naturel, et c'est d'ailleurs celui qui est utilisé par Xilinx, ce qui apparaît notamment dans la syntaxe des fichiers MHS.

Cependant, ce n'est pas le paradigme utilisé ici. En effet, nous considérerons que c'est la carte, constituée de l'ensemble des composants externes au FPGA, qui constitue un composant hiérarchique (fig. 5.2). On peut voir cela comme une sorte de retournement topologique : ce qui était à l'intérieur du composant hiérarchique se retrouve à l'extérieur, et vice versa.

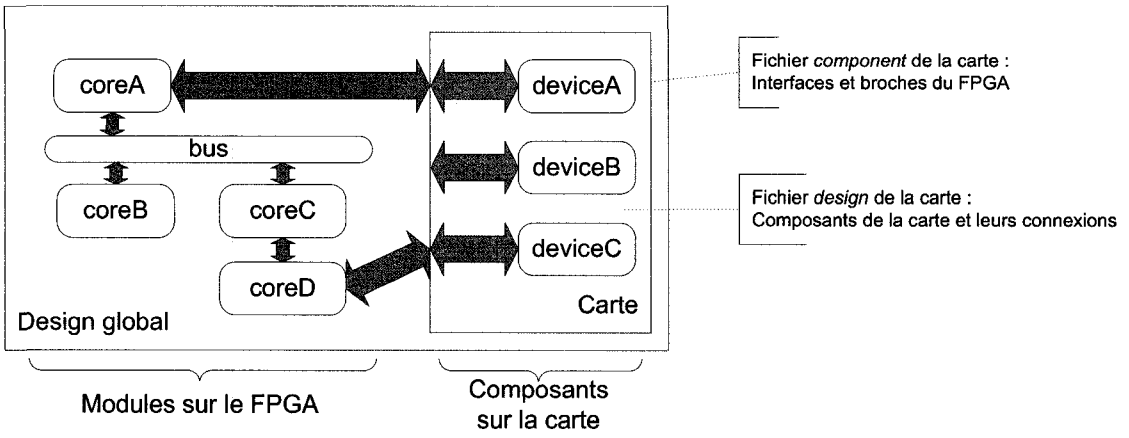


Figure 5.2: Représentation des cartes pour l'outil de raffinement à bas niveau

Cette conception est motivée par le besoin d'exprimer simplement les composants de la carte et leurs connexions avec le FPGA. De plus, la structure intérieure d'un composant hiérarchique IP-XACT n'est pas modifiable en fonction des instances du composant. Ce n'est pas gênant pour les composants matériels de la carte, dont les caractéristiques sont fixées. Pour la description du FPGA, cela est plus contraignant, puisque son design interne est généré (et donc différent) à chaque appel de l'outil de raffinement bas niveau. Il vaut donc mieux placer les composants de la carte dans la partie fixe du design, à savoir le composant hiérarchique.

Il est important de mentionner que toutes ces considération ont été faites sous l'hypothèse que la carte ne contient qu'un seul FPGA. Dans le cas où plusieurs FPGA sont présents, il ne faudrait considérer que l'un d'entre eux pour utiliser l'outil. L'utilisateur devrait ensuite modifier manuellement le design pour répartir les composants sur les FPGA disponibles. Cependant, une telle limitation est assez contraignante, en particulier parce que les composants sur la carte qui sont accessibles uniquement par les autres FPGA ne pourront pas être utilisés lors de la génération automatique du design.

Comme pour tout composant hiérarchique, deux fichiers IP-XACT sont donc utilisés pour la description des cartes : un fichier *component* et un fichier *design* (cf. fig. 5.2).

Fichier *component* Le type de composant spécifié dans ses extensions est *board*. Le modèle du FPGA est aussi indiqué, ainsi que sa position dans la chaîne JTAG (cf. annexe I.6.2.2). Le rôle du fichier est de décrire l'interface entre le FPGA et les autres composants de la carte. Comme tout fichier *component*, le fichier contient une énumération des ports et interfaces utilisées. La correspondance avec les broches physiques du FPGA *(pinout)* est spécifiée dans les extensions : pour chaque port, on peut associer une ou plusieurs déclarations UCF dans des balises *space:ucfString* (voir annexe I.6.2.1).

Fichier *design* Ce fichier décrit le design intérieur du super-composant. Les composants qui y sont instanciés sont tous de type *device*. Leurs liens avec les ports du super composant sont faits par des connexions hiérarchiques.

5.3 Méthode

5.3.1 Différences avec la méthode haut niveau

S'agissant d'un raffinement, le principe est identique à celui utilisé pour raffinement à haut niveau de la section 4.5. Il existe cependant plusieurs différences.

La première différence importante intervient dans l'exploration possibilités de designs raffinés. Lors de la sélection des composants bas niveau, il faut éliminer des choix :

1. les IP qui ne sont pas utilisables pour le FPGA ciblé,
2. les IP qui ne conviennent pas pour la version de XPS installée,
3. les composants de carte *(devices)* qui ne figurent pas sur la carte choisie.

De même qu'à haut niveau, l'exploration retourne une liste de designs possibles. Si elle contient plusieurs choix, l'utilisateur sélectionne celui qui convient. La représentation complète du design est alors créée, et l'utilisateur et les générateurs peuvent la modifier.

La classe représentant les designs à bas niveau est spéciale. Sa spécificité se situe dans le gestionnaire des instances de composants. Il est en effet nécessaire de mémoriser une correspondance entre les instances de composants disponibles sur la carte et ceux utilisés dans le design. Cela permet d'éviter d'utiliser des composants qui ne sont pas disponibles, ou d'en instancier plus qu'il n'y en a sur la carte. La correspondance est implémentée sous forme de table de hachage. Les clés de la table sont les instances de composants telles qu'elles apparaissent dans le design interne de la description de la carte. Les valeurs sont les instances du design global.

5.3.2 Génération du projet XPS

Une fois que le nouveau design est créé, l'instance de composant hiérarchique représentant la carte est contracté. On obtient donc une configuration correspondant à la figure 5.2. Il est ainsi possible de générer le fichier MHS décrivant le design. Les ports globaux sont établis à partir de la liste des connexions faites sur ce composant. Leurs directions sont inversées par rapport à celles des ports de la carte : on décrit ici un composant correspondant à l'extérieur de la carte. Puis les instances de composant sont listées avec leurs paramètres et les connexions de leur port et interface. La syntaxe des fichiers MHS pour les connexions est ici prise en compte. En particulier, les interconnexions générées sont de type normal lorsqu'elles relient un canal à un autre composant, et de type « point-à-point » pour les autres cas. La table 5.1 indique les informations utilisées pour construire les fichiers du projet. La fonction de ces fichiers est détaillée dans l'annexe III.

5.4 Cas particuliers et générateurs

Comme on l'a déjà vu, le bas niveau possède de nombreuses particularités. Certaines ont été traitées de façon générique, en définissant des extensions spéciales pour contenir les

Fichier	Informations utilisées
XPS (fichier principal)	application de chaque processeur et les fichiers qu'elle utilise
MHS	design après raffinement
UCF *(pinout)*	extensions des ports sur le fichier *component* de la carte
MSS (pilotes logiciels)	extensions des IP
linker script	générateur du processeur
download.cmd	position du FPGA dans la chaîne JTAG, dans les extensions de la carte

Tableau 5.1: Emplacement des informations pour la génération des fichiers XPS

informations nécessaires. Cependant, ce traitement ne suffisant pas à gérer tous les cas, des générateurs ont dû être ajoutés. Cette partie détaille les plus important d'entre eux.

5.4.1 Composants utilisateur

Lors du raffinement à haut niveau, les composants utilisateur ne sont généralement pas modifiés. En effet, par le principe de séparation des communications et des calculs, les interfaces de composants de l'application n'évoluent pas. Le raffinement de ces modules peut être réalisé en détaillant plus les calculs, mais cela n'est pas effectué par l'outil de raffinement, qui se concentre uniquement sur l'architecture. En revanche, pour le bas niveau, il est nécessaire de remplacer les composants utilisateur. Le langage d'implémentation est en effet changé, ainsi que les interfaces : il ne doit plus y avoir de port transactionnel.

Le remplacement des composants utilisateur est confié aux générateurs. Cette tâche est en effet dépendante du type de composant, et il n'est pas possible de la traiter simplement de façon générique. Le type des générateurs est *preTransform*. Comme on l'a vu dans la section 4.5.5.1, ces générateurs ont accès au dossier contenant les composants pour le projet courant. Dans le cas de XPS, il s'agit du dossier *pcores*. Le générateur peut alors créer le

sous-dossier dédié au nouveau composant, y ajouter les déclarations Xilinx MPD et PAO ainsi que des fichiers source. Dans les mises en œuvres qui ont été réalisées, les fichiers source générés sont des modèles à compléter. Ils possèdent tous les ports requis, mais il est nécessaire d'y ajouter manuellement l'implémentation de leur fonctionnalité. Cependant, bien que cela n'ait pas été testé, il est tout à fait possible de créer un générateur utilisant un synthétiseur comportemental de manière à automatiser complètement le raffinement. De plus, comme le générateur a également accès au design original de haut niveau, il est possible d'analyser les connexions des composants de manière à faire apparaître ou non certains ports dans les fichier bas niveau générés.

5.4.2 Horloges

À haut niveau, les horloges sont de simples composants possédant un port de sortie, qui fournit le signal d'horloge. Sa fréquence peut être choisie simplement en paramétrant l'instance d'horloge. Sur une carte, la situation est différente : l'horloge est un composant physique à l'extérieur du FPGA. Le signal est fourni sur une de ses broches. Par conséquent, la fréquence ne peut pas être choisie librement. Au mieux la carte fournit plusieurs signaux avec des fréquences différentes, mais cela constitue néanmoins un choix plus que limité. Pour conserver quand même un libre choix de la fréquence des signaux d'horloge que les composants sur le FPGA utilisent, on peut avoir recours à des DCM *(Digital Clock Manager)*. Ces composants sont capables de gérer les signaux d'horloge, et en particulier de multiplier et diviser leur fréquence.

Un générateur *postTransform* a été créé de manière à ce que les composants du FPGA aient la même fréquence d'horloge que celle définie dans le design haut niveau. La partie générique de la transformation (exécutée avant le générateur) connecte les composants à une horloge de la carte. Sa fréquence est indiquée dans par ses paramètres, déclarés dans le design intérieur de la carte. Lorsque le générateur est exécuté, il a donc accès aux

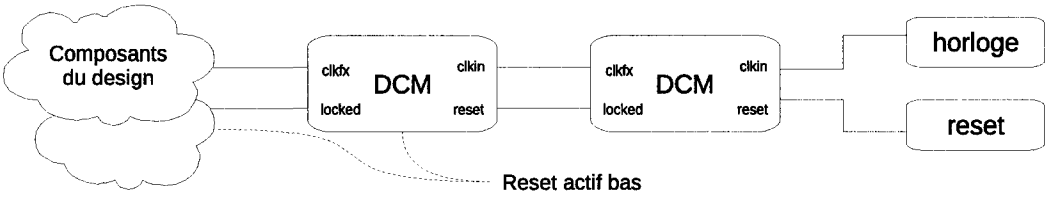


Figure 5.3: Chaînes de DCM avec signaux d'horloge et reset

fréquences dans le design d'origine et dans le nouveau design. Il peut donc calculer le rapport entre ces deux fréquences et insérer un ou plusieurs DCM en chaîne entre le port d'horloge du FPGA et les composants qui l'utilisent.

