Prototype virtuel pour la génération des architectures mixtes logicielles/matérielles

C. Valderrama

Laboratoire TIMA

THÈSE

No attribué par la bibliothèque

|__|__|__|__|__|__|__|__|__|__|

pour obtenir le grade de

DOCTEUR de **l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE**

Discipline : **Micro-électronique**

Présentée et soutenue publiquement

par

Carlos Alberto VALDERRAMA

Le 29 Octobre 1998

PROTOTYPE VIRTUEL POUR LA GENERATION DES ARCHITECTURES MIXTES LOGICIELLES/MATERIELLES

Directeur de thèse : Ahmed Jerraya

Jury :

Messieurs:	Bernard Courtois	*Président*
	Jean Paul Calvez	*Rapporteur*
	Eduardo Sanchez	*Rapporteur*
	Pierre Paulin	*Examinateur*
	Ahmed Jerraya	*Examinateur*

Thèse préparée au sein du Laboratoire TIMA à Grenoble 46, Avenue Félix Viallet, 38031 Grenoble.

Résumé

L'objectif de ce travail de thèse est le développement d'une méthodologie pour la génération rapide d'architectures flexibles et modulaires pour les systèmes distribués. Cette approche, appelé aussi "prototypage virtuel", est une étape essentielle dans le processus de conception conjointe des systèmes mixtes logiciel/matériel. Les approches de recherche dans ce domaine sont motivées par le besoin urgent de prototypes pour valider la spécification, par la disponibilité des outils et des environnements de synthèse pour les parties logicielles et matérielles.

Le prototypage virtuel permet à la fois la manipulation du domaine logiciel ainsi que du domaine matériel. Il prend en entrée une architecture hétérogène composée d'un ensemble de modules distribués issu du découpage matériel/logiciel et génère des descriptions exécutables pour des éléments matériels et logiciels.

Ce travail décrit une stratégie de prototypage virtuel pour la co-synthèse (génération des modules matériels et logiciels sur une plate-forme architecturale) et la co-simulation (c'est-à-dire la simulation conjointe de ces deux composants) dans un environnement unifié. Ces travaux définissent également le développement d'un environnement de co-simulation distribué et flexible permettant l'utilisation de différents outils de simulation, de langages, la génération de modèles matériels et logiciels synthésisables et l'ordonnancement des modèles multiprocesseurs sur une architecture monoprocesseur.

Cette approche, présentée dans la conférence ED&TC, a obtenu le prix de l'année 1995. Des outils ont été mis en pratique dans l'environnement de conception conjointe Cosmos. Ce travail a aussi fait l'objet d'un transfert de technologie au profit de SGS-Thomson Microelectronics. Les outils développés au cours de cette thèse ont été utilisés pour les projets Européens COMITY (particulièrement utilisé par l'Aérospatiale Missiles à Toulouse et Intracom en Grèce) et CODAC, et par d'autres groupes comme le FZI de l'université de Tübingen et PSA à Paris.

Mots Clés : synthèse au niveau système, conception conjointe matériel/logiciel, co-simulation, prototypage, prototypage virtuel.

Abstract

The objective of this work is to develop a methodology for the generation of flexible and modular architectures for distributed systems. This approach (also called " virtual prototyping ") is an essential stage in the process of joint design (codesign) of mixed software/hardware systems.

Virtual prototyping takes as input a heterogeneous architecture made up of a whole of distributed modules resulting from software/hardware partitioning. It generates executable descriptions for software and hardware elements. Research approaches in this field are justified by the evolution of technology, the urgent need for prototypes to validate the specification, and by the availability of tools and synthesis environments for the design of software and hardware parts.

One of the major difficulties of virtual prototyping is that it allows at the same time to handle both, software and hardware. This work describes a strategy of virtual prototyping for the cosynthesis (generation of the modules material and software on an architectural platform) and cosimulation (i.e. the joint simulation of these two kind of components) in a unified environment, the development of a distributed and flexible cosimulation environment allowing the use of several simulation tools and languages, the generation of hardware/software synthesizable models and mono-processor architecture software generation for a set of communicating processes.

This approach, presented in the ED&TC conference, got the best paper award in 1995. The tools developed during this thesis were put into practice in the Cosmos codesign environment. One of them was transferred to SGS-Thomson Microelectronics. The tools were also used for the Europeans projects COMITY (particularly used by Aerospace the Missiles in Toulouse and Intracom in Greece) and CODAC, and by other groups like the FZI of the university of Tübingen and PSA in Paris.

Key Words: system level synthesis, hardware/software codesign, cosimulation, prototyping, virtual prototyping.

Remerciements

Ce travail a été effectué au sein du laboratoire des Techniques de l'Informatique et de la Micro-électronique pour l'Architecture d'ordinateurs (TIMA). Je tiens à témoigner ma reconnaissance à tous ceux qui m'ont aidé durant ces années, dans le groupe System Level Synthesis du TIMA, et plus spécialement à Philippe Lemarrec, Sabiha Arab, Christian et Christine Lenne, François Naçaval et tout particulièrement à Monsieur Ahmed Jerraya pour son aide dans l'orientation de mon travail, et son précieux soutien. Par ailleurs, j'adresse mes plus sincères remerciements à tous les membres de l'équipe qui ont accepte de lire et corriger les diverses versions de cette thèse.

Je tiens à remercier tout particulièrement Dieu pou m'avoir accompagné pendant tout ce séjour en France. Ma famille et moi tenons également à témoigner toute notre gratitude à nos frères français qui nous ont accueillis avec amitiés en leur pays magnifique, la France, que nous n'oublierons jamais. Je remercie aussi le gouvernement Brésilien, en particulier aux organismes CAPES et COFECUB pour avoir accepté de financer l'ensemble de mes travaux.

Enfin, merci à tous les miens de m'avoir soutenu tout au long de cette période de travaille et d'étude, et à qui je souhaite dédier cette thèse.

A Dieu,
A ma Famille

Sommaire condensé

Table des matières

Liste de figures

Chapitre 1

Introduction

Ce chapitre d'introduction définit les motivations et les objectifs de cette thèse. Il présente une vue d'ensemble des problèmes existants lors de la conception conjointe de systèmes mixtes logiciels/matériels et plus particulièrement le prototypage virtuel. Ce chapitre situe la contribution de ce travail. L'organisation de cette thèse est fournie à la fin du chapitre.

1.1 Motivation

Au cours de ces dernières années, les techniques de développement des circuits intégrés (ASICs) ont suivi une évolution très importante. Parallèlement, l'amélioration des architectures matérielles a pu être constatée, concernant en particulier celles contenant des processeurs ou des éléments programmables (comme FPGA, PLD).

Un système est souvent formé par une collection d'ensembles matériels (circuits) et de logiciels (code exécutable) en étroite interaction. Si ce système est purement matériel, il présente des garanties de faible surface, de faible consommation et de performances, alors que la flexibilité, le temps de développement et les possibilités de réutilisation sont ses points faibles [1]. Ainsi, un tel système coûte très cher quand il faut le faire évoluer. A l'opposé, une solution logicielle, à base de processeurs standards, offre un degré maximum de flexibilité, ainsi qu'un temps de développement très réduit puisqu'aucune conception physique n'est nécessaire. De plus, les outils de développement sont disponibles et faciles d'emploi [2][3]. La haute densité d'intégration permet de réunir sur une même puce des circuits spécifiques et des processeurs.

L'utilisation conjointe de processeurs dont les performances atteignent aujourd'hui des niveaux très élevés, et de circuits spécialisés nécessite de nouvelles méthodes de conception. Ces méthodes doivent en particulier permettre au concepteur de faire le découpage d'un système afin de trouver le meilleur compromis entre une réalisation logicielle et une réalisation matérielle. Généralement les parties critiques, qui nécessitent un temps de réponse rapide (comme c'est le cas des applications temps réel), sont réalisées en matériel par des ASICs ou des FPGAs ; les parties moins critiques sont réalisées en logiciel pour être exécutées par des microprocesseurs ou micro-contrôleurs.

La conception conjointe de logiciel/matériel contribue à la réalisation de systèmes fiables en utilisant une validation appliquée au modèle du système entier. Ce type de conception permet aussi d'augmenter la productivité en permettant la conception simultanée des parties logicielles et des parties matérielles.

1.2 Conception conjointe de logiciel/matériel

La conception de systèmes contenant à la fois du logiciel et du matériel n'est pas nouvelle. Cependant, le développement conjoint du logiciel et du matériel est un domaine de recherche récent. Pour maîtriser la complexité croissante des systèmes et pour répondre aux critères de performances attendus, il est important d'aborder la conception des systèmes mixtes en terme de conception conjointe de matériel/logiciel [92][93]. Ce type d'approche agit aussi au plus haut niveau d'abstraction, c'est à dire le niveau système.

Le processus de conception est un ensemble de tâches qui transforment un modèle en une architecture. Le modèle décrit le fonctionnement du système et l'architecture décrit la façon dont il sera réalisé. Au début de ce processus, seules les fonctionnalités du système sont connues. Le travail du concepteur consiste donc à décrire cette fonctionnalité par des langages qui sont basés sur le modèle le plus approprié. La puissance d'expression de ces langages est très dépendante du type d'application considéré. Les descriptions qui sont données en entrée peuvent être soit dans un langage de description matériel (VHDL, Verilog, HardwareC, etc.), soit logiciel (C ou assembleur) ou encore dans un langage de description au niveau système (SDL, StateCharts, CSP, SpecCharts, etc.).

Une approche typique de synthèse part d'un langage de spécification de niveau système, puis réalise une répartition des fonctions entre logiciel et matériel ; ensuite les blocs résultants sont synthétisés. La synthèse au niveau système peut être séparée en deux activités principales qui sont le

découpage et la synthèse de communication. Le découpage de la spécification d'un système génère un ensemble de partitions (modules matériels, modules logiciels, et blocs de communication) qui seront transposées sur une architecture cible. La synthèse de communication permet de raffiner les interfaces des sous-systèmes communicants [89]. Etant donné que les processus des diverses partitions peuvent dialoguer, cette étape doit définir non seulement les interfaces d'entrée/sortie entre les partitions, mais aussi les protocoles qui coordonnent la communication entre les processus. Par exemple, elle permet la définition d'un modèle de communication fixe (par exemple, communication point-à-point), communication à travers une mémoire partagée, ou communication avec un protocole qui peut avoir différents degrés de complexité.

Nous nous intéressons au développement d'une méthodologie unifiée de conception conjointe logiciel/matériel (appelée codesign), permettant la liaison des outils de codesign (comportant le partitionnement et la synthèse de communication) et des outils spécifiques de synthèse logicielle/matérielle. Cette liaison, qui est établie par une étape intermédiaire appelée prototypage virtuel de systèmes hétérogènes, est constitué une partie importante du processus de cosynthèse. Elle vise à produire une architecture hétérogène ou mixte, composée de parties logicielle et matérielle, qui sert à l'implantation de la spécification initiale. La partie matérielle peut être décrite en langage VHDL comportemental pour ensuite être synthétisé sur un ASIC ou FPGA. La partie logicielle destiné à un processeur sera décrite en C.

1.3 Objectifs

L'objectif principal de cette thèse est de définir une approche de conception de systèmes mixtes logiciels/matériels communiquant entre-eux.

Ce travail décrit une stratégie de modélisation permettant l'utilisation du prototype virtuel pour la cosynthèse (la génération des modules logiciels et matériels sur une plate-forme architecturale) et la cosimulation (la simulation conjointe des composants logiciels et matériels) dans un environnement unifié. Le but est de maîtriser la complexité croissante des systèmes ainsi que d'accélérer le processus de conception.

Le développement d'une méthodologie flexible et modulaire pour le prototypage rapide de systèmes électroniques mixtes, logiciel et matériel, débute avec un prototype virtuel, c'est-à-dire une architecture hétérogène composée d'un ensemble de modules distribués matériels et logiciels. Pendant le processus de conception, chaque module nécessite plusieurs étapes de raffinement à l'aide d'outils spécifiques qui génèrent des spécifications intermédiaires. Chacun d'eux devra être validé par la cosimulation avec le reste du système [6].

Le prototypage virtuel est une étape qui génère des descriptions exécutables pour chaque processus issu du partitionnement. Ces processus doivent être décrits dans des langages de spécification correspondant aux modules logiciels (en langage C) et matériels (en VHDL) respectivement [7][8]. Une telle approche requière des cosimulations hétérogènes (C/ VHDL) pour divers niveaux (niveau comportemental, cycle d'horloge et logique). Evidemment, pour assurer la cohérence de la conception, ce même modèle C/ VHDL doit être utilisé pour la cosimulation et la cosynthèse.

La méthodologie proposée considère les concepts et problèmes suivants :

- Le développement d'un modèle unifié pour la cosimulation et la cosynthèse : la définition du modèle et de l'environnement comprend : la spécification des modules matériels et logiciels à l'aide de langages dédiés, la communication entre les modules logiciels et matériels, la cohérence entre les résultats de cosimulation et de cosynthèse, le support de plate-formes multiples orientées

vers la cosimulation, la cosynthèse et la réutilisation. Le but est la génération automatique de C et VHDL en partant des résultats du partitionnement.

- Le développement d'un environnement de cosimulation distribué et flexible : la cosimulation distribuée des modules logiciels et matériels nécessite l'utilisation de différents outils de simulation et de langages de description, l'exécution concurrente d'outils de mise au point des parties matérielles et logicielles, l'abstraction de la communication matériel-logiciel, et la génération automatique d'interfaces permettant l'utilisation transparente du lien entre les environnements de simulation de matériel et logiciel.
- Génération de modèles matériels pour la synthèse : transposition des modules matériels pour la synthèse d'architecture.
- Génération de modèles logiciels pour la compilation : l'ordonnancement des modèles multiprocesseurs sur une architecture monoprocesseur. Contrairement aux modèles matériels, la synthèse de logiciel implique l'exécution séquentielle de processus auparavant concurrents.

1.4 Contribution

Le travail de cette thèse fait partie d'un projet de codesign appelé Cosmos. La contribution de cette thèse se situe au niveau du prototype virtuel. Les principales contributions de cette thèse sont :

- La spécification et le développement d'un outil qui génère automatiquement des descriptions en langage C et VHDL en partant du langage intermédiaire SOLAR (le format intermédiaire de COSMOS) et la définition des modèles générés pour obtenir un environnement unifié et flexible pour le prototypage virtuel et la synthèse. Les descriptions matérielles sont générées en vue de la synthèse architecturale.
- La conception, le développement et l'application d'un outil de génération d'interface C-VHDL pour la cosimulation, nommé VCI (VHDL-C interface). La cosimulation, autrement dit la simulation distribuée des modules logiciels et matériels, exige la création d'un lien entre les outils de simulation basés sur des modèles d'exécution différents [9]. En partant d'une description abstraite de l'interface entre le matériel et le logiciel, l'outil VCI génère un environnement de cosimulation fiable et personnalisé pour une application spécifique. L'approche proposée dans cette thèse a été validée par un projet SGS-Thomson Microelectronics, pour la re-synthèse du "single-chip Videotelephone Codec" (STi1100 [6]) et une version industrielle de VCI fait actuellement partie de l'ensemble des outils de CAO de ST Microelectronics.

Un premier prototype de cette approche a été mis en pratique dans l'environnement de conception conjointe Cosmos. Actuellement, d'autres thèses sont en cours au sein du laboratoire. Elles constituent une continuation des travaux présentés dans cette thèse. Ces travaux offrent, pour une partie, la possibilité d'étendre le prototypage virtuel à des langages autres que C et VHDL et en outre, de pouvoir valider le prototype à chacune des étapes du processus de codesign.

1.5 Plan de la thèse

Le chapitre 2 présente les caractéristiques des environnements de codesign, des langages de spécification et des architectures d'implantation. Ce chapitre donne une meilleure compréhension des problèmes associés au prototypage virtuel (étape intermédiaire entre la spécification niveau système et la génération de l'architecture cible). L'approche de prototypage virtuel est basée sur le choix du style de la spécification en entrée, le domaine d'application, le modèle de l'architecture cible, et les étapes de synthèse. Les techniques traditionnelles utilisées pour les systèmes de conception conjointe

logiciel/matériel ainsi que l'état de l'art des environnements de codesign seront décrits brièvement afin de dégager les points communs et les différences entre elles. Les caractéristiques des langages seront détaillées. Ensuite, une comparaison entre ces langages de spécification sera réalisée. Les architectures d'implantation sont classées et comparées pour relever les caractéristiques et les domaines d'application de chacune d'elles.

Le chapitre suivant présente des outils développés dans le cadre de ce travail. Le chapitre 3 décrit l'environnement de cosimulation qui a été développé pour permettre la vérification des systèmes mixtes matériel/logiciel. Seule la génération d'interface C-VHDL sera commentée. La génération des modules C et VHDL sera présentée dans le chapitre 4. Cette génération sera illustrée par un exemple dans l'annexe A.

Le chapitre 4 présente le prototypage virtuel pour les systèmes mixtes matériel/logiciel. Il décrit une méthodologie pour le prototypage rapide des systèmes électroniques fortement modulaires et flexibles comprenant le logiciel et le matériel. Il détaille la génération du prototype virtuel en partant de la répartition des fonctionnalités d'un système entre le logiciel et le matériel jusqu'à sa réalisation sur une architecture comportant des processeurs et des composants matériels.

Le chapitre 5 présente l'approche de conception conjointe de systèmes mixtes logiciel/matériel qui est le cadre dans lequel s'est déroulée cette thèse. Le chapitre donne une vue globale de l'environnement Cosmos de conception conjointe de logiciel/matériel. Il décrit les différentes phases d'une conception en partant d'une spécification jusqu'à sa réalisation sur une architecture qui comporte des processeurs et des composants matériels.

Le chapitre 6 présente les conclusions et les perspectives attachées aux travaux menés dans le cadre de cette thèse, les aspects, qui commencent à devenir d'actualité et qui n'ont pas été traité dans cette thèse, seront discutés.

Plusieurs détails relatifs aux travaux menés dans le cadre de cette thèse seront présentés en annexe. L'annexe donne quelques exemples décrits dans les différents langages utilisés pour le prototypage virtuel, en particulier le langage intermédiaire SOLAR, C et VHDL.

Chapitre 2

Conception de systèmes mixtes logiciel/matériel

Ce chapitre contient une analyse des techniques traditionnelles utilisées pour les systèmes de conception conjointe logiciel/matériel, ainsi que l'état de l'art des outils existants. Pour mieux comprendre les concepts et les choix fait aux cours de cette thèse, ce chapitre présente les caractéristiques des environnements de conception, les langages de spécification et les architectures mises en œuvre.

2.1 Introduction

Le but de ce chapitre est de présenter l'état de l'art des environnements de conception conjointe logiciel/matériel [1]. Les méthodologies proposées pour le codesign diffèrent selon les choix stratégiques suivants : le style de la spécification en entrée, le modèle de l'architecture cible et les étapes de synthèse. Ce chapitre donne une meilleure compréhension des problèmes associés au prototypage virtuel par rapport à ces trois sujets et les choix faits au cours de cette thèse.

Le processus de conception est un ensemble de tâches qui transforme un modèle de haut niveau en une architecture. Au début de ce processus, seules les fonctionnalités du système sont connues. A chaque étape du processus de conception, l'architecture commence à émerger avec l'ajout de détails supplémentaires. Le travail du concepteur consiste donc à décrire cette fonctionnalité à l'aide de langages basés sur le modèle le plus approprié. La conception concurrente matérielle/logicielle est une approche qui intègre dans un même environnement la conception du matériel et celle du logiciel. Les modèles qui sont donnés en entrée peuvent être décrits soit dans un langage de description de matériel (VHDL, Verilog, HardwareC, etc.), soit de logiciel (C ou assembleur), soit dans un langage de description au niveau système (SDL, StateCharts, CSP, SpecCharts, etc.). Le langage utilisé pour le cahier des charges représente un compromis entre la puissance d'expression et la facilité de la mise en place. Cette puissance d'expression est très dépendante des types d'applications considérés.

La synthèse au niveau système peut être résumée en deux activités principales qui sont le découpage logiciel/matériel et la synthèse de communication. Le découpage de la spécification d'un système génère un ensemble de partitions (puces, composants matériels, modules logiciels, blocs de communication) qui seront transposées sur une architecture cible. La synthèse de la communication permet de raffiner les interfaces des sous-systèmes communicants [11]. Elle permet, par exemple, la définition des protocoles de communication ou de l'interface d'entrée/sortie de chaque partition. Les approches de synthèse qui seront présentées se basent sur l'un des modes de communication suivants : communication fixe (par exemple, communication point-à-point), communication à travers une mémoire partagée, et communication avec un protocole qui peut avoir différents degrés de complexité. La plupart de ces approches réalisent un découpage sur un graphe du flux de contrôle et de données ; la communication étant alors limitée par la simplicité des protocoles considérés.

L'architecture est destinée à supporter un modèle en spécifiant sa mise en œuvre. Le modèle décrit le fonctionnement du système; par contre, l'architecture décrit sa réalisation. Généralement, le concepteur se rend compte que, en fonction du domaine d'application, certaines architectures sont plus efficaces que d'autres pour l'implantation de certains modèles. Pour cela il est nécessaire de considérer plusieurs possibilités de mise en œuvre avant que le processus de conception aboutisse à sa fin. Ainsi, l'architecture cible devra être flexible, modulaire et devra maximiser la réutilisation de composants existants.

En général, deux types d'architectures peuvent être pris en compte : les architectures basées sur une unité de contrôle centrale et les architectures distribuées. Pour mettre en relief les caractéristiques et les domaines d'applications de chacune, une description et une analyse des architectures sont présentées. On s'intéressera plus précisément aux architectures mixtes logiciel/matériel supportées dans les environnements de conception conjointe. Ces architectures comportent des modules matériels et des modules logiciels.

2.2 Systèmes de Conception Conjointe Logiciel/Matériel

Une approche typique de synthèse part d'un langage de spécification de niveau système pour

réaliser une répartition des fonctions entre logiciel et matériel et, ensuite, synthétiser la communication entre les blocs résultants (figure 2.1). L'étape suivante consiste en la génération de code qui est suivie de la compilation du logiciel et de la synthèse comportementale des parties matérielles [12][13]. Le prototypage virtuel, qui est une étape intermédiaire entre la spécification au niveau système et la génération de l'architecture cible, établit un lien entre les outils du codesign et les outils spécifiques pour la synthèse logicielle/matérielle.

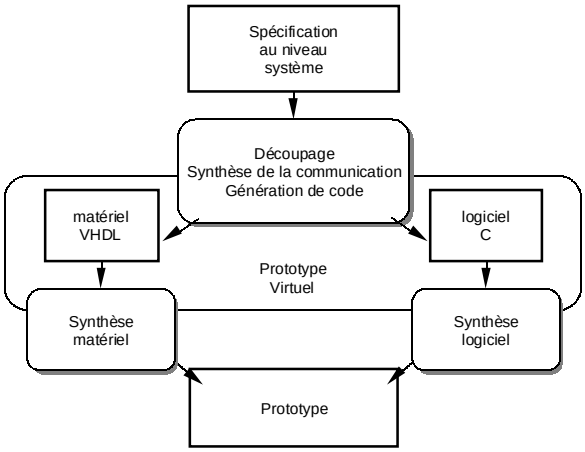


figure 2.1 : Approche typique de synthèse

La conception mixte fournit au concepteur des méthodes et outils pour explorer plusieurs possibilités de découpage du système et évaluer les performances des partitions [14]. L'analyse des compromis de mise en œuvre, tant au niveau matériel qu'au niveau logiciel, est très utile pour permettre la sélection de l'architecture du système qui s'accommode aux mieux des contraintes diverses (coûts de développement, contraintes temps réel, fiabilité, etc.) [15]. Par ailleurs, la combinaison dans une même représentation des ensembles matériels et logiciels nécessite un modèle ou un langage permettant d'unifier les sémantiques associées à ces ensembles. Certaines méthodes de conception mixte utilisent un seul langage pour la description de tout le système. D'autres utilisent à la fois un langage pour le logiciel et un langage pour le matériel et essaient de les intégrer dans un même environnement [16]. Ainsi, l'un des problèmes inhérents à la conception mixte matérielle/logicielle, et auquel il faut souvent faire face, est la modélisation de l'interaction entre plusieurs ensembles matériels et logiciels.

Le cycle de développement d'un système logiciel/matériel est présenté dans la figure 2.2. Le point de départ est toujours un cahier des charges qui correspond à une formulation des besoins. Les informations fournies concernent l'application dans son ensemble et expriment les objectifs souhaités. Une fois le cahier des charges défini, le système à concevoir doit être décrit selon une vue purement externe. La spécification inclut toutes les contraintes que doit satisfaire ce système. La phase de conception générale exprime la structure du système sur le plan fonctionnel. L'organisation interne et le comportement de chacune des fonctions sont explicités. La phase de conception détaillée a pour objectif de trouver des schémas d'implantation pour le logiciel et les structures du matériel qui sont nécessaires. Ensuite, la phase de réalisation décrit la solution finale en termes de logiciel et de matériel. Elle conduit à un système opérationnel.

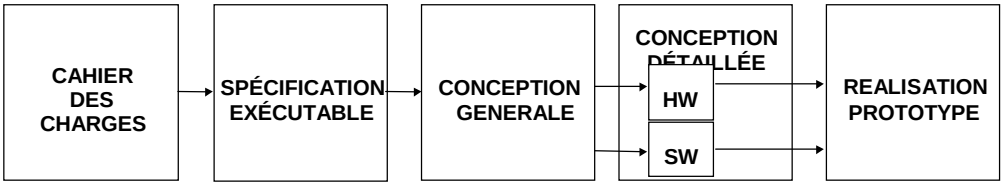


figure 2.2 : Etapes du cycle de développement d'un système

Une approche de niveau système est souvent invoquée naturellement en tant que condition nécessaire à la maîtrise de la complexité. Dans le contexte des systèmes électroniques complexes, ceci se traduit par une approche de développement simultané du matériel et du logiciel [17]. Parmi les avantages d'une telle approche, on peut citer celui du compromis réaliste entre matériel et logiciel qui répond au mieux aux performances souhaitées, où celui d'une meilleure utilisation des possibilités de la technologie actuelle en ce qui concerne les répartitions matérielles/logicielles.

Plusieurs projets (Cosyma, SpecSyn, MCSE, Cosmos) sont en train d'être développés afin de créer des environnements de codesign. Certaines approches proposent de partir d'une spécification système type (SDL, StateChart, Esterel). Ces langages offrent la possibilité de spécifier un système à un haut niveau d'abstraction où les parties logicielles et matérielles ne sont pas déterminées. A partir de ces spécifications, on procède à des étapes de découpage et d'allocations pour aboutir à un modèle abstrait composé de parties matérielles et logicielles communicant à travers des unités de communication. Les parties matérielles sont généralement spécifiées en VHDL, mais la description des parties logicielles varie d'un environnement à un autre. De plus, d'autres projets en cours se distinguent par des choix pragmatiques en limitant le flot de conception et/ou en limitant le domaine d'application. Parmi ces projets on peut citer RASSP [20] et CoWare [18]. La suite de cette section introduit quelques méthodologies de conception de systèmes mixtes logiciels/matériels.

2.2.1 RASSP

Le projet RASSP [15] propose une approche pragmatique qui consiste à spécifier les parties logicielles et matérielles à l'aide d'un langage de description matériel (VHDL). Cette méthodologie a pour cible la réalisation des applications spécifiques dédiées au traitement du signal. Le modèle d'architecture est composé d'un processeur numérique du signal (Digital Signal Processor) préalablement spécifié au niveau transfert de registre (RTL) et des modules matériels. La communication entre les deux parties est pilotée par le processeur. Une bibliothèque fournit les modèles de description de l'architecture du processeur. Le modèle ISA (Instruction Set Architecture) décrit au niveau cycle d'horloge toutes les instructions que le processeur peut traiter. Le modèle FBM (Full Behavioral Model) décrit le fonctionnement du circuit et spécifie son interface externe.

La particularité de la méthodologie du projet RASSP consiste en l'utilisation de modèles très détaillés du processeur qui exécute le code logiciel. Cela permet d'avoir une simulation très précise au cycle d'horloge près. Néanmoins, cette approche présente quelques limitations. En effet, RASSP n'offre pas d'outil de découpage logiciel/matériel efficace, et l'allocation logiciel/matériel est effectuée manuellement. Pour les parties logicielles, le langage VHDL n'est pas approprié pour la spécification du code des processeurs logiciels.

2.2.2 LIRMM

Un environnement d'aide à la conception conjointe de systèmes dédiés matériels/logiciels est en cours de développement au LIRMM (Montpellier) [19]. Il s'agit d'une méthodologie interactive qui vise essentiellement le prototypage des systèmes hétérogènes. Les applications traitées touchent principalement le traitement d'images en temps réel. Les descriptions d'entrée sont de haut niveau pour le matériel, et en code assembleur pour le processeur de traitement du signal. Le découpage logiciel/matériel est effectué manuellement suivant les performances requises pour chaque module. L'architecture cible est construite autour d'un noyau de processeur de traitement du signal, une mémoire et des modules matériels programmables. Le résultat de la conception est validé sur une carte de prototypage à base des FPGAs.

Cet environnement se distingue par la conception des interfaces de communication. Ces

interfaces intègrent le protocole de contrôle des sous modules du système. La réutilisation des modules est la principale contrainte de conception dans cet environnement.

2.2.3 MCSE

L'approche MCSE (Méthode et modèle de Conception des systèmes Electroniques) définit une méthodologie complète qui couvre toutes les étapes de développement des systèmes [153]. La spécification d'entrée comprend des spécifications fonctionnelles, exprimées dans un langage graphique fondé sur le modèle de processus communicants, et des spécifications non fonctionnelles exprimées sous forme de contraintes (temps, performances, technologie, etc.). Le partitionnement repose sur les estimations de performances recueillies à partir du modèle de performances de MCSE. Le modèle, dit non-interpreté, est formé de deux parties : la structure et le comportement. Le modèle structural résulte de la composition de une structure fonctionnelle et de l'architecture matérielle donnée par le processus d'allocation. Le modèle comportemental de chaque fonction est une abstraction du comportement algorithmique. Les attributs et les paramètres associés à ce modèle indiquent les propriétés de tous les composants. Ce modèle de performances est utilisé pour la vérification du système dans une étape de cosimulation matériel/logiciel au niveau modèle (simulation VHDL dite non interprétée) [154]. Des simulations en VHDL et en C++ sont également possibles. Elles permettent d'assister l'activité de partitionnement matériel/logiciel. Dans une deuxième phase, le modèle de performances et l'architecture matérielle sont employés pour obtenir par la synthèse les descriptions matérielles et logicielles. Les deux descriptions sont utilisées, pour une vérification finale, par une cosimulation interprété détaillé.

2.2.4 Ptolemy

L'environnement de codesign Ptolemy, de l'université de Berkeley, permet le développement d'applications de traitement du signal et de systèmes communicants [21][22][62]. Le modèle initial est formé par un ou plusieurs programmes, s'exécutant sur des composants programmables. Au niveau système, cette description peut inclure des contraintes de conception telles que : les contraintes temps-réel, la vitesse, la surface, la taille du code, la consommation, etc. L'architecture cible comprend une variété de processeurs qui peuvent avoir une configuration parmi plusieurs : système monoprocesseur, architectures parallèles à mémoire partagée, avec bus partagé ou avec passage de messages.

L'approche est formée de trois étapes : le découpage matériel/logiciel, la synthèse du logiciel et du matériel et la synthèse des interfaces matérielles/logicielles. Le découpage, guidé par les contraintes d'entrée, essaie d'optimiser des fonctions de coûts telles que le coût des communications, la surface ou la vitesse. La simulation du système a lieu une fois que la synthèse du logiciel et du matériel est faite [23].

2.2.5 Cosyma

L'environnement de codesign Cosyma, de l'université de Braunschweig, est dédié aux systèmes complexes [24][25]. La description en entrée est une spécification textuelle en langage C^x. Le langage C^x est une extension au langage C permettant d'inclure des contraintes de temps et le parallélisme entre les processus. L'architecture cible utilise un seul processeur avec un seul circuit intégré (ASIC), une mémoire et un bus.

La description d'entrée est traduite dans une représentation interne appelée graphe syntaxique étendu. Ensuite, une simulation préliminaire est réalisée pour extraire les informations nécessaires au découpage. Le découpage matériel/logiciel est automatique et basé sur un algorithme de recuit simulé (ang. "simulated annealing"). Les parties à réaliser en logiciel sont traduites en langage C et les parties

à réaliser en matériel sont traduites en langage HardwareC.

2.2.6 CoWare

L'environnement nommé CoWare permet la conception des systèmes distribués comportant plusieurs processeurs logiciels programmables du type DSP (Digital Signal Processor) ou micro-contrôleurs, et des modules matériels dédiés [26][27]. L'architecture utilisée permet d'avoir un modèle composé de processeurs interconnectés par des canaux de communications du type point-à-point. La communication entre les modules utilise un protocole rendez-vous, qui est synchronisé par des signaux d'attentes. Un module logiciel est constitué par un cœur de processeur programmable, une mémoire et une unité d'E/S qui assure la communication matérielle avec l'environnement. Les composants sont décrits en langage C (pour les modules logiciels) et en langage VHDL (pour les parties matérielles). Les modules utilisés sont généralement des composants existants rassemblés pour former une architecture multiprocesseur. Cet environnement permet la génération automatique d'interfaces et la cosimulation à plusieurs niveaux d'abstraction.

2.3 Langages de spécification

Cette section discute les langages utilisés pour la conception des systèmes mixtes logiciel/matériel. Le choix du langage le plus approprié pour définir le cahier des charges est une tâche importante, car il y a une pléthore de langages de spécification [29]. Le choix d'un langage résulte généralement d'une pondération de plusieurs critères tels que la puissance expressive du langage, les capacités d'automatisation fournies par le modèle sous-tendant le langage et de la disponibilité des outils et méthodes supportant le langage. D'ailleurs, pour une application donnée, plusieurs langages peuvent être utilisés pour la spécification des divers modules qui font partie du système.

La spécification d'un système mixte logiciel/matériel peut suivre une des deux approches :

- Spécification homogène : un langage unique est utilisé pour la spécification du système global comprenant des parties matérielles et des parties logicielles.
- Spécification hétérogène : des langages spécifiques sont utilisés pour les parties matérielles et logicielles.

Chacune des deux stratégies de conception implique une organisation différente du processus de codesign, donc de son environnement.

2.3.1 Spécification homogène

Le cahier des charges d'une spécification homogène implique l'utilisation d'un langage unique pour spécifier le système global. Un environnement générique de codesign basé sur un modèle homogène est schématisé dans la figure 2.3. Le système est décrit comme étant un ensemble de fonctions et de contraintes, et cela indépendamment de toute considération matérielle ou logicielle. A ce niveau, un langage de spécification système peut convenir, ainsi que toute autre représentation fournissant un modèle exécutable du système. Le cahier des charges passe par une étape de découpage en parties logicielles et matérielles, visant à obtenir une première approche de l'architecture cible. Le résultat qui en découle, le prototype virtuel, est une architecture faite de composants matériels et logiciels. La dernière étape, qui consiste à effectuer la synthèse du matériel et la génération de code pour les processeurs, produit un premier prototype du système.

Dans de tels environnements de codesign, la question clef est la correspondance entre les concepts utilisés dans le modéle initial et les concepts fournis par l'architecture cible. Par exemple, établir la correspondance entre la spécification au niveau système, comprenant des concepts de très

haut niveau tels que le contrôle distribué ou la communication, et la représentation faite par des langages de bas niveau tels que C et VHDL est une tâche non négligeable [8][30][31].

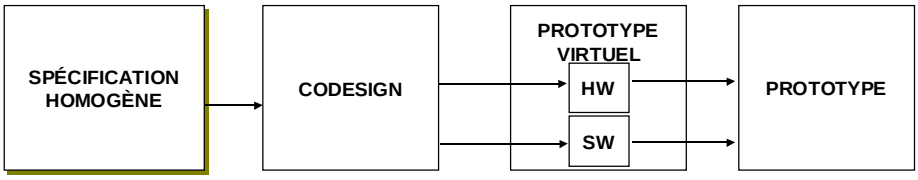


figure 2.3 : Flot de codesign pour une spécification homogène

Plusieurs environnements de codesign suivent cette option. Néanmoins, afin de réduire la distance entre le modèle de spécification et le prototype, ces outils démarrent par un modèle de spécification de bas niveau par rapport au niveau système. Certaines de ces approches ont étendu des langages mono-flôt pour supporter des concepts matériels et de la communication. L'environnement de codesign Cosyma accepte une spécification en langage C^x extension du langage C [24][25]. La description en entrée de l'environnement Vulcan, de Standford, est une autre extension du langage C appelée HardwareC [32][41][57]. Lycos et Castle utilisent le langage C [33][34][104]. D'autres outils de codesign utilisent le langage VHDL [35]. On peut noter que peu d'outils ont essayé de partir d'un modèle de niveau plus élevé. Dans ces derniers nous trouvons Polis [36] qui utilise Esterel [37], SpecSyn de Irvine [38][58] qui utilise SpecCharts [39], et l'environnement décrit dans [31] qui utilise LOTOS. Cosmos, qui utilise le langage SDL, appartient aussi à ce groupe.

Peu d'approches ont utilisé des langages de spécification de systèmes distribués tel que SDL, LOTOS ou ESTELLE [16][31][46]. Cela est dû principalement à la distance existant entre les concepts manipulés par ces langages et ceux des langages de description matérielle. En général, la traduction directe de tels langages est seulement possible en introduisant des restrictions sur le modèle d'exécution, les aspects dynamiques ou les modèles de communication disponibles. Dans [110], le modèle de communication Estelle est traduit en VHDL. Dans [31], les processus LOTOS sont traduits en automates d'états finis VHDL et chaque schéma de communication à une seule implémentation basée sur un module de synchronisation prédéfini extrait d'une bibliothèque VHDL. Dans [16], Glunz présente une approche de traduction SDL vers VHDL. Le modèle de communication SDL peut être changé à l'aide d'une bibliothèque de protocoles.

2.3.2 Spécification hétérogène

Le cahier des charges des systèmes à base de spécification hétérogène permet l'utilisation de langages spécifiques pour les parties matérielles et logicielles. Un environnement générique de codesign basé sur un modèle hétérogène est donné par la figure 2.4. Le codesign commence par un prototype virtuel. Dans ce cas, le codesign est une correspondance des parties logicielles et matérielles sur des processeurs dédiés. La partie logicielle, traditionnellement écrite en assembleur, est souvent décrite en langage C. L'utilisation de ce langage donne la possibilité de réutilisation du code et sa validation indépendamment du processeur cible [4][5]. Pour les parties matérielles, les deux langages de description les plus utilisés actuellement sont VHDL et Verilog [42][150]. Ces langages traitent les caractéristiques spécifiques de la conception de matériel et les niveaux d'abstraction, permettant l'ajout des détails résultant de la synthèse.

Les points clef avec un tel arrangement sont la validation de la spécification et la génération des interfaces. L'utilisation d'une description multi-language exige de nouvelles techniques de validation capables de manipuler de tels modèles. Au lieu de la simulation nous aurons besoin de cosimulation et au lieu de la vérification nous aurons besoin de coverification. Par ailleurs, le cahier des charges multi-language entraîne un problème d'interface entre sous-ensembles qui sont décrits par des langages

différents. Ces interfaces doivent être raffinées par rapport à l'architecture cible à partir de la spécification initiale. CoWare [18][27] est un environnement typique qui supporte un tel flot de codesign. L'environnement utilise une description mixte donnée en VHDL pour le matériel, et C pour le logiciel. Cet environnement tient compte de la cosimulation.

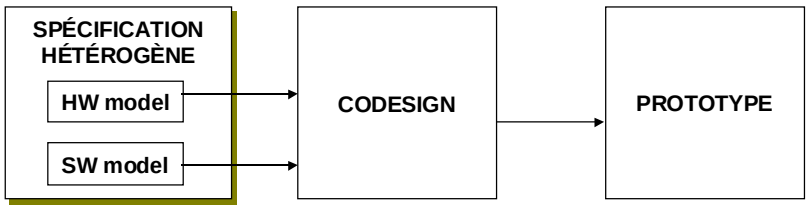


figure 2.4 : Flot de codesign pour une spécification hétérogène

2.4 Architectures logicielle/matérielle

Une architecture indique comment le système sera réellement mis en œuvre. Une architecture est constituée d'un ensemble de composants, d'une organisation et d'un modèle d'interaction entre les différents composants [56]. Le but de la conception d'architecture est de décrire pour un système donné le nombre de composants, le type de chaque composant, et le type de connexion entre ces divers composants. Les composants sont les éléments de base pour réaliser une application donnée. Dans le cas des architectures mixtes logicielles/matérielles on distinguera les composants logiciels, les composants matériels et les composants de communication.

Les architectures couvrent le large spectre des systèmes partant des simples contrôleurs jusqu'aux systèmes à processeurs parallèles hétérogènes. Généralement, certaines architectures sont plus efficaces pour l'implémentation de certains modèles ou pour supporter une gamme d'applications.

2.4.1 Organisation de l'architecture

L'organisation de l'architecture est définie par la composition ou l'assemblage d'éléments de base. Dans le cas des architectures mixtes logiciel/matériel on s'intéressera aux rôles joués par le contrôle global du système et par les différents éléments de base. On distinguera deux types d'organisation : l'architecture monoprocesseur et l'architecture multiprocesseur. Une architecture monoprocesseur est composée d'un processeur maître et d'un ensemble de composants matériels spécifiques. Dans ce cas, un seul processeur est responsable de l'état global du système. Il réalise le contrôle au niveau global et les autres éléments agissent en tant qu'esclaves. Dans le second type d'organisation, l'architecture est composée d'un réseau de processeurs autonomes communicants. Les processeurs agissent de manière indépendante. L'interaction entre les différents composants constitue la communication. Cette communication permet l'échange des informations et du contrôle. Ces échanges peuvent être synchrones ou asynchrones. Ils peuvent être aussi distribués ou gérés par des contrôleurs spécifiques. Une autre classification [61], basée sur le domaine d'application, permet de classer les architectures en : architectures d'application spécifique comme des systèmes de DSP, les processeurs d'usage universel tels que les RISCs, les architectures à processeurs parallèles tels que les VLIWs et les machines SIMD et MIMD.

Une architecture peut être représentée par un diagramme de composants interconnectés (figure 2.5). Sa représentation contient un ensemble de nœuds et un ensemble d'arcs. Les nœuds représentent les composants. Ils sont des objets définis par leur type et par le nombre d'entrées/sorties (Unité Arithmétique et Logique, microprocesseur, mémoire, unité d'entrée/sortie, ASIC, FPGA ou même des sous systèmes complexes). Les arcs sont les différents types de connexions entre ces composants (bus, fils ou même modules de communication sophistiqués).

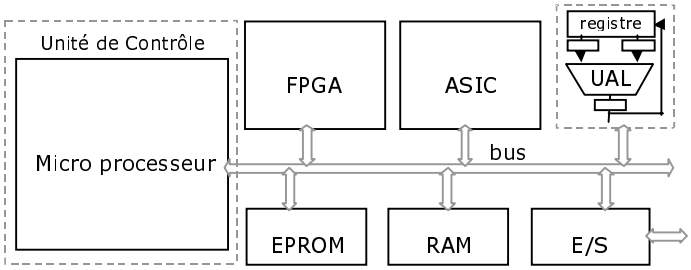


figure 2.5 : Exemple d'architecture

2.4.2 Composants de l'architecture

On appelle architecture mixte logicielle/matérielle une organisation qui combine à la fois des composants logiciels et des composants matériels. Les systèmes complexes existants comportent une multitude de composants. Les composants trouvés dans ce type de système sont les processeurs (composants logiciels), des ASICs et des FPGAs (composants matériels).

Processeurs

Nous pouvons classer les processeurs en processeurs d'usage universel, processeurs parallèles et processeurs spécifiques [139][140][141][142].

Les processeurs d'usage universel, tels que le RISC (Reduced-Instruction-Set Computer), sont optimisés pour réaliser des cycles d'horloge courts, un nombre restreint de cycles par instruction, et une parallélisation efficace du flot d'instruction (pipelining). Ils offrent un degré maximum de flexibilité, ainsi qu'un temps de développement très réduit puisqu'aucune conception matérielle n'est nécessaire et que les outils de développement logiciel sont disponibles. Par contre le bas coût en surface, la basse consommation et surtout les performances ne sont pas assurées. Comme le montre la figure 2.6, le chemin de données d'un processeur RISC se compose généralement d'un grand registre et d'une Unité Arithmétique et Logique (UAL). La simplicité du contrôle et du chemin de données dans le RISC a comme conséquence un cycle d'horloge court et un rendement plus élevé. Cependant, la simplicité plus grande des architectures RISC exige un compilateur plus sophistiqué. En outre, étant donné que le nombre d'instructions est réduit, le compilateur RISC devra employer un ordre dans la séquence des instructions RISC afin de mettre en application des exécutions complexes.

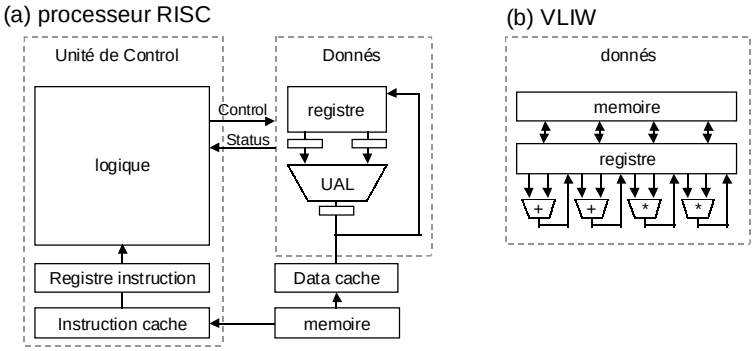


figure 2.6 : Exemples de microprocesseurs

L'architecture à processeurs parallèles telles que les machines VLIW, SIMD ou MIMD exploitent le parallélisme en utilisant de multiples unités fonctionnelles dans son chemin de données (figure 2.6) [152]. Une instruction VLIW (Very-Long-Instruction-Word) contient une zone pour chaque unité fonctionnelle. Chaque zone d'une instruction VLIW indique l'adresse de la source et les opérandes destination, ainsi que l'opération à exécuter par l'unité fonctionnelle. Comme résultat, une instruction VLIW est généralement très large, puisqu'elle doit contenir approximativement une instruction

standard pour chaque unité fonctionnelle.

Le processeur programmable spécifique, aussi appelé ASIP (Application-Specific Instruction-set Processor) est développé pour une application donnée tout en optimisant son architecture interne, son ensemble d'instructions et les programmes de son application. Les applications adaptées varient d'un simple algorithme à un vaste domaine comme celui du traitement du signal. L'approche consiste par la suite à réutiliser ce processeur avec son logiciel pour la conception de systèmes distribués complexes. La réutilisation des composants diminue le temps de conception et le coût des systèmes.

ASICs

Les ASICs (Application-Specific Integrated Circuit) offrent les garanties de bas coût en surface, basse consommation et de performances. Par contre la flexibilité, le temps de développement et les possibilités de ré-utilisation sont ses points faibles. Les concepteurs utilisent plus les ASICs pour les parties d'une application bien maîtrisées, laissant le reste pour une implantation en logiciel, et ceci dans un but de flexibilité et de coût. Les circuits ASICs ont fait l'objet de beaucoup d'intérêt pendant la dernière décennie à cause du développement d'outils de synthèse, souvent appelés outils de synthèse de haut niveau. Ces outils ont été développés pour augmenter la productivité de conception des circuits en réduisant le temps de conception. Pourtant, cette approche de conception n'est pas toujours adoptée.

Certaines applications doivent être programmables. C'est pourquoi les constructeurs des microprocesseurs mettent à disposition des concepteurs le cœur de leurs processeurs pour être utilisés comme une fonction a incorporer dans le circuit. L'utilisation d'un cœur de processeur programmable avec un chemin de données sur commande offre plusieurs avantages. Il permet l'amélioration des performances, car les composants critiques sont réalisés par un chemin de données sur mesure. Par conséquent, la communication logiciel/matériel peut être plus rapide. On obtient ainsi, une réduction de la surface et de la consommation par l'intégration du logiciel et du matériel sur un même support. Ce type de conception de circuit est attractif pour les applications embarquées, tels que les applications digitales pour le téléphone cellulaire. La conception de ce type de système nécessite le découpage de l'application en parties logicielles et matérielles, tout en explorant les différentes possibilités d'implantation.

FPGAs

Les FPGAs (Field Programmable Gate Arrays) sont souvent utilisés à côté des composants cités plus haut [152]. Ils fournissent des circuits logiques et numériques avec une complexité moyenne de l'ordre de quelques milliers de portes environ et peuvent être utilisés pour des applications spécifiques dans un seul circuit intégré. La popularité des FPGAs peut s'expliquer par leur facilité d'utilisation.

Les FPGAs peuvent servir pour le prototypage rapide, l'émulation matérielle et l'accélération de fonctions. Pour le prototypage rapide, un FPGA permet de transférer une nouvelle application sur silicium en quelques heures, alors que la réalisation des circuits intégrés (ASICs) nécessite des semaines voire des mois. Ceci est très utile pour l'évaluation de plusieurs options d'architecture pour une application donnée. Les FPGAs permettent l'émulation des applications pour lesquelles la simulation logique s'avère difficile à cause d'événements survenant en temps réel. Ainsi, un FPGA peut servir comme circuit de prototypage. Il peut aussi fournir une aide précieuse pour le développement d'algorithmes parallèles en supportant des configurations multiples.

2.4.3 Taxonomie d'architectures

Une architecture peut varier d'un simple contrôleur à une machine massivement parallèle. Une des caractéristiques envisageables dans un environnement de codesign est la flexibilité de

l'architecture. Une architecture flexible est une architecture facile à manipuler et à transformer. Généralement, le concepteur change fréquemment des détails du système à concevoir pour améliorer la performance ou réduire le coût. Dans un environnement de codesign on doit pouvoir traiter une grande classe d'applications. Pour cela, il est nécessaire d'avoir la possibilité de modéliser et de concevoir plusieurs variantes d'architectures.

En dépit des différentes variétés d'architectures, il est néanmoins possible de les répertorier dans l'une des deux classes suivantes : architectures monoprocesseur et architectures multiprocesseurs. Un aperçu sur ces deux classes d'architectures est présenté dans les sections suivantes.

Architectures monoprocesseur

Une architecture monoprocesseur est composée d'un processeur qui réalise le contrôle global et un chemin de données. Le chemin de données est composé d'unités fonctionnelles. Une unité fonctionnelle peut être un opérateur arithmétique, des mémoires, une unité d'entrée/sortie ou un copossesseur. Ces unités fonctionnelles sont synchronisées par le contrôleur maître qui séquence les événements. La figure 2.7 montre un exemple de ce type d'architecture. Cette architecture peut être réalisée en matériel, en logiciel ou par une combinaison des deux. Le rôle du contrôleur peut être accompli par une simple machine d'états finis, par un microprocesseur ou par un processeur spécifique. Le chemin de données peut être composé d'un ensemble d'ASICs ou de processeurs programmables agissant comme co-processseurs. Le contrôleur synchronise les événements dans chaque composant de l'architecture. Le bus, les registres et les multiplexeurs assurent le transfert de données entre les différents composants de l'architecture.

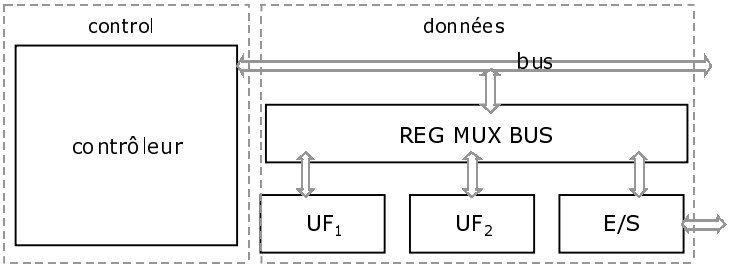


figure 2.7 : Modèle d'organisation d'une architecture monoprocesseur

Dans le domaine du codesign, plusieurs travaux ont été menés pour permettre la conception des architectures mixtes logiciel/matériel basées en ce modèle. Certains de ces travaux, LIRMM [19], Cosyma [24], Vulcan [97], utilisent un modèle d'architecture composé d'un processeur logiciel, d'une mémoire et d'un accélérateur matériel (ASICs). Le modèle utilisé est illustré par la figure 2.8. Quelques exemples de ce type d'architecture sont présentés à la suite.

L'environnement Vulcan [97], appliqué aux systèmes temps-réel, utilise un seul processeur, une mémoire et un bus de communication. Tout est intégré dans le même circuit. Le processeur utilise un seul niveau de mémoire et d'adressage pour les instructions et les données. Le transfert des données entre le processeur et les unités fonctionnelles est fait par des routines d'interruptions. Ces interruptions sont associées à un temporisateur. L'architecture cible de l'environnement SpecSyn [58] peut être utilisé pour une multitude d'applications. Elle est basée sur un microprocesseur standard complété par des ASICs. Un estimateur de logiciel fournit des métriques pour le logiciel généré en fonction du processeur cible. Ces métriques (temps d'exécution, taille de l'exécutable, etc.) permettent l'identification de goulots d'étranglement dans le système qui peut entraîner une migration du logiciel au matériel. L'environnement Codes [59], développé à Siemens, utilise une plate-forme formée d'un processeur, d'une mémoire, de composants standards et de circuits spécifiques issus de la synthèse. Cette architecture supporte un modèle de communication abstrait qui permet la représentation du

système à travers d'unités communicantes. Ce modèle, appelé PRAMs (Parallel Random Access Machines), supporte la communication point-à-point et la communication par diffusion.

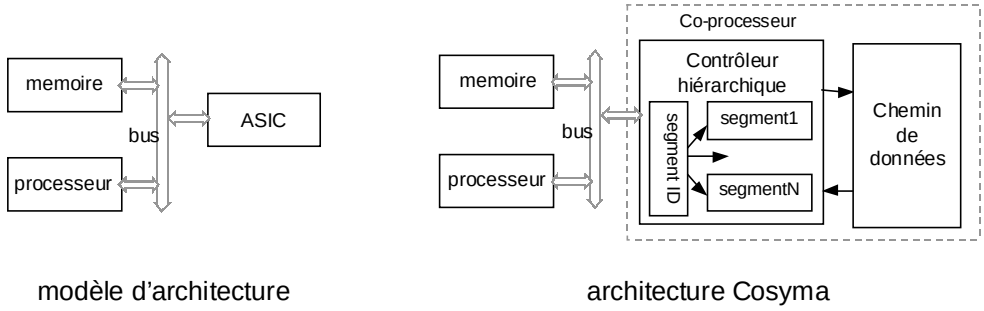


figure 2.8 : Architecture monoprocesseur pour le codesign

Architecture synchrone utilisant plusieurs processeurs esclaves

Ce type d'architecture synchrone met en œuvre plusieurs processeurs utilisés comme co-processeurs (accélérateurs ou processeurs esclaves). Dans ce modèle, l'architecture cible comprend un contrôleur principal et un ensemble de co-processeurs. Chaque co-processeur est un processeur qui exécute le code d'instructions spécifiques à la tache qui lui est allouée. La communication entre les modules est assurée par une unité centrale unique qui contrôle et synchronise la communication au travers d'un bus commun. Cette architecture est illustrée par la figure 2.9. Elle est utilisée par le projet Tosca [28] pour permettre l'accélération de l'exécution d'algorithmes de traitement du signal. La fonction globale est répartie entre sous-tâches, chacune étant réalisée par un co-processeur. Tous les processeurs exécutant ces tâches sont des modules logiciels identiques. Seule l'unité de contrôle et de synchronisation est réalisée en matériel. Même si cette architecture s'avère très performante pour des applications bien ciblées, l'inconvénient majeur réside dans le fait qu'elle ne peut pas supporter des modules différents des processeurs utilisés. D'autre part, cette architecture est difficile à mettre en œuvre, car si l'algorithme à intégrer n'est pas modulaire, on ne pourra pas le découper en sous tâches indépendantes qui seront exécutées sur les différents co-processeurs.

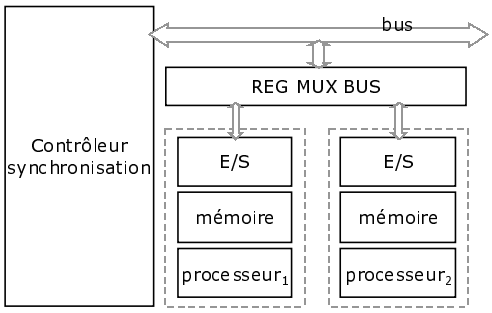


figure 2.9 : Exemple d'architecture synchrone (Tosca)

Le projet RASSP (Rapid Prototyping of Application Specific Signal Processors) [20], propose une méthodologie pour accélérer et réduire le coût de conception dans un environnement de codesign. Il vise à produire une architecture monoprocesseur qui est composée d'un processeur principal et des co-processeurs (accélérateurs matériels) pour exécuter les calculs complexes. Cette méthodologie propose la génération automatique de modèles de microprocesseurs et de DSPs existantes tels que RISC i860xp, ADSP 21060 ou PowerPC 601. Une architecture multiprocesseur peut être également construite à partir de plusieurs modules interconnectés par des unités de communication.

Architecture Spyder: système de développement à processeur reconfigurable

Le projet Spyder vise à développer un processeur reconfigurable, adaptable à chaque application,

ainsi que son environnement de programmation [152]. Cette approche superscalaire, avec plusieurs unités de traitement travaillant en parallèle, utilise une architecture de type VLIW et des unités de traitement reconfigurables, taillées sur mesure pour chaque application.

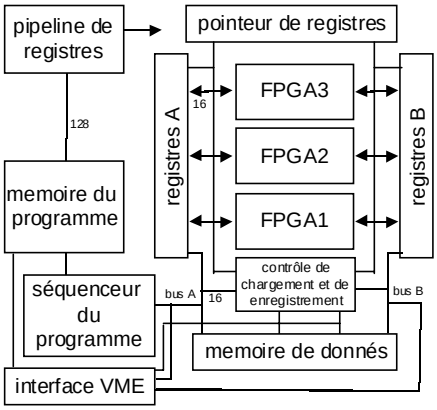


figure 2.10 : Architecture Spyder

Les différentes unités de traitement, des circuits FPGAs, sont commandées en parallèle, directement par les bits de l'instruction. Le processeur Spyder est réalisé sur une carte VME implantée dans une station de travail qui sert de machine hôte. Un compilateur spécialisé traduit un programme écrit dans un langage de haut niveau, C++, en schémas logiques directement utilisables par les outils de conception pour configurer les unités de traitement.

Architectures multiprocesseur

Dans un système multiprocesseur, on profite de l'avantage du parallélisme en utilisant des éléments fonctionnant d'une manière concurrente. Chaque élément de l'architecture doit comporter son propre contrôleur (ou processeur) et des ressources locales (figure 2.11). Chaque contrôleur communique avec les autres à travers sa propre interface de communication ou via un contrôleur de communication intermédiaire. Dans ce dernier cas, le contrôleur de communication joue le rôle d'arbitre. Ce genre d'architecture permet la distribution de tâches à l'ensemble des contrôleurs. Pour les systèmes complexes, cette architecture simplifie le processus de développement. Les choix importants sont le nombre de processeurs, le partage de ressources, les possibilités de communication directe, et la taille des mémoires locales.

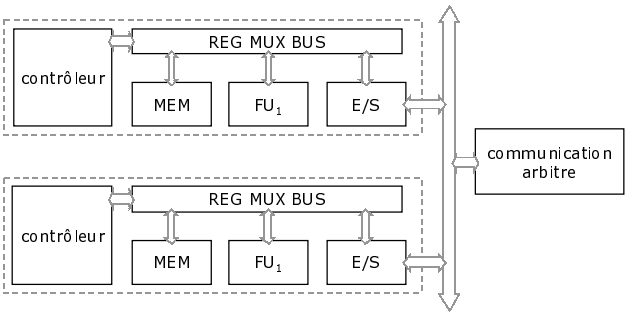


figure 2.11 : Modèle d'organisation d'une architecture multiprocesseur

Dans une architecture multiprocesseur, le système est organisé comme un ensemble de modules qui sont spécifiés séparément. Chaque module peut être un processeur (logiciel), des ASICs (matériel) ou mixte (logiciel/matériel), comme est montré par la figure 2.12.

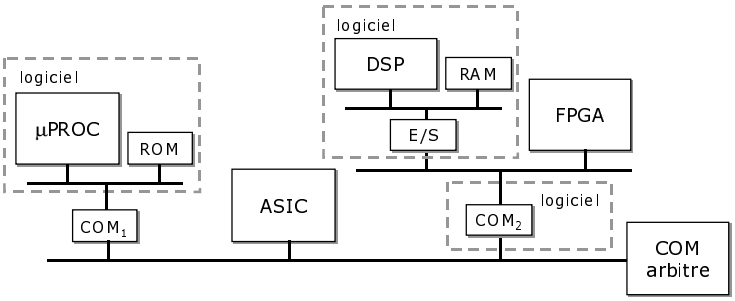


figure 2.12 : Exemple d'architecture multiprocesseur

Chaque module de communication assure le transfert des données et des commandes entre certaines modules de l'architecture. Suivant la technologie, il peut être un simple composant passif (par exemple une mémoire partagée) ou assez sophistiquée (comme un contrôleur d'entrée/sortie). Dans ce dernier cas, il peut correspondre à un circuit existant, comme un contrôleur d'interruption ou un DMA (Direct Memory Access). Il peut aussi être un module logiciel, par exemple un microprocesseur dédié à la communication.

Communication

Les architectures multiprocesseurs utilisent trois types de communication : point-à-point, par mémoire partagée (communication processeur/mémoire/processeur) et par passage de message dans un système à mémoire distribuée (communication processeur/processeur) [145]. Les trois types de communication sont représentés dans la figure 2.13.

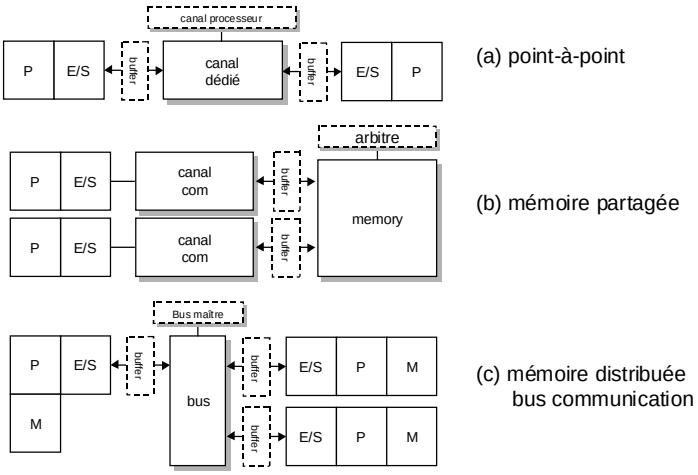


figure 2.13 : Communication dans les architectures multiprocesseurs

La communication point-à-point (figure 2.13(a)) entre deux processeurs est le type de communication le plus simple. Le canal physique est consacré à l'interconnexion des deux processeurs. S'il n'y a aucune mémoire tampon (buffer) la communication nécessite un mécanisme de rendez-vous où les deux processeurs sont actifs et prêts à participer de la communication en même temps. La communication par mémoire partagée utilise une mémoire globale comme moyen d'échange de donnés entre les processeurs. La figure 2.13(b) montre un ensemble de processeurs connectés par des bus indépendants à une mémoire commune. L'accès à la mémoire doit être protégé par des arbitres. Par exemple, on doit prévoir le cas où deux processeurs essayent d'écrire dans la même adresse de la mémoire en même temps. Pour éviter de telles situations de conflit on doit contrôler l'accès de la mémoire. Les systèmes à mémoire distribuée évitent les problèmes de chevauchement de données. Dans un système à mémoire distribuée, chaque processeur a accès seulement à sa propre mémoire

(figure 2.13(c)). D'autre part l'accès aux données dans une mémoire commune est possible par passage de messages entre les processeurs à travers le bus. Le bus relie plusieurs processeurs et il est employé pour partager le canal physique de communication.

Architecture à Mémoire Distribuée SynDex

L'architecture utilisée par l'environnement de conception de systèmes temps réel distribués SynDex [60] est présenté par la figure 2.14. Cet environnement permet également de concevoir des applications multiprocesseurs de traitement du signal. L'architecture générée est composée de modules logiciels. Chaque module comprend une unité de calcul (CPU), une unité d'entrée/sortie (E/S) et des mémoires locales ou partagées (RAM). La mémoire locale reçoit les instructions à exécuter. Les mémoires peuvent aussi échanger des données sans l'intervention du processeur. La communication entre deux processeurs est utilisée pour le transfert de données entre les processeurs connectés. Le module d'entrée/sortie permet au système de communiquer avec l'environnement.

