Cosimulation multiniveaux dans un flot de conception multilangage

Ph. Lemarrec

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|_|_|_|_|_|_|_|_|_|_|

THESE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Microélectronique

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

présentée et soutenue publiquement

par

Philippe LE MARREC

le 28 Juin 2000

Titre :

Cosimulation Multiniveaux Dans Un Flot de Conception Multilangage

Directeur de Thèse :

Ahmed Amine JERRAYA

JURY

M :	Pierre	GENTIL	, Président
M. :	Mondher	ATTIA	, Rapporteur
M^{me} :	Judith	BENZAKKI	, Rapporteur
M. :	Ahmed Amine	JERRAYA	, Directeur de thèse
M^{me} :	Dominique	BORRIONE	, Examinatrice

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|_|_|_|_|_|_|_|_|_|_|

THESE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Microélectronique

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

présentée et soutenue publiquement

par

Philippe LE MARREC

le 28 Juin 2000

Titre :

Cosimulation Multiniveaux Dans Un Flot de Conception Multilangage

Directeur de Thèse :

Ahmed Amine JERRAYA

JURY

M :	Pierre	GENTIL	, Président
M. :	Mondher	ATTIA	, Rapporteur
M^{me} :	Judith	BENZAKKI	, Rapporteur
M. :	Ahmed Amine	JERRAYA	, Directeur de thèse
M^{me} :	Dominique	BORRIONE	, Examinatrice

A ma femme pour m’avoir soutenu tout le long de
mes travaux et supporté mes humeurs de rédaction,

Remerciements

Je tiens à remercier tous les gens qui m'ont aidé et encouragé durant ces travaux.

Cette thèse n'aurait pu voir le jour sans le soutien et les efforts d'un certain nombre de personnes que je souhaiterais remercier ici. C'est avec une émotion particulière que j'ai l'occasion d'apporter le témoignage écrit de ma reconnaissance à tous ceux qui m'ont gratifié du soutien et de la confiance qui m'étaient nécessaires pour mener à bien ces recherches.

Je tiens à remercier Madame Dominique BORRIONE, Professeur au laboratoire TIMA, pour m'avoir fait l'honneur de présider le jury.

Je remercie Madame Judith BENZAKKI, Professeur à l'université d'Evry qui m'a fait l'honneur d'être rapporteur et Monsieur Pierre GENTIL, Professeur de l'INPG, pour sa participation en tant que membre du jury.

Il me fait grand plaisir de reconnaître l'immense contribution du Docteur Ahmed Amine JERRAYA, Directeur de cette thèse, qui m'a donné de son temps et communiqué de son savoir. Ses conseils m'ont été très profitables tout au long de mes travaux. Grand merci à lui qui ne se représentera jamais assez le respect que je lui dois.

Je saurais également gré à Mr Mondher ATTIA, pour avoir accepté d'être rapporteur de cette thèse et aussi pour m'avoir accueilli pendant neuf mois au centre de recherche de Peugeot-Citroën. Grâce à lui, j'ai pu bénéficié d'une expérience très profitable au sein du monde automobile.

Je souhaiterais aussi exprimer ma reconnaissance à Olivier CAYROL pour son assistance et son profond intérêt pour le projet.

Un grand merci à toutes les personnes de la Direction des Recherches et Affaires Scientifiques qui m'ont permis de découvrir le monde automobile pour lequel j'ai ressenti un vif engouement.

Je tiens à exprimer mes plus sincères remerciements au Dr Carlos Alberto VALDERRAMA qui m'a transféré son savoir au début de ces travaux, pour le temps précieux qu'il m'a accordé et la confiance qu'il m'a témoignée.

Je dois aussi féliciter Pascal COSTE , Thierry ROUDIER, Gabriela NICOLESCU et fabianno HESSEL qui auront évalué l'outil dans ses moindres détails et permis l'amélioration des travaux sur l'étude de nombreux exemples. Je tiens à leur attester ma profonde reconnaissance.

Ceci vaut aussi pour les nombreux stagiaires qui se sont essayés sur les premières versions de MCI. Leur patience irréprochable et les remarques dont ils m'ont fait part m'ont été d'une aide considérable.

Enfin, un grand merci à la fameuse équipe SLS du TIMA pour leur soutien et les grands moments que nous avons partagés (à Sonja, Fabianno, Pascal, Sabi, Zoltan, Wonder, Rodolphe, Imed, Amer, Lovic, Nacer et les autres).

Résumé

De nos jours, la réalisation d'un système électronique hétérogène composé de différents sous-systèmes (logiciel, matériel, mécanique, etc..), démarre par une spécification de haut niveau qui est ensuite divisée en différentes parties modélisées en différents langages. Ces nouvelles méthodes requièrent donc de nouvelles techniques d'aide à la conception et de validation afin de réduire le temps de mise sur le marché. La vérification par simulation de tels systèmes consiste à simuler conjointement les différentes parties du système en utilisant les simulateurs appropriés à leur type de modélisation. Ce type de simulation appelé cosimulation doit être réalisable à tous les niveaux de la spécification.

Le sujet de cette thèse consiste à étudier un outil de validation par cosimulation. Cet outil doit pouvoir vérifier un système complet tout au long de son flot de conception par simulation à l'aide de simulateurs concurrentiels communicants. Chaque partie du système est simulée, éventuellement sur une machine distante pour respecter la délocalisation des groupes de travail, mais surtout par un simulateur spécifique à son domaine d'application. La cosimulation multiniveaux peut être de deux types. Le premier est la cosimulation fonctionnelle. Elle permet une validation de l'interconnexion des sous-systèmes entre eux. Le second est la cosimulation temporelle. Elle permet d'ajouter à la cosimulation fonctionnelle une synchronisation en temps des différents simulateurs.

L'interface de cosimulation a été utilisée avec succès sur des applications industrielles, tout particulièrement sur des exemples du monde automobile chez PSA (Peugeot-Citroën). Au niveau multilangage, elle permet actuellement d'inclure des langages tels que SDL, COSSAP, MATLAB, SABER, VHDL et le C, bénéficiant ainsi d'une variété de langages pour la spécification.

Multilevel Co-simulation In a Multilanguage Design Flow

Abstract

Nowadays, the realization of an heterogeneous electronic system composed by different sub-systems (software, hardware, mechanical parts, etc...) begins with a high level specification which is then divided into different parts often modelized in different languages. These new methods thus imply new co-design and validation techniques so as to reduce the time to market. The verification by simulation of such systems consists in the joined simulation of the different parts of the system by using simulators which are appropriate to the type of module. This kind of simulation is called co-simulation and must be usable at each abstraction level of the specification.

The subject of this thesis concentrates on the study of a validation tool based on co--simulation. The aim of this tool is to verify a complete system all through its design flow by simulation implying the parallel and communicating execution of simulators. Each part of the system is, if necessary, simulated on a remote machine so as to maintain the delocalization of the working groups. The multilevel co-simulation may be of two types : the first one is the functional co-simulation which enables the validation of the interconnection of the sub-systems. The second one is the timed co-simulation which allows the addition of a temporal synchronization between the simulators.

The co-simulation environment has been successfully used on industrial applications, and particularly on mechatronic examples. It was, for instance, the case at PSA (Peugeot-Citroën). At the multilanguage level, co-simulation currently permits the introduction of such languages as SDL, COSSAP, MATLAB, SABER, VHDL and C. Thanks to co-simulation, designers benefit from many languages for the validation of their specification.

Spécialité : Microélectronique

Mots-Clés : cosimulation, spécification, langages, spécification multilangage, conception multilangage, simulation multi-domaine.

Intitulé et adresse du laboratoire :

Laboratoire TIMA - groupe System Level Synthesis
46, Avenue Félix Viallet
38031 Grenoble Cedex

Table des matières

Introduction

Ces dernières années, le monde de la conception microélectronique et particulièrement les outils d'aide à la conception ont bénéficié de l'essor du marché de la fabrication des circuits intégrés. De nos jours, les circuits intégrés deviennent de plus en plus complexes. La nécessité d'une mise sur le marché accélérée des systèmes intégrés conduit l'industrie de la microélectronique ainsi que la recherche à s'investir dans la conception au niveau système. Cette densité d'intégration sur silicium de systèmes complexes et la flexibilité croissante des nouvelles méthodes de conception ont permis d'accroître le champ d'applications des systèmes embarqués. Les domaines d'applications de ces circuits intégrés concernent désormais la conception de la téléphonie mobile, la télévision numérique, la visiophonie, ou encore des secteurs d'activités comme les systèmes mécatroniques, avioniques, etc..

Les exigences croissantes inhérentes aux besoins de modélisation des systèmes complexes ont incité les industriels à fournir des langages de modélisation de haut niveau dont les fonctionnalités sont adaptées de la façon la plus appropriée aux différents domaines d'applications. D'un autre côté, les concepteurs sont amenés à utiliser une plus grande variété de langages pour la conception d'un même système. Si ce système nécessite l'utilisation de plusieurs équipes de travail, de nouvelles méthodes de partitionnement et de modélisation sont nécessaires pour user au mieux de la spécificité des groupes de travail. L'intérêt principal de ces nouvelles méthodes de conception est donc de profiter de la spécialité des concepteurs le plus longtemps possible durant le flot de conception. Dès qu'il est possible, l'introduction d'outils de génération automatique, tels que les outils de co-conception,[1] est réalisée de façon à accélérer la réalisation du système et épargner au concepteur la réalisation des détails de certaines parties du système. Afin de pouvoir vérifier l'exactitude du comportement du système obtenu tout au long du flot de conception, de nouvelles méthodes de validation multilangages apparaissent. Ces méthodes de validation sont réalisées par des techniques de cosimulation. La cosimulation est tout d'abord apparue afin de valider des systèmes mixtes logiciel/matériel. Depuis peu, elle s'étend vers la validation des systèmes multilangages. Toutefois, la cosimulation doit permettre la validation du système à chaque étape de son raffinement, c'est à dire à tous les niveaux d'abstraction de la conception. La cosimulation multilangage et multiniveaux est donc devenue une approche essentielle de la validation des systèmes complexes avant leur implantation sur silicium.

L'objectif de cette thèse est donc de proposer un environnement de cosimulation multilangage capable de simuler un système complet aux différents niveaux d'abstraction durant le flot de conception.

Cette thèse présente les quatre points suivants :

Elle définit tout d'abord, dans le chapitre 1, la conception de systèmes multilangages. Ce

[1] Les outils de co-conception sont plus souvent connus sous le nom d'outils de "co-design".

chapitre introduit les contextes industriels et présente les intérêts majeurs de ce type de modélisation. Ensuite, il traite de l'approche des outils de co-conception pour ces types de systèmes. Enfin, il expose les possibilités de validation multilangages.

Le chapitre 2 décrit les concepts de cosimulation en présentant les techniques existantes de validation par cosimulation, l'utilisation de la cosimulation à différents niveaux d'abstraction durant le flot de conception. Enfin, il détaille l'évaluation des principaux outils de cosimulation existants.

Le chapitre 3 expose un environnement de cosimulation multilangage et multiniveaux dédié à la conception des systèmes hétérogènes multilangages. On y trouve la description des concepts utilisés, les domaines d'applications ciblés ainsi que les modèles de synchronisation utilisés par cet environnement. Ce chapitre se conclut en détaillant les résultats obtenus et les améliorations possibles.

Enfin, le chapitre 4 analyse les performances de l'outil de cosimulation étudié au chapitre 3 sur des exemples d'applications. La dernière partie présente un bilan des perspectives liées aux travaux menés dans le cadre de cette thèse.

Chapitre 1

CONCEPTION DE SYSTÈMES COMPLEXES MULTILANGAGES

1.1 Contexte

L'évolution des technologies a rendu possible l'intégration sur une même puce de processeurs programmables exécutant du logiciel et de circuits spécifiques. Les concepteurs ont donc bénéficié de gains d'espace qui les ont conduit à implémenter des circuits toujours plus complexes. Du fait de ces complexités croissantes, de nouveaux outils de conception ainsi que des nouvelles méthodologies sont apparus afin d'aider les concepteurs à réaliser leurs circuits. Le progrès principal permet aux concepteurs de réaliser des systèmes complexes sans avoir à se soucier de la difficulté réelle du circuit à plus bas niveau d'abstraction (niveau porte par exemple). L'usage d'un outil de co-conception engendre une meilleure maîtrise de la compréhension du système par une certaine abstraction de la complexité.

Actuellement, le monde de la modélisation se tourne de plus en plus vers une description du système à très haut niveau d'abstraction, suivi d'un partitionnement en sous-systèmes englobant un ensemble d'étapes de raffinement. Ces partitionnements sont réalisés en fonction du type d'implémentation, des ressources humaines et surtout des contraintes de temps. Afin d'aider le concepteur dans ses tâches de raffinement, de nouveaux outils de conception conjointe et de validation ont émergé. Leur apparition a engendré de nouvelles méthodologies de modélisation et de conception utilisées dans la réalisation de systèmes plus complexes. On peut penser qu'à l'avenir, la conception des systèmes va tendre vers une modélisation de haut niveau [Ous97].

1.1.1 Motivations de l'Industrie Microélectronique

La motivation première du secteur de la microélectronique demeure le temps de mise sur le marché.[1] L'intérêt essentiel est de pouvoir réaliser des circuits toujours plus complexes en un minimum de temps afin de faire face à la concurrence. C'est pourquoi, depuis quelques dizaines d'années, l'enjeu des industriels et de la recherche est de proposer des méthodes, des outils et des techniques d'aide et d'automatisation de la conception pour pouvoir faciliter la réalisation rapide de circuits toujours plus complexes.

Actuellement, des solutions existent pour réaliser des systèmes mixtes logiciel/matériel. En

[1] Plus connu en anglais sous la forme "time to market".

général, le système est spécifié par un langage homogène de haut niveau, il est ensuite découpé en sous-systèmes pour obtenir une ou des partie(s) logicielle(s) et matérielle(s). Ces étapes ont été peu à peu automatisées par les outils de co-conception. La complexité grandissante ainsi que l'intérêt porté à de nouveaux secteurs d'applications, incitent le monde de la microélectronique à se pencher sur la synthèse de haut niveau et la modélisation multilangage.

Jusque là, le temps de mise sur le marché a favorisé les concepteurs des systèmes à opter pour du matériel pour les parties critiques du système et une partie logicielle pour le reste. Bénéficiant des avantages du logiciel en terme de rapidité de développement, de paramétrage et de coût, les concepteurs de systèmes embarqués arrivent à tenir leurs délais de conception, en incorporant davantage de logiciel lors de la modélisation de leur système. Néanmoins, les systèmes contiennent d'autres parties qui ne peuvent être modélisées ni en logiciel, ni en matériel. C'est le cas des environnements qui interagissent avec le système (stimuli, organes d'affichage, systèmes mécaniques, hydrauliques, etc..). Ces systèmes sont difficilement modélisables dans les langages classiques de description du logiciel et matériel. En effet, certaines parties peuvent être réalisées de façon plus aisée par des langages dédiés à leur type de modélisation.

Désormais, la volonté des concepteurs est de pouvoir modéliser des applications en utilisant des langages de haut niveau qui supportent la complexité intrinsèque à ces systèmes. L'utilisation d'une multitude de langages dans la modélisation des systèmes de haut niveau apporte aux concepteurs une alternative à la conception des systèmes hétérogènes. Chaque langage est dédié à la modélisation d'une partie spécifique du système sur laquelle il reste spécialisé. Enfin, pour faire face à la concurrence, il devient nécessaire de disposer d'outils d'automatisation de la conception ainsi que d'outils de validation afin de pouvoir vérifier la cohérence de la conception tout au long du flot. Les outils de validation permettent ainsi de vérifier le système hétérogène multilangage avant son implémentation.

1.1.2 Apparition d'Outils de Co-conception et de Cosimulation

Les outils de co-conception et de cosimulation sont apparus avec le début de l'automatisation de la conception de systèmes mixtes logiciels/matériels. Ces outils ainsi que la méthodologie qui leur est associée représentent un domaine de recherche en vogue et demeurent le point de départ de la conception des systèmes hétérogènes et multilangages [JRV+97]. Les outils ont émergé pour maîtriser la complexité croissante des systèmes et pour répondre aux critères de performance et de temps exigés. Le processus de conception est un ensemble de tâches qui transforme un modèle en une architecture. Le modèle décrit le fonctionnement du système et l'architecture décrit la façon dont il sera réalisé. Une approche typique d'un outil de co-conception consiste à modéliser le système à partir de langages de haut niveau (niveau système), puis à réaliser, après plusieurs étapes de raffinement, une répartition des fonctions entre logiciel et/ou matériel. Enfin, les modèles qui résultent de l'outil de co-conception sont synthétisés par une étape de ciblage sur processeur pour la partie logicielle et par utilisation d'outils de synthèse existants pour les parties matérielles.

En plus de ce partitionnement s'ajoute une étape de synthèse de la communication [Dav97]. Elle permet de définir et de transposer sur une architecture cible les méthodes de communication entre les blocs matériels et logiciels. Certaines approches de co-conception prennent en compte l'étape de synthèse de la communication après le partitionnement [GG96][XGR96] alors que d'autres déterminent simultanément l'allocation de la communication puis l'ordonnancement de l'ensemble des tâches [YW95].

Plusieurs projets de co-conception existent à l'heure actuelle (Cosyma [OBE+97], SpecSyn [GV95], Cosmos [VRD+97], Codes [Buc94], Tosca [ABFS94], Ptolemy [LR93], ...). Ces approches proposent de partir d'une spécification système type de haut niveau, pouvant être C^X, SpecCharts [VNG95], SDL [ST87], StateCharts [Har87] ou SpeedChart. Néanmoins, après avoir obtenu les modèles générés en C, VHDL[2] ou en d'autres formats à partir de ces outils de co-conception, une étape de validation est nécessaire. Pour cela, il faut valider de façon conjointe le bon comportement des modèles générés. Ainsi, parallèlement aux outils de co-conception, sont apparus les outils de cosimulation logiciel-matériel [Row94][CT95] (CoWare, Cosyma, Ptolemy, VCI, etc..)

L'apport des outils de co-conception a permis d'accroître la complexité des systèmes à modéliser et a remis en question la façon de concevoir les systèmes à très haut niveau. Certes, les outils de co-conception des systèmes mixtes logiciel/matériel sont de plus en plus utilisés. Cependant, la gestion des ressources pour la conception des systèmes a fini par orienter la modélisation de haut niveau des systèmes complexes vers un type de modélisation hétérogène et une modélisation multilangage.

1.1.3 Modélisation des Systèmes Hétérogènes

La spécification des systèmes embarqués nécessite l'utilisation de plusieurs formalismes en raison de la diversité des natures des fonctions à spécifier. Ceci pose des problèmes de choix des formalismes, des méthodes de spécification, et d'intégration de ces formalismes. Ainsi, les critères de succès de la tâche de spécification des systèmes complexes sont tels qu'une grande variété de langages de spécification doit être combinée [ZJ93].

La modélisation des systèmes hétérogènes permet l'utilisation de langages spécifiques pour les différentes parties du système à concevoir. Chacun des langages utilisés lors de la spécification est très efficace pour un type d'application donné. Par exemple, les langages SDL et StateCharts sont des langages très adaptés aux spécifications à base d'états. D'autres sont plus adaptés aux modèles flot de données (LUSTRE [CPHP87] et SILAGE) ou aux systèmes continus (MATLAB [Mat98], MATRIXX [IS98]), alors que d'autres sont très utilisés pour les descriptions d'algorithmes, comme le C, C++.

Quand un système doit être décrit par des groupes de conception séparés, ils peuvent avoir des cultures, des expertises, des méthodes de travail, et des styles d'écriture différents. La spécification d'un système, par conséquent, doit pouvoir offrir la possibilité aux différents concepteurs de modéliser les différentes parties dans des langages différents. Cette flexibilité d'utilisation offre la possibilité de spécifier des sous-systèmes en utilisant les outils les plus appropriés associés à des cultures spécialisées idéales.

Avec la modélisation hétérogène, la définition d'un système consiste en une multitude de modèles des différents sous-systèmes. L'intérêt de ce type de modélisation est qu'elle peut être exécutée, conçue et validée séparément en utilisant le langage le plus approprié. Le système complet est validé au final par des techniques de cosimulation.

Les outils de co-conception existants qui proposent une entrée hétérogène sont des environnements partant d'un prototype virtuel[3] de très bas niveau qui décrit une architecture abstraite

[2]VHDL : (VHSIC "Very High Speed Integrated Circuit" Hardware Description Language) est un langage de description formelle pour être utilisé dans toutes les phases de conception des systèmes numériques.

[3]Le prototype virtuel permet à la fois la manipulation du domaine logiciel ainsi que du domaine matériel qui prend en entrée une architecture hétérogène composée d'un ensemble de modules distribués issus du découpage

où le partitionnement logiciel/matériel est déjà réalisé. L'étape de co-conception consiste donc à réaliser un ciblage des parties logicielles et des parties matérielles sur des processeurs dédiés. Les points clés des systèmes hétérogènes sont la validation et la réalisation des interfaces entre les modules (Cosimulation, Synthèse des interfaces).

Des outils de conception et de validation hétérogènes existent mais ils restent utilisables à très bas niveau. Cependant, la modélisation hétérogène devient une alternative pour la conception des systèmes complexes impliquant l'utilisation de plusieurs ressources et d'habitudes de travail.

Pour les circuits intégrés, la validation des systèmes hétérogènes multilangages par cosimulation est apparue par le biais de la cosimulation matérielle/logicielle de bas niveau alors que dans d'autres domaines le multilangage existe depuis les années 1970, comme par exemple le domaine du logiciel où le C peut être lié à d'autres programmes comme l'assembleur.

1.2 Conception de Systèmes Complexes

La conception de systèmes électroniques complexes nécessite désormais la coopération de plusieurs équipes de conception ayant des cultures différentes. Ces équipes agissent selon leurs propres méthodologies de travail et leurs langages de modélisation. Elles maîtrisent donc l'utilisation des outils, sur lesquels elles ont l'habitude de travailler, de la modélisation au déboggage. C'est pourquoi, il est primordial de disposer de nouvelles méthodes de travail afin d'utiliser leurs capacités sans pour autant troubler leur méthodologie de travail. Ainsi, de nouvelles méthodes de spécification et de conception apparaissent la où plusieurs méthodes et langages sont nécessaires pour la conception d'un même système. Les méthodes de spécification, de conception et de vérification multilangages ont attiré les concepteurs. A l'heure où les outils convergent vers la modélisation de haut niveau et la synthèse au niveau système [Are00], de nombreux secteurs, comme les télécommunications, l'aéronautique, la mécanique et d'autres encore, sont sensibles à ces perspectives notamment les concepteurs qui sont amenés à fabriquer des systèmes communicants avec un environnement extérieur.

1.2.1 Domaines d'Applications de la Conception des Systèmes Complexes

Les domaines d'applications concernés par la conception de systèmes complexes sont de plus en plus vastes. Ces applications peuvent aussi bien concerner le pilotage automatique d'un avion, la régulation d'une suspension de voiture ou encore la modélisation d'un émetteur haute fréquence. Globalement, les méthodologies liées à la conception des systèmes complexes s'appliquent aux concepteurs de systèmes où la coordination d'équipes différentes de modélisation est primordiale. Elles concernent également et surtout les concepteurs de systèmes embarqués pour l'aviation, l'automobile, les télécommunications, etc... Ces types de systèmes consistent à échanger ou réguler des informations liées au monde extérieur (analogique, hydraulique, mécanique, ...). Dans ce type de systèmes, seul le sous-système de régulation sera implémenté sur un circuit et la modélisation de l'environnement extérieur sert comme base de simulation pour la validation du système tout le long de son flot de conception. La conception des systèmes complexes est apparue, par exemple, dans des domaines d'applications comme les télécommunications et la conception automobile

logiciel/matériel et génère des descriptions exécutables pour des éléments matériels et logiciels.

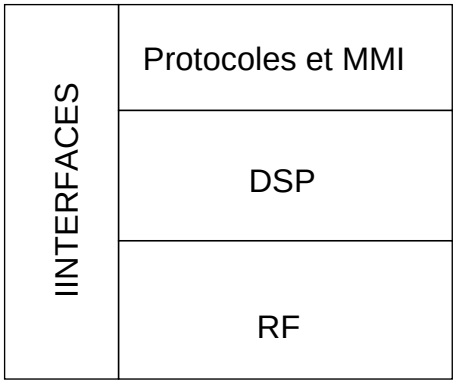


FIG. 1.1 – Architecture hétérogène d'un terminal de télécommunication pour mobiles

1.2.1.1 Le Monde Automobile

L'utilisation de l'électronique dans le monde automobile devient de plus en plus importante. Il est, par ailleurs, intéressant de noter que la part de l'électronique dans une voiture représente plus de 30% du prix de sa conception [RBCH96]. La conception conjointe de parties mécaniques avec des parties électroniques, et particulièrement des circuits à base de microcontrôleurs, représente une part considérable de la conception de systèmes mécatroniques. A titre d'exemple, la BMW série 7 ne comporte pas moins de 70 microcontrôleurs, ce qui porte la part de l'électronique à 15% du prix total du véhicule. Dans les approches traditionnelles de conception, les parties électroniques et mécaniques sont modélisées par différents groupes et l'intégration complète du système est exécutée à la fin du flot de conception. Ce schéma peut impliquer des retards provenant d'une ou de plusieurs parties du système. La modélisation d'une partie logicielle comme celle d'une partie matérielle peut nécessiter plus de temps que celle d'une partie mécanique, du fait de sa durée de validation. Des problèmes de compatibilité lors de l'assemblage final peuvent survenir car certains groupes de conception peuvent ne pas respecter le cahier des charges. Tous ces aléas peuvent finalement induire un coût supplémentaire de conception (cf. §4.2). C'est pourquoi, la conception multilangage et notamment la cosimulation peuvent apporter une souplesse dans la modélisation des différentes parties du système et permettre une validation plus rapide de celui-ci en le simulant au fur et à mesure de son avancement.

1.2.1.2 Les Systèmes de Télécommunications

Les circuits pour les télécommunications renferment à la fois des parties analogiques et numériques. De ce fait, le domaine des télécommunications, qui représente un énorme marché potentiel, s'intéresse de plus en plus à la spécification de ses systèmes [Bol97]. Par exemple, la conception d'un terminal mobile (eg. GSM[4]) consiste à utiliser la collaboration de plusieurs équipes spécialisées. On attribue à chaque équipe un domaine spécifique de la conception. La modélisation d'un système complexe comme le GSM (cf. 1.1) comprend essentiellement quatre parties :

1. Les protocoles et MMI[5] :
   Cette partie gère les protocoles de communication de haut niveaux, la gestion des don-

[4]GSM : Standard Européen des cellulaires.

nées ainsi que l'interface utilisateur. Cette partie est généralement conçue par une équipe logicielle qui utilise des langages de haut niveau comme SDL ou C++.

2. Le sous-système DSP[6] :
   Ce sous-système représente la partie traitement du signal et la correction d'erreurs de transmission. Il est généralement modélisé par un groupe spécialisé qui utilise des outils spécifiques comme MATLAB/SIMULINK ou COSSAP [RC95].
3. La partie Radio Fréquence :
   Cette partie sert à établir une connexion et à moduler les données. Elle est généralement formée par un groupe de conception analogique utilisant d'autres types d'outils et méthodes spécifiques comme CMS [HP98].
4. Le sous-système interface :
   Il gère la communication entre les trois autres parties. Il peut inclure des bus complexes et un système de mémoires sophistiqué. Il est généralement modélisé par un groupe de conception matériel qui utilise des outils classiques de l'EDA.[7]

Les points clés de la conception d'un tel système sont la validation du système complet et la synthèse des interfaces entre les différents sous-systèmes. Evidemment, la plupart de ces sous-systèmes contiendront des parties logicielles et matérielles.

La modélisation implique donc l'utilisation de plusieurs langages par les différentes équipes de modélisation. Ce type de système est constitué de quatre sous-systèmes hétérogènes qui sont traditionnellement conçus par des groupes séparés qui peuvent être géographiquement distants. L'évaluation de langages de spécification devient donc nécessaire pour la spécification des systèmes complexes de télécommunications [JKS+98]. Dans un système tel que le GSM, les protocoles de communication sont modélisés en langage SDL, la partie DSP pourra être modélisée par des langages adaptés tels que COSSAP et MATLAB, la partie radiofréquence peut se reporter à des logiciels comme ADS, VHDL-Ams,[8] tandis que les interfaces sur silicium pourront être modélisées en VHDL.

Pour maintenir un tel projet, sa gestion entre plusieurs groupes de conception séparés est indispensable.

1.2.2 Les Langages de Description

Cette section est basée sur les travaux conjoints réalisés par l'équipe SLS du laboratoire TIMA [JRM+99].

La spécification est une tâche essentielle dans la conception d'un système et les langages servant à la modélisation ont également un rôle primordial. La sélection d'un langage se résume généralement à un compromis entre plusieurs critères : la puissance d'expression du langage, les possibilités d'automatisation fournies par les modèles à travers le langage et les capacités supportées par l'outil [DBDH99]. Dans le cadre de certaines applications, plusieurs langages sont nécessaires afin de modéliser le même bloc (applications contrôle/données, continu/discrétisé). De ce fait, il n'existe pas de règles précises quant à l'utilisation d'un ou des langages de modélisation des systèmes.

[5]MMI : Men Machine Interface, où interface homme-machine.
[6]DSP : "Digital Signal Processing", partie traitement du signal.
[7]EDA : Electronic Design Automation.
[8]VHDL-Ams est l'adaptation du VHDL pour la conception de circuits analogiques.

Les langages de spécification sont d'abord apparus en ingénierie du logiciel dans le but de supporter les premières étapes d'un développement logiciel [Dav95]. Le coût élevé du développement logiciel et sa maintenance ont prouvé la nécessité de se concentrer sur la spécification et le besoin d'analyse des étapes de conception. Le premier langage de description matériel était dédié à la spécification des ISC.[9] DDL, PMS et ISP sont des exemples typiques introduits dans les années 60-70 pour la spécification matérielle [GVNG94]. Depuis, une multitude de langages est présente dans la terminologie scientifique. Ces derniers sont le résultat de plusieurs champs de recherche. Un langage est souvent dédié à un domaine d'application précis, du fait de ses possibilités de modélisation et du secteur d'application qu'il cible. Il est établi qu'un langage comme le VHDL est associé à la modélisation matérielle comme le langage C l'est au logiciel. Toutefois, de nouveaux langages de plus haut niveau apparaissent comme les langages ESTEREL [BC84], LOTOS [ISO89], SDL [ST87], et plus récemment de nouveaux travaux du groupe SLDL[Sch98] essayent de définir le futur langage de modélisation des sytèmes. Enfin, le champ d'application de la modélisation des systèmes embarqués s'élargissant, des langages divers comme MATLAB [Mat98] ou MATRIXX ont fait leur apparition. Ils permettent de modéliser les environnements extérieurs lors de la validation du système.

1.2.2.1 Les Modèles d'Exécution d'un Langage de Description

La réalisation d'une spécification multilangage bien adaptée au comportement d'une application nécessite une étude sur les langages de spécification. Pour cette raison, les modèles d'exécution, les concepts et les caractéristiques des langages utilisés sont à prendre en compte.

Les propriétés d'un langage comme sa facilité d'expression découlent de son modèle d'exécution fondamental. En fait, les langages de spécification diffèrent principalement selon la manière dont ils fournissent une vision des composants de base, le lien et la composition de ces composants. Les composants de base sont décrits par leur comportement, qui peut être orienté contrôle ou données. Les liens définissent la communication entre les modules et la composition sert à décrire la hiérarchie. De nombreuses classifications de langages de spécification ont été proposées dans la littérature. La plupart d'entre eux se concentrent sur le style de spécification. D. Gasjki [GVNG94], par exemple distingue cinq styles de spécification :

- orienté états,
- orienté activités,
- orienté structure,
- orienté données,
- hétérogène.

Les modèles orientés, respectivement états et activités, offrent une description du comportement à travers des machines d'états et des transformations. Le modèle orienté structure se concentre sur la structure hiérarchique du système. Enfin, Les modèles orientés données offrent une spécification système basée sur la modélisation des informations. Delgado dans [KPSG99] utilise d'autres techniques de classification et distingue neuf modèles d'exécution différents.

Une classification objective des langages de spécification est définie de façon plus précise quand elle est basée sur un modèle d'exécution plutôt qu'un style d'écriture. En fait, le style de la spécification reflète la syntaxe et non les aspects du modèle d'exécution. Le modèle d'exécution d'une spécification donnée est la combinaison de deux concepts orthogonaux : le modèle de communication et le modèle du contrôle.

[9] ISC : "Instruction Set Computers". Calculateurs sur jeu d'instruction.

Type \ Modèle de communication	mono-processus	distribué
flot de controle	SCCS, StateChart Esterel, SML	CCS, CSP, VHDL OCCAM, SDL
flot de données	SILAGE LUSTRE, SIGNAL	flot de données asynchrone

FIG. 1.2 – Modèles d'exécution de langages de spécification

- Le modèle de communication d'un langage de spécification contient deux variantes :
  - un modèle d'exécution synchrone (un seul flot d'exécution),
  - un modèle d'exécution distribuée (plusieurs processus s'exécutant en parallèle avec une communication explicite régulée par une synchronisation).
- Le modèle de contrôle peut être classifié en :
  - flot de contrôle : le modèle est constitué d'une machine d'états finis,
  - flot de données : c'est un ensemble de blocs fonctionnels connectés. Un bloc est activé lorsque suffisamment de données sont présentes en entrée. A chaque exécution, un bloc consomme des entrées et produit un résultat en sortie. Les modèles de flot de données existants sont :
    - flot de données synchrone : le nombre de données consommées et produites est constant d'une activation à l'autre.
    - flot de données dynamique : les blocs sont complètement indépendants et le nombre de données consommées et produites peut varier d'une exécution à l'autre.

Globalement, nous avons principalement 5 modèles d'exécution qui peuvent être définis par les notions d'exécution concurrentielles et de synchronisation. La figure 1.2 présente différents langages par rapport à ces modèles.

1.2.2.2 Les Langages de Spécification des Systèmes

La plus grande utilisation de ces langages s'exerce dans des domaines tels que :

1. la conception des systèmes VLSI : la recherche dans ce domaine a produit des langages de description matériels (HDLs). ISPS [Bar81], CONLAN [Pa83] et plus récemment HardwareC [KM88], SpecCharts, SpecC [NVG91] et VHDL [IEE94]. Ces langages ont pour but de pouvoir spécifier les parties matérielles. Il existe certains langages de cette catégorie dédiés pour des applications spécifiques comme COSSAP et SPW [KeA95] pour la conception basée sur les DSPs,
2. la spécification des protocoles : plusieurs langages ont été créés pour la modélisation des protocoles. Ils sont aussi utilisés pour valider ces protocoles, c'est pourquoi ces langages sont basés sur les techniques de description formelle. SDL, LOTOS et ESTELLE [Glu93][BD87] sont des langages caractéristiques de cette modélisation,
3. la modélisation des systèmes réactifs : les systèmes réactifs sont des applications temps-réel avec une réaction rapide à l'environnement. ESTEREL, LUSTRE et SIGNAL [GG87] sont des langages typiques pour la spécification des systèmes réactifs. Petri-Nets [Pet81] peut aussi être inclus dans cette catégorie,

4. la programmation : la plupart des langages de programmation ont été utilisés pour la description matérielle. Ces langages sont Fortran, C, Pascal, ADA mais aussi C++ et JAVA. Bien que ces langages apportent de grandes facilités pour la spécification des systèmes matériels, ils ne permettent pas de modéliser facilement les notions de temps ou les modèles d'exécutions parallèles. Des travaux de recherche ont essayé d'étendre les langages de programmation pour la modélisation matérielle. Par exemple, dans le but d'utiliser le langage C comme langage de spécification, il est nécessaire de le surcharger en y introduisant les concepts d'exécution parallèle, la communication, la hiérarchie structurelle, les interfaces et la synchronisation. Des approches comme HardwareC, SpecC ou SystemC [SD] essayent de remplir ces caractéristiques,
5. les langages de programmation parallèle : les programmes parallèles sont très proches de la spécification matérielle à cause de la concurrence. Pourtant, ils manquent généralement de concepts de temps et offrent des aspects dynamiques qui sont difficiles à implémenter en matériel. La plupart des travaux dans le domaine de la programmation parallèle sont basés sur CSP et CCS [Mil83]. Ces méthodes ont produit plusieurs langages comme OCCAM et Unity [BRX94] qui ont été utilisés pour la spécification des systèmes,
6. la programmation fonctionnelle et les notations algébriques : plusieurs tentatives ont été entreprises pour utiliser la programmation fonctionnelle et les notations algébriques pour la spécification matérielle. VDM, Z et B en sont des exemples. VDM [Jon90] (Vienna Development Method) est basé sur la logique par prédicats et a l'avantage d'être un standard ISO. La faiblesse de VDM réside principalement dans l'impossibilité de modéliser le parallélisme ainsi que son manque d'outils et de verbosité. Z [Spi89] est un langage basé sur les prédicats similaires à VDM. Z permet de diviser la spécification en petites parties appelées "schemes". Ces modules décrivent au même moment les aspects statiques et dynamiques des systèmes. B [Abr97] est composé d'une méthode et d'un environnement. Il a été développé par J.R. Abrial qui a participé à la définition du langage Z. B est complètement formalisé, sa sémantique est bien décrite et il dissocie de surcroît les deux tâches de spécification et de conception,
7. l'analyse structurelle : afin de maîtriser le développement de grandes applications logicielles, plusieurs méthodologies de conception ont été introduites. L'analyse structurelle fournit une approche systématique pour structurer le code et les données dans le cas d'applications logicielles conséquentes. La clé de voûte de l'analyse structurelle repose sur la décomposition des grandes applications en plus petites dont la gestion est plus aisée. Plusieurs améliorations de l'analyse structurelle initiale ont été introduites avec SART. Avec l'apparition de la programmation objet, plusieurs nouvelles techniques d'analyses ont fait leur apparition dans la littérature, HOOD et OMT [Mar96] par exemple. La dernière évolution de ces techniques d'analyses est le langage de modélisation unifié (UML) [KKM]. Le but d'UML est de rassembler plusieurs notations à travers un seul langage. Toutes ces techniques fournissent des outils performants pour la modélisation de grands systèmes mais restent inadaptées pour ensuite réaliser des étapes de synthèse et de validation,
8. les langages continus : ils sont basés sur les équations différentielles et sont utilisés pour la modélisation à très haut niveau de tout type de systèmes. Les langages les plus connus sont MATLAB, MATRIXX, Mathematica [WOL98] et SABER [Sab]. Ces outils permettent la modélisation de nombreux systèmes grâce à l'accession à une multitude de bibliothèques spécialisées dans différents domaines. Ils sont très utilisés dans des domaines

comme les DSPs, la conception mécanique et l'hydraulique. Ces langages permettent de disposer aussi d'une grande puissance d'expression. Néanmoins, ils utilisent énormément les méthodes de calcul sur les flottants, ce qui limite leur utilisation pour des étapes de synthèse. De plus, l'utilisation intensive de leurs bibliothèques spécialisées en fait des outils très flexibles et puissants.

1.2.2.3 Comparaison des Langages de Spécification

Il est peu probable qu'un seul langage de prédilection puisse supporter tous les types d'applications [RHJ+95]. Un langage de spécification est généralement sélectionné sur les simples critères du type d'application à modéliser mais aussi et surtout en fonction des habitudes du concepteur (réutilisabilité, lisibilité, documentation, etc..). Une multitude de langages sont présentées dans la littérature. Ceux-ci sont le résultat de différents champs de recherche. Les trois critères essentiels qui peuvent aider à la sélection d'un langage de spécification sont [Bru87] :

- la puissance d'expression du langage. Elle est relative au modèle de calcul. Elle se définit comme étant la facilité ou la difficulté à modéliser le comportement d'un système. Ses principaux composants sont la communication, la modélisation des exécutions concurrentielles, de la synchronisation, de la description des données et des modèles de simulation,
- la puissance analytique : elle est relative à l'analyse, la transformation et la vérification du format. Cet aspect est principalement fonction de l'offre et des concepts de l'outil. La puissance analytique d'un langage se concentre sur la définition formelle du langage. Certains langages ont des sémantiques formelles (Z, D, SCCS). Dans ces cas, les techniques mathématiques peuvent être appliquées pour transformer, analyser et appliquer des règles de preuves sur la spécification. L'existence d'une sémantique formelle permet une analyse plus facile de la spécification. Ceci permet aussi de prouver certaines propriétés comme la cohérence. Cette puissance inclut aussi les possibilités de construire des outils autour des langages analytiques,
- le coût d'utilisation : la clarté du modèle, les outils existants, la standardisation, et tous les aspects qui font que la maîtrise et l'utilisation de l'outil sont plus aisées. La lisibilité d'une spécification joue également un rôle important dans l'efficacité de son exploitation. Une spécification graphique est beaucoup plus lisible et peut être plus facilement compréhensible par certains concepteurs, alors que d'autres préfèrent le format textuel. Les formats graphiques et textuels sont donc complémentaires. Le support d'un outil autour d'un langage de spécification est aussi très important pour pouvoir tirer pleinement parti de l'expression du langage. Les supports d'un outil peuvent être les éditeurs, les outils de simulation, les outils d'applications de preuves, les déboggueurs, les environnements de prototypage, etc..

Le tableau 1.1 représente un certain nombre de ces langages et retrace les principales composantes des différents langages utilisés dans la modélisation des systèmes. Il est clair que les critères montrent bien que la comparaison des langages de spécification est difficile.

VHDL offre un excellent coût d'utilisation. Pourtant sa puissance d'expression est assez lente pour la modélisation de la communication et les structures de données.

SDL peut être un bon choix pour la spécification des systèmes. Il offre des facilités acceptables pour la plupart des critères. Le point faible de SDL est la spécification du temps, où seulement la notion de "*timer*" est utilisable. De plus, SDL restreint la communication à des

	HDL	SDL	MATLAB	StateCharts Esterel	SPW COS-SAP	C, C++	JAVA, UML, ...
exécution concurrentiel	*	***	*	**	**	/	**
notion de temps	***	* ?	***	**	*	/	*
algorithmes de calcul	***	* ?	***	***	***	***	***
FSMs, contrôle des interruptions	**	***	? ? ?	***	/	**	**
Analyse formelle	*	***	/	**	/	/	/
Coût d'utilisation	***	***	***	*	*	***	? ? ?
Bibliothèques spécifiques	coeurs HW	Protocoles	DSP, maths, mécaniques	?	DSP	***	? ?

*** : Le langage est excellent

** : le langage offre des possibilités acceptables

* : le langage offre des possibilités limitées

? : non vérifié, ne justifie pas une étoile

TAB. 1.1 – Résumé de langages de spécification

modèles asynchrones. Il introduit un modèle plus générique de la communication en utilisant les RPC[10] et les nouvelles versions à venir devraient introduire plus de capacités algorithmiques.

StateCharts et **Esterel** sont deux langages asynchrones. Esterel offre des concepts puissants pour la modélisation du temps. Néanmoins, son modèle de communication est restreint à la spécification des systèmes synchrones. StateCharts offre un faible coût d'utilisation, pourtant comme Esterel il possède des modèles de communication restreints. La puissance d'expression est augmentée par l'existence de différents types de hiérarchies (activités, états, modules). Les langages synchrones réalisent très difficilement la modélisation des systèmes distribués. Par exemple, Esterel et StateCharts supposent que les transitions de toutes les entités parallèles doivent prendre le même temps. Ceci signifie que si un module est décrit au niveau cycle d'horloge, tous les autres modules doivent être décrits au même niveau.

SPW et **COSSAP** sont deux environnements orientés DSP offerts par les vendeurs de l'EDA. Le fait qu'ils soient basés sur des langages propriétaires rend leur coût d'utilisation relativement élevé car ils manquent de standardisation et d'outils génériques supportant ces langages. Ces environnements font une utilisation prononcée de bibliothèques, ce qui rend la vérification formelle difficile à appliquer.

MATLAB est utilisé par plus de 400 000 ingénieurs à travers le monde et dans plusieurs domaines d'applications. MATLAB offre une bonne puissance d'expression pour la modélisation des algorithmes de haut niveau. De plus il supporte très bien la notion de temps. Par contre, les restrictions majeures de MATLAB sont relatives au modèle de communication (uniquement des fils qui communiquent des flottants), à l'expression des modèles de calcul basés sur les machines d'états (nouvelle version inclut la modélisation de modèles comme StateCharts) et la sémantique formelle.

[10]RPC : "Remote Procedure Call" appels de procédures distantes.

C et **C++** offrent une grande marge d'utilisation mais ils n'offrent pas de concepts généraux quant à la spécification des systèmes comme la concurrence et les aspects temporels. L'utilisation du C et du C++ pour la spécification des systèmes est très populaire. En réalité, la plupart des concepteurs sont familiers avec le C et dans beaucoup de domaines d'applications la spécification initiale est réalisée en C ou C++. Il est aussi très facile de construire des bibliothèques temps-réel pour étendre le langage afin qu'il supporte les exécutions parallèles et la notion de temps. Enfin, ces types d'extension ne sont pas standardisés, ils rendent les outils de spécification de systèmes spécifiques à des environnements dédiés.

JAVA et **UML** sont porteurs d'espoirs dans la communauté logicielle. Ils sont très étudiés pour leur application à la co-conception. UML est juste une notation émergeante tandis que JAVA semble être adopté par une grande partie de la recherche. Il propose des modèles d'une très grande puissance d'expression. Par contre, JAVA n'offre pas de concepts de temps, mais seulement la modélisation par des "*timers*" utilisant les concepts des RPC. La puissance d'utilisation de JAVA n'est pas encore connue.

1.3 Concepts de la Conception Multilangage

Cette section est le résultat d'un travail collectif publié dans [JRM+99].

Lorsqu'un grand système doit être conçu par des groupes différents, la conception requiert la coopération de plusieurs équipes utilisant différents langages durant l'ensemble du flot. Le bénéfice d'une meilleure vision et compréhension de cette conception est obtenu en séparant les tâches à réaliser tout en maîtrisant les travaux des équipes spécialisées. Dans les grands groupes de conception, les spécificités sont séparées en groupe de travail et même en sites géographiques différents. Dans le but de coordonner un projet de conception de systèmes complexes multilangages, une équipe supplémentaire peut éventuellement être nécessaire pour réguler les différentes tâches à réaliser. Par ailleurs, elle peut fixer les contraintes aux différentes équipes et vérifier que chacune d'entre elles respecte la spécification initiale. Il sera également important de modéliser la coordination des équipes entre-elles, de définir les interconnexions entre les sous-systèmes et de vérifier le respect de ces interconnexions par chacune des équipes de modélisation.