5.4.3 Signal de réinitialisation

Le signal de réinitialisation *(reset)* est fourni par un port du FPGA, connecté à un composant sur la carte. Lorsqu'un DCM est ajouté, il possède une entrée pour un tel signal, et une sortie *locked* qui est active haute lorsque le signal d'horloge de sortie est stabilisé. Le générateur dédié à ces signaux connecte le port d'entrée au signal extérieur, et le port *locked* aux composants, en changeant si besoin leurs paramètres pour indiquer que le signal *reset* est actif bas. Dans le cas où une chaîne de DCM a été utilisée, le port d'entrée *reset* de chaque DCM est connectée au port *locked* du précédent. Ici aussi, les paramètres sont modifiés en fonction du niveau actif. Une telle configuration est présentée figure 5.3.

5.4.4 Processeur

Dans le design de départ, au niveau TL2, une gestion précise de la mémoire n'est pas forcément indispensable. Par exemple, il est possible que toute la mémoire utilisée par un modèle de processeur soit sous forme d'un bloc de BRAM, même si la taille de la mémoire utilisée est insuffisante pour que cela soit faisable en pratique. Ainsi, un générateur dédié à la gestion de la mémoire par les processeurs (en l'occurrence MicroBlaze) a été créé. Il

s'agit d'un générateur *postTransform*, qui est donc appelé après le processus de raffinement. Il propose à l'utilisateur de choisir les mémoires de données et instructions parmi les instances existantes, ou d'en créer de nouvelles. Pour chaque mémoire, l'adresse de base et la taille sont sélectionnées. Les choix de l'utilisateur sont enregistrés et seront utilisés par un second générateur, de type *normal* pour créer le fichier *linker script*.

Si des mémoires se retrouvent non utilisées par le processeur, le générateur en propose la suppression à l'utilisateur. Dans le cas où la mémoire d'instructions se situe dans une mémoire externe (SDRAM), un pont entre le bus de données et le bus d'instructions est créé. Ce pont permet au processeur d'écrire dans la mémoire d'instructions lors de son initialisation *via* le bus des données, ce qui serait impossible autrement.

5.5 Outils supplémentaires pour la génération de fichiers IP-XACT

La création de documents IP-XACT peut se révéler fastidieuse, en particulier pour le bas niveau, où les composants et les interfaces possèdent de nombreux ports. Heureusement, dans le cas de XPS, une grande partie de l'information nécessaire est déjà présente dans des fichiers de la bibliothèque de composants Xilinx. Ce ne sont pas des fichiers au format XML, *a fortiori* IP-XACT. Des outils ont donc été conçus pour lire ces documents, et créer des modèles de fichiers IP-XACT à compléter.

5.5.1 Lecteur de MPD

Les fichiers MPD *(Microprocessor Peripheral Definition)* décrivent les paramètres, ports et interfaces de composants. À partir de ces informations, il est possible de générer un fichier *component*. Ce fichier contient les ports, les interfaces avec la correspondance entre ports logiques et physiques, ainsi que les paramètres du composant. Certaines informations doivent ensuite être ajoutées manuellement :

1. l'identifiant VLNV du composant,
2. les identifiants des interfaces : *busType* et *abstractionType*, ainsi que leur mode (maître, esclave, ...),
3. les extensions.

Il est également possible de générer un squelette de document *abstractionDefinition* pour chacune des interfaces du composant. Cela est utile dans le cas où l'interface est d'un type non encore décrit dans la bibliothèque IP-XACT.

5.5.2 Lecteur de XBD

Les fichiers XDB *(Xilinx Board Definition)* sont consacrées à la définitions des cartes sur lesquelles les systèmes sont exécutés. Le lecteur de XBD génère le fichier *component* des cartes. De même que pour le lecteur de MPD, le fichier généré contient les ports et interfaces utilisées, et la correspondance entre ports logiques et physiques. Les chaînes UCF indiquant l'emplacement des ports sur les broches du FPGA sont également générées dans les extensions des ports. Les squelettes de fichiers *abstractionDefinition* peuvent aussi être créés. La travail manuel restant est un peu plus important que pour le lecteur de MPD. En plus des informations énumérées pour le lecteur de MPD, il faut ajouter le fichier décrivant le design intérieur.

5.5.3 Lecteur de sc_main

La fichier principal d'une simulation SystemC contient un grand nombre d'informations relatives au design simulé : la liste des instances de composants avec leur type et valeurs de paramètres, ainsi que les connexions (ad hoc). Un outil a été conçu pour créer un fichier *design* à partir de tels fichiers. Il a été initialement créé pour pouvoir bénéficier de l'outil de raffinement bas niveau présenté dans ce chapitre, à partir d'un design créé par un logiciel

non compatible IP-XACT (SpaceStudio par exemple), ou manuellement. L'analyseur ne respecte pas rigoureusement toutes les règles syntaxiques de C++. Le design généré doit donc être vérifié. Cependant, cet outil a été testé avec succès sur plusieurs configurations générées par SpaceStudio. Son exécution se déroule comme suit.

En premier lieu, un design vide est créé. Puis le fichier est analysé, en différenciant les déclarations d'instances de composants et celles des connexions.

Pour chaque déclaration d'instance de *sc_module* c, on recherche dans la bibliothèque les composants dont le nom (dans l'identifiant VLNV) correspond à celui de c. Vu leur statut particulier dans SystemC (cf. sec. 6.1), une règle spéciale est créée pour les horloges. Les *sc_clock* sont remplacées par des composants générant un signal d'horloge.

Les connexions sont également analysées. Il s'agit de connexions port-à-port (ad hoc). Le traitement des connexions transactionnelles et *wire* est différent. Pour chaque déclaration de connexion transactionnelle SystemC, une nouvelle connexion du design est créée. S'il s'agit d'une connexion « port-à-interface », elle est transformée en connexion « port-à-export », d'après la liste des exports déclarés dans la description du composant. Pour les connexions *wire*, la déclaration SystemC associe les ports à des signaux. L'analyseur SystemC maintient une table de hachage multiple associant une liste de ports à chaque signal. Une fois le parcours du fichier terminé, une connexion *wire* est ajoutée au design pour chaque entrée dans la table de hachage. Ces connexions possèdent le même nom que le signal et associent tous les ports connectés à un même signal.

Enfin, les connexions ad hoc sont remplacées lorsque cela est possible par des interconnexions (reliant des interfaces). Si le passage d'une interconnexion à un ensemble de connexions ad hoc est habituel, le contraire ne l'est pas. La règle utilisée ici est la suivante : si toutes les connexions ad hoc correspondant à une interconnexion existent, alors l'interconnexion est créée et la connexion ad hoc supprimée.

On obtient donc un design possédant des instances de composants et des interconnexions pour les relier. Bien que le lecteur de fichiers SystemC ait été créé pour utiliser le raffinement bas niveau, il est possible de profiter de toutes les fonctionnalités de la plateforme après son exécution.

CHAPITRE 6

APPLICATION

6.1 Présentation du système

Ce chapitre présente un cas d'utilisation de la plateforme détaillée dans ce mémoire. Son objectif est double. D'une part, cela met en évidence le type d'interactions qui peuvent se produire entre l'utilisateur et la plateforme. En particulier, il est possible de constater leur simplicité, ce qui constitue un des objectifs majeurs. D'autre part, cette utilisation valide le bon fonctionnement de la plateforme, à la fois pour sa fonctionnalité de base (gestion des composants et designs), et pour les outils. L'interface utilisateur se fait sous la forme d'une console possédant ses commandes propres. Il est possible de grouper des suites de commandes dans des scripts pour les exécuter successivement. L'annexe IV liste les commandes disponibles et leur fonction.

La mise en œuvre de ce chapitre consiste à créer un design à haut niveau, puis de le raffiner une première fois pour obtenir un modèle transactionnel plus précis, puis une seconde fois pour créer une implémentation sur une carte. Entre ces opérations de raffinement, des outils du chapitre 4 sont utilisés.

La bibliothèque de composants utilisés ici est SpaceLib. Pour qu'elle puisse être documentée avec IP-XACT, elle a subi les deux modifications suivantes :

1. Des *sc_export* ont été ajoutés aux composants implémentant une *sc_interface*. L'export est connecté au module lui-même, tel que représenté dans la figure 2.3 p. 26. La compatibilité avec les simulations n'utilisant pas les exports est donc conservée.
2. Un composant *horloge* a été ajouté. En effet, les horloges SystemC *(sc_clock)* peuvent être vues à la fois comme des signaux et comme des composants : d'une part

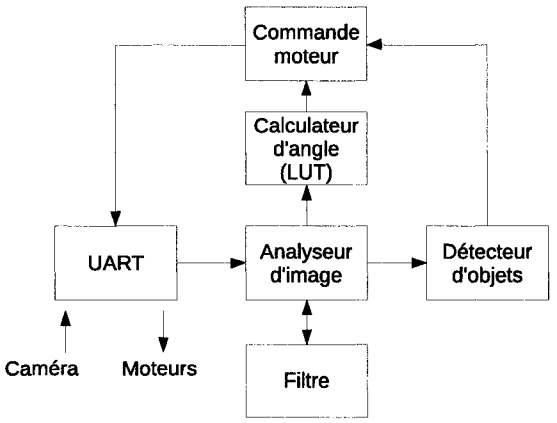


Figure 6.1: Principe de fonctionnement du rover

elles peuvent se connecter directement aux ports d'entrée d'horloge des composants, et d'autre part elles possèdent des paramètres (nom, période et rapport cyclique). Cela ne correspond pas aux paradigmes de IP-XACT. Un module générant une horloge sur un de ses ports de sortie a donc été créé. Ses paramètres sont ceux du *sc_clock* qui se trouve à l'intérieur.

L'utilisation de SpaceLib implique que les fonctions de communication appelées par les modules soient celles de Space. Cependant, du fait de la généricité de la plateforme, toute autre technologie pourrait être utilisée.

Le système considéré ici est un contrôleur de véhicule automatique rover. Il a déjà été utilisé comme étude de cas de la plateforme Space [26, 32]. Cela permet de réutiliser les modules de l'application déjà existants, et de se concentrer sur les fonctionnalités de gestion d'architecture, qui constituent le rôle de la plateforme.

Le principe de fonctionnement du système est représenté figure 6.1[1]. Le véhicule possède une caméra. Ses données sont envoyées au système *via* un port sériel RS232 sur un UART. Le système analyse cette image et envoie des commandes au moteur en conséquence, toujours par le UART. Le but est que le véhicule ne quitte pas une piste tracée au sol.

[1]Cette figure, ainsi que la description dans ce paragraphe, reposent en grande partie sur [26].

L'analyseur d'image lit une image au complet (8×8 pixels) et utilise le filtre pour en enlever le bruit ligne par ligne. Puis il calcule la direction pour rester sur la piste et l'envoie au calculateur d'angle. Ce dernier convertit la direction en angle, à l'aide d'une table de conversion (LUT, *look-up table*). L'angle est ensuite transmis au module de commande des moteurs, qui envoie au UART les commandes pour les moteurs gauche et droit. Parallèlement, un détecteur d'objets analyse l'image et informe le contrôleur des moteurs si des obstacles sont détectés à proximité. Dans ce cas, la vitesse est réduite, ce qui permet d'augmenter la précision pour la position du véhicule, et donc d'éviter les obstacles.

6.2 Création du premier design

Initialisation La première étape consiste à créer un projet vide *demoRover*, et d'y ajouter un design vide. Le design (ou configuration) se nomme ici *conf_elix*.

```
newproject demoRover
newconf conf_elix
```

La première commande crée également le dossier correspondant au projet dans le système de fichiers, ainsi que son arborescence interne.