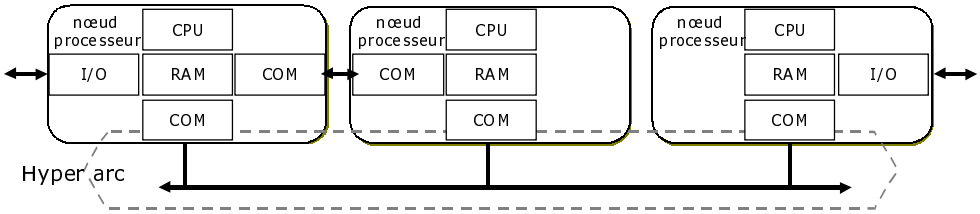


figure 2.14 : Architecture SynDex

Architecture distribuée de Ptolemy

L'environnement Ptolemy est utilisé pour des applications de traitement du signal et des systèmes communicants [62]. La spécification d'entrée est formée par plusieurs programmes. Cette spécification peut être synthétisée pour une variété d'architectures cible : système monoprocesseur, architecture parallèle à mémoire partagée, architecture parallèle à bus partagé, architecture parallèle avec passage de messages. Le découpage de la spécification est guidé par les contraintes de surface, de vitesse et de flexibilité. Lors de la génération de code pour les processeurs programmables, le découpage essaye d'optimiser les fonctions coûts telles que les coûts de communication, l'espace mémoire local et global, etc.

2.4.4 Résumé des architectures

Il y a une grande variété d'architectures cible pour la conception conjointe logicielle/matérielle. Cette variété est le résultat de l'optimisation manuelle d'architecture à une application simple ou à un domaine d'application sous de diverses contraintes et fonctions de coût. L'optimisation mène aux systèmes spécialisés. Plusieurs facteurs de coût et de performances dépendent directement du choix de l'architecture d'implémentation. L'utilisation d'une architecture spécifique, bien adaptée à une application déterminée, est plus efficace. Cependant, ce type d'architecture n'est pas assez flexible pour bien s'adapter dans un environnement de codesign à d'autres types d'application.

2.5 Bilan sur les systèmes de codesign

Cette section classe les outils les plus connus pour la conception conjointe logiciel/matériel. La liste des outils ne se veut pas exhaustive mais représentative du domaine. Ces outils sont développés soit dans des universités, soit dans des centres de recherche et développement appartenant à des industriels. La figure 2.3 recapitule les différents outils de conception conjointe logiciel/matériel étudiés.

Pour chaque outil, le tableau donne :

- le langage de spécification utilisé en entrée,
- le type d'applications visé,
- les étapes de conception suivies et
- l'architecture cible.

OUTILS	Spécification Système	Type d'application	Approche de conception	Architecture cible
SpecSyn (Irvine)	SpecCharts	Systèmes de contrôle et de communication	Découpage logiciel/matériel Estimation Implantation	Multi-processeur +ASICs
Ptolemy (Berkeley)	Blocs interconnectés multi-langage	Traitement du signal et systèmes communicants	Découpage Sythèse de matériel, logiciel, et d'interfaces Cosimulation	Paramétrable Parallel ou Mono-processeur
Vulcan (Standford)	HardwareC	Systèmes temps réel	Hw/Sw partitioning Estimation	CPU+ASIC +Bus +memory
Cosyma (Braunschweig)	C^x	Systèmes Complexes	Découpage logiciel/matériel Estimation Prototypage	CPU+ASIC +Bus +memory
Codes (Siemens)	SDL StateCharts	Systèmes de communication	Découpage matériel/logiciel Synthèse Prototypage	Multi-processeur +FPGA +ASIC
Tosca (Italtel)	Speed Chart	Systèmes de communication	Découpage matériel/logiciel Synthèse	Mono-processor +coprocessors
SynDex (INRIA)	SIGNAL	Traitement du signal	Découpage matériel/logiciel Ordonnancement Synthèse Compilation	Distribuée Processeurs communicants
SAW (Carnegie Mellon)	CSP	Accélération de fonctions logicielles par des accérérateurs matériels	Découpage matériel/logiciel Synthèse Prototypage	CPU+PCB (FPGA+ASIC)
MCSE IRESTE	Formalisme MCSE	Systèmes de contrôle et de communication	Parttionnement interactif évaluations de performances simulation au niveau modèle (non interprétée) cosimulations VHDL et C++	Multi-processeurs
CoWare (IMEC)	C + VHDL + CoWare	Traitement du signal et systèmes communicants	Synthèse d'interfaces Ordonnancement Cosimulation	Multi-processeurs +ASICs

figure 2.15 : Tableau recapitulatif des outils de conception logiciel/matériel

Les langages de spécifications peuvent être soit mono-flot (par exemple le langage C), soit multi-flot (les systèmes décrits en langage SDL). L'architecture cible peut être soit monoprocesseur, soit multiprocesseurs. Une architecture monoprocesseur se compose généralement d'une unité de contrôle centrale (CPU) avec un ou plusieurs ASICs (ou FPGAs). Dans ce cas, les composants matériels jouent le rôle d'accélérateurs pour la partie logicielle. Pour les architectures multiprocesseurs, le contrôle est distribué entre ces éléments. Pour ce qui concerne les étapes de synthèse, celles-ci dépendent du langage de spécification initial et de l'architecture cible. Généralement, les différentes approches réalisent un découpage logiciel/matériel basé sur des estimations, ensuite compilation des parties logicielles et synthèse des parties matérielles. La phase finale est l'intégration des différentes parties sur une architecture prototype. Le résultat peut servir à l'émulation ou bien au prototypage.

2.6 Le projet Cosmos et le prototypage virtuel

Cosmos est une méthodologie et un outil qui comble la distance existant entre les outils du niveau système et les outils de synthèse. A partir d'un modèle du système décrit en SDL, Cosmos réalise le découpage et la synthèse de la communication pour ensuite générer une architecture distribuée composée des modules logiciels et de modules matériels. Toutes ces étapes de conception sont basées sur une forme intermédiaire appelée Solar [83]. Le prototypage virtuel [9] produit à partir de Solar du code exécutable pour chacun des modules en fonction de leur utilisation (figure 2.16). Le prototype virtuel est une architecture hétérogène représentée par du code C (processeurs virtuels logiciels) et du code VHDL (processeurs virtuels matériels).

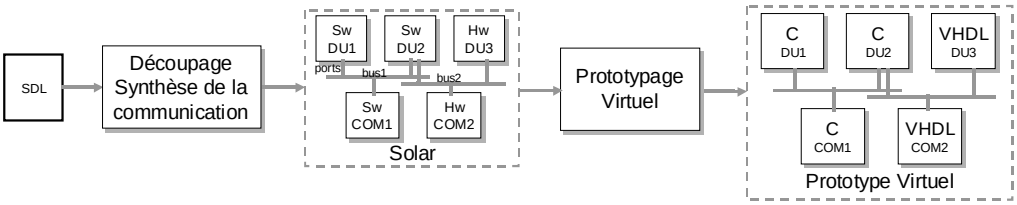


figure 2.16 : Etape de génération du modéle C/VHDL

Le prototype virtuel peut être exécuté à partir des mêmes stimuli utilisés pour la vérification de la spécification initiale décrite en SDL. C'est un moyen de vérifier, par simulation, que les étapes de synthèse de niveau système n'ont pas provoqué de dérives du comportement initial. Par ailleurs, ces descriptions exécutables (en C ou en VHDL) permettent d'estimer avec plus de précisions certaines caractéristiques de performance d'un système.

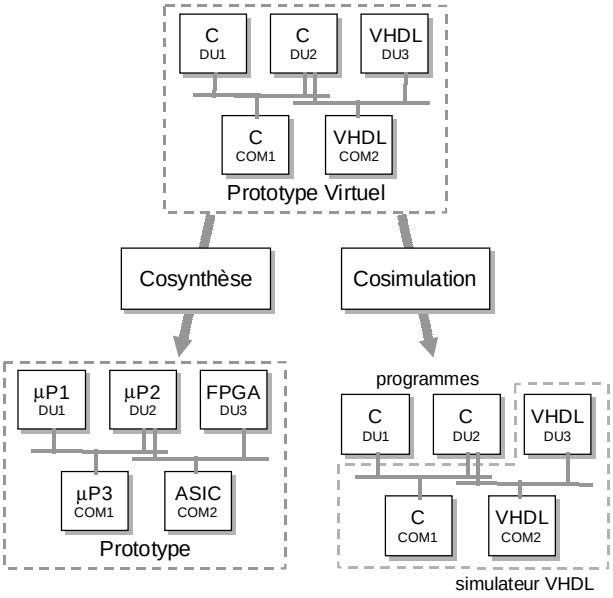


figure 2.17 : Le prototypage virtuel

La méthodologie proposée considère le problème du développement d'un modèle unifié pour la cosimulation et la cosynthèse (figure 2.17). Les descriptions produites sont à la fois simulables et synthétisables. La génération de modèles matériels pour la synthèse consiste à effectuer la transposition des modules matériels pour la synthèse d'architecture (on considère le style de modélisation lié à la plate-forme architecturale choisie, l'outil de synthèse et aussi la cosimulation). La génération de modèles logiciels pour la compilation requière l'ordonnancement des modèles multiprocesseurs sur une architecture monoprocesseur. Contrairement aux modèles matériels, la synthèse de logiciel implique l'exécution séquentielle de processus auparavant concurrents.

La cosimulation des modules distribués logiciels et matériels nécessite l'utilisation de différents outils de simulation et de langages de description (C pour les éléments logiciels et VHDL pour les éléments matériels), exécutions concurrentes d'outils de mise au point des parties matérielles et logicielles, et abstraction de la communication matériel-logiciel.

2.7 Conclusion

Les méthodologies proposées pour le codesign diffèrent selon les quatre choix stratégiques suivants : le style de la spécification en entrée, le domaine d'application, le modèle de l'architecture cible, et les étapes de synthèse. C'est pour cette raison que dans ce chapitre nous avons présenté les principales approches de description de systèmes, les environnements de conception et les différentes architectures pour la conception de systèmes mixtes logiciel/matériel.

L'étude des architectures a permis de relever les principales caractéristiques de chacune. D'entre elles, l'architecture mono-processeur constitue l'élément de base d'un système distribué. On peut considérer cette architecture comme un module parmi d'autres qui participent à la composition d'un système plus complexe. Une architecture mixte doit être flexible et modulaire pour pouvoir s'adapter à diverses applications.

Les environnements de codesign suivent une des deux approches de spécification de systèmes : la spécification homogène ou la spécification hétérogène. La combinaison dans une même représentation des ensembles matériels et logiciels nécessite soit 'un modèle ou langage unifiant la sémantique associée à ces ensembles (spécification homogène), soit des langages spécifiques pour chaque ensemble essayant de les intégrer dans un même environnement (spécification hétérogène). Par rapport à la première approche, nous devons obtenir, à partir d'une représentation unique, un prototype virtuel où chaque ensemble (logiciel/matériel) est décrit par un langage approprié. Ensuite, une représentation hétérogène doit faire face au problème de modélisation de l'interaction entre les ensembles matériels et logiciels. D'ailleurs, fréquemment la validation de tels systèmes n'est ainsi possible qu'une fois que matériel et logiciel ont été réalisés.

Chapitre 3

Génération automatique des interfaces VHDL-C pour la cosimulation distribuée

---

La simulation d'un système mixte matériel/logiciel représente un élément important dans les méthodologies de codesign. La cosimulation permet la vérification fonctionnelle du système complet. Cette étape peut avoir lieu à différentes étapes du processus de codesign. Ceci permet la vérification des sous-ensembles matériels et logiciels dans l'ensemble du processus de conception. Ce chapitre décrit la spécification, la conception et la réalisation de l'outil de cosimulation VCI, développée pour permettre la vérification des systèmes mixtes matériel/logiciel à chaque étape de la méthodologie de codesign Cosmos, ainsi que les résultats obtenus lors des expérimentations de l'outil.

3.1 Introduction

Les méthodologies modernes de conception de systèmes peuvent amener à l'utilisation de plusieurs langages pour la spécification des différentes parties du système. Au cours du processus de conception, chaque module va être raffiné grâce à des outils spécifiques, et validé par la cosimulation avec le reste du système. Il est donc nécessaire de valider ces parties à chaque niveau d'abstraction [44][55][12].

Ce chapitre décrit la spécification, la conception et la réalisation de l'outil de cosimulation C-VHDL, VCI, ainsi que les résultats obtenus lors de son expérimentation. Le rôle principal de la cosimulation est de permettre la validation conjointe des modules matériels et logiciels décrits dans des langages spécifiques. Ceci implique l'utilisation d'un environnement de cosimulation composé d'outils de mise au point matériels/logiciels qui seront exécutés en parallèle. La réalisation d'un environnement de cosimulation couvre plusieurs aspects : l'établissement de liens entre les simulateurs pour permettre l'échange de données, l'intégration de ces liens dans les simulateurs, la synchronisation de l'exécution des simulateurs, et l'échange de données de types différents. Pour faciliter son fonctionnement, le lien entre les simulateurs doit rester transparent en ce qui concerne son intégration, tout en tenant compte des diverses couches de communication et des mécanismes de synchronisation existant entre les simulateurs.

Ce chapitre expose dans sa première section les motivations de ce travail. La section 3.3 présente les techniques de cosimulation. La section 3.4 retrace l'état de l'art en matière d'outils de cosimulation. La section 3.5 décrit la cosimulation distribuée C/VHDL. Dans la section 3.6 est décrit l'outil de cosimulation VCI, développé pour la génération automatique des interfaces de cosimulation C/VHDL. Dans la section 3.7, les concepts ci-dessus sont appliqués à des exemples expérimentaux. La section 3.8 fait une présentation du prototype virtuel utilisé par le projet de codesign Cosmos. Enfin, nous concluons ce chapitre par des résultats, des perspectives et des directions à suivre dans le cadre de travaux futurs.

3.2 Motivations

L'un des problèmes du codesign est la validation mixte matérielle/logicielle. Dans le flot de conception classique, cette validation est réalisée relativement tard. Elle est traditionnellement accomplie une fois que le prototype est disponible, en utilisant les émulateurs en circuit (in-circuit emulators) ou d'autres techniques [57][88]. Il est observé que plus la validation intervient tôt dans la conception d'un système, plus une erreur est corrigée rapidement [147]. Afin de réduire les coûts et d'économiser les efforts inutiles, il est important de s'assurer que les fonctionnalités modélisées sont conformes aux attentes pendant tout le processus de développement. Cette vérification peut être réalisée par la cosimulation.

Un environnement de cosimulation nécessite l'intégration d'outils et de modèles différents pour qu'ils puissent coopérer et échanger des données. On considère une architecture composée de modules logiciels décrits en langage C et de modules matériels décrits en langage VHDL. La partie matérielle est typiquement simulée à l'aide de simulateurs d'événements discrets ou de simulateurs cycle-driven. Le processeur qui exécute le logiciel peut être modélisé à divers niveaux de détails. L'intégration et l'exécution simultanée de ces outils posent un problème d'ingénierie important en raison des limitations des outils existants à offrir des interfaces d'échange avec des outils externes [146].

Dans certains cas [6], il est davantage nécessaire de valider la fonctionnalité du logiciel avec l'environnement sans l'existence d'un modèle du processeur. La conception d'un système intégré

contenant un processeur spécifique a la particularité que le logiciel embarqué doit être développe et validé avant que le matériel ne soit réalisé. La validation du logiciel embarqué intervient donc avant ou pendant le développement du processeur sur lequel il devra s'exécuter [15][6]. Néanmoins, ce problème peut être résolu au moyen des techniques de cosimulation C/VHDL. En effet, cette validation peut être effectuée en faisant abstraction de l'interface qui existe entre le processeur et son environnement. Cette abstraction peut avoir plusieurs couches de détail qui vont d'une simple fonction d'E/S jusqu'aux fonctions complexes de communication.

Les besoins d'un outil de cosimulation ont motivé le développement d'un outil de génération automatique des interfaces de cosimulation C-VHDL, nommé VCI [107][7]. Cet outil, à partir d'une description abstraite de l'interface entre le matériel et le logiciel, produit un environnement de cosimulation distribuée, fiable et personnalisé pour la vérification des systèmes mixtes matériels/logiciels à chacune des étapes de la méthodologie de codesign.

3.3 Les techniques de cosimulation

La cosimulation logiciel/matériel consiste à vérifier que les parties matérielles et logicielles d'un système fonctionnent correctement ensemble. Ceci inclut la simulation des modules matériels, des processeurs et des logiciels que les processeurs exécutent. En général, dans le cadre d'une approche de codesign, la conception débute par des modules logiciels décrits en langage C et des modules matériels décrits en langage VHDL [6]. Les méthodes de validation dit traditionnelles utilisent aussi un modèle du processeur pour la partie logicielle. Ce modèle peut exister sous plusieurs formes : physique, matériel ou logiciel [105][15].

Le modèle physique du processeur est le plus précis (normalement de l'ordre de nano-secondes pour tous les pins), mais dont l'exécution est la plus lente (de l'ordre de 1 à 100 instructions par seconde). Le modèle matériel, souvent en langage VHDL, comportemental ou RTL, est aussi accompagné du compilateur C. Les modèles entièrement fonctionnels peuvent être précis au niveau du cycle (*cycle-accurate*) ou au niveau de l'instruction (*instruction-level*) avec une performance de l'ordre de 50 à 1000 instructions par seconde. Dans ce cas le logiciel écrit en C est compilé en code assembleur. Ce code assembleur est ensuite chargé dans la mémoire de programme du modèle VHDL du processeur et simulé conjointement avec le reste du système par un simulateur VHDL unique. Le modèle logiciel du processeur est un programme appelé simulateur du jeu d'instructions (*Instruction-Set Simulator*). Ce type de modèle peut être beaucoup plus efficace que le modèle matériel avec une performance de l'ordre de 10 000 à un million d'instructions par seconde. Ce modèle est fréquemment intégré à l'environnement du simulateur matériel (C'est le cas des outils commerciaux Cossap de Synopsys [148] et SPW de Cadence [149]).

La cosimulation fonctionnelle C-VHDL [107] fait intervenir le code C, à la place du code assembleur. Le code C est exécuté en parallèle avec le simulateur VHDL du reste du système. Cette cosimulation ne requiert aucune description du processeur, mise-à-part la définition des signaux d'interface du processeur pour sa connexion avec le reste du système. Le logiciel (pour simplifier nous ne considérons qu'un seul) est compilé sur la machine hôte de la simulation et mise en communication avec le simulateur VHDL. Si cette communication est assurée par des méthodes de liaison asynchrones (tel que le temps de communication n'a aucun effet sur la fonctionnalité), il est possible d'avoir une méthode de simulation encore plus rapide au prix d'une perte de précision [105][15]. Cette perte de précision est du au manque d'information associée au nombre de cycles d'horloge par instruction correspondants au processeur cible. Les avantages d'une telle approche sont : une validation fonctionnelle complète du système indépendamment du processeur cible et a plusieurs niveaux d'abstraction, l'utilisation d'outils de développement logiciel standards et la possibilité

d'intégration de plusieurs outils de simulation.

3.4 Outils de cosimulation

Un bon nombre de travaux sont actuellement effectués pour couvrir plusieurs aspects de la cosimulation.

Quelques approches se servent d'un mécanisme de communication fixé selon l'architecture cible. En utilisant le modèle de processeur et les modèles de communication de bas niveaux. Les environnements fournissent la simulation à grain fin aux dépens du temps de simulation et de la flexibilité. Le projet Cobra utilise un environnement de prototypage fondé sur FPGAs, appelé Sparrow, supportant l'intégration d'un processeur standard dans des conditions de temps-réel, ainsi que l'émulation du processeur [57]. La plupart de ces approches fournissent un environnement de cosimulation et de mise au point efficace. Cependant, leur inconvénient principal est d'être restrictifs à un modèle de communication lié à l'architecture cible.

D'autres environnements permettent une cosimulation distribuée logiciel/matériel à l'aide d'outils de mise au point spécifiques. Cependant, ces environnements utilisent des modèles de communication spécifiques à l'application. Donc, ils limitent la possibilité de migration vers d'autres solutions d'implantation. Dans [101], des composants matériels et logiciels (décrits dans Verilog et C++, respectivement) sont traités (dans l'environnement Unix) en tant que processus séparés qui communiquent par des sockets BSD (*Berkeley Software Distribution*). Parce qu'ils visent une application spécifique, le logiciel utilise un modèle de communication fixe, ce qui permet plus tard l'utilisation du modèle de processeur associé. Une approche semblable est décrite dans [95] ; l'aide de l'outil consiste à tracer la spécification sur un processeur simple communiquant avec de multiples ASICs. Le logiciel exécute sur le processeur de développement et communique avec un simulateur matériel par le mécanisme de communication d'interprocessus IPC (*Inter Process Communication*), en utilisant le passage de messages.

Une approche plus générique de la cosimulation, décrite dans [106], exécute la cosimulation du système mixte logiciel/matériel à diverses étapes du processus de codesign. La méthodologie met à jour deux descriptions séparées pour le matériel et le logiciel. Des primitives de communication sont insérées dans les descriptions matérielles/logicielles pour modéliser la communication dans le système. Ce modèle permet à la communication entre le matériel et le processus logiciel d'avoir lieu sans connaissance des détails explicites de l'interface intrinsèque du bus. Les primitives requises pour manipuler la communication sont classés comme suit : le transfert de données synchronisé (*synchronized data transfer*), le transfert de données non synchronisé (ou non tamponné), et la synchronisation sans transfert de données. Bien que cette approche étende l'espace des architectures supportées (en travaillant à un niveau plus élevé d'abstraction de la communication), l'interface de cosimulation matériel/logiciel est limitée parce qu'elle devra tenir compte que de ces modes de communication.

3.4.1 Cosyma

Cosyma se sert d'un arrangement fixe de communication. Il fournit un simulateur du processeur pour l'analyse de l'exécution et la vérification cible [24]. La description de l'entrée de Cosyma est un modèle textuel en langage C^X, qui est une extension du langage C. Il permet d'inclure des contraintes de temps et le parallélisme entre les processus. La stratégie, adaptée au partitionnement matériel/logiciel, commence, à partir d'une solution purement logicielle, par extraire les pièces critiques pour les porter à l'extérieur, dans le matériel. Les parties à réaliser en logiciel sont traduites en langage C et les parties à réaliser en matériel sont traduites en langage HardwareC. L'architecture

cible utilise seulement un processeur avec un circuit intégré (ASIC), une mémoire et un bus. En utilisant le processeur et les modèles inférieurs de communication, les environnements fournissent une simulation de grain fin, aux dépens du temps et de la flexibilité de la simulation.

3.4.2 Ptolemy

L'environnement de codesign Ptolemy [62][94] utilise un modèle de système orienté objet. L'environnement supporte différents modèles de conception encapsulés dans des objets appelés "domaines" (domains). Un domaine réalise un modèle de calcul d'un sous-ensemble. Pour la cosimulation, l'environnement exige une description complète du système. Le travail décrit dans [23] 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 du flux de contrôle/données du système.

3.4.3 CoWare

Parmi tous les environnements de cosimulation existants, CoWare [27] est celui qui se rapproche le plus des travaux présentés dans cette thèse. Il est orienté vers les systèmes distribués et couvre plusieurs domaines d'application. CoWare offre une grande flexibilité pour la synthèse des interfaces et la spécification C-VHDL. Il est fondé sur un modèle d'abstraction puissant, capable de manipuler une grande classe de modèles de communication. Les interfaces sont décrites au niveau de l'application, permettant de ce fait une grande modularité. Cependant, l'utilisateur de CoWare est prié de fournir des bibliothèques de communication grandes et sophistiquées, ce qui est tout à fait difficile à concevoir et à mettre au point.

3.4.4 Notre approche

Le modèle de cosimulation décrit ici est plus facile à utiliser que [27] et utilise un modèle plus flexible que [106] et [23]. Le lien de cosimulation créé entre les outils de mise au point logiciels/matériels repose sur l'interface d'E/S associée au module logiciel. Ainsi, l'outil n'a pas besoin d'une description complète du système pour produire les liens de cosimulation. En conséquence, le module logiciel peut être utilisé comme tout autre composant matériel. Ceci permet de manipuler toutes sortes d'architectures distribuées. Du coté du logiciel, le lien est réalisé par des primitives d'E/S simples. La communication peut donc être abrégée dans plusieurs couches, ce qui permet la cosimulation à différents niveaux d'exactitude et une transition douce vers le processus de cosynthèse. Conformément à la configuration du lien entre les simulateurs, l'exécution des modules logiciels peut être faite dans un ou plusieurs CPUs. Une autre flexibilité de notre 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. Le style de modélisation et le fondement de cette méthodologie sont décrits dans le chapitre 6 et publiés dans [9].

3.5 Cosimulation distribuée logiciel/matériel

Cette section traite de l'environnement de cosimulation distribuée développé dans ce travail pour le prototypage virtuel de systèmes hétérogènes. Le but d'un environnement de cosimulation distribuée est de fournir une méthode pour la simulation des systèmes logiciel/matériel. Dans l'environnement, on visualise les modules matériels et logiciels en tant que composants séparés en communication par l'intermédiaire d'une interface.

Une partie importante de la simulation porte sur le mode d'interaction entre le matériel et le logiciel. Cette interaction peut être validée dans un environnement de cosimulation distribuée.

L'environnement, composé de simulateurs du matériel et du logiciel, permet d'exécuter les modules matériels (décrits en VHDL) et logiciels (écrits en langage C) en tant que processus parallèles communiquants. Cette cosimulation peut avoir lieu à différentes étapes du processus de codesign. Ainsi, des modules spécifiés à plusieurs niveaux d'abstraction peuvent être validés. Les niveaux d'abstraction du matériel incluent le niveau comportemental et le niveau transfert des registres (*RTL*). Les niveaux d'abstraction du logiciel sont le C de haut niveau et le niveau du cycle (*cycle-true*). En ce qui concerne la cosimulation, Il est nécessaire de spécifier l'interface correspondant au module logiciel et l'établissement de liens entre les simulateurs pour permettre l'échange de données. Ces liens permettent l'intégration des simulateurs, la synchronisation de la cosimulation et l'homogénéisation de l'échange de données de type différent.

3.5.1 Modèles de cosimulation

La question clé qui se pose pour la définition d'un environnement de cosimulation logiciel/matériel est le lien entre les différents simulateurs. En se fondant sur le mode d'exécution des simulateurs, il existe principalement deux modes de simulation : le mode maître-esclave et le mode distribué.

Le mode maître-esclave comprend un seul simulateur maître et un ou plusieurs simulateurs esclaves. Dans ce cas, les simulateurs esclaves sont appelés en utilisant des techniques d'appels de procédure (*procedure call*) qui peuvent être mises en application selon l'une ou l'autre des deux voies présentées ci-dessous :

- En appelant des procédures étrangères (par exemple des procédures écrites en langage C) par le simulateur principal ou
- en encapsulant le simulateur esclave dans un appel de procédure.
- La figure 3.1(a) représente un modèle de cosimulation typique maître-esclave. Celui-ci est détaillé en utilisant la cosimulation des modèles mixtes C-VHDL.

La plupart des simulateurs commerciaux de VHDL (par exemple VSS de Synopsys [85] et LeapFrog de Cadence [150]) fournissent des moyens de base permettant d'appeler des fonctions C pendant la simulation VHDL d'après le modèle maître-esclave de la figure 3.1(b). L'accès aux fonctions C est possible en utilisant l'attribut VHDL "foreign" dans la spécification de l'architecture VHDL. Cet attribut permet la déclaration des fonctions écrites dans des langages autres que VHDL qui seront appelés pendant la simulation. Ces fonctions peuvent être employées pour indiquer le comportement d'une entité VHDL, comme cela est représenté par la figure 3.1(b). L'exécution des fonctions, que nous désignons comme des procédures C d'émulation VHDL (VEC), suit le modèle des systèmes réactifs (a chaque événement d'entrée est produit un événement de sortie).

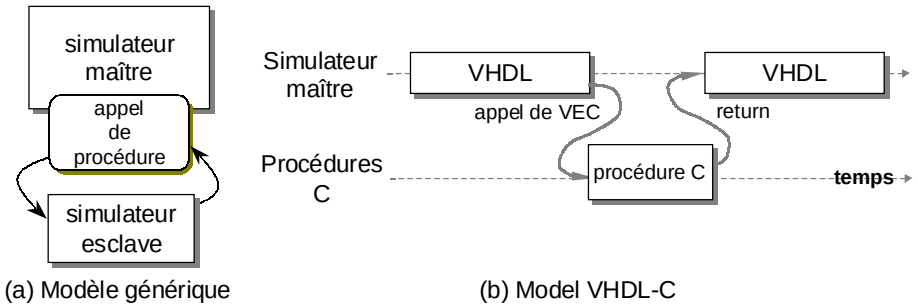


figure 3.1 : Cosimulation maître-esclave

Bien qu'utile, le mode maître-esclave présente une contrainte fondamentale : le besoin de découper le programme C en un jeu d'opérations exécutées une à une par l'appel de procédure. Ceci

exige un changement crucial du modèle d'exécution du programme C, en particulier pour des applications orientées vers le contrôle. Ces applications ont besoin de multiples points d'interaction avec le reste du matériel [84]. Pour des applications orientées vers le contrôle, ce modèle exige que le logiciel soit découpé en tranches et un arrangement sophistiqué où la procédure sauvegarde le point de sortie pour les futures invocations de la procédure. Par ailleurs, le modèle ne permet pas une véritable simultanéité : quand la procédure C s'exécute, le simulateur est en veille.

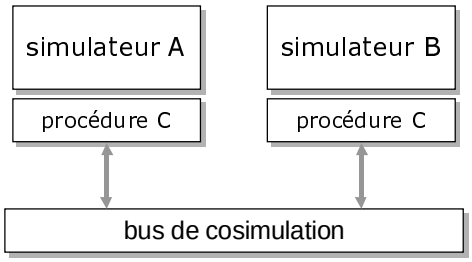


figure 3.2 : Modèle de cosimulation distribuée

Le modèle de cosimulation distribuée surmonte ces restrictions. La figure 3.2 représente un modèle générique de cosimulation distribuée. Cette approche repose sur un protocole du réseau de communication, qui est utilisé comme bus de cosimulation. Chaque simulateur communique par ce bus. Le bus, ainsi que ses mécanismes de communication, sont uniquement utilisés pour connecter les simulateurs. Il est responsable des transferts de données entre les différents simulateurs et aussi chargé de synchroniser l'exécution des simulateurs. En conséquence, il n'existe aucune relation entre ce bus de cosimulation et le modèle de communication employé par les modules C/VHDL. La mise en place de ce bus peut être fondée sur les mécanismes standard du système, tels que les IPC (communication entre processus) ou les sockets dans le cas du système Unix [101]. Il peut également être mis en application comme plate-forme spécifique de simulateurs (*ad-hoc simulator backplane*) [98].

Ce modèle comporte plusieurs avantages par rapport aux modèles de cosimulation précédents :

- **modularité** : les différents modules peuvent être conçus concurremment en utilisant différents outils et méthodes de conception ;
- **flexibilité** : les différents modules peuvent être simulés à plusieurs niveaux d'abstraction au cours du processus de conception en utilisant le même banc d'essai.

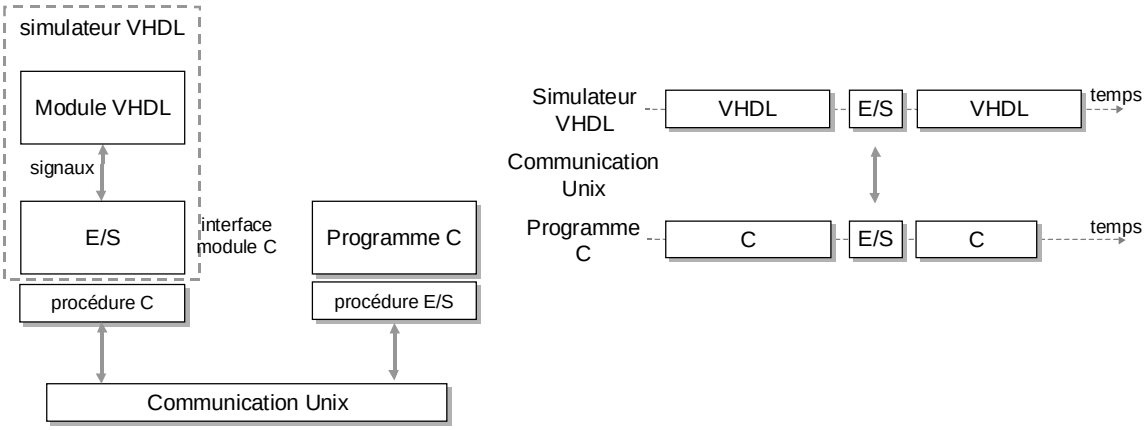


figure 3.3 : cosimulation distribuée C-VHDL

La figure 3.3 représente un modèle de cosimulation distribuée C-VHDL en utilisant les moyens de communication du système Unix. Par conséquent, les mécanismes de communication supportés sont les IPC (communication entre processus) et les sockets. La capacité d'appeler une procédure C dans la simulation VHDL est toujours utilisée, mais la procédure contient seulement le code pour la synchronisation des simulateurs et l'échange de données avec le bus de communication. Ainsi, le

concepteur peut maintenir le code d'application C dans sa forme initiale. En outre, le simulateur de VHDL et le programme C peuvent fonctionner simultanément. Le point essentiel de cette méthode est l'encapsulation de la voie de communication Unix et des couches de procédures VEC. Cette implémentation peut être tout à fait encombrante, en particulier lorsqu'elle est répétée pour plusieurs applications. Cela a motivé le développement d'un outil, qui produit automatiquement ces couches.

3.5.2 Bus de cosimulation

Le bus de cosimulation sert à l'échange de données entre le simulateur VHDL et l'application C. Le modèle de cosimulation distribuée C-VHDL utilise les moyens de communication du système Unix. Les mécanismes de communication plus courants sont les IPCs, les sockets et les RPC [90].

Les IPCs (Inter-Process Communication) offrent un moyen de communication, restreint à une seule machine, qui repose sur trois modes : la mémoire partagée, le sémaphore et le passage de messages. Le mode de communication par mémoire partagée utilise un segment de mémoire qui est accessible en lecture ou en écriture par les processus connectés. Les sémaphores utilisent les mécanismes d'exclusion mutuelle (droit d'accès) pour éviter les conflits d'utilisation du service. Le passage de messages utilise une file pour stocker les messages. Ces messages ont une identification du destinataire. Cette identification permet d'établir une liaison bidirectionnelle entre deux processus. Les primitives de lecture et d'écriture peuvent être bloquantes ou non. Les primitives bloquantes attendent indéfiniment la lecture (file vide) ou l'écriture (file pleine). Les primitives non bloquantes retournent un message d'erreur.

Les sockets, basés sur le modèle client-serveur, permettent la communication entre machines connectées à un réseau. Le serveur doit initialiser le socket et (à la demande de connexion du client) ouvrir un tube (*pipe*) entre lui et le client. Les données sont transférées par le tube dans un sens bidirectionnel.

Les RPC (*Remote Procedure Call*) sont basés sur l'appel de procédures à distance. Le serveur (ou remote) exécute une routine de service en réponse d'une requête du serveur (*call*).

Nous cherchons un modèle de communication pouvant fournir le plus large éventail de mécanismes, suivant les besoins. C'est pourquoi nous avons choisi le mécanisme IPC pour réaliser la communication dans une seule machine. Pour certains modes de synchronisation, garantir la séquentialité des événements est nécessaire. Dans ces cas, la file de messages sera utilisée comme moyen de communication. On peut utiliser la mémoire partagée pour les cas où les données sont consommées en même temps qu'ils sont produits. Pour la communication en réseaux on utilisera les sockets.

3.5.3 Abstraction de la communication VHDL-C

L'abstraction de l'interaction logiciel/matériel 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 de l'application, la communication est effectuée par des primitives de haut niveau, indépendamment de la réalisation. La figure 3.4 représente trois niveaux de communication entre les modules [95].

La complexité de l'interface matérielle/logicielle 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 niveau, 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. La génération automatique

de l'interface entre les simulateurs pour ce niveau d'abstraction exige un processus de sélection du protocole de communication [10][63][17].

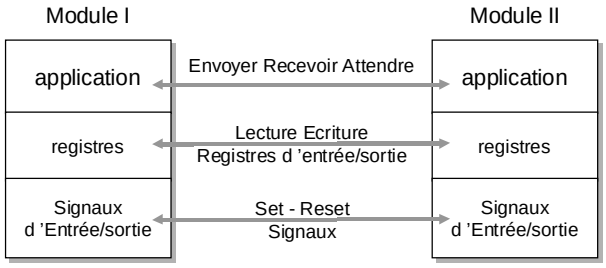


figure 3.4 : niveaux d'abstraction de la communication entre les modules

Le niveau intermédiaire utilise des registres. La communication est exécutée par la lecture/écriture sur des registres d'E/S. Ces exécutions peuvent cacher le décodage d'adresses physiques et la gestion des interruptions. La génération automatique des interfaces entre les simulateurs choisit une mise en place pour les exécutions de E/S. Par exemple, CoWare est un environnement typique de cosimulation agissant à ce niveau [5].

Au niveau le plus bas, tous les détails de la mise en place du protocole sont connus et la communication est exécutée par des liens directs (par exemple, des signaux de VHDL).

3.5.4 Les niveaux d'abstraction C-VHDL

Les environnements de cosimulation C-VHDL diffèrent par le niveau d'abstraction et le modèle de communication utilisés. La conception du système intégré sur puce combine généralement les processeurs programmables, exécutant le logiciel, et les composants matériels d'application spécifique (ASICs) [82][107]. L'utilisation des techniques de codesign fondées sur des modèles C-VHDL permet la cospécification, la cosimulation et la cosynthèse de l'ensemble du système. Les expériences [15] prouvent que ces nouvelles approches de codesign sont beaucoup plus souples et efficaces que la méthode traditionnelle, où la conception du logiciel et celle du matériel ont été complètement séparées.

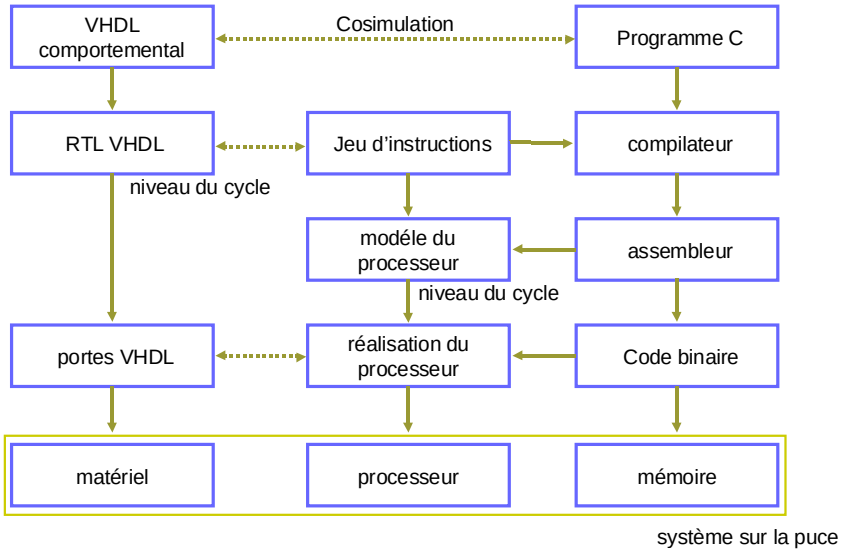


figure 3.5 : codesign C-VHDL pour les processeurs embarqués (embedded processors)

La figure 3.5 représente une méthode typique de codesign fondée sur les langages C-VHDL [107]. La conception commence par une spécification mixte C-VHDL et manipule trois niveaux d'abstraction : le niveau fonctionnel, le niveau du cycle (ou cycle accurate) et le niveau des portes (ou

gate-level).

Au niveau fonctionnel, le logiciel est décrit comme un programme C et le matériel par un modèle comportemental VHDL. A ce niveau, la communication logiciel/matériel peut être décrite au stade de l'application. La cosimulation C-VHDL peut être exécutée pour la vérification de la spécification initiale. Le code C est exécuté sur le poste de travail (workstation) et le VHDL est simulé au niveau comportemental. A ce niveau, seules des vérifications fonctionnelles peuvent être effectuées. Des vérifications précises du temps d'exécution peuvent être effectuées aux deux niveaux suivants.

Au niveau du cycle (cycle-accurate level), le matériel est décrit au stade RTL et le logiciel au stade de l'assembleur (assembly). A ce niveau, la communication C-VHDL peut être contrôlée au niveau du cycle. On peut aussi exécuter le code C sur un modèle du processeur. Ceci peut se produire en utilisant un modèle matériel (en VHDL) du processeur au niveau du cycle ou un modèle logiciel (un programme) au niveau d'instructions. Le modèle précis au niveau du cycle peut être obtenu automatiquement ou manuellement à partir du niveau fonctionnel. Le modèle VHDL au niveau RTL peut être produit en utilisant la synthèse comportementale. L'assembleur est produit par la compilation [83][85] ; la difficulté principale concerne l'amélioration de l'interface logiciel/matériel pour le passage d'un niveau à l'autre [108].

Le niveau des portes décrit la réalisation. Le modèle VHDL est raffiné par la synthèse du niveau RTL au niveau des portes. La mise en place du logiciel dans le processeur est fixe. Le processeur peut être abstrait au niveau de macro-instructions ou conçu au niveau des portes. Toute la communication entre le matériel et le logiciel est détaillée au niveau des signaux. Dans ce contexte, la cosimulation peut être exécutée pour vérifier et corriger le temps d'exécution de tout le système.

3.6 Génération d'interfaces de cosimulation VHDL-C

Dans l'approche de la cosimulation, le procédé le plus pénible et le plus générateur d'erreurs est la génération du lien entre les environnements de simulation matériel et logiciel. Pour surmonter ce problème, nous avons mis en œuvre un outil de génération automatique d'interfaces de cosimulation, nommé VCI (interface VHDL-C). Cet outil prend comme entrée un fichier de configuration défini par l'utilisateur, qui indique les caractéristiques désirées de la cosimulation (interface d'E/S et synchronisation entre les outils de mise au point) et produit une interface prête à effectuer la cosimulation VHDL/C. Cette interface permet de gérer plusieurs applications C embarquées dans un même système. Elle est indépendante du style de description ou du protocole de communication utilisateur et suffisamment robuste pour supporter un fonctionnement imprévisible de l'application.

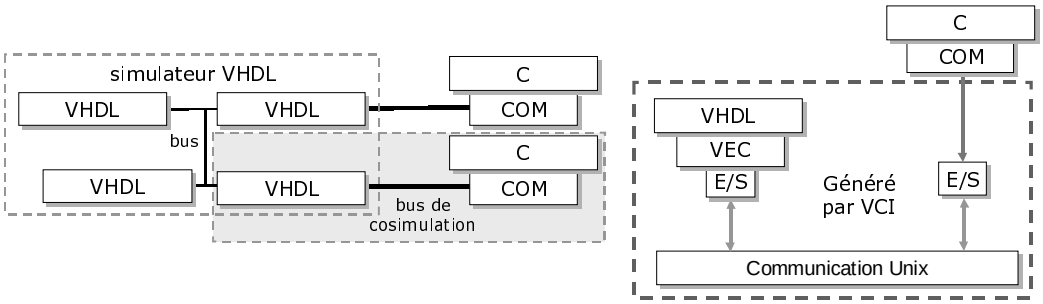


figure 3.6 : Modèle de cosimulation distribuée VHDL-C

Un système mixte logiciel/matériel est représenté en VHDL comme un ensemble de modules interconnectés. Le côté gauche de la figure 3.6 représente un système formé par l'interconnexion de quatre composants. Deux d'entre eux sont des modules logiciels, introduits par l'utilisateur, qui seront exécutés par des programmes ou par un autre simulateur. Dans notre exemple, chaque module logiciel

corresponds à un programme C donné par l'utilisateur. Le programme C peut employer un protocole de communication pour échanger l'information avec le monde externe. Dans l'hypothèse où le programme C agit avec un ensemble de modules matériels, le protocole de communication exécute des opérations d'E/S sur les ports du coté du matériel (bus Hw) en appelant des primitives d'E/S.

L'outil VCI produit, pour chaque programme C, une interface de cosimulation VHDL-C. En partant du fichier de configuration défini par l'utilisateur, le programme C est encapsulé dans une entité VHDL pour être utilisée comment composant du système distribué. La partie droite de la figure 3.6 représente en détail la structure d'interconnexion VHDL-C pour l'un des modules logiciels. Toutes les parties mises en valeur dans la figure sont produites automatiquement par VCI (voir la prochaine section).

Chaque programme C et l'ensemble des modules matériels sont traités comme des processus Unix indépendants mis en communication par des bus de cosimulation. En d'autres termes, un bus de cosimulation est le lien entre les primitives d'E/S employés par le programme C et son interface (les ports d'E/S de l'entité de VHDL). L'entité VHDL accède au bus par les procédures VEC. Ces procédures accomplissent les tâches de conversion des types de données VHDL-C, la synchronisation entre les simulateurs ainsi que la propagation des événements entre le simulateur VHDL et le bus de cosimulation.

3.6.1 Flot pour la génération d'interfaces C-VHDL

L'outil de génération de l'interface VHDL-C (VCI) crée automatiquement les éléments nécessaires pour la cosimulation distribuée C/VHDL. La figure 3.7 récapitule l'utilisation de l'outil VCI pour produire les fichiers nécessaires pour la cosimulation. Selon le fichier d'entrée (description d'interface VCI), VCI produit les fichiers VHDL et C requis pour interconnecter le programme C à la structure VHDL pendant la cosimulation. Il est important de noter que l'outil peut alternativement être utilisé pour produire l'interconnexion entre deux modules logiciels et la connexion d'autres types de simulateurs au simulateur VHDL (voir section 3.6.4).

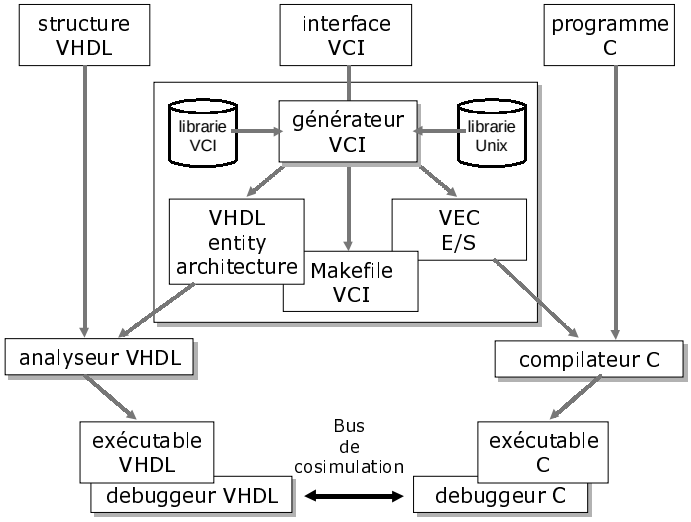


figure 3.7 : flot de génération de l'interface de cosimulation VHDL-C

L'entrée de VCI est un fichier qui spécifie l'interface du programme C et fournit d'autres informations nécessaires pour personnaliser la cosimulation ; ceci sera détaillé dans la prochaine section. La sortie résultante comprend l'interface VHDL/bus de cosimulation du côté matériel, l'interface C/bus de cosimulation du côté logiciel, et un fichier *make-file* qui contient un script Unix pour la génération de l'interface. L'interface VHDL/bus est composée de trois parties : l'entité VHDL

(qui encapsule le programme C), une description de son architecture VHDL (qui contient la déclaration des procédures VEC présentées par l'attribut VHDL "*foreign*"), et l'interface VEC/bus (procédures C exécutées par le simulateur de VHDL qui communiquent avec la partie logicielle par les primitives d'E/S). L'interface C/bus se compose des primitives E/S (employés pour le programme C pour communiquer à travers le bus de cosimulation). Finalement, l'exécution du fichier *make-file* fait le lien de ces primitives au programme C et compile l'interface VHDL du programme. Après la compilation, cette interface VHDL sera prête à être utilisée dans la structure VHDL.

3.6.2 L'environnement de cosimulation

L'environnement de cosimulation est initialisé par le lancement des simulateurs (VHDL et programmes C). Cette opération est effectuée pour un simple fichier shell. Le bus de cosimulation est établi par le programme C. La simulation ne commence réellement que quand tous les bus de cosimulation ont été crées. Cela garantit une initialisation correcte de toutes les communications. A la fin de la cosimulation l'utilisateur peut quitter le simulateur VHDL, ce qui entraine aussi la destruction de tous les processus C et des bus de cosimulation.

La figure 3.8 représente un écran de cosimulation correspondant à un modèle distribué composé de deux modules C et des modules VHDL. Le côté droit de l'écran correspond au simulateur de VHDL (en haut à droite) et les formes d'onde de la simulation de la structure VHDL (en bas à droite). La partie gauche représente deux fenêtres: elles correspondent à l'exécution du programme pour la mise au point du code C (en haut à gauche) et à la trace d'exécution de ces programmes (en bas à gauche).

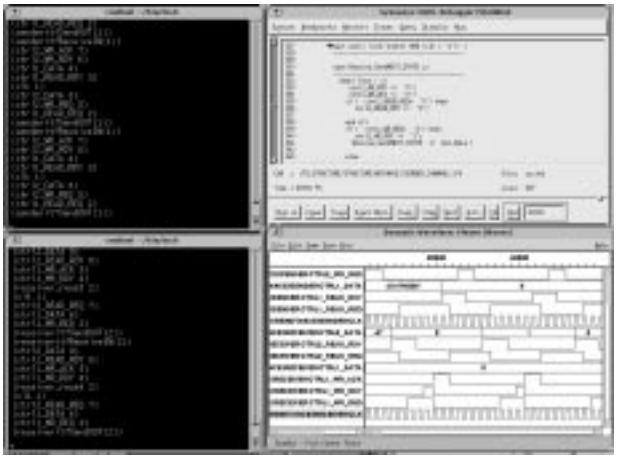


figure 3.8 : écran de cosimulation

3.6.3 Fichier de spécification VCI

L'outil VCI trace les ports situés du côté matériel (VHDL) dans des variables situées du côté logiciel. Les événements produits des deux côtés sont propagés à travers un canal de communication Unix. L'interface contrôle la propagation des événements, synchronisant l'exécution des simulateurs du matériel et du logiciel. Ce mécanisme est produit à partir d'un fichier qui décrit l'interface du module logiciel en termes de ports VHDL.

Le côté gauche de la figure 3.9 correspond à une description d'entrée typique pour l'outil VCI. L'entité VHDL générée est utilisée pour encapsuler le programme C. Cette entité est représentée sur le côté droit de la figure 3.9. Le fichier d'entrée VCI décrit l'interface du module logiciel comme un bloc matériel. Il spécifie le nom, la direction (IN, OUT et INOUT), le type et la sensibilité des ports. Le type d'un port est indiqué par son nom de type et par sa dimension dans le cas des vecteurs. L'outil supporte les types prédéfinis VHDL 1076-1987, standard logic d'IEEE, les vecteurs et les types définis

par l'utilisateur. La clause de sensibilité permet d'indiquer quels sont les signaux utilisés pour le contrôle de la synchronisation. Elle sera décrite dans la prochaine section.

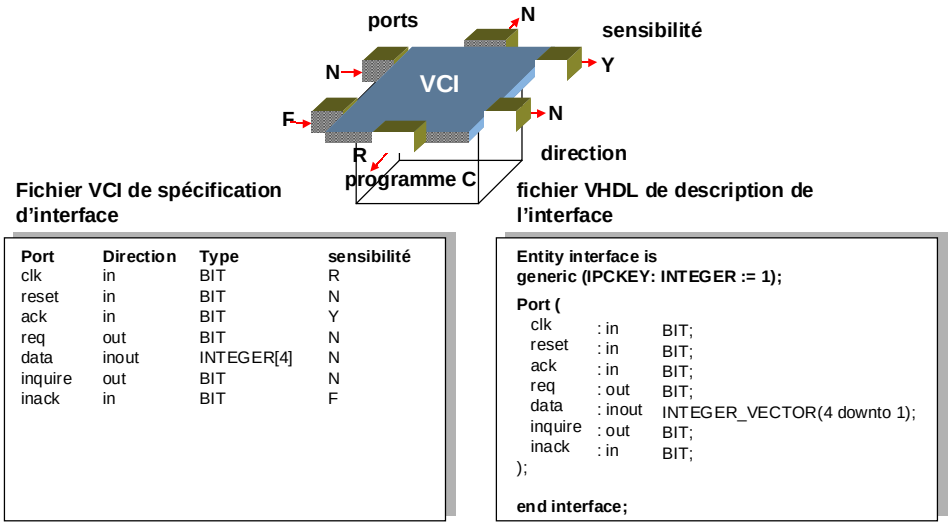


figure 3.9: Fichier de spécification VCI et interface VHDL équivalente

3.6.4 Synchronisation

La configuration de l'environnement de cosimulation est indépendante de la spécification de l'interface. Les clauses de sensibilité et les modes de couplage indiquent de quelle manière les simulateurs échangent leurs données à travers le bus de cosimulation. Pour la mise en application du bus de communication on peut utiliser soit la communication par IPC (file de messages ou mémoire partagée) soit la communication par socket.

L'outil VCI permet à l'utilisateur, en combinant les clauses de sensibilité et les modes de couplage entre les simulateurs, de produire une interface selon l'environnement de cosimulation souhaité. Par défaut, une transaction est engagée régulièrement entre le simulateur VHDL et l'application C. Cette régularité élimine immédiatement tout risque de blocage de la communication. Cette synchronisation permet aussi de gérer plusieurs applications C connectés au simulateur VHDL.

Les clauses de sensibilité

La clause de sensibilité est employée pour énoncer les conditions nécessaires à la propagation des événements sur les ports de l'interface. La clause de sensibilité peut être R (front montant), F (front descendant) et Y (oui pour l'événement). Les clauses déclarent que les valeurs sur les ports de l'interface sont actualisées lorsque les événements du port choisi changent de bas en haut (état R) ou de haut en bas (état F) ou chaque fois qu'il y a un événement sur le port (état Y). L'utilisateur peut indiquer le mode désiré de synchronisation en associant des clauses de sensibilité aux ports de l'interface. Dans le cas où tous les signaux d'entrée de l'interface seraient désignés comme sensibles, le mode de transfert par événement est indiqué. Lorsque la clause de sensibilité n'est pas utilisée (aucune sensibilité ou N), une cosimulation avec prélèvement des entrées à des intervalles de temps fixés par l'utilisateur sera effectuée. Par défaut l'intervalle de temps est fixé par le temps de base du simulateur. Dans ce dernier cas, un simulateur continu peut être connecté (ce qui est utile pour connecter les simulateurs continus comme Spice ou Matlab). Lorsque l'interface n'a pas de ports d'entrée (par exemple un générateur de signaux ou d'interruption), alors sera effectuée une cosimulation avec prélèvement des sorties à des intervalles de temps fixe.

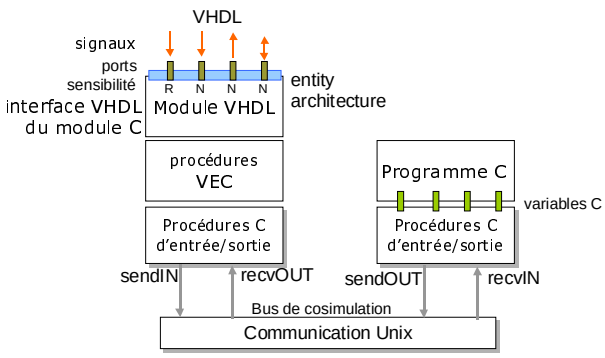


figure 3.10: Interface VHDL-C

La figure 3.10 représente la structure de l'interface C-VHDL produite par l'outil VCI. Au cours de la cosimulation, une procédure VEC est appelée chaque fois qu'il existe un événement sur un ou plusieurs des ports d'entrée de l'entité VHDL associée au programme C. Après que tous les événements d'entrée du temps correspondant ont été programmés, si l'événement sur l'un des ports d'entrée satisfait la clause de sensibilité (R, F ou Y), on exécute les primitives d'E/S (sendIN et recvOUT) pour actualiser les valeurs de l'interface à travers le bus. Du côté logiciel, les primitives d'E/S, sendIN et recvOUT sont employés pour actualiser les valeurs associées aux ports d'E/S de l'interface. Dans l'exemple suivant, le bus est mis en application par une unique file de messages bidirectionnelle (IPC message queue). Le bus utilise deux structures de données (INS et OUTS). L'information permutée entre les deux parties contient les valeurs de tous les ports de l'interface.

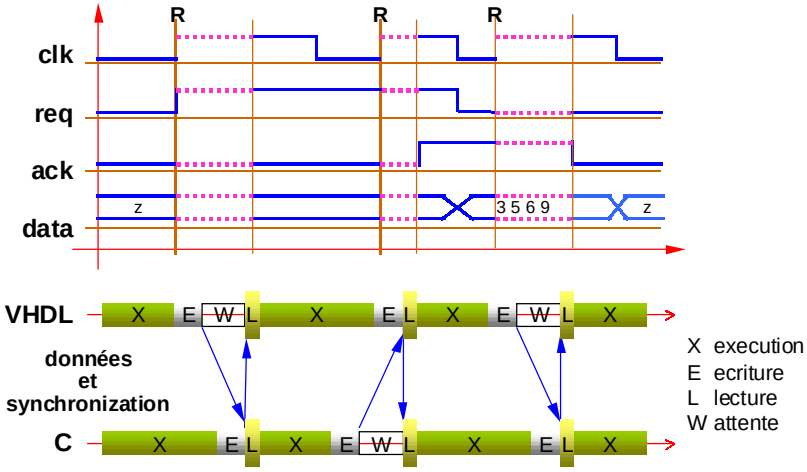


figure 3.11: Exemple de synchronisation

La figure 3.11 est un exemple de synchronisation dans laquelle on utilise un signal d'entrée d'horloge (*clk*) comme signal d'échantillonnage. Ce genre de synchronisation est normalement employé pour modéliser la fréquence d'exécution du logiciel. En indiquant la sensibilité du port de l'horloge comme "montant" ou R, chaque fois que le signal d'horloge change de zéro à un les valeurs de l'interface sont permutées à travers le bus. Ce mode de synchronisation est généralement employé pour exécuter une cosimulation au niveau du cycle (*cycle-true cosimulation).* On exécute le simulateur et les programmes C en parallèle jusqu'au moment où une exécution d'E/S est exigée du côté logiciel ou si la condition de sensibilité d'un quelconque de ces ports de l'interface est satisfaite.

Les modes de couplage entre les simulateurs

Les modes de couplage désignent de quelle façon les simulateurs échangent leurs valeurs à

travers l'interface. La vitesse et l'efficacité de la cosimulation dépendent du mode de couplage et des moyen de communication Unix utilisés.

L'utilisateur dispose du mode de couplage maître-esclave qui peut être utile pour les applications orientées données. Ce mode, représenté par le schéma du temps d'exécution de la figure 3.12(a), produit une exécution alternée des simulateurs. Le simulateur esclave (programme C) attend l'arrivée des événements d'entrée qui provient du simulateur maître (simulateur VHDL). Le simulateur maître est arrêté pendant l'exécution du simulateur esclave en attendant les événements générés par ce dernier.

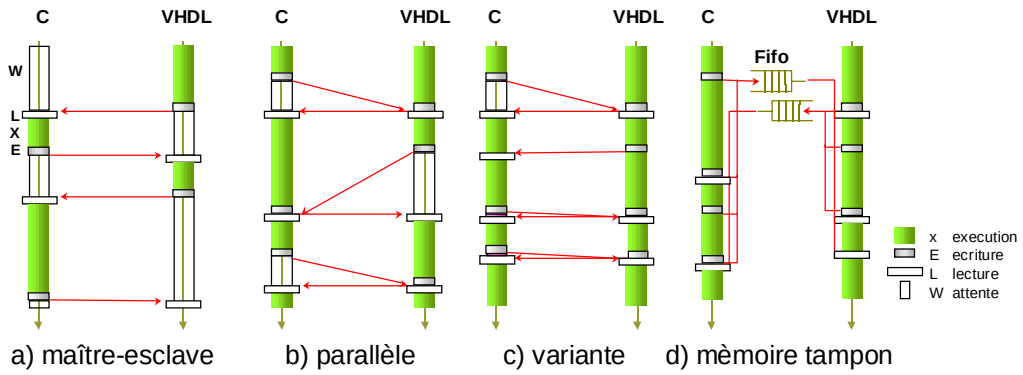


figure 3.12: Modes de couplage

Le mode de couplage parallèle, représenté par la figure 3.12(b), permet une exécution simultanée des simulateurs. Les deux simulateurs progressant en parallèle et la synchronisation est assurée des deux côtés. Les clauses de sensibilité sont utilisées du côté VHDL pour réguler les transactions. Un événement de sortie du VHDL ou un prélèvement d'entrée peut commander la transaction. Du côté C la transaction n'est engagée que lorsqu'une instruction d'E/S est invoquée, quelle que soit la direction de cette entrée-sortie. Ainsi, nous nous assurons que l'échange n'est effectué que si cela est nécessaire. Chaque simulateur envoie ses données et reste en attente pour la réception des données de l'autre côté. Pour éviter que l'exécution du simulateur VHDL soit bloquée pendant la période de cette attente (empêchant l'exécution des autres processus), des événements de réveil (*wake-up*) sont stockés dans la file interne du simulateur jusqu'à la réception de la donnée. Cet échange s'effectue chaque fois que les deux simulateurs commandent une transaction. Nous notons qu'avec ce mécanisme, un message ne peut être écrit que si le précédent a été lu. Par conséquent, le choix du bus de cosimulation n'affecte pas la cosimulation.

Une variante du mode parallèle est représentée par la figure 3.12(c). Le programme C prend immédiatement les valeurs d'entrée, mais attend que les valeurs de sortie soient prises par le simulateur VHDL. Du côté VHDL, le simulateur ne prend les valeurs d'entrée que s'ils sont présents. Ce mode permet un enchaînement d'événements en fonction du temps, ce qui permet la synchronisation avec un simulateur continu.

Le dernier mode de couplage utilise un bus de cosimulation basé sur des files de messages. Les transactions commandées du côté du programme C doivent être gérées par un arbitre. Cet arbitre doit synchroniser le temps de simulation des deux simulateurs. Cette option est fournie pour permettre la connexion de l'interface à une plate-forme de cosimulation (simulator backplane).

3.7 Applications

L'outil VCI a été utilisé pour la conception de plusieurs applications. Cette section détaille l'utilisation de VCI pour la conception de deux exemples : un système industriel appelé Videophone Codec et un contrôleur de moteurs. Le premier exemple illustre une méthode complète de codesign

utilisant la cosimulation C-VHDL. Le second exemple est retenu pour illustrer la procédure de génération de l'interface et l'utilisation de l'interface pour une cosimulation C-VHDL-Matlab.

3.7.1 Le Videotéléphone Codec

L'outil de génération de l'interface VHDL-C (VCI) a été utilisé dans un projet de SGS-Thomson : une nouvelle conception du Videotéléphone Codec (le STi1100 [108]). La cosimulation C-VHDL est employée pour valider la fonctionnalité du logiciel embarqué (embedded software), qui sera ensuite compilé dans le code binaire en utilisant un compilateur multi-cible. La même source C doit être utilisée pour la cosimulation et pour l'implémentation. Cette méthodologie a été utilisée pour deux processeurs du système. Les résultats ont prouvé que cette validation fonctionnelle permet de réduire la durée du cycle de conception. Plus de details sur cette application se trouvent dans [151].

Architecture du système

Le Videotéléphone Codec prend comme entrée un flux continu d'images, exécute le compactage et le codage de celles-ci. Il émet des chaînes de bits (*bits-streams*) sur la ligne téléphonique. En parallèle, des chaînes de bits sont reçues de la ligne téléphonique, décodée et décompressé, puis affichée sur l'écran. Le système (représenté par la figure 3.13) regroupe un ensemble d'opérateurs. Ces opérateurs communiquent par un bus de commande et un bus de données (contrôlé par un contrôleur de mémoire).

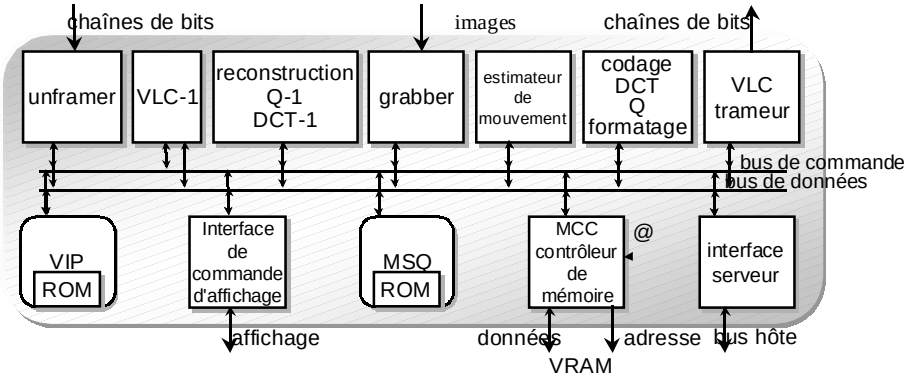


figure 3.13 : Schéma fonctionnel du Videotéléphone Codec

Le bus de commande est contrôlé par un processeur programmable (MSQ). Le contrôleur de mémoire (MCC) arbitre l'accès des différents opérateurs à la mémoire RAM de la vidéo (VRAM). Tous les opérateurs en rapport avec le bus central utilisent le même protocole de communication. Certains de ces opérateurs sont des blocs matériels (DCT, I-DCT, contrôleur d'affichage et estimateur de mouvement), d'autres sont les processeurs d'application spécifique (les processeurs MSQ et VIP). Une description détaillée de la fonctionnalité de la puce est donnée dans [109].

Flot de conception

La figure 3.14 représente le flux de conception pour les parties logicielles du Videotéléphone Codec. Des algorithmes C pilotent les deux ASIPs, le MSQ, MCC et le VIP (processeur d'image Vliw). Ils sont compilés en micro-code en utilisant un compilateur multi-cible (*retargetable compiler*) [82].