1.3.1 Flot d'une Conception Multilangage

Le flot d'une conception multilangage d'un système combine trois techniques de base :

1. La spécification du système.
   Elle implique l'utilisation de différents langages pour réaliser les spécifications partielles des sous-systèmes et la définition des interconnexions entre eux. La spécification initiale doit être scrupuleusement étudiée car une optimisation à haut niveau impliquera généralement un gain non négligeable de performances à bas niveau.
2. La validation des sous-systèmes et du système complet.
   Elle démarre par une validation des sous-systèmes séparément, puis par une étape de validation conjointe du système complet par des types de validation multiparadigmes (cf. §1.5). Ainsi, par ces méthodes de validation, le concepteur peut débogguer et analyser les différents modules de son système. La validation doit intervenir à chaque étape du flot de conception et à chaque étape de raffinement. Le fait d'utiliser des spécifications

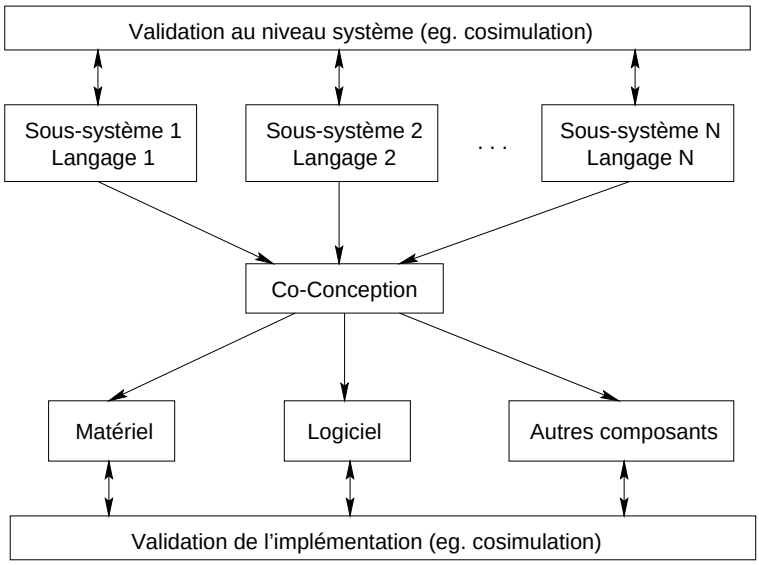


FIG. 1.3 – Conception multilangage

multilangages pousse les concepteurs à remplacer les techniques de simulation par de la cosimulation, et donc au lieu de réaliser de la vérification, des phases de co-vérification sont exécutées.

3. Le raffinement des différents composants du système tout au long du flot de conception. Il comprend :

– le raffinement des modules. Ce sont les techniques utilisées par les concepteurs pour descendre les niveaux d'abstraction avec les modèles décrits dans des langages différents (c'est une étape de synthèse pour chacun des modules),
– le raffinement des interactions entre les modules. C'est l'étape de synthèse d'interface multilangage qui doit permettre de définir les interfaces entre les sous-systèmes. Ces interfaces devront également être raffinées lorsque la spécification initiale doit être ciblée sur un prototype.

Le schéma de la figure 1.3 représente un flot générique de co-conception partant d'une spécification à plusieurs niveaux d'abstraction. Chacun des sous-systèmes (1, 2, .., n) de la spécification initiale doit être raffiné pour être décomposé en partie logicielle et matérielle. Le processus de co-conception doit s'attaquer au raffinement des interfaces et de la communication entre les sous-systèmes.

1.3.2 Interfaces de Communication des Différents Sous-Systèmes

La conception de systèmes embarqués hétérogènes engendre une étape de synthèse de la communication. Cette approche définit, dans le cadre des systèmes multilangages, un modèle de communication et permet le raffinement des interconnexions entre les blocs décrits dans des langages différents [HCM+99]. En effet, une communication est nécessaire entre les différents blocs définissant le système. Différents types de schéma de communication, de protocoles et d'adaptations sont nécessaires selon les langages de spécification utilisés et les besoins du système. Le type d'interconnexions et le protocole utilisé influent sur les performances du système et peuvent même rendre son implémentation impossible si le concepteur les néglige.

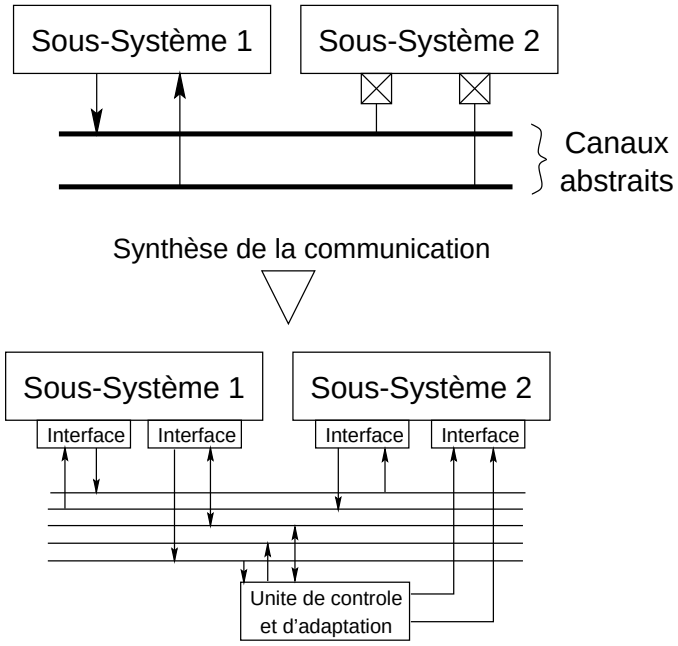


FIG. 1.4 – Synthèse des interfaces

Dans le cas de spécification multilangage, le problème réside dans la façon de connecter des ports de communication qui peuvent être différents d'un langage à l'autre. Le problème de la synthèse des interfaces se résume à produire une/des interface(s) entre des ports de communication différents afin d'adapter la communication sur un support tel qu'un signal, un bus, ou éventuellement un composant de contrôle. Cette synthèse de la communication représentée sur la figure 1.4 s'effectue en parallèle avec le raffinement des différents modules du système.

La spécification multilangage utilise de façon conjointe différents langages pour réaliser les sous-systèmes. De plus, certains des sous-systèmes à concevoir peuvent être des blocs réutilisables : IP,[11] processeurs programmables, ASICs, etc... A haut niveau, l'ensemble des sous-systèmes connectés, communiquent entre eux via un canal de communication. Les canaux offrent des primitives de communication de haut niveau (envoyer, recevoir) et des interfaces pour des connexions particulières (connecteurs, IP). Les primitives et les types de connexions aux canaux peuvent être différents pour chaque application.

1.4 Méthode de Spécification pour la Co-conception

Actuellement, il existe deux méthodes principales de spécification des systèmes lesquelles sont appelées spécification homogène et hétérogène. La première utilise un seul langage de très haut niveau. La seconde utilise le concept multilangage. Initialement, le concept de description hétérogène a été utilisé pour la spécification de systèmes mixtes logiciel/matériel ou le matériel et le logiciel sont décrits en utilisant des langages différents. Dans un deuxième temps, le concept a été étendu pour supporter des systèmes hétérogènes quelconques.

[11] IP : "Intellectual Properties" ce sont des blocs réutilisables accessibles à partir de bibliothèques.

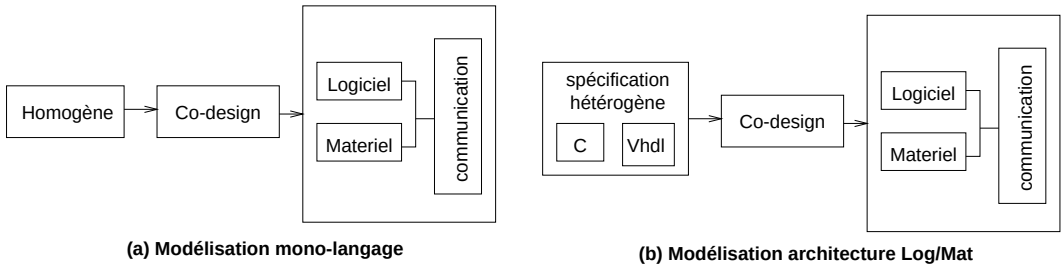


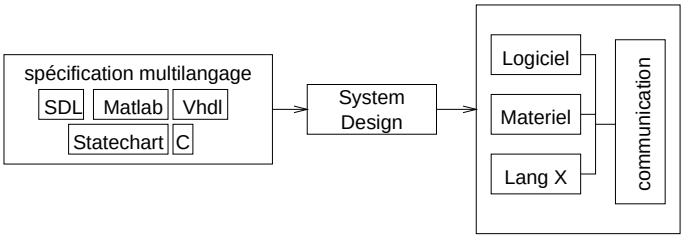
FIG. 1.5 – Co-conception homogène et hétérogène

1.4.1 Spécification Homogène et Hétérogène pour les Systèmes Mixtes Logiciels/Matériels

La spécification homogène implique l'utilisation d'un langage unique pour spécifier le système complet. Un environnement générique basé sur un modèle homogène est schématisé sur la figure 1.5a. Le système est décrit comme étant un ensemble de fonctions et de contraintes, et ceci indépendamment de toutes considérations logicielles ou matérielles. Cette spécification peut être indépendante de la future implémentation et du partitionnement logiciel/matériel à venir. Dans ce cas, la co-conception a pour but de transformer le langage initial en modules écrits en langage logiciel et matériel. Le résultat de la conception génère, la plupart du temps, une architecture à base de processeurs et d'ASICs.[12] Cet ensemble est généralement appelé "prototype virtuel" et peut se présenter sous forme d'un ou de plusieurs langages. En général, la spécification initiale peut être simulée afin de vérifier, dans un premier temps, que le système est correctement spécifié. Le but des outils de co-conception est de réduire le fossé entre le modèle spécifié et le prototype virtuel. La plupart des outils tentent de suivre ce schéma en partant d'un langage de spécification au niveau système. Néanmoins, le ciblage d'un langage de spécification de système de haut niveau, utilisant des concepts de haut niveau comme des modèles de contrôle distribués communicant par des canaux abstraits, vers un langage de bas niveau comme le C et le VHDL n'est pas trivial [DMJ97]. Seulement très peu d'outils ont essayé de partir d'une spécification de haut niveau. Dans ces outils, nous trouvons Polis [LSVH96][BGJ+97] qui utilise Esterel [Ber91], l'outil SpecSyn de Irvine [GVNG94] qui utilise SpecC [RD98], l'environnement décrit dans [KeA95] qui utilise LOTOS, ou encore l'environnement Cosmos qui utilise SDL. Peu d'approches ont utilisé des langages de spécification de systèmes distribués tels que SDL, LOTOS ou ESTELLE. Ceci est dû à la distance existante entre les différents concepts de ces langages et la description matérielle. La plupart des outils de co-conception démarre avec un modèle de spécification de bas niveau dans le but de réduire le fossé entre la spécification et le prototype virtuel. C'est le cas de l'environnement de co-conception Cosyma qui accepte une spécification en langage C^X qui est l'extension du langage C ou l'environnement Vulcan, de Standford, qui utilise HardwareC [GM92][KKR94]. Ces environnements utilisent des langages mono-flot étendus pour supporter les concepts matériels et la communication.

D'autres encore utilisent le langage VHDL [Wol94] comme modèle de spécification. En général, la complexité d'un outil de co-conception est relative à l'écart qui sépare le langage de spécification de la réalisation matérielle. L'automatisation de la co-conception impose souvent des restrictions sur les modèles d'exécution, sur les aspects dynamiques ou sur les modèles de communication disponibles.

[12] "Application Specific Integrated Circuit" : Circuit intégré dédié à une application.



(c) Modélisation multilangage au niveau système

FIG. 1.6 – Co-conception multilangage au niveau système

La spécification hétérogène permet l'utilisation de langages spécifiques par la modélisation des parties logicielles et matérielles. Un environnement générique de co-conception basé sur la spécification hétérogène est donné sur la figure 1.5b. La spécification démarre avec un prototype virtuel où le partitionnement logiciel/matériel est réalisé. La partie logicielle, par exemple, est modélisée en C plutôt qu'en assembleur et une partie matérielle est de préférence modélisée en VHDL ou Verilog.[13] Dans ce cas, la co-conception consiste à cibler les parties logicielles et matérielles sur des processeurs dédiés. Ce sont des schémas de co-conception qui restent relativement de bas niveau, où les différentes parties sont modélisées par des langages de spécification de niveau supérieur. L'utilisation de langages hétérogènes de plus haut niveau offre la possibilité de réutilisation du code à plus bas niveau. Une partie logicielle peut ainsi être validée sur différentes architectures de processeurs alors qu'une partie matérielle peut être raffinée par une étape de synthèse afin d'obtenir certains détails pour modifier la spécification. La problématique d'un tel schéma de conception est la validation et l'interfaçage des modèles entre eux. CoWare [CoW96] et Seamless [Gra96] sont des environnements typiques de cette méthode de co-conception et proposent des solutions performantes d'interfaçage dans le cas des modules logiciels/matériels. L'utilisation de ces méthodes de co-conception requiert dans les deux cas des nouvelles techniques de validation capables de vérifier les spécifications multiparadigmes et le prototype virtuel obtenu. L'étape de simulation est remplacée par de la cosimulation et la vérification par de la co-vérification.

Les spécifications homogènes et hétérogènes sont des méthodologies couramment employées de nos jours. Néanmoins, la plupart des outils s'appliquent typiquement à la conception logicielle/matérielle et c'est pourquoi ils ne sont pas adaptés pour des applications plus vastes qui impliquent d'autres domaines tels que la mécanique, l'hydraulique, ou encore l'électronique de puissance.

1.4.2 Méthodologie Multilangage au Niveau Système

On appelle méthodologie multilangage au niveau système une approche de conception pour les applications complexes mettant en jeu l'utilisation de plusieurs langages. Le flot de co-conception est représenté sur la figure 1.6. La spécification se fait à l'aide de langages divers facilitant la décomposition du système en sous-systèmes. Puis, l'ensemble du système est traité par un module d'aide à la conception qui est capable, en entrée, d'accepter une multitude de langages de spécification. En sortie, après différentes étapes (partitionnement, ré-ordonnancement,

[13] Verilog est un autre langage de description de haut niveau comme le VHDL. Verilog est souvent préféré au VHDL pour sa simplicité.

etc..) il peut fournir les modules matériels et logiciels sous différentes formes (VHDL ou Verilog pour le matériel). Ces modules peuvent enfin être synthétisés par des outils commerciaux existants. De telles approches multi-formalismes ont été étudiées dans le but d'obtenir des implémentations matérielles/logicielles : c'est le cas des projets parallèles ASAR [BP94] et PSYCHE [FIR+97] du laboratoire LaMI. Quelques normalisations ainsi que des environnements d'exploration d'architecture au niveau système apparaissent (SLDL, ADL (RAPID [RW93]), AREXSYS). Néanmoins, cette méthodologie nécessite la possibilité d'affiner les modules et de les utiliser pour la validation par cosimulation multilangage et ce aux différents niveaux de la conception. Ainsi de nouvelles techniques de cosimulation orientées multilangage à multiniveaux sont nécessaires pour pouvoir valider le système à concevoir avant son prototypage. De plus, cette spécification multilangage apporte de nouvelles études sur la synthèse des interfaces. En effet, l'interfaçage des sous-systèmes, lorsqu'ils sont décrits dans différents langages, nécessite un formalisme au niveau des types de connexion [HCM+99].

1.5 Validation des Systèmes Multilangages

La conception d'un système complexe peut donc nécessiter l'utilisation de plusieurs langages et de plusieurs équipes de travail parfois géographiquement distribuées. Ainsi, de nouvelles méthodes de spécification apparaissent pour pallier à ces besoins. C'est pourquoi, les concepteurs ont besoin de nouvelles méthodes de validation tenant compte de l'utilisation de différents langages tout au long du flot de conception.

1.5.1 Différentes Méthodes de Validation

La méthode de validation la plus traditionnelle des systèmes multilangages est celle du prototypage. Celle-ci ne représente pas l'ensemble des avantages (coût, flexibilité) en terme de modularité d'une solution de validation par simulation mais permet de vérifier dans un certain nombre de cas, et de façon efficace, les systèmes multilangages. Elle est encore très utilisée dans l'industrie et restera longtemps une méthode de validation en conditions réelles intéressante avant la production de masse. Toutefois, les concepteurs d'aujourd'hui recherchent des approches moins coûteuses par simulation. Il existe deux approches principales pour la simulation des systèmes multilangages qui respectent au mieux la notion de modularité et l'utilisation des cultures différentes. Ces deux principales approches sont : l'approche compositionnelle et l'approche basée sur la cosimulation.

1.5.1.1 Validation des Systèmes par Prototypage

La méthode de validation par prototypage demeure la plus ancienne méthode de validation et s'applique également à des systèmes multilangages. La figure 1.7 représente un schéma de validation par prototypage dans le monde automobile. Elle consiste en une réalisation partielle ou totale du système qui doit être conçu. Cette méthodologie est bien souvent utilisée dans le cas où le système est constitué d'un module logiciel/matériel communicant avec un environnement extérieur, mécanique par exemple. La partie matérielle est la plupart du temps constituée

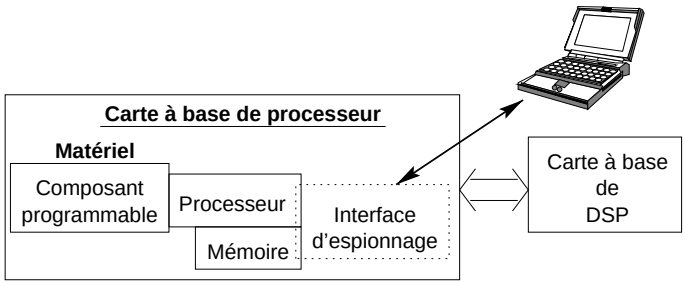


FIG. 1.7 – Méthode de prototypage de systèmes logiciel/matériel et mécanique dans l'automobile.

d'un composant programmable (ASIC, FPGA,[14] PLD,[15] etc..) alors que la partie logicielle se compose d'un processeur standard. De cette façon, la partie matérielle comme la partie logicielle pourront être rapidement modifiées. Le résultat de la compilation de la partie logicielle sera ciblée dans la mémoire du processeur et la partie matérielle sera ciblée sur le composant programmable par l'utilisation d'un outil de synthèse (FPGA-compiler, par exemple). Ce processeur est souvent réutilisé pour d'autres conceptions afin de réduire le prix de revient et le premier prototype sert souvent de base pour la conception d'autres systèmes. Il existe même des interfaces autour de ce processeur pour pouvoir espionner et visualiser les traces de son exécution lors de la phase de validation. Une interface peut être réalisée, par exemple, par un noyau temps réel sur le processeur dont une partie a pour tâche de transférer dans une zone mémoire l'état du processeur lorsqu'il est nécessaire. Elle peut encore être la programmation de l'interface série d'un microcontrôleur pour communiquer à un ordinateur extérieur toutes les informations de déboggage.

L'environnement extérieur, dans le cas d'une partie mécanique, peut être modélisé sur une carte à base de DSP[16] programmé. Cette carte permet donc de modéliser l'environnement extérieur sous forme d'équations mathématiques. Dans d'autres cas, l'environnement sera modélisé suivant les possibilités de raffinement du langage de modélisation utilisé. Dans le cas d'un environnement mécanique, il est donc aisé de modéliser l'environnement extérieur en MATLAB, de le simuler puis enfin de générer le code DSP pour l'implémenter sur une carte et reproduire en mode réel l'émulation du monde extérieur. Cette méthodologie est très utilisée dans les systèmes de sécurité où les tests en environnement réel sont risqués. Il est donc judicieux dans le cas de système de sécurité, de passer par cette étape de prototypage, même si cette méthodologie est coûteuse.

Des variantes existent :

– l'émulation d'une ou plusieurs parties du système [PSH98],
– la réécriture de modules matériels en logiciel, comme par exemple la réécriture de la partie matérielle (VHDL ou Verilog) en logiciel qui est implémentée sur le processeur du prototype et ainsi vérifier le système sans la réalisation d'une partie matérielle,
– Tests en conditions réelles de parties du système.

Cette méthode de validation, dont la mise en place est longue et compliquée, reste une bonne solution à bas niveau d'abstraction. Néanmoins, le prototypage a un coût important et l'utili-

[14]Field Programmable Gate Array : Composant programmable pour la composition de fonctions basées sur le contrôle/données.

[15]Programmable Logic Device : Composant programmable pour la description de fonctions logiques.

[16]Digital Signal Processing : processeur dédié aux traitement de signal

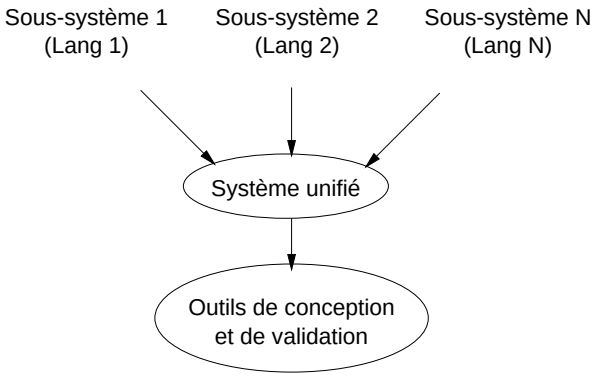


FIG. 1.8 – Approche compositionnelle

sation de la simulation est une solution nettement plus intéressante car elle peut remplacer de nombreuses étapes intermédiaires de la validation des systèmes.

1.5.1.2 La Méthode de Validation Multilangage Compositionnelle

La méthode compositionnelle, représentée sur la figure 1.8, a pour but d'intégrer les spécifications partielles des sous-systèmes dans une représentation unifiée qui est ensuite utilisée pour la vérification du comportement global du système. Ceci permet la mise en place d'une vérification totale de la cohérence et de la constance du système en vue d'identifier les liens de traçabilité et de faciliter l'intégration des nouveaux langages de spécification [Dav95]. Plusieurs approches ont été proposées [ZJ93][Wil92][Rei87] et leur principale intention est de prouver les propriétés d'exécution parallèle des systèmes entre eux. La réalisation d'un tel environnement dans le cadre de la validation des systèmes multilangages nécessite la traduction des différents modèles en un langage unifié ou encore la spécification de tous les modèles dans le même langage.

Elle peut aussi consister à compiler les modèles par leurs compilateurs respectifs pour obtenir autant de modèles exécutables que le système contient de blocs, puis à réaliser une édition de liens entre les blocs communicants. La simulation consiste à exécuter le modèle exécutable résultant et par conséquent à reproduire le comportement fonctionnel du système global. Par ailleurs, il faut éventuellement ajouter des fonctionnalités supplémentaires afin de visualiser les traces de la simulation. Cette méthode est tout de même plus rapide que la méthode par cosimulation car cette simulation se rapproche plus à une exécution. En effet, l'échange des données entre les sous-systèmes se fait de façon statique par entrelacement du code lors de la compilation et non pas de façon dynamique comme pour la méthode basée sur la cosimulation. La méthode compositionnelle est souvent utilisée pour des simulations à un niveau d'abstraction donné, par exemple au niveau architecture (compilation de la partie logicielle en C et de la partie matérielle en SystemC et édition de lien en vue d'une exécution), et permet de valider le système sans pouvoir réutiliser les différents modèles dans les niveaux d'abstraction inférieurs de la simulation. Le fait de changer un sous-système à simuler nécessite la restructuration de l'ensemble de l'environnement de cosimulation.

1.5.1.3 La Méthode de Validation Multilangage Basée sur la Cosimulation

La méthode de vérification basée sur la cosimulation (cf. Figure 1.9) consiste à combiner

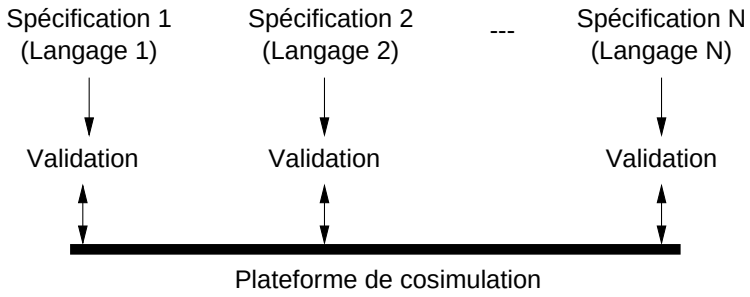


FIG. 1.9 – Approche par cosimulation

les environnements de simulation de chaque spécification partielle. La méthode de cosimulation est une solution de validation multilangage émergeante. En effet, elle réalise une intégration superficielle des spécifications partielles par une exécution parallèle des différents simulateurs mis en jeu. Les éléments clés d'un environnement de cosimulation sont la communication entre ces différents simulateurs, la synchronisation qui régule les données en fonction des niveaux d'abstraction et les types de langages simulés (MATLAB a une exécution séquentielle, le SDL exécute des processus de façon parallèle). La validation des systèmes multilangages par cosimulation consiste à sélectionner les simulateurs pour chaque module constituant le système, de valider indépendamment chaque modèle avec son simulateur, pour enfin simuler le système complet dans l'environnement de cosimulation. Cet environnement est basé sur un bus de cosimulation, aussi appelé plate-forme,[17] qui contrôle, régule, et aiguille les données calculées par chacun des simulateurs.

1.5.2 Problèmes Liés aux Méthodes Actuelles de Validation Multilangage

Les méthodes actuelles de validation des systèmes multilangages (par prototypage ou par émulation) restent des méthodes de bas niveau. En effet, dans l'industrie, il est très courant de rencontrer des méthodes de validation par prototypage par exemple. Ces méthodes sont fiables mais engendrent un temps de conception et des coûts de fabrication importants.

Lors de l'utilisation des méthodes de prototypage, les concepteurs de système s'aperçoivent bien souvent que le cahier des charges entre les équipes conceptrices n'est pas respecté ou du moins n'a pas été complètement suivi. Certaines petites erreurs de plus haut niveau peuvent impliquer aux concepteurs la redéfinition d'un ou de plusieurs modèles dans le flot de conception suivant la hiérarchie de modélisation du système. De plus, il s'avère parfois difficile de coordonner les modèles conçus qui proviennent d'équipes différentes : des problèmes de taille de données ou de type peuvent être complexes à résoudre lors de la connexion des prototypes à bas niveau. Pour débogguer la plate-forme de prototypage, un analyseur de signaux ou du code supplémentaire qui utilise les temps morts de l'exécution sont obligatoires. Si la réalisation d'un changement de paramètres intervient, il faut parfois régénérer tout le prototype.

Les méthodes de validation par simulation deviennent des alternatives très prisées des nouveaux concepteurs, car leur flexibilité et leur coût de mise en oeuvre sont très réduits face aux méthodes basées sur le prototypage.

En ce qui concerne la méthode compositionnelle, il est souvent très difficile de produire

[17] Backplane : plate-forme de cosimulation. Elle est composée d'un bus et d'un contrôleur qui permet d'aiguiller les communications à travers le bus.

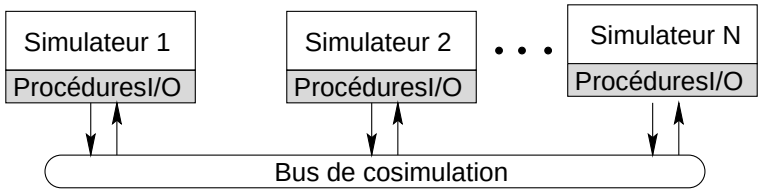


FIG. 1.10 – Modèle de cosimulation distribuée

un format unifié pour l'ensemble du système. Une alternative est la modélisation unifiée où le matériel est spécifié dans un langage similaire à la modélisation des autres parties du système (par exemple C [GDZ98][SD]). Néanmoins, ces approches nécessitent des bibliothèques complexes et impliquent un temps de développement coûteux. Elles sont difficilement extensibles à d'autres langages sans le développement conséquent d'une nouvelle bibliothèque de modélisation. Pour parfaire la traduction de l'écriture d'un modèle au langage unifié, une vérification formelle est nécessaire, ce qui complique énormément sa réalisation et la validation du traducteur. Lorsque la notion de temps est primordiale pour une vérification plus poussée de bas niveau, l'annotation du langage unifié peut s'avérer ingérable car elle implique la connaissance des temps correspondants aux instructions des différents langages.

Enfin, la méthode compositionnelle dissocie la simulation de la synthèse. Il est donc généralement impossible de réutiliser les modèles validés pour les synthétiser car ceux-ci ont été transformés avant la simulation. L'adjonction de code supplémentaire essentiel à l'affichage des traces et du déboggage peut rendre la chaîne de compilation d'autant plus longue.

La méthode de cosimulation compositionnelle est rapide et suffit dans bien des cas à valider le système. Malgré tout, la grande diversité des langages utilisés lors de la spécification a orienté la validation vers une cosimulation distribuée où plusieurs simulateurs s'exécutent de façon concurrentiels. La méthode par cosimulation est certainement la plus lente des méthodes car elle est liée aux performances des simulateurs inclus dans l'environnement de simulation. Pourtant, elle reste la plus confortable des solutions au niveau de l'utilisateur, car elle permet la réutilisation des blocs comme IP de simulation au niveau système, et reste un environnement basé sur une plate-forme et donc se charge d'une grande partie de la gestion des échanges de données

1.6 Approche de la Cosimulation Multilangage Distribuée

L'approche de validation par cosimulation devient une alternative essentielle dans la vérification des systèmes complexes. La méthode par cosimulation est préférée en raison de sa vitesse de mise en oeuvre et surtout du faible coût et de l'importante flexibilité qui en découlent. Il existe deux modèles de cosimulation : la cosimulation maître-esclave et la cosimulation distribuée (cf. §2.5). La cosimulation distribuée semble la plus adaptée aux systèmes multilangages. En effet, elle nécessite l'adaptation du simulateur à la plate-forme de simulation alors que pour la méthode compositionnelle, elle exige une adaptation vers le langage unifié des modèles simulés.

De plus, les systèmes à valider comprennent plusieurs parties de natures différentes et seule la validation par cosimulation distribuée permet d'évaluer le parallélisme d'exécution entre ces différents blocs.

Cette approche est basée sur un protocole de communication appelé bus de cosimulation. La

figure 1.10 montre un modèle générique de cosimulation distribuée. Chaque simulateur communique avec le bus de cosimulation à travers des primitives d'entrées-sorties. Ces primitives sont des appels externes au simulateur et sont réalisées par chaque simulateur de façon spécifique. Le bus de cosimulation est en charge du transfert des données entre les différents simulateurs. Il agit comme un contrôleur des communications réalisées par les primitives externes. L'implémentation du bus peut être basée sur les propriétés de communication standards du système, comme les IPC, les "*sockets*" ou autres supports de communication. Il peut également être implémenté comme une plate-forme de cosimulation "*ad hoc*" [LDA93b].[18] Ce type d'environnement de cosimulation distribuée présente des avantages précieux : une modularité et flexibilité pour les différents modules. La modularité consiste en une conception concurrentielle des différents modules via l'utilisation d'outils différents et de méthodes distinctes. La notion de flexibilité implique que ces différents modules puissent être simulés à différents niveaux d'abstraction durant tout le flot de conception.

Cette approche permet l'utilisation des simulateurs existants sur le marché. De plus en plus de travaux de recherche se tournent désormais vers la cosimulation distribuée pour la validation des systèmes multilangages. Des expériences existent déjà dans des domaines d'application bien précis. Des plate-formes de cosimulation permettant de simuler de façon conjointe des modèles de différents domaines d'application sont apparues (Cosimulation StateMate-MatrixX [Pfe97], Cosimulation COSSAP-MATLAB [LSPM95], etc..). Nous présentons dans le chapitre 3 l'outil MCI qui permet la cosimulation multilangage distribuée et cible différents secteurs de l'industrie des systèmes embarqués.

La cosimulation distribuée va dans le sens de la conception multilangage. Elle a pour but de connecter un simulateur quelconque à son environnement. L'utilisateur peut donc user d'une variété d'outils de spécification à différents niveaux d'abstraction. La modélisation des modèles qui doivent être simulés peut faire l'objet d'une conception concurrentielle. Du fait que les équipes conçoivent en des temps de conception sensiblement différents, certaines équipes achèvent leurs conceptions pendant que d'autres la poursuivent. Ce principe stimule l'esprit de concurrence et motive l'équipe en pleine conception à accélérer son activité. Le bénéfice final réside en une conception amplement plus rapide du système.

Aussi, la personne qui coordonne les équipes de modélisation peut se charger d'effectuer la simulation du système complet conformément à son rôle. Il a simplement besoin de décrire la coordination des simulateurs qui doivent être utilisés afin de rendre son environnement de cosimulation opérationnel. Il ne nécessite donc pas d'une connaissance spécifique de chaque sous-système. La cosimulation distribuée comme tout environnement de cosimulation, implique une notion de communication des blocs simulés entre eux. Les blocs proviennent de langages différents et les connexions entre ces blocs sont souvent très abstraites (problèmes de compatibilité des types, de taille des données, etc...). C'est pourquoi, il est intéressant de formaliser les connexions par des interfaces de communication afin de pouvoir connecter les sous-systèmes entre eux.

1.7 Conclusion

La spécification au niveau système est un aspect important de l'évolution des méthodes et des outils de conception des systèmes. Elle concerne les langages de spécification qui sensibi-

[18] Ad hoc : (loc. lat. "à cet effet") destiné expressément à cet usage.

lisent les concepteurs et les formes internes qui sont utilisées par les outils.

Une grande quantité de langages existe pour spécifier les systèmes. Ils proviennent de domaines industriels et de recherche différents : conception VLSI, protocoles, systèmes réactifs, programmation (parallèle ou non), notations algébriques, etc... Chacun de ces langages est très approprié dans un domaine d'application restreint. Evidemment, les langages sont aussi choisis pour leur qualité de modélisation, de puissance d'expression, de convivialité, etc.. Les expériences ont montré que dans le domaine des langages de spécification, il n'y a pas de langage unique et universel qui pourrait supporter la modélisation de tous les types d'applications. La conception des systèmes hétérogènes peut recourir à l'utilisation de plusieurs langages de spécification au vue d'une conception des différentes parties d'un système. Le point clé dans ce cas est la validation multilangage, la cosimulation et l'interfaçage des modules entre eux. Tout porte à croire que dans le futur, la conception multilangage va se développer et couvrir de multiples langages de spécification des systèmes. Elle permettra peut être d'allier l'utilisation de différents langages et la coopération d'équipes spécialisées pour obtenir plus facilement et plus rapidement la conception de très grands systèmes hétérogènes. En parallèle à ces méthodologies de conception, les outils de validation multilangage permettront de vérifier la cohérence des différentes modélisations à chaque étape du flot de conception.

Chapitre 2

LA COSIMULATION

La cosimulation est née de l'émergence des outils de co-conception. Sa toute première fonction était de réaliser la simulation mixte des systèmes logiciels et matériels. En effet, la cosimulation permet la validation fonctionnelle du système complet avant son implémentation. Cette étape de validation peut intervenir à différents moments lors du raffinement du système ou du processus de co-conception. Ce chapitre présente la cosimulation et insiste plus particulièrement sur ses intérêts, les techniques utilisées, les outils existants et les difficultés qui lui sont inhérentes.

2.1 Introduction à la Cosimulation

Les méthodologies modernes de conception de systèmes peuvent induire l'utilisation de plusieurs langages dans la spécification des différentes parties du système. Au cours d'un processus de co-conception, une partie du système spécifié ou son ensemble sont raffinés à l'aide d'outils spécifiques. A certaines étapes du processus, il est nécessaire de vérifier la conformité fonctionnelle du système obtenu. Pour cela, le concepteur doit avoir recours à des outils de validation tels que la cosimulation qui permet de vérifier l'exécution parallèle des sous-systèmes communicants. Elle constitue ainsi, une approche incontournable pour la vérification des résultats de sortie des outils de co-conception à tous les niveaux d'abstraction.

2.1.1 Définition de la Cosimulation

La cosimulation d'un système réside en la simulation conjointe de l'ensemble de ses sous-systèmes. Chaque sous-système est décrit dans un langage qui lui est propre. Ces sous-systèmes peuvent être décrits par des équipes de conception séparées et à des niveaux d'abstraction différents. Ainsi, de par la cosimulation, il est possible de simuler l'ensemble du système en coordonnant et en échangeant les données calculées et interprétées par chaque sous-système. Elle doit faire face aux problèmes de coordination et de synchronisation et doit permettre d'obtenir un résultat qui ne doit pas changer la fonctionnalité de la future implémentation du système.

2.1.2 Motivations de l'Utilisation de la Cosimulation

L'introduction sur le marché des outils de co-conception a engendré les premières motivations de l'utilisation des outils de cosimulation. Traditionnellement, dans le flot de conception

d'un système hétérogène, la validation est bien souvent réalisée de manière tardive. Elle est, la plupart du temps, accomplie une fois que le prototype est disponible, via l'utilisation des émulateurs ou d'autres techniques [KKR94][Cha96]. Néanmoins, il est observé que plus la validation d'un système intervient tôt dans le flot de conception, plus vite l'erreur sera corrigée [CHP95]. De plus, la complexité des systèmes s'est accrue, entraînant un risque d'erreurs de conception grandissant. C'est pourquoi, pour une nécessité de coût de validation, la cosimulation a pris son importance.

La validation d'un système complet peut nécessiter une validation conjointe ou partielle des modules qui le constituent. Un concepteur peut avoir besoin de valider une partie d'un système sans se soucier du reste de l'application. Ceci peut notamment être intéressant si le système est conçu par des équipes de développement distinctes. Les parties interdépendantes nécessitant un développement rapide peuvent être très tôt validées conjointement. Simultanément, un autre groupe peut développer les parties plus complexes, qui seront validées ultérieurement.

D'autre part, afin de minimiser le coût de conception d'un système, les phases de prototypage doivent être évitées le plus possible. Il est vrai que l'émulation est une alternative qui permet d'enrayer les coûts liés au prototypage. Cependant, elle impose de prendre le même support d'implémentation (processeur identique, carte modèle de base) et sa mise en oeuvre est parfois trop longue.

La simulation est de loin la solution la moins coûteuse en dépit d'un temps de simulation conséquent.

En conséquence, des solutions comme la cosimulation présentent des avantages à condition que leur mise en oIJeuvre soit relativement aisée.

2.2 La Cosimulation dans le Flot de Conception

La Cosimulation est basée sur l'exécution conjointe de simulateurs. Chaque simulateur exécute une partie spécifique du circuit à réaliser. Une interface de cosimulation consiste à extraire les données calculées de chacun des simulateurs et échanger leurs valeurs. Ces données échangées doivent être vérifiées pour qu'elles respectent les contraintes de types, de taille, etc... La cosimulation doit apporter au concepteur la possibilité de simuler, donc de vérifier son circuit avant son implémentation sans que celle-ci n'interagisse dans la fonctionnalité du système. Il est également intéressant de pouvoir débogguer simultanément les modèles simulés tout en gardant une transparence vis à vis de l'interconnexion. Aussi, la cosimulation a pour rôle d'accompagner le concepteur dans ses validations tout au long du flot de conception (à plusieurs niveaux d'abstraction). La cosimulation multilangage (cf. §1.6) étend la cosimulation (communément associée à la cosimulation matérielle-logicielle) à l'utilisation de systèmes plus diversifiés comme les systèmes mécatroniques. La validation du système par cosimulation multilangage consiste à valider une partie électronique constituée de modules logiciels/matériels (C et VHDL), avec son environnement extérieur (mécanique, électromécanique, etc...).

Ce type de modèle peut correspondre à un système mécanique, un circuit spécifique pour les télécommunications, comportant par exemple un circuit numérique et son module de haute fréquence. La modélisation de grands projets multilangages peut nécessiter la participation de plusieurs groupes de concepteurs. Ces groupes de travail peuvent, de plus, provenir d'entreprises différentes, et utiliser des méthodes de travail dissemblables. Le concept de la spécification

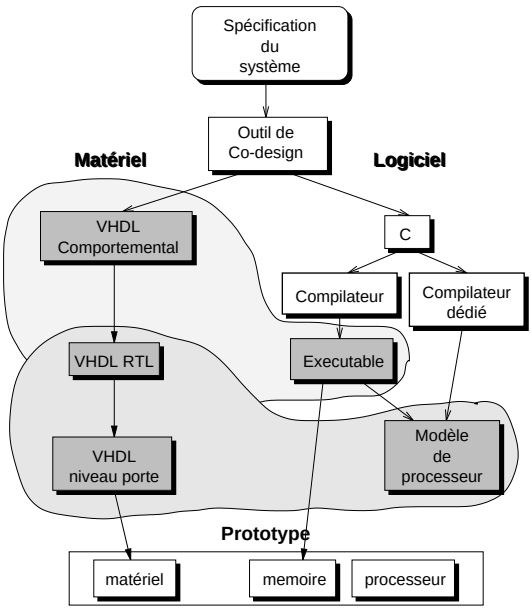


FIG. 2.1 – La cosimulation dans un flot de co-conception

multilangage, que nous avons plus amplement détaillé dans le premier chapitre, pousse à adopter une coordination entre les modules à concevoir, un certain formalisme et des conventions.

2.2.1 Intégration dans un Flot de Co-conception

La figure 2.1 représente l'intérêt de la cosimulation dans un flot de co-conception.

La cosimulation d'un système obtenu par le biais d'un outil de co-conception intervient, dans un premier temps, au niveau d'une vérification fonctionnelle des modèles C et VHDL générés. C'est la simulation conjointe du C compilé et exécuté sur la machine de cosimulation et du VHDL comportemental ou RTL[1] chargé sur un simulateur comme l'outil VSS de Synopsys. Ce type de cosimulation, c'est à dire celle mise en oeuvre par exécution native du code, évite en premier lieu un surcoût dû au retour à la spécification en cas d'erreur. En effet, elle permet de vérifier le bon comportement du système sans se soucier des éventuels problèmes de plus bas niveau comme par exemple l'allocation de processeurs et la synthèse matérielle réalisée avec par exemple des outils comme DC de Synopsys. L'étape suivante dans la validation par cosimulation peut-être la validation du système à très bas niveau par une cosimulation temporelle consistant en la simulation conjointe du logiciel et du VHDL RTL. Le C compilé est exécuté sur un simulateur de processeur et le VHDL RTL est vérifié par le biais d'un simulateur. Cette étape de très bas niveau, quoique lente, permet de bien reproduire le comportement réel du système avant son implantation.

La cosimulation permet de valider et de détecter les éventuelles erreurs de l'outil de co-conception. Le pourcentage d'erreurs, lors de la vérification finale, par émulation ou par prototypage devient ainsi faible.

[1] Vhdl RTL : Vhdl au niveau transfert de registres (Register Tranfer Level)

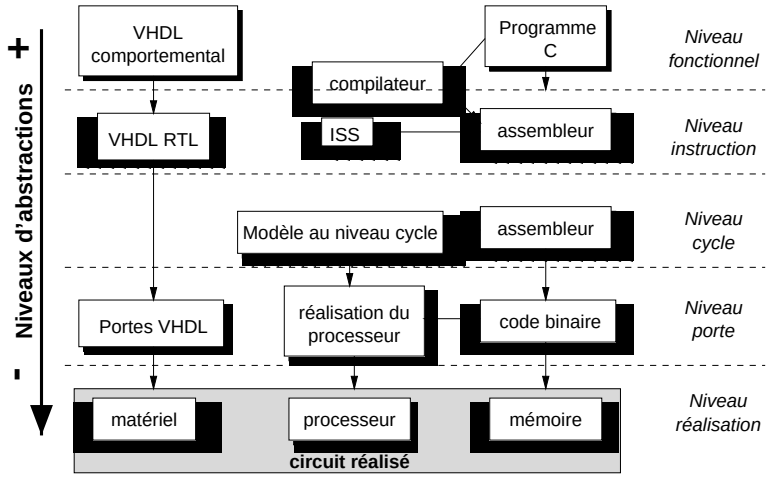


FIG. 2.2 – Les principaux niveaux d'abstraction

2.2.2 Les Différents Niveaux d'Abstraction de la Cosimulation

Les environnements de cosimulation diffèrent par leurs niveaux d'abstraction et le modèle de communication utilisé. La conception d'un système sur silicium[2] combine généralement des processeurs programmables, exécutant la partie logicielle, et des composants matériels d'applications spécifiques (ASICs) [PLMS94][LNV+97]. Dans le cas des systèmes multilangages, le système complet spécifié est habituellement constitué d'une partie logicielle/matérielle qui agit sur un environnement extérieur (mécanique, hydraulique, etc..). La partie logicielle/matérielle est le plus souvent raffinée à l'aide d'un outil de co-conception, tandis que l'environnement extérieur est modélisé séparément. Cet environnement extérieur ne faisant pas partie de la phase de prototypage, demeure dans sa modélisation initiale tout le long du flot de conception. Il agit ainsi comme un programme de test pour la partie logicielle/matérielle. L'abstraction est définie sur les modèles à simuler, et les interfaces qui connectent ces modèles.

2.2.2.1 Les Différents Niveaux d'Abstraction

Les différents niveaux d'abstraction d'un environnement de cosimulation sont issus des différentes étapes de simulation durant le flot de conception. La figure 2.2 illustre ces différents niveaux dans un flot de conception C-VHDL. Ce flot couvre quatre niveaux d'abstraction en plus du niveau de la réalisation :

– Au niveau fonctionnel :
  à ce niveau d'abstraction, le logiciel est décrit par un programme en langage C et le VHDL par un modèle comportemental. La communication entre le logiciel et le matériel peut être décrite au stade de l'application et une cosimulation C-VHDL peut être réalisée afin de vérifier la conformité du comportement de la spécification du système. Le code C est exécuté sur le poste de travail[3] et le VHDL est simulé au niveau comportemental par un simulateur ou un déboggueur approprié. De cette étape, seules les vérifications fonctionnelles sont obtenues. Ce type de cosimulation est aussi appelé simulation au niveau architecture ou exécution native. Afin d'augmenter la précision de la simulation, il est

[2]Système sur silicium (SOC : System On Chip)

nécessaire de descendre dans le niveau d'abstraction.

– Au niveau instruction :
le niveau instruction est la modélisation de l'exécution logicielle en une notion de nombre d'instructions. L'exécution logicielle est approximée par un modèle du processeur au niveau instructions.[4] Ce niveau permet de vérifier la compilation du programme de la partie logicielle et son adaptation à un processeur dédié à son exécution. Le programme compilé fournit dans la plupart des cas un programme assembleur. De plus, le modèle du processeur au niveau instruction permet de tester son initialisation qui est une valeur ajoutée au programme à simuler. La partie matérielle peut être simulée soit au niveau comportemental soit au niveau RTL par les simulateurs appropriés. Enfin, il sera possible de coordonner les notions de temps des simulateurs pour obtenir une première évaluation temporelle du système.

– Au niveau du cycle :
le niveau cycle[5] permet une simulation plus fine du système. Le matériel est décrit au niveau RTL et le C au niveau assembleur. Néanmoins, à ce niveau l'assembleur est injecté dans un modèle de processeur au niveau cycle voire niveau broches ou encore un modèle optimisé comme un modèle BFM[6] du processeur[7] qui peut être un modèle C ou VHDL. Le modèle précis au niveau du cycle peut être obtenu automatiquement ou manuellement à partir du niveau fonctionnel. Le modèle VHDL RTL de la partie matérielle peut être produit en utilisant la synthèse comportementale. La difficulté majeure réside dans la définition des interfaces logicielles/matérielles pour le passage d'un niveau à un autre.

– Au niveau portes[8] : il décrit la réalisation. Le modèle VHDL est raffiné par la synthèse du niveau RTL au niveau porte. La mise en place du logiciel dans le processeur est fixée. Le processeur peut être abstrait au niveau des macro-instructions ou conçu au niveau portes. Toute la communication entre le matériel et le logiciel est détaillée au niveau des signaux. Au deux derniers niveaux (niveau cycle et niveau portes), la cosimulation peut être exécutée pour vérifier et corriger les temps d'exécution de tout le système et vérifier les éventuels aléas.

2.2.2.2 L'Abstraction de la Communication

L'abstraction de l'interaction entre modules décrits dans des langages différents peut être indiquée à différents niveaux : du niveau de la réalisation jusqu'au niveau de l'application. Au niveau de la réalisation, la communication est exécutée par des fils. Par contre, au niveau application, la communication est effectuée par des primitives de haut niveau, indépendamment de la réalisation. La figure 2.3 représente trois niveaux de communication entre deux

[3]Host Code Execution.
[4]ISS : Instruction Set Simulator.
[5]Cycle-accurate level : niveau cycle d'horloge.
[6]BFM : "Bus Fonctionnal Model" : Les échanges sont actifs sur les changements d'état du bus du processeur. Ce modèle permet d'optimiser le taux d'échange lors de cosimulation.
[7]Pin Accurate Model : Modèle précis au niveau broches.
[8]Gate Level : Précision au niveau portes.