Création des composants utilisateur Puis les cinq composants utilisateurs sont ajoutés au projet.

```
newusermodule ObjectDetector
[...]
newusermodule ImgAnalizer
```

Lorsque cette fonction est appelée, la bibliothèque est analysée pour rechercher les composants qui sont marqués comme base de composant utilisateur. Dans le cas présent, un seul composant répond à ce critère. C'est le module virtuel *SpaceBaseModule*, dont l'identifiant VLNV est « *spacecodesign.com, base, SpaceBaseModule, 0.2* ». S'il y en

avait plusieurs, le choix serait laissé à l'utilisateur. Le processus détaillé section 4.6 est alors exécuté. Les nouveaux composants sont enregistrés dans le projet et des modèles de fichiers source sont générés. Ceux-ci peuvent alors être modifiés par l'utilisateur pour qu'ils décrivent l'application.

Instanciation des composants Tous les composants nécessaires à la création du premier design sont maintenant disponibles. Le but de ce design est de valider la fonctionnalité globale. Les composants qui doivent y figurer sont les cinq composants utilisateur ainsi qu'un UART et un modèle de canal sans estimation de temps (UTF). Tous les composants seront connectés à ce canal.

```
newinst spacecodesign.com base ObjectDetector 0.2 my_ObjectDetector
[instanciation des autres composants utilisateur]
newinst spacecodesign.com base ImgAnalizer 0.2 my_ImgAnalizer
newinst spacecodesign.com serial SpaceUART 0.1 uart
newinst spacecodesign.com channel UTFChannel 0.2 channel
```

Les quatre premiers arguments de la commande *newinst* sont l'identifiant VLNV du composant, et le dernier est le nom de la nouvelle instance.

L'instanciation des composants utilisateur implique l'exécution automatique d'outils. D'une part, ils possèdent un port d'entrée *reset*, qui correspond à une interface marquée comme signal obligatoire. L'outil dédié aux signaux obligatoires (section 4.2) ajoute une nouvelle instance de gestionnaire *reset*, et la connecte aux nouveaux modules. De plus, le UART possède une interface d'horloge. Celle-ci est donc connectée à une nouvelle instance d'horloge. D'autre part, les composants utilisateur possèdent deux paramètres correspondant à la période d'horloge. L'un correspond à la partie numérale, l'autre à l'unité. L'outil d'importation de paramètres (section 4.3) définit la dépendance entre les paramètres de l'horloge et ceux des composants utilisateur. Si l'horloge n'avait pas existé à ce stade, elle aurait été instanciée pour satisfaire ces dépendances.

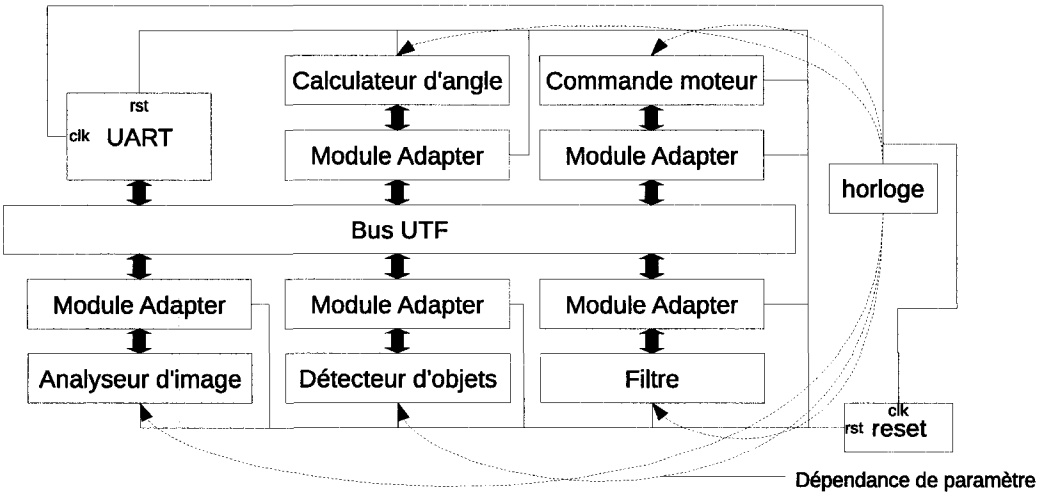


Figure 6.2: Premier design du rover

De plus, les composants utilisateur possèdent un générateur de type *postInstantiation*. Son rôle est de gérer les identifiants de composants Space utilisés pour les communications. Pour chacun des cinq modules de l'application, une nouvelle constante est déclarée ; elle correspond à l'identifiant. Une valeur est attribuée à cette variable, de manière à ce que chaque identifiant soit unique dans le design.

Ajout des connexions À ce stade, les instances ne sont pas connectées, à l'exception des interfaces d'horloge et *reset*. L'ajout de connexions se fait avec la commande suivante :

```
bind my_ObjectDetector channel
```

L'opération est bien sûr répétée pour tous les composants à connecter au canal. Cela appelle la génération d'adaptateurs (cf. sec. 4.4). Un adaptateur de type *ModuleAdapter* est inséré entre chaque module utilisateur et le canal. Le UART étant compatible avec le bus, la connexion peut se faire directement, sans adaptateur. Le design obtenu est représenté figure 6.2.

Création de la simulation Il est désormais possible de créer les fichiers de simulation, en appelant la commande *generate*. Cela exécute les éventuels générateurs normaux, puis

crée les fichiers nécessaires à la simulation SystemC :

1. le fichier principal contenant la fonction *sc_main*, dans laquelle les instances de composants sont instanciées et connectées, et la simulation du système démarrée,
2. son fichier d'en-tête *(header)*,
3. un fichier *PlatformDefinitions.h* contenant des définition de constantes utilisées dans ce design, en particulier les valeurs des identifiants,
4. le *makefile* à utiliser pour compiler l'exécutable de la simulation.

6.3 Second design et partitionnement

Raffinement L'outil de raffinement est appelé pour créer un design au niveau inférieur, qui se nomme *Simtek* dans le cas de Space.

```
chglevel simtek conf_simtek
```

On obtient alors le design de la figure 6.3. Les dépendances de paramètres pour les modules utilisateurs sont toujours présentes, mais n'y sont pas représentées pour des raisons de lisibilité. Les composants utilisateur ont été simplement copiés. Les valeurs de leurs paramètres n'ont donc pas été changées. Le canal UTF a été remplacé par un modèle de bus OPB de niveau TL2, et le modèle de UART par un modèle de plus bas niveau, qui peut se connecter directement sur le bus OPB. L'arbitre du bus est inclus dans le composant OPB, il n'apparaît donc pas.

Ajout d'un processeur Le niveau de cette étape étant celui destiné à l'exploration architecturale, il est pertinent d'y ajouter un processeur. Pour ne pas perdre le design courant, il est d'abord copié. Le modèle de processeur est un simulateur de jeu d'instructions (ISS) de MicroBlaze. Comme il doit être instancié avec d'autres composants, on utilise ici un composant hiérarchique constitué du modèle de processeur et de ses dépendances. Puis il est étendu, pour faire apparaître les sous-composants.

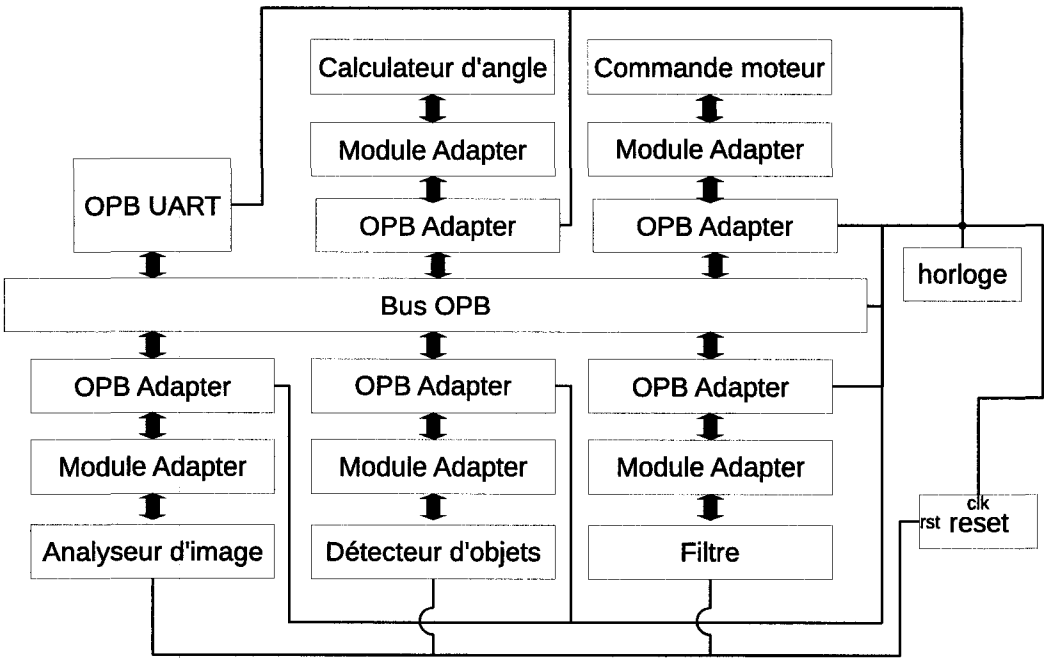


Figure 6.3: Design après raffinement

```
copyconf conf_simtek1ub
newinst spacecodesign.com iss uBlazeSuperComp 0.2 uBSC
expand uBSC
```

Le nom de l'instance de l'ISS qui est ajoutée lors de l'expansion du super-composant se nomme *uBlaze*. Celui-ci supportant l'implémentation logicielle des composants utilisateurs, certains d'entre eux sont déplacés :

```
move2sw my_ObjectDetector uBlaze
move2sw my_LRCTRL uBlaze
move2sw my_ImgAnalizer uBlaze
```

On obtient alors l'architecture de la figure 6.4. L'horloge et le contrôleur de réinitialisation n'y sont pas représentés. Parmi les modules du super-composant, on trouve un adaptateur de l'ISS. Ce composant Space a pour but de gérer les communications matériel-logiciel. Un *timer* est aussi ajouté. Ces deux modules sont connectés à un contrôleur d'interruptions, lui-même connecté au processeur. De plus, une mémoire de type *BRAM* est ajoutée au design. Elle contient les sections de données et d'instructions de l'exécutable de processeur.

