TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES EXTRAITS DE CODE

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

API	Interface de programmation, de l'anglais « Application Programming Interface »
BNF	« Backus-Naur Form »
CDR	« Common Data Representation »
COM	« Component Object Model »
CORBA	« Common Object Request Broker Architecture »
FPGA	Réseau prédiffusé programmable par l'utilisateur, de l'anglais « Field Programmable Gate Array »
FLI	« Foreign Language Interface »
GIOP	« General Inter-ORB Protocol »
HIL	« Hardware-in-the-loop »
HLA	« High Level Architecture »
ICO	« Integrated Circuit ORB »
IDL	Langage de définition d'interface, de l'anglais « Interface Definition Language »
IIOP	« Internet Inter-ORB Protocol »
IPC	Communication inter processus, de l'anglais « Inter Process Communication »
JRMP	« Java Remote Method Protocol »
LACIME	Laboratoire de Communication et d'Intégration de MicroÉlectronique
MIDL	Langage de définition d'interface de Microsoft, de l'anglais « Microsoft Interface Definition Language »
NDR	« Network Data Representation »
OMA	« Object Management Architecture »

OMG	« Object Management Group »
ORB	« Object Request Broker »
ORPC	« Object RPC »
OSCI	« Open SystemC Initiative »
RPC	« Remote Procedure Call »
RMI	« Remote Method Invocation »
RTL	Niveau transfert de registres, de l'anglais « Register to Transistor Level »
SDR	Radio logicielle, de l'anglais « Software Defined Radio »
SoC	Système sur puce, de l'anglais « System on Chip »
TAO	« The ACE ORB »
TF	Modèle fonctionnel avec notion de temps, de l'anglais « Timed Functional Model »
UTF	Modèle fonctionnel sans notion de temps, de l'anglais « UnTimed Functional Model »
VHPI	« VHDL Programmable Interface »

INTRODUCTION

Au cours des dernières années, la complexité de la conception de circuits intégrés a augmentée de manière impressionnante forçant ainsi les concepteurs à introduire de l'hétérogénéité dans leur flot de conception pour tenter de réduire l'écart de productivité. Un système sur puce (SoC) est un bon exemple d'intégration complexe de composantes hétérogènes au sein d'un même système. Les composantes sont souvent conçues en fonction d'un paradigme de modélisation propre à leurs caractéristiques. Cela a pour résultat que des outils et langages spécialisés sont utilisés à diverses étapes de la conception et de la vérification. De plus, une composante peut s'exprimer à différents niveaux d'abstraction durant la conception. D'un niveau d'abstraction à l'autre, le paradigme de modélisation varie.

La vérification par simulation de tels systèmes hétérogènes implique l'exécution de modèles provenant de paradigmes différents comme un tout cohérent. Cette hétérogénéité rend la vérification de design dispendieuse. Les experts s'entendent pour dire que la vérification fonctionnelle requiert de 50% à 75% des ressources allouées à la conception d'un design (temps et effort) [10].

Les prototypes fonctionnels virtuels sont une bonne méthode pour implémenter un système hétérogène. Ils forment une spécification exécutable du système, avec un degré variable de contraintes architecturales, pouvant être utilisés pour maîtriser les algorithmes en cause. L'évaluation des performances à bas niveau d'abstraction peut se faire en intégrant du *Hardware-in-the-loop* (HIL) au prototype fonctionnel virtuel. Au sein de systèmes hétérogènes, la vérification requiert des mécanismes complexes pour permettre la communication inter composantes. Les solutions traditionnelles offertes par l'industrie se divisent essentiellement en deux possibilités. La première est de supporter, à l'interne, un nombre limité de langages de modélisation couvrant divers niveaux d'abstraction. Cette méthode manque de flexibilité et dépend du désir d'une entreprise à supporter des langages de modélisation. La deuxième, souvent appelée « couplage ad-hoc », fournit un mécanisme pour s'interfacer au simulateur, mais est habituellement une solution spécifique à un ou deux simulateurs. MathWorks *EDA Simulator Link MQ*

(autrefois appelé *Link for ModelSim*) est un exemple de « couplage ad-hoc » où seulement deux simulateurs sont supportés soit Mentor Graphics *ModelSim* et *QuestaSim*.

La contribution principale de ce mémoire est de proposer une dorsale de communication générique pour la conception et la vérification de systèmes complexes. Les différents acteurs impliqués communiquent à travers une *Common Object Request Broker Architecture* (CORBA) *i.e.* une architecture d'objets distribués commune telle que proposée par [42]. Notre travail se distingue de [40] par l'utilisation de CORBA non seulement pour observer un système, mais aussi pour interagir avec celui-ci. De plus, notre intégration au simulateur de l'OSCI ne requiert aucune modification au noyau de ce dernier.

Le premier chapitre présente le contexte de la recherche. Le deuxième chapitre définit le projet de recherche. Le chapitre 3 présente les spécifications et le choix de l'architecture de la dorsale. Le chapitre 4 traite de l'implémentation et de l'intégration de la dorsale. Le chapitre 5 montre comment vérifier un design avec cet environnement. Le chapitre 6 fait présente une étude de cas ainsi qu'une comparaison de cette dorsale avec une solution commerciale existante. Enfin, la conclusion, les améliorations à apporter et les travaux futurs sont offerts à la fin de ce mémoire.

CHAPITRE 1

CONCEPTION ET VÉRIFICATION FONCTIONNELLE DE DESIGNS EN ÉLECTRONIQUE NUMÉRIQUE

Ce chapitre décrit le contexte de la recherche en décrivant les caractéristiques des designs électroniques ainsi que les méthodes utilisées pour leur conception et leur vérification fonctionnelle. De plus, nous y énonçons la problématique, les objectifs et les hypothèses.

1.1 Historique

Les produits électroniques sont de plus en plus complexes. La plupart du temps, ils intègrent une panoplie de composantes de nature différente ou provenant de sources variées (voir la Figure 1.1 pour un exemple). Cette hétérogénéité complexifie grandement la conception et la vérification des systèmes. Ces processus font intervenir de nombreux outils étant souvent incompatibles les uns avec les autres.

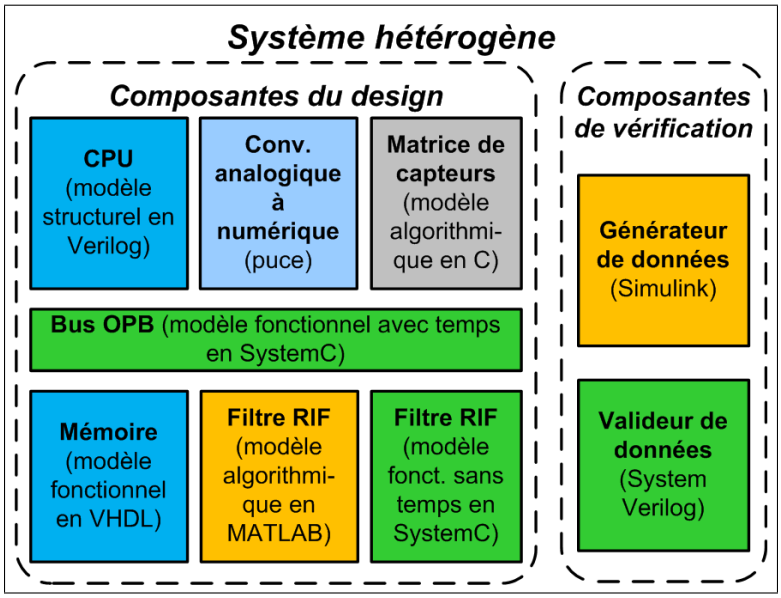

Figure 1.1 Exemple de système hétérogène en conception électronique.

Il est fort utile de pouvoir faire la vérification fonctionnelle d'un système dans son entièreté au fur et à mesure que la conception avance. C'est-à-dire de s'assurer que la logique du système effectue correctement les bonnes opérations en toutes circonstances, et ce telles qu'elles sont définies dans les spécifications du design. Nombre de problèmes font surface qu'au moment de l'intégration des différentes composantes d'un système. La vérification d'un système, comme un tout cohérent, ayant des composantes simulées dans divers simulateurs n'est pas chose facile. En effet, non seulement les simulateurs présentent habituellement leur propre interface programmable, en plus ils agissent dans un paradigme qui leur est propre. Un simulateur matériel prenant comme entrée une représentation au niveau transfert de registres d'une composante est à priori incompatible avec un simulateur algorithmique utilisant des formules mathématiques. L'abstraction et la semi-automatisation de la communication entre des simulateurs diminuent donc grandement le fardeau reposant sur les épaules des ingénieurs électriques.

Les chercheurs en vérification s'intéressent particulièrement à l'intégration d'outils de conception et de vérification provenant de sources différentes ainsi qu'aux mécanismes facilitant la réutilisation de composantes existantes. La forte compétition dans des domaines telle que la téléphonie cellulaire est au cœur de ces efforts. La fenêtre d'opportunité pour la mise en marché rapetisse forçant ainsi les concepteurs à livrer leurs produits plus rapidement pour faire face à leurs concurrents.

1.1.1 Vérification fonctionnelle en traitement de signal

Soit l'exemple de la vérification fonctionnelle en traitement de signal. Historiquement, la conception d'un filtre numérique débute par l'utilisation d'outils algorithmiques tels que MathWorks MATLAB/Simulink. Lorsque le filtre possède les caractéristiques désirées, celui-ci est raffiné à un niveau d'abstraction inférieur se rapprochant de la cible prévue où on vérifie ses caractéristiques de nouveau. Ce processus est répété à plusieurs reprises jusqu'à l'atteinte d'une représentation assez proche de la cible permettant une confiance suffisante dans le modèle pour lancer la première vague de production. Typiquement, il y a une cassure entre les étapes de raf-

finage *i.e.* non seulement on doit réécrire le modèle mais également les artéfacts[1] permettant la vérification et la validation du filtre. Cette cassure est d'autant plus encline à l'introduction d'erreurs lorsque la réécriture des artéfacts ne se limite pas qu'à une simple traduction vers un autre langage *e.g.* il est fréquent d'avoir à changer la représentation des nombres.

1.1.2 Support multi niveaux d'abstraction

Un design peut être décrit à divers niveaux d'abstraction *i.e.* les composantes peuvent être implémentées à des niveaux d'abstraction différents au cours de la réalisation du design. De même, par commodité, certains domaines peuvent imposer leurs représentations ajoutant à la complexité de la création d'un tel design. Ce type de design est aussi appelé système hétérogène. La Figure 1.2 présente des exemples de représentations en fonction des niveaux d'abstraction ou des domaines d'application. Les niveaux matériel, transfert de registres (RTL) et conception système sont des niveaux d'abstraction alors que la vérification et la modélisation système sont plutôt des domaines.

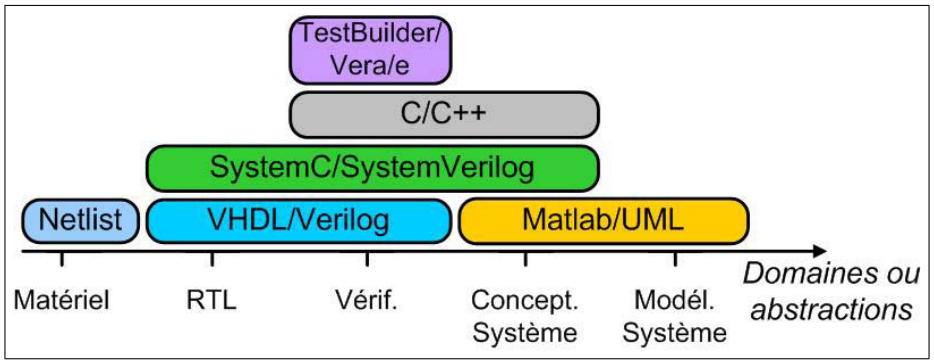

Figure 1.2 Exemples de représentations en microélectronique numérique.

Ainsi, pour vérifier un système hétérogène, il faut que deux composantes exprimées avec des représentations distinctes puissent communiquer. Il existe plusieurs solutions se divisant essentiellement en 2 catégories : l'approche par représentation intermédiaire ainsi que l'approche par adaptation de fonctionnalité et de données.

[1] *e.g.* le banc d'essai.

Représentation intermédiaire

Avec l'approche par représentation intermédiaire, on cherche à ramener tous nos modèles à une représentation intermédiaire commune tout en conservant un niveau de détail satisfaisant pour notre utilisation. Ce nouveau modèle unifié est simulé au sein du même moteur de simulation.

Comme c'est le même moteur de simulation qui exécute le modèle, l'ajout du support pour une nouvelle représentation (*e.g.* nouveau langage de modélisation ou nouveau niveau d'abstraction) peut être fastidieux. Dans certains cas, c'est impossible, car à moins de travailler pour l'entreprise développant le simulateur, on n'a pas accès au code source de l'application.

Certains simulateurs commerciaux supportant divers niveaux d'abstraction simultanément utilisent une représentation intermédiaire ou du moins des techniques s'y apparentant. À titre d'exemple, le simulateur de l'Open SystemC Initiative (OSCI) est un simulateur pour le langage SystemC supportant plusieurs niveaux d'abstraction. De même, Mentor Graphics ModelSim est un simulateur supportant plusieurs niveaux d'abstraction ainsi que plusieurs langages de modélisation. Ces deux simulateurs supportent les niveaux transfert de registres et système mais se spécialisent dans l'un ou dans l'autre de ces niveaux *e.g.* le simulateur de l'OSCI n'est pas l'outil idéal pour la simulation au niveau transfert de registres puisqu'il ne supporte pas les deux langages de modélisation les plus utilisés : Verilog et VHDL. Enfin, aucun de ces deux outils ne supportent le niveau algorithmique.

Adaptation de fonctionnalité et de données

Avec cette approche, les composantes exprimées à des niveaux d'abstraction incompatibles entre eux demeurent intactes. Au lieu de transformations vers une représentation intermédiaire, une adaptation de fonctionnalité et de données est effectuée.

Ainsi, on délaisse habituellement l'utilisation d'un seul simulateur pour en utiliser deux ou plus, tout en ouvrant la porte au *Hardware-in-the-loop* (HIL). Le support pour de nouveaux langages de modélisation ou de niveaux d'abstraction requiert donc que les outils impliqués

dans la simulation des modèles aient une interface programmable. Pour qu'une interaction avec d'autres outils soit possible, au minimum, ils doivent posséder des mécanismes permettant une communication avec le monde extérieur. Par monde extérieur on désigne tout ce qui est en dehors de l'outil tel qu'un fichier sur le disque dur, la mémoire vive, une interface de connexion (*socket*), *etc*.

Dans la littérature, l'objet effectuant l'adaptation est fréquemment appelé *transactor* [6, 11, 12]. À titre d'exemple, la boite à outils *EDA Simulator Link MQ* de MathWorks MATLAB utilise une forme de *transactor* pour faire le lien entre un design simulé au niveau transfert de registres (RTL) dans Mentor Graphics ModelSim et un design simulé au niveau algorithmique dans MathWorks MATLAB.

Puisque les liens entre les composantes d'un tel système ne sont pas directs, cette technique est généralement accompagnée d'une architecture de communication assurant la liaison entre les simulateurs ou les outils. Dans le cas spécifique de *EDA Simulator Link MQ*, l'architecture est cachée, mais deux média s'offrent à nous : la mémoire partagée ou une interface de connexion.

1.1.3 Cosimulation

L'objectif de la cosimulation est d'utiliser des outils de conception au niveau conception système afin de faciliter l'exécution et l'analyse. Ces outils sont complémentaires au design et ne se substituent pas à des composantes du design.

Ainsi, des logiciels comme MathWorks MATLAB ou GNU Octave peuvent être utilisés pour générer du trafic ou pour calculer les densités spectrales des canaux de communication d'une radio.

Pour permettre la cosimulation, il faut interfacer notre design à vérifier avec l'outil. Il existe des solutions du côté industriel et académique.

1.1.4 Méthodes de vérification fonctionnelle pour un système hétérogène

Pour la vérification fonctionnelle d'un système hétérogène incluant au moins une composante au niveau matériel ou transfert de registres, trois méthodes sont possibles.

Simulation

La stratégie la plus commune et certainement la moins coûteuse est la simulation d'une représentation au niveau transfert de registres. C'est une méthode simple d'utilisation, précise, flexible, mais très lente. Du simple traçage de forme d'onde à l'exécution pas à pas, la qualité des mécanismes pour le débogage d'un design varie d'une solution à l'autre, mais y est habituellement excellente. Il existe une panoplie de simulateurs de matériel autant du côté académique qu'industriel *e.g.* Mentor Graphics ModelSim, Altera Quartus II, GHDL et Icarus Verilog. Pour compenser la lenteur de cette méthode, certaines entreprises (*e.g.* Synopsys et EVE) offrent des accélérateurs de simulation matériels.

Hardware-in-the-loop

Le *Hardware-in-the-loop* (HIL) réfère à la conjonction de matériel et de logiciels pour faire de la vérification. La méthode de HIL la plus courante consiste à synthétiser une représentation au niveau transfert de registres et d'utiliser des *field programmable gate arrays* (FPGAs). Ces derniers sont rapides (typiquement 20MHz et plus) et peu coûteux. Des outils tels que Chipscope de Xilinx ou SignalTap d'Altera permettent une excellente observabilité tout en requérant peu d'espace dans le design. Cependant, leur utilisation n'est pas toujours possible *e.g.* il manque d'espace dans le FPGA pour inclure cet outil, le cas à tester n'est pas supporté par l'outil, le design contient une technologie pour laquelle il n'y a pas d'outil équivalent, *etc*. Dans ces cas, il est parfois difficile, voire impossible, d'observer des signaux internes au design. Ainsi, le deverminage d'un design y est beaucoup plus difficile. De plus, l'implémentation d'un design complexe devant être partitionnée sur plusieurs FPGAs s'avère une tâche ardue et encline à l'erreur. Pour palier à ce problème, certains [14, 28, 33, 34] tentent d'automatiser ce processus pour le rendre efficace.

Le HIL est la seule méthode permettant également de faire de la cosimulation avec une puce existante.

Émulation

Aussi connue sous le nom *In-Circuit Emulation* (ICE), l'émulation matériel est une technique qui consiste à imiter le comportement de matériel à l'aide d'un système matériel spécifiquement conçu à cet effet. Bien que plus coûteuse que la simulation, les principaux avantages de l'émulation sont sa rapidité d'exécution, mais surtout sa paramétrisation. En effet, les émulateurs permettent l'ajustement du degré de raffinement de l'émulation. Cela permet de choisir un bon compromis entre la précision et la rapidité d'exécution. Malgré que les outils de débogage ne soient pas aussi puissants qu'avec un simulateur, l'observabilité d'un design y est de loin supérieure au HIL. Bien que la rapidité d'exécution varie, elle est de l'ordre de 1 à 3 MHz ou encore de 10 000 à 100 000 fois plus rapide que la simulation. Considérant la rapidité d'exécution et les outils de débogage, cette solution est généralement considérée comme étant un excellent compromis entre la simulation et le HIL. Bien qu'il existe certaines solutions d'émulation de matériel ouvertes (*e.g.* QEMU [7]) les meilleures sont commerciales, fermées, mais supportées par des entreprises (*e.g.* EVE et Cadence). Enfin, cette solution ce limite souvent aux circuits et processus connus et communs.

1.2 Survol des architectures de communication

Lorsque la simulation d'un système nécessite plusieurs simulateurs ou outils, une architecture de communication est nécessaire.

Dans l'industrie et dans le domaine académique, il y a essentiellement 2 approches :

– Architecture et protocole(s) personnalisés [4, 5, 27]

– Architecture et protocole(s) normalisés [25, 32, 37]

Il existe une panoplie d'architectures normalisées. Certaines demeures peu communes *e.g.* la *High Level Architecture* (HLA) [16] développée par le département états-unien de la défense

ou encore GLOBUS [18], une architecture élaborée par un regroupement d'universités. Ainsi, le Tableau 1.1 résume les caractéristiques des 3 architectures normalisées les plus courantes.

Tableau 1.1 Architectures de communication normalisées

	CORBA	**Java-RMI**	**DCOM**
Organisme	OMG	Sun Microsystems	Microsoft
Définition d'objet	IDL	Interface Java	MIDL
Représentation lors du transport	CDR	Sérialisation Java ou CDR	NDR
Protocole de transport	IIOP	JRMP ou RMI-IIOP	ORPC

Pour un comparatif complet de Java-RMI, CORBA et DCOM, voir le Tableau 1.1 dans [3]. Pour un comparatif de DCOM et CORBA, voir [2, 41]. Pour une comparaison de HLA, CORBA et Java-RMI, voir [13].