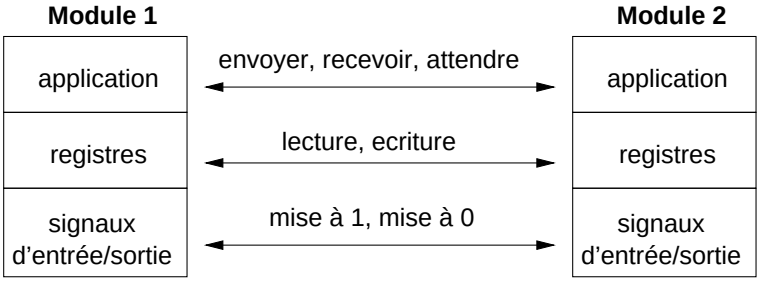


FIG. 2.3 – Niveaux d'abstraction de la communication entre deux modules

modules [TAS93]. La complexité de l'interface dépend du niveau de communication entre les modules. Au niveau de l'application, la communication est effectuée par des primitives de haut niveau comme envoyer "*send*", recevoir "*receive*" et attendre "*wait*". A ce stade, le protocole de communication peut être caché par des procédures. Ceci permet la spécification d'un module sans avoir à se soucier du protocole de communication qui sera utilisé plus tard. Il est nécessaire d'avoir un processus de sélection du protocole de communication pour réaliser l'interface entre les deux simulateurs [Wol93][IOJ95][IJ95]. Le niveau intermédiaire utilise des registres et la communication est exécutée par la lecture/écriture sur des registres d'E/S.[9] Ces exécutions peuvent cacher le décodage d'adresses physiques et la gestion des interruptions. La génération automatique des interfaces entre les simulateurs sélectionne le type d'architecture mis en place pour les exécutions des E/S. Par exemple, CoWare est un environnement typique de cosimulation agissant à ce niveau [Bol97].

2.2.3 Concept du Fichier de Coordination

Le fichier de coordination sert à décrire le système complet. Il décrit les différents sous-systèmes et les interconnexions entre eux. Il permet une clarification de la définition des tâches et plus généralement facilite la modularité de conception. Il sert également de base pour la génération automatique des interfaces de communication entre les sous-systèmes. Il est décrit en général dans des formats orientés description d'interconnexions ("*netlist*"). Ce format peut être le format EDIF, VHDL, SOLAR à plus bas niveau ou encore la plupart des langages de description y compris le C. Bien entendu, pour que ce format soit utilisable dans un maximum de niveaux d'abstraction, il doit être suffisamment flexible pour pouvoir modéliser des canaux abstraits de communication à haut niveau et des fils à bas niveau.

Ce fichier de coordination est utilisé à la fois pour la conception et la cosimulation. Dans le cas d'une cosimulation basée sur une plate-forme, il est possible de définir le type de routage des données et l'adaptation des types et éventuellement les conversions et les différents simulateurs à utiliser pour les différents sous-systèmes.

Ce même fichier de coordination est généralement utilisé pour raffiner les protocoles de communication entre les différents sous-systèmes. L'interprétation des liens entre les différents sous-systèmes hétérogènes nécessite une étape spécifique de synthèse de la communication.

Dans le cas de la conception multilangage, il sert de base pour le bénéfice d'une meilleure vision et compréhension de la conception. Le fichier de coordination permet, lors de la conception, la séparation des tâches à réaliser en les distribuant dans des équipes spécialisées. Dans

[9] E/S : l'abréviation signifie Entrées-Sorties.

les grands groupes de conception les spécificités sont séparées en groupe de travail et même dans certains cas, en sites géographiques différents. Pour coordonner un projet de conception de système complexe multilangage, une équipe supplémentaire peut être nécessaire pour réguler les différentes tâches à réaliser, fixer les contraintes aux différentes équipes et vérifier que chacune d'entre elles respecte les contraintes de coordination. Cette équipe doit alors superviser et maîtriser la conception globale du système. Pour ce faire, elle a pour objectif de partitionner de façon efficace la spécification initiale, de distribuer les partitionnements dans les différents groupes de modélisation, de coordonner les tâches en terme de délais et enfin de valider le système par des techniques telles que la cosimulation.

L'utilisation d'équipes différentes entraîne également une maîtrise complète des temps de réalisation de chacune d'entre elles. En effet, une équipe de modélisation hydraulique peut réaliser sa modélisation dix fois plus vite qu'une équipe de modélisation logicielle. Il est important d'en tenir compte et pour cela le concepteur peut éventuellement avoir recours à des outils d'ordonnancement comme le réseau PERT pour l'aider à gérer les délais en fonction des tâches lorsque la complexité devient trop importante.

2.3 Les Différents Types de Moteurs de Simulation

Il existe deux approches principales pour la cosimulation de systèmes hétérogènes : l'approche mono-moteur et l'approche multi-moteurs.

2.3.1 Cosimulation Mono-Moteur

La cosimulation mono-moteur (cf. Figure 2.4) correspond à la modélisation compositionnelle (cf. §1.5.1.2). Elle consiste à intégrer les sous-systèmes à simuler en une représentation unifiée qui sera utilisée pour la vérification et la conception du comportement global du système. Cette vérification peut se faire par une simple compilation et exécution, voire déboggage. Cette méthode permet d'avoir une complète cohérence et une vérification parfaite de la définition des interconnexions. Néanmoins, cette méthode nécessite une visualisation complète du modèle afin d'identifier les moyens d'ajouter des liens pour tracer les signaux et visualiser les variables nécessaires au déboggage éventuel. De plus, l'intégration d'un nouveau langage peut s'avérer très complexe du fait qu'il soit nécessaire de transformer sa syntaxe et sa sémantique pour le rendre compatible à une représentation unifiée.

L'utilisation d'une telle méthode peut être intéressante si la représentation unifiée est suffisamment complète pour représenter les systèmes à des niveaux d'abstraction allant du niveau très bas au niveau très haut. Cependant, ce type de simulation ne peut représenter la notion d'exécution parallèle sans une méthode de ré-ordonnancement dynamique des tâches au cours de la simulation. Elle n'est pas très représentative des problèmes d'interfaçage de bas niveau.

Cette méthode comporte des aspects très attractifs à condition que :
- les applications ne soient pas concurrentes ou parallèles,
- la génération du code intermédiaire soit facilement et rapidement réalisable pour chaque sous-système,
- la chaîne de compilation soit rapide,

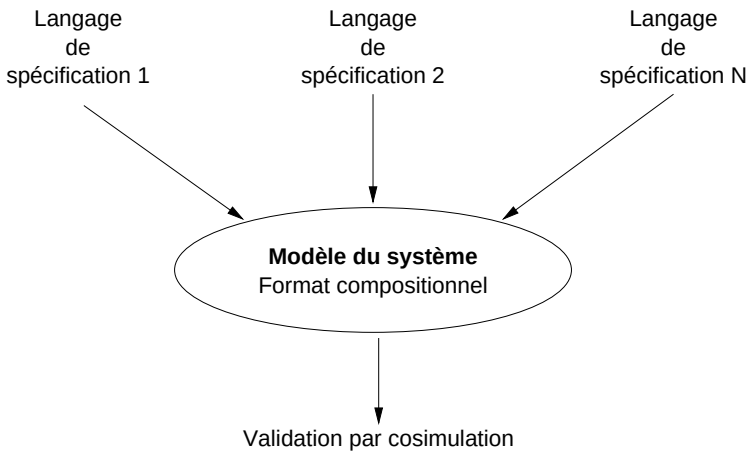


FIG. 2.4 – Cosimulation mono-moteur

– l'édition des liens des modules puisse être aisément mise en oeuvre afin de permettre l'établissement d'un lien avec un programme d'espionnage des échanges de données.

L'avantage de cette méthode est qu'elle peut être très rapide à condition que la chaîne finale de compilation du langage intermédiaire soit performante au niveau exécution.

2.3.2 Cosimulation à Base de Plusieurs Moteurs

La cosimulation multi-moteurs consiste à fournir un moteur de simulation pour chaque langage engagé dans le processus de cosimulation (voir figure 2.5). Chaque module est simulé par un simulateur approprié. La cosimulation devient un échange de données entre simulateurs dont le cheminement est tracé en accord avec un fichier de coordination. L'avantage d'une telle méthode permet l'utilisation d'outils existants pour réaliser la simulation et le déboggage. Néanmoins, il faut s'assurer que les simulateurs placés dans l'environnement de cosimulation permettent d'extraire leurs données de simulation vers l'extérieur, via une API[10] ou une interface de communication vers le langage C. Il faut réaliser des modules d'entrées/sorties permettant d'interfacer le simulateur avec le modèle de communication. Le fichier de coordination inclus dans l'environnement doit pouvoir identifier l'ensemble des données qui transitent dans le support de communication. Le modèle de communication doit, à partir du fichier de coordination, réaliser une vérification des interconnexions ou/et des types, exécuter les résolutions si celles-ci sont nécessaires et contrôler les directions des données qui transitent. Le fichier de coordination doit définir les simulateurs de l'environnement et décrire l'interconnexion des simulateurs comme la déclaration d'entités et leurs instantiations dans un fichier VHDL. Le fichier de coordination décrit le système complet, en y définissant les applications, les connexions, sans oublier la déclaration des types et les propriétés supplémentaires utiles au modèle de communication pour qu'il puisse correctement échanger les données. Le comportement des modules en interne des simulateurs n'interviennent pas dans ce fichier de coordination : un ensemble de boîtes noires est uniquement visible.

L'approche multi-moteurs est un environnement de simulation qui offre à l'utilisateur une solution modulaire et généralement souple d'utilisation.

[10] API : (Application Programmable Interface) bibliothèque permettant aux développeurs un accès réservé aux ressources du simulateur.

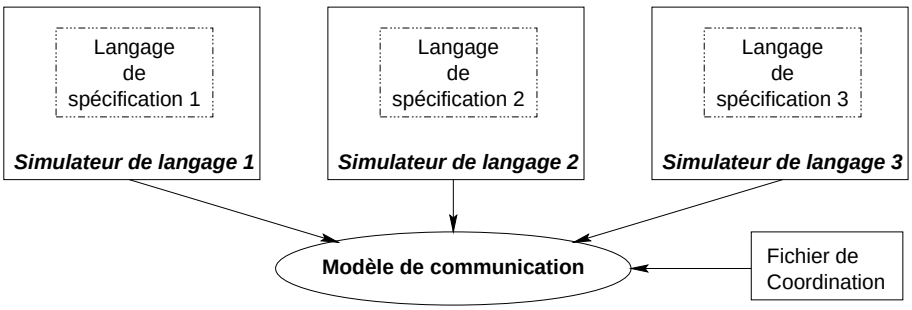


FIG. 2.5 – Cosimulation multi-moteurs

Cependant, elle implique une nette perte de performance face à une autre méthode telle que celle mono-moteur.

2.3.2.1 Interfaçage des Outils de Simulation Dans un Environnement de Cosimulation

L'interfaçage d'un outil de simulation dans un environnement de cosimulation dépend du type de cosimulation mise en oeuvre :

Dans le cas d'une méthode mono-moteur, l'interface consiste en une conversion de format. Étant donné que la spécification est réalisée en un langage X, alors que la simulation est exécutée dans un langage intermédiaire Y, le problème est de passer du langage X au langage Y. Ce passage consiste en la traduction de X en Y. Pour ceci, il faut essentiellement tenir compte de la différence de syntaxe et de sémantique des deux langages. En effet, la traduction d'un langage à un autre n'est pas forcément simple. Les langages peuvent avoir des concepts différents (types différents, possibilités différentes).

Dans le cas d'une méthode à plusieurs moteurs, la méthode consiste à interfacer ces simulateurs. La difficulté repose sur la possibilité d'extraire des données échangées entre un simulateur et son environnement extérieur. Actuellement, la plupart des simulateurs commerciaux permettent d'échanger leurs résultats vers l'extérieur, via des appels systèmes (eg. des fonctions C) ou offrent même la possibilité de diriger l'évolution du simulateur par l'intermédiaire d'appels de fonctions externes. Cependant, l'accès ou la connaissance de ces primitives sont bien souvent la propriété du fournisseur du simulateur. De ce fait, les bibliothèques utilisées pour établir la communication entre le simulateur et l'extérieur sont en général mal documentées. Par ailleurs, si l'on considère le faible nombre d'utilisateurs de telles primitives, cela n'incite pas les fournisseurs à s'y investir davantage.

Enfin, un problème essentiel apparaît dans le cas où la notion de synchronisation temporelle est prise en compte dans la simulation. En effet, il est parfois difficile d'extraire l'avancement temporel d'un simulateur durant son exécution. De plus, certains simulateurs ne contiennent pas de notions temporelles : ce sont des simulateurs purement événementiels. La cosimulation entre les modules qui utilisent des notions de temps différents est parfois difficile à mettre en oeIJuvre. Il faut noter que dans le cas de la méthode mono-moteur, la cosimulation avec notion temporelle est difficilement réalisable. Une fois le système traduit en langage intermédiaire, il faut réaliser un ordonnancement de l'ensemble des taches parallèles. Cette opération d'ordonnancement complique énormément la méthode mono-moteur.

2.3.2.2 Conception Modulaire

Ces deux méthodes de cosimulation permettent la conception modulaire. La spécification du système complet peut se décomposer en plusieurs sous-systèmes. Le concepteur peut donc organiser ses équipes de travail de la même façon qu'il découpe son système. Il peut aussi réorganiser les modules à concevoir en fonction des délais de réalisation, des difficultés potentielles de conception, et du choix de la description du sous-système. Cette modularité peut aussi impliquer une conception concurrentielle des différents groupes de conception permettant ainsi d'accélérer la réalisation globale du projet. La modularité reste ainsi présente dans les deux différentes méthodes, puisque dans les deux approches les sous-systèmes sont validés séparément avant d'être simulés conjointement.

Dans la méthode multi-moteurs, on conserve cette modularité tout au long du flot de conception. En effet la plupart du temps le concepteur d'un sous-système connaît également l'utilisation du simulateur ou du déboggueur du sous-système qu'il doit simuler. En cas d'erreur, ce concepteur est capable, de part son expérience, de localiser le problème potentiel décelé lors de la simulation.

Dans le cas de l'utilisation de la méthode mono-moteur, cette modularité s'arrête au moment où les différents sous-systèmes sont traduits en un même langage pour être ensuite simulés. En ce qui concerne la méthode multi-moteurs, la modularité est existante à tous les niveaux d'abstraction, car ce sont également les différents outils de simulation de chaque sous-système qui interviennent pour la simulation du système final.

2.3.2.3 Simulation à Plusieurs Niveaux d'Abstraction

La flexibilité d'un outil de cosimulation se distingue de part sa possibilité de simuler à différents niveaux d'abstraction. Dans la méthode mono-moteur, descendre d'un niveau d'abstraction consiste en deux points :

- tout d'abord, la ré-exécution de toute la traduction des langages de spécification vers le langage intermédiaire. Ce langage intermédiaire peut être complètement différent d'un niveau à un autre,[11]
- ensuite la ré-exécution de la chaîne de compilation du modèle complet en langage unifié ou de chaque sous-système, à condition qu'une édition de lien réalise l'assemblage des sous-systèmes pour former le système complet.

Dans la méthode multi-moteurs, descendre d'un niveau se résume en deux étapes :

- en premier lieu, le changement du fichier de coordination afin de mettre à jour les types qui pourraient avoir subi une transformation,
- ensuite l'utilisation du simulateur adéquat pour le nouveau modèle si toutefois il existe. L'utilisation d'un même simulateur ou d'un simulateur différent se fait selon :
  - la capacité du simulateur à simuler au niveau d'abstraction inférieur. Dans ce cas on utilise le même simulateur,
  - l'étendue des transformations subies par le modèle (par exemple le passage dans un outil de co-conception (SDL vers C & VHDL)). Dans ce cas, on utilise un simulateur

[11] Le passage d'un niveau d'abstraction à un autre nécessite des informations supplémentaires. Si le langage intermédiaire n'est pas adapté pour les rajouter, un nouveau langage sera nécessaire.

différent. Il faut modifier l'environnement de cosimulation pour permettre son intégration.

La simulation peut également se faire sur des niveaux d'abstraction mixtes. Le concepteur peut simuler son système en définissant certains sous-systèmes à des niveaux d'abstraction différents par rapport à d'autres.

Il peut donc simuler, dans le même environnement, une partie de son système à très bas niveau d'abstraction (proche de la réalisation - prototypage), alors que le reste est simulé à très haut niveau d'abstraction (StateCharts, SDL, ...).

2.3.2.4 Possibilité de Génération Automatique des Modules à Cosimuler

Les modules qui doivent être cosimulés, nécessitent de légères modifications avant d'être insérés dans l'environnement. Ces modifications ont deux sens différents suivant les deux approches. Dans le cas de la cosimulation mono-moteur, ces modifications désignent davantage l'étape de synthèse des interfaces, alors que dans l'approche multi-moteurs elles permettent de spécifier aux modèles les ports qui vont être cosimulés.

- Dans le premier cas, il s'agit de structurer la communication par l'adjonction de code pour faire correspondre les types des données échangées entre les modèles. Cette étape fait partie de la phase de synthèse d'interface qui consiste à choisir le protocole de communication entre les modules en fonction des niveaux d'abstraction. Cette génération automatique des modifications des modèles pour la méthode mono-moteur doit, en quelque sorte, se faire en étroite collaboration avec l'étape de la synthèse d'interface. Ainsi, l'édition de lien final des blocs entre eux n'engendre pas de problèmes.
- Dans le second cas, cette synthèse d'interface peut se faire plus tard dans le flot de conception. Dans ces conditions, le moteur de simulation doit se charger de la gestion de la communication et de la correspondance des types et des protocoles. La génération automatique des modifications intervient, ici, par l'adjonction de code à l'intérieur des modèles ou par génération d'un programme dont l'objectif est d'extraire les données nécessaires du simulateur via son API. Ainsi, ces interfaces vont permettre de spécifier au(x) simulateur(s) les données à échanger vers l'extérieur durant la simulation. La génération automatique consiste, d'une part, à modifier les modèles spécifiés et, d'autre part, à générer une partie du modèle de communication (figure 2.5). C'est le cas notamment de l'outil VCI (page 65).

  L'insertion et la génération automatique des interfaces consistent, pour la méthode multi-moteurs, à :
  - faire référence à une bibliothèque de ports dont l'architecture à été spécifiée pour écrire ou lire les données vers l'extérieur du simulateur durant son exécution (Procédures I/O[12]),
  - déclarer par ajout de code à l'intérieur des modèles, les ports qui doivent être cosimulés. Ces lignes permettent d'indiquer au simulateur qu'il doit faire appel à une procédure d'entrée/sortie lorsque le moteur de simulation rencontre un de ces ports. Ces déclarations sont faites différemment suivant les simulateurs :
    - séquentialisées en insérant tout au long du séquencement, des fonctions prototypes d'entrées/sorties afin d'extraire les données simulées vers l'extérieur à chaque exé-

cution des itérations rajoutées,
- instanciées en réalisant dans la déclaration une instanciation des ports à cosimuler.

Le problème le plus difficile à régler consiste à gérer les différences de types entre les modèles à simuler. Le générateur doit être capable de prendre en compte ces problèmes et doit se ramener à des types intermédiaires pour les faire correspondre à tous les modèles. Dans le cas de certains types complexes (structures, unions, objets, ...), la génération automatique peut s'avérer complexe.

2.4 Modélisation du Temps

La difficulté majeure de la réalisation de la cosimulation provient de la notion de temps qui est différente entre le sous-système logiciel embarqué, le matériel et l'environnement qui l'entoure. Pour simuler de tels systèmes avant leur implémentation, l'insertion du temps dans les outils de cosimulation est nécessaire [YC97][LLSV98][HH93]. Nous présenterons dans cette section les différences de fonctionnement entre la simulation fonctionnelle d'un système et la simulation au niveau temporel.

2.4.1 Validation Fonctionnelle

La validation fonctionnelle consiste à vérifier un système à très haut niveau d'abstraction, lorsque la notion de temps n'est pas nécessaire à la simulation. Les temps de propagation ne sont pas à prendre en compte puisque les échanges de données entre les simulateurs sont perçus comme des événements au niveau du temps d'exécution. Cette validation fonctionnelle consiste à échanger les données dans l'ordre de leurs modifications. Ce type de simulation permet de valider les aspects du système qui sont indépendants du temps. Dans le cas où la communication entre les modules à simuler utilise un modèle indépendant du temps, alors ce type de simulation peut aussi valider la communication du système. Dans ce genre de validation il est nécessaire de dissocier deux types de comportement de simulation : d'une part l'existence de simulateurs séquentiels et d'autre part de l'existence de simulateurs événementiels.

2.4.1.1 Les Simulateurs Séquentiels et Evénementiels

Les simulateurs séquentiels et événementiels se distinguent par deux comportements différents comme le suggère la figure 2.6. Les simulateurs séquentiels sont définis par un ensemble de séquences d'instructions et où chacune des instructions possède un temps d'exécution non nul. Les simulateurs événementiels agissent eux sur des événements dont la modification peut avoir un temps nul.

- Pour les simulateurs séquentiels, la lecture comme l'écriture d'une donnée prend un temps λ d'exécution simulé. Ainsi, il ne peut y avoir dans le modèle d'exécution séquentiel deux données modifiées au même moment. Le temps de simulation d'un modèle séquentiel est donc proportionnel à l'avancement de l'exécution du programme. Sur l'exemple de la figure 2.6, les modifications des variables "*Out1*", "*Out2*" et "*Out3*" n'apparaîtront

[12] Procédures I/O : Procédures d'entrées sorties inclues dans une bibliothèque liées au moteur de simulation du simulateur et permettant de communiquer vers l'extérieur.

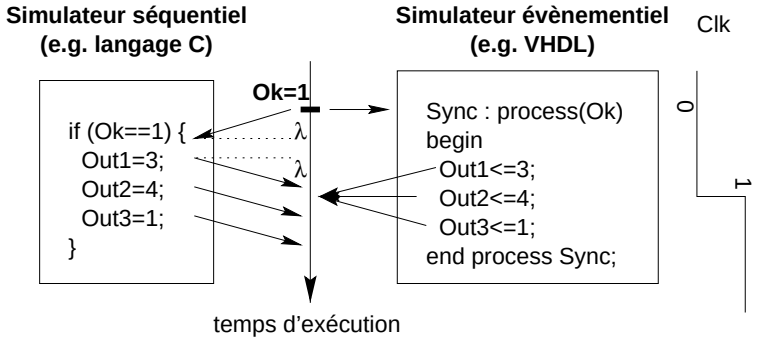


FIG. 2.6 – Simulateurs séquentiels et événementiels

jamais au même temps mais successivement et au moins espacé du temps d'écriture λ (si le temps d'écriture est égal au temps de lecture).

– A l'opposé, les simulateurs événementiels sont en général cadencés par une horloge ou par les modifications d'événements sur les données. C'est pourquoi, on distingue généralement dans les simulateurs événementiels, les variables (comportement séquentiel) et les signaux (comportement événementiel). Dans ce type d'exécution, la lecture de la modification d'un signal correspond à un temps d'exécution en lecture qui peut être égal à zéro. La modification des signaux peut intervenir sur un événement comme le front d'une horloge. Dans le cas de notre figure "*Out1*", "*Out2*" et "*Out3*" sont modifiés en même temps. Il n'y a pas d'ordre dans la modification de ces signaux, car cette modification intervient pour les trois signaux au même moment c'est à dire lors de l'évènement produit par le changement de OK.

2.4.1.2 Couplage des Simulateurs Séquentiels et Evénementiels

Il arrive souvent que des simulateurs événementiels et séquentiels soient utilisés dans un même environnement de simulation. Dans ce cas de figure, il faut tenir compte du fait que le simulateur événementiel puisse avoir plusieurs données modifiées à échanger en un même temps. Il faut donc obligatoirement avoir les moyens d'identifier le début et la fin de l'événement du simulateur. Entre ces points événementiels, il faudra échanger les données en interdisant d'une façon ou d'une autre aux simulateurs destinataires de lire ces données entre ces deux points événementiels.

La figure 2.7 illustre le mécanisme : le simulateur événementiel A doit échanger trois valeurs (Val1, Val2, et Val3) vers deux simulateurs B et C. Afin de respecter le bon transfert des données, il faut que les simulateurs B et C (et surtout C puisqu'il reçoit deux données) soient bloqués pendant cet échange. Cette particularité est uniquement valable lorsque des simulateurs événementiels échangent leurs données vers des simulateurs séquentiels et seulement si le simulateur cible a plus d'une données à transmettre au simulateur destinataire. Afin de pallier à ce problème, il faut :

– pouvoir déterminer le début et la fin d'un événement durant la simulation :
  – soit par l'utilisation de fonctions de bibliothèques appropriées (accessibles dans les API des simulateurs par exemple),
  – soit par la vérification de l'évolution du temps du simulateur ou du contrôle du passage d'un événement à un autre durant la simulation.

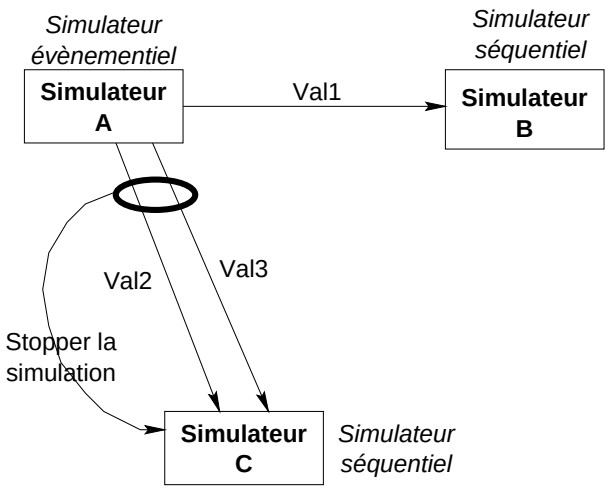


FIG. 2.7 – Synchronisation entre simulateurs événementiels et séquentiels

- pouvoir bloquer les simulateurs destinataires pendant le transfert des données modifiées durant l'événement :
  - soit par l'ajout d'un jeu de fonctions systèmes permettant d'interrompre le simulateur en cours d'exécution pour le mettre en état de veille. Il ne reprend alors son activité que lorsqu'il reçoit la dernière donnée de l'événement. Un ordre d'interruption peut être envoyé par le simulateur éditeur d'un événement et se propager à tous les simulateurs destinataires de celui-ci. Cet ordre peut bloquer le simulateur pendant la modification des données jusqu'à un nouvel ordre de déblocage,
  - soit par l'insertion d'une information d'arrêt dans la trame de la première donnée modifiée de l'événement et une information de fin d'événement dans la dernière trame envoyée. Entre ces deux trames, le simulateur sait qu'il n'est pas autorisé à lire les données modifiées lui parvenant, mais qu'il doit plutôt attendre la fin de ces modifications.
- pouvoir inclure une synchronisation supplémentaire dans le bus de cosimulation pour considérer les données qui arrivent dans un événement comme faisant partie du même temps simulé.

De toute façon, l'objectif est de respecter un ordre de transmission des données dans le bus, c'est à dire tenir compte du comportement du modèle récepteur de ces données. L'essentiel est de respecter l'ordre d'envoi des données, tout particulièrement si le système est sous forme CDFG,[13] car l'ordre d'envoi des données est une information importante dans le sens où les données doivent arriver avant les signaux de contrôle.

2.4.2 Validation Temporelle

La validation temporelle est une étape incontournable pour la validation à très bas niveau. Contrairement à la cosimulation fonctionnelle, où la simulation est contrôlée par des événements sur les données, la cosimulation temporelle consiste à échanger les données dans des fenêtres temporelles [LDA93a]. Il convient néanmoins de prendre certaines précautions dans ce

[13]CDFG : Control-Data Flow Graph : c'est un modèle très utilisé pour représenter les systèmes orientés transfert de données. Plusieurs outils de co-conception utilisent le modèle CDFG.

type de simulation, sachant que si une donnée n'est pas consommée dans une fenêtre temporelle donnée, elle peut être perdue. Ce type de simulation implique que les simulateurs associent une fenêtre de validité définie par un intervalle temporel de simulation spécifique à chaque opération. Etant donné que la cosimulation peut impliquer l'exécution concurrentielle de plusieurs simulateurs qui ont différentes vitesses d'exécution et de base temporelle,[14] un modèle de synchronisation est donc nécessaire pour coordonner l'exécution parallèle des différents simulateurs. Il existe plusieurs modèles de synchronisation.

Le plus simple utilise le modèle maître-esclave (cf. §2.5.1). Avec ce schéma [HH93], un simulateur joue le rôle de maître alors que les autres procèdent comme étant les esclaves. Le maître fixe les tranches temporelles pendant lesquelles les esclaves doivent simuler et s'aligner. Ce modèle est simple mais impose de nombreuses restrictions sur l'organisation du système.

Le schéma, le plus général est le modèle asynchrone. Dans ce cas, les simulateurs échangent leurs données avec leur temps [CHL97]. Les simulateurs lents empilent leurs données pour respecter le temps. Par contre, le retour en arrière des simulateurs reste la complexité d'une telle implémentation.

Au niveau des simulateurs, la notion du temps ou l'extraction du temps de simulation est nécessaire pour pouvoir réaliser un environnement de cosimulation temporelle. Son obtention peut impliquer :

- d'annoter le code source ou le modèle à simuler, afin de représenter le temps d'exécution de ses instructions,
- d'avoir la possibilité de connaître l'évolution du temps lors de la simulation soit par des primitives, soit par espionnage du simulateur.

La validation temporelle consiste donc à synchroniser l'évolution du temps des simulateurs pour permettre une simulation la plus réelle possible. Elle implique la mise en place d'une synchronisation qui peut être du type maître-esclave ou distribuée avec un modèle de synchronisation qui peut être, par exemple, celui de la barrière temporelle.

2.4.2.1 Granularité de la Précision en Fonction des Simulateurs

Du fait de la nécessité d'annoter le temps dans le(s) modèle(s) lors d'une cosimulation temporelle, on se trouve confronté à certaines granularités. C'est le cas d'un programme C grâce auquel l'on peut annoter la durée d'exécution de ses procédures, fonctions, etc.. voire exiger une simulation plus réaliste. A très bas niveau, le concepteur aura donc besoin de cibler son code C sur un processeur donné [LLSV98]. Ainsi, la cosimulation à très bas niveau devra contenir un modèle simulable du processeur sur lequel est implémenté le code du logiciel compilé. La notion du temps qui, à haut niveau, représentait le nombre approximatif d'instructions exécutées, représente au niveau du modèle de processeur une notion d'instructions assembleur exécutées et au niveau cycle représente le nombre de cycles exécutés. Descendre dans les niveaux d'abstraction implique généralement une fréquence d'échanges plus fine : l'instruction assembleur peut représenter de 1 à plus de 10 cycles d'horloge.

Plus la granularité de la simulation est fine, plus les fenêtres temporelles de simulation sont fines, plus la précision est grande et plus les modèles de simulation sont lents. La différence peut exister suivant les simulateurs ou les modèles d'exécution utilisés, mais de toute évidence la simulation comme la cosimulation nécessite une quantité de calcul plus conséquente.

[14] La base temporelle est parfois très différente. Dans le cas d'un système mécatronique, la partie mécanique peut être simulée en seconde alors que la partie matérielle en Nanoseconde.

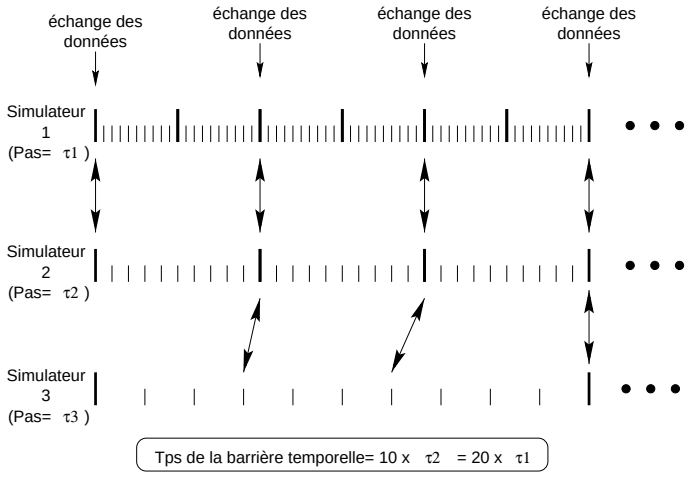


FIG. 2.8 – Synchronisation par barrière temporelle

2.4.3 Modèle de la Barrière Temporelle

Le modèle de synchronisation à base de barrière temporelle "*lock step model*" (figure 2.8) consiste à synchroniser l'ensemble des simulateurs sur un pas global de simulation. Son avantage vient de sa simplicité de mise en oeIJuvre même s'il apparaît comme étant un inconvénient lorsqu'on utilise des simulateurs possédant des vitesses de simulation différentes.

Lorsqu'on introduit la notion de temps dans la simulation, il convient de dissocier les notions de temps suivantes représentées sur la figure 2.8 :

- Le pas de simulation des simulateurs représenté par la lettre τ. Il correspond à la valeur utilisée pour discrétiser la simulation. Par exemple, nano-seconde pour le VHDL, l'instruction pour un ISS ou la milli-seconde pour la simulation mécanique.
- Le temps global simulé est représenté par le temps de la barrière temporelle T. Il correspond à la quantité de calcul nécessaire par les simulateurs pour obtenir le résultat simulé.
- Le temps réel global de la simulation correspond au temps machine nécessaire pour réaliser le temps global simulé. Ce temps est directement dépendant de l'occupation de la machine, des modèles et à la performance des algorithmes des simulateurs.

$$Temps\,Global\,simule\,d'une\,simulation \sim \sum_{t=0}^{T} \left(\sum_{\tau_1=0}^{t} Ts_1 \parallel \sum_{\tau_2=0}^{t} Ts_2 \parallel \cdots \parallel \sum_{\tau_n=0}^{t} Ts_n \right)$$

$$Temps\,reel\,global \approx \sum_{t=0}^{T} \left(\max(\sum_{\tau_1=0}^{t} Ts_1, \sum_{\tau_2=0}^{t} Ts_2, \cdots, \sum_{\tau_n=0}^{t} Ts_n) + ND_t \times TpsD \right)$$

Ts_n : Temps réel simulé mis par le simulateur n pour réaliser un pas de simulation.
τ_n : Valeur d'un pas de simulation du simulateur n.
t : Temps de la barrière temporelle. C'est la valeur d'un pas global de simulation.
ND_t : Nombre de données modifiées au pas t de la barrière temporelle.
$TpsD$: Temps de traitement d'une donnée modifiée.

Ce modèle, quoique simple, permet de reproduire un réalisme temporel presque parfait de la simulation. D'autres procédés sont envisageables afin d'accélérer le mécanisme (insertion de FIFO,[15] stockage des données pré-calculées) mais il en résulte une perte de convivialité qui

[15]FIFO : procédé de stockage des données sous forme de file ("First-In, First-Out").

est due au décalage des temps locaux de simulation. En effet, si l'on utilise des simulateurs qui s'exécutent à des vitesses différentes un décalage des résultats obtenus par les simulateurs peut avoir lieu entre eux lors de leurs visualisations, au moment d'un déboggage par exemple. Le problème fondamental d'un tel modèle, et d'une synchronisation temporelle en général, intervient lorsqu'on simule entre eux des simulateurs dont les temps de calcul Ts sont différents. Par exemple, si l'on simule un modèle VHDL dont le pas de calcul τ est calculé en nano-seconde avec un modèle Matlab qui lui doit calculer en seconde, dans le pire des cas, le simulateur VHDL devra effectuer 10^9 fois plus d'itérations de calcul que le modèle Matlab. Ainsi, le temps de simulation nécessaire pour simuler un pas global d'une seconde de la barrière temporelle sera complètement dépendant de la vitesse du simulateur VHDL pour simuler cette seconde.

De plus, le choix du pas de simulation temporelle, qui spécifie la fréquence des échanges des simulateurs peut être judicieux. Il est évident que si le temps de la barrière est fixé au plus petit pas de simulation des simulateurs, alors la précision de la simulation est idéale. Néanmoins, le temps de simulation va s'accroître de façon exponentielle si les pas de simulation des simulateurs sont très distincts. Il faut donc laisser le libre choix au concepteur de trouver un compromis entre la précision de la cosimulation et la rapidité globale de la simulation. Bien évidemment, si la fréquence des échanges n'est pas homogène le pas global de simulation doit être le plus fin possible.

Enfin, l'élaboration d'une simulation temporelle exige nécessairement de pouvoir extraire le temps des simulateurs et éventuellement de mettre en place des moyens dans le but de commander l'arrêt d'un simulateur au niveau temporel. L'arrêt d'un simulateur dans un environnement de simulation temporel peut être réalisé de différentes façons :

- soit en bloquant le simulateur par des fonctions systèmes (signaux, sémaphore,..). Dans ce cas, dans les moments d'attente, le simulateur est la plupart du temps incontrôlable au niveau de sa GUI.[16] De ce fait, le simulateur perd de sa convivialité et l'environnement de cosimulation doit pouvoir contrôler ces mécanismes de blocage, déblocage des simulateurs en fonction du temps de simulation,
- soit en utilisant des fonctions de contrôle extérieures au simulateur. Pour cela, il est nécessaire d'avoir la possibilité de démarrer et d'arrêter le simulateur de son extérieur ou de pouvoir contrôler l'évolution du simulateur : le faire reculer , re-simuler, etc... Ce type de fonctions est rarement disponible, mais cette approche permet de garder une grande convivialité de simulation, cruciale principalement lors du déboggage.

La cosimulation temporelle est un atout essentiel pour simuler un système à très bas niveau. Néanmoins, le prix d'une telle simulation implique un temps de calcul souvent énorme, surtout si les applications mises en jeu ont des pas de calcul très différents. Enfin, la plupart du temps, suivant les possibilités des simulateurs, elle engendre une perte de la convivialité de simulation (GUI bloqué, affichage saccadé, etc..).

2.5 Modèles de Synchronisation

La synchronisation est l'élément essentiel d'un outil de cosimulation. Alors que dans la méthode mono-moteur elle est implicite, dans la méthode multi-moteurs elle représente le fonctionnement de l'outil et doit être scrupuleusement étudiée et vérifiée.

[16] GUI : Graphical User Interface où interface graphique

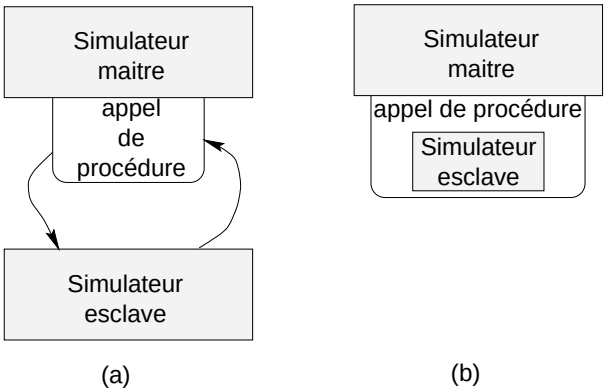


FIG. 2.9 – Cosimulation Maître-Esclave

Dans la méthode mono-moteur, la synchronisation est inclue dans le langage intermédiaire. Les primitives d'échanges de données sont introduites dans le langage avant compilation et sont perçues comme des affectations au moment de l'exécution. C'est pourquoi, ces affectations vont décrire les échanges de données sans nécessiter de synchronisation spécifique. Dans le modèle distribué, la simulation se comporte comme un ensemble de processus s'exécutant en parallèle, où les processus sont les simulateurs mis en IJoeuvre dans l'environnement. Suivant les besoins, la synchronisation doit définir la manière dont les données vont s'échanger. Il existe deux modèles principaux de synchronisation : le modèle maître-esclave et le modèle distribué.

Le mode maître-esclave comprend un simulateur maître et un ou plusieurs simulateurs esclaves. Dans ce cas, les simulateurs esclaves sont exécutés en utilisant des appels de procédures.[17] Ces appels de procédures peuvent être implémentés selon les possibilités suivantes :

- en appelant des procédures étrangères (par exemple, des procédures écrites en C) par le simulateur principal et qui sont liées au simulateur esclave par édition de lien (Figure 2.9-a),
- en encapsulant le simulateur esclave dans un appel de procédure. Le simulateur esclave est perçu comme une fonction par le simulateur principal (Figure 2.9-b).

La plupart des simulateurs commerciaux proposent des moyens de base permettant de réaliser des appels extérieurs (*FMI*[18] pour Leapfrog (VHDL-Cadence), *CLI*[19] pour VSS (VHDL-Synopsys) [Inc98], FKI[20] pour Voyager (VHDL-Ikos)), ou fournissent même des bibliothèques permettant de commander le simulateur par un programme extérieur (*ClientSDL* pour Object-Geode, "*Engine*" pour Matlab).

Bien qu'utile et bien souvent facile à mettre en place, la cosimulation maître-esclave présente des inconvénients. Le mode de synchronisation représenté sur la figure 2.10-a reproduit l'échange de données entre deux simulateurs. Les échanges en lecture/écriture entre les deux simulateurs se font de façon alternée. Ce mode de synchronisation ne permet pas d'exécuter les simulateurs de façon parallèle. Lorsque le simulateur maître s'exécute, alors le simulateur esclave est stoppé, et vice versa.

Les modes de synchronisation utilisables pour le modèle distribué sont représentés sur la figure 2.10a et b. Le mode de synchronisation parallèle permet une exécution simultanée des

[17] Un appel de procédure ("procedure call") bloque l'appelant pendant l'exécution de cette procédure.
[18] FMI : Foreign Model Interface.
[19] CLI : C Language Interface.
[20] FKI : Foreign Kernel Interface.

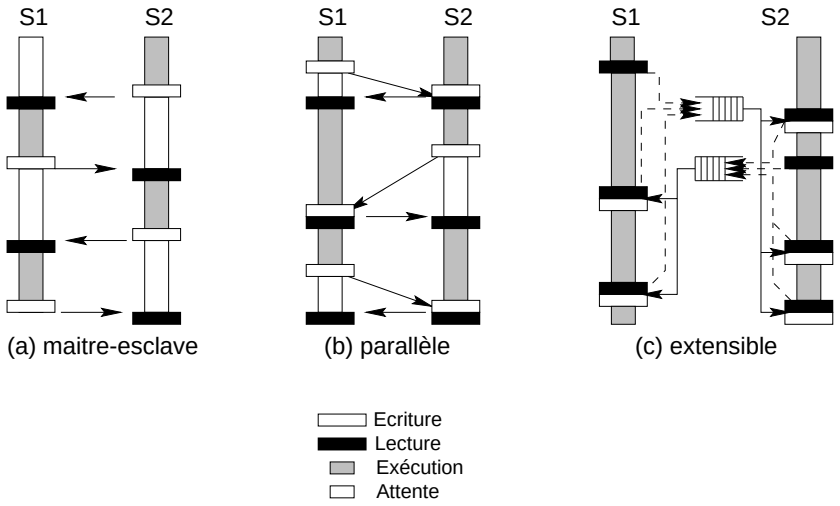


FIG. 2.10 – Mode de synchronisation entre deux simulateurs

simulateurs. Les deux simulateurs progressent en parallèle et la synchronisation est assurée des deux côtés. Dans les deux sens les transactions de données ne sont effectuées que lorsqu'une instruction d'E/S est invoquée par un simulateur. Ainsi on s'assure que l'échange n'est effectué qui si cela est nécessaire. Chaque simulateur envoie ses données et reste en attente pour la réception des données provenant de l'autre côté. Des variantes du mode parallèle existent : elles peuvent consister à bloquer l'évolution du simulateur tant que le simulateur n'a pas reçu une donnée précise en entrée. Ces variantes de diverses sortes permettent généralement de partager des signaux entre plusieurs applications (cf. signal CLK), et/ou synchroniser des applications entre-elles par un enchaînement d'événements en fonction du temps.

Un dernier mode de synchronisation qui peut être utilisé, et parfois implémenté de façon intrinsèque par le support de communication du bus de cosimulation, est le mode extensible de la figure 2.10c. Ce mode permet d'optimiser le parallélisme d'exécution des simulateurs en utilisant un moyen de stockage pour les données.

Ces différents modes de synchronisation peuvent être gérés à l'intérieur des primitives d'E/S ou encore par un arbitre qui gère les données qui transitent. Cet arbitre définit le routeur des données dans les plate-formes de cosimulation. Le support de communication utilisé par le bus de cosimulation contient également un mode de synchronisation qui lui est propre (généralement sécurisé et bidirectionnel). C'est pourquoi les modes de synchronisation utilisés pour l'environnement de cosimulation doivent également tenir compte du type de support utilisé pour le transport des données.

2.5.1 Modèle de Synchronisation Maître-Esclave

Ce type de cosimulation est bien adapté pour les applications sous forme flux de données ("*data flow*"), ou pour les applications qui transfèrent de grands blocs de données ("*burst*"). Or il n'est pas très fonctionnel lorsqu'il s'agit d'applications basées sur le contrôle. Pour des applications orientées contrôle, ce modèle exige que le logiciel soit découpé en tranches et un arrangement sophistiqué, tel que la sauvegarde des sorties, doit être réalisé pour les futures invocations de la procédure.

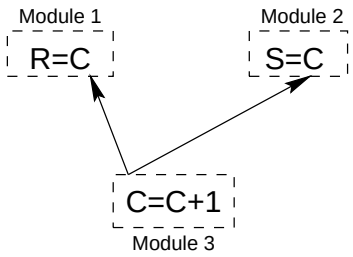


FIG. 2.11 – Problème de l'ordre d'exécution des simulateurs

Dans le cas de la cosimulation maître-esclave les liens entre les simulateurs sont relativement simples lorsque le nombre de simulateurs ne dépasse pas deux. S'il y a plus de deux simulateurs, cette méthode peut engendrer une dépendance de l'ordre d'échange des données en fonction de l'ordre d'exécution des modules.

L'exemple de la figure 2.11 illustre ce problème. Si un module se charge d'incrémenter une valeur alors que deux autres se chargent de lire cette même valeur, cela peut poser un problème.

Si l'on considère le module comme le producteur de données et les modules 1 et 3 comme les récepteurs, on s'aperçoit qu'il est impossible d'avoir **S** égal à **R** après l'exécution des différentes affectations. En effet, si le module 1 lit la valeur **C** en mode maître-esclave, cela signifie l'exécution du module 3. Ainsi, la valeur **C** sera incrémentée dans ce module 3 à la sortie de l'appel de procédure du module 1. De plus l'opération est identique pour le module 2. Il lira la valeur incrémentée à la sortie de son appel de procédure. Dans tous les cas et même si la lecture se fait dans un ordre différent, **R** ne sera jamais égal à **S** par cette méthode de synchronisation maître-esclave.

En fonctionnement normal, ceci n'est pas vrai car les modules 1 et 2 doivent prendre une valeur différente suivant leurs vitesses d'exécution. Il peut notamment arriver que la valeur C lue par les modules 1 et 2 soit identique si par exemple ces modules (1 et 2) sont beaucoup plus rapides que le processus 3. Dans une méthode maître-esclave, ce principe est difficilement modélisable étant donné que le simulateur qui lit bloque le simulateur lu. Ainsi, la valeur lue entre le processus 1 et 2 aura toujours un intervalle d'au moins 1.

Il est clair que dans une méthode de cosimulation maître-esclave, l'environnement ne peut contenir de connexions à branches multiples. De plus, la notion de temps de propagation des données n'existe pas et nécessite un changement complet du modèle afin de pouvoir l'insérer.