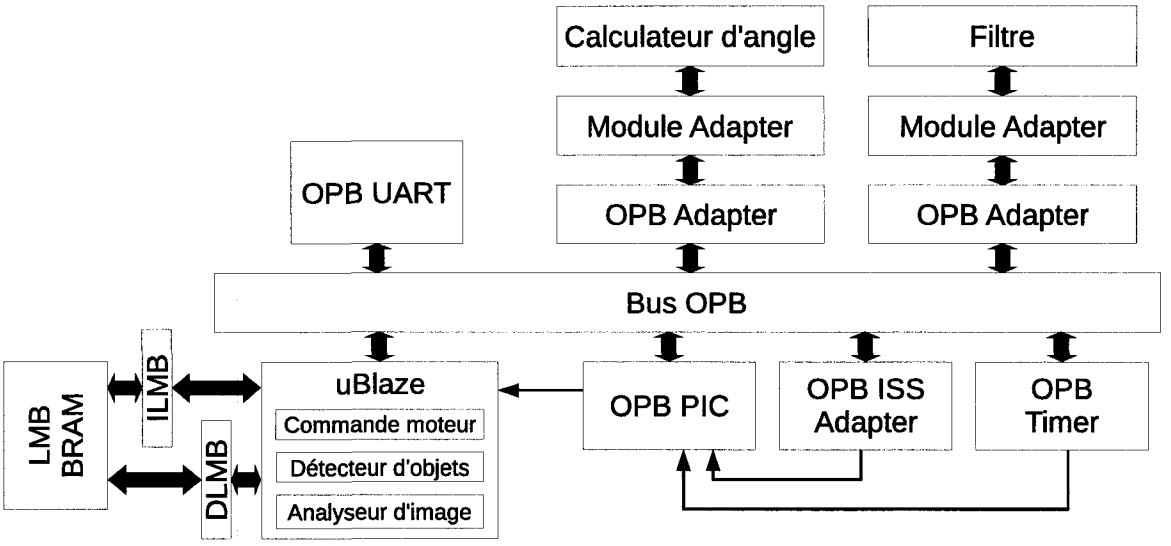


Figure 6.4: Design avec un modèle de processeur

La génération des fichiers de la simulation peut ensuite être exécutée. Cette fois, non seulement les fichiers décrivant l'architecture matérielle en SystemC sont générés, mais aussi ceux destinés à la création du logiciel exécuté par le modèle de processeur. Cela est réalisé par les générateurs du processeur, qui interrogent la plateforme pour obtenir l'ensemble des composants logiciels, puis créent les fichiers en conséquence. Ce sont des fichiers source, ainsi qu'un *makefile* et un *linker script*. Le chemin des fichiers source est enregistré dans l'objet correspondant à l'instance de processeur.

6.4 Génération bas niveau

Ce dernier design, avec l'application répartie entre une implémentation matérielle et une exécution logicielle faite par un Microblaze peut être implémenté sur la carte. La commande a exécuter est la suivante :

```
genx xilinx.com board V2Multimedia 1 system
```

Le dernier argument est le nom du projet XPS à générer. Les quatre autres arguments *xilinx.com, board, V2Multimedia, 1* sont les identifiants de la carte. Il s'agit donc de la

slices	3 788	35%
bascules	4 160	19%
tables de conversion *(LUT)*	5 597	26%
entrées-sorties	65	10%
BRAM	8	14%
multiplicateurs 18×18	3	5%
horloges globales	4	25%
DCM	1	12%

Tableau 6.1: Ressources utilisées sur le FPGA

carte Xilinx Multimedia [33]. Elle possède un FPGA Virtex-II XC2V2000-FF896. Elle contient aussi cinq banques de mémoire (ZBT RAM) de taille 512Ki×32 bits chacune. Ces cinq éléments sont indépendants et connectés directement aux broches du FPGA. La carte possède aussi entre autres deux horloges et un port sériel RS 232.

Le processus détaillé dans le chapitre 5 est alors exécuté. Une horloge de la carte est utilisée avec un DCM de manière à conserver la période choisie dans les modèles transactionnels. Le générateur du MicroBlaze demande alors quelles mémoires utiliser pour le logiciel (cf. sec. 5.4.4). Pour cette architecture, les sections de données et d'instructions sont placées dans une mémoire de la carte, en raison de la taille importante occupée par le logiciel (313 024 octets).

Après implémentation des composants utilisateur matériels, le système a été synthétisé *via* XPS. La table 6.1 contient des données extraites du rapport de synthèse. En outre, le rapport indique que le système peut fonctionner avec une période minimale de 17,353 ns, soit une fréquence maximale de 57,627 MHz.

Le système a ensuite été téléchargé sur la carte. Un simulateur envoie les données correspondant aux images de la caméra par le port RS 232. Après traitement par l'application sur la carte, les commandes des moteurs sont retransmises au simulateur qui calcule l'image suivante, et ainsi de suite. L'application ainsi obtenue fonctionne correctement.

CHAPITRE 7

CONCLUSION

Le travail présenté dans ce mémoire montre qu'il est possible de créer une plateforme de conception remplissant les trois objectifs définis dans l'introduction : réutilisation, modèles à haut niveau et automatisation.

La création des outils génériques et leur intégration dans un environnement de conception constituent la principale contribution de ce travail de maîtrise. La généricité, couplée à l'utilisation d'un format de données standard permet la réutilisation des composants, designs et interfaces.

Ce format IP-XACT convient pour les besoins de description généraux. Cependant, il ne suffit pas en l'état pour couvrir toutes les fonctionnalités apportées par la plateforme. Il a donc été nécessaire de l'étendre, en utilisant les sections *vendorExtensions* des générateurs non standard.

Le fait que IP-XACT ait été conçu de manière extensible est très utile. Grâce à cela, tous les fichiers de description utilisés sont conformes à la norme. Il est donc possible de les utiliser tels quel dans n'importe quelle autre plateforme compatible IP-XACT. Réciproquement, comme les extensions ne sont pas obligatoires pour la plateforme, elle est capable de lire tout fichier IP-XACT valide.

Lorsque le comportement de la plateforme doit être propre à un certain modules, il est mis en œuvre à l'intérieur d'un générateur. La plateforme elle-même reste ainsi générique.

La cas d'utilisation présenté au chapitre 6 permet de faire les constatations suivantes : D'une part, une fois la bibliothèque chargée, l'exécution des commandes entrées dans

la plateforme est perçue comme instantanée par l'utilisateur. D'autre part, les résultats produits par les différents outils (adaptateurs, raffinement, implantation matérielle, etc.) sont identiques à ceux de SpaceStudio. Pour une bibliothèque existante donnée, la généricité n'apporte pas d'inconvénient à l'utilisateur par rapport à une solution codée en dur.

Pour les composants *normaux*, c'est-à-dire sans générateur, l'ajout se fait très rapidement : il suffit de décrire le composant à ajouter. Les composants utilisant un ou plusieurs générateurs, en particulier les processeurs et les bases de composants utilisateur, la tâche est plus longue et compliquée. Cependant, il faut remarquer que l'utilisation des générateurs n'augmente pas le travail nécessaire à l'ajout d'un composant. La complexité est simplement déplacée. Au lieu d'implémenter une fonctionnalité dans le cœur de la plateforme, on la place à l'extérieur. De plus, les fonctions de la partie générique restent utilisables, ce qui diminue la quantité de travail. Par exemple, supposons que l'on veuille ajouter un nouveau type de composant utilisateur, avec de nouvelles interfaces. Dans une plateforme non générique, il faudrait modifier les parties gérant les adaptateurs. Ce n'est pas le cas pour la plateforme générique présentée ici. On peut donc conclure que même si l'ajout de certains composants peut s'avérer coûteuse en termes de temps de développement, le travail total reste inférieur à celui d'une plateforme non générique.

La mesure (quantitative) des avantages apportés par la généricité aurait demandé de créer un nouvel ensemble de composants (bus, processeur, périphériques) et de comparer leur temps d'intégration dans la plateforme générique et dans une solution non générique (SpaceStudio par exemple). Ensuite, un cas d'utilisation aurait permis d'observer le temps mis par l'utilisateur pour développer le système et de comparer les systèmes produits. En principe, ils devraient être similaires. Cette opération aurait demandé un temps de développement important, d'abord pour créer les modèles SystemC (pour vérifier la validité du système en le simulant), mais surtout dans la modification de SpaceStudio. Il n'a donc pas été possible de réaliser ces mesures ; cela pourrait constituer un travail intéressant pour

le futur.

Lors du développement de la plateforme, il y a eu à plusieurs reprises un dilemme entre l'ajout d'une nouvelle fonctionnalité ou son placement dans les générateurs. Par exemple, la dépendance de paramètres (cf. section 4.3) peut être considérée comme une fonctionnalité propre à Space, ou utilisable pour tout type de composant. Dans le premier cas, il faudrait la placer dans les générateurs, et dans le second elle est présente dans la plateforme, avec utilisation de balises d'extensions dans les documents IP-XACT. Il a été décidé que ce second choix s'appliquait, car d'autres types de composants, non propres à Space pourraient utiliser l'importation de paramètres. Un autre exemple est celui des identifiants Space. Comme quasiment tous les composants de la bibliothèque utilisées (SpaceLib) les utilisent, il était tentant d'en faire une fonctionnalité générale de la plateforme, avec recours des extensions. Cependant, comme il s'agit d'une technique propre à Space, il est préférable de placer la gestion des identifiants dans un générateur.

En résumé, les bases d'une plateforme de conception générique à haut niveau ont été posées dans ce mémoire. Dans le cadre d'une exploitation industrielle, il serait utile d'y ajouter une interface utilisateur graphique. Si on considère une architecture MVC (Modèle, Vue, Contrôleur), le modèle (gestion des composants et design), et le contrôleur (outils) ont déjà été implémentés. Il reste à créer une vue dans laquelle les données du modèle sont mises en forme de manière graphique, et capable d'appeler les outils du contrôleur.

RÉFÉRENCES

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, vol. 4, 1965.

[2] P. Magarshack, "Improving SoC design quality through a reproducible design flow," *Design and Test of Computers, IEEE*, vol. 19, n° 1, p. 76–83, jan./fév. 2002.

[3] L. Cai et D. Gajski, "Transaction level modeling : an overview," in *CODES+ISSS '03 : Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis.* New York, NY, USA : ACM, 2003, p. 19–24.

[4] F. Ghenassia et A. Clouard, "TLM : An overview and brief history," in *Transaction Level Modeling with SystemC : TLM Concepts and Applications for Embedded Systems*, F. Ghenassia, Ed. Springer, 2005, ch. 1, p. 1–22.

[5] N. Laug, G. Bois, et M.-A. Cantin, "Assisted creation and refinement of transactional level specifications based on IP-XACT," in *IP based electronic system Conference and Exhibition (IP08)*, Grenoble, France, déc. 2008.

[6] L. Maillet-Contoz et J.-P. Strassen, "TLM modeling techniques," in *Transaction Level Modeling with SystemC : TLM Concepts and Applications for Embedded Systems*, F. Ghenassia, Ed. Springer, 2005, ch. 3, p. 57–94.

[7] A. Rose, S. Swan, J. Pierce, et J.-M. Fernandez, "Transaction Level Modeling in SystemC," *Open SystemC Initiative*, 2005.

[8] OCP-IP, *Open Core Protocol Specification*, 2006. [En ligne]. Disponible : http://www.ocpip.org/socket/ocpspec/ [Consulté le 15 septembre 2008].

[9] Open SystemC Initiative (OSCI), *OSCI TLM-2.0 User Manual*, juin 2008.

[10] Xilinx, Inc., *EDK concepts, tools and techniques*, 2008. [En ligne]. Disponible : http://www.xilinx.com/ise/embedded/edk91i_docs/edk_ctt.pdf [Consulté le 3 novembre 2008].

[11] Altera Corporation, *Introduction to the Quartus II software*, 2008. [En ligne]. Disponible : http://altera.com/literature/manual/intro_to_quartus2.pdf [Consulté le 3 novembre 2008].

[12] T. Ismail, M. Abid, et A. Jerraya, "Cosmos : a codesign approach for communicating systems," sept. 1994, p. 17–24.

[13] W. Cescirio, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. Jerraya, et M. Diaz-Nava, "Component-based design approach for multicore socs," 2002, p. 789–794.

[14] J. Chevalier, O. Benny, M. Rondonneau, G. Bois, E. M. Aboulhamid, et F.-R. Boyer, "Space : a hardware/software SystemC modeling platform including an RTOS," p. 91–104, 2004.