Le code d'application C (exécutable) est compilé sur le poste de travail et cosimulé avec le modèle VHDL du reste du système (on ne prend pas en compte le modèle du processeur). On utilise seulement l'interface du processeur. Une fois la fonctionnalité du logiciel validée, le même code C est compilé dans le code de la mémoire ROM en utilisant un compilateur multi-cible. Le code produit est

chargé dans la mémoire ROM du modèle VHDL du processeur, pour ensuite être simulé avec le reste du système au niveau RT.

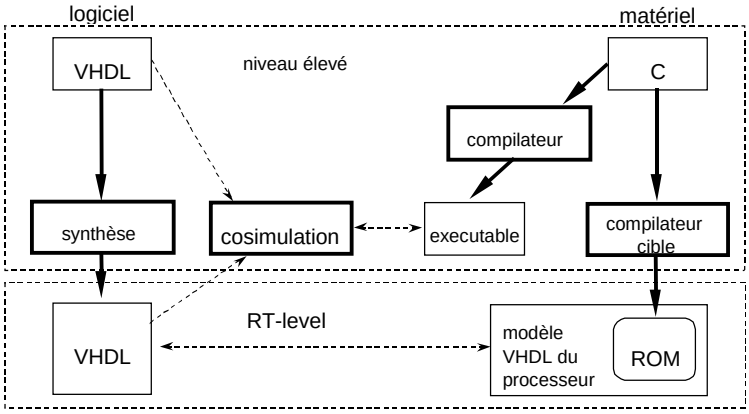


figure 3.14 : Flux de conception pour un processeur encastré et son logiciel

Configuration de l'utilisateur

La cosimulation concerne l'application C et le modèle VHDL du reste du système. Etant donné la spécification de l'interface du processeur, l'outil VCI produit automatiquement l'interface C-VHDL pour la cosimulation. Ensuite, le code généré est joint au code d'application et compilé sur le poste de travail [151].

Pour préserver des contraintes de la réalisation du système, l'utilisateur a été amené à encapsuler l'interface de cosimulation produite par l'outil VCI. Du côté matériel, le bus du système exécute un mécanisme de communication spécifique. Ce bus (contrôlé par deux signaux d'horloge) inclut les types VHDL définis par l'utilisateur et les modules VHDL développés par plusieurs concepteurs. Du côté logiciel, le processeur cible utilise une fonction d'E/S de bibliothèque appelée "io_transaction". Cette fonction aurait deux types de contenu : le premier sera utilisé pour la cosimulation et le second pour la cosynthèse.

Fonction d'adaptation C

Dans la partie C, il est nécessaire de définir une syntaxe stricte pour que les fonctions d'E/S gardent un code C simple pour la cosimulation et la compilation sur le processeur cible. Ces fonctions doivent être traduites pour la cosimulation et compilées en assembleur (tracé sur la mémoire RAM) pour le processeur cible. Par conséquent, pour la cosimulation on a introduit la fonction "*io_transaction*" pour faire appel aux primitives d'E/S produites par VCI (*MSQSendOUT* et *MSQReceiveIN*). Ces fonctions font l'écriture et lecture des ports de l'interface à chaque opération d'E/S.

Bloc de connexion VHDL

Dans la partie VHDL, le système utilise une interface générique qui fait abstraction des ports d'E/S du processeur. Cette abstraction est exigée pour préserver le système des modifications postérieures du processeur cible en développement. Cette interface générique doit se conformer aux conditions de synchronisation du bus du système. Par conséquent, l'interface est reliée au bus du système par un bloc de connexion [151].

Synchronisation

La synchronisation entre les programmes C et le simulateur VHDL a été mise en application par un protocole de communication alterné synchrone fondé sur l'échange continu des mises à jour. Les échanges réguliers de données C-VHDL sont commandés à l'aide d'une horloge d'évaluation (clk_eval), dérivée de l'horloge principale. Le transfert de données se fait alternativement sur chaque front du signal d'horloge. Pour mettre en application ce mécanisme de synchronisation, la sensibilité n'a été seulement placée que sur les front de l'horloge d'évaluation (dans le fichier d'entrée VCI). Les autres ports ne sont pas sensibles. La figure 3.15 représente la synchronisation entre les parties C et VHDL. Il s'agit du modèle similaire au décrit précédemment dans la section 3.6.4.

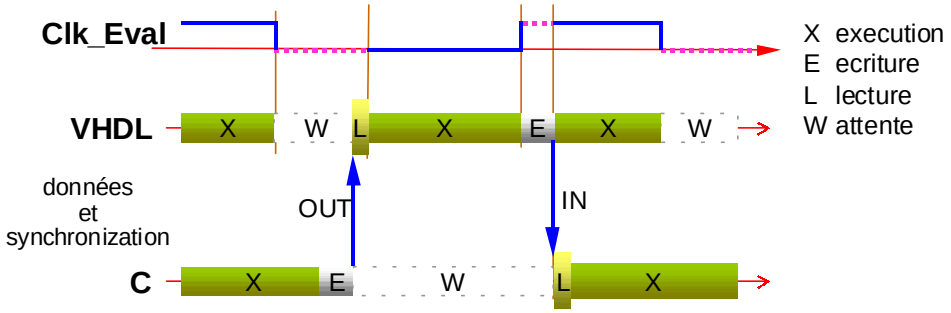


figure 3.15 : Synchronisation personnalisée C-VHDL

La transaction s'effectue en deux étapes. A un moment donné, l'application C engage une transaction en appelant une fonction spécifique appelée *c_transaction*. Celle-ci envoie une mise-à-jour de tous les ports de sortie et se bloque en attente de lecture de mise-à-jour des ports d'entrée. Le simulateur VHDL, qui était en attente de lecture, lit alors les mise-à-jours de sortie. Après avoir fait la mise-à-jour des signaux internes, il reprend la simulation pour un demi cycle. Au demi cycle suivant, le simulateur envoie les mise-à-jour des ports d'entrée et reprend la simulation. Cela débloque l'application C qui était en attente de lecture, qui peut alors mettre à jour ses propres variables et reprendre son exécution interne jusqu'à la prochaine entrée-sortie.

Afin d'optimiser la fréquence des transactions entre les simulateurs, un second mode de synchronisation a été implémenté. Il permet de modifier la fréquence des transactions pendant la cosimulation selon l'avancement de celle-ci. Ce mode est particulièrement adapté aux applications où la communication n'est pas très fréquente ou pas constante durant l'ensemble de la cosimulation [151].

3.7.2 Résultats expérimentaux

L'application de cette méthodologie de cosimulation à un grand projet industriel permet de mesurer la méthode au niveau de la durée du cycle de conception et de la vitesse de la simulation.

Nous comparons la cosimulation C-VHDL à la simulation entièrement effectuée en VHDL (modèle du processeur cible avec le programme chargé dans la mémoire ROM). Le délai d'exécution de la simulation (simulation runtime) est défini comme suit :

- pour la cosimulation C-VHDL, le délai d'exécution de la simulation est l'ajout du temps passé pour exécuter l'algorithme C compilé (y compris la communication Unix), et le temps passé pour simuler le reste du système dans VHDL ;
- pour la simulation VHDL avec le modèle du processeur, le délai d'exécution de la simulation est le délai d'exécution du simulateur VHDL (modèle du processeur cible et de son environnement).

Cette expérience a été faite pour six algorithmes différents [151]. La figure 3.16 correspond à une comparaison entre le délai d'exécution de la cosimulation et le délai d'exécution de la simulation

VHDL obtenue pour les différents algorithmes. Les algorithmes sont placés sur l'axe des abscisses dans un ordre croissant de complexité.

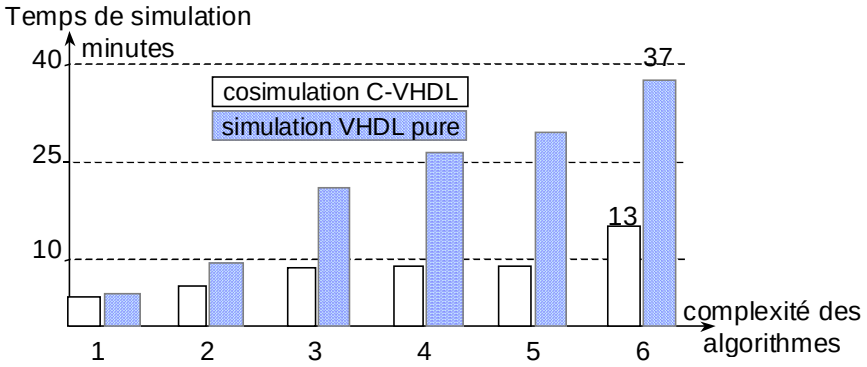


figure 3.16 : Simulation RTL contre la cosimulation

La cosimulation est toujours plus rapide que la simulation VHDL pure, et cette différence s'accroît avec la complexité des algorithmes. Deux raisons peuvent expliquer ces résultats :

- le simulateur VHDL dépense beaucoup de temps pour simuler le comportement du processeur pour chaque instruction d'assemblage, tandis que l'instruction C équivalente est simplement exécutée sur le poste de travail ;
- le code supplémentaire dû à la couche de communication IPC devient négligeable comparé au code d'application, à mesure que la complexité de l'algorithme augmente.

Ces résultats dépendent de la configuration de la plate-forme de cosimulation. En dépit du fait d'utiliser seulement un canal Unix par chaque module logiciel, la cadence de la propagation des données entre les simulateurs dépend du genre de canal Unix choisi (IPC ou socket). La vitesse de cosimulation dépend également du mode de synchronisation des simulateurs et du nombre de processeurs employés par la plate-forme de cosimulation. Ces variables affectent la vitesse et l'exactitude de la simulation. Cependant, les résultats sont tout à fait importants dans le domaine des systèmes embarqués. L'environnement de cosimulation permet d'utiliser les outils de développement standard du langage C (debugger, profiler, browser) pour valider le code du logiciel bien avant de concevoir le processeur cible, et rapidement modéliser et valider l'interface du processeur. La conception simultanée du code du logiciel et du processeur réduit considérablement la durée du cycle de conception.

3.7.3 Le contrôleur de moteurs : Application à la cosimulation C-VHDL-Matlab

Le contrôleur de moteurs ajuste les paramètres de position et de vitesse d'un ensemble de moteurs [89][111]. Ce système commande un ensemble de moteurs produisant un mouvement continu uniforme. Par exemple, la commande dans un espace 2-D a besoin d'un moteur pour chaque axe (X et Y) et d'un système associé de commande de mouvement. Ce système peut manipuler jusqu'à 18 moteurs. Cependant, afin de simplifier la présentation, nous retiendrons une application comprenant deux moteurs.

Le contrôleur de vitesse communique avec un ordinateur central (Hôte-Sw), et deux moteurs (moteurX) composent le système global (représenté par la figure 3.17). Le système est divisé en sous-systèmes communiquant et en unités de communication associées (fifoX). Chaque moteur est décrit par un programme Matlab, qui modélise son comportement. Ceci permet d'utiliser un modèle précis pour les moteurs. Les modules logiciels sont interconnectés aux bus matériels par différents types de

ports : des ports d'entrée-sortie (du type real et integer) pour les signaux de données, et des vecteurs de bits pour les signaux de contrôle.

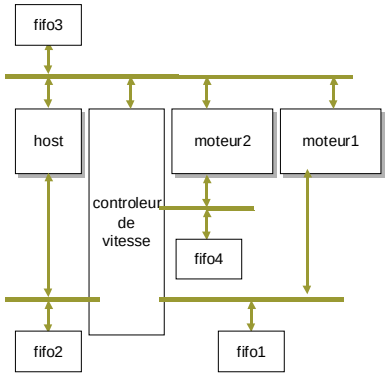


figure 3.17 : Système adaptatif de contrôle du moteur

Génération des interfaces

L'outil VCI a été utilisé pour produire l'interface VHDL-C pour chacun des modules logiciels et Matlab à partir de la description de l'interface correspondante. Par exemple, la figure 3.18 est une description de l'interface correspondant à l'un des moteurs. Des modules Matlab sont traités de la même façon que le programme C.

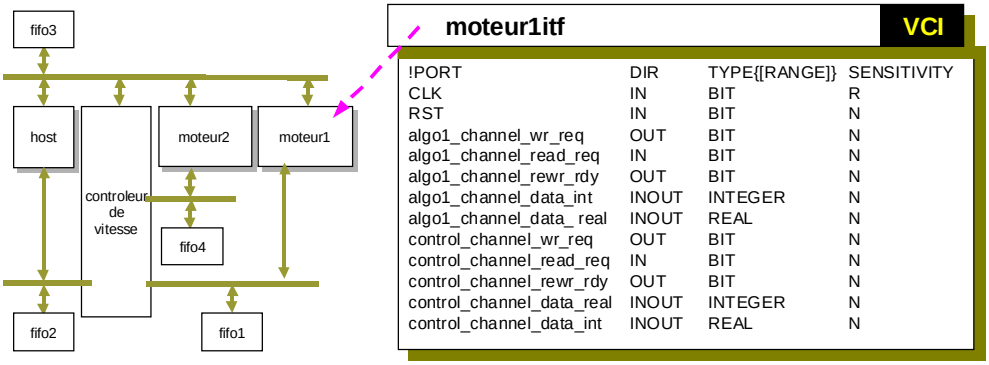


figure 3.18 : Fichier d'entrée VCI pour l'application de moteur

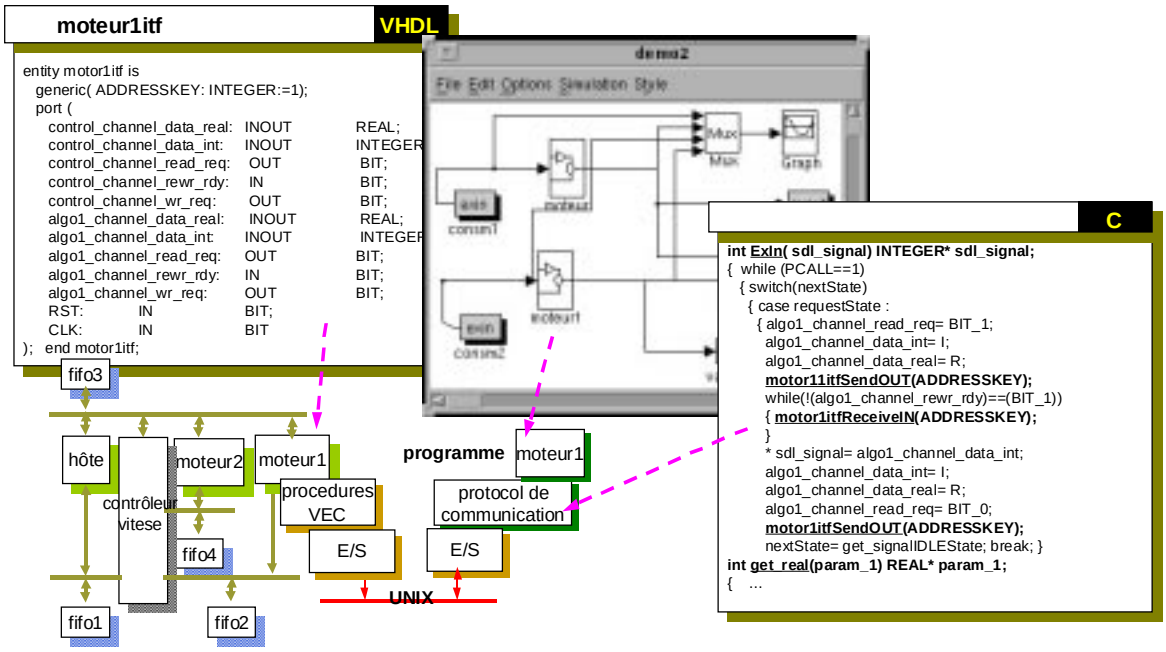


figure 3.19 : Modèles de cosimulation VHDL-C pour le moteur motor1

Selon le fichier d'entrée (moteur1itf.vci), l'outil VCI produit l'entité VHDL (utilisée pour interconnecter le module logiciel au bus matériel), les procédures VEC et les primitives d'E/S. La figure 3.19 représente le programme Matlab (moteur1) correspondant à l'un des moteurs ainsi que le protocole de communication utilisé (moteur1fu), les deux étant définis par l'utilisateur. Le programme Matlab communique en utilisant un protocole de communication composé de plusieurs primitives (*ExIn, ExOut*) aussi définies par l'utilisateur. Chaque primitive de communication (cellule Matlab qui encapsule un programme C) met en application son mécanisme de communication correspondant en appelant les primitives d'E/S produits par l'outil VCI (motor1itfSendOut, et motor1itfReceiveIn).

La cosimulation C-VHDL-Matlab

La cosimulation dans l'exemple retenu requiert un simulateur VHDL, le programme C de la machine hôte et les modèles Matlab des moteurs [111].

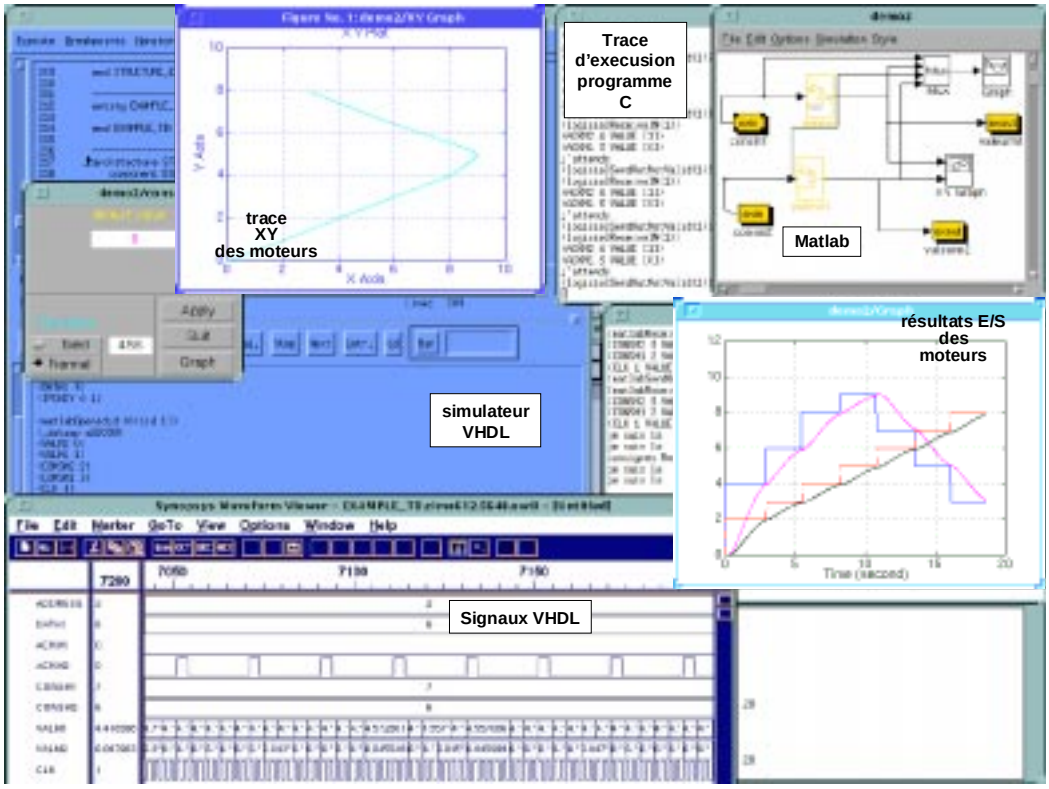


figure 3.20 : Résultats de la cosimulation du contrôleur du moteur

La figure 3.20 représente un écran de cosimulation avec des formes d'onde correspondant à la simulation VHDL du contrôleur, l'exécution du programme C de la machine hôte et l'exécution des deux moteurs (*moteurX*). Chaque moteur a été modélisé et simulé par le simulateur continu Matlab connecté par l'outil VCI au simulateur VHDL. Les résultats de l'exécution des moteurs (*XY-plot*) et les valeurs d'E/S des moteurs pendant la simulation apparaissent sur le côté droit de la figure 3.20. Les valeurs discrètes d'E/S correspondent aux valeurs du contrôleur et les valeurs continues correspondent à la position des moteurs.

Dans le cas de cette application, l'utilisation de la cosimulation a évité de construire le prototype prévu au commencement du projet. La cosimulation a permis d'ajuster les paramètres de l'algorithme du contrôleur et valider la spécification de tous les deux (du logiciel fonctionnant sur la machine hôte et du modèle matériel du contrôleur). Le contrôleur a été simulé à trois niveaux d'abstraction [88]. Le modèle initial du contrôleur a été donné en VHDL comportemental. L'application d'un outil de synthèse comportemental a produit un modèle VHDL RTL qui a été cosimulé dans le même

environnement. L'application de la synthèse au niveau RTL a produit un modèle de portes qui à été également cosimulé. La même interface VHDL-C, produite par l'outil VCI, a été utilisée pour la cosimulation à tous les niveaux d'abstraction.

3.8 Intégration dans le projet Cosmos

L'outil VCI a aussi été utilisé par la phase de prototypage virtuel dans Cosmos [9]. Dans Cosmos, le flot de conception commence à partir d'un modèle du système décrite en SDL. Cosmos réalise le découpage et la synthèse de la communication pour générer une architecture distribuée composée de modules logiciels et de modules matériels. L'outil S2CV (Solar-to-C/VHDL), développé au cours de cette thèse, génère automatiquement des descriptions en langage C et VHDL en partant du langage intermédiaire Solar (le format intermédiaire de Cosmos) [8]. Les modèles générés en langage C et en langage VHDL ont été définis pour obtenir un environnement unifié et flexible qui permet la cosimulation et la synthèse du prototype virtuel. Une description structurelle du système est faite en VHDL pour décrire l'interconnexion entre les modules logiciels et matériels. Un fichier script est généré pour la compilation de tous les modules, la génération des interfaces de cosimulation et pour le lancement de la cosimulation.

Le format intermédiaire Solar, qui sera décrit dans le chapitre Cosmos, a été conçu pour fournir des facilités de modelisation à plusieurs niveaux d'abstraction ce qui permet le raffinement du modèle entre ces différents niveaux. Solar est définie par un ensemble de concepts qui permettent une conciliation efficace entre les aspects associés au langage et ceux qui correspondant à l'architecture. Une description Solar représente un système distribué où le comportement est recouvert sur un ensemble d'unités de conception. Ces unités de conception s'exécutent indépendamment des autres, jusqu'au moment où ils requièrent l'échange de données. Cet échange se fait à travers de signaux (ports) ou par l'utilisation de canaux de communication qui joignent les différentes unités de conception.

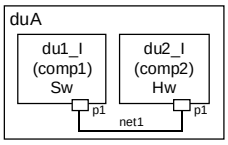
Les descriptions matérielles sont générées en vue de la synthèse architecturale. Les résultats de celle-ci sont fortement influencés par le style utilisé pour la modélisation ainsi que par l'outil de synthèse. Il est donc nécessaire de considérer ces éléments lors de la génération. Quant aux modèles logiciels, leur génération doit se faire en considérant l'architecture monoprocesseur cible. Ainsi, les spécifications initialement concurrentes doivent dans certains cas être transformées pour une exécution séquentielle.

Représentation structurelle

En Solar, une unité de conception permet la spécification de la structure d'un système donné ou le comportement d'un module. Une unité de conception structurelle est composée d'une interface (ensemble de ports), d'instances d'autres unités de conception (structurelles ou comportementales) et d'un réseau d'interconnexion. Les instances permettent l'utilisation d'unités de conception déjà existantes en décrivant une partie ou la totalité de leurs interfaces. Ceci évite la redéfinition des unités de conception déjà disponibles et permet la réutilisation de composants existants. D'autre part, plusieurs instances d'une même unité de conception peuvent être utilisées dans un même système. Le réseau d'interconnexion relie les instances des unités de conception.

La figure 3.21 résume la correspondance entre une unité de conception structurelle Solar et son équivalent en VHDL. Une unité de conception Solar et son interface sont traduites en une entité VHDL renfermant l'ensemble des ports de l'interface. La vue (*View*) associée à cette unité correspond à une architecture VHDL contenant l'ensemble d'instances et de réseaux d'interconnexion. Les instances sont traduites en composants VHDL (le mot clef *viewref* spécifie le nom de l'unité de

conception associée a cette instance). La traduction du réseau d'interconnexion Solar en VHDL exige la définition d'un signal qui interconnecte les ports des instances par l'instruction de connexion "*port map*". Le mot clef "*property design*" est employé pour préciser que l'unité de conception associée a cette instance sera traduite comme un composant logiciel ou matériel. Cette information est requise pendant la phase de configuration VHDL où chaque composant est associé à la entité et à l'architecture qu'il représente. Ainsi, l'architecture du système sera composée par de processus C dans le cas d'implémentation logiciel du composant et par des architectures VHDL dans les cas de composants matériels.



Solar

```
(DESIGNUNIT duA
    (VIEW viewA
        (VIEWTYPE "structure")
    (INTERFACE
        (PORT ...)
    (CONTENTS
        (INSTANCE du1_I
            (VIEWREF behavior comp1)
            (PORTINSTANCE (ARRAY p1 ... ))
            (PROPERTY DESIGN SOFTWARE))
        (INSTANCE du2_I
            (VIEWREF behavior comp2)
            (PORTINSTANCE (ARRAY p1 ... ))
            (PROPERTY DESIGN HARDWARE))
        (NET net1
            (JOINED
            (PORTREF p1 (INSTANCEREF du1_I))
            (PORTREF p1 (INSTANCEREF du2_I))
```

VHDL

```
entity duA  is
    port ( ... );
end du_name;
architecture viewA of duA is
signal net1 : BIT_VECTOR(1 to 2) ;
component comp1
    port (  p1 : in BIT_VECTOR(1 TO 2); ... );
end component;
component comp2
    port (  p1 : in BIT_VECTOR(1 TO 2); ... );
end component;
begin
    du1_I : comp1
    port map (  p1 => net1, ... );
    du2_I : comp2
    port map ( p1 => net1, ... );
end viewA;
configuration confstruct of duA is
    for viewA
        for du1_I : comp1
        use entity WORK.comp1(CLI); end for;
        for du2_I : comp2
        use entity WORK.comp2(behavior); end for;
    end for;
end confstruct;
```

figure 3.21 : Correspondance entre Solar et VHDL structurelle

Représentation comportementale

En Solar, la table d'état est le constructeur de base pour la description fonctionnelle. Une table d'état est une FSM étendue composée par une combinaison illimitée d'états et de tables d'état (hiérarchie comportementale). Des tables d'état sont employées pour indiquer le comportement des unités de conception, des unités de communication, des procédures et des fonctions. Tous ces états peuvent être exécutés soit séquentiellement, soit en parallèle ou tous les deux. Une table d'état a des attributs pour manipuler des exceptions, des variables globales et des états par défaut.

Solar

```
(STATETABLE table1
    (STATELIST init request finish)
    (ENTRYSTATE init)
    (STATE init
        (IF (= signaldata '1')
            (THEN
                (WAIT (UNTIL (= signaldata '0'))))
        (ASSIGN signalreg '1')
        (NEXTSTATE request))
    (STATE request
    )
    (STATE finish
    )
)
```

VHDL

```
architecture viewA of duA is
begin
process
    type table1StateType is (table1Idle, init, request, finish);
    variable table1NextState: table1StateType := init;
    begin
    wait until (rising_edge(CLK)) or (RST='1');
    if (RST='1') then .....
    StateTabletable1 : loop
    case table1NextState is
        when (init) =>
            if (signaldata = '1') then
                wait until (signaldata= '0');
            end if;
            signalreg <= '1';
            table1NextState:= request;
            exit StateTabletable1;
        when (request) => ...
        when (finish) => ...
        When (others) =>
            table1NextState:= init;
    end case;
end loop;
end viewA;
```

C

```
int duA()
{   typedef enum { table1Idle, init, request, finish } table1StateType;
    table1StateType table1NextState= init;
    int AGAIN=1;
    while (AGAIN==1)
    {   if (AGAIN==1) oneCylceStep();
        switch (table1NextState)
        {   case init:
            {   if (signaldata==BIT_1)
                    actualizeInput();
                signalreg= BIT_1;
                MapAddr(mtypesignalreg);
                table1NextState= request;
                break;
            }
            case request: ...
            case finish: ...
            default: table1NextState= init;
            break;
        }
    }
}
```

figure 3.22 : Correspondance entre Solar comportementale et le code C/VHDL généré

Un état Solar est défini comme un triple State := (A, T, C) où A est un ensemble d'actions à exécuter, T est un ensemble de transitions à prendre de l'état actuel et C est une condition qui détermine quelle transition prendre. L'ensemble de conditions inclut des exceptions globales aussi bien

que des essais internes. Les transitions entre les états ne sont pas restreintes à un niveau hiérarchique spécifique. En d'autres termes, les transitions peuvent traverser des bornes hiérarchiques (transitions globales). L'ensemble de transitions est déterminé par l'instruction NEXTSTATE dans un état et par les transitions globales qui affectent l'état. Cette construction permet une transition à un autre état quelconque dans la hiérarchie. Une action peut être une expression simple, un ensemble d'expressions à exécuter en parallèle ou ordonnée, ou ce peut être une déclaration hiérarchique. Une expression simple peut être une affectation, un rapport de branchement ou un appel de procédé. L'écoulement de commande peut être dans l'état, d'un état à un autre ou un appel de procédé. Le premier type concerne les opérateurs classiques comme IF, CASE, WHILE, EXIT et WAIT qui passent d'une commande vers l'autre dans le même état.

L'étape de la génération du code traduit les unités de conception fonctionnelles Solar en C ou VHDL selon que l'instance VHDL associée est logiciel ou matériel, respectivement (figure 3.22). Dans le cas de génération de code C, l'automate sera traduit comme un programme C. Le code C qui est généré est une instruction *SWITCH* où chaque état est représenté par une instruction CASE. Pour ce qui concerne les instructions sur des signaux (comme les instructions *WAIT* ou *ASSIGN*), les procédures de communication utilisent des primitives d'entrée/sortie qui font la synchronisation et l'échange de donnés à travers l'interface du programme. L'interface entre le programme et les ports de l'entité VHDL associée est généré par VCI. Dans le cas de la génération de code VHDL l'automate sera traduit comme un processus VHDL (*process*) contenant l'automate d'états finis. Les transitions entre l'état actuel et le prochain état sont synchronisées par le front montant d'un signal externe qui donne un intervalle de temps abstrait pour obtenir une trace équivalent à l'exécution du model SDL pendant la simulation. Chaque processus VHDL contient un ensemble d'instructions séquentielles. Parmi ces instructions, on distingue les opérations conditionnelles (*if*, *case*, etc.), les boucles (*loop* et *while*), les appels de procédures, les affectations de signaux et de variables, et les instructions d'attente (*wait until*). La description VHDL fait appel à des procédures de communication. Une procédure contient les protocoles d'échange de données avec le monde extérieur (par l'intermédiaire des ports).

3.9 Conclusion

Ce chapitre a décrit une solution de cosimulation C-VHDL qui offre des avantages par rapport aux approches précédentes. L'environnement de cosimulation permet la manipulation de toutes sortes d'architectures distribuées sans avoir à se soucier du mécanisme de communication utilisé, la cosimulation à différents niveaux d'abstraction ainsi qu'une douce transition vers le processus de cosynthèse. Cette méthodologie permet de garder le code d'application C dans sa forme originale pour la cosimulation et la cosynthèse. Par ailleurs, on peut utiliser des outils standard de simulation VHDL et de mise au point des programmes C. Cette flexibilité est obtenue grâce à un outil de génération automatique d'interfaces de cosimulation capable de créer un lien entre le logiciel et les environnements de simulation matériels. L'approche peut manipuler de nombreux modules logiciels ou matériels ; les outils de mise au point peuvent être utilisés conjointement ; des programmes de mise au point génériques ou spécifiques peuvent être utilisés ; et plusieurs modes de synchronisation peuvent être indiqués pour supporter différents types de scénarios de cosimulation.

Le lien de cosimulation logiciel/matériel repose sur l'interface d'E/S associée au module logiciel. Ainsi, l'outil n'a pas besoin d'une description complète du système pour produire la plate-forme de cosimulation. Ainsi, ce lien est employé par des primitives d'E/S du module logiciel. Par conséquent, la communication peut être abrégée dans plusieurs couches, ce qui permet la cosimulation à différents niveaux d'abstraction. Du côté matériel, les modules logiciels peuvent être utilisés comme des composants. Ceci permet de manipuler toutes sortes d'architectures distribuées. Les modules logiciels

peuvent être exécutés sur un ou plusieurs CPUs, selon la configuration du lien entre les simulateurs. Une autre flexibilité correspond à la capacité de spécifier plusieurs modes de synchronisation entre les simulateurs. Ceci permet de changer l'efficacité de la cosimulation en dehors de l'interface spécifiée.

L'utilisation de l'outil VCI dans de grandes applications industrielles a été validée dans [7]. Ce travail a été aussi à l'origine du développement de nouveaux outils de cosimulation permettant l'utilisation d'outils commerciaux (comme Synopsys et Cadence) et d'autres langages (comme Matlab et SDL) [111]. En effet, une version industrielle de VCI, nommé CoSim, fait actuellement partie de l'ensemble d'outils d'Unicad de SGS-Thomson Microelectronics [151].

Chapitre 4

Prototypage virtuel pour les systèmes mixtes logiciel/matériel

---

Le but de cette chapitre est de présenter une méthodologie pour le prototypage rapide de systèmes électroniques flexibles et modulaires contenant des parties logicielles et matérielles. La contribution principale de ce travail est l'aptitude à manier une large gamme d'architectures. Nous supposons que le partitionnement du système en parties matérielles et parties logicielles est déjà fait. Le prototypage virtuel est une étape qui génère du code exécutable pour chaque module du système. Cette étape du processus de codesign produit un prototype virtuel, c'est-à-dire une architecture hétérogène composée d'un ensemble de modules distribués décrits en VHDL, pour des éléments matériels, et en C, pour des éléments du logiciel. Ces modules interagissent par des modules de communication. L'approche proposée pour le prototypage virtuel est basée sur une stratégie de modélisation qui permet l'utilisation du prototype virtuel pour la cosynthèse (transposition des modules logiciels et matériels sur une plate-forme architecturale) et la cosimulation (qui est la simulation conjointe de composants logiciels et matériels) dans un environnement unifié. Cette stratégie emploie le concept de bibliothèque de vues multiples pour abstraire les détails de mise en œuvre et les problèmes de communication entre les modules. Ce chapitre a été publié dans : Design Automation for embedded Systems, Kluwer Academics Publishers, R. Camposano et W. Wolf editors, Vol. 2(3/4), May 1997. Pour cette raison l'article est reproduit comme il est.

4.1 Virtual Prototyping

The joint specification, design and synthesis of mixed hardware/software systems, also called codesign, is a recent development issue. The interest in codesign is driven by increasing complexity and the need for early prototypes to validate the specification and provide the customer with feedback during the design process [32][81].

Codesign aims to produce a heterogeneous architecture of mixed hardware/software components that implement an initial specification. As shown in figure 4.1, a typical Rapid System Prototyping (RSP) codesign flow starts with a system specification given in an existing language (such as SDL or StateChart) [17]. At this stage, the specification is composed of a set of interacting modules. The next step is partitioning. The partitioning process transforms a system level specification into a heterogeneous architecture composed of hardware and software modules. This model is generally called virtual prototype. The virtual prototype is a simulatable model of the system. The final step is prototyping. The prototyping or architecture mapping, produces an architecture that implements or emulates the initial specification.

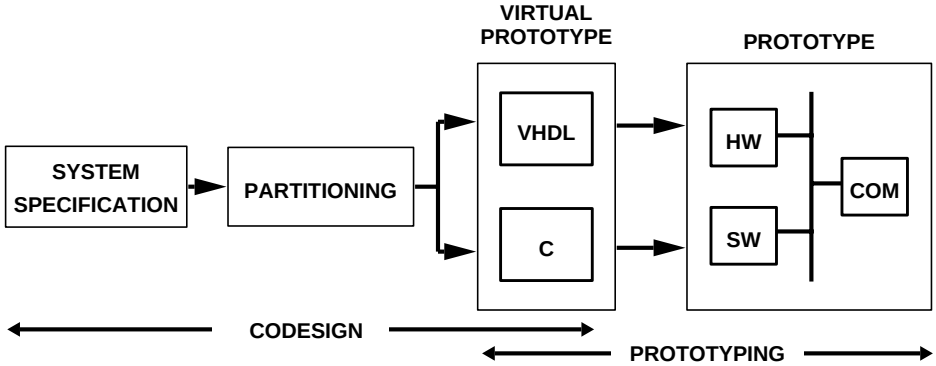


figure 4.1 : RSP and codesign flows.

One of the main difficulties in having this full design flow working is the link between the two last steps e.g. Virtual Prototyping and Prototyping. As it is shown more ahead, to take care of these steps most of codesign environments restrict the architectural model used.

The goal of this work is to develop a methodology for fast prototyping of highly modular and flexible electronic systems including both, software and hardware components. The main contribution of this work is the ability to handle a wide range of architectures. We assume that hardware/software partitioning is already made. Of course, this assumes that partitioning is made with a set of architectures in mind. This may be very useful in the case where the goal is to map a given function onto an existing architecture, where the communication is already fixed. In this case, the communication model is abstracted as a library component. At this stage we start with a heterogeneous architecture composed of a set of distributed modules, represented in VHDL for hardware elements and in C for software elements, communicating through communication modules (located into a library of components).

This work concentrates on the use of the virtual prototype for both cosynthesis (mapping hardware and software modules onto an architectural platform) and cosimulation (that is the joint simulation of hardware and software components) into a unified environment.

In the following section, we give a brief overview of existing codesign solutions and challenges. In section 4.2, we introduce the concept of the unified environment for cosimulation and cosynthesis. In section 4.3, we detail the modular and flexible architectural model. Section 4.4 describes the communication unit concept, the modular specification, and the unified model for cosimulation and

cosynthesis. The above concepts will be clarified by a real example (in section 4.5). Finally, in section 4.6, we conclude with perspectives and directions of future work.

4.1.1 Objectives

This work deal with the cosimulation and cosynthesis starting from mixed C, VHDL descriptions. The goal of this work is to combine the cosimulation and cosynthesis into a unified environment. The definition of a joint environment for cosynthesis and cosimulation posses the following challenges:

- Communication between the HW and SW modules,
- Coherence between the results of cosimulation and cosynthesis and
- Support for multiple platforms aimed at cosimulation and cosynthesis.

The first issue is essentially due to the following reasons: Mismatch in the HW/SW execution speeds, communication influenced by data dependencies and support for different protocols [92].

The second issue is coming from the fact that different environments are used for simulation and synthesis. In order to evaluate the HW, the cosimulation environment generally uses a cosimulation library that provides means for communication between the HW and the SW. On the other hand, the cosynthesis produces code and/or HW that will execute on a real architecture. If enough care is not taken, this could result in two different descriptions for cosimulation and cosynthesis.

The third issue is imposed by the target architecture. In general, the codesign is the mapping of a system specification onto a HW-SW platform that includes a processor to execute the SW and a set of Asics to realize the HW. In such a platform (ex. a standard PC with an extended FPGA card), the communication model is generally fixed. Of course, the goal is to be able to support as many different platforms as possible.

This chapter presents a flexible modeling strategy that allows taking care of the three previously mentioned problems. The general model allows separating the behavior of the modules (hardware and software) and the communication units. Inter-modules interaction is abstracted using communication primitives that hide the implementation details of the communication units.

4.1.2 Related work

Several researchers have described frameworks and methodologies for codesign [11] [21] [81] [93] [95] [96]. Codesign environments differ by the way in which the Hw/Sw are described, abstraction level and communication model used. Most of the previous work have been targeted towards either cosimulation or cosynthesis and very few of them tried to combine both. However, they do not address all the three problems mentioned in the previous section, especially that of supporting multiple platforms.

Codesign environments differs by the way in which the initial specification is described, synthesized, and the partitioned descriptions are evaluated. In the VULCAN cosynthesis system, the input language is HardwareC and the design system tries to gradually move hardware to software [97]. The Cosyma cosynthesis system (cosynthesis for embedded architectures) also starts with a single specification of the system given in C^x (a super-set of the ANSI C standard) [24]. The approach is software-oriented, and the input specification is translated into an internal graph representation suitable for partitioning [25]. These codesign tools needs to be linked to a rapid system prototyping environment including both cosimulation and cosynthesis.

Codesign environments also differs by the way in which they handle cosimulation and cosynthesis. Different methodologies have been applied to cosimulation, cosynthesis and codesign

platforms [76][92][98][96][99][100]. Most of the previous work have been target either cosimulation or cosynthesis and very few of them tried to combine both [96][99][100]. A multi-paradigm simulation environment (Ptolemy), described in [21], supports cosimulation of different domains and a variety of hardware-software cosimulation techniques. In Ptolemy, the design process starts from a single specification, and the simulation is performed on the mixed Hw/Sw description obtained after partitioning. The tool presents a nice approach to deal with coherence between co-simulation/co-synthesis and support for multiple platforms, but it uses a restricted class of architectures.

Communication between hardware and software is one of the main issues when dealing with cosimulation and cosynthesis. Some approaches make use of a fixed communication scheme depending on a fixed architectural platform [11][32][95][99], in which case the first two problems addressed earlier are easily handled. Although some of these architectures are flexible (e.g. supports a variable number of hardware and software processors) they use a fixed communication model to exchange information between protocols [24][57][97]. The COSYMA system makes use of a model of the Sw processor for Hw/Sw communication. In this manner, COSYMA ensures coherence between cosimulation and cosynthesis and provides estimation of performances. The VULCAN system also uses a processor model for the software component [97]. Therefore, it provides a very accurate simulation. In [101], hardware and software descriptions are treated as separate Unix processes which communicate through BSD (Berkeley Software Distribution) sockets. Since its targeting a specific application, it uses a fixed communication model between the hardware and software. A similar approach is described in [95]. The tool assists in mapping a specification onto a single processor with multiple Asics. Software executes on the development processor and communicates with a hardware simulator through Unix inter-process communication mechanisms, using message passing. The COBRA project uses a prototyping environment based on a fixed architecture [57]. The project makes use of a FPGA based prototyping board called SPARROW. Therefore, it supports standard processor integration under real-time conditions as well as processor emulation.

A more general approach for codesign, described in [62], performs simulation of Hw/Sw system at various stages of the codesign process. The methodology maintains two separate descriptions for Hw and Sw through the entire codesign process and communication is modeled as a message passing system. Communication primitives are inserted into the Hw/Sw descriptions to model the system communication. Using this model allows the communication between the Hw and Sw processes to take place without knowledge of the explicit details of the underlying bus interface. To handle the interaction between the Hw and Sw, four data transfer primitives are used. These primitives are inserted into the Hw and Sw descriptions to model the system communication. Although this approach increases the range of supported architectures (by working at a higher abstraction level of communication), the Hw/Sw cosimulation interface is implemented in terms of the supported communication modes. This approach solves the three explained challenges above, but even so; the user can only use the supported communication modes.

4.2 Methodology

The proposed methodology starts from a heterogeneous description composed of three parts: HW components described in VHDL, SW components as C programs, and communication component(s) to connect the above two parts (figure 4.2). The communication components, located into HW (VHDL) and SW (C) libraries, help to hide the possibly complex behavior of an existing platform. This methodology enables the user to profit from a wide range of communication schemes; abstracted by communication components and each sub-system can be treated independently of the communication scheme.

This methodology combines the co-simulation and co-synthesis into a unified environment, supporting multiple platforms [9]. The first step is to validate the heterogeneous description using a HW/SW cosimulation. The validation is performed by simultaneous cosimulation of VHDL and C descriptions. We use a VHDL-based cosimulation environment for VHDL descriptions. In that case, a VHDL entity is used to connect a HW module with that of SW.

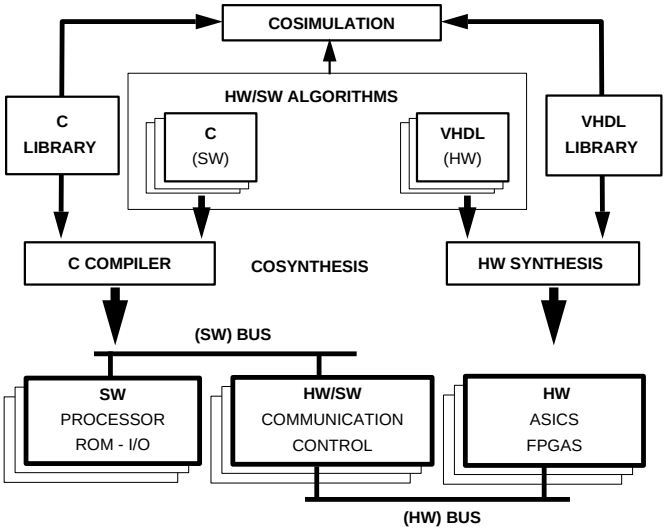


figure 4.2 : Methodology.

The same description will be used for co-synthesis as well. Each module can be synthesized using the corresponding tool. Hardware synthesis tools treat hardware (VHDL) components, while software (C) components are handled by available software compilers. High-level synthesis tools map HW components into Asics or FPGAs. Software compilers map SW components into processor, ROM, and I/O. The communication units are placed into a library of components and are not synthesized.

The modeling approach hides specific HW/SW implementation details and communication schemes, thus allowing the cosynthesis and cosimulation to start from the same description. System-level interaction is abstracted using communication primitives that hide the underlying communication protocol. Therefore, each sub-system can be treated independently of the communication scheme. This methodology enables the user to profit from a wide range of communication schemes. This will be introduced in the following section.

4.3 Architectural model

We use a modular and flexible architectural model. The general model, shown in figure 4.3(a), is composed of three kind of components: software components (aimed to execute C programs), hardware components (implements the VHDL descriptions) and communication components. This model serves as a platform onto which a mixed hardware/software system is mapped. Communication modules come from a library; they correspond to existing communication models that may be as simple as a handshake or as complex as a layered network.

The proposed architectural model is general enough to allow the representation of a broad class of Hw/Sw platforms. The model allows different implementation of mixed hardware/software systems, distributed architectures and several communication models. As shown in figure 4.3(b), a typical architecture will be composed of several hardware modules, several software modules and communication modules linking HW and/or SW modules.

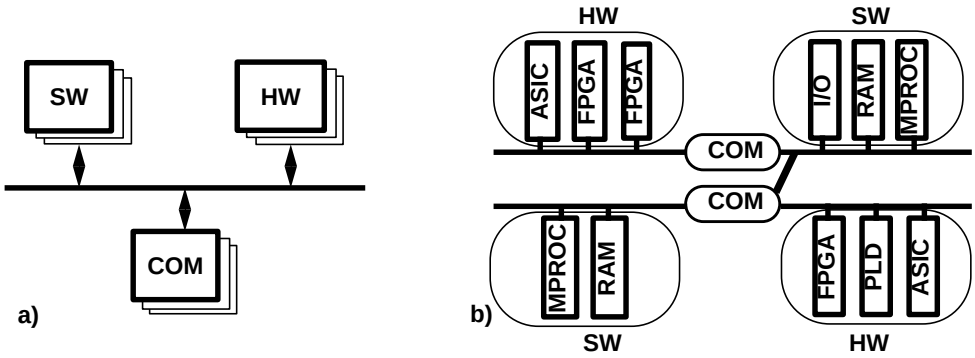


figure 4.3 : Architectural Model: a) architectural model. b) HW/SW platform.

4.4 Unified model for cosimulation and cosynthesis

This section details the unified model used for cosimulation and cosynthesis. The communication unit concept will be introduced first, then modular specification of the design will be explained. Finally, it will be shown that the combination of these concepts to the availability of multiple views for communication primitives allows modular design and design re-use of existing sub-systems.

4.4.1 The communication unit concept

Communication between sub-systems is performed via communication units which acts as a communication server for processors [102]. A communication unit is an entity able to execute a communication scheme invoked through a procedure call mechanism. The communication unit provides a fixed set of communication primitives (also called communication procedures, methods or services) to control access to the desired communication scheme. The model is known as the remote procedure call (RPC) [93].

To communicate, a sub-system calls a primitive (for memory access or message sending for example) as an ordinary procedure call. The communication unit unpacks the parameters, and sends a reply back to the caller. In other words, the communication unit acts as a co-processor (or server). Communication procedures, which correspond to the visible part of the communication unit, hide the communication protocol. The rest is completely transparent to the user and contains ports linking the method parameters to the controller.

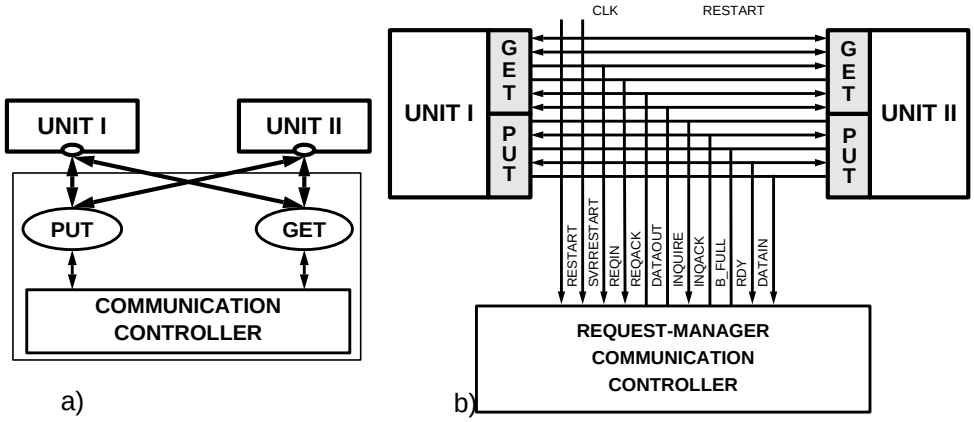


figure 4.4 : The communication unit concept: a) conceptual view, b) implementation view.

The communication unit can include a controller, which guards its current state as well as conflict-resolution functions. The complexity of the controller may range from a simple handshake protocol to as complex as a layered protocol. The procedures interact with the controller, which in turn modifies the unit's global state and synchronizes the communication. The communication modules are used as black boxes. The decision to make them in Hw or Sw is a partitioning decision that should be

made before cosimulation.

The figure 4.4(a) shows an abstract view of a communication unit linking two processes (Unit_I and Unit_II). Each process can be designed independently of one another. In this conceptual view, the communication unit is an object that can execute one or several procedures (get and put) that may share some common resource(s) (communication controller). This model hides all implementation details of the communication protocol. figure 4.4(b) shows a possible implementation of figure 4.4(a). All the procedure calls are expanded according to the protocol selected. Each module will include an interface that control data exchange with communication controllers.

The communication unit is a flexible communication model that allows to model most system level properties, such as message passing, shared resources, and other more complex protocols (as Communication Networks). This mechanism allows design re-use, because, a communication unit may correspond to either an existing communication platform, or a design produced by external tools, or to a subsystem resulting from an early design session. This concept is similar to the concept of system function library in programming languages.

4.4.2 Modular specification

The basic idea behind this work is to allow for modular specification of the design [103]. The communication scheme is described separately from the rest of the system. The use of procedures allows hiding the details related to the communication unit. All accesses to the interface of the communication unit are made through these procedures. The procedures fix the protocol of exchanging parameters between the sub-systems and the communication unit. This kind of model is very common in the field of telecommunication [104]. In this model, the communication primitives hide the communication network. The network may be generated automatically by the partitioning step or given by the designer.

The figure 4.5 shows a process that performs internal computations and exchange data with the external world. The process can be described as a FSM or made up of straight-line code segments [6]. The communication between the processes is done by using two procedures: get and put. A call to the put or get procedure indicates an interaction with another process to perform an input (get) or output (put) operation. In spite of the FSM-oriented form of description used within the environment, the style does not alter the interaction performed by the communication procedures. As we can see, the only information necessary to perform the desired communication scheme is the fact that we can execute two methods named get and put. The rest of the channel is completely transparent to the user.

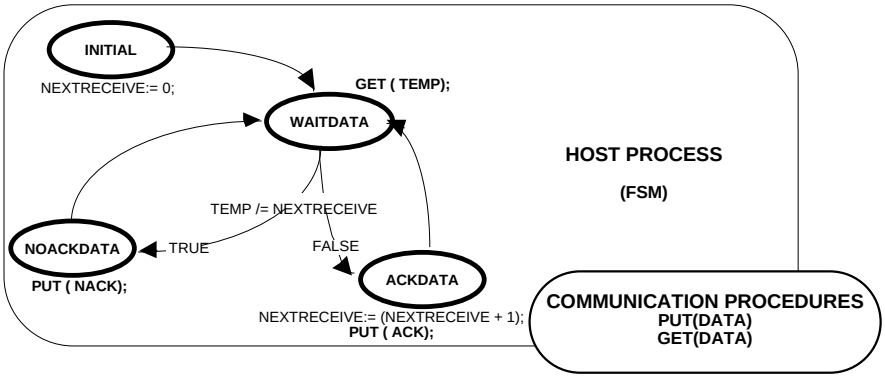


figure 4.5 : Access to the interface of a communication procedure.

We may have several protocols executing the same method. The communication between the sub-systems may be executed by one of the schemes (synchronous, asynchronous, serial, parallel, etc.) described in the library of communication units. The choice of a given communication unit will not

only depend on the communication to be executed, but also on the performances required and the implementation technology.

4.4.3 Multiple views for communication primitives

The use of a modular description scheme allows for separate evolution of the different modules. During the design process, several representations of the same object may be used at different design steps. During virtual prototyping, the modules are described at the behavioral level and the communication units are modeled as hardware or software processes. Later, after implementation, the communication units correspond to existing units.

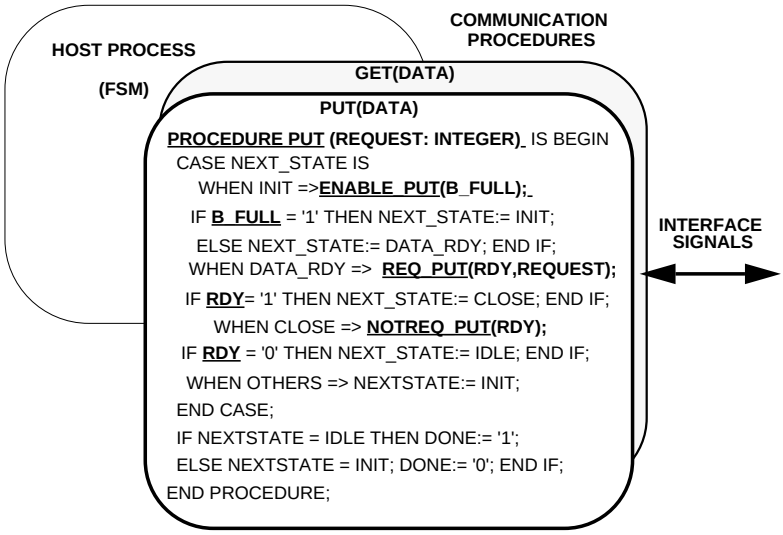


figure 4.6 : Hardware view (VHDL) of a communication procedure.

In order to allow the use of a communication unit at different design steps we need to describe its communication procedures into different views. Additionally, in order to be able to connect a communication unit to both, HW and SW modules, we have HW and SW views of a communication procedure. To support different applications, the number and type of views for each procedure will depend also on the co-simulation and co-synthesis environments used. The HW view (given in VHDL) may be common to both co-simulation and co-synthesis. In the case where we use different synthesis systems supporting different abstraction levels (e.g. a behavioral synthesis and an RTL synthesis), we may need different views for the communication procedures. The figure 4.6 gives a hardware view for the procedure put for message passing based protocol. The procedure describes the interaction of the communication primitive with the controller by using internal signals and handshakes. This VHDL procedure, at the behavioral level, uses a FSM model to describe its interaction through the interface signals. Within the put procedure, specific commands (like ENABLE_PUT, REQ_PUT, or NOTREQ_PUT) or single signal assignment operations performs this interaction.

The figure 4.7 shows different software views of the communication procedure put. The two SW views are needed for co-simulation and co-synthesis respectively. The software simulation view (used for simulation) hides the simulation environment. In the present version, we use the Synopsys C-language Interface (CLI) as the target architecture for simulation, then the procedure is expanded into CLI routines [85]. Of course, other co-simulation models can be used. For example, if we use the Inter-processes Communication (IPC) model of UNIX¨, this communication procedure call will be expanded to system routines using the IPC mechanism. The software synthesis view (used for synthesis) hides the compilation environment. The view will depend on target architecture selected. If communication is entirely done by software executing on a given operating system, communication

procedure calls are expanded into system calls, making use of communication mechanisms available within the system. For example, if the communication is to be executed on a standard processor, the call becomes an access to a bus routine written as an assembler code. In the example, the software synthesis view makes use of existing mechanisms available within the IBM/PC system. The communication can also be executed as embedded software on a hardware data path controlled by a micro-coded controller, in which case, this communication procedure call will become a call to a standard micro-code routine.

SW SIMULATION VIEW (CLI)

• HIDES THE SIMULATOR COMMUNICATION DETAILS

```
INT PUT(REQUEST)
 INTEGER REQUEST;
 { SWITCH( NEXT_STATE )
   { CASE INIT : { /* INIT STATE STATEMENTS */ } BREAK;
    CASE DATA_RDY :
    {
       CLIOUTPUT(DATAIN,REQUEST);
       CLIOUTPUT(REQ,BIT_1);
      IF( CLIGETPORTVALUE(RDY) == BIT_1)
        NEXT_STATE = CLOSE; } BREAK;
    CASE CLOSE :{ /* CLOSE STATE STATEMENTS */  } BREAK;
    DEFAULT :  NEXTSTATE := INIT; }
  IF( NEXTSTATE == IDLE ) RETURN 1;
  ELSE { NEXTSTATE == INIT; RETURN 0;  } }
```

SW SYNTHESIS VIEW (IBM/PC)

• HIDES THE ARCHITECTURE COMMUNICATION DETAILS

```
INT PUT(REQUEST)
 INTEGER REQUEST;
 { SWITCH( NEXT_STATE )
   { CASE INIT : { /* INIT STATE STATEMENTS */ } BREAK;
    CASE DATA_RDY :
    {
       OUTPORT(DATAIN,REQUEST);
       OUTPORT(REQ,BIT_1);
      IF(  INPORT(RDY) == BIT_1)
        NEXT_STATE = CLOSE; } BREAK;
    CASE CLOSE :{ /* CLOSE STATE STATEMENTS */  } BREAK;
    DEFAULT :  NEXTSTATE := INIT; }
  IF( NEXTSTATE == IDLE ) RETURN 1;
  ELSE { NEXTSTATE == INIT; RETURN 0;  } }
```

figure 4.7 : Different software views of a communication procedure.

In this case, we need one HW view given in VHDL, one SW simulation view given in C, and a SW synthesis view specific to each target architecture. The difficulty found with the use of multiple views of a communication primitive is the management of the different views to support multiple applications and platforms within the environment. We assume the existence of trusted librarian able to produce coherent views. This general problem exists when libraries are used. The librarian may be an automatic generation tool or a designer.

4.5 An example

This approach has been used for modeling an Adaptive Motor Speed Controller system. The system adjusts the position and speed parameters of a set of motors (to avoid discontinuous operation problems). For example, the control in a 2-D space needs one motor for each axis (X and Y) and an associated control system for a continuous movement. The system can handle up to 18 motors. However, in order to simplify the presentation, we will explain a single motor application.

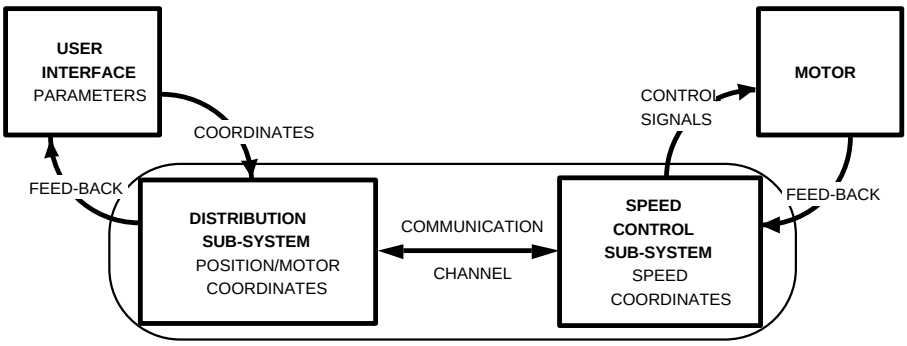


figure 4.8 : Adaptive motor controller system.

As shown in figure 4.8, the controller is composed of two sub-systems, Distribution and Speed Control, communicating through a communication channel. The Distribution sub-system provides the travelling distance to the Speed Control sub-system. With the specified position and the current state of the motor, the Speed Control sub-system computes the needed speed of the motor and sends

modulated pulses as motor control signals.

The system is partitioned into communicating HW/SW sub-systems and associated communication units (figure 4.9). A software abstract model describes the Distribution sub-system and a hardware abstract model describes the Speed Control sub-system. In the early stage of the design, the motor is modeled as a hardware processor.

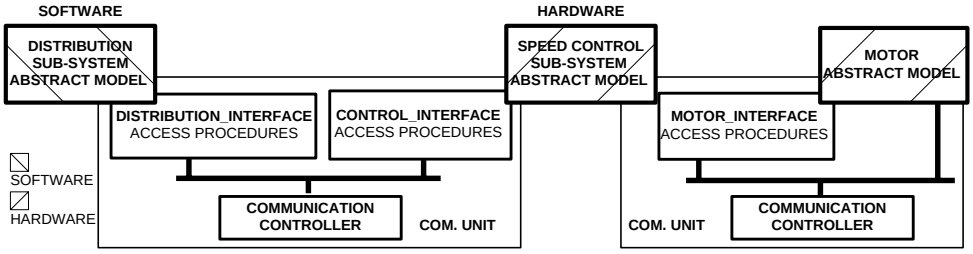


figure 4.9 : The adaptive motor controller: virtual prototype.

The communication between the Speed Control sub-system (hardware) and the Distributor sub-system (software) is described using a communication unit composed of two groups of access procedures (Distribution_Interface and Control_Interface). A communication unit (accessed by a collection of procedures called Motor_Interface) achieves the communication between the Speed Control sub-system and the motor (modeled as a HW process). The use of the above communication units enables the description of the sub-systems independent of the architectural platform that may be chosen.

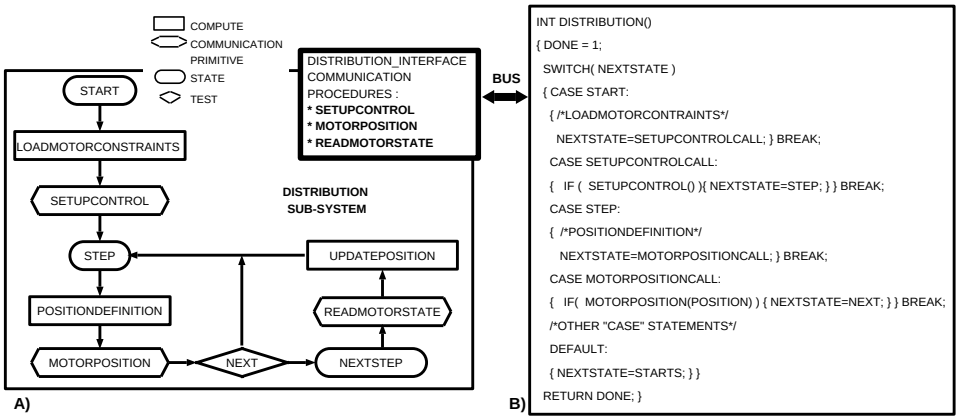


figure 4.10 : Distribution sub-system.

The Distribution sub-system is a software model. The figure 4.10(a) shows its main computation steps and the main communication primitives used by this subsystem. It activates the Speed Control sub-system of the motor by specifying the maximum position value and the maximum number of speed-pulses. The total translation distance of the motor is divided into segments and is sent to the Speed Control sub-system as bundles of data. The initialization data, motor selection and position co-ordinates are transmitted to the Speed Control sub-system by the Distribution_Interface access procedures (SetupControl, MotorPosition, and ReadMotorState) which communicate through the I/O interface. The figure 4.10(b) shows an extract of the C code corresponding to the Distribution Sub-system. In this case, the code is organized as a finite state machine composed of states, transitions, and calls of communication procedures.

A hardware model in VHDL language describes the Speed Control sub-system (figure 4.11). This sub-system uses communication procedures, which are described in VHDL. The sub-system is composed of three parallel processes named: Position, Core and Timer. The Position process actualizes motor position and co-ordinates. It communicates with the Distribution sub-system using the Control_Interface access procedures by sending the actual motor state (via ReturnMotorState

access procedure) and waiting for the new co-ordinates and motor constraint parameters (ReadMotorConstraints and ReadMotorPosition access procedures). The Core process computes the residual position and the next operation conditions. It communicates with the two other units using simple VHDL signals. The Timer process sends a set of motor control pulses to the motor and reads the motor co-ordinates using the Motor_Interface access procedures (SendMotorPulses and ReadSampledData).