2.5.2 Modèle Distribué

Le modèle de cosimulation distribué (voir figure 2.12) surmonte les restrictions du modèle maître-esclave. Cette approche repose sur un bus de cosimulation qui est utilisé comme protocole de communication. Chaque simulateur communique via ce bus. Le bus est responsable du routage des données entre les différents simulateurs de l'environnement de cosimulation. De plus, il est en charge de synchroniser l'exécution des simulateurs. Les modules accèdent à ce bus via des procédures d'E/S. Les procédures se chargent d'extraire les données du simulateur et de les communiquer au bus de cosimulation et vice et versa. Les primitives d'E/S utilisent toujours des appels de fonction très simples qui permettent de réaliser les échanges avec l'extérieur. Plutôt que d'appeler le simulateur esclave (méthode maître-esclave), la méthode consiste juste à lire et écrire les données dans le bus. Ainsi chaque simulateur agit en tant que maître.

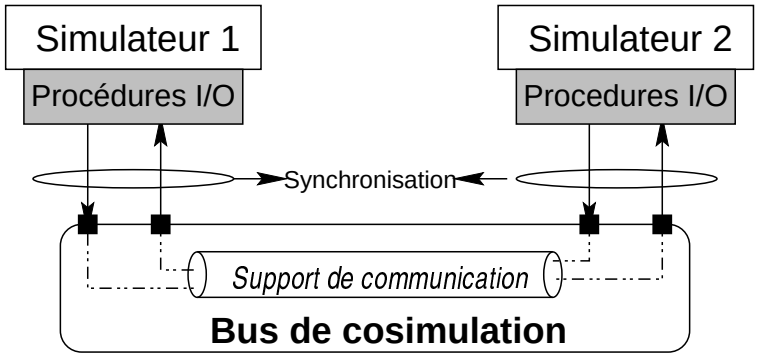


FIG. 2.12 – Cosimulation distribuée

Le bus de cosimulation est en charge d'arbitrer les conflits d'accès aux Données. Il réalise la synchronisation et l'aiguillage des données. De ce fait, comme dans toute conception de bus, un contrôleur est adjoint pour coordonner les données. Pour ce faire il est nécessaire de posséder :

- un programme indépendant qui s'occupe de l'aiguillage des données qui transitent dans le bus de cosimulation (routeur),
- des procédures d'E/S qui permettent de lire les données au fur et à mesure de leur présence en entrée. En sortie, elles doivent gérer les données sans générer un sur-empilement au niveau de l'accès au bus de cosimulation et/ou un dépassement de capacité[21] du bus (bande passante respectée).

Ce modèle comporte plusieurs avantages par rapport à un modèle comme le maître-esclave. Tout d'abord, ce modèle respecte l'exécution des modèles concurrents. En effet, chaque simulateur s'exécute en parallèle et accède au bus dès qu'il le désire via ses primitives d'E/S. Il permet aussi de garder une flexibilité dans l'utilisation des outils et une modularité de simulation durant tout le flot de conception. Désormais, la complexité de ce modèle de cosimulation distribué se concentre sur la plate-forme de cosimulation : la gestion des accès au bus de cosimulation et la coordination des données par le contrôleur du bus. Aussi, elle impose la mise en place de la synchronisation aux différents niveaux d'abstraction et enfin l'élaboration de la plate-forme doit aussi tenir compte du support de communication utilisé pour réaliser le bus de cosimulation. Ce support doit être suffisamment performant en offrant la possibilité d'une communication stable, distante mais surtout il doit offrir un flux suffisamment grand et rapide pour ne pas ralentir l'environnement de cosimulation lors d'échanges de données par gros débit.

2.5.3 Définition du Bus de Cosimulation

Le bus de cosimulation est le support qui va permettre à l'environnement simulé d'échanger ses données. Le support de communication utilisé définit le moyen de transport des données, et les modèles de synchronisation de l'environnement doivent tenir compte de son implémentation et de sa synchronisation intrinsèque. Pour réaliser les échanges de données, ce support de communication doit être choisi ou réalisé. La plupart des systèmes d'exploitation offrent une multitude de services au niveau de la communication. Néanmoins, ceci impose le choix d'un compromis entre les performances de ces supports et leur flexibilité.

[21] Dépassement de capacité ou plus communément appelé "Out of bounds".

Les différents aspects qui influent sur les performances de la communication dans le bus de cosimulation doivent tenir compte des points suivants :

- Les différents simulateurs de l'environnement de cosimulation peuvent être implantés sur la même machine ou sur des machines distantes. La simulation distante implique des temps de communication dépendants de l'architecture du réseau et de ses performances intrinsèques.
- La quantification des échanges. En effet, les modules simulés peuvent avoir à échanger une faible quantité de données ou au contraire un grand nombre. C'est pourquoi la bande passante du support de communication doit être suffisamment grande afin d'éviter l'engorgement des données dans le support de communication.
- Les niveaux de la communication utilisés. Les différents systèmes d'exploitation offrent généralement plusieurs services à différents niveaux pour la communication. Ces services sont directement liés aux différentes couches réseaux d'un système [Tan89] :
  - A haut niveau les services sont généralement des applications comme JAVA, CORBA [GS96],[22] etc.. Ces outils apportent une facilité d'utilisation de la communication mais ils utilisent néanmoins des services de plus bas niveau qui impliquent de ce fait un surcoût des temps de communication. Toutefois, ils sont généralement très flexibles car ils proposent des primitives très complètes pour l'utilisation de tout type de données et offrent une abstraction de la communication lors de transmission distante.
  - A bas niveau les services sont beaucoup plus rapides, mais moins flexibles d'utilisation. Ces services sont par exemple TCP[23] et UDP.[24] Ils permettent de communiquer rapidement des données soit par flot de données, soit par trames sécurisées.

2.5.3.1 Moyens de communication utilisés par la cosimulation

Sur la plupart des systèmes d'exploitation, les moyens de communication sont les suivants (dans un ordre croissant de rapidité de transmission) [Cur91] :

- les RPC[25]
- les "*Sockets*" [26] domaine Internet
- les "*Sockets*" domaine local

Suivant les besoins, les "*sockets*" demeurent un mode de communication bas niveau qui existe sur tous les systèmes d'exploitation et qui procure ainsi une portabilité de la communication. Le protocole de communication d'une "*socket*" peut être choisi. Les "*sockets*" de domaine internet utilisent le protocole TCP/IP permettant ainsi la communication à travers le monde mais peuvent s'appliquer à d'autres types de protocoles. Les "*sockets*" de domaine local permettent quant à elles, une communication locale par un protocole fixé par le système d'exploitation. Elles sont plus rapides tout en fonctionnant de façon identique.

- les IPCs[27]
  Les IPCs sont un ensemble d'accès système utilisé pour gérer des messages entre processus, des mémoires partagées et des sémaphores. Pour communiquer entre processus il est

[22] CORBA : "Common Object Request Broker Architecture".

[23] TCP : (Tranfer Communication Protocol) protocole pour les transferts de communications.

[24] UDP : (User Data Protocol) : Protocole de transfert de données utilisateur.

[25] RPC : abréviation de Remote Procedure Call en anglais, permet de réaliser des appels de procédures distants. Ils concernent la couche réseaux "session" mais ont été conçus pour être rapides

[26] "Socket" : nom anglais de prise, est un support de communication de la couche "transport". C'est un point d'accès auxquels des connections peuvent être rattachées.

donc possible d'utiliser :
- les messages qui proposent un ensemble de primitives pour communiquer avec la synchronisation adaptée. Le système d'exploitation gère lui même le protocole de communication et garantit la distribution et le stockage des messages via une pile système.
- une mémoire partagée[28] avec un sémaphore de synchronisation garantissant les meilleures performances en terme de rapidité. Le mode de communication utilise un segment de mémoire qui est accessible en lecture/écriture par les processus connectés. Néanmoins, cette méthode de communication nécessite à l'utilisateur d'implémenter ses propres synchronisations en terme de lecture/écriture dans la mémoire partagée.

– Les primitives de haut niveau (e.g. JAVA). Ce sont des outils qui surchargent les communications de bas niveau en offrant des primitives très flexibles pour traiter des structures de données allant jusqu'au transport de classes d'objets. Ces primitives gèrent automatiquement la communication en utilisant des primitives de bas niveau de communication.

2.5.3.2 Mesures de performances des modes de communication

Un certain nombre de mesures ont été effectuées dans le but d'évaluer les performances en vitesse des méthodes de communication sur un système Unix. Les tests réalisés couvrent l'ensemble des méthodes de communication du système V.[29] Seuls les messages appartenant à la famille des IPCs n'ont pas été testés car leurs performances se situent entre celles de la mémoire partagée (avec sémaphore) et de la "*socket*" locale. Les figures 2.13a et 2.13b représentent respectivement les mesures de performances sur des échanges de 20 et de 160 caractères. En abscisse on représente le nombre de cycle d'horloge effectués par le processeur. En ordonnée le temps réel en seconde qu'il a nécessité pour réaliser l'ensemble des échanges. Ces mesures ont été effectuées sur 3 machines différentes (Sun Server Ultra-sparc 2*250 Mhz sous Solaris 7, Sun Ultra-sparc 10 sous Solaris 7, et Tadpole_S3GX sparc sous Solaris 5). De plus, les échanges sont réalisés dans trois cas : mémoire partagée + sémaphore, "*socket*" locale, et "*socket*" domaine internet.

Les Mesures effectuées ont permis certaines constatations sur les performances des différents supports de communication disponibles sous Unix :

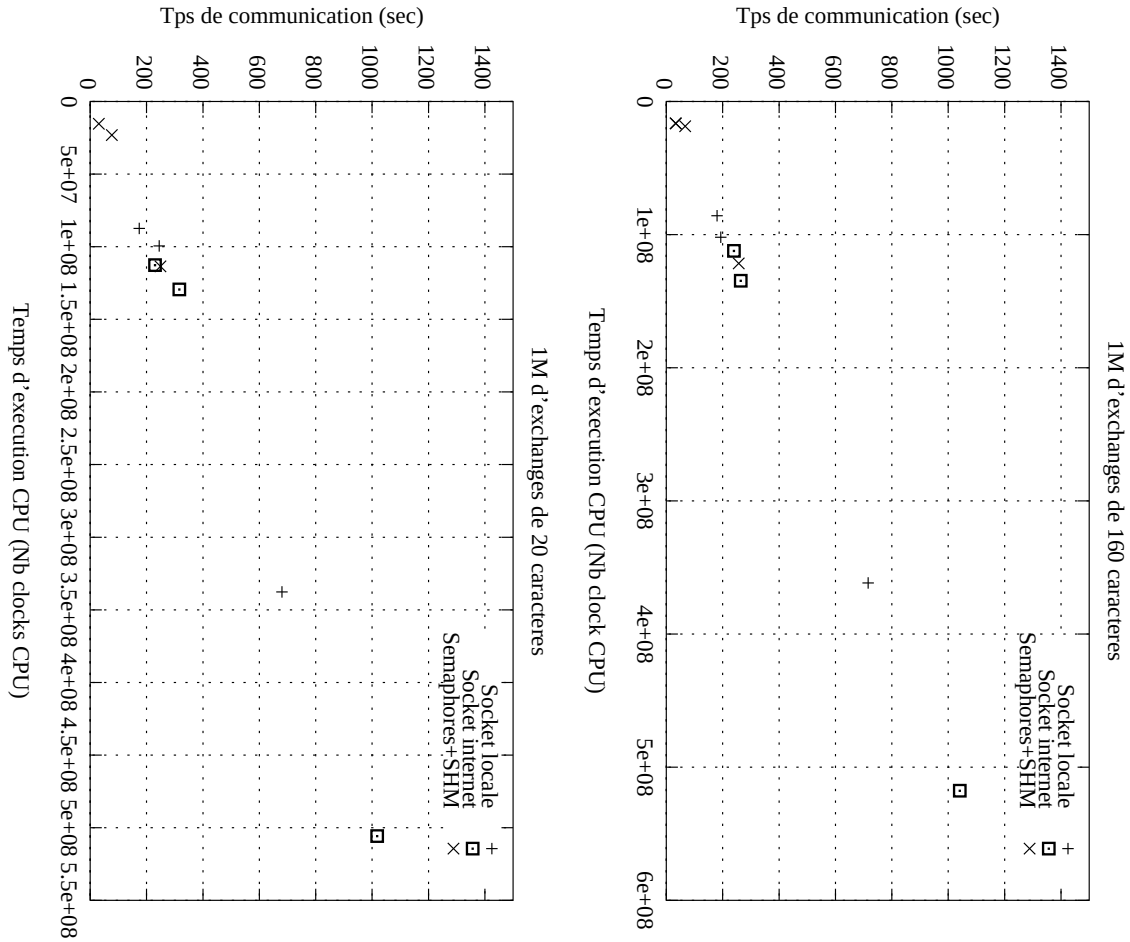
1. Le nombre d'octets échangés implique une très légère modification du temps de communication. On peut certainement dire qu'il y a une étroite relation avec la longueur des trames éthernet du système. Ce facteur, à priori linéaire, est valable quelque soit le support de communication. Celui-ci correspond en fait au temps de traitement CPU[30] des caractères. La différence de temps entre 20 et 160 caractères traités est la différence de temps qui correspond à la différence du nombre d'appel de routines systèmes multiplié par le temps de réaction du système à ces appels.
2. Les temps des appels systèmes diffèrent en fonction des supports de communication.

$$Tps_{(Socket\,Internet)} \gg Tps_{(Socket\,locale)} \geq Tps_{(Semaphore+Shm)}$$

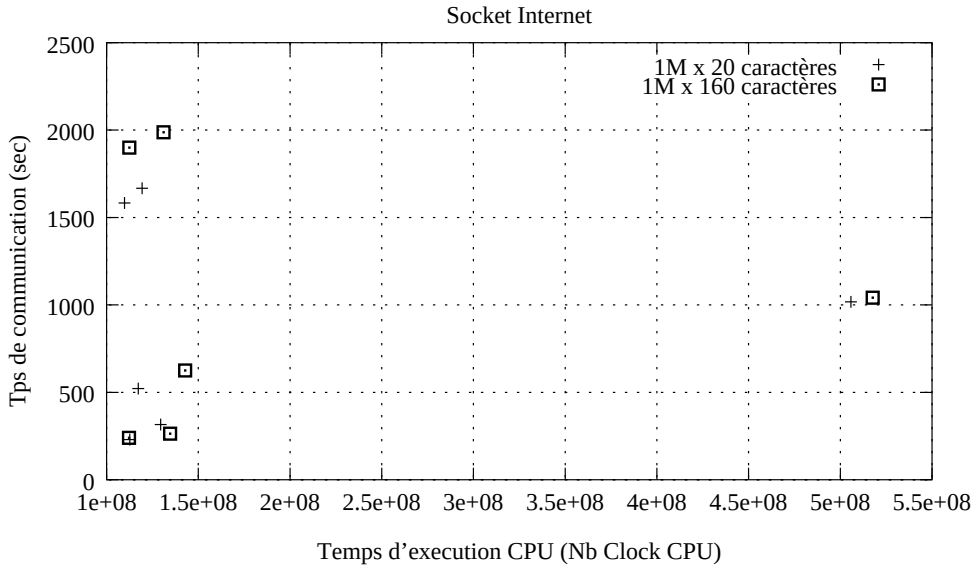
[27] Les IPCs ("Inter Process Communication") sont un moyen de communication standard sous Unix uniquement. Néanmoins des bibliothèques de développement utilisées pour les modéliser existent aussi sur d'autres systèmes d'exploitations.

[28] Une mémoire partagée est souvent abrégée par SHM "SHared Memory".

[29] le système V d'unix est le standard annoncé par AT&T depuis 1983



(a) : même machine



(b) : plusieurs machines

FIG. 2.13 – Mesures de performances de systèmes de communication

3. Les systèmes d'exploitation Solaris 2.5 et Solaris 2.7 sont différents. Solaris 2.7 a sensiblement diminué la différence de performance entre les "*sockets*" locales et les sémaphores+SHM. Ces différences sont dues aux optimisations apportées au système d'exploitation de Solaris. L'optimisation est de l'ordre de 20% de moins en temps d'appel système pour Solaris 2.7. De la même manière, cela diminue l'occupation du processeur.
4. Sur les "*sockets*" de domaine internet, le temps de communication dépend énormément des performances de la carte réseau et essentiellement du temps de réaction de cette carte.
5. La communication par jeu de sémaphore et d'une mémoire partagée est très intéressante car elle divise le temps CPU par 5 par rapport aux autres solutions. Elle diminue également en moyenne par 5.5 le temps global d'une communication par rapport à celle réalisée sur "*socket*" locale.

Néanmoins, ces mesures peuvent varier selon l'occupation du réseau pour la "*socket*" internet et de la charge CPU pour l'ensemble.

2.5.3.3 Choix du support de communication pour le bus de cosimulation

La cosimulation de modules différents entre eux ne présente à priori pas de comportement comparable à un modèle de communication client-serveur. En effet, un module, comme par exemple le logiciel, peut aussi bien être le contrôleur du circuit (rôle de maître ou serveur) qu'un co-processeur (rôle d'esclave ou client). La cosimulation requiert une communication équilibrée : les échanges sont bidirectionnels et initiés par l'une ou l'autre des parties. Pour la cosimulation, nous cherchons un modèle de communication pouvant fournir le plus large éventail de mécanismes possible tout en respectant le transport des données. Il faut pouvoir choisir un support de communication respectant les taux de transfert, la rapidité du transport, la flexibilité par la communication distante et le respect des modes de synchronisation. Pour certains modes de synchronisation, garantir la séquentialité des événements est nécessaire. Dans ces cas, les messages IPC ou une "*socket*" locale pourront être utilisés. On peut utiliser la mémoire partagée pour les cas où les données sont consommées en même temps qu'elles sont produites. Enfin, pour la communication distante on utilisera une "*socket*" pour communiquer entre les différentes machines utilisées dans l'environnement de cosimulation.

2.6 Outils de Cosimulation Existants

De nombreux outils de cosimulation matériel - logiciel existent, aussi bien sur le marché que dans le monde universitaire [BHL94][CT95][KL93][KKR94][BST92][GM92]. Les environnements complets de conception conjointe disposent également de moyens de cosimulation. Dans cette étude, nous nous intéressons aux domaines d'application de ces outils. Si la cosimulation au niveau cycle est proposée dans presque tous les environnements considérés, la cosimulation C-VHDL au niveau fonctionnelle est plus rare, et la cosimulation multilangage est au stade d'émergence. Les différents outils existants et les techniques de modélisation utilisées par la cosimulation sont présentés ci-dessous et le tableau 2.1de la page 68 les récapitule.

[30] le CPU ("Central Processing Unit") désigne l'unité centrale de calcul ou le processeur.

2.6.1 Outils Commerciaux

2.6.1.1 Seamless de Mentor Graphics

Seamless de Mentor Graphics [Gra96] fournit une interface de cosimulation pour connecter un simulateur de jeux d'instructions à un simulateur matériel. Les principales caractéristiques de cet outil sont une optimisation des échanges entre les deux parties selon la densité de la communication, ainsi qu'une interface de programmation standard. Cet outil autorise une certaine flexibilité dans la localisation de l'interface matériel-logiciel utilisée pendant la cosimulation (indépendamment de l'interface matériel-logiciel du système réel). L'outil Seamless permet de simuler la mémoire en logiciel, indépendamment de sa réalisation finale dans le système (nécessairement matérielle).

Seamless propose plusieurs niveaux d'optimisation, permettant d'optimiser soit tous les échanges entre les modèles simulés, soit seulement les échanges de données ou encore les échanges instructions (fetch) du processeur. En contrepartie de ces optimisations, on obtient une perte de précision dans la validation. Enfin, la vitesse maximale atteinte selon Mentor est de l'ordre de 100000 instructions par seconde. Cette vitesse est dépendante du mode de simulation utilisé et du mode d'optimisation des données. En simulation au niveau cycle la vitesse de simulation est rapportée à quelques dizaines d'instructions par seconde.

Une caractéristique intéressante de cet outil concerne la réalisation de la communication entre le simulateur matériel et le simulateur logiciel. Dans la configuration standard de l'outil, le simulateur VHDL et le simulateur de jeu d'instructions sont fournis par Mentor Graphics, ou par Cadence pour le simulateur Verilog.

En cas de besoin, un simulateur de jeu d'instructions client peut être intégré à l'outil. Pour cela, une interface de programmation standard (API) est disponible. Le travail de recyclage de l'interface et d'intégration est assuré par Mentor Graphics, car il requiert une expertise dans la programmation interne de l'outil.

Avantages et Inconvénients

L'outil Seamless autorise la cosimulation de plusieurs modules, chacun étant associé à une API propre. Les mémoires de chacun des processeurs sont simulées indépendamment. En conséquence, la cosimulation de plusieurs processeurs et d'une mémoire partagée ne peut se faire qu'en implémentant cette mémoire dans la partie matérielle. De ce fait, l'optimisation des accès proposée par Mentor est impossible dans le cas d'une architecture multi-processeurs avec une mémoire partagée.

De plus, la cosimulation avec le simulateur VHDL VSS de chez Synopsys, n'est pas envisageable.

Par ailleurs, Seamless propose une grande variété de modèles de processeurs.

2.6.1.2 EagleI

L'outil EagleI de Eagle Design Automation [Aut95][Aut96] propose un environnement de cosimulation matériel-logiciel multiniveaux, basé sur la technologie de processeurs logiciels virtuels VSP.[31] Le processeur virtuel est une boîte noire ayant la même interface matérielle que le processeur réel, connectée d'une part au logiciel et d'autre part au reste du système. La connexion avec le reste du système est assurée par un module d'interface matériel, décrivant

[31] VSP : Virtual Software Processor.

le comportement des signaux sur le bus avec un réalisme au niveau cycle. La connexion avec le logiciel est implémentée différemment selon la configuration choisie (C, simulateur logiciel ou émulateurs), tout en conservant la même interface. La possibilité du maintien du même environnement de cosimulation du système tout au long du cycle de conception, par un simple changement de configuration suivant les éléments disponibles, permet d'assurer une continuité dans le flot de validation (les mêmes stimuli de test peuvent êtres réutilisés).

La configuration de cosimulation C-VHDL, appelée VSP/Link, consiste à compiler le code C du logiciel sur la station de travail, puis à l'exécuter en parallèle avec la simulation VHDL du reste du système. La communication entre les deux parties est assurée par un ensemble de fonctions C de base (typiquement "`read_port`" et "`write_port`"), qui interagissent avec les ports matériels modélisés dans le module d'interface. Ainsi, la gestion des signaux d'interruption est assurée. Dans cette configuration, la mémoire du processeur est modélisée par le logiciel (mémoire de la station de travail) assurant une vitesse de cosimulation bien plus grande qu'avec un modèle matériel. Cette vitesse peut varier entre 5 Kips (milliers d'instructions par seconde) et 3 Mips (millions d'instructions par seconde). Le développement d'un modèle VSP/Link pour un nouveau processeur (fonctions de communication et modèle matériel de l'interface) est estimé entre 4 et 12 semaines.

La configuration de cosimulation matériel-VHDL, appelée VSP/Sim fait appel à un simulateur de jeu d'instructions qui exécute le code assembleur correspondant au logiciel et au système d'exploitation final. La communication est à nouveau assurée au niveau cycle avec le reste du système. La précision de simulation logicielle (réalisme temporel) est bien meilleure qu'avec le modèle VSP/Link, par contre la vitesse de simulation est nettement plus faible.

La dernière configuration possible (VSP/Tap) fait appel à un émulateur matériel, implémentant la fonctionnalité du processeur cible ("*In-Circuit Emulation*"). Cette étape finale consiste à valider encore plus finement le logiciel embarqué dans des conditions proches du temps réel. Elle intervient après la modélisation du processeur en VHDL, donc assez tard dans le flot. Elle autorise cependant une validation du système en temps-réel avant le prototypage matériel qui est bien plus coûteux.

Avantages et Inconvénients

L'outil EagleI autorise la simulation de plusieurs processeurs et comme Seamless, propose un grand nombre de modèles de processeurs. Toutefois, EagleI propose la connexion d'outils divers pour le développement de certaines parties du système comme par exemple des liens avec les outils de Cadence, IKOS, Mentor, et quickturn.

2.6.1.3 Cossap et Spw

L'environnement COSSAP [RC95], représenté sur la figure 2.14 part d'un modèle de description du système à haut niveau (saisi graphiquement par composition d'éléments de bibliothèques) et simulable à haut niveau. Le partitionnement, spécifié par l'utilisateur, produit un modèle VHDL pour la partie matérielle (comportemental ou RTL synthétisable), et un modèle logiciel en C ou en assembleur. Le modèle C peut être standard, ou bien optimisé pour un processeur donné s'il est disponible en bibliothèque. L'utilisateur peut par ailleurs introduire son propre code C ou assembleur dans des blocs COSSAP. Le modèle C du logiciel est rarement utilisé pour produire le code assembleur final. Cela est essentiellement dû à l'inexistence de compilateurs C performants dans le domaine du traitement du signal. En pratique, le modèle

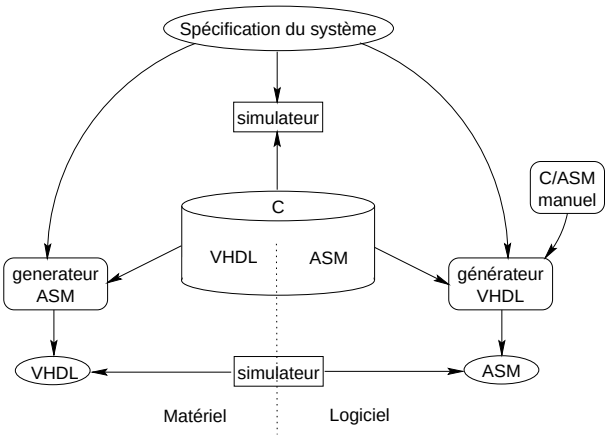


FIG. 2.14 – Flot de conception dans l'environnement COSSAP

C est uniquement utilisé pour la validation de la fonctionnalité à haut-niveau (enchaînement des traitements, modèles de données, etc...) et du partitionnement matériel/logiciel. Ce modèle sert de référence pour choisir et ordonner les composants de base de la bibliothèque. Alors, les modèles C bas-niveau ou assembleur de ces composants sont effectivement utilisés pour la réalisation finale. Les modèles de la bibliothèque sont écrits à la main, et fournis par Synopsys. Après compilation éventuelle des blocs écrits en C, nous obtenons un code assembleur unique pour l'application. C'est alors que ce code assembleur est introduit dans un simulateur de jeux d'instructions, pour être validé dans un environnement matériel.

Une telle approche n'est pas adaptée à un flot de conception continue du logiciel, depuis le modèle en C haut-niveau jusqu'au code assembleur. La validation effective du logiciel réellement implémenté intervient au niveau assembleur et nécessite un simulateur de jeux d'instructions dédié. Mais l'obstacle majeur de cette méthodologie pour la conception et le développement de processeurs dédiés, réside dans le fait que le code (C bas-niveau ou assembleur) utilisé pour la réalisation doit être disponible sous forme de fonctions de base, dans une bibliothèque. De ce fait, toute exploration d'architecture est évidemment impossible (il faudrait générer de nouveau la bibliothèque complète) et pénalise le développement et la validation rapide de systèmes dédiés.

Alta Group de Cadence Design Systems offre un environnement de conception et de validation d'un système à base de processeur de traitement du signal, SPW [Gro96b][Gro96a]. La méthodologie est très similaire à celle de Cossap de Synopsys. La validation du logiciel final est effectuée grâce à la simulation du code assembleur par un simulateur de jeu d'instructions, lequel est relié à l'environnement matériel.

2.6.1.4 CoWare

CoWare, développé initialement à l'IMEC (Leuven, Belgique) puis industrialisé par CoWare Inc., est un environnement de co-conception matériel-logiciel à haut niveau [CoW96][RVBM96]. Utilisant les langages de programmation et de description classiques (C, C++, VHDL), il est ouvert aux outils de développement standards.

Flot de conception

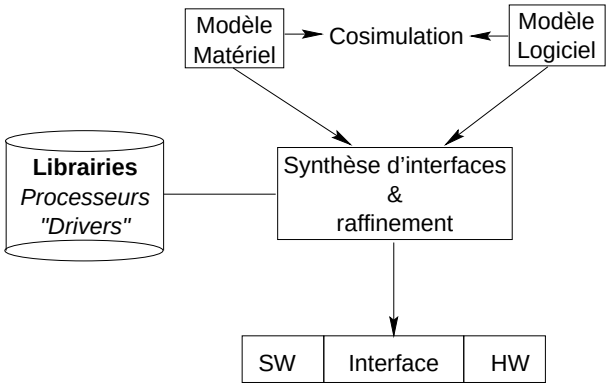


FIG. 2.15 – CoWare : Décomposition

Le système est décrit sous forme de "*thread*",[32] c'est à dire un ensemble de blocs de code écrits en C, C++, SystemC[33] ou VHDL, pouvant éventuellement être exécutés en parallèle. Les interactions entre les blocs (passage de valeurs) ne peuvent avoir lieu qu'au début ou à la fin d'un "*thread*" (comme une fonction). Lorsque ce partitionnement est validé, la partie logicielle est optimisée afin de minimiser le nombre de "*threads*". Pour des applications mono-flot, nous obtenons en fait un programme séquentiel unique en C ou C++. Celui-ci peut alors être compilé (si un compilateur est disponible) ou servir de référence pour l'écriture d'un code assembleur. Dans le cas d'applications de contrôle complexes, un noyau temps réel peut être invoqué (s'il est disponible en bibliothèque). Pour le choix du processeur embarqué sur lequel doit s'exécuter le logiciel, deux approches sont possibles : soit un cIJoeur de processeur standard est utilisé, soit un processeur dédié est conçu. L'utilisateur fait alors appel à des primitives de communication existantes, spécifiques à l'interface.

Validation par la cosimulation

Le point de départ de CoWare est un système mixte C-VHDL organisé sous forme de "*threads*" parallèles. Ce modèle peut alors être cosimulé. Le mécanisme utilisé est l'appel de procédures à distance (RPC), bien adapté au modèle de "*threads*". Le modèle C utilisé pour la partie logicielle n'est cependant pas exactement celui qui sera implémenté, en raison essentiellement de l'existence de "*threads*". La phase de minimisation du nombre de "*threads*" introduit une modification sensible du code C, lequel sera ensuite compilé.

Après compilation du logiciel, il est possible d'effectuer une cosimulation au niveau assembleur. En effet, l'environnement CoWare rend possible la connexion d'un simulateur de jeu d'instructions au simulateur VHDL. Quelques procédures de base du simulateur de jeu d'instructions (initialisation, simulation d'un seul cycle) doivent être accessibles par l'environnement.

Modélisation de l'interface

L'interface entre le matériel et le logiciel (i.e. l'interface du processeur embarqué) est directement disponible si le processeur est en bibliothèque. Sinon, l'interface peut être générée. En fait, les interfaces (standards ou nouvelles) sont décrites suivant plusieurs niveaux d'abstraction :

[32] Les "threads" sont des processus de programmes parallèles partageant le même espace de données

[33] SystemC est une bibliothèque de classes pouvant modéliser une partie matérielle en C++. SystemC est actuellement du domaine public.

- à haut niveau, les données sont définies par des types du langage C,
- ensuite, une taille est associée à chaque donnée,
- enfin, un ensemble de ports est alloué pour transmettre la donnée (il peut y avoir un vecteur de bits pour la donnée elle-même et un ou plusieurs bits d'acquittement, associés à des délais de temporisation).

Avantages et Inconvénients

L'environnement CoWare est particulièrement adapté au développement rapide de systèmes embarqués. La disponibilité de coeIJurs de processeurs et des interfaces correspondantes, aussi bien en VHDL qu'en C, permet d'obtenir facilement un prototype réaliste. Les possibilités de cosimulation, aussi bien au niveau C qu'au niveau assembleur, autorisent une validation continue tout au long de la conception, et ce dans un environnement unifié. La validation fonctionnelle de l'application complète et la cosimulation C-VHDL intervenant avant le choix du processeur et de l'interface, peuvent être effectuées très tôt dans le flot de conception. L'utilisation des outils de développement standards aussi bien en VHDL qu'en C avantage la conception rapide et le déboggage du système.

Le niveau d'abstraction élevé de la description de l'interface peut être un atout pour l'écriture de protocoles complexes. De plus, il facilite l'interaction entre les modules décrits à différents niveaux d'abstraction.

L'utilisation d'un modèle évolué de représentation du système à haut niveau (multi-"*threads*") peut s'avérer être un choix pertinent pour le développement de systèmes à contrôle relativement complexe, nécessitant un noyau multi-tâches en temps réel.

Cependant, la traduction du modèle logiciel original (contenant beaucoup de "*threads*") en un programme C compilable introduit une modification sensible de celui-ci, nécessitant une validation ultérieure.

Enfin l'environnement de CoWare n'accepte pas l'instanciation de plusieurs processeurs .

2.6.1.5 Synthesia

Synthesia [Alt95a][Alt95b][AG94] propose un environnement de co-vérification matériel-logiciel destiné à la conception conjointe d'un système, décrit en VHDL pour la partie matérielle, et en C (ou C++ et Ada) pour la partie logicielle. Cette co-vérification intervient après le partitionnement du système, dans le but de valider celui-ci avant la réalisation finale. L'ensemble du matériel est simulé sur un simulateur VHDL unique, alors que le ou les logiciels sont directement exécutés sur la station de travail. La communication entre le simulateur et les programmes est assurée par un mécanisme de communication RPC. L'interface entre le matériel et le logiciel est constituée d'un ensemble de données, de types simples et standards (bit, octets, mots) ou complexes (structures, types énumérés, unions). Les simulateurs actuellement supportés sont Leapfrog de Cadence et VHDL Testbench. L'approche de Synthésia est parfaitement adaptée à la conception d'applications logicielles en C embarquées, autorisant l'utilisation d'outils standards de développement en C.

Avantages et Inconvénients

Nous pouvons identifier deux inconvénients à cette approche. En premier lieu, une mauvaise gestion de la communication peut entraîner des erreurs liées à la cosimulation et non pas à l'application de l'utilisateur. En second lieu, le code applicatif est notablement différent de celui qui sera implémenté.

Concernant le réalisme temporel de la co-vérification avec Synthesia, nous notons l'absence d'un modèle d'annotation du programme C, permettant de simuler les délais de traitement de chaque instruction assembleur. Il serait à prévoir en fonctionnement réel. Un tel modèle permettrait de retarder la simulation VHDL pour certaines parties de l'application, et donc d'approcher un réalisme en terme de cycles d'instructions.

2.6.1.6 CORBA et Plug&Sim

CORBA est une partie du projet OMG (Object Management Architecture). Il offre une communication flexible et des méthodes adaptées pour réaliser des environnements distribués hétérogènes d'applications orientés objets. Pour inclure une application, le concepteur doit utiliser la bibliothèque d'objets OMG IDL[34] pour définir les types d'interfaces et les méthodes utilisées pour chaque application. L'implémentation de ces méthodes est séparée de l'architecture de l'application et hérite des possibilités de déclaration, de manipulation et de communication d'objets de CORBA.

Plug&Sim d'Integrated Systems Inc propose un environnement de cosimulation très flexible et ouvert où deux applications pouvant être cosimulées. Cette plate-forme est basée sur CORBA et permet une exécution parallèle de la simulation sur une même machine ou à travers un réseau. Les types de données ne sont pas limités mais la connexion ne peut se faire sur plus de deux simulateurs.

2.6.2 Outils de Recherche

2.6.2.1 Ptolemy

L'environnement de co-conception Ptolemy, de l'université de Berkeley, permet le développement d'applications de traitement du signal et de systèmes communicants [KL93][BHL94]. Il utilise un modèle de systèmes orientés objets. L'environnement supporte différents modèles de conception encapsulés dans des objets appelés "*domains*".[35] Un domaine réalise un modèle de calcul d'un sous-ensemble modélisé par une hiérarchie d'objets ("*Blocks*", "*Galaxy*", "*Stars*", ...). Dans le cadre de la cosimulation, l'environnement exige une description complète du système. Le travail décrit dans [KKS+96] étend l'environnement de Ptolemy pour la cosimulation. L'outil fournit la génération automatique de l'interface entre un noyau logiciel et un noyau matériel. Il exige, comme description d'entrée, un graphe de flux de contrôle/données du système.

2.6.2.2 Cosyma

Cosyma décrit dans [OBE+97], se sert d'un arrangement fixe de communication. Il fournit un simulateur du processeur pour l'analyse de l'exécution et la vérification [EHB+95]. La description de l'entrée de Cosyma est un modèle textuel en langage C^X. Ce langage fait parti des nombreux HDL[36] existants. Il est basé sur le langage C et permet la description d'applications en modélisant les délais entre opérations, ainsi que les débits d'entrée ou de sortie du système.

[34] IDL : "Interface Definition Language" ou langage de définition des interfaces.
[35] domaines.
[36] Langage de description de haut niveau ("High Description Language")

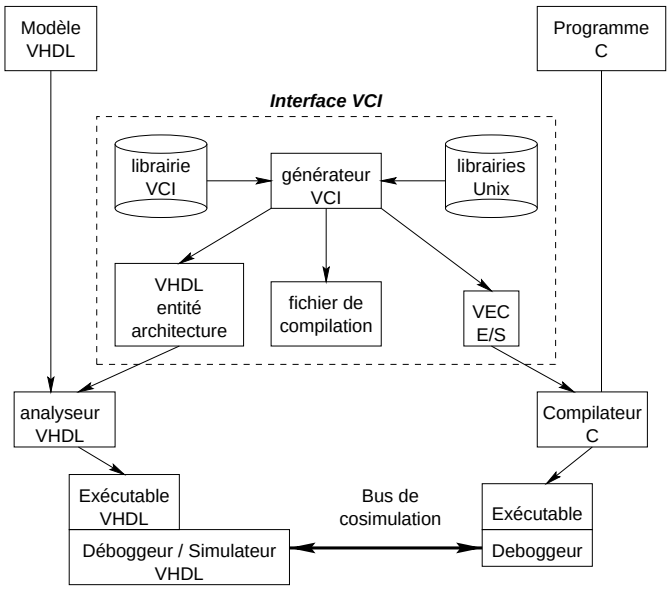


FIG. 2.16 – Interface de cosimulation C-VHDL (VCI)

La stratégie, adaptée au partitionnement matériel / logiciel, commence à partir d'une solution purement logicielle. Les parties critiques du logiciel sont portées vers la partie matérielle. Les parties logicielles et matérielles sont à terme traduites respectivement en langage C et HardwareC [KM88]. L'architecture cible utilise uniquement un processeur doté d'un circuit intégré (ASIC), d'une mémoire et d'un bus. En utilisant le processeur et les modèles inférieurs de communication, les environnements fournissent une simulation de grain fin, au dépend du temps et de la flexibilité de la simulation. Enfin, certains travaux ont permis l'extension de Cosyma pour l'estimation de coût [HHE94].

2.6.2.3 VCI (VHDL-C Interface)

VCI [VNPJ96](figure 2.16), est un système de simulation C-VHDL distribué. Il utilise un modèle plus flexible que dans [CT95] et [KKS+96]. Il a la particularité de générer les interfaces de communication entre les modèles VHDL et C, qui permettront d'établir la communication entre ces modèles lors de la cosimulation.

Le lien de cosimulation créé entre les outils de mise au point logiciels et matériels repose sur l'interface d'E/S existante entre les modules. Ainsi, l'outil n'a pas besoin d'une description complète du système pour produire les liens de simulation entre les modèles. En conséquence, le module logiciel peut être utilisé comme tout autre composant matériel. Ceci permet de manipuler toutes sortes d'architectures distribuées. En ce qui concerne la partie logicielle, le lien est réalisé par des primitives d'E/S simples. La communication peut donc être modélisée dans plusieurs couches et ainsi, on peut réaliser une cosimulation à différents niveaux d'abstraction. L'exécution des modules logiciels peut être faite dans un ou plusieurs processeurs. Une autre flexibilité de cet outil est sa capacité à spécifier le mode de synchronisation entre les simulateurs. Ceci permet d'ajuster l'efficacité de la simulation sans qu'il soit nécessaire de changer l'interface. VCI a aussi servi de base à MCI (cf. §3) pour l'extension de son modèle de cosimulation distribuée vers le multilangage et la simulation de modules distants

2.6.2.4 Cass

Le simulateur Cass [PHG97b] est un simulateur précis au niveau cycle dans la prédiction d'événements. Il utilise la propriété d'absence de cycles dans les communications combinatoires inter-processus (CIC) [PHG97a]. Son système de simulation est fondé sur les propriétés d'échanges des systèmes simulés. Un composant doit être capable de communiquer avec plusieurs autres composants, et la réutilisation d'un des composants n'est possible que si le concepteur utilise une interface standard de communication. Le modèle de communication doit avoir une très haute fréquence et une latence très faible. Le principe est fondé sur ces propriétés et du fait qu'un système est décrit au mieux à travers une collection de processus séquentiels communicants [GVNG94][Hoa85]. La partie matérielle est modélisée en un jeu de machines d'états finis.

Si le système ne contient pas de boucles dans les CIC, il est possible alors d'établir un ré-ordonnancement statique des échanges de données, à la compilation. La condition est que chaque processus du système soit évalué une et une seule fois par cycle. Le simulateur devient une boucle dans laquelle chaque processus est appelé une fois. Les liens de communication inter-processus sont décrits comme une liste d'interconnexions de processus entre eux où les échanges sont réalisés à la fin de chaque cycle.

Avantages et Inconvénients

Ce type de moteur de simulation atteint des performances de l'ordre de 150 Kcycles par seconde. L'inconvénient est que les modèles du système doivent être écrits avec précision afin d'optimiser la vitesse de simulation. De plus, la synthèse n'est pas directement possible à partir de ces modèles, mais il est toutefois intéressant pour la validation d'une architecture logicielle/matérielle.

2.6.2.5 RAPID

RAPID [RW93] est un outil de développement de prototype pour la conception des systèmes embarqués de la conception au développement. Il permet notamment de créer des simulations fonctionnelles d'un système embarqué. Le modèle du système à simuler comprend l'ensemble des conceptions du système et la fonctionnalité d'un prototype physique. Il fournit également un générateur de code qui permet d'obtenir un prototype basé sur les StateCharts et sur la méthodologie objets. Des objets visuels, comme un clavier, peuvent être ajoutés au modèle à simuler. Ces objets simulent le comportement des composants externes. Cette méthode est une méthode compositionnelle. Le modèle est généré, puis compilé pour obtenir un exécutable de simulation qui peut être exécuté.

2.6.2.6 ADL

Les ADLs (Architecture Description Languages) [ISZ98] sont dédiés à la définition d'architecture. Le concept est basé sur les notions de composants, de connecteurs et de configuration. Les composants sont, soit des primitives pour l'encapsulation du logiciel, soit des composites pour structurer l'application. Les modes de communication et l'implémentation des modèles des composants sont définis avec les interfaces. Un connecteur spécifie les contraintes sur les

composants connectés en terme d'interfaces et de compatibilité d'opérations. Ce langage est en phase de standardisation et apportera la possibilité aux concepteurs de combiner des parties logicielles et matérielles pour former une application complète.

2.7 Conclusion

Les méthodes de validation par cosimulation peuvent être utilisées pour valider les différents modèles d'un système durant tout le flot de conception. Elles permettent la manipulation de toutes sortes d'architectures distribuées sans avoir à se soucier du mécanisme de communication utilisé. La plupart des approches permettent l'utilisation d'outils de mise au point et de déboggage pouvant être utilisés conjointement. Aussi, plusieurs modes de synchronisation peuvent être utilisés pour supporter différents scénarios de simulation. Malgré tout, les différents environnements apparus avec les outils de co-conception sont bien souvent dédiés aux conceptions mixtes logiciel/matériel et seulement très peu de travaux proposent de réaliser une cosimulation au niveau système. Les environnements existants ne sont pas suffisamment génériques pour accepter la cosimulation de systèmes hétérogènes multilangages. Certaines adaptations ont été tentées afin d'implémenter de nouveaux langages sur certains environnements, mais l'investissement de travail reste à chaque fois conséquent lors de l'intégration d'un nouveau langage. Les nouvelles méthodologies de conception des systèmes hétérogènes nécessitent désormais des environnements de cosimulation de plus haut niveau. De plus, il est nécessaire de pouvoir simuler conjointement des parties d'un système de haut niveau avec des parties de plus bas niveau ou encore avec un modèle représentant un environnement extérieur. Les nouvelles méthodologies associées à la conception des systèmes hétérogènes multilangages impliquent une plus grande modularité des environnements de cosimulation. Cette modularité doit exister aussi bien dans les niveaux d'abstraction que dans l'utilisation de différents langages. La cosimulation multilangage et multiniveaux devient l'enjeu des nouvelles méthodologies de conception.

	Propriétaire	Entrée	Utilisation	Type	Proc	Comm
Seamless	Mentor Graphics	VHDL, Verilog, Assembleur	niveau instructions et cycle	D	MP	SHM
EagleI	Synopsys	VHDL, Assembleur	niveau instruction et cycle	D	MP	
Plug N Sim	Integrated Systems	Statemate, MatrixX, simulateurs compatibles CORBA	niveau système	D		CORBA
RAPID	Univ. Standford	StateCharts, objets	niveau système	C		
Cosyma	Univ. Braunschweig	C^X	fonctionnelle, Proc-ASIC	D		
COSSAP	Synopsys	Modules, C, VHDL, assembleur	fonctionnelle, Procs-VHDL, DSP	D		IPC
SPW	Cadence	Modules, C, VHDL, assembleur	fonctionnelle, Procs-VHDL, DSP	D		
CoWare	CoWare	C, C++, VHDL	fonctionnelle, ISS-VHDL-IP	D		RPC
Synthesia	Synthesia	C, C++, Ada, VHDL	VHDL-HCE	D		RPC
Ptolemy	Univ. Berkeley	Blocs interconnectés multilangages	fonctionnelle	D		
VCI	Lab. TIMA	VHDL, C	fonctionnelle	D		IPC
Cass	Lab. LIP6	Processeur - VHDL(FSM)	niveau cycle	D	SP	
ADL	Lab. INRIA	connector, interfaces et configuration	modélisation	C		RPC (JAVA), PIPE
CORBA	OMG	Applications	modélisation au niveau système	D		TCP/IP, SHM

C : cosimulation par une méthode compositionnelle

D : cosimulation distribuée

SP : Mono-processeur au niveau ciblage

MP : Multi-processeurs au niveau ciblage

HCE : Code exécuté directement sur la machine

TAB. 2.1 – Récapitulatif d'outils de cosimulation

Chapitre 3

MCI : UN OUTIL POUR LA COSIMULATION MULTILANGAGE ET MULTINIVEAUX

L'outil MCI ("*Multilanguage Cosimulation Interface*") [HLJ98] est un outil de validation à différents niveaux d'abstraction par cosimulation dédiée aux applications multilangages décrites dans plusieurs langages. Il est extensible à d'autres langages car il permet l'ajout de simulateurs à son environnement. Ce chapitre présente l'outil MCI dans ses détails, à commencer par ses domaines d'applications, ses objectifs et son principe de fonctionnement. Ensuite, l'interfaçage et les diverses méthodes de synchronisation sont introduits en fonction des niveaux d'abstraction.

3.1 Domaines d'Applications de MCI

L'outil de cosimulation multilangage MCI cible un grand nombre de domaines d'applications allant de l'automobile aux télécommunications. Les travaux liés à la cosimulation multilangage ont démarré dans le cadre d'un projet de coopération entre le laboratoire TIMA et le centre de recherche de PSA.[1] MCI a été par la suite appliqué à d'autres domaines. C'est le cas par exemple de la simulation d'un modem VDSL sur réseau CAD_{NET}, projet réalisé avec le CNET et ST-microelectronics.

