HAL
open science

Découpage transformationnel pour la conception de systèmes mixtes logiciel/matériel

G.F. Marchioro

To cite this version:

G.F. Marchioro. Découpage transformationnel pour la conception de systèmes mixtes logiciel/matériel. Micro et nanotechnologies/Microélectronique. Institut National Polytechnique de Grenoble - INPG, 1998. Français. NNT : . tel-00002985

HAL Id: tel-00002985

https://theses.hal.science/tel-00002985

Submitted on 11 Jun 2003

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

THESE

pour obtenir le grade de
DOCTEUR DE L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Discipline: Micro-électronique

présentée et soutenue publiquement
par

Gilberto Fernandes MARCHIORO

le 26 novembre 1998

Titre:

Découpage Transformationnel pour la Conception de Systèmes Mixtes Logiciel/Matériel

---

Directeur de thèse:
M. Ahmed Amine JERRAYA

JURY

M.	Guy MAZARÉ	Président
Mme.	Donatella SCIUTO	Rapporteur
M.	Michel ISRAËL	Rapporteur
M.	Tarek Ben ISMAIL	Examinateur
M.	Ahmed Amine JERRAYA	Examinateur

A ma mère,
à mon père, et
à Camilla.

Remerciements

Les petits ruisseaux font
les grandes rivières

Je voudrais tout d'abord remercier mon directeur de recherche Ahmed Amine Jerraya pour son support, dévouement et surtout pour m'avoir soutenu tout au long de ma recherche. Je remercie aussi monsieur Bernard Courtois pour m'avoir accueilli dans le Laboratoire TIMA et monsieur Guy Mazaré pour m'avoir fait l'honneur de présider ce jury. Je voudrais exprimer ma gratitude à Donatella Sciuto et Michel Israël pour avoir accepté la lourde tâche de rapporter sur cette thèse. Pour la participation au jury, je remercie monsieur Tarek Ben Ismail.

Je remercie mon camarade, Jean-Marc Daveau, pour ses commentaires et ses suggestions qui ont été très utiles pour mon travail et bien sûr pour son agréable compagnie dans le bureau 128. Je remercie Camilla Bom, qui m’a beaucoup aidé à corriger la syntaxe de la thèse. Son travail a été remarquable, même si les termes techniques étaient en dehors de son domaine de travail. Je remercie Nascer Zergainoh pour son aide durant la dernière phase de cette thèse. Je tiens encore à remercier mes amis qui ne m’ont jamais laissé rester dans le laboratoire en dehors des horaires de travail. Je remercie pour leur accueil tous les français. Pendant ces quatre années je me suis senti chez moi, et aussi important que ce travail de recherche a été, ce séjour m'a permis de m'enrichir culturellement. La France est un pays magnifique, ce qui rend le départ d'autant plus difficile.

Il y a autant de personnes à remercier... Finalement, je remercie le gouvernement Brésilien pour avoir financé ma recherche. Je trouve exemplaire ceux qui croient que seulement l’étude et la préparation professionnelle peuvent surmonter les périodes de crise économique du pays. Aussi, je remercie le Capes/Cofecub de son aide financière pour l’achat des matériaux techniques.

J’espère que vous apprécierez cette thèse, que vous apprendrez beaucoup à sa lecture et que vous l’utiliserez.

Sommaire Condensé

Résumé

Ce travail de thèse développe une nouvelle méthodologie pour la conception conjointe du logiciel et du matériel. Cette méthodologie met en pratique une approche transformationnelle de découpage capable de manipuler des systèmes distribués et des architectures multiprocesseurs. Cette approche semi-automatique de découpage réalise le lien entre une spécification au niveau système et une architecture logicielle/matérielle de manière rapide permettant une exploration rapide de l'espace des solutions.

Le principal domaine d'application de ce travail est la conception des architectures distribuées, composées de processeurs logiciels (e.g. programmes) et processeurs matériels (e.g. ASICs). Par rapport à l'approche classique de conception, utilisée dans la plus grande partie des systèmes de conception conjointe existants, cette approche supporte des architectures flexibles composées de processeurs qui communiquent en utilisant un réseau de communication complexe. La principale contribution de ce travail est l'application de l'approche transformationnelle à la conception conjointe logicielle/matérielle d'architectures multiprocesseurs.

L'utilisateur débute le processus de conception avec une spécification au niveau système et une solution architecturale en tête, et réalise des raffinements pour adapter cette spécification initiale à l'architecture. Ce passage se fait par une approche transformationnelle basée sur des raffinements successifs. Chaque étape de raffinement fixe un ensemble de détails permettant de réduire le fossé entre spécification et réalisation. Dans ce modèle, la spécification initiale est donnée en langage SDL. Cette spécification est convertie dans un modèle intermédiaire plus adapté aux algorithmes de synthèse. Le modèle intermédiaire est raffiné pour produire un modèle C/VHDL distribué.

Des méthodes d'estimation ont aussi été développées. Elles permettent à l'utilisateur: de réaliser une exploration rapide de l'espace des solutions; de chercher à réduire le coût de réalisation; de vérifier le résultat des différentes stratégies de raffinement; et d'étudier les possibles critères d'optimisation de l'architecture. La méthodologie présentée ci-dessus est mise en pratique dans un environnement de conception conjointe appelé Cosmos.

Mots-clés : Conception conjointe logicielle/matérielle, découpage transformationnel, estimation de performance, exploration de l'espace des solutions, raffinement de spécifications système.

Abstract

Transformational Partitioning for the Co-Design of Mixed Hardware/Software Systems

This thesis develops a new methodology for hardware/software co-design. It introduces a transformational partitioning approach capable of handling distributed systems and multiprocessor target architectures. This semi-automatic partitioning approach allows to link a system-level specification to a hardware/software architecture in a short design time with fast exploration of the design space.

The main application domain of this work is the design of distributed architectures composed of software processors (e.g. Instruction-level programs) and hardware processors (e.g. ASICs). In contrast to the classical co-design approach implemented by most of existing co-design systems, this approach deals with flexible architectures composed of processors communicating through a complex network. The main contribution of this work is the application of the transformational approach to hardware/software co-design of multi-processor architecture.

The user starts the design process with a system-level specification and an architectural solution in mind and realizes refinements to transform the initial specification into the architectural solution. Each incremental refinement step fixes some implementation detail. In this model, the initial specification is given in the SDL language. This specification is converted to an intermediate model adapted to the synthesis algorithms. The intermediate model is refined and implemented on a distributed C/VHDL model.

The estimation methods developed allows: a fast exploration of the design space; to reduce the implementation costs; to verify the consequence of different refinement strategies; and study the different optimization criteria of architectures. The methodology presented here is implemented in a co-design environment called Cosmos.

Keywords : Hardware/software co-design, transformational partitioning, design space exploration, performance estimation.

Sommaire

Liste des Figures

Liste des Tableaux

Liste des Equations

- Computers in the future may weigh no more than 1.5 tons.

Popular Mechanics, forecasting the relentless march of science, 1949.

- Heavier-than-air flying machines are impossible.

Lord Kelvin, president, Royal Society, 1895.

- Who the hell wants to hear actors talk?

H.M. Warner, Warner Brothers, 1927.

- We don't like their sound, and guitar music is on the way out.

Decca Recording Co. rejecting the Beatles, 1962.

- The abdomen, the chest, and the brain will forever be shut from the intrusion of the wise and humane surgeon.

Sir John Eric Ericksen, British surgeon, appointed Surgeon-Extraordinary to Queen Victoria 1873.

- There is no reason anyone would want a computer in their home.

Ken Olson, president, chairman and founder of Digital Equipment Corp., 1977.

- Everything that can be invented has been invented.

Charles H. Duell, Commissioner, U.S. Office of Patents, 1899.

- Airplanes are interesting toys but of no military value.

Marechal Ferdinand Foch, Professor of Strategy, Ecole Superieure de Guerre.

- This 'telephone' has too many shortcomings to be seriously considered as a means of communication. The device is inherently of no value to us.

Western Union internal memo, 1876.

- 640K ought to be enough for anybody.

Bill Gates, CEO of Microsoft, 1981.

- The wireless music box has no imaginable commercial value. Who would pay for a message sent to nobody in particular?

David Sarnoff's associates in response to his urgings for investment in the radio in the 1920s.

- I think there is a world market for maybe five computers.

Thomas Watson, chairman of IBM, 1943.

- Louis Pasteur's theory of germs is ridiculous fiction.

Pierre Pachet, Professor of Physiology at Toulouse, 1872.

- Drill for oil? You mean drill into the ground to try and find oil? You're crazy.

Drillers who Edwin L. Drake tried to enlist to his project to drill for oil in 1859.

- But what ... is it good for?

Engineer at the Advanced Computing Systems Division of IBM, 1968, commenting on the microchip.

- I have traveled the length and breadth of this country and talked with the best people, and I can assure you that data processing is a fad that won't last out the year.

The editor in charge of business books for Prentice Hall, 1957.

- If I had thought about it, I wouldn't have done the experiment. The literature was full of examples that said you can't do this.

Spencer Silver on the work that led to the unique adhesives for 3-M "Post-It" Notepads.

- The concept is interesting and well-formed, but in order to earn better than a ‘C’, the idea must be feasible.

A Yale University management professor in response to Fred Smith's paper proposing reliable overnight delivery service.
(Smith went on to found Federal Express Corp.)

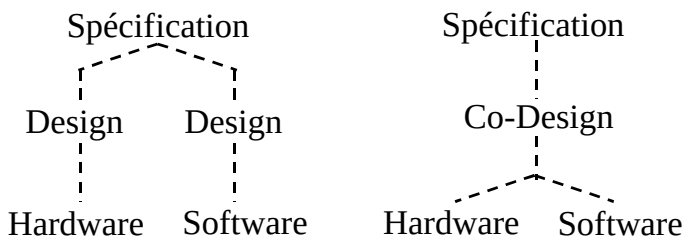
- Professor Goddard does not know the relation between action and reaction and the need to have something better than a vacuum against which to react. He seems to lack the basic knowledge ladled out daily in high schools.

1921 New York Times editorial about Robert Goddard's revolutionary rocket work.

Imagination is more important than knowledge.
Albert Einstein

Chapitre 1

Introduction



Ce chapitre présente les motivations, les objectifs et les contributions de ce travail de recherche au domaine de la conception conjointe logicielle/matérielle des systèmes distribués. Nous constatons que avant l'arrivée de la conception conjointe le concepteur de systèmes séparait le développement du logiciel du développement du matériel. Actuellement, la recherche dans le domaine de la conception conjointe a pour objectif de permettre le développement parallèle du logiciel et du matériel, en étroite interaction, pendant toutes les étapes de la conception. Dans ce contexte, nous présentons une nouvelle méthodologie de conception conjointe basée sur une approche de découpage transformationnel pour des systèmes multiprocesseurs.

1.1 Introduction

La conception au niveau système est définie comme l'ensemble des tâches qui réalisent la conversion d'une spécification du niveau système en une architecture exécutable, composée d'un ensemble de modules interconnectés. Ces modules peuvent être transposés sur des composants logiciels ou matériels. La réalisation matérielle permet généralement une performance plus élevée, par contre, la réalisation logicielle implique un coût plus faible, un temps de conception plus petit et une plus grande flexibilité aux changements.

Une nouvelle génération de méthodes et d'outils est en train d'émerger et de mûrir. Ces méthodes sont capables de manipuler des systèmes mixtes logiciels/matériels au niveau comportemental. Il existe actuellement plusieurs environnements de conception conjointe disponibles qui mettent en pratique ces méthodes. Les principaux systèmes sont [Buc94, Ern95, Gaj94, Sri95, Wol95, The94, Mar97]. La conception conjointe conduit à une augmentation drastique de la productivité parce qu'elle permet la conception de manière concurrente des différents parties d'un système distribué et l'automatisation du processus de découpage. Cependant, la conception conjointe impose quatre défis à l'utilisateur:

1. Le développement de méthodes modernes de conception conjointe a créé un espoir exagéré en promettant des méthodes de découpage automatique. Ces méthodes seraient capables de générer une solution optimale à partir d'une spécification fonctionnelle en réduisant le temps et le coût de la conception. Plusieurs approches efficaces de découpage automatique sont reportées dans la littérature [Bar94, Lag41, Wol94, Vah92, Har95, Rou97]. Malheureusement, la plus grande partie de ces travaux de recherche restreignent le problème à un seul domaine d'application ou bien utilisent des méthodes d'estimation simples. Ces restrictions empêchent l'application de ces méthodes de découpage sur des systèmes complexes comportant plusieurs processeurs logiciels et matériels.
   Les outils de conception conjointe actuels souffrent de la faiblesse des approches de découpage automatique et de la non existence de méthodes d'estimation universelles pouvant être utilisées pendant le découpage pour sélectionner la meilleure solution.
   L'évaluation d'une architecture distribuée est un processus complexe, parce qu'elle dépend d'un grande nombre de critères: l'efficacité, la fiabilité, la maintenance, la portabilité, l'usage, etc.. Chaque critère peut entraîner une longue liste de sous-critères. Par exemple, efficacité comprend la vitesse, le coût, la consommation d'énergie, le volume et la superficie. Pour chaque critère, une valeur de mesure doit être associée. En plus, le poids de ces critères peut être différent pour chaque domaine d'application et chaque technologie utilisée. Par exemple, la fiabilité est le principal critère lorsque l'on manipule des systèmes concernant la sécurité de la vie humaine. Pour d'autres domaines d'application, comme par exemple les systèmes multimédia portables, le coût et la consommation d'énergie devient les critères principaux. Il devient clair qu'il est très difficile de définir une procédure d'évaluation réaliste, même sur un domaine d'application spécifique;
2. L'utilisateur a besoin de comprendre le résultat du découpage automatique pour être capable d'analyser ce résultat. L'outil de conception conjointe doit fournir des facilités capables d'établir la correspondance entre la spécification initiale et l'architecture finale;
3. Normalement, l'utilisateur a une bonne solution en tête lorsqu'il commence le processus de conception conjointe. Cette solution peut être partielle, comme le modèle de communication ou le nombre de processeurs de l'architecture cible. Partant de cela,

l'outil de conception conjointe doit prendre en considération les solutions partielles et permettre le contrôle du processus de conception conjointe par l'utilisateur;

4. La conception de systèmes complexes est généralement un processus interactif où plusieurs solutions doivent être explorées avant de trouver la 'meilleure' solution. La conception conjointe doit permettre une exploration facile de l'espace des solutions. Cette exploration doit être basée sur des critères spécifiques sélectionnés par l'utilisateur.

Tant que ces défis ne sont pas résolus, la conception conjointe continuera à être restreinte à des spécialistes et à des domaines d'application spécifiques.

1.2 Contribution

La principale contribution de cette thèse est le développement d'une approche transformationnelle de conception conjointe basée sur des estimations de performance. Cette approche résoudre les quatre défis de la conception conjointe de systèmes complexes, cités ci-dessous:

1. La méthodologie développée dans cette thèse est basée sur les transformations guidées par l'utilisateur. Il s'agit d'utiliser l'expertise de l'utilisateur pour les prises de décisions du découpage. L'interaction avec l'utilisateur réduit le problème critique du manque de modèle d'estimation. La méthode proposée est une approche réaliste pour traiter la complexité du découpage;
2. La méthodologie, basée sur des transformations, est implantée dans un environnement interactif où l'utilisateur peut analyser le résultat de chaque transformation pour mieux comprendre le résultat du découpage;
3. Dans cette approche, si l'utilisateur a une bonne solution architecturale en tête, il peut arriver à cette solution en utilisant une séquence de primitives de transformation;
4. L'exploration de l'espace des solutions est facilitée par un outil d'estimation rapide. Ceci permet à l'utilisateur d'évaluer plusieurs solutions architecturales en partant de la même spécification d'origine.

Le projet Cosmos est un travail de groupe résultat de plusieurs thèses. Parfois il est difficile d'isoler les contributions de chaque thèse. Les contributions spécifiques de ce travail seront mentionnés explicitement et afin de faciliter la lecture, la méthode Cosmos sera introduite dans son ensemble.

1.3 Plan de la thèse

Ce travail de thèse est divisé en sept chapitres:

- **Chapitre 1 - Introduction**. Ce chapitre présente les motivations, les objectifs et les contributions de ce travail de recherche au domaine de la conception conjointe logicielle/matérielle des systèmes distribués. Il présente une nouvelle méthodologie de conception conjointe basée sur une approche de découpage transformationnel pour des systèmes multiprocesseurs;
- **Chapitre 2 - La Conception Conjointe de Systèmes Mixtes Logiciel/Matériel et le Système Cosmos**. Ce chapitre est organisé en deux parties. La première partie présente les principaux concepts, méthodologies et environnements existants de conception conjointe. Cette introduction a pour objectif de préparer le lecteur à la deuxième partie du chapitre, qui décrit la méthodologie et l'environnement Cosmos;

- **Chapitre 3 - Format Intermédiaire Solar**. Ce chapitre présente le format intermédiaire Solar, utilisé pour la représentation des systèmes pendant les étapes de raffinement et de synthèse. L'utilisation de Solar apporte plusieurs avantages au processus de conception, entre autres: une manipulation et un raffinement aisé de la spécification par les outils du système; et une représentation des aspects langage et architecture de la spécification pendant la conception d'un système mixte;
- **Chapitre 4 - Découpage Transformationnel**. Ce chapitre présente une approche transformationnelle de découpage logiciel/matériel qui couvre le processus de conception conjointe à travers des transformations guidées par l'utilisateur. Cette approche permet une transformation rapide d'une spécification système en une architecture distribuée;
- **Chapitre 5 - Estimation de Performances**. Ce chapitre présente le modèle d'estimation de performance développé au cours de ce travail de recherche. L'estimation de performance permet une exploration rapide de l'espace des solutions, car l'utilisateur dispose d'un ensemble d'informations liées à la réalisation. L'utilisateur peut valider chaque réalisation et peut aussi optimiser sa propre stratégie de raffinement de la spécification;
- **Chapitre 6 - Résultats et Evaluation.** Ce chapitre présente l'application de l'approche de découpage transformationnel Cosmos sur un système appartenant au domaine de la télécommunication: une carte d'interface du réseau ATM. La carte ATM est destinée à lier des programmes d'application à un réseau, en utilisant la couche physique;
- **Chapitre 7 - Conclusion**. Ce chapitre donne les conclusions et les perspectives concertants à ce travail.

La période de réalisation de chacune de ces étapes est présenté dans la section Calendrier de la Thèse. Le support audiovisuel utilisé pendant la soutenance de la thèse est disponible sur l'adresse internet : http://tima-cmp.imag.fr/Homepages/marchior/soutenance/.

Chapitre 2

La Conception Conjointe de Systèmes Mixtes Logiciel/Matériel et le Système Cosmos

Ce chapitre est organisé en deux parties. La première partie présente les principaux concepts, méthodologies et environnements existants de conception conjointe. Cette introduction a pour objectif de préparer le lecteur à la deuxième partie du chapitre, qui décrit la méthodologie et l'environnement Cosmos. Au sein du projet Cosmos, ce travail de recherche a contribué à plusieurs enrichissements: l'interface graphique interactive, la construction de plusieurs outils, l'unification du format intermédiaire, la définition d'une approche d'estimation de performance et la validation de la méthodologie avec la réalisation de plusieurs exemples.

2.1 Position du problème

La conception conjointe du matériel et du logiciel (*co-design*) est une technique qui intègre, dans un même environnement, la conception du matériel et la conception du logiciel. La conception commence à partir d'une spécification au niveau système et conduit à un premier prototype d'architecture, qui respecte les contraintes de la conception. La conception conjointe offre à l'utilisateur un ensemble de méthodes et d'outils pour le prototypage rapide et pour l'évaluation de performance des systèmes complexes.

Les méthodologies regroupent les aspects techniques et les aspects d'organisation de la conception. Elles coordonnent l'utilisation conjointe de plusieurs outils de conception et la coopération de plusieurs aspects liés à tous les niveaux de développement d'un système électronique. Les équipes du logiciel et les équipes du matériel peuvent travailler en parallèle, dans un environnement de coopération et de collaboration qui réduit le cycle de développement de la conception.

La Figure 1 illustre les étapes traditionnelles du cycle de conception d'un système développé dans le milieu industriel. A partir de la spécification informelle du système et des contraintes de la conception, la définition de l'architecture est suivie par le découpage des tâches qui vont être réalisées par les équipes du matériel et du logiciel. L'équipe du matériel réalise la conversion de la spécification dans un format exécutable (langage de description de matériel), puis elle réalise la synthèse et la génération des circuits intégrés, en utilisant des outils de CAO. L'équipe du logiciel est responsable de l'écriture du code qui va être compilé et exécuté sur des processeurs d'usage général. Ensuite, les équipes réalisent l'intégration physique des deux parties. Mais nous constatons que le manque d'interaction entre ces deux équipes, pendant les étapes de développement de la conception et pendant les prises de décisions, peut générer plusieurs problèmes d'intégration.

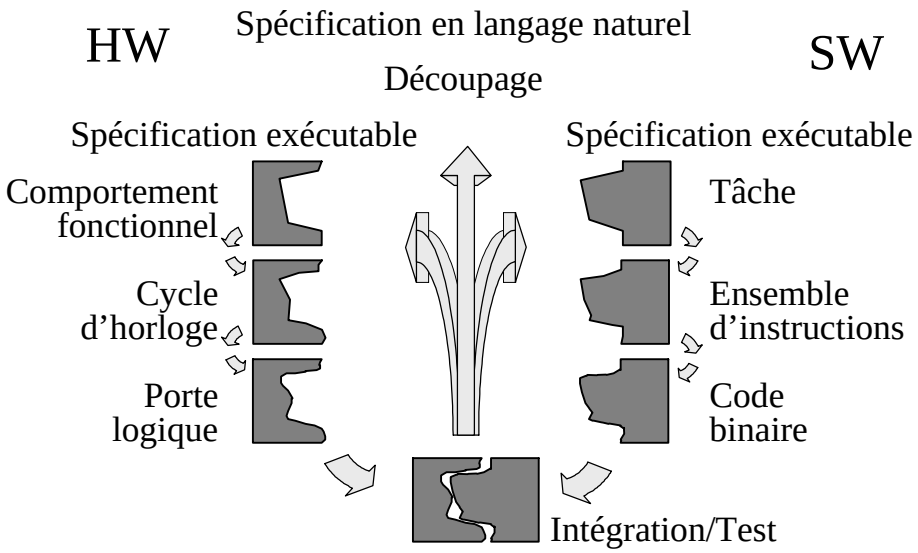


Figure 1: Problème d'intégration logiciel/matériel.

Nous estimons que 80% des problèmes d'intégration peuvent être évités par l'utilisation d'une méthodologie de conception conjointe du logiciel et du matériel. La Figure 2 illustre les étapes de la conception conjointe d'un système. La spécification informelle, avant d'être découpée entre les équipes du logiciel et les équipes du matériel,

est convertie en un format exécutable. La spécification exécutable peut être validée, ce qui permet la détection précoce des erreurs de fonctionnalité. Les équipes du logiciel et du matériel travaillent en parallèle et à chaque étape de conception elles réalisent l'intégration et le test des spécifications. Les opérations de test et d'intégration, appliquées après chaque étape, génèrent une augmentation du temps de conception, mais réduisent la propagation des erreurs à des étapes ultérieures de la conception. L'intégration finale, lors de l'étape de prototypage est réalisée sans difficultés majeures. Nous constatons que la méthodologie de conception conjointe réduit le temps total de conception, puisqu'elle réduit de beaucoup le nombre de retours à des étapes antérieures de conception occasionnés par la détection tardive d'erreurs.