Dans le cas des architectures personnalisées, leurs caractéristiques varient trop pour être résumées. Cependant, certains mécanismes de communication sont communs : les mécanismes POSIX de communication inter processus (IPC) ainsi que les appels de procédure à distance (RPC).

1.2.1 Acceptation ou normalisation

Notons que CORBA est une norme de l'Object Management Group (OMG) et est basé sur l'*Object Management Architecture* (OMA) ainsi que le *Component Object Model* (COM). DCOM est une norme de Microsoft et est purement une extension du COM. Quant à lui, Java-RMI est basé sur les mécanismes de représentation de classe de la machine virtuelle Java. DCOM est la seule de ces architectures à ne pas posséder de spécification complète (BNF formel). Avec une grande majorité d'articles scientifiques faisant référence à CORBA, dans le domaine de la simulation distribuée, CORBA se démarque comme étant l'architecture normalisée la plus courante.

1.2.2 Efficacité et performance

Globalement, la maximisation des performances d'une architecture d'objets distribués passe par l'optimisation des communications. Pour ce faire, de manière générale, la tendance [31] est de faire un mélange intelligent des méthodes de simulations orientées événements et orientées données.

Ainsi, les techniques communes consistent à minimiser la fréquence des communications entre les simulateurs, à minimiser la période active des simulateurs et à maximiser le parallélisme.

Architectures personnalisées

L'efficacité et la performance des architectures personnalisées sont à évaluer au cas par cas. Ces deux critères ne sont pas pris en considération dans le cadre de ce projet, car la nature même des architectures personnalisées entre en conflit avec nos spécifications du chapitre 2.

Architectures normalisées

La performance d'une architecture dépend fortement de l'implémentation. Ainsi, on cherche à optimiser les composantes critiques de l'architecture *e.g.* l'ORB dans le cas de CORBA et DCE-RPC dans le cas de DCOM.

Dans le cas de Java-RMI, un reproche qui revient souvent est la lenteur de l'interface de programmation (API) de réflection [3]. Pour ce qui est de CORBA, l'architecture est bien définie, complète, modulaire et implémentée pour plusieurs plateformes. De plus, il existe de nombreuses implémentations temps réel d'ORBs [9, 26, 44, 46]. Lorsque l'ORB utilisé n'est pas une implémentation temps réel et que le système d'exploitation est un de ceux de Microsoft, les performances de DCOM sont comparables à celles de CORBA [1]. Cependant, l'architecture de DCOM est monolithique et repose sur des optimisations spécifiques à une seule plateforme [2].

1.2.3 Support multi environnements ou adaptabilité aux environnements hétérogènes

Dans le cadre de ce projet, le support multi environnements signifie la capacité à exprimer les objets dans plusieurs langages de programmation ainsi que la capacité de l'architecture de communication à s'exécuter simultanément sur différents systèmes d'exploitation et architectures matérielles. Chacune des architectures présentées fournit un ou plusieurs services s'occupant de faire la liaison entre les objets *i.e.* transporter l'information d'un objet à un autre. Certaines architectures offrent un outil, appelé *language mapper*, automatisent en partie ou en totalité la création des objets dans un langage d'implémentation supporté.

Bien qu'il soit indépendant du langage d'implémentation, tel que mentionné dans la sous-section précédente, DCOM est particulièrement optimisée pour une plateforme. Par le passé, Microsoft et d'autres entreprises ont tenté d'implémenter et de supporter DCOM pour d'autres architectures, mais n'ont pas réussi à s'imposer[2]. Cela fait de DCOM une architecture qui n'est pas adaptée aux environnements hétérogènes. Afin de contourner ce problème, certains ont développé des ponts permettant à DCOM d'interagir avec CORBA et vice versa [17, 41].

Tout comme DCOM, CORBA est indépendant du langage d'implémentation [39]. Le langage de programmation utilisé peut ne pas être un langage orienté-objet *e.g.* l'ORB peut être implémenté dans un langage tel que le C. Enfin, CORBA est également indépendant du système d'exploitation et de l'architecture matérielle. À lui seul, The ACE ORB (TAO)[3] [44] est disponible pour plusieurs systèmes d'exploitation tels que VxWorks, LynxOS, Solaris, Windows et GNU/Linux ainsi que pour plusieurs architectures matérielles telles que x86, x86_64, alpha, arm, hppa, mips, s390, *etc*. Il y a des implémentations de CORBA pour des systèmes d'exploitation temps réel ainsi que pour du matériel. Cela fait de CORBA une architecture qui s'intègre bien aux environnements hétérogènes.

Pour sa part, Java-RMI se limite au langage Java sauf en cas d'utilisation du protocole de communication inter ORB (IIOP) au lieu du protocole *Java Remote Method Protocol* (JRMP). Dans

[2] Autrefois, l'entreprise SoftwareAG offrait une implémentation de DCOM pour Unix et GNU/Linux mais ce support a disparu à partir de la version 8.0 de leur produit EntireX.

[3] TAO est une implémentation libre de CORBA.

un tel cas, l'infrastructure RMI à travers l'IIOP offre l'architecture CORBA. Tous les langages supportés par CORBA deviennent donc utilisables. De plus, sans passer par RMI-IIOP, dans le cas où le *Java Naming and Directory Interface* (JNDI) sert pour les liaisons, on peut directement interagir avec les services et objets CORBA. Finalement, tout comme CORBA, Java est indépendant du système d'exploitation et de l'architecture matérielle dans la mesure où il y existe une machine virtuelle Java. Une machine virtuelle Java permet d'interpréter et d'exécuter du pseudo-code binaire (une forme de représentation intermédiaire) Java. À elle seule, l'entreprise Sun Microsystems fournit des machines virtuelles Java pour les systèmes d'exploitation GNU/Linux, Microsoft Windows et Sun Solaris pour les architectures x86, x86_64, sparc et ia64. Le projet GNU de compilateur pour Java (GCJ) supporte encore davantage de systèmes d'exploitation et d'architectures matérielles. Ainsi, Java-RMI est également une architecture qui s'intègre bien aux environnements hétérogènes. Cependant, contrairement à CORBA, il n'existe pas d'implémentation matérielle et la nature de Java-RMI fait en sorte que cette architecture de communication n'est pas une bonne candidate pour les systèmes temps réel.

Notez qu'on peut également directement utiliser CORBA sous Java puisqu'il existe des implémentations natives d'ORBs ainsi qu'un *language mapper* CORBA pour Java.

1.2.4 Support de multiples niveaux d'abstraction

Bien que CORBA permette de travailler à divers niveaux d'abstraction, ce n'est pas le cas de toutes les architectures de communication. Ainsi, dans tous les cas observés [8, 15, 35], un langage de niveau conception système (*e.g.* SystemC ou C#) sert de passerelle entre les différents niveaux d'abstraction. Dans ces cas, le support de multiples niveaux d'abstraction repose plutôt sur la combinaison des simulateurs utilisés.

1.3 Conclusion

La conception et la vérification fonctionnelle d'un système électronique complexe requiert presque inévitablement un environnement hétérogène permettant la simulation simultanée de plusieurs niveaux d'abstraction. En cette matière, la création d'une représentation intermédiaire

flexible est ardue et coûteuse. Pour sa part, la méthode d'adaptation de fonctionnalité et de données se fait bien et s'insère aisément avec une dorsale de communication. Cette méthode est donc à privilégier puisqu'elle simplifie grandement la tâche.

Parmi les architectures de communication normalisées, le choix de CORBA semble s'imposer de par sa flexibilité, ses performances ainsi que de par son acceptation dans les milieux académiques et commerciaux. Il existe d'ailleurs des précédents en la matière où l'intégration de l'architecture d'objets distribués CORBA s'est faite au niveau système (SystemC). C'est ce qui fût fait dans le cadre du projet CARH de l'université Virginia Tech [40].

CHAPITRE 2

SPÉCIFICATION DE L'ENVIRONNEMENT DE CONCEPTION ET DE VÉRIFICATION FONCTIONNELLE

Ce chapitre présente les spécifications, les hypothèses de travail ainsi que les utilisateurs potentiels des résultats de ce projet de recherche. La conclusion résume les objectifs de ce travail.

2.1 Spécifications

Cette section définit les spécifications que la dorsale de communication doit rencontrer. Pour permettre sa mise en application dans d'autres contextes et sa réutilisation, la dorsale doit rencontrer certaines caractéristiques. La dorsale doit être :

a. Générique : capable de supporter des composantes hétérogènes, de s'exécuter sur différentes plateformes et de transporter des signaux autant que des objets ;

b. Flexible : capable de transporter l'information à travers plusieurs média et protocoles de communication et de s'intégrer avec d'autres applications de plusieurs manières différentes *e.g.* mécanismes POSIX, mémoire partagé, instanciation, *etc* ;

c. Extensible et évolutive : capable de s'adapter aux changements et permettre des ajouts ne faisant pas partie de la version initiale *e.g.* ajout du support pour un nouveau simulateur, ajout du nouveau protocole de communication, *etc* ;

d. Ouverte : reposer sur des normes ouvertes ;

e. Simple d'utilisation : automatiser le plus possible ce qui sort du domaine de l'électronique.

Les sous-sections suivantes apportent des précisions sur ces caractéristiques.

2.1.1 Composantes hétérogènes

La dorsale doit permettre la conception et la simulation d'un système constitué de composantes hétérogènes. C'est-à-dire que ces dernières doivent pouvoir être exprimées avec des langages de modélisation différents et/ou à des niveaux d'abstraction différents. Cette exigence se résume au support de multiples simulateurs/outils et à la possibilité d'effectuer une adaptation de fonctionnalité ou de signaux.

2.1.2 Simulateurs et outils de conception

Pour la vérification, l'exécution de modèles de composantes passe par un simulateur. Puisque le support de composantes hétérogènes est essentiel, il faut que la dorsale permette l'intégration de plusieurs simulateurs. La majorité des simulateurs offre des mécanismes permettant d'interagir avec d'autres applications. Puisque les méthodes et langages de programmation utilisés varient, la dorsale de communication proposée doit être suffisamment flexible pour s'intégrer à ce qui est disponible. De plus, l'indépendance vis-à-vis du système d'exploitation est un atout considérable.

2.1.3 Communication

Pour maximiser la flexibilité de la dorsale, cette dernière doit supporter divers média et protocoles de communication. Ainsi, l'architecture doit permettre la communication autant à travers un bus de bas niveau qu'à travers un protocole crypté de haut niveau. L'implémentation initiale doit minimalement supporter l'échange entre les simulateurs par la mémoire partagée et le protocole TCP/IP ainsi que permettre l'ajout ultérieur de protocoles.

2.1.4 Adaptation de fonctionnalité et des données

Le passage d'un niveau d'abstraction à un autre requiert parfois une adaptation en fonctionnalité et/ou de données. Ainsi, la dorsale doit permettre l'implémentation du concept de *transactor* tel qu'exprimé dans [6, 12]. Un *transactor* s'insère entre deux composantes afin de faire l'adaptation de fonctionnalité et de données. Par exemple, le *transactor* peut faire une conver-

sion d'une représentation de nombres à virgule flottante (IEEE754) vers une représentation de nombres signés complément deux à virgule fixe et vice versa. Ou encore, il peut prendre en charge la gestion de signaux additionnels présents dans une implémentation faite à un bas niveau d'abstraction (*e.g.* transfert de registres), mais absents dans une implémentation algorithmique.

2.1.5 Normalisation

Au-delà de la crédibilité, l'utilisation de normes ouvertes facilite la réutilisation et l'intégration à d'autres applications. Du même coup, en utilisant des normes éprouvées, on évite de réinventer la roue pour se concentrer sur une utilisation optimale. Cela est d'autant plus vrai dans le cas où les normes utilisées jouissent d'une grande acceptation dans l'industrie et dans le domaine académique.

2.1.6 Simplicité d'utilisation

L'objectif de cette dorsale est de faire de la conception et de la vérification en électronique. Ainsi, l'aspect logiciel doit être simplifié pour rendre la dorsale accessible à des utilisateurs qui ne sont pas des informaticiens. Ainsi, il est souhaitable d'appliquer des techniques de génération automatisée à partir de gabarit là où c'est applicable. Dans le même esprit, la complexité de la communication ne doit pas être apparente à l'utilisateur.

2.2 Hypothèses de travail

Voici les hypothèses ayant principalement orientées ces travaux de recherches :

– Les utilisateurs de cette dorsale sont prêts à vivre avec un temps d'exécution légèrement plus long (de l'ordre de 20%) au profit d'une plus grande flexibilité. Ainsi, la possibilité d'intégrer plus de deux simulateurs leur est plus intéressante que d'être limité à des solutions plus performantes, mais avec un fort couplage entre deux outils ;
– MathWorks MATLAB/Simulink est un outil populaire parmi les concepteurs de systèmes numériques, surtout pour faire une première implémentation à partir des spécifications ;

– Les utilisateurs éventuels voudront pouvoir facilement intégrer du HIL ;
– Les utilisateurs éventuels ne veulent pas devoir se soucier de la complexité ajoutée par le choix d'une telle architecture de communication ;
– Le protocole de communication TCP/IP est le plus couramment utilisé sur Internet et son support dès la première itération permettra une meilleure acceptation de cette dorsale de communication.

2.3 Utilisateurs

Les utilisateurs principalement visés par cette dorsale sont les concepteurs de systèmes numériques. Plus généralement, les développeurs de systèmes embarqués et les autres chercheurs en microélectronique peuvent également utiliser cette dorsale.

2.4 Conclusion

Le but de ce mémoire est de proposer une dorsale de communication générique pour la conception et la vérification de circuits électroniques hétérogènes. Pour cela, la dorsale doit être suffisamment flexible pour supporter différents simulateurs, langages de modélisation et systèmes d'exploitation. La dorsale ne doit pas imposer de média de communication ni de protocole de communication. Elle doit supporter plusieurs protocoles tout en permettant l'intégration de nouveaux protocoles. La complexité logicielle, entre autres liée à la communication, doit être cachée de l'utilisateur. L'architecture doit être extensible et évolutive pour permettre la vérification de système complexe. L'architecture doit permettre l'intégration du concept de *transactor*, essentiel à la vérification avec des composantes exprimées à des niveaux d'abstraction différents. Enfin, l'architecture doit reposer sur des normes ouvertes. Ainsi, CORBA est l'architecture de communication qui s'impose pour la réalisation de la dorsale.

CHAPITRE 3

ARCHITECTURE DE LA DORSALE DE COMMUNICATION

Ce chapitre décrit les éléments clefs de notre méthodologie de vérification soit le modèle d'architecture, l'environnement de communication ainsi que les outils impliqués dans notre flot de vérification hétérogène. Les outils conceptuels sont présentés en quatre sections soit les composantes passerelles, les adaptateurs d'outil, les enveloppes de composante et les enveloppes de client ou de serveur. Une section expliquant brièvement l'initialisation de la simulation clôture ce chapitre.

L'environnement de communication proposé et la méthodologie qui y est associée firent l'objet d'une présentation à un atelier de l'OMG en mars 2007 [20].

3.1 Modèle d'architecture

CORBA est une norme d'architecture d'objets distribués définie par l'Object Management Group (OMG) utilisée dans un large spectre d'applications. Les applications vont des appareils bas-niveau de communication militaire jusqu'aux applications logiciels haut-niveau. Les interfaces des objets sont décrites à l'aide du langage de définition des interfaces (IDL) tel que défini par l'OMG.

Les objets peuvent être implémentés à l'aide de divers langages et sur diverses plateformes. En fait, en termes de langage et de plateforme, CORBA est agnostique. Les enveloppes de client et serveur sont autogénérées à partir des fichiers de définition d'interface. Un *language mapper* s'occupe de générer les objets CORBA dans le langage d'implémentation désiré. Il existe des *language mappers* pour plusieurs langages. Certains sont officiellement documentés [38] tels que ceux pour Ada, C, C++, Java, Python et PL/1 alors que d'autres existent (*e.g.* SystemC, Eiffel et VHDL) sans être endossés par l'OMG.

De plus, CORBA possède un cadriciel d'extension des protocoles de communication (*Extensible Transport Framework* (ETF)). Ainsi, CORBA n'est pas limité à un protocole de transport *i.e.* des greffons implémentant un nouveau protocole de transport peuvent être ajoutés à l'architecture existante.

3.2 Environnement de communication

La communication est un des plus gros défis en vérification hétérogène. Le choix d'une architecture de communication basée sur la distribution d'objets est justifié par plusieurs facteurs. La vérification de composantes implémentées à divers niveaux d'abstraction implique habituellement l'utilisation de multiples langages de modélisation. Ainsi, plusieurs simulateurs doivent communiquer ensemble. L'analyse de données ne requérant pas de modifications au design ni de tâches fastidieuses est souhaitable. La vérification matérielle à l'aide de carte de prototypage ou de développement tôt dans le processus est un atout. L'architecture de communication présentée permet de faire tout cela. Le coût en découlant est mineur. Il s'agit du temps requis pour interfacer un outil à l'architecture : la conception d'adaptateurs d'outil. Une vue simplifiée de l'architecture est présentée à la Figure 3.1. Chaque simulateur ou outil requiert un ORB s'occupant de l'intercommunication. Le simulateur, ou outil, agissant comme maître de simulation possède de un à plusieurs clients CORBA. Les autres simulateurs ou outils sont des serveurs exposant leurs services/composantes via le *Naming Service* de CORBA.

3.3 Composantes passerelles

Les composantes incluses dans le maître de simulation, mais qui n'y sont pas simulées sont des composantes passerelles. C'est-à-dire que ces composantes sont en fait des clients CORBA envoyant des requêtes aux serveurs CORBA qui, à leur tour, communiquent avec le simulateur des composantes visées. Le maître de simulation n'a pas conscience de cette duperie.

Les composantes passerelles sont un type d'adaptateur d'outil et intègre généralement l'enveloppe de composante et l'enveloppe client. Lorsque la composante contient beaucoup de

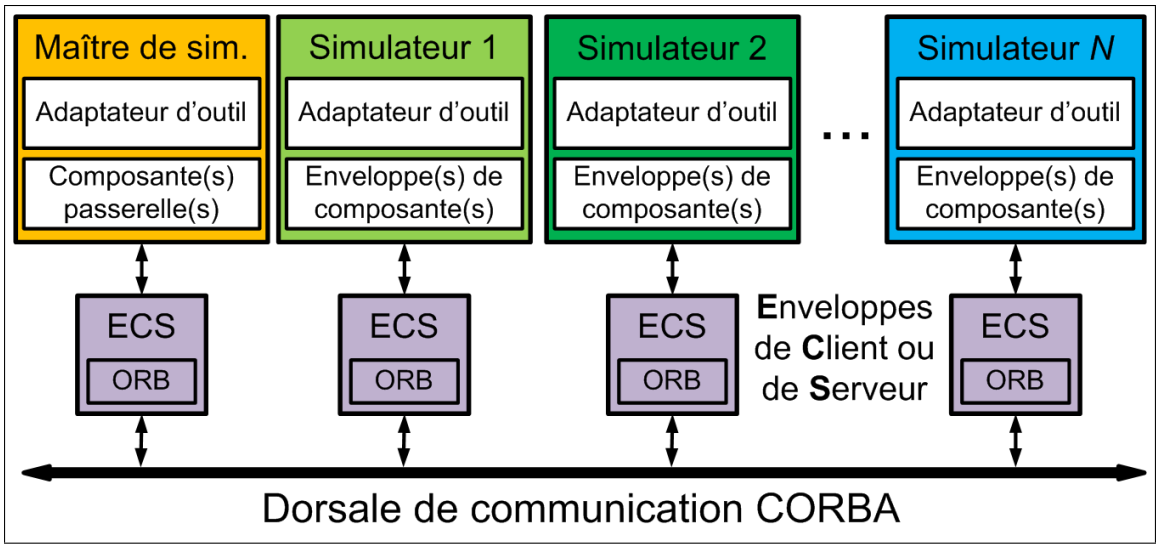

Figure 3.1 Architecture de la dorsale de communication CORBA.

signaux requérant une adaptation de fonctionnalité et de données, il est suggéré de séparer l'enveloppe de composante de la composante passerelle.