3.1.1 Systèmes Mécatroniques

Les premiers résultats de cet outil ont été obtenus sur des applications diverses du monde automobile. Les trois applications suivantes ont été traitées :

– les premiers essais des versions initiales de MCI ont été réalisés sur une application qui permet d'éliminer les vibrations parasites d'une pédale d'accélération (cf. application §4.2). L'application est un sous-système du projet "Tulipe", ou voiture urbaine électrique. Il consiste à réguler la variation de position de la pédale d'accélération du véhicule avec son moteur. En effet, étant donné que le véhicule est électrique, la dynamique de son moteur est importante. De ce fait, il est essentiel de supprimer les vibrations intempestives

[1] Cette collaboration a eu lieu entre le laboratoire TIMA et la Direction des Recherches et Affaires Scientifiques de PSA (Peugeot et Citroën à Vélizy-Villacoublay).

du pied du conducteur dans la consigne de commande de l'accélération véhicule. L'ensemble du projet fut modélisé en trois langages : C, VHDL et MATLAB. La cosimulation multilangage a donc consisté à simuler de façon conjointe les parties matérielle, logicielle et mécanique à différents niveaux d'abstraction. A bas niveau, ces travaux nous ont permis d'utiliser un modèle VHDL du processeur dans le but de réaliser la cosimulation du système au niveau cycle avant son implémentation réelle.

– La seconde application avait pour objectif de modéliser la suspension hydractive d'un véhicule. La partie véhicule et sa suspension ont été spécifiées en MATLAB/SIMULINK,[2] tandis que la partie électronique de contrôle a été modélisée pour être exécutée sur un microcontrôleur. La fonction de la partie électronique est de contrôler les flux hydrauliques à travers les suspensions, afin d'obtenir l'assiette véhicule la plus plane possible, quelque soit le comportement du conducteur.
– La troisième application fut la simulation conjointe entre un simulateur MATLAB et un simulateur SABER.[3] La simulation de SABER est beaucoup plus fine que MATLAB en granularité, mais l'affichage des résultats n'est possible qu'à la fin de la simulation. Notre cosimulation a donc permis d'afficher les résultats de SABER dans MATLAB et de paramètrer SABER au cours de la simulation.

3.1.2 Systèmes de Télécommunications

Depuis plus d'un an, une étroite collaboration entre le CNET et le laboratoire TIMA a permis de promouvoir le réseau haut débit CAD_{NET} ainsi que de valider l'outil. Ainsi, plusieurs applications multilangages ont pu être réalisées sur ce réseau. Actuellement, la validation d'un protocole VDSL[4] est en cours avec la collaboration d'une équipe de ST-Microelectronics. L'intérêt d'un outil de validation multilangage pour des applications de télécommunication est de pouvoir modéliser des systèmes de transmission de façon modulaire. Différentes parties de ce système ont été réalisées en C, VHDL, COSSAP et SDL.

MCI propose la validation complète d'un tel système puisqu'il est ouvert à tous ces types de langages et devrait même évoluer en intégrant des accélérateurs de simulation tel que Voyager de IKOS.[5] IKOS permettrait une simulation plus rapide au niveau portes des parties matérielles, et offrirait donc la possibilité de simuler des systèmes de plus grande envergure.

3.1.3 MCI : Outil Evolutif

MCI a été avant tout conçu pour permettre la mise en oeuvre plus rapide et plus aisée des applications multilangages. C'est pourquoi l'utilisateur doit avoir la possibilité d'ajouter aisément un nouvel outil dans son environnement. Ainsi, grâce à MCI, l'intégration d'un nouveau simulateur est relativement simple à condition que cet outil contienne une API minimale de communication extérieure. Etant donné que la plupart des outils commerciaux proposent cette option et s'orientent dans ce sens, MCI devient ainsi très évolutif.

Aujourd'hui, l'outil permet de cosimuler des modules décrits dans les langages tels que :

[2] Simulink est l'interface graphique de Matlab.

[3] Saber est un logiciel de modélisation et de simulation des systèmes electro-magnétique, hydraulique, etc.. Il contient un moteur de simulation dédié, extrèmement précis.

[4] Le VDSL est le nouveau modem haut débit.

[5] Voyager de IKOS est un accélérateur de simulation VHDL au niveau porte.

- COSSAP de Synopsys,
- SDL d'Object Geode,
- les langages C/C++, ainsi que les modèles de simulation de jeux d'instruction décrits en C (Intel 8051, ST10 de ST-Microelectronics, C167 de Siemens,[6] Motorola 68000),
- SABER,
- MATLAB,
- VHDL (Leapfrog de Cadence ou VSS de Synopsys).

Cette liste a été construite au fur et à mesure des besoins des utilisateurs. Plusieurs autres langages peuvent être ajoutés, et ce sans grande difficulté. C'est la raison pour laquelle cette liste n'est pas exhaustive et tend à s'accroître selon la demande des utilisateurs.

3.2 Objectifs et Caractéristiques de l'outil MCI

L'objectif de MCI est de proposer un environnement de cosimulation permettant la simulation distribuée d'applications définies en plusieurs langages. Il permet d'accompagner le concepteur dans la validation du système à plusieurs niveaux d'abstraction, allant du niveau système au niveau le plus proche du prototypage. MCI doit être simple d'emploi et surtout suffisamment générique pour pouvoir facilement intégrer de nouveaux simulateurs et adapter de nouvelles méthodes de communication pour le transport des données.

3.2.1 Vérification par Simulation d'un Système Complet

Actuellement, de plus en plus de systèmes sont conçus avec des outils de très haut niveau. Le schéma de la figure 3.1 représente le principe de conception d'un système hétérogène. Le système débute souvent par une description de très haut niveau (SDL, StateCharts,..) pour être décomposé suivant sa fonctionnalité globale. Il est donc déjà intéressant d'user d'une simulation de haut niveau afin de vérifier la conformité de la première réalisation par rapport à la spécification initiale.

Les parties de l'environnement du système, si elles existent, sont séparées de la spécification pour être partitionnées et modélisées. Dans ce cas, une première cosimulation est très utile afin de vérifier que la modélisation de la partie électronique est satisfaisante vis à vis de l'environnement modélisé par des langages comme MATLAB, SABER, etc...

Dans une seconde étape, la spécification de cette partie électronique pourra être raffinée par des outils de co-conception. Dès lors, de nombreux cas de figures de cosimulation sont possibles. La validation par cosimulation peut consister à valider la partie électronique, non raffinée et de haut niveau, avec le reste du système et son environnement. Elle peut également consister en une validation de la sortie de l'outil de co-conception, voire la validation d'une partie du système.

La cosimulation multilangage joue un rôle important dans la modélisation des systèmes hétérogènes. Elle offre la possibilité au concepteur de vérifier plusieurs configurations possibles. MCI offre la possibilité aux concepteurs de valider leur système par cosimulation de sous-systèmes décrits dans des langages différents. Il permet ainsi de s'ouvrir à la modélisation multilangage et à la conception concurrente.

[6] Le processeur C167 de Siemens est iddentique au ST10 de ST-Microelectronics.

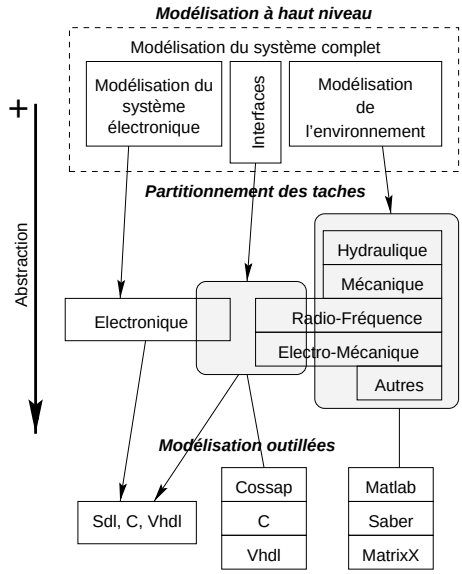


FIG. 3.1 – Schéma de conception de systèmes hétérogènes

3.2.2 Conception Concurrente

L'intérêt d'un outil de cosimulation réside dans la possibilité de vérifier un système complet par simulation conjointe de ses composantes. Il facilite le travail concurrent des différentes équipes de modélisation sollicitées. En effet, la plupart du temps, les différents sous-systèmes n'ont pas les mêmes délais de conception. L'outil MCI offre cette possibilité de modélisation concurrente en introduisant l'approche de coordination. Les différentes parties du modèle sont perçues par MCI comme des instances, et chacune d'elles représente la modélisation d'une partie décrite dans un langage particulier. Cette modélisation peut être confiée à différents groupes de conception spécifiques. Chaque groupe a donc pour objectif de spécifier et de valider localement son sous-système conformément aux techniques de la conception multilangage. Un fichier de coordination doit décrire pour la simulation les liens existants entre les différents sous-systèmes. Grâce à ce fichier de coordination, MCI offre la possibilité aux différents sous-système modélisés d'être simulés de façon conjointe avec leurs simulateurs respectifs. La plateforme de cosimulation de MCI permet de gérer les échanges entre les différents sous-systèmes et donnent une vision globale du comportement du système complet simulé. Le système peut être conçu de façon équilibrée et certains sous-systèmes peuvent être validés pendant que d'autres restent en phase de modélisation. Ainsi l'outil permet d'éliminer certaines pertes de temps dans la validation dûes à l'attente de l'achèvement d'un sous-système pour valider le système complet. Le système sera validé plus tôt et le temps de mise sur le marché sera réduit de façon significative.

Néanmoins, les liens entre les sous-systèmes, en plus d'être spécifiés dans le fichier de coordination comme une liste d'interconnexions, doivent aussi être annotés dans les différents modèles à simuler. Aussi, il est nécessaire de fournir un moyen automatique de générer l'environnement de cosimulation.

3.2.3 Génération Automatique de l'Environnement de Cosimulation

Afin de réaliser les interconnexions des différents sous-systèmes qui doivent être simulés avec l'environnement de cosimulation, l'outil MCI doit apporter :

– soit des annotations de code (graphiques ou textuelles),
– soit des instantiations de composants dans les modèles qui doivent être simulés.

MCI propose un ensemble de bibliothèques simples de primitives, de composants et de modules graphiques qui permettent d'avoir une génération automatique des annotations à produire pour chaque langage. L'outil contient donc un ensemble de bibliothèques pour chaque langage (VHDL, C, MATLAB, COSSAP, etc...). Ces bibliothèques simples introduisent le concept de ports de cosimulation dans les architectures des modèles permettant ainsi la communication avec l'environnement de cosimulation de MCI.

Ainsi, un outil de co-conception lié à ces bibliothèques est capable de générer de façon automatique les fichiers impliqués dans la cosimulation.

3.2.4 Utilisation des Outils de Simulation Existants

MCI est un outil basé sur le concept de la simulation distribuée (cf. §2.5.2). De ce fait, l'ensemble des outils de simulation utilisé dans l'environnement sont des outils de simulation déjà existants. L'intérêt est de constamment respecter les aspects du multilangage où l'utilisateur possède la plus grande liberté possible d'utilisation des outils qui lui sont familiers.

L'utilisation d'outils existants par MCI apporte plusieurs avantages :

– L'environnement de cosimulation bénéficie des propriétés de simulation des différents outils, de leur interface homme-machine, de leurs possibilités de mise au point et ainsi l'environnement s'affranchit des complexités de simulation des différents sous-systèmes.
– Cette utilisation permet aux concepteurs de conserver leur environnement de simulation et évite ainsi de perturber leurs habitudes de travail.
– Elle favorise la compatibilité avec la chaîne d'outils disponibles autour de l'outil de simulation et du modèle simulé. Par exemple, à partir d'un modèle MATLAB il est possible de générer un modèle C pour DSP.
– Elle peut apporter une simulation hiérarchique à l'intérieur d'un même simulateur. Certains simulateurs possèdent des liens avec d'autres simulateurs (ex : MatrixX avec StateMate via CORBA), ou peuvent eux-aussi encapsuler des sous-systèmes de langages différents sous forme d'appel de fonctions externes (ex : appel de fonctions C dans SDL, Fortran ou C dans MATLAB, VHDL dans COSSAP, etc..).

Néanmoins, il impose à MCI de respecter la convivialité des outils de simulation durant les phases de cosimulation. Ainsi, il convient d'éviter les blocages des moteurs de simulation, de l'interface graphique utilisateur et de maintenir la vitesse de simulation élevée. Enfin, pour augmenter la flexibilité de l'outil, MCI doit pouvoir accepter un maximum d'outils en entrée, et pour cela doit posséder un moyen simple d'intégrer un nouveau simulateur à son environnement.

3.2.5 Simulation Géographiquement Distribuée

A l'heure où les moyens de communication deviennent de plus en plus performants, il est intéressant d'envisager la possibilité de simuler sur des plate-formes distantes. C'est pourquoi, MCI étend le modèle distribué à un modèle géographiquement distribué. Comme l'illustre la

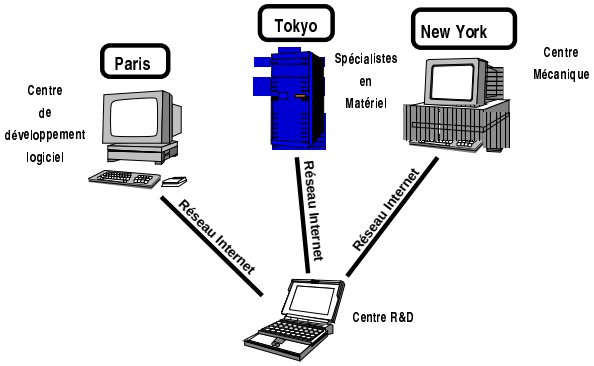


FIG. 3.2 – Environnement de cosimulation géographiquement distribué

figure 3.2, un utilisateur peut donc simuler son système complet sur des machines géographiquement distantes. Cette fonctionnalité offre aux différents groupes de conception la possibilité de coopérer au niveau de la simulation et donc au niveau de la conception. Ce type de simulation permet une plus large utilisation des compétences et des ressources intellectuelles par des équipes spécialisées.

De plus, cette possibilité de simulation permet de contourner les problèmes de licences.[7] La simulation distribuée ne nécessite pas de posséder les licences et les simulateurs sur la même machine, et d'effectuer de l'affichage distribué. Elle permet au contraire de distribuer les moteurs de simulation dans les groupes spécifiques possédant généralement les licences d'utilisation pour les simulateurs dédiés. Il est possible grâce à cette méthode de combiner les machines et les logiciels de simulation afin de partager les ressources CPU disponibles pour les différents sous-systèmes simulés, dans le but d'accélérer la simulation du système global.

L'utilisation de MCI ne nécessite donc pas d'organiser une équipe pour simuler le système complet. Les modèles qui doivent être simulés peuvent provenir d'équipes spécialisées distantes et sont parfois, dans un premier temps, mis au point localement. Les interfaces, ou boîtes noires, des modèles conçus sont ensuite décrites dans un même fichier de coordination qui sera utilisé par MCI pour établir l'environnement complet de cosimulation. A partir de ce fichier, MCI exécute les différents simulateurs distants et établit des liens de communication entre eux. La simulation peut donc être réalisée soit en local, soit à distance. Il est certain qu'à l'heure actuelle, une simulation distante est plus lente à cause de la communication par réseaux. Ce problème sera supprimé dans quelques années grâce à l'installation de réseaux haut débit (e.g., Réseau CADNET). La visualisation des traces et de tous les éléments liés à la simulation peut se faire soit sur plusieurs machines, soit de manière centralisée. Cette visualisation distribuée est intéressante car elle permet aux spécialistes d'analyser et de juger le déroulement de leur simulation. Bien souvent, ce sont les propres concepteurs des modèles simulés qui peuvent comprendre très précisément le déroulement de la simulation. La plate-forme de cosimulation de MCI est constituée d'un routeur permettant de visualiser les interactions entre les différentes simulations et d'effectuer les échanges suivant les comportements des simulateurs.

[7] Les licenses d'utilisation d'un logiciel sont bien souvent à prendre en compte car elles sont, pour certains logiciels, excessivement onéreuses.

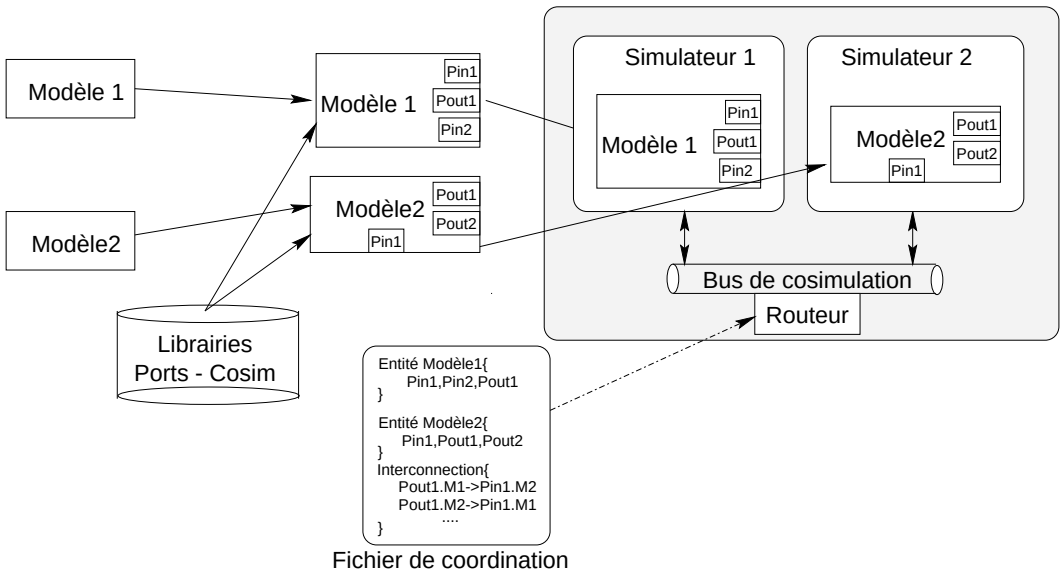


FIG. 3.3 – Principe général de fonctionnement au niveau utilisateur

3.3 Principe de Fonctionnement de l'Outil

MCI est un outil conçu de façon à ce qu'une certaine transparence de fonctionnement soit maintenue vis à vis de l'utilisateur. Son rôle est d'offrir une plate-forme de cosimulation suffisamment flexible proposant à la fois l'utilisation et l'intégration de simulateurs existants. Le système complet est décrit dans un fichier de coordination servant également à exécuter les différents simulateurs et à gérer les données à échanger. L'outil doit pouvoir gérer les différences de modèles d'exécution simulés, en introduisant des synchronisations adaptées, et appliquer des méthodes de conversion de type issues des différents modèles simulés. Enfin, sa structure en couches facilite la communication et ses blocs de fonctionnalités distinctes permettent la gestion des structures de données. Nous présentons ici la constitution de l'outil, les différentes approches mises en oeuvre ainsi que les méthodes de synchronisation.

3.3.1 Etude du Fonctionnement au Niveau Utilisateur

Pour un utilisateur, la succession des opérations nécessaires à la construction d'un environnement de cosimulation avec MCI est représentée sur la figure 3.3. L'utilisateur de MCI doit tout d'abord préparer les modèles qu'il désire cosimuler avant de les inclure dans l'environnement. Pour ce faire, le concepteur utilise des primitives de bibliothèques de cosimulation, adaptées à son type de langage. Ces primitives, appelées ports EXINS-EXOUTS[8] ou "ports de cosimulation", s'ajoutent sous forme de primitives, de fonctions ou de composants supplémentaires au modèle.Ils permettent d'encapsuler les modules afin de pouvoir les connecter en vue de la cosimulation. Ces ports EXINS-EXOUTS possèdent leurs propres architectures décrites dans la bibliothèque fournie par l'environnement MCI. Ces ports assurent la communication entre les modèles et les environnements de cosimulation.

[8]Exins-Exouts pour l'abréviation d'entrées externes ("External IN") et de sorties externes ("External OUT") au simulateur.

Pour établir l'environnement de cosimulation et spécifier les types d'échanges de données à effectuer, l'utilisateur doit ensuite composer un fichier de coordination contenant les paramètres de la simulation globale. Ce fichier regroupe la déclaration des modules (encapsulant modèles et simulateurs), l'ensemble de leurs ports EXINS-EXOUTS ainsi que la liste de leurs interconnexions. Ce fichier de coordination permet à MCI d'associer les simulateurs au modèle à simuler, d'exécuter les différents simulateurs, et d'établir la connexion et l'échange des données. Le paragraphe suivant décrit les transformations des modèles avant leur simulation suivant différents langages et détaille le fichier de coordination.

3.3.2 Encapsulation des Modules Pour la Cosimulation

Chaque module de l'environnement de cosimulation se compose d'un modèle adapté pour la cosimulation et d'un simulateur adéquat pour effectuer la simulation. Les modèles à simuler doivent donc être transformés et adaptés à la cosimulation avant de pouvoir faire partie d'un module. Les transformations qui doivent être apportées aux modèles sont décrites ci-dessous :

Avant d'être compilés et simulés, ils nécessitent l'introduction de ports EXINS-EXOUTS qui permettront à l'environnement de cosimulation de communiquer avec l'intérieur du module à cosimuler. Ces ports sont présents dans des bibliothèques de ports EXINS-EXOUTS qui contiennent toutes les primitives d'entrées/sorties nécessaires pour lier les modèles à simuler et l'environnement de cosimulation. Une bibliothèque d'EXINS-EXOUTS peut éventuellement contenir des *Exinouts*, ports d'entrée et de sortie auxquels certaines conventions, comme la résolution, seront à appliquer suivant les nécessités.

Il existe exactement une bibliothèque pour chaque langage pouvant être inséré dans l'environnement MCI. Les déclarations ajoutées par les bibliothèques de MCI sont dépendantes du langage, de sa syntaxe et de ses propriétés de description. Le principe général pour l'élaboration des bibliothèques est d'utiliser les déclarations standards des langages. Les possibilités de déclaration peuvent être les suivantes :

– la définition de procédures dédiées à la cosimulation pour les langages procéduraux (par exemple C, C++).
  La définition des primitives peut être attribuée à la déclaration de prototypes de fonctions qui seront liées, à l'édition de lien, aux fonctions équivalentes de la bibliothèque MCI.
  Par exemple avant d'intégrer le modèle dans l'environnement de cosimulation, pour le langage C, le concepteur aura besoin de :

1. Référencer la bibliothèque d'EXINS-EXOUTS :

   L'utilisateur inclut la référence à la bibliothèque par la commande :

   #include <exinexoutc.h>

2. Définir les ports à cosimuler :

   L'utilisateur introduit dans son modèle une déclaration du port qu'il désire connecter à l'environnement de cosimulation :

   Port<*Type*>(<*Nom de la variable*>) ;

3. Initialiser les ports :

L'utilisateur introduit au début de son programme un appel de fonction pour initialiser ses ports avant toute exécution, selon le modèle suivant :

CPort<*Type*>Decl(&<*Nom de la variable*>,"<*Nom de la variable*>", <*Valeur initiale*>, <*Direction du port*>) ;

4. Ajouter des primitives d'échanges :

   Pour chaque transmission d'une donnée, le modèle doit contenir une primitive de communication afin de mettre à jour les ports au niveau du simulateur et/ou de l'environnement

   Output_<*Type*>(&<*Nom de la variable*>) ;

   ou

   Input_<*Type*>(&<*Nom de la variable*>) ;

– la définition de composants dans les langages proposant la notion de modules (par exemple VHDL, COSSAP, MATLAB).
Dans ce cas, la bibliothèque de MCI propose un ensemble de composants qui doivent connectés aux ports du modèle. Ces composants peuvent se présenter sous forme graphique ou textuelle (VHDL). Pour les modèles graphiques, l'outil MCI dispose de bibliothèques graphiques, et leur utilisation ne consiste qu'en un simple "glisser-poser" [9] . C'est le cas de MATLAB, COSSAP et SABER. Pour les simulateurs graphiques, comme MATLAB, les EXINS-EXOUTS (e.g., figure 3.4) sont créés de façon à ce que l'utilisateur n'ait plus qu'à les utiliser grâce à quelques mouvements de la souris. En réalité, sous l'affichage graphique les EXINS-EXOUTS contiennent une couche écrite en langage C permettant la connexion à l'environnement de cosimulation lors de l'exécution. Le fait de les placer sur la feuille de travail équivaut à la déclaration de ces ports au niveau procédural.
Pour un modèle textuel écrit en VHDL, la déclaration consiste uniquement à instancier les composants de la bibliothèque de cosimulation VHDL. Le concepteur doit :

1. Référencer la bibliothèque pour le VHDL :

   L'utilisateur fait référence au composant VHDL de la bibliothèque EXINS-EXOUTS.

   use WORK.EXINEXOUT_COMP.all

2. Définir les ports à cosimuler :

   Pour définir un signal comme étant un port à cosimuler, l'utilisateur doit connecter ce signal à une instance de type *Exin*, *Exout*, ou *Exinout* de sa bibliothèque, suivant son type. Pour réaliser l'instanciation d'un port de sortie, il doit inclure :

   <*Instance Name*> : EXOUT_<*Type*>

   Generic Map ("<*Nom du signal à cosimuler*>"), <*Taille du vecteur si le signal est un tableau*>

   Port Map (<*Nom du signal à cosimuler*>)

[9] De l'expression anglaise "Drag and drop".

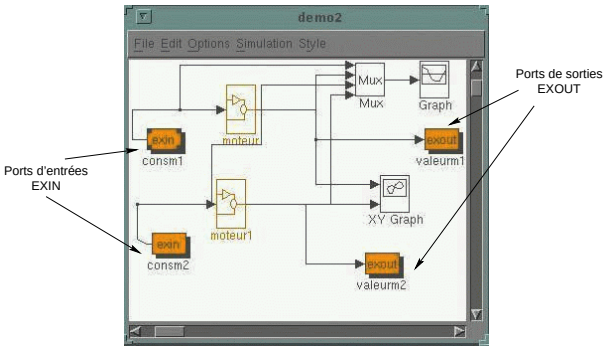


FIG. 3.4 – Modèle MATLAB avec ses EXINS-EXOUTS.

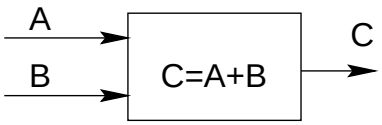


FIG. 3.5 – Exemple d'un additionneur

Lorsque l'utilisateur a instancié tous les signaux qu'il veut cosimuler, son modèle est prêt pour être compilé et exécuté dans l'environnement de cosimulation.

– L'utilisation de propriétés syntaxiques du langage (par exemple SDL).
Le langage SDL est un cas particulier car il communique via un canal connecté vers l'environnement. Cet environnement est considéré à défaut comme étant l'interface utilisateur. Néanmoins, *Object_Geode* propose une API permettant d'intercepter ce canal. Ainsi, si l'utilisateur désire cosimuler un port, il doit utiliser un canal SDL connecté à l'environnement. Lors de la connexion avec MCI, tout ce qui passe dans ce canal est pris en compte par la cosimulation. La déclaration du canal peut être faite de façon graphique comme textuelle.

3.3.2.1 Exemple de modification de modèles pour la cosimulation

Deux exemples de programme contenant les modifications apportées sont représentés dans les tableaux 3.1 et 3.2. Le premier est un programme C et le second est un modèle VHDL. Le même programme consiste à lire 2 entiers, à calculer leur somme et la communiquer à l'environnement (cf. figure 3.5).

Dans tous les cas, pour insérer un modèle dans l'environnement de cosimulation, l'utilisateur doit déclarer les ports qui doivent être cosimulés à l'intérieur des différents modèles. L'architecture des ports est contenue dans les bibliothèques de MCI. Le lien entre ces déclarations et les architectures équivalentes est établi au moment de l'édition de liens ou à la compilation des différents modèles.

3.3.2.2 Compilation des Modèles adaptés pour la cosimulation

La compilation des modèles est en général relativement facile ou du moins elle n'est pas plus contraignante que la compilation classique d'un modèle sans ports EXINS-EXOUTS.

Avant Transformation	**Modèle adapté à la cosimulation**
	#include <exinexoutc.h>
	Portint(a) ;
	Portint(b) ;
	Portint(c) ;
int main() {	int main() {
	CPortintDecl(&a,"a",0,EXIN) ;
	CPortintDecl(&b,"b",0,EXIN) ;
while (1) {	while (1) {
scanf("%d",a) ;	**Input_int(&a) ;**
scanf("%d",b) ;	**Input_int(&b) ;**
c=a+b ;	c=a+b ;
printf("c=%d\n",c) ;	**Output_int(&c) ;**
} /* End While */	} /* End While */
} /* End Main */	} /* End Main */

TAB. 3.1 – Transformation d'un programme C

Avant Transformation	**Modèle adapté à la cosimulation**
library Synopsys ;	library Synopsys ;
use Synopsys.types.all ;	use Synopsys.types.all ;
use Synopsys.attribute.all ;	use Synopsys.attributes.all ;
	use work.EXINEXOUT_COMP.all ;
entity Addition is	entity Addition is
Port (a,b : in integer	Port (a,b : in integer
c : out integer	c : out integer
) ;	) ;
end Addition ;	end Addition ;
architecture behavioral of Addition is	architecture behavioral of Addition is
	Instance_A :EXIN_INTEGER
	Generic Map("a")
	Port Map(a) ;
	Instance_B :EXIN_INTEGER
	Generic Map("b")
	Port Map(b) ;
	Instance_C :EXOUT_INTEGER
	Generic Map("c")
	Port Map(c) ;
begin	begin
c<=a+b ;	c<=a+b ;
end Behavioral ;	end Behavioral ;

TAB. 3.2 – Transformation d'un programme VHDL

- Dans le cas de modèles graphiques, la compilation, si elle a lieu, est relativement transparente et ne pose pas de problèmes.
- Dans les autres cas, elle consiste à utiliser les outils traditionnels de compilation en fonction des langages (e.g., *gcc* ou *cc* pour le C, *vhdlan* pour le VHDL, *gsmcomp* pour le SDL).

C'est l'édition de liens qui, dans la plupart des cas, va établir l'association des primitives ajoutées au modèle avec les architectures de ces primitives contenues dans les bibliothèques de MCI. Ces bibliothèques, chacune associée à un langage particulier, sont statiques ou dynamiques. Dans le premier cas, la bibliothèque sera reliée au moment de l'édition de lien, avant la simulation (Par exemple pour le langage C : *gcc -o ModelExecutable Model.c -lexinexoutc)*. Dans le deuxième cas, le lien est établi en standard par le compilateur, mais sera effectivement réalisé au moment de l'exécution lorsque le simulateur fera appel à une primitive de cosimulation (e.g. VHDL).

3.3.3 Fichier de Coordination

Pour l'utilisateur, le fichier de coordination est l'élément essentiel d'un environnement de cosimulation. Ce fichier fixe les chemins des données à échanger et décrit les interconnexions entre les modules. Chaque module désigne un simulateur et son modèle à simuler. De plus, ce fichier contient les propriétés essentielles qui vont permettre de configurer l'environnement de cosimulation :

1. la liste des modules et les ports que chaque modèle contient,
2. les propriétés telles que le nom de machine, l'adresse IP, etc.. pour permettre l'exécution des simulations distantes,
3. l'énumération des types de chaque module,
4. la liste des interconnexions entre les ports des modules ("*netlist*").

Les détails de la description du fichier de coordination sont présentés dans le paragraphe 3.4.

3.3.4 Architecture des bibliothèques de Cosimulation

Les ports à cosimuler, déclarés dans les modèles, sont liés à des bibliothèques fournies par l'environnement de MCI. Ces bibliothèques permettent en effet, aux ports déclarés de communiquer avec l'environnement de la cosimulation. Elles sont propres à chaque simulateur et leur principe de fonctionnement doit être simple. Plus le développement d'une bibliothèque est court, plus la réalisation de l'intégration d'un nouvel outil est rapide ("*plugin*"). C'est une des raisons pour lesquelles les primitives d'accès à l'environnement de cosimulation sont de la plus grande simplicité et revêtent la forme suivante :

- ***AddNewCosimPort*** : Insertion d'un nouveau port à l'initialisation d'un simulateur,
- ***WriteValue*** (Ecrire une valeur) ***ReadValue*** (Lire une valeur)
- ***WriteEvent*** (Ecrire un événement) ***ReadEvent*** (Lire un événement)
- ***EndTimedStep*** (Fin de pas de simulation pour la cosimulation temporelle à plus bas niveau)
- ***CloseCosim*** (Fin de la simulation).

Ces primitives sont proposées par une bibliothèque propre à MCI (bibliothèque dynamique *libliste.so*) et sont utilisées pour toutes les nouvelles bibliothèques qui doivent être réalisées.

La construction des bibliothèques ne suit pas un schéma spécifique, elles dépendent des simulateurs. Pour chaque langage, nous présentons les aspects principaux de la construction des différentes bibliothèques :

1. Le langage C
   La bibliothèque du langage C est la plus simple à concevoir. Elle consiste à définir dans un programme C les fonctions associées aux primitives (***CPort<Type>Decl***, ***Output<Type>***, etc..). Ce programme C sera ensuite compilé en une bibliothèque statique ou dynamique *libexinexoutc.a* qui est utilisée pour chaque modèle C de l'environnement de cosimulation. L'association de la bibliothèque au modèle se fera lors de l'édition de lien avant la simulation.

2. Le langage VHDL
   La bibliothèque est réalisée par l'intermédiaire de l'outil *cli* pour VSS de Synopsys, et par le compilateur C gcc pour Leapfrog. Dans les deux cas, la construction est similaire. Elle consiste pour Synopsys à créer une bibliothèque *libCLI.so* et pour Leapfrog à générer une bibliothèque *libfmi.so*. Ces deux bibliothèques sont liées aux modèles à simuler lors de l'élaboration. Pour fabriquer la bibliothèque dynamique, il faut réaliser un programme C à partir de fonctions prototypes fixées par les distributeurs des logiciels. Ces primitives diffèrent suivant les fournisseurs, mais utilisent un formalisme qui doit être respecté pour chaque port réalisé (cf. Documentation CLI et/ou Documentation FMI). VSS, le simulateur de Synopsys, utilise trois primitives principales : **EXINEXOUT_Open** pour l'ouverture du modèle à simuler par le simulateur, **EXINEXOUT_Eval** lors de l'apparition d'un événement sur le port et **EXINEXOUT_close** pour terminer la simulation. Leapfrog en utilise deux : **C_init** appelé à chaque nouveau statut du simulateur (démarrage, sauvegarde, restitution du contexte, etc..) et **C_stim** à chaque événement sur un port (premier appel, événement, passage à zéro, etc..). En plus, un très grand nombre de fonctions peut être utilisé dans le but d'extraire des informations du simulateur.
   De plus, il faut établir un paquetage[10] VHDL décrivant les différents ports accessibles et permettant aux compilateurs de lier l'exécutable de la simulation avec les bibliothèques *libfmi.so* ou *libCLI.so*, lors de l'élaboration. Ce fichier contient la déclaration des instances *Exin*, *Exout* et *Exinout* pour tous les types. Les architectures, quant à elles, sont déclarées comme étrangères afin de faire les liens avec les bibliothèques (*libfmi* ou *libCLI*). Ainsi, pour chaque port du paquetage, la déclaration des architectures porte la mention "*FOREIGN*" .

   Citons ci-dessous l'exemple de déclaration de l'*Exout* entier dans le paquetage EXINSEXOUTS pour Leapfrog ou Synopsys :

```
library IEEE;
use ieee.std_logic_arith.all;

entity EXOUT_INTEGER is Generic ( NOM : string );
Port ( PORTOUT : in INTEGER );
end EXOUT_INTEGER;
```

```
architecture C_EXOUT_INTEGER_Model of EXOUT_INTEGER is
attribute FOREIGN of C_EXOUT_INTEGER_Model :
architecture is "clib :EXOUT_INTEGER";
begin
end C_EXOUT_INTEGER_Model;
```

Dans le cas du simulateur VHDL, ceci signifie qu'à la mise à jour d'un port instancié sur un *Exout* entier, le simulateur appelle la procédure **EXOUT_INTEGER_eval** pour VSS ou **EXOUT_INTEGERC_stim** pour Leapfrog respectivement de la bibliothèque libCLI.so ou libfmi.so avec les paramètres de ce port. Ainsi, cet appel réalise la mise à jour des données vers l'environnement de cosimulation en appelant les primitives offertes par la bibliothèque *libliste.so* de MCI.
La manière d'utiliser ce paquetage consiste en une simple déclaration au début du modèle par la directive *use*.

3. Pour MATLAB, il faut de même créer une bibliothèque dynamique pour chaque port (**exin_integer.mexsol** et **exout_integer.mexsol** par exemple). Elle est utilisée lors de l'exécution de la simulation du modèle. La référence à ces bibliothèques est introduite lorsque que l'on ajoute un port *Exin* ou *Exout* sur le schéma graphique du modèle. Pour établir ces ports, il faut créer un programme C contenant des fonctions prototypes fixées par le constructeur, les compléter et les compiler par l'outil *cmex* fourni par MATLAB. Les différents prototypes qui peuvent être utilisés lors de la construction de la bibliothèque sont relatifs aux phases d'exécution de la simulation. Il en existe plusieurs pour l'initialisation : **mdlInitializeSizes**, **mdlInitializeSampleTimes**, **mdlInitializeConditions**. Les autres prototypes disponibles décrivent le déroulement de la simulation : **mdlStart**, **mdlOutputs**, **mdlUpdate**, **mdlDerivatives**, et **mdlTerminate**. A l'exécution de la simulation, le simulateur fait appel à toutes ces procédures et ceci pour tous les ports déclarés. Il consiste donc, pour la cosimulation, à spécifier l'initialisation des ports (**mdlInitializeConditions**) et la mise à jour de leurs données à travers la primitive **mdlOutputs.** A l'intérieur de ces primitives MATLAB redéfinies, l'utilisateur doit se servir des primitives de la bibliothèque *libliste.so* de MCI.

4. Pour COSSAP et SABER, le principe est pratiquement le même que pour MATLAB, à l'exception que les prototypes disponibles sont limités à trois. En effet, ici les prototypes sont minimaux, ils se limitent à **Init**, **Eval**, et **Close**. Ensuite, la compilation de ces primitives doit être contenue dans une bibliothèque statique pour COSSAP et dans trois bibliothèques dynamiques pour SABER (un prototype par bibliothèque). Ces bibliothèques sont référencées de la même façon que MATLAB lorsqu'on ajoute graphiquement un port *Exin* ou *Exout* dans le modèle. La mise à jour des valeurs pour la cosimulation se fait par les primitives de MCI dans la procédure **Eval** appelée à chaque évaluation de port par le simulateur.

5. Pour SDL, le cas est particulier. Nous aurions pu entreprendre la même opération que pour les autres modèles en référençant les bibliothèques par des prototypes déclarés sous forme d'ADT.[11] Néanmoins, cela implique de propager, comme dans le langage C, les primitives d'échanges de données dans toute la description SDL (Ajout de prototype à toutes les entrées-sorties de la cosimulation). Cette opération est délicate, car elle modifie

énormément la description du modèle.
Nous avons donc choisi d'utiliser les canaux abstraits connectés à l'environnement. Afin de réaliser la connexion avec l'environnement de cosimulation, nous avons remodelé le client SDL à notre convenance. Ce client est une API fournie par Object_Geode pour communiquer vers l'extérieur. MCI le lance lui-même et communique avec lui en utilisant ses primitives et celles de l'API Object_Geode.

Toutes ces bibliothèques communiquent avec l'environnement de cosimulation par l'intermédiaire de primitives offertes par MCI. Ces primitives sont simples et permettent juste d'extraire et de lire les données des simulateurs dans une mémoire partagée. Cette mémoire est un tampon de données entre les simulateurs et l'environnement de cosimulation. L'ajout d'un nouvel outil consiste donc à étudier les possibilités dont il dispose pour utiliser les primitives offertes par MCI. Plus la réalisation de la bibliothèque est simple, plus l'introduction du simulateur dans l'environnement est rapide. Parfois, la mise en oeuvre d'une bibliothèque est longue.

La raison majeure provient bien souvent de la pauvreté de la documentation !

Afin que le fonctionnement du processus soit correct, MCI doit créer les mémoires partagées avant que les simulateurs n'utilisent les primitives de lecture/écriture. Ainsi, MCI doit réguler l'ordre des opérations et lancer lui-même les différents simulateurs. Nous présentons maintenant, ce schéma d'exécution pour bien comprendre le déroulement des opérations.

3.3.5 Algorithmes pour la Constitution de l'Environnement de MCI

La cosimulation se déroule en trois étapes importantes :

- La première est la constitution de la structure de données à partir du fichier de coordination. Elle met en place un routeur qui se réfère à cette structure pour aiguiller l'ensemble des données qui transitent entre les modules.
- La seconde étape est l'exécution des différentes interfaces avec leurs paramètres, l'établissement des connexions avec le routeur. Les interfaces (une par simulateur) ont un rôle de gestion des données entre le simulateur et une mémoire partagée servant de tampon. Elles s'occupent d'échanger correctement les données entre le routeur et le simulateur conformément aux règles de synchronisations et en fonction du simulateur.
- La dernière étape est l'opération de routage consistant à orienter les données en fonction du fichier d'interconnexion, de vérifier et de faire les éventuelles concordances de types, de réaliser les adressages multiples, les conversions et les résolutions de types. Cette étape est appelée le mode établi.

3.3.5.1 Initialisation de la Cosimulation

Le processus de lancement de la cosimulation est schématisé sur la figure 3.6. Le premier processus lancé par l'utilisateur (symbolisé par 1 sur la figure 3.6a) est MCI. Nous appelons ce processus le routeur, car son rôle principal en mode établi est le routage des données. Il lit le fichier de coordination (2) et construit une structure de données adaptée qui sera utilisée tout au long de l'exécution par le routeur des données. Une fois sa structure constituée, le routeur

[10] Un fichier "package" est une bibliothèque pour le VHDL. C'est un fichier contenant des déclarations de types, de procédures, d'instances, etc.. Pour l'utiliser, il faut le référencer par la directive "use".

[11] "ADT" : "Abstract Data Type". Définition de type abstrait dans SDL (Object Geode).

exécute autant de processus interfaces (3) qu'il y a de modules dans le fichier de coordination. Dans le cas où le module doit être exécuté sur une autre machine, le lancement de l'interface doivent s'effectué sur la machine cible. Le routeur créé deux "*sockets*" :

- une "*socket*" locale proposant une connexion aux interfaces (modules) s'exécutant sur la machine locale du routeur,
- une "*socket*" de domaine Internet pour proposer la connexion des interfaces distantes.

Après un moment d'attente, le routeur établit enfin la connexion entre lui même et les différentes interfaces qu'il a exécutées. Toutes les informations communiquées entre le routeur et les interfaces sont désormais transmises via ce support de communication. Le routeur se met alors en veille dans l'attente d'une identification de ses différentes interfaces via la "*socket*". Cette identification précise que l'interface a bien fonctionné, que le simulateur s'est bien exécuté, que l'échange des données est bien opérationnel et que la simulation peut donc démarrer.

Avant de démarrer, les interfaces communiquent au routeur la liste des ports qu'elles ont détectée. En échange, le routeur vérifie la cohérence de ces ports et transmet aux interfaces des numéros associés à chaque port. Ces numéros permettent d'accélérer la vitesse des échanges entre le routeur et les interfaces, en référençant les données modifiées uniquement par ces numéros.

La relation qui existe entre le routeur et ses interfaces se résume en une adaptation et une synchronisation des modules, quelque soit leur type et leur distance. C'est la modélisation du bus de cosimulation. Les interfaces sont des processus qui permettent de relier les simulateurs et le bus de cosimulation. Elles ont aussi pour rôle de créer la mémoire tampon et de lancer les simulateurs pour enfin échanger les données vers le routeur.

Le flot d'exécution de l'interface, représenté sur la figure 3.6b, se compose tout d'abord d'un sémaphore de synchronisation général (4) qui se bloque au premier appel. Ce sémaphore permet d'exécuter une à une les interfaces, s'il est nécessaire de lancer plusieurs interfaces sur la même machine. Ceci est très important si l'on veut éviter d'avoir des problèmes de connexions entre un simulateur et une interface qui ne lui est pas associée. L'interface crée ensuite une mémoire partagée (5) qui lui sert d'intermédiaire avec le simulateur et lance le simulateur avec le modèle à simuler (6). L'interface se met en veille et attend la modification de la mémoire partagée qui doit se faire par le simulateur. Si cette modification n'intervient pas au bout d'un certain temps, l'interface considère, soit que le simulateur ne contient pas d'EXINS-EXOUTS, soit qu'il n'arrive pas à se connecter à cette mémoire partagée pour quelque raison que ce soit. Dans ce cas l'environnement de cosimulation s'arrête et tue tous les processus et simulateurs lancés.

Lors de son initialisation, le simulateur appelle la primitive d'initialisation des EXINS-EXOUTS. Cette procédure se connecte simplement sur la mémoire crée par l'interface (7) et écrit les ports à cosimuler dans la mémoire partagée. Elle débloque donc l'interface qui s'était mise en veille (8).

L'interface, qui a désormais détecté la présence de données dans la mémoire, communique les ports inscrits vers le routeur (étape d'identification), lève le sémaphore global (9) et se prépare au début des échanges.

Les interfaces se servent de la mémoire partagée (fig. 3.7) comme de mémoire tampon entre les simulateurs et les interfaces. Elles créent également des sémaphores qui se chargent de synchroniser les lectures et écritures des données dans cette mémoire. Les références des sémaphores sont inscrites dans la mémoire partagée par l'interface qui les crée. Ces références

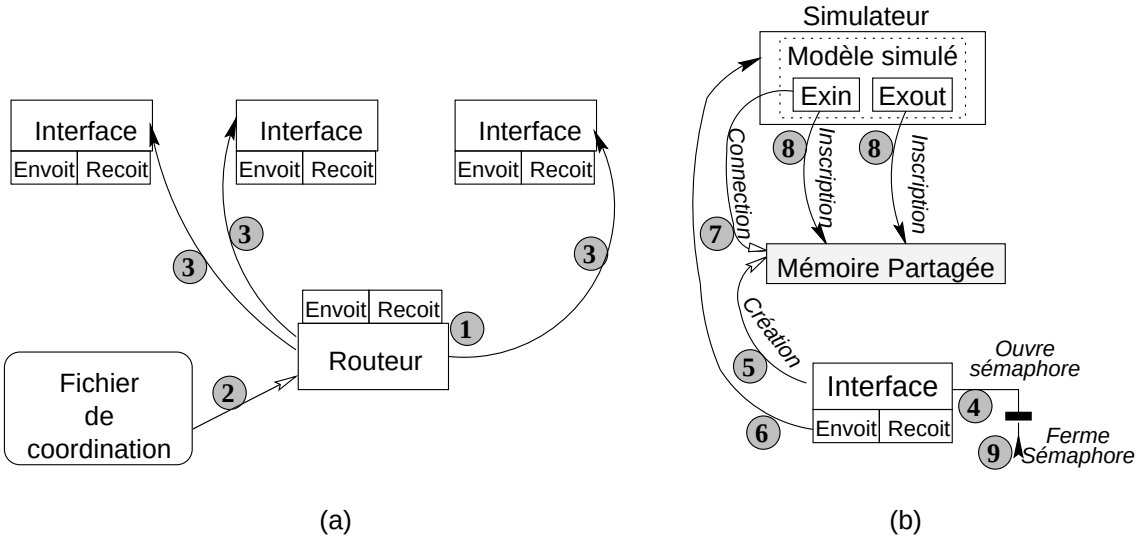


FIG. 3.6 – Ordre d'exécution des processus

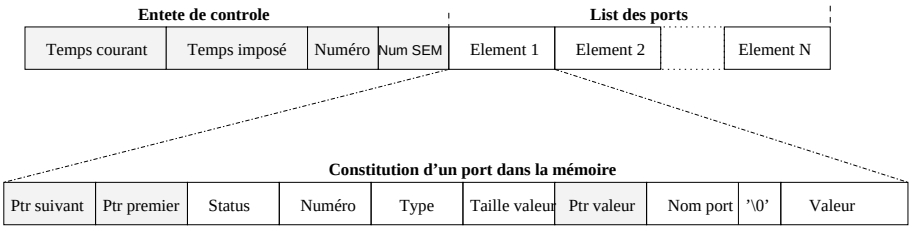


FIG. 3.7 – Structure de la mémoire partagée de MCI

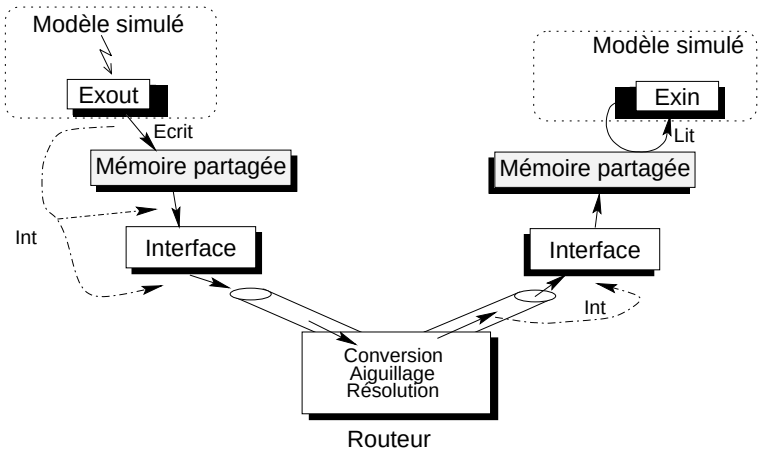


FIG. 3.8 – Flot de transfert d'une donnée modifiée

peuvent être utilisées à tout moment par un simulateur pour se connecter au sémaphore de synchronisation des lectures et écritures de cette mémoire.