[15] Open SystemC Initiative, "SystemC 2.2," 2007. [En ligne]. Disponible : http://www.systemc.org/ [Consulté le 15 septembre 2008].

[16] F. Bouchhima, G. Nicolescu, M. Aboulhamid, et M. Abid, "Discrete-continuous simulation model for accurate validation in component-based heterogeneous SoC design," *Rapid System Prototyping, 2005. (RSP 2005). The 16th IEEE International Workshop on*, p. 181–187, juin 2005.

[17] Magillem Design Services, *MPA : Magillem Platform Assembly.* [En ligne]. Disponible : http://magillem.com/index.php?rubrique=4163 [Consulté le 3 novembre 2008].

[18] E. Lee et S. Neuendorffer, *MoML : A Modeling Markup Language in SML : Version 0.4*, mars 2000.

[19] The SPIRIT consortium, "IP-XACT v1.4 : A specification for XML meta-data and tool interfaces," mars 2008. [En ligne]. Disponible : http://www.spiritconsortium.org/tech/docs/ [Consulté le 3 août 2008].

[20] B. Stroustrup, *The C++ Programming Language*, col. Addison-Wesley series in computer science. Reading, MA : Addison-Wesley, 1986.

[21] The World Wide Web Consortium, *Namespaces in XML 1.0 (Second Edition)*, août 2006. [En ligne]. Disponible : http://www.w3.org/TR/REC-xml-names/ [Consulté le 7 septembre 2008].

[22] T. Berners-Lee, R. Fielding, et L. Masinter, "Uniform resource identifier (URI) : Generic syntax," RFC 3986, IETF, jan. 2005. [En ligne]. Disponible : http://tools.ietf.org/html/rfc3986 [Consulté le 20 août 2008].

[23] W3C, *SOAP version 1.2*, avril 2007. [En ligne]. Disponible : http://www.w3.org/TR/soap12 [consulté le 7 septembre 2008].

[24] C. G. Lasater, *Design Patterns*. Wordware Publishing, 2007.

[25] B. Meyer, *Conception et programmation orientées objet*. Eyrolles, 1997.

[26] L. Moss, M.-A. Cantin, G. Bois, et E. M. Aboulhamid, "Automation of communication refinement and hardware synthesis within a system-level design methodology," *Rapid System Prototyping, 2008. RSP '08. The 19th IEEE/IFIP International Symposium on*, p. 75–81, juin 2008.

[27] M. López-Vallejo et J. C. López, "On the hardware-software partitioning problem : System modeling and partitioning techniques," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 8, n° 3, p. 269–297, 2003.

[28] Forte Design Systems, *Cynthesizer User's Guide For Cynthesizer 3.3*, San Jose, CA, 2007.

[29] A. Gerstlauer, H. Yu, et D. Gajski, "RTOS modeling for system level design," 2003, p. 130–135.

[30] R. Oakley, "Qnx microkernel technology : a scalable approach to real-time distributed and embedded systems," *Proceedings of the Symposium on Operating System Principles. ACM*, 1997.

[31] S. Goyette, "Élaboration d'un modèle d'abstraction des communications point-à-point pour une plateforme (SoC) multiprocesseur hétérogène," M.Sc.A., École Polytechnique de Montréal, Qc, Canada, 2008.

[32] L. Filion, M.-A. Cantin, L. Moss, E. M. Aboulhamid, et G. Bois, "Space codesign : A SystemC framework for fast exploration of hardware/software systems," in *Design & Verification Conference and Exhibition (DVCON'07)*, San Jose, CA, 2007.

[33] Xilinx, *MicroBlaze and multimedia development board user guide*, août 2002. [En ligne]. Disponible : http://www.xilinx.com/products/boards/multimedia/ [Consulté le 20 septembre 2008].

ANNEXE I

EXTRAITS DE FICHIERS IP-XACT

I.1 Signal requis

```
1  <spirit:busInterface>
2    <spirit:name>clock</spirit:name>
3    <spirit:busType spirit:vendor="spiritconsortium.org"
         spirit:library="busdef.clock" spirit:name="clock"
         spirit:version="1.0"/>
4    <spirit:abstractionType spirit:vendor="spiritconsortium.org"
         spirit:library="busdef.clock" spirit:name="clock_rtl"
         spirit:version="1.0"/>
5    <spirit:slave/>
6    <spirit:portMaps>
7      <spirit:portMap>
8        <spirit:logicalPort>
9          <spirit:name>CLK</spirit:name>
10       </spirit:logicalPort>
11       <spirit:physicalPort>
12         <spirit:name>OPB_Clk</spirit:name>
13       </spirit:physicalPort>
14     </spirit:portMap>
15   </spirit:portMaps>
16   <spirit:vendorExtensions>
17     <space:busInterfaceExtensions>
18       <space:mandatorySignal>true</space:mandatorySignal>
19     </space:busInterfaceExtensions>
20   </spirit:vendorExtensions>
21 </spirit:busInterface>
```

I.2 Importation de valeur de paramètre

I.2.1 Dans le fichier component

```
1  <spirit:modelParameter>
2    <spirit:name>dClockPeriod</spirit:name>
```

```
3    <spirit:value spirit:id="dClock_periodID" />
4    <spirit:vendorExtensions>
5      <space:modelParameterExtensions>
6        <space:origin>
7          <space:componentRef spirit:vendor="spacecodesign.com"
               spirit:library="clock" spirit:name="Clock" spirit:version
               ="0.1" />
8          <space:configurableElement space:referenceId="PeriodID" />
9        </space:origin>
10     </space:modelParameterExtensions>
11   </spirit:vendorExtensions>
12 </spirit:modelParameter>
```

I.2.2 Dans le fichier design

```
1  <spirit:componentInstance>
2    <spirit:instanceName>Clock1</spirit:instanceName>
3    <spirit:componentRef spirit:library="clock" spirit:name="Clock"
        spirit:vendor="spacecodesign.com" spirit:version="0.1"/>
4    <spirit:configurableElementValues>
5      <spirit:configurableElementValue spirit:referenceId="
          ctor_name_id">Clock1</spirit:configurableElementValue>
6    </spirit:configurableElementValues>
7  </spirit:componentInstance>
8  <spirit:componentInstance>
9    <spirit:instanceName>my_LRCTRL</spirit:instanceName>
10   <spirit:componentRef spirit:library="base" spirit:name="LRCTRL"
        spirit:vendor="spacecodesign.com" spirit:version="0.2"/>
11   <spirit:configurableElementValues>
12     <spirit:configurableElementValue spirit:referenceId="
          ctor_name_id">my_LRCTRL</spirit:configurableElementValue>
13     <spirit:configurableElementValue spirit:referenceId="
          dClock_periodID">40</spirit:configurableElementValue>
14     <spirit:configurableElementValue spirit:referenceId="UnitID">
```

```
        SC_NS</spirit:configurableElementValue>
15      <spirit:configurableElementValue spirit:referenceId="ucIDID">4</
            spirit:configurableElementValue>
16      <spirit:configurableElementValue spirit:referenceId="
            ucAccessPriorityID">4</spirit:configurableElementValue>
17    </spirit:configurableElementValues>
18    <spirit:vendorExtensions>
19    <space:componentInstanceExtensions>
20      <space:paramDependencies>
21      <space:paramDependency space:parameterID="dClock_periodID">
            Clock1</space:paramDependency>
22      <space:paramDependency space:parameterID="UnitID">Clock1</
            space:paramDependency>
23      </space:paramDependencies>
24    </space:componentInstanceExtensions>
25    </spirit:vendorExtensions>
26  </spirit:componentInstance>
```

I.3 Description du logiciel

I.3.1 Fichier component du processeur

```
1  <space:supportedSW>
2    <space:supportedOS>
3      <space:name>standalone</space:name>
4      <space:version>1.00.a</space:version>
5      <space:supportedComponents>
6        <space:supportedComponent spirit:vendor="spacecodesign.com"
             spirit:library="base" spirit:name="SpaceBaseModule"
             spirit:version="0.2" />
7      </space:supportedComponents>
8    </space:supportedOS>
9  </space:supportedSW>
```

I.3.2 Instance de processeur dans un design

```
<spirit:componentInstance>
  <spirit:instanceName>uBlaze</spirit:instanceName>
  <spirit:componentRef spirit:library="ISS" spirit:name="uBlaze"
      spirit:vendor="spacecodesign.com" spirit:version="0.1"/>
  <spirit:configurableElementValues>
    <spirit:configurableElementValue spirit:referenceId="
        ctor_name_id">uBlaze</spirit:configurableElementValue>
    <spirit:configurableElementValue spirit:referenceId="
        zBinaryFileNameID">uBlaze.mb.elf</
        spirit:configurableElementValue>
    <spirit:configurableElementValue spirit:referenceId="
        DebuggerPortID">0</spirit:configurableElementValue>
    <spirit:configurableElementValue spirit:referenceId="
        InstructionLMBSizeID">0x10000</
        spirit:configurableElementValue>
    <spirit:configurableElementValue spirit:referenceId="
        DataLMBSizeID">0x10000</spirit:configurableElementValue>
    <spirit:configurableElementValue spirit:referenceId="verboseID">
        false</spirit:configurableElementValue>
    <spirit:configurableElementValue spirit:referenceId="
        moduleStackSizeID">4096</spirit:configurableElementValue>
    <spirit:configurableElementValue spirit:referenceId="heapSizeID"
        >0</spirit:configurableElementValue>
    <spirit:configurableElementValue spirit:referenceId="QLengthID">
        100</spirit:configurableElementValue>
    <spirit:configurableElementValue spirit:referenceId="QValueID">5
        </spirit:configurableElementValue>
  </spirit:configurableElementValues>
  <spirit:vendorExtensions>
    <space:componentInstanceExtensions>
      <space:hierarchicalComponentRef space:subComponent="uBlaze"
          space:superComponent="uBSC"/>
```

```
19      <space:swInstances>
20        <spirit:componentInstance>
21          <spirit:instanceName>my_ObjectDetector</
                spirit:instanceName>
22          <spirit:componentRef spirit:library="base" spirit:name="
                ObjectDetector" spirit:vendor="spacecodesign.com"
                spirit:version="0.2"/>
23          <spirit:configurableElementValues>
24            <spirit:configurableElementValue spirit:referenceId="
                  ctor_name_id">my_ObjectDetector</
                  spirit:configurableElementValue>
25            <spirit:configurableElementValue spirit:referenceId="
                  dClock_periodID">40</spirit:configurableElementValue>
26            <spirit:configurableElementValue spirit:referenceId="
                  UnitID">SC_NS</spirit:configurableElementValue>
27            <spirit:configurableElementValue spirit:referenceId="
                  ucIDID">1</spirit:configurableElementValue>
28            <spirit:configurableElementValue spirit:referenceId="
                  ucAccessPriorityID">1</
                  spirit:configurableElementValue>
29          </spirit:configurableElementValues>
30          <spirit:vendorExtensions>
31            <space:componentInstanceExtensions>
32              <space:paramDependencies>
33              <space:paramDependency space:parameterID="
                    dClock_periodID">Clock1</space:paramDependency>
34              <space:paramDependency space:parameterID="UnitID">
                    Clock1</space:paramDependency>
35              </space:paramDependencies>
36            </space:componentInstanceExtensions>
37          </spirit:vendorExtensions>
38        </spirit:componentInstance>
39        <spirit:componentInstance>
40          <spirit:instanceName>my_LRCTRL</spirit:instanceName>
```

```
41          <spirit:componentRef spirit:library="base" spirit:name="
                LRCTRL" spirit:vendor="spacecodesign.com"
                spirit:version="0.2"/>
42          <spirit:configurableElementValues>
43            <spirit:configurableElementValue spirit:referenceId="
                  ctor_name_id">my_LRCTRL</
                  spirit:configurableElementValue>
44            <spirit:configurableElementValue spirit:referenceId="
                  dClock_periodID">40</spirit:configurableElementValue>
45            <spirit:configurableElementValue spirit:referenceId="
                  UnitID">SC_NS</spirit:configurableElementValue>
46            <spirit:configurableElementValue spirit:referenceId="
                  ucIDID">4</spirit:configurableElementValue>
47            <spirit:configurableElementValue spirit:referenceId="
                  ucAccessPriorityID">4</
                  spirit:configurableElementValue>
48          </spirit:configurableElementValues>
49          <spirit:vendorExtensions>
50            <space:componentInstanceExtensions>
51              <space:paramDependencies>
52              <space:paramDependency space:parameterID="
                    dClock_periodID">Clock1</space:paramDependency>
53              <space:paramDependency space:parameterID="UnitID">
                    Clock1</space:paramDependency>
54              </space:paramDependencies>
55            </space:componentInstanceExtensions>
56          </spirit:vendorExtensions>
57        </spirit:componentInstance>
58        <spirit:componentInstance>
59          <spirit:instanceName>my_ImgAnalizer</spirit:instanceName>
60          <spirit:componentRef spirit:library="base" spirit:name="
                ImgAnalizer" spirit:vendor="spacecodesign.com"
                spirit:version="0.2"/>
61          <spirit:configurableElementValues>
```

```
62              <spirit:configurableElementValue spirit:referenceId="
                    ctor_name_id">my_ImgAnalizer</
                    spirit:configurableElementValue>
63              <spirit:configurableElementValue spirit:referenceId="
                    dClock_periodID">40</spirit:configurableElementValue>
64              <spirit:configurableElementValue spirit:referenceId="
                    UnitID">SC_NS</spirit:configurableElementValue>
65              <spirit:configurableElementValue spirit:referenceId="
                    ucIDID">5</spirit:configurableElementValue>
66              <spirit:configurableElementValue spirit:referenceId="
                    ucAccessPriorityID">5</
                    spirit:configurableElementValue>
67            </spirit:configurableElementValues>
68            <spirit:vendorExtensions>
69              <space:componentInstanceExtensions>
70                <space:paramDependencies>
71                <space:paramDependency space:parameterID="
                      dClock_periodID">Clock1</space:paramDependency>
72                <space:paramDependency space:parameterID="UnitID">
                      Clock1</space:paramDependency>
73                </space:paramDependencies>
74              </space:componentInstanceExtensions>
75            </spirit:vendorExtensions>
76          </spirit:componentInstance>
77        </space:swInstances>
78        <space:globalSourceFiles>
79          <spirit:file>
80            <spirit:name>uBlaze/main_uBlaze.cpp</spirit:name>
81            <spirit:fileType>systemCSource</spirit:fileType>
82            <spirit:isIncludeFile>false</spirit:isIncludeFile>
83          </spirit:file>
84        </space:globalSourceFiles>
85      </space:componentInstanceExtensions>
86    </spirit:vendorExtensions>
```

```
87 </spirit:componentInstance>
```