3.4 Adaptateurs d'outil

Pratiquement tous les simulateurs offrent une interface externe permettant à un développeur d'interagir programmatiquement avec l'outil. C'est la porte d'entrée pour les adaptateurs d'outil permettant l'interconnexion des objets CORBA avec les composantes simulées. Les tâches de l'adaptateur incluent le contrôle du flot de simulation (le démarrage, la suspension, la reprise et l'arrêt). C'est également l'adaptateur qui fournit les mécanismes nécessaires à l'enveloppe de composante pour l'initialisation de la composante ainsi que la lecture et l'écriture de signaux.

Dans certains cas, l'adaptateur d'outil n'est qu'un conteneur ou une librairie partagée à l'intérieur duquel l'enveloppe de composante est implémentée. Le FLI de ModelSim, le VHPI de GHDL et les *S-Functions* de Simulink en sont des exemples.

3.5 Enveloppes de composante

Le passage d'un domaine de modélisation à un autre requiert parfois une adaptation de fonctionnalité ou de données. Dans ces cas, l'enveloppe de composante agit comme un *transactor*. Étant entre la composante et le client CORBA, c'est essentiellement le rôle de l'enveloppe de composante. Un bloc Simulink, un module SystemC ou un design VHDL sont des exemples de composantes.

Dans les cas où il n'est pas nécessaire d'effectuer une adaptation de fonctionnalité ou de données, l'enveloppe de composante ne fait que connecter les signaux correspondants. C'est-à-dire qu'il ne fait que transmettre les signaux de la composante au client ou au serveur CORBA et vice versa. À noter qu'avant d'atteindre le client ou le serveur CORBA, les signaux passent au travers de l'adaptateur d'outil et de l'enveloppe de client ou de serveur.

3.6 Enveloppes de client ou de serveur

Responsable de l'initialisation, de l'exécution et de la destruction des objets CORBA, l'enveloppe de client ou de serveur est également responsable de faire passer les messages (signaux dans ce contexte) reçus de l'ORB vers l'enveloppe de composante et vice versa.

De plus, l'implémentation actuelle des enveloppes de client ou de serveur permet au concepteur de configurer les coordonnés du *Naming Service* CORBA (*CORBA endpoints*) à l'exécution des clients et serveurs.

3.7 Initialisation de la simulation

Lors de l'initialisation, le maître de simulation demande à sa composante de s'initialiser. Lorsque la composante passerelle reçoit ce signal, elle configure les paramètres CORBA avec ce que le concepteur a spécifié en paramètres[1] et demande au client de s'initialiser à son tour. Le client demande au *Naming Service* de lui fournir l'adresse du service offrant l'interface de la composante désirée. Le *Naming Service* répond avec l'adresse du serveur, s'il en existe au

[1] Au chapitre 5, il sera vu que cela prend la forme d'un paramètre d'un bloc Simulink.

moins un. De là, le client indique à la composante passerelle que l'initialisation est terminée. Dès lors la composante passerelle démarre le client et la simulation peut débuter. La Figure 3.2 illustre ce processus, l'axe vertical représente le temps et l'axe horizontal les différents acteurs impliqués dans l'exécution de la tâche.

Certains outils conceptuels n'apparaissent pas sur la figure puisqu'ils sont inclus dans un autre acteur. Cela permet de faciliter l'interprétation de la figure et d'éviter d'encombrer inutilement le diagramme de séquence avec des outils ne faisant que relayer des appels. Ainsi, l'adaptateur d'outil n'apparait pas puisque la composante passerelle est une forme d'adaptateur d'outil. De même, l'enveloppe de composante n'apparait pas sur la figure puisqu'elle est généralement incluse dans la composante passerelle. Enfin, l'enveloppe de client est omise puisqu'elle ne fait que le relais entre la composante passerelle et le client.

3.8 Conclusion

Le domaine de la conception électronique contient une pléthore de langage de modélisation et d'outils excellant dans leur domaine respectif. Cet environnement de conception et de vérification encourage le concepteur à utiliser le meilleur outil pour la tâche à accomplir. De plus, il y a une séparation claire de la fonction des différents outils conceptuels permettant au concepteur de programmer de façon modulaire lorsque la complexité le justifie. Enfin, comme la communication se fait à l'aide d'ORBs, il suffit d'interfacer un outil à un ORB pour l'intégrer à cette architecture.

Ainsi, un concepteur peut simuler, comme un tout cohérent, un système composé de modèles SystemC (niveau système) simulés avec le simulateur de l'OSCI, des modèles VHDL (niveau RTL) simulés avec ModelSim (ou GHDL) et des modèles algorithmiques de référence exécutés sur MATLAB. De surcroit, le concepteur peut utiliser MATLAB/Simulink pour générer les données en entrée et valider les données en sortie, et ce, pour tous ses modèles sans considération du simulateur.

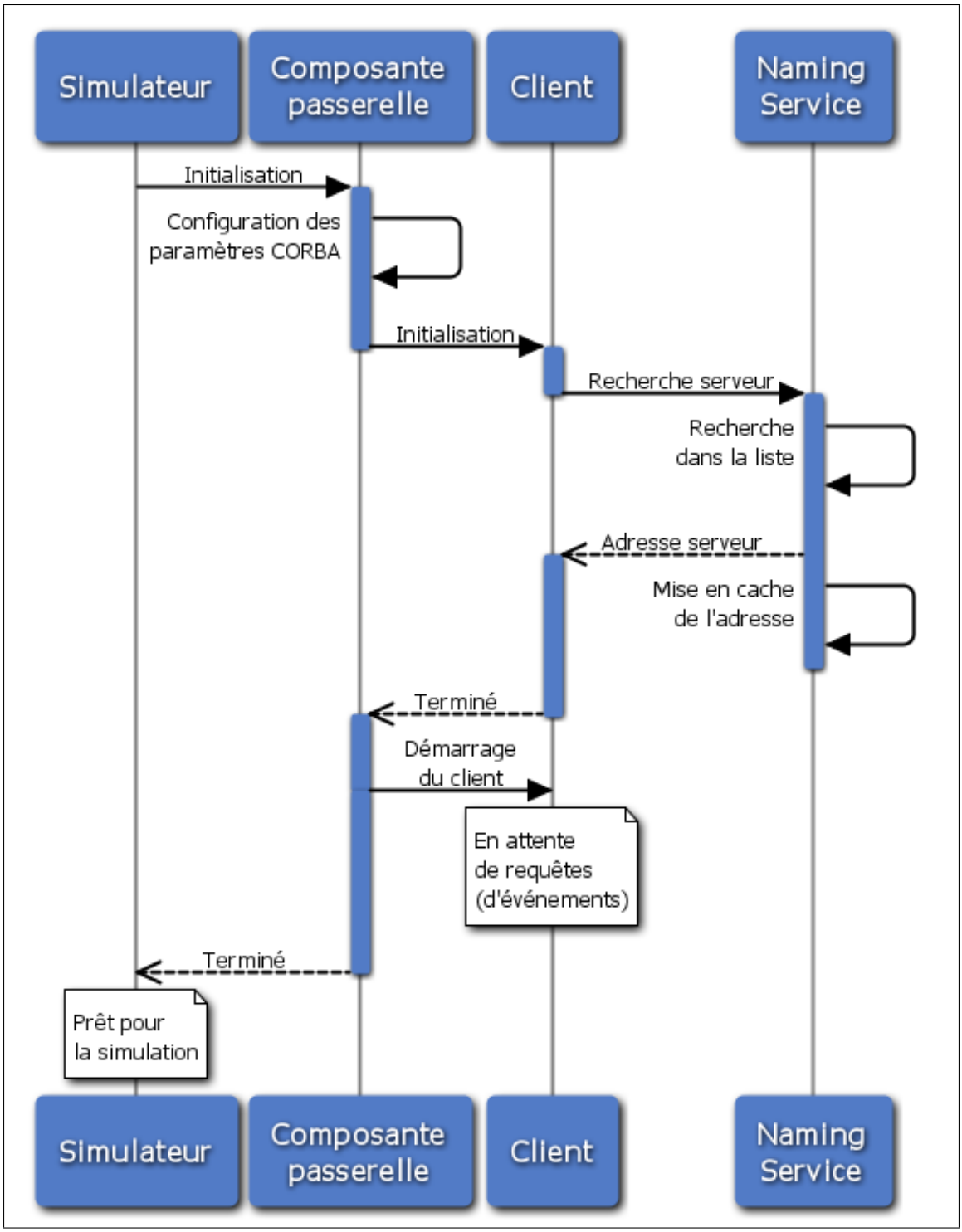

Figure 3.2 Initialisation de la simulation.

CHAPITRE 4

IMPLÉMENTATION ET INTÉGRATION DE LA DORSALE

Comme mentionné dans la section précédente, l'implémentation initiale reproduit le flot de vérification traditionnel tel qu'illustré à la Figure 4.1. Ainsi, chaque acteur requiert un ORB. Nous utilisons une implémentation logicielle et libre : The ACE ORB (TAO). Les interfaces des composantes sont décrites en IDL et sont traduites dans le langage de programmation désiré à l'aide d'un outil tel TAO-IDL. Ce dernier génère automatiquement les interconnexions *i.e.* les objets CORBA ou ORBs.

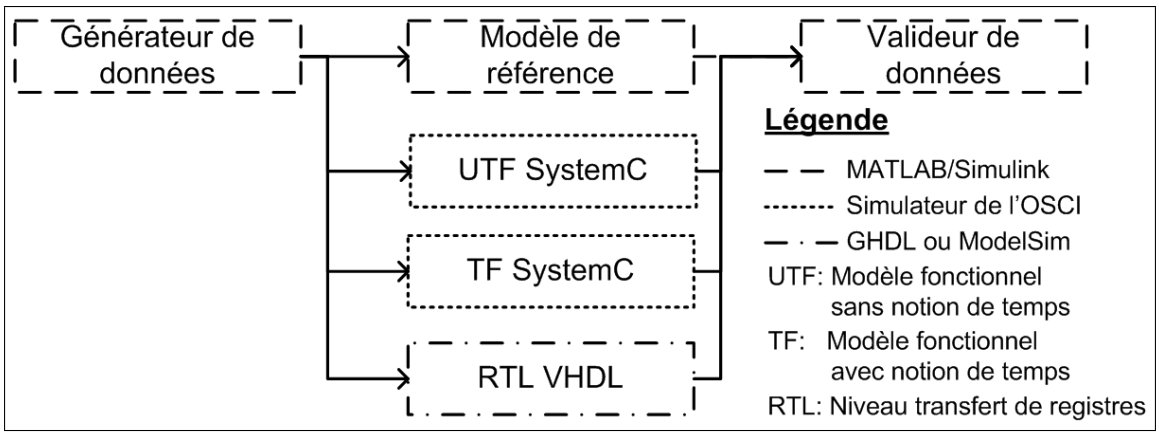

Figure 4.1 Reproduction du flot de vérification traditionnel.

Les ORBs, agissant comme serveur ou client, requièrent une enveloppe, une enveloppe de composante et ultimement un adaptateur d'outil. Comme dans la plupart des cas l'implémentation de l'enveloppe de client ou serveur est triviale, dans les sections suivantes, l'accent est mis sur l'implémentation des adaptateurs d'outil.

Ce chapitre fit l'objet d'une affiche lors de la compétition TEXPO au symposium annuel de CMC Microsystèmes en octobre 2007 [19] ainsi que d'une présentation lors de la huitième réunion du groupe d'utilisateurs nord-américains de SystemC (NASCUG VIII) en marge de la *Design and Verification Conference* (DVCon) en février 2008 [21].

4.1 Caractéristiques communes

Certaines caractéristiques communes à toutes les implémentations découlent des contraintes des outils utilisées alors que d'autres ont pour but de maximiser l'efficacité et la performance de l'environnement proposé. Tout d'abord, l'environnement combine les techniques orientées données et orientées événements. Alors que la vérification est orientée données, la communication est déclenchée par des événements tels que la réception de nouvelles données. Puis, pour minimiser la période active des simulateurs, l'horloge des simulateurs est indépendante de celle du maître de simulation. Dans le même ordre d'idée, les signaux d'une même composante sont regroupés dans un même objet pour minimiser les communications entre les simulateurs. De plus, entre la réception de deux objets les simulateurs sont mis en veille *i.e.* qu'ils attendent la réception d'un événement avant de reprendre les calculs. Enfin, afin de maximiser le parallélisme, chaque simulateur possède son propre ORB.

4.1.1 Absence d'une horloge globale

L'horloge globale fréquemment appelée l'horloge système n'est pas transmise (par défaut) à travers la dorsale. Comme mentionné ci-haut, ceci a le double avantage de diminuer la fréquence des communications ainsi que de permettre aux simulateurs de se mettrent en veille dans l'attente de nouvelles données. En contrepartie, l'utilisateur de la dorsale doit s'assurer qu'il y a cohérence dans le temps dans le cas où ses préoccupations s'étendent au-delà de la fonctionnalité.

Enfin, l'absence d'une horloge globale facilite l'intégration du concept de *Hardware-in-the-loop* (HIL) à cette dorsale. En effet, une carte de prototypage avec un FPGA contient sa propre horloge et cette dernière n'est pas aussi facilement manipulable qu'en simulation logicielle.

Au chapitre 6 on présentera des cas où l'horloge est générée par une enveloppe de composante du côté du simulateur esclaves. Notez cependant que l'ajout d'une horloge peut se faire en incluant un signal d'horloge à l'interface d'une composante. Ainsi, la dorsale de communication

a la flexibilité nécessaire pour permettre au maître de simulation de distribuer une horloge aux composantes exécutées sur des simulateurs esclaves.

4.1.2 Communications bloquantes

Concernant les clients et serveurs CORBA générés automatiquement, notons que la communication entre les ORBs est bloquante. Lorsqu'un serveur reçoit un message d'un client, il lit le message, met à jour les signaux entrants de la composante, simule la composante pour le nombre requis de cycles, lit les signaux sortants de la composante, les renvoie dans un message et attend la réception d'un nouveau message provenant d'un client. La Figure 4.2 illustre ce flot de simulation. Notez que l'enveloppe de serveur n'apparait pas ici, car dans le cas général cette dernière ne fait que le lien entre le serveur et l'adaptateur d'outil.

De même, un client envoie un message à un serveur, attend le message contenant les résultats, met à jour ses sorties lors de la réception du message et continu ses opérations. Les messages envoyés par les clients incluent les signaux[1] entrant des composantes du design. Les messages reçus par les clients incluent les signaux sortant des composantes du design.

Ce flot de simulation généralisé s'applique pour tous les types de simulateurs. Cependant, comme mentionné dans la sous-section ci-dessous, en cas d'utilisation des mécanismes POSIX, ceux-ci s'insèrent entre le serveur et le simulateur.

4.1.3 Mécanismes POSIX

Lorsqu'un moteur de simulation peut être instancié à partir d'un processus externe tout en offrant un contrôle complet sur le flot de simulation, l'utilisation des mécanismes POSIX n'est pas nécessaire puisque tout peut se faire au sein d'un même processus. Ce cas est illustré à la Figure 4.3(A).

Cependant, dans le cas d'un simulateur offrant peu ou pas de mécanismes pour contrôler la simulation, deux processus sont nécessaires. Tel qu'illustré à la Figure 4.3(B), un premier pro-

[1] Au sens électronique du terme.

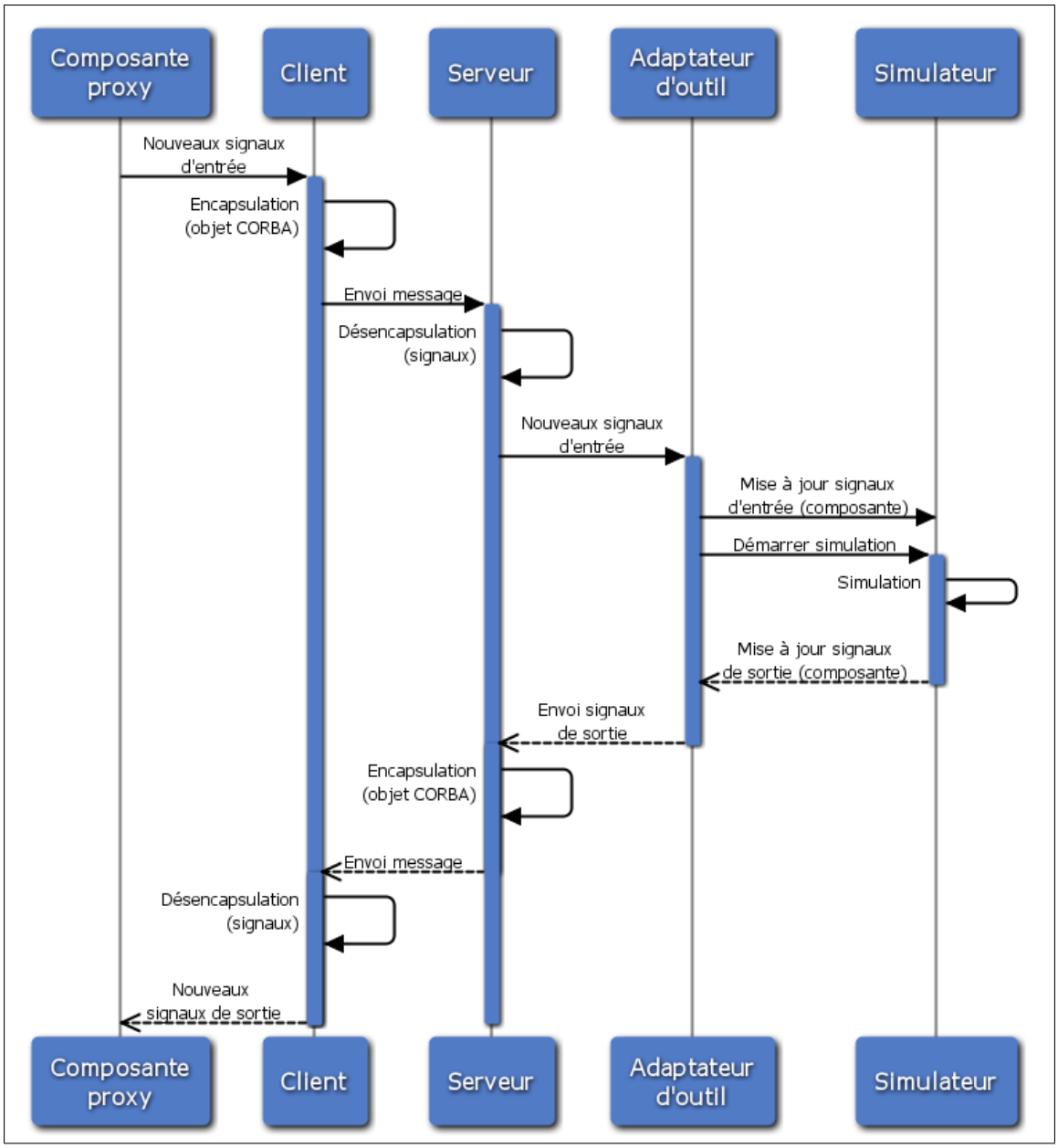

Figure 4.2 Flot de simulation généralisé.

cessus s'occupe de la communication avec la dorsale de communication alors que l'autre effectue la simulation. Dans ce cas, un mécanisme d'échange d'informations entre processus est nécessaire. Plusieurs mécanismes de communication inter processus permettent cela *e.g.* les interfaces de connexion Unix (*Unix sockets*), les canaux de communication (*pipes*), les files d'attente de messages (*message queues*)), *etc*. Parmi les solutions disponibles, les outils

POSIX de communication inter processus (sémaphores et mémoire partagée) sont utilisés puisqu'ils permettent d'efficacement gérer la synchronisation ainsi que l'échange d'informations. En effet, la latence de la mémoire partagée est minime par rapport aux autres possibilités.