3.3.5.2 Schéma Explicatif du Fonctionnement de MCI durant l'Exécution de la simulation

En mode établi, c'est à dire lorsque l'ensemble des processus interfaces et simulateurs fonctionne correctement, le routeur de MCI se charge d'aiguiller les données modifiées. La figure 3.8 représente le parcours complet d'une donnée lorsqu'elle est modifiée entre deux modules. Dans le cas présenté, ces deux modules sont distants, mais le fonctionnement reste similaire si ces deux modules fonctionnent sur la même machine. Tout d'abord, chaque simulateur est lancé et a créé, à son initialisation, l'ensemble des ports cosimulés dans la mémoire partagée. Les interfaces sont en mode veille lorsqu'aucune donnée ne transite dans l'environnement. Si un simulateur modifie une donnée en écriture, celle-ci est écrite dans la mémoire partagée via l'*Exout*. Aussitôt, l'interface se réveille et intercepte la donnée pour la communiquer au routeur via la "*socket*". Le premier mécanisme de synchronisation consiste à lire toutes les données modifiées sans en perdre une seule et s'assurer que les données sont transitées dans l'ordre de leur modification. Par sa structure, le routeur reconnaît le port qui communique la donnée et élabore la liste des connexions pour ce port. Pour tous les ports connectés, le routeur écrit la donnée traitée dans les "*socket*" correspondantes. L'arrivée d'une donnée réveille l'interface qui se trouve en réception et l'écrit au bon emplacement dans la mémoire partagée, pour enfin se remettre en veille. Le modèle récepteur (le simulateur destinataire) peut alors lire cette donnée modifiée à sa demande via un port *Exin*.

Afin de gérer un parallélisme dans l'arrivée des données, le routeur utilise des "*threads*". Il crée autant de "*threads*", qu'il y a d'interfaces. Les "*threads*" sont très pratiques car ils ont la particularité d'être multi-processus tout en partageant la même structure de données. Un système de synchronisation entre "*threads*" est utile si le routeur doit traiter des résolutions multiples ou des priorités dans les signaux qui transitent.

On s'aperçoit que le routage des données doit respecter un chemin particulier. Il convient donc à l'utilisateur de définir ce ou ces chemins dans un format suffisamment complet pour définir les interconnexions et toutes les propriétés nécessaires au bon fonctionnement de l'envi-

ronnement. L'ensemble de ces informations forme la structure de données du routeur et provient du fichier de coordination écrit en SOLAR [JO94].

3.4 Coordination Inter-Modules

La coordination des modules entre eux doit être décrite de la manière la plus précise possible. C'est pourquoi, il est important d'utiliser un langage cohérent, capable de modéliser un système sous forme d'instance et de connexions. De plus, il doit être capable de représenter les types utilisés par l'ensemble des langages qui sont impliqués dans l'environnement de cosimulation. Ce langage de coordination peut posséder une syntaxe très simple, car seules les interconnexions des différents modules doivent être représentées et en aucun cas le fichier de coordination ne devra décrire l'architecture d'un module. Dans le cadre des travaux sur MCI, nous avons choisi d'utiliser le langage SOLAR développé au laboratoire TIMA. Ce langage de modélisation possède une syntaxe complète pour la description des systèmes et est donc très adapté pour ce type de description, à la fois pour décrire les modules mais aussi pour ses possibilités de définition de types.

3.4.1 Utilisation du Langage Intermédiaire SOLAR

La syntaxe de description SOLAR est suffisamment complète et très bien adaptée pour construire un fichier de coordination comportant des propriétés, des listes de modules ou instances, des ports, des descriptions de types et surtout des interconnexions ou des "*nets*".

La description de l'environnement de cosimulation, par l'intermédiaire de ce fichier de coordination, est nécessaire pour MCI. La syntaxe SOLAR, est donc utilisée en entrée de l'outil. Les informations contenues dans le fichier de coordination peuvent être décrites manuellement ou de façon automatique par un outil de "*co-design*". Une description manuelle du fichier consiste à inclure :

- les modules, à l'aide du mot clé *Instance* en SOLAR. Le paramètre associé à l'instance définit le nom du fichier contenant le modèle à exécuter par le simulateur. L'*instance* doit aussi contenir les propriétés[12] suivantes :
  - **le type de module** (logiciel, matériel, mécanique, etc..). Il permet de définir le mode à adopter pour le traitement et la synchronisation des données. De plus, il permet de connaître, sauf cas particulier, le simulateur qui doit être exécuté pour le modèle du module,
  - **une propriété** nom de machine qui permet de spécifier la machine sur laquelle le module va être attitré. Cette propriété est utilisée par l'environnement lors de l'exécution des processus et permet de connaître le type de support de communication à utiliser ("*socket*" Internet ou locale).
- la liste des ports du module. Elle regroupe uniquement le nom des ports à cosimuler des différents modèles de l'environnement. Ceci correspond à la définition des ports d'une entité en langage VHDL (mot clé "*portinstance*"),
- la liste des interconnexions des *instances* et de leurs ports. Pour cela on utilise les mots clés "*net*", "*joined*", "*portref*"et "*instanceref*",
- la description des types et directions des différents ports déclarés. Cette description est utile pour les conversions et les résolutions. En SOLAR, il est nécessaire de représenter

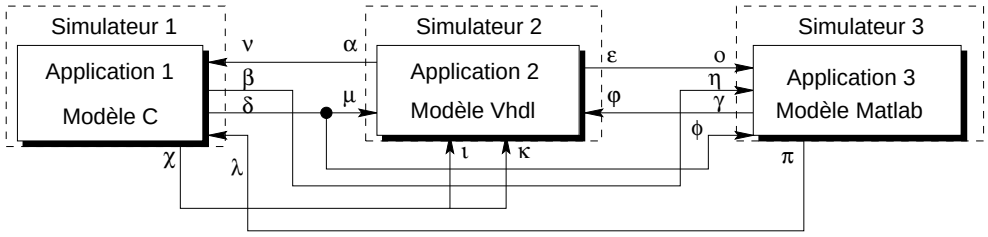


FIG. 3.9 – Schéma d'interconnexions

des architectures sommaires(“*DesignUnit*” en SOLAR) pour chaque modules pour représenter les types des ports.

3.4.2 Description des Interconnexions

La description des interconnexions des modules (“*instances*”) entre-elles est réalisée par les “*nets*” de SOLAR. Dans un “*net*” il suffit de spécifier une liste de ports pour les connecter ensemble. On peut donc spécifier le(s) port(s) source(s) et le(s) port(s) destination(s) qui lui est(sont) connecté(s). La figure 3.9 illustre un système à cosimuler, et pour bien comprendre la modélisation SOLAR, nous avons décrit le fichier de coordination équivalent (figure 3.10). Le couple modèle + simulateur est représenté en “*instances*” et le nom de “l'*instance*” porte le nom du modèle à simuler (figure 3.10b). Les ports à échanger α, β, γ, etc.. se retrouvent dans la définition des “*instances*”, car ils représentent les connexions extérieures aux modèles. Les types de ces ports sont déclarés dans les architectures (“*designunit*”) associées aux “*instances*” (figure 3.10c). Et les connexions des ports entre eux sont définies par un ensemble de “*nets*” (figure 3.10b).

Cette étape d'écriture du fichier de coordination peut se faire de façon automatique en relisant les modèles. Dans ce cas, seules les interconnexions doivent être spécifiées. Ce fichier de coordination est lue au début de l'exécution de l'environnement et sert à construire la structure de donnée de l'environnement de cosimulation, utilisée par le routeur pour aiguiller les données.

3.4.3 Vérification des Interconnexions et Types de Données

Les données du fichier de coordination permettent, dans un premier temps, de vérifier la cohérence des interconnexions et les types propres aux ports afin d'éviter toutes erreurs. Pour cela, un certain nombre de tests sont effectués :

1. le parcours de la “*netlist*” d'interconnexions en :
   - contrôlant l'ensemble des références “d'*instances*” dans chaque *net*,
   - contrôlant si les ports de chaque *nets* sont bien déclarés dans les “*instances*” associées par les “*instanceref*”,
   - vérifiant qu'il n'existe pas de port non connecté.
2. Parcourir l'ensemble des architectures (“*DesignUnit*”) et de la “*netlist*” d'interconnexions en parallèle :

[12]Les propriétés sont désignées par le mot clé PROPERTY en SOLAR.

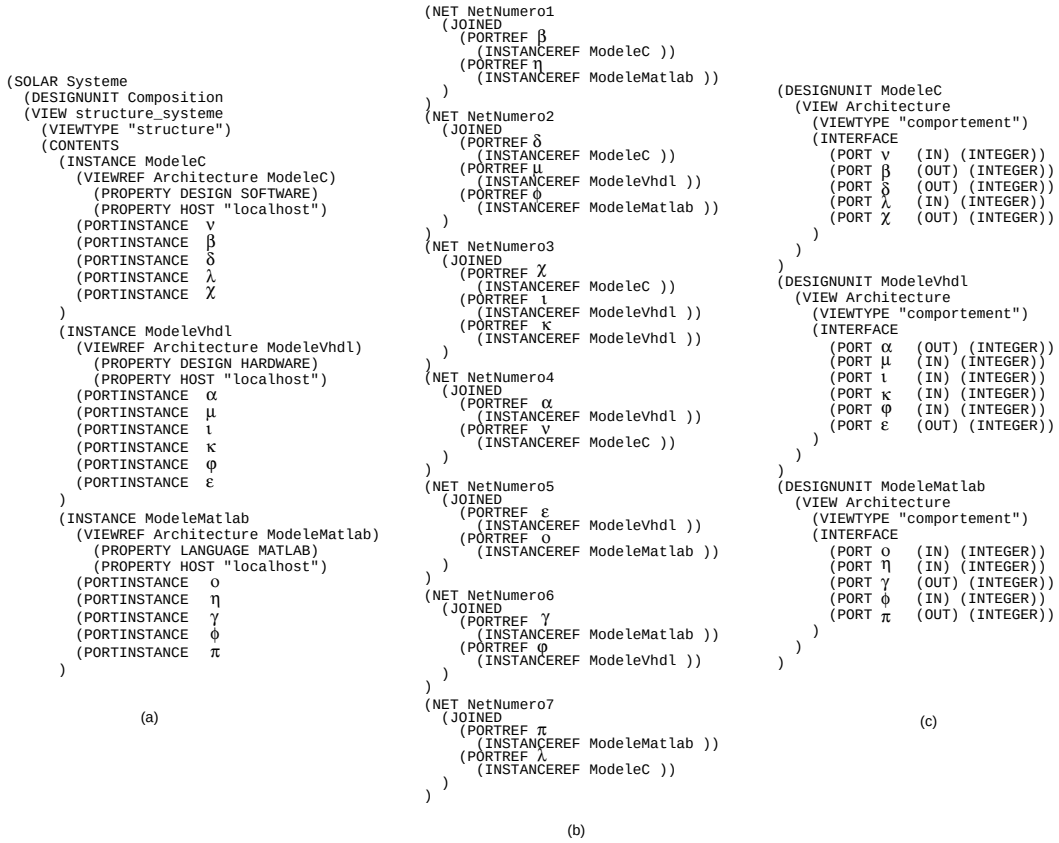


FIG. 3.10 – Exemple de fichier de coordination

Représentation SOLAR	C	VHDL	MATLAB	COSSAP	SDL	SABER
BIT	char(1)	bit(1)	×	char	×	×
CHAR	char(1)	×	×	char	char	×
INTEGER	int	integer	×	integer	integer	×
REAL	float	real	double(8)	float	real	double(8)
STD_LOGIC (_V..)	×	std_logic(1)	×	×	×	×
SIGNED || UNSIGNED	×	signed || unsigned(8)	×	×	×	×

× : Représentation non supporté par MCI pour le langage

STD_LOGIC(_V..) : std_logic et std_logic_vector

(n) : Taille physique réservée du type dans la mémoire partagée en octets

TAB. 3.3 – Représentation des types disponibles

- pour vérifier la bonne connexion d'un port de sortie (*OUT* ou *INOUT*) sur un port en entrée (*IN* ou *INOUT*),
- vérifier l'écriture des types,
- contrôler la compatibilité et les conversions des types interconnectés.

Les types simples disponibles pour l'environnement de cosimulation MCI sont représentés dans le tableau 3.3. Nous décrivons ici les types possibles de représentation pour chaque langage dans le format unifié SOLAR, mais aussi le format des types tels qu'ils sont interprétés au niveau des ports de cosimulation EXIN-EXOUTS à la sortie des simulateurs. En plus, nous indiquons la taille physique des valeurs en octets, lorsqu'elle n'est pas dépendante de l'architecture de la machine qui simule. Pour les modules distants, les interfaces effectuent localement les conversions nécessaires pour les types de données dépendants de l'architecture de la machine utilisée.

MCI, via le routeur, contrôle les types de connexions avant de réellement démarrer sa cosimulation . Un module cosimulé écrit, dans sa phase d'initialisation, les ports à cosimuler (EXINS-EXOUTS) dans une mémoire partagée réservée. Il crée ainsi un champ dans la mémoire partagée, correspondant au type et à la taille de la donnée qui est échangée (fig. 3.7). Pendant l'identification de ses interfaces (cf. §3.3.5), le routeur établit une communication entre lui-même et ses interfaces et vérifie la compatibilité entre les types écrits dans les différentes mémoires partagées et ceux définis dans le fichier de coordination. Pour ce faire, il se conforme au tableau 3.3.

3.4.3.1 Echange de Données Particulières

Le traitement des types est un atout important de l'outil MCI. En plus des types simples, il permet maintenant de modéliser des tableaux et de définir plus précisément la taille des types. Il permet aussi de connecter des types différents.

Il est capable de convertir les types entre eux en tenant compte si nécessaire du tableau 3.4. Cependant, il convient d'être vigilant quant à ces conversions. C'est pourquoi, le routeur avertit l'utilisateur de toutes ces modifications. En effet, même si un type converti a une signification au niveau simulation, comme par exemple un Bit connecté sur un STD_LOGIC, il nécessite une adaptation particulière à plus bas niveau, comme la synthèse d'interfaces ou le ciblage en fonction de l'architecture.

	Types Destination					
	BIT	CHAR	INTEGER	REAL	STD_LOGIC(_V..)	(UN)SIGNED
BIT	o	x	x	x	o - x	x
CHAR	x	o	o	o	x - o	o
INTEGER	x	o	o	o	x - o	o
REAL	x	x	o	o	x - x	x
STD_LOGIC(_V..)	o - x	x - o	x - o	x - x	o	x - o
(UN)SIGNED	x	o	o	x	o - o	o - o

× : Conversion non supportée

o : Conversion supportée

STD_LOGIC(_V..) : std_logic et std_logic_vector

TAB. 3.4 – Conversion de types

Pour ne pas compliquer la gestion du routeur, qui imposerait une perte de vitesse de traitement, nous limitons les types permis au niveau VHDL à des types relativement simples mais modulables en taille (STD_LOGIC, SIGNED, INTEGER, REAL). En général, les autres modules utilisent des types similaires à ceux qui existent en langage C (char, int, float, double).

3.5 Méthodes de Synchronisation

Les méthodes de synchronisation dans un outil de cosimulation multilangage doivent être présentes pour respecter les différences de comportement des différents langages utilisés. De plus, elles doivent être appropriées au niveau d'abstraction de la simulation. Si l'on considère que la simulation est distante, l'outil doit de plus tenir compte du support de communication et de sa synchronisation intrinsèque. Les changements de niveaux d'abstraction qui interviennent dans les différentes phases de cosimulation du flot de validation d'un système influent sur le comportement de l'ensemble de l'environnement et impose une adaptation de la synchronisation. Dans cette partie, nous présentons les différentes synchronisations de l'outil, leur localisation, leur fonctionnement, et les incidences de fonctionnement en fonction des niveaux d'abstraction de la simulation.

3.5.1 Les Différents Types de Synchronisations Utilisés dans MCI

La clé de voûte d'un environnement de cosimulation est la synchronisation qu'il adopte. Dans notre interface, elle est strictement nécessaire pour obtenir un ordre précis de lecture et d'écriture dans le bus de cosimulation. Etant donné que l'environnement de cosimulation doit être un environnement multilangage ou géographiquement distribué, l'outil se compose en deux niveaux principaux de synchronisation. Comme le montre la figure 3.11, un premier niveau intervient entre la machine distante et celle qui exécute le routeur (entre les interfaces et le routeur). Il a pour rôle essentiel de transporter les données. Un second niveau permet de gérer les données en local en fonction des types de simulateurs (cf. §2.4.1.2) (entre le simulateur et l'interface).

En plus d'être nécessaire, cette synchronisation est très importante pour la vitesse d'exécution de l'environnement : elle est aussi utilisée pour éviter les processus de s'exécuter inutile-

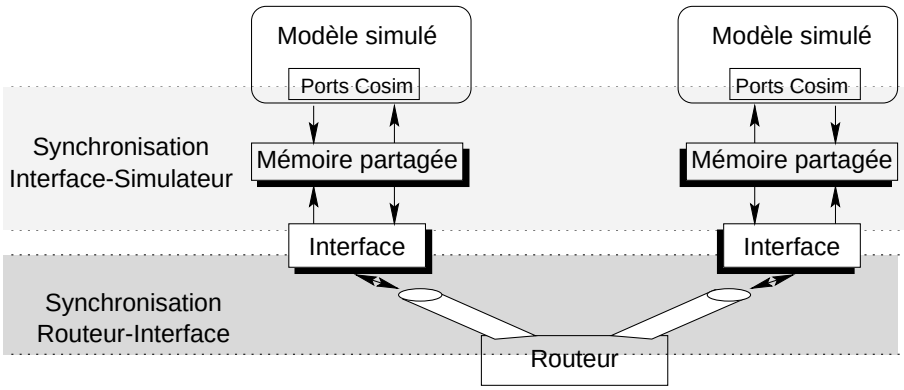


FIG. 3.11 – Représentation des synchronisations dans MCI

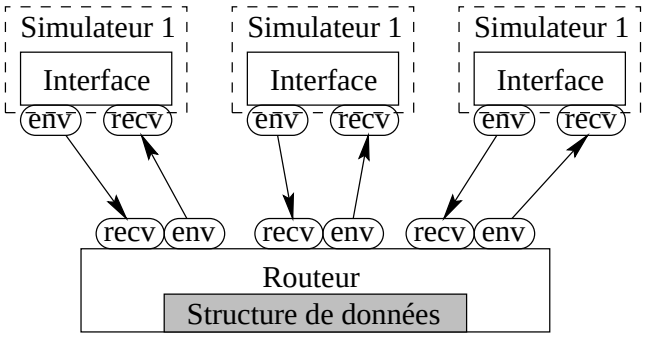


FIG. 3.12 – Synchronisation Interfaces - routeur

ment. Le temps CPU économisé peut donc être réutilisé par les simulateurs et le routeur, et ainsi accélérer la simulation globale de façon notoire.

3.5.1.1 La synchronisation entre les Interfaces et le Routeur

La synchronisation entre les *interfaces* et le *routeur* doit se faire de façon bidirectionnelle. Puisque cette synchronisation et ce transport peuvent apparaître entre deux machines distantes, nous avons opté pour l'utilisation des "*sockets*", comme moyen de transport.[13] Elles ont la propriété de se mettre en mode bloquant, ce qui permet d'avoir une synchronisation sécurisée des données qui transitent.[14] Elles peuvent également débloquer automatiquement, par interruption, les processus connectés à cette "*socket*", dès l'arrivée d'une donnée. Ayant une propriété de latence, elle permet aussi de stocker des données, telle une FIFO, et donc de créer une synchronisation extensible. L'environnement de cosimulation nécessite également une synchronisation entre l'action des simulateurs et le routeur. C'est pourquoi il est important d'établir soit une synchronisation supplémentaire particulière, soit un protocole de communication.

Dans MCI, la synchronisation entre les interfaces et le routeur est exécutée de façon bidirectionnelle via le support de communication "*socket*", comme l'indique la figure 3.12. La

[13]Les sockets ont un avantage, car on peut modifier leur type (domaine local ou domaine internet) sans changer leurs méthodes d'accès.

[14]La synchronisation est réalisée par un protocole poignée de main "Handshake".

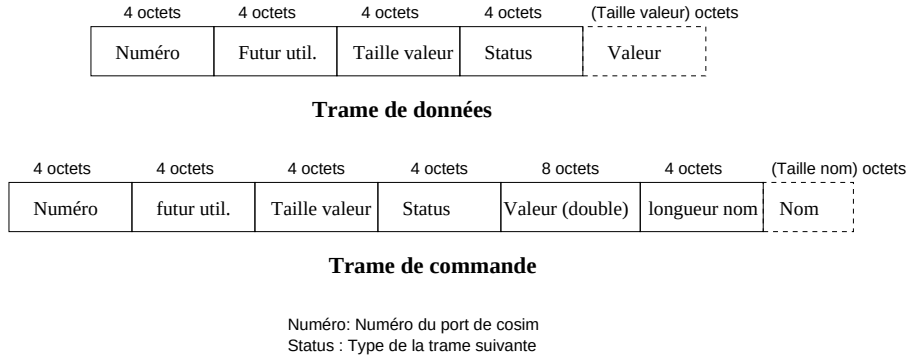


FIG. 3.13 – Schéma du protocole de communication dans MCI pour la synchronisation Routeur-Interfaces.

synchronisation est réalisée, d'une part par la "*socket*" bloquante elle-même et d'autre part par une synchronisation entre les interfaces et le routeur via un protocole d'échange de commande à travers la "*socket*".

La synchronisation de la "*socket*" est gérée par un processus d'envoi et de réception, que ce soit du côté du routeur ou des interfaces (cf. figure 3.12). Les processus d'envoi utilisent des primitives d'écriture dans la "*socket*", comme des appels de procédure. Les processus de réception sont des processus dormants, qui se réveillent à la présence unidirectionnelle d'une donnée dans la "*socket*". Les processus routeur + ("*recv*" et "*send*"[15]) et interface + ("*recv*") sont en temps normal en veille. Si une donnée arrive sur la "*socket*" du routeur par l'envoi d'une valeur par le processus interface ("*send*"), le processus récepteur du routeur se réveille sur interruption et s'occupe d'un premier raffinement de la donnée avant de la fournir au routage (conversions système ("*ntohl*", "*htonl*"), traitement du type de trame). Du côté de l'interface, le fonctionnement est similaire puisque la donnée qui arrive réveille directement le processus de réception. Il se charge à son tour d'écrire la donnée au bon endroit dans la mémoire partagée.

La définition d'un protocole d'échange est essentielle pour synchroniser les interfaces et le routeur. Par exemple, pour les simulateurs événementiels et d'autres cas particuliers (échange de temps pour la synchronisation de bas niveau, échange de paramètre de simulation) des informations supplémentaires sont utiles (ordres à transmettre). Pour cela, il existe un protocole d'échange comprenant deux types de protocoles, comme l'indique la figure 3.13. Le premier est le protocole de données qui sert à la communication des données modifiées. Dans le cas des échanges classiques de données modifiées, seul ce protocole est utilisé. Il doit être par ailleurs optimal pour diminuer les temps de communication. Il sert également à transporter les commandes principales de synchronisation comme (Début d'un événement, fin d'un événement, prochain pas de simulation, Fin de pas de simulation, transports d'erreurs, etc..). Pour cela, les différentes commandes transportées sont détectées par la valeur du "status" de la trame. Lorsque le protocole de données ne suffit pas à transmettre l'ordre, le deuxième protocole (le protocole de commande) permet de spécifier certains ordres plus complets éventuellement sous forme de chaîne de caractères. Un protocole de commande étant optionnel et moins fréquemment utilisé, il est toujours précédé d'un protocole de donnée dont le statut spécifie son arrivée, ceci pour diminuer la charge de communication.

Il existe dans MCI une vraie conversation entre les interfaces et le routeur pour la communi-

[15]Recevoir et envoyer.

cation d'un certain nombre de consignes. Elles sont essentiellement nécessaires à l'initialisation (pour l'échange de nom de ports et de numéro), à bas niveau, et lors d'échanges d'événements. Elles permettent de lier le routeur avec ses interfaces.

3.5.1.2 La Synchronisation entre les Blocs Interfaces et les Simulateurs

L'interface sert aussi à coordonner la synchronisation des données avec des simulateurs (cf §2.4.1.2). Il existe à ce niveau trois différents modes de synchronisation en fonction des types de simulateurs utilisés dans nos travaux.

Ces synchronisations ont été développées dans MCI, avec des sémaphores (cf. IPC Unix) qui permettent à la fois une synchronisation rapide et une optimisation des ressources du système. Afin de mettre en oeuvre la synchronisation entre les interfaces et les simulateurs, nous avons construit deux procédures d'envoie de données qui sont utilisées dans les ports de cosimulation en sortie (*Exouts*), lors de leur création. Ces fonctions font parties des primitives de MCI qui sont utilisées lors de la fabrication des bibliothèques pour l'accès à l'environnement de cosimulation des simulateurs (cf. §3.3.4). Elles réagissent différemment selon le type de simulateur mais stockent leurs valeurs localement dans une mémoire partagée associée à l'application (fig. 3.7).

Les trois différents modes représentés sur la figure 3.14 sont les suivants :

1. Le premier mode (Figure 3.14a) est un principe qui est adaptable à tous les simulateurs ayant un fonctionnement purement séquentiel dans le temps. C'est le cas, par exemple, du langage C qui modifie ses variables au fur et à mesure de son exécution. Pour cette synchronisation deux sémaphores sont utiles : l'un au niveau du simulateur, l'autre au niveau de l'interface.

   (a) L'écriture d'une donnée externe du simulateur devant être transmise au routeur au moment où elle est écrite. La fonction d'écriture vers l'extérieur (côté simulateur) écrit la valeur modifiée, lève le sémaphore au niveau interface, et bloque son propre sémaphore. L'interface qui était en veille se débloque, lit la donnée et lève le sémaphore du simulateur pour lui permettre de reprendre son exécution. Dès lors, l'interface peut en parallèle transmettre la donnée vers le routeur et se remettre en veille.

   (b) La lecture n'utilise pas de synchronisation particulière puisque ce type de simulateurs lit une valeur extérieure au moment où son exécution le définit. Si la donnée est inchangée, l'exécution lit tout de même cette donnée inchangée. La modification d'une donnée sera donc lue à la prochaine directive de lecture du programme exécuté.

2. Le deuxième mode de synchronisation (figure 3.14b) s'applique à l'ensemble des simulateurs événementiels qui ont la possibilité de communiquer vers l'extérieur via une interface C (e.g., VSS de Synopsys et son interface *CLI*). Dans cette gamme de simulateurs, les données sont échangées à l'extérieur du simulateur sur des événements et ceci par un appel de procédure C. Bien que ces simulateurs puissent communiquer via le C, on rencontre bien souvent un changement de comportement lors de l'exécution. Le mécanisme interne d'évaluation du simulateur est faussé à la sortie de l'interface C à cause de la séquentialité de l'exécution C. Le simulateur prend implicitement un ordre d'échange des données lorsqu'un événement apparaît. Cet ordre est fixé par la façon dont le modèle

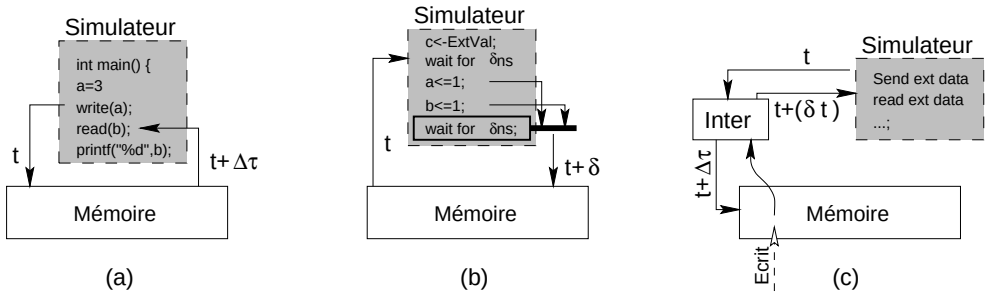


FIG. 3.14 – Types de simulateur

simulé aura été écrit. C'est pourquoi, pour corriger un éventuel désordre, il est nécessaire d'utiliser une synchronisation à base de deux sémaphores modélisant les événements :

(a) En écriture, la première donnée modifiée décrit le comportement :

i. Si la donnée est la seule modifiée, alors le comportement rentre dans le cas énoncé en 1.

ii. Si ce n'est pas le cas, alors plusieurs données sont modifiées. Elles sont donc écrites dans la mémoire partagée une à une, mais la dernière lève le sémaphore de l'interface et stoppe son exécution. Ce sémaphore réveille le processus *Interface* qui lit l'ensemble des données modifiées de la mémoire et les communique au routeur en précisant pour la première un début d'événement et la dernière une fin d'événement. Lorsque l'interface a repéré les données modifiées, elle peut relever le sémaphore du simulateur avant/après avoir transmis les données au routeur.

(b) La lecture des données est également réalisée sur des événements. Néanmoins, il n'est pas possible de réaliser directement le comportement d'un événement sans avoir une implémentation compliquée par des "*threads*". C'est pourquoi, dans ce cas la façon d'agir est plutôt une détection d'événements. On précise au simulateur de faire de la relecture de ses entrées à cosimuler à intervalles réguliers.[16] Afin de s'assurer qu'on ne perde pas d'informations, ce temps minimum de réactualisation est fixé à l'unité de la base de temps du simulateur. Il est vrai que cette méthode, même si elle est efficace, n'est pas rapide et consomme beaucoup de ressources processeur. Il n'y a donc pas d'utilisation de synchronisation particulière si ce n'est qu'une recherche de modification des entrées de la mémoire partagée, par le simulateur, selon une fréquence donnée.

3. En ce qui concerne la synchronisation des simulateurs séquentiels, une synchronisation particulière est nécessaire pour ceux qui communiquent vers l'extérieur via un module indépendant du simulateur (figure 3.14c) . C'est le cas du simulateur SDL, puisqu'il utilise un processus à part qui commande le simulateur (le *Postmaster* pour Telelogic, le *ClientSDL* pour Object_Geode, ***Inter*** sur notre schéma). Dans ce cas, l'utilisation de deux sémaphores est encore nécessaire, mais l'avantage essentiel reste qu'entre ***Inter*** et le simulateur, le schéma reste identique dans le cas d'un simulateur événementiel. Par cette synchronisation supplémentaire, il n'est pas nécessaire d'ajouter de nouvelles primitives aux niveaux lecture/écriture du simulateur.

(a) En sortie, la modification de données, à communiquer vers l'extérieur, est automatiquement interceptée par le process ***Inter*** (Nom générique de *Postmaster* ou *ClientSDL*). Dans ce cas, à partir d'***Inter***, on utilise le mode de synchronisation de la figure 3.14a. Ce type de simulateurs événementiels définit l'ordre d'envoi des signaux selon une FIFO gérée au sein du simulateur. C'est pourquoi, il n'est pas nécessaire d'ajouter un mécanisme de contrôle de l'ordre des valeurs modifiées durant l'événement.

(b) En entrée, la synchronisation se fait entre l'interface et le processus ***Inter***. Lorsque l'interface reçoit une donnée dans la mémoire partagée, il débloque par sémaphore le processus ***Inter*** qui en temps normal d'inactivité est en veille. Le réveil de ***Inter*** bloque l'exécution de l'interface afin qui n'y ait pas d'écrasement de données arrivant successivement. Enfin, le processus ***Inter*** envoie la donnée au simulateur via une commande de l'API et s'arrête après avoir débloqué l'interface.

4. Enfin, il est intéressant de noter qu'une quatrième synchronisation a été ajoutée dans toutes les primitives de lecture de la mémoire partagée du côté simulateur pour les événements. Cette synchronisation est utilisée lorsque le *routeur* communique à l'*interface* un événement. En effet, il faut interdire la lecture du simulateur lors de la mise à jour d'un ensemble de données provenant d'un événement. Pour cela, chaque primitive de lecture contrôle l'état d'un sémaphore. En l'absence d'événement le sémaphore est égal à 1, la primitive n'est pas bloquante et peut donc lire la donnée normalement. Si un événement intervient, l'*interface* baisse ce sémaphore. L'*interface* ne reste pas bloquée, elle peut donc mettre à jour les données de l'événement. Par contre, les primitives d'accès en lecture du côté simulateur vont se bloquer lorsqu'elles vont vouloir lire dans la mémoire. A la fin de la mise à jour des données de l'événement reçu, l'*interface* lève ce sémaphore et l'ensemble des primitives peut à nouveau lire normalement les données.

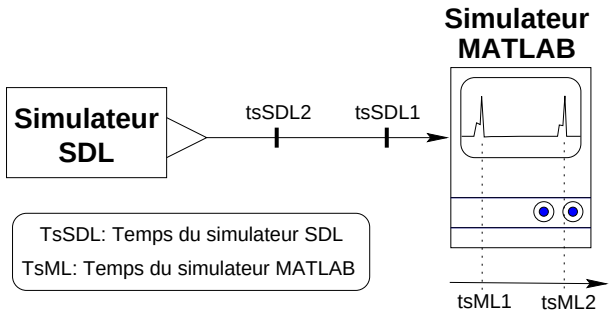
3.5.2 Synchronisation Temporelle et Non Temporisée

La synchronisation est aussi dépendante du niveau d'abstraction de la simulation. Si la cosimulation d'un système complet consiste à simuler conjointement du VHDL comportemental avec du SDL, la synchronisation globale du système consiste à synchroniser les événements échangés entre les deux modules. Par contre, si la cosimulation doit vérifier le bon fonctionnement d'un processeur et d'une partie matérielle (ASIC), la vérification doit se faire à plusieurs niveaux. La vérification par cosimulation du système est généralement utilisé dans un premier temps, pour synchroniser l'échange des données sans temps de propagation des données. Puis dans un second temps, elle est utilisée pour vérifier le système complet à bas niveau en synchronisant les données avec la notion de temps.

3.5.2.1 Synchronisation Non Temporisée

La cosimulation ne fait en aucun cas intervenir la notion de temps. Les données sont échangées dans l'ordre temporel de leur modification.

[16]Cette technique est aussi appelé "polling" en anglais.



$$(TsSDL2 - TsSDL1) \neq (TsML2 - TsML1)$$

FIG. 3.15 – Illustration d'un modèle SDL-MATLAB en boucle ouverte

Dans ce cas de simulation, la notion de temps peut tout de même être présente localement, la nuance est que ces temps ne sont absolument pas partagés entre les modules cosimulés : le temps reste interne à chaque simulateur. Prenons le cas d'un système mécatronique en boucle ouverte, comme l'étude de l'asservissement d'un moteur face à une consigne envoyée par un modèle SDL. Le modèle à simuler est présenté en figure 3.15. Le moteur est modélisé par MATLAB ainsi que l'asservissement. Dans ce cas, la consigne reçue par MATLAB et provenant du SDL agit sur l'asservissement qui est représenté en temps réel dans l'environnement MATLAB. Par contre, au niveau système, la cosimulation reste fonctionnelle puisqu'elle ne prend pas en compte la durée du traitement SDL et le temps de propagation de la communication entre les deux modèles simulés.

La cosimulation à un niveau non temporisée consomme peu de ressources système. Il s'agit d'une simulation uniquement comportementale où, donc, le temps des sous-systèmes n'est pas pris en compte pour la cosimulation du système global. La fonctionnalité est respectée si la description prend soin de synchroniser les échanges de données par des protocoles de communication, par exemple asynchrones (cf. protocole "rendez-vous").

Elle permet, néanmoins, de vérifier que les interconnexions inter-systèmes sont respectées et permet de valider de façon rapide les protocoles de communication définis dans les différents modèles simulés. Enfin, si au niveau du système cosimulé, il n'existe pas de boucle entre les sous-systèmes, ce type de cosimulation peut suffire. Pour le niveau d'abstraction proche du prototypage, la validation temporelle est nécessaire afin de détourner les aléas de la validation fonctionnelle. Ainsi, un modèle de synchronisation bas niveau doit exister.

3.5.2.2 Synchronisation Temporelle

Située à un niveau plus proche de la réalisation physique du système, la synchronisation à bas niveau d'abstraction doit intégrer la gestion du temps et permettre d'obtenir une simulation en temps-réel. Ce type de simulation nécessite une compatibilité au niveau des simulateurs mis en jeu. Il doit être possible d'extraire la notion de temps de simulation pour chaque type de simulateur introduit dans l'environnement de cosimulation. Dès lors, on peut synchroniser les échanges de résultats des simulateurs de façon temporelle. Néanmoins, il faut rajouter un modèle de synchronisation de temps au niveau du routeur.

Il existe des méthodes performantes de synchronisation des temps. Mais leur implémentation est compliquée car elle impose de nombreuses contraintes au niveau des simulateurs (e.g.

retour en arrière[17]). Il faudrait que le processus de gestion des données puisse agir sur l'avancement de la simulation par chaque simulateur et les faire recalculer si nécessaire. Dans notre cas, il est préférable d'utiliser le modèle de la barrière temporelle (cf. §2.4.3). Ce modèle, quoique plus lent, a la propriété de ne pas étaler les traces lors de la simulation. Il est simple à mettre en oeuvre et n'impose qu'une contrainte : la fourniture d'une notion temporelle par les simulateurs.

Néanmoins, un problème provenant de la partie logicielle subsiste, puisque celle-ci ne contient pas de notion temporelle concrète. Cet aspect ne doit pas être négligé car c'est la partie la plus utilisée dans les systèmes embarqués du fait de ses propriétés de maintenance et de paramétrage. Pour pallier à cela deux possibilités s'offrent à nous :

- soit on réalise l'intégration du temps dans le code C sous forme d'annotation du langage source.[18] Celle-ci présente l'avantage de conserver une rapidité conséquente d'exécution du code à simuler. Cependant elle ne permet pas d'obtenir une très bonne granularité de temps.
- soit on a besoin d'un modèle d'implémentation pour le logiciel. Cette méthode consiste à réaliser un modèle du processeur sur lequel le code est ciblé pour y être exécuté. Cette exécution peut se faire pas à pas selon une granularité très fine (par instructions ou par cycles d'horloge[19]). Néanmoins, les comportements des divers processeurs (microprocesseurs et microcontrôleurs) existants sont très variés. La fabrication d'un modèle processeur nécessite un grand investissement et c'est pourquoi, on ne peut effectuer ce portage logiciel qu'au cas par cas, avec une forte intervention humaine.

A un niveau intermédiaire de la simulation temporelle, on peut éventuellement se contenter d'une annotation sommaire de la partie logicielle. Par contre, pour modéliser les parties logicielles au plus bas niveau, des modèles de processeurs sont indispensables. Nous avons donc utilisé pour nos applications des modèles de processeurs C167,[20] ST10, 8051 et 68000 des entreprises Siemens, ST-Microelectronics, Intel et Motorola, respectivement. Pour permettre la cosimulation à très bas niveau, nous avons opté pour l'utilisation de la barrière temporelle comme modèle de synchronisation.

Implémentation du Modèle de la Barrière Temporelle

La synchronisation par modèle "*lock step*" (fig. 3.16) est réalisée par une consigne de temps envoyée par le routeur. Le routeur est maître de l'avancement du temps des simulateurs, c'est pourquoi le temps d'avancement est d'abord évalué en fonction des applications mises en oeuvre et de la granularité de simulation exigée. Le temps d'avancement est fixé par l'utilisateur dans le fichier de coordination par une propriété "**TIMESTEP <Valeur du pas global>**". Les mémoires partagées sont étendues et contiennent désormais, en plus des informations sur les ports à cosimuler, un champ qui définit le temps imposé par le routeur et le temps courant du simulateur (cf. Entête de la mémoire partagée sur la figure 3.7). Les méthodes de synchronisation

[17] "rollback" en anglais : consiste à revenir en arrière dans la simulation. La plupart des simulateurs n'offrent pas cette possibilité.

[18] L'opération d'annotation consiste à inscrire des informations temporelles dans le code exécuté qui sont interprétées uniquement par le simulateur.

[19] En anglais "instruction accurate", "cycle accurate" voire même "pin accurate" pour une précision au niveau des broches du processeur.

[20] Le modèle C167 a été utilisé lors de la collaboration avec PSA, et est un modèle VHDL au niveau cycle de chez Synopsys.

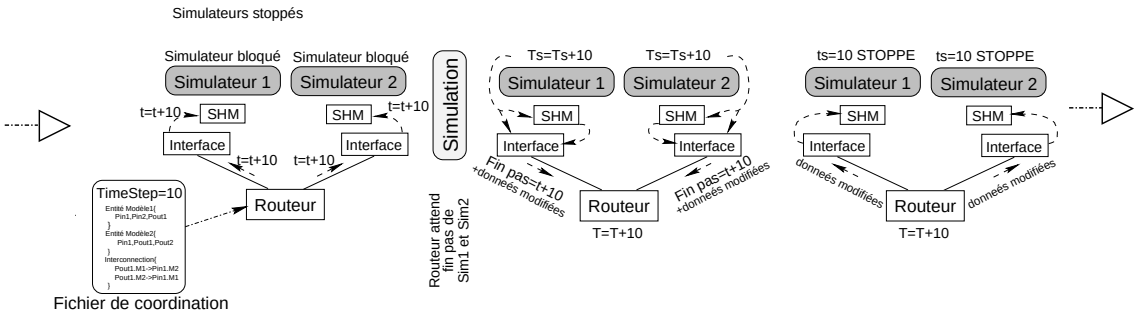


FIG. 3.16 – Schéma de fonctionnement du modèle de la barrière temporelle dans MCI

changent quelque peu et deviennent plus simples.

Au niveau de la synchronisation entre les simulateurs et leurs interfaces l'algorithme devient :

Côté simulateur (solution sans sémaphore)

SI (TLOCAL<=TIMPOSE) ALORS
  CONTINUER SIMULATION
  SIMULATION=1 ;
SINON (TLOCAL>TIMPOSE) ALORS
  REAPPEL À T=T+0
  SIMULATION=0
FIN SI
SI (SIMULATION=0) ALORS
  DÉBLOQUE SÉMAPHORE INTERFACE

Côté simulateur (solution avec sémaphore)

SI (TLOCAL<=TIMPOSE) ALORS
  CONTINUER SIMULATION
SINON (TLOCAL>TIMPOSE) ALORS
  DÉBLOQUE SÉMAPHORE INTERFACE
  BLOQUE SÉMAPHORE SIMULATEUR
FIN SI

Côté interface (réception)

ATTENDRE REQUÊTE DE LA SOCKET
SI (ORDRE = DONNÉE)
  METTRE A JOUR DONNÉE
SI (ORDRE = PAS SUIVANT)
  METTRE A JOUR TIMPOSE

Côté interface (envoi)

TANT QUE VRAI FAIRE
  BLOQUE LE SÉMAPHORE INTERFACE
  ENVOYER VIA LA SOCKET LES DONNÉES MODIFIÉES
  ENVOYER ORDRE FIN DE PAS
  SI (SOLUTION ECONOME) DEBLOQUER SÉMAPHORE SIMULATEUR
FIN TANT QUE

TLOCAL : Temps du simulateur pour un module en local.

TGLOBAL : Temps Imposé par le routeur. C'est le temps du pas global de simulation additionné de la valeur du "*TimeStep*" à chaque itération.

La méthode sans sémaphores s'applique aux simulateurs graphiques et présente l'avantage de permettre une simulation en temps-réel tout en maintenant une maîtrise de l'interface graphique. La solution avec sémaphores est plus économique en temps machine. Cependant, elle implique un blocage partiel des interfaces graphiques utilisées par les simulateurs. Ce blocage intervient entre le moment où le simulateur a fini son pas global et le moment où tous les simulateurs ont fini leur pas. Si les simulateurs ont une vitesse de traitement du pas de calcul identique, cette solution peut alors s'appliquer (extrêmement rare). Le fait de bloquer le processus par sémaphore présente l'avantage d'éviter la consommation inutile de ressources processeur. Par contre, du fait de l'arrêt total du processus, ceci provoque un saccadement de l'interface graphique. Ce désavantage aboutit trop souvent à une impossibilité d'utiliser l'interface graphique

et de ce fait, la convivialité de déboggage n'est pas maintenue.

Quant au routeur, il reçoit à la fois les données de ses interfaces, mais aussi les ordres pour effectuer de nouveaux pas. L'algorithme au niveau du routeur devient le suivant :

Synchronisation au niveau du routeur

```
TANT QUE VRAI FAIRE
   TGLOBAL=TGLOBAL+ PAS DE CALCUL
   TANT QUE VRAI FAIRE
      POUR CHAQUE INTERFACE FAIRE
         ENVOIT CONSIGNE TEMPS GLOBAL
      FIN POUR
      POUR CHAQUE INTERFACE FAIRE
         TANT QUE (INFORMATION <> FIN DE PAS) FAIRE
            ATTENDRE INFORMATION SOCKET
              SI DONNEE PRÉSENTE DANS LA SOCKET ALORS TRAITER LA DONNEE
              SI (FIN DE PAS) PASSER A L'INTERFACE SUIVANTE
         FIN TANT QUE
      FIN POUR
   FIN TANT QUE
FIN TANT QUE
```

Le routeur envoie les temps imposés à chacun de ses modules. Les simulateurs effectuent leur simulation, entre deux pas de calcul. Les données modifiées par les simulateurs entre deux pas, ne sont pas directement transmises par les interfaces vers le routeur. Lorsque les simulateurs terminent leur pas, ils réveillent l'interface qui leur associée. Durant le pas, cette interface communique au routeur l'ensemble des données et l'ordre de fin de ce pas. Le routeur attend cet ordre de fin de calcul qui doit provenir de chacune des interfaces. Une fois l'ensemble des acquittements de fin de pas reçu, il traite l'ensemble des données en respectant la "*netlist*" d'interconnexions. Il expédie les données traitées aux différents simulateurs via les interfaces et leur envoie l'ordre de calculer un nouveau pas (à Temps Global + valeur du "*TimeStep*"). De nouveau, le routeur se met en attente de la fin de pas des simulateurs, etc...