I.4 Liens directs dans un design

instance1, *instance2* et *spaceDirectLink* sont des instances de composants du design dans lequel se trouvent ces déclarations.

```
1  <space:directConnections>
2    <space:directConnection>
3      <space:name>directLink</space:name>
4      <space:componentInstances>
5        <spirit:instanceName>instance1</spirit:instanceName>
6        <spirit:instanceName>instance2</spirit:instanceName>
7      </space:componentInstances>
8      <space:channel>spaceDirectLink</space:channel>
9    </space:directConnection>
10 </space:directConnections>
```

I.5 Instance de composant hiérarchique étendue dans un design

I.5.1 Extensions globales du design

```
1  <spirit:vendorExtensions>
2    <space:designExtensions>
3      <space:superInstances>
4      <spirit:componentInstance>
5        <spirit:instanceName>uBSC</spirit:instanceName>
6        <spirit:componentRef spirit:library="iss" spirit:name="
             uBlazeSuperComp" spirit:vendor="spacecodesign.com"
             spirit:version="0.2"/>
7      </spirit:componentInstance>
8      </space:superInstances>
```

```
9    </space:designExtensions>
10 </spirit:vendorExtensions>
```

I.5.2 Extensions d'une instance de sous-composant

```
1  <spirit:vendorExtensions>
2    <space:componentInstanceExtensions>
3      <space:hierarchicalComponentRef space:subComponent="OPBTimer"
           space:superComponent="uBSC" />
4    </space:componentInstanceExtensions>
5  </spirit:vendorExtensions>
```

I.6 Description du bas niveau

I.6.1 Extensions des IP

Aucun modèle de FPGA et version de XPS n'est spécifiée, l'IP est donc utilisable pour tout FPGA et toute version de XPS.

```
1  <spirit:vendorExtensions>
2    <space:componentExtensions>
3      <space:componentType>functional</space:componentType>
4      <space:IPDefinition>
5        <space:mssInfo>
6          <space:name>intc</space:name>
7          <space:version>1.00.c</space:version>
8        </space:mssInfo>
9      </space:IPDefinition>
10   </space:componentExtensions>
11 </spirit:vendorExtensions>
```

I.6.2 Définition des cartes

I.6.2.1 Définition d'un port

```
<spirit:port>
  <spirit:name>CONN_BANK0_WEND_Z</spirit:name>
  <spirit:wire>
    <spirit:direction>in</spirit:direction>
    <spirit:vector>
      <spirit:left>3</spirit:left>
      <spirit:right>0</spirit:right>
    </spirit:vector>
  </spirit:wire>
  <spirit:vendorExtensions>
    <space:portExtensions>
      <space:ucfStrings>
        <space:ucfString space:index="0">LOC=G29</space:ucfString>
        <space:ucfString space:index="0">FAST</space:ucfString>
        <space:ucfString space:index="1">LOC=F29</space:ucfString>
        <space:ucfString space:index="1">FAST</space:ucfString>
        <space:ucfString space:index="2">LOC=H24</space:ucfString>
        <space:ucfString space:index="2">FAST</space:ucfString>
        <space:ucfString space:index="3">LOC=J24</space:ucfString>
        <space:ucfString space:index="3">FAST</space:ucfString>
      </space:ucfStrings>
    </space:portExtensions>
  </spirit:vendorExtensions>
</spirit:port>
```

I.6.2.2 Extensions globales du composant *carte*

```
<spirit:vendorExtensions>
        <space:componentExtensions>
    <space:componentType>board</space:componentType>
```

```
4     <space:description>Xilinx Virtex-II Multimedia FF896 Development
          Board </space:description>
5     <space:fpga>
6       <space:fpgaRef space:family="virtex2" space:device="XC2V2000"
            space:package="FF896" space:speedgrade="-4" />
7       <space:jtagPosition>2</space:jtagPosition>
8     </space:fpga>
9         </space:componentExtensions>
10 </spirit:vendorExtensions>
```

ANNEXE II

TYPES DE COMPOSANTS

Cette annexe récapitule les types de composants qui ont été définis lors de l'élaboration de la plateforme. Le type est mentionné dans les extensions du fichier *component*, dans une balise *space:componentType*. La liste qui suit est hiérarchisée en fonction de la distinction type/sous-type.

1. *functional* : C'est le type par défaut. Les composants fonctionnels possèdent des interfaces directes.
   (a) *processor* : Les processeurs sont des composants spéciaux. Ils décrivent les types de composants qu'ils peuvent supporter dans leurs extensions. Leurs instances possèdent une collection interne regroupant les composants qu'ils exécutent.
   (b) *device* : Composants sur une carte.
2. *adapter* : Les adaptateurs peuvent posséder des interfaces directes et miroir.
3. *channel* : Représente tout type de canal de communication. Possède des interfaces miroir seulement.
   (a) *directLink* : Type spécial pour les liens directs.
4. *bridge* : Les ponts sont destinés à relier des canaux. Ils possèdent des interfaces directes.
5. *board* : Carte. Ce sont des composants hiérarchiques utilisés pour la génération sur FPGA. Leurs sous-modules sont des *devices*.
6. *virtual* : Type ne pouvant pas être instancié. Utilisé pour grouper les composants dérivant d'un même module virtuel.

ANNEXE III

FICHIERS RENCONTRÉS POUR LA GÉNÉRATION DE FICHIERS BAS NIVEAU

Cette annexe introduit différents fichiers rencontrés lors de la génération de projets XPS. Un projet XPS est consitué de plusieurs fichiers obligatoires pour qu'il puisse être chargé et que ses outils puissent fonctionner pour créer la plateforme sur FPGA.

III.1 Fichiers du projet à générer

Fichier XMP *(Xilinx Microprocessor Project)* Il s'agit du fichier principal du projet. Il contient essentiellement les emplacements des autres fichiers du projets. Il décrit aussi le logiciel, en indiquant l'emplacement des fichiers source, les options du compilateur, l'emplacement du linker script et de l'exécutable.

Fichier MHS *(Microprocessor Hardware Specification)* Ce fichier correspond au fichier *design* de IP-XACT. Il contient une liste d'instances de composants *(IP)*. Les types composants à utiliser sont identifiés par leur nom et leur version. Pour chaque instance, on définit les valeurs de ses paramètres et les connexions de ses ports et interfaces. La structure est donc différente de IP-XACT, pour lequel la déclaration des instances et

de leurs paramètres est distincte de celle des connexions. Il est possible de réaliser des connexions port à port. À ce niveau, il n'y a pas de port transctionnel. Pour décrire la connexion d'un port, on l'associe à signal. Tous les ports associés à un même signal sont connectés ensemble. On peut également connecter des interfaces. Ce type de connexion se divise en deux catégories :

1. « Interface-à-bus ». Il s'agit du genre de connexion habituel. Les interfaces impliquées sont de type *maître*, *esclave*, *maître_esclave* ou *moniteur*. Dans la déclaration de l'instance de composant à connecter au canal, l'interface à connecter est associée à une instance de bus est connectée.
2. Connexions « point-à-point » (selon la terminologie Xilinx). Une connexion point-à-point implique une interface *initiateur* et une interface *cible*. Pour déclarer une connexion point-à-point, on associe à chacune des deux interfaces un même identifiant. La déclaration est donc semblable à celle des connexions port-à-port où l'identifiant associé à chaque port correspond à un signal. Dans les connexions point-à-point, il s'agit d'un groupe de signaux. Elles sont utilisées pour réaliser des connexions impliquant deux composants qui ne sont pas des canaux. C'est en particulier le cas pour les connexions entre les adaptateurs et les composants fonctionnels.

Les fichiers MHS possèdent aussi des ports globaux. Ces ports pourront être utilisés pour communiquer avec des composants situés à l'extérieur du FPGA. L'architecture des composants sur le FPGA correspond donc à un composant hiérarchique. À la différence de IP-XACT, les ports du super-composant (ports globaux du MHS) et son design intérieur se situent dans un même fichier.

Fichier UCF *(User Constraints File)* Ce fichier associe aux ports globaux du MHS présentés précédemment des broches du FPGA. Il est donc nécessaire de connaître la manière dont les composants de la carte sont connectés au FPGA pour pouvoir configurer correctement ce fichier.