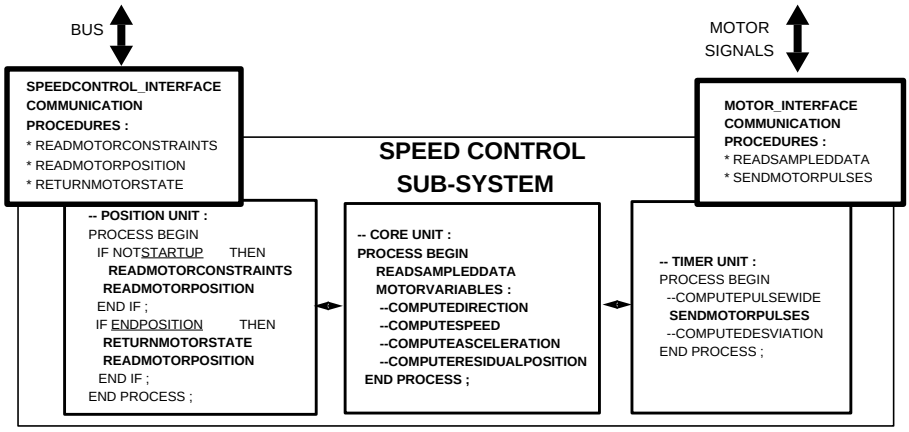


figure 4.11 : Control system (VHDL).

We simulated the system at two different abstraction levels. The entire system was simulated first at the behavioral level to verify functionality. After Hw synthesis, a second cosimulation was performed in order to verify timing constraints. For this purpose, we used a distributed co-simulation environment, composed of a VHDL simulator for hardware modules and C-debuggers for software modules (C-programs) communicating via a software bus. This software bus relies on the Unix IPC layer. The software bus and the elements used to interface VHDL and C modules during simulation where automatically created by a VHDL-C interface generation tool (named VCI) [6][7].

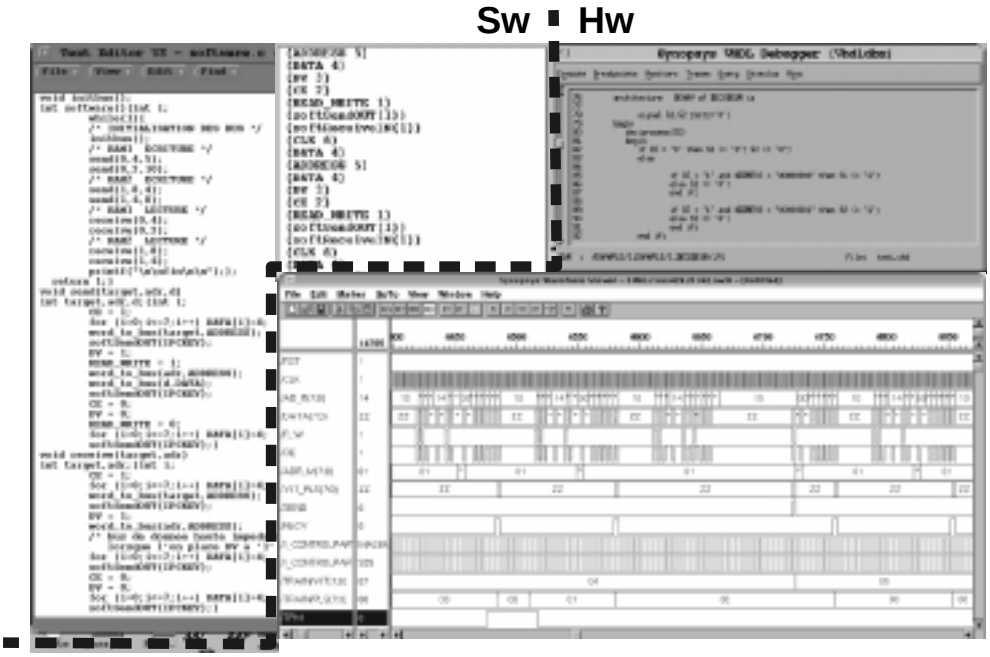


figure 4.12 : Motor controller cosimulation test-bench.

The system was described as a VHDL structure to allow the interconnection of Hw and Sw parts. The Sw subsystem (a C-program executed on the workstation) was interconnected to the Hw part by using the Sw simulation view of its communication procedures. For simulation purpose, these communication procedures were associated to cosimulation primitives making use of the Unix/IPC

mechanism. The Hw part was simulated within the VHDL simulator. The results of cosimulation are the waveforms generated by the VHDL simulator and the C-program execution. During the debug phase, we can execute the C-program and the VHDL simulator in a step by step mode in order to follow the execution. The figure 4.12 is a screen capture showing a simple cosimulation session consisting of the execution of the C-program (left-side) and the VHDL simulation of the Hw part (right-side).

After co-simulation, the co-synthesis was made using an existing platform. In the example we used an architecture composed of a PC-AT communicating with an FPGA based board via the extension bus of the PC. The communication primitives have been selected based on the target architecture. The software primitives correspond to C procedures that make use of specific system calls (I/O routines) requiring some physical addresses. The communication primitives used by the hardware side are written in order to respect the timing and the protocol considerations required by the PC and the motor signals.

As shown in figure 4.13, the Distribution sub-system (a C program) was compiled on a 386-based processor which communicates with an FPGA development board (the Speed Control sub-system) through a 16-bit parallel bus (synchronous communication, 10 MHz). The Speed Control sub-system was synthesized onto a Xilinx 4000-series FPGA associated with memories (EPROM's) and a microcomputer interface. An analysis of the prototype system indicates that this solution confirms the results obtained during cosimulation. The implementation correctly implements the system functionality while meeting the real-time constraints. In order to map this application onto another target architecture, we need to have the corresponding communication primitives. One can note that the target can be a complex multiprocessor architecture.

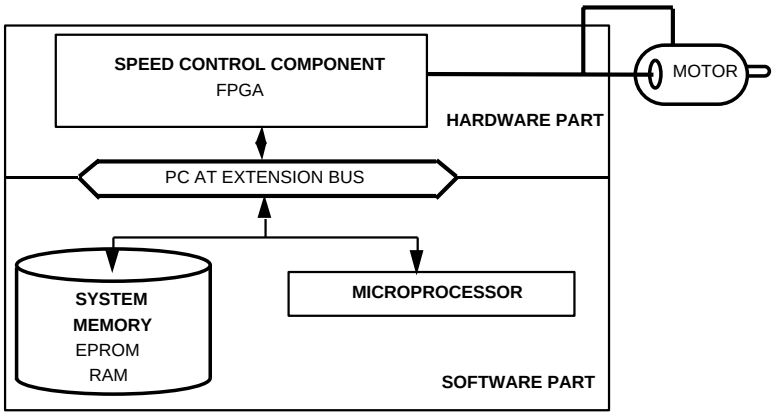


figure 4.13 : The adaptive motor controller system prototype.

4.6 Conclusion

This chapter presented a virtual prototyping strategy for co-design based of mixed C, VHDL specifications. The same descriptions can be used for both cosimulation and cosynthesis. It also allows accommodating several architectural models using a library of communication models enabling the abstraction of existing communication schemes and design re-use. In other words, the same module descriptions are usable with different architectures in terms of their underlying communication protocols.

Chapitre 5

Méthodologie de conception dans Cosmos

Ce chapitre présente Cosmos, une nouvelle méthode pour la conception conjointe de logiciel et de matériel. Cosmos permet la conception de systèmes temps-réel distribués, composés de processeurs programmables pour le logiciel, de modules matériels et de réseaux de communication complexes pour leur interconnexion. De tels systèmes peuvent être mis sur puce, sur une carte ou dans un système géographiquement distribué. C'est une stratégie transformationnelle de conception capable de manipuler des systèmes multiprocesseurs et des architectures distribuées. Cette approche permet le découpage semi-automatique d'une spécification à l'aide d'un ensemble de transformations guidées par l'utilisateur via une interface graphique. Ces transformations sont basées sur des primitives qui effectuent le partitionement fonctionnel, la réorganisation structurelle et la synthèse de communication. Ces transformations permettent d'effectuer une exploration rapide de l'espace de solutions pour réussir à produire un premier prototype dans un temps de conception minimisé. Plusieurs exemples de conception à partir d'un cahier des charges au niveau système (en langage SDL) jusqu'à la génération d'une architecture distribuée logicielle/matérielle (décrite en langage C et VHDL) illustrent cette approche. Nous prouvons ainsi que l'utilisation de cette approche transformationnelle permet :

- *d'employer l'expertise du concepteur pendant le processus de codesign,*
- *de comprendre les résultats du codesign,*
- *de tenir compte des solutions partielles existantes,*
- *d'avoir une grande exploration de l'espace de solutions et*
- *pour un système, de rester à un très haut niveau de spécification.*

Ce chapitre présente le résultat du travail de plusieurs membres de l'équipe de synthèse au niveau système du laboratoire TIMA. Il a été publié dans "Hardware-Software Codesign : Principles and Practice," Kluwer Academic Publishers, éditeurs Jorgen Staustrup et Wayne Wolf, pp. 235-261, London, 1997. Pour cette raison le chapitre est reproduit comme il est.

5.1 Introduction

Codesign is becoming a bottleneck in the process of designing complex electronic systems under short time-to-market and low cost constraints. In this chapter, the word system means a multiprocessor distributed real time system composed of programmable processors executing software and dedicated hardware processors communicating through a complex network. Such a system may be implemented as a single chip, a board, or a geographically distributed system.

In a traditional design methodology, designers make the hardware/software partitioning at an early stage during the development cycle. Different groups design the different parts of the system. The integration of the different parts of the system leads generally to a late detection of errors meaning higher cost and longer delay needed for the integration step. Besides, this early partitioning restrains the ability to investigate a better partitioning trade-off. The different parts of the system are generally oversized in order to reduce last-minute risks.

5.1.1 Requirements for codesign of multiprocessor systems

A new generation of methods is emerging and maturing, these methods are able to handle mixed hardware/software systems at the system-level. They are called codesign tools [62] [63] [24] [64] [65] [66] [34]. Codesign may provide a drastic increase in productivity by making easier concurrent design of the different parts of a distributed system and by the automation of the partitioning and the integration steps. However, codesign issues several challenges:

- The development of modern codesign methods has created an exaggerated hope for having general purpose automatic partitioning tools that would start from a functional specification and produce an optimal solution reducing design time and cost. Indeed, several successful automatic partitioning approaches have been reported in the literature [22][35][67][68][69][70]. However, most of these works restrict the problem to a single application domain or make use of simple estimation methods. These restrictions limit the applicability of these partitioning methods to complex systems including several hardware and software processors. What makes partitioning difficult is the non-availability of a universal estimation method that can be used during partitioning for selecting the right solution.
- The evaluation of a distributed architecture is a complex process depending on a large number of criteria such as: efficiency, reliability, manageability, portability, and usability. Some of these criteria can entail a long list of sub-criteria. For example, efficiency may involve speed, cost, power consumption, volume, or area. For each criterion, a metric value has to be associated. In addition, the weights of these criteria may be different according to the application domain as well as the technology used. For instance, reliability will be the major criterion when dealing with systems concerning human life security. For other systems, such as portable multi-media systems, cost, and power consumption will be the major criteria. It is then clear that it is quite hard to define a realistic evaluation procedure even for a specific application domain. A realistic issue, for dealing with the partitioning, seems being semi-automatic methods allowing mixing manual and automatic design.
- The designer needs to understand the results of automatic partitioning in order to be able to analyze it. This means that codesign tools should provide facilities to show the correspondence between the initial specification and the resulting architecture.
- It is often the case when the designer has a good solution in mind before to start the codesign process. This may be a partial solution like fixing a communication model or fixing the number

of processors of the resulting architecture. This means that codesign tools should take into account this partial solution and allow the designer to control the codesign process.

- The design of a complex system is generally an iterative process into which several solutions need to be explored before finding the "right" one. This means that codesign should allow easy design space exploration.

As long as these mentioned challenges are not solved, codesign will remain restricted to specialists for specific applications.

5.1.2 Previous work

Most current researches in codesign fall in one of three categories:

- **ASIP (Application Specific Integrated Processor) codesign**. In this case, the designer starts with an application, builds a specific programmable processor and translates the application into software code executable by the specific processor [6][71][72]. In this scheme, the hardware/software partitioning includes instruction set design [73]. In this case, the cost function is generally related to area, execution speed, and/or power.
- **Hardware/software synchronous system codesign**. In this case the target architecture of codesign is a software processor acting as a master controller, and a set of hardware accelerators acting as co-processors. Within this scheme two kinds of partitioning have been developed: software oriented partitioning [24][33] and hardware oriented partitioning [74]. Most of the published works in codesign fall in this scheme. They generally use a simple cost function related to area, to processor cost for software and to speed for hardware. Vulcan [74], Codes [75], Tosca [76], and Cosyma [24] are typical codesign tools for synchronous systems.
- **Hardware/Software for distributed systems**. In this case, codesign is the mapping of a set of communicating processes (task graph) onto a set of interconnected processors (processor graphs). This codesign scheme includes behavioral decomposition, processor allocation and communication synthesis [64][77]. Most of the existing partitioning methods restrict the cost function to parameters such as real time constraints [35][78] or cost [79][80]. CoWare [27] handles very well multiprocessor codesign during the latest design phases. However, it starts from a C/VHDL where partitioning is already done. SpecSyn [81] is a precursor for the codesign of multiprocessors. It allows automatic partitioning and design space exploration. However, SpecSyn does not help the designer to understand the produced architecture and it does not allow a codesign with partial solution. Siera [65] provides a powerful scheme for codesign of multiprocessor based on the re-use of components. It allows codesign with partial solutions. However, it does not provide automatic partitioning. Ptolemy provides a powerful environment for codesign of multiprocessor architectures [62]. However, its partitioning is restricted to DSP applications and does not allow partial solutions [22]. Wolf's group studies system-level analysis/performance and cosynthesis of distributed multiprocessor architectures [35]. They have developed an efficient algorithm to bound the computation time of a set of processes on a multiprocessor system, a synthesis method which takes into account communication and computation cost, and scheduling and allocation algorithms for multiprocessors.

In this chapter, we will restrict our discussion to codesign tools of the third category.

5.1.3 Contribution

The main contribution of this chapter is to present a user guided transformational approach for codesign. We present the underlying design methodology and show the efficiency of our refinement-

based approach for hardware/software codesign. The intention of this approach is to solve the challenges mentioned in the section 5.1.1. It covers the design process through a set of user guided transformations that allows semi-automatic partitioning synthesis with predictable results. The lack of realistic estimation method is compensated by the expertise of the designer. A large design space exploration is available through multiple trials and a fast feedback as an implementation of the initial specification can be quickly obtained.

The approach described in this chapter is implemented in the Cosmos hardware/software codesign environment. Cosmos is a methodology and a tool intended to fill the gap between system-level tools and existing synthesis tools. Several aspects of the Cosmos approach have already been presented in the literature. This chapter focuses on the methodological aspects from the designer's point of view and not on the algorithms and techniques used in Cosmos. Most of the details of the intermediate model Solar are given in [83], the behavioral transformation primitives are detailed in [63], the communication synthesis methods are explained in [17] and the code generation (C/VHDL) techniques can be found in [9]. However, for a better clarity of the chapter, we will introduce the models and techniques used when needed.

The first section will present the main principles of Cosmos. Section 5.3 will focus on design models for codesign. The different models obtained after each refinement step will be detailed. Section 5.4 details the main steps of the codesign flow. This models and refinement steps will be clarified by an example, in section 5.5. Finally, we conclude showing the results and the strength of our methodology.

5.2 Cosmos: A global view

Within Cosmos, codesign is decomposed into four major steps (figure 5.1). Functional decomposition is aimed to split large behaviors that need to be executed on several processors. This step may create additional communication. The virtual processor allocation fixes the number of processors and assigns an execution processor to each function. Each abstract processor may be implemented in hardware or in software. Each of the different partitions will contain one or several functions coming from the initial specification. Several functions may be assigned to the same partition allowing them to share functional units or communication resources.

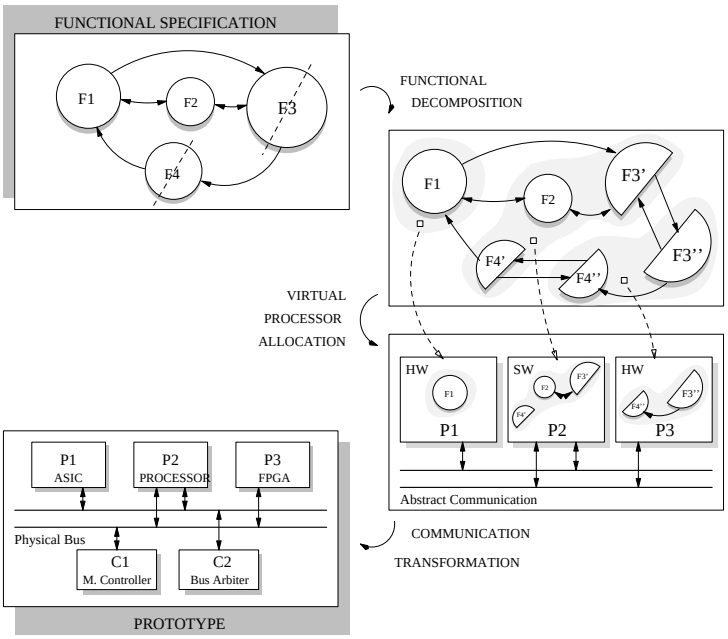


figure 5.1 : Transformational partitioning in Cosmos methodology

After the functional decomposition and virtual processor allocation follows the communication transformation where processors communicating through high-level communication scheme are transformed into processors communicating through buses and sharing communication control. The final step is prototyping. This includes the generation of C-VHDL models of the architecture and the mapping of this model onto the target architecture. In figure 5.1, the processes P1 and P3 will be translated to a behavioral VHDL description for hardware synthesis and the process P2 will be translated to the C language in order to produce an executable code. In order to obtain the prototype, the C code has to be compiled onto software processors and VHDL needs to be synthesized onto Asics or FPGAs.

The user guided transformational methodology assumes that the designer starts with an initial specification and an architectural solution in mind. System design from specification to implementation is performed through a set of primitives allowing the designer to transform the system, following an incremental refinement scheme, in a distributed model that matches to the architectural solution. All the refinement transformations are performed automatically. The designer who uses his knowledge and experience to achieve the desired solution makes the decisions.

Each step reduces the gap between specification and realization by fixing some implementation details (communication protocol, generating software or hardware code) or by preparing future implementation steps (merging and scheduling several processes to execute them in a single processor). The transformations must satisfy the designer's imposed constraints without changing the functionality of the system.

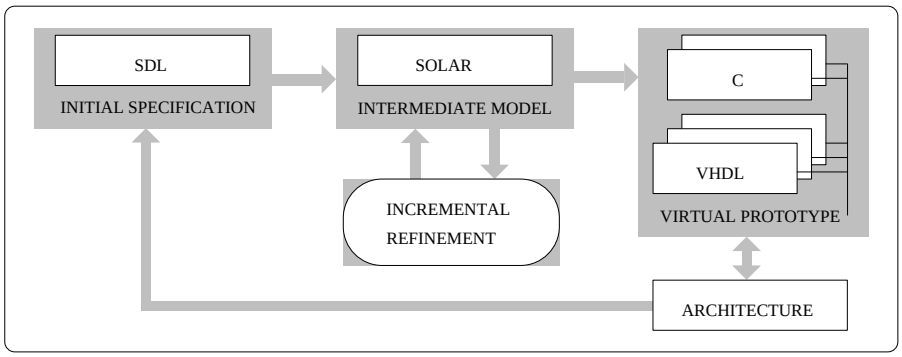


figure 5.2 : Specification models used during codesign

In the case of Cosmos, the user guided transformational approach makes use of three specification formats, as shown in figure 5.2. The initial specification is given in the system-level specification language SDL. The user controls the refinement process through a set of transformation primitives. All the refinement process is based on an intermediate form called Solar. The output is a virtual prototype of the architecture given in a distributed C/VHDL model.

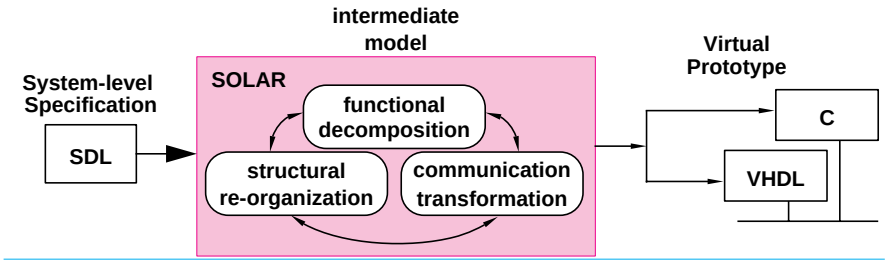


figure 5.3 : Primitives of Cosmos partitioning

Within Cosmos, the partitioning steps are implemented through a set of primitives that performs basic transformations such as split, merge, move, flat and map. The system provides three sets of

primitives working on the system structure, behavior, and communication. The designer guides the interaction process and chooses the transformations needed in order to obtain the desired solution. A new implementation can be obtained by changing the primitives sequence activation. The partitioning flow is shown in figure 5.3.

The organization of partitioning into several small steps reduces the complexity of the problem. The designer controls the partitioning history within an interactive environment, through a fine grain control of the synthesis process. This methodology can be seen as a human guided compilation where the designer spends additional effort to produce an efficient implementation [86].

To facilitate the user interaction for incremental transformations, a graphical interface has been developed. The interface provides a good control of the design process and a graphical view of Solar objects. This will be explained in the following sections.

5.3 Design models used by Cosmos

This section introduces the five design models used within the Cosmos codesign environment: SDL, Solar, the communication model, the mixed C/VHDL, and the target architecture. A special attention will be given to communication models and its transformations.

5.3.1 Target architecture

We use a modular and flexible architectural model. The general model, shown in figure 5.4(a), is composed of three kind of components: software components (aimed to execute C programs), hardware components (implements the VHDL descriptions) and communication components. This model serves as a platform onto which a mixed hardware/software system is mapped. Communication modules come from a library; they correspond to existing communication models that may be as simple as a handshake or as complex as a layered network.

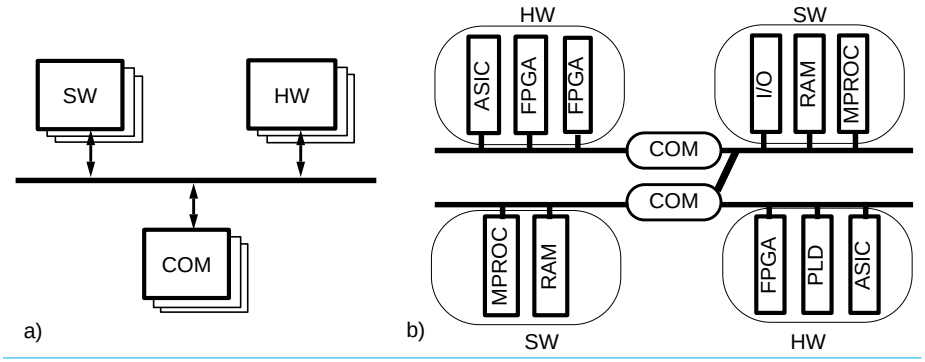


figure 5.4 : Architectural Model: a) architectural model. b) Hardware/software platform.

The proposed architectural model is general enough to represent a large class of hardware/software platforms. It allows different implementation of mixed hardware/software systems, distributed architectures and several communication models. As shown in figure 5.4(b), a typical architecture will be composed of several hardware modules, several software modules and communication modules which are linking hardware and/or software modules.

5.3.2 System specification with SDL

SDL (specification and description language) is intended for the modeling and simulation of real time, distributed and telecommunication systems and is standardized by the ITU [91]. A system described in SDL is regarded as a set of concurrent processes that communicate with each others using signals. SDL support different concepts for describing systems such as structure, behavior, and

communication. SDL is intended for describing large designs at the system level. There are two SDL formats, a textual, and a graphical one.

Structure

A hierarchy of blocks describes the static structure of a system. A block can contain other blocks, resulting in a tree structure or a set of processes to describe the behavior of a terminal block. Processes are connected with each other and to the boundary of the block by signal-routes. Blocks are connected together by channels. Channels and signal-routes are a way of conveying signals that are used by the processes to communicate. Signals exchanged by the processes follow a communication path made up of signal-routes and channels from the sending to the receiving process (figure 5.5). SDL also support dynamic features that are software oriented like dynamic process creation and dynamic addressing.

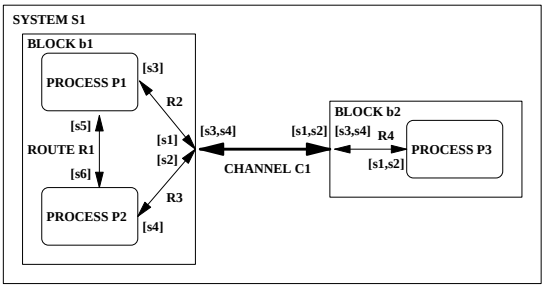


figure 5.5 : SDL signal-routes and channels

Behavior

The behavior of a system is described by a set of autonomous and concurrent processes. A process, described by a finite state machine, communicates asynchronously with other processes by signals. Each process has an input queue where signals are buffered on arrival. Signals are extracted from the input queue by the process in the order in which they arrived. In other words, signals are buffered in a first-in-first-out order. The arrival of an expected signal in the input queue validate a transition and the process can then execute a set of actions such as manipulating variables, procedures call and emission of signals. The received signal determines the transition to be executed. When a signal has initiated a transition, it is removed from the input queue.

(a)

```
PROCESS SpeedControl;
    DCL Vin integer;
    DCL Vout, Vout_1 integer;
    DCL CtrlConst integer;
    START ;
        TASK Vout:=0, Vout_1:=0, Vin:=0;
        OUTPUT NewSpeed1(0);
        NEXTSTATE WaitK;
    STATE WaitK;
        INPUT ControlConst1(CtrlConst);
        NEXTSTATE WaitSpeed;
        ENDSTATE;
    STATE WaitSpeed;
        INPUT MDReady1;
        TASK Vout_1:=Vout;
        TASK Vout:=Vout_1 + CtrlConst*(Vin - (Vout_1/UPSCALE));
        OUTPUT NewSpeed1(Vout/UPSCALE);
        NEXTSTATE WaitSpeed;
        INPUT SpeedCmd1(Vin);
        NEXTSTATE WaitSpeed;
        INPUT ControlConst1(CtrlConst);
        NEXTSTATE WaitSpeed;
        ENDSTATE;
ENDPROCESS SpeedControl;
```

(b)

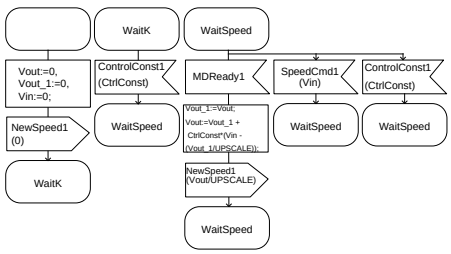


figure 5.6 : SDL process specification: a) textual form b) graphical form

The figure 5.6 represents an SDL process specification. State Start represents the default state. Input represents the guard of a transition. This transition will be triggered when the specified signal is extracted from the input queue. Task represents an action to perform when the transition is executed and Output emit a signal with its possible parameters.

In SDL, variables are owned by a specific process and cannot be modified by others processes. The synchronization between processes is achieved using the exchange of signals only. SDL includes communication through revealed and exported-shared variables. However, they are single-writer-multiple-readers. Shared variables are not recommended in the SDL92 standard [91]. Each process has a unique address (Pid) which identify it. A signal always carries the address of the sending and the receiving processes in addition to possible values. The destination address may be used if the destination process cannot be determined statically and the address of the sending process may be used to reply to a signal.

Communication

Signals are transferred between processes using signal-routes and channels. If the processes are contained in different blocks, signals must traverse channels. There are two major differences between channels and signal-routes. First, channels may perform a routing operation on signals; channel routes a signal to different channels (signal-routes) connected to it at the frontier of a block depending on the receiving process. Communication through signal-routes is timeless while a communication through a channel is non-deterministically delayed. No assumption can be made on the delay and no ordering can be presumed for signals using different delaying paths. Second, signal-routes only deliver signal to the destination process. A signal-route does not have to be connected to more than one channel. The figure 5.5 represents the structure and communication specification of an SDL system.

Channels and signal-routes may be both uni and bi-directional. If many signals are transferred on the same channel or signal-route, their ordering is preserved. When going through a channel or signal route, signals are not allowed to overtake each other. However there is no specific ordering of signals arriving in the input queue of a process that have followed different channels and signal-routes.

5.3.3 Solar: A system-level model for codesign

Solar [29][83] is the intermediate format used within Cosmos. The different refinement steps use Solar as an intermediate representation. Each step makes transformations on a Solar model. The use of an intermediate format for synthesis brings the advantages to make the system independent from description languages. For example, in the case of Cosmos, several input languages can be used as input specifications. In [29] Solar is used for codesign strategy from a multi-language specification combining SDL [44] and StateCharts [49]. Additionally, an intermediate format can be defined in such a way to be more suitable for synthesis algorithms. Finally, in this case, Solar has a graphical representation that makes easier to show the intermediate results to the designer.

Solar supports high-level communication concepts including channels and global variables shared over concurrent processes. It is possible to model most system-level communication schemes such as message passing, shared resources and other more complex protocols.

Solar models system-level constructions in a synthesis-oriented manner. The underlying model combines two powerful system-level concepts: extended finite state machines (EFSM) for behavioral description and, remote procedure call (RPC) for the specification of high-level communication (figure 5.7)[90].

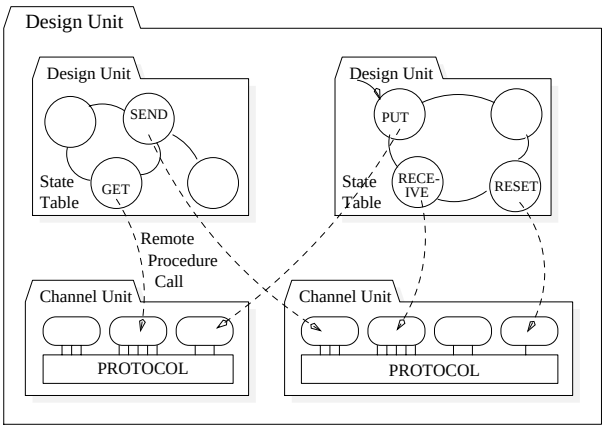


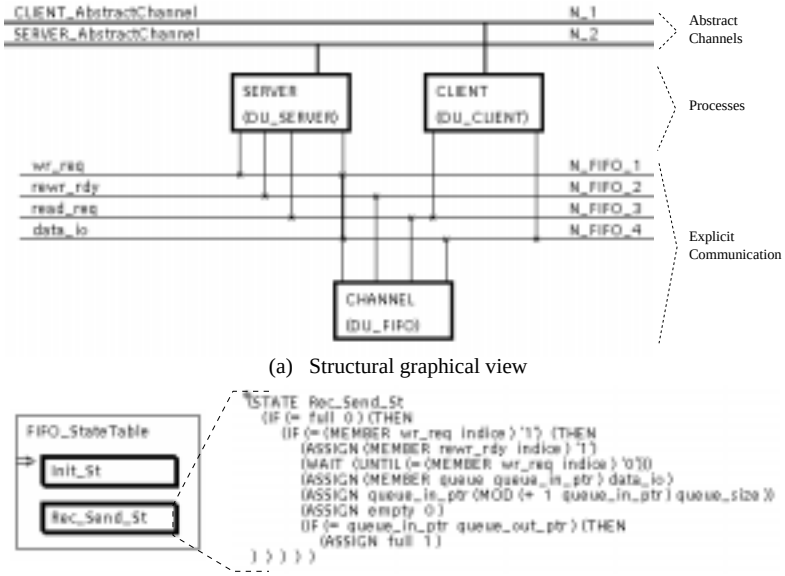
figure 5.7 : EFSMs using RPC communication: the Solar model

The basic concepts are: State Tables, Design Units, and Channel Units:

- The State Table is the basic constructor for behavior description. A state table is an EFSM composed of an unlimited combination of states and other state tables (behavioral hierarchy). All of these states can be executed sequentially, concurrently, or both. A state table has attributes to handle exceptions, global variables, reset and default states. Transitions between states are not level restricted. In other words, transitions may traverse hierarchical boundaries (global transitions).
- The Design Unit construct allows the structuring of a system description into a set of interacting subsystems (processes). These subsystems interact with the environment using a well-defined boundary. A design unit can be specified as a set of communicating design units (structural hierarchy) or as a set of interacting state tables. The communication between design units can be performed in two different ways, first by means of classic port concept where single wires send data in one or two directions or by means of communication channels with a well-defined protocol.
- The Channel Unit performs the communication between any number of design units. The model mixes the principles of monitors and message passing, also known as remote procedure call [90]. The use of a RPC to invoke channel services allows a flexible representation, with a clear semantic. These communication schemes can be described separately from the rest of the system, allowing modular design and specification. A channel unit consists of many individual connections and not only it acts as a transport mechanism for the communicated data, it also provides a handshaking interface to ensure both the synchronization and the avoidance of access conflicts. The channel is composed of a controller, a set of methods, and a set of interconnection signals. The controller stores the resource current state. The access to the channel is governed by a fixed set of methods (services) that are the visible part of the channel. The utilization of an extensible library of protocols allows the reuse of existing components. If a communication unit, to implement the required protocol, services, and average rate, is not found in the library, the designer can adapt an existing communication unit (increase the bus width or buffer size for example) rather than building from scratch.

The figure 5.8 shows a graphical and a textual view of a Solar description. In figure 5.8(a), we can see two processes named DU_SERVER and DU_CLIENT represented by their respectively instances SERVER and CLIENT. These processes communicate with each other by means of signal nets (wr_req, rewr_rdy, read_req and data_io) connected to a channel controller named CHANNEL that implements a FIFO protocol. The CLIENT AbstractChannel and SERVER AbstractChannel correspond to abstract communication channels while CHANNEL represents an implementation of a

communication protocol. The figure 5.8(b) shows a state table (FIFO_StateTable) and two states (Init_St and Rec_Send_St) belonging to the channel controller. A Solar textual view of each graphical element and their hidden structure can also be seen (figure 5.8(c)).



(a) Structural graphical view

(b) Functional graphical view (c) Functional textual view

figure 5.8 : Graphical and textual view of a Solar description

5.3.4 Communication modeling and refinement concepts

In this section, we describe our communication-modeling scheme for system level synthesis. These model aims at representing and implementing the communication scheme used in specification languages. Our model is general enough to accommodate different communication schemes such as message passing or shared memory and allows an efficient implementation of a system level communication specifications.

Communication model

At the system level, a design is represented by a set of processes communicating through abstract channels (figure 5.7). An abstract channel is an entity able to execute a communication scheme invoked through a procedure call mechanism. It offers high-level communication primitives that are used by the processes to communicate. Access to the channel is controlled by this fixed set of communication primitives and relies on the remote procedures call [90] of these primitives. A process that is willing to communicate through a channel makes a remote procedure call to a communication primitive of that channel. Once the remote procedure call is done, the communication is executed independently of the calling process by the channel unit. The communication primitives are transparent to the calling processes. This allows processes to communicate by means of high-level communication schemes while making no assumption on the implementation of the communication.

There is no pre-defined set of communication primitives, they are defined as standard procedures and are attached to the abstract channel. Each application may have a different set of communication primitives (send_int, send_short, send_atm, etc.). The communication primitives are the only visible part of an abstract channel.

The use of remote procedure call allows separating communication specification from the rest of the system. These communication schemes can be described separately. In our approach, the detailed I/O structure and protocols are hidden in a library of communication components. The figure 5.9 shows a conceptual communication over an abstract communication network. The processes

communicate through three abstract channels: c1, c2 and c3. C1 and c2 offers services svc1, svc2 and c3 offers services svc3, svc4 (services svc1 and svc2 offered by abstract channel c2 are not represented).

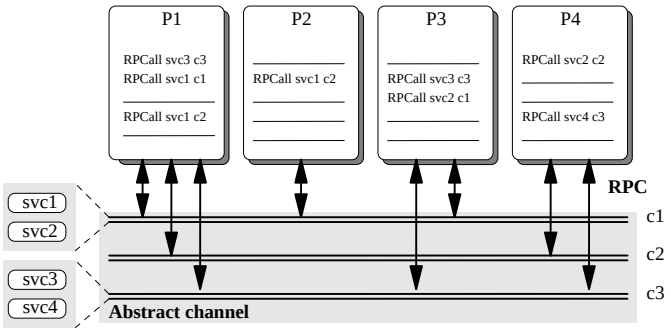


figure 5.9 : Specification of communication with abstract channels

Communication unit modeling

We define a communication unit as an abstraction of a physical component. Communication units are selected from the library and instantiated during the communication synthesis step. Conceptually, the communication unit is an object that can execute one or several communication primitives with a specific protocol. A communication unit is composed of a set of primitives, a controller, and an interface. The complexity of the controller may range from a simple handshake to a complex layered protocol. This modular scheme hides the details of the realization in a library where a communication unit may have different implementations depending on the target architecture (hardware/software). Communication abstraction in this manner enables a modular specification, allowing communication to be treated independently from the rest of the design.

5.3.5 Communication refinement

Communication synthesis aims to transform a system, composed of processes that communicate via high level primitives through abstract channels, into a set of interconnected processors that communicate via signals and shared communication control. Starting from such a specification two steps are needed. The first is aimed to fix the communication network structure and protocols used for data exchange. This step is called protocol selection or communication unit allocation. The second step, called interface synthesis, adapts the interface of the different processes to the selected communication network.

Protocol Selection and Communication Unit Allocation

Allocation of communication units starts with a set of processes communicating through abstract channels (figure 5.10) and a library of communication units (figure 5.11).

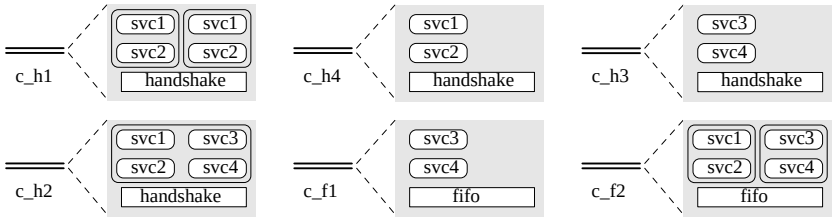


figure 5.10 : Library of communication units

Each communication units is an abstraction of some physical components. This step chooses the appropriate set of communication units from the library in order to provide the services required by the communicating processes. The communication between the processes may be executed by one of the

schemes described in the library. The choice of a given communication unit will not only depend on the communication to be executed but also on the performances required and the implementation technology of the communicating processes. This step fixes the protocol used by each communication primitive by choosing a communication unit with a specific protocol for each abstract channel. It also determines the interconnection topology of the processes by fixing the number of communication units and the abstract channels executed on it.

An example of communication unit allocation for the system of figure 5.9 is given in figure 5.11. Starting with the library of communication units of figure 5.10, the communication unit c_h1 has been allocated for handling the communication offered by the two abstract channels: c1 and c2. The communication unit c_h1 is able to execute two independent communication requiring services svc1 and svc2. Communication unit c_f1 has been allocated for abstract channel c3.

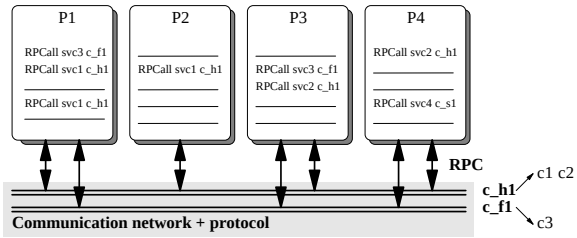


figure 5.11 : System after allocation of communication units

Interface synthesis

Interface synthesis selects an implementation for each of the communication units from the implementation library (figure 5.12) and generates the required interfaces and interconnections for all the processes using the communication units (figure 5.13).

The library may contain several implementations of the same communication unit. Each communication unit is realized by a specific implementation selected from the library concerning data transfer rates, memory buffering capacity, and the data bus width. The interface of the different processes is adapted according to the implementation selected and interconnected. The result of interface synthesis is a set of interconnected processors communicating through signals, buses and possible additional dedicated components selected from the implementation library such as bus arbiter, FIFO, etc. With this approach, it is possible to map communication specification into any protocol, from a simple handshake to a complex protocol.

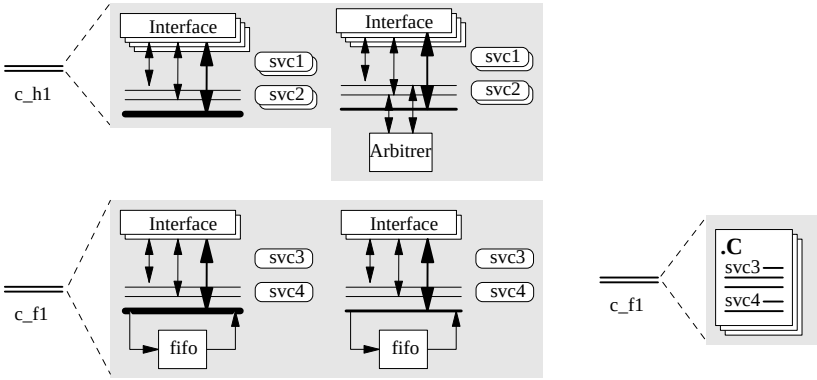


figure 5.12 : Implementation library

Starting from the system of figure 5.11, the result of interface synthesis task is detailed in figure 5.13. The communication unit c_h1 has two possible implementations, one with an external bus arbiter for scheduling the two communications, the other with the arbiter distributed in the interfaces. Any of the two implementations may be selected.

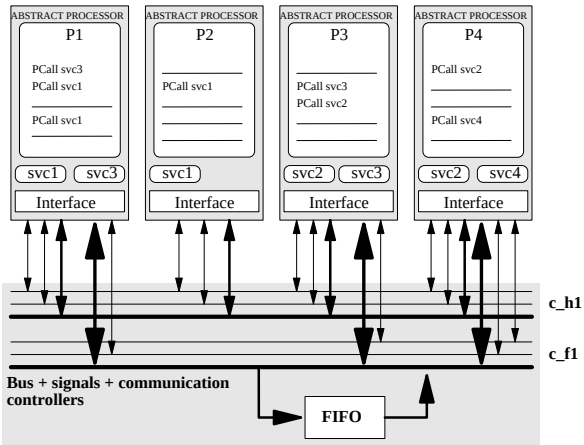


figure 5.13 : System after interface synthesis

5.3.6 Virtual prototyping using C and VHDL models

The architecture to be generated, also called a virtual prototype, will be composed of a set of distributed modules, represented in VHDL for hardware elements and in C for software elements, communicating through communication modules (located into a library of components). The virtual prototype is a simulatable model of the system. This model will be used for the production of the final prototype. The prototyping (also called architecture mapping) produces an architecture that implements or emulates the virtual prototype.

The virtual prototyping step concentrates on the generation and use of the virtual prototype for both cosynthesis (mapping hardware and software modules onto an architectural platform) and cosimulation (that is the joint simulation of hardware and software components) into a unified environment [8].

The joint environment for cosynthesis and cosimulation provides support for multiple platforms aimed at cosimulation and cosynthesis, communication between C and VHDL modules and coherence between the results of cosimulation and cosynthesis. The model used by the environment allows separating module behavior and communication. The interaction between the modules is abstracted using communication primitives that hide implementation details of the communication unit.

5.3.7 C-VHDL communication model

The essential issue in virtual prototyping is the modeling of hardware/software communication. In our case, we use the modular scheme provided by Solar, to produce a modular C-VHDL specification suitable for both, cosimulation and cosynthesis.

The use of a modular description scheme allows for separate evolution of the different modules. During the design process, several representations of the same object may be used at different design steps. During virtual prototyping, the modules are described at the behavioral level and the communication units are modeled as hardware or software processes. Later, after implementation, the communication units correspond to existing units.

In order to allow the use of a communication unit at different design steps we need to describe its communication procedures into different views. Additionally, in order to be able to connect a communication unit to both, hardware and software modules, we have hardware and software views of a communication procedure. The different views of communication procedures may be seen at different libraries required to link the design with different applications.

To support different applications, the number and type of views for each procedure will depend

also on the co-simulation and co-synthesis environments used. The hardware view (given in VHDL) may be common to both co-simulation and co-synthesis. In the case where we use different synthesis systems supporting different abstraction levels (e.g. a behavioral synthesis and an RTL synthesis), we may need different views for the communication procedures. The figure 5.14 gives a hardware view for the procedure put for message passing based protocol. The procedure describes the interaction of the communication primitive with the controller by using internal signals and handshakes. This VHDL procedure, at the behavioral level, uses a FSM model to describe its interaction through the interface signals. Within the put procedure, specific commands (like ENABLE_PUT, REQ_PUT, or NOTREQ_PUT) or single signal assignment operations performs this interaction.

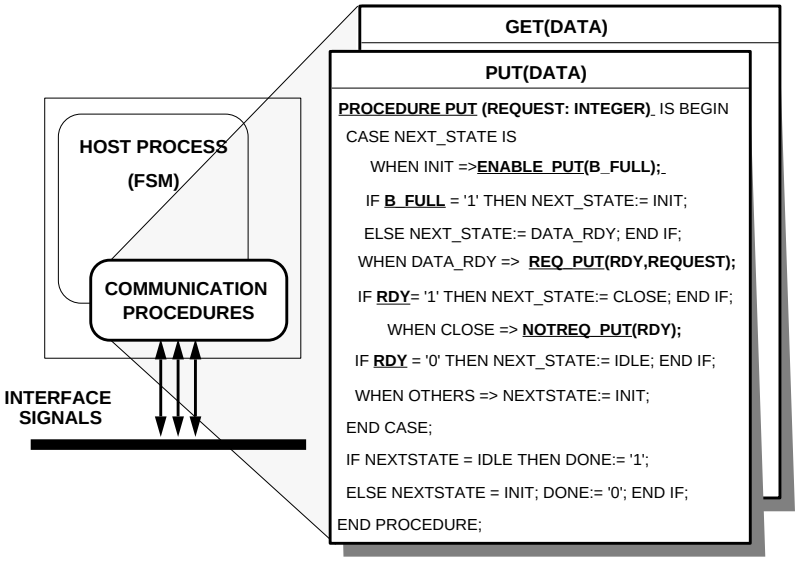


figure 5.14 : Hardware view (VHDL) of a communication procedure.

The figure 5.15 shows two possible software views of the communication procedure put. The two software views are needed for co-simulation and co-synthesis respectively. The software simulation view (used for simulation) hides the simulation environment. In the present version, we use the Synopsys C-language Interface (CLI) as the target architecture for simulation, then the procedure is expanded into CLI routines [85]. Of course, other co- simulation models can be used. The software synthesis view (used for synthesis) hides the compilation environment. The view will depend on target architecture selected. If the communication is entirely a software executing on a given operating system, communication procedure calls are expanded into system calls, making use of communication mechanisms available within the system. If the communication is to be executed on a standard processor, the call becomes an access to a bus routine written as an assembler code.

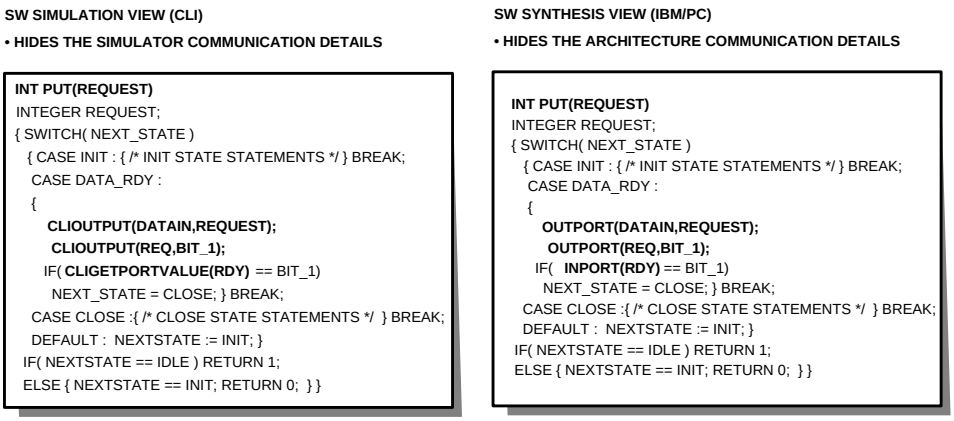


figure 5.15 : Different software views of a communication procedure.

Software views only differ on the I/O primitives to be used. These primitives depend on the target processor or on the cosimulation environment. In other words, abstraction of I/O primitives allows the use of the same communication procedure for synthesis or simulation. For example, if we use the Inter-processes Communication (IPC) model of UNIX, I/O primitives will be expanded to system routines using the IPC mechanism [84]. In this case, I/O primitives make use of existing mechanisms available within the Unix system. The communication can also be performed by an embedded software on a hardware data path controlled by a micro-coded controller, in which case, the I/O primitives will become standard micro-code routines.

We need one hardware view given in VHDL, one software simulation view given in C, and a software synthesis view specific to each target architecture. The difficulty found with the use of multiple views of a communication primitive is the management of the different views to support multiple applications and platforms within the environment. This general problem exists when libraries are used. We assume the existence of trusted librarian able to produce coherent views. The librarian may be an automatic generation tool or a designer.

5.4 Design steps

Cosmos starts with a multi-thread description and produces a multiprocessor architecture. The main steps, after system specification, are: compilation, partitioning, communication synthesis and architecture generation.

5.4.1 SDL Compilation

The design flow begins with an SDL description, but all the synthesis steps use Solar. SDL allows handling concepts as system, block, process, and channel. Translating most SDL concepts (except communication concepts) into Solar is straightforward.

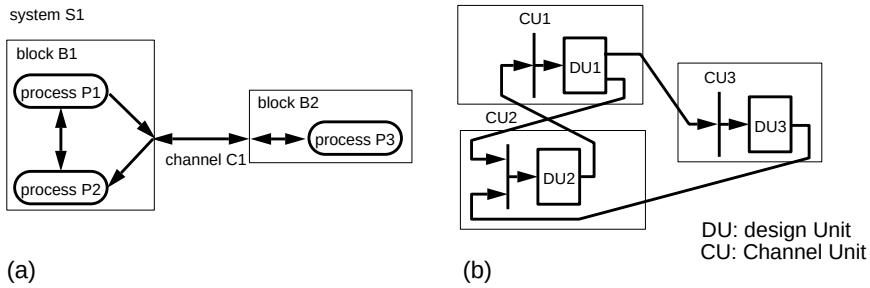


figure 5.16 : Corresponding models: (a) SDL, (b) Solar

However, translating of SDL communication concepts (channel and signal route) causes description reorganization. In SDL, communication is based on message passing. Processes communicate through signal routes and channels group all communication between blocks. Solar communication methodology lets systems containing processes and channels to be modeled. The figure 5.16 summarizes this scheme. The figure 5.16(a) shows a system example with two communicating blocks. Block B1 contains two processes (P1, P2) that communicate with process P3, a process that belong to block B2. Translating this system onto Solar produces the structure shown in figure 5.16(b). Each SDL process is translated into a Solar Design Unit (DU) containing an extended FSM and a Channel Unit (CU). The SDL-to-Solar translation is performed automatically.

5.4.2 Restriction for hardware synthesis

SDL support a general and abstract communication model that is not well suited for hardware synthesis [30]. This is mainly due to the fact that signals can be routed through channels. In other

words, the destination of a signal can be determined dynamically by the address of the receiver. In SDL, the dynamic routing scheme is mainly intended for use with the dynamic process creation feature. This feature is very software oriented and is difficult to map in hardware. Nevertheless we can restrict the SDL communication model for hardware synthesis without losing too much of its generality and abstraction. The restriction imposed on SDL will concern its dynamical aspects such as process creation and message routing.

A wide subset of SDL is supported including:

- System, process, multiple instances.
- State, state*, state*(), save, save*, continuous signals, enabling condition.
- input, output, signals with parameters, task, label, join, nextstate, stop, decision, procedure call, imported and exported variables.

Feature not supported includes dynamic creation of processes, Pid (supported for multiple instances processes), channel substructure, non determinism (input any, input none, decision any).

As stated in section 5.4.1 dynamics aspects of SDL are not considered for hardware synthesis. To avoid any routing problem and obtain an efficient communication, we will restrict ourselves to the case where the destination process of a signal can be statically determined. Communication structure can then be flattened at compile time. A signal emitted by a process through a set of channels must have a single receiver among the processes connected to these channels. In such a case, channels only forward signals from one boundary of a block to another. No routing decision may be taken, as there is only one path for a signal through a set of channels. Therefore channels and signal-routes will not be represented in the final system. A process that is emitting a signal will write it directly in the input queue of the destination process without going through several channels. Flattening the communication eliminates the communication overhead that occurs when traversing several channels.

Each SDL process will be translated into the corresponding finite state machine. During the partitioning step state machines may be split and merged to achieve the desired solution. This step may generate additional communication like shared variables. All the communication protocols and implementation details will be fixed by the communication synthesis step regardless from where it has been generated (initial specification or partitioning).

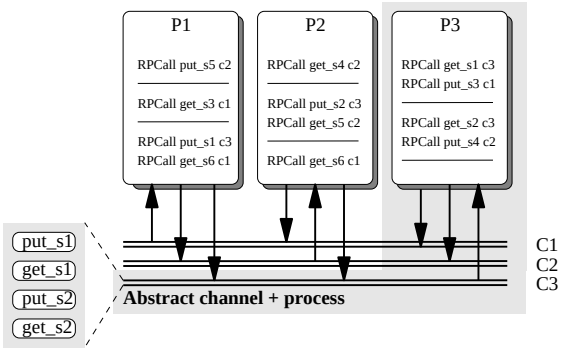


figure 5.17 : Modeling SDL communication with abstract channels

In SDL, each process has a single implicit queue used to store incoming messages. Therefore, we will associate one abstract channel to each process (figure 5.17). This abstract channel will stand for the input queue and will offer the required communication primitives. During the communication synthesis steps, a communication unit able to execute the required communication scheme will be selected from the library. This approach allows the designer to choose from the library a communication unit that provides an efficient implementation of the required communication. Despite the fact that SDL offers only one communication model, several different protocols may be allocated

from the library for different abstract channels. Each signal will be translated as two communication primitives offered by the abstract channel to read and write the signal and its parameters in the channel.

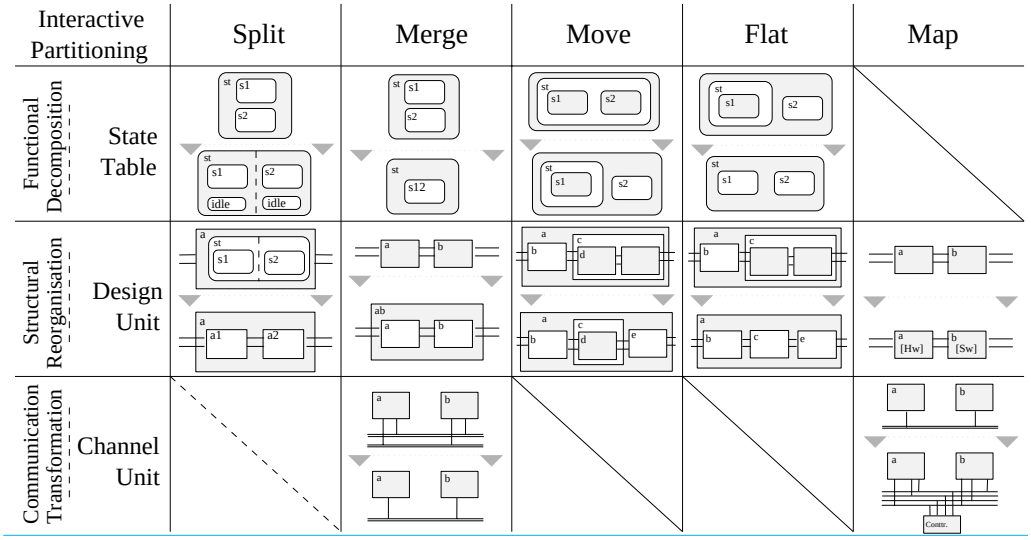


figure 5.18 : Decomposition/composition and communication primitives

The figure 5.18 represents the refined SDL model corresponding to the system of figure 5.5 for synthesis. Each SDL process is mapped to a process containing the behavioral part of the specification and an abstract channel that offers communication primitives to send and receive signals. Each process will read from its own channel and write into other processes' channel. An SDL specification is therefore represented by a set of processes and abstract channels. As stated before channels and signal-routes are not represented.

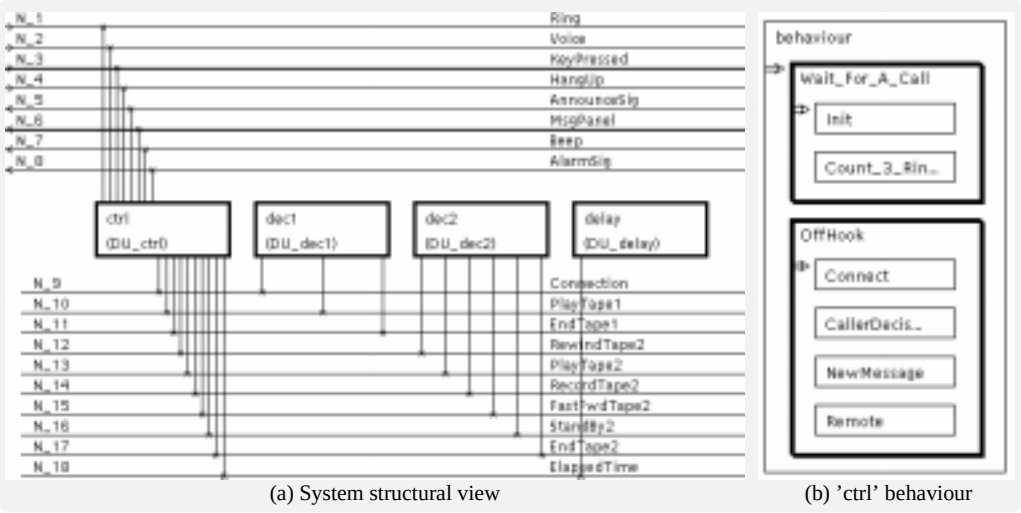
5.4.3 Hardware/software partitioning and communication refinement

As stated before, the partitioning process is composed of three kinds of transformations. The transformations that can be used are functional decomposition, structural reorganization and communication transformation. Functional decomposition acts on the state tables allowing refining behavioral descriptions. Structural reorganization acts on design units allowing refining the structure of the system. Communication transformation acts on channel units allowing refining the communication protocols. All these refinements make use of a set of five primitives called split, merge, move, flat and map allowing to decompose, compose and transform Solar objects. The figure 5.18 summarizes these partitioning primitives. The effect of these primitives is explained in the following paragraphs using a codesign of an Answering Machine [87]. In this example, we will perform a sequence of primitives' calls in order to achieve a desired realization.

The figure 5.19(a) shows the initial system description coded in Solar. The answering machine is composed of four processes: a controller (ctrl), two decks (dec1 and dec2) one to read the announcement and another one to record the messages and, a time counter (delay). The controller is the main process, which manages the resources utilization represented by the other processes. Boxes with external I/O ports (nets 1 to 8) and internal signals (nets 9 to 18) represent the four processes. The ctrl process has a behavior description (called Behavior in figure 5.19(b)), represented by a state table composed of two machines (Wait-For-A-Call and OffHook). Indented boxes represent the machines' hierarchy. The boxes aligned vertically (horizontally) have a sequential (parallel) execution.

The desired implementation is guided to divide the ctrl behavior in two partitions. Functions belonging to the Remote machine will be realized in software and the other functions in hardware. The

software will be converted into C code and executed by a standard processor. The hardware will be translated into behavioral VHDL and realized by an application specific or a programmable circuit. The appropriate set of functions belonging to each partition is a result of the designer's interactions.



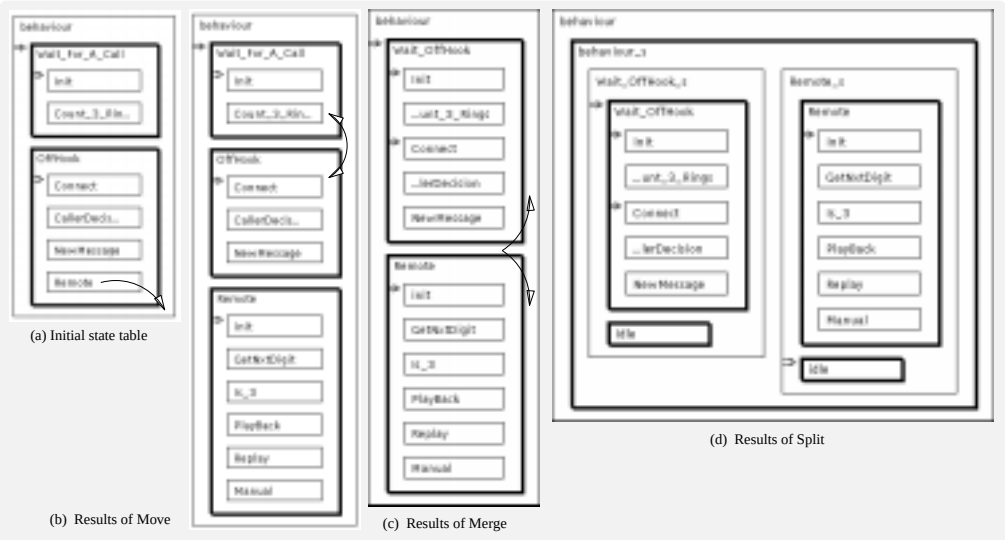
(a) System structural view (b) 'ctrl' behaviour

figure 5.19 : Initial specification of the Answering Machine

Functional decomposition primitives

The functional decomposition primitives are used to transform behaviors i.e. state tables. A detailed description of these primitives with algorithms can be found in [63]. A brief description follows:

- Split decomposes a behavior (state table) into a set of subsystems.
- Merge groups a set of states or state tables into a unique machine.
- Move transforms the hierarchy of a given machine.
- Flat reduces the height of the machine hierarchy.



(a) Initial state table

(b) Results of Move

(c) Results of Merge

(d) Results of Split

figure 5.20 : Functional decomposition primitives

In the case of the Answering machine, the partitioning begins with the functional decomposition primitives in the following sequence (figure 5.20): a move of the Remote machine to a higher hierarchical level, a merge of the Wait-For-A-Call and OffHook machines and, a split of the behavior machine. We use the split primitive to transform the sequential machines into parallel machines in

order to allow the separation in two communicating processes. To simulate a parallel execution, a control signal and an idle state are annexed to each machine.

Structural reorganization primitives

The structural refinement goal is to distribute the behavior into a set of design units that correspond to the final processors. In this case, the application of the primitives will transform the structure hierarchy. Hereby follows a description of the structural reorganization primitives:

- The Split primitive works on the behavior of a process in order to cut it into a set of independent processes. Each process will communicate with the others by means of abstract channels. Data shared between the split processes are converted to abstract channels in the new representation. In each generated process, the access to data shared is replaced by calls to remote services offered by the channel. The figure 5.21 shows the result of splitting a system composed of two concurrent machines that share a common variable. The processes generated can only access the shared data through the abstract channel named X_variable_channel.

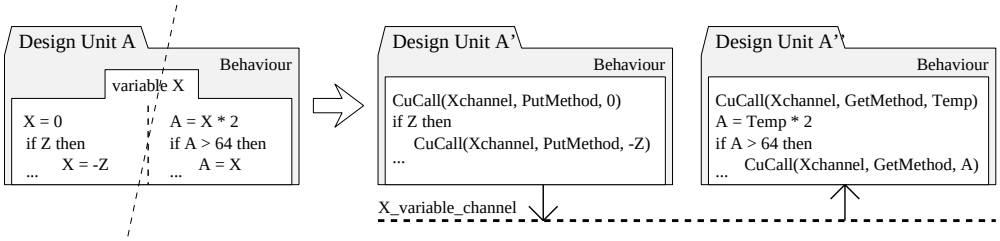
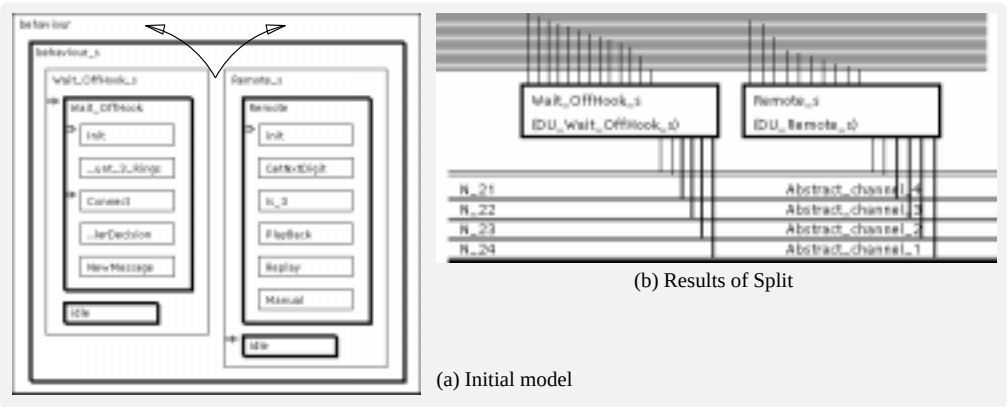


figure 5.21 : Structural Split operation

- The Merge primitive groups a set of processes into a new structural process. When they are all leaf processes, with a behavioral description inside, the algorithm realizes a behavioral merge operation. When we have non-leaf processes, with a process hierarchy inside, the algorithm works on the structure. In the first case, the approach serves to create a behavioral process with a parallel execution of each process behavior involved in the merge operation. Communication channels between these processes are converted to internal global variables. In the second case, a structural process is created with instance calls to the processes involved on the merge operation. A new hierarchical level is created and the interfaces adjusted.
- The Move primitive moves a design unit in the hierarchy. The goal is to put together processes that will be mapped to the same processor in the target architecture. We adjust the signal and channel nets to this new structure.
- Flat performs a structural flattening operation on the hierarchy to remove a hierarchy level.
- Map permits the identification of hardware and software realization options for each process.