3.6 Conclusion

L'outil MCI a pour objectif d'apporter aux concepteurs un environnement de cosimulation multilangage qui leur permet de valider leurs systèmes tout le long du flot de conception. Il doit aussi être très flexible et s'adapter à divers domaines d'applications utilisant une pléthoria de langages distincts. Pour pouvoir accepter les contraintes multiples de la validation multilangage, MCI a été conçu de façon générique et suffisamment modulaire pour pouvoir accepter les divers langages utilisés par les concepteurs de systèmes.

L'interface de cosimulation réalisée, est basée sur une communication à base de "*socket*" permettant ainsi une cosimulation distante. L'intégration d'un langage se fait par l'utilisation de ports d'entrées-sorties dont l'objectif est de permettre au modèle simulé d'échanger ses données vers le bus de cosimulation. Un fichier de coordination permet de décrire l'ensemble du système à cosimuler. Il décrit la liste des simulateurs et leurs ports ainsi que leurs interconnexions. Il sert à générer les ports d'entrées-sorties pour chaque simulateur avant la cosimulation et à échanger les données durant la simulation. La cosimulation multiniveaux se répartit en deux manières

d'échanger les données : la première est la cosimulation fonctionnelle qui permet une validation de l'interconnexion des sous systèmes entre eux, en vérifiant le bon échange des données calculées entre les différents simulateurs. MCI utilise, pour cela, des méthodes de synchronisation qui adaptent automatiquement le transport des données en fonction des caractéristiques des langages et du mode de simulation exigé. La seconde est la cosimulation temporelle qui permet d'ajouter à la cosimulation fonctionnelle une synchronisation en temps des différents simulateurs. Néanmoins, elle implique une sélectivité dans les logiciels à utiliser, car tous les simulateurs n'ont pas cette représentation du temps simulé.

Enfin, l'outil MCI a fait l'objet d'un transfert technologique en janvier 1999 entre le laboratoire TIMA représenté par l'INPG et la société S.A. AREXSYS à Meylan. L'outil MCI est actuellement, au sein de cette société, un outil de cosimulation à part entière commercialisé sous le nom de CosiMate^{TM}. L'outil commercial a beaucoup évolué, mais la méthodologie demeure la même à quelques améliorations. CosiMate apporte des possibilités supplémentaires par rapport à MCI. Il est plus souple d'utilisation, plus convivial, et plus rapide. Néanmoins, il s'oriente vers la cosimulation multilangage à haut niveau d'abstraction. Il a aussi gardé les possibilités de vérification des systèmes obtenus à partir d'outil de co-conception puisque actuellement, il sert d'outil de validation pour la sortie de l'outil de co-conception ArchiMate^{TM}.

Chapitre 4

RÉSULTATS - EVALUATION

4.1 Résultats Expérimentaux : Applications de la Cosimulation

La manière la plus concrète de valider un outil ainsi que sa méthodologie est de le submerger d'exemples d'horizons divers. Il est indéniable que les différents exemples testés sur l'outil MCI ont permis de le faire considérablement évoluer, de montrer ses points faibles et d'introduire des éléments sur les évolutions possibles. L'outil et la méthodologie ont été testés en premier lieu grâce à la collaboration de PSA. Cette collaboration nous a permis d'évaluer les besoins d'un outil de cosimulation et les problèmes à résoudre pour qu'il soit exploitable.

4.2 Applications Mécatroniques (PSA : Peugeot - Citroën)

Au sein de PSA, plusieurs exemples ont été testés et simulés. Cette collaboration nous a permis d'évaluer l'enjeu de la cosimulation sur des projets industriels. Dans le domaine automobile, la conception des systèmes mécatroniques nécessite la création en parallèle de composants logiciels, matériels et mécaniques. Dans les approches de conceptions traditionnelles, les différentes parties sont modélisées par des groupes distincts et l'intégration du système complet est réalisée à la fin, lors de l'implémentation sur prototype. Ce schéma de conception introduit des retards et des coûts dûs à l'introduction de problèmes de communication, de disproportions des temps de réalisation, de non respect du cahier des charges, etc ... Nous avons donc proposé notre approche pour tenter de réduire le temps de mise sur le marché en réalisant une validation plus rapide du système complet. Cette validation est la cosimulation des différents modèles constituant le système.

Dans ces projets, les différentes cosimulations mises en oeuvre sont constituées de modules C-VHDL-MATLAB. Les parties logicielles ont été modélisées en C, les parties matérielles en VHDL, et les parties mécaniques en MATLAB-SIMULINK. Dans ces exemples d'applications, nous avons pu expérimenter deux types de cosimulation à niveaux d'abstraction différents : d'une part la cosimulation non-temporelle et d'autre part la cosimulation temporelle [MVHJ98]. La dernière est utilisée pour valider le système par simulation temporelle afin d'évaluer le comportement temps-réel du système à concevoir.

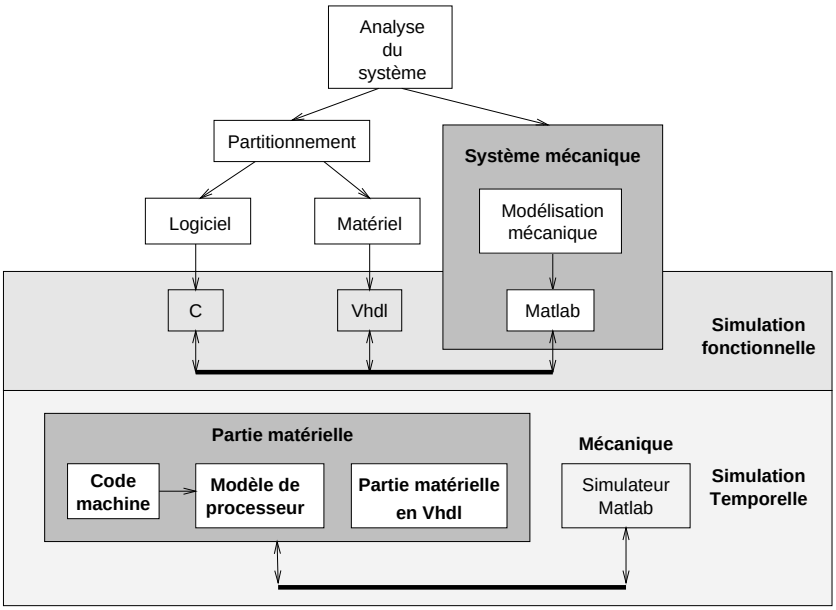


FIG. 4.1 – Méthodologie de simulation

4.2.1 L'approche Mécatronique

La plupart des applications mécatroniques impliquent trois types de composants : le matériel, le logiciel, et la partie mécanique. Lorsque différents langages interviennent pour la spécification des différentes parties, la partie validation du système implique l'utilisation des techniques de cosimulation. On différencie deux principaux types de cosimulation : la cosimulation fonctionnelle pour une vérification du respect de la majorité du cahier des charges. En revanche, la cosimulation temporelle est utilisée pour une vérification temps-réel du système virtuel C-VHDL-MATLAB réalisée à partir de la spécification du système.

La cosimulation fonctionnelle vérifie le bon comportement du système, établit la cohérence des interconnexions des sous-systèmes entre eux. Les échanges des données entre simulateurs sont contrôlés par des événements. Dans ce cas seulement, l'ordonnancement des données est réalisé sans la notion de synchronisation de temps des simulateurs entre eux.

Dans le cas de la cosimulation temporelle, les données sont échangées dans des fenêtres temporelles de simulation. En effet, les simulateurs sont synchronisés sur le temps global du système. Cette simulation est appelée temps-réel simulé.

L'approche que nous avons utilisée chez PSA est basée sur l'environnement de cosimulation VCI [VRD+97]. VCI permet de faire fonctionner concurrentiellement plusieurs simulateurs en utilisant les IPCs d'Unix pour modéliser le bus de communication entre ces différents simulateurs. Nous avons, pour nos besoins, modifié VCI pour qu'il nous permette à la fois de réaliser une cosimulation temporelle et une cosimulation fonctionnelle.

La figure 4.1 représente la méthodologie utilisée pour produire et valider la spécification initiale pour une application automobile comprenant une partie logicielle, matérielle et mécanique. Globalement, la conception démarre toujours par l'analyse des besoins du système à concevoir. La définition des différentes fonctionnalités du système est décrite à haut niveau (cf. langage SDL, SA/RT). Ensuite, les concepteurs réalisent un partitionnement manuel en séparant la partie mécanique de la partie électronique (logiciel et matérielle). La partie mécanique

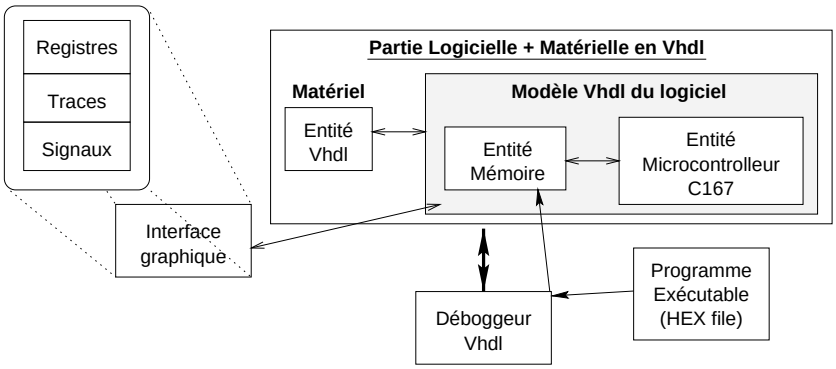


FIG. 4.2 – Modélisation de la partie électronique

est modélisée par une équipe spécialisée qui décrit le sous-système avec MATLAB. La partie électronique est enfin séparée et modélisée en VHDL pour le matériel et en C pour le logiciel. Dans les applications mécatroniques le logiciel est beaucoup plus conséquent que la partie matérielle (parfois inexistante). En effet, le logiciel est couramment utilisé car il permet de redéfinir des paramètres du sous-système logiciel sans pour autant être obligé d'effectuer, comme pour la partie matérielle, une étape de prototypage.

Durant toutes ces étapes de conception, la cosimulation doit être utilisée pour valider les différents sous-systèmes à différents niveaux d'abstraction.

Partant d'une spécification C-VHDL-MATLAB, le processus de conception inclut principalement deux étapes de validation qui doivent être réalisées avant l'implémentation. La première cosimulation à réaliser est la validation fonctionnelle du système alors que la seconde étape est la vérification temporelle du système. Pour mener à bien ce type de cosimulation nous avons introduit un modèle au niveau cycle du microcontrôleur cible de chez PSA . Celui-ci est implémenté au final sur la voiture. Ce modèle est nécessaire pour émuler l'exécution du logiciel au niveau du cycle d'horloge. La cosimulation devient une simulation au niveau cycle d'horloge. Le modèle utilisé dans le cadre de ces projets est un modèle VHDL du microcontrôleur C167 de chez Siemens [Gro95]. Ce modèle est composé à la fois d'une entité processeur et d'une mémoire utilisée en grande partie pour stocker le programme exécutable obtenu avec un compilateur C pour C167. L'organisation matérielle du modèle processeur est représentée sur la figure 4.2. Dans ce cas de figure la cosimulation C-VHDL-MATLAB se transforme en une cosimulation VHDL-MATLAB. La cosimulation utilise les déboggueurs MATLAB (SIMULINK) et VSS-Debugger (vhdldbx de Synopsys). Le modèle de processeur ouvre une interface graphique auxiliaire qui décrit l'ensemble du processeur (Registres, traces, E/S, etc...). Il est, de plus, possible de modifier ces éléments et de contrôler les signaux externes et internes du processeur à l'intérieur du visualiseur de traces du déboggueur Synopsys. L'évolution des calculs du simulateur MATLAB est synchronisée sur un multiple de l'horloge cadençant le VHDL. Ce pas de calcul est donc global au système et doit être relativement petit pour obtenir un grain de simulation acceptable pour la partie mécanique.

La configuration de l'exécution d'une telle cosimulation est représentée sur la figure 4.3. L'environnement débute par l'exécution d'un script qui se charge d'exécuter un programme de contrôle de MATLAB (programme développé autour de la bibliothèque "*Engine*"[1]). Ce script

[1]Les fonctions "Engine" fournit par Mathworks permettent de commander le simulateur matlab par un l'intermédiaire d'un programme C.

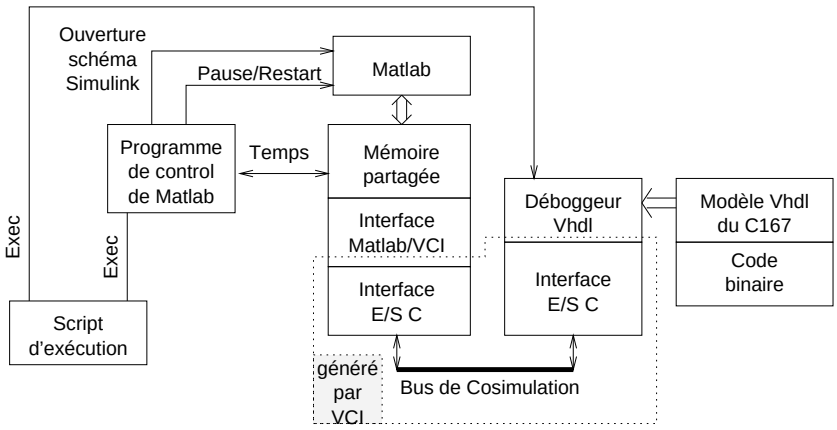


FIG. 4.3 – Configuration de la cosimulation synchronisé implémentée

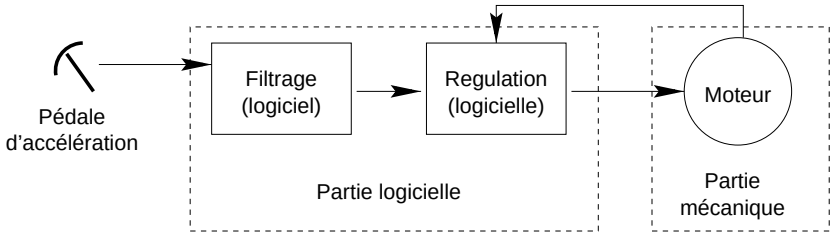


FIG. 4.4 – Application de régulation de la pédale d'accélération

exécute ensuite le déboggueur VHDL. Ce dernier intègre automatiquement, à sa simulation, le modèle du processeur qui, à son tour, charge le code binaire au format "*Intel-HEX*" à l'intérieur de sa mémoire. Le programme de contrôle de MATLAB utilise l'API "*Engine*" de MATLAB pour respectivement exécuter la session Simulink du modèle MATLAB, qui doit être simulé, et communiquer avec le moteur de simulation MATLAB. La communication via les fonctions "*Engine*", permet de lire la position en temps de la simulation MATLAB et de commander cette simulation suivant le temps de l'environnement de cosimulation. D'un côté le temps est extrait du simulateur MATLAB et de l'ordre, il est imposé par l'intermédiaire des interfaces E/S générés par VCI. Le fonctionnement est celui de la barrière temporelle à ceci près que VCI a été étendu pour pouvoir manipuler les temps des différents simulateurs.

Deux applications principales ont été décrites suivant cette méthodologie sur le site de PSA. La première est le contrôle des variations parasites du mouvement de la pédale d'accélération sur un véhicule électrique (Figure 4.4). La deuxième a été la réalisation de la régulation d'une suspension hydraulique (suspension hydractive). Ces deux systèmes contiennent un modèle comportemental du véhicule ou de la partie mécanique à contrôler, en MATLAB. La partie logicielle comprend le modèle VHDL du C167 et son programme compilé, et éventuellement en plus une partie matérielle instanciée.

Dans l'exemple de la figure 4.4, le système régule les variations dûes au pied du chauffeur sur la pédale et les vibrations du moteur pour ne pas faire osciller la consigne du moteur. De ce fait, un filtre de Kalmann est utilisé afin d'éliminer les vibrations parasites engendrées. La partie contrôle et régulation de l'application est réalisée par une carte électronique contenant une partie matérielle et logicielle modulables (l'utilisateur a le libre choix d'implémenter ce filtre soit en

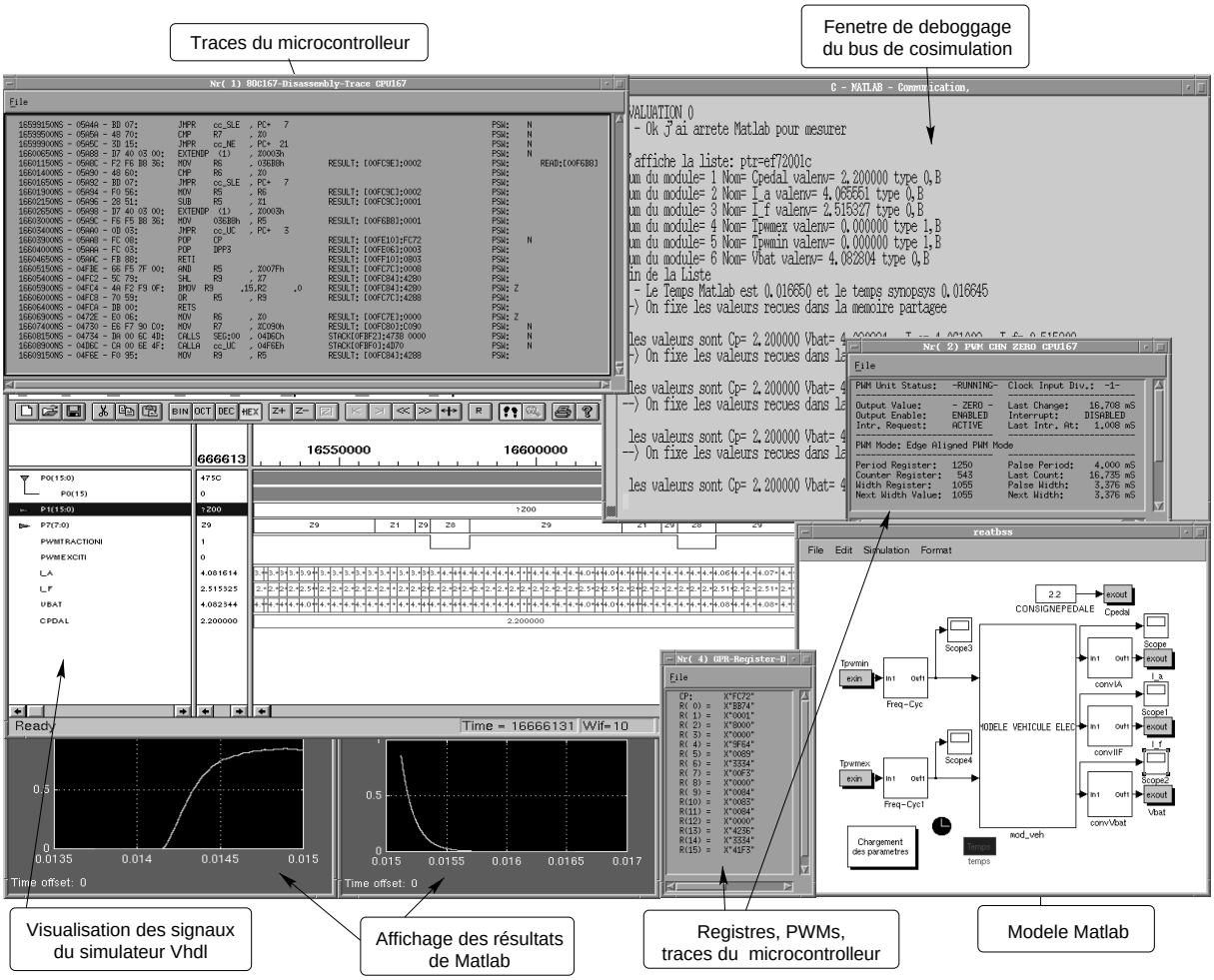


FIG. 4.5 – Simulation temporelle de l'application de régulation des parasites

matériel soit en logiciel). La partie comportement véhicule, celle du moteur en particulier, est modélisée par un outil de modélisation mécanique (MATLAB). La figure représente la copie d'écran de la cosimulation temporelle effectuée sur cet exemple. Sur la copie d'écran on peut voir à la fois le déboggueur VHDL et son visualiseur de traces, et le modèle C167 exécutant le logiciel compilé. De plus, l'environnement MATLAB/SIMULINK simule la partie mécanique et il est donc possible de modifier un paramètre du modèle durant la simulation.

4.2.2 Résultats de l'Expérience PSA

L'expérience de PSA, nous a permis de cibler les besoins et de comprendre la faisabilité de la cosimulation sur des applications industrielles.

La cosimulation fonctionnelle est très rapide et facile à mettre en oeuvre sans posséder une complète connaissance des composantes constituant le système. Dans les deux applications cette étape de validation, qui intervient à haut niveau du raffinement de la spécification, reste une bonne solution pour valider le comportement du système tout en conservant le contrôle des interconnexions entre les différents sous-systèmes. Evidemment, nous avons plus de difficultés en ce qui concerne la cosimulation temporelle. Malgré l'aboutissement de la simulation de nos

deux exemples principaux, la constatation est que le temps de ces cosimulations est trop important. Dans notre cas précis, la vitesse de simulation est imposée par le simulateur le plus lent, c'est à dire le modèle de processeur en VHDL. Ce modèle ne permet que de simuler quelques dizaines d'instructions assembleurs par seconde sur une Ultra-Sparc, ce qui rend l'utilisation de la simulation impossible au niveau industriel. La simulation d'une milliseconde de temps réel demande approximativement 15 minutes de temps machine. Dans une application mécatronique, comme une régulation de suspension, la simulation prend un sens lorsqu'elle s'est produite sur plusieurs minutes. Par ailleurs, bien souvent le processeur contient une étape d'initialisation qui dure quelques dizaines de millisecondes, voire plus. Ce temps représente une perte de temps simulé et ne profite en rien aux résultats de la simulation. Il est donc nécessaire de miser sur la vitesse de simulation. Par la suite, nous avons orienté et testé le même type de simulation avec l'utilisation d'un modèle C des processeurs. Ce qui nous a permis d'atteindre 1 million d'instructions par seconde pour la partie logicielle.

4.3 Applications Résultant de MUSIC : Outil de Co-conception

De nombreuses applications ont résulté de l'outil de co-conception MUSIC. Les exemples ont tout d'abord été des tests croisés permettant de valider les deux outils à la fois. Ensuite, des exemples multilangages plus conséquents ont été cosimulés. Au fur et à mesure de l'évolution de l'outil, des cosimulations de plus en plus complexes, au niveau temps et débit d'échanges, ont été réalisées. Par la suite, d'autres projets sont nés dans le laboratoire comme par exemple la modélisation en C d'un microcontrôlleur ST10 (ST-Microelectronics), puis d'un microcontrôleur 8051 (Intel) et même d'un processeur 68000 (Motorola). Ces modèles de processeurs ont servi à construire des cosimulations temporisées de plus grande rapidité, afin d'améliorer les performances de la simulation temporelle.

Dans cette partie nous présentons une application de la méthodologie multilangage sur la base d'un exemple de régulation de moteurs.

4.3.1 Description du Flot de Conception MUSIC

L'outil de co-conception MUSIC offre une méthodologie de conception et de synthèse de haut niveau des systèmes mixtes logiciels/matériels. Il a été étendu pour accepter, en entrée de son flot de conception, des systèmes décrits à la fois en langage de spécification SDL et avec son environnement. L'environnement peut-être spécifié dans d'autres langages qui sont adaptés à leur domaine respectif de modélisation (e.g. COSSAP pour le traitement du signal, MATLAB pour les systèmes continus).

Le flot de conception de l'outil MUSIC, représenté sur la figure 4.6), est composé de quatre étapes principales :

1. La capture de la spécification du système. Elle est réalisée à très haut niveau d'abstraction en SDL. Une spécification SDL d'un système, représentée sur la figure 4.7, le décrit à travers un nombre de sous-systèmes appelés blocs reliés par des canaux de communication. Les blocs sont décomposés en processus concurrents échangeant des données à travers des acheminements de message. Chaque processus décrit un automate d'états finis dont les transitions peuvent être actives sur des signaux.
   La partie environnementale est développée séparément en utilisant les outils appropriés.

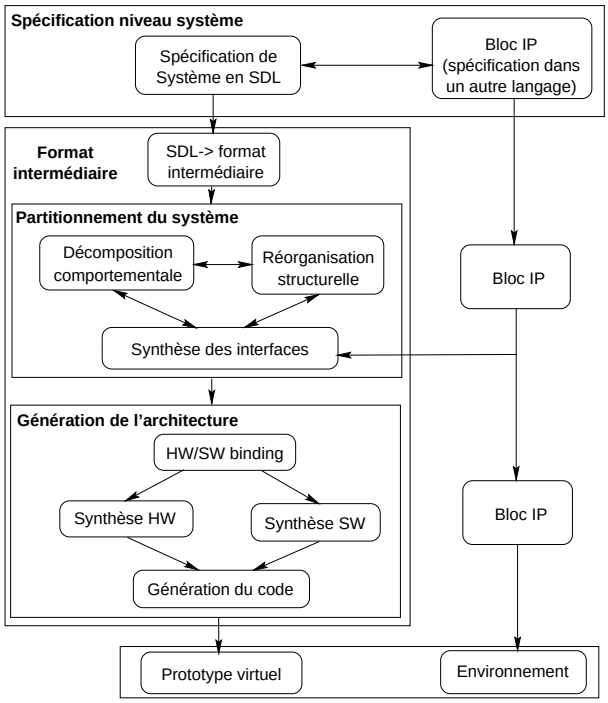


FIG. 4.6 – Flot de conception avec MUSIC

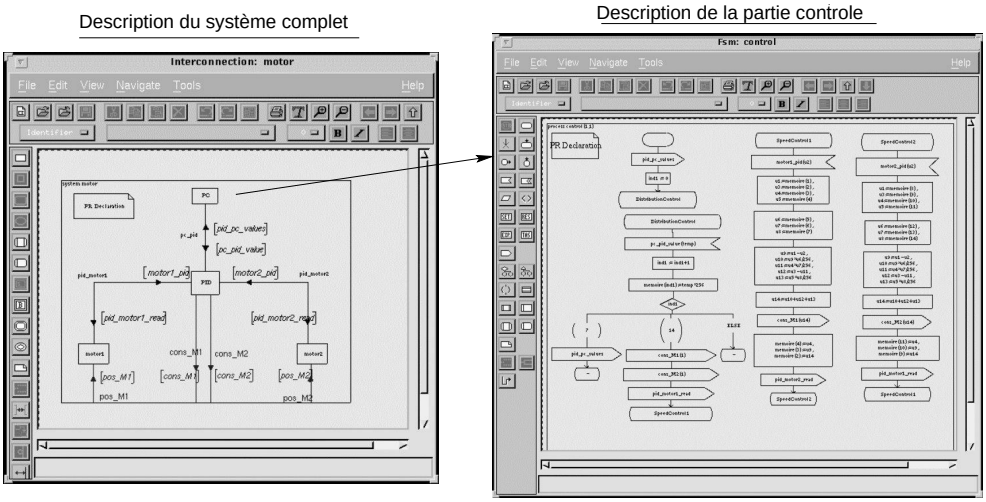


FIG. 4.7 – Description SDL de l'exemple du contrôleur

2. La traduction de la spécification SDL en langage intermédiaire SOLAR. Ce langage intermédiaire sert pour toutes les transformations qui suivent. Il est utilisé pour réaliser tous les raffinements, les ordonnancements, etc...
3. Le partitionnement du système. Il définit les raffinements structurels des différents blocs architecturaux.
4. La synthèse de communication. Cette étape importante définit la communication entre les langages. A ce stade, la communication interlangage (C, VHDL, MATLAB par exemple) est décrite à un niveau où tous les protocoles sont explicites à l'intérieur des modèles.. Une cosimulation doit être réalisée pour valider le partitionnement et les protocoles de communication. Des vérifications temporelles peuvent également être réalisées à ce niveau.
5. La génération de l'architecture. Cette étape consiste à désigner[2] les blocs (logiciel et matériel), puis à opérer à une étape de réordonnancement comportemental des états finis du système pour correctement générer les architectures.
6. La réalisation du prototype.

Un système décrit en SDL peut donc générer un système C-VHDL(RTL)-(Environnement) à travers l'outil MUSIC.

Dans cette approche de co-conception il est toutefois nécessaire de vérifier le comportement du système complet aux différentes phases de la co-conception. Dans une telle approche où les différents modules sont conçus avec des langages différents, la cosimulation devient l'outil nécessaire pour ce type de validation. Pour les différentes étapes de conception d'un système multilangage, nous avons donc utilisé MCI pour effectuer la validation. Nous avons également utilisé MCI pour le prototype virtuel multilangage généré par MUSIC. Nous présentons ici les différentes étapes de validation sur un exemple d'application simple : un contrôleur de moteurs.

4.3.2 Application : Le Contrôleur de Moteurs

Cette partie montre les résultats d'une application de co-conception multilangage dans le cas d'un modèle multilangage. L'application concerne le contrôle d'un bras de robot. Ce système peut être divisé en deux parties : les moteurs commandant le bras du robot et le contrôleur. Le langage SDL est utilisé pour modéliser le contrôleur alors que MATLAB, ou COSSAP dans une autre configuration, modélisent le comportement physique des moteurs. Le contrôleur du bras du robot peut ajuster les paramètres de vitesse et de position de deux moteurs. Quatre signaux sont échangés entre le modèle MATLAB et le modèle SDL (2 pour chaque moteurs). Le premier signal contrôle le moteur et le second lit la position courante du bras. Le schéma de base est représenté sur la figure 4.8a). Le bloc GENERATEUR calcule une trajectoire pour tous les moteurs commandant le bras du robot. Le bloc CAPTEUR reçoit les positions et les transmet au contrôleur. Le bloc contrôleur a pour rôle de donner les ordres aux moteurs seulement lorsque c'est possible.

La spécification SDL est composée de quatre blocs comprenant six processus : un générateur, un distributeur, un contrôleur par moteur et un échantillonneur par moteur. La figure 4.8b montre la structure des processus du système du contrôleur. Le générateur produit deux signaux, un ordre et une adresse. Le distributeur renvoie l'ordre au contrôleur du moteur désigné

[2]Etape de "binding".

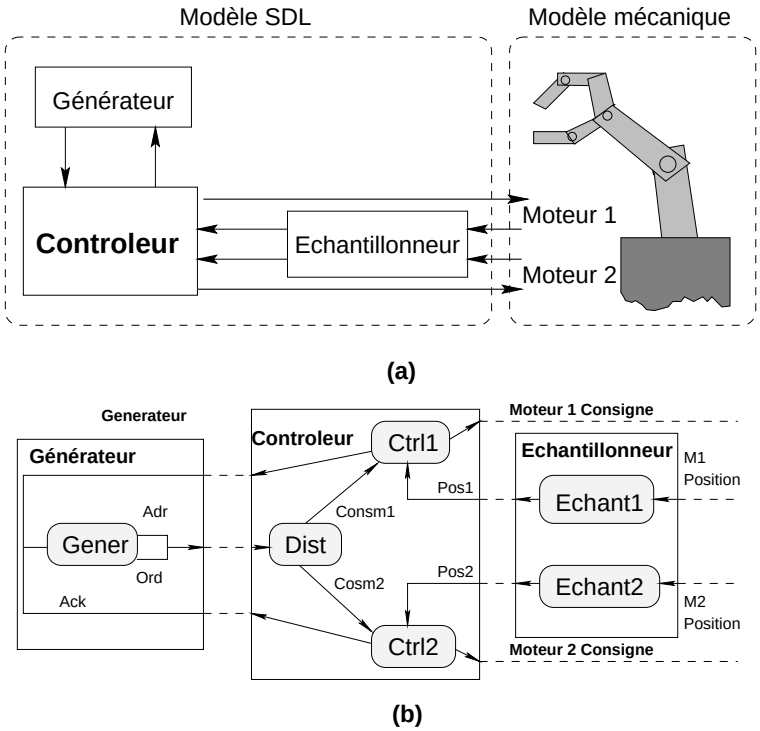


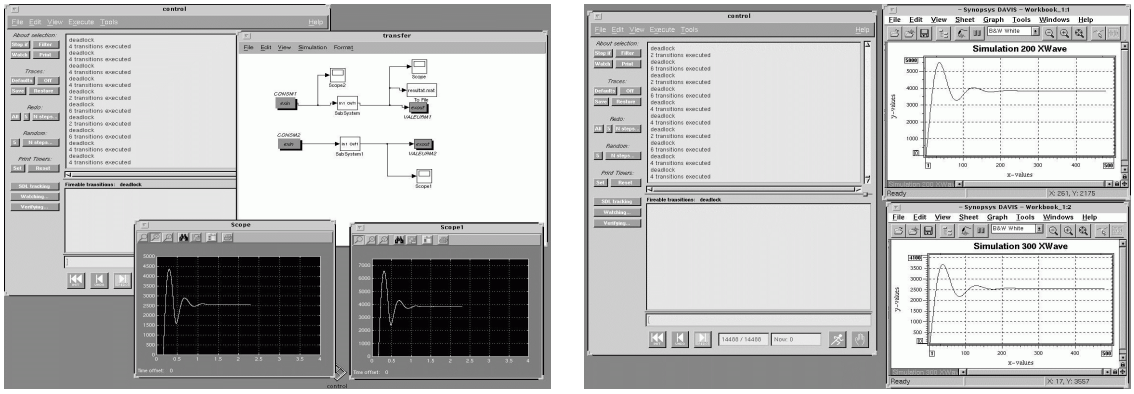
FIG. 4.8 – Application du contrôleur de moteurs

par l'adresse. Un processus de contrôle donne l'ordre au moteur et "*scan*" sa position. Quand un moteur est sur le point de réaliser son ordre, le processus de contrôle informe le générateur qu'il est disponible pour une nouvelle opération. Etant donné que la position est un signal continu, chaque processus échantillonneur se charge de digitaliser les signaux.

La modélisation du comportement physique des moteurs a été réalisée dans deux configurations différentes : l'une en MATLAB et l'autre en COSSAP.

Enfin, le système complet et ses interconnexions sont décrits dans un fichier de coordination qui sert d'entrée à MCI. A ce niveau le fichier est constitué de deux blocs interconnectés par des canaux abstraits. On utilise donc le format SOLAR pour décrire notre système à simuler. Pour les étapes suivantes l'utilisation de MUSIC permet de générer automatiquement le fichier de coordination pour pouvoir cosimuler avec MCI.

Dans un premier temps, il est intéressant de modéliser les moteurs en MATLAB ou en COSSAP et de garder la spécification SDL initiale. La première vérification consiste en la validation du système complet dont les sous-systèmes modélisant les moteurs sont réalisés en MATLAB ou en COSSAP. Les copies d'écran de la figure 4.9 représentent deux cosimulations de très haut niveau d'abstraction SDL-MATLAB sur la figure a et SDL-COSSAP sur la figure b. Ce type de modélisation et de cosimulation présente un sérieux avantage. MATLAB est très approprié pour simuler un moteur et offre donc toutes les possibilités pour modifier les paramètres au cours de la simulation. De plus, elle permet de séparer les moteurs de la spécification SDL, qui est introduite dans l'outil de co-conception. Avec ce type de cosimulation, on vérifie ainsi que la spécification réalisée à haut niveau est conforme au cahier des charges. Il est vrai que la modélisation et la simulation entière du système peut se faire directement en SDL, mais certains sous-systèmes comme la modélisation des moteurs est plus rapide à réaliser en MATLAB qu'en SDL. C'est pourquoi, le concepteur est amené à découper sa spécification avant de la modéliser.



(a) (b)

FIG. 4.9 – Cosimulations à haut niveau

Il sera beaucoup plus rapide de modéliser la partie mécanique (environnement extérieur) et le reste en SDL et de cosimuler l'ensemble du système, plutôt que de tout modéliser en SDL et de le simuler, pour ensuite le partitionner.

Dans ce type de simulation, le système est validé à très haut niveau avec son environnement extérieur avant d'être raffiné par un outil de co-conception.

Cosimulation Non Temporelle au Niveau Architecture

Le C généré par MUSIC est un C non dédié à un processeur. Le VHDL est un VHDL RTL qui peut directement passer à l'étape de synthèse. MUSIC génère également les modèles directement utilisables par l'outil de cosimulation et fournit un fichier SOLAR de coordination pour décrire les connexions des modules générés.

A ce nouveau niveau, la cosimulation C-VHDL-Environnement peut être exécutée (figure 4.10). Le modèle VHDL est simulé sur un déboggueur VHDL, l'environnement est toujours simulé par son ou ses simulateurs appropriés (MATLAB sur la figure 4.10), et le C sera exécuté sur la machine ou par l'intermédiaire d'un déboggueur pour analyser son exécution (l'outil DDD "Data Display Debugger" sur la figure). Si un modèle nécessite d'être simulé sur une autre machine, l'utilisateur doit tout simplement éditer ce fichier de coordination et y adjoindre le nom de la nouvelle machine désirée.

Le concepteur peut parfois avoir besoin de simuler son système avec la notion de temps sans pour autant utiliser un modèle du processeur. Il est possible à ce niveau d'obtenir la notion de temps par l'annotation temporelle approximative des temps d'exécution du langage C et par adjonction d'une propriété "*TIMESTEP*" dans le fichier de coordination. Les fichiers C sont recompilés en mode temps-réel, et l'outil de cosimulation synchronise les "temps semi-virtuels" des simulateurs entre eux. Le programme C généré par MUSIC est en fait annoté en temps à chaque passage dans la boucle de la machine d'état. Les simulateurs du modèle extérieur sont inclus dans cette synchronisation temporelle uniquement s'ils contiennent la gestion du temps à l'intérieur de leurs simulations (MATLAB par exemple). Ce type de cosimulation en mode annotation permet plus souvent d'ajuster les vitesses des simulateurs entre eux, plutôt que de permettre une vérification temporelle.

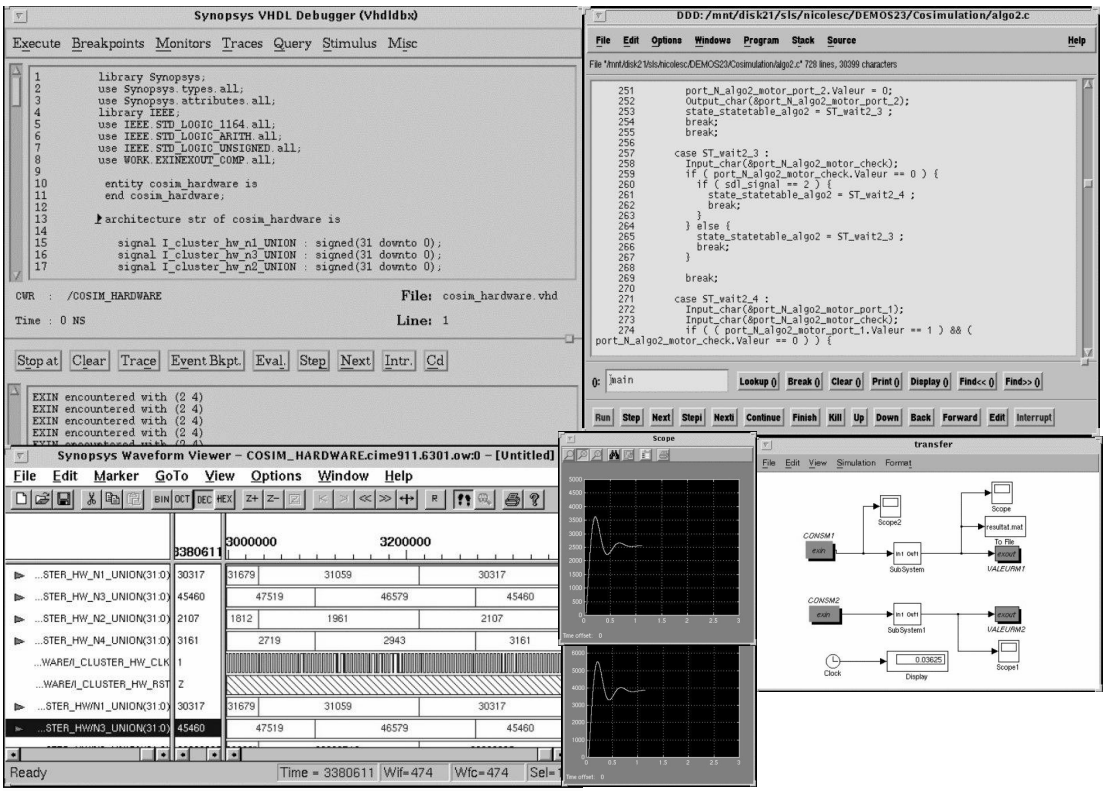


FIG. 4.10 – Cosimulation par exécution native

4.3.2.1 Cosimulation Temporelle au Niveau Cycle

Pour pouvoir à nouveau descendre dans les niveaux d'abstraction, l'introduction de modèles de processeur est indispensable. Le C généré est donc modifié suivant les caractéristiques du processeur et des ports dont il dispose pour échanger ses données. Si ces ports ne sont pas suffisants, il faut ajouter des interfaces, etc.. Le "*targeting*"[3] permet de correctement implanter le C généré dans un processeur. Le C obtenu est compilé par le compilateur associé au processeur et le code binaire obtenu est inscrit dans sa mémoire. La cosimulation synchronisée de l'ensemble processeur(s)-VHDL et de l'environnement extérieur est la dernière étape de validation avant l'implémentation du prototype virtuel.

Cette cosimulation est synchronisée en respectant le modèle de la barrière temporelle et permet de valider par simulation le système complet, à l'origine en SDL, à très bas niveau. La simulation de l'environnement extérieur est toujours réalisée avec le simulateur MATLAB. La figure 4.11) représente cette simulation avec un microcontrôleur (Intel 8051). De nombreuses configurations ont été réalisées autour du 8051 et/ou du ST10 et de nouveaux travaux interviennent également dans l'étude des performances en fonction du partitionnement des modules matériels et logiciels sur différents processeurs.

[3] L'étape de "targeting" consiste à cibler le code logiciel pour l'adapter à un processeur donné.

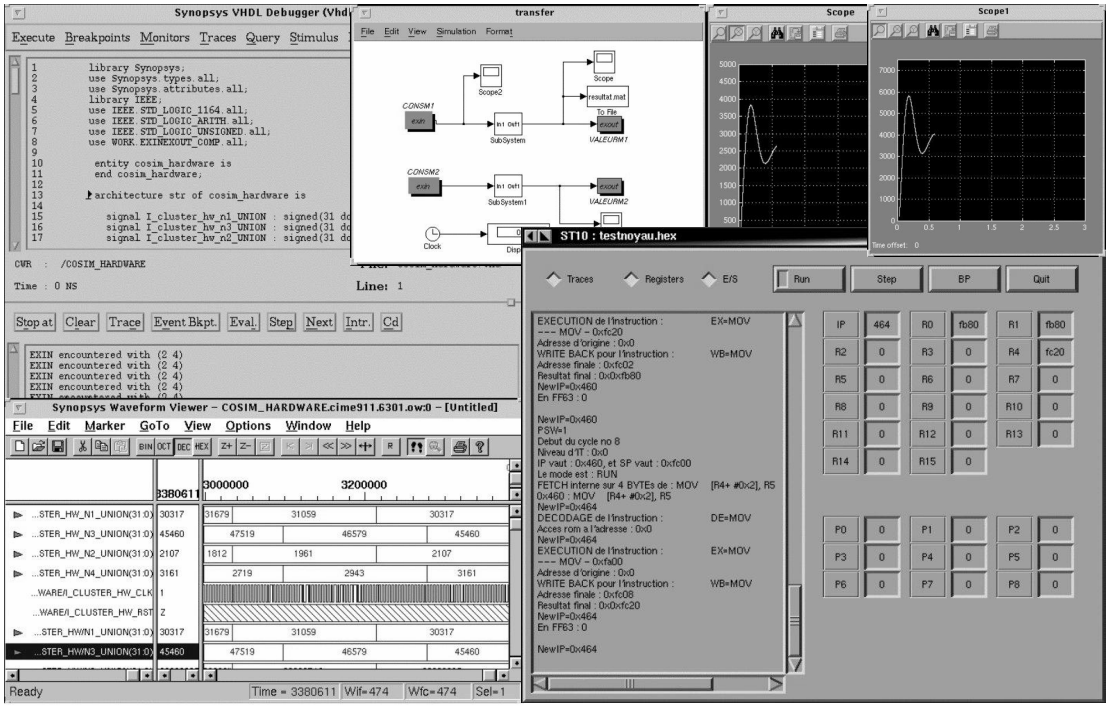


FIG. 4.11 – Simulation au niveau cycle d'horloge

	même machine		Machine différente	
	C-C	Proc-Proc	C-C	Proc-Proc
Temps d'un cycle (ms)	39.9	43.2	68.3	70.7
Nb Cycle/s	25	23	15	14

TAB. 4.1 – Temps de cosimulation

4.3.3 Résultats de la Cosimulation

Les applications de cosimulation à la fois temporelles et non-temporelles ont été nombreuses. Différentes configurations possibles ont été testées et ceci à tous les niveaux. Elles ont permis à la fois de consolider l'outil de cosimulation et de vérifier ses performances. Dans le cadre des estimations de performances, le tableau 4.1 illustre les disproportions des temps d'exécution en fonction du niveau d'abstraction utilisé et du type de support de communication emprunté par l'environnement de cosimulation. Les différences de performances entre les types de simulation sur une seule machine et sur des machines différentes sont significatives. Il est donc plus convenable de faire fonctionner les applications, qui demandent beaucoup de ressources, sur la même machine que le routeur. Ceci diminue les temps de communication comparativement aux performances du réseau (ici $\sim$40% de gain). Ces chiffres prouvent aussi que des améliorations de performances au niveau communication (choix d'un nouveau support ou mode de communication) sont envisageables. On remarque aussi que la valeur du nombre de cycles simulés par seconde est équivalente aussi bien à l'exécution native du code, qu'à son insertion dans un modèle. Bien entendu, le nombre de cycles à simuler au niveau processeurs est beaucoup plus conséquent que le nombre d'instructions au niveau architecture.

D'autres chiffres comme ceux du tableau 4.2 représente une estimation globale des temps

	Simulation SDL	Cosimulation niveau architecture	Cosimulation niveau cycle
Pas de simulation	Transaction	Instruction	Cycle d'horloge
Nb Cycles total	3300	23400	2451000
Cycle/s	330	25	23
Tps global de simulation	10s	15m 35s	30h 22m 39s

TAB. 4.2 – Temps de simulation à différents niveaux d'abstraction

de simulation en fonction des niveaux d'abstraction de la cosimulation par MCI dans le cas du contrôleur de robot. A titre indicatif, ces chiffres sont aussi dépendants de la qualité du code généré par l'outil MUSIC. Le temps de la simulation résulte de la granularité des pas de simulation. Un pas correspond à un cycle de simulation. Evidemment, ces cycles de simulation ont différentes représentations d'un niveau d'abstraction à un autre :

- Au niveau système, un pas de simulation correspond à une transition entre des processus parallèles. Pour notre application une simulation complète requiert 3300 cycles.
- Au niveau architecture, un pas de simulation représente un pas d'évaluation du simulateur de VHDL comportemental. Dans notre cas 23400 cycles sont nécessaires pour aboutir à une simulation complète.
- Au niveau processeur, un pas de simulation représente une instruction assembleur pour un modèle ISS du processeur. Dans notre cas 2451000 cycles sont nécessaires pour terminer la simulation.