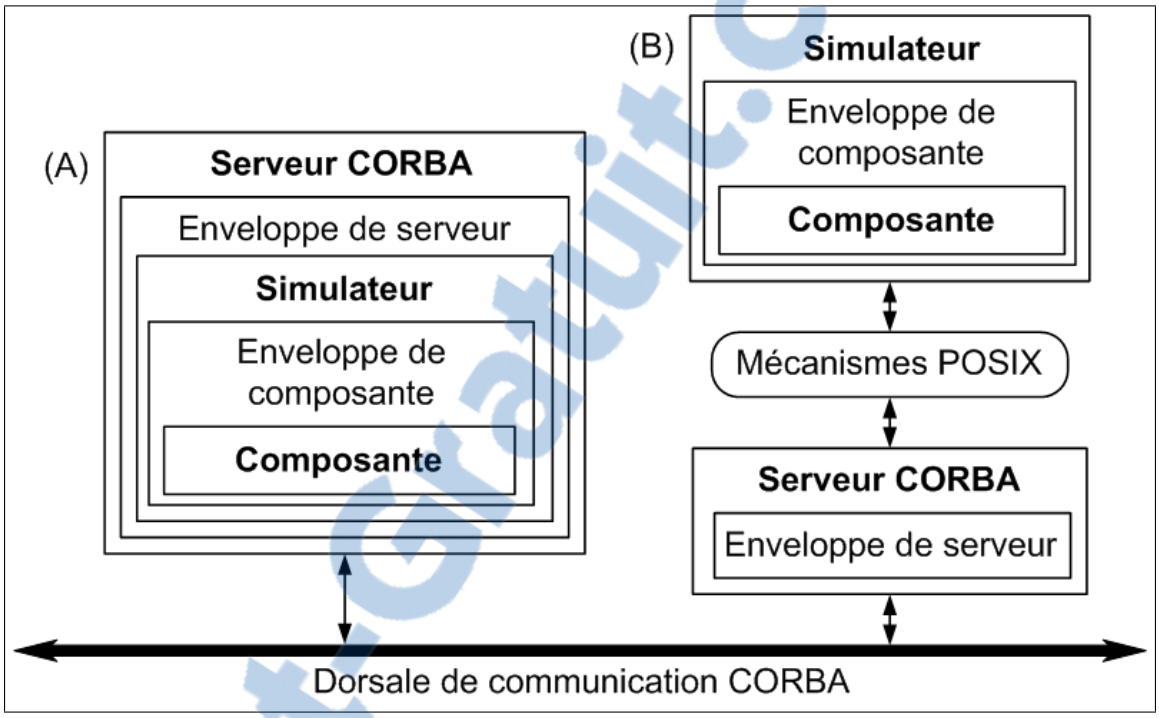

Figure 4.3 Intégration de simulateur à la dorsale de communication CORBA. (A) Simulateur et serveur CORBA dans un même processus. (B) Simulateur et serveur CORBA dans des processus distincts.

Pour le deuxième cas, ce sont l'enveloppe de serveur et l'enveloppe de composante qui partagent un espace de mémoire et deux sémaphores. Un des sémaphores indique à l'enveloppe serveur que la composante est prête à recevoir des signaux alors que l'autre sémaphore indique à l'enveloppe de composante que de nouveaux signaux sont disponibles. Lorsque le client envoie un message contenant des signaux au serveur, l'enveloppe de serveur extrait les signaux du message, copie ceux-ci dans l'espace de mémoire partagée et active le sémaphore. La disponibilité de ce sémaphore réveille l'enveloppe de composante qui récupère les nouvelles données d'entrées dans la mémoire partagée, met à jour les signaux du design et [re]démarre la simulation. Lorsque la simulation est terminée, l'enveloppe de composante récupère les signaux de sorties pour les copier dans la mémoire partagée, émet un sémaphore à l'enveloppe de serveur

et se remet en attente de la disponibilité de son sémaphore. Ce flot d'exécution est illustré dans la Figure 4.4 où, au départ, tous les acteurs sont en attente. Le client est en attente d'une action de l'usager, le serveur est en attente d'un message provenant du client, l'enveloppe de serveur est en attente de nouveaux signaux d'entrée, l'enveloppe de la composante attend sa sémaphore et le simulateur attend la fin de l'exécution de la méthode en cours d'exécution dans l'enveloppe de composante.

La nomenclature utilisée pour les sémaphores et les espaces de mémoire partagée indique le nom du simulateur et de la composante pour éviter les conflits de nom entre simulateurs et composantes. À titre d'exemple, soit des sémaphores pour une composante appelée « FirRtl » simulée à l'aide ModelSim, les deux sémaphores requis porteront le nom de « ms_app_firrtl » et « ms_lib_firrtl ». Les espaces de mémoire partagée porteront le même nom.

4.2 MathWorks MATLAB/Simulink

MATLAB/Simulink est un langage de modélisation au niveau algorithmique et un environnement de calcul numérique utilisé pour simuler des systèmes dynamiques. Dans le cas de notre implémentation initiale, MATLAB/Simulink est utilisé comme maître de simulation. De ce fait, MATLAB/Simulink ne possède que des clients CORBA : le contrôle du flot de simulation n'est pas requis (dans le cas où ce serait nécessaire, MATLAB/Simulink offre ces mécanismes). L'adaptateur d'outil est implémenté à l'aide de fichiers MEX de niveau 2 (*level two MEX files*) : les *S-Functions*.

Les *S-Functions* fournissent un mécanisme pour étendre les fonctionnalités de Simulink via des fonctions de rappel. L'architecture de Simulink est orientée données. Une méthode, « mdlOutputs », est appelée à chaque fois qu'une nouvelle donnée fait son entrée. C'est à l'appel de cette méthode que les signaux sont transmis vers l'enveloppe de client et que les adaptations de fonctionnalité et de données sont effectuées. La sortie de la composante passerelle est mise à jour avant de quitter la méthode « mdlOutputs ». L'extrait de code 4.1 montre un exemple de méthode « mdlOutputs » pour une composante passerelle où aucune adaptation de fonctionnalité ni de données n'est nécessaire.

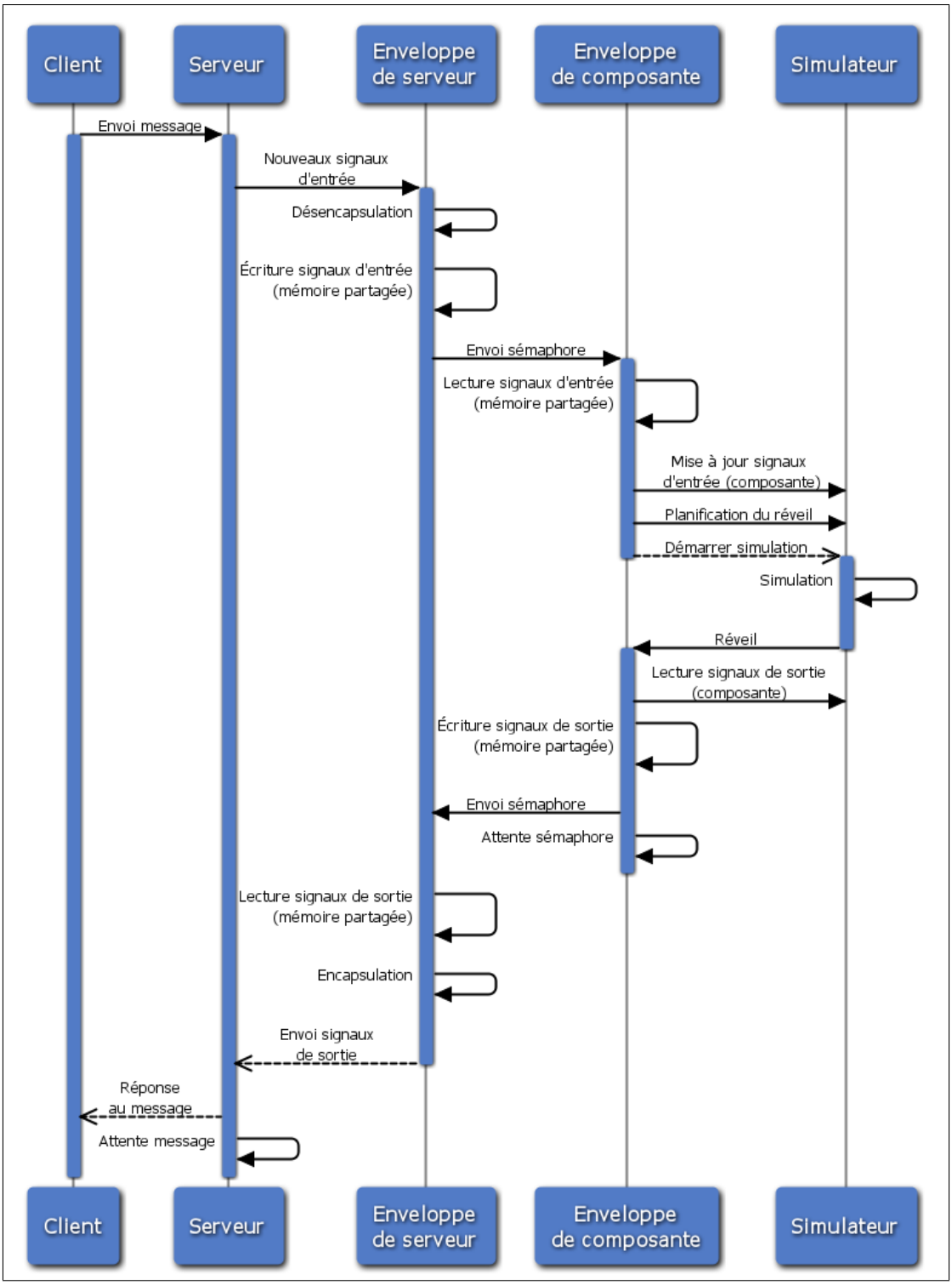

Figure 4.4 Flot de simulation avec les mécanismes POSIX.

```
/* Paramètres:
 * tid: Identificateur de tâche
 * *S: Pointeur sur la structure Simulink
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
  /* Récupération l'objet C++ du ORB */
  CommLink *c = (CommLink *) ssGetPWorkValue(S, 0);
  /* Les entrées du filtre */
  InputRealPtrsType u = ssGetInputPortRealSignalPtrs(S, 0);
  /* La sortie du filtre */
  real_T *y = (real_T *)ssGetOutputPortRealSignal(S, 0);

  /* Si l'initialisation de l'ORB client est un succès, tenter l'envoi des données. */
  if(c->orb)
    {
      try {
        /* Envoi des données à la vrai composante.
         * Ici update() est la méthode de mise à jour des signaux de la composante telle que définie
         * par le concepteur dans le fichier de description des interfaces.
         */
        CORBA::Boolean result = c->orb->update(
                                               *u[0], // Données entrantes
                                               *u[1], // Mode d'opération
                                               *u[2], // Nombre d'étages du filtre
                                               *y ); // Données sortantes
      } catch(const CORBA::Exception& ex) {
        /* Traitement d'une erreur de communication avec le serveur */
      }
    } else {
      /* Traitement d'une erreur d'initialisation de l'ORB client.
       * Il peut s'agir d'un problème de connexion au Naming Service CORBA ou d'une incapacité à
       * trouver un serveur implémentant l'interface désirée.
       */
    }
}
```

Extrait 4.1 Exemple 1 de méthode « mdlOutputs » d'une composante passerelle pour Simulink.

L'extrait de code 4.2 montre une méthode similaire où une adaptation de données est effectuée à l'aide de macros.

4.3 Simulateur de l'Open SystemC Initiative

Le langage SystemC est en fait une librairie C++. Ainsi, le simulateur de l'OSCI supporte nativement le langage C++. De plus, il offre des mécanismes de contrôle du flot de simulation.

```
/* Les deux macros faisant office de transactor.
 * Conversion du format flottant double précision à Q15.16 et vice versa.
 * Note à propos de float2fix: Afin de simplifier les calculs, l'arrondissement à l'entier le plus près est
 * volontairement omis. L'arrondissement se fait implicitement lors de la conversion vers en entier
 * i.e. la portion fractionnaire est éliminée.
 */
#define fix2float(v) (((real_T)(v))/(65536))
#define float2fix(v) ((int32_T)(v*65536))

static void mdlOutputs(SimStruct *S, int_T tid)
{
  /* Récupération l'objet C++ du ORB */
  CommLink *c = (CommLink *) ssGetPWorkValue(S, 0);
  /* Les entrées du filtre */
  InputRealPtrsType u = ssGetInputPortRealSignalPtrs(S, 0);
  /* La sortie du filtre */
  real_T *y = (real_T *)ssGetOutputPortRealSignal(S, 0);

  /* Si l'initialisation de l'ORB client est un succès, tenter l'envoi des données. */
  if(c->orb)
    {
      try {
        /* Envoi des données à la vraie composante.
         * Ici aussi update() est la méthode définie par le concepteur.
         */
        CORBA::Long output = 0.5;
        CORBA::Boolean result = c->orb->update( float2fix(*u[0]), // Données entrantes
                                                *u[1],  // Mode d'opération
                                                *u[2],  // Nombre d'étages du filtre
                                                output); // Données sortantes
        /* Mise à jour de la sortie */
        y[0] = fix2float(output);
      } catch(const CORBA::Exception& ex) {
        /* Traitement d'une erreur de communication avec le serveur */
      }
    } else {
      /* Traitement d'une erreur d'initialisation de l'ORB client.
       * Il peut s'agir d'un problème de connexion au Naming Service CORBA ou d'une incapacité à
       * trouver un serveur implémentant l'interface désirée.
       */
    }
}
```

Extrait 4.2 Exemple 2 de méthode « mdlOutputs » d'une composante passerelle pour Simulink.

De ce fait, il est simple d'implémenter l'enveloppe de composante et l'adaptateur d'outil dans une même classe.

L'essentiel du contrôle du flot de simulation est fait à l'aide de la méthode « sc_start ». Les signaux sont directement accessibles. Dans les cas où la fonctionnalité ou les données requièrent une adaptation, les fonctions C++ ou SystemC standards sont utilisées.

L'extrait de code 4.3 présente un exemple simple d'une méthode de l'enveloppe de composante. Cette méthode met à jour les signaux d'entrées de la composante, fait la simulation pour un cycle et renvoi la sortie au client à travers la dorsale. Dans cet exemple, aucune adaptation de fonctionnalité ou de données n'est nécessaire.

```
/* Modèle fonctionnel sans notion de temps d'une composante (untimed functional model).
 * Ici aussi update() est la méthode définie par le concepteur.
 */
CORBA::Boolean FirUtf_i::update(
                                        ::CORBA::Double dataIn,
                                        ::CORBA::Boolean runCfg,
                                        ::CORBA::Short nbrTaps,
                                        ::CORBA::Double_out dataOut
                                        )
  ACE_THROW_SPEC((CORBA::SystemException))
{
  try {
    /* Mise à jour des entrées */
    dataSignal.write(dataIn);
    cfgRunSignal.write(runCfg);
    nbrTapsSignal.write(nbrTaps);
    /* Simulation pour 1 cycle */
    sc_start();
    /* Mise à jour de la sortie */
    dataOut = dataOutSignal.read();
  } catch(exception& e) {
    /* Traitement d'une erreur de simulation. */
    return false;
  }
  return true;
}
```

Extrait 4.3 Exemple de méthode d'une enveloppe de composante pour le simulateur de l'OSCI.

4.4 Mentor Graphics ModelSim

ModelSim est un simulateur et débogueur de matériel supportant les langages Verilog, SystemVerilog, SystemC et VHDL. Il possède une interface permettant l'ajout d'extension à des

designs VHDL : le *Foreign Language Interface* (FLI). Cette interface permet au concepteur de lire et écrire des signaux de designs VHDL à partir des langages C, C++ ou Fortran. Notre code C++ utilisant l'interface FLI est compilé et utilisé sous forme de librairie partagée. Cette librairie partagée est chargée à travers une fausse composante VHDL comme l'illustre l'extrait de code 4.4.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity tester is
   port ([...]); -- Ports de la composante
end tester;

architecture a0 of tester is
   attribute foreign of a0 : architecture is
      "tester_init tester_fli.so;"; -- Appel de la fonction tester_init de la librairie partagée.
begin
end a0;
```

Extrait 4.4 Chargement de l'adaptateur d'outil pour ModelSim.

L'interface FLI permet de définir des méthodes de rappel qui s'exécutent lorsque surviennent des événements tels que l'élaboration du design, la mise à jour d'un signal, *etc*. L'interface FLI permet un contrôle très limité sur le flot de simulation se résumant essentiellement à l'arrêt de la simulation et la planification du réveil d'un processus VHDL. De par sa nature, il n'est donc pas possible de d'instancier et de contrôler le moteur de simulation à partir d'un autre processus.

Ainsi, par manque de contrôle sur le flot de simulation, l'interaction avec les objets CORBA se fait à l'aide des mécanismes POSIX. L'adaptateur d'outil et le serveur CORBA sont exécutés dans des processus distincts. Les sémaphores et la mémoire partagée POSIX servent pour la synchronisation et le partage de données.

Au moment de l'élaboration, un pilote (*driver*) est créé pour chacun des signaux se propageant dans le design. Un processus VHDL appelé « updatedesign » est ensuite créé à l'aide de la méthode « mti_CreateProcess ». Enfin, les méthodes « initposix » et « initdesign » sont appelées

pour initialiser les mécanismes POSIX et le design. Cette deuxième méthode, « initdesign », est également responsable de planifier le premier réveil du processus « updateoutput ». La planification du réveil d'un processus VHDL se fait à l'aide de la méthode « mti_ScheduleWakeup ». L'extrait de code 4.5 présente notre méthode appelée au moment de l'élaboration du design.

```
void tester_init(
                        mtiRegionIdT region,
                        char *param,
                        mtiInterfaceListT *generics,
                        mtiInterfaceListT *ports
                        )
{
  inst_rec_ptr ip;
  mtiInterfaceListT *p;

  ip = (inst_rec_ptr)mti_Malloc(sizeof(inst_rec));
  mti_AddRestartCB( mti_Free, ip ); // Libérer la mémoire lors d'un redémarrage

  num_ports = 0;
  for ( p = ports; p; p = p->nxt ) {
    /* Création de la liste de port contenant leur nom et leur taille. */
    [...]
    /* Création de pilote pour les entrées du design (sortie de notre adaptateur d'outil).
       Les sorties du design ne requièrent pas de pilote. */
    if( p->port_dir != MTI_DIR_IN ) {
      ip->drivers[num_ports] = mti_CreateDriver(p->u.port);
    }
    tester_ports[num_ports].number = num_ports;
    num_ports++;
  }
  /* Appeler notre fonction de mise à jour de la sortie à chaque événement. */
  ip->test_values = mti_CreateProcess((char*)"test", updatedesign, ip);

  /* Initialisation de notre mémoire partagée et de nos sémaphores POSIX */
  initposix();

  /* Initialisation du design */
  initdesign(ip);
}
```

Extrait 4.5 Exemple de fonction appelée à l'élaboration pour ModelSim.

La méthode « updatedesign » est au centre de l'adaptateur d'outil. Elle est responsable d'attendre la réception de signaux d'entrées destinés au design, de mettre à jour ce dernier, de planifier le prochain réveil du processus et de transmettre les signaux de sorties à l'enveloppe de serveur CORBA. Lors de son exécution, cette méthode commence par transmettre les signaux

de sorties correspondant aux résultats de la simulation nouvellement achevée et se met en attente du sémaphore. Lorsque le sémaphore est disponible, elle récupère les nouvelles données d'entrées dans la mémoire partagée, met à jour les signaux du design et planifie son prochain réveil. Ce cycle se répète lors des exécutions subséquentes. L'exécution de la méthode « updatedesign » au temps 0 est un cas particulier où la transmission des signaux de sorties n'est pas effectuée puisqu'aucune simulation n'a encore eue lieu.

Pendant que le processus de simulation attend la réception du sémaphore, le processus du serveur CORBA est toujours actif et est en attente d'un message provenant du client CORBA. Lorsque le client envoie un message contenant des signaux au serveur, l'enveloppe de serveur extrait les signaux du message, copie ceux-ci dans l'espace de mémoire partagée et active le sémaphore. La disponibilité de ce sémaphore réveille le processus de simulation et de ce fait la méthode « updatedesign ».

Les interfaces programmables de ModelSim et GHDL étant conceptuellement similaires, ce flot d'exécution est également celui de l'adaptateur d'outil pour GHDL.

4.5 GHDL

GHDL est un simulateur libre, basé sur GCC, pour le VHDL. Tel que mentionné ci-haut, GHDL possède certains points communs avec ModelSim. GHDL possède une interface pour étendre les fonctionnalités de base du simulateur : le *VHDL Programmable Interface* (VHPI). Il n'est pas possible de contrôler le flot de simulation de GHDL de manière satisfaisante pour notre application. Ainsi, les sémaphores et la mémoire partagée POSIX y servent également pour intégrer la dorsale CORBA à GHDL. Malgré que l'implémentation de VHPI ne soit que partielle, similairement au FLI de ModelSim, le VHPI de GHDL permet l'interaction avec un design à travers des méthodes de rappel.

La lecture ou l'écriture de signaux requiert l'association d'une méthode à la méthode de rappel « cbReadOnlySync[h] ». Une fois que cela est fait, notre méthode associée est appelée à chaque mise à jour d'un signal. Au premier appel de « cbReadOnlySync[h] », la méthode « initdesign »

initialisant le design est appelée. Lors des appels suivants, la méthode « updateinput » met à jour les signaux à l'aide de la fonction « vpi_put_value » alors que la méthode « updateoutput » lit les signaux de sortie avec la fonction « vpi_get_str » pour ensuite les copier dans la mémoire partagée.