(b) Results of Split

(a) Initial model

figure 5.22 : Structural reorganization primitives

The figure 5.22 is a continuation of our example and it shows the structural reorganization primitives. In the figure, we have the machine behavior_s composed of two concurrent machines, Wait-OffHook-s and Remote-s. The use of the structural reorganization primitives begins with the utilization of a split to divide the behavior_s in two different processes. In this case, four extra channels are inserted, each one representing a global variable.

Communication transformation

This step transforms a system composed of processes that communicate via high-level primitives (through abstract channels) into interconnected processes that communicate via signals. This step was introduced in section 5.3.5.

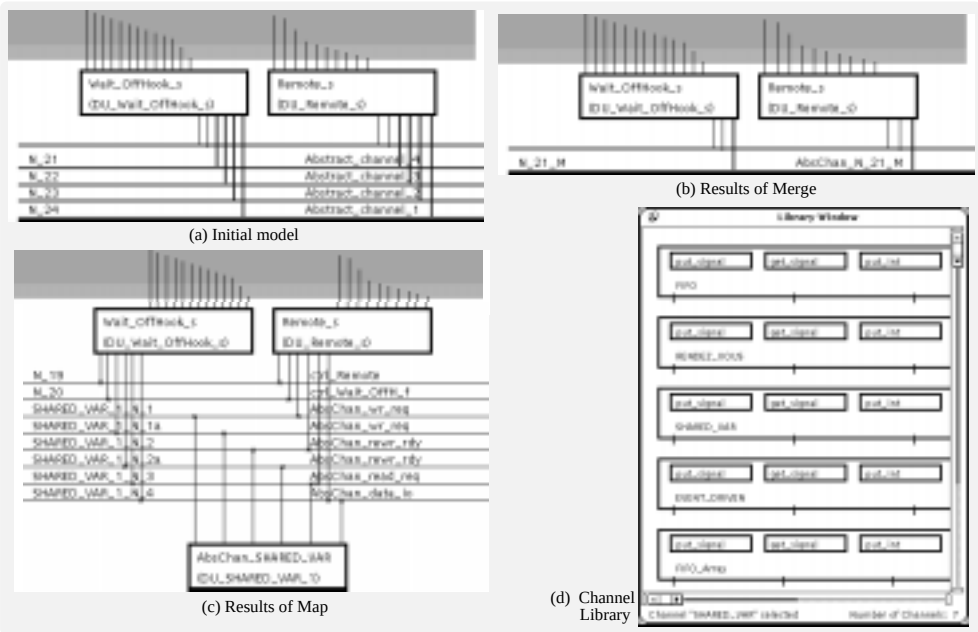


figure 5.23 : Communication transformation primitives

The protocol selection step allows to choose the appropriate communication unit from a library to implement the protocol [17][63]. This operation replaces the abstract channel calls by procedure calls to remote services belonging to the communication unit selected.

The interface building adds to each process the needed code to control the communication according to the selected protocol. A communication controller may also be inserted. At this moment, the channel controller can be seen as a normal process (design unit).

Cosmos makes use of a channel library composed of basic protocol templates. Two primitives are available for the communication transformation:

- Map generates the interfaces (figure 5.23(c)).
- Merge transformation allows replacing two abstract channels by another abstract channel. In several cases, this leads to a cost reduction.

The figure 5.23 shows the use of the primitives merge and map on channels. In the example we start with a system composed of two design units (Wait-OffHook-s and Remote-s) communicating through four abstract channels (figure 5.23(a)). The figure 5.23(b) shows the results of the application of the merge transformation. The four abstract channels are merged into a single new one. The figure 5.23(c) shows the results of the application of the map transformation. In this case the abstract channel is expanded using a specific implementation (shared variables) from the library (figure 5.23(d)).

5.4.4 Architecture generation

Architecture generation corresponds to the translation of Solar into executable code (C and VHDL) to allow the cosimulation and cosynthesis of the system. The generated architecture, also called virtual prototype, is a heterogeneous architecture composed of a set of distributed modules, represented in VHDL for hardware elements and in C for software elements, communicating through communication modules (located into a library of components). The virtual prototype is a simulatable model of the system. The general model, introduced in section 5.3.6, allows to separate the behavior of the modules (hardware and software) and the communication units. Inter-modules interaction is abstracted using communication primitives that hide the implementation details of the communication units. The final step is prototyping. The prototyping or architecture mapping produces an architecture that implements or emulates the initial specification.

5.4.5 VHDL-C cosimulation interface

A distributed co-simulation environment is composed of VHDL simulators for hardware modules and C-debuggers for software modules (C-programs). The link between hardware and software simulation environments and the elements used to interface VHDL and C modules during cosimulation are automatically created by the VHDL-C interface generation tool called VCI [7]. The tool takes as input a user defined configuration file, which specifies the desired configuration characteristics (I/O interface and synchronization between debugging tools) and produces a ready-to-use VHDL/C cosimulation interface.

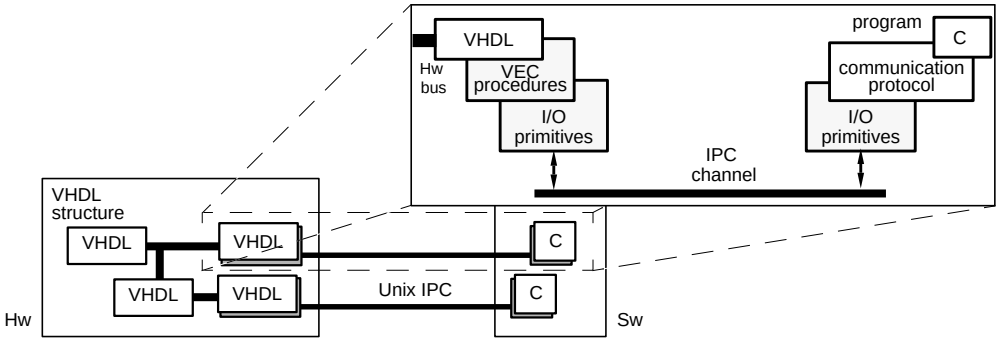


figure 5.24 : Distributed VHDL-C cosimulation model

The system configuration is described in VHDL as a set of interconnected blocks. The figure 5.24 details a system composed of four interconnected modules (VHDL structural view). Two of them are software modules. Each software module is encapsulated in a VHDL entity and its behavior is given by a C-program.

The top box of figure 5.24 shows the detail of the VHDL-C communication structure for one of the software modules. All the dashed parts are generated automatically using VCI. The C-program is connected to the rest of the system through a communication protocol. The protocol allows the C-program to exchange information with the external world. When the program interacts with a set of Hw modules, I/O primitives are used by the communication protocol to perform I/O operations over the ports of the hardware part (Hw bus). On the hardware side, the I/O primitives connect the C-program to the hardware bus through a VHDL entity that encapsulates the ports used.

The C-program and the set of hardware modules are treated as separated Unix processes communicating through the IPC mechanism. In other words, an IPC channel is the link between the I/O primitives used by the C-program and the I/O ports of the VHDL entity. The channel is accessed by the VHDL entity through VHDL emulation C-procedures (VEC-procedures). This access is possible by using the foreign VHDL attribute within an associated VHDL architecture. The foreign

attribute allows parts of the code to be written in languages other than VHDL. In this manner, VEC-procedures (declared by the foreign attribute), instead of VHDL-code, can be executed during simulation. The VEC-procedures perform the tasks of VHDL-C data-type conversion, synchronization and propagation of events between the VHDL simulator and the IPC channel.

5.4.6 C-VHDL model generation

The figure 5.25 shows the overall C-VHDL generation flow.

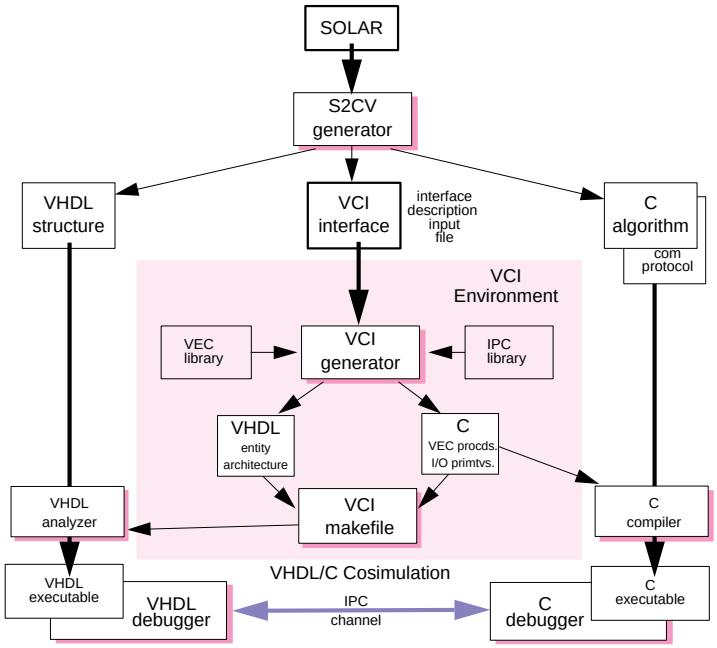


figure 5.25 : VHDL-C interface generation flow

Starting from a refined Solar model, a first tool, called s2cv produces C and VHDL models, for software and hardware processes respectively. S2cv produces also an interface file that specifies the C-VHDL interface of software modules and cosimulation directives. The C-VHDL interface is generated to be used during the cosimulation step. The cosimulation directives are used to generate a cosimulation test bench.

The first step makes use of several transformations of Solar modules to produce hardware and software descriptions. Hardware modules are described in behavioral VHDL, which can be used for synthesis or cosimulation. Software modules are described in C language, which can be targeted to different platforms. The Solar structural view is also represented in VHDL as a set of components interconnected by signals. The rest of this section will detail the VHDL-C interface generation tool (VCI).

In the second step, VCI creates automatically the elements used to interface VHDL and C modules during simulation. According to the input file (VCI interface description), The tool generates the VHDL and C files needed to interconnect the C-program to the VHDL structure during cosimulation. The tool can alternatively be used to generate a VHDL interface for another simulator.

The input of VCI is a file that specifies the C-program interface and additional information for cosimulation. The resulting output consists of the VHDL/channel interface on the hardware side, the C/channel interface on the software side, and a make-file. The VHDL/channel interface is composed of three parts: the VHDL entity (that encapsulates the C- program interface), a VHDL architecture

(contains the declaration of the VEC-procedures introduced by the foreign VHDL attribute) and the VEC/channel interface (C-procedures executed by the VHDL simulator that communicates with the software part through the IPC channel). The C/channel interface is composed of the I/O primitives (used by the communication protocol of the software module to exchange data through the channel). These primitives and the communication protocol are compiled and linked to the C-program creating the executable code. Finally, the make-file execution generates automatically the hardware side that will be part of the VHDL structure to be simulated and starts the distributed simulation environment. The environment will be composed of a set of simulators (VHDL simulator(s) and C debugger(s)) interconnected by cosimulation channels.

5.4.7 Prototyping

The starting point of prototyping is the virtual prototype, composed of a set of interconnected hardware modules (in VHDL), software modules (in C), and communication modules. The result of prototyping is a functional architecture integrating all the modules into processors, FPGAs, and already existing circuits (figure 5.26) [88].

The prototyping consists of the realization of an application prototype. The prototype can be a first version of the final machine or a temporary realization of the application. This step consists of integrating the software parts and the hardware parts on a physical support. The result can be a circuit, a circuit board, or a complex system. In the first case, the Sw, hardware and the communication modules are integrated within a single chip. In other cases, one can use the processors and existing architectures that can be adapted to validate the application. This step is necessary to understand the real performance of the solution. It consists of assembling the different parts of the system.

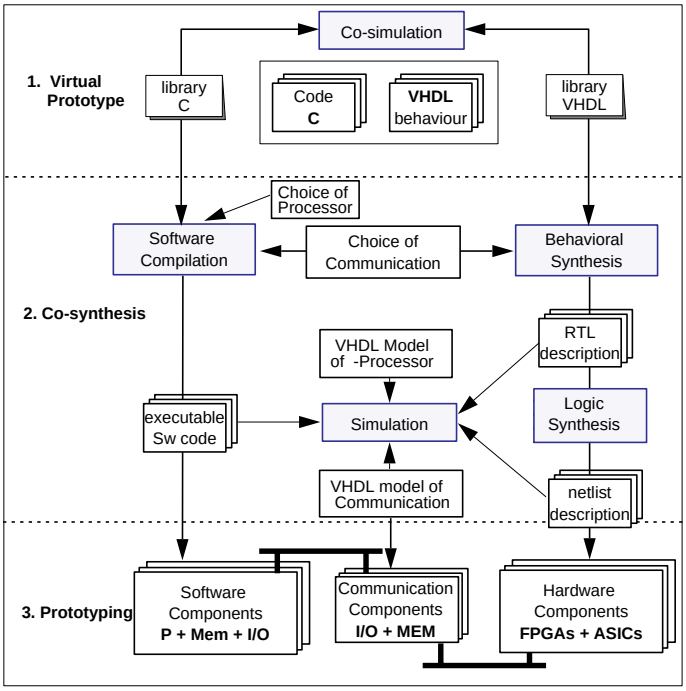


figure 5.26 : Prototyping design flow

The C-VHDL model used in Cosmos allows combining co-simulation and co-synthesis within a single environment supporting a large variety of architectures and platforms. VHDL and C descriptions can be simulated together to validate the specification at different steps of the prototyping process. The simulated descriptions are also used for cosynthesis. The cosynthesis will produce code and/or hardware that will execute on a real architecture. In general, this step consists on mapping the system onto a Hw-Sw platform that includes a processor to execute the software and a set of Asics to

realize the Hw. As shown on figure 5.26, the resulting prototype is generated by using the executable code generators for the software parts to be realized, the logic synthesis, placement and routing tools for the hardware parts to be realized. The compilers translate the C-programs into an assembler code to be executed on the processors, whereas, the synthesis tools transform the VHDL descriptions down to Asics or FPGAs.

Hardware design

The design of complex hardware modules necessitates several iterations in the design flow. These iterations permit the designer in refining the selected hardware components as well as their codes of descriptions. The starting point of the hardware design is a behavioral description of the module in the IEEE standard VHDL language. Finally, the result is a circuit net-list. The design flow consists of two synthesis steps: An architectural synthesis and a logic synthesis. The architectural synthesis permits us to analyze trade-off between different architectures algorithmically and to pass from a behavioral level to Register Transfer Level (RTL). The RTL description obtained describes the functional architecture of the circuit. The logic synthesis compiles the VHDL RTL description to optimize the design for area, performance, and power and generates a net-list of the circuit. At each level of abstraction (behavior, RTL, net-list), one conducts a co-simulation of the hardware module with that of software. This permits us to validate the functionality of the circuit at each step of the synthesis process (namely, architectural synthesis and logic synthesis).

Software design

A software module is described in ANSI C standard language. This description is used at the beginning for co-simulation. The same description, including communication procedures and I/O functions, is used for software synthesis (compilation for a microprocessor based architecture). Only the I/O functions for co-simulation and co-synthesis are different. They are provided by a librarie for co-simulation or co-synthesis. In order to establish the necessary library for the compilation of the software code, one needs to know the type of the architecture upon which the software will execute and in particularly the type of the microprocessor.

The software module consists of a processor executing a machine code. Principally the type of the processor decides the performance as well as the cost of this module. The C-program is generated independently of the processor. This allows postponing the processor selection decision to a late stage in the design process. This choice is indispensable since for the compilation of the C-code, we needs to describe the procedures used by the software conforming to the microprocessor on which they will be executed. The processor can be a standard microprocessor or an Application Specific Instruction Processor (ASIP).

The performance requirements for the execution of the software code will determine the type of processor. For a signal processing application, where the software code will consist of several functions of complex calculations (example: sum of products, operations on matrices, etc.), a DSP processor is more suitable. On the other hand, for simple calculations on data types, a standard microprocessor would be sufficient. The software module described and co-simulated in C language is compiled with the procedures of calculation and communication in order to obtain a code that can be executed by the microprocessor of the software module. This compilation is realized either directly on a software module or by a compiler specific to the microprocessor.

An interesting method for simulating this code consists of using a VHDL processor model that can execute the code. This description executes all the machine code supported by this processor and gives the same result as can be obtained from a real architecture. At this level, one can simulate the

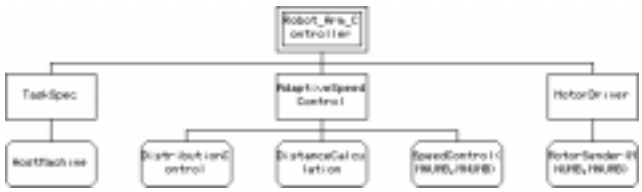
hardware modules described at the RTL or net-list level with the software modules composed of the compiled code and the processor model upon which they will be executed.

5.5 Application

This section details the application of the transformational approach to a codesign example. The example is a large application, a Robot Arm Controller. In the example we will illustrate the overall design flow of Cosmos, from a system-level specification given in SDL to a distributed hardware/software architecture described in C/VHDL. The use of SDL allows for fivefold reduction of the size of system specification when compared to the C/VHDL models. All the transformations applied in this section are fast enough to look instantaneous during an interactive session. None of these primitives require more than 5 seconds CPU time on a Sparc-20 workstation. The Robot Arm Controller implementation regarding to the design space exploration and high-level synthesis can be found in [89] and [88] respectively.

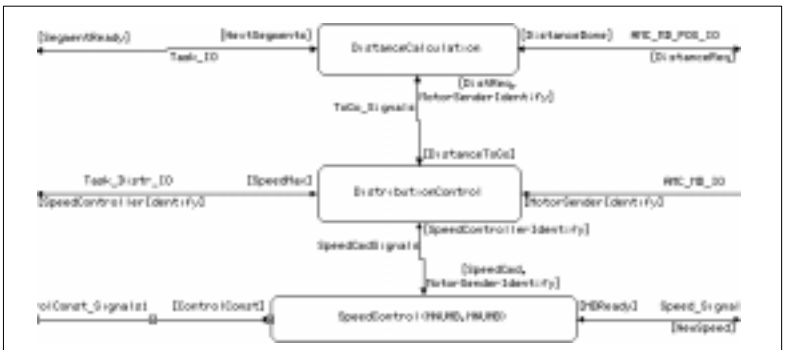
5.5.1 Robot arm controller codesign example

The Robot Arm Controller is a codesign system that adjusts the position and speed parameters of eighteen motors belonging to a robot arm. This computation is intended to avoid discontinuous motor operation problems. The change in motor speed should follow a smooth curve for acceleration and deceleration for mechanical reasons. The basic block diagram of this system, in SDL, is shown in the figure 5.27(a). The constraints for this design are: the time needed to complete the total movement, which must be minimal, the change in velocity must be as smooth as possible and the overshoot at the end position that must be minimal.



(a) System structure

(b) 'Robot_Arm_Controller' process structure



(c) 'AdaptativeSpeedControl' process structure

figure 5.27 : SDL input graphical representation

The HostMachine process is the interface between the user and the system; it takes care of storage

segments and motors constants calculation. In a brief functional description, the AdaptativeSpeedControl block receives positions from the HostMachine block and calculates the speed that each motor needs to arrive at the same time at the desired position. The Distance calculation stores up to date information about the distance to go for every motor according to the current position. The Distribution controller identifies the motor that must run at maximal speed and sends scaled commands to the speed controller. The SpeedControl block is responsible to compute the number of speed control pulses with a specified final position and a current state of a motor. This block contains a digital filter based on a Fuzzy Logic algorithm to generate the smooth acceleration curve. The block converts the input steps into a curve that respects the worst-case maximum load acceleration and deceleration curves of the motors. The motor with the longest distance to go runs at the maximal speed and the speed of the other motors are adapted to this one. The Motor Driver is the interface to the stepper motor, which converts speed into frequency pulses. The MotorSender block converts the pulses into control signals to each motor.

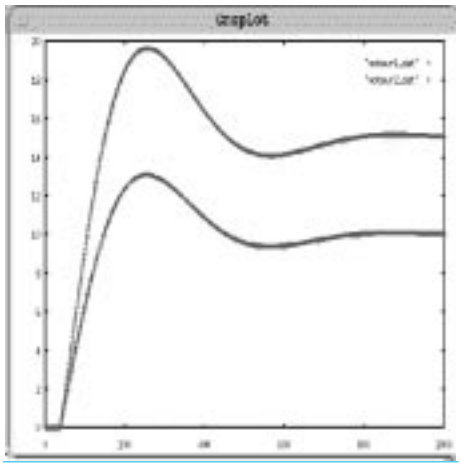


figure 5.28 : Two motors system SDL simulation results

The SDL model was simulated in order to proof the function and to allow the codesign process. The figure 5.28 shows the results of the SDL simulation of 2 motors system.

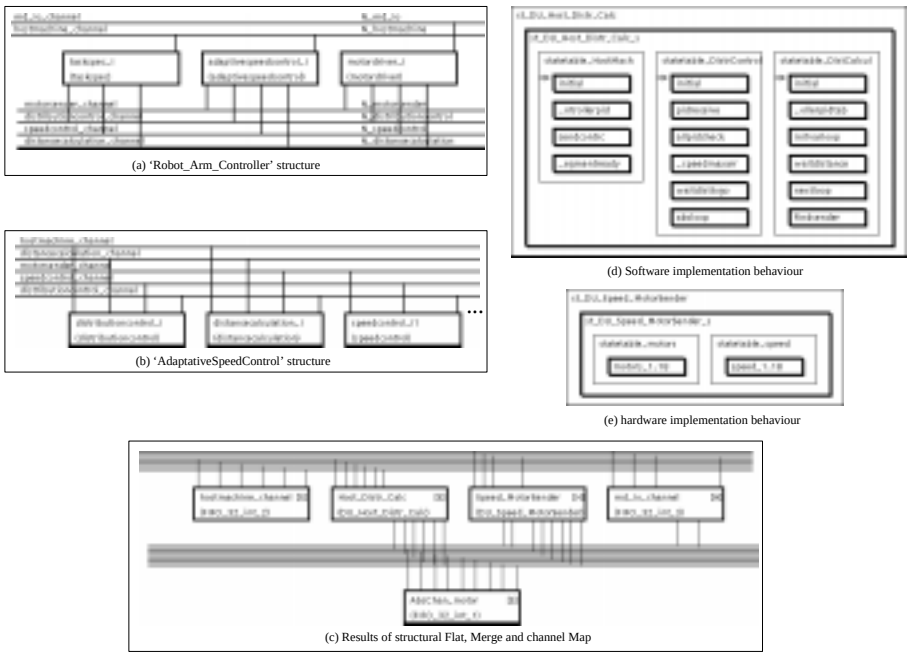
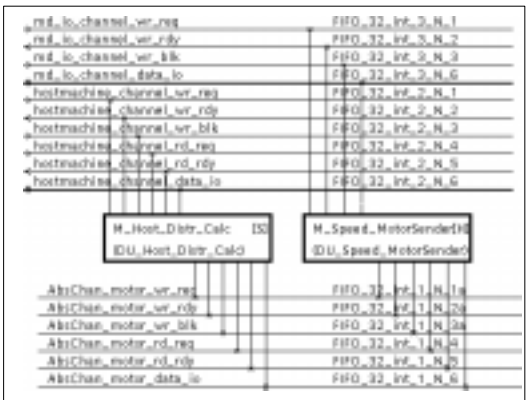


figure 5.29 : Refinement steps

The figure 5.29 shows the use of refinement primitives on the Solar representation. During

partitioning the SpeedControl and the MotorSender, blocks were assigned to compose the hardware part and the other blocks the software part. In this architecture implementation, the processes with extensive computation were assigned to have a hardware implementation to respect the system time constraints. The figure 5.29(a) shows the top hierarchy translated from SDL to Solar. The three top modules are connected to 2 external communication units and to four internal ones. The figure 5.29(b) shows the structure of the adaptive speed control block. The figure 5.29(c) shows the result of a set of transformations, including flat and merge operations on the initial structure and, merge and map on the channels. The abstract channels were mapped on protocols from the library. In figure 5.29(d) and figure 5.29(e) we can see a graphical representation of the finite state machines resulting after the application of the structural merge operations. The initial design requires duplicating the SpeedControl and MotorSender blocks eighteen times. An instance for each motor was necessary to arrive to the desired operation throughput. However, only one instance is required for the other blocks. In figure 5.30 we can see the final system structure and parts of the generated C/VHDL codes.



(a) 'Robot_Arm_Controller' final structure

(b) Partial C-code of 'distancecalculation' process

(b) Partial VHDL-code of 'speedcontrol' process

figure 5.30 : Virtual prototype

In order to verify the cosynthesis process, we have built a prototype platform for two motors. The structure of this platform is shown in figure 5.31. The prototype is composed of the speed controller (hardware) communicating to the host machine (software) and two motors. Each motor is described by a C-program in order to model its behavior using an algorithmic model. Software modules are interconnected to buses through different kind of ports: input-output ports of type real and integer for data signals, and bit vector ports for control signals. The SDL communication was translated into four channel units (CU1 to CU4). CU1 transfers responses from the motors, CU4 and CU2 transfer commands from the speed control unit to the motors. CU4 uses a double port memory for bi-directional communication between the host and the speed controller.

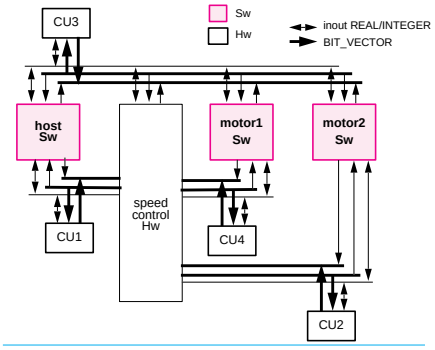


figure 5.31 : Adaptive motor controller system

The VCI generation tool was used to generate the VHDL-C interface for each of the software modules starting from the interface description shown in figure 5.32. According to the input file, VCI generates the VHDL entity used to interconnect the module to the hardware bus (figure 5.33), the VEC procedures and I/O primitives.

motor1itf **VCI**

```
!
!PORT                    DIR   TYPE{[RANGE]}SENSITIVITY
!
CLK                      IN    BIT        R
RST                      IN    BIT        N
algo1_channel_wr_req     OUT   BIT        N
algo1_channel_rewr_rdy   IN    BIT        N
algo1_channel_read_req   OUT   BIT        N
algo1_channel_data_int   INOUT INTEGER    N
algo1_channel_data_real  INOUT REAL       N
control_channel_wr_req   OUT   BIT        N
control_channel_rewr_rdy IN    BIT        N
control_channel_read_req OUT   BIT        N
control_channel_data_int INOUT INTEGER    N
control_channel_data_real INOUT REAL      N
```

figure 5.32 : VCI input file for the motor application

Each C-program, corresponding to the behavioral description of one of the motors, communicates by using primitives (e.g. get_signal, get_int, get_real, put_int, etc.). Each communication primitive (C-procedure) implements a specific communication scheme by calling the I/O primitives (motor1itfSendOut, and motor1itfReceiveIn) generated for cosimulation. This scheme was described by the figure 5.24.

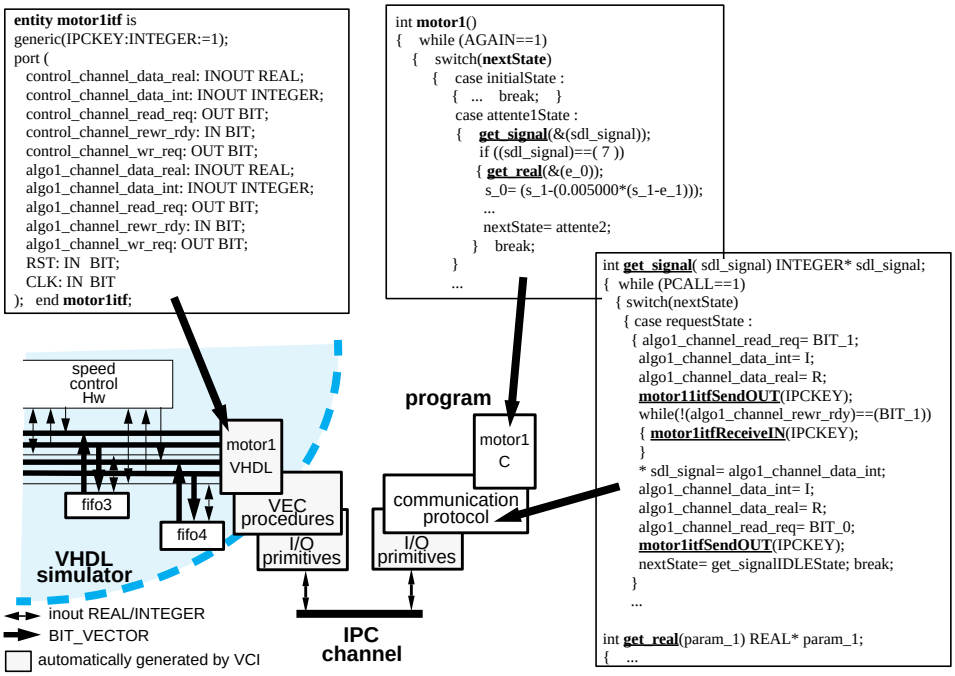


figure 5.33 : VHDL-C motor1 cosimulation models

5.5.2 C-VHDL cosimulation

The cosimulation of the example requires a VHDL simulator and three C-programs (two C modules for modeling the motors and the third to be executed on the host machine). Each software modules was debugged in its corresponding debugging tool and the speed of each motor was traced by using the gnuplot tool. The figure 5.34 shows a cosimulation screen showing VHDL waveforms corresponding to the simulator of the controller and the speed curve corresponding to the two motors. In this case, the cosimulation allowed tuning the control algorithms in order to fit with the requirements of the application.

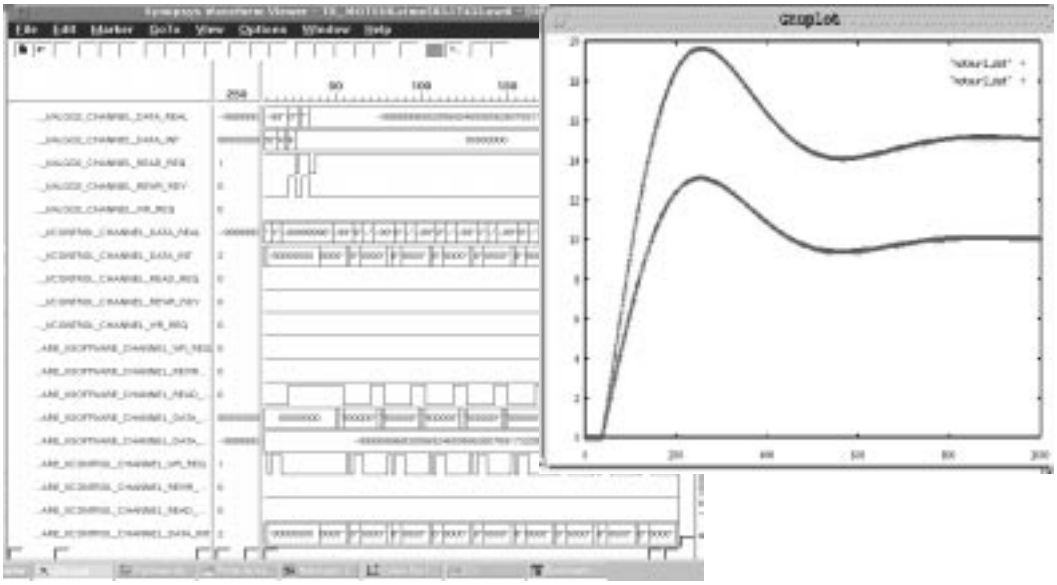


figure 5.34 : Motor controller VHDL cosimulation results

For this application, the use of cosimulation avoided to build a prototype initially planned in the project. The cosimulation allowed to tune the parameters of controller algorithm and to validate the specification of both, the software running on the host computer and of the hardware model of the controller. The controller was simulated at three abstraction levels [88]. The initial model was given in behavioral VHDL. The application of a behavioral synthesis tool produced a VHDL-RTL model that was also cosimulated in the same environment. The application of RTL synthesis produced a gate model that was also simulated. The same hardware/software interface generated by VCI was used for the three cosimulations.

5.5.3 Architecture generation

We implemented the modules using standard design tools, such as C compilers for software modules, and the architectural synthesis tool AMICAL (developed at the TIMA laboratory) and the Synopsys logic synthesis tool [22] for hardware.

The speed calculation module is specified in behavioral VHDL. The high-level synthesis process is performed in three steps: scheduling, functional units' allocation and architecture generation. The figure 5.35 represents the global architecture of the speed calculation hardware module, automatically generated. The initial description (in behavioral VHDL) contains about 150 lines of VHDL code. The generated architecture is described into 3740 lines of RT-level VHDL code. The architectural synthesis generates an architecture composed of a controller composed of 39 states and a FSM with 108 transitions. The controller commands the data path using 49 control lines. This controller sequences the operations executed by the functional units. It is generated automatically during the synthesis process. The data-path is constituted by 5 functional units (ALU, 8x8 multiplier, 16/6

divider, I/O RAM and I/O motor), 9 registers of 8 bits and MUXs (3 Mux (2), 5 Mux (30) and 8 Mux (4), where Mux (i) means a multiplier with i entries).

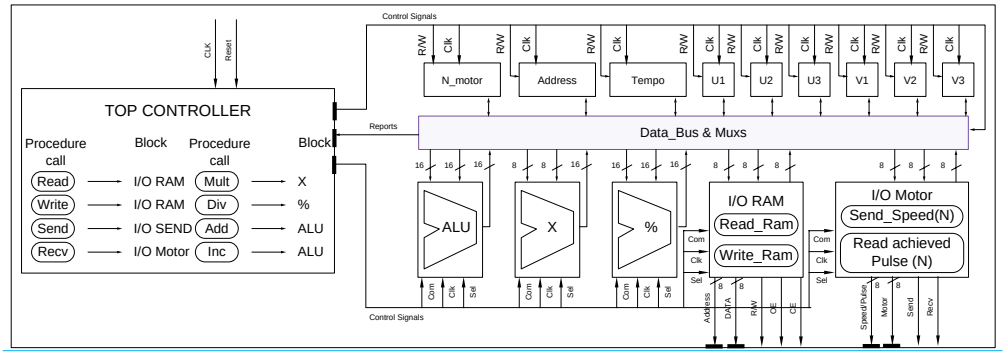


figure 5.35 : Generated hardware architecture

The speed calculation unit is connected with the software distribution module and the motor interface module. The hardware/software communication is assured by a shared memory to store the I/O parameters. Thus, the speed control encloses two additional functional units (figure 5.35): IO_RAM (between the speed control and the dual port RAM memory) and IO_Motor (between the speed control and the motors interface). Each unit executed a specific protocol synchronized by an external clock signal. The IO_RAM unit executes two operations: Read_Ram (read memory operation) and Write_RAM (writes the memory). The unit executes two operations: Send_Speed (send the speed parameters) and Recv_Pulse (receiving the consumed pulses). The motor interface module, allows to transform the speed value (8 bits value) issued by the speed controller into a width-modulated pulse to run the motor.

After performing architectural (RT-level module generation) and logic synthesis (producing a gate-level description), the hardware module was mapped onto a FPGA (Field-Programmable Gate Array). This step maps the net-list of gates onto a FPGA chip. The obtained module, containing 10,000 gates, was transposed on a Xilinx FPGA.

The software platform considered was a personal computer (PC). The choice of this platform was motivated by its flexibility. In effect, the architecture of the system permits several types of extensions over the system bus. The software module and its communication procedures were compiled using a standard C compiler. A 4kx8 bits dual port memory was used for the communication between the PC and the speed controller. An extension card connects a bus, supporting the two hardware modules as well as the dual port memory, assuring the hardware/software communication.

5.6 Evaluation

The design methodology and prototyping within the Cosmos project permits to have an executable model much earlier within the design cycle. A system level specification serves as a basis for deriving an implementation. The use of system-level specification languages offers concepts and methods adapted to the description of complex systems, providing formal methods for verifying a validating the system behavior. The codesign process is an interactive process where several solutions can be explored at an early stage of the development cycle. In this context, the Cosmos project is fast enough to allow the exploration of several solutions in short time intervals.

The figure 5.36 shows the codesign flow and gives the synthesis time for different abstraction levels associated to the motor controller development example. While we can perform a fast simulation and verification at the system level (SDL), because the size of system-level descriptions is usually 5-10 times smaller than equivalent lower-level descriptions, the gap between the specification and the realization includes many design decisions and implementation details that need to be

performed in short time intervals. The Cosmos methodology allows a fast transformation of the system specification given in SDL to a distributed hardware/software architecture. The coverification of the resulting hardware/software virtual prototype is less expensive than working directly with its prototype. This coverification facilitates the understanding of the hardware/software communication and to detect errors present in the initial specification. In other words, the time necessary for the cosimulation of software modules (each processor executing less than 1000 lines of code per second) with high-level descriptions (10 to 100 thousand lines per second) is smaller as compared to the cosimulation using the low-level ones.

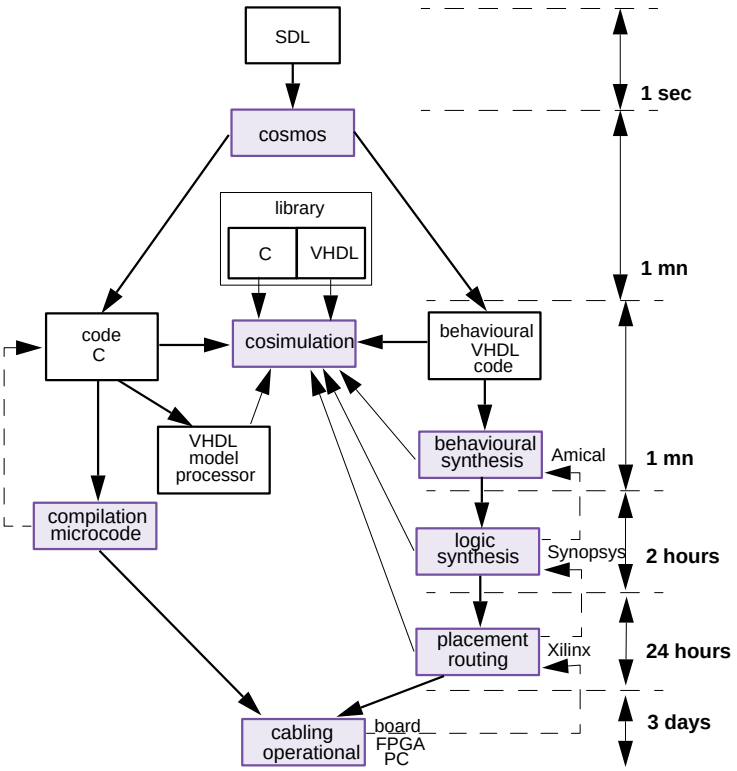


figure 5.36 : Time for traversing the design process

The link between cosimulation and cosynthesis within a unified environment permit to realize a multi-level cosimulation using the generated VHDL during the different intermediate phases of synthesis. In this manner, it becomes possible to cosimulate a system described at several levels of abstraction such as C-VHDL (RTL), C net-list or ever a micro-code net-list. The cosimulation allows a first validation of the virtual prototype and to correct synchronization errors. The communication of a hardware module with its external environment needs to be taken into account as a constraint to be satisfied. This is possible while the software and hardware are verified together during the initial stage of the specification.

Besides, the rapidity of execution of a high-level synthesis tool permits an iterative architecture exploration. In fact, the typical synthesis process of bigger hardware module necessitates an elapsed time of less than 10 minutes on a Sun Sparc 10 station. A lower level design iteration such as a logic synthesis may require several hours of even some days for a circuit that is relatively small such as the one involving 10,000 gates. The use of high-level synthesis permits to reduce the number of iterations of the lower level synthesis.

The main lack of the present version of cosmos is the availability of powerful estimation techniques that may guide the designer during architecture exploration.

5.7 Conclusion

This chapter introduced cosmos; a codesign tool based on user guided transformational partitioning. The design process starts with a SDL model (system level) and produces a distributed C-VHDL implementation. The tool allows:

- To use the expertise of the designer during the codesign process thanks to user-guided transformations.
- To understand the results of the codesign process thanks to the interactivity of the codesign process.
- To take into account partial existing solution through the execution of a specific set of transformations that produces the required target solution.
- A large design space exploration thanks to a fast and interactive interface. To start with a very high-level specification language, SDL, in order to produce an implementation of the system.

Chapitre 6

Conclusions et perspectives

6.1 Conclusions et perspectives

L'objectif principal de ce travail de thèse est d'étudier et de proposer des solutions aux divers problèmes qui apparaissent pendant la conception conjointe de systèmes mixtes logiciel/matériel. L'approche considère tout particulièrement les problèmes associés au prototype virtuel. Une approche globale des problèmes associés et les idées menant à leurs résolutions a été présentée. Cette approche a consisté à développer une méthodologie flexible et modulaire pour la conception de systèmes mixtes logiciel/matériel. Le premier problème dans la conception de tels systèmes est le développement simultané des aspects logiciels et matériels d'un système. Ce problème de conception logiciel/matériel est présent à tous les niveaux de la conception, tels que la spécification, la simulation, la génération d'architecture, et la réalisation physique.

Les motivations et les objectifs de cette thèse ont été définis dans le chapitre 1. Ce chapitre présente une vue de l'ensemble des problèmes existants lors de la conception conjointe des systèmes mixtes logiciels/matériels et plus particulièrement lors de l'étape de prototype virtuel.

Dans le chapitre 2, une présentation de l'état de l'art de la conception conjointe à travers les plus connues, des langages de spécification utilisés et des architectures d'implantation a été effectuée. Les techniques traditionnelles utilisées pour les systèmes de conception conjointe logiciel/matériel ainsi que l'état de l'art des environnements de conception ont été décrit afin de dégager les points communs et les différences entre-elles. Les caractéristiques des langages de spécification, ainsi que les architectures d'implantation, ont aussi été détaillées et comparées pour relever les caractéristiques et les domaines d'applications de chacune.

Le chapitre 3 décrit l'environnement de cosimulation développé permettant la vérification des systèmes mixtes matériel/logiciel à chaque étape du processus de conception. Les caractéristiques du système, les différentes descriptions dont il est composé et les descriptions synthétisées peuvent être toutes simulées dans cet environnement. Ce qui permet au concepteur de vérifier par simulation des sous-ensembles matériels et logiciels dans l'ensemble du processus de conception. La solution de cosimulation C-VHDL présentée a permis l'utilisation conjointe d'outils standards de simulation aussi bien pour la mise au point du VHDL que du C. Pour permettre cette cosimulation on a développé l'un outil de génération de l'interface C-VHDL nommé VCI. Il génère un environnement de cosimulation fiable et personnalisé pour une application spécifique en partant d'une description abstraite de l'interface entre les modules matériels et logiciels. Ce travail a été aussi à l'origine du développement de nouveaux outils de cosimulation permettant l'utilisation d'outils commerciaux (comme Synopsys et Cadence) et d'autres langages [6][111]. En effet, une version industrielle de VCI, nommé CoSim, fait actuellement partie de l'ensemble d'outils d'Unicad de SGS-Thomson Microelectronics [151]. L'outil a été également employé par d'autres groupes comme le FZI de l'université de Tübingen pour des expériences avec le langage Java (C. Trautwein, "Technical Memo Java Connector," Feb. 1997), PSA à Paris pour l'extension Matlab (P. Le Marrec, "Cosimulation C-VHDL-Matlab," Rapport de stage, DEA Micro-électronique, 1997), et Aérospatiale à Toulouse.

Le chapitre 4 détaille le prototype virtuel pour les systèmes mixtes matériels/logiciels. Il inclut la distribution des fonctionnalités d'un système entre le logiciel et le matériel jusqu'à sa réalisation sur une architecture qui comporte des processeurs et des composants de matériel. La modélisation proposée a permis la génération de code C et VHDL à partir des langages SDL et Solar. Pour cette étape, un outil qui génère automatiquement des descriptions en langage C et VHDL en partant du langage intermédiaire Solar a été développé. Les modèles générés en langage C pour les modules logiciels et en langage VHDL pour les modules matériels ont été définis pour obtenir un environnement unifié et flexible pour le prototype virtuel et la synthèse C/VHDL. Les modèles utilisés

pour résoudre les restrictions de synthèse et de simulation ne dépendent pas du langage de spécification et peuvent être appliqués à d'autres types de langages. Cet outil effectue aussi l'ordonnancement des modèles multiprocesseurs en langage C pour une architecture monoprocesseur. L'outil a été appliqué pour produire des modèles en C et en VHDL dans les projets Européen COMITY (particulièrement utilisé par l'Aérospatiale Missiles en France et Intracom en Grece) ainsi que CODAC. Un premier prototype de cette approche a été mis en pratique dans l'environnement de conception conjointe Cosmos. Cette approche, présentée dans la conférence ED&TC, a obtenu le prix de l'année 1995.

Le chapitre 5 donne une vue globale de l'environnement Cosmos de conception conjointe de logiciel/matériel. Il décrit les différentes phases d'une conception en partant d'une spécification niveau système jusqu'à sa réalisation sur une architecture qui comporte des processeurs et des composants matériels.

Des outils de compilation d'architecture appliqués aux circuits existent et sont relativement bien maîtrisés. Par contre, les études dans le domaine de la spécification, modélisation, et la synthèse de systèmes complets, correspondant à un besoin réel des industriels, ne font qu'émerger. Ce travail constitue ce qu'on pourrait appeler une première génération de systèmes de conception mixte logiciel/matériel. Actuellement, d'autres thèses sont en cours au sein du laboratoire. Elles constituent une continuation aux travaux présentés dans cette thèse. Ces travaux offrent, pour une partie, la possibilité d'étendre le prototypage virtuel a des langages autres que C et VHDL et pour permettre la validation du prototype à chacune des étapes du processus de codesign. Plusieurs points restent à développer, par exemple l'adaptation des interfaces des sous-systèmes communicants par rapport à l'architecture cible, la réutilisation de composants décrits dans plusieurs langages et à différents niveaux d'abstraction, ainsi que la validation de la spécification à tous les niveaux d'une conception mixte logicielle/matériel avec vérification précise des contraintes de temps réel.

Bibliographie

[1] M.Strik, J. Van Meerbergen, "Efficient Code Generation for In-House DSP Cores", Proceedings of EDTC, pages 244-249, 1995.

[2] P. Lapsley, "NSP Show Promise on Pentium, PowerPC", Microprocessor Report, pages 11-15, May 1995.

[3] M. Slater, "System Architects Look to the Future", Microprocessor Report, pp. 22-24, December 1995.

[4] N. Gehani, "C : An Advanced Introduction, ANSI C Edition", Computer Science Press, 1988.

[5] I. Bolsens, "Specification, Cosimulation and Hardware/Software Interfacing for Telecom Systems", Leuwen Codesign Course, Feb. 1997.

[6] P.G. Paulin, J. Fréhel, M. Harrand, E. Berrebi, C. Liem, F. Naçabal, J.-C. Herluison, "High-Level Synthesis and Codesign Methods: An Application to the Videophone Codec", EuroDAC/EuroVHDL, Brighton, U.K., Invited Paper, Sept. 1995.

[7] C. Valderrama, F. Naçabal, P. Paulin, A. Jerraya, "Automatic Generation of Interfaces for Distributed C-VHDL Cosimulation of Embedded Systems: an Industrial Experience", 7th International Workshop on Rapid Systems Prototyping, pp. 72-77, Thessaloniki, Greece, June 1996. Suported by a Brazilian Governement Fellowship CAPES/COFECUB 144-94 Brazil.

[8] C. Valderrama, A. Changuel, A. Jerraya, "Virtual Prototyping for Modular and Flexible Hardware/Software Systems", Journal of Design Automation for Embedded Systems, Kluwer Academic Publishers, Raul Camposano, Wayne Wolf Ed., Vol. 2, n. 3/4, May 1997. Suported by a Brazilian Governement Fellowship CAPES/COFECUB 144-94 Brazil.

[9] C. A. Valderrama, A. Changuel, P.V. Raghavan, M. Abid, T. Ben Ismail, A. A. Jerraya, " A Unified Model for Cosimulation and Cosynthesis of Mixed Hardware/Software Systems ", in Proc. of the European Design and Test Conference ED&TC'95, IEEE CS Press, Paris, France, March 1995. Suported by a Brazilian Governement Fellowship CAPES/COFECUB 144-94 Brazil. Best paper awards 1996.

[10] W. Wolf, “Guest Editor’s Introduction: Hardware-Software Codesign”, IEEE Design & Test of Computers, vol. 10, n. 3, pp. 5, Sept. 1993.

[11] E. Walkup, G. Boriello, “Automatic Synthesis of Device Drives for Hardware-Software Codesign”, Int’l. Workshop on Hardwaer-Software Codesign, Cambridge, IEEE CS Press, October 1993.

[12] G. Boriello, K. Buchenrieder, R. Camposano, E. Lee, R. Waxman, W. Wolf, “Hardware/Software Codesign”, IEEE Design & Test of Computers, Round Table, pp. 83-91, March 1993.

[13] K. Keutzer, “hardware-Software Codesign and ESDA”, Proc. 31st Design Automation conference (DAC), IEEE CS Press, pp. 435-436, June 1994.

[14] M. Voss, T. Ben Ismail, A. Jerraya, K. Kapp, “Towards a Theory for Hardware/Software Codesign”, Proc. Third Int’l Workshop on Hardware/Software Codesign (CODES/CASHE), IEEE Press, pp. 173-180, Grenoble, France, Sept. 1994.

[15] M. A. Richards, “The Rapid Prototyping of Application Specific Signal Processors (RASSP) Program: Overview and Status”, Proc. Int’l Workshop Rapid System Prototyping, IEEE CS Press, Order No. 5885, pp 1-6, Los Alamitos, Calif., 1994.

[16] W. Glunz, T. Kruse, T. Rossel, D. Monjau, Ïntegrating SDL and VHDL for system-Level Hardware Design”, Proc. IFIP Conf. Hardware Description Languages (CHDL), Plub. Elsevier Science, Ottawa, Canada, April, 1993.

[17] T. Ben Ismail, A. Jerraya, “Synthesis Steps and Design Models for Codesign”, IEEE Computer, special issue on Rapid-prototyping of Microelectronics Systems, Vol. 28, n. 2, pp. 44-52, February 1995.

[18] H. De Man, I. Bolsens, B. Lin, K.Van Rompaey, S. Vercauteren, D. Verkest, “Co-design for DSP systems”, NATO ASI Hardware/Software Codesign, Tremezzo, June 1995.

[19] G. Cambon, C. Vial, L. Maillet-Contoz, L. Torres, “Environnement d’aide à la Conception Concurrente de Systèmes Dédiés Matériel/Logiciel”, Workshop Codesign, CNET, Grenoble, 1995.

[20] V. Madisetti, T. Egolf, S. Famorzadeh, L-R. Dung, “Virtual Prototyping of Embedded DSP Systems”, Proc. of IEEE ICASSP 95, 1995.

[21] A. Kalavade, E. A. Lee, “A Hardware-Software Codesign Methodology for DSP Applications”, IEEE Design and Test of Computers, Vol.10, No.3, pp.16-28, Sept. 1993.

[22] A. Kalavade, E. A. Lee, “The Extended Partitioning Problem: Hardware/Software Mapping, Scheduling, and Implementation-Bin Selection”, Proceedings of Sixth International Workshop on Rapid Systems Prototyping, North Carolina, pp 12-18, June, 1995.

[23] K. Kim , Y. Kim, Y. Shin, K. Choi, S. Ha, “An Integrated Hardware-Software Cosimulation Environment with Automated Interface Generation”, Proc. Of 7th IEEE Int’l Workshop on Rapid System Prototyping, pp. 66-71, June 1996.

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

[25] A. Österling, T. Benner, R. Ernst, D. Herrmann, T. Scholz ,W.Ye, “The Cosyma System”, Hardware /Software Co-Design: Principles and Practice, Kluwer Academic Publishers, J. Staunstrup and W. Wolf edditors, pp. 263-305, 1997.

[26] G. Goossens, I. Bolsens, B. Lin, F. Catthoor, “Design of Heterogeneous ICs for Mobile and Personal Communication Systems”, Proc. Int’l Conf. on Computer Aided Design (ICCAD), IEEE CS Press, pp. 524-531, San Jose, California, Nov. 1994.

[27] K. Van Rompaey, D. Verkest, I. Bolsens, 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, p252, Sept. 1996.

[28] S. Antoniazzi, A. Balboni, W. Fornaciari, D. Sciuto, “A Methodology for Control-Dominated Systems Codesign”, Proc. Third Int’l Workshop on Hardware/Software Codesign (CODES/ CASHE), IEEE CS Press, pp. 2-9, Grenoble, France, Sept. 1994.

[29] M. Romdhani, R. P. Hautbois, A. Jeffroy, P. Chazelles, A. A. Jerraya, “Evaluation and Composition of Specification Languages, an Industrial Point of View”, Proc. IFIP Conf. Hardware Description Languages (CHDL), Japan, pp. 519-523, Septembre 1995.

[30] J. M. Daveau, G. F. Marchioro, C. A. Valderrama and A. A. Jerraya, “VHDL Generation from SDL Specification”, In Carlos D. Kloos and Eduard Cerny, editors, Proceedings of CHDL, pages 182--201. IFIP, Chapman-Hall, April 1997.

[31] C. Delgado Kloss, M. Lopez, et al. “From Lotos to VHDL”, Current Issues in Electronic Modelling , vol.3, Sept. 1995.

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

[33] J. Madsen, J. Brage, “Codesign Analysis of a Computer Graphics Application”, Journal: Design Automation of Embedded Systems, vol.1, no.1-2, January 1996.

[34] M.Theibinger, P.Stravers, H.Veit, “CASTLE: an interactive environment for hardware-software co-design”, In Proceedings of International Workshop on Hardware-Software Co-Design, pages 203-210, 1994.

[35] W. Wolf , “Hardware-Software Co-Design of Embedded Systems”, Proceedings of the IEEE, vol 82. no 7, pp. 967-989, July 1994.

[36] L. Lavagno, A. Sangiovanni-Vicentelli, H. Hsieh, “Embedded System Codesign: Synthesis and Verification”, in Hardware/Software Codesign, (G. De Michelli and M. Sami editors), pp. 213-242, Kluwer, 1996.

[37] G. Berry, “Hardware Implementation of Pure Esterel”, in Proceedings of the ACM Workshop on Formal Methods in VLSI Design, January 1991.

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

[39] F. Vahid, S. Narayan, D. Gajski, “SpecCharts: A VHDL Front-end for Embedded Systems”, IEEE Transactions on CAD of Integrated Circuits and Systems, vol. 14(6), pp. 694-06, 1995.

[40] M. Barbacci, “Instruction Set Processor Specification (ISPS): The Notation and Its Application”, IEEE trans. on Computers, vol. 30, pp. 24-40, January 1981.

[41] D. Ku, G. De Micheli, “HardwareC, a Language for Hardware Design”, Tech. Rep. CSL-TR-88-362,

Computer Systems Laboratory, Standford University, Aug. 1988.

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

[43] G. Holzmann, “Design and Validation of Computer Protocols. Englewood Cliffs, N.J: Prentice-Hall, 1991.

[44] R. Saracco, P. A. J. Tilanus, “CCITT SDL: An Overview of the Language and its Applications”, Computer Networks & ISDN SYstems, Special Issue on CCITT SDL, Vol. 13(2), pp. 65-74, 1987.

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

[46] S. Budowki, P. Dembinski, “An Introduction to Estelle: A Specification Language for Distributed Systems”, Computer Networks and ISDN Systems, vol. 13, no. 2, pp. 2-23, 1987.

[47] N. Halbwachs, F. Lagnier, C. Ratel, “Programming and verifying Real-time Systems by Means of the Synchronous Data-flow Lustre”, IEEE Transactions on Software Engineering, vol. 18, Sept. 1992.

[48] T. Gautier, P. Guernic, “Signal, a Declarative Language for Synchronous Programming of Real-time Systems”, Computer Science, Formal Languages and Computer Architectures, vol. 274, 1987.

[49] D. Harel, “Statecharts: a Visual Formalism for Complex Systems”, Science of Computer Programming 8, North-Holland, pp.231-274, 1987.

[50] J. Peterson, “Petri Net Theory and the Modeling of Systems”, Englewood Cliffs, N.J., Prentice-Hall, 1981.

[51] R. Milner, “Calculi for Synchrony and Asynchrony”, Theoretical Computer Science, vol. 23, pp. 267-310, 1983.

[52] C. Jones, “Systematic Software Development Usign VDM”, C.A.R. Hoare Series, Prentice Hall International Series in Computer Science, 1990.

[53] J. Spivey, “An Introduction to Z Formal Specifications”, Software Engineering Journal, pp. 40-50, January 1989.

[54] J. Abrial, “The B Book, Assigning Programs to Meaning”, Cambridge University Press, 1997.

[55] J. Bruijnin, “Evaluation and Integration of Specification Languages”, Computer Networks and ISDN Systems, vol. 13, no. 2, pp. 75-89, 1987.

[56] A. Changuel, M. Abid, C. Valderrama, A. Jerraya, “Environnement de Conception et de Prototypage de Systèmes Mixtes Logiciels/Matériels”, Journal Technique des Sciences Informatiques (TSI), 1996.

[57] G. Koch, U. Kebshull, W. Rosenstiel, “A prototyping Environment for Hardware/Software Codesign in the COBRA Project”, Proc. Third International Workshop Hw/Sw Codesign, Grenoble, France, pp.10-16, Sept. 1994.

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

[59] K.Buchenrieder, A.Sedlmeier, C.Veith, “Hw/Sw Codesign with PRAMs Using COdes”, Proc. IFIP Conf. Hardware Description Languages, Elsevier Science, Amsterdam, The Netherlands, pp. 55-68, 1993.

[60] X. Lavarenne, O. Seghrouchni, Y. Sorel, M. Sorine, “TheSynDex Software Environment for Real-time Distributed Systems Design and Implementation”, Proc. European Control Conference, July 1991.

[61] D. Gajski, J. Zhu, R. Dömer, “Essential Issues in Codesign”, Hardware/Software Co-Design: Principles and Practice, Kluwer Academic Publishers, J. Staunstrup and W. Wolf edditors, pp. 1-44, 1997.

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

[63] T. Ben Ismail, K. O'Brien, A. A. Jerraya,"PARTIF: Interactive System-level Partitioning", VLSI Design Vol. 3 no 3-4, pp. 333-345, 1995.

[64] D. Gajski, F. Vahid, S. Narayan, "A System Design Methodology: Executable-Specification Refinement", Proc. European Design & Test Conference (EDAC-ETC-EuroASIC), Paris, France, IEEE CS Press, pp. 458-463, February 1994.

[65] M. Srivastava, R. Brodersen, " SIERA: A unified framework for rapid-prototyping of system-level hardware and software", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pp. 676-693, June 1995.

[66] W. Wolf, " Object-Oriented Co-Synthesis of Distributed Embedded Systems", in Procs CHDL'95, IFIP, 1995.

[67] E. Barros, W. Rosentiel, X. Xiong, "A Method for Partitioning UNITY Language in Hardware and Software", Proc.of European Design Automation Conf. (Euro-DAC), 1994.

[68] E. D. Lagnese, D.E. Thomas, "Architectural partitioning of system level synthesis of integrated circuits", IEEE Trans. CAD/ICAS, vol. 10, no. 7, pp. 847-860, July 1991.

[69] F. Vahid, D. D. Gajski, "Specification Partitioning for System Design", Proceedings of the 29th Design Automation Conference (DAC), IEEE Press, June 1992.

[70] W. Hardt, R. Camposano, "Specification Analysis for HW/SW - Partitioning", In 3. GI/ITG Workshop, Passau, Germany, March 1995.

[71] D. Lanneer, J. VanPraet, W. Geurts, G. Goossens, "Chess: Retargetable Code Generation for Embedded DSP Processors", in Code Generation for Embedded Processors, ed. by P. Marwedel, G. Goossens, Kluwer Academic Publishers, 1995.

[72] P. Marwedel, G. Goessens, "Code Generation for Embedded Processors (DSP)", Kluwer Academic Publishers, 1995.

[73] A. Alomary, T. Nakata, Y. Honma, "An ASIP Instruction Set Optimization Algorithm with Functional Module Sharing Constraint", In ICCAD'93, pp. 526-532, 1993.

[74] R. Gupta, G. De Micheli, "Hardware-Software Cosynthesis using Reprogrammable Components", IEEE Design & Test of Computers, Vol 10, n.3, pp 29-41, September 1993.

[75] K. Buchenrieder, "A Prototyping Environment for Control-Oriented HW/SW Systems using StateCharts, Activity-Charts and FPGA's", Proc. Euro-DAX with Euro-VHDL, Grenoble, France, IEEE CS Press, pp.60-65, September 1994.

[76] P. Camurati, F. Corno, P.Prinetto, C. Bayol, B. Soulas, "System Level Modeling and Verification: A Comprehensive Design Methodology", Proc. European Design & Test Conference, Paris, France, IEEE CS Press, February 1994.

[77] T-Y. Yen, W. Wolf, "Communication synthesis for Distributed Embedded Systems", in Proceedings, ICCAD-95, pp. 64-69, IEEE Computer Society Press, 1995.

[78] M. Chiodo, D.Engels, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, K. Suzuki, A. Sangiovanni-Vincentelli, "A case Study in Computer Aided Codesign of Embedded Controllers", Design Automation for Embedded Systems, Vol.1, No.1-2, pp.51-67, Jan. 1996.

[79] J. Gong, D. Gajski, S.Narayan, "Software Estimation from Executable Specifications", Proc. European Design & Automation Conference (EuroDAC), IEEE CS Press, Grenoble, France, September 1994.

[80] B. Lin, S. Vercauteren, H. D. Man, "Embedded Architecture Cosynthesis and System Integration,"in Proceedings, Fourth International Workshop on Hardware/Software Codesign, pp. 2-9, IEEE Computer Society Press, 1996.

[81] D. Gajski, F. Vahid, "Specification and Design of Embedded Systems", IEEE Design & Test fo Computers, pp. 53-67, Spring 1995.

[82] P.Paulin, C. Liem, T.May, 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), KluwerAcademic Publishers, 1994.

[83] A. A. Jerraya, K. O'Brien, "Solar: an intermediate format for system-level modeling and synthesis", in J. Rozenblit and K. Buchenrieder, eds., Computer-Aided Software/Hardware Engineering, J. Rozenblit, K. Buchenrieder (eds) IEEE Prss, Piscataway, N. J., 1994.

[84] P.Paulin, M. Cornero, C. Liem, F.Naçabal, C. Donawa, S. Sutarwala, T.May, C. Valderrama, "Trends in Embedded System Technology: An Industrial Perspective", Hardware/Software Co-design, Kluwer Academic Publishers, 1996.

[85] "Synopsys VHDL System Simulator Interfaces Manual: C-language Interface", Synopsys Inc., Version 3.0b, June 1993.

[86] C. W. Krueger, "Software Reuse", ACM Computing Surveys, vol.24, no.2, pp.131-183, June 1992.

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

[88] A. Changuel, R. Rolland, A. A. Jerraya, "Design of an Adaptative Motors Controller based on Fuzzy Logic using Behavioural Synthesis", In Proc. European DAC with EURO-VHDL 96, Geneva, Switzerland, pp. 48-52, September 1996.

[89] M. Abid, A. Changuel, A. Jerraya, "Exploration of Hardware/Software Design Space Through a Codesign

of Robot Arm Controller", In Proc. EDA Conference with EURO-VHDL 96, Geneva, Switzerland, pp. 42-47, September 1996.

[90] G. R. Andrews, "Concurrent Programming, Principles and Practice", Benjamin/ Cummings (eds), Redwood City, Calif., pp. 484-494, 1991.

[91] ITU-T, "Z.100 Functional Specification and Description Language", Recommendation Z.100 - Z.104, March 1993.

[92] K. Ten Hagen, H. Meyer, "Timed and Untimed Hardware/Software Cosimulation: Application and Efficient Implementation," International Workshop on Hardware-Software Codesign, Cambridge, October 1993.

[93] T. Ben Ismail, M. Abid, K. O'Brien, A. Jerraya, "An Approach for Hardware-Software Codesign," RSP'94, Grenoble, France, June 1994.

[94] A. Kalavade, E. Lee, "A Hardware-Software Codesign Methodology for DSP Applications," IEEE Design and Test of Computers, pp.16-28, September 1993.

[95] D. Thomas, J. Adams, H. Schmit, "A Model and Methodology for Hardware-Software Codesign," IEEE Design & Test of Computers, Vol.10, N3, pp.6-15, September 1993.

[96] H. Fleukers, J. Jess, "ESCAPE: A Flexible Design and Simulation Environment," Proc. of The Synthesis and Simulation Meeting and International Interchange, SASIMIÕ93, pp.277-288, Oct.1993.

[97] R. Gupta, C. Coelho, G. De Micheli, "Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components," Proc. 29th Design Automation Conf. ACM/IEEE CS Press, pp. 225-234, 1992.

[98] W. Loucks, B. Doray, D. Agnew, "Experiences in Real Time Hardware-Software Cosimulation," Proc VHDL Int. Users Forum (VIUF), Ottawa, Canada, pp.47-57, April 1993.