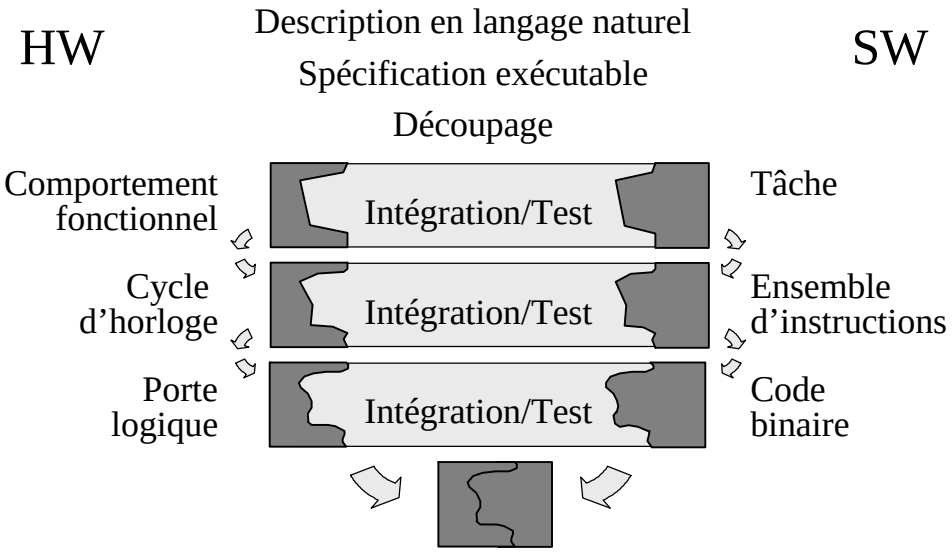


Figure 2: Solution au problème d'intégration: le *Co-design*.

2.2 Etat de l'art dans le domaine de la conception de systèmes mixtes logiciel/matériel

La conception conjointe logicielle/matérielle constitue un thème de recherche qui concerne les professionnels de conception de circuits, de conception de logiciel et de spécification de systèmes. L'état de l'art de la conception conjointe, présenté dans cette section, est basé sur les différentes méthodologies existantes. Ces méthodologies divergent entre elles sur trois aspects: le style de spécification d'entrée, le modèle d'architecture et les étapes de synthèse.

Les lignes du Tableau 1 énumèrent les types d'architectures et les colonnes énumèrent les modèles de spécification.

Modèle de spécification /Modèle d'architecture	Mono-flot	Multi-flot
ASIP	Conception du jeu d'instructions du processeur et conception du compilateur	Ordonnancement des taches, conception du jeu d'instructions et conception du compilateur
Monoprocesseur	Découpage, génération du code et conception du processeur	Ordonnancement des taches, découpage, génération du code et conception des processeurs
Multiprocesseur	Découpage, synthèse de la communication et conception des processeurs	Découpage, synthèse de la communication et conception des processeurs

Tableau 1: Taxonomie des méthodologies.

Le modèle de spécification peut utiliser un langage permettant un seul flot d'exécution (comme un programme C ou un processus VHDL) ou un langage multi-flot, composé par un ensemble de sous-systèmes communicants (comme les systèmes en langage SDL, Esterel ou StateChart). Le choix du langage de spécification dépend principalement du domaine d'application.

Les modèles d'architecture peuvent être classés en trois categories:

- **ASIP** (*Application Specific Integrated Processor*): L'utilisateur commence la conception avec une application, puis il développe un processeur programmable dédié à l'application et traduit l'application en code du processeur [Lan95 Mar95 Pau96]. Le découpage inclut la conception du jeu d'instructions [Alo93]. La fonction 'coût' est normalement liée à la surface, à la vitesse d'exécution et à la consommation;
- **Monoprocesseur** (systèmes synchrones): L'architecture cible est composée d'un processeur logiciel, qui agit comme contrôleur maître, et d'un ensemble d'accélérateurs matériels, qui agissent comme co-processeurs. La plus grande partie des systèmes de conception existants visent une architecture cible de ce type. Deux approches de découpage sont utilisées sur ce modèle: le découpage orienté logiciel [Ern95] et le découpage orienté matériel [Gup93]. La fonction 'coût' est simple et les résultats sont liés à la surface, au coût du processeur logiciel et à la performance du processeur matériel. Vulcan [Gup93], Codes [Buc94], Tosca [Cam94], et Cosyma [Ern95] sont des outils de conception typiques pour des systèmes synchrones;
- **Multiprocesseur** (systèmes distribués): La conception consiste en la transposition d'un ensemble de processus communicants (graphe de tâches) en un ensemble de processeurs interconnectés (graphe de processeurs). Ce schéma englobe la décomposition comportementale, l'allocation des processeurs et la synthèse de la communication [Wol94, Gaj94]. Plusieurs approches de découpage utilisant ce modèle restreignent la fonction coût à des paramètres du type contraintes de temps réel [Chi96, Wol94] et de coût [Gon94].

Cette classification est indépendante de la technologie utilisée. Chaque modèle peut être réalisé sur un circuit intégré unique, sur une carte ou sur plusieurs cartes distribuées géographiquement. Les étapes de synthèse et les algorithmes de découpage sont fortement dépendants du langage de spécification et du modèle d'architecture cible choisi. Sur une architecture monoprocesseur, l'étape principale est la génération du code à partir de la spécification. Sur une architecture multiprocesseur, les étapes de découpage et d'ordonnancement sont nécessaires, et elles doivent être liées à une stratégie efficace d'exploration de l'espace des solutions.

2.2.1 Tableau récapitulatif des méthodologies de conception

Plusieurs grands groupes de recherche ont comme sujet principal la conception conjointe. Les groupes les plus connus sont résumés dans le Tableau 2. Les lignes du tableau énumèrent les différents groupes de recherche et les colonnes énumèrent les caractéristiques.

	Langage de spécification	**Technique de découpage**	**Archi-tecture**	**Validation logicielle/matérielle**
Ptolemy	FDS (Flot de données synchrone)	Découpage automatique orienté logiciel; optimisation du temps global d'exécution	Mono-processeur	Processus légers communicant à travers des portes (*sockets*) ; communication synchrone
Cosyma	C^x (C parallèle)	Découpage automatique orienté logiciel, basé sur des estimations dynamiques de performance	Mono-processeur	Processus communicant à travers le modèle CSP; utilisation de portes (*sockets*)
SpecSyn	SpecChart	Découpage automatique basé sur la technique de groupement	Multi-processeur	Processus communicants à travers le modèle CSP
Co-Saw	CSP	Découpage interactif basé sur la technique de groupement	Mono-processeur	*Sockets* du système UNIX
Cobra	VHDL	Découpage non-automatique orienté matériel	Mono-processeur	Simulation dirigée au VHDL. Le logiciel est décrit en langage d'assemblage
Vulcan II	HardwareC	Découpage automatique orienté matériel	Mono-processeur	Co-simulation au niveau assembleur; communication dirigée par des événements discrets
Rassp	VHDL	Découpage non-automatique orienté matériel	Mono-processeur	Co-simulation à base de modèles détailles des processeurs et ASICs au niveau des portes
Lycos	VHDL ou C	Découpage manuel	Mono-processeur	Processus communicants
Tosca	SpeedChart	Découpage manuel	Mono-processeur	Validation par prototypage physique
Codes	SDL, StateChart	Découpage manuel	Mono-processeur	Validation par prototypage physique
MCSE	Formalisme MCSE	Découpage interactif basé sur des estimations de performance	Multi-processeur	Co-simulation au niveau modèle, suivie par la simulation VHDL et C++
CoWare	POPE	Synthèse des interfaces et raffinement des processeurs	Multi-processeur	Validation par prototypage physique

Tableau 2: Méthodologies de conception conjointe.

2.2.2 Etapes de conception

Le processus de conception conjointe s'adapte parfaitement aux étapes du cycle de développement des systèmes électroniques. La conception conjointe englobe l'ensemble des étapes de conception qui, à partir d'une spécification exécutable, conduisent à un premier prototype virtuel du système. La Figure 3 illustre le contexte de la conception conjointe qui exécute fondamentalement deux opérations: le découpage logiciel/matériel et la synthèse de la communication. Une fois ces étapes terminées, la conception continue jusqu'à la synthèse architecturale et l'intégration du prototype physique.

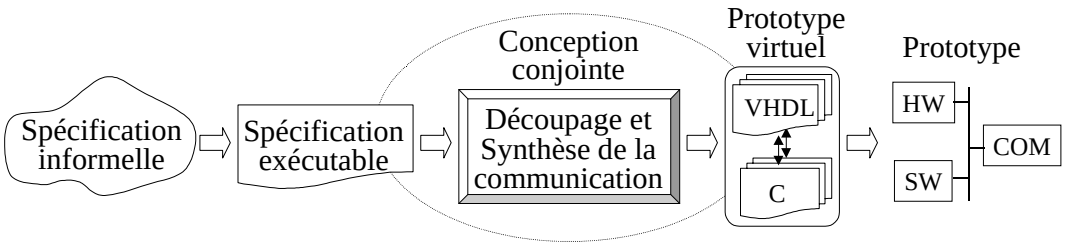


Figure 3: Flot de conception.

Les principales étapes de conception sont résumées ci-dessous:

1. La **Modélisation du système** spécifie la fonctionnalité désirée du système et ses restrictions. La fonctionnalité est décomposée en modules qui, ensemble, forment le modèle conceptuel du système. La spécification peut se baser sur un langage de description au niveau système ou sur une autre description qui fournit un modèle exécutable. La validation de la description utilise des simulateurs ou d'autres techniques de vérification. Le résultat de cette étape est la génération d'une spécification fonctionnelle, libre de tous les détails de réalisation;
2. Le **découpage logiciel/matériel basé sur l'estimation de performance** explore les alternatives de conception pour identifier celles qui s'adaptent le mieux aux contraintes du système. Cette étape réalise la transposition des fonctions du système sur des composants logiciels et matériels. Les composants sont des ASICs, des processeurs programmables, des FPGAs, des mémoires et des bus. La synthèse de la communication traduit les primitives de communication à un niveau de détail accepté par les outils de synthèse;
3. Le **prototypage** est divisé en deux étapes: le prototypage virtuel, qui intègre les techniques de co-simulation et de co-synthèse; et le prototypage physique, qui produit un premier exemplaire du système. La création d'une réalisation pour chaque composant est réalisée par l'intermédiaire des techniques de conception logicielle/matérielle classiques. Le prototypage consiste à effectuer la synthèse du matériel et la génération du code (ou microcode) logiciel à partir des descriptions générées par les étapes précédentes.

La Figure 4 présente les étapes et le flot de conception, en utilisant une méthodologie de conception conjointe. Ces étapes sont détaillées dans les sections qui suivent.

2.2.3 Modélisation du système

La première étape de la conception d'un système électronique est la modélisation de sa fonctionnalité. La question est de savoir comment décrire la fonctionnalité d'un système à un niveau de détails suffisant permettant de prévoir son comportement complet, sans ambiguïté.

Normalement, la modélisation utilise une stratégie de décomposition de la spécification du système (en langage naturel) en modules fonctionnels (en un langage exécutable). La spécification de chaque module exécutable peut utiliser un langage different, bien adapté au module. Nous constatons que les systèmes qui imposent un langage d'entrée unique pour la description de tous les modules restreignent le domaine d'application de la conception. L'avantage de l'utilisation de plusieurs modèles et langages de spécification est qu'ils permettent une représentation facile de différentes visions du système. Ces visions sont nécessaires aux différentes phases de la conception et aux différents domaines d'application.

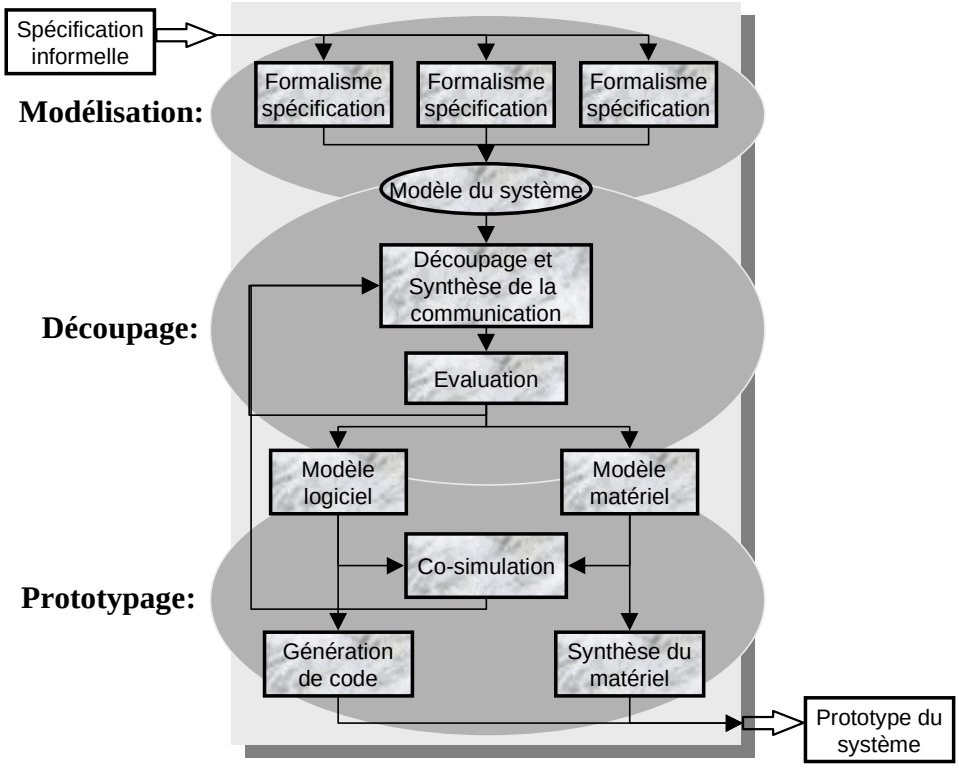


Figure 4: Etapes et flot de la conception conjointe.

La difficulté de l'étape de spécification vient du fait que les langages naturels, utilisés dans la première description du système, sont souvent ambigus et incomplets. Ce qui réduit leur capacité à détailler les fonctions du système. Il est nécessaire que l'utilisateur soit très attentif pendant la génération du modèle exécutable.

Le modèle exécutable permet la représentation du système par un ensemble de sous-systèmes ou de modules fonctionnels. Les modules doivent offrir les caractéristiques suivantes:

- être formels et non ambigus;
- être complets, de façon à ce qu'ils décrivent la totalité du comportement désiré;
- permettre une compréhension facile pour la correction, la maintenance et la réutilisation;
- être modulaires, avec une interface et un comportement bien définis.

Dans les étapes qui suivent, cette fonctionnalité est transposée sur les composants physiques de l'architecture. Les spécifications au niveau système sont basées sur quatre concepts: la concurrence, la hiérarchie, la communication et la synchronisation.

- La **concurrence** permet l'exécution parallèle d'opérations, d'unités fonctionnelles ou des processus, suivant la granularité de la spécification [Gaj94b]. Nous pouvons classifier la concurrence, suivant l'ordre de l'exécution des opérations, en deux classes: les spécifications qui imposent explicitement l'ordre d'exécution des opérations parallèles (modèles orientés contrôle, comme les CSP et MEF concurrents); et les spécifications où l'ordre d'exécution des opérations est définie par la dépendance entre les données (modèles orientés données, comme les graphes de flot de données);
- La **hiérarchie** est nécessaire pour manipuler la complexité. Elle permet la décomposition de systèmes en sous-systèmes plus petits et plus facile à traiter. Nous distinguons deux types de hiérarchie: la hiérarchie du comportement qui permet de

cacher des parties du comportement par l'utilisation de constructions comme les procédures et les sous-états; et la hiérarchie structurelle qui permet la décomposition du système en un ensemble de sous-systèmes;

- La **communication** permet l'échange des données et des informations de contrôle entre des modules concurrents [And91]. Nous distinguons deux types de communications: la communication par échange de message, où les sous-systèmes exécutent des opérations spécifiques pour envoyer et recevoir des données; et la communication par mémoire partagée où chaque communication nécessite la lecture ou l'écriture sur une position spécifique de la mémoire pour échanger des données. L'appel des procédures à distance (RPC) est un modèle hybride qui permet la modélisation des deux approches [And93];
- La **synchronisation** coordonne la communication et l'échange d'information entre les sous-systèmes concurrents [And93, Boc93]. Nous distinguons deux modes de synchronisation de la communication: le mode synchrone (protocole du type *rendez-vous* ou *mémoire partagée* avec des sémaphores); et le mode asynchrone (protocole du type Fifo ou *mémoire partagée* avec des moniteurs).

Ces concepts sont généralement supportés par les langages au niveau système, mais la vision ou l'interprétation de ces concepts change d'un langage à l'autre.

2.2.4 Decoupage logiciel/matériel

L'étape de découpage a pour objectif de diviser le comportement d'un système en un ensemble de partitions. Chaque partition doit s'exécuter soit en logiciel, soit en matériel [Vah92]. Les différentes partitions vont contenir une ou plusieurs fonctions provenant de la spécification initiale. Ces fonctions seront implantées sur des processeurs programmables pour la partie logicielle et sur des ASICs pour la partie matérielle. Les partitions logicielles seront compilées pour le processeur et les partitions matérielles sont synthétisées en un composant d'émulation ou directement en circuit intégré. Le nombre de partitions générées par le découpage correspond au nombre de processeurs de l'architecture cible, comme illustre la Figure 5.

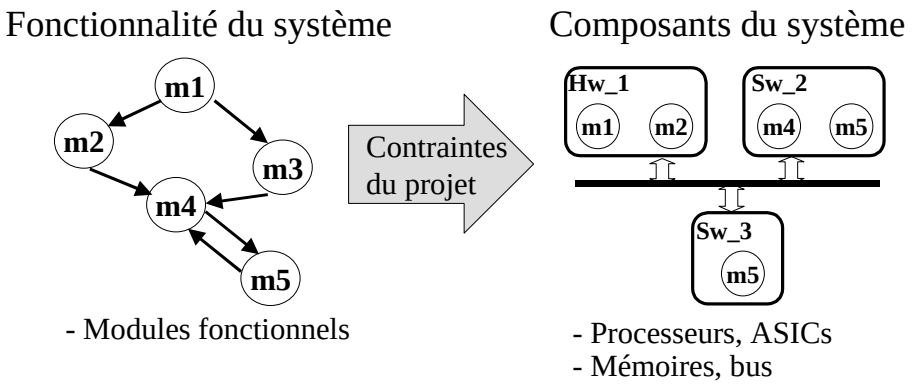


Figure 5: Découpage logiciel/matériel.

Lors du découpage, il s'agit de répondre aux questions suivantes:

- Combien de processeurs faut-il pour exécuter une spécification donnée ?
- Avec quelle technologie (logicielle ou matérielle) doit-on réaliser chaque processeur ?
- Sur quels processeurs doit-on exécuter chaque fonction d'une spécification ?
- Quel est le surcoût de la communication engendré par le découpage ?

- Quel est le gain (performance, surface, communication, etc.) obtenu après une opération de raffinement ?

L'utilisateur (ou l'algorithme automatique de découpage) a la difficile tâche de trouver un bon compromis entre les paramètres de surface, de performance et de temps de communication des différentes partitions. Généralement, les partitions qui font partie du chemin critique de l'application, et qui demandent un temps de réponse court, sont réalisées en matériel. Les autres partitions sont réalisées en logiciel afin de réduire le coût de la réalisation. Une fonction peut être affectée à plusieurs processeurs et plusieurs fonctions peuvent être affectées à un processeur unique. Les différentes alternatives de réalisation sont analysées par des résultats d'estimation. Pendant le découpage, non seulement les partitions, mais l'architecture cible aussi est raffinée.

Les étapes traditionnelles de découpage sont: la sélection des composants, qui fixe le nombre et le type des processeurs de l'architecture; le découpage fonctionnel, qui attribue des fonctions aux composants; et le calcul des caractéristiques de la réalisation.

Le bon choix de la granularité des partitions est contraint par la minimisation des critères d'interconnexion, de surface et de temps de réponse du système. Ces trois critères sont normalement corrélés et l'optimisation d'un de ces critères induit la dégradation des autres. Le découpage peut être vu comme un problème d'optimisation de performance du système, en prenant en considération le critère 'coût'.

2.2.4.1 Algorithmes et méthodes de découpage

Il existe trois approches de découpage qui font la relation entre les modules fonctionnels de la spécification et les composants structurels de l'architecture:

- La **recherche exhaustive,** qui consiste à vérifier toutes les combinaisons modules/composants possibles pour un système. Cette technique est impraticable dans la réalité, car la relation module/composant génère un espace de solutions exponentiel ($\text{composants}^{\text{modules}}$);
- La **recherche interactive** consiste à trouver une solution qui respecte les contraintes du système avec l'intervention manuelle de l'utilisateur. Cette méthode est basée sur des prises de décision effectuées par l'utilisateur. Le raffinement dispose d'un environnement du type boîte à outils, composé par un ensemble de primitives de transformation et de découpage. Ces primitives exécutent des opérations simples sur la spécification, sans changer la fonctionnalité d'origine. Le critère de choix des primitives à appliquer est laissé à l'utilisateur, qui utilise son expérience pour trouver une solution qui respecte les contraintes. La technique interactive est très répandue dans le milieu industriel, dans la mesure où il n'existe pas actuellement de méthodes d'estimation universelles pour guider les approches automatiques;
- La plus grande partie des systèmes existants utilisent une approche de découpage basée sur la **recherche automatique**. Cette approche met en pratique une heuristique pour trouver une solution qui s'adapte aux contraintes. L'heuristique a pour objectif de réduire l'espace des solutions et par conséquence le nombre de réalisations candidates au test. Cette approche, comme l'approche de recherche interactive, peut ne pas trouver la solution optimale coût/performance. Finalement, les systèmes automatiques utilisent une fonction coût pour évaluer les partitions. Normalement ces systèmes sont restreints à des modèles d'architecture monoprocesseur et à un domaine d'application spécifique.

Les algorithmes de recherche automatique, utilisés par les systèmes existants de conception conjointe, exécutent une des techniques suivantes (ou une variation de ces techniques):

- **Groupement hiérarchique** (*Hierarchical clustering*). Ce type d'algorithme groupe les objets d'une partition s'ils génèrent une réduction de la fonction coût du système;
- **Groupement multi-étages** (*Multi-stage clustering*). Ce type d'algorithme est semblable au précédent, à la difference, qu'il utilise une fonction coût différente à chaque étape du groupement;
- **Migration par groupe** (*Groupe migration*). Ce type d'algorithme déplace les objets qui, dans différentes partitions, génèrent une réduction du coût global du système;
- **Recuit simulé** (*Simulated Annealing*). Ce type d'algorithme est semblable au précédent, avec une option supplémentaire qui accepte des déplacements générant une augmentation momentanée de la fonction coût quand la réduction n'est pas possible;
- **Evolution génétique** (*Genetic Evolution*). Ce type d'algorithme traite les réalisations comme des générations et les étapes de raffinement comme des méthodes d'évolution de la nature, à base de sélection, croisement et mutation;
- **Programmation linéaire entière** (*Integer Linear Program*). Réalise le découpage en utilisant un algorithme à base de formulations de programmation linéaire.

Pour être complète, la classification des méthodes de découpage doit se baser sur les caractéristiques suivantes: le niveau d'abstraction de la description d'entrée, la granularité de la spécification, la flexibilité d'allocation des composants du système, les métriques d'estimation, l'algorithme de découpage, le flot de contrôle du découpage, et la technique d'interaction avec l'utilisateur.

Actuellement, il existe trois cultures différentes de chercheurs qui travaillent sur le découpage:

- La **culture logicielle** [Ern93] qui commence la conception avec une spécification initiale entièrement logicielle et cherche à migrer du code vers le matériel. Les parties critiques du système sont identifiées et affectées à une réalisation matérielle;
- La **culture matérielle** [Gup93] qui part d'une spécification initiale entièrement matérielle. Les parties non critiques sont identifiées et affectées à une réalisation logicielle, ce qui permet la réduction du coût de la réalisation;
- La **culture système** [Gaj95] ne se limite pas à un type particulier de spécification d'entrée. Les différentes parties d'une spécification sont affectées à une réalisation logicielle ou matérielle qui satisfait les contraintes de conception.

2.2.5 Synthèse de la communication

La synthèse de la communication permet la modélisation et la réalisation du schéma de communication utilisé par les langages de spécification. Cette étape réalise l'échange d'informations entre les modules de la description, i.e. la communication entre les processeurs de l'architecture.