Fichier MSS (Microprocessor Software Specification) Ce fichier contient des informations spécifiques au logiciel exécuté sur les processeurs du design. Il indique les compilateur est le sytème d'exploitation pour chaque processeur. Chaque composant est également associé à un pilote *(driver)*.

Linker script Il ne s'agit pas de fichier spécifique à Xilinx, mais à l'éditeur de liens utilisé (ld). Il en existe un pour chaque instance de processeur. Il permet de définir l'emplacement des différentes sections de l'exécutable.

Fichier *etc/download.cmd* Ce fichier contient la commande à exécuter pour charger le fichier de programmation du FPGA (bitstream) sur le FPGA. Sa génération dépend de la position du FPGA dans la chaîne des périphériques JTAG (Joint Test Action Group).

III.2 Description des composants

Fichiers MPD Ces fichiers correspondent aux fichiers *component* de IP-XACT. Ils décrivent les paramètres, ports et interfaces pour chaque type de composant. La plupart des fichiers MPD utilisés dans un projet ne se situent pas à l'intérieur de celui-ci, mais appartiennent à la bibliothèque de composants Xilinx. Ces fichiers-là ne sont bien sûr pas générés par la plateforme. En revanche, il est possible d'ajouter des composants au projet lui-même. Leurs descriptions se trouvent alors dans le sous-dossier *pcores*.

Fichiers PAO *(Peripheral Analyze Order)* Ils accompagnent les fichiers MPD. Leur rôle est d'indiquer les fichiers à utiliser pour la simulation et la synthèse du composant, ainsi que l'ordre dans lequel ils doivent être compilés.

III.3 Fichiers XBD *(Xilinx Board Definition)*

Ces fichiers se trouvent dans la bibliothèque Xilinx. Leur rôle est la description de cartes. Chaque fichier contient une liste d'interfaces. Généralement, chacune d'entre elles correspond à un composant périphérique sur la carte. Cependant, il n'est pas impossible qu'un tel composant possède plusieurs interfaces. La description de carte possède de plus une section *fpga*, qui fait le lien entre les ports de toutes ses interfaces et les broches du FPGA. Cette information est donc intéressante pour la génération du fichier UCF du projet.

ANNEXE IV

COMMANDES DE LA PLATEFORME

abstractconnec *inst* affiche les connexions abstraites (c.-à-d. sans adaptateurs) de l'instance de composant *inst*

bind *ic1 ic2* connecte les instances de composant *ic1* et *ic2*

bind *ic1 port1 ic2 port2* connecte le port *port1* de *ic1* au port *port2* de*ic2*

bind *ic1 if1 ic2 if2* connecte l'inhterface *if1* de *ic1* à l'interface *if2* de *ic2*

binding affiche les déclarations de connexions SystemC

callgen lance le processus de génération pour le design courant

cd *rep* définit *rep* comme répertoire courant

cd définit le dossier utilisateur comme répertoire courant

chglevel *niveau nom* raffine le design courant vers le niveau indiqué, en indiquant le nom du nouveau design

configure *inst* configure l'instance de composant *inst* en mode interactif

contract *inst* contracte l'instance de composant hiérarchique *inst*

compinfo *vendor library name version* affiche des informations sur le composant identifié par son VLNV

copyconf *nom* copie le design (configuration) courant

expand inst étend l'instance de composant hiérarchique *inst*

genx *VLNVCarte rep* effectue le raffinement bas niveau du design courant, pour la carte identifiée par *VLNVCarte*, et dans le répertoire *rep*

help affiche l'aide

instdeclar affiche les déclarations des instances de composant (avec leurs paramètres)

loaddesign ***fichier*** charge le design situé dans *fichier*

loadlib charge la bibliothèque de composants

ls liste les fichiers du répertoire courant

lsconf liste les designs (configurations) du projet courant

lscomp liste les composants disponibles

move2sw ***inst proc*** déplace l'instance de composant *inst* vers une implémentation logicielle dans l'instance de processeur *proc*

newconf ***nom*** crée un nouveau design (configuration)

newinst ***vendor library name version [nomInst]*** crée une nouvelle instance, éventuellement en précisant son nom. S'il est absent, son nom sera généré.

newproject ***rep*** crée un nouveau projet dans le répertoire *rep*. S'il existe déjà, il est chargé.

newusermodule ***nom*** crée un nouveau composant utilisateur

pwd affiche le chemin du répertoire courant

quit termine le programme

rminst ***nomInst*** enlève une instance du design courant. Supprime aussi ses connexions.

run ***fichier*** exécute le script situé dans *fichier*

saveproject enregistre le projet courant

selectconf ***conf*** sélectionne le design (configuration) *conf* parmi ceux du projet courant

setparam ***nomInst IDParam valeur*** définit la valeur d'un paramètre sur une instance de composant

spiritdesign ***[fich]*** enregistre le design courant dans le fichier *fich*. S'il est absent, son contenu est affiché en console.

ANNEXE V

ASSISTED CREATION AND REFINEMENT OF TRANSACTIONAL LEVEL SPECIFICATIONS BASED ON IP-XACT

IP 08

Session : Standard & Integration

ASSISTED CREATION AND REFINEMENT OF TRANSACTIONAL LEVEL SPECIFICATIONS BASED ON IP-XACT

Nicolas Laug, Guy Bois, Marc-André Cantin
École Polytechnique
Microelectronics research group
Montréal, Québec, Canada
{laug, bois, cantin}@grm.polymtl.ca

Abstract

High abstraction levels, design automation and reuse are currently the three main axes to solve the productivity gap issue. The new Spirit consortium's IP-XACT v1.4 standard provides normalized file formats to describe transactional and RTL (Register Transfer Level) components. Since information on connecting and configuring designs is given, the combination of reuse and ESL is now enabled. This paper describes a high level platform prototype based on IP-XACT. It supports SystemC models and simulations; and also provides a set of tools for fast design generation and simple user experience. The platform has been successfully implemented and used to generate a system including a processor in an FPGA.

1. Introduction

In electronic design, the pressure of Moore's law and time-to-market challenge has lead to a great need of productivity in the system-on-chip industry. The difference between the increase of chip complexity and the growth of productity is often referred as the *productivity gap*. Because of it, the design cost of system on chips is constantly increasing. To solve that issue, several responses have emerged.

First, Electronic System Level (ESL) [1] focuses on high levels of abstraction. A model of the system can be simulated in the early stages of the design flow, even before hardware-software partitioning. That approach has several advantages. First, it helps to create executable functional specification. Then fast architecture exploration can be done, including hardware/software partitioning. Finally, high-level models can be also used for verification of RTL components [2, 3].

Secondly, productivity can be increased thanks to component reuse, which avoids reinventing the wheel for every system, using reliable modules. Most reuse efforts focus on standard interface definitions. At high levels, OSCI (Open SytemC Initiative) TLM-2.0 (Transaction Level Modeling) defines a SystemC standard for transactional modeling. At low-level, wrapper libraries (for instance OCP — Open Core Protocol [4]) allow the user to create bus interface neutral cores, which can be connected to buses using adapters. With those standards one can create components once for all, using adapters when necessary. However, managing large libraries of components inevitably needs to support multiple types of interfaces. Therefore, a standard format that documents component interfaces is necessary. That is precisely one of the goals of Spirit consortium's IP-XACT [5]. It supports ESL components, including transactional ports since its latest 1.4 version.

Finally, it is essential to provide tools that help architecture engineers to modify and refine their system simply and rapidly. Indeed, modifying a design inevitably includes a part of tedious work. Automating that work wherever it is possible would improve usability of high-level modeling, fostering its adoption.

This paper proposes principles and an implementation of a platform that combines those three features. It manages a library of high-level components, which can be used to build the first model. Then at each refinement step, library components are replaced by more accurate modules, down to a physical implementation. At every level, the user has access to tools to edit the model in a very simple manner, without worrying about implementation details.

High abstractions levels are enabled by supporting functional descriptions and transactional ports, as SystemC [6] does. IP-XACT files store information on how to parameter and connect components. More precisely, we use four of the seven XML (Extensible Markup Language) file types defined by the standard:

A bus definition contains general information on an interface type, such as whether it is addressable or direct (in this case, direct master-slave con-

nections are allowed).

Abstraction definitions describe the ports used to implement a bus interface at a given abstraction level.

A component has bus interfaces, ports and parameters. Bus interfaces on a component are defined by their abstraction definition and the interface type (e.g. master or slave).

A design describes a set of instances of components. Values are given to their parameters, and connections are defined, whether they connect simple ports (ad hoc connections) or whole bus interfaces (interconnections).

Without excluding other technologies, the program benefits from Space platform [7]. Indeed, it provides all the features required by this application. A library of SystemC modules at various abstraction levels is available. Modules at the lowest levels can be mapped to hardware cores. Additionally, Space supports software implementation of application modules.

This paper has the following structure: Section 2 presents how and why IP-XACT needs to be extended to create the platform. Then, Section 3 gives an overview of the most important helper tools, and examples of such extensions. Section 4 details which information is needed to generate implementations on Field Programmable Gate Array (FPGA) boards, and finally Section 5 shows an example of use of the platform. Section 6 presents the conclusion.

2. IP-XACT as basis

2.1. Vendor extensions

Tools we developed all need some specific information which is missing in common IP-XACT files. Fortunately, the standard defines special sections of files called *vendor extensions*. While the rest of documents must respect a strict syntax specified by an XML schema, vendor extensions can contain any information.

To be able to validate input files, we have defined an XML schema for extensions we added. Other IP-XACT compliant programs will simply ignore them, so interoperability is not reduced. Reciprocally, our extensions are not mandatory, so any IP-XACT valid file can be processed by our platform. However, some tools will not work if their extensions are missing.

Sections 3 and 4 show the main vendor extensions used by the platform.

2.2. Generators

Some components may need to be managed in a particular manner. For instance, a component may need to import the value of a parameter from an other component; the files used to build software must be generated depending on the processor type; etc. To support those special cases, components may refer to *generators*. They are executable programs called by the design environment, able to interact with the current design.

IP-XACT defines a communication interface between the design environment and generators: the Tight Generator Interface (TGI). It is a set of functions implemented by the design environment and called by generators. The communication is made according to the SOAP standard [8]. The latter, mostly known in the context of web services, defines XML-based message passing used for remote procedure calls in heterogeneous environments. Since the interface is completely neutral, generators using the TGI do not depend on design environment implementation. As well as the description file format, the TGI may appear insufficient in some situations, so it has also been extended.

Since the platform is implemented in Java, it can invoke Java methods of generators, giving reference to objects it uses. Therefore, they have a broad access to the platform's inner behavior. Such generators are located in JAR files in the component library, keeping them in the outside of the platform. The major drawback is that independence of generators with design environment is lost. That is why that type of generator should be used only when the TGI cannot realize the required functionality. There are two types of such situations:

1) Some generators require access to special, non-standard information. In particular, processor generators are responsible for creating files to compile its software (detailed in Section 3.2). And they must also generate appropriate connections to implement point-to-point links (detailed in Section 3.3) involving components it executes.

2) The need to define several roles for generators, such that of them is called at a specific moment. Five types have been defined: *User component generators* create IP-XACT descriptions of user components, with possibly source file stubs. *Post-instantiation generators* are called just after an instance of a component has been added to a design. They are mostly used to operate dynamic changes, using the listener design pattern to obtain instant reaction to user operations. *Pre-transformation generators* are able to select which type of component must replace a component in a refinement process, whereas *post-transformation generators* are called just after a refinement. They can modify the new design model from characteristics of the original one

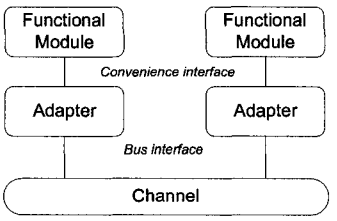


Figure 1. Layered communication model

(cf. Section 3.4). At last, *normal generators* are called during the generation step, as defined in the IP-XACT standard. This is the default type. Standard IP-XACT generators (using the TGI) are run at this moment.