[99] S. Lee, J. Rabaey, "A Hardware Software Cosimulation Environment," International Workshop on Hardware-Software Codesign, Cambridge, October 1993.

[100] N. Rethman, P. Wilsey, "RAPID: A Tool for Hardware/ Software Tradeoff Analysis," Proc. CHDL'93, Ottawa, Canada, April 1993.

[101] D. Becker, R. Singh, S. Tell, "An Engineering Environment for Hardware/Software Cosimulation," Proc. Design Automation Conference ACM, pp.129-134, 1992.

[102] K. O'Brien, T. Ben Ismail, A. Jerraya, "A Flexible Communication Modeling Paradigm for System-level Synthesis," International Workshop on Hardware-Software Codesign, Cambridge, October 1993.

[103] D. Ungar, R. Smith, C. Chambers, U. Holzle, "Object, Message, and Performance: How They Coexist in Self," IEEE Computer, October1992.

[104] G. Bochmann, "Specification Languages for Communication Protocols," Proc. CHDL'93, Ottawa, Canada, April 1993.

[105] J. Rowson, “HardwareSoftware Co-Simulation”, Proceedings of the 31st Design Automation Conference, pp. 439-440, San Diego, CA, USA, June 1994.

[106] Coumeri, D. E. Thomas, “A Simulation Environment for Hardware-Software Codesign”, Proc. ICCD Conference, IEEE CS Press, Oct. 1995.

[107] C. Liem, C. Valderrama, F. Naçabal, P. Paulin, A. Jerraya, “System-on-a-Chip Cosimulation and Compilation”, IEEE Design&Test of Computers, Vol. 14(2), pp. 16-25, April-June 1997.

[108] SGS-Thomson Microelectronics, “Sti1100 VideoPhone Codec: Preliminary Data Specification”, August 1993.

[109] M. Harrand et al., “A single Chip Videophone Video Encoder/decoder”, Proc. Of IEEE International Solid-State Circuits Conference, pp. 231-274, 1987.

[110] Wytrebwicz, S. Budkowski, "Communication Protocols Implemented in Hardware: VHDL generation from Estelle," Current Issue in Electronic Modelling, Vil. 3, pp. 77-98, Sept. 1995.

[111] P. Lemarec, C. Valderrama, F. Hessel, A. Jerraya, "Hardware, Software and Mechanical Cosimulation for Automotive Applications", RSP'98, Leuven, Belgium, June, 1998.

[112] R.K. Gupta, G. De Micheli, “A co-synthesis approach to embedded system design automation”, Design Automation for Embedded Systems, An International Journal, Volume 1, no.1-2, pp.121-145, Jan. 1996.

[113] B. Dasarathy, “Timing constraints of real-time systems: constructs for expressing them, method of validating them”, IEEE Trans. Software Engineering, vol. SE-11, no. 1, pp. 80-86, January 1985.

[114] Y. Shin, K. Choi, “Thread-based software synthesis for embedded system design," In Proceedings, The

European Design & Test Conference (ED&TC), pp.282-286, mar. 1996.

[115] J. Madsen, J.P. Brage, “Modeling Shared Variables in VHDL," Proceedings ACM’94, pp. 486-491, 1994.

[116] F. Vahid, S. Narayan, D. Gajski, “A Transformation for Integrating VHDL Behavioral Specification with Synthesis and Software Generation," Proceedings of ACM’94, pp. 552-557,1994.

[117] P. Eles, K. Kuchcinski, Z. Peng, M. Minea, “Synthesis of VHDL Concurrent Processes," Proceedings of ACM’94, pp. 540-545, 1994.

[118] S.C. Cheng, J. A. Stankovic, K. Ramamritham, “Scheduling algorithms for hard real-time systems," IEEE Tutorial on Hard real-time Systems, pp. 150-173, 1985.

[119] G. Sih, E.A. Lee, ”A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," IEEE Trans. Parallel and Distributed Systems, 4(20, pp. 175-187, Feb. 1993.

[120] J.L. Pino, E.A. Lee, “Hierarchical static scheduling of data-flow graphs onto multiple processors," Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing, 1994.

[121] P. Hoang, J. Rabaey, “Scheduling of DSP programs onto multiprocessors for maximum throughput," IEEE trans. On Signal Processing, 41(4), pp. 2225-2235, June 1993.

[122] C. Liu, J.W. Lanyland, “Scheduling algorithms for multiprogramming in a hard real-time environment," J.ACM, 20(1), pp.46-61, 1973.

[123] D. Peng, K.G. Shin, “Static allocation of periodic tasks with precedence constraints in distributed real-time systems," Proc. International Conference on Distributed Computing Systems, 1989.

[124] P. Chou, G. Borriello, “Interval scheduling: fine-grained code scheduling of embedded systems”, In Proceedings, 32nd. Design Automation Conference (DAC), San Francisco, CA, June 1995, pp.462-467.

[125] S. Vercauteren, B. Lin, and H. De Man, “A strategy for real-time kernel support in application-specific Hw/ Sw embedded architectures“, 33rd Design Automation conference, Las Vegas, USA, Jun. 1996.

[126] R. Ernst, J. Henkel, T. Benner, “Hardware-software cosynthesis for micro-controllers," IEEE Design & Test of Computers, pp. 64-75, 1993.

[127] M. Edwards, J. Forrest, “A development environment for the cosynthesis of embedded software-hardware systems," In Proc. of the European Design & Test Conference, 1994.

[128] B.G. Hald, J. Madsen, “A flexible architecture representation for high-level synthesis," Proceedings APCHDL, pp. 247-250, 1994.

[129] J. Madsen, “Interface modeling and synthesis," Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark, Nov. 1995.

[130] S. Narayan, D. Gajski, “Protocol generation for communication channels," IN Proc. Of DAC, pp. 547-548, 1994.

[131] S. Narayan, D. Gajski, “Interfacing incompatible protocols using interface process generation," 32nd. Design Automation Conference, 32’DAC, pp. 468-473, Moscone Center, San Francisco, CA, USA, June 12-16, 1995.

[132] K. Olukotun, R. Helaihel, J. Levitt, R. Ramirez, "A Software-hardware Cosynthesis Approach to Digital System Simulation," IEEE Micro, pp. 48- 58, Aug. 1994.

[133] T. Adam, M. Chandy, J. Dickson, "A Comparison of List Schedules for Parallel Processing Systems," Comm. ACM, Vol. 17, No. 12, pp. 685-690, Dec. 1974.

[134] M. Chiodo, P. Giusto, A. Jurecska, H. Hsieh, A. Sangiovanni-Vicentelli, L. Lavagno, "Hardware-Software Codesign of Embeeded Systems", IEEE Micro, pp. 26-36, Aug. 1994.

[135] P. Chou, E. Walkup, G. Boriello, "Scheduling for reactive Real-Time Systems," IEEE Micro, pp37-47, Aug. 1994.

[136] V. Mooney, T. Sakamoto, G. De Micheli, "Run-time Scheduler Synthesis for Hardware-Software Systems and Application to Robot Control Design," 5th. Int'l Workshop on Hardware/Software Codesign, Braunschweig, Germany, 1997.

[137] D. Thomas, P. Moorby, "The Verilog Hardware Description Language," Boston, Kluwer Academoc Publishers, pp. 146-167, 1991.

[138] P. Zave, M. Jackson, "Conjunciton as Composition," ACM trans. On Software engineering and methodology, 8(2), pp. 379-411, October 1993.

[139] M. Dolle and M. Schlett, "A Cost-Effective RISC/DSP Microprocessor for Embedded Systems", IEEE Micro, pp. 32-40, 1995.

[140] M. Flynn, V. Milutinovic editor, "High-Level Language Computer Architecture", Computer Science Press, 1989.

[141] B. Cole, "Processor Architectures", EE-times, Feb. 1995.

[142] I. Bolsens et al., "Hardware/sotware co-design of digital telecomunication systems,"Proceedings of the IEEE, vol. 85, pp. 391-418, March 1997.

[143] C. Liem, "Retargetable Compilers for Embedded Core Processors: Methods and Experiences in Industrial Applications", Kluwer Academic Publishers, 1997.

[144] A. Jerraya, M. Romdhani, C. Valderrama, P. Lemarrec et alias, "Languages for System Level Specification and Design", Book chapter, "Hardware-Software Codesign: Principles and Practice, Kluwer Academic Publishers, editors Jorgen Staustrup and Wayne Wolf, pp. 209-236, London, 1997.

[145] R. Ernst, "Target Architectures", Hardware /Software Co-Design: Principles and Practice, Kluwer Academic Publishers, J. Staunstrup and W. Wolf edditors, pp. 113-148, 1997.

[146] G. Fisher, "Rapid System Prototyping in an Open System Environment," Proceedings of the International Workshop on Rapid System Prototyping (RSP) IEEE CS Press, Grenoble, France, June 1994.

[147] J. Calvez, D. Heller, O. Pasquier, "System Performance Modeling and Analysis with VHDL : Benefits and Limitations", Proceedings of VHDL-Forum for CAD in Europe Conference, April 1995.

[148] P. Runstadler, R. Crevier, "Virtual Prototyping for System Design and Verification", Synopsys Documentation, March 1995.

[149] Alta Group, "Signal Processing WorkSystem : DSP Processor Models User's Guide", Cadence Design Systems, June 1996.

[150] Cadence, "Overview of the Leapfrog C interface", Cadence Design Systems, June 1994.

[151] F. Naçabal, "Outils pour l'exploration d'architectures programmables embarquées dans le cadre d'applications industrielles", Thèse de Doctorat, Laboratoire TIMA, Grenoble, France, Feb. 1998.

[152] C. Iseli, E. Sanchez, "Spyder: A SURE (Superscalar and Reconfigurable) Processor", The Journal of Supercomputing, 9(3), pp. 231-252, Kluwer Academic Publishers, 1995.

[153] J-P. Calvez, "A System Specification Model and Method," CIEM Current Issues In Electronic Modeling, vol 4. , High-level System Modeling: Specification and Design Methodologies", Kluwer Academic Publishers, Editors J. Bergé, O. Levia, J. Rouillard, 1996.

[154] J-P. Calvez, D. Heller, O. Pasquier, "Uninterpreted Cosimulation for Performance evaluation of Hw/Sw Systems,"proc. 4th International Workshop on Hardware/Software Codesign, pp. 132-139, CODES/CASHE'96, Pittsburg, USA, March 1996.

Publications personnelles

[155] N. Voros, S. Tsasakou, C. Valderrama, S. Arab, A. Birbas, M. Birbas, V. Mariatos and A. Andritsou , "Hardware - Software Co-design of embedded systems using multiple formalisms for application development ", IFIP International Conference FORTE/PSTV'98, Formal Description Techniques & Protocol Specification, Testing, and Verification, Paris, France, Nov. 1998.

[156] Workshop. N. Voros, S. Tsasakou, C. Valderrama, S. Arab, A. Birbas, M. Birbas, V. Mariatos and A. Andritsou, "ATM protocol design: A challenge for modern HW-SW codesign methodologies", 10th GI/ITG/GMM/IEEE Workshop, Herrenberg,1998.

[157] S. Tsasakou, N.Voros C. Valderrama, S.Arab,V. Mariatos, A. Birbas and M. Birbas , "A Hardware-Software co-design methodology for embedded telecommunication systems", EMMSEC'98, European Multimedia, Microprocessor Systems, and Electronic Commerce Conference and Exposition, Bordeaux, France, Sept. 1998.

[158] Journal. N. Voros, S. Tsasakou, C. Valderrama, S. Arab, A. Birbas, M. Birbas, V. Mariatos and A. Andritsou, "Hardware-Software co-design in practice: An approach using Multiple specification formalisms for industrial telecommunication applications", IEEE Micro Magazine,1998.

[159] P. Le Marrec, C. Valderrama, F. Hessel, A. Jerraya ,"Hardware, Software and Mechanical Cosimulation for Automotive Applications", IEEE International Workshop on Rapid Systems Prototyping RSP'98, Leuven, Belgium, June 1998.

[160] M. Abid, T. Ismail, A. Changuel, C. Valderrama, M. Romdhani, A. Jerraya, "Hardware/Software Codesign Methodology for Design of Embedded Systems", Journal, Integrated Computer Aided Engineering, vol. 5(1), pp. 6983, 1998.

[161] M.Abid, A. Changuel, C. Valderrama, A. Jerraya, "Conception de systèmes mixtes logiciels-matériels à partir de modèles C/VHDL", Journal, TSI, Technique et Science Informatiques, Hermes, vol 17(2), Feb. 1998.

[162] C.Liem, F. Nacabal, C. Valderrama, P. Paulin, A. Jerraya , "Cosimulation and Software Compilation Methodologies for the System-on-a-Chip in Multimedia", Journal, IEEE Design & Test of Computers, special issue on "Design, Test & ECAD in Europe", vol. 14(2), pp. 16-25, April-June1997.

[163] C.Valderrama, C. Liabeuf, F.Naçabal, P. Paulin, A. Jerraya, "Hardware/Software Cosimulation", Workshop, 7tn workshop on Synthesis and System Integration of Mixed technologies - SASIMI'97, pp. 110-119, Osaka - Japan, Dec. 1997.

[164] C. Valderrama, P. Le Marrec, A. Jerraya , "Multilanguage Cosimulation", Workshop, IEEE Second International High level Design Validation and Test Workshop - HLDVT'97, Oakland CA USA, Nov. 1997.

[165] A. Jerraya, M. Romdhani, C. Valderrama, P. Lemarrec et alias, "Languages for System Level Specification and Design", Book chapter, "Hardware-Software Codesign: Principles and Practice, Kluwer Academic Publishers, editors Jorgen Staustrup and Wayne Wolf, pp. 209-236, London, 1997.

[166] C. Valderrama, M. Romdhani, A. Jerraya et alias, "Cosmos: A Transformational Co-Design Tool for Multiprocessor Architectures", Book chapter, Hardware-Software Codesign: Principles and Practice, Kluwer Academic Publishers, editors Jorgen Staustrup and Wayne Wolf, pp. 235-261, London, 1997.

[167] C. A. Valderrama, François Naçabal, Pierre Paulin, Ahmed A. Jerraya, "Automatic VHDL-C Interface Generation for Distributed Cosimulation: Application to Large Design Examples", Journal, To appear Journal Design Automation For Embedded Systems - Special Issue on Rapid Prototyping, Kluwer academic publishers, 1997.

[168] M. Abid, T. Ismail, A. Changuel, C. Valderrama, A. Jerraya, "Design of Embedded Hardware/software Systems Using COSMOS Methodology", Journal, IEE Integrated Computer Aided Engineering, 1997.

[169] C.Valderrama, "Prototypage Virtuel pour la Génération d'Architectures Flexibles et Modulaires", Workshop, Colloque CAO de circuits integres et systemes, Villard de Lans, France, 15-17 Janvier 1997.

[170] J.Daveau, C. Valderrama, A. Jerraya, "VHDL generation from SDL specification", Conference, XIII IFIP Conference on Computer Hardware Description Languages (CHDL97), Toledo, Spain, April 1997.

[171] P.Paulin, M. Cornero, C. Liem, F. Nacabal, C. Donawa, S. Sutarwala, T. May, C. Valderrama, "Trends in embedded systems technology: An industrial perspective", Book chapter, Hardware/Software Codesign, Edited by M.G.Sami, G.DeMicheli, Kluwer Academic Publishers, NATO ASI Series, Vol. 310, pp. 311-338, London, Uk, 1996.

[172] J. Daveau, C. Valderrama, A. Jerraya et alias, "Cosmos: an SDL Based Hardware/Software Codesign Environment", Journal, Current Issues In Electronic Modelling - CIEM - Special issue on Co-Design & Co-Verification, Kluwer Academic Publishers, vol. 8, MA.USA, 1996.

[173] C. Valderrama, F. Naçabal, P. Paulin, A. Jerraya , "Automatic generation of interfaces for distributed c-vhdl cosimulation of embedded systems: an industrial ex perience", Workshop, 7th IEEE International Workshop on Rapid Systems Prototyping, pp. 72-77, Thessaloniki, Greece, June 1996.

[174] C. Valderrama, A. Changuel, A. Jerraya , "Virtual prototyping for modular and flexible hardware/software systems", Journal, Design Automation for Embedded Systems An International Journal Special Issue, Kluwer Academic Publishers, 2(3/4) pp.267-283, London, 1997. Suported by a Brazilian Governement Fellowship CAPES/COFECUB 144-94 Brazil.

[175] C.Valderrama, A. Changuel, P.V. Raghavan, M. Abid, T. Ben Ismail, A. Jerraya , "A unified model for co-simulation and co-synthesis of mixed hardware/software systems", Conference, The European Design and Test Conference ED&TC'95, Paris France, 6-9 March 1995. The best paper awards ED&TC'95

[176] C.Valderrama, A. Changuel, P.V. Raghavan, M. Abid, T. Ben Ismail, A. Jerraya, "An unified environment for co-simulation of hetherogeneous hadware/software systems", Conference, VHDL-Forum for CAD in Europe (VFE) EURO-MICRO VHDL, Grenoble, France, September 1994.

[177] VHDL design environment (93), Ms.Sc. These. COPPE/UFRJ, Rio de Janeiro, Brazil.

[178] VHDL development environment: compilation and intermediate format generation with simulation forecast (93), Conference. VIII Congresso da Sociedade Brasileira de Microeletronica (VIII SBMicro), Campinas, Brazil.

Glossaire

Cet appendice liste les sigles, les acronysmes et la terminologie utilisée dans cette thèse.

ACU : *Address Calculation Unit :* Unité de calcul d'addresses
ALU : *Arithmetic and Logic Unit :* Unité Arithmétique et Logique
ANSI : *American National Standards Institute*
API : *Application Programming Interface*
ASIC : *Application Specific Integrated Circuit :* Circuit intégré dédié à une application
ASIP : *Application Specific Integrated Processor :* Processeur intégré d'application spécifique (à jeu d'instructions dédié)
ATM : *Application Specific Integrated Processor :* Protocole de communication asynchrone à haut débit
C : Langage de programmation C
C^x : Langage de spécification qui est une extension au langage C pour décrire le parallélisme entre processus et les contraintes de temps
CAD : *Computer Aided Design :* CAO
CAO : Conception Assistée par ordinateur
CASHE : *Computer aided software/hardware engineering*
CCITT : *Consultative Committee on International Telegraphy and Telephony :* Organisme de normalisation des protocoles de communication
CFG : *Control Flow Graph :* Graphe de flot de contrôle
CLI : *C Language Interface :* Interface em langage C (pour le simulateur VHDL VSS de Synopsys)
CODEC : *Coder/Decoder :* Codeur/Décodeur
CODES : Outil de conception logicielle/matérielle à Siemens
Codesign : Conception conjointe matériel/logiciel : méthodologie destine à réduire l'espace existante entre l'implémentation de fonctionnalités du système en matériel (ASICs) ou logiciel (Processeurs)
CMOS : *Complementary Metal-Oxide-Semiconductor :* Technologie de fabrication des circuits intégrés
COSMOS : Projet de conception mixte logiciel/matériel développé dans l'équipe SLS du laboratoire TIMA
COSYMA : Outil de conception logicielle/matérielle à l'Université de Braunshweig
CPU : *Central Processing Unit :* Unité centrale de calcule
CSP : *Communicating Sequential Processes :* Langage de description de processus séquentiels communicants
CU : *Channel Unit :* Canal de communication
DAG : *Direct acyclique graph :* Graphe acyclique directe
DCT : *Discrete Cosinus Transform :* Treansformée en cosinus discrète
DCU : *Data Calculation Unit :* Unité de calcul de données
DFG : *Data Flow Graph :* Graphe de flot de donne
DSP : *Digital Signal Processing (processor) :* (processeur de) traitement numérique du Signal.
DU : *Design Unit :* Unité de conception
EDIF : *Electronic Design Interchange Format :* Format d'échange pour la conception de systèmes électronique
EEPROM : *Electrically Erasable Programable Read Only Memory :* memóire morte re-programmable
EFSM : *Extended Finite State Machine :* Machine d'états finis étendus (hierarchique et RPC)
FMI : *Foreign Model Interface :* Interface en langage C (Leapfrog de Cadence)
FPGA : *Field Programmable Gate Array :* Circuit programmable pour l'émulation
FSM : *Finite State Machine :* Machine d'état fini
FSMD : *Finite State Machine with Datapath :* Machine d'états finis avec chemin de données

GCC : *GNU c compiler :* compilateur C de GNU
HardwareC : Langage basé sur une extension au langage C pour intégrer le parallélisme et des aspects pour la description de matériel.
HDL : *Hardware Description Language :* Langage de description matérielle
IDCT : *Inverse Discrete Cosinus Transform :* Transformée en cosinus discrète inverse
IEEE : *Institut of Electrical and Electronic Engineers :* Organisation de normalisation dans le domaine électronique
IMEC : *Interuniversitair Micro-Elecktronica Centrum (Belgique)*
INPG : Institut National Polytechnique de Grenoble
I/O: *Input/Output :* unité ou procédure d'Entre/Sortie E/S
IP : *Internet Protocol :* Protocole Internet
IPC : *Inter Process Communication :* Méchanisme de communication inter-processus sous UNIX
ISO : *International Standards Organization :* Organisme de standardisation internationale
ITU : *International Telecommunications Union*
IVT : *Integrated Videotelephone Terminal :* Circuit de visiophonie de SGS-Thomson
LSI : *Large Scale Integrated Circuit :* Circuit intégré a large échelle
MAC : *Multiply Accumulator*
MCSE: Méthode et modèle de conception des systèmes développés à l'IRESTE de Nantes
MCU : *Microcontroller Unit* : Unité de micro-contrôle
MEF : Machine d'états finis
MIPS : *Million instructions Per Second :* Million d'instructions par seconde
MMIO : *Memory-Mapped Input/Output :* Entrée/Sortie adressée par la mémoire
MPU : *Multi processor Unit :* Unité multiprocesseur
MPEG : *Motion Picture Experts Group :* Standard Européen de codage de vidéo numérique
MSQ : *Micro-SeQuencer :* Processeur dédié embarqué dans le circuit IVT
PARTIF : *PARTItioning of extended Finite state machines :* Outil de découpage au niveau système développé par T. Ben Ismail au TIMA/SLS (Grenoble, France)
PE : *Processing Element :* Elément de calcule
PID : *Process Identification Descriptor :* Descripteur d'identification de un processus
PLD : *Programmable Logic Device :* Composant logique programmable
Ptolemy : Environnement de simulation et de conception logiciel/matériel développé à l'université de Berkeley (Californie, USA)
RAM : *Random Access Memory :* Mémoire vive
RAPID : Outil de conception logicielle/matérielle à l'Université de Cincinnati
RISC : Reduced Instruction Set Computer : Ordinateur à jeu d'instructions réduit
ROM : *Read Only Memory :* Mémoire morte
RPC : *Remote Procedure Call :* Appel de procédure à distance
RTL : *Register Transfer Level :* Représentation au niveau transfert des registres
SDL : *Specification and Description Language :* Langage de spécification et de description
SHM : *Shared Memory :* Méchanisme de communication par partage de mémoire
SOLAR : Format intermédiaire pour la représentation de systèmes à différents niveaux de spécification (utilisé dans le projet Cosmos, développé à TIMA)
SpecCharts : Langage de spécification au niveau système et du type StateCharts avec instructions très semblables à celles décrites en VHDL comportemental. Ce langage a été développé à l'Université d'Irvine
SpecSyn : Outil de conception logicielle/matérielle à l'Université d'Irvine
SpeedChart : Langage de description graphique et textuelle qui utilise le formalisme de StateCharts
ST : *State Table :* Table d'états
ST : *SGS-Thomson Microelectronics*
StateCharts : Langage synchrone ayant un formalisme visuel et permettant de décrire des MEFs ainsi que leur contrôle en fonction du temps
TCP : *Transmission Control Protocol :* Protocole de transmission asynchrone
TIMA : Techniques de l'informatique et de la Microélectronique pour l'Architecture d'ordinateurs
TOSCA : Outil de conception logicielle/matérielle à Italtel (Italie*)*
UAL : Unité Arithmétique et Logique
UNITY : Langage de spécification développé à l'Universitté de Tübingen en Allemagne

UNIX : Système d'exploitation
VERILOG : Langage de description du matériel
VHDL : *VHSIC Hardware Description Language :* Langage standard IEEE de description du matériel
VHSIC : *Very High Speed Integrated Circuit :* Circuit intégré à très haute vitesse
VIP : *VLIW Image processor :* Processeur dédié empbarqué dans le circuit IVT
VLC : *Variable Length Coding :* Codage à logeur variable
VLSI *:* *Very Large Scale Integration circuit :* Circuit a intégration à très grande échelle
VLIW *:* *Very Large Instruction Word :* Mot-instruction teès large
VOP *:* *Vértical Operation*
VSS : *VHDL Synopsys Simulator :* Simulateur VHDL de Synopsys

Chapitre 7

Annexe A

Exemple du contrôleur adaptatif de moteur : Les fichiers générés

Dans cette annexe seront présentés des exemples de descriptions, dans le format intermédiaire appelé Solar et les fichiers C-VHDL générés pour l'exemple du contrôleur adaptatif de moteur. Le format Solar sert de modèle pour toutes les étapes de synthèse. Les fichiers C et VHDL sont générés par l'outil s2cv. Pour la cosimulation, pour chaque module logiciel est généré un fichier du type .vci qui décrit son interface.

7.1 Etape de génération de code C/VHDL

Dans l'étape de génération de code, l'outil S2CV est utilisé pour produire le prototype virtuel à partir du fichier Solar de l'exemple du contrôleur de moteur. Cette génération est représentée par la figure 7.1.

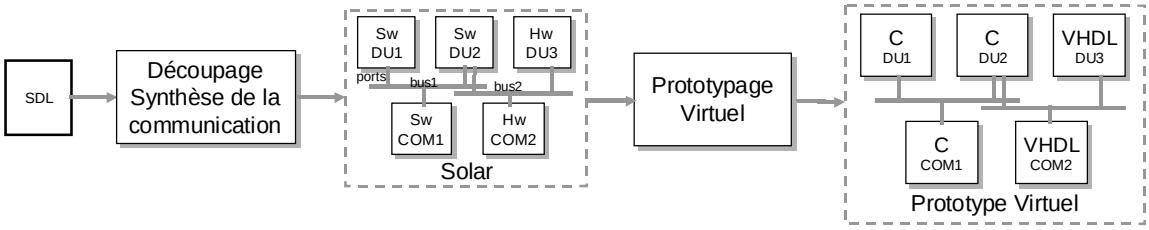


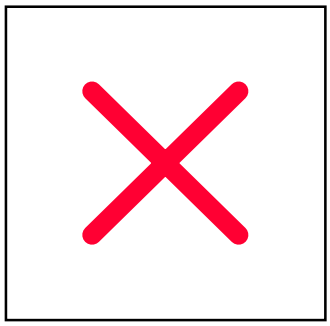
figure 7.1 : Étape de génération d'un prototype virtuel

Les différentes parties de l'exemple (résultat de l'étape de synthèse de communication) sont traduites séparément dans l'étape de prototypage virtuel. La sortie du prototypage virtuel est une architecture hétérogène représentée par du code C (processeurs virtuels logiciels) et du code VHDL (processeurs virtuels matériels). Les codes correspondants aux processus de communication, que ce soient les procédures d'appel locales ou les contrôleurs de communication, sont extraits de la bibliothèque de réalisation de la communication. Bien entendu, ils correspondent au mode de réalisation logiciel ou matériel qui leur est affecté. Ils sont donc décrits soit en code C, soit en VHDL.

7.1.1 Description Solar de l'exemple de Contrôle de Moteur

La description d'entrée de S2CV (le fichier Solar de l'exemple de contrôle de moteur) présente

une vision structurelle composée des instances des unités qui sont interconnectées par les réseaux (NETs). La description Solar structurelle (DESIGNUNIT *control*) est traduite en VHDL par S2CV pour permettre l'interconnexion des différentes parties C et VHDL.



```
(
SOLAR moteur
                (Version: "0")
  (DESIGNUNIT moteur
    (VIEW Structure
      (VIEWTYPE "Structure")
      (CONTENTS
        (INSTANCE algo2_I
          (VIEWREF algo2_behaviour algo2 )
          (PORTINSTANCE control_channel_wr_req )
          (PORTINSTANCE control_channel_rewr_rdy )
          (PORTINSTANCE control_channel_data_int )
          (PORTINSTANCE control_channel_data_real )
          (PORTINSTANCE algo2_channel_rewr_rdy )
          (PORTINSTANCE algo2_channel_read_req )
          (PORTINSTANCE algo2_channel_data_int )
          (PORTINSTANCE algo2_channel_data_real )
            (PROPERTY DESIGN SOFTWARE )          )
        (INSTANCE control_I
          (VIEWREF control_behaviour control )
          (PORTINSTANCE software_channel_wr_req )
          (PORTINSTANCE software_channel_rewr_rdy )
          (PORTINSTANCE software_channel_data_int )
          (PORTINSTANCE software_channel_data_real )
          (PORTINSTANCE algo1_channel_wr_req )
          (PORTINSTANCE algo1_channel_rewr_rdy )
          (PORTINSTANCE algo1_channel_data_int )
          (PORTINSTANCE algo1_channel_data_real )
          (PORTINSTANCE control_channel_rewr_rdy )
          (PORTINSTANCE control_channel_read_req )
          (PORTINSTANCE control_channel_data_int )
          (PORTINSTANCE control_channel_data_real )
          (PORTINSTANCE algo2_channel_wr_req )
          (PORTINSTANCE algo2_channel_rewr_rdy )
          (PORTINSTANCE algo2_channel_data_int )
          (PORTINSTANCE algo2_channel_data_real )
            (PROPERTY DESIGN HARDWARE )          )
        (INSTANCE algo2_channel_FIFO_4
          (VIEWREF FIFO_behaviour DU_FIFO_4 )
          (PORTINSTANCE (ARRAY algo2_channel_wr_req 2 ))
          (PORTINSTANCE (ARRAY algo2_channel_rewr_rdy 2 ))
          (PORTINSTANCE (ARRAY algo2_channel_read_req 2 ))
          (PORTINSTANCE algo2_channel_data_int )
          (PORTINSTANCE algo2_channel_data_real )
            (PROPERTY DESIGNTYPE CHANNELUNIT )
            (PROPERTY DESIGN HARDWARE )          )
        (INSTANCE software_I
          (VIEWREF software_behaviour software )
          (PORTINSTANCE software_channel_rewr_rdy )
          (PORTINSTANCE software_channel_read_req )
          (PORTINSTANCE software_channel_data_int )
          (PORTINSTANCE software_channel_data_real )
          (PORTINSTANCE control_channel_wr_req )
          (PORTINSTANCE control_channel_rewr_rdy )
          (PORTINSTANCE control_channel_data_int )
          (PORTINSTANCE control_channel_data_real )
            (PROPERTY DESIGN SOFTWARE )          )
        (INSTANCE algo1_I
          (VIEWREF algo1_behaviour algo1 )
          (PORTINSTANCE algo1_channel_rewr_rdy )
          (PORTINSTANCE algo1_channel_read_req )
          (PORTINSTANCE algo1_channel_data_int )
          (PORTINSTANCE algo1_channel_data_real )
          (PORTINSTANCE control_channel_wr_req )
          (PORTINSTANCE control_channel_rewr_rdy )
          (PORTINSTANCE control_channel_data_int )
          (PORTINSTANCE control_channel_data_real )
            (PROPERTY DESIGN SOFTWARE )          )
        (INSTANCE control_channel_FIFO_3
          (VIEWREF FIFO_behaviour DU_FIFO_3 )
          (PORTINSTANCE (ARRAY control_channel_wr_req 4 ))
          (PORTINSTANCE (ARRAY control_channel_rewr_rdy 4 ))
          (PORTINSTANCE (ARRAY control_channel_read_req 4 ))
          (PORTINSTANCE control_channel_data_int )
          (PORTINSTANCE control_channel_data_real )
            (PROPERTY DESIGNTYPE CHANNELUNIT )
            (PROPERTY DESIGN HARDWARE )          )
        (INSTANCE algo1_channel_FIFO_2
          (VIEWREF FIFO_behaviour DU_FIFO_2 )
          (PORTINSTANCE (ARRAY algo1_channel_wr_req 2 ))
          (PORTINSTANCE (ARRAY algo1_channel_rewr_rdy 2 ))
          (PORTINSTANCE (ARRAY algo1_channel_read_req 2 ))
          (PORTINSTANCE algo1_channel_data_int )
          (PORTINSTANCE algo1_channel_data_real )
            (PROPERTY DESIGNTYPE CHANNELUNIT )
            (PROPERTY DESIGN HARDWARE )          )
        (INSTANCE software_channel_FIFO_1
          (VIEWREF FIFO_behaviour DU_FIFO_1 )
          (PORTINSTANCE (ARRAY software_channel_wr_req 2 ))
          (PORTINSTANCE (ARRAY software_channel_rewr_rdy 2 ))
          (PORTINSTANCE (ARRAY software_channel_read_req 2 ))
          (PORTINSTANCE software_channel_data_int )
          (PORTINSTANCE software_channel_data_real )
            (PROPERTY DESIGNTYPE CHANNELUNIT )
            (PROPERTY DESIGN HARDWARE )          )
        (NET FIFO_4_N_1a
          (JOINED
            (PORTREF algo2_channel_wr_req
              (INSTANCEREF control_I )
            )
            (PORTREF (MEMBER algo2_channel_wr_req 2 )
              (INSTANCEREF algo2_channel_FIFO_4 )
            )
          )
        )
        (NET FIFO_4_N_2
          (JOINED
            (PORTREF algo2_channel_rewr_rdy
              (INSTANCEREF algo2_I )
            )
            (PORTREF (MEMBER algo2_channel_rewr_rdy 1 )
              (INSTANCEREF algo2_channel_FIFO_4 )
            )
          )
        )
        (NET FIFO_4_N_2a
          (JOINED
            (PORTREF algo2_channel_rewr_rdy
              (INSTANCEREF control_I )
            )
            (PORTREF (MEMBER algo2_channel_rewr_rdy 2 )
              (INSTANCEREF algo2_channel_FIFO_4 )
            )
          )
        )
        (NET FIFO_4_N_3
          (JOINED
            (PORTREF algo2_channel_read_req
              (INSTANCEREF algo2_I )
            )
            (PORTREF (MEMBER algo2_channel_read_req 1 )
              (INSTANCEREF algo2_channel_FIFO_4 )
            )
          )
        )
        (NET FIFO_4_N_4
          (JOINED
            (PORTREF algo2_channel_data_int
              (INSTANCEREF algo2_I )
            )
            (PORTREF algo2_channel_data_int
              (INSTANCEREF control_I )
            )
            (PORTREF algo2_channel_data_int
              (INSTANCEREF algo2_channel_FIFO_4 )
            )
          )
        )
        (NET FIFO_4_N_5
          (JOINED
            (PORTREF algo2_channel_data_real
              (INSTANCEREF algo2_I )
            )
            (PORTREF algo2_channel_data_real
              (INSTANCEREF control_I )
            )
            (PORTREF algo2_channel_data_real
              (INSTANCEREF algo2_channel_FIFO_4 )
            )
          )
        )
        (NET FIFO_3_N_1
          (JOINED
            (PORTREF control_channel_wr_req
              (INSTANCEREF software_I )
            )
            (PORTREF (MEMBER control_channel_wr_req 1 )
              (INSTANCEREF control_channel_FIFO_3 )
            )
          )
        )
        (NET FIFO_3_N_1a
          (JOINED
            (PORTREF control_channel_wr_req
              (INSTANCEREF algo2_I )
            )
            (PORTREF (MEMBER control_channel_wr_req 2 )
              (INSTANCEREF control_channel_FIFO_3 )
            )
          )
        )
        (NET FIFO_3_N_1b
```

```
  (JOINED
    (PORTREF control_channel_wr_req
      (INSTANCEREF algo1_I )
    )
    (PORTREF (MEMBER control_channel_wr_req 3 )
      (INSTANCEREF control_channel_FIFO_3 )
    )
  )
)
(NET FIFO_3_N_2
  (JOINED
    (PORTREF control_channel_rewr_rdy
      (INSTANCEREF software_I )
    )
    (PORTREF (MEMBER control_channel_rewr_rdy 1 )
      (INSTANCEREF control_channel_FIFO_3 )
    )
  )
)
(NET FIFO_3_N_2a
  (JOINED
    (PORTREF control_channel_rewr_rdy
      (INSTANCEREF algo2_I )
    )
    (PORTREF (MEMBER control_channel_rewr_rdy 2 )
      (INSTANCEREF control_channel_FIFO_3 )
    )
  )
)
(NET FIFO_3_N_2b
  (JOINED
    (PORTREF control_channel_rewr_rdy
      (INSTANCEREF algo1_I )
    )
    (PORTREF (MEMBER control_channel_rewr_rdy 3 )
      (INSTANCEREF control_channel_FIFO_3 )
    )
  )
)
(NET FIFO_3_N_2c
  (JOINED
    (PORTREF control_channel_rewr_rdy
      (INSTANCEREF control_I )
    )
    (PORTREF (MEMBER control_channel_rewr_rdy 4 )
      (INSTANCEREF control_channel_FIFO_3 )
    )
  )
)
(NET FIFO_3_N_3c
  (JOINED
    (PORTREF control_channel_read_req
      (INSTANCEREF control_I )
    )
    (PORTREF (MEMBER control_channel_read_req 4 )
      (INSTANCEREF control_channel_FIFO_3 )
    )
  )
)
(NET FIFO_3_N_4
  (JOINED
    (PORTREF control_channel_data_int
      (INSTANCEREF software_I )
    )
    (PORTREF control_channel_data_int
      (INSTANCEREF algo2_I )
    )
    (PORTREF control_channel_data_int
      (INSTANCEREF algo1_I )
    )
    (PORTREF control_channel_data_int
      (INSTANCEREF control_I )
    )
    (PORTREF control_channel_data_int
      (INSTANCEREF control_channel_FIFO_3 )
    )
  )
)
(NET FIFO_3_N_5
  (JOINED
    (PORTREF control_channel_data_real
      (INSTANCEREF software_I )
    )
    (PORTREF control_channel_data_real
      (INSTANCEREF algo2_I )
    )
    (PORTREF control_channel_data_real
      (INSTANCEREF algo1_I )
    )
    (PORTREF control_channel_data_real
      (INSTANCEREF control_I )
    )
    (PORTREF control_channel_data_real
      (INSTANCEREF control_channel_FIFO_3 )
    )
  )
)
(NET FIFO_2_N_1a
  (JOINED
    (PORTREF algo1_channel_wr_req
      (INSTANCEREF control_I )
    )
    (PORTREF (MEMBER algo1_channel_wr_req 2 )
      (INSTANCEREF algo1_channel_FIFO_2 )
    )
  )
)
(NET FIFO_2_N_2
  (JOINED
    (PORTREF algo1_channel_rewr_rdy
      (INSTANCEREF algo1_I )
    )
    (PORTREF (MEMBER algo1_channel_rewr_rdy 1 )
      (INSTANCEREF algo1_channel_FIFO_2 )
    )
  )
)
(NET FIFO_2_N_2a
  (JOINED
    (PORTREF algo1_channel_rewr_rdy
      (INSTANCEREF control_I )
    )
    (PORTREF (MEMBER algo1_channel_rewr_rdy 2 )
      (INSTANCEREF algo1_channel_FIFO_2 )
    )
  )
)
(NET FIFO_2_N_3
  (JOINED
    (PORTREF algo1_channel_read_req
      (INSTANCEREF algo1_I )
    )
    (PORTREF (MEMBER algo1_channel_read_req 1 )
      (INSTANCEREF algo1_channel_FIFO_2 )
    )
  )
)
(NET FIFO_2_N_4
  (JOINED
    (PORTREF algo1_channel_data_int
      (INSTANCEREF algo1_I )
    )
    (PORTREF algo1_channel_data_int
      (INSTANCEREF control_I )
    )
    (PORTREF algo1_channel_data_int
      (INSTANCEREF algo1_channel_FIFO_2 )
    )
  )
)
(NET FIFO_2_N_5
  (JOINED
    (PORTREF algo1_channel_data_real
      (INSTANCEREF algo1_I )
    )
    (PORTREF algo1_channel_data_real
      (INSTANCEREF control_I )
    )
    (PORTREF algo1_channel_data_real
      (INSTANCEREF algo1_channel_FIFO_2 )
    )
  )
)
(NET FIFO_1_N_1a
  (JOINED
    (PORTREF software_channel_wr_req
      (INSTANCEREF control_I )
    )
    (PORTREF (MEMBER software_channel_wr_req 2 )
      (INSTANCEREF software_channel_FIFO_1 )
    )
  )
)
(NET FIFO_1_N_2
  (JOINED
    (PORTREF software_channel_rewr_rdy
      (INSTANCEREF software_I )
    )
    (PORTREF (MEMBER software_channel_rewr_rdy 1 )
      (INSTANCEREF software_channel_FIFO_1 )
    )
  )
)
(NET FIFO_1_N_2a
  (JOINED
    (PORTREF software_channel_rewr_rdy
      (INSTANCEREF control_I )
    )
    (PORTREF (MEMBER software_channel_rewr_rdy 2 )
      (INSTANCEREF software_channel_FIFO_1 )
    )
  )
)
(NET FIFO_1_N_3
  (JOINED
    (PORTREF software_channel_read_req
      (INSTANCEREF software_I )
    )
    (PORTREF (MEMBER software_channel_read_req 1 )
      (INSTANCEREF software_channel_FIFO_1 )
    )
  )
)
(NET FIFO_1_N_4
  (JOINED
    (PORTREF software_channel_data_int
      (INSTANCEREF software_I )
    )
    (PORTREF software_channel_data_int
      (INSTANCEREF control_I )
    )
    (PORTREF software_channel_data_int
      (INSTANCEREF software_channel_FIFO_1 )
    )
  )
)
(NET FIFO_1_N_5
  (JOINED
    (PORTREF software_channel_data_real
      (INSTANCEREF software_I )
    )
    (PORTREF software_channel_data_real
      (INSTANCEREF control_I )
    )
    (PORTREF software_channel_data_real
      (INSTANCEREF software_channel_FIFO_1 )
    )
  )
)
)
)
)
```

7.1.2 Description Solar comportementale pour un module logiciel : *moteur2* (*algo2*)

Chaque module (DESIGNUNIT) a une représentation Solar comportementale. La description Solar pour un des moteurs (DESIGNUNIT algo2), qui sera traduite en C par S2CV, est représentée dans ce qui suit :

```
(DESIGNUNIT  algo2
    (VIEW  algo2_behaviour
      (VIEWTYPE "behaviour")
      (INTERFACE
        (PORT  control_channel_wr_req (DIRECTION  OUT ) (BIT ))
        (PORT  control_channel_rewr_rdy (DIRECTION  IN ) (BIT ))
        (PORT  control_channel_data_int (DIRECTION  INOUT ) (TYPEREF
RESOLVED_INTEGER ) )
        (PORT  control_channel_data_real (DIRECTION  INOUT ) (TYPEREF
RESOLVED_REAL ) )
        (PORT  algo2_channel_rewr_rdy (DIRECTION  IN ) (BIT ))
        (PORT  algo2_channel_read_req (DIRECTION  OUT ) (BIT ))
        (PORT  algo2_channel_data_int (DIRECTION  INOUT ) (TYPEREF
RESOLVED_INTEGER ) )
        (PORT  algo2_channel_data_real (DIRECTION  INOUT ) (TYPEREF
RESOLVED_REAL ) )
      )
      (PROCEDURE  control_channel_put_real
        (PARAMETER  sdl_signal (INTEGER ))
        (PARAMETER  param_1 (REAL ) )
        (STATETABLE  put_real
          (STATELIST  request  requestwait  putparam  paramwait )
          (ENTRYSTATE  request )
          (STATE  request
            (IF (=  control_channel_rewr_rdy  '0')
              (THEN
                (ASSIGN  control_channel_data_int  sdl_signal )
                (ASSIGN  control_channel_data_real  REAL_LOW )
                (ASSIGN  control_channel_wr_req  '1')
                (NEXTSTATE   requestwait )
              )
            )
            (NEXTSTATE   request )
          )
          (STATE  requestwait
            (WAIT  (UNTIL (=  control_channel_rewr_rdy  '1')))
            (ASSIGN  control_channel_wr_req  '0')
            (ASSIGN  control_channel_data_int  INTEGER_LOW )
            (ASSIGN  control_channel_data_real  REAL_LOW )
            (NEXTSTATE   putparam )
          )
          (STATE  putparam
            (ASSIGN  control_channel_data_int  INTEGER_LOW )
            (ASSIGN  control_channel_data_real  param_1 )
            (ASSIGN  control_channel_wr_req  '1')
            (NEXTSTATE   paramwait )
          )
          (STATE  paramwait
            (WAIT  (UNTIL (=  control_channel_rewr_rdy  '1')))
            (ASSIGN  control_channel_data_real  REAL_LOW )
            (ASSIGN  control_channel_data_int  INTEGER_LOW )
            (ASSIGN  control_channel_wr_req  '0')
          )
        )
      )
      (PROCEDURE  algo2_channel_get_signal
        (PARAMETER  sdl_signal (INTEGER ))
        (STATETABLE  get_signal
          (STATELIST  request  requestwait )
          (ENTRYSTATE  request )
          (STATE  request
            (IF (=  algo2_channel_rewr_rdy  '0')
              (THEN
                (ASSIGN  algo2_channel_read_req  '1')
                (ASSIGN  algo2_channel_data_int  INTEGER_LOW )
                (ASSIGN  algo2_channel_data_real  REAL_LOW )
                (NEXTSTATE   requestwait )
              )
            )
            (NEXTSTATE   request )
          )
          (STATE  requestwait
            (WAIT  (UNTIL (=  algo2_channel_rewr_rdy  '1')))
            (ASSIGN  sdl_signal  algo2_channel_data_int )
            (ASSIGN  algo2_channel_data_int  INTEGER_LOW )
            (ASSIGN  algo2_channel_data_real  REAL_LOW )
            (ASSIGN  algo2_channel_read_req  '0')
          )
        )
      )
      (PROCEDURE  algo2_channel_get_real
        (PARAMETER  param_1 (REAL ) )
        (STATETABLE  get_real
          (STATELIST  request  waitrdy )
          (ENTRYSTATE  request )
          (STATE  request
            (IF (=  algo2_channel_rewr_rdy  '0')
              (THEN
                (ASSIGN  algo2_channel_read_req  '1')
                (ASSIGN  algo2_channel_data_real  REAL_LOW )
                (ASSIGN  algo2_channel_data_int  INTEGER_LOW )
                (NEXTSTATE   waitrdy )
              )
            )
            (NEXTSTATE   request )
          )
          (STATE  waitrdy
            (WAIT  (UNTIL (=  algo2_channel_rewr_rdy  '1')))
            (ASSIGN  param_1  algo2_channel_data_real )
            (ASSIGN  algo2_channel_data_real  REAL_LOW )
            (ASSIGN  algo2_channel_data_int  INTEGER_LOW )
            (ASSIGN  algo2_channel_read_req  '0')
          )
        )
      )
    )
    (CONTENTS
      (VARIABLE  sdl_signal (INTEGER ))
      (VARIABLE  sdl_to (INTEGER ))
      (VARIABLE  sdl_sender (INTEGER ))
      (VARIABLE  s_0 (REAL ) )
      (VARIABLE  s_1 (REAL ) )
      (VARIABLE  e_0 (REAL ) )
      (VARIABLE  e_1 (REAL ) )
      (CONSTANT  pid_to_moteur2_read (INTEGER )(INITIALVALUE  3 ) )
      (CONSTANT  pid_to_moteur2_data (INTEGER )(INITIALVALUE  4 ) )
      (CONSTANT  moteur2_to_pid (INTEGER )(INITIALVALUE  5 ) )
      (CONSTANT  self (INTEGER )(INITIALVALUE  2 ) )
      (STATETABLE  statetable_algo2
        (STATELIST  initial  attente1  attente2 )
        (ENTRYSTATE  initial )
        (STATE  initial
          (ASSIGN  e_1  0.000000)
          (ASSIGN  s_0  0.000000)
          (ASSIGN  s_1  0.000000)
          (NEXTSTATE   attente1 )
        )
        (STATE  attente1
          (PCALL  algo2_channel_get_signal
            (PARAMETERASSIGN  sdl_signal  sdl_signal )
          )
          (CASE
            (ALT (=  sdl_signal  pid_to_moteur2_data )
              (PCALL  algo2_channel_get_real
                (PARAMETERASSIGN  param_1  e_0 )
              )
              (ASSIGN  s_0 (-  s_1 (*  0.005000(-  s_1  e_1 ))))
              (ASSIGN  s_1  s_0 )
              (ASSIGN  e_1  e_0 )
              (NEXTSTATE   attente2 )
            )
            (DEFAULT
              (CUCALL  deadlock_signal  algo2_channel
                (PARAMETERASSIGN  sdl_signal  sdl_signal )
              )
              (NEXTSTATE   attente1 )
            )
          )
        )
        (STATE  attente2
          (PCALL  algo2_channel_get_signal
            (PARAMETERASSIGN  sdl_signal  sdl_signal )
          )
          (CASE
            (ALT (=  sdl_signal  pid_to_moteur2_read )
              (PCALL  control_channel_put_real
                (PARAMETERASSIGN  sdl_signal  moteur2_to_pid )
                (PARAMETERASSIGN  param_1  s_0 )
              )
              (NEXTSTATE   attente1 )
            )
            (DEFAULT
              (CUCALL  deadlock_signal  algo2_channel
                (PARAMETERASSIGN  sdl_signal  sdl_signal )
              )
              (NEXTSTATE   attente2 )
            )
          )
        )
      )
    )
  )
)
```

7.1.3 Description Solar comportementale pour un module matériel : le contrôleur de vitesse (*control*)

Chaque module (DESIGNUNIT) a une représentation Solar comportementale. La description

Solar pour un module matériel (le contrôleur de vitesse : *DESIGNUNIT control*) qui sera traduite en VHDL par S2CV, est représentée par le modèle suivant :

```
(DESIGNUNIT control
   (VIEW control_behaviour
     (VIEWTYPE "behaviour")
     (INTERFACE
       (PORT software_channel_wr_req (DIRECTION OUT ) (BIT ))
       (PORT software_channel_rewr_rdy (DIRECTION IN ) (BIT ))
       (PORT software_channel_data_int (DIRECTION INOUT ) (TYPEREF
RESOLVED_INTEGER ) )
       (PORT software_channel_data_real (DIRECTION INOUT ) (TYPEREF
RESOLVED_REAL ) )
       (PORT algo1_channel_wr_req (DIRECTION OUT ) (BIT ))
       (PORT algo1_channel_rewr_rdy (DIRECTION IN ) (BIT ))
       (PORT algo1_channel_data_int (DIRECTION INOUT ) (TYPEREF
RESOLVED_INTEGER ) )
       (PORT algo1_channel_data_real (DIRECTION INOUT ) (TYPEREF
RESOLVED_REAL ) )
       (PORT control_channel_rewr_rdy (DIRECTION IN ) (BIT ))
       (PORT control_channel_read_req (DIRECTION OUT ) (BIT ))
       (PORT control_channel_data_int (DIRECTION INOUT ) (TYPEREF
RESOLVED_INTEGER ) )
       (PORT control_channel_data_real (DIRECTION INOUT ) (TYPEREF
RESOLVED_REAL ) )
       (PORT algo2_channel_wr_req (DIRECTION OUT ) (BIT ))
       (PORT algo2_channel_rewr_rdy (DIRECTION IN ) (BIT ))
       (PORT algo2_channel_data_int (DIRECTION INOUT ) (TYPEREF
RESOLVED_INTEGER ) )
       (PORT algo2_channel_data_real (DIRECTION INOUT ) (TYPEREF
RESOLVED_REAL ) )
     )
     (PROCEDURE software_channel_put_signal
       (PARAMETER sdl_signal (INTEGER ))
       (STATETABLE put_signal
         (STATELIST request requestwait )
         (ENTRYSTATE request )
         (STATE request
           (IF (= software_channel_rewr_rdy '0')
             (THEN
               (ASSIGN software_channel_data_int sdl_signal )
               (ASSIGN software_channel_data_real REAL_LOW )
               (ASSIGN software_channel_wr_req '1')
               (NEXTSTATE requestwait )
             )
           )
           (NEXTSTATE request )
         )
         (STATE requestwait
           (WAIT (UNTIL (= software_channel_rewr_rdy '1')))
           (ASSIGN software_channel_data_int INTEGER_LOW )
           (ASSIGN software_channel_data_real REAL_LOW )
           (ASSIGN software_channel_wr_req '0')
         )
       )
     )
     (PROCEDURE algo1_channel_put_real
       (PARAMETER sdl_signal (INTEGER ))
       (PARAMETER param_1 (REAL ) )
       (STATETABLE put_real
         (STATELIST request requestwait putparam paramwait )
         (ENTRYSTATE request )
         (STATE request
           (IF (= algo1_channel_rewr_rdy '0')
             (THEN
               (ASSIGN algo1_channel_data_int sdl_signal )
               (ASSIGN algo1_channel_data_real REAL_LOW )
               (ASSIGN algo1_channel_wr_req '1')
               (NEXTSTATE requestwait )
             )
           )
           (NEXTSTATE request )
         )
         (STATE requestwait
           (WAIT (UNTIL (= algo1_channel_rewr_rdy '1')))
           (ASSIGN algo1_channel_wr_req '0')
           (ASSIGN algo1_channel_data_int INTEGER_LOW )
           (ASSIGN algo1_channel_data_real REAL_LOW )
           (NEXTSTATE putparam )
         )
         (STATE putparam
           (ASSIGN algo1_channel_data_int INTEGER_LOW )
           (ASSIGN algo1_channel_data_real param_1 )
           (ASSIGN algo1_channel_wr_req '1')
           (NEXTSTATE paramwait )
         )
         (STATE paramwait
           (WAIT (UNTIL (= algo1_channel_rewr_rdy '1')))
           (ASSIGN algo1_channel_data_real REAL_LOW )
           (ASSIGN algo1_channel_data_int INTEGER_LOW )
           (ASSIGN algo1_channel_wr_req '0')
         )
       )
     )
     (PROCEDURE algo1_channel_put_signal
       (PARAMETER sdl_signal (INTEGER ))
       (STATETABLE put_signal
         (STATELIST request requestwait )
         (ENTRYSTATE request )
         (STATE request
           (IF (= algo1_channel_rewr_rdy '0')
             (THEN
               (ASSIGN algo1_channel_data_int sdl_signal )
               (ASSIGN algo1_channel_data_real REAL_LOW )
               (ASSIGN algo1_channel_wr_req '1')
               (NEXTSTATE requestwait )
             )
           )
           (NEXTSTATE request )
         )
         (STATE requestwait
           (WAIT (UNTIL (= algo1_channel_rewr_rdy '1')))
           (ASSIGN algo1_channel_data_int INTEGER_LOW )
           (ASSIGN algo1_channel_data_real REAL_LOW )
           (ASSIGN algo1_channel_wr_req '0')
         )
       )
     )
     (PROCEDURE control_channel_get_signal
       (PARAMETER sdl_signal (INTEGER ))
       (STATETABLE get_signal
         (STATELIST request requestwait )
         (ENTRYSTATE request )
         (STATE request
           (IF (= control_channel_rewr_rdy '0')
             (THEN
               (ASSIGN control_channel_read_req '1')
               (ASSIGN control_channel_data_int INTEGER_LOW )
               (ASSIGN control_channel_data_real REAL_LOW )
               (NEXTSTATE requestwait )
             )
           )
           (NEXTSTATE request )
         )
         (STATE requestwait
           (WAIT (UNTIL (= control_channel_rewr_rdy '1')))
           (ASSIGN sdl_signal control_channel_data_int )
           (ASSIGN control_channel_data_int INTEGER_LOW )
           (ASSIGN control_channel_data_real REAL_LOW )
           (ASSIGN control_channel_read_req '0')
         )
       )
     )
     (PROCEDURE control_channel_get_real
       (PARAMETER param_1 (REAL ) )
       (STATETABLE get_real
         (STATELIST request waitrdy )
         (ENTRYSTATE request )
         (STATE request
           (IF (= control_channel_rewr_rdy '0')
             (THEN
               (ASSIGN control_channel_read_req '1')
               (ASSIGN control_channel_data_real REAL_LOW )
               (ASSIGN control_channel_data_int INTEGER_LOW )
               (NEXTSTATE waitrdy )
             )
           )
           (NEXTSTATE request )
         )
         (STATE waitrdy
           (WAIT (UNTIL (= control_channel_rewr_rdy '1')))
           (ASSIGN param_1 control_channel_data_real )
           (ASSIGN control_channel_data_real REAL_LOW )
           (ASSIGN control_channel_data_int INTEGER_LOW )
           (ASSIGN control_channel_read_req '0')
         )
       )
     )
     (PROCEDURE algo2_channel_put_real
       (PARAMETER sdl_signal (INTEGER ))
       (PARAMETER param_1 (REAL ) )
       (STATETABLE put_real
         (STATELIST request requestwait putparam paramwait )
         (ENTRYSTATE request )
         (STATE request
           (IF (= algo2_channel_rewr_rdy '0')
             (THEN
               (ASSIGN algo2_channel_data_int sdl_signal )
               (ASSIGN algo2_channel_data_real REAL_LOW )
               (ASSIGN algo2_channel_wr_req '1')
               (NEXTSTATE requestwait )
             )
           )
           (NEXTSTATE request )
         )
         (STATE requestwait
           (WAIT (UNTIL (= algo2_channel_rewr_rdy '1')))
           (ASSIGN algo2_channel_wr_req '0')
           (ASSIGN algo2_channel_data_int INTEGER_LOW )
           (ASSIGN algo2_channel_data_real REAL_LOW )
           (NEXTSTATE putparam )
         )
         (STATE putparam
           (ASSIGN algo2_channel_data_int INTEGER_LOW )
           (ASSIGN algo2_channel_data_real param_1 )
           (ASSIGN algo2_channel_wr_req '1')
           (NEXTSTATE paramwait )
         )
         (STATE paramwait
           (WAIT (UNTIL (= algo2_channel_rewr_rdy '1')))
           (ASSIGN algo2_channel_data_real REAL_LOW )
           (ASSIGN algo2_channel_data_int INTEGER_LOW )
           (ASSIGN algo2_channel_wr_req '0')
         )
       )
     )
     (PROCEDURE algo2_channel_put_signal
       (PARAMETER sdl_signal (INTEGER ))
       (STATETABLE put_signal
         (STATELIST request requestwait )
         (ENTRYSTATE request )
         (STATE request
           (IF (= algo2_channel_rewr_rdy '0')
             (THEN
               (ASSIGN algo2_channel_data_int sdl_signal )
               (ASSIGN algo2_channel_data_real REAL_LOW )
               (ASSIGN algo2_channel_wr_req '1')
               (NEXTSTATE requestwait )
             )
           )
           (NEXTSTATE request )
         )
         (STATE requestwait
           (WAIT (UNTIL (= algo2_channel_rewr_rdy '1')))
           (ASSIGN algo2_channel_data_int INTEGER_LOW )
           (ASSIGN algo2_channel_data_real REAL_LOW )
           (ASSIGN algo2_channel_wr_req '0')
         )
       )
     )
     (CONTENTS
```

```
(VARIABLE sdl_signal (INTEGER ))
(VARIABLE sdl_to (INTEGER ))
(VARIABLE sdl_sender (INTEGER ))
(VARIABLE memoire (TYPEREF memory ) )
(VARIABLE u1 (REAL ) )
(VARIABLE u2 (REAL ) )
(VARIABLE u3 (REAL ) )
(VARIABLE u4 (REAL ) )
(VARIABLE u5 (REAL ) )
(VARIABLE u6 (REAL ) )
(VARIABLE u7 (REAL ) )
(VARIABLE u8 (REAL ) )
(VARIABLE u9 (REAL ) )
(VARIABLE u10 (REAL ) )
(VARIABLE u11 (REAL ) )
(VARIABLE u12 (REAL ) )
(VARIABLE u13 (REAL ) )
(VARIABLE u14 (REAL ) )
(VARIABLE temp (REAL ) )
(TYPEDEF (ARRAY memory 14 )(REAL ) )
(TYPEDEF mindexint (TYPEREF natural ) )
(CONSTANT pc_to_pid (INTEGER )(INITIALVALUE 2 ) )
(CONSTANT moteur2_to_pid (INTEGER )(INITIALVALUE 5 ) )
(CONSTANT moteur1_to_pid (INTEGER )(INITIALVALUE 8 ) )
(CONSTANT pid_to_pc (INTEGER )(INITIALVALUE 1 ) )
(CONSTANT pid_to_moteur2_read (INTEGER )(INITIALVALUE 3 ) )
(CONSTANT pid_to_moteur2_data (INTEGER )(INITIALVALUE 4 ) )
(CONSTANT pid_to_moteur1_read (INTEGER )(INITIALVALUE 6 ) )
(CONSTANT pid_to_moteur1_data (INTEGER )(INITIALVALUE 7 ) )
(CONSTANT self (INTEGER )(INITIALVALUE 4 ) )
(STATETABLE statetable_control
  (STATELIST initial attente_donnee_pc_1 attente_donnee_pc_2
attente_donnee_pc_3 attente_donnee_pc_4 attente_donnee_pc_5 attente_donnee_pc_6
attente_donnee_pc_7 attente_donnee_pc_8 attente_donnee_pc_9 attente_donnee_pc_10
attente_donnee_pc_11 attente_donnee_pc_12 attente_donnee_pc_13
attente_donnee_pc_14 calcul1 calcul2 )
  (ENTRYSTATE initial )
  (STATE initial
    (PCALL algo1_channel_put_real
      (PARAMETERASSIGN sdl_signal pid_to_moteur1_data )
      (PARAMETERASSIGN param_1 0.000000)
    )
    (PCALL algo2_channel_put_real
      (PARAMETERASSIGN sdl_signal pid_to_moteur2_data )
      (PARAMETERASSIGN param_1 0.000000)
    )
    (NEXTSTATE attente_donnee_pc_1 )
  )
  (STATE attente_donnee_pc_1
    (PCALL control_channel_get_signal
      (PARAMETERASSIGN sdl_signal sdl_signal )
    )
    (CASE
      (ALT (= sdl_signal pc_to_pid )
        (PCALL control_channel_get_real
          (PARAMETERASSIGN param_1 temp )
        )
        (ASSIGN (MEMBER memoire 1 ) temp )
        (PCALL software_channel_put_signal
          (PARAMETERASSIGN sdl_signal pid_to_pc )
        )
        (NEXTSTATE attente_donnee_pc_2 )
      )
      (DEFAULT
        (CUCALL deadlock_signal control_channel
          (PARAMETERASSIGN sdl_signal sdl_signal )
        )
        (NEXTSTATE attente_donnee_pc_1 )
      )
    )
  )
  (STATE attente_donnee_pc_2
    (PCALL control_channel_get_signal
      (PARAMETERASSIGN sdl_signal sdl_signal )
    )
    (CASE
      (ALT (= sdl_signal pc_to_pid )
        (PCALL control_channel_get_real
          (PARAMETERASSIGN param_1 temp )
        )
        (ASSIGN (MEMBER memoire 2 ) temp )
        (PCALL software_channel_put_signal
          (PARAMETERASSIGN sdl_signal pid_to_pc )
        )
        (NEXTSTATE attente_donnee_pc_3 )
      )
      (DEFAULT
        (CUCALL deadlock_signal control_channel
          (PARAMETERASSIGN sdl_signal sdl_signal )
        )
        (NEXTSTATE attente_donnee_pc_2 )
      )
    )
  )
  (STATE attente_donnee_pc_3
    (PCALL control_channel_get_signal
      (PARAMETERASSIGN sdl_signal sdl_signal )
    )
    (CASE
      (ALT (= sdl_signal pc_to_pid )
        (PCALL control_channel_get_real
          (PARAMETERASSIGN param_1 temp )
        )
        (ASSIGN (MEMBER memoire 3 ) temp )
        (PCALL software_channel_put_signal
          (PARAMETERASSIGN sdl_signal pid_to_pc )
        )
        (NEXTSTATE attente_donnee_pc_4 )
      )
      (DEFAULT
        (CUCALL deadlock_signal control_channel
          (PARAMETERASSIGN sdl_signal sdl_signal )
        )
        (NEXTSTATE attente_donnee_pc_3 )
      )
    )
  )
  (STATE attente_donnee_pc_4
    (PCALL control_channel_get_signal
      (PARAMETERASSIGN sdl_signal sdl_signal )
    )
    (CASE
      (ALT (= sdl_signal pc_to_pid )
        (PCALL control_channel_get_real
          (PARAMETERASSIGN param_1 temp )
        )
        (ASSIGN (MEMBER memoire 4 ) temp )
        (PCALL software_channel_put_signal
          (PARAMETERASSIGN sdl_signal pid_to_pc )
        )
        (NEXTSTATE attente_donnee_pc_5 )
      )
      (DEFAULT
        (CUCALL deadlock_signal control_channel
          (PARAMETERASSIGN sdl_signal sdl_signal )
        )
        (NEXTSTATE attente_donnee_pc_4 )
      )
    )
  )
  (STATE attente_donnee_pc_5
    (PCALL control_channel_get_signal
      (PARAMETERASSIGN sdl_signal sdl_signal )
    )
    (CASE
      (ALT (= sdl_signal pc_to_pid )
        (PCALL control_channel_get_real
          (PARAMETERASSIGN param_1 temp )
        )
        (ASSIGN (MEMBER memoire 5 ) temp )
        (PCALL software_channel_put_signal
          (PARAMETERASSIGN sdl_signal pid_to_pc )
        )
        (NEXTSTATE attente_donnee_pc_6 )
      )
      (DEFAULT
        (CUCALL deadlock_signal control_channel
          (PARAMETERASSIGN sdl_signal sdl_signal )
        )
        (NEXTSTATE attente_donnee_pc_5 )
      )
    )
  )
  (STATE attente_donnee_pc_6
    (PCALL control_channel_get_signal
      (PARAMETERASSIGN sdl_signal sdl_signal )
    )
    (CASE
      (ALT (= sdl_signal pc_to_pid )
        (PCALL control_channel_get_real
          (PARAMETERASSIGN param_1 temp )
        )
        (ASSIGN (MEMBER memoire 6 ) temp )
        (PCALL software_channel_put_signal
          (PARAMETERASSIGN sdl_signal pid_to_pc )
        )
        (NEXTSTATE attente_donnee_pc_7 )
      )
      (DEFAULT
        (CUCALL deadlock_signal control_channel
          (PARAMETERASSIGN sdl_signal sdl_signal )
        )
        (NEXTSTATE attente_donnee_pc_6 )
      )
    )
  )
  (STATE attente_donnee_pc_7
    (PCALL control_channel_get_signal
      (PARAMETERASSIGN sdl_signal sdl_signal )
    )
    (CASE
      (ALT (= sdl_signal pc_to_pid )
        (PCALL control_channel_get_real
          (PARAMETERASSIGN param_1 temp )
        )
        (ASSIGN (MEMBER memoire 7 ) temp )
        (PCALL software_channel_put_signal
          (PARAMETERASSIGN sdl_signal pid_to_pc )
        )
        (NEXTSTATE attente_donnee_pc_8 )
      )
      (DEFAULT
        (CUCALL deadlock_signal control_channel
          (PARAMETERASSIGN sdl_signal sdl_signal )
        )
        (NEXTSTATE attente_donnee_pc_7 )
      )
    )
  )
  (STATE attente_donnee_pc_8
    (PCALL control_channel_get_signal
      (PARAMETERASSIGN sdl_signal sdl_signal )
    )
    (CASE
      (ALT (= sdl_signal pc_to_pid )
        (PCALL control_channel_get_real
          (PARAMETERASSIGN param_1 temp )
        )
        (ASSIGN (MEMBER memoire 8 ) temp )
        (PCALL software_channel_put_signal
          (PARAMETERASSIGN sdl_signal pid_to_pc )
        )
        (NEXTSTATE attente_donnee_pc_9 )
      )
      (DEFAULT
        (CUCALL deadlock_signal control_channel
          (PARAMETERASSIGN sdl_signal sdl_signal )
        )
        (NEXTSTATE attente_donnee_pc_8 )
      )
    )
  )
  (STATE attente_donnee_pc_9
    (PCALL control_channel_get_signal
      (PARAMETERASSIGN sdl_signal sdl_signal )
    )
    (CASE
      (ALT (= sdl_signal pc_to_pid )
```

```
        (PCALL control_channel_get_real
          (PARAMETERASSIGN param_1 temp )
        )
        (ASSIGN (MEMBER memoire 9 ) temp )
        (PCALL software_channel_put_signal
          (PARAMETERASSIGN sdl_signal pid_to_pc )
        )
        (NEXTSTATE attente_donnee_pc_10 )
      )
      (DEFAULT
        (CUCALL deadlock_signal control_channel
          (PARAMETERASSIGN sdl_signal sdl_signal )
        )
        (NEXTSTATE attente_donnee_pc_9 )
      )
    )
  )
  (STATE attente_donnee_pc_10
    (PCALL control_channel_get_signal
      (PARAMETERASSIGN sdl_signal sdl_signal )
    )
    (CASE
      (ALT (= sdl_signal pc_to_pid )
        (PCALL control_channel_get_real
          (PARAMETERASSIGN param_1 temp )
        )
        (ASSIGN (MEMBER memoire 10 ) temp )
        (PCALL software_channel_put_signal
          (PARAMETERASSIGN sdl_signal pid_to_pc )
        )
        (NEXTSTATE attente_donnee_pc_11 )
      )
      (DEFAULT
        (CUCALL deadlock_signal control_channel
          (PARAMETERASSIGN sdl_signal sdl_signal )
        )
        (NEXTSTATE attente_donnee_pc_10 )
      )
    )
  )
  (STATE attente_donnee_pc_11
    (PCALL control_channel_get_signal
      (PARAMETERASSIGN sdl_signal sdl_signal )
    )
    (CASE
      (ALT (= sdl_signal pc_to_pid )
        (PCALL control_channel_get_real
          (PARAMETERASSIGN param_1 temp )
        )
        (ASSIGN (MEMBER memoire 11 ) temp )
        (PCALL software_channel_put_signal
          (PARAMETERASSIGN sdl_signal pid_to_pc )
        )
        (NEXTSTATE attente_donnee_pc_12 )
      )
      (DEFAULT
        (CUCALL deadlock_signal control_channel
          (PARAMETERASSIGN sdl_signal sdl_signal )
        )
        (NEXTSTATE attente_donnee_pc_11 )
      )
    )
  )
  (STATE attente_donnee_pc_12
    (PCALL control_channel_get_signal
      (PARAMETERASSIGN sdl_signal sdl_signal )
    )
    (CASE
      (ALT (= sdl_signal pc_to_pid )
        (PCALL control_channel_get_real
          (PARAMETERASSIGN param_1 temp )
        )
        (ASSIGN (MEMBER memoire 12 ) temp )
        (PCALL software_channel_put_signal
          (PARAMETERASSIGN sdl_signal pid_to_pc )
        )
        (NEXTSTATE attente_donnee_pc_13 )
      )
      (DEFAULT
        (CUCALL deadlock_signal control_channel
          (PARAMETERASSIGN sdl_signal sdl_signal )
        )
        (NEXTSTATE attente_donnee_pc_12 )
      )
    )
  )
  (STATE attente_donnee_pc_13
    (PCALL control_channel_get_signal
      (PARAMETERASSIGN sdl_signal sdl_signal )
    )
    (CASE
      (ALT (= sdl_signal pc_to_pid )
        (PCALL control_channel_get_real
          (PARAMETERASSIGN param_1 temp )
        )
        (ASSIGN (MEMBER memoire 13 ) temp )
        (PCALL software_channel_put_signal
          (PARAMETERASSIGN sdl_signal pid_to_pc )
        )
        (NEXTSTATE attente_donnee_pc_14 )
      )
      (DEFAULT
        (CUCALL deadlock_signal control_channel
          (PARAMETERASSIGN sdl_signal sdl_signal )
        )
        (NEXTSTATE attente_donnee_pc_13 )
      )
    )
  )
  (STATE attente_donnee_pc_14
    (PCALL control_channel_get_signal
      (PARAMETERASSIGN sdl_signal sdl_signal )
    )
    (CASE
      (ALT (= sdl_signal pc_to_pid )
        (PCALL control_channel_get_real
          (PARAMETERASSIGN param_1 temp )
        )
        (ASSIGN (MEMBER memoire 14 ) temp )
        (PCALL algo1_channel_put_signal
          (PARAMETERASSIGN sdl_signal pid_to_moteur1_read )
        )
        (NEXTSTATE calcul1 )
      )
      (DEFAULT
        (CUCALL deadlock_signal control_channel
          (PARAMETERASSIGN sdl_signal sdl_signal )
        )
        (NEXTSTATE attente_donnee_pc_14 )
      )
    )
  )
  (STATE calcul1
    (PCALL control_channel_get_signal
      (PARAMETERASSIGN sdl_signal sdl_signal )
    )
    (CASE
      (ALT (= sdl_signal moteur1_to_pid )
        (PCALL control_channel_get_real
          (PARAMETERASSIGN param_1 u2 )
        )
        (ASSIGN u1 (MEMBER memoire 1 ))
        (ASSIGN u3 (MEMBER memoire 2 ))
        (ASSIGN u4 (MEMBER memoire 3 ))
        (ASSIGN u5 (MEMBER memoire 4 ))
        (ASSIGN u6 (MEMBER memoire 5 ))
        (ASSIGN u7 (MEMBER memoire 6 ))
        (ASSIGN u8 (MEMBER memoire 7 ))
        (ASSIGN u9 (- u1 u2 ))
        (ASSIGN u10 (* u9 u6 ))
        (ASSIGN u11 (* u4 u7 ))
        (ASSIGN u12 (- u3 u11 ))
        (ASSIGN u13 (* u5 u8 ))
        (ASSIGN u14 (+ (+ u10 u12 ) u13 ))
        (PCALL algo1_channel_put_real
          (PARAMETERASSIGN sdl_signal pid_to_moteur1_data )
          (PARAMETERASSIGN param_1 u14 )
        )
        (ASSIGN (MEMBER memoire 4 ) u4 )
        (ASSIGN (MEMBER memoire 3 ) u9 )
        (ASSIGN (MEMBER memoire 2 ) u14 )
        (PCALL algo2_channel_put_signal
          (PARAMETERASSIGN sdl_signal pid_to_moteur2_read )
        )
        (NEXTSTATE calcul2 )
      )
      (DEFAULT
        (CUCALL deadlock_signal control_channel
          (PARAMETERASSIGN sdl_signal sdl_signal )
        )
        (NEXTSTATE calcul1 )
      )
    )
  )
  (STATE calcul2
    (PCALL control_channel_get_signal
      (PARAMETERASSIGN sdl_signal sdl_signal )
    )
    (CASE
      (ALT (= sdl_signal moteur2_to_pid )
        (PCALL control_channel_get_real
          (PARAMETERASSIGN param_1 u2 )
        )
        (ASSIGN u1 (MEMBER memoire 8 ))
        (ASSIGN u3 (MEMBER memoire 9 ))
        (ASSIGN u4 (MEMBER memoire 10 ))
        (ASSIGN u5 (MEMBER memoire 11 ))
        (ASSIGN u6 (MEMBER memoire 12 ))
        (ASSIGN u7 (MEMBER memoire 13 ))
        (ASSIGN u8 (MEMBER memoire 14 ))
        (ASSIGN u9 (- u1 u2 ))
        (ASSIGN u10 (* u9 u6 ))
        (ASSIGN u11 (* u4 u7 ))
        (ASSIGN u12 (- u3 u11 ))
        (ASSIGN u13 (* u5 u8 ))
        (ASSIGN u14 (+ (+ u10 u12 ) u13 ))
        (PCALL algo2_channel_put_real
          (PARAMETERASSIGN sdl_signal pid_to_moteur2_data )
          (PARAMETERASSIGN param_1 u14 )
        )
        (ASSIGN (MEMBER memoire 11 ) u4 )
        (ASSIGN (MEMBER memoire 10 ) u9 )
        (ASSIGN (MEMBER memoire 9 ) u14 )
        (PCALL algo1_channel_put_signal
          (PARAMETERASSIGN sdl_signal pid_to_moteur1_read )
        )
        (NEXTSTATE calcul1 )
      )
      (DEFAULT
        (CUCALL deadlock_signal control_channel
          (PARAMETERASSIGN sdl_signal sdl_signal )
        )
        (NEXTSTATE calcul2 )
      )
    )
  )
)
```