```
int readonlysync (struct t_cb_data* cbdata)
{
  /* Variables de synchronisation, d'horloge et d'initialisation */
  static bool rising_edge = true;
  static bool init = true;
  static bool firstpass = true;

  /* Synchronisation de l'horloge: l'événement ReadOnlySynch est déclenché à chaque
      changement sur l'horloge que ce soit un front montant ou un front descendant.
      L'écriture et la lecture de données doivent se faire que sur les fronts montants. */
  if(rising_edge) {
    if( init ) {
      initdesign(); // Initialisation du design
      init = false;
    } else {
      /* À la première passe les entrées n'ont pas encore été mise à jour alors on
          ne fait pas de mise à jour de la sortie. */
      if( firstpass ) {
        firstpass = false;
      } else {
        updateoutput(); // Mise à jour des sorties
      }
      updateinput(); // Mise à jour des entrées
    }
  }
  /* Mise à jour de l'horloge */
  rising_edge = !rising_edge;

  return true;
}
```

Extrait 4.6 Exemple de méthode de rappel pour GHDL.

L'extrait de code 4.6 présente un exemple de méthode de rappel attachée à l'événement « cbReadOnlySync[h] ».

4.6 Conclusion

Ce chapitre montre qu'avec TAO, l'intégration de la dorsale se fait très bien là où les outils possèdent une interface C++[2]. Le cas le plus simple, illustré à la Figure 4.3(A), est celui des moteurs de simulation pouvant être instanciés à partir d'un processus externe tout en offrant un contrôle complet sur le flot de simulation. Dans le cas des simulateurs offrant peu ou pas de mécanismes pour contrôler la simulation, deux processus sont utilisés. Ainsi, le simulateur et le serveur CORBA sont exécutés dans des processus distincts tels qu'illustrés à la Figure 4.3(B). Dans ce dernier cas, les outils POSIX de communication inter processus (sémaphores et mémoire partagée) permettent d'efficacement gérer la synchronisation ainsi que l'échange d'informations. Tel que mentionné à la sous-section 4.1.3, n'importe quel mécanisme permettant l'échange d'informations entre processus est utilisable. Cependant, la mémoire partagée est la solution la plus performante.

Enfin, rappelons deux limitations importantes. Tout d'abord, telle que mentionné à la sous-section 4.1.2, la communication entre les ORBs est bloquante. Un serveur ne peut donc pas traiter un message entrant s'il n'a pas encore répondu au message précédent. Il en va de même pour un client *i.e.* le client ne peut envoyer un nouveau message tant qu'il n'a pas reçu la réponse à son message précédent. Ensuite, telle que mentionné à la sous-section 4.1.1 : par défaut, l'implémentation initiale ne partage pas l'horloge système entre les simulateurs. Ainsi, le temps n'est pas cohérent à travers le système. Tel que montré au chapitre 6, ces limitations n'empêchent en rien l'utilisation de cette dorsale pour faire la réalisation progressive d'un design en passant par plusieurs niveaux d'abstraction. Mais avant de passer à l'expérimentation, le chapitre 5 présente comment intégrer un design à cette dorsale de communication.

[2] Au chapitre 5, on verra que TAO possède un outil d'autogénération des serveurs et des clients CORBA en C++.

CHAPITRE 5

INTÉGRATION D'UN DESIGN ÉLECTRONIQUE NUMÉRIQUE

Les étapes préalables au démarrage de la simulation sont décrites dans la Figure 5.1 ci-dessous. La première étape pour vérifier ou concevoir un design électronique avec cet environnement est d'identifier l'outil qui sera utilisé comme maître de simulation. Deuxièmement, il faut déterminer une stratégie de nomenclature. Troisièmement, on crée un fichier IDL décrivant les composantes du design qui ne seront pas simulées à l'aide du maître de simulation. Quatrièmement, on génère les clients et serveurs liés aux composantes simulées hors du maître de simulation. Cinquièmement, toujours pour les composantes simulées hors du maître de simulation, on génère les artéfacts correspondants aux clients et serveurs. Sixièmement, on adapte les artéfacts à nos besoins. Enfin, on compile les clients et les serveurs pour ensuite démarrer le *Naming Service* et les serveurs.

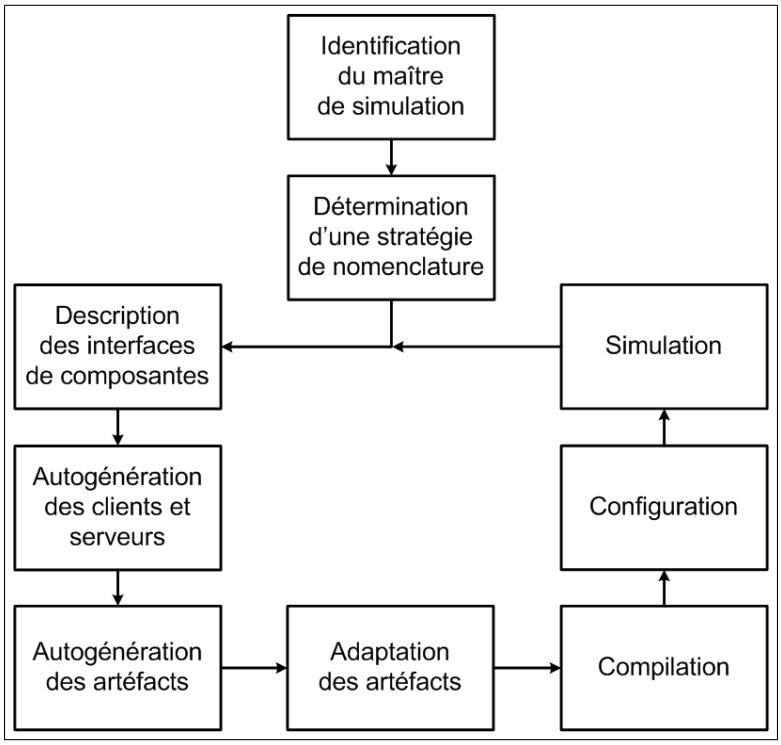

Figure 5.1 Flot d'intégration d'un design électronique numérique.

Tel qu'illustré sur la figure 5.1, lorsque les résultats de la simulation sont satisfaisants, on est prêt à raffiner notre modèle et à recommencer le flot d'intégration à partir de la description des interfaces.

Ce flot d'intégration ainsi que l'exemple de scénario ci-dessous furent l'objet d'une présentation lors de la séance d'affiches du SYTACom-S en juillet 2008 [22].

5.1 Exemple de scénario

Pour faciliter la compréhension de ce chapitre, on reprend le flot de vérification traditionnel présenté à la Figure 4.1 (p.25) du chapitre précédent. Dans ce scénario, illustré à la Figure 5.2, une première version algorithmique d'un filtre numérique à réponse impulsionnelle finie est développée à l'aide de MATLAB. Par la suite, une deuxième et une troisième versions du filtre sont conçues au niveau système. La deuxième version est un modèle fonctionnel sans notion de temps utilisant une représentation de nombres à point flottant. La troisième version est également un modèle fonctionnel, mais cette fois avec une notion de temps et une représentation de nombres à point fixe. Enfin, une quatrième version du filtre est conçue au niveau transfert de registres avec une représentation de nombres en point fixe. Le filtre au niveau algorithmique simulé avec MATLAB sert tout au long de la conception comme modèle de référence.

Étant un outil algorithmique avec des boîtes à outils graphiques, MATLAB est un outil de choix pour la génération et la validation des données. C'est donc ce dernier qui est utilisé pour générer les données d'entrées des filtres ainsi que pour comparer les données de sorties des filtres. Puisque MATLAB est au centre de ce scénario, il est utilisé comme maître de simulation. Ainsi, MATLAB a besoin de trois composantes passerelles (clients CORBA) faisant le pont avec les outils simulant les trois autres versions du filtre (serveurs CORBA).

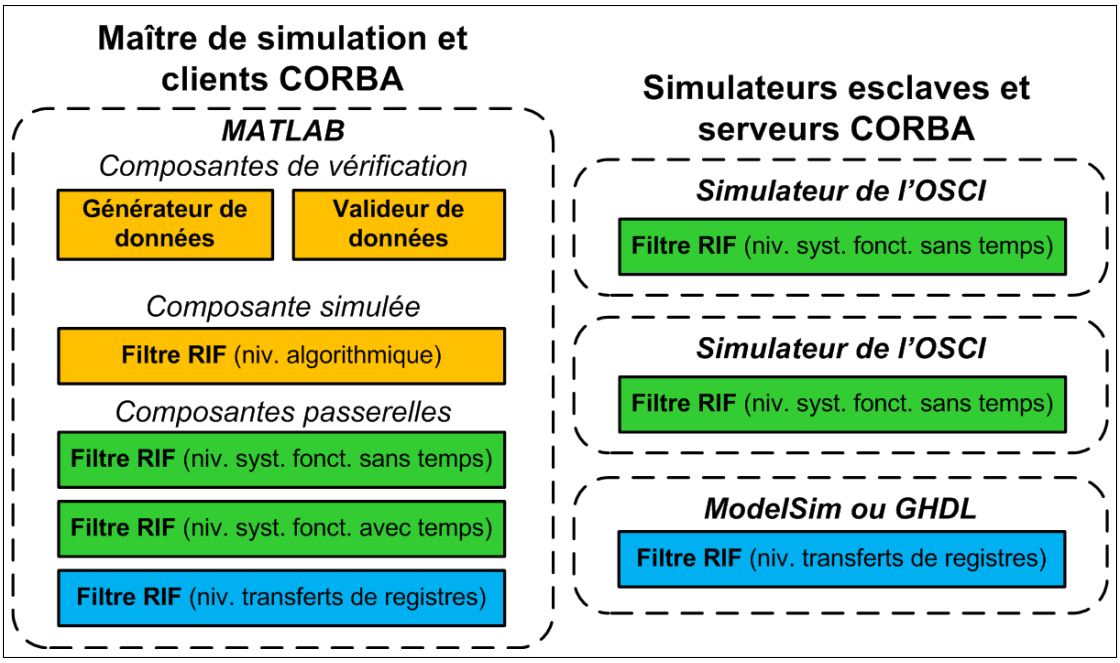

Figure 5.2 Exemple de scénario d'intégration.

5.2 Choix du maître de simulation

L'architecture de la dorsale de communication proposée permet l'utilisation de n'importe quel outil comme maître de simulation. Bien qu'il n'y ait pas de règle absolue pour le choix du maître de simulation, voici quelques questions pouvant guider le choix de ce dernier :

– Quel est l'outil avec lequel il y a le plus d'interaction ?
– Quel est l'outil dominant *i.e.* exécutant la majorité des modèles ?
– Quel est le profil des usagers utilisant l'environnement de conception et de vérification ?
– Dans le cas où la combinaison des simulateurs varie, quel est l'outil présent tout au long de la réalisation du design ?
– Est-ce que l'outil pressenti comme maître de simulation possède des mécanismes de contrôle du flot d'exécution ?

Considérant que nos utilisateurs cibles font beaucoup de traitement de signal, qu'ils sont accoutumés à MATLAB et que cet outil est utilisé tout au long de la réalisation, notre implémentation actuelle ne couvre que MATLAB/Simulink. Le choix du maître de simulation revient à l'utili-

sateur de nos travaux, mais requiert de la programmation dans le cas où le choix du maître de simulation n'est pas MATLAB/Simulink.

Notez qu'il serait possible de créer un maître de simulation standard avec aucune autre fonction que de créer l'environnement. Ce maître de simulation pourrait être générique et réutilisable d'un projet à l'autre. Dans un contexte plus large cela serait un avantage puisque MATLAB, bien qu'il soit bien pour le traitement de signal, est assez pauvre pour des systèmes plus complexes.

5.3 Stratégie de nomenclature

Avant de décrire les composantes à l'aide de fichiers IDL, il faut déterminer la nomenclature qui sera utilisée. Il n'y a pas de recette parfaite couvrant tous les besoins, cependant on distingue deux cas fréquents. Le premier est l'instanciation multiple d'un même modèle d'une composante et le deuxième est l'utilisation simultanée de multiples modèles d'une même composante.

Pour le premier cas, l'instanciation multiple d'un même modèle d'une composante, il est recommandé de décrire qu'une seule interface et une seule méthode dans le fichier IDL et d'utiliser plusieurs contextes de nommage CORBA pour faire la distinction entre les instanciations. À titre d'exemple, on pourrait associer un contexte de nommage CORBA à chacun des ordinateurs utilisés pour exécuter des serveurs CORBA. Le contexte de nommage d'un serveur est défini dans l'enveloppe de serveur. L'extrait de code 5.6 (p.51) contient un exemple où le contexte de nommage est « CCBExample ».

Dans le deuxième cas, l'utilisation simultanée de multiples modèles d'une même composante, il est plutôt suggéré de définir un nom d'interface unique à chacun des modèles de la composante, mais de conserver le même nom de méthode. Indistinctement des signaux d'entrées et de sorties utilisées qui peuvent varier d'un niveau d'abstraction à l'autre, cela permet de clairement identifier le niveau d'abstraction utilisé et de réduire la possibilité d'erreur de manipulation humaine. Par exemple, l'interface d'un filtre numérique à réponse impulsionnelle

finie implémenté au niveau d'abstraction algorithmique pourrait s'appeler « FirHighLevel » alors que l'implémentation du même filtre au niveau d'abstraction transfert de registres pourrait s'appeler « FirLowLevel ». L'extrait de code 5.1 de la section 5.4 illustre cet exemple. Dans cet exemple, la méthode de mise à jour des signaux du filtre s'appelle « update ».

Il y a deux autres méthodes possibles, cependant on recommande de les éviter. La première est d'utiliser le même nom d'interface pour tous ses modèles, mais d'utiliser un nom de méthode différent par modèles. Ici le problème est que cette séparation n'est pas explicite à l'utilisateur des clients CORBA. C'est-à-dire que le nom de la méthode est spécifié dans l'enveloppe du client et non au moment de la configuration. La deuxième méthode à éviter est encore d'utiliser le même nom d'interface pour tous les modèles, mais d'utiliser plusieurs fichiers IDL. Ainsi, dans le cas où les signaux d'entrées et de sorties diffèrent, le *Naming Service* CORBA saura guider le client CORBA vers le serveur attendu par l'utilisateur. Cependant, dans le cas contraire, le *Naming Service* CORBA pointera le client CORBA vers le premier serveur CORBA de sa liste offrant cette interface.

Une fois la stratégie de nomenclature déterminée, on décrit l'interface de nos composantes dans un ou plusieurs fichiers IDL.

5.4 Description de composantes

Le design requiert au moins un fichier IDL décrivant l'interface des composantes. Ce fichier définit les signaux entrant et sortant des composantes qui ne sont pas simulées à l'aide du maître de simulation. Ce sont les composantes qui seront sujettes aux raffinements, à la migration vers un autre langage de modélisation ou au calcul distribué.

Le nombre de fichiers IDL requis dépend de la stratégie de nomenclature employée. Cependant, dans le cas où une des stratégies recommandées est utilisée, un seul fichier IDL par design est nécessaire puisque plusieurs composantes ainsi que plusieurs modèles d'une même composante peuvent être décrits au sein d'un même fichier. Référez-vous à l'extrait de code 5.1 pour

```
/* Modèle haut niveau avec entrées et sorties de données au format point flottant double précision. */
interface FirHighLevel
{
   boolean update(in double dataIn,
                  in boolean runCfg,
                  in unsigned short nbrTaps,
                  out double dataOut);
};

/* Modèle bas niveau avec entrées et sorties de données au format point fixe Q15.16. */
interface FirLowLevel
{
   boolean update(in long dataIn,
                  in boolean runCfg,
                  in unsigned short nbrTaps,
                  out long dataOut);
};
```

Extrait 5.1 Exemple de définition d'interface pour un filtre ou fichier IDL.

un exemple de fichier IDL décrivant l'interface de deux modèles d'un même filtre à réponse impulsionnelle finie paramétrique programmable.

Une fois que les interfaces des composantes sont décrites dans un ou plusieurs fichiers IDL, le script « ccb_gen » peut autogénérer les fichiers manquants nécessaires à l'intégration.

5.5 Autogénération des artéfacts

Pour faciliter l'utilisation de la dorsale de communication CORBA, nous avons créé un script Perl appelé « ccb_gen ». Il s'agit essentiellement d'une utilisation de la notion de gabarit, des expressions régulières étendues ainsi que du module « CORBA::IDLtree » [30]. L'extrait de code 5.2 présente une des fonctions de « ccb_gen » où une S-Function pour MATLAB/Simulink est créée à partir d'un gabarit. L'extrait de code 5.3 montre des fragments du gabarit de *S-Function*.

« ccb_gen » génère automatiquement certains fichiers dont :

– Les enveloppes de composante et leur entête
– Les enveloppes de client
– Les enveloppes de serveur

```
sub insert_sfunc($$) {
    # 2 arguments: gabarit et nom de l'interface
    my ($file, $if) = @_;
    die "$file est introuvable!\n" unless( −e $file );
    # Lecture du gabarit
    open(FH, $file);
    my @lines = <FH>;
    close(FH);
    # Insertion de code dans le gabarit
    foreach(@lines) {
        # Nom de l'interface
        $_ =~ s/%%interface_name%%/$if/g;
        # Initialisation des ports d'entrée
        $_ =~ s/%%in_ports%%/$inportsinit/g;
        # Initialisation des ports de sortie
        $_ =~ s/%%out_ports%%/$outportsinit/g;
        # Récupération des ports de l'objet Simulink
        $_ =~ s/%%ports_binding_init%%/$portsbindinginit/g;
        # Association des signaux de l'objet CORBA avec les ports de la S−Function
        $_ =~ s/%%ports_binding%%/$portsbinding/g;
    }
    # Retourne une S−Function pour MATLAB/Simulink (enveloppe de composante)
    return join("",@lines);
}
```

Extrait 5.2 Fonction de « ccb_gen » créant une *S-Function* à partir d'un gabarit.

– Les applications serveur

– Les interfaces des composantes

Ces fichiers sont générés pour tous les simulateurs supportés en appelant le script avec le chemin vers le fichier IDL en paramètre tel qu'illustré dans l'extrait de code 5.4. Ces fichiers requièrent peu ou pas de modifications. Aucune modification n'est requise lorsqu'un seul contexte de nommage CORBA est utilisé (voir la section 5.3 ci-haut) et qu'aucune adaptation de fonctionnalité et de données n'est requise. La section 5.6 suivante donne davantage de détails sur la modification des artéfacts.

Pour le simulateur de l'OSCI, la version initiale de l'interface de composante est autogénérée par l'outil TAO-IDL. Elle est ensuite modifiée par le script « ccb_gen » pour y ajouter une méthode appelée à l'élaboration de la composante. L'extrait de code 5.5 présente un exemple complet d'interface pour une composante appelée FirUtf.

```
static void mdlOutputs(SimStruct *S, int_T tid)
{
  /* Récupération de l'objet C++ du ORB */
  CommLink *c = (CommLink *) ssGetPWorkValue(S, 0);
  /* Récupération des pointeurs vers les ports de la S-Function */
  %%ports_binding_init%%

  if(c->orb)
    {
      try {
        /* Envoi des signaux au serveur %%interface_name%% */
        %%ports_binding%%
      }
      catch(const CORBA::Exception& ex) {
        static char msg[256];
        sprintf(msg, "(%%interface_name%%) Exception CORBA: %s", ex._name());
        ssSetErrorStatus(S, msg);
      }
    }
  else
    {
      ssSetErrorStatus(S, "(%%interface_name%%) Incapable d'initialiser l'ORB.");
    }
}
```

Extrait 5.3 Fragments du gabarit de *S-Function* pour « ccb_gen ».