La spécification d'entrée de l'étape de synthèse de la communication est composée d'un ensemble de processeurs abstraits représentant les différentes partitions, résultant du découpage. Ces processeurs communiquent entre eux par un réseau conceptuel de communication. Le résultat de la synthèse est un ensemble de processeurs interconnectés par des signaux qui peuvent être contrôlées par une unité de protocole. La synthèse de la communication transforme un système composé d'un ensemble de processus communicants en un système composé d'un ensemble de processeurs interconnectés.

En général, la synthèse de la communication est divisée en deux parties: la **sélection du protocole** et la **synthèse de l'interface** [Dav96b, Fre97]. La sélection du

protocole consiste à sélectionner, parmi les canaux existants dans la bibliothèque, celui qui s'adapte mieux à la communication entre les unités concernées. La communication peut utiliser un protocole qui contient une fonction de résolution des conflits d'accès ou un protocole qui réalise l'échange des signaux directement entre les unités. Après l'allocation du canal, l'étape de synthèse de l'interface répartie les éléments nécessaires à la réalisation du canal sur l'ensemble du système.

2.2.6 Exploration de l'espace des solutions

L'exploration de l'espace des solutions est une phase importante du processus de conception d'un système. L'objectif de l'exploration est de trouver la solution de découpage, de synthèse de communication et d'architecture qui respecte les contraintes de la conception et qui représente un bon compromis entre les critères de surface et de performance.

La tâche d'évaluation d'une réalisation n'est pas triviale, car elle demande un grand effort et une quantité de calcul considérable. La méthodologie de conception conjointe doit mettre en pratique une technique qui réalise l'évaluation des réalisations avec un niveau de précision et une vitesse de calcul acceptable. Ces deux critères sont contradictoires, puisque plus de fidélité demande un temps de calcul plus long. L'exploration de l'espace des solutions englobe plusieurs étapes de la conception, comme le découpage, l'allocation et l'estimation. Normalement, plusieurs interactions sont nécessaires entre ces étapes avant que le concepteur arrive à une solution efficace.

La tâche d'exploration est complexe, car chaque choix de découpage peut influencer l'allocation. Par exemple, une solution de découpage et d'allocation destiné à la réalisation d'un système avec un nombre réduit de processeurs de grande performance peut être plus coûteuse que l'utilisation d'un nombre plus élevé de processeurs moins performants. Des constatations de ce type peuvent sembler normales pour l'utilisateur mais elles sont moins triviales pour les algorithmes de conception qui peuvent se trouver piégés dans des extremum locaux de la fonction d'évaluation.

Normalement, l'utilisateur a une bonne idée de l'architecture la plus appropriée au système. Cette idée peut se baser sur son expertise, sur la volonté de réutiliser des composants pré-conçus, sur des tendances du marché, ou sur d'autres raisons difficiles à modéliser. Les approches automatiques existantes réalisent une exploration restreinte et partielle de l'espace des solutions, une fois que la spécification d'entrée, l'architecture cible et l'algorithme d'estimation imposent de fortes contraintes.

2.2.7 Prototypage virtuel et physique

Le prototypage virtuel est l'étape qui génère le code exécutable pour chaque processeur abstrait. Les descriptions générées sont simulables et peuvent être utilisées par les outils de synthèse classique. Chaque sous-système est traduit séparément et le système résulte en une architecture hétérogène représentée par le code C et le code VHDL. Les codes correspondant aux processus de communication sont normalement obtenus à partir de la bibliothèque de réalisation des protocoles.

La simulation du prototype virtuel peut avoir comme vecteur de test d'entrée les mêmes valeurs utilisées lors de la vérification de la spécification initiale. Cette forme de vérification, par simulation, détecte les changements du comportement initial du système par les étapes de synthèse. La simulation des descriptions C et VHDL permet également

la vérification des propriétés du système, comme la synchronisation des protocoles de communication et le décompte du nombre de cycles d'exécution des processus.

Le prototypage physique produit une architecture qui exécute ou émule la spécification initiale. Dans cette étape, les compilateurs et les outils de synthèse doivent être adaptés à l'architecture cible. Les composants programmables (FPGAs) peuvent être utilisés pour exécuter la partie matérielle.

2.3 La méthodologie Cosmos

Cosmos est une méthodologie et un environnement de conception conjointe. Les principales caractéristiques de Cosmos sont :

- Le **domaine d'application** de Cosmos couvre les systèmes distribués et communicants;
- Le **modéle de spécification** est traduit dans un format intermédiaire utilisé pendant les étapes de raffinement. Cosmos utilise un modèle appelé Solar, basé sur des machines d'états finis étendues qui communiquent à travers des appels de procédure à distance;
- Le **processus de synthèse** détermine quelle partie de la méthodologie est exécutée automatiquement par les outils et quelle partie est exécutée manuellement par l'utilisateur. Dans la méthodologie Cosmos, les transformations sont exécutées automatiquement par les outils et les décisions de conception sont prises par l'utilisateur. La flexibilité du processus de synthèse est assurée par l'interaction continue avec l'utilisateur et par l'utilisation d'une bibliothèque de modèles de communication;
- Cosmos utilise comme **entrée la langage SDL** et **produit un modèle distribué en C/VHDL**;
- Le **flot de conception** du découpage utilise une approche transformationnelle.

2.3.1 Les modèles de conception

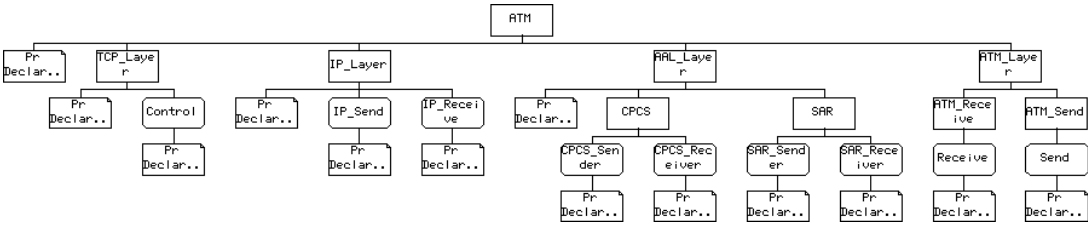
Cette section présente les cinq modèles qui composent la méthodologie Cosmos. Ces modèles sont : SDL, Solar, le modèle de communication, C/VHDL et l'architecture cible.

2.3.1.1 La spécification du système en SDL

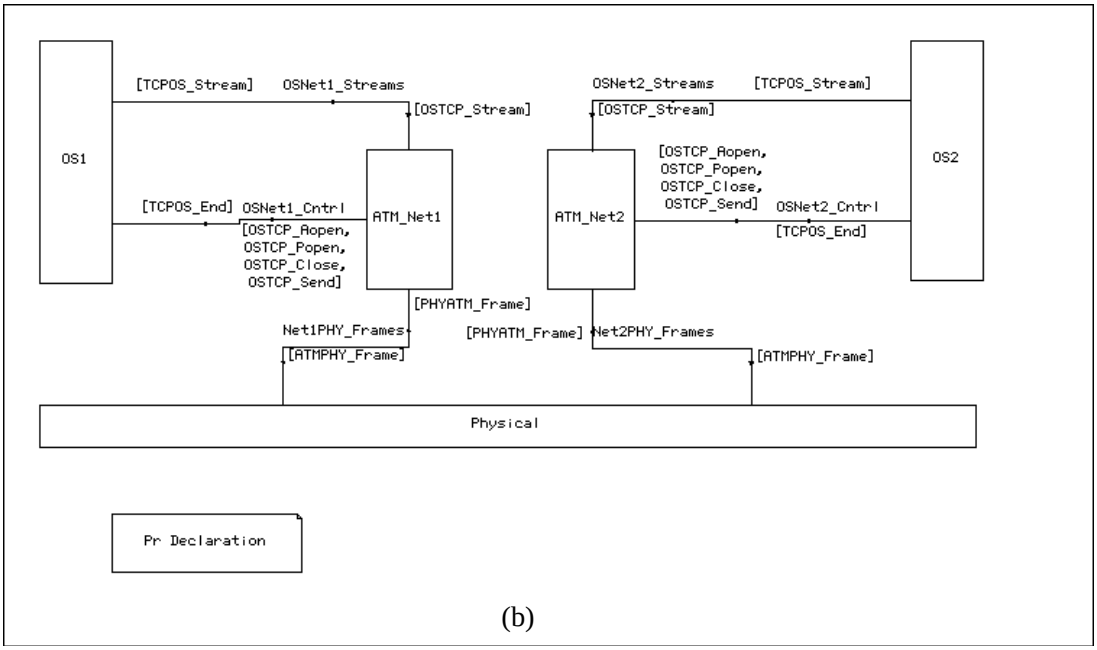
Le langage SDL (*Specification and Description Language*, standard ITU [Bel91, Fae94, Sar97]) est dédié à la modélisation et à la simulation des systèmes temps réel, distribués et dédiés à la télécommunication. Un système décrit en SDL est vu comme un ensemble de processus concurrents qui communiquent par des signaux. SDL supporte la description de systèmes complexes au niveau système. L'utilisateur dispose d'un environnement pour la capture de la spécification et la validation de sa fonctionnalité. La manipulation de la description peut être faite de façon graphique ou textuelle. Les concepts de base pour la description des systèmes sont:

- La **structure** : La structure statique d'un système est décrite par une hiérarchie de blocs. Un bloc peut contenir d'autres blocs formant une structure en forme d'arbre, comme montre la Figure 6.a. Un bloc peut aussi contenir un ensemble de processus, qui décrivent le comportement des blocs terminaux. Les processus sont connectés avec d'autres processus ou avec la frontière du bloc par des routes de signaux. Les blocs

sont connectés entre eux par des canaux (voir Figure 6.b). SDL supporte aussi des caractéristiques dynamiques orientées logiciel, comme la création dynamique de processus et l'adressage dynamique. La Figure 6.c présente la structure d'une spécification SDL dans le format textuel.



(a)



(b)

```
SYSTEM NET;

SYNTYPE ATM_INDEX = INTEGER CONSTANTS 1:48
ENDSYNTYPE ATM_INDEX;
SIGNAL  PHYATM_Frame(ATM_FRAME), ATMPHY_Frame(ATM_FRAME);

  BLOCK ATM_Net1;
    SUBSTRUCTURE;
      CHANNEL OSTCP_Streams
        FROM ATM TO ENV WITH TCPOS_Stream;
        FROM ENV TO ATM WITH OSTCP_Stream;
      ENDCHANNEL OSTCP_Streams;
      CHANNEL OSTCP_Cntrl
        FROM ENV TO ATM WITH OSTCP_Aopen, OSTCP_Popen;
        FROM ATM TO ENV WITH TCPOS_End;
      ENDCHANNEL OSTCP_Cntrl;
      CHANNEL ATMPHY_Frames
        FROM ATM TO ENV WITH ATMPHY_Frame;
        FROM ENV TO ATM WITH PHYATM_Frame;
      ENDCHANNEL ATMPHY_Frames;
      CONNECT Net1PHY_Frames AND ATMPHY_Frames;
      CONNECT OSNet1_Cntrl AND OSTCP_Cntrl;
      CONNECT OSNet1_Streams AND OSTCP_Streams;
      BLOCK ATM REFERENCED COMMENT '#ref ATM.pr';
    ENDSUBSTRUCTURE;
  ENDBLOCK ATM_Net1;
```

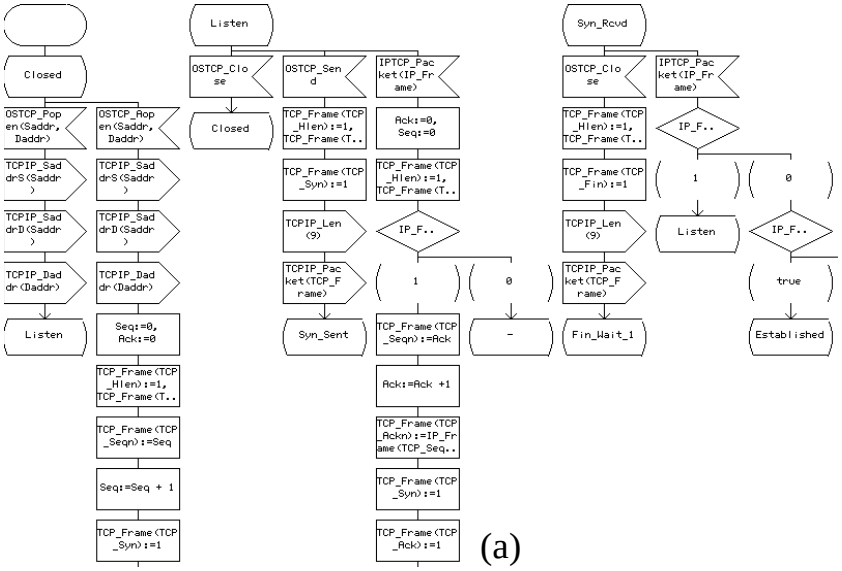
(c)

Figure 6: Structure d'une spécification SDL.

- Le **comportement** : Le comportement d'un système est décrit par un ensemble de processus concurrents et autonomes. Un processus est formé par une machine d'états finis qui communique avec d'autres processus de façon asynchrone par des signaux. Chaque processus dispose d'une file d'attente qui sert de mémoire tampon aux signaux qui arrivent. Les signaux sont retirés de la file d'entrée par le processus dans l'ordre

d'arrivée (Fifo). L'arrivé d'un signal attendu, dans la file d'entrée, active une transition et le processus peut exécuter l'ensemble des actions de son comportement. Les actions sont du type: manipulation de variables, appel de procédures et émission des signaux. Le signal reçu détermine la transition à exécuter. Quand un signal active une transition, ce signal est enlevé de la file d'entrée.

La Figure 7 présente une partie du comportement d'un processus. L'état *Start* représente l'état d'initialisation <<défaut>>. *Input* représente la garde d'une transition. Cette transition va être activée aussitôt que le signal spécifié est enlevé de la file d'entrée. *Task* représente une action et *Output* émet un signal avec ses paramètres éventuels.



(a)

```
PROCESS Control;
  DCL Saddr       INTEGER;
  DCL Daddr       INTEGER;
  TIMER T1;
   START ;
    NEXTSTATE Closed;
  STATE Listen;
    INPUT OSTCP_Close;
      NEXTSTATE Closed;
    INPUT OSTCP_Send;
       TASK TCP_Frame(TCP_Syn):=1;
       OUTPUT TCPIP_Len(9);
       OUTPUT TCPIP_Packet(TCP_Frame);
      NEXTSTATE Syn_Sent;
    INPUT IPTCP_Packet(IP_Frame);
      TASK Ack:=0, Seq:=0;
      DECISION IP_Frame(TCP_Syn);
      ( 1 ):
          TASK TCP_Frame(TCP_Seqn):=Ack;
          TASK Ack:=Ack +1;
          TASK TCP_Frame(TCP_Ackn):=IP_Frame(TCP_Seqn)+1;
          TASK TCP_Frame(TCP_Syn):=1;
          TASK TCP_Frame(TCP_Ack):=1;
          OUTPUT TCPIP_Len(9);
          OUTPUT TCPIP_Packet(TCP_Frame);
          NEXTSTATE Syn_Rcvd;
        ( 0 ):
          NEXTSTATE -;
      ENDDECISION;
  ENDSTATE;
  . . .
```

(b)

Figure 7: Comportement d'une spécification SDL.

En SDL, une variable appartient à un processus spécifique et peut seulement être accédée par ce processus. La synchronisation entre les processus est faite par l'échange des signaux. Chaque processus dispose d'une adresse d'identification propre (Pid). Chaque signal contient l'adresse du processus d'origine. L'adresse du processus de destination est nécessaire seulement quand la destination du signal ne peut pas être déterminée

statiquement. Les signaux normalement contiennent des données de types et de tailles variés.

Les signaux sont transférés entre les processus par des routes de signaux ou par des canaux. Si les processus appartiennent à des blocs différents, les signaux doivent traverser des canaux. Dans ce cas, les canaux exécutent une opération de routage des signaux et la communication subit un délais non déterministe. La communication à travers des routes de signaux est instantanée.

2.3.1.2 Le modèle Solar

Le modèle de description Solar est utilisé principalement pendant le découpage transformationnel [Mar97]. Les différentes étapes du raffinement utilisent Solar comme une représentation intermédiaire. Chaque étape réalise des transformations sur un modèle Solar.

L'utilisation d'un format intermédiaire rend la conception conjointe indépendante du langage de spécification. De cette façon, plusieurs langages peuvent être utilisés comme spécification d'entrée. Chaque langage ayant des caractéristiques propres et étant adapté à un domaine d'application propre. En plus, un format intermédiaire est défini de façon à être plus adapté aux algorithmes de synthèse.

Solar supporte des concepts de communication de haut niveau, comme les canaux et les variables globales partagées entre processus concurrents. Il est possible de modéliser la plus grande partie des schémas de communication au niveau système, comme le passage de message, le partage de ressources et d'autres protocoles plus complexes. Solar modélise les constructions du niveau système d'une manière orientée vers la synthèse. Le modèle combine deux concepts puissants du niveau système:

- les **machines d'états finis étendue** (EFSM) pour la description du comportement;
- l'**appel de procédure à distance** (RPC) pour la spécification de la communication de haut niveau.

Un système en Solar est représenté comme étant une hiérarchie d'unités de conception communicantes, comme l'illustre la Figure 8. Le modèle Solar est détaillé dans le chapitre 3.

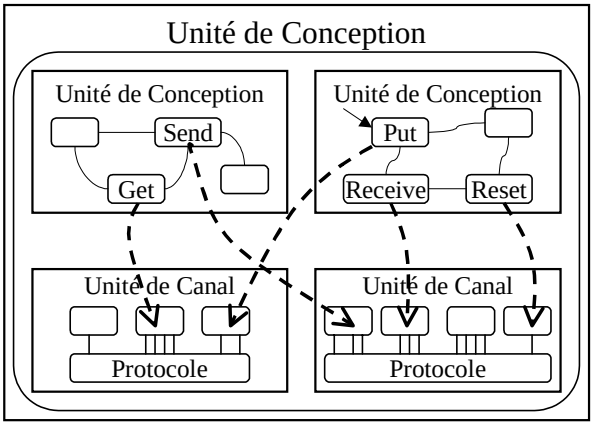


Figure 8: Modèle Solar - EFSMs et RPC.

2.3.1.3 Le modèle de communication

Le modèle de communication, pour la synthèse au niveau système, permet la représentation et la réalisation des schémas de communication utilisés par les langages de

spécification [Dav96, Cam93, Nar95]. Ce modèle est suffisamment général pour représenter différents schémas de communication, comme l'échange de messages ou les mémoires partagées, et il permet une réalisation efficace des spécifications de communications au niveau système.

Au niveau système, un module est représenté par un ensemble de processus qui communiquent par des canaux abstraits. Un canal abstrait est une entité capable d'exécuter un schéma de communication requis, activé par un mécanisme d'appel de procédure, comme l'illustre la Figure 9. Ce modèle dispose de primitives de communication de haut niveau qui sont utilisées pour la communication entre les processus.

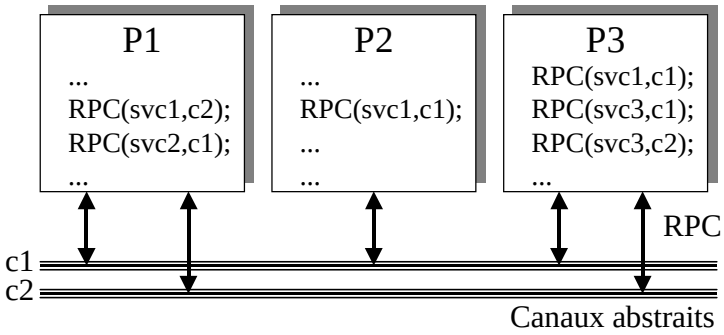


Figure 9: Communication au niveau système.

Conceptuellement, l'unité de communication est un objet qui exécute une ou plusieurs primitives de communication avec un protocole spécifique. Une unité de communication est composée d'un ensemble de primitives, d'une interface et d'un contrôleur.

Les accès aux canaux sont contrôlés par un ensemble fixe de primitives de communication (*svc1*, *svc2* et *svc3* de la Figure 9) activées par des appels de procédures à distance (RPC [Bir84]). Un processus qui désire communiquer à travers un canal, fait un appel de procédure à distance aux primitives de communication du canal. Une fois l'appel de procédure réalisé, la communication est exécutée indépendamment du processus qui a activé l'appel. Les primitives de communication sont transparentes aux processus et sont exécutées par le canal, de sorte que la réalisation de ces primitives est cachée au processus appelant. Cette approche permet de modéliser la communication entre processus en utilisant des schémas de communication de haut niveau, qui font abstraction de la réalisation du canal.

Les primitives d'accès disponibles pour chaque canal sont définies comme des procédures standards. Chaque canal peut avoir un ensemble différent de primitives de communication. Les primitives les plus courantes sont *send_msg*, *get_msg*, *send_int* et *get_int* qui réalisent l'envoi et la réception de messages et de valeurs entières. Les primitives de communication forment la seule partie visible d'un protocole.

L'utilisation des appels de procédures à distance permet la séparation entre la spécification de la communication et la spécification du système. Dans l'approche Cosmos, la structure d'entrée/sortie et les protocoles sont disponibles dans une bibliothèque de composants de communication. La Figure 10 illustre les protocoles de communication qui font partie de la bibliothèque standard de Cosmos. Le protocole *Rendez Vous* est destiné à l'interconnexion de deux unités seulement. Sa réalisation est simple et il n'utilise pas de contrôleur externe dédié à la communication, puisque le comportement du protocole est distribué entre les primitives d'accès. Les protocoles

Mémoire Partagée et *Fifo* utilisent un contrôleur de communication qui sauvegarde et gère les messages et les données.

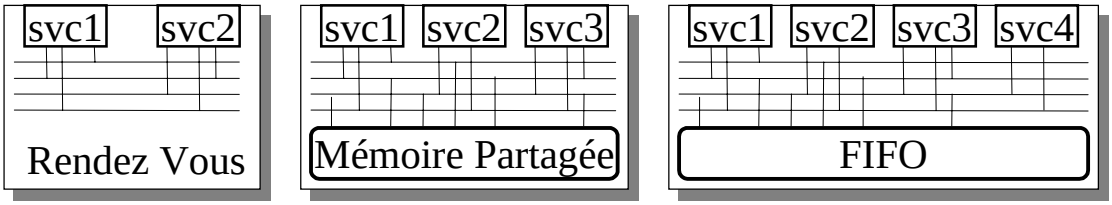


Figure 10: Bibliothèque d'unités de communication standard.

Une unité de communication (ou protocole) est une abstraction d'un composant physique. Les unités de communication sont sélectionnées dans la bibliothèque et allouées pendant l'étape de synthèse de la communication.

2.3.1.4 Le modèle C/VHDL

L'architecture générée, aussi appelée prototype virtuel, est une architecture hétérogène composée d'un ensemble de modules distribués. Les modules matériels sont représentés en VHDL et les modules logiciels son représentés en C. Les éléments de communication, originaires de la bibliothèque, sont représentés en VHDL ou en C suivant leur réalisation. Le prototype virtuel est un modèle exécutable du système utilisé pour la génération du prototype final.

L'étape de transposition de l'architecture produit une architecture qui exécute ou émule la spécification initiale. Cette étape utilise le prototype virtuel pour la co-synthèse (attribution des modules matériels et logiciels sur la plate-forme architecturale) et pour la co-simulation (la simulation conjointe des composants matériels et logiciels).

La partie centrale du prototypage virtuel est la modélisation de la communication entre le matériel et le logiciel. L'utilisation d'un modèle de communication modulaire, modèle Solar, permet l'évolution de la communication pendant les étapes de synthèse. De cette façon, plusieurs représentations d'un même objet peuvent être utilisées. Pendant la spécification système et le découpage, la communication est représentée de façon abstraite par des canaux. Après la synthèse de la communication, la communication est réalisée par un protocole, des procédures et des signaux d'interconnexion. Le prototypage virtuel modélise la communication comme étant un ensemble de processus logiciels ou matériels. Après la réalisation, les unités de communication sont associées à des programmes existants dans le cas de modules logiciels et des composants décrits en VHDL dans le cas matériel.

Pour permettre l'utilisation des unités de communication dans les différentes étapes de la conception, il est nécessaire de décrire les procédures de communication en différentes versions logicielles et matérielles.