7.1.4 Description VHDL structurelle : structure plus configuration des instances

La description structurelle Solar de l'exemple (DESIGNUNIT *moteur*) est traduite en VHDL par S2CV pour permettre l'interconnexion des différentes parties C et VHDL. Le fichier VHDL fait l'interconnexion des instances. Toutes les instances sont configurées et les signaux externes (*clock* et *reset*) sont connectés dans les modules de cosimulation. La description VHDL équivalente est représentée ci-dessous :

```
-- s2cv version: 3.02
-- File: controleur_10.vhd
-- Solar file: controleur_10.solar
-- COSMOS/TIMA/INPG valderr@verdon.imag.fr
 ---------------------------------------------------------
library IEEE;
   use IEEE.STD_LOGIC_1164.all;
   use IEEE.STD_LOGIC_ARITH.all;
library SYNOPSYS;
   use SYNOPSYS.attributes.all;
library WORK;
   use WORK.controleur_10pkg.all;
   use WORK.user.all;
 ---------------------------------------------------------
entity moteur is
   Generic
   (
     IPCKEY: INTEGER:= 1
   );
   Port (
     CLK: IN BIT;
     RST: IN BIT
   );
end moteur;
 ---------------------------------------------------------
library IEEE;
   use IEEE.STD_LOGIC_1164.all;
   use IEEE.STD_LOGIC_ARITH.all;
library SYNOPSYS;
   use SYNOPSYS.attributes.all;
library WORK;
   use WORK.controleur_10pkg.all;
   use WORK.user.all;
 ---------------------------------------------------------
 architecture Structure of moteur is
   ---------------------------------------------------------
   signal FIFO_4_N_1a: BIT; --5007291196
   signal FIFO_4_N_2: BIT; --5007291196
   signal FIFO_4_N_2a: BIT; --5007291196
   signal FIFO_4_N_3: BIT; --5007291196
   signal FIFO_4_N_4: RESOLVED_INTEGER; --5007291196
   signal FIFO_4_N_5: RESOLVED_REAL; --5007291196
   signal FIFO_3_N_1: BIT; --5007291196
   signal FIFO_3_N_1a: BIT; --5007291196
   signal FIFO_3_N_1b: BIT; --5007291196
   signal FIFO_3_N_2: BIT; --5007291196
   signal FIFO_3_N_2a: BIT; --5007291196
   signal FIFO_3_N_2b: BIT; --5007291196
   signal FIFO_3_N_2c: BIT; --5007291196
   signal FIFO_3_N_3c: BIT; --5007291196
   signal FIFO_3_N_4: RESOLVED_INTEGER; --5007291196
   signal FIFO_3_N_5: RESOLVED_REAL; --5007291196
   signal FIFO_2_N_1a: BIT; --5007291196
   signal FIFO_2_N_2: BIT; --5007291196
   signal FIFO_2_N_2a: BIT; --5007291196
   signal FIFO_2_N_3: BIT; --5007291196
   signal FIFO_2_N_4: RESOLVED_INTEGER; --5007291196
   signal FIFO_2_N_5: RESOLVED_REAL; --5007291196
   signal FIFO_1_N_1a: BIT; --5007291196
   signal FIFO_1_N_2: BIT; --5007291196
   signal FIFO_1_N_2a: BIT; --5007291196
   signal FIFO_1_N_3: BIT; --5007291196
   signal FIFO_1_N_4: RESOLVED_INTEGER; --5007291196
   signal FIFO_1_N_5: RESOLVED_REAL; --5007291196
   ---------------------------------------------------------
   component algo2
     Generic
     (
       IPCKEY: INTEGER:= 1
     );
     Port (
       CLK: IN BIT;
       RST: IN BIT;
       control_channel_wr_req: OUT BIT;
       control_channel_rewr_rdy: IN BIT;
       control_channel_data_int: INOUT RESOLVED_INTEGER;
       control_channel_data_real: INOUT RESOLVED_REAL;
       algo2_channel_rewr_rdy: IN BIT;
       algo2_channel_read_req: OUT BIT;
       algo2_channel_data_int: INOUT RESOLVED_INTEGER;
       algo2_channel_data_real: INOUT RESOLVED_REAL
     );
   end component;
   ---------------------------------------------------------
   component control
     Generic
     (
       IPCKEY: INTEGER:= 1
     );
     Port (
       CLK: IN BIT;
       RST: IN BIT;
       software_channel_wr_req: OUT BIT;
       software_channel_rewr_rdy: IN BIT;
       software_channel_data_int: INOUT RESOLVED_INTEGER;
       software_channel_data_real: INOUT RESOLVED_REAL;
       algo1_channel_wr_req: OUT BIT;
       algo1_channel_rewr_rdy: IN BIT;
       algo1_channel_data_int: INOUT RESOLVED_INTEGER;
       algo1_channel_data_real: INOUT RESOLVED_REAL;
       control_channel_rewr_rdy: IN BIT;
       control_channel_read_req: OUT BIT;
       control_channel_data_int: INOUT RESOLVED_INTEGER;
       control_channel_data_real: INOUT RESOLVED_REAL;
       algo2_channel_wr_req: OUT BIT;
       algo2_channel_rewr_rdy: IN BIT;
       algo2_channel_data_int: INOUT RESOLVED_INTEGER;
       algo2_channel_data_real: INOUT RESOLVED_REAL
     );
   end component;
   ---------------------------------------------------------
   signal multi29_27: BIT_VECTOR(2 downto 1);
   signal multi29_9: BIT_VECTOR(2 downto 1);
   signal multi29_10: BIT_VECTOR(2 downto 1);
   component DU_FIFO_4
     Generic
     (
       IPCKEY: INTEGER:= 1
     );
     Port (
       CLK: IN BIT;
       RST: IN BIT;
       algo2_channel_wr_req: IN BIT_VECTOR(2 downto 1);
       algo2_channel_rewr_rdy: OUT BIT_VECTOR(2 downto 1);
       algo2_channel_read_req: IN BIT_VECTOR(2 downto 1);
       algo2_channel_data_int: INOUT RESOLVED_INTEGER;
       algo2_channel_data_real: INOUT RESOLVED_REAL
     );
   end component;
   ---------------------------------------------------------
   component software
     Generic
     (
       IPCKEY: INTEGER:= 1
     );
     Port (
       CLK: IN BIT;
       RST: IN BIT;
       software_channel_rewr_rdy: IN BIT;
       software_channel_read_req: OUT BIT;
       software_channel_data_int: INOUT RESOLVED_INTEGER;
       software_channel_data_real: INOUT RESOLVED_REAL;
       control_channel_wr_req: OUT BIT;
       control_channel_rewr_rdy: IN BIT;
       control_channel_data_int: INOUT RESOLVED_INTEGER;
       control_channel_data_real: INOUT RESOLVED_REAL
     );
   end component;
   ---------------------------------------------------------
   component algo1
     Generic
     (
       IPCKEY: INTEGER:= 1
     );
     Port (
       CLK: IN BIT;
       RST: IN BIT;
       algo1_channel_rewr_rdy: IN BIT;
       algo1_channel_read_req: OUT BIT;
       algo1_channel_data_int: INOUT RESOLVED_INTEGER;
       algo1_channel_data_real: INOUT RESOLVED_REAL;
       control_channel_wr_req: OUT BIT;
       control_channel_rewr_rdy: IN BIT;
       control_channel_data_int: INOUT RESOLVED_INTEGER;
       control_channel_data_real: INOUT RESOLVED_REAL
     );
   end component;
   ---------------------------------------------------------
   signal multi42_5: BIT_VECTOR(4 downto 1);
   signal multi42_6: BIT_VECTOR(4 downto 1);
   signal multi42_26: BIT_VECTOR(4 downto 1);
   component DU_FIFO_3
     Generic
     (
       IPCKEY: INTEGER:= 1
     );
     Port (
       CLK: IN BIT;
       RST: IN BIT;
       control_channel_wr_req: IN BIT_VECTOR(4 downto 1);
       control_channel_rewr_rdy: OUT BIT_VECTOR(4 downto 1);
       control_channel_read_req: IN BIT_VECTOR(4 downto 1);
       control_channel_data_int: INOUT RESOLVED_INTEGER;
       control_channel_data_real: INOUT RESOLVED_REAL
     );
   end component;
   ---------------------------------------------------------
   signal multi44_22: BIT_VECTOR(2 downto 1);
   signal multi44_23: BIT_VECTOR(2 downto 1);
   signal multi44_41: BIT_VECTOR(2 downto 1);
   component DU_FIFO_2
     Generic
```

```
    (
      IPCKEY: INTEGER:= 1
    );
    Port (
      CLK: IN BIT;
      RST: IN BIT;
      algo1_channel_wr_req: IN BIT_VECTOR(2 downto 1);
      algo1_channel_rewr_rdy: OUT BIT_VECTOR(2 downto 1);
      algo1_channel_read_req: IN BIT_VECTOR(2 downto 1);
      algo1_channel_data_int: INOUT RESOLVED_INTEGER;
      algo1_channel_data_real: INOUT RESOLVED_REAL
    );
  end component;
  ---------------------------------------------------------
  signal multi46_18: BIT_VECTOR(2 downto 1);
  signal multi46_19: BIT_VECTOR(2 downto 1);
  signal multi46_37: BIT_VECTOR(2 downto 1);
  component DU_FIFO_1
    Generic
    (
      IPCKEY: INTEGER:= 1
    );
    Port (
      CLK: IN BIT;
      RST: IN BIT;
      software_channel_wr_req: IN BIT_VECTOR(2 downto 1);
      software_channel_rewr_rdy: OUT BIT_VECTOR(2 downto 1);
      software_channel_read_req: IN BIT_VECTOR(2 downto 1);
      software_channel_data_int: INOUT RESOLVED_INTEGER;
      software_channel_data_real: INOUT RESOLVED_REAL
    );
  end component;
  ---------------------------------------------------------
begin
  ---------------------------------------------------------

  algo2_I: algo2
    Generic Map (
      IPCKEY=> 1
    )
    Port Map(
      CLK=> CLK,
      RST=> RST,
      control_channel_wr_req=> FIFO_3_N_1a,
      control_channel_rewr_rdy=> FIFO_3_N_2a,
      control_channel_data_int=> FIFO_3_N_4,
      control_channel_data_real=> FIFO_3_N_5,
      algo2_channel_rewr_rdy=> FIFO_4_N_2,
      algo2_channel_read_req=> FIFO_4_N_3,
      algo2_channel_data_int=> FIFO_4_N_4,
      algo2_channel_data_real=> FIFO_4_N_5
    );
  ---------------------------------------------------------
  control_I: control
    Generic Map (
      IPCKEY=> 2
    )
    Port Map(
      CLK=> CLK,
      RST=> RST,
      software_channel_wr_req=> FIFO_1_N_1a,
      software_channel_rewr_rdy=> FIFO_1_N_2a,
      software_channel_data_int=> FIFO_1_N_4,
      software_channel_data_real=> FIFO_1_N_5,
      algo1_channel_wr_req=> FIFO_2_N_1a,
      algo1_channel_rewr_rdy=> FIFO_2_N_2a,
      algo1_channel_data_int=> FIFO_2_N_4,
      algo1_channel_data_real=> FIFO_2_N_5,
      control_channel_rewr_rdy=> FIFO_3_N_2c,
      control_channel_read_req=> FIFO_3_N_3c,
      control_channel_data_int=> FIFO_3_N_4,
      control_channel_data_real=> FIFO_3_N_5,
      algo2_channel_wr_req=> FIFO_4_N_1a,
      algo2_channel_rewr_rdy=> FIFO_4_N_2a,
      algo2_channel_data_int=> FIFO_4_N_4,
      algo2_channel_data_real=> FIFO_4_N_5
    );
  ---------------------------------------------------------
  multi29_27(2)<= FIFO_4_N_1a;
  FIFO_4_N_2<= multi29_9(1);
  FIFO_4_N_2a<= multi29_9(2);
  multi29_10(1)<= FIFO_4_N_3;
  algo2_channel_FIFO_4: DU_FIFO_4
    Generic Map (
      IPCKEY=> 3
    )
    Port Map(
      CLK=> CLK,
      RST=> RST,
      algo2_channel_wr_req=> multi29_27,
      algo2_channel_rewr_rdy=> multi29_9,
      algo2_channel_read_req=> multi29_10,
      algo2_channel_data_int=> FIFO_4_N_4,
      algo2_channel_data_real=> FIFO_4_N_5
    );
  ---------------------------------------------------------
  software_I: software
    Generic Map (
      IPCKEY=> 4
    )
    Port Map(
      CLK=> CLK,
      RST=> RST,
      software_channel_rewr_rdy=> FIFO_1_N_2,
      software_channel_read_req=> FIFO_1_N_3,
      software_channel_data_int=> FIFO_1_N_4,
      software_channel_data_real=> FIFO_1_N_5,
      control_channel_wr_req=> FIFO_3_N_1,
      control_channel_rewr_rdy=> FIFO_3_N_2,
      control_channel_data_int=> FIFO_3_N_4,
      control_channel_data_real=> FIFO_3_N_5
    );
  ---------------------------------------------------------
  algo1_I: algo1
    Generic Map (
      IPCKEY=> 5
    )
    Port Map(
      CLK=> CLK,
      RST=> RST,
      algo1_channel_rewr_rdy=> FIFO_2_N_2,
      algo1_channel_read_req=> FIFO_2_N_3,
      algo1_channel_data_int=> FIFO_2_N_4,
      algo1_channel_data_real=> FIFO_2_N_5,
      control_channel_wr_req=> FIFO_3_N_1b,
      control_channel_rewr_rdy=> FIFO_3_N_2b,
      control_channel_data_int=> FIFO_3_N_4,
      control_channel_data_real=> FIFO_3_N_5
    );
  ---------------------------------------------------------
  multi42_5(1)<= FIFO_3_N_1;
  multi42_5(2)<= FIFO_3_N_1a;
  multi42_5(3)<= FIFO_3_N_1b;
  FIFO_3_N_2<= multi42_6(1);
  FIFO_3_N_2a<= multi42_6(2);
  FIFO_3_N_2b<= multi42_6(3);
  FIFO_3_N_2c<= multi42_6(4);
  multi42_26(4)<= FIFO_3_N_3c;
  control_channel_FIFO_3: DU_FIFO_3
    Generic Map (
      IPCKEY=> 6
    )
    Port Map(
      CLK=> CLK,
      RST=> RST,
      control_channel_wr_req=> multi42_5,
      control_channel_rewr_rdy=> multi42_6,
      control_channel_read_req=> multi42_26,
      control_channel_data_int=> FIFO_3_N_4,
      control_channel_data_real=> FIFO_3_N_5
    );
  ---------------------------------------------------------
  multi44_22(2)<= FIFO_2_N_1a;
  FIFO_2_N_2<= multi44_23(1);
  FIFO_2_N_2a<= multi44_23(2);
  multi44_41(1)<= FIFO_2_N_3;
  algo1_channel_FIFO_2: DU_FIFO_2
    Generic Map (
      IPCKEY=> 7
    )
    Port Map(
      CLK=> CLK,
      RST=> RST,
      algo1_channel_wr_req=> multi44_22,
      algo1_channel_rewr_rdy=> multi44_23,
      algo1_channel_read_req=> multi44_41,
      algo1_channel_data_int=> FIFO_2_N_4,
      algo1_channel_data_real=> FIFO_2_N_5
    );
  ---------------------------------------------------------
  multi46_18(2)<= FIFO_1_N_1a;
  FIFO_1_N_2<= multi46_19(1);
  FIFO_1_N_2a<= multi46_19(2);
  multi46_37(1)<= FIFO_1_N_3;
  software_channel_FIFO_1: DU_FIFO_1
    Generic Map (
      IPCKEY=> 8
    )
    Port Map(
      CLK=> CLK,
      RST=> RST,
      software_channel_wr_req=> multi46_18,
      software_channel_rewr_rdy=> multi46_19,
      software_channel_read_req=> multi46_37,
      software_channel_data_int=> FIFO_1_N_4,
      software_channel_data_real=> FIFO_1_N_5
    );
  ---------------------------------------------------------
end Structure;
---------------------------------------------------------
-- synopsys_synthesis_off --
configuration moteur_Structure of moteur is
  for Structure
    for algo2_I: algo2
      use entity WORK.algo2itf(CLI);
    end for;
    for control_I: control
      use entity WORK.control(control_behaviour);
    end for;
    for algo2_channel_FIFO_4: DU_FIFO_4
      use entity WORK.DU_FIFO_4(FIFO_behaviour);
    end for;
    for software_I: software
      use entity WORK.softwareitf(CLI);
    end for;
    for algo1_I: algo1
      use entity WORK.algo1itf(CLI);
    end for;
    for control_channel_FIFO_3: DU_FIFO_3
      use entity WORK.DU_FIFO_3(FIFO_behaviour);
    end for;
    for algo1_channel_FIFO_2: DU_FIFO_2
      use entity WORK.DU_FIFO_2(FIFO_behaviour);
    end for;
    for software_channel_FIFO_1: DU_FIFO_1
      use entity WORK.DU_FIFO_1(FIFO_behaviour);
    end for;
  end for;
end moteur_Structure;
-- synopsys_synthesis_on --
---------------------------------------------------------
library IEEE;
  use IEEE.STD_LOGIC_1164.all;
  use IEEE.STD_LOGIC_ARITH.all;
library SYNOPSYS;
  use SYNOPSYS.attributes.all;
library WORK;
  use WORK.controleur_10pkg.all;
  use WORK.user.all;
---------------------------------------------------------
entity TB_moteur is
end TB_moteur;
---------------------------------------------------------
library IEEE;
  use IEEE.STD_LOGIC_1164.all;
  use IEEE.STD_LOGIC_ARITH.all;
library SYNOPSYS;
  use SYNOPSYS.attributes.all;
```

```
library WORK;
  use WORK.controleur_10pkg.all;
  use WORK.user.all;
 ----------------------------------------------------
architecture Structure of TB_moteur is
  ----------------------------------------------------
  signal CLK: BIT:= '1';
  signal RST: BIT:= '0';
  component moteur
    Generic
    (
      IPCKEY: INTEGER:= 1
    );
    Port (
      CLK: IN BIT;
      RST: IN BIT
    );
  end component;
  ----------------------------------------------------
begin
  ----------------------------------------------------
  Imoteur: moteur
  Generic Map (
    IPCKEY=> 1
  )
  Port Map (
    CLK=> CLK,
    RST=> RST
  );
  ----------------------------------------------------
  CLK<= NOT CLK after 1 ns;
  ----------------------------------------------------
end Structure;
 ----------------------------------------------------
library IEEE;
  use IEEE.STD_LOGIC_1164.all;
  use IEEE.STD_LOGIC_ARITH.all;
library SYNOPSYS;
  use SYNOPSYS.attributes.all;
library WORK;
  use WORK.controleur_10pkg.all;
  use WORK.user.all;
 ----------------------------------------------------
-- synopsys_synthesis_off --
configuration TB_moteur_Structure of TB_moteur is
for Structure
  for Imoteur: moteur
    use configuration WORK.moteur_Structure;
  end for;
 end for;
end TB_moteur_Structure;
-- synopsys_synthesis_on --
----------------------------------------------------
```

7.1.5 Description VHDL comportementale du module Solar matériel : contrôleur (*control*)

La description comportementale du module Solar contrôleur de vitesse (DESIGNUNIT *control*) de l'exemple de contrôle de moteur, est traduite en VHDL comportemental.

```
----------------------------------------------------
library IEEE;
 use IEEE.STD_LOGIC_1164.all;
 use IEEE.STD_LOGIC_ARITH.all;
library SYNOPSYS;
 use SYNOPSYS.attributes.all;
library WORK;
 use WORK.controleur_10pkg.all;
 use WORK.user.all;
----------------------------------------------------
entity control is
  Generic
  (
    IPCKEY: INTEGER:= 1
  );
  Port (
    CLK: IN BIT;
    RST: IN BIT;
    software_channel_wr_req: OUT BIT;
    software_channel_rewr_rdy: IN BIT;
    software_channel_data_int: INOUT RESOLVED_INTEGER;
    software_channel_data_real: INOUT RESOLVED_REAL;
    algo1_channel_wr_req: OUT BIT;
    algo1_channel_rewr_rdy: IN BIT;
    algo1_channel_data_int: INOUT RESOLVED_INTEGER;
    algo1_channel_data_real: INOUT RESOLVED_REAL;
    control_channel_rewr_rdy: IN BIT;
    control_channel_read_req: OUT BIT;
    control_channel_data_int: INOUT RESOLVED_INTEGER;
    control_channel_data_real: INOUT RESOLVED_REAL;
    algo2_channel_wr_req: OUT BIT;
    algo2_channel_rewr_rdy: IN BIT;
    algo2_channel_data_int: INOUT RESOLVED_INTEGER;
    algo2_channel_data_real: INOUT RESOLVED_REAL
  );
end control;
----------------------------------------------------
library IEEE;
 use IEEE.STD_LOGIC_1164.all;
 use IEEE.STD_LOGIC_ARITH.all;
library SYNOPSYS;
 use SYNOPSYS.attributes.all;
library WORK;
 use WORK.controleur_10pkg.all;
 use WORK.user.all;
----------------------------------------------------
architecture control_behaviour of control is
 ----------------------------------------------------
begin
 ----------------------------------------------------
 ----------------------------------------------------
 process
   constant pc_to_pid: INTEGER:= 2;
   constant moteur2_to_pid: INTEGER:= 5;
   constant moteur1_to_pid: INTEGER:= 8;
   constant pid_to_pc: INTEGER:= 1;
   constant pid_to_moteur2_read: INTEGER:= 3;
   constant pid_to_moteur2_data: INTEGER:= 4;
   constant pid_to_moteur1_read: INTEGER:= 6;
   constant pid_to_moteur1_data: INTEGER:= 7;
   constant self: INTEGER:= 4;
   variable sdl_signal: INTEGER;
   variable sdl_to: INTEGER;
   variable sdl_sender: INTEGER;
   variable memoire: memory;
   variable u1: REAL;
   variable u2: REAL;
   variable u3: REAL;
   variable u4: REAL;
   variable u5: REAL;
   variable u6: REAL;
   variable u7: REAL;
   variable u8: REAL;
   variable u9: REAL;
   variable u10: REAL;
   variable u11: REAL;
   variable u12: REAL;
   variable u13: REAL;
   variable u14: REAL;
   variable temp: REAL;
   variable PCALL: INTEGER:= 1;
   ----------------------------------------------------
   procedure software_channel_put_signal(sdl_signal: IN INTEGER );
   procedure software_channel_put_signal(sdl_signal: IN INTEGER ) is
    ----------------------------------------------------
    type put_signal_StateType is (put_signalIdle, request, requestwait);
    variable put_signal_NextState: put_signal_StateType:= request;
   begin
    ----------------------------------------------------
    PCALL:= 1;
    while (PCALL = 1) loop
      ----------------------------------------------------
      StateTable_put_signal: loop
        case put_signal_NextState is
          ----------------------------------------------------
          when(request) =>
            if (software_channel_rewr_rdy = '0') then
              software_channel_data_int<= sdl_signal;
              software_channel_data_real<= REAL_LOW;
              software_channel_wr_req<= '1';
              put_signal_NextState:= requestwait;
              exit StateTable_put_signal;
            end if;
            put_signal_NextState:= request;
            exit StateTable_put_signal;
          ----------------------------------------------------
          when(requestwait) =>
            if NOT(software_channel_rewr_rdy = '1') then
              wait until ((software_channel_rewr_rdy = '1'));
            end if;
            software_channel_data_int<= INTEGER_LOW;
            software_channel_data_real<= REAL_LOW;
            software_channel_wr_req<= '0';
          ----------------------------------------------------
            put_signal_NextState:= put_signalIdle;
          ----------------------------------------------------
          when OTHERS =>
            PCALL:= 0;
            put_signal_NextState:= request;
          ----------------------------------------------------
        end case;
        exit StateTable_put_signal;
      end loop StateTable_put_signal;
      ----------------------------------------------------
      if (PCALL=1) then
        wait until (rising_edge(CLK));
      end if;
    end loop;
   end software_channel_put_signal;
   ----------------------------------------------------
   procedure algo1_channel_put_real(sdl_signal: IN INTEGER; param_1: IN REAL );
   procedure algo1_channel_put_real(sdl_signal: IN INTEGER; param_1: IN REAL ) is
    ----------------------------------------------------
    type put_real_StateType is (put_realIdle, request, requestwait, putparam, paramwait);
    variable put_real_NextState: put_real_StateType:= request;
   begin
    ----------------------------------------------------
    PCALL:= 1;
    while (PCALL = 1) loop
      ----------------------------------------------------
      StateTable_put_real: loop
        case put_real_NextState is
```

```
        ---------------------------------------------------
        when(request) =>
          if (algo1_channel_rewr_rdy = '0') then
            algo1_channel_data_int<= sdl_signal;
            algo1_channel_data_real<= REAL_LOW;
            algo1_channel_wr_req<= '1';
            put_real_NextState:= requestwait;
            exit StateTable_put_real;
          end if;
          put_real_NextState:= request;
          exit StateTable_put_real;
        ---------------------------------------------------
        when(requestwait) =>
          if NOT(algo1_channel_rewr_rdy = '1') then
            wait until ((algo1_channel_rewr_rdy = '1'));
          end if;
          algo1_channel_wr_req<= '0';
          algo1_channel_data_int<= INTEGER_LOW;
          algo1_channel_data_real<= REAL_LOW;
          put_real_NextState:= putparam;
          exit StateTable_put_real;
        ---------------------------------------------------
        when(putparam) =>
          algo1_channel_data_int<= INTEGER_LOW;
          algo1_channel_data_real<= param_1;
          algo1_channel_wr_req<= '1';
          put_real_NextState:= paramwait;
          exit StateTable_put_real;
        ---------------------------------------------------
        when(paramwait) =>
          if NOT(algo1_channel_rewr_rdy = '1') then
            wait until ((algo1_channel_rewr_rdy = '1'));
          end if;
          algo1_channel_data_real<= REAL_LOW;
          algo1_channel_data_int<= INTEGER_LOW;
          algo1_channel_wr_req<= '0';
        ---------------------------------------------------
          put_real_NextState:= put_realIdle;
        ---------------------------------------------------
        when OTHERS =>
          PCALL:= 0;
          put_real_NextState:= request;
        ---------------------------------------------------
      end case;
      exit StateTable_put_real;
    end loop StateTable_put_real;
    -----------------------------------------------------
    if (PCALL=1) then
      wait until (rising_edge(CLK));
    end if;
  end loop;
end algo1_channel_put_real;
-----------------------------------------------------
procedure algo1_channel_put_signal(sdl_signal: IN INTEGER );
procedure algo1_channel_put_signal(sdl_signal: IN INTEGER ) is
  -----------------------------------------------------
  type put_signal_StateType is (put_signalIdle, request, requestwait);
  variable put_signal_NextState: put_signal_StateType:= request;
begin
  -----------------------------------------------------
  PCALL:= 1;
  while (PCALL = 1) loop
    -----------------------------------------------------
    StateTable_put_signal: loop
      case put_signal_NextState is
        ---------------------------------------------------
        when(request) =>
          if (algo1_channel_rewr_rdy = '0') then
            algo1_channel_data_int<= sdl_signal;
            algo1_channel_data_real<= REAL_LOW;
            algo1_channel_wr_req<= '1';
            put_signal_NextState:= requestwait;
            exit StateTable_put_signal;
          end if;
          put_signal_NextState:= request;
          exit StateTable_put_signal;
        ---------------------------------------------------
        when(requestwait) =>
          if NOT(algo1_channel_rewr_rdy = '1') then
            wait until ((algo1_channel_rewr_rdy = '1'));
          end if;
          algo1_channel_data_int<= INTEGER_LOW;
          algo1_channel_data_real<= REAL_LOW;
          algo1_channel_wr_req<= '0';
        ---------------------------------------------------
          put_signal_NextState:= put_signalIdle;
        ---------------------------------------------------
        when OTHERS =>
          PCALL:= 0;
          put_signal_NextState:= request;
        ---------------------------------------------------
      end case;
      exit StateTable_put_signal;
    end loop StateTable_put_signal;
    -----------------------------------------------------
    if (PCALL=1) then
      wait until (rising_edge(CLK));
    end if;
  end loop;
end algo1_channel_put_signal;
-----------------------------------------------------
procedure control_channel_get_signal(sdl_signal: INOUT INTEGER );
procedure control_channel_get_signal(sdl_signal: INOUT INTEGER ) is
  -----------------------------------------------------
  type get_signal_StateType is (get_signalIdle, request, requestwait);
  variable get_signal_NextState: get_signal_StateType:= request;
begin
  -----------------------------------------------------
  PCALL:= 1;
  while (PCALL = 1) loop
    -----------------------------------------------------
    StateTable_get_signal: loop
      case get_signal_NextState is
        ---------------------------------------------------
        when(request) =>
          if (control_channel_rewr_rdy = '0') then
            control_channel_read_req<= '1';
            control_channel_data_int<= INTEGER_LOW;
            control_channel_data_real<= REAL_LOW;
            get_signal_NextState:= requestwait;
            exit StateTable_get_signal;
          end if;
          get_signal_NextState:= request;
          exit StateTable_get_signal;
        ---------------------------------------------------
        when(requestwait) =>
          if NOT(control_channel_rewr_rdy = '1') then
            wait until ((control_channel_rewr_rdy = '1'));
          end if;
          sdl_signal:= control_channel_data_int;
          control_channel_data_int<= INTEGER_LOW;
          control_channel_data_real<= REAL_LOW;
          control_channel_read_req<= '0';
        ---------------------------------------------------
          get_signal_NextState:= get_signalIdle;
        ---------------------------------------------------
        when OTHERS =>
          PCALL:= 0;
          get_signal_NextState:= request;
        ---------------------------------------------------
      end case;
      exit StateTable_get_signal;
    end loop StateTable_get_signal;
    -----------------------------------------------------
    if (PCALL=1) then
      wait until (rising_edge(CLK));
    end if;
  end loop;
end control_channel_get_signal;
-----------------------------------------------------
procedure control_channel_get_real(param_1: INOUT REAL );
procedure control_channel_get_real(param_1: INOUT REAL ) is
  -----------------------------------------------------
  type get_real_StateType is (get_realIdle, request, waitrdy);
  variable get_real_NextState: get_real_StateType:= request;
begin
  -----------------------------------------------------
  PCALL:= 1;
  while (PCALL = 1) loop
    -----------------------------------------------------
    StateTable_get_real: loop
      case get_real_NextState is
        ---------------------------------------------------
        when(request) =>
          if (control_channel_rewr_rdy = '0') then
            control_channel_read_req<= '1';
            control_channel_data_real<= REAL_LOW;
            control_channel_data_int<= INTEGER_LOW;
            get_real_NextState:= waitrdy;
            exit StateTable_get_real;
          end if;
          get_real_NextState:= request;
          exit StateTable_get_real;
        ---------------------------------------------------
        when(waitrdy) =>
          if NOT(control_channel_rewr_rdy = '1') then
            wait until ((control_channel_rewr_rdy = '1'));
          end if;
          param_1:= control_channel_data_real;
          control_channel_data_real<= REAL_LOW;
          control_channel_data_int<= INTEGER_LOW;
          control_channel_read_req<= '0';
        ---------------------------------------------------
          get_real_NextState:= get_realIdle;
        ---------------------------------------------------
        when OTHERS =>
          PCALL:= 0;
          get_real_NextState:= request;
        ---------------------------------------------------
      end case;
      exit StateTable_get_real;
    end loop StateTable_get_real;
    -----------------------------------------------------
    if (PCALL=1) then
      wait until (rising_edge(CLK));
    end if;
  end loop;
end control_channel_get_real;
-----------------------------------------------------
procedure algo2_channel_put_real(sdl_signal: IN INTEGER; param_1: IN REAL );
procedure algo2_channel_put_real(sdl_signal: IN INTEGER; param_1: IN REAL ) is
  -----------------------------------------------------
  type put_real_StateType is (put_realIdle, request, requestwait, putparam, paramwait);
  variable put_real_NextState: put_real_StateType:= request;
begin
  -----------------------------------------------------
  PCALL:= 1;
  while (PCALL = 1) loop
    -----------------------------------------------------
    StateTable_put_real: loop
      case put_real_NextState is
        ---------------------------------------------------
        when(request) =>
          if (algo2_channel_rewr_rdy = '0') then
            algo2_channel_data_int<= sdl_signal;
            algo2_channel_data_real<= REAL_LOW;
            algo2_channel_wr_req<= '1';
            put_real_NextState:= requestwait;
            exit StateTable_put_real;
          end if;
          put_real_NextState:= request;
          exit StateTable_put_real;
        ---------------------------------------------------
        when(requestwait) =>
          if NOT(algo2_channel_rewr_rdy = '1') then
            wait until ((algo2_channel_rewr_rdy = '1'));
          end if;
          algo2_channel_wr_req<= '0';
          algo2_channel_data_int<= INTEGER_LOW;
          algo2_channel_data_real<= REAL_LOW;
          put_real_NextState:= putparam;
          exit StateTable_put_real;
        ---------------------------------------------------
        when(putparam) =>
          algo2_channel_data_int<= INTEGER_LOW;
          algo2_channel_data_real<= param_1;
          algo2_channel_wr_req<= '1';
          put_real_NextState:= paramwait;
```

```
          exit StateTable_put_real;
        ------------------------------------------------------------
        when(paramwait) =>
          if NOT(algo2_channel_rewr_rdy = '1') then
            wait until ((algo2_channel_rewr_rdy = '1'));
          end if;
          algo2_channel_data_real<= REAL_LOW;
          algo2_channel_data_int<= INTEGER_LOW;
          algo2_channel_wr_req<= '0';
        ------------------------------------------------------------
          put_real_NextState:= put_realIdle;
        ------------------------------------------------------------
        when OTHERS =>
          PCALL:= 0;
          put_real_NextState:= request;
        ------------------------------------------------------------
      end case;
      exit StateTable_put_real;
    end loop StateTable_put_real;
    ------------------------------------------------------------
    if (PCALL=1) then
      wait until (rising_edge(CLK));
    end if;
  end loop;
end algo2_channel_put_real;
------------------------------------------------------------
procedure algo2_channel_put_signal(sdl_signal: IN INTEGER );
procedure algo2_channel_put_signal(sdl_signal: IN INTEGER ) is
  ------------------------------------------------------------
  type put_signal_StateType is (put_signalIdle, request, requestwait);
  variable put_signal_NextState: put_signal_StateType:= request;
begin
  ------------------------------------------------------------
  PCALL:= 1;
  while (PCALL = 1) loop
    ------------------------------------------------------------
    StateTable_put_signal: loop
      case put_signal_NextState is
        ------------------------------------------------------------
        when(request) =>
          if (algo2_channel_rewr_rdy = '0') then
            algo2_channel_data_int<= sdl_signal;
            algo2_channel_data_real<= REAL_LOW;
            algo2_channel_wr_req<= '1';
            put_signal_NextState:= requestwait;
            exit StateTable_put_signal;
          end if;
          put_signal_NextState:= request;
          exit StateTable_put_signal;
        ------------------------------------------------------------
        when(requestwait) =>
          if NOT(algo2_channel_rewr_rdy = '1') then
            wait until ((algo2_channel_rewr_rdy = '1'));
          end if;
          algo2_channel_data_int<= INTEGER_LOW;
          algo2_channel_data_real<= REAL_LOW;
          algo2_channel_wr_req<= '0';
        ------------------------------------------------------------
          put_signal_NextState:= put_signalIdle;
        ------------------------------------------------------------
        when OTHERS =>
          PCALL:= 0;
          put_signal_NextState:= request;
        ------------------------------------------------------------
      end case;
      exit StateTable_put_signal;
    end loop StateTable_put_signal;
    ------------------------------------------------------------
    if (PCALL=1) then
      wait until (rising_edge(CLK));
    end if;
  end loop;
end algo2_channel_put_signal;
------------------------------------------------------------
------------------------------------------------------------
type statetable_control_StateType is (initial, attente_donnee_pc_1, attente_donnee_pc_2,
attente_donnee_pc_3, attente_donnee_pc_4, attente_donnee_pc_5, attente_donnee_pc_6,
attente_donnee_pc_7, attente_donnee_pc_8, attente_donnee_pc_9, attente_donnee_pc_10,
attente_donnee_pc_11, attente_donnee_pc_12, attente_donnee_pc_13,
attente_donnee_pc_14, calcul1, calcul2);
variable statetable_control_NextState: statetable_control_StateType:= initial;
------------------------------------------------------------
begin
  ------------------------------------------------------------
  wait until (rising_edge(CLK)) OR (RST='1');
  if (RST='1') then
    statetable_control_NextState:= initial;
    wait until (rising_edge(CLK));
  end if;
  ------------------------------------------------------------
  StateTable_statetable_control: loop
    case statetable_control_NextState is
      ------------------------------------------------------------
      when(initial) =>
        algo1_channel_put_real
        (
          sdl_signal=> 7,
          param_1=> 0.000000
        ); --PCall
        algo2_channel_put_real
        (
          sdl_signal=> 4,
          param_1=> 0.000000
        ); --PCall
        statetable_control_NextState:= attente_donnee_pc_1;
        exit StateTable_statetable_control;
      ------------------------------------------------------------
      when(attente_donnee_pc_1) =>
        control_channel_get_signal
        (
          sdl_signal=> sdl_signal
        ); --PCall
        if (sdl_signal = 2) then
          control_channel_get_real
          (
            param_1=> temp
          ); --PCall
          memoire(1):= temp;
          software_channel_put_signal
          (
            sdl_signal=> 1
          ); --PCall
          statetable_control_NextState:= attente_donnee_pc_2;
          exit StateTable_statetable_control;
        end if;
      ------------------------------------------------------------
      when(attente_donnee_pc_2) =>
        control_channel_get_signal
        (
          sdl_signal=> sdl_signal
        ); --PCall
        if (sdl_signal = 2) then
          control_channel_get_real
          (
            param_1=> temp
          ); --PCall
          memoire(2):= temp;
          software_channel_put_signal
          (
            sdl_signal=> 1
          ); --PCall
          statetable_control_NextState:= attente_donnee_pc_3;
          exit StateTable_statetable_control;
        end if;
      ------------------------------------------------------------
      when(attente_donnee_pc_3) =>
        control_channel_get_signal
        (
          sdl_signal=> sdl_signal
        ); --PCall
        if (sdl_signal = 2) then
          control_channel_get_real
          (
            param_1=> temp
          ); --PCall
          memoire(3):= temp;
          software_channel_put_signal
          (
            sdl_signal=> 1
          ); --PCall
          statetable_control_NextState:= attente_donnee_pc_4;
          exit StateTable_statetable_control;
        end if;
      ------------------------------------------------------------
      when(attente_donnee_pc_4) =>
        control_channel_get_signal
        (
          sdl_signal=> sdl_signal
        ); --PCall
        if (sdl_signal = 2) then
          control_channel_get_real
          (
            param_1=> temp
          ); --PCall
          memoire(4):= temp;
          software_channel_put_signal
          (
            sdl_signal=> 1
          ); --PCall
          statetable_control_NextState:= attente_donnee_pc_5;
          exit StateTable_statetable_control;
        end if;
      ------------------------------------------------------------
      when(attente_donnee_pc_5) =>
        control_channel_get_signal
        (
          sdl_signal=> sdl_signal
        ); --PCall
        if (sdl_signal = 2) then
          control_channel_get_real
          (
            param_1=> temp
          ); --PCall
          memoire(5):= temp;
          software_channel_put_signal
          (
            sdl_signal=> 1
          ); --PCall
          statetable_control_NextState:= attente_donnee_pc_6;
          exit StateTable_statetable_control;
        end if;
      ------------------------------------------------------------
      when(attente_donnee_pc_6) =>
        control_channel_get_signal
        (
          sdl_signal=> sdl_signal
        ); --PCall
        if (sdl_signal = 2) then
          control_channel_get_real
          (
            param_1=> temp
          ); --PCall
          memoire(6):= temp;
          software_channel_put_signal
          (
            sdl_signal=> 1
          ); --PCall
          statetable_control_NextState:= attente_donnee_pc_7;
          exit StateTable_statetable_control;
        end if;
      ------------------------------------------------------------
      when(attente_donnee_pc_7) =>
        control_channel_get_signal
        (
          sdl_signal=> sdl_signal
        ); --PCall
        if (sdl_signal = 2) then
          control_channel_get_real
          (
            param_1=> temp
          ); --PCall
          memoire(7):= temp;
          software_channel_put_signal
          (
            sdl_signal=> 1
          ); --PCall
          statetable_control_NextState:= attente_donnee_pc_8;
          exit StateTable_statetable_control;
```

```
  end if;
-----------------------------------------------------------
when(attente_donnee_pc_8) =>
  control_channel_get_signal
  (
    sdl_signal=> sdl_signal
  ); --PCall
  if (sdl_signal = 2) then
    control_channel_get_real
    (
      param_1=> temp
    ); --PCall
    memoire(8):= temp;
    software_channel_put_signal
    (
      sdl_signal=> 1
    ); --PCall
    statetable_control_NextState:= attente_donnee_pc_9;
    exit StateTable_statetable_control;
  end if;
-----------------------------------------------------------
when(attente_donnee_pc_9) =>
  control_channel_get_signal
  (
    sdl_signal=> sdl_signal
  ); --PCall
  if (sdl_signal = 2) then
    control_channel_get_real
    (
      param_1=> temp
    ); --PCall
    memoire(9):= temp;
    software_channel_put_signal
    (
      sdl_signal=> 1
    ); --PCall
    statetable_control_NextState:= attente_donnee_pc_10;
    exit StateTable_statetable_control;
  end if;
-----------------------------------------------------------
when(attente_donnee_pc_10) =>
  control_channel_get_signal
  (
    sdl_signal=> sdl_signal
  ); --PCall
  if (sdl_signal = 2) then
    control_channel_get_real
    (
      param_1=> temp
    ); --PCall
    memoire(10):= temp;
    software_channel_put_signal
    (
      sdl_signal=> 1
    ); --PCall
    statetable_control_NextState:= attente_donnee_pc_11;
    exit StateTable_statetable_control;
  end if;
-----------------------------------------------------------
when(attente_donnee_pc_11) =>
  control_channel_get_signal
  (
    sdl_signal=> sdl_signal
  ); --PCall
  if (sdl_signal = 2) then
    control_channel_get_real
    (
      param_1=> temp
    ); --PCall
    memoire(11):= temp;
    software_channel_put_signal
    (
      sdl_signal=> 1
    ); --PCall
    statetable_control_NextState:= attente_donnee_pc_12;
    exit StateTable_statetable_control;
  end if;
-----------------------------------------------------------
when(attente_donnee_pc_12) =>
  control_channel_get_signal
  (
    sdl_signal=> sdl_signal
  ); --PCall
  if (sdl_signal = 2) then
    control_channel_get_real
    (
      param_1=> temp
    ); --PCall
    memoire(12):= temp;
    software_channel_put_signal
    (
      sdl_signal=> 1
    ); --PCall
    statetable_control_NextState:= attente_donnee_pc_13;
    exit StateTable_statetable_control;
  end if;
-----------------------------------------------------------
when(attente_donnee_pc_13) =>
  control_channel_get_signal
  (
    sdl_signal=> sdl_signal
  ); --PCall
  if (sdl_signal = 2) then
    control_channel_get_real
    (
      param_1=> temp
    ); --PCall
    memoire(13):= temp;
    software_channel_put_signal
    (
      sdl_signal=> 1
    ); --PCall
    statetable_control_NextState:= attente_donnee_pc_14;
    exit StateTable_statetable_control;
  end if;
-----------------------------------------------------------
when(attente_donnee_pc_14) =>
  control_channel_get_signal
  (
    sdl_signal=> sdl_signal
  ); --PCall
  if (sdl_signal = 2) then
    control_channel_get_real
    (
      param_1=> temp
    ); --PCall
    memoire(14):= temp;
    algo1_channel_put_signal
    (
      sdl_signal=> 6
    ); --PCall
    statetable_control_NextState:= calcul1;
    exit StateTable_statetable_control;
  end if;
-----------------------------------------------------------
when(calcul1) =>
  control_channel_get_signal
  (
    sdl_signal=> sdl_signal
  ); --PCall
  if (sdl_signal = 8) then
    control_channel_get_real
    (
      param_1=> u2
    ); --PCall
    u1:= memoire(1);
    u3:= memoire(2);
    u4:= memoire(3);
    u5:= memoire(4);
    u6:= memoire(5);
    u7:= memoire(6);
    u8:= memoire(7);
    u9:= (u1 - u2);
    u10:= (u9 * u6);
    u11:= (u4 * u7);
    u12:= (u3 - u11);
    u13:= (u5 * u8);
    u14:= ((u10 + u12) + u13);
    algo1_channel_put_real
    (
      sdl_signal=> 7,
      param_1=> u14
    ); --PCall
    memoire(4):= u4;
    memoire(3):= u9;
    memoire(2):= u14;
    algo2_channel_put_signal
    (
      sdl_signal=> 3
    ); --PCall
    statetable_control_NextState:= calcul2;
    exit StateTable_statetable_control;
  end if;
-----------------------------------------------------------
when(calcul2) =>
  control_channel_get_signal
  (
    sdl_signal=> sdl_signal
  ); --PCall
  if (sdl_signal = 5) then
    control_channel_get_real
    (
      param_1=> u2
    ); --PCall
    u1:= memoire(8);
    u3:= memoire(9);
    u4:= memoire(10);
    u5:= memoire(11);
    u6:= memoire(12);
    u7:= memoire(13);
    u8:= memoire(14);
    u9:= (u1 - u2);
    u10:= (u9 * u6);
    u11:= (u4 * u7);
    u12:= (u3 - u11);
    u13:= (u5 * u8);
    u14:= ((u10 + u12) + u13);
    algo2_channel_put_real
    (
      sdl_signal=> 4,
      param_1=> u14
    ); --PCall
    memoire(11):= u4;
    memoire(10):= u9;
    memoire(9):= u14;
    algo1_channel_put_signal
    (
      sdl_signal=> 6
    ); --PCall
    statetable_control_NextState:= calcul1;
    exit StateTable_statetable_control;
  end if;
-----------------------------------------------------------
end case;
exit StateTable_statetable_control;
end loop StateTable_statetable_control;
-----------------------------------------------------------
end process;
end control_behaviour;
-----------------------------------------------------------
-- synopsys_synthesis_off --
configuration control_control_behaviour of control is
 for control_behaviour
 end for;
end control_control_behaviour;
-- synopsys_synthesis_on --
-----------------------------------------------------------
```

7.1.6 Description du *package* VHDL pour les types introduits par l'unité de communication

Les types définis par l'utilisateur et les types vecteurs sont définis dans un *package* VHDL par S2CV.

```
--s2cv version: 3.02
-- File: controleur_10pkg.vhd
-- Solar file: controleur_10.solar

 -----------------------------------------------------
package controleur_10pkg is
   function rising_edge(signal CLK:bit) return BOOLEAN;
 type REAL_VECTOR is array (NATURAL range <>) of REAL;
 subtype memory is REAL_VECTOR(14 downto 1);
 subtype mindexint is natural;
   type INTEGER_VECTOR is array (NATURAL range <>) of INTEGER;
end controleur_10pkg;
-----------------------------------------------------

package body controleur_10pkg is
 -----------------------------------------------------
 function rising_edge(signal CLK:bit) return BOOLEAN is
 begin
   return (CLK'event and CLK='1' and CLK'last_value='0');
 end rising_edge;
 -----------------------------------------------------
end controleur_10pkg;
-----------------------------------------------------
```

7.1.7 Description C du corps du module logiciel : *moteur1* (*algo1*)

La description Solar pour un des moteurs (DESIGNUNIT algo2) est traduite en C par S2CV en trois niveaux : instance, corps et primitives de communication.

```
/*
                H-file: algo1.h
                Solar2C-VHDL(3.02): SOLAR-COSMOS SLS/TIMA/INPG
                Report problems to: valderr@verdon.imag.fr
*/

#ifndef _algo1_H_

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
/*DU(algo1)*/
/*******************************************************/
/*IncludesDU*/
#include "algo1itf.h"
/*******************************************************/
extern int PCALL;
extern int IPCKEY;
 /*TypesDU*/
extern void oneCycleStep();
extern void actualizeOuput();
extern int actualizeInput();
extern int  algo1();
   extern int algo1_channel_get_signal( /*INTEGER* sdl_signal */);
/*declareProcedureObject_CC*/
   extern int algo1_channel_get_real( /*REAL* param_1 */); /*declareProcedureObject_CC*/
   extern int control_channel_put_real( /*INTEGER* sdl_signal,REAL* param_1 */);
/*declareProcedureObject_CC*/

#define _algo1_H_
#endif /*algo1.h*/


/*
                C-file: algo1.c
                Solar2C-VHDL(3.02): SOLAR-COSMOS SLS/TIMA/INPG
                Report problems to: valderr@verdon.imag.fr
*/

/*DU(algo1)*/
/*******************************************************/
/*IncludesDU*/
#include "algo1.h"

int PCALL= 1 ;
/*ViewType("behaviour")*/
 /*InterfaceProcessDU(algo1itf.vci)*/
 /*Iface*/
int algo1()
{
                FILE* fichier;
   /*View*/
   INTEGER pid_to_moteur1_read=  6 ; /*CONSTANT*/
   INTEGER pid_to_moteur1_data=  7 ; /*CONSTANT*/
   INTEGER moteur1_to_pid=  8 ; /*CONSTANT*/
   INTEGER self=  3 ; /*CONSTANT*/
   INTEGER sdl_signal; /*VARIABLE*/
   INTEGER sdl_to; /*VARIABLE*/
   INTEGER sdl_sender; /*VARIABLE*/
   REAL s_0; /*VARIABLE*/
   REAL s_1; /*VARIABLE*/
   REAL e_0; /*VARIABLE*/
   REAL e_1; /*VARIABLE*/
   INTEGER sdl_signal_131; /*PCallParamVar*/
   REAL param_1_132; /*PCallParamVar*/
   INTEGER sdl_signal_88; /*PCallParamVar*/
   REAL param_1_88; /*PCallParamVar*/
   /*BeginContents(algo1_behaviouralgo1)*/
   /*******************************************************/
   /*DeclareST*/
   typedef enum
   {
     idle_statetable_algo1,
     initial_136State,
     attente1_136State,
     attente2_136State
   } statetable_algo1_StateType;
   statetable_algo1_StateType statetable_algo1_NextState= initial_136State;
   statetable_algo1_StateType statetable_algo1InitialState= initial_136State;
   /*******************************************************/
   int AGAIN= 1;
   while (AGAIN==1)
   {
     if (AGAIN==1)
     { /*OneCycleByStep*/
       oneCycleStep();
     }
     switch(statetable_algo1_NextState) /*defineST*/
     {
       case initial_136State :  /*CaseState*/
       {
         e_1=  0.000000 /*RealvalObject_CC*/;
         s_0=  0.000000 /*RealvalObject_CC*/;
         s_1=  0.000000 /*RealvalObject_CC*/;
         statetable_algo1_NextState= attente1_136State;
         break; /*GoToNextState*/
       } /*EndOfCaseState*/
       case attente1_136State :  /*CaseState*/
       {
         algo1_channel_get_signal(&(sdl_signal)); /*PCall*/
         /*SwitchCase*/
         if ((sdl_signal)==( 7 )) /*IfAlt*/
         {
           algo1_channel_get_real(&(e_0)); /*PCall*/
           s_0= (s_1-( 0.005000 /*RealvalObject_CC*/*(s_1-e_1)));
           s_1= s_0;
           e_1= e_0;
           fichier=fopen("moteur1.dat","a");
                                                                    fprintf(fichier,"%f\n",s_0);
                                                                    fclose(fichier);
           statetable_algo1_NextState= attente2_136State;
           break; /*GoToNextState*/
         }
         else
         ; /*Nothing*/
       } /*EndOfCaseState*/
       case attente2_136State :  /*CaseState*/
       {
         algo1_channel_get_signal(&(sdl_signal)); /*PCall*/
         /*SwitchCase*/
         if ((sdl_signal)==( 6 )) /*IfAlt*/
         {
           control_channel_put_real(&(moteur1_to_pid),&(s_0)); /*PCall*/
           statetable_algo1_NextState= attente1_136State;
           break; /*GoToNextState*/
         }
         else
         ; /*Nothing*/
       } /*EndOfCaseState*/
     }
   }
   return AGAIN;
   /*EndOfContents*/
}
/*******************************************************/
/*CloseCCFile(algo1.c)*/
```