Le fait de descendre dans les niveaux d'abstraction implique une augmentation exponentielle du nombre d'itérations à effectuer au cours de la simulation. Néanmoins, le temps d'exécution d'une itération diminue dans la descente des niveaux. On constate, indépendamment de la qualité de la génération que la descente dans ces niveaux d'abstraction reproduit des temps globaux de simulation qui se dégradent. Il est important de signaler que les chiffres obtenus proviennent de la première version stable de MCI. Même s'ils représentent bien la perte de rapidité en fonction des niveaux d'abstraction, ils ne retracent pas parfaitement les performances de l'outil lui-même.

Les performances de l'outil MCI dépendent des configurations, des machines utilisées et des types de modèles simulés. De nettes améliorations ont été apportées au fur et à mesure des travaux. D'une part, en optimisant les méthodes de communication. D'autre part, en apportant certaines modifications et optimisations au niveau du routage et des synchronisations. Enfin, une recherche de diminution des appels systèmes au niveau sémaphores, mémoires partagées et écriture "*socket*" a considérablement accentué la rapidité.

Les chiffres du tableau 4.3 représentent les performances de l'outil à bas niveau. Ces résultats ont été déterminés à partir d'un exemple contenant deux parties logicielles exécutées sur deux modèles de processeur Motorola 68000, plus une partie de contrôle en VHDL RTL. Les deux modèles de processeur sont exécutés sur une machine en local avec le routeur de MCI, et le VHDL est simulé par l'intermédiaire du déboggueur vhdldbx sur une machine distante à travers un réseau de communication de 10 Mb/s. Les processeurs mis en jeu dans cet environnement sont des modèles C du 68000, capables d'exécuter 22000 instructions par seconde. En

	Partie initialisation	Régime établi	Régime établi+ "*debug*"
Nb Pas globaux	100000	300000	300000
Temps de simulation	18' 17"	41' 38"	48' 15"
Tps réel simulé	8.33 ms	25 ms	25 ms
Nb Inst. Proc 1	97571	297211	297285
Nb Inst. Proc 2	95233	288757	286075
Inst./Sec	87.88	117.29	100.75
Nb appel au bus	30529	62258	73614
Nb Bus appels/Sec	27.83	24.92	25.43
Tps pour simuler 1ms	2' 12"	1' 40"	1' 56"

TAB. 4.3 – Temps de simulation sur un exemple 68000-68000-VHDL

considérant l'horloge VHDL connectée sur l'horloge des processeurs cadencés à 12 Mhz, nous pouvons affirmer que l'environnement est capable de simuler avec une vitesse approchant les 8 μs de simulation réelle par seconde (pour cet exemple), soit la simulation réelle d'1 ms en environ 2 minutes. On distingue tout de même trois phases sensiblement différentes durant la simulation :

- Dans la période d'initialisation, les processeurs échangent énormément de données (30539 échanges). La vitesse de simulation chute légèrement.
- Au régime établi, le processeur a atteint une moyenne d'échanges fixe. La vitesse de simulation augmente puisqu'elle passe de 2'12" à 1'40" stationnaire pour simuler 1ms.
- Enfin, la partie "*debug*" montre qu'il est possible de visualiser les informations renseignant l'utilisateur (conversions, adaptation de type) sans trop dégrader les performances.

Ces mesures, n'ont pu être, faute de moyens, réalisées sur la même machine (Outils Synopsys installés sur un réseau différent). Cependant, nous pouvons constater que la communication joue un rôle important dans la vitesse de l'environnement de cosimulation. Lorsque les échanges diminuent la vitesse augmente sensiblement.

4.4 Analyse des Performances

Les critères principaux qui peuvent influencer la cosimulation sont liés :

- Aux performances des simulateurs.

Le choix des simulateurs influe énormément sur la rapidité de la simulation. Par exemple, le simulateur LEAPFROG de Cadence est beaucoup plus rapide que le simulateur VSS de Synopsys. De même, pour le simulateur SDL, il sera plus judicieux d'utiliser Telelogic pour sa rapidité de simulation.

- Au nombre des simulateurs inclus dans l'environnement.

Si la cosimulation mise en place comprend la simulation de plusieurs sous-systèmes, la charge du processeur doit être repartie entre les différents simulateurs et le routeur de l'environnement de cosimulation. Il sera dans certains cas préférable de répartir les simulateurs sur des machines différentes pour optimiser le temps CPU. Il faut néanmoins faire attention au nombre d'échanges impliqués entre le routeur et les simulateurs distants, car une "*socket*" de domaine Internet est requise, ce qui diminue considérablement la vitesse de transmission des données.

- A la quantité d'informations échangées entre les simulateurs

Les échanges transitent via le routeur de l'environnement de cosimulation. Ceci signifie qu'une donnée échangée parcourt deux "*sockets*" ainsi qu'une étape de traitement (conversions, adaptation, ...). Si le nombre de données à échanger est trop grand entre deux applications, il peut y avoir un effet d'entonnoir du à la bande passante du support de communication. Pour l'éviter, le concepteur peut partitionner son(ses) modèle(s) pour limiter les communications par le même support.

– Au niveau d'abstraction de la simulation

Un environnement de cosimulation devient lent en descendant dans les niveaux d'abstraction, car les simulateurs qu'il implique doivent calculer beaucoup plus de cycles et parce que la fréquence des échanges augmente proportionnellement aux nombres de cycles. De plus, à bas niveau, la cosimulation temporelle implique la synchronisation de ces simulateurs sur l'avancement du temps. Pour la méthode de la barrière temporelle, elle consiste à attendre le dernier simulateur (le simulateur le plus lent) avant de passer au pas suivant. Si la finesse de la granularité de la barrière temporelle est augmentée le temps de simulation devient exponentiel.

4.5 Résultats de MCI

L'outil MCI a effectivement été testé sur de nombreux exemples. Grâce à eux, nous avons pu faire évoluer la méthodologie et la vérifier. Il a été utilisé par de nombreuses personnes et l'ensemble des remarques et des besoins de chacun ont engendré une considérable amélioration de l'outil à beaucoup de points de vue (rapidité, convivialité, possibilité, ...). Les multiples expériences menées au cours de ces travaux de recherche, avec tout d'abord PSA, puis le CNET ont conduit à une multiplicité des exemples de cosimulation multilangage à différents niveaux d'abstraction. Enfin, pour permettre le développement du nombre d'applications, un lien entre l'outil MUSIC [VRD+97] et MCI a été établi.

4.5.1 Mise en Place Relativement Aisée de Systèmes Multilangages

Un des principaux atouts de l'outil est sa facilité d'utilisation lorsque l'on veut mettre en place une application multilangage. En effet, chaque application est décrite selon une entité à priori inconnue de l'utilisateur. On y énumère les ports à cosimuler, en décrivant leurs types. Enfin, une simple "*netlist*" d'interconnexions est nécessaire pour la description du système complet. La définition du langage qui doit être utilisée pour chaque entité est décrite par une propriété où l'utilisateur spécifie le nom parmi une liste des langages disponibles. Le reste du processus est automatique. En respectant une certaine structure des fichiers à exécuter, l'outil guide l'utilisateur et simule en l'informant au maximum des éléments qui s'échangent. Un environnement de cosimulation se met en place très facilement manuellement et même de façon automatique avec l'outil de co-conception MUSIC.

4.5.2 Lien avec l'Outil de Co-conception MUSIC

MCI, qui était au départ un outil indépendant pour des applications particulières, s'est interconnecté avec l'outil MUSIC pour pouvoir à la fois valider la sortie de MUSIC, par l'exécution de cosimulations diverses, et pour valider les concepts de MCI. Le fichier de coordination de

MCI est devenu, au fur et à mesure de l'avancement des travaux, un fichier en langage intermédiaire SOLAR dans le but d'être également compatible avec MUSIC.

MUSIC est un outil de co-conception permettant de partir d'une description de haut niveau en SDL pour générer une description du système en C et/ou VHDL RTL. Ceci s'effectue sous le contrôle de l'utilisateur. Ce dernier définit le partitionnement, peut regrouper des taches, les dissocier, et désigner les parties à générer en logiciel ou en matériel. L'outil de cosimulation vient désormais en aval de cette génération puisqu'il permet de valider et de vérifier que cette génération est correcte et de visualiser les chronogrammes de la simulation du système avant son implémentation.

MUSIC génère donc les fichiers logiciels et matériels ainsi que la "*netlist*" d'interconnexions. L'utilisateur compile les différents fichiers et peut aussitôt exécuter une cosimulation avec MCI. L'avantage de ceci, est que l'utilisateur a la possibilité de modifier un paramètre au niveau SDL et de refaire sa simulation après génération en moins d'une minute parfois. Ce gain de temps améliore très sensiblement le temps de mise sur le marché.

4.6 Problèmes Subsistants et Améliorations Possibles

Les problèmes subsistants dans la cosimulation sont orientés essentiellement sur la vitesse de simulation. L'aspect essentiel du gain en vitesse est basé sur le type de support de communication utilisé. Pour l'utilisation d'une application distante, il faut impérativement se servir d'une "*socket*" de domaine internet. Par contre, si la cosimulation est locale il peut être intéressant d'éliminer la "*socket*" locale et de réaliser plutôt une communication par mémoire partagée. Il faudrait donc prévoir deux modes de fonctionnement de l'environnement suivant le type de simulateurs connectés. Ainsi, la vitesse de simulation peut être accélérée à condition aussi que le concepteur adopte une méthodologie d'utilisation. Il doit faire fonctionner localement les applications les plus lourdes en consommation de ressources.

La cosimulation temporelle peut aussi être améliorée par l'utilisation d'un moteur de synchronisation plus adapté. Le modèle de la barrière temporelle est simple, sûr et facile à mettre en oeuvre. Pourtant, d'autres méthodes d'optimisation des points morts doivent être envisagées. Un système par FIFO peut être mis en place pour stocker et distribuer les résultats pré-simulés des simulateurs rapides vers les simulateurs les plus lents. Cette technique doit permettre un gain de temps global de simulation.

La gestion des types et les conversions automatiques sont complexes mais très utiles pour connecter des applications dont les types ne sont pas forcément compatibles. Pour parfaire ce complexe mécanisme, il serait intéressant de compléter la résolution des types en fonction des différentes applications même si la plupart d'entre-elles ne l'utilisent pas. Cette résolution solverait notamment les éventuels problèmes de connexion entre applications différentes. Celle-ci réaliserait la simulation selon une spécification fournie par l'utilisateur et pourrait être reprise, par la suite, pour la synthèse des interfaces.

4.7 Conclusion

Les exemples d'applications pour l'outil MCI ont été nombreux et enrichissants. Ils ont participé très étroitement à l'évolution de l'outil et à sa validation. L'expérience au sein du groupe PSA a permis d'évaluer les besoins industriels dans le domaine du multilangage. Depuis, MCI

s'est étendu autour d'exemples multiples sous des configurations très hétérogènes et variées. La connexion de MCI avec l'outil MUSIC a permis la réalisation du lien entre l'outil de co-conception et l'outil de validation. La génération de nombreux exemples a permis de consolider l'outil MCI et d'étendre son environnement à des caractéristiques non explorées. L'effervescence de nouveaux exemples a démontré que des points faibles sont encore inhérents à l'outil, notamment en ce qui concerne la communication. De plus, même si l'outil est devenu un outil de validation fiable et performant au niveau système, il n'en demeure pas moins que de nouveaux travaux méritent d'être poursuivis. Il serait intéressant, par exemple, de faire des recherches sur des algorithmes de gestion du temps plus performants que l'algorithme de la barrière temporelle. Il serait peut-être également bénéfique d'entreprendre des travaux permettant d'extraire les résultats de la cosimulation pour établir des analyses qui faciliteraient les explorations d'architectures. L'outil est suffisamment générique pour pouvoir être amélioré. Ainsi, tout porte à croire que demain des exemples très conséquents pourront être validés par MCI.

Conclusions et Perspectives

L'accroissement des techniques de conception a engendré l'intégration de systèmes de plus en plus complexes sur le marché de la microélectronique. Les circuits développés sont de plus en plus performants en terme de rapidité et occupent de moins en moins d'espace. Cependant, la complexité grandissante liée à cet essor a poussé les concepteurs à utiliser des outils de co-conception. De ce fait, afin que les concepteurs maîtrisent la complexité, la recherche s'oriente de plus en plus vers de nouveaux outils et de nouvelles méthodes de co-conception orientés système et multilangage. Le premier chapitre de cette thèse a présenté ces nouvelles méthodes de conception et de validation multilangage.

L'apparition des outils de validation a suivi l'évolution des outils de conception. Depuis l'apparition des premières méthodes de co-conception, les premières méthodes de validation par cosimulation sont apparues. Nous avons retracé les principales caractéristiques de la cosimulation dans le second chapitre. L'arrivée des nouvelles méthodes de conception multilangages a engendré l'extension de ces outils de validation vers de nouveaux langages. Actuellement, il existe quelques applications de validation multilangages mais elles sont dédiées à l'utilisation de langages précis. C'est pourquoi, nous avons présenté, dans le chapitre 3, un outil de cosimulation multilangage adapté à un maximum de langages. L'outil que nous proposons ciblait au départ les applications bien spécifiques comme la modélisation mécatronique. Il s'étend désormais au concept fondamental de la conception multilangage. De plus, par l'adjonction de synchronisations et d'adaptations aux différents modes de simulation, il permet désormais de valider un système complet multilangage à tous les niveaux de son abstraction. Sa conception sous forme de couches lui confère une minimisation de l'effort quant à l'ajout de nouveaux simulateurs. A priori, tout simulateur possédant une interface C, pour communiquer vers son extérieur, peut s'insérer dans la liste des langages compatibles avec notre outil. La liste actuelle est composée du langage C, VHDL, COSSAP, SDL, MATLAB, et SABER. Cette liste a été étoffée conformément à la demande des utilisateurs. De plus, l'outil permet de cosimuler du plus haut niveau d'abstraction au plus bas niveau. Ainsi, il offre à l'utilisateur la possibilité de valider sa conception à chacune des étapes des différents raffinements. L'élaboration de modèles de processeurs a permis d'évaluer ses performances au plus bas niveau de la simulation. Grâce à l'outil nous avons pu reproduire en temps réel de nombreux systèmes avant l'étape de prototypage. Enfin, il a engendré l'ouverture vers de nouveaux travaux de recherche. C'est le cas, entre autre, de l'exploration d'architecture, l'étude d'architecture multi-processeurs, et la synthèse d'interface multilangage. Cette dernière est nécessaire pour l'adaptation des futurs outils de co-conception aux méthodologies multilangages. Actuellement, l'outil de validation réalise lui-même ces adaptations au sein du routeur et prévient l'utilisateur des modifications engendrées (conversions, adaptation de type, etc..).

De nombreuses applications ont permis d'évaluer ses performances et ses propriétés aussi bien dans le monde de la recherche que dans le domaine des systèmes industriels. La connexion

avec l'outil de co-conception MUSIC permet à la fois de valider la génération de l'outil de co-conception et d'encourager les utilisateurs à créer de nouveaux exemples de systèmes à co-simuler. Les résultats de MCI ont été améliorés au fur et à mesure des travaux réalisés et l'écart des possibilités de simulation au niveau abstraction s'est creusé. Même si les performances à bas niveau ne concurrencent pas directement les outils commerciaux, il n'en demeure pas moins que MCI est un outil de validation par cosimulation suffisamment performant et flexible pour permettre la validation d'importants systèmes industriels tout au long de leur flot de conception.

Les travaux à venir se distinguent en plusieurs catégories. La première consiste à améliorer la rapidité de l'outil. En effet, une étape d'optimisation des synchronisations et de la communication est envisagée, afin d'augmenter les taux de transferts de l'outil et d'accélérer l'environnement de cosimulation. Aussi, il serait intéressant de faire une corrélation entre la gestion des types du routeur et l'étape de synthèse d'interface multilangage. Cette étude pourrait permettre d'évaluer les transformations des données à plusieurs niveaux d'abstraction en fonction des différents langages. Une étape de cosimulation pourrait évaluer les différentes incompatibilités de connexions, tout en autorisant la continuité de la simulation. Les résultats de ces incompatibilités pourraient être reliés à un outil de synthèse d'interface multilangage. Celui-ci permettrait d'ajouter, aux différents modèles du système, les méthodes nécessaires pour descendre automatiquement les niveaux d'abstraction. Dans le même esprit, une extension de la résolution des types est à envisager concernant les langages. Enfin, l'amélioration essentielle consiste à agrandir les capacités d'entrée de l'outil, et de pouvoir accepter le plus grand nombre de langages possible. Cette extension est primordiale pour couvrir un maximum de domaines d'activité de la microélectronique.

Néanmoins, l'outil reste un environnement de plus en plus stable pour la validation des systèmes multilangages. Un nombre important d'exemples a pu être exploité à partir de l'outil et a également engendré la validation d'une grande partie de la chaîne de l'outil de co-conception MUSIC. On peut supposer qu'il saura se faire une place sur le marché des outils de validation lorsque les concepteurs auront pris conscience de l'utilité de la modélisation et de la conception multilangages.

Enfin, la plus grande consécration de cet outil, est son transfert technologique entre le laboratoire TIMA et la société AREXSYS. L'outil est vendu sous l'appellation de CosiMate^{TM} et permet la validation des systèmes multilangages.

Bibliographie

[ABFS94] S. Antoniazzi, A. Balboni, W. Fornaciari, and D. Sciuto. "A Methodology for Control-Dominated Systems Codesign". In IEEE CS Press, editor, *Third International Workshop on Hardware/Software Codesign (CODES/CASHE)*, pages 2–9, Grenoble, France, Septembers 1994.

[Abr97] J. R. Abrial. "The Book. Assigning Programs to Meaning". *Cambridge University Press*, 1997.

[AG94] M. Altmae and P. Gibson. "Hardware/Software Coverification". In *Proceedings of VHDL-Forum for CAD*, April 1994.

[Alt95a] M. Altmae. "Hardware/Software Coverification". In *Proceedings of EDA Traff*, March 1995.

[Alt95b] M. Altmae. "Synthesia tools documentation". Technical report, Synthesia, August 1995.

[Are00] Arexsys. "Arexsys : Product Description, http ://www.arexsys.com". Technical report, Arexsys Inc., 2000.

[Aut95] Eagle Design Automation. "Providing Virtual Solutions for Embedded Systems". Technical report, Eagle Design Automation, December 1995.

[Aut96] Eagle Design Automation. "Co-Verification of Embedded Systems Design Using Virtual System Integration". Technical report, Eagle Design Automation, 1996.

[Bar81] M. R. Barbacci. "Instruction set processor specification (isps) : The notation and its applications". In *IEEE trans. on computer, c30(1)*, pages 24–40, January 1981.

[BC84] G. Berry and L. Cosserat. "The Esterel Synchronous Programming Language and its Mathematical Semantics. Language for Synthesis". Ecole National Superieure de Mines de Paris, 1984.

[BD87] S. Budowki and P. Dembinski. "An introduction to Estelle : A Specification Language for Distributed Systems". In *Computer Networks and ISDN Systems*, volume 13, pages 2–23, 1987.

[Ber91] G. Berry. "Hardware Implementation of Pure Esterel". In *The ACM Workshop on Formal Methods in VLSI Design*, Jan. 1991.

[BGJ+97] Felice Balarin, Paolo Giusto, Attila Jurecska, Claudio Passerone, Ellen Sentovich, Bassam Tabbara, Massimiliano Chiodo, Harry Hsieh, Luciano Lavagno, Alberto Sangiovanni-Vincentelli, and Kei Suzuki Hardbound. *Hardware-Software Co-Design of Embedded Systems, The POLIS Approach*, volume 404 of *ENGINEERING AND COMPUTER SCIENCE*. Kluwer Academic Publishers, May 1997. ISBN 0-7923-9936-6.

[BHL94] J. Buck, S. Ha, and E. Lee. “Ptolemy : A Framework for Simulating and Prototyping Heterogeneous Systems”. *International Journal of Computer Simulation*, January 1994.

[Bol97] I. Bolsens. “Specification, Cosimulation and Hardware/Software Interfacing for Telecom Systems”. *Leuven Codesign Course*, February 1997.

[BP94] J. Benzakki and P.Asar. “Framework and Multi-Formalism : the ASAR Project”. In *Fourth International IFIP 10.5 Working Conference on Electronic Design Automation Frameworks*, Gramado, Brazil, november 1994.

[Bru87] J. Bruijnin. “Evaluation and integration of specification languages”. *Computer Networks and ISDN Systems, 13(2)*, pages 75–89, 1987.

[BRX94] E. Barros, W. Rosentiel, and X. Xiong. “A method for partitionning unity language in hardware and software”. In IEEE CS Press, editor, *European Design Automation Conference 5EuroDAC)*, pages 220–225, September 1994.

[BST92] D. Becker, R. Singh, and S. G. Tell. “An Engineering Environment for Hardware/Software Co-Simulation”. In *29th ACM/IEEE Proc. of the Design Automation Conference*, pages 129–134, 1992.

[Buc94] K. Buchenreider. “A prototyping Environment for Control-Oriented HW/SW Systems using State-Charts, Activity-Charts and FPGAs”. In IEEE CS Press, editor, *Euro-DAC with Euro-VHDL*, pages 60–65, Grenoble, France, September 1994.

[Cha96] A. Changuel. *Prototypage rapide d’architectures mixtes logiciels/matériel à partir de modèles mixtes C-VHDL*. PhD thesis, Institut National Polytechnique de Grenoble, Octobre 1996.

[CHL97] W. Chang, S. Ha, and E. Lee. “Heterogeneous simulation mixing discrete event models with dataflow”. *Invited Paper, Journal on VLSI Signal Processing*, 13(1), January 1997.

[CHP95] J. P. Calvez, D. Heller, and O. Pasquier. “System Performance Modeling and Analysis with VHDL : Benefits and Limitations”. In *Proceedings of VHDL-Forum Europe Conference*, April 1995.

[CoW96] CoWare. “CoWare : Product Description”. Technical report, CoWare, 1996.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J.A. Plaice. “LUSTRE : a declarative language for real-time programming”. In *Fourteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of programming languages*, pages 178–188, Munich, Germany, 1987.

[CT95] S.L. Coumeri and D.E. Thomas. “A Simulation Environment For Hardware-Software Codesign”. In *Proc. ICCD Conf.* IEEE CS Press, Oct. 1995.

[Cur91] D. A. Curry. *Using C on the UNIX System*, o’reilly & associates edition, February 1991.

[Dav95] A. Davis. *Software Requirements : Analysis and Specification*. Elsiever, NY, 1995.

[Dav97] J-M. Daveau. *Specifications Systemes et Synthese de la Communication Pour Le Co-Design Logiciel/Materiel*. PhD thesis, Institut National Polytechnique de Grenoble, Dec 1997.

[DBDH99] J. M. Delosme, J. Benzakki, B. Djafri, and R. Hamouche. “Java, VHDL-AMS, Ada or C for System Level Specification ?”. In *DATE’99*, Munich, Germany, March 1999.

[DMJ97] J.-M. Daveau, G. Fernandes Marchioro, and A.A. Jerraya. “VHDL Generation from SDL Specification”. In Carlos D. Kloos and Eduard Cerny, editors, *CHDL*, pages 182–201. Chapman-Hall, Apr. 1997. IFIP.

[EHB+95] R. Ernst, J. Henkel, Th. Benner, W. Ye, U. Holtmann, D. Hermann, and M. Trawny. “The COSYMA Environment for Hardware/Software Cosynthesis”. *Journal of Microprocessors and Microsystems, Butterworth-Heinemann*, 1995.

[FIR+97] L. Freund, M. Israel, F. Rousseau, J.M. Berge, M. Auguin, C. Belleudy, and G. Gogniat. “A codesign experiment in acoustic echo cancellation : GDMDFa”. *ACM Transaction on Design Automation for Embedded Systems*, 2(4), Octobre 1997.

[GDZ98] D. Gajski, Rainer Dömer, and Jianwen Zhu. “IP-Centric Methodology and Design with the SpecC Language”. In *Contribution to NATO-ASI Workshop on System Level Synthesis*, Il Ciocco, Barga, Italy, Aug. 1998.

[GG87] T. Gautier and P. Le Guernic. “Signal, a declarative language for synchronous programming of real-time systems”. In *Computer Science, Formal Languages and Computer Architectures*, page 274, 1987.

[GG96] J. Gong and D. D. Gajski. “Model Refinement for Hardware-Software Codesign”. In *IEEE Design & Test of Computers*, pages 270–274, Los Alamitos, CA., March 1996.

[Glu93] W. Glunz. “Methodology for using SDL for Hardware Design at abstract Levels”. Technical report, Siemens AG, sept. 1993.

[GM92] R.K. Gupta and G. De Micheli. “System-level Synthesis using Re-programmable Components”. In *Proc. Third European Conf. Design Automation*, pages 2–7. IEEE CS Press, 1992.

[Gra96] Mentor Graphics. “A Cosimulation Tool from Mentor Graphics for ST20”. Technical report, Mentor Graphics Inc, 1996.

[Gro95] Compiled Designs Group. *VHDL-SABC167 manual*. Synopsys, October 1995.

[Gro96a] Alta Group. “Application-Specific Design Automation Tools”. Technical report, Cadence Design Systems, July 1996.

[Gro96b] Alta Group. “Signal Processing WorkSystem : DSP Processor Models User’s Guide”. Technical report, Cadence Design Systems, June 1996.

[GS96] A. Gokhale and D. C. Schmidt. “The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks”. In *GLOBECOM’96*, pages 50–56, London, England, November 1996. IEEE.

[GV95] D. Gajski and F. Vahid. “Specification and Design of Embedded Hardware-Software Systems”. *IEEE Design & Test of Computers*, pages 53–67, 1995. Spring.

[GVNG94] D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice Hall, Inc. Englewood Cliffs, New Jersey, 1994.

[Har87] D. Harel. “Statecharts : A Visual Formalism for Complex Systems”. *Science of Computer Programming*, 8 :231–274, 1987.

[HCM+99] F. Hessel, P. Coste, P. Le Marrec, N. Zergainoh, J. M. Daveau, and A. A. Jerraya. "Communication Interface Synthesis for Multilanguage Specifications". *RSP'99, 10th IEEE International Workshop on Rapid System Prototyping*, June 1999.

[HH93] K.T. Hagen and H.Meyer. "Timed and Untimed Hardware/ Software Cosimulation : Application and Efficient Implementation". In *International Workshop on Hardware-Software Codesign*, Cambridge, Oct. 1993.

[HHE94] D. HermanN, J. Henkel, and R. Ernst. "An Approach to the Adaptation of Estimated Cost Parameters in the Cosyma System". In *Third Int'l Wshp on Hardware/Software Codesign Codes/CASHE*, Grenoble, Sept. 1994. IEEE CS Press.

[HLJ98] F. Hessel, P. Lemarrec, and A. Jerraya. "MCI - Multilanguage Distributed Co-Simulation Tool". *International IFIP WG 10.3 / WG 10.5 Workshop on Distributed and Parallel Embedded Systems*, oct. 1998.

[Hoa85] C. A. R. Hoare. "Communicating sequential Processes". *Prentice Hall*, 1985.

[HP98] HP. *HP Advanced Design System*. 1998. http ://www.tmo.hp.com/tmo/hpeesof/products/ads/adsoview.html.

[IEE94] IEEE. *Institute of Electrical and Electronics Engineers. IEEE Standard VHDL Language Reference Manual, Std 1076-1993*, 1994.

[IJ95] T. Ben Ismail and A. Jerraya. "Synthesis Steps and Design Models for Codesign". In *IEEE Computer, special issue on Rapid-prototyping of Microelectronics Systems*, pages 44–52, February 1995.

[Inc98] Synopsys Inc. *Synopsys VHDL System Simulator Interfaces Manual : C-language Interface*. Synopsys Inc., June 1998.

[IOJ95] T. Ben Ismail, K. O'Brien, and A. A. Jerraya. "PARTIF : Interactive system-level Partitionning". In *VLSI Design*, pages 333–345, 1995.

[IS98] Inc. Integrated Systems. *MATRIXx*. 1998. Http ://Www.Isi.com/Products/MATRIXx/.

[ISO89] IS 8807 ISO. "LOTOS a Formal Description Technique based on The Temporal Ordering of Observational Behaviour", February 1989.

[ISZ98] V. Issarny, T. Saridakis, and A. Zarras. "A Survey of Architecture Description Languages". Technical report, ESPRIT LTR Project Nř24962 C3DS Deliverable A3.1, 1998.

[JKS+98] Axel Jantsch, Shashi Kumar, Ingo Sander, Bengt Svantesson, Johnny Öberg, and Ahmed Hemani. "Evaluation of Languages for Specification of Complex Telecom Systems". *Internal report*, 1998.

[JO94] A.A. Jerraya and K. O'Brien. "SOLAR : An Intermediate Format for System-level Modeling and Synthesis". In J.Rozenblit and K.Buchenrieder, editors, *Computer Aided Software/Hardware Engineering*. IEEE Press, 1994.

[Jon90] C. B. Jones. "Systematic software development using vdm.". In C.A.R Hoare Series, editor, *Prentice Hall International Series in Computer Science*, 1990.

[JRM+99] A. A. Jerraya, M. Romdhani, Ph. Le Marrec, F. Hessel, P. Coste, C. Valderrama, G. F. Marchioro, J. M. Daveau, and N. E.D. Zergainoh. "Multilanguage Specification for System Design". In A. Jerraya and J. Mermet, editors, *System-Level*

Synthesis, NATO ASI 1998, pages 103–136. Kluwer Academic Publishers. ISBN : 0-7923-5749-3, 1999.

[JRV+97] A.A. Jerraya, M. Romdhani, C. Valderrama, Ph. Le Marrec, F. Hessel, G. Marchioro, and J.M. Daveau. *Hardware and Software Co-design : Principles and Practice*, chapter Languages for system-level specification and design, pages 235–262. Kluwer Academic Publishers, 1997.

[KeA95] C. Delgado Kloos and Marin Lopez et All. "From Lotos to Vhdl". *Current Issues in Electronic Modelling*, 3, Sept. 1995.

[KKM] H. Karathanasis, D. Karkas, and D. Metafas. "DECT Source Algorithmic Components". Technical report, d0.1s1 AS PIS ESPRIT project (EP20287) http ://www.rational.com/uml/index.shtml.

[KKR94] G. Koch, U. Kebshull, and W. Rosenstiel. "A prototyping Environment for Hardware/Software Codesign in the COBRA Project". In *International Workshop on Hardware-Software Codesign*, Grenoble, France, Sep. 1994.

[KKS+96] K. Kim, Y. Kim, Y. Shin, K. Choi, and S. Ha. "An Integrated Hardware-Software Cosimulation Environment with Automated Interface Generation". In *Proceedings of 7th IEEE International Workshop on Rapid System Prototyping*, pages 66–71, June 1996.

[KL93] A. Kalavade and E.A. Lee. "A Hardware-Software Codesign Methodology for DSP Applications". In *IEEE Design & Test of Computers*, pages 16–28, Sep. 1993.

[KM88] D. Ku and G. De Micheli. "HardwareC, a Language for Hardware Design". Technical report, Tech. Rep. CSL-TR-88-362, Computer Systems Laboratory, Standford University, August 1988.

[KPSG99] C. D. Kloos, S. Pickin, L. Sanchez, and A. Groba. "High-Level Specification Languages for Embedded System Design". In A. Jerraya and J. Mermet, editors, *System-Level Synthesis*, NATO ASI 1998, pages 137–174. Kluwer Academic Publishers. ISBN : 0-7923-5748, 1999.

[LDA93a] W. Loucks, B.J. Doray, and D.G. Agnew. "Experiences in Real Time Hardware-Software Cosimulation". *Proc VHDL Int. Users Forum (VIUF), Ottawa, Canada,*, pages 47–57, April 1993.

[LDA93b] W.M. Loucks, B.J. Doray, and D.G. Agnew. "Experiences In Real Time Hardware-Software Cosimulation". In *Proc VHDL Int. Users F o rum (VIUF)*, pages 47–57, Otawa, Canada, Apr. 1993.

[LLSV98] J. Liu, M. Lajolo, and A. Sangiovanni-Vincentelli. "Software Timing Analysis Using HW/SW Cosimulation and Instruction Set Simulator". In *6th International Workshop on Hardware/Software Co-Design (Codes/CASHE'98)*, pages 67–71, 1998.

[LNV+97] C. Liem, F. Naçabal, C. Valderrama, P. Paulin, and A.A. Jerraya. "Cosimulation and Software Compilation Methodologies for the System-on-a-Chip in Multimedia". *IEEE Design & Test of Computers*, 1997. special issue on "Design, Test & ECAD in Europe".

[LR93] S. Lee and J.M. Rabaey. "A Hardware Software Cosimulation Environment". In *International Workshop on Hardware-Software Codesign*, Cambridge, Oct. 1993.

[LSPM95] U. Lambrette, B. Schmandt, G. Post, and H. Meyr. “Cossap-Matlab Cosimulation”. In *ICSPAT95*, Boston, oct. 1995.

[LSVH96] L. Lavagno, A. Sangiovani-Vincentelli, and H. Hsieh. *Embedded System Codesign : Synthesis and Verification*, pages 213–242. Kluwer Academic, Boston, 1996.

[Mar96] *Integrated System Design with an Object Oriented Methodology*. CIEM : Current Issues in Electronic Modelling, Kluwer Academic Publishers, September 1996.

[Mat98] MathWorks. *MATLAB 5/SIMULINK 2*. MathWorks Inc., 1998. http ://www.mathworks.com/.

[Mil83] R. Milner. “Calculi for synchrony and asynchrony”. In *Theoritical Computer Science*, pages 267–310, 1983.

[MVHJ98] P. Le Marrec, C. Valderrama, F. Hessel, and A. Jerraya. “Harware, Software and Mechanical Cosimulation for Automotive Applications”. In *RSP’98, 9th IEEE International Workshop on Rapid System Prototyping*, Leuven, Belgium, june 1998.

[NVG91] S. Narayan, F. Vahid, , and D. Gajski. “System Specification and Synthesis with the Specchart Language”. In *Int’l Conf. On Computer-Aided Design (ICCAD)*, pages 226–296. IEEE CS Press, Nov. 1991.

[OBE+97] A. Österling, T. Benner, R. Ernst, D. Herrmann, T. Scholz, and W. Ye. *The Cosyma System*, chapter Hardware/Software Co-Design : Principles and Practice, J. Staunstrup and W. Wolf editors, pages 263–305. Kluwer Academic Publishers, 1997.

[Ous97] J.K. Ousterhout. “Scripting : Higher Level Programming for the 21st Century”. In *IEEE/ACM International Conference on Computer Aided Design*, page White Paper, Nov. 1997.

[Pa83] P. Piloty and al. “Conlan report”. In In Springer Verlag, editor, *Lecture notes in computer science*, page 151, Berlin, 1983.

[Pet81] J. L. Peterson. “Petri Net Theory and the modeling of Systems”. In *Prentice-Hall*, Englewood Cliffs, N.J., 1981.

[Pfe97] R. Pfeifer. “Simulation Backplane 1.5b5”, Oct. 1997. Evaluation report.

[PHG97a] F. Petrot, D. Hommais, and A. Greiner. “Cycle precise core based hardware/software system simulation with predictable event propagation”. In *Proceeding of the 23rd Euromicro Conference*, pages 182–187, Budapest, Hungary, September 1997.

[PHG97b] F. Petrot, D. Hommais, and A. Greiner. “A Simulation Environment for Core Based Embedded Systems”. In *Proceeding of the 30th Annual Simulation Symposium*, pages 86–91, Atlanta, Georgia, April 1997.

[PLMS94] P. Paulin, C. Liem, T. May, and S. Sutarwala. “DSP Design Tool Requirements for Embedded Systems : A Telecommunication Industrial Perspective”. *in journal of VLSI Signal Processing (special issue on synthesis for real-time DSP), Kluwer Academic Publishers*, 1994.

[PSH98] M. Pfaff, M. Schutti, and R. Hagelauer. “SHarC : Coupling Simulation and Physical Hardware to Improve Design Validation”. In *Design Automation and Test in Europe Conference*, pages 129–133, PARIS, 1998.

[RBCH96] A. Rault, Y. Bezard, A. Coustre, and T. Halconruy. “Systems Integration in the Car Industry”. PSA, Peugeot-Citroen 78140 Velizy-Villacoublay, France, 1996.

[RC95] P. Runstadler and R. Crevier. “Virtual Prototyping for System Design and Verification”. Technical report, Synopsys documentation, March 1995.

[RD98] Daniel D. Gajski Rainer Dömer, Jianwen Zhu. “The SpecC Language Reference Manual”. Technical Report ICS-TR-98-13, University of California, Irvine, Mar. 1998.

[Rei87] S.P. Reis. “Working in the Garden Environment Form Conceptual Programming”. *IEEE Software*, 4 :16–27, Nov. 1987.

[RHJ+95] M. Romdhani, R.P. Hautbois, A. Jeffroy, P. de Chazelles, and A.A. Jerraya. “Evaluation and Composition of Specification Languages, an Industrial Point of View”. In *Proc. IFIP Conf. Hardware Description Languages (CHDL)*, pages 519–523, Sept. 1995.

[Row94] J.A. Rowson. “Hardware/Software Co-Simulation”. *Proceedings of the 31st Design Automation Conference, San Diego, CA, USA*, pages 439–440, 1994.

[RVBM96] K. Van Rompaey, D. Verkest, I. Bolsens, and H. De Man. “CoWare - A Design Environment for Heterogeneous Hardware/Software Systems”. In *Proceedings of the European Design Automation Conference Euro-DAC’96, Geneva, Switzerland*, page 252, September 1996.

[RW93] N.L. Rethman and P.A. Wilsey. “RAPID : A Tool For Hardware/ Software Tradeoff Analysis”. In *Proc. CHDL’93*, Otawa,Canada, Apr. 1993. Elsevier Science.

[Sab] Saber Technology Ltd, http ://www.saber.bm. *Saber*.

[Sch98] E. Schulz. “The New System-Level Design Language”. *Integrated System Design*, pages 26–34, jul. 1998.

[SD] CoWare Synopsys and Frontier Design. *SystemC v1.0*. Synopsys.

[Spi89] J. M. Spivey. “An introduction to z formal specifications”. *Software Engineering Journal*, pages 40–50, January 1989.

[ST87] R. Sarracco and P. A. J. Tilanus. “CCITT SDL : An Overview of the Language and its Applications”. *Computer Networks & ISDN Systems, Special Issue on CCITT SDL*, 13(2) :65–74, 1987.

[Tan89] A. Tanenbaum. *Computer Networks*, chapter 7. Prentice Hall International, Inc., Englewood Cliffs, N.J., 1989.

[TAS93] D.E. Thomas, J.K. Adams, and H. Schmit. “A Model and Methodology for Hardware-Software Codesign”. *IEEE Design & Test of Computers*, 10(3) :16–28, Sep. 1993.

[VNG95] F. Vahid, S. Narayan, and D. Gajski. “Specchart s : A Vhdl Front-End for Embedded Systems”. *IEEE trans. on CAD of Integrated Circuits and Systems*, 14(6) :694–706, 1995.

[VNPJ96] C.A. Valderrama, F. Nacabal, P. Paulin, and A.A. Jerraya. “Automatic generation of interfaces for distributed C-Vhdl Cosimulation of Embedded Systems : An Industrial Experience”. *Seventh IEEE International Workshop on Rapid System Prototyping*, pages 72–77, June 1996.

[VRD+97] C. Valderrama, M. Romdhani, J.M. Daveau, G. Marchioro, A. Changuel, and A.A.Jerraya. *Hardware and Software Co-design : Principles and Practice*, chapter COSMOS : A Transformational Codesign Tool for Multiprocessor Architectures, pages 307–357. Kluwer Academic Publishers, 1997.

[Wil92] D.S. Wile. "Integrating Syntaxes and their Associated Semantics". Technical Report RR-92-297, Univ. Southern California, 1992.

[Wol93] W. Wolf. "Guest Editor's Introduction : Hardware-Software Codesign". In *IEEE Design & Test of Computers*, September 1993.

[Wol94] W. Wolf. "Hardware-Software Codesign of Embedded Systems". *IEEE*, 27(1) :42–47, Jan. 1994.

[WOL98] WOLFRAM. *Mathematica*. 1998. http ://www.wolfram.com/mathematica/.

[XGR96] X. Xiong, P. Gutberlet, and W. Rosentiel. "Automatic Generation of Interprocess Communication in the PARGON System". In *RSP'96, 7th IEEE International Workshop on Rapid System Prototyping*, Thessalouiki, Greece, 1996.

[YC97] S. Yoo and K. Choi. "Optimistic Timed HW-SW Cosimulation". In *APCHDL'97*, pages 43–49, 1997.

[YW95] T. Y. Yen and W. Wolf. "Communication Synthesis for Distributed Embedded Systems". In *IEEE/ACM International Conference on Computer Aided Design*, pages 288–294, November 1995.

[ZJ93] P. Zave and M. Jackson. "Conjunction as Composition". In *ACM trans. On Software engineering and methodology*, volume 8, pages 379–411, Oct. 1993.

Annexe A : Publications

La liste des publications relative aux travaux exposés dans cette thèse est donnée ci-dessous :

Publications dans des ouvrages

- A. A. Jerraya, M. Romdhani, Ph. Le Marrec, F. Hessel, P. Coste, C. Valderrama, G. F. Marchioro, J. M. Daveau, N. E. Zergainoh. “Multilanguage Specification for System Design and Codesign”. Chapter in System-level Synthesis, NATO ASI 1998, edited by A. Jerraya and J. Mermet, ISBN : 0-7923-5749-3, Kluwer Academic Publishers, 1999.

- F. Hessel, P. Le Marrec, C. A. Valderrama, M. Romdhani, A. A. Jerraya. “MCI - Multilanguage Distributed Co-Simulation Tool”. Chapter in Distributed and Parallel Embedded Systems, DIPES’98 organized by IFIP WG10.3/WG 10.5, edited by F. Ramming, ISBN : 0-7923-8614-0, Kluwer Academic Publishers, 1999.

- A. A. Jerraya, M. Romdhani, C. A. Valderrama, Ph. Le Marrec, F. Hessel, G. Marchioro and J. M. Daveau. “Languages for system-level specification and design”. Chapter in Hardware/Software Co-design : Principles and Practice (J. Staunstrup and W. Wolf eds.), pp. 235-262, ISBN : 0-792-38013-4, Kluwer Academic Publishers, 1997.

Publications dans des conférences internationales

- F. Hessel, P. Coste, P. Le Marrec, N. Zergainoh, J. M. Daveau, A. A. Jerraya. “Communication Interface Synthesis for Multilangage Specifications”. RSP99, pp. 15-20, Clearwater, Florida, 1999.

- P.Coste, F. Hessel, P. Le Marrec, Z. Sugar, M. Romdhani, R. Suescun, N. Zergainoh, A. A. Jerraya. “Multilanguage design of heterogeneous Systems”. CODES 99, pp. 54-58, Roma, Italy, 1999

- P. Le Marrec, C. A. Valderrama, F. Hessel, A. A. Jerraya. “Hardware, Software and Mechanical Cosimulation for Automotive Applications”. RSP98, Leuven, Belgium, pp. 202-206, 1998.

- C. Valderrama, P. Le Marrec, A. A. Jerraya. “Multilanguage Cosimulation”. in Journal IEEE

Second International High Level Design Validation and Test Workshop - HLDVT 97, Oakland CA USA, 1997.

- F. Hessel, P. Le Marrec, C. A. Valderrama, M. Romdhani, A. A. Jerraya. “MCI - Multilanguage Distributed Co-Simulation Tool”. DIPES’98, Paderborn, Germany, Oct 1998

_ F. Hessel, P. Coste, P. Le Marrec, N. Zergainoh, G. Nicolescu, J. M. Daveau, A. A. Jerraya. “Interlanguage Communication Synthesis for Heterogeneous Specifications - DAFES 2000.

- G. Nicolescu, P. Coste, F. Hessel, P. Le Marrec, A. A. Jerraya. “Multilanguage Design of a Robot Arm Controller : Case Study”. VLSI’99.

- F. Hessel, P. Coste, G. Nicolescu, P. Le Marrec, N. Zergainoh, A. Jerraya. “Multi-level Communication Synthesis of Heterogeneous Multilanguage Specification”. ICCD’2000

Cosimulation Multiniveaux Dans un Flot de Conception Multilangage

Résumé : De nos jours, la réalisation d'un système électronique hétérogène composé de différents sous-systèmes (logiciel, matériel, mécanique, etc..), démarre par une spécification de haut niveau qui est ensuite divisée en différentes parties modélisées en différents langages. Ces nouvelles méthodes requièrent donc de nouvelles techniques d'aide à la conception et de validation afin de réduire le temps de mise sur le marché. La vérification par simulation de tels systèmes consiste à simuler conjointement les différentes parties du système en utilisant les simulateurs appropriés à leur type de modélisation. Ce type de simulation appelé cosimulation doit être réalisable à tous les niveaux de la spécification.

Le sujet de cette thèse consiste à étudier un outil de validation par cosimulation. Cet outil doit pouvoir vérifier un système complet tout au long de son flot de conception par simulation à l'aide de simulateurs concurrentiels communicants. Chaque partie du système est simulée, éventuellement sur une machine distante pour respecter la délocalisation des groupes de travail, mais surtout par un simulateur spécifique à son domaine d'application. La cosimulation multiniveaux peut être de deux types. Le premier est la cosimulation fonctionnelle. Elle permet une validation de l'interconnexion des sous-systèmes entre eux. Le second est la cosimulation temporelle. Elle permet d'ajouter à la cosimulation fonctionnelle une synchronisation en temps des différents simulateurs.

L'interface de cosimulation a été utilisée avec succès sur des applications industrielles, tout particulièrement sur des exemples du monde automobile chez PSA (Peugeot-Citroën). Au niveau multilangage, elle permet actuellement d'inclure des langages tels que SDL, COSSAP, MATLAB, SABER, VHDL et le C, bénéficiant ainsi d'une variété de langages pour la spécification.

Multilevel Co-simulation In a Multilanguage Design Flow

Abstract : Nowadays, the realization of an heterogeneous electronic system composed by different sub-systems (software, hardware, mechanical parts, etc...) begins with a high level specification which is then divided into different parts often modelized in different languages. These new methods thus imply new co-design and validation techniques so as to reduce the time to market. The verification by simulation of such systems consists in the joined simulation of the different parts of the system by using simulators which are appropriate to the type of module. This kind of simulation is called co-simulation and must be usable at each abstraction level of the specification.

The subject of this thesis concentrates on the study of a validation tool based on co-simulation. The aim of this tool is to verify a complete system all through its design flow by simulation implying the parallel and communicating execution of simulators. Each part of the system is, if necessary, simulated on a remote machine so as to maintain the delocalization of the working groups. The multilevel co-simulation may be of two types : the first one is the functional co-simulation which enables the validation of the interconnection of the sub-systems. The second one is the timed co-simulation which allows the addition of a temporal synchronization between the simulators.

The co-simulation environment has been successfully used on industrial applications, and particularly on mechatronic examples. It was, for instance, the case at PSA (Peugeot-Citroën). At the multi-language level, co-simulation currently permits the introduction of such languages as SDL, COSSAP, MATLAB, SABER, VHDL and C. Thanks to co-simulation, designers benefit from many languages for the validation of their specification.

Le Marrec Philippe, Juin 2000, 132 pages.

(Version papier.) ISBN 2-913329-49-7
(Version électronique) ISBN 2-913329-50-0