2.3.1.5 L'architecture cible

Cosmos utilise une architecture modulaire et flexible. Le modèle général de cette architecture, représenté dans la Figure 11, est composé par trois types de composants: les composants logiciels qui exécutent le code C; les composants matériels qui exécutent des descriptions VHDL; et les composants de communication. Ce modèle sert de plate-forme aux systèmes mixtes logiciel/matériel. Les modules de communication viennent d'une

bibliothèque. Cette bibliothèque est composée de modèles de communication simples ou complexes.

Le modèle d'architecture proposé est suffisamment général pour représenter une grande classe de plates-formes logicielle/matérielle existantes. Ce modèle permet différentes réalisations d'un système mixte (architectures distribuées et plusieurs modèles de communication). Une architecture typique est composée de plusieurs modules matériels, de plusieurs modules logiciels et de modules de communication.

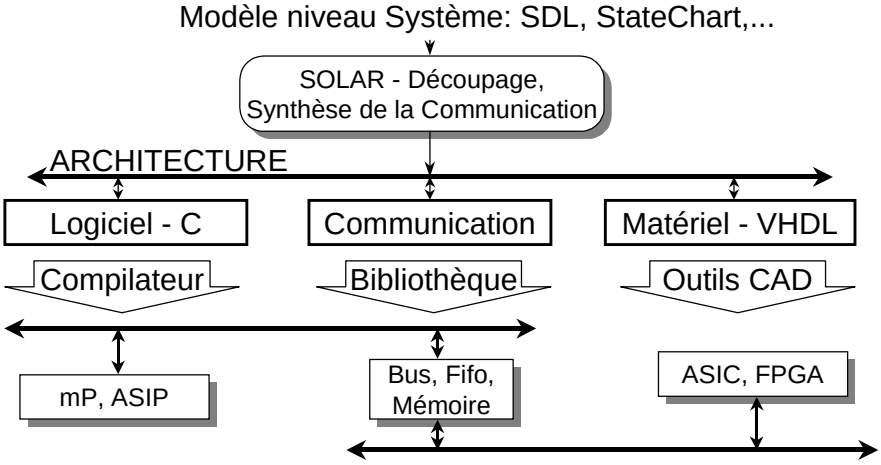


Figure 11 : Modèle d'architecture cible.

2.3.2 L'environnement Cosmos

L'interaction avec le système Cosmos se fait à partir d'un environnement graphique à base de fenêtres. Cet environnement graphique apporte à Cosmos tous les avantages d'une interface conviviale pour l'utilisateur. L'utilisateur peut facilement activer des primitives de transformation, visualiser le format intermédiaire de façon graphique et examiner rapidement la structure de la réalisation. La construction de cet environnement a été nécessaire puisque seulement un environnement graphique peut concrétiser l'utilisation d'une méthodologie semi-automatique de découpage basée essentiellement sur l'interaction continue avec l'utilisateur.

Pendant la construction de l'environnement graphique, plusieurs objectifs ont été pris en considération :

- **L'intégration des outils** qui font partie de la méthodologie Cosmos (Sas, Partif, Estim et S2cv). Les outils doivent être activés automatiquement, d'une façon transparente à l'utilisateur, et les données d'entrée et de sortie doivent être gérées par l'environnement;
- Permettre **l'unification du format intermédiaire** Solar. Tous les outils qui font partie d'une version de l'environnement doivent utiliser la même version du format intermédiaire Solar. L'unification oblige les constructeurs d'outils à respecter les standards imposés par la méthodologie;
- Permettre **la création des versions fermées** du système (environnement et outils). Les versions permettent la distribution, la détection des bogues et le contrôle des modifications des outils;
- **Faciliter l'apprentissage et la compréhension** de la méthodologie Cosmos ;
- Permettre à l'utilisateur de **visualiser clairement et facilement la structure hiérarchique et le comportement** d'une spécification en Solar ;

- Permettre une **vérification facile du fonctionnement des outils**. Ainsi, les erreurs sur les descriptions d'entrées et de sortie peuvent être facilement trouvées ;
- Permettre une facile **exploration de l'espace des solutions**. La bonne adéquation de l'environnement à la méthodologie permet à l'utilisateur d'essayer plusieurs alternatives de réalisation;
- Permettre à l'utilisateur de **guider le flot de conception**. L'utilisateur doit avoir le contrôle sur le flot de raffinement, ce qui lui permet de guider le processus à la réalisation désirée;
- **Présenter de façon claire les résultats** des opérations invoquées par l'utilisateur. La présentation des résultats de l'estimation de performance doit être directe et compréhensible par l'utilisateur.

L'environnement graphique Cosmos est implanté sur station de travail du type SUN-Sparc et utilise la bibliothèque graphique Open Look (Xview [Sun98]). L'interaction avec l'utilisateur se fait par l'intermédiaire d'un écran graphique, couleur ou monochrome, du type Console SUN ou terminal X. L'interface est basée sur des fenêtres et l'affichage de toutes les informations relatives à la conception permet une parfaite interaction avec l'utilisateur.

Les prochaines versions de Cosmos vont utiliser le package de fonctions Xclass'95 [Xcl95]. Ce paquet graphique a l'avantage d'être moins dépendant du système d'exploitation et du système de fenêtres que le paquet utilisé actuellement. Xclass est un paquet graphique basé sur Xlib et écrit en C++. Ce paquet est de domaine public et la présentation des fenêtres ressemble au style Win'95.

Il est aussi possible de lancer les outils Cosmos en dehors de l'environnement graphique, par la ligne de commande ou par l'utilisation d'un *script* prédéfinie (*shell Unix*).

2.3.2.1 L'interaction avec l'utilisateur

L'interaction avec l'utilisateur est basée sur des fenêtres permettant l'affichage des informations et l'entrée de commandes par l'utilisateur. La première fenêtre présentée à l'utilisateur est la fenêtre *Principale*. Dans cette fenêtre l'utilisateur indique le fichier qui contient la spécification initiale du système. Cette spécification, dans la version actuelle de Cosmos, doit être modélisée en SDL ou Solar. Lorsque la spécification d'entrée est présentée en SDL, l'environnement active l'outil *Sas* pour réaliser la traduction au format intermédiaire Solar. La Figure 12 présente la hiérarchie des fenêtres de l'interface graphique.

Cosmos dispose d'une fenêtre graphique de la spécification. Cette visualisation est réalisée dans la fenêtre *Source*. Chaque fois qu'une primitive de raffinement est appliquée, la spécification résultante de la transformation est présentée dans la fenêtre *Destination*. La fenêtre *Historique de développement* trace l'évolution du processus de transformation. Les composants des bibliothèques, i.e. les protocoles de communication, les processeurs logiciels, les réalisations matérielles et les mémoires, sont présentés dans la fenêtre *Bibliothèque*. Cette fenêtre permet la sélection manuelle des ressources pendant l'étape de transposition de l'architecture. La fenêtre *Estimation* affiche: la hiérarchie structurelle de la spécification; la transposition des composants de la bibliothèque aux processus; et les informations liées à la performance des processus. Les informations détaillées de performance sont disponibles dans la fenêtre *Métriques d'estimation*.

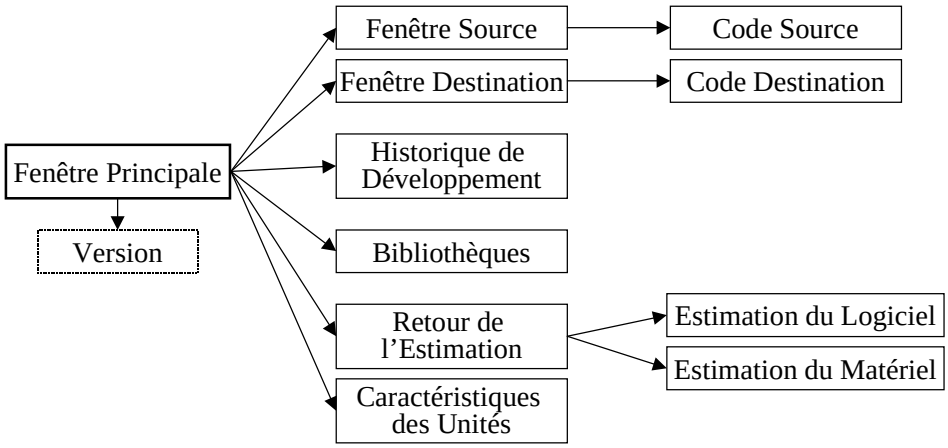


Figure 12 : Hiérarchie des fenêtres de l'interface Cosmos.

Le dialogue interface/utilisateur peut s'effectuer à différentes étapes de la synthèse:

- Pour interpréter les commandes. Par exemple, pour permettre l'activation des primitives de transformation par des boutons de l'interface;
- Pour montrer le résultat des commandes. Par exemple, la fenêtre *Résultat* permet à l'utilisateur d'examiner les conséquences des transformations;
- Pour permettre à l'utilisateur d'intervenir sur certaines opérations. Par exemple, dans la synthèse de la communication, l'utilisateur a la possibilité de réaliser manuellement l'allocation du protocole à partir de la bibliothèque;
- Pour vérifier si la commande activée par l'utilisateur est permise. L'outil est sensible au contexte et n'autorise qu'un sous ensemble d'actions dans chaque instant. Par exemple, le découpage d'une unité de conception est inactivé quand le comportement de l'unité n'est pas parallèle.

La Figure 13 présente la fenêtre *Principale* de l'environnement Cosmos. Cette fenêtre affiche l'ensemble des primitives de raffinement. L'application de chaque primitive dépend du contexte d'utilisation de la transformation. Les contextes d'utilisation sont: la décomposition fonctionnelle, la réorganisation de la structure, la synthèse de la communication et l'estimation de performance. La fenêtre *Version* montre le logo du système et la version des outils.

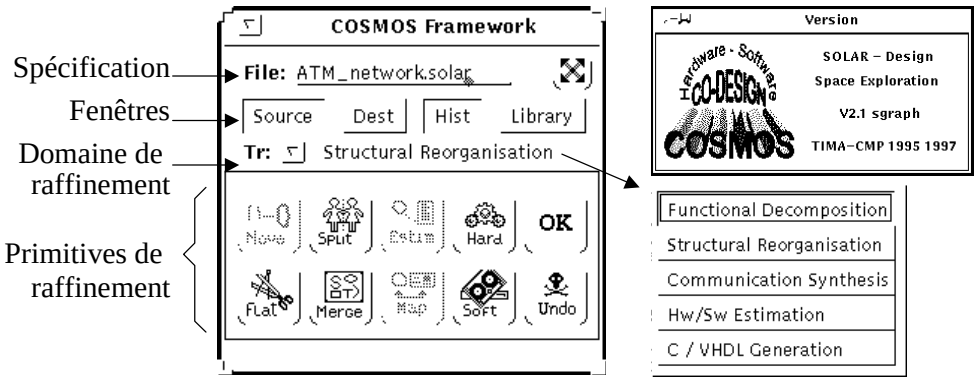


Figure 13: Fenêtre *Principale*.

L'appendice E détaille la fonctionnalité des fenêtres de l'environnement Cosmos.

2.3.2.2 Les outils du système

L'environnement Cosmos dispose de cinq outils de raffinement, illustrés dans la Figure 14. Ces outils sont modulaires, de façon à ce qu'ils puissent être utilisés à travers l'interface graphique ou par des commandes de ligne. La figure liste les outils dans l'ordre normal d'activation.

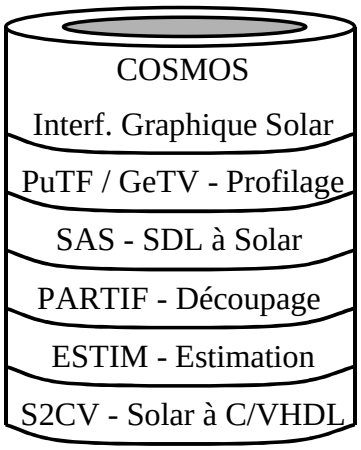


Figure 14: Outils internes de Cosmos.

Ci-dessous suit une brève description de chaque outil:

- **Cosmos** est l'environnement graphique qui intègre les outils du système. Cet environnement contient l'interface Solar;
- **Putf/Getv** sont les outils de calcul de la fréquence d'exécution des instructions. Ils contrôlent l'ajout et la vérification des indicateurs de trace [Mar97a];
- **Sas** est le traducteur de SDL vers Solar [Dav97]. Cet outil utilise l'interface de programmation ObjectGeode (API [Obj97]) pour la génération du code Solar, à partir d'une spécification en SDL;
- **Partif** est l'outil de découpage interactif [Mar97]. Cet outil exécute l'ensemble des primitives de découpage transformationnel et de synthèse de la communication;
- **Estim** est l'outil qui fournit une estimation quantitative des unités du système [Mar97a]. Les mesures d'estimation sont le résultat du calcul de la fréquence d'exécution, et de la caractérisation des processeurs et réalisations. Cet outil à pour objectif d'aider l'utilisateur à prendre des décisions pendant le découpage logiciel/matériel;
- **S2cv** est le traducteur de Solar en C/VHDL [Val97]. La traduction est nécessaire quand l'utilisateur arrive à la réalisation désirée. Cet outil génère automatiquement les interfaces entre les codes C et VHDL, ce qui permet la co-simulation distribuée.

2.3.2.3 Le flot d'exécution de Cosmos

La Figure 15 illustre le flot de conception typique de la méthodologie Cosmos. L'utilisateur débute la conception avec une spécification initiale du système et une architecture cible en tête. La spécification est donnée normalement en SDL, mais d'autres langages de spécification sont partiellement acceptés, comme C et VHDL. La spécification est validée à l'aide d'un simulateur fonctionnel. Cette spécification suit une étape de calcul de la fréquence d'exécution, où l'utilisateur introduit des données représentatives afin d'obtenir des valeurs de fréquence d'exécution proches à l'exécution réelle.

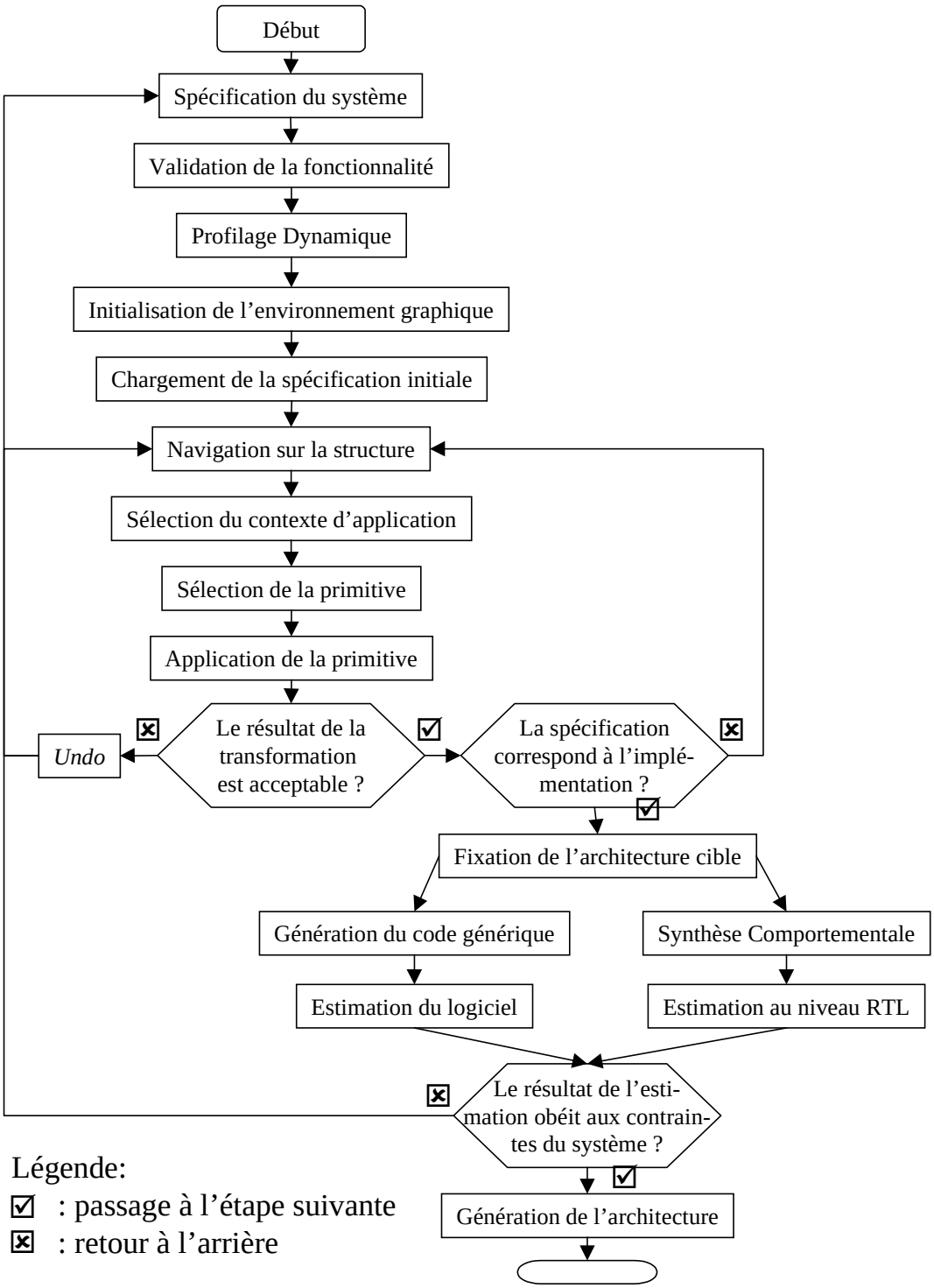


Figure 15: Le flot de conception avec Cosmos.

L'utilisateur peut utiliser l'environnement Cosmos pour visualiser graphiquement la spécification. L'utilisateur réalise le raffinement de la spécification à l'aide des primitives de transformation pour arriver à la réalisation désirée. A tout moment, l'opération de retour à des étapes antérieures de raffinement est possible.

Quand l'utilisateur obtient le découpage désiré, l'étape suivante est le choix de l'architecture cible avec l'allocation des processeurs logiciels, matériels et des mémoires. L'estimation du code logiciel utilise un format générique pour la représentation de la spécification. Ce format générique est une représentation de la spécification à un niveau plus proche des langages des processeurs.

L'estimation du côté matériel utilise la synthèse comportementale suivie de l'estimation au niveau RTL. L'estimation fournit des valeurs approximatives des caractéristiques de la réalisation.

2.3.2.4 La traduction de SDL en Solar

La traduction du modèle SDL au modèle Solar ne pose pas de difficultés conceptuelles, une fois que la distance cognitive entre ces deux modèles a été réduite. Il existe des similitudes entre les concepts de base de SDL et de Solar, ce qui facilite la traduction automatique. Cette traduction est divisée en trois parties: la traduction du comportement, la traduction de la structure et la traduction de la communication. La traduction de la communication exige une attention spéciale car SDL décrit la communication à un niveau très abstrait, alors que le modèle Solar, même s'il appartient à un haut niveau d'abstraction, impose des restrictions de représentation. Les détails de cette traduction sont donnés en [Dav97b].

La traduction des aspects de la communication obéit aux règles suivantes (voir Figure 16):

- **L'expansion de la file d'attente implicite**. Chaque processus SDL possède une file d'attente implicite, dans laquelle tous les messages à destination de ces processus sont stockés. La traduction génère un canal abstrait qui représente la file d'attente de chaque processus;
- **La mise à plat de la communication**. La communication en SDL réalise le routage des signaux, ce qui normalement génère une réalisation coûteuse. La solution choisie est une mise à plat de la communication. Cela impose néanmoins quelques restrictions pendant la description du système mais elle permet une réalisation plus efficace du schéma de communication SDL;
- **Les signaux SDL**. Chaque canal alloué pendant la synthèse de la communication dispose de ses propres primitives de communication, permettant de lire et d'écrire des signaux sur un canal à travers des RPCs;

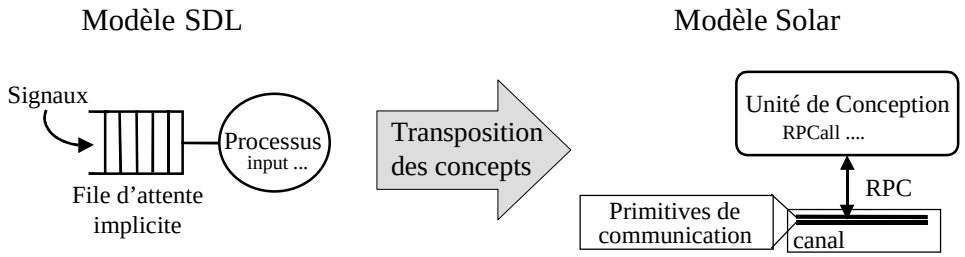


Figure 16: Traduction de SDL en Solar.

2.3.2.5 La bibliothèque de communication

La bibliothèque de communication de Cosmos permet l'allocation et la réutilisation des protocoles de communication. La réutilisation signifie qu'un protocole défini pour une application peut être réutilisé dans d'autres applications. De la même façon, le résultat de la conception conjointe peut être une unité réalisant un protocole complexe destiné à faire partie de la bibliothèque de communication, comme la carte du réseau ATM (voir chapitre 6).

Avec Cosmos, l'utilisateur n'a pas besoin de décrire les protocoles dédiés à chaque application, sauf lorsque l'application a besoin d'un protocole particulier qui n'est pas

disponible dans la bibliothèque. Dans ce cas, l'utilisateur peut adapter à son besoin un protocole de la bibliothèque. Ce nouveau protocole peut être inclut à la bibliothèque s'il est assez général pour être réutilisé par d'autres applications.

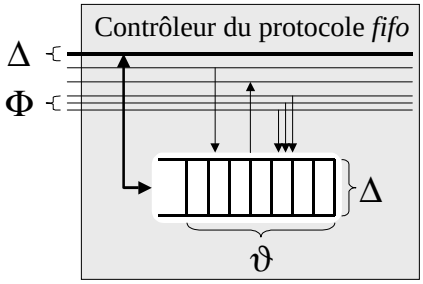
Dans la version actuelle de Cosmos, l'écriture des protocoles est faite de façon manuelle, en code Solar. Pour aider l'écriture des protocoles, un éditeur graphique du format intermédiaire Solar est prévu pour les prochaines versions de Cosmos. Avec l'éditeur, l'utilisateur va pouvoir décrire de façon plus rapide le comportement des protocoles.

L'utilisation de la bibliothèque des protocoles est divisée en trois étapes :

- L'**abstraction** est l'étape où l'utilisateur, ou l'administrateur Cosmos, décrit les caractéristiques essentielles et le comportement des protocoles qui vont faire partie de la bibliothèque;
- L'**allocation** est l'étape qui sélectionne un protocole de la bibliothèque à partir de ses caractéristiques et restrictions;
- L'**intégration** est l'étape qui adapte à la spécification le protocole alloué. Cette adaptation peut modifier l'interface, les paramètres et le comportement du protocole.

Les canaux sont composés d'une partie fixe, son comportement, et d'une partie variable, ses paramètres. Par exemple, pendant l'étape d'abstraction du protocole *fifo*, l'attention a été dirigée à la paramétrisation de plusieurs caractéristiques, comme: la taille de la mémoire *fifo*, le type des éléments de la *fifo* et le nombre d'unités interconnectées. Cette paramétrisation facilite la réutilisation.

Avec la paramétrisation, l'utilisateur dispose d'une plus grande liberté lors de l'étape d'allocation. Pendant l'étape d'intégration, les paramètres reçoivent des valeurs fixes pour adapter le protocole à l'environnement. La Figure 17 présente les paramètres référents au protocole *fifo* spécifiés par l'utilisateur pendant l'étape de l'abstraction du protocole.



Paramètres:
Φ: nombre de processus interconnectés
ϑ: taille de la mémoire *fifo*
Δ: type des données

Figure 17: Paramètres de réalisation du protocole *fifo*.

La valeur Φ correspond au nombre de processus interconnectées au canal. Le modèle de communication Cosmos assume que chaque canal connecte un ou plusieurs processus sources de données à un processus destination de données. La valeur Δ indique le type de la donnée (entière, flottante, définie par l'utilisateur, etc.). Cette valeur définit la taille du bus de communication et la taille des éléments qui composent la mémoire *fifo*. La valeur ϑ indique le nombre d'éléments de la mémoire *fifo*.