7.1.8 Description C des procédures de communication utilisées par le module logiciel : *moteur1* (*algo1*)

Les procédures de communication, décrites en Solar dans la bibliothèque de communication, sont traduites en C par S2CV.

```
/*
                Functional Unit C-file: algo1fu.c
                Solar2C-VHDL(3.02): SOLAR-COSMOS SLS/TIMA/INPG
                Report problems to: valderr@verdon.imag.fr
*/

#include "algo1.h"


void oneCycleStep()
{
  if (BitAddr(0))
  { /*ActualizePortValues*/
    actualizeOuput();
  }
  else
  {  /*JustReceive*/
    actualizeInput();
  }
}

void actualizeOuput()
{
  algo1itfSendOUT(IPCKEY);
}

int actualizeInput()
{
  algo1itfReceiveIN(IPCKEY);
  return 0;
}
  /*****************************************************/
int algo1_channel_get_signal( sdl_signal)
INTEGER* sdl_signal;
{
    /*****************************************************/
    /*DeclareST*/
    typedef enum
    {
      idle_get_signal,
      request_83State,
      requestwait_83State
    } get_signal_StateType;
    get_signal_StateType get_signal_NextState= request_83State;
    get_signal_StateType get_signalInitialState= request_83State;
    /*****************************************************/
    PCALL= 1;
    while (PCALL==1)
    { /*ProcedureBody*/
      switch(get_signal_NextState) /*defineST*/
      {
        case request_83State :  /*CaseState*/
        {
          if ((algo1_channel_rewr_rdy)==(BIT_0)) /*If*/
          { /*Then*/
            MapAddr(mtypealgo1_channel_read_req);
            algo1_channel_read_req= BIT_1;
            MapAddr(mtypealgo1_channel_data_int);
            algo1_channel_data_int= INTEGER_LOW;
            MapAddr(mtypealgo1_channel_data_real);
            algo1_channel_data_real= REAL_LOW;
            get_signal_NextState= requestwait_83State;
            break; /*GoToNextState*/
          }
          get_signal_NextState= request_83State;
          break; /*GoToNextState*/
        } /*EndOfCaseState*/
        case requestwait_83State :  /*CaseState*/
        {
          /*WaitObject_CC*/
          if (BitAddr(0))
          { /*ActualizePortValues*/
            actualizeOuput();
          }
          while(!(algo1_channel_rewr_rdy)==(BIT_1))
          { /*ReceivePortValues*/
            actualizeInput();
          }
          * sdl_signal= algo1_channel_data_int;
          MapAddr(mtypealgo1_channel_data_int);
          algo1_channel_data_int= INTEGER_LOW;
          MapAddr(mtypealgo1_channel_data_real);
          algo1_channel_data_real= REAL_LOW;
          MapAddr(mtypealgo1_channel_read_req);
          algo1_channel_read_req= BIT_0;
        /*NoBreakOnProc*/
          get_signal_NextState= idle_get_signal;
          break;
        }
        /*****************************************************/
        default : /*DefaultState*/
        { PCALL= 0;
          get_signal_NextState= get_signalInitialState;
        }
        /*****************************************************/
      }
      if (PCALL==1)
      {
        oneCycleStep();
      } /* if (PCALL==1) */
    } /* end while (PCALL==1) */
    return PCALL;
} /*EndOfProcedure algo1_channel_get_signal*/

  /*****************************************************/
int algo1_channel_get_real( param_1)
REAL* param_1;
{
    /*****************************************************/
    /*DeclareST*/
    typedef enum
    {
      idle_get_real,
      request_117State,
      waitrdy_117State
    } get_real_StateType;
    get_real_StateType get_real_NextState= request_117State;
    get_real_StateType get_realInitialState= request_117State;
    /*****************************************************/
    PCALL= 1;
    while (PCALL==1)
    { /*ProcedureBody*/
      switch(get_real_NextState) /*defineST*/
      {
        case request_117State :  /*CaseState*/
        {
          if ((algo1_channel_rewr_rdy)==(BIT_0)) /*If*/
          { /*Then*/
            MapAddr(mtypealgo1_channel_read_req);
            algo1_channel_read_req= BIT_1;
            MapAddr(mtypealgo1_channel_data_real);
            algo1_channel_data_real= REAL_LOW;
            MapAddr(mtypealgo1_channel_data_int);
            algo1_channel_data_int= INTEGER_LOW;
            get_real_NextState= waitrdy_117State;
            break; /*GoToNextState*/
          }
          get_real_NextState= request_117State;
          break; /*GoToNextState*/
        } /*EndOfCaseState*/
        case waitrdy_117State :  /*CaseState*/
        {
          /*WaitObject_CC*/
          if (BitAddr(0))
          { /*ActualizePortValues*/
            actualizeOuput();
          }
          while(!(algo1_channel_rewr_rdy)==(BIT_1))
          { /*ReceivePortValues*/
            actualizeInput();
          }
          * param_1= algo1_channel_data_real;
          MapAddr(mtypealgo1_channel_data_real);
          algo1_channel_data_real= REAL_LOW;
          MapAddr(mtypealgo1_channel_data_int);
          algo1_channel_data_int= INTEGER_LOW;
          MapAddr(mtypealgo1_channel_read_req);
          algo1_channel_read_req= BIT_0;
        /*NoBreakOnProc*/
          get_real_NextState= idle_get_real;
          break;
        }
        /*****************************************************/
        default : /*DefaultState*/
        { PCALL= 0;
          get_real_NextState= get_realInitialState;
        }
        /*****************************************************/
      }
      if (PCALL==1)
      {
        oneCycleStep();
      } /* if (PCALL==1) */
    } /* end while (PCALL==1) */
    return PCALL;
} /*EndOfProcedure algo1_channel_get_real*/

  /*****************************************************/
int control_channel_put_real( sdl_signal, param_1)
INTEGER* sdl_signal;
REAL* param_1;
{
    /*****************************************************/
    /*DeclareST*/
    typedef enum
    {
      idle_put_real,
      request_90State,
      requestwait_90State,
      putparam_90State,
      paramwait_90State
    } put_real_StateType;
    put_real_StateType put_real_NextState= request_90State;
    put_real_StateType put_realInitialState= request_90State;
    /*****************************************************/
    PCALL= 1;
    while (PCALL==1)
    { /*ProcedureBody*/
      switch(put_real_NextState) /*defineST*/
      {
        case request_90State :  /*CaseState*/
        {
          if ((control_channel_rewr_rdy)==(BIT_0)) /*If*/
          { /*Then*/
```

```
      MapAddr(mtypecontrol_channel_data_int);
      control_channel_data_int= * sdl_signal;
      MapAddr(mtypecontrol_channel_data_real);
      control_channel_data_real= REAL_LOW;
      MapAddr(mtypecontrol_channel_wr_req);
      control_channel_wr_req= BIT_1;
      put_real_NextState= requestwait_90State;
      break; /*GoToNextState*/
    }
    put_real_NextState= request_90State;
    break; /*GoToNextState*/
  } /*EndOfCaseState*/
  case requestwait_90State :  /*CaseState*/
  {
    /*WaitObject_CC*/
    if (BitAddr(0))
    { /*ActualizePortValues*/
      actualizeOuput();
    }
    while(!(control_channel_rewr_rdy)==(BIT_1))
    { /*ReceivePortValues*/
      actualizeInput();
    }
    MapAddr(mtypecontrol_channel_wr_req);
    control_channel_wr_req= BIT_0;
    MapAddr(mtypecontrol_channel_data_int);
    control_channel_data_int= INTEGER_LOW;
    MapAddr(mtypecontrol_channel_data_real);
    control_channel_data_real= REAL_LOW;
    put_real_NextState= putparam_90State;
    break; /*GoToNextState*/
  } /*EndOfCaseState*/
  case putparam_90State :  /*CaseState*/
  {
    MapAddr(mtypecontrol_channel_data_int);
    control_channel_data_int= INTEGER_LOW;
    MapAddr(mtypecontrol_channel_data_real);
    control_channel_data_real= * param_1;
    MapAddr(mtypecontrol_channel_wr_req);
    control_channel_wr_req= BIT_1;
    put_real_NextState= paramwait_90State;
    break; /*GoToNextState*/
  } /*EndOfCaseState*/
  case paramwait_90State :  /*CaseState*/
  {
    /*WaitObject_CC*/
    if (BitAddr(0))
    { /*ActualizePortValues*/
      actualizeOuput();
    }
    while(!(control_channel_rewr_rdy)==(BIT_1))
    { /*ReceivePortValues*/
      actualizeInput();
    }
    MapAddr(mtypecontrol_channel_data_real);
    control_channel_data_real= REAL_LOW;
    MapAddr(mtypecontrol_channel_data_int);
    control_channel_data_int= INTEGER_LOW;
    MapAddr(mtypecontrol_channel_wr_req);
    control_channel_wr_req= BIT_0;
  /*NoBreakOnProc*/
    put_real_NextState= idle_put_real;
    break;
  }
  /*****************************************************/
  default : /*DefaultState*/
  { PCALL= 0;
    put_real_NextState= put_realInitialState;
  }
  /*****************************************************/
 }
 if (PCALL==1)
 {
   oneCycleStep();
 } /* if (PCALL==1) */
} /* end while (PCALL==1) */
return PCALL;
} /*EndOfProcedure control_channel_put_real*/

/*CloseFFFile(algo1fu.c)*/
```

7.1.9 Description de l'interface du module logiciel pour la cosimulation (fichier VCI) : *moteur1* (*algo1*)

L'interface correspondant à la description Solar d'un des moteurs (DESIGNUNIT *algo2*) est générée pour la cosimulation (format VCI) dans un fichier indépendant.

```
! VCI-file: algo1itf.vci
! Solar2C-VHDL(3.02): SOLAR-COSMOS SLS/TIMA/INPG
! Report problems to: valderr@verdon.imag.fr
!
!PORT        DIR        TYPE{[RANGE]}        SENSITIVITY
!
CLK IN BIT  R
RST IN BIT  N
algo1_channel_rewr_rdy IN BIT N
algo1_channel_read_req OUT BIT N
algo1_channel_data_int INOUT RESOLVED_INTEGER N
algo1_channel_data_real INOUT RESOLVED_REAL N
control_channel_wr_req OUT BIT N
control_channel_rewr_rdy IN BIT N
control_channel_data_int INOUT RESOLVED_INTEGER N
control_channel_data_real INOUT RESOLVED_REAL N
```

7.1.10 Fichier C généré automatiquement par VCI pour la cosimulation avec le module logiciel

Ce fichier contient les procédures d'entrée/sortie, les procédures d'émulation VHDL-VEC (*VHDL emulation C procedures*) et de conversion des types VHDL-C utilisés pour la cosimulation VHDL-C.

```
/* H-Interface file: algo1itf.h */
/* VCI INTERFACE:
            VHDL-C language interface - Unix CLI Synopsys
            Author: Carlos Valderrama (CA.Valderrama@imag.fr)
            version 5.05 MSQ
 */

/*91AJyG6*/

#ifndef _VCI_algo1itf_H
#define _VCI_algo1itf_H

/*HIncludes*/

#include <stdio.h>

#ifdef CLILIB
#include "cli.h"
#include "clictyp.h"
#else
#include "generic.h"
#endif

/*ConvertionTypeToValue*/

#ifndef CLILIB   /*91AJAnM*/

/*H-File global variables*/

extern RESOLVED_REAL                    control_channel_data_real;
extern RESOLVED_INTEGER                 control_channel_data_int;
extern BIT                              control_channel_rewr_rdy;
extern BIT                              control_channel_wr_req;
extern RESOLVED_REAL                    algo1_channel_data_real;
extern RESOLVED_INTEGER                 algo1_channel_data_int;
extern BIT                              algo1_channel_read_req;
extern BIT                              algo1_channel_rewr_rdy;
extern BIT                              RST;
extern BIT                              CLK;
extern short int                        _send_map[11];

#define MaskAddr(mn_)          (1<<mn_-1)
#define _DIOmap(mn_)           _send_map[0]=_send_map[mn_]= 1
#define MapAddr(mn_)           _DIOmap(mn_)
#define BitAddr(mn_)           (_send_map[mn_]==1)
#define MapClr(mn_)            _send_map[mn_]= 0
#endif          /*91AJAnM*/

#define mtypecontrol_channel_data_real          10
#define trangecontrol_channel_data_real         0
#define mtypecontrol_channel_data_int           9
#define trangecontrol_channel_data_int          0
#define mtypecontrol_channel_rewr_rdy           8
#define trangecontrol_channel_rewr_rdy          0
#define mtypecontrol_channel_wr_req             7
#define trangecontrol_channel_wr_req            0
#define mtypealgo1_channel_data_real            6
#define trangealgo1_channel_data_real           0
#define mtypealgo1_channel_data_int             5
#define trangealgo1_channel_data_int            0
```

```
#define mtypealgo1_channel_read_req          4
#define trangealgo1_channel_read_req         0
#define mtypealgo1_channel_rewr_rdy          3
#define trangealgo1_channel_rewr_rdy         0
#define mtypeRST                             2
#define trangeRST                            0
#define mtypeCLK                             1
#define trangeCLK                            0
#define mtype_OUTS                           1999 /*C->VHDL OUT ports*/
#define mtype_INS                            2999 /*VHDL->C IN ports*/

extern int                           IPCKEY;
extern TIME                          _actual_t;

#ifdef CLILIB /*CliProcedures*/

/*Open_VCIProcedures*/
extern int
algo1itfOpen(/*did,iid*/ );
/*cliDID did;
  cliIID iid;
 */

/*ErrorCLI*/
extern int
algo1itfError( /*did,iid*/ );
/* cliDID did;
   cliIID iid;
 */

/*CloseCLI*/
extern int
algo1itfClose( /*did,iid*/ );
/*cliDID did;
  cliIID iid;
 */

/*EvalCLI*/
extern int
algo1itfEval(/*did,iid,ppid,np*/);
/*cliDID did;
  cliIID iid;
  cliPID *ppid;
  int np;
 */

#else /*CProcedures*/

/*ReceiveINAll(local-C)*/
extern int
algo1itfReceiveINAll();

/*SendOUTNotValid(local-C)*/
extern int
algo1itfSendOUTNotValid();

/*SendOUTAll(local-C)*/
extern int
algo1itfSendOUTAll();

/*ReceiveIN(local-C)*/
extern int
algo1itfReceiveIN(/*ipckey*/ );
/* int ipckey;
 */

/*SendOUT(local-C)*/
extern int
algo1itfSendOUT(/*ipckey*/ );
/* int ipckey;
 */

/*SendINAll(external-C)*/
extern int
algo1itfSendINAll();

/*ReceiveOUTAll(external-C)*/
extern int
algo1itfReceiveOUTAll();

/*SendIN(external-C)*/
extern int
algo1itfSendIN(/*ipckey*/ );
/* int ipckey;
 */

/*SendINNotValid(external-C)*/
extern int
algo1itfSendINNotValid();

/*ReceiveOut(external-C)*/
extern int
algo1itfReceiveOUT(/*ipckey*/ );
/* int ipckey;
 */

#endif
#endif          /*91AJyG6*/

/* C-Interface file: algo1itf.c */
/* VCI INTERFACE:
                VHDL-C language interface - CLI Synopsys
                Author: CAValderrama (CA.Valderrama@imag.fr)
                version 5.05 MSQ
 */
/* C-Interface file: algo1itf.c */

/*CIncludes*/
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include "algo1itf.h"

/*-----------------------------------------------------*/


typedef struct _INS             /*97H13807*/
{
  int                           valid;
  short int                     send_map[11];
  RESOLVED_REAL                         control_channel_data_real;
  RESOLVED_INTEGER                      control_channel_data_int;
  BIT                           control_channel_rewr_rdy;
  RESOLVED_REAL                         algo1_channel_data_real;
  RESOLVED_INTEGER                      algo1_channel_data_int;
  BIT                           algo1_channel_rewr_rdy;
  BIT                           RST;
  BIT                           CLK;
} _INS;

typedef struct _OUTS            /*97H13807*/
{
  int                           valid;
  short int                     send_map[11];
  RESOLVED_REAL                         control_channel_data_real;
  RESOLVED_INTEGER                      control_channel_data_int;
  BIT                           control_channel_wr_req;
  RESOLVED_REAL                         algo1_channel_data_real;
  RESOLVED_INTEGER                      algo1_channel_data_int;
  BIT                           algo1_channel_read_req;
} _OUTS;

#ifdef CLILIB    /*91ACIRC*/

cliAFTER _AFTERevents;

typedef struct _VALUES /*VALUESStruct*/
{
  cliVALUE                      IPCKEY;
  cliVALUE                      control_channel_data_real;
  cliVALUE                      control_channel_data_int;
  cliVALUE                      control_channel_rewr_rdy;
  cliVALUE                      control_channel_wr_req;
  cliVALUE                      algo1_channel_data_real;
  cliVALUE                      algo1_channel_data_int;
  cliVALUE                      algo1_channel_read_req;
  cliVALUE                      algo1_channel_rewr_rdy;
  cliVALUE                      RST;
  cliVALUE                      CLK;
} _VALUES;
static _VALUES _values;

typedef struct _DATAS /*DatasStruct*/
{
  int                           ipckey;
  TIME                          last_t;
  int                           sent;
  int                           receive;
  int                           first_change;
  cliGID                        IPCKEY;
  _INS                          ins;
  _OUTS                         outs;
  cliPID                        control_channel_data_real;
  cliPID                        control_channel_data_int;
  cliPID                        control_channel_rewr_rdy;
  cliPID                        control_channel_wr_req;
  cliPID                        algo1_channel_data_real;
  cliPID                        algo1_channel_data_int;
  cliPID                        algo1_channel_read_req;
  cliPID                        algo1_channel_rewr_rdy;
  cliPID                        RST;
  cliPID                        CLK;
} _DATAS;
_DATAS *_datasp;

#define _last_t  _datasp->last_t
#define _sent                   _datasp->sent
#define _receive                _datasp->receive
#define _1st_change _datasp->first_change
#define _ins _datasp->ins
#define _outs _datasp->outs

#define _DEresetDATASp(dtp_) \
  (dtp_)->last_t= _NULLtime; \
  (dtp_)->sent= 0; (dtp_)->receive= 0


/*ConvertionTypeToValue*/

#endif /*91ACIRC*/


#ifndef CLILIB   /*91AJAnM*/

/*C-File global variables*/

static int                      _1st_change= 1;
RESOLVED_REAL                           control_channel_data_real;
RESOLVED_INTEGER                        control_channel_data_int;
BIT                             control_channel_rewr_rdy;
BIT                             control_channel_wr_req;
RESOLVED_REAL                           algo1_channel_data_real;
RESOLVED_INTEGER                        algo1_channel_data_int;
BIT                             algo1_channel_read_req;
BIT                             algo1_channel_rewr_rdy;
BIT                             RST;
BIT                             CLK;
static          _INS            _ins;           /*defInsVar*/
static          _OUTS           _outs;          /*defOutsVar*/
short int                       _send_map[11]= {0,0,0,0,0,0,0,0,0,0,0};

#endif          /*91AJAnM*/

short int                       _NULLmap[11];
#define _DIOResetMap(dir_) \
memcpy((char*)dir_.send_map,(char*)_NULLmap,11)
TIME                            _actual_t;

#define _oIO(pn_)               ((_outs.pn_!=pn_)&&(_ins.pn_!=pn_))
#define _DIOScmp(vn_,pn_)       vn_=(_oIO(pn_))? 1: 0
#define _IOvv(pt_,pn_,vn_)      ((pt_*)pn_)[vn_]
#define _IOvo(pt_,pn_,vn_)      ((pt_*)_outs.pn_)[vn_]
#define _IOvi(pt_,pn_,vn_)      ((pt_*)_ins.pn_)[vn_]
#define _cIOvo(pt_,pn_,vn_)     (_IOvv(pt_,pn_,vn_)!=_IOvo(pt_,pn_,vn_))
#define _cIOvi(pt_,pn_,vn_)     (_IOvv(pt_,pn_,vn_)!=_IOvi(pt_,pn_,vn_))
#define _oIOv(pt_,pn_,vn_)      (_cIOvo(pt_,pn_,vn_)&&_cIOvi(pt_,pn_,vn_))
#define _DIOVcmp(vn_,pn_,pz_,pt_)       for(vn_=pz_;vn_>0;vn_--)if(_oIOv(pt_,pn_,(vn_-
1)))){vn_= 1;break;}
```

```
#define _DIOold(vn_,pn_)          (_1st_change||(!BitAddr(0)&&(vn_!=0)))
#define _DIOnew(vn_,pn_,mn_)(BitAddr(mn_)||_DIOold(vn_,pn_))

#define MapPinouts(dir_,mn_)   dir_.send_map[mn_]= 1
#define BitPinouts(dir_,mn_)     ((dir_.send_map[mn_])!=0)
#define ClrPinouts(dir_,mn_)     dir_.send_map[mn_]= 0

#ifdef CLILIB /*CliProcedures*/

#ifdef STATUS
#define _DSopen(sfn_) \
   _statfip =(FILE*)OpenStatusFile(); \
   SimTimeStatus \
   ( _statfip, IPCKEY, sfn_, \
     _actual_t.high, _actual_t.low,_actual_t.timebase \
   );
#else /*NO STATUS*/
#define _DSopen(sfn_)
#endif /*STATUS*/


#ifdef DEBUG
TIME                _enexttime;
#define _DDENEXTeventTIME() \
                    { _VECgetNEXTeventTIME(&_enexttime); \
                      _VECshowNEXTeventTIME(&_enexttime); }
#else /*NO DEBUG*/
#define _DDENEXTeventTIME()
#endif /*DEBUG*/

#ifdef DEBUG
#define _DDeproc(epn_) _DEProc(epn_,did,iid,np,ppid)
#else /*NO DEBUG*/
#define _DDeproc(epn_)
#endif /*DEBUG*/

#define _DEvp2id(pov_,pon_,tcv_,trz_) \
   tcv_(&(_values.pov_),trz_); \
   _DEPortName2PID(did,pon_,_datasp->pov_); \
   _DEPort(pon_,_datasp->pov_);

#define _DEsp2id(pov_,pon_,tcv_) \
   tcv_(&(_values.pov_)); \
   _DEPortName2PID(did,pon_,_datasp->pov_); \
   _DEPort(pon_,_datasp->pov_);

#define _DEsg2id(gev_,gen_,tcv_) \
   tcv_(&(_values.gev_)); \
   _DEGenericName2GID(did,gen_,_datasp->gev_); \
   _DEGeneric(gen_,_datasp->gev_);

#define _DESgenericGet(gev_,ttv_) \
   cliGetGenericValue \
   (did,iid,_datasp->gev_,(&(_values.gev_))); \
   algo1itfError(did,iid); \
   gev_= ttv_(_values.gev_)

/*Open_VCIProcedures*/
int
algo1itfOpen( did,iid )
cliDID did;
cliIID iid;
{
/*LocalVariables*/
   printf("\n(algo1itfOpen(did %d)(iid %d))\n",did,iid);
   _AFTERevents= _NULLafter;

/*MemoryAllocation*/
   _datasp= (_DATAS*)cliAllocMem(sizeof(_DATAS));
   printf("(_datasp %p)\n",(void*)_datasp);

/*_DATASStructInit*/

_DEsp2id(control_channel_data_real,"control_channel_data_real",RESOLVED_REALVALUE);

_DEsp2id(control_channel_data_int,"control_channel_data_int",RESOLVED_INTEGERVALUE)
;
   _DEsp2id(control_channel_rewr_rdy,"control_channel_rewr_rdy",BITVALUE);
   _DEsp2id(control_channel_wr_req,"control_channel_wr_req",BITVALUE);
   _DEsp2id(algo1_channel_data_real,"algo1_channel_data_real",RESOLVED_REALVALUE);
   _DEsp2id(algo1_channel_data_int,"algo1_channel_data_int",RESOLVED_INTEGERVALUE);
   _DEsp2id(algo1_channel_read_req,"algo1_channel_read_req",BITVALUE);
   _DEsp2id(algo1_channel_rewr_rdy,"algo1_channel_rewr_rdy",BITVALUE);
   _DEsp2id(RST,"RST",BITVALUE);
   _DEsp2id(CLK,"CLK",BITVALUE);
   _DEsg2id(IPCKEY,"IPCKEY",INTEGERVALUE);
   _DEresetDATASp(_datasp);
   cliGetSimTime(&_actual_t);
   _DESgenericGet(IPCKEY,ToINTEGER);
   _datasp->ipckey= IPCKEY;
   printf("(IPCKEY %d %d)\n",_datasp->IPCKEY,IPCKEY);
   ClrPinouts(_ins,0);
   _1st_change= 1;

/*SaveInstanceData*/
   cliSaveInstanceData(did,iid,(void*)_datasp);

   algo1itfError(did,iid);
   _DSopen("algo1itf");
   _DSclose("algo1itf");

   return 0;
}

/*ErrorCLI*/
int
algo1itfError( did,iid )
cliDID did;
cliIID iid;
{
   if (cli_fatals)
   {
     printf("(cli_fatals %d)\n",cli_fatals);
     algo1itfClose(did,iid);
   }
   if (cli_errno)
     printf( "(cli_errno %d)\n",cli_errno);

   return (0);
}

/*CloseCLI*/
int
algo1itfClose( did,iid )
cliDID did;
cliIID iid;
{
   printf("\n(algo1itfClose(did %d)(iid %d))\n",did,iid);
   deleteCHANNEL(_datasp->ipckey);

   return (0);
}

#define _DESportGet(pov_,ttv_) \
   cliGetPortValue \
   (did,iid,_datasp->pov_,(&(_values.pov_))); \
   algo1itfError(did,iid); \
   _ins.pov_= ttv_(_values.pov_)

#define _DEVportGet(pov_,ttv_,trz_) \
   cliGetPortValue \
   (did,iid,_datasp->pov_,(&(_values.pov_))); \
   algo1itfError(did,iid); \
   memcpy((char*)_ins.pov_,(char*)ttv_(_values.pov_),sizeof(_ins.pov_))


#define _DESportPut(pov_,ftv_) \
   _values.pov_= ftv_(_outs.pov_); \
   cliScheduleOutput \
   (did,iid,_datasp->pov_,(&(_values.pov_)),_RNULLafter); \
   algo1itfError(did,iid);


#define _DEVportPut(pov_,ftv_,trz_) \
   _values.pov_= ftv_(_outs.pov_,trz_); \
   cliScheduleOutput \
   (did,iid,_datasp->pov_,(&(_values.pov_)),_RNULLafter); \
   algo1itfError(did,iid);

/*EvalCLI*/
int
algo1itfEval(did,iid,ppid,np)
cliDID did;
cliIID iid;
cliPID *ppid;
int np;
{
/*LocalVariables*/
   int event_found= 0;
   int k;
   int result;
   int ok2send= 0;
   int ok2receive= 0;
   int i;

   _datasp= (_DATAS*)cliRestoreInstanceData(did,iid);
   IPCKEY= _datasp->ipckey;
   _DDeproc("algo1itfEval");
   cliGetSimTime(&_actual_t);
   _DDENEXTeventTIME();


   if (np>0) /*drqPFE*/
   { _DSopen("algo1itf");
     for (i=0;i<np;i++)/*drr10a*/
     {
       if (ppid[i]==_datasp->control_channel_data_real)
       {
         /*GetPortValue: NotSensoredSignal(N)*/
         /*CLIValue2Variable*/
         MapPinouts(_ins,mtypecontrol_channel_data_real);
         _DESportGet(control_channel_data_real,ToRESOLVED_REAL);

         _DDport

(_ins.control_channel_data_real,"control_channel_data_real",mtypecontrol_channel_data_real,
ssRESOLVED_REAL,RESOLVED_REAL,0);
         _DSport

(_ins.control_channel_data_real,"control_channel_data_real",ssRESOLVED_REAL,RESOLVE
D_REAL,0);
       } /*ppid[control_channel_data_real]*/
       if (ppid[i]==_datasp->control_channel_data_int)
       {
         /*GetPortValue: NotSensoredSignal(N)*/
         /*CLIValue2Variable*/
         MapPinouts(_ins,mtypecontrol_channel_data_int);
         _DESportGet(control_channel_data_int,ToRESOLVED_INTEGER);

         _DDport

(_ins.control_channel_data_int,"control_channel_data_int",mtypecontrol_channel_data_int,ssR
ESOLVED_INTEGER,RESOLVED_INTEGER,0);
         _DSport

(_ins.control_channel_data_int,"control_channel_data_int",ssRESOLVED_INTEGER,RESOLVE
D_INTEGER,0);
       } /*ppid[control_channel_data_int]*/
       if (ppid[i]==_datasp->control_channel_rewr_rdy)
       {
         /*GetPortValue: NotSensoredSignal(N)*/
         /*CLIValue2Variable*/
         MapPinouts(_ins,mtypecontrol_channel_rewr_rdy);
         _DESportGet(control_channel_rewr_rdy,ToBIT);

         _DDport

(_ins.control_channel_rewr_rdy,"control_channel_rewr_rdy",mtypecontrol_channel_rewr_rdy,ss
BIT,BIT,0);
         _DSport
         (_ins.control_channel_rewr_rdy,"control_channel_rewr_rdy",ssBIT,BIT,0);
       } /*ppid[control_channel_rewr_rdy]*/
       if (ppid[i]==_datasp->algo1_channel_data_real)
       {
         /*GetPortValue: NotSensoredSignal(N)*/
         /*CLIValue2Variable*/
         MapPinouts(_ins,mtypealgo1_channel_data_real);
         _DESportGet(algo1_channel_data_real,ToRESOLVED_REAL);
```

```
        _DDport

(_ins.algo1_channel_data_real,"algo1_channel_data_real",mtypealgo1_channel_data_real,ssR
ESOLVED_REAL,RESOLVED_REAL,0);
        _DSport

(_ins.algo1_channel_data_real,"algo1_channel_data_real",ssRESOLVED_REAL,RESOLVED_
REAL,0);
      } /*ppid[algo1_channel_data_real]*/
      if (ppid[i]==_datasp->algo1_channel_data_int)
      {
        /*GetPortValue: NotSensoredSignal(N)*/
        /*CLIValue2Variable*/
        MapPinouts(_ins,mtypealgo1_channel_data_int);
        _DESportGet(algo1_channel_data_int,ToRESOLVED_INTEGER);

        _DDport

(_ins.algo1_channel_data_int,"algo1_channel_data_int",mtypealgo1_channel_data_int,ssRESO
LVED_INTEGER,RESOLVED_INTEGER,0);
        _DSport

(_ins.algo1_channel_data_int,"algo1_channel_data_int",ssRESOLVED_INTEGER,RESOLVED
_INTEGER,0);
      } /*ppid[algo1_channel_data_int]*/
      if (ppid[i]==_datasp->algo1_channel_rewr_rdy)
      {
        /*GetPortValue: NotSensoredSignal(N)*/
        /*CLIValue2Variable*/
        MapPinouts(_ins,mtypealgo1_channel_rewr_rdy);
        _DESportGet(algo1_channel_rewr_rdy,ToBIT);

        _DDport

(_ins.algo1_channel_rewr_rdy,"algo1_channel_rewr_rdy",mtypealgo1_channel_rewr_rdy,ssBIT,
BIT,0);
        _DSport
        (_ins.algo1_channel_rewr_rdy,"algo1_channel_rewr_rdy",ssBIT,BIT,0);
      } /*ppid[algo1_channel_rewr_rdy]*/
      if (ppid[i]==_datasp->RST)
      {
        /*GetPortValue: NotSensoredSignal(N)*/
        /*CLIValue2Variable*/
        MapPinouts(_ins,mtypeRST);
        _DESportGet(RST,ToBIT);

        _DDport
        (_ins.RST,"RST",mtypeRST,ssBIT,BIT,0);
        _DSport
        (_ins.RST,"RST",ssBIT,BIT,0);
      } /*ppid[RST]*/
      if (ppid[i]==_datasp->CLK)
      {
        /*GetPortValue: SensoredSignal(R|F|Y)*/
        /*CLIValue2Variable*/
        MapPinouts(_ins,mtypeCLK);
        _DESportGet(CLK,ToBIT);

        _DDport
        (_ins.CLK,"CLK",mtypeCLK,ssBIT,BIT,0);
        _DSport
        (_ins.CLK,"CLK",ssBIT,BIT,0);
        if (_ins.CLK == BIT_1)
        { event_found= 1;
        }
      } /*ppid[CLK]*/
    } /*drr10a*/
    ; /*97H14956*/
    _DSclose("algo1itf");
  } /*drqPFE*/
  if /*ReturnIfSent*/
  ( (_last_t.high==_actual_t.high)&&
    (_last_t.low==_actual_t.low)
  )
  { if (_sent)
    { if (_receive==0)
      { return 0;
      }
    }
  }
  else
  { _sent= 0;
  }
  if (_1st_change) /*Err10a*/
  {
    _1st_change= 0;
    ok2send= ok2receive= 1;
  }
  else
  {
    if (event_found) /*TestEventFound*/
    {
      cliScheduleWakeup(did,iid,&_AFTERevents);
      return 0;
    }
  } /*Err10a*/
  if (np==0) /*NpNullAction*/
  {
    if (_sent==0)
    { ok2send= 1; ok2receive= 1;
    }
    if (_receive)
    { ok2receive= 1;
    }
  }
  if (ok2send) /*0KrTsFA*/
  { /*SendVhdlData*/
    _ins.valid= 1;
    MapPinouts(_ins,0);
    sendMSG
    (
      _datasp->ipckey,mtype_INS,
      sizeof(_ins),(char*)&(_ins)
    );
    ; /*97H14956*/
    _DDiop("algo1itfSendIN",_datasp->ipckey);
    _sent= 1;
    algo1itfError(did,iid);
    _last_t= _actual_t;
    _DIOResetMap(_ins); /*97H14436*/
    result= -1;
  } /*0KrTsFA*/
  if (ok2receive) /*97H12612*/
  {
    /*look= timeInSeconds();*/
    result=
    receiveNoWaitMSG
    (
      _datasp->ipckey,
      mtype_OUTS,sizeof(_outs),(char*)&(_outs)
    );
    if (result!=-1) /*A91ACIR*/
    {
      _receive= 0; /*97H121130*/
      if (_outs.valid==0) /*291ACId*/
      {
        ClrPinouts(_outs,0);
        return 0;
      } /*291ACId*/
    } /*A91ACIR*/
    else
    {
      _receive= 1;
      cliScheduleWakeup(did,iid,&_AFTERevents);
      return 0;
    }
    _DDiop("algo1itfReceiveOUT",_datasp->ipckey);
    if (BitPinouts(_outs,mtypecontrol_channel_data_real)) /*5rqPFA*/
    {
      /*OutputPortValue*/
      ClrPinouts(_outs,mtypecontrol_channel_data_real);
      _DDport

(_outs.control_channel_data_real,"control_channel_data_real",mtypecontrol_channel_data_real
,ssRESOLVED_REAL,RESOLVED_REAL,0);
      _DESportPut(control_channel_data_real,FromRESOLVED_REAL);
    } /*5rqPFA*/
    if (BitPinouts(_outs,mtypecontrol_channel_data_int)) /*5rqPFA*/
    {
      /*OutputPortValue*/
      ClrPinouts(_outs,mtypecontrol_channel_data_int);
      _DDport

(_outs.control_channel_data_int,"control_channel_data_int",mtypecontrol_channel_data_int,ss
RESOLVED_INTEGER,RESOLVED_INTEGER,0);
      _DESportPut(control_channel_data_int,FromRESOLVED_INTEGER);
    } /*5rqPFA*/
    if (BitPinouts(_outs,mtypecontrol_channel_wr_req)) /*5rqPFA*/
    {
      /*OutputPortValue*/
      ClrPinouts(_outs,mtypecontrol_channel_wr_req);
      _DDport

(_outs.control_channel_wr_req,"control_channel_wr_req",mtypecontrol_channel_wr_req,ssBIT,
BIT,0);
      _DESportPut(control_channel_wr_req,FromBIT);
    } /*5rqPFA*/
    if (BitPinouts(_outs,mtypealgo1_channel_data_real)) /*5rqPFA*/
    {
      /*OutputPortValue*/
      ClrPinouts(_outs,mtypealgo1_channel_data_real);
      _DDport

(_outs.algo1_channel_data_real,"algo1_channel_data_real",mtypealgo1_channel_data_real,ss
RESOLVED_REAL,RESOLVED_REAL,0);
      _DESportPut(algo1_channel_data_real,FromRESOLVED_REAL);
    } /*5rqPFA*/
    if (BitPinouts(_outs,mtypealgo1_channel_data_int)) /*5rqPFA*/
    {
      /*OutputPortValue*/
      ClrPinouts(_outs,mtypealgo1_channel_data_int);
      _DDport

(_outs.algo1_channel_data_int,"algo1_channel_data_int",mtypealgo1_channel_data_int,ssRES
OLVED_INTEGER,RESOLVED_INTEGER,0);
      _DESportPut(algo1_channel_data_int,FromRESOLVED_INTEGER);
    } /*5rqPFA*/
    if (BitPinouts(_outs,mtypealgo1_channel_read_req)) /*5rqPFA*/
    {
      /*OutputPortValue*/
      ClrPinouts(_outs,mtypealgo1_channel_read_req);
      _DDport

(_outs.algo1_channel_read_req,"algo1_channel_read_req",mtypealgo1_channel_read_req,ssBI
T,BIT,0);
      _DESportPut(algo1_channel_read_req,FromBIT);
    } /*5rqPFA*/
    ClrPinouts(_outs,0);
  } /*97H12612*/
  return (0);
}

#else /*CProcedures*/

/*ReceiveINAll(local-C)*/
int
algo1itfReceiveINAll()
{
  int result= -1;
  while (1) /*F91ACIR*/
  {
    result=
    receiveNoWaitMSG
    (
      IPCKEY,
      mtype_INS,sizeof(_ins),(char*)&(_ins)
    );
    if (result!=-1) /*A91ACIR*/
    {
      if (_ins.valid==0) /*291ACId*/
      {
        ClrPinouts(_ins,0);
        return 0;
      } /*291ACId*/
      break;
    } /*A91ACIR*/
  } /*F91ACIR*/
```

```
  /*ActualizePortsToReceive*/
  if (BitPinouts(_ins,mtypecontrol_channel_data_real)) /*0rqPFA*/
  {
    if (_1st_change) /*f91ACcJ*/
    { _outs.control_channel_data_real= _ins.control_channel_data_real;
    } /*f91ACcJ*/
    ClrPinouts(_ins,mtypecontrol_channel_data_real);
    control_channel_data_real= _ins.control_channel_data_real;
    _DDport

(control_channel_data_real,"control_channel_data_real",mtypecontrol_channel_data_real,ssRE
SOLVED_REAL,RESOLVED_REAL,0);
  } /*0rqPFA*/
  /*ActualizePortsToReceive*/
  if (BitPinouts(_ins,mtypecontrol_channel_data_int)) /*0rqPFA*/
  {
    if (_1st_change) /*f91ACcJ*/
    { _outs.control_channel_data_int= _ins.control_channel_data_int;
    } /*f91ACcJ*/
    ClrPinouts(_ins,mtypecontrol_channel_data_int);
    control_channel_data_int= _ins.control_channel_data_int;
    _DDport

(control_channel_data_int,"control_channel_data_int",mtypecontrol_channel_data_int,ssRESOL
VED_INTEGER,RESOLVED_INTEGER,0);
  } /*0rqPFA*/
  /*ActualizePortsToReceive*/
  if (BitPinouts(_ins,mtypecontrol_channel_rewr_rdy)) /*0rqPFA*/
  {
    ClrPinouts(_ins,mtypecontrol_channel_rewr_rdy);
    control_channel_rewr_rdy= _ins.control_channel_rewr_rdy;
    _DDport

(control_channel_rewr_rdy,"control_channel_rewr_rdy",mtypecontrol_channel_rewr_rdy,ssBIT,
BIT,0);
  } /*0rqPFA*/
  /*ActualizePortsToReceive*/
  if (BitPinouts(_ins,mtypealgo1_channel_data_real)) /*0rqPFA*/
  {
    if (_1st_change) /*f91ACcJ*/
    { _outs.algo1_channel_data_real= _ins.algo1_channel_data_real;
    } /*f91ACcJ*/
    ClrPinouts(_ins,mtypealgo1_channel_data_real);
    algo1_channel_data_real= _ins.algo1_channel_data_real;
    _DDport

(algo1_channel_data_real,"algo1_channel_data_real",mtypealgo1_channel_data_real,ssRESO
LVED_REAL,RESOLVED_REAL,0);
  } /*0rqPFA*/
  /*ActualizePortsToReceive*/
  if (BitPinouts(_ins,mtypealgo1_channel_data_int)) /*0rqPFA*/
  {
    if (_1st_change) /*f91ACcJ*/
    { _outs.algo1_channel_data_int= _ins.algo1_channel_data_int;
    } /*f91ACcJ*/
    ClrPinouts(_ins,mtypealgo1_channel_data_int);
    algo1_channel_data_int= _ins.algo1_channel_data_int;
    _DDport

(algo1_channel_data_int,"algo1_channel_data_int",mtypealgo1_channel_data_int,ssRESOLVE
D_INTEGER,RESOLVED_INTEGER,0);
  } /*0rqPFA*/
  /*ActualizePortsToReceive*/
  if (BitPinouts(_ins,mtypealgo1_channel_rewr_rdy)) /*0rqPFA*/
  {
    ClrPinouts(_ins,mtypealgo1_channel_rewr_rdy);
    algo1_channel_rewr_rdy= _ins.algo1_channel_rewr_rdy;
    _DDport

(algo1_channel_rewr_rdy,"algo1_channel_rewr_rdy",mtypealgo1_channel_rewr_rdy,ssBIT,BIT,
0);
  } /*0rqPFA*/
  /*ActualizePortsToReceive*/
  if (BitPinouts(_ins,mtypeRST)) /*0rqPFA*/
  {
    ClrPinouts(_ins,mtypeRST);
    RST= _ins.RST;
    _DDport
    (RST,"RST",mtypeRST,ssBIT,BIT,0);
  } /*0rqPFA*/
  /*ActualizePortsToReceive*/
  if (BitPinouts(_ins,mtypeCLK)) /*0rqPFA*/
  {
    ClrPinouts(_ins,mtypeCLK);
    CLK= _ins.CLK;
    _DDport
    (CLK,"CLK",mtypeCLK,ssBIT,BIT,0);
  } /*0rqPFA*/
  ClrPinouts(_ins,0);
  _1st_change= 0; /*97H12851*/
  _DDiop("algo1itfReceiveIN",IPCKEY);

  return 0;

} /*ReceiveINAll(local-C)*/

/*SendOUTNotValid(local-C)*/
int
algo1itfSendOUTNotValid()
{
  _outs.valid= 0;
  MapPinouts(_outs,0);
  sendMSG
  (
    IPCKEY,mtype_OUTS,
    sizeof(_outs),(char*)&(_outs)
  );
  ; /*97H14956*/
  _DDiop("algo1itfSendOUTNotValid",IPCKEY);

  return 0;
} /*SendOUTNotValid(local-C)*/

/*SendOUTAll(local-C)*/
int
algo1itfSendOUTAll()
{
  int _send_diff= 0;
  ClrPinouts(_outs,0);
  ;
  /*Actualize OUT port [control_channel_data_real]*/
  _DIOScmp(_send_diff,control_channel_data_real);
  if (_DIOnew(_send_diff,control_channel_data_real,mtypecontrol_channel_data_real))
  { _DIOmap(mtypecontrol_channel_data_real);
  }
  if (BitAddr(mtypecontrol_channel_data_real)) /*erqPFA*/
  {
    _outs.control_channel_data_real= control_channel_data_real;
    MapPinouts(_outs,mtypecontrol_channel_data_real);
    MapClr(mtypecontrol_channel_data_real);
    _DDport

(control_channel_data_real,"control_channel_data_real",mtypecontrol_channel_data_real,ssRE
SOLVED_REAL,RESOLVED_REAL,0);
  } /*erqPFA*/
  /*Actualize OUT port [control_channel_data_int]*/
  _DIOScmp(_send_diff,control_channel_data_int);
  if (_DIOnew(_send_diff,control_channel_data_int,mtypecontrol_channel_data_int))
  { _DIOmap(mtypecontrol_channel_data_int);
  }
  if (BitAddr(mtypecontrol_channel_data_int)) /*erqPFA*/
  {
    _outs.control_channel_data_int= control_channel_data_int;
    MapPinouts(_outs,mtypecontrol_channel_data_int);
    MapClr(mtypecontrol_channel_data_int);
    _DDport

(control_channel_data_int,"control_channel_data_int",mtypecontrol_channel_data_int,ssRESOL
VED_INTEGER,RESOLVED_INTEGER,0);
  } /*erqPFA*/
  /*Actualize OUT port [control_channel_wr_req]*/
  if (BitAddr(mtypecontrol_channel_wr_req)) /*erqPFA*/
  {
    _outs.control_channel_wr_req= control_channel_wr_req;
    MapPinouts(_outs,mtypecontrol_channel_wr_req);
    MapClr(mtypecontrol_channel_wr_req);
    _DDport

(control_channel_wr_req,"control_channel_wr_req",mtypecontrol_channel_wr_req,ssBIT,BIT,0);
  } /*erqPFA*/
  /*Actualize OUT port [algo1_channel_data_real]*/
  _DIOScmp(_send_diff,algo1_channel_data_real);
  if (_DIOnew(_send_diff,algo1_channel_data_real,mtypealgo1_channel_data_real))
  { _DIOmap(mtypealgo1_channel_data_real);
  }
  if (BitAddr(mtypealgo1_channel_data_real)) /*erqPFA*/
  {
    _outs.algo1_channel_data_real= algo1_channel_data_real;
    MapPinouts(_outs,mtypealgo1_channel_data_real);
    MapClr(mtypealgo1_channel_data_real);
    _DDport

(algo1_channel_data_real,"algo1_channel_data_real",mtypealgo1_channel_data_real,ssRESO
LVED_REAL,RESOLVED_REAL,0);
  } /*erqPFA*/
  /*Actualize OUT port [algo1_channel_data_int]*/
  _DIOScmp(_send_diff,algo1_channel_data_int);
  if (_DIOnew(_send_diff,algo1_channel_data_int,mtypealgo1_channel_data_int))
  { _DIOmap(mtypealgo1_channel_data_int);
  }
  if (BitAddr(mtypealgo1_channel_data_int)) /*erqPFA*/
  {
    _outs.algo1_channel_data_int= algo1_channel_data_int;
    MapPinouts(_outs,mtypealgo1_channel_data_int);
    MapClr(mtypealgo1_channel_data_int);
    _DDport

(algo1_channel_data_int,"algo1_channel_data_int",mtypealgo1_channel_data_int,ssRESOLVE
D_INTEGER,RESOLVED_INTEGER,0);
  } /*erqPFA*/
  /*Actualize OUT port [algo1_channel_read_req]*/
  if (BitAddr(mtypealgo1_channel_read_req)) /*erqPFA*/
  {
    _outs.algo1_channel_read_req= algo1_channel_read_req;
    MapPinouts(_outs,mtypealgo1_channel_read_req);
    MapClr(mtypealgo1_channel_read_req);
    _DDport

(algo1_channel_read_req,"algo1_channel_read_req",mtypealgo1_channel_read_req,ssBIT,BIT,
0);
  } /*erqPFA*/
  if (BitAddr(0)) /*WrqPFA*/
  {
    _outs.valid= 1;
    MapPinouts(_outs,0);
    sendMSG
    (
      IPCKEY,mtype_OUTS,
      sizeof(_outs),(char*)&(_outs)
    );
    MapClr(0); /*97H13015*/
    _DIOResetMap(_outs); /*97H14436*/
  } /*WrqPFA*/
  ; /*97H14956*/
  _1st_change= 0; /*97H12851*/
  _DDiop("algo1itfSendOUT",IPCKEY);

  return 0;
} /*SendOUTAll(local-C)*/

/*ReceiveIN(local-C)*/
int
algo1itfReceiveIN(ipckey)
int ipckey;
{
  IPCKEY= ipckey;
  if (_1st_change==0)
  { /*SrqnFV*/
    algo1itfSendOUTNotValid();
  } /*SrqnFV*/
  algo1itfReceiveINAll();

  return 0;

} /*ReceiveIN(local-C)*/

/*SendOUT(local-C)*/
int
```

```
algo1itfSendOUT(ipckey)
int ipckey;
{
  IPCKEY= ipckey;
  if (_1st_change==1)
  { /*30Se814*/
    algo1itfReceiveINAll();
  } /*30Se814*/
  algo1itfSendOUTAll();
  algo1itfReceiveINAll();

  return 0;

} /*SendOUT(local-C)*/

/*SendINAll(external-C)*/
int
algo1itfSendINAll()
{
  int _send_diff= 0;
  ClrPinouts(_ins,0);
  ;
  /*Actualize OUT port [control_channel_data_real]*/
  _DIOScmp(_send_diff,control_channel_data_real);
  if (_DIOnew(_send_diff,control_channel_data_real,mtypecontrol_channel_data_real))
  { _DIOmap(mtypecontrol_channel_data_real);
  }
  if (BitAddr(mtypecontrol_channel_data_real)) /*erqPFA*/
  {
    _ins.control_channel_data_real= control_channel_data_real;
    MapPinouts(_ins,mtypecontrol_channel_data_real);
    MapClr(mtypecontrol_channel_data_real);
    _DDport

(control_channel_data_real,"control_channel_data_real",mtypecontrol_channel_data_real,ssRE
SOLVED_REAL,RESOLVED_REAL,0);
  } /*erqPFA*/
  /*Actualize OUT port [control_channel_data_int]*/
  _DIOScmp(_send_diff,control_channel_data_int);
  if (_DIOnew(_send_diff,control_channel_data_int,mtypecontrol_channel_data_int))
  { _DIOmap(mtypecontrol_channel_data_int);
  }
  if (BitAddr(mtypecontrol_channel_data_int)) /*erqPFA*/
  {
    _ins.control_channel_data_int= control_channel_data_int;
    MapPinouts(_ins,mtypecontrol_channel_data_int);
    MapClr(mtypecontrol_channel_data_int);
    _DDport

(control_channel_data_int,"control_channel_data_int",mtypecontrol_channel_data_int,ssRESOL
VED_INTEGER,RESOLVED_INTEGER,0);
  } /*erqPFA*/
  /*Actualize OUT port [control_channel_rewr_rdy]*/
  if (BitAddr(mtypecontrol_channel_rewr_rdy)) /*erqPFA*/
  {
    _ins.control_channel_rewr_rdy= control_channel_rewr_rdy;
    MapPinouts(_ins,mtypecontrol_channel_rewr_rdy);
    MapClr(mtypecontrol_channel_rewr_rdy);
    _DDport

(control_channel_rewr_rdy,"control_channel_rewr_rdy",mtypecontrol_channel_rewr_rdy,ssBIT,
BIT,0);
  } /*erqPFA*/
  /*Actualize OUT port [algo1_channel_data_real]*/
  _DIOScmp(_send_diff,algo1_channel_data_real);
  if (_DIOnew(_send_diff,algo1_channel_data_real,mtypealgo1_channel_data_real))
  { _DIOmap(mtypealgo1_channel_data_real);
  }
  if (BitAddr(mtypealgo1_channel_data_real)) /*erqPFA*/
  {
    _ins.algo1_channel_data_real= algo1_channel_data_real;
    MapPinouts(_ins,mtypealgo1_channel_data_real);
    MapClr(mtypealgo1_channel_data_real);
    _DDport

(algo1_channel_data_real,"algo1_channel_data_real",mtypealgo1_channel_data_real,ssRESO
LVED_REAL,RESOLVED_REAL,0);
  } /*erqPFA*/
  /*Actualize OUT port [algo1_channel_data_int]*/
  _DIOScmp(_send_diff,algo1_channel_data_int);
  if (_DIOnew(_send_diff,algo1_channel_data_int,mtypealgo1_channel_data_int))
  { _DIOmap(mtypealgo1_channel_data_int);
  }
  if (BitAddr(mtypealgo1_channel_data_int)) /*erqPFA*/
  {
    _ins.algo1_channel_data_int= algo1_channel_data_int;
    MapPinouts(_ins,mtypealgo1_channel_data_int);
    MapClr(mtypealgo1_channel_data_int);
    _DDport

(algo1_channel_data_int,"algo1_channel_data_int",mtypealgo1_channel_data_int,ssRESOLVE
D_INTEGER,RESOLVED_INTEGER,0);
  } /*erqPFA*/
  /*Actualize OUT port [algo1_channel_rewr_rdy]*/
  if (BitAddr(mtypealgo1_channel_rewr_rdy)) /*erqPFA*/
  {
    _ins.algo1_channel_rewr_rdy= algo1_channel_rewr_rdy;
    MapPinouts(_ins,mtypealgo1_channel_rewr_rdy);
    MapClr(mtypealgo1_channel_rewr_rdy);
    _DDport

(algo1_channel_rewr_rdy,"algo1_channel_rewr_rdy",mtypealgo1_channel_rewr_rdy,ssBIT,BIT,
0);
  } /*erqPFA*/
  /*Actualize OUT port [RST]*/
  if (BitAddr(mtypeRST)) /*erqPFA*/
  {
    _ins.RST= RST;
    MapPinouts(_ins,mtypeRST);
    MapClr(mtypeRST);
    _DDport
    (RST,"RST",mtypeRST,ssBIT,BIT,0);
  } /*erqPFA*/
  /*Actualize OUT port [CLK]*/
  if (BitAddr(mtypeCLK)) /*erqPFA*/
  {
    _ins.CLK= CLK;
    MapPinouts(_ins,mtypeCLK);
    MapClr(mtypeCLK);
    _DDport
    (CLK,"CLK",mtypeCLK,ssBIT,BIT,0);
  } /*erqPFA*/
  if (BitAddr(0)) /*WrqPFA*/
  {
    _ins.valid= 1;
    MapPinouts(_ins,0);
    sendMSG
    (
      IPCKEY,mtype_INS,
      sizeof(_ins),(char*)&(_ins)
    );
    MapClr(0); /*97H13015*/
    _DIOResetMap(_ins); /*97H14436*/
  } /*WrqPFA*/
  ; /*97H14956*/
  _1st_change= 0; /*97H12851*/
  _DDiop("algo1itfSendIN",IPCKEY);

  return 0;

} /*SendINAll(external-C)*/

/*ReceiveOUTAll(external-C)*/
int
algo1itfReceiveOUTAll()
{
  int result= -1;
  while (1) /*F91ACIR*/
  {
    result=
    receiveNoWaitMSG
    (
      IPCKEY,
      mtype_OUTS,sizeof(_outs),(char*)&(_outs)
    );
    if (result!=-1) /*A91ACIR*/
    {
      if (_outs.valid==0) /*291ACId*/
      {
        ClrPinouts(_outs,0);
        return 0;
      } /*291ACId*/
      break;
    } /*A91ACIR*/
  } /*F91ACIR*/
  /*ActualizePortsToReceive*/
  if (BitPinouts(_outs,mtypecontrol_channel_data_real)) /*0rqPFA*/
  {
    if (_1st_change) /*f91ACcJ*/
    { _ins.control_channel_data_real= _outs.control_channel_data_real;
    } /*f91ACcJ*/
    ClrPinouts(_outs,mtypecontrol_channel_data_real);
    control_channel_data_real= _outs.control_channel_data_real;
    _DDport

(control_channel_data_real,"control_channel_data_real",mtypecontrol_channel_data_real,ssRE
SOLVED_REAL,RESOLVED_REAL,0);
  } /*0rqPFA*/
  /*ActualizePortsToReceive*/
  if (BitPinouts(_outs,mtypecontrol_channel_data_int)) /*0rqPFA*/
  {
    if (_1st_change) /*f91ACcJ*/
    { _ins.control_channel_data_int= _outs.control_channel_data_int;
    } /*f91ACcJ*/
    ClrPinouts(_outs,mtypecontrol_channel_data_int);
    control_channel_data_int= _outs.control_channel_data_int;
    _DDport

(control_channel_data_int,"control_channel_data_int",mtypecontrol_channel_data_int,ssRESOL
VED_INTEGER,RESOLVED_INTEGER,0);
  } /*0rqPFA*/
  /*ActualizePortsToReceive*/
  if (BitPinouts(_outs,mtypecontrol_channel_wr_req)) /*0rqPFA*/
  {
    ClrPinouts(_outs,mtypecontrol_channel_wr_req);
    control_channel_wr_req= _outs.control_channel_wr_req;
    _DDport

(control_channel_wr_req,"control_channel_wr_req",mtypecontrol_channel_wr_req,ssBIT,BIT,0);
  } /*0rqPFA*/
  /*ActualizePortsToReceive*/
  if (BitPinouts(_outs,mtypealgo1_channel_data_real)) /*0rqPFA*/
  {
    if (_1st_change) /*f91ACcJ*/
    { _ins.algo1_channel_data_real= _outs.algo1_channel_data_real;
    } /*f91ACcJ*/
    ClrPinouts(_outs,mtypealgo1_channel_data_real);
    algo1_channel_data_real= _outs.algo1_channel_data_real;
    _DDport

(algo1_channel_data_real,"algo1_channel_data_real",mtypealgo1_channel_data_real,ssRESO
LVED_REAL,RESOLVED_REAL,0);
  } /*0rqPFA*/
  /*ActualizePortsToReceive*/
  if (BitPinouts(_outs,mtypealgo1_channel_data_int)) /*0rqPFA*/
  {
    if (_1st_change) /*f91ACcJ*/
    { _ins.algo1_channel_data_int= _outs.algo1_channel_data_int;
    } /*f91ACcJ*/
    ClrPinouts(_outs,mtypealgo1_channel_data_int);
    algo1_channel_data_int= _outs.algo1_channel_data_int;
    _DDport

(algo1_channel_data_int,"algo1_channel_data_int",mtypealgo1_channel_data_int,ssRESOLVE
D_INTEGER,RESOLVED_INTEGER,0);
  } /*0rqPFA*/
  /*ActualizePortsToReceive*/
  if (BitPinouts(_outs,mtypealgo1_channel_read_req)) /*0rqPFA*/
  {
    ClrPinouts(_outs,mtypealgo1_channel_read_req);
    algo1_channel_read_req= _outs.algo1_channel_read_req;
    _DDport

(algo1_channel_read_req,"algo1_channel_read_req",mtypealgo1_channel_read_req,ssBIT,BIT,
0);
  } /*0rqPFA*/
  ClrPinouts(_outs,0);
  _1st_change= 0; /*97H12851*/
```

```
   _DDiop("algo1itfReceiveOUT",IPCKEY);

   return 0;

} /*ReceiveOUTAll(external-C)*/

/*SendIN(external-C)*/
int
algo1itfSendIN(ipckey)
int ipckey;
{
   IPCKEY= ipckey;
   algo1itfSendINAll();
   algo1itfReceiveOUTAll();

/*RecordInOuts*/
   control_channel_data_real= _ins.control_channel_data_real;
   control_channel_data_int= _ins.control_channel_data_int;
   algo1_channel_data_real= _ins.algo1_channel_data_real;
   algo1_channel_data_int= _ins.algo1_channel_data_int;
   return 0;

} /*SendIN(external-C)*/

/*SendINNotValid(external-C)*/
int
algo1itfSendINNotValid()
{
   _ins.valid= 0;
   MapPinouts(_ins,0);
   sendMSG
   (
     IPCKEY,mtype_INS,
     sizeof(_ins),(char*)&(_ins)
   );
   ; /*97H14956*/
   _DDiop("algo1itfSendINNotValid",IPCKEY);

   return 0;
} /*SendINNotValid(external-C)*/

/*ReceiveOut(external-C)*/
int
algo1itfReceiveOUT(ipckey)
int ipckey;
{
   IPCKEY= ipckey;
   if (_1st_change==0)
   { /*SrqnFV*/
     algo1itfSendINNotValid();
   } /*SrqnFV*/
   algo1itfReceiveOUTAll();

   return 0;

} /*ReceiveOut(external-C)*/

#endif

/*CEnd*/
```

7.1.11 Fichier VHDL généré automatiquement par VCI pour l'encapsulation d'un module logiciel : *moteur1* (*algo1*)

Ce fichier VHDL définit l'interface du module C (*moteur1-algo1*) pour son utilisation comme composant dans la structure VHDL de l'exemple. L'interface correspondant à la description Solar d'un des moteurs (DESIGNUNIT *algo2*), décrite par le format VCI, est convertie par VCI en *entity* VHDL.

```
-- VHDL file
-- file algo1itf.vhd
--          VCI5.05: VHDL-C language interface
--          repport problems to: CA.Valderrama(valderr@verdon.imag.fr)

package algo1itfpkg is
end algo1itfpkg;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
library WORK;
use WORK.algo1itfpkg.all;
use WORK.user.all;

entity algo1itf is
          generic (
          IPCKEY: INTEGER:=1
          );
          port (
          control_channel_data_real:   INOUT    RESOLVED_REAL;
          control_channel_data_int:    INOUT    RESOLVED_INTEGER;
          control_channel_rewr_rdy:    IN       BIT;
          control_channel_wr_req:      OUT      BIT;
          algo1_channel_data_real:     INOUT    RESOLVED_REAL;
          algo1_channel_data_int:      INOUT    RESOLVED_INTEGER;
          algo1_channel_read_req:      OUT      BIT;
          algo1_channel_rewr_rdy:      IN       BIT;
          RST:          IN          BIT;
file
          CLK:          IN          BIT
          );
end algo1itf;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
library WORK;
use WORK.algo1itfpkg.all;
use WORK.user.all;
library SYNOPSYS;
use SYNOPSYS.attributes.all;

architecture CLI of algo1itf is
          attribute FOREIGN of CLI : architecture is
          "SYNOPSYS:CLI";
          attribute CLI_ELABORATE of CLI : architecture is
          "algo1itfOpen";
          attribute CLI_EVALUATE of CLI : architecture is
          "algo1itfEval";
          attribute CLI_ERROR of CLI : architecture is
          "algo1itfError";
          attribute CLI_CLOSE of CLI : architecture is
          "algo1itfClose";
          attribute CLI_PIN of CLK : signal is CLI_EVENT;
          begin
end CLI;

-- algo1itf.vhd end VHDL
```

7.1.12 Fichier *SHELL* (*make*) pour cosimuler le prototype virtuel

Le fichier *shell* généré par S2CV permet de créer l'environnement de cosimulation C-VHDL de l'exemple. Ce fichier sert à valider l'exemple par la cosimulation. Le *shell* exécute la compilation de tous les fichiers (C, VHDL et VCI) générés par S2CV et initialise l'environnement de cosimulation (simulateur VHDL et programmes C).

```
#! /bin/csh

echo '*** RUN-file: controleur_10.run'
echo '*** Solar2C-VHDL(3.02): SOLAR-COSMOS SLS/TIMA/INPG'
echo '*** Report problems to: valderr@verdon.imag.fr'
make -f controleur_10.mak
make -f controleur_10.mak

# MANUAL TANSFORMATION

cp moteur1.ini moteur1.dat
cp moteur2.ini moteur2.dat
xterm -e gnuplot moteur.plot &

# END MANUAL TANSFORMATION

xterm -bg RoyalBlue -fg White -T "C-program [algo2_I]" -name "algo2_I" -e 'algo2_I' &
xterm -bg RoyalBlue -fg White -T "C-program [software_I]" -name "software_I" -e 'software_I' &
xterm -bg RoyalBlue -fg White -T "C-program [algo1_I]" -name "algo1_I" -e 'algo1_I' &
vhdldbx TB_moteur_Structure ; kpc vhdlsim; kpc algo2_I ; kpc software_I ; kpc algo1_I ; kpc gnuplot;  kpc msgsvr ; ipcc; ipcs

echo '*** done controleur_10.run'
```

PROTOTYPE VIRTUEL POUR LA GENERATION DES ARCHITECTURES MIXTES LOGICIELLES/MATERIELLES

Résumé. L'objectif de ce travail de thèse est le développement d'une méthodologie pour la génération rapide d'architectures flexibles et modulaires pour les systèmes distribués. Cette approche, appelé aussi "prototypage virtuel", est une étape essentielle dans le processus de conception conjointe des systèmes mixtes logiciel/matériel. Les approches de recherche dans ce domaine sont motivées par le besoin urgent de prototypes pour valider la spécification, par la disponibilité des outils et des environnements de synthèse pour les parties logicielles et matérielles. Le prototypage virtuel permet à la fois la manipulation du domaine logiciel ainsi que du domaine matériel. Il prend en entrée une architecture hétérogène composée d'un ensemble de modules distribués issu du découpage matériel/logiciel et génère des descriptions exécutables pour des éléments matériels et logiciels.

Ce travail décrit une stratégie de prototypage virtuel pour la co-synthèse (génération des modules matériels et logiciels sur une plate-forme architecturale) et la co-simulation (c'est-à-dire la simulation conjointe de ces deux composants) dans un environnement unifié. Ces travaux définissent également le développement d'un environnement de co-simulation distribué et flexible permettant l'utilisation de différents outils de simulation, de langages, la génération de modèles matériels et logiciels synthésisables et l'ordonnancement des modèles multiprocesseurs sur une architecture monoprocesseur. Cette approche, présentée dans la conférence ED&TC, a obtenu le prix de l'année 1995. Des outils ont été mis en pratique dans l'environnement de conception conjointe Cosmos. Ce travail a aussi fait l'objet d'un transfert de technologie au profit de SGS-Thomson Microelectronics. Les outils développés au cours de cette thèse ont été utilisés pour les projets Européens COMITY (particulièrement utilisé par l'Aérospatiale Missiles à Toulouse et Intracom en Grèce) et CODAC, et par d'autres groupes comme le FZI de l'université de Tübingen et PSA à Paris. Mots Clés : synthèse au niveau système, conception conjointe matériel/logiciel, co-simulation, prototypage, prototypage virtuel.

VIRTUAL PROTOTYPING FOR THE GENERATION OF MIXED HARDWARE / SOFTWARE ARCHITECTURES

Abstract. The objective of this work is to develop a methodology for the generation of flexible and modular architectures for distributed systems. This approach (also called " virtual prototyping ") is an essential stage in the process of joint design (codesign) of mixed software/hardware systems. Virtual prototyping takes as input a heterogeneous architecture made up of a whole of distributed modules resulting from software/hardware partitioning. It generates executable descriptions for software and hardware elements. Research approaches in this field are justified by the evolution of technology, the urgent need for prototypes to validate the specification, and by the availability of tools and synthesis environments for the design of software and hardware parts.

This work describes a strategy of virtual prototyping for the cosynthesis (generation of the modules material and software on an architectural platform) and cosimulation (i.e. the joint simulation of these two kind of components) in a unified environment, the development of a distributed and flexible cosimulation environment allowing the use of several simulation tools and languages, the generation of hardware/software synthesizable models and mono-processor architecture software generation for a set of communicating processes. This approach, presented in the ED&TC conference, got the best paper award in 1995. The tools developed during this thesis were put into practice in the Cosmos codesign environment. One of them was transferred to SGS-Thomson Microelectronics. The tools were also used for the Europeans projects COMITY (particularly used by Aerospace the Missiles in Toulouse and Intracom in Greece) and CODAC, and by other groups like the FZI of the university of Tübingen and PSA in Paris. Key Words: system level synthesis, hardware/software codesign, cosimulation, prototyping, virtual prototyping.

Carlos Alberto Valderrama Sakuyama, Octobre 1998, 143 pages

2-913329-09-8 Version électronique ..... 2-913329-08-X Version papier