Just like vendor extensions, non-standard generators are simply ignored by third party design environments, so the component description remains usable. The difference is that the additional component features will not be enabled.

3. Necessary features

3.1. Adapter generation

As explained by Gajski and Kai [9], TLM should be implemented focusing on a separation between communication and computation concerns. Thus, adapters are needed between processing elements and communication channels, leading to a layered model (Figure 1), as described by [10, 11]. It is important to note that those adapters are just the *glue* to make the simulation work; designers should make abstraction of such components.

To generate adapters in a design, the platform needs to split components in three categories: channels, adapters and functional component. The type of a component is defined in its vendor extensions, and the default is functional.

When the user asks for a connection between a functional component and a channel, the platform first checks if those components have compatible interfaces. If not, a chain of adapters from the library is built between the components. In most cases, one or two adapters are sufficient.

For instance, the need of two adapters has been encountered in the following situation: 1) using the Space environment, a module adapter converts the Space convenience interface into a common bus interface; 2) a second layer of adapters connects it to a bus model (such as OPB). A third adapter may be necessary to simulate an RTL module in a TLM environment [12].

The principles of the method are shown in Algorithm 1. Adapters have two sides: the first one has

Algorithm 1: Search of adapter chain principle

Data: adapters library, maxAdapters, components startComponent and endComponent
Result: chain of adapters to use, or null if not found
1 **begin**

software tasks, run by one or more processor. It is a different point of view than the IP-XACT one, in which partitioning is considered to be fully defined: the standard let only describe which executable image is run by a processor, as well as how to build it (source files, compiler and linker). Using extensions, it is possible to introduce our representation of software in description files.

First, the processor's component file specifies which type of components it can execute. They should be base components of application modules. Second, on the design file, each processor instance lists the software component instances it executes in its vendor extensions. The syntax of their description is the standard one, normally used for hardware instances. Therefore, it is very easy to bring them back to a hardware model. The executable model run by the processor is built by its generators.

3.3. Point-to-point communication

Functional modules could require to be connected directly, for performance purposes. Point-to-point connections or FIFO (First In, First Out) connections are examples of dedicated communications links. This is particularly useful in data flow applications, in which buses can be bottlenecks. Therefore, we define a dedicated communication link as a subtype of a channel. The tool looks for such links to directly connect two components. If none are found, Algorithm 1 is used to insert the proper adapters in order to make the connection possible. Direct links can also be useful for defining hardware accelerators or coprocessors. For instance, Xilinx MicroBlaze softcore processor supports Fast Simplex Links, which are FIFOs for which control is hooked in the processor's pipeline. Through IP-XACT generators, our methodology supports creating a link between a piece of software running on a MicroBlaze and a hardware accelerator. However, using different processors would lead to different implementations of the generators.

3.4. Refinement

In an ESL based design flow, refinement steps are part of the important milestones. To reach the goal of implementing a user-friendly environment, assistance must be provided for that operation. Basically, the principle is to replace functional components and channels by equivalent more precise models. The generated architecture is similar, and some parameter values are imported.

It is clear that a mechanism to find equivalent components is necessary. The chosen solution is adding an inheritance relationship between component classes, just like in object-oriented programming. Thus, it is possible to group components of

Algorithm 2: Refinement method overview

1 Conn ← set of abstract connections, i.e. making abstraction of adapters
foreach *non-adapter component* c

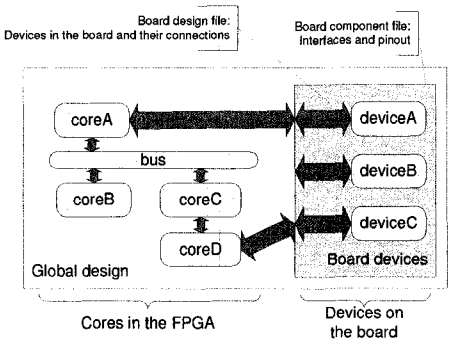


Figure 2. Board representation

tain types of FPGA might be available. In this case, the list of supported FPGAs is added to the component definition file. When selecting appropriate equivalent components, the refinement algorithm will exclude incompatible components (line 2 of Algorithm 2).

5. Application to a real case

5.1. Implementation technologies

For high-level modeling, the implementation is based on SystemC [6] as the system level design language. That means that components use SystemC and the platform generates SystemC files to simulate the design. As mentioned in the introduction, modules come from the library of the Space platform.

FPGA refinement generates a Xilinx Platform Studio [13] project. It uses IP cores provided by Xilinx, as well as Space components. Since Xilinx has it own description format (MPD for cores and XBD for boards), external tools have been created to build IP-XACT file stubs from those files, making integration of new cores less tedious.

5.2. Design example

The platform has been tested using the case study of a land rover guiding system [7]. The purpose of this section is both validating its functionality and giving an overview of the user interface. We have followed a top-down approach, starting from an untimed model down to an FPGA implementation. All user actions are made through a console with simple commands, hiding the internal complexity.

The first step is the creation of the application processing elements, represented as user components:

```
newusermodule ObjectDetector
```

That command creates the IP-XACT file corresponding to the new component, as well as SystemC file stubs. To achieve this, the design environment collaborates with the generator of the base component. The same action is repeated for every application module, then the user implements the functionality in the generated files.

After that, an empty design is created, specifying the name of the abstraction level (here Elix, a timed/untimed communication level [7]). Components can now be instantiated in it.

```
newconf elix confElix
newinst spacecodesign.com base
    ObjectDetector 0.2 myObjectDetector
[other user components]
newinst spacecodesign.com channel
    UTFChannel 0.2 channel
newinst spacecodesign.com serial
    SpaceUART 0.1 uart
```

Automatic operations are transparently performed here. For instance, the user components have parameters corresponding to the clock period, as specified in their vendor extensions. Therefore, a new clock instance is automatically generated, and the parameter values are imported from the clock. If the clock period is changed, dependent parameters are automatically updated.

Then the connections are made. For instance:

```
bind myObjectDetector channel
```

As seen in Section 3.1, adapters are automatically added to the design and connected to the functional components and the channel. Of course, adapters instantiated in this way can benefit from the same mechanisms as in manual instantiation.

The generation of files builds source files and a makefile to be used to compile a simulation of the system. The IP-XACT file describing the current design is also created.

The refinement to the lower level (here Simtek, a cycle accurate computation model [9]) can now be executed:

```
chglevel simtek confSimtek
```

By calling that command, user components — which are allowed at every transactional level — are simply copied to the new design. On the other hand, since untimed channels are not allowed in Simtek configuration, the platform looks for equivalent allowed components. The OPB bus satisfies those requirements, so it is instantiated to replace the untimed channel. Connections to the components are generated again (cf. Section 3.4); this time two layers of adapters are necessary.

Simtek is the level in the Space technology where hardware-software partitioning can be done, thus it is possible to add a MicroBlaze simulator in the design. The MicroBlaze is available in the library as a hierarchical component containing the processor itself, and other peripherals like an interrupt controller, a timer, etc. All of them are instantiated together. Since the

MicroBlaze description specifies that it can execute application modules, the user components we created can be moved in the software domain. That operation is done in a single line. For instance:

```
set2sw myObjectDetector uBlaze
```

Files generated for that configuration include the same than the previous one, plus files created by the MicroBlaze's generator, to build the executable it runs.

Finally, the design is exported to an XPS project, for an FPGA implementation (Section 4). The board identifiers are given in the command line:

```
genx xilinx.com board V2Multimedia 1 outDir
```

Components of the previous design become either by IP cores or board devices. A Digital Clock Manager (DCM) is added to generate a clock signal with the period specified in the original configuration from one provided by the board. Location of the software's memory sections can be selected, and the linker script is generated accordingly. For this design, we chose to store instruction and data sections in an external RAM chip. Memory controllers are also automatically instantiated and connected to on-chip buses. The only manual task is implementing hardware application components in VHDL file stubs. All other files files required in a Xilinx Platform Studio project are generated, so when XPS is launched, it is ready for synthesis of the FPGA bitstream.

The system has been successfully synthesized and tested on the Xilinx multimedia development board. It uses 3763 slices (34%) of the board's Virtex II FPGA, and it can run at a maximum frequency of 61.909MHz.

6. Conclusion

Version 1.4 of IP-XACT is a strong basis for IP description exchange, in high and low abstraction levels. However, to build a powerful and easy to use design platform, tools must be created, and those tools need additional information. Thanks to the vendor extensions mechanism, that data can be added in description files, while keeping our platform compatible with IP-XACT, whether it be for importing or exporting description files.

From that format, we have successfully developed the core of a design platform able to quickly create FPGA-based systems, following a top-down methodology. To build a complete design environment, a graphical user interface may be developed upon it.

Acknowledgments

The authors would like to acknowledge the SpaceCodesign team, in particular Benoit Pilote for his deep explanations of SpaceStudio requirements. They also want to thank NSERC for their monetary support and CMC Microsystems for their technical support.

References

[1] B. Bailey, G. E. Martin, and A. Piziali. *ESL Design and Verification: A Prescription for Electronic System-Level*. The Morgan Kaufmann series in systems on silicon. Morgan Kaufmann, Amsterdam, Netherlands, 2007.

[2] F. Ghenassia and A. Clouard. TLM: An overview and brief history. In F. Ghenassia, editor, *Transaction Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*, chapter 1, pages 1–22. Springer, 2005.

[3] S. Abdi and D. Gajski. Model validation for mapping specification behaviors to processing elements. *High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International*, pages 101–106, Nov. 2004.

[4] OCP International Partnership. *Open Core Protocol specification version 2.2*, 2006. `http://www.ocpip.org/`.

[5] The SPIRIT consortium. IP-XACT v1.4: A specification for XML meta-data and tool interfaces, Mar. 2008. `http://www.spiritconsortium.org/tech/docs/`.

[6] Open SystemC Initiative. SystemC. `http://www.systemc.org/`.

[7] L. Filion, M.-A. Cantin, L. Moss, E. M. Aboulhamid, and G. Bois. Space codesign: A SystemC framework for fast exploration of hardware/software systems. In *Design & Verification Conference and Exhibition (DVCON'07)*, San Jose, CA, 2007.

[8] W3C. *SOAP Version 1.2 specification*, Apr. 2007. `http://www.w3.org/TR/soap12`.

[9] L. Cai and D. Gajski. Transaction level modeling: an overview. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24, New York, NY, USA, 2003. ACM.

[10] L. Maillet-Contoz and J.-P. Strassen. TLM modeling techniques. In F. Ghenassia, editor, *Transaction Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*, chapter 3, pages 57–94. Springer, 2005.

[11] A. Rose, S. Swan, J. Pierce, and J.-M. Fernandez. Transaction Level Modeling in SystemC. `http://www.systemc.org/`.

[12] L. Moss, M.-A. Cantin, G. Bois, and E. M. Aboulhamid. Automation of communication refinement and hardware synthesis within a system-level design methodology. *Rapid System Prototyping, 2008. RSP '08. The 19th IEEE/IFIP International Symposium on*, pages 75–81, June 2008.

[13] Xilinx. Platform studio. `http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm`.