```
./ccb_gen.pl --idl /tmp/FirUtf.idl
```

Extrait 5.4 Exemple d'appel du script Perl « ccb_gen ».

5.6 Adaptation des artéfacts

Comme mentionné à la section précédente, les artéfacts autogénérés par le script « ccb_gen » peuvent requérir quelques modifications pour s'intégrer à notre environnement. Par exemple, dans les cas où plusieurs contextes de nommage CORBA sont utilisés, une modification aux enveloppes de serveur est nécessaire afin de changer le contexte par défaut « CCBExample » par celui qui est désiré.

Similairement, dans le cas où une adaptation de fonctionnalité et de données est à effectuer, il est envisageable de la faire dans l'une ou l'autre des enveloppes. Cela dit, pour plus de cohérence, il est recommandé de le faire dans l'enveloppe de composante ou de composante

```
#ifndef FIRUTF_I_H_
#define FIRUTF_I_H_

#include "FirUtfS.h"

class FirUtf_i : public virtual POA_FirUtf
{
   virtual CORBA::Boolean update (
                                          ::CORBA::Double data,
                                          ::CORBA::Boolean select,
                                          ::CORBA::Short ordre,
                                          ::CORBA::Double_out Y
   )
   ACE_THROW_SPEC((CORBA::SystemException));

public:
   /* Méthode pour l'élaboration de la composante */
   void init_systemc(void);

};
#endif /* FIRUTF_I_H_ */
```

Extrait 5.5 Exemple d'interface de composante pour le simulateur de l'OSCI.

passerelle. Pour l'adaptation de données, on recommande de le faire dans l'enveloppe de la composante passerelle afin que l'interface décrite dans le fichier IDL soit facilement associable à la composante réelle. Pour ce qui est de l'adaptation de fonctionnalité, on recommande plutôt de le faire dans l'enveloppe de la composante puisque cette adaptation peut impliquer un flot de simulation particulier *e.g.* respect d'une séquence de signaux s'étalant sur cinq cycles afin d'obtenir un signal de sortie valide.

5.6.1 Composantes passerelles et clients CORBA

Les composantes passerelles se font passer pour de vraies composantes. Dans les faits, elles transmettent les requêtes à un client CORBA qui les retransmet à un serveur CORBA. C'est ce dernier qui est responsable de fournir le comportement fonctionnel de la composante à simuler. Les composantes passerelles sont composées de deux éléments : une enveloppe de composante et une enveloppe de client. Le tableau 5.1 donne un exemple où la composante décrite dans le fichier IDL s'appelle FirUtf.

Tableau 5.1 Fichiers pour l'intégration de composantes passerelles avec MATLAB/Simulink comme maître de simulation

Élément d'intégration	**Fichier**
Entête de l'enveloppe de client	FirUtfC.h
Enveloppe de client	CorbaComm.cpp
Enveloppe de composante	SfuncFirUtf.cpp

Enveloppes de composante

La structure de base d'une enveloppe de composante est autogénérée à l'aide du script « ccb_gen ». Pour l'essentiel, une enveloppe de composante fait la correspondance entre le ou les signaux fournis par le maître de simulation et les signaux de la composante. Elle peut également effectuer une adaptation de fonctionnalité ou de données si le changement de niveau d'abstraction le requiert. Dans les cas où ce n'est pas nécessaire d'effectuer une adaptation, aucune modification n'est nécessaire. Les macros « fix2float » et « float2fix » de l'extrait de code 4.2 donne un exemple simple d'adaptation de données.

Enfin, bien que ce soit l'enveloppe de client qui ait besoin des trois paramètres CORBA nécessaires à la simulation, ces derniers sont spécifiés par l'enveloppe de composante. Rappelons que ces paramètres sont l'emplacement du *Naming Service* CORBA, le contexte de nommage CORBA (*naming context*) et le nom de l'interface. Il est recommandé de les spécifier à l'aide de l'interface graphique avant l'exécution de la simulation (voir la section 5.8 et la Figure 5.3 de la page 54).

Enveloppes de client

Aucune modification n'est nécessaire aux enveloppes de client puisque les paramètres pouvant varier peuvent être spécifiés dynamiquement (voir section 5.8). Notez cependant que le script « ccb_gen » actuel ne supporte que MATLAB/Simulink.

5.6.2 Simulateurs esclaves ou serveurs CORBA

Bien que l'intégration d'une composante à la dorsale de communication dépend du simulateur utilisé, « ccb_gen » se charge de générer les fichiers requis à partir des gabarits qui lui sont offerts. Ainsi, les modifications requises peuvent être généralisées pour les trois simulateurs actuellement supportés : ModelSim, GHDL et le simulateur de l'OSCI. Le tableau 5.2 donne un exemple des fichiers générés pour chacun des simulateurs où la composante décrite dans le fichier IDL s'appelle FirRtl.

Tableau 5.2 Fichiers pour l'intégration à ModelSim, à GHDL ou au simulateur de l'OSCI comme serveurs

Élément d'intégration	**Fichier**
Application serveur (ModelSim et GHDL seulement)	CorbaServerImpl.cpp
Enveloppe de serveur	CorbaServer.cpp
Entête de l'enveloppe de composante	FirRtl_i.h
Enveloppe de composante (Sim. de l'OSCI)	FirRtl_i.cpp
Enveloppe de composante (ModelSim)	tester_fli.cpp
Enveloppe de composante (GHDL)	tester_vpi.cpp

Enveloppes de serveur

L'enveloppe de serveur s'occupe de l'interaction avec l'ORB. À partir du code autogénéré par le script « ccb_gen » (voir l'extrait de code 5.6), une seule modification peut être nécessaire : la modification du contexte de nommage CORBA.

Applications serveur et enveloppes de composante

Seul ModelSim et GHDL possèdent une application serveur. En effet, tel que mentionné au chapitre 4, les simulateurs ModelSim et GHDL exécutent le serveur CORBA dans un processus séparé du simulateur.

L'application serveur est autogénérée par le script « ccb_gen ». Elle contient trois méthodes : deux pour les mécanismes POSIX et une pour le passage des objets CORBA. Les deux pre-

```
#include "FirRtl_i.h"
#include <orbsvcs/CosNamingC.h>
#include <iostream>

int main( int argc, char *argv[] )
{
  try {
    /* Initialisation de l'ORB. */
    CORBA::ORB_var orb = CORBA::ORB_init( argc, argv );
    /* Obtention d'un pointeur sur la racine de l'adaptateur d'objet portable (POA). */
    CORBA::Object_var obj = orb->resolve_initial_references( "RootPOA" );
    PortableServer::POA_var poa = PortableServer::POA::_narrow( obj.in() );
    /* Activation du gestionnaire de POA. */
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    mgr->activate();
    /* Recherche du Naming Service. */
    obj = orb->resolve_initial_references("NameService");
    CosNaming::NamingContext_var root = CosNaming::NamingContext::_narrow(obj.in());
    if (CORBA::is_nil(root.in())) { [...] } /* Aucun Naming Service de trouvé. */
    /* Association à un contexte de nommage et création de ce dernier si nécessaire. */
    CosNaming::Name name;
    name.length( 1 );
    name[0].id = CORBA::string_dup("CCBExample"); /* Contexte de nommage. */
    try { CORBA::Object_var dummy = root->resolve(name); }
    catch(const CosNaming::NamingContext::NotFound&) { /* Le contexte n'existe pas encore. */
      CosNaming::NamingContext_var dummy = root->bind_new_context( name );
    }
    /* Association à un nom de composante; création et activation de l'objet de la composante. */
    name.length( 2 );
    name[1].id = CORBA::string_dup("FirUtf");
    FirUtf_i servant;
    PortableServer::ObjectId_var oid = poa->activate_object(&servant);
    obj = poa->id_to_reference(oid.in());
    FirUtf_var comp_obj = FirUtf::_narrow(obj.in());
    root->rebind(name, comp_obj.in());
    /* Élaboration de la composante. */
    servant.init_systemc();
    /* Démarrage et destruction de l'ORB. */
    orb->run();
    orb->destroy();
  }
  catch(const CORBA::Exception& ex) { [...] } /* Exception CORBA, retourner 1. */
  return 0; /* Pas d'erreur. */
}
```

Extrait 5.6 Exemple d'enveloppe de serveur.

mières méthodes s'occupent de l'initialisation et de la destruction des sémaphores et des espaces de mémoire partagée. La troisième méthode gère l'échange des données avec l'adaptateur d'outil. Il faut faire attention à ce que les noms de sémaphores et d'espaces de mémoire

partagée entre l'application serveur et l'enveloppe de composante soient uniques. En effet, lorsqu'on exécute, sur un même ordinateur, plusieurs instances d'une application serveur pour un même simulateur et pour une même composante, les valeurs définies par défaut doivent être changées pour éviter les conflits. Ce cas d'usage est commun lorsqu'une composante est paramétrisable. Par exemple, un filtre numérique paramétrique peut avoir une première instance où il est configuré en filtre passe-bas et une deuxième instance où il est configuré en passe-haut.

Pour modifier le nom des sémaphores et des espaces de mémoire partagée, il faut modifier l'application serveur et l'enveloppe de composante. Dans les deux cas, la variable à modifier s'appelle « posix_name_suffix ». À titre d'exemple, l'extrait de code ci-dessous montre un cas où la valeur de la variable a été changée pour l'application serveur.

```
void FirRtl_i::init_shm(void)
{
  /* Définition du nom des variables POSIX. */
  posix_name_prefix = "ms"; /* Préfixe pour ModelSim. */
  posix_name_suffix = "firrtl_lp"; /* Variable à changer. Était firrtl. */
  posix_app_var = new char[strlen(posix_name_prefix)+strlen(posix_name_suffix)+6];
  posix_lib_var = new char[strlen(posix_name_prefix)+strlen(posix_name_suffix)+6];
  strcpy(posix_app_var, posix_name_prefix);
  strcat(posix_app_var, "_app_");
  strcat(posix_app_var, posix_name_suffix);
  strcpy(posix_lib_var, posix_name_prefix);
  strcat(posix_lib_var, "_lib_");
  strcat(posix_lib_var, posix_name_suffix);

  /* Initialisation des sémaphores et des espaces de mémoire partagée. */
  ssem = sem_open(posix_app_var, O_RDWR);
  if( ssem == SEM_FAILED ) perror("sem_open");
  csem = sem_open(posix_lib_var, 0);
  if( csem == SEM_FAILED ) perror("sem_open");
[...]
```

Extrait 5.7 Modification du nom des sémaphores et des espaces de mémoire partagée pour une application serveur.

Enfin, lorsqu'on désire faire une adaptation de fonctionnalité ou de données dans l'enveloppe de composante du côté serveur (*i.e.* l'adaptation n'est pas prise en charge du côté de la composante passerelle), il faut insérer cette adaptation nous-mêmes. En effet, le script « ccb_gen » n'a pas une connaissance suffisante de la composante et de nos intentions pour le faire lui-même.

5.7 Compilation

L'environnement contient plusieurs librairies et applications et leur compilation se fait séparément et diffère en fonction du système d'exploitation et du compilateur utilisés. Sous GNU/-Linux ou autres implémentations de Unix, l'utilisation des GNU Autotools et de GCC est la norme. Nous y avons donc opté pour cette solution. Dans le cas où le système d'exploitation est une des saveurs de Microsoft Windows, nous utilisons Microsoft Visual Studio 2005 et le compilateur y étant associé. Au choix, on peut également compiler les *S-Functions* directement à la ligne de commande de MATLAB.

Une fois que la compilation de toutes les librairies et applications est terminée, il faut configurer les clients CORBA.

5.8 Configuration

L'étape de configuration consiste à faire le lien entre les composantes passerelles et les serveurs. Essentiellement, deux mécanismes sont disponibles soit le nom de l'interface et le contexte de nommage (*naming context*). Plusieurs conventions de nommage sont possibles.

À titre d'exemple, là où le nom et la signature de l'interface d'une composante sont les mêmes pour les implémentations à différents niveaux d'abstraction, les contextes de nommage peuvent servir à sélectionner le niveau d'abstraction désiré. Une fois la convention établie, il faut faire pointer les composantes passerelles vers leur serveur respectif.

La configuration d'une composante passerelle se fait à partir de la boite de dialogue des paramètres de la *S-Function*. La Figure 5.3 montre un exemple spécifique où les paramètres sont les suivants :

– Protocole : IIOP ;
– Adresse IP du *Naming Service* : 142.137.20.230 ;
– Port : 2809 ;
– Contexte de nommage : CCBExample ;

– Interface : FirUtf.

Évidemment, ces paramètres doivent être modifiés pour s'adapter à votre environnement *e.g.* la machine hébergeant votre *Naming Service* CORBA n'a certainement pas l'adresse IP ci-dessus.

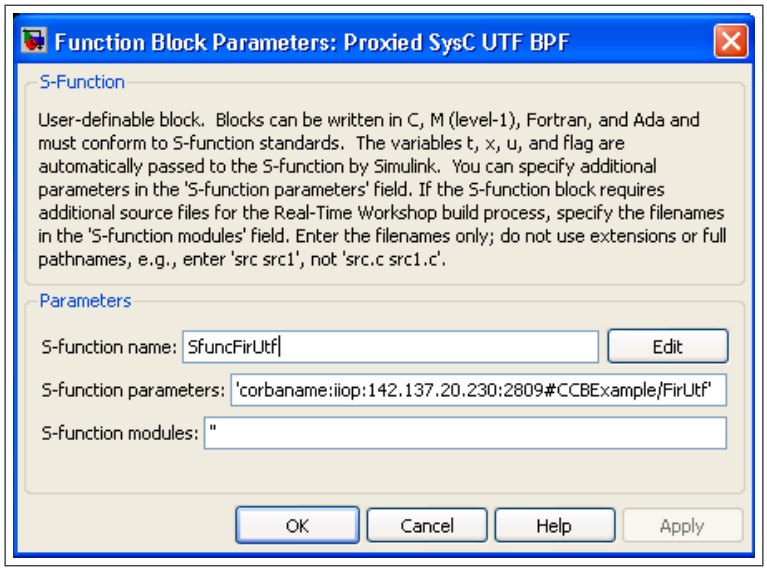

Figure 5.3 Configuration d'une composante passerelle.

Lorsque la composante passerelle est générée à partir du script « ccb_gen », le nom de la *S-Function* est « Sfunc<*nom de l'interface*> » *e.g.* si l'interface est FirRtl, la *S-Function* s'appelle « SfuncFirRtl ».

5.9 Exécution

L'exécution d'une simulation doit se faire dans un ordre précis. Le *Naming Service* CORBA doit être démarré en premier pour que les serveurs puissent s'y enregistrer lors du démarrage. Le maître de simulation contenant les clients doit être exécuté en dernier puisqu'il s'adresse au *Naming Service* pour trouver les serveurs.

5.9.1 *Naming Service* CORBA

Le *Naming Service* a minimalement besoin d'un protocole, d'une adresse et d'un port d'écoute[1]. Ils sont spécifiés à l'aide du paramètre « ORBEndpoint ». L'extrait de code 5.8 présente un exemple de démarrage du *Naming Service* écoutant sur le port 2809 de son adresse IP 142.137.20.230.

```
Naming_Service −ORBEndpoint iiop://142.137.20.230:2809 −ORBDottedDecimalAddresses 1
```

Extrait 5.8 Exemple de démarrage du *Naming Service*.

Dans cet exemple, le protocole utilisé est l'*Internet Inter-ORB Protocol* (IIOP). IIOP est l'implémentation du *General Inter-ORB Protocol* (GIOP) pour TCP/IP.

L'utilisation du paramètre « ORBDottedDecimalAddresses » permet de contourner les dépassements de délais causés par un serveur de nom de domaine ne supportant pas les requêtes inversées (*Reverse DNS lookups*)[2].

5.9.2 Simulateurs esclaves ou serveurs CORBA

Les différents serveurs CORBA supportent les mêmes paramètres. Pour certains simulateurs le démarrage se fait en deux temps. En effet, comme mentionné au chapitre 4, dans le cas de ModelSim et de GHDL le serveur et le simulateur sont deux processus séparés communiquant entre eux par les mécanismes POSIX.

Les paramètres minimaux à passer au serveur sont le protocole à utiliser ainsi que l'adresse et le port du *Naming Service*. L'extrait de code 5.9 présente un exemple de démarrage de ModelSim.

Comme mentionné à la section 4.4, l'adaptateur d'outil de ModelSim est une librairie partagée chargée à travers une fausse composante VHDL. Dans le cas de GHDL, l'adaptateur d'outil

[1] Référez-vous à la documentation de TAO pour en savoir plus sur les autres paramètres du *Naming Service*.
[2] Tel que celui de l'École de technologie supérieure.