Les valeurs des paramètres Φ e Δ sont calculées automatiquement par la primitive d'allocation. Le paramètre ϑ (taille de *fifo*), par contre, reçoit une valeur défaut (égale à 512). L'utilisateur peut manuellement changer cette valeur.

2.4 Conclusion

Ce chapitre a présenté les principaux concepts, étapes et défis de la conception conjointe. Cette introduction a permit au lecteur d'appréhender le contexte actuel de la conception conjointe afin de mieux pouvoir comprendre et critiquer la méthodologie et l'environnement Cosmos, présenté dans la deuxième partie du chapitre.

Les principales contributions de l'auteur à Cosmos ont étés:

- Le développement d'un environnement d'intégration des outils appartenant à la méthodologie Cosmos (Sas, Partif, Estim et S2cv). Les outils sont activés automatiquement et d'une façon transparente à l'utilisateur. Les données d'entrée et de sortie sont gérées par l'environnement;
- L'unification du format intermédiaire Solar. Tous les outils qui font partie d'une version de l'environnement utilisent la même version du format intermédiaire Solar. L'unification a obligé les constructeurs d'outils à respecter les standards imposés par la méthodologie;
- La création des versions fermes du système (environnement et outils). Les versions fermes ont permis la distribution de Cosmos, la détection des bogues et le contrôle des modifications des outils;
- La construction des outils de calcul de la fréquence d'exécution (PuTF et GetTV), les outils d'estimation de performance du logiciel (Estim) et le complément de l'outil de découpage (Partif);
- La conception de plusieurs exemples, comme: le contrôleur de bras de robot et la carte du réseau ATM;
- La définition et l'écriture des canaux qui composent la bibliothèque de communication;
- L'inclusion des nouvelles étapes dans le cycle de conception comme le calcul de la fréquence d'exécution, la transposition de l'architecture et les estimations de performance;
- La définition d'une approche d'estimation de performance.

Ces nouvelles caractéristiques ont permis:

- De rendre plus facile l'apprentissage et la compréhension de la méthodologie Cosmos;
- La visualisation claire et facile de la structure hiérarchique et du comportement du format Solar ;
- Une exploration facile de l'espace des solutions. La bonne adéquation de l'environnement à la méthodologie permet à l'utilisateur d'essayer plusieurs alternatives de réalisation;
- De mieux guider le flot de conception. L'utilisateur a le contrôle sur le flot de raffinement, ce qui lui permet d'aboutir à la réalisation désirée;
- De présenter de façon claire les résultats des opérations invoquées par l'utilisateur. La présentation des résultats de l'estimation de performance est directe et compréhensible par l'utilisateur.
- La vérification facile du fonctionnement des outils. Les erreurs sur les descriptions générées sont facilement identifiées.

Chapitre 3

Format Intermédiaire Solar

Some dream of doing great things,
while others stay awake and do them.

Ce chapitre présente le format intermédiaire, appelé Solar, utilisé pour la représentation des systèmes pendant les étapes de raffinement et de synthèse. Solar est considéré comme format intermédiaire puisqu'il n'a pas l'intention d'être un nouveau langage de représentation au niveau système. L'utilisation de Solar apporte plusieurs avantages au processus de conception, entre autres: une manipulation et un raffinement aisé de la spécification par les outils du système; et une représentation des aspects langage et architecture de la spécification pendant la conception d'un système mixte.

3.1 Introduction

La représentation des systèmes complexes, composés de parties matérielles et logicielles, impose plusieurs défis de conception. La communauté scientifique travaille actuellement sur plusieurs projets qui ont le même objectif : remplir l'espace conceptuel existant entre le niveau système et les outils de conception de circuit intégré. Si l'on prend leur modèle de représentation comme point de départ, nous pouvons classer ces conceptions dans quatre catégories :

- Les systèmes qui utilisent un **langage de spécification standard au niveau système**:
  - ☞ Exemples de langages: SDL, Esterel, Lotos, StateCharts, CSP, Estelle ;
  - ☝ Avantages: ces langages sont bien définis et documentés; ils disposent normalement d'environnements de spécification et de simulation déjà développés;
  - ☟ Inconvénients: ces langages nécessitent un travail difficile pour les adapter aux algorithmes de synthèse; le raffinement de ces langages peut imposer des limitations sur le format de représentation; certains concepts manipulés par ces langages ont une transcription difficile au niveau matériel;
- Les systèmes qui utilisent un **langage de spécification pour le matériel ou le logiciel**:
  - ☞ Exemples de langages: VHDL, Verilog, C++, Java;
  - ☝ Avantages: ces langages disposent d'un grand ensemble d'outils pour aider à la spécification et à la vérification; la documentation est complète;
  - ☟ Inconvénients: ces langages ont des restrictions pour la représentation des concepts au niveau système, comme par exemple la communication, la synchronisation et le contrôle de processus; la spécification n'est pas indépendante de la réalisation;
- Les systèmes qui créent un **nouveau langage de spécification**:
  - ☞ Exemple de langage: SpecChart;
  - ☝ Avantage: le langage peut être bien adapté à la méthodologie;
  - ☟ Inconvénients: il est nécessaire de développer des outils pour manipuler ce nouveau langage, comme par exemple les simulateurs et les éditeurs graphiques; langage spécifique et non standard;
- Les systèmes qui créent un **format intermédiaire**:
  - ☞ Exemples de formats intermédiaires: BIF, SIF, CDFG, SLIF;
  - ☝ Avantages: le format intermédiaire permet une représentation facile des concepts système; il permet une manipulation facile par des outils de synthèse; il permet une conversion facile de et vers les langages standards de description matériel et logiciel;
  - ☟ Inconvénient: la manipulation du format intermédiaire ne peut pas être faite manuellement et nécessite des outils;

L'analyse des avantages et inconvénients des quatre approches montre clairement que la solution optimale pour la représentation des systèmes complexes en vue, de la synthèse, est le couplage de la première et la quatrième solution. En effet, les inconvénients de la première solution sont compensés par les avantages de la quatrième, et vice-versa. Cette solution optimale a été adoptée par la méthodologie Cosmos. La spécification initiale est donnée en langage SDL. Elle est ensuite traduite dans le format intermédiaire Solar, adapté aux algorithmes de synthèse.

Les prochaines sections présentent le modèle de représentation Solar, utilisé dans la méthodologie Cosmos. L'objectif de ce modèle est de permettre la représentation des

systèmes complexes de façon concise et expressive. Le modèle permet aussi la minimisation de l'effort intellectuel requis par l'utilisateur pour décrire le système, ainsi que la réduction de la complexité des algorithmes pour manipuler la représentation.

3.2 Format intermédiaire

La plupart des outils de conception conjointe utilisent un format de représentation interne. Cette représentation modélise le système pendant tout le processus de raffinement, à partir de la spécification jusqu'à l'architecture.

La spécification initiale du système est normalement décrite dans un format plus adapté à l'utilisateur, appelé langage naturel (voir Figure 18). Ce format est converti manuellement en une spécification d'entrée exécutable, qui peut être un langage textuel (C ou VHDL) ou une représentation graphique (StateCharts ou SDL). Le format intermédiaire est une répresentation de la spécification du système, du modèle et du langage. Ce format a une lecture et une manipulation plus facile pour les outils de synthèse que le langage exécutable d'entrée.

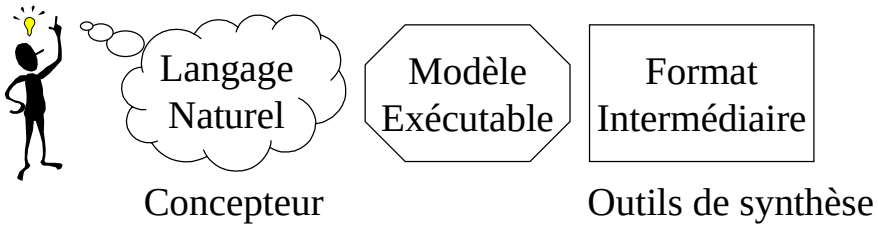


Figure 18: Format intermédiaire.

Nous pouvons distinguer deux types de modèles intermédiaires: les modèles orientés langage et les modèles orientés architecture. Ces deux modèles sont utilisés pour la spécification du système pendant les étapes de transformation. Les formats intermédiaires orientés langage sont normalement basés sur une représentation à base de graphes. Ces formats modélisent bien les concepts manipulés par les langages de spécification. Les formats intermédiaires orientés architecture sont normalement basés sur le modèle des machine d'états finis (MEF). Ces formats sont proches des concepts nécessaires pour représenter les architectures.

Le type de format intermédiaire sélectionné restreint la performance des outils de conception conjointe de différentes façons:

- Le **format intermédiaire orienté langage**: facilite la transformation de la spécification et la manipulation des concepts de haut niveau, comme l'abstraction de la communication et l'abstraction des types de données. Par contre, ce format impose certaines contraintes pendant la génération de l'architecture, la spécification des architectures partielles et la spécification des contraintes physiques;
- Le **format orienté architecture**: il rend plus difficile la traduction et la représentation des concepts de haut niveau. Par contre, ce format facilite la spécification des concepts liés à l'architecture, comme les protocoles de communication et la synchronisation. Ce format normalement génère des réalisations plus efficaces.

La Figure 19 représente le modèle générique de conception conjointe qui combine ces deux types de modèles intermédiaires. Premièrement, la spécification initiale au niveau système est traduite dans un format intermédiaire orienté langage. Un ensemble d'étapes de raffinement peut être appliqué sur cette représentation de haut niveau. Ensuite,

la conversion du modèle raffiné est réalisée dans un format intermédiaire orienté architecture. Dans le format orienté architecture, les raffinements permettent la préparation de la spécification à la génération de l'architecture.

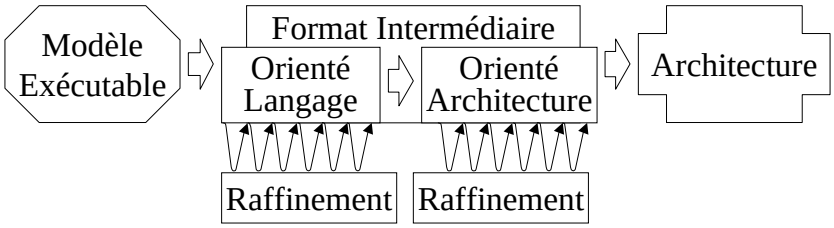


Figure 19: Modèles orientés langage et architecture.

Le modèle générique, à base de deux formats intermédiaires, a une réalisation difficile, puisqu'il nécessite l'utilisation des traducteurs entre les formats, et différents outils pour chaque format. Pour cette raison, les méthodologies actuelles cherchent un modèle de représentation adéquat et unique.

3.3 Etat de l'art

Plusieurs modèles de représentation des systèmes pour la conception conjointe sont proposés dans la littérature [Gaj94b, KuD88, Cam89, Jer97, Moo96, Bud97, Car96]. La plus grande partie des travaux est basée sur une représentation à base de **graphes de flot** ou à base de **machines d'états finis**. Les graphes de flot correspondent au format orienté langage. Des variations des graphe flot sont: les graphes de flot de données; les graphes flot de contrôle; et les graphes flot de contrôle et de données. Les machines d'états finis correspondent au format orienté architecture. Des variations des machines d'états finis sont: les machines d'états finis avec chemin de donnée; et les machines d'états finis avec co-processeur.

Les formats intermediaires orientés langage:

- **Graphe flot de données** (DFG): c'est la représentation la plus populaire pour la synthèse au niveau système. Les noeuds représentent les opérateurs et les arc représentent les valeurs. La fonction d'un noeud est de générer une nouvelle valeur dans les sorties, basée sur les valeurs d'entrée. Dans le niveau système, les noeuds peuvent représenter un calcul complexe ou même un processeur. Ce modèle est puissant pour la représentation du calcul, mais il est restreint pour la représentation des structures de contrôle;
- **Graphe flot de contrôle** (CFG): c'est la représentation la plus adaptée pour modéliser le contrôle. Il permet la représentation d'opérations du type boucles, appels de procédures, exceptions, et synchronisation. Les noeuds représentent des opérations et les arcs représentent des relations de séquencement. Ce modèle impose des restrictions sur l'analyse du flot de données et sur les transformations;
- **Graphe flot de contrôle et de données** (CDFG): c'est une extension des DFG avec des noeuds de contrôle. Ce modèle est adapté à la représentation des applications orientées flot de données.

Les formats intermédiaires orientés architecture:

- **Machine d'états finis avec chemin de données** (FSMD): est une MEF étendue avec des opérations sur des données. Le modèle peut contenir des variables internes, et les transitions peuvent inclure des opérations logiques et arithmétiques sur les variables;

- **Machine d'états finis avec co-processeur** (FSMC): c'est une FSMD avec des opérations exécutées par des co-processeurs. Les expressions peuvent contenir des opérations complexes exécutées dans des unités de calcul dédiées. Plusieurs outils de conception conjointe génèrent des architectures à base de FSMC [Ern95, Mad96b]. Dans ces systèmes, l'architecture est composée d'un processeur logiciel qui agit comme contrôleur et des co-processeurs matériels qui agissent comme accélérateurs.

3.4 Le format intermédiaire Solar

Solar est un modèle de représentation destiné à faire le lien entre les outils de CASE et les outils de CAO de circuits intégrés. Solar est composé d'un modèle de représentation de données et d'un langage textuel. Le format textuel est lisible mais il n'est pas destiné à être un nouveau langage de spécification système. Solar permet à la fois la spécification et la synthèse conjointe pour les systèmes contenant du matériel et du logiciel. La Figure 20 représente le contexte d'utilisation de Solar.

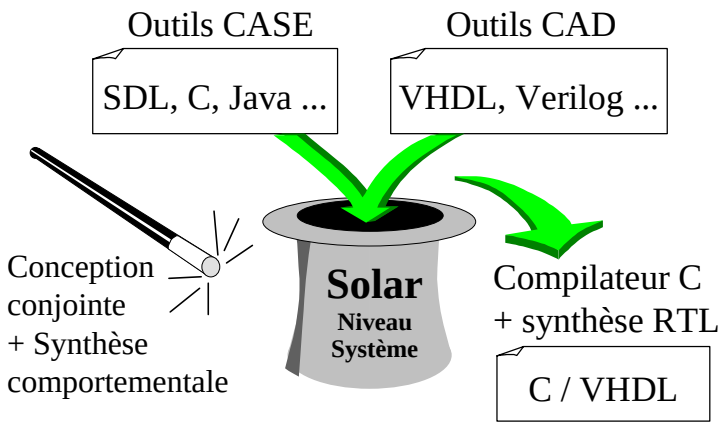


Figure 20: Le contexte Solar.

Les objectifs initiaux de la construction du modèle Solar étaient :

- L'identification des structures de données qui permettent l'accommodation des besoins nécessaires aux étapes de co-spécification, de conception conjointe et de co-synthèse. Ces structures de données doivent former un modèle concis;
- Permettre la construction d'outils de synthèse au niveau système basés sur un format de représentation unique. Ces outils doivent être capables de générer un ensemble de spécifications à un niveau plus bas, adapté aux outils de synthèse existants;
- Permettre l'utilisation d'un modèle unique pour les conceptions de systèmes logiciels/matériels complexes et distribués;
- Fournir des caractéristiques qui permettent la modélisation orientée langage et la modélisation orientée architecture dans un unique modèle de représentation. De cette façon, la représentation peut simultanément manipuler les concepts fonctionnels et les concepts de réalisation facilitant ainsi considérablement la transition entre les différentes étapes de la conception.

3.4.1 Niveaux d'abstraction Solar

Le format intermédiaire Solar facilite la transition d'une spécification initiale à une réalisation, puisqu'il supporte trois niveaux d'abstraction: le niveau système, le niveau comportemental et le niveau transfert de registres. Chaque niveau défini un ensemble de concepts qui peuvent être raffinés dans ceux du niveau inférieur (Figure 21). Les outils sont construits sur cette structure pour réaliser l'ensemble de transformations de la spécification. Par exemple, les outils de découpage, de synthèse de la communication et de génération de code appartiennent à cette structure.

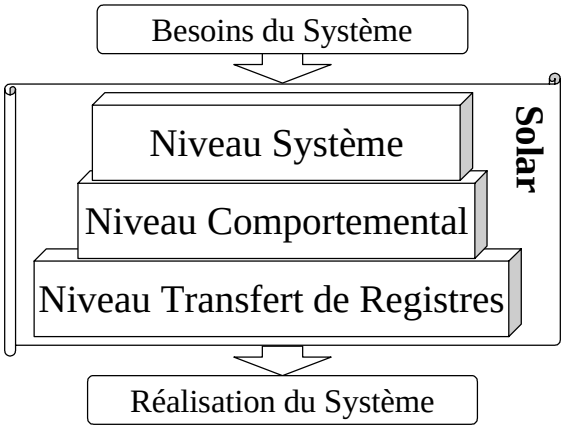


Figure 21: Niveaux d'abstraction Solar

- **Abstraction au niveau système**: c'est une abstraction de haut niveau des fonctionnalités et des contraintes du système à développer. À ce niveau, le système est vu comme étant une entité qui interagit avec l'environnement à travers d'une frontière bien définie. Cette définition est appliquée de la même façon à un système embarqué complexe comme à un circuit intégré ou à une simple routine C. Une vision au niveau système est représentée par un système distribué où les sous-systèmes communiquent à travers des méthodes standards de communication. Les différents sous-systèmes sont décrits de façon hiérarchique et représentent des comportements concurrents. Les concepts clefs de ce niveau d'abstraction, disponibles dans la plus part des langages système, sont: la communication, la hiérarchie et la concurrence;
- **Abstraction au niveau comportemental:** ce niveau d'abstraction repose sur les machines d'états finis communicantes. Ce modèle est très populaire parmis la communauté de synthèse de haut niveau et de conception conjointe. Il utilise normalement un modèle de MEF étendue pour permettre le calcul des données. Le modèle de MEF communicantes n'est pas fréquemment supporté par des outils de conception conjointe. SpecSyn [Gaj98] et Polis [Pol98] utilisent des MEFs interconnectées. Dans Solar, le modèle de machine d'états finis étendue supporte la hiérarchie, la concurrence et la communication entre les MEFs individuelles. Dans une machine, un état peut être: un état feuille, un ensemble d'états séquentiels ou un ensemble d'états parallèles. La représentation Solar des MEF peut modéliser la plus grande partie des langages orientés contrôle. Elle permet la modélisation des comportements complexes aussi que ceux permis par les graphes de flot de contrôle;
- **Abstraction au niveau transfert des registres**: c'est le plus bas niveau d'abstraction d'une représentation intermédiaire. Il est utile pour décrire les détails de l'architecture du circuit intégré. Une description RTL est composée d'un contrôleur, d'une structure du chemin de donnée, et d'un réseau d'interconnexion. Un modèle RTL peut être rangé dans une bibliothèque et réutilisé pour d'autres projets.

3.4.2 Principaux concepts Solar

Solar supporte plusieurs niveaux d'abstractions, lui permettant de modéliser des circuits intégrés aussi bien que des systèmes logiciel/materiel distribués. La modélisation multi-niveau permet le raffinement de la spécification entre les différents niveaux d'abstraction.

Le modèle Solar combine deux puissants concepts du niveau système :

- Un **modèle de machine d'états finis étendue** pour décrire le comportement, représenté par des tables d'états;
- Un **mécanisme à base d'appel de procédures à distance** pour spécifier la communication de haut niveau, représentée par des canaux.

Comme montre la Figure 22, Solar modélise la structure du système par une hiérarchie des unités de conception. Un système est composé d'un ensemble d'unités qui communiquent. Le comportement des unités feuilles est décrit par des tables d'états. La communication entre les unités est modélisée par des appels de procédures à distance. L'unité canal est composée d'un contrôleur, d'un ensemble de services et d'un ensemble de signaux d'interconnexion.

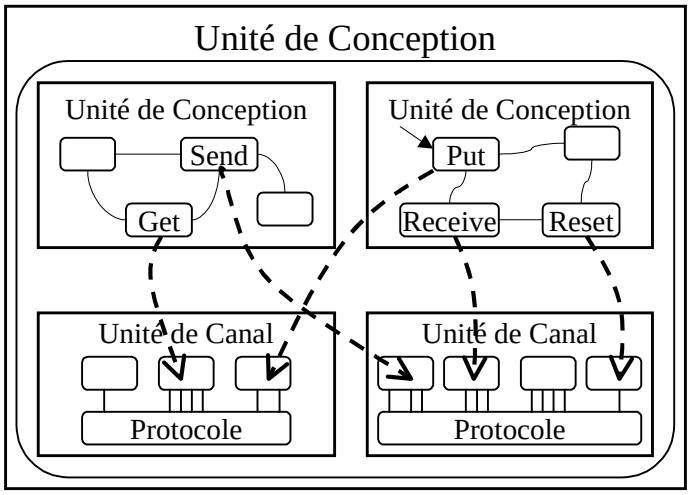


Figure 22: Concepts de base de Solar.

Les constructeurs Solar sont:

- La **Table d'Etat** est le constructeur de base pour décrire le comportement, représenté par une machine d'états finis étendue;
- Les **Unités de Conception** permettent la structuration de la description du système en un ensemble de sous-systèmes en interaction (processus). Ces sous-systèmes interagissent avec l'environnement en utilisant une frontière bien définie;
- Les **Unités de Canal** réalisent la communication entre les unités de conception. L'unité canal contient la spécification du protocole de communication;
- Les **Unités Fonctionnelles** sont utilisées pour spécifier des opérateurs complexes (DCT, FFT). Ce concept permet la spécification et la réutilisation d'opérateurs partagés entre MEF séquentielles.

La Figure 23 montre la représentation graphique et textuelle d'une description Solar. Dans la Figure 23.a nous pouvons voir deux processus appellés *DU_SERVER* et *DU_CLIENT* représentés par ces respectives instances *SERVER* et *CLIENT*. Ces processus communiquent par l'intermédiaire des signaux (*wr_req*, *rewr_rdy*, *read_req* et

data_io) connectés à un contrôleur de canal appelé *CHANNEL*. Le contrôleur exécute un protocole du type *fifo*. *CLIENT_AbstractChannel* et *SERVER_AbstractChannel* correspondent à des canaux abstraits de communication. *CHANNEL* représente la réalisation d'un protocole de communication. La Figure 23.b montre une table d'états (*FIFO_StateTable*) composée de deux états (*Init_St* et *Rec_Send_St*) appartenant au contrôleur du canal. La description textuelle de chaque élément graphique et sa structure interne est aussi disponible, comme le montre la Figure 23.c.

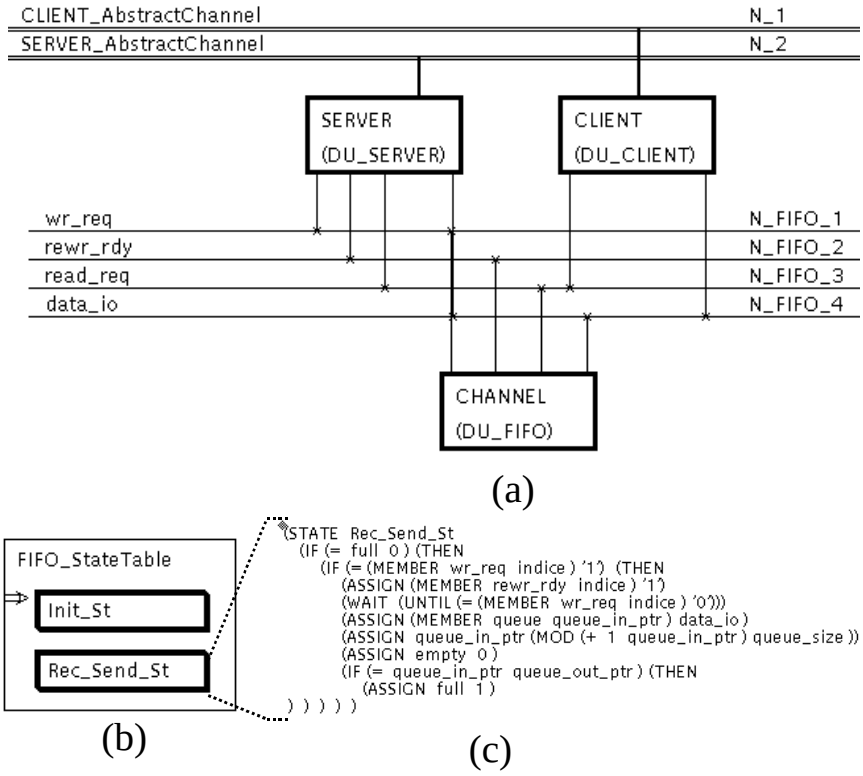


Figure 23: Exemple de représentation Solar.

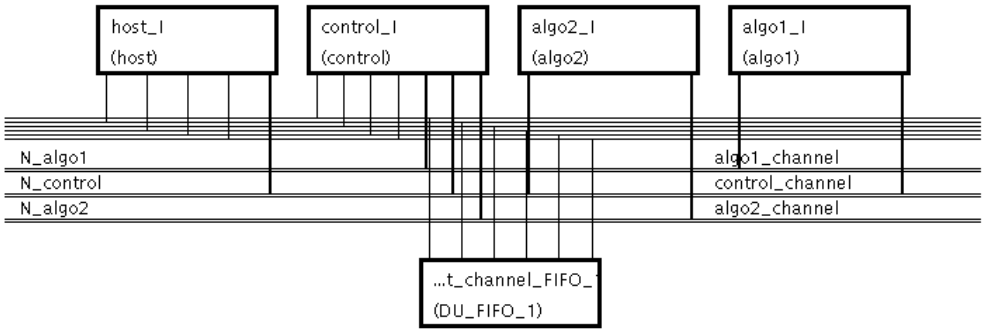
La représentation textuelle Solar (Figure 23.c) utilise une syntaxe comparable au langage EDIF. Ce style de syntaxe facilite les opérations de lecture et d'enregistrement de la spécification, et fournit une méthode unifiée d'extension du format intermédiaire. Chaque construction commence avec une parenthèse suivi d'un mot clef, ensuite nous trouvons une liste d'items et la parenthèse est fermée, *(ASSIGN empty 0)* par exemple. Ces items peuvent cacher d'autres constructions imbriquées, des jetons (*tokens*) ou des identificateurs.

La manipulation d'une description Solar est réalisée à l'aide d'un environnement graphique. L'édition de la description textuelle Solar est possible, mais non conseillée. Solar dispose d'un ensemble de fonctions pour le traitement de la structure interne de données. Ces fonctions réalisent principalement le chargement et la sauvegarde de la spécification. Pour le raffinement de la spécification, une API (*Aplication Program Interface*) est disponible. Cette interface permet la manipulation des objets Solar et possède des fonctions comme la création de nouveaux objets, la destruction, l'enchaînement et la sélection des objets.

3.4.2.1 La philosophie Solar

Une description Solar représente un système distribué où le comportement est divisé en un ensemble d'unités de conception. Ces unités sont exécutées indépendamment jusqu'au moment où la synchronisation devient nécessaire. La synchronisation est réalisée par des signaux qui traversent des portes, ou en utilisant des messages véhiculés par des canaux de communication.

La Figure 24 montre la représentation graphique d'un système composé d'un ensemble d'unités de conception. Les unités, représentées par des boîtes, s'exécutent en parallèle et communiquent par des signaux et des messages. Les instructions de synchronisation entre les processus communicants sont aussi représentées dans la figure. La synchronisation par des signaux utilise l'instruction *WAIT* (attente) et la synchronisation par des messages utilise l'instruction *CUCALL* (appel de procédure distante).




```
Synchronisation par signaux :
Processus 1: (WAIT (UNTIL (=  channel_FIFO_wr_req  '1')))
Processus 2: (ASSIGN channel_FIFO_wr_req '1')

Synchronisation par messages :
Processus 1: (CUCALL get_signal N_algo1 (PARAMETERASSIGN  sdl_signal signal ))
Processus 2: (CUCALL put_signal N_algo1 (PARAMETERASSIGN  sdl_signal signal ))
```

Figure 24: Concurrence et synchronisation en Solar.

3.4.2.2 Le comportement

Solar définit le concept de **table d'états,** qui permet la modélisation d'unités contenant des comportements complexes. La table d'états est une extension du modèle classique de machine d'états finis à laquelle ont été ajoutés deux concepts: la hiérarchie et le parallélisme. Chaque état d'une table d'états peut contenir des états feuilles ou d'autres tables d'états, formant la hiérarchie du comportement. Les états peuvent être exécutés de façon parallèle ou séquentielle.

Une table d'états contient des attributs permettant manipuler les exceptions, les variables globales, les initialisations et les états par défauts, comme montre la Figure 25. La transition entre états n'est pas restreinte à des niveaux hiérarchiques, cela veut dire qu'une transition peut traverser les frontières de la hiérarchie. Un état est composé de trois parties: les actions qui vont être exécutées; l'ensemble des transitions qui peuvent être réalisées à partir de l'état; et les conditions qui déterminent quelle transition va être réalisée.

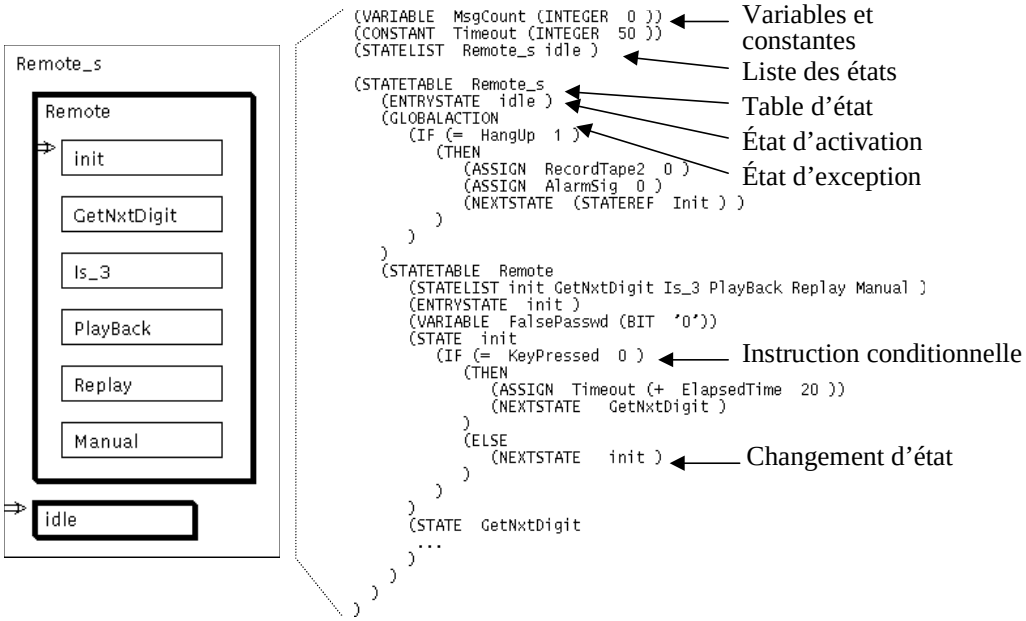


Figure 25: Comportement des unités.

Les variables et les constantes sont de type prédéfini ou de type défini par l'utilisateur. Les appels de procédures permettent, soit le passage du contrôle à des unités fonctionnelles, soit l'activation des procédures de communication. Les actions globales décrivent le comportement exécuté en parallèle avec la table d'états. Ce comportement permet la modélisation des exceptions.

Les tables d'états sont utilisées pour spécifier le comportement des unités de conception (processus), des unités de canaux, des unités fonctionnelles, des procédures et des fonctions. L'utilisateur peut aussi décrire le comportement en utilisant un langage externe, du type C ou VHDL. Dans ce cas, l'instruction *foreing* indique dans quel fichier le comportement est défini.

3.4.2.3 La structure du système

Avec le format intermédiaire Solar, un système est structuré par un ensemble de sous-systèmes communicants, modélisés par des **unités de conception**. Une unité de conception peut être composée d'un ensemble d'unités de conception communicantes (la hiérarchie structurelle) ou d'un ensemble de tables d'états en interaction (la hiérarchie comportementale). La communication entre les unités de conception peut se dérouler de deux façons différentes: à travers des ports d'entrées/sorties, dans ce cas, de simples signaux envoient des données dans une ou deux directions; ou à travers des canaux de communication, qui permettent à l'utilisateur de spécifier la communication en utilisant un protocole prédéfini.

La Figure 26 présente un exemple de système composé de cinq unités de conception communicantes. Ces unités sont interconnectées par un ensemble de signaux et de canaux. Les signaux sont représentés par des lignes horizontales simples tandis que les canaux par des lignes horizontales doubles.

Une unité de conception est caractérisée par son interface et son contenu (*contents*). L'interface spécifie les ports et les accès utilisés pour communiquer avec l'extérieur. Le contenu spécifie le corps de l'unité. Le contenu peut être structurel ou comportemental. L'attribut *viewtype* indique le type d'abstraction utilisé (*behavior* ou *structure*). Le contenu d'une unité du type structurelle se compose d'une liste d'instances

d'unités et d'un réseau d'interconnexion (Figure 26 côté droit). Le contenu d'une unité du type comportementale se compose d'une liste de déclarations et de tables d'états (Figure 26 en bas).

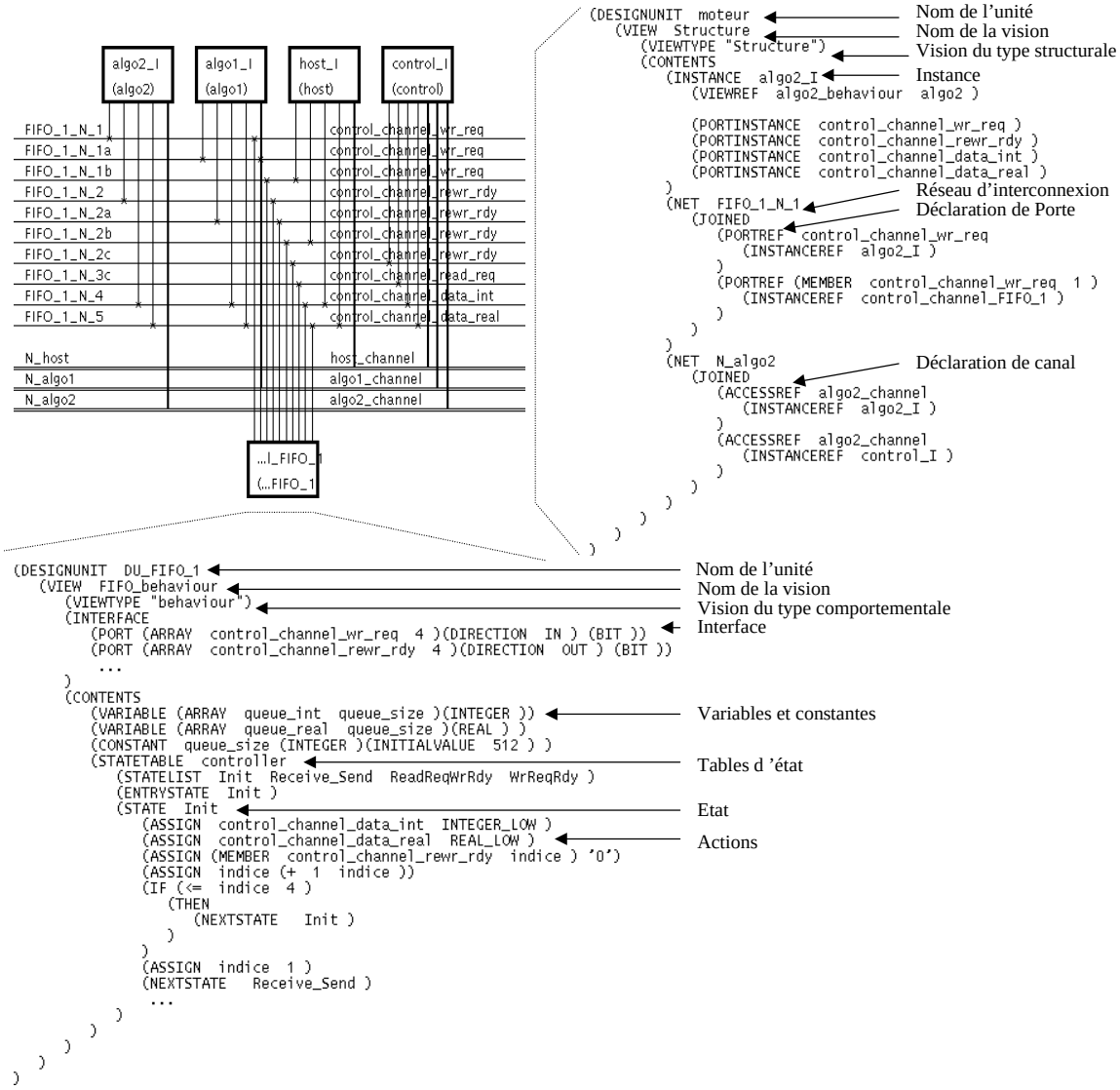


Figure 26: Structure des unités

3.4.2.4 La communication

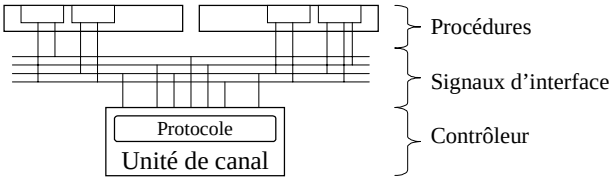
La communication entre sous-systèmes utilise les **canaux**. L'unité de canal permet la communication entre un nombre varié d'unités de conception. Le modèle d'unité de canal combine le concept de moniteurs et de passage de messages, connu aussi comme l'appel de procédure à distance (RPC [And91]). L'utilisation de RPC pour invoquer les services de canaux permet une représentation flexible, avec une sémantique claire et efficace, permettant la modélisation de schémas de communication existants. Ces schémas de communication peuvent être décrits séparément du reste du système, permettant ainsi la modularisation de la conception et de la spécification.

Le principe des moniteurs permet le contrôle d'accès à des ressources. Le contrôle est basé sur l'encapsulation des ressources et des opérations qui les manipulent. Ces opérations ont le format de procédures et sont appelées par les processus concurrents qui

partagent les ressources. De cette façon, la réalisation des ressources est complètement indépendante des processus accédant à celles-ci.

Une unité de canal consiste en plusieurs connexions individuelles (signaux) qui agissent comme un mécanisme d'échange de données ou comme un mécanisme de synchronisation du type *hand-shake*. Le canal peut être vu comme une ressource partagée dont l'accès est contrôlé par un nombre fixe de procédures. Une fois le canal activé, la communication peut procéder de plusieurs façons, comme par exemple: un transfert parallèle or séquentiel, un passage de messages synchrone ou asynchrone.

Le modèle du canal se compose (voir Figure 27) d'un contrôleur qui mémorise l'état courant de la ressource, d'un ensemble de signaux d'interconnexion, d'une interface qui encapsule les ports d'entré/sortie, et d'un ensemble de procédures (ou méthodes). Les procédures et les ports constituent les parties visibles du canal, et sont les seuls moyens d'y accéder. Le contrôleur est une fonction privée, inaccessible aux unités interconnectées, qui mémorise l'état actuel de la ressource. Le contrôleur interagit avec les procédures par des signaux pour permettre la synchronisation et l'exclusion mutuelle.



(b) Modèle conceptuel

```
(CHANNELUNIT  FIFO
   (VIEW  FIFO_behaviour
      (INTERFACE
         (PORT (ARRAY  wr_req  Number_of_DU )(DIRECTION  IN ) (BIT ))
         (PORT (ARRAY  rewr_rdy  Number_of_DU )(DIRECTION  OUT ) (BIT ))
          ...
         (METHOD  put_signal
            (PARAMETER  sdl_signal (INTEGER ))
            (STATETABLE  put_signal
               (STATELIST  request  requestwait )
               (ENTRYSTATE  request )
               (STATE  request
                  (IF (=  rewr_rdy  '1')
                     (THEN
                        (WAIT  (UNTIL (=  rewr_rdy  '0')))
                     )
                  )
                  (ASSIGN  data_int  sdl_signal )
                  (ASSIGN  data_real  REAL_LOW )
                  (ASSIGN  wr_req  '1')
                  (NEXTSTATE   requestwait )
               )
               (STATE  requestwait
                  (WAIT  (UNTIL (=  rewr_rdy  '1')))
                  (ASSIGN  data_int  INTEGER_LOW )
                  (ASSIGN  data_real  REAL_LOW )
                  (ASSIGN  wr_req  '0')
               )
            )
         )
         ...
      )
```

Interface

Procédure de accès au canal

Contrôleur du canal

```
(CONTENTS
   (VARIABLE (ARRAY  queue_int  queue_size )(INTEGER ))
   (VARIABLE  queue_in_ptr (INTEGER )(INITIALVALUE  1 ) )
   (VARIABLE  queue_out_ptr (INTEGER )(INITIALVALUE  1 ) )
   (VARIABLE  indice (INTEGER )(INITIALVALUE  1 ) )
   (VARIABLE  full (INTEGER )(INITIALVALUE  0 ) )
   (VARIABLE  empty (INTEGER )(INITIALVALUE  1 ) )
   (CONSTANT  queue_size (INTEGER )(INITIALVALUE  512 ) )
   (STATETABLE  controller
      (STATELIST  Init  Receive_Send  ReadReqWrRdy  WrReqRdy )
      (ENTRYSTATE  Init )
      (STATE  Init
         (ASSIGN  data_int  INTEGER_LOW )
         (ASSIGN  data_real  REAL_LOW )
         (ASSIGN (MEMBER  rewr_rdy  indice ) '0'
         (ASSIGN  indice (+  1  indice ))
         (IF (<=  indice  Number_of_DU )
            (THEN
               (NEXTSTATE   Init )
            )
         )
         (ASSIGN  indice  1 )
         (NEXTSTATE   Receive_Send )
      )
      ...
   )
)
```

(d) Description Solar

Figure 27: Canal de communication.

3.4.3 Les objets Solar

La structure du format intermédiaire Solar utilise une approche Orienté Objet [Boo97]. L'approche orientée objet apporte les avantages suivants: elle facilite l'utilisation de la structure de données Solar par les outils de synthèse; elle facilite la transition entre les différentes étapes de conception; elle facilite les extensions futures de Solar; et permet une représentation et une compréhension facile de la structure de données.

Le format intermédiaire Solar est composé d'un ensemble d'objets où chaque objet modélise une partie du système et regroupe toutes les informations liées à l'entité modélisée, comme illustre la Figure 28.

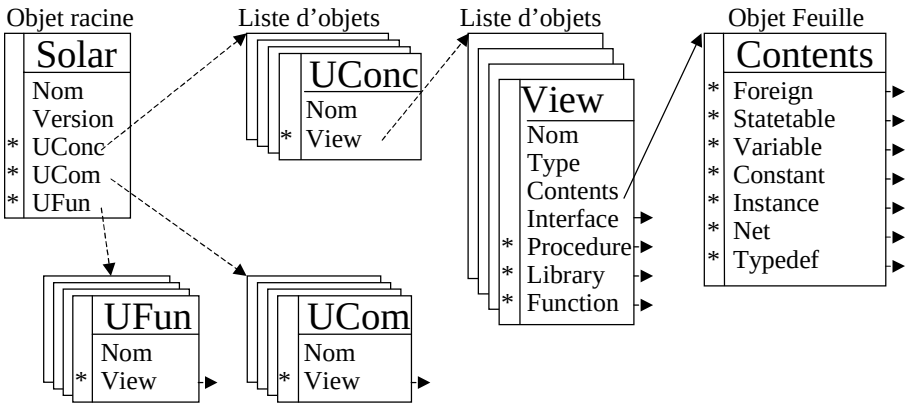


Figure 28: Structure des objets Solar.

La Figure 29 présente un objet Solar. Les objets sont composés de deux parties :

- La **Partie privée** est la vision interne de la classe, qui permet sa mise en oeuvre et cache son comportement;
- La **Partie publique** d'une classe (l'interface) fournit la vue extérieure et met l'accent sur l'abstraction, tandis qu'elle cache sa structure et son comportement. Cette interface contient les déclarations de toutes les opérations applicables aux instances (objets) de cette classe.

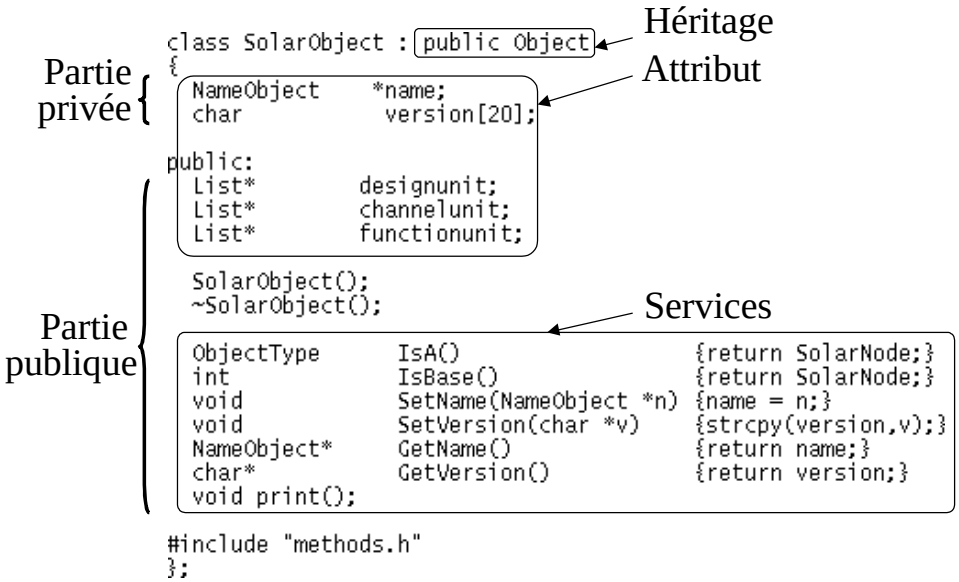


Figure 29: Les différents parties qui composent un objet Solar.

Les **Attributs** forment l'état d'une classe. Les attributs englobent toutes les propriétés de l'objet, plus les valeurs courantes de chaque propriété. Pour décrire une classe, l'utilisateur doit toujours essayer d'encapsuler l'état de l'objet (partie privée) plutôt que de l'exposer (partie publique).

Les **méthodes** décrivent le comportement d'une classe. Le comportement est la façon dont un objet agit et réagit, en terme de changement d'état et de transmission de messages.

Chaque objet représente une instance d'une certaine classe, toutes les classes étant des membres d'une hiérarchie de classes unifiées par des relations d'**Héritage**. L'appendice D présente la description détaillée des principaux objets qui composent le

format intermédiaire Solar. Une description complète de tous ces objets est disponible dans le manuel de référence Solar [Rom97].

3.4.3.1 La hiérarchie des objets Solar

La structure de données Solar est composée d'un ensemble d'objets. La Figure 30 présente la structure hiérarchique des objets où le niveau d'indentation représente l'héritage des attributs des objets.

Dans cette structure hiérarchique nous identifions trois types d'objets :

- Les **objets originaires des classes abstraites** (*Object*, *Action*, *Simpact*, *Value* et *Typeval*). Une classe abstraite est la classe qui n'a pas d'instance. Elle existe afin que ses sous-classes concrètes enrichissent sa structure et son comportement, généralement en mettant en oeuvre ses opérations abstraites;

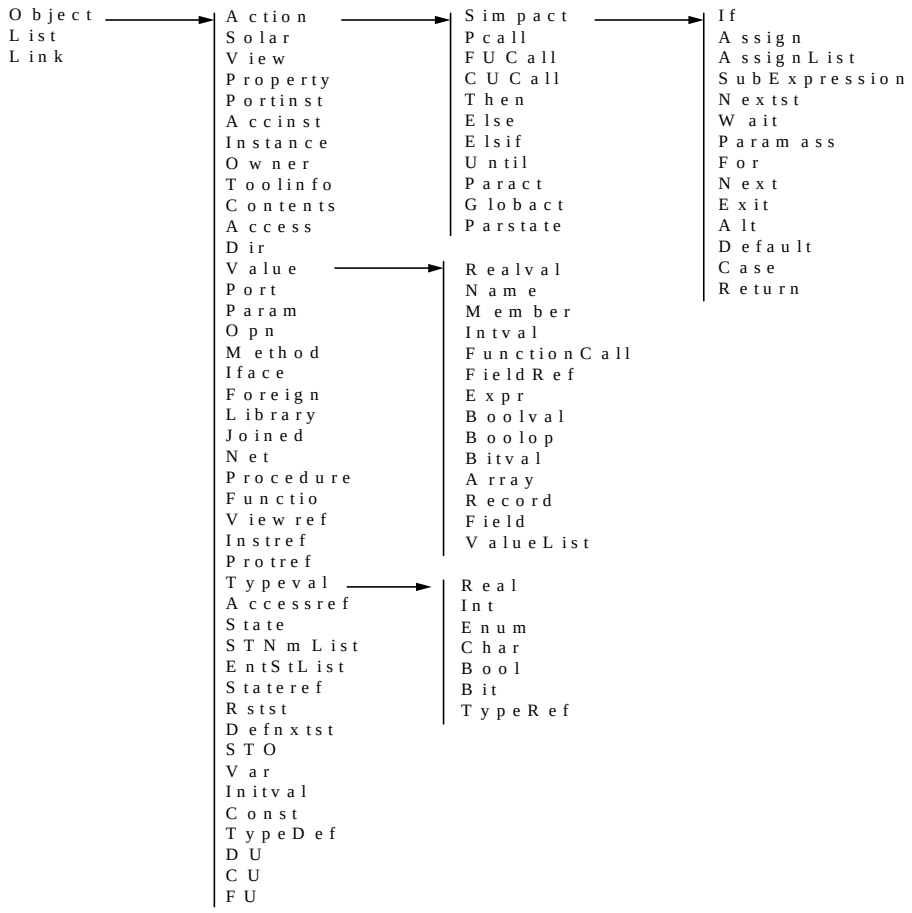


Figure 30: Diagramme de hiérarchie des objets Solar.

- Les **objets originaires des classes de base** (*Object*, *List* et *Link*). Une classe de base est la classe la plus générale dans la structure. Nous pouvons dire que la classe *Objet* est une classe de base primaire puisqu'elle sert de super-classe ultime pour toutes les autres classes.
- Les **objets originaires des classes concrètes** (tous les autres objets). La classe concrète est une classe où la mise en oeuvre est concrète, et qui peut avoir des instances (objets);

3.4.4 L'interface graphique Solar

L'interface Solar permet la visualisation et la manipulation des spécifications représentées sous le format intermédiaire Solar. La représentation graphique permet à l'utilisateur de comprendre instantanément la structure et les composants de la spécification. De cette façon, après chaque raffinement, l'utilisateur peut vérifier le changement de la structure et du comportement des unités occasionné par l'application de la primitive.

La représentation graphique du comportement des unités et de la structure du système obéit aux normes suivantes:

- Le **comportement des unités** est représenté par des boîtes imbriquées pour représenter la hiérarchie des machines (états ou tables d'états), comme le montre la Figure 31. Dans un même niveau de la hiérarchie, les machines représentées verticalement ont une exécution séquentielle, tandis que les machines représentées horizontalement ont une exécution parallèle. L'état initial de chaque machine est représenté par une flèche. Les instructions appartenant aux machines ne sont pas visibles graphiquement.

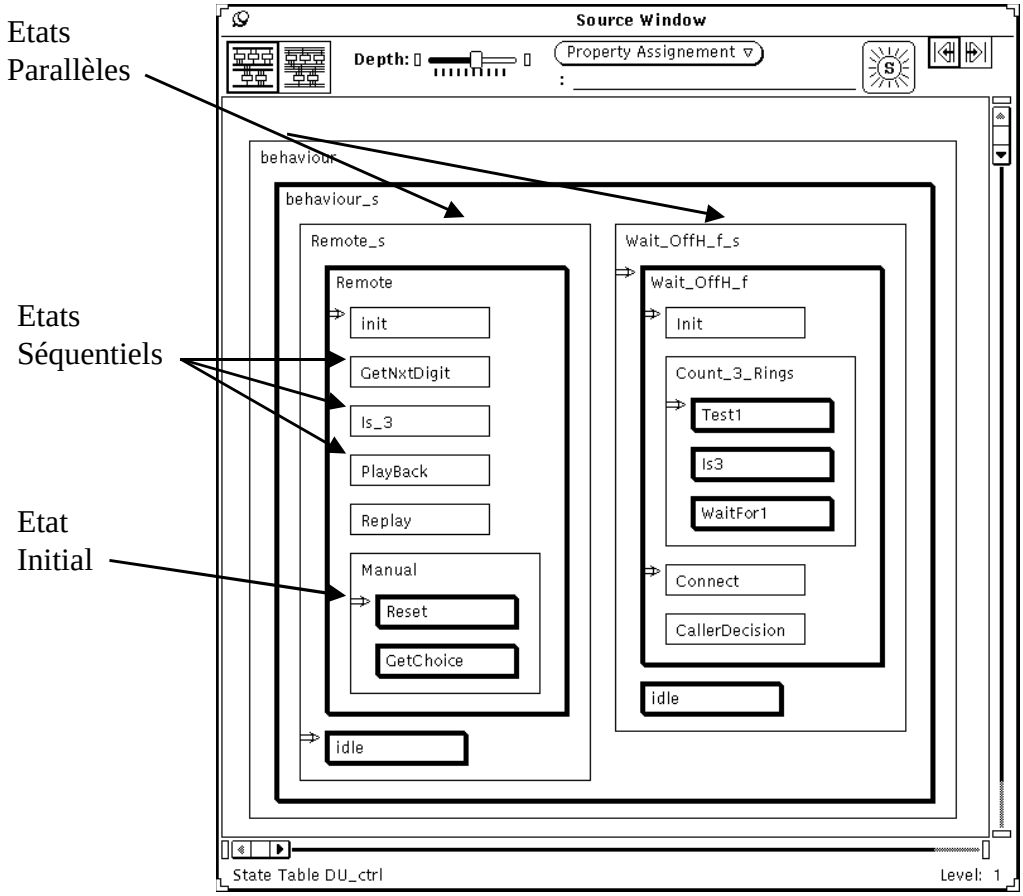


Figure 31: Représentation graphique du comportement des unités.

- La **structure du système** (voir Figure 32) est représentée graphiquement par les unités et les interconnexions. Les unités sont représentées par des boîtes et les interconnexions par des lignes horizontales. Dans les boîtes se trouvent le nom de l'instance, le nom de l'unité et le choix de réalisation (logicielle *[S]* ou matérielle *[H]*). Les unités de conception sont disposées entre les signaux d'interface et les signaux internes, tandis que les unités canaux sont disposées au-dessous des signaux internes et des canaux abstraits. Pour chaque signal, est représenté le nom du réseau (à gauche) et le nom de l'accès (à droite). Les signaux du type bit sont représentés par des lignes

simples et les signaux d'autres types par des lignes épaisses. La direction des signaux d'interface est représentée par les symboles: d'entrée (>), de sortie (<) ou bidirectionnel (<>). Les canaux abstraits sont représentés par des lignes doubles. Les lignes verticales, qui connectent les unités aux signaux et aux canaux, indiquent les interconnexions et les accès aux canaux.

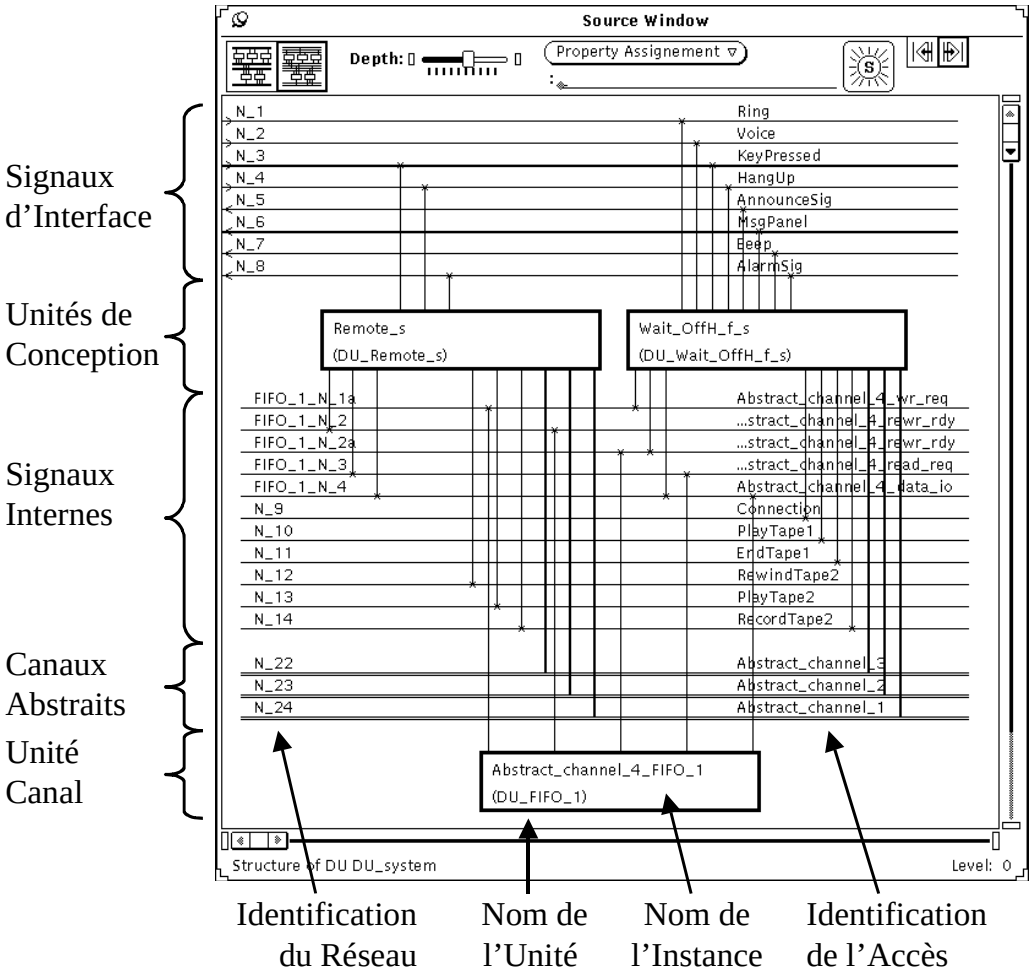


Figure 32: Représentation graphique de la structure des unités.

3.5 Conclusion

Dans ce chapitre, le format intermédiaire Solar a été présenté dans sa version actuelle. Initialement, ce format a été développé dans le cadre d'une conception de recherche visant la réalisation d'une passerelle entre les outils de CASE et les outils de CAO de circuits intégrés [Jer94]. Les principaux objectifs visés étaient: l'identification des structures de données permettant l'accommodation des besoins nécessaires aux étapes de co-spécification, de conception conjointe et de co-synthèse; un format concis permettant la construction d'outils de synthèse au niveau système basés sur un format de représentation unique; l'utilisation de ce modèle pour la conception de systèmes matériel/logiciel complexes et distribués; et finalement, fournir des moyens permettant la modélisation orientée à la fois langage et à la fois architecture dans un modèle unique de représentation. Pendant le déroulement de ce travail de recherche, Solar a été validé et a reçu plusieurs mises à jour.

Les principales contributions de l'auteur au format Solar sont:

- **L'unification du format entre les différents outils du système** (Partif, Amical, Pat, Sas et S2cv), puisqu'au début de ce travail de thèse chaque outil disposait de sa propre version Solar, avec ses propres objets;
- **L'ajout de nouveaux objets à la demande suivant les besoins de nouveaux algorithmes**, comme par exemple: des nouveaux types de données définis par l'utilisateur (*Typedef*), la description de parties de la spécification de façon externe (*Library* et *Foreign*), l'inclusion des valeurs initiales à des types de données (*Initval*), etc.;
- **La mise à jour de plusieurs classes existantes**;
- **La réalisation d'une interface graphique a Solar.**

Cette opération de maturation a été possible grâce à plusieurs facteurs: l'utilisation du format intermédiaire Solar sur **plusieurs exemples**, comme la carte du réseau ATM et le contrôleur de bras de robot; la construction et la mise en pratique de **nouveaux outils** qui manipulent les caractéristiques du modèle Solar qui n'étaient pas testés précédemment, comme les classes qui manipulent les types de données, les vecteurs et les réseaux de communication; l'intégration des outils dans un **environnement unique**; et finalement l'**interface graphique** Solar qui a donné une identité au format intermédiaire.

Chapitre 4

Découpage Transformationnel

Those that can, use applications.
Those that can't, write them!

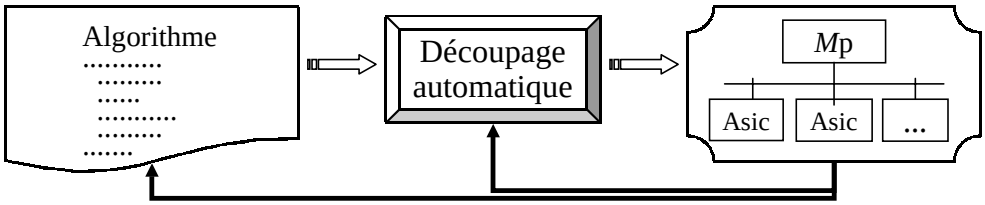
Ce chapitre présente une approche transformationnelle de découpage logiciel/matériel. Cette approche couvre le processus de conception conjointe à travers des transformations guidées par l'utilisateur permettant un découpage semi-automatique. Les transformations sont basées sur un ensemble de primitives. Ces primitives réalisent des opérations de décomposition de la fonctionnalité, de réorganisation de la structure et de transformation de la communication. Cette approche permet une transformation rapide d'une spécification système en une architecture distribuée.

4.1 Introduction

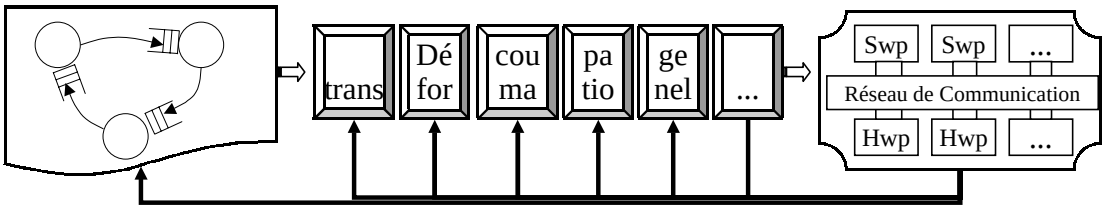
Le découpage logiciel/matériel est en passe de devenir le goulet d'étranglement du processus de développement des systèmes électroniques complexes. Le terme système, dans ce chapitre, signifie un système multiprocesseur, distribué et de temps réel. Ce système est composé de processeurs programmables, qui exécutent le code logiciel, et de processeurs matériels dédiés, interconnectés par un réseaux de communication complexe. Un tel système peut être réalisé sur un seul circuit électronique, sur une carte ou sur un système géographiquement distribué.

Dans une méthodologie de conception traditionnelle, l'utilisateur réalise le découpage logiciel/matériel tôt dans le cycle de développement. Les différentes parties du système sont conçues par différents groupes. L'intégration de ces différentes parties amène généralement à une détection tardive des erreurs. La propagation des erreurs à des étapes avancées de la conception génère une augmentation du coût de la conception et une augmentation du temps nécessaire à l'étape d'intégration. En plus, ce découpage prématuré restreint la possibilité d'investigation d'une meilleure relation entre les parties matérielles et logicielles. L'utilisateur est obligé de surdimensionner les différentes parties du système dans le but de réduire les risques lors de l'intégration.

La conception conjointe commence avec une spécification au niveau système, réalise le découpage logiciel/matériel et produit une architecture logicielle/matérielle mixte, comme détaille le chapitre 2. Les approches de conception peuvent être classées par trois concepts : **la description d'entrée**, **l'approche de découpage** et **l'architecture cible**. La Figure 33 illustre deux schémas de conception conjointe.



(a) Approche Classique : architecture monoprocesseur, découpage en une étape



(b) Approche Transformationnelle: architecture multiprocesseur, raffinement transformationnel

Figure 33: Approches de conception conjointe.

Quand les modèles utilisés pour la spécification initiale et pour l'architecture cible sont simples, le découpage automatique est alors possible (voir Figure 33.a). Pour cette raison, plusieurs approches classiques de conception conjointe commencent avec une spécification au niveau système sous la forme d'un algorithme et produisent une architecture monoprocesseur classique. Cette architecture est composée d'un processeur central, agissant comme contrôleur maître, et d'un ensemble d'accélérateurs matériels agissant comme co-processeurs. Dans ce cas, le découpage consiste à trouver les parties critiques de la description dans le but de les exécuter sur des co-processeurs matériels.

Par contre, la méthodologie de découpage transformationnel proposée dans ce travail débute avec un modèle de niveau d'abstraction très élevé (Figure 33.b). Ce modèle permet la spécification de processus distribués qui communiquent en utilisant des protocoles asynchrones. De plus, cette méthodologie prend pour cible une architecture multiprocesseur, plus générale que l'architecture monoprocesseur. Ce choix augmente l'espace existant entre la spécification d'entrée et l'architecture cible. C'est justement dans le but de remplir cet espace conceptuel que l'approche de découpage transformationnel a été créée.

Dans cette approche le processus de découpage est décomposé en un ensemble d'étapes de raffinement. Chaque étape exécute une transformation spécifique, comme par exemple : fixer le nombre et le type des processeurs, assigner des parties du comportement aux processeurs, réaliser l'allocation des ressources de communication et synchronisation, exécuter la synthèse de l'interface et d'autres étapes de la conception conjointe. Comme il va être montré explicitement dans la suite de ce chapitre, ce modèle permet un découpage efficace et une exploration facile de l'espace des solutions pour les systèmes multiprocesseurs.

La principale contribution de ce chapitre est qu'il présente une approche de découpage logiciel/matériel guidée par l'utilisateur et basée sur des raffinements incrémentaux. Dans la suite de ce chapitre la méthodologie de conception sera présentée et ainsi que l'efficacité de cette approche.

L'approche développée traite le processus de conception à travers un ensemble de transformations permettant le découpage semi-automatique avec des résultats prévisibles. L'efficacité de ce schéma de découpage est illustré à travers de plusieurs exemples.

La section 4.2 présente l'approche de découpage transformationnel. La section 4.3 introduit les trois types de raffinements: transformation du comportement, décomposition de la structure et transformation de la communication. Ces raffinements sont détaillés respectivement dans les sections 4.4, 4.5 et 4.6. La section 4.7 illustre le découpage transformationnel en utilisant deux exemples. Finalement les sections 4.8 et 4.9 présentent la conclusion et montrent les résultats et la puissance de cette méthodologie.

4.2 Le découpage transformationnel

La méthodologie de découpage transformationnel suppose que l'utilisateur débute la conception avec une spécification initiale et une solution architecturale en tête. L'utilisateur dispose d'un ensemble de primitives qui peuvent être enchaînées à la demande pour transformer le modèle initial en une architecture. Toutes les transformations sont exécutées automatiquement. Les décisions sont prises par l'utilisateur qui utilise son expérience et connaissance pour arriver à la solution désirée.

Chaque étape réduit la distance conceptuelle existante entre la spécification et la réalisation par l'ajout de détails de réalisation (choix du protocole de communication, génération du code pour le logiciel ou le matériel). Les transformations doivent satisfaire aux restrictions imposées par l'utilisateur sans changer la fonctionnalité du système. Le flot de découpage transformationnel est illustré dans la Figure 34.

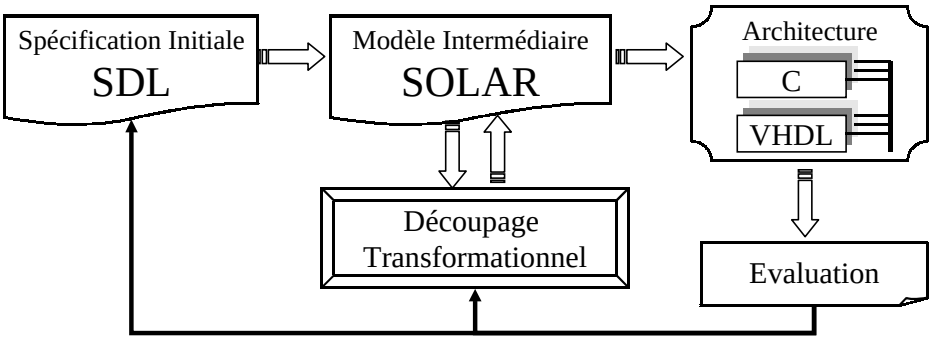


Figure 34: Découpage transformationnel.

4.2.1 Définition d'une transformation

Une transformation peut être définie comme l'application d'une fonction sur une spécification pour générer une réalisation, comme montre la Figure 35. La réalisation qui en résulte a le même comportement que la spécification d'origine mais est plus détaillée [Kru92].

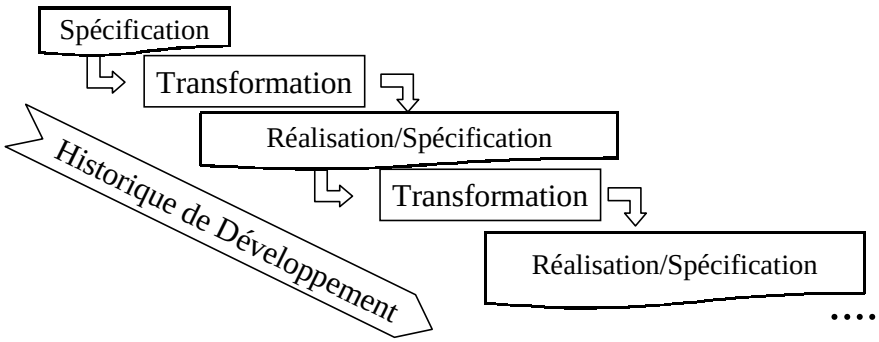


Figure 35: Raffinement d'une spécification.

Pendant les étapes de raffinement, une spécification S_i génère une réalisation S_{i+1} qui se comporte comme une spécification pour la prochaine transformation T_x. La transformation d'une spécification peut être représentée par l'Equation 1.

$$S_0 \times T_x \Rightarrow S_1 ,$$
$$S_i \times T_x \Rightarrow S_{i+1}$$

Equation 1: La transformation d'une spécification.

Chaque transformation possède une condition d'applicabilité qui détermine les restrictions à l'application de cette transformation. Une séquence de transformations exécutées est appelée un **historique de développement**. L'historique de développement peut être sauvegardé et appliqué de nouveau après un changement de la spécification. L'historique de développement est représenté par l'Equation 2.

$$((S_0 \times T_0) \times T_1) \ldots \times T_n \Rightarrow S_{n+1}$$

Equation 2: L'historique de développement.

4.2.2 Etapes du raffinement

Le système dispose de trois ensembles de primitives qui travaillent soit sur la **structure**, le **comportement** ou la **communication**. L'utilisateur guide le processus d'interaction et choisit les transformations nécessaires pour obtenir la solution désirée. Une nouvelle réalisation peut être obtenue par le changement de la séquence des primitives activées.

La Figure 36 présente le flot de découpage. Le découpage commence avec en entrée une spécification du système en SDL. Ensuite l'utilisateur réalise le raffinement de la spécification en utilisant les trois ensembles de primitives de transformation incrémentalle sur le modèle Solar. Finalement le code C/VHDL est généré.

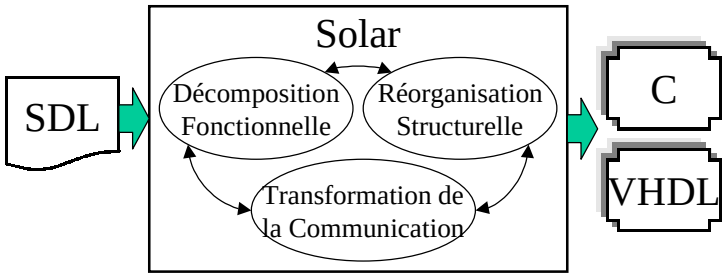


Figure 36: Flot de découpage.

La **décomposition fonctionnelle** permet la division des comportements qui ont besoin d'être exécutés sur plusieurs processeurs différents. Cette étape travaille sur des tables d'états et génère éventuellement des éléments de communication additionnels pour que les différentes parties du comportement puissent communiquer (voir Figure 37).

La **réorganisation de la structure** fixe le nombre de processeurs abstraits et décide de l'affectation des différentes fonctions sur les processeurs alloués. Chacune des différentes partitions peuvent contenir une ou plusieurs fonctions venant de la spécification initiale. Plusieurs fonctions peuvent être affectées à une simple partition afin de permettre le partage des unités fonctionnelles ou des ressources de communication. Cette étape travaille sur des unités de conception. Chaque processeur abstrait peut être réalisé en matériel ou en logiciel.

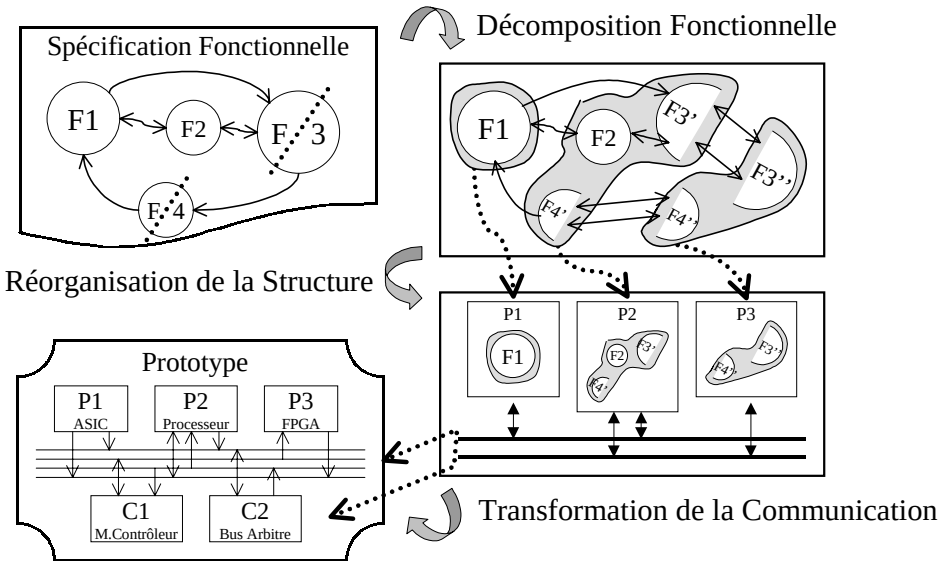


Figure 37: Etapes du découpage transformationnel.

La décomposition de la fonctionnalité et la réorganisation de la structure, sont suivies par la **transformation de la communication**. Dans cette étape, les processus communicants (à travers un schéma de communication de haut niveau) sont transformés en processeurs communicants à travers des signaux et partageant le contrôle de la communication. Cette étape travaille sur des canaux de communication.

L'organisation du découpage en plusieurs étapes réduit la complexité du problème. L'utilisateur contrôle l'historique de développement de découpage en disposant d'un environnement interactif qui lui permet un contrôle précis du processus de synthèse. Nous pouvons considérer cette méthodologie comme une compilation guidée par l'utilisateur, où le concepteur utilise des efforts additionnels pour produire une réalisation plus efficace [Kru92]. L'utilisateur dispose aussi d'un bon contrôle du processus de conception et pour faciliter l'interaction pendant les transformations incrémentalles, un environnement graphique a été développé. Cet environnement est présenté dans le chapitre 3.

4.3 Les primitives de transformation

Le processus de découpage est décomposé de trois ensembles de transformations : la décomposition fonctionnelle, la réorganisation de la structure et la transformation de la communication. Toutes ces transformations utilisent un ensemble de primitives qui exécutent des transformations telles que : diviser, grouper, déplacer, aplatir et affecter. Ces transformations permettent de décomposer, de composer et de transformer des objets Solar. Le Tableau 3 résume ces primitives de transformation.

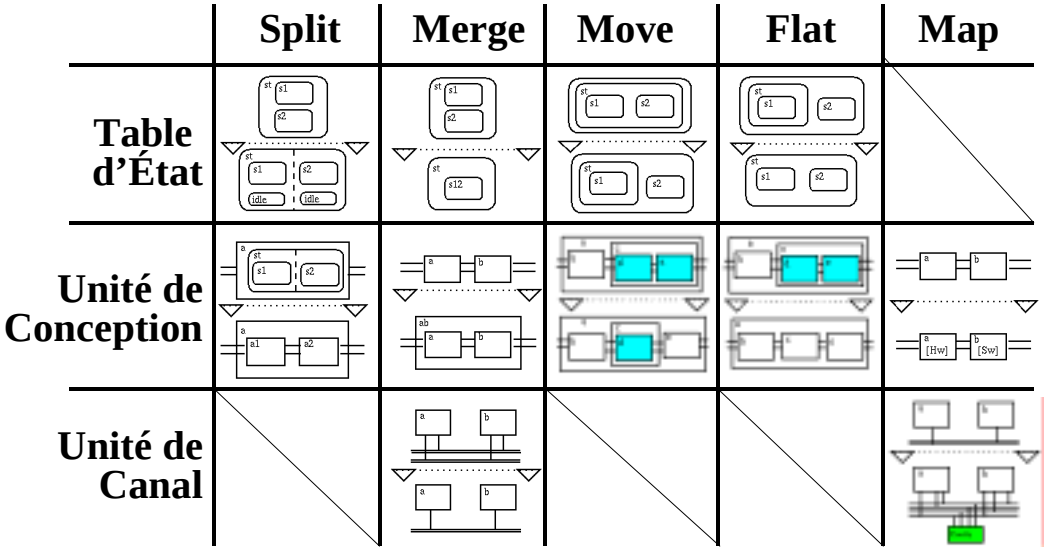


Tableau 3: Primitives de transformation.

Avant l'application de chaque primitive de transformation, le système vérifie les **conditions d'applicabilité**. L'utilisateur peut appliquer les primitives dans un ordre quelconque, tant que les conditions d'applicabilité sont respectées. L'effet de ces primitives est illustré dans les prochains paragraphes en utilisant l'exemple de conception conjointe d'un répondeur téléphonique (*Answering Machine* [Gaj94]). Dans cet exemple, une séquence de primitives de transformation est exécutée dans le but d'arriver à la réalisation désirée. Un autre exemple plus élaboré est illustré dans la section 4.7.2.

La Figure 38 représente la spécification système initiale du répondeur téléphonique. Cette spécification est modélisée en Solar et comporte quatre processus : un contrôleur

(*ctrl*) ; deux cassettes (*dec1* et *dec2*) pour lire l'annonce et pour enregistrer les messages ; et un compteur du *timer*. Le contrôleur est le processus principal, il est responsable de la coordination de l'exécution des ressources représentées par les autres processus. Cette description au répondeur téléphonique peut être dérivée d'un modèle SDL ou d'un autre langage au niveau système.

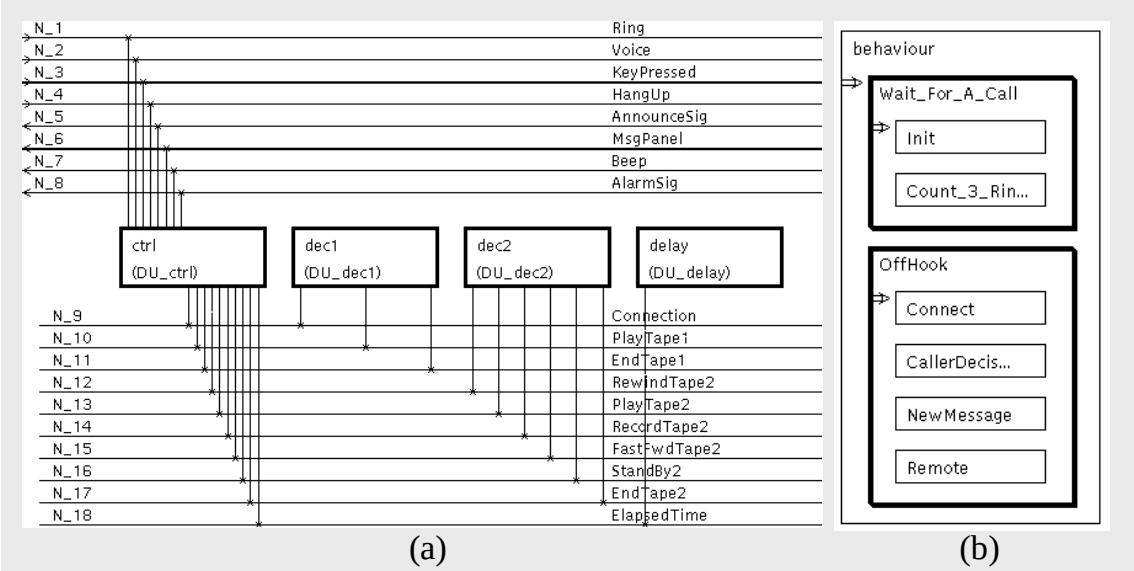


Figure 38: Exemple d'un répondeur téléphonique.

La Figure 38.a montre les quatre processus (représentés par des boîtes), les ports externes d'entrée/sortie (interconnexions 1 à 8) et les signaux internes (interconnexions 9 à 18). Le processus *ctrl* a une description comportementale, appellée *behaviour*, présentée dans la Figure 38.b. Elle est décrite par une table d'états composée de deux machines (*Wait_For_A_Call* et *OffHook*). La hiérarchie des machines est représentée par des boîtes imbriquées dans la représentation graphique. Les boîtes allignées verticalement ont une exécution séquentielle et les boîtes allignées horizontalement ont une exécution parallèle (Figure 39.d).

Les sections qui suivent utilisent l'exemple du répondeur téléphonique pour illustrer l'effet des primitives de transformation. Dans cet exemple, les transformations ont pour but de raffiner la spécification pour arriver à une réalisation où le comportement du processus *ctrl* est divisé en deux partitions. Les fonctions appartenant à la machine *Remote* vont être réalisées en logiciel et les autres fonctions vont être réalisées en matériel. La partie logicielle est traduite en code C et exécutée sur un processeur standard. La partie matérielle est convertie en VHDL comportemental et réalisé sur un processeur dédié ou sur un circuit programmable. Comme indiqué précédemment, l'ensemble des fonctions appartenant à chaque partition est le résultat de l'intervention de l'utilisateur.

Une description détaillée des primitives de transformation structurelles est disponible dans l'appendice A.

4.4 La décomposition fonctionnelle

Les primitives de décomposition fonctionnelle sont utiles pour transformer le comportement, i.e. les tables d'états. Ces primitives ont fait l'objet de la thèse de Tarek Ben Ismail [Ism96]. Une brève description des primitives appartenant à la décomposition

fonctionnelle est donnée dans ce qui suit (le terme machine est utilisé pour représenter un état ou une table d'états):

- ***Split*** (diviser): cette primitive décompose une machine séquentielle en un ensemble de sous-machines parallèles. Chaque machine qui en résulte peut être placée dans un processeur différent. Le contrôle des processus est réalisé par des signaux et par des états d'attente ajoutés à chaque bloc résultant de la division ;
- ***Merge*** (grouper) : cette primitive groupe un ensemble de machines séquentielles en une seule machine. Le regroupement peut permettre le partage de ressources et la réduction du coût de communication (registres et unités fonctionnelles) ;
- ***Move*** (déplacer) : cette primitive transforme la hiérarchie d'une machine. La transformation peut être utilisée pour déplacer le code d'une réalisation logicielle à une réalisation matérielle, et vice-versa ;
- ***Flat*** (aplatir): cette primitive met à plat la hiérarchie d'une machine.

Dans le cas du répondeur téléphonique, le découpage commence avec l'application des primitives de décomposition fonctionnelle dans l'ordre suivante (voir Figure 39):

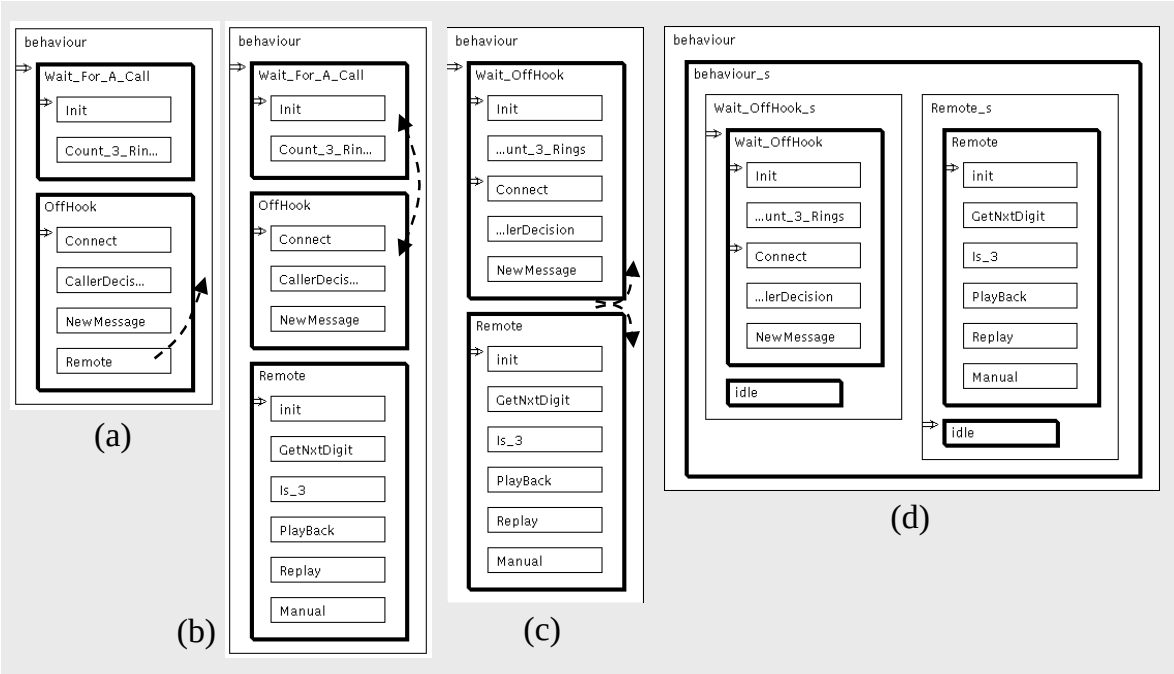


Figure 39: La décomposition fonctionnelle du répondeur téléphonique.

- Déplacer la machine *Remote* au niveau racine de la hiérarchie (*Move Remote *), voir Figure 39.a ;
- Grouper les machines *Wait_For_A_Call* et *OffHook* (*Merge Wait_For_A_Call OffHook*), voir Figure 39.b ;
- Décomposer la machine séquentielle *behaviour* en un ensemble de machines parallèles (*Split behaviour*), voir Figure 39.c.

A ce moment, le comportement du processus *ctrl* est composé de deux machines parallèles prêtes pour le découpage en processus communicants. Nous pouvons voir, dans la Figure 39.d, les états d'attente (*idle*) ajoutés à la spécification permettant l'exécution mutuellement exclusive des deux machines.

4.5 La réorganisation de la structure

L'objectif de la réorganisation de la structure est de distribuer le comportement du système sur l'ensemble des unités de conception, où chaque unité correspond à un processeur de l'architecture. C'est la principale transformation du découpage logiciel/matériel. Les primitives de réorganisation de la structure permettent la transformation de la hiérarchie structurelle du système. Ces primitives peuvent être appliquées plusieurs fois pendant la réorganisation structurelle. Le raffinement des interconnexions entre les unités est réalisé parallèlement a celui de la structure. Ces primitives sont:

- ***Split*** (diviser): cette primitive raffine le comportement des processus parallèles dans le but de découper ces processus en un ensemble d'unités de conception indépendantes. Les unités résultantes communiquent via des canaux abstraits et des signaux d'entré/sortie. Les données partagées par les machines d'états découpées (variables globales) sont représentées par des canaux abstraits dans la nouvelle description. Ces canaux abstraits se transforment normalement en une réalisation du type variable partagée, lors de la synthèse de la communication. Dans chaque unité générée, les accès à des variables globales sont remplacés par des appels à des services de communication offerts par le canal. Typiquement les services *Get* et *Put* appartenant au canal sont utilisés pour permettre l'accès aux valeurs des variables. Les accès concurrents aux variables appartenant aux processus parallèles sont contrôlés par le contrôleur du canal. La Figure 40 représente le résultat du découpage d'un système composé de deux machines concurrentes qui partagent une variable commune. Après l'opération de découpage structurel, les processus générés peuvent partager des données en utilisant le canal abstrait appelé *X_Abstract_Channel*;

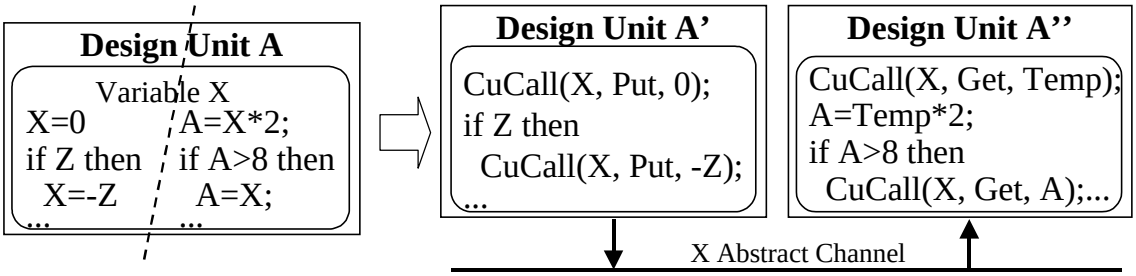


Figure 40: La primitive *Split* sur la structure.

- ***Merge*** (grouper): cette primitive groupe un ensemble d'unités en une nouvelle unité de conception. Cette opération est généralement utile pour rassembler les differents modules affectés au même processeur de l'architecture. La primitive *merge* travaille de deux manières différentes, suivant le type de processus impliqué dans l'opération:
  - **Groupement comportemental**: Quand les unités qui ont subie l'opération de groupement appartiennent au niveau comportemental, la primitive réalise le groupement des machines d'états finis des unités. Dans ce cas, la primitive réalise la création d'une unité de conception responsable de l'exécution concurrente du comportement des unités qui ont subie l'opération de groupement. Dès que possible, les canaux et les signaux de communication entre ces processus sont convertis en variables internes;
  - **Groupement hiérarchique**: Quand les unités qui ont subie l'opération de groupement appartiennent au niveau structurel de la hiérarchie, la primitive travaille sur la structure des unités. Une unité structurelle de conception est créée

pour contenir des instances aux unités qui ont subie l'opération de groupement. Le réseau de communication est adapté à la nouvelle hiérarchie;

- ***Move*** (déplacer): cette primitive déplace une unité de conception dans la hiérarchie. Elle est normalement utile pour préparer la structure à l'opération de groupement. L'objectif de grouper est mettre ensemble les processus qui vont être exécutés sur un même processeur dans l'architecture cible. Le réseau de communication est adapté à la nouvelle structure ;
- ***Map*** (affecter): cette primitive permet l'affectation de l'option de réalisation matérielle/logicielle à chaque unité de conception ;
- ***Flat*** (aplatir): cette primitive met à plat la structure hiérarchique des unités du système.

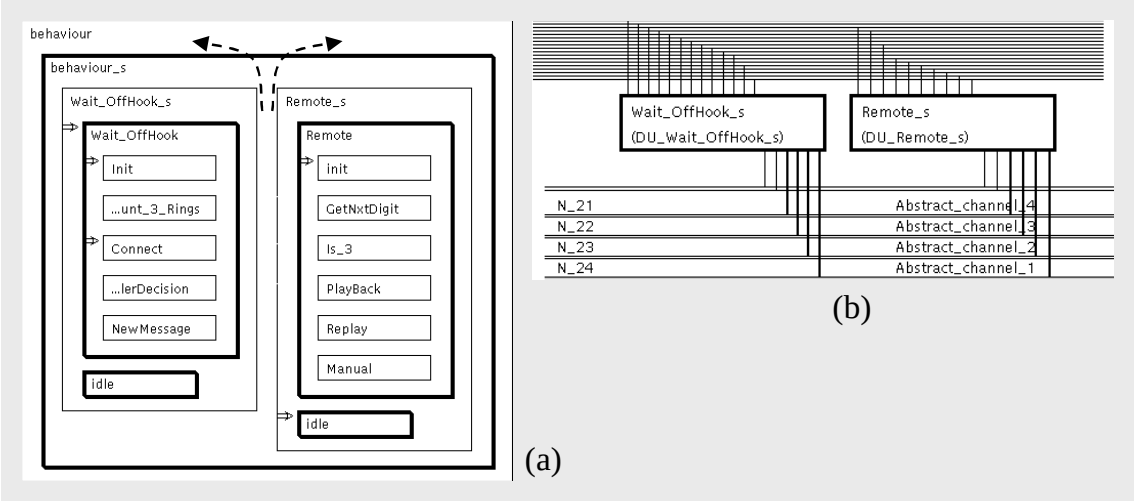


Figure 41: La réorganisation de la structure du répondeur téléphonique (*Split*).

La Figure 41 présente la suite de notre exemple du répondeur téléphonique et montre l'utilisation des primitives de réorganisation de la structure. La Figure 41.a présente la machine *behaviour_s*, composée de deux machines concurrentes: *Wait_OffHook_s* et *Remote_s*. L'utilisation des primitives de réorganisation de la structure commence avec l'application de la primitive *split* qui divise la machine *behaviour_s* en deux unités distinctes. Dans cet exemple (Figure 41.b), quatre canaux ont été créés. Ces canaux remplacent les variables globales communes des machines découpées.

4.6 La transformation de la communication

Cette étape transforme un système composé de processus qui communiquent à l'aide des primitives de haut niveau (canaux abstraits) en un système composé de processus interconnectés qui communiquent à l'aide des signaux [Dav98]. Cette transformation est réalisée en se basant sur une bibliothèque de communication composée d'un ensemble de protocoles simples. La Figure 42.a représente deux processus qui communiquent via des canaux abstraits.

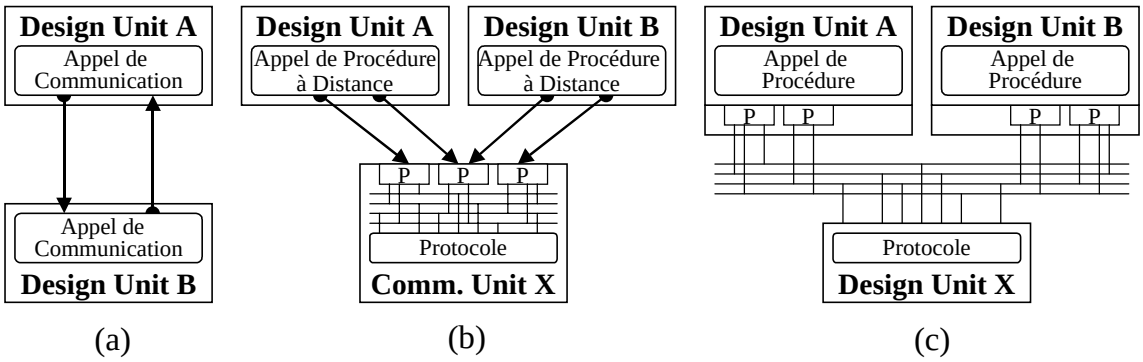


Figure 42: Modèle de communication.

L'allocation du protocole est la première étape de la transformation de la communication (Figure 42.b). L'allocation se base sur les caractéristiques de chaque protocole et peut être réalisée soit automatiquement par des outils disponibles, soit manuellement par l'utilisateur. Le choix d'un protocole pour réaliser un canal abstrait doit prendre en considération les performances nécessaires et le coût de la réalisation. La réalisation d'un protocole remplace aussi les appels de procédures à distance (RPC) par des procédures de communication locales. Les procédures de communication résultent de l'expansion des méthodes du canal aux unités interconnectées. Ensuite, l'étape de construction de l'