```
# Démarrage du simulateur
vsim tb −do vsim.do &
# Démarrage du serveur CORBA
./corbaserver −ORBDefaultInitRef iiop://142.137.20.230:2809 −ORBDottedDecimalAddresses 1
```

Extrait 5.9 Exemple de démarrage de ModelSim.

est également une librairie partagée mais elle est chargée par la ligne de commande avec le paramètre « vpi ».

5.9.3 Maître de simulation

Une fois que les composantes passerelles sont configurées, il ne faut qu'ouvrir le modèle Simulink et démarrer la simulation. Bien que ce soit spécifique à MATLAB/Simulink, le principe serait le même pour un autre maître de simulation.

5.10 Conclusion

Ce chapitre montre la facilité avec laquelle les artéfacts requis pour l'intégration d'un design à la dorsale de communication proposée sont créés. En effet, notre script Perl « ccb_gen » permet l'automatisation presque entière de la création des artéfacts. Essentiellement, la partie n'étant pas automatisée est la création des routines d'adaptation de fonctionnalité ou de données. Bien que la création des routines d'adaptation de données pourrait être automatisée dans certains cas (*e.g.* conversion entre plusieurs formats de représentation de nombres), l'adaptation de fonctionnalité est plus compliquée et requiert une bonne connaissance du comportement de la composante.

Du côté du maître de simulation, la configuration des clients est facilitée par l'utilisation des mécanismes de Simulink permettant la configuration à l'exécution. Cela permet de changer les simulateurs utilisés sans avoir à recompiler le code source.

Le chapitre suivant montre l'utilisation de la dorsale de communication pour la réalisation progressive d'un design *i.e.* du niveau algorithmique jusqu'au niveau transfert de registres.

CHAPITRE 6

EXPÉRIMENTATION

L'intégration de la dorsale de communication CORBA à un environnement de conception électronique permet la vérification d'un système hétérogène ayant des composantes électroniques simulées sur des outils différents. Dans ce chapitre, la section 6.1 présente une étude de cas où un filtre numérique est progressivement implémenté alors que la section 6.2 compare la dorsale de communication CORBA proposée avec une solution commerciale existante.

Ce chapitre fit l'objet d'un article présenté lors de la première édition de la *Microsystems and Nanoelectronics Research Conference* tenue en marge du symposium annuel de CMC Microsystèmes en octobre 2008 [23].

6.1 Étude de cas : implémentation d'un filtre

Cette section présente les résultats suite à l'utilisation de la dorsale de communication proposée pour l'implémentation et la vérification d'un circuit numérique simple : un filtre numérique à réponse impulsionnelle finie paramétrique programmable de 40 étages. L'implémentation est faite à quatre niveaux d'abstraction différents *i.e.* algorithmique, fonctionnel sans notion de temps, fonctionnel avec notion de temps et transfert de registre.

6.1.1 Architecture

La Figure 6.1 montre l'architecture du filtre à implémenter. La valeur des coefficients $h[0]$ à $h[39]$ dépend de la nature du filtre. Ici, les coefficients utilisés sont ceux d'un passe-bande et furent obtenus à l'aide de l'outil de conception de filtres « fdatool » inclus dans MathWorks MATLAB.

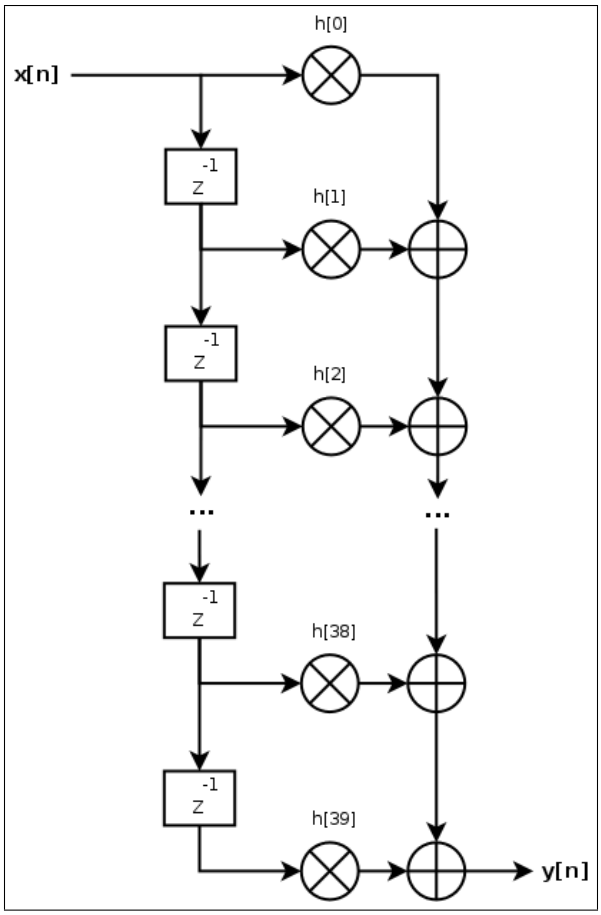

Figure 6.1 Diagramme du filtre numérique.

6.1.2 Formats de représentation des nombres

Dans cette étude de cas, deux formats de représentation des nombres sont utilisés : le point flottant et le point fixe. Dans le premier cas, il s'agit du format point flottant à double précision tel que défini par le IEEE [29]. Pour ce qui est du format point fixe, il s'agit du format *Q* tel que défini par l'entreprise Texas Instruments [45].

Format *Q*

La notation utilisée de ce format est *Qm.n* où :

– m est le nombre de bits utilisés pour représenter la portion entière du nombre. m est en format complément deux et exclut le bit le plus significatif. La portion entière est donc représentée sur $m+1$ bits ;
– n est le nombre de bits utilisés pour représenter la portion fractionnaire du nombre. n est également en format complément deux ;
– le bit le plus significatif fait partie de la portion entière et n'est pas comptabilisé ni dans m ni dans n[1].

Ainsi, pour un format *Qm.n* quelconque, le nombre est emmagasiné comme étant un entier signé de $m+n+1$ bits. Sa plage est donc $[-2^m;\ 2^m - 2^{-n}]$ et sa résolution, constante sur toute la plage, est de 2^{-n}.

Par exemple, soit le format *Q1.2*. Ce format a les caractéristiques suivantes :
– requiert $1+2+1=4$ bits ;
– plage de $[-2^1; 2^1 - 2^{-2}] = [-2{,}0; 1{,}75]$;
– résolution de $2^{-2} = 0{,}25$.

Donc, un nombre au format *Q1.2* peut prendre les valeurs : $\{-2{,}0; -1{,}75; ...; 0; ...; 1{,}5; 1{,}75\}$.

Soit le format *Q15.16* utilisé pour l'implémentation. Ce format a les caractéristiques suivantes :
– requiert $15+16+1=32$ bits ;
– plage de $[-2^{15}; 2^{15} - 2^{-16}] = [-32768{,}0; 32767{,}999984741]$;
– résolution de $2^{-16} = 1,5259 \times 10^{-5}$.

Conversion de format

La conversion du format point flottant au format *Q* et vice versa est simple à faire en logiciel. Pour passer du format point flottant au format *Qm.n*, on multiplie le nombre en point flottant par 2^n, on l'arrondit à l'entier le plus près et on le convertit au format entier signé (complément deux). Pour faire l'inverse, passer du format *Qm.n* au format point flottant, on convertit d'abord le nombre directement en point flottant et on le divise ensuite par 2^n. Puisque qu'il y a un

[1] Texas Instruments désigne ce bit comme étant le bit de signe malgré que le format est le complément deux.

arrondissement lors de la conversion du format point flottant au format *Q*, dans certains cas, cette opération n'est pas réversible.

Par exemple, soit la convertion du nombre au format point flottant $1{,}234$ vers le format *Q1.2* suivie d'une reconversion au format point flottant retourne la valeur $1{,}25$. La démarche est la suivante :

a. Format point flottant vers le format *Q1.2* : $1{,}234 \times 2^n = 1{,}234 \times 4 = 4{,}936 \Rightarrow 5$;

b. Format *Q1.2* vers le format point flottant : $5/2^n = 5/4 = 1{,}25 \neq 1{,}234$;

Comme l'illustre cet exemple, la conversion du format point flottant au format *Q* n'est pas toujours réversible.

6.1.3 Méthodologie de conception et de vérification

L'approche choisie est similaire à celle présentée à la Figure 3-4 de [11] *i.e.* une approche de haut en bas avec une réutilisation verticale du banc d'essai. La conception commence avec la mise en place d'un banc d'essai sous Simulink. Au départ, le banc d'essai contient un générateur et un valideur de données. On ajoute ensuite une première implémentation du filtre. Ce filtre de niveau algorithmique conçu avec un modèle Simulink sert de modèle de référence. Ensuite, on raffine notre modèle de filtre au niveau système purement fonctionnel pour graduellement se rendre à une implémentation au niveau transfert de registres (RTL) en VHDL.

Le format point flottant à double précision est utilisé pour le générateur de données, le modèle algorithmique, le modèle fonctionnel sans notion de temps ainsi que pour le valideur de données. Le format *Q15.16* est utilisé pour les modèles fonctionnel avec notion de temps ainsi que transfert de registres.

À chaque itération, le nouveau filtre est ajouté au banc d'essai reproduisant le flot de vérification traditionnel illustré à la Figure 4.1 de la page 25. Il est alors comparé au modèle de référence de niveau algorithmique ou à l'implémentation du niveau d'abstraction précédent.

6.1.4 Résultats

Après implémentations du filtre aux quatre niveaux d'abstraction, le modèle Simulink est tel qu'illustré à la Figure 6.2.

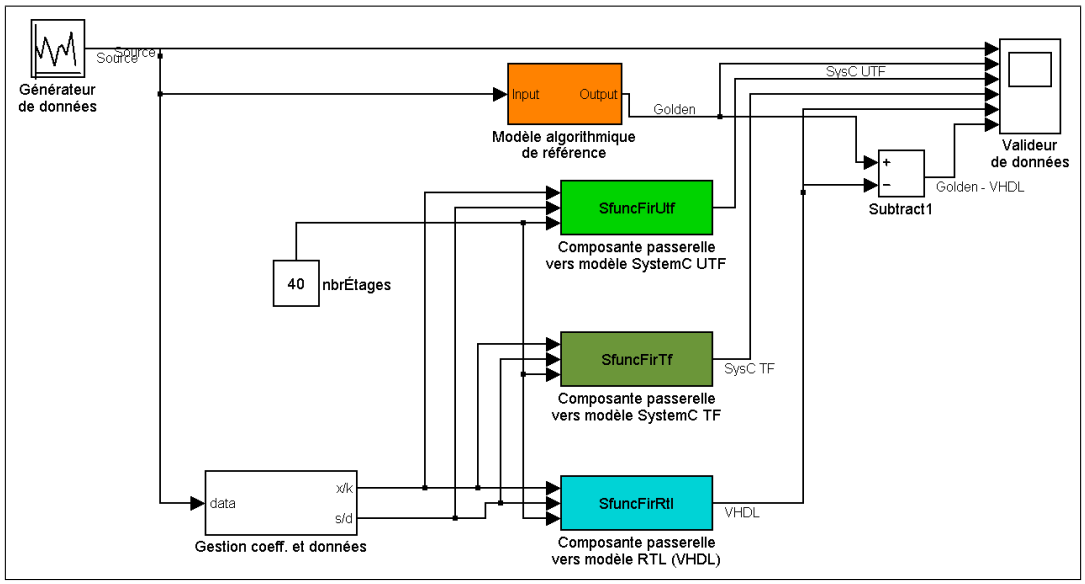

Figure 6.2 Modèle Simulink avec quatre niveaux d'abstraction.

Les résultats sont validés graphiquement. De haut en bas, la Figure 6.3 présente le signal d'entrée, la sortie des quatre implémentations du filtre et la différence entre le modèle de référence et la sortie du filtre au niveau RTL. Les sorties des filtres sont présentées dans l'ordre des niveaux d'abstraction : algorithmique, fonctionnel sans notion de temps, fonctionnel avec notion de temps et transfert de registre.

Notons que l'erreur observée a trois origines. Au début, *i.e.* entre 0 et $0{,}005$ seconde, les filtres reçoivent leurs coefficients et ne sont donc pas prêts à filtrer. Par la suite, *i.e.* entre $0{,}005$ et $0{,}01$ seconde, les filtres traitent les premières données, mais le résultat n'est pas encore exacte, car il dépend des données précédentes s'étant propagées dans les 40 étages du filtre. Enfin, pour le reste de la durée de la simulation l'erreur est de $\pm 1 \times 10^{-4}$. Il s'agit d'une erreur de quantification, car le modèle algorithmique utilise une représentation en point flottant

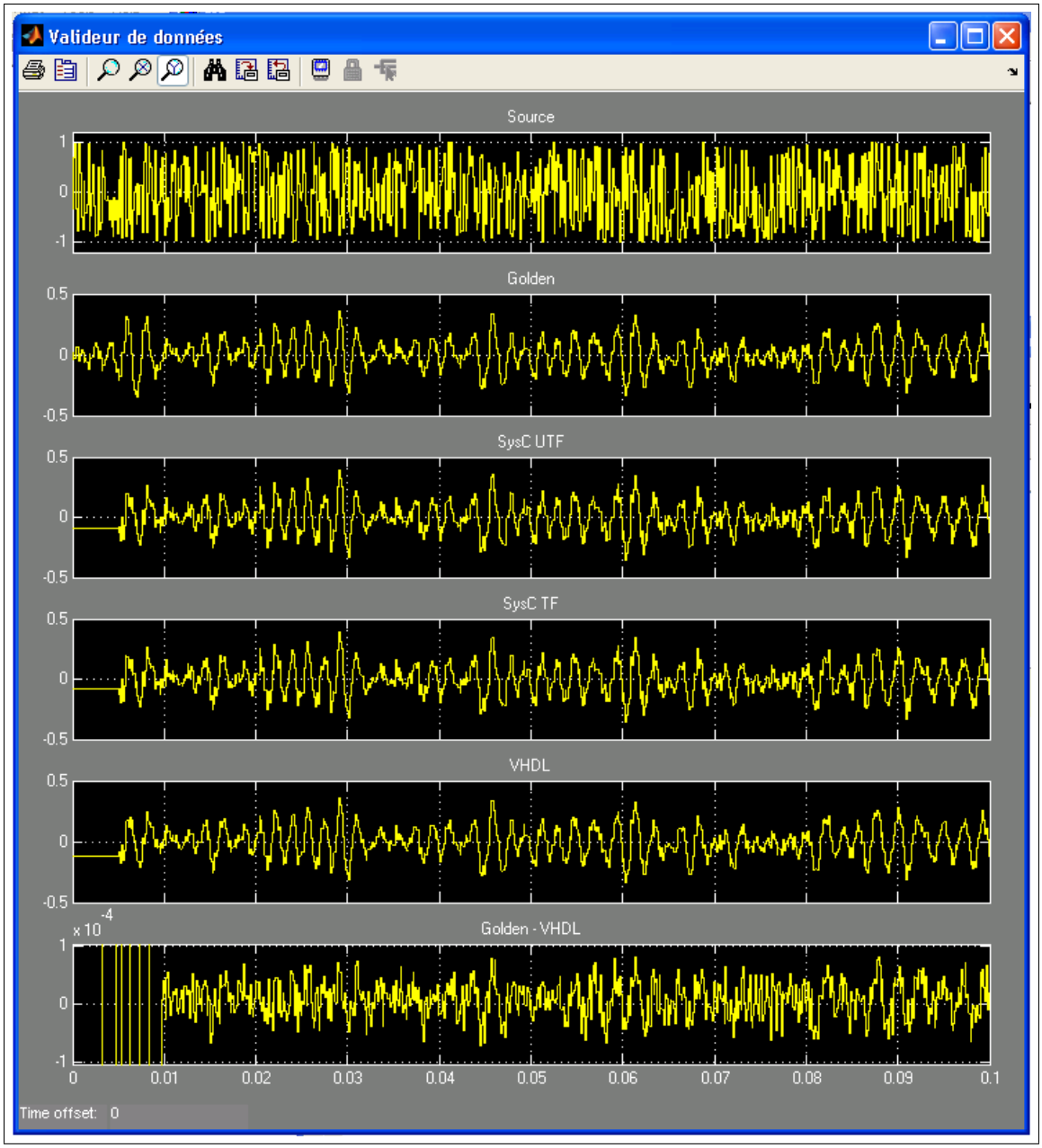

Figure 6.3 Résultats de la simulation des quatre implémentations du filtre.

double précision alors que l'implémentation RTL utilise la représentation *Q15.16* (résolution de $1{,}5259 \times 10^{-5} \Rightarrow$ erreur maximale théorique de $\pm 40 \times 1{,}5259 \times 10^{-5} = \pm 6{,}10352 \times 10^{-4}$).

6.2 Comparaison avec une solution commerciale

6.2.1 Présentation

MathWorks *EDA Simulator Link MQ*, anciennement *Link for ModelSim*, est une boite à outils permettant la cosimulation entre MATLAB et ModelSim. Bien que limité à ces deux outils, la comparaison de cette solution avec la dorsale proposée est significative car similaire en fonctionnalités.

6.2.2 Méthodologie

Le banc d'essai utilisé est similaire à celui de la section 6.1 dans le sens où c'est Simulink qui génère les signaux et qui les compare. L'implémentation au niveau transfert de registres du filtre numérique à réponse impulsionnelle finie paramétrique programmable de 40 étages est la même pour les deux possibilités. Il s'agit exactement du même module écrit en VHDL puisqu'aucune des deux solutions ne requiert de modifications au design. Les deux solutions partagent également un adaptateur de composante chargé de l'initialisation de la composante *i.e.* le chargement des coefficients du filtre. Notez que cet adaptateur pourrait être intégré une des enveloppes de composante mais puisque cette adaptation est requise pour tous les modèles, à l'exception de celui au niveau algorithmique, il est adéquat de l'implémenter dans le banc d'essai.

6.2.3 Intégration

Pour la dorsale CORBA, l'adaptation des données se fait à l'aide de macros dans l'enveloppe de composante. Pour MathWorks *Link for ModelSim*, l'adaptation des données et la synchronisation doivent être effectuées dans Simulink. De plus, MathWorks *Link for ModelSim* requiert un pas de simulation entier. Cela est pris en considération pour la comparaison.

Les Figures 6.4 et 6.5 présentent les modèles Simulink pour la solution commerciale et pour la solution proposée. Chaque solution utilise trois instanciations du modèle de filtre afin d'exécuter un simulateur par ordinateur (voir la sous-section suivante).

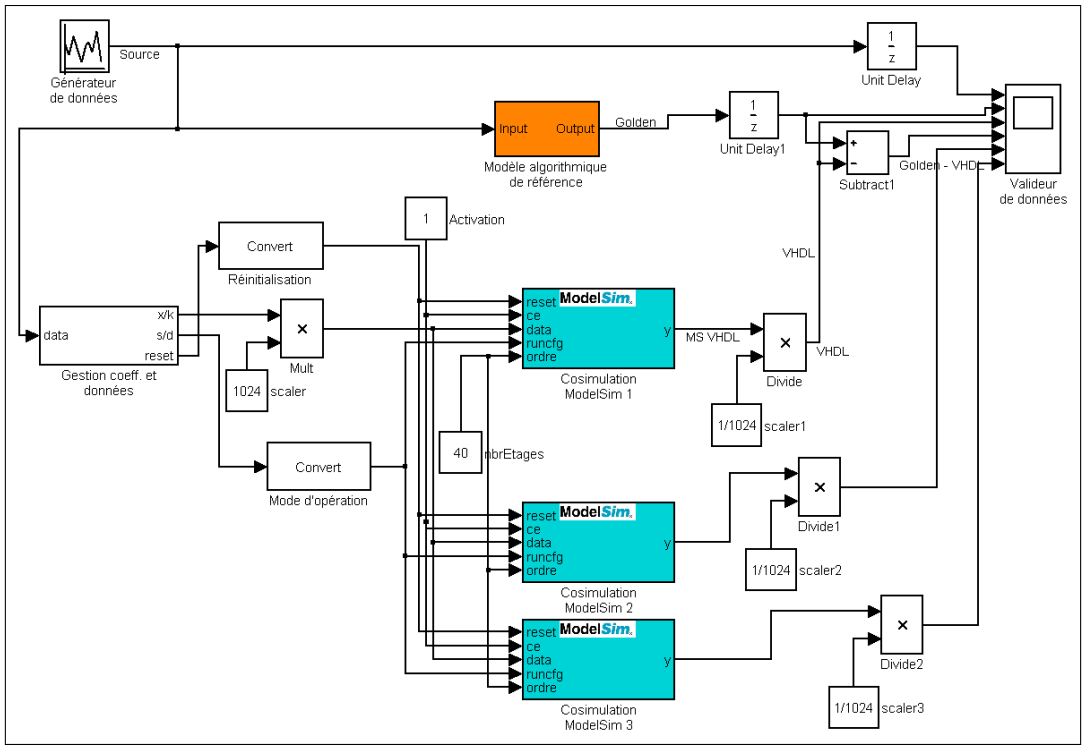

Figure 6.4 Modèle Simulink utilisant MathWorks *Link for ModelSim*.

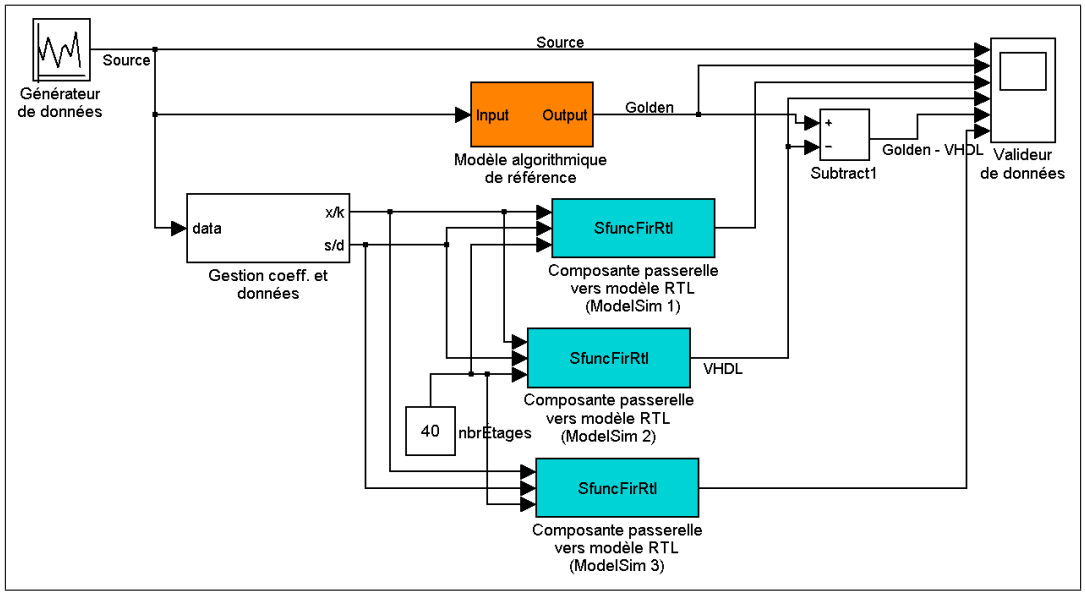

Figure 6.5 Modèle Simulink utilisant la dorsale de communication CORBA.

Il est à noter que lors de l'utilisation de la dorsale de communication CORBA, les signaux supplémentaires spécifiques à un niveau d'abstraction sont pris en charge par l'enveloppe de composante. Ils n'apparaissent donc pas dans le modèle Simulink.

6.2.4 Performance

Cette section présente des résultats obtenus avec la dorsale proposée et avec la solution commerciale de MathWorks. La vérification par simulation fut exécutée sur quatre ordinateurs en parallèle. MATLAB/Simulink fût exécuté sur la machine ayant MS Windows XP comme système d'exploitation tandis que les trois autres machines identiques exécutèrent un filtre numérique chacun. Le Tableau 6.1 présente davantage de précision sur les machines utilisées.

Tableau 6.1 Machines utilisées pour la vérification

Machine	**Système d'exploitation**	**Applications**
Intel Xeon E7525 3.6GHz (3GB RAM)	MS Windows XP SP2 (32bit)	MATLAB/Simulink 7.4.0
Dual Intel Quad Core Xeon E5405 2GHz (16GB RAM)	Ubuntu GNU/Linux 8.04 (64bit)	Mentor Graphics ModelSim 6.3f
Dual Intel Quad Core Xeon E5405 2GHz (16GB RAM)	Ubuntu GNU/Linux 8.04 (64bit)	Mentor Graphics ModelSim 6.3f
Dual Intel Quad Core Xeon E5405 2GHz (16GB RAM)	Ubuntu GNU/Linux 8.04 (64bit)	Mentor Graphics ModelSim 6.3f et *CORBA Naming Service*

Comme le montre le Tableau 6.2 ou la Figure 6.6, la différence de temps d'exécution entre l'utilisation de la dorsale de communication CORBA (CCB) et la solution commerciale *Link for ModelSim* de MathWorks (L4M) n'excède pas 20% lors de la simulation d'un filtre numérique à réponse impulsionnelle finie paramétrique programmable de 40 étages.

Cette dégradation des performances est majoritairement due à la latence inhérente aux mécanismes CORBA *e.g.* encapsulation et désencapsulation des messages. Dans un cas où on aurait beaucoup plus de composantes, on s'attend à ce que l'augmentation de la latence soit linéaire[2] telle que présentée dans les articles [36, 43].

[2] Avec les systèmes d'exploitation GNU/Linux, LynxOS, Solaris et VxWorks.

Tableau 6.2 Comparaison du temps d'exécution entre *Link for ModelSim* de MathWorks et la dorsale de communication CORBA

Temps simulé (s)	**Temps réel (s)**		**Différence**
	Link for ModelSim	Dorsale de comm. CORBA	
2,5	12,125	13,582	12,02%
5	23,493	26,865	14,35%
7,5	34,677	40,203	15,94%
10	45,995	54,761	19,06%
15	68,362	80,884	18,32%
25	113,285	135,188	19,33%
40	181,276	216,591	19,48%
55	248,892	296,968	19,32%
70	316,247	377,368	19,33%
85	384,831	459,675	19,45%
100	451,637	538,745	19,29%

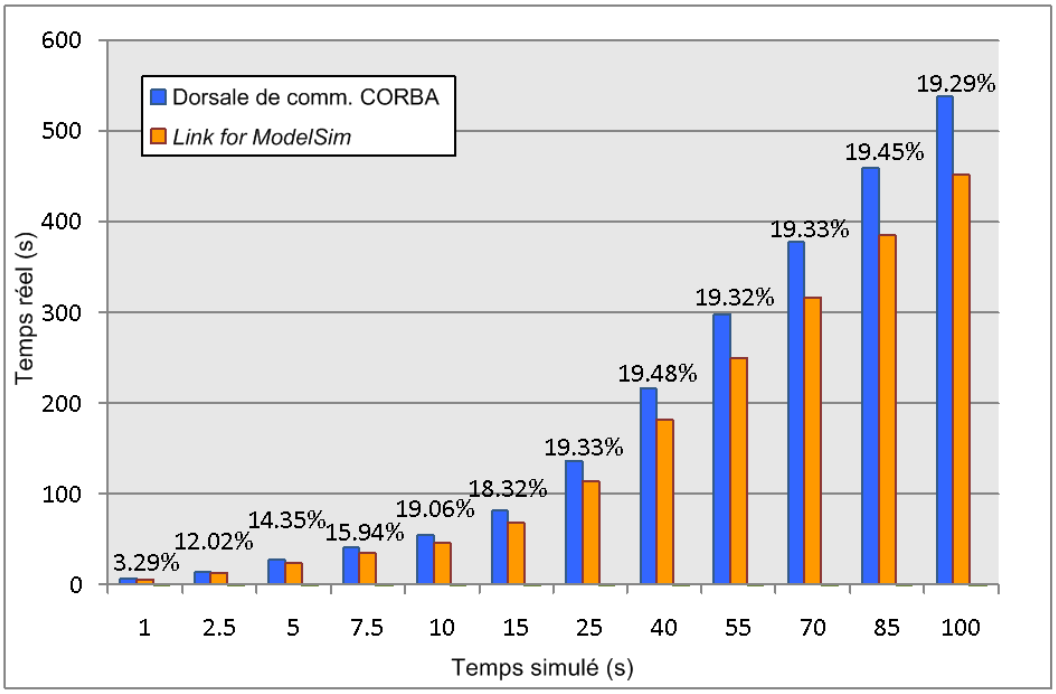

Figure 6.6 Comparaison du temps d'exécution entre *Link for ModelSim* de MathWorks et la dorsale de communication CORBA.

6.3 Conclusion

Bien que la composante réalisée à la section 6.1 soit simple, cette étude de cas montre qu'avec cette dorsale, MATLAB/Simulink devient utilisable pour la stimulation et la validation de composantes à des niveaux d'abstraction autre qu'algorithmique et exprimées dans des langages autres que ceux supportés par MATLAB/Simulink. Autrement dit, grâce à la dorsale de communication CORBA proposée, il est possible de faire de la vérification fonctionnelle d'un système hétérogène.

De plus, la dorsale de communication proposée permet la conception et la vérification d'un design en utilisant une méthodologie de haut en bas avec une réutilisation verticale du banc d'essai. Dans le cas présenté, l'adaptation de fonctionnalité est assumée par un modèle Simulink composé de deux scripts *M-file* simples. L'adaptation de données pour l'implémentation au niveau RTL est quant à elle intégrée à l'enveloppe de composante sous forme de macro.

La dégradation des performances, présentée à la section 6.2, semble significative. Cependant, il est attendu qu'elle devienne négligeable lorsque la communication entre les composantes d'un design est minimale. Autrement dit, cette différence devrait s'atténuer là où le temps nécessaire pour faire la simulation excède grandement le temps requis pour transmettre les signaux. Dans [24], Gokhale et al. montrent aux Figures 33 et 34 que la latence de TAO est quasi-constante[3] même si la taille des messages augmentent. Ainsi, il est également possible de diminuer l'importance de cette dernière en regroupant des signaux de temps différents en blocs. Notons également que contrairement à la solution commerciale de MathWorks, notre architecture générique de conception et de vérification permet l'intégration d'une large gamme d'applications. De plus, notre architecture permet un support, transparent à l'usager, de différents langages de modélisation et de niveaux d'abstraction.

Avec ses enveloppes de composantes et ses enveloppes de serveurs pouvant agir comme *transactor*, cet environnement de conception et de vérification offre au concepteur la possibilité de

[3] Avec la version optimisé du protocole IIOP. Cette version du protocole IIOP est incluse dans les versions de TAO parues après 1997.

faire du raffinement ciblé de composante. Le concepteur peut exprimer une composante dans un autre formalisme ou déplacer la composante sur un autre simulateur sur le même réseau et continuer à simuler le design comme un système cohérent comme si toutes les composantes étaient toutes exprimées au même niveau d'abstraction et roulaient dans le même simulateur. Enfin, le concepteur peut rapidement faire de l'exploration architecturale en modifiant les paramètres d'une composante passerelle pour la faire pointer sur une autre implémentation.

CONCLUSION

Telle que mentionné dans l'introduction, la contribution principale de ce mémoire est de proposer une dorsale de communication générique pour la conception et la vérification de systèmes complexes. Cette dorsale de communication se base sur une architecture d'objets distribués. Notre travail se distingue de [40] par l'utilisation de CORBA non seulement pour observer un système, mais aussi pour interagir avec celui-ci. De plus, notre intégration au simulateur de l'OSCI ne requiert aucune modification au noyau de ce dernier.

Notre architecture de conception et de vérification permet la cosimulation de composantes exprimées à des niveaux d'abstraction différents ainsi qu'exprimées à l'aide de langages de modélisation différents, simulées avec différents simulateurs ou outils sur diverses plateformes ou divers systèmes d'exploitation. Notre architecture est flexible de par sa nature et permet l'intégration de nouveaux outils telle qu'illustrée à la Figure 3.1 de la page 21. Cette architecture permet également l'ajout ultérieur de protocoles de communication.

La facilité d'intégration fait la promotion de bonnes pratiques de conception telles que l'exploration architecturale, la réutilisation de code, la vérification matérielle hâtive et le fin raffinement ciblé de composante d'un design. D'autre part, puisque l'implémentation actuelle supporte nativement TCP/IP, les simulateurs peuvent être répartis au sein d'un réseau TCP/IP pour permettre un calcul distribué. De plus, que ce soit avec le protocole TCP/IP ou autre, le *Hardware-in-the-loop* est possible.

L'autogénération presqu'entière des artéfacts contribue également à la simplicité d'utilisation en dissimulant une portion de la complexité de l'architecture de communication tout en diminuant de manière significative le temps requis pour l'intégration. Cela permet de se concentrer sur l'adaptation de fonctionnalité et de données.

La section 6.2 montre une perte de performance lorsqu'on compare notre solution avec une solution commerciale. D'une part, nous croyons que cette perte est un compromis acceptable considérant que notre dorsale est une solution générique par rapport aux solutions commer-

ciales à fort couplage. D'autre part, cette perte peut être largement compensée par l'utilisation de *Hardware-in-the-loop*. Enfin, n'importe quel simulateur ou outil peut être intégré à l'architecture proposée du moment qu'il possède une interface programmable ou un autre mécanisme permettant éventuellement l'intégration avec un ORB.

RECOMMANDATIONS

Les travaux futurs incluent l'évaluation de l'impact de la simulation distribuée d'un gros design, l'amélioration des performances et la migration vers un maître de simulation autre que MATLAB. La vérification avec *Hardware-in-the-loop* à l'aide d'un FPGA et du *OpenFusion Integrated Circuit ORB* (ICO) de PrismTech est également à considérer.

Une première idée pour améliorer les performances de la communication de manière significative serait de regrouper des signaux pour des temps différents en blocs. L'idée serait de reproduire le principe des salves de données couramment utilisé sur des bus de donnée tels que l'OPB.

Dans le cas où la notion de temps deviendrait une priorité, il serait essentiel de propager le temps dans les communications entre les simulateurs. Inévitablement, cela nécessite également de mettre à jour le temps système au minimum sur le simulateur maître.

BIBLIOGRAPHIE

[1] Adamopoulos, D.X., G. Pavlou et C.A. Papandreou: *Performance Evaluation of Distributed Object Platforms for Telecommunications Service Engineering Activities*. Signal Processing, Communications and Computer Science. Proceedings of the 4th IMACS/IEEE World Multiconference on Circuits, Systems, Communications and Computers (CSCC'00), pages 131–136.

[2] Adamopoulos, D.X., G. Pavlou, C.A. Papandreou et E. Manolessos: *Distributed Object Platforms in Telecommunications : A Comparison Between DCOM and CORBA*. British Telecommunications Engineering, 18(2) :43–49, 1999.

[3] Amar, Abdelkader: *Envrionnement [sic] fonctionnel distribué et dynamique pour systèmes embarqués*. Thèse de doctorat, Université des sciences et technologies de Lille, 2003.

[4] Amory, A., F. Moraes, L. Oliveira, N. Calazans et F. Hessel: *A Heterogeneous and Distributed Co-simulation Environment [Hardware/Software]*. Dans *Integrated Circuits and Systems Design, 2002. Proceedings. 15th Symposium on*, pages 115–120, 9-14 Sept. 2002.

[5] Atef, D., A. Salem et H. Baraka: *An Architecture of Distributed Co-simulation Backplane*. Dans *Circuits and Systems, 1999. 42nd Midwest Symposium on*, tome 2, pages 855–858vol.2, 8-11 Aug. 1999.

[6] Balarin, F. et R. Passerone: *Functional Verification Methodology Based on Formal Interface Specification and Transactor Generation*. Design, Automation and Test in Europe, 2006. DATE '06. Proceedings, 1 :1–6, March 2006.

[7] Bellard, F.: *QEMU, a Fast and Portable Dynamic Translator*. Dans *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[8] Benini, L., D. Bertozzi, D. Bruni, N. Drago, F. Fummi et M. Poncino: *SystemC Cosimulation and Emulation of Multiprocessor SoC Designs*. Computer, 36(4) :53–59, April 2003, ISSN 0018-9162.

[9] Bensley, E., P. Krupp, M. Squadrito, B. Thuraisingham et T. Wheeler: *Object-oriented Approach for Designing Evolvable Real-time Command and Control Systems*. Dans *Proceedings of the Workshop on Object-Oriented Real-Time Dependable Systems, Feb*, 1996.

[10] Bergeron, Janick: *Writing Testbenches using SystemVerilog*. Springer, 2006.

[11] Boland, Jean-François: *A Multi-Abstraction Level Platform for the Validation and Verification of Complex Digital Designs*. Thèse de doctorat, McGill University, 2007.

[12] Bombieri, N., N. Deganello et F. Fummi: *Integrating RTL IPs into TLM Designs Through Automatic Transactor Generation*. Design, Automation and Test in Europe, 2008. DATE '08, pages 15–20, March 2008.

[13] Buss, A. et L. Jackson: *Distributed Simulation Modeling : a Comparison of HLA, CORBA, and RMI*. Simulation Conference Proceedings, 1998. Winter, 1 :819–825 vol.1, Dec 1998.

[14] Chen, Yuanfeng, Pushan Tang, Jinmei Lai et Jiarong Tong: *A Universal Hierarchical FPGA Partitioning Algorithm*. ASIC, 2005. ASICON 2005. 6th International Conference On, 2 :811–814, Oct. 2005.

[15] Chung, M.K., S. Yang, S.H. Lee et C.M. Kyung: *System-level HW/SW Co-simulation Framework for Multiprocessor and Multithread SoC*. Dans *VLSI Design, Automation and Test, 2005.(VLSI-TSA-DAT). 2005 IEEE VLSI-TSA International Symposium on*, pages 177–180, 2005.

[16] Dahmann, J.S.: *High Level Architecture for Simulation*. Distributed Interactive Simulation and Real Time Applications, 1997., First International Workshop on, pages 9–14, Jan 1997.

[17] Daniel, J., B. Traverson et V. Vallee: *Active COM : an Inter-working Framework for CORBA and DCOM*. Dans *Distributed Objects and Applications, 1999. Proceedings of the International Symposium on*, pages 211–222, 5-6 Sept. 1999.

[18] Foster, I. et C. Kesselman: *The Globus Project : a Status Report*. Heterogeneous Computing Workshop, 1998. (HCW 98) Proceedings. 1998 Seventh, pages 4–18, Mar 1998, ISSN 1097-5209.

[19] Giard, Pascal, Jean-François Boland et Jean Belzile: *CORBA Based Communication for Verification*. TEXPO 2007 Research Competition and Exhibition, CMC Microsystems 2007 Annual Symposium, Oct. 2007.

[20] Giard, Pascal, Jean-François Boland et Jean Belzile: *Hardware in the Loop Functional Verification Methodology*. Object Management Group's Third Software Based Communications Workshop, Mar. 2007.

[21] Giard, Pascal, Jean-François Boland et Jean Belzile: *CORBA Based Co-Verification Methodology For SystemC*. 8th North American SystemC User Group Meeting, Feb. 2008.

[22] Giard, Pascal, Jean-François Boland et Jean Belzile: *CORBA Based Communication For Design and Verification*. SYTACom-S Poster Session 2008, Jul. 2008.

[23] Giard, Pascal, Jean-François Boland et Jean Belzile: *CORBA Communication Backplane For Design and Verification*. Microsystems and Nanoelectronics Research Conference, 2008. MNRC 2008. 1st, pages 121–124, Oct. 2008.

[24] Gokhale, AS et DC Schmidt: *Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks*. IEEE Transactions on Computers, 47(4) :391–413, 1998.

[25] Guofu, Ding: *Study of DCOM-based Simulation Applied in Locomotive & Track Coupling Dynamics*. Parallel and Distributed Computing, Applications and Technologies, 2003. PDCAT'2003. Proceedings of the Fourth International Conference on, pages 652–654, Aug. 2003.

[26] Habert, S. et L. Mosseri: *COOL : Kernel Support for Object-oriented Environments*. Dans *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*, pages 269–275. ACM New York, NY, USA, 1990.

[27] Hines, Ken et Gaetano Borriello: *Dynamic Communication Models in Embedded System Co-Simulation*. Dans *Design Automation Conference*, pages 395–400, 1997.

[28] Hung, Yu-Shan, Chih-Hung Lee, Su-Fen Tseng et Tsai-Ming Hsieh: *Minimum Cost Complex Resource FPGA Partition with Performance Refining*. Circuits and Systems, 2002. MWSCAS-2002. The 2002 45th Midwest Symposium on, 2 :II–107–II–110 vol.2, Aug. 2002.

[29] IEEE Computer Society: *IEEE Standard for Binary Floating-Point Arithmetic*. IEEE Standard 754-1985, 1985.

[30] Kellogg, Oliver M.: *CORBA-IDLtree-1.4*. http://search.cpan.org/dist/CORBA-IDLtree/.

[31] Kim, Dohyung, Chan-Eun Rhee, Youngmin Yi, Sungchan Kim, Hyunguk Jung et Soonhoi Ha: *Virtual Synchronization for Fast Distributed Cosimulation of Dataflow Task Graphs*. Dans *System Synthesis, 2002. 15th International Symposium on*, pages 174–179, 2002.

[32] Koga, M. et Y. Kikukawa: *Integration of Platform-dependent Simulators using Distributed Object and Native Language Interface*. SICE 2002. Proceedings of the 41st SICE Annual Conference, 1 :374–379 vol.1, Aug. 2002.

[33] Krupnova, H., C. Rabedaoro et G. Saucier: *FPGA Partitioning for Rapid Prototyping : a 1 Million Gate Design Case Study*. Rapid System Prototyping, 1999. IEEE International Workshop on, pages 128–133, Jul 1999.

[34] Kuznar, Roman et Baldomir Zajc: *Multi-way Netlist Partitioning into Heterogeneous FPGAs and Minimization of Total Device Cost and Interconnect*. 1994.

[35] Lapalme, J., EM Aboulhamid, G. Nicolescu, L. Charest, FR Boyer, JP David et G. Bois: *ESys. Net : a new solution for embedded systems modeling and simulation*. ACM SIGPLAN Notices, 39(7) :107–114, 2004.

[36] Levine, D.L., S. Flores-Gaitan et D.C. Schmidt: *An Empirical Evaluation of OS Support for Real-time CORBA Object Request Brokers*. Dans *Proceedings of the Multimedia Computing and Networking 2000 (MMCN00) conference*. Citeseer, 2000.

[37] Lianfeng, Yang, Wu Jin, Nasirjan et Wei Tongli: *The Development of Deep Sub Micrometer Semiconductor Device Simulation Based on the CORBA Platform*. Solid-State and Integrated Circuit Technology, 1998. Proceedings. 1998 5th International Conference on, pages 443–446, 1998.

[38] Object Management Group: *CORBA Language Mapping Specifications*. http://www.omg.org/technology/documents/formal/corba_language_mapping_specs.htm.

[39] Object Management Group: *IDL Syntax & Semantics*. http://www.omg.org/cgi-bin/doc?formal/02-06-39.

[40] Patel, H.D., D.A. Mathaikutty, D. Berner et S.K. Shukla: *CARH : Service-oriented Architecture for Validating System-level Designs*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 25(8) :1458–1474, Aug. 2006.

[41] Redlich, J. P., M. Suzuki et S. Weinstein: *Distributed Object Technology for Networking*. Communications Magazine, IEEE, 36(10) :100–111, Oct. 1998.

[42] Schmerler, S., Y. Tanurhan et K.D. Muller-Glaser: *A Backplane Approach for Cosimulation in High-level System Specification Environments*. Eurodac, 00 :262, 1995.

[43] Schmidt, D.C., M. Deshpande et C. O'Ryan: *Operating System Performance in Support of Real-time Middleware*.

[44] Schmidt, D.C., A.S. Gokhale, T.H. Harrison et G. Parulkar: *TAO : a High-performance Endsystem Architecture for Real-time CORBA*. Communications Magazine, IEEE, 35(2) :72–77, 1997.

[45] Texas Instruments: *TMS320C64x DSP Library Programmer's Reference*, 2002.

[46] Wolfe, V.F., L.C. DiPippo, R. Ginis, M. Squadrito, S. Wohlever, I. Zykh et R. Johnston: *Real-time CORBA*. Real-Time and Embedded Technology and Applications Symposium, IEEE, 0 :148, 1997, ISSN 1080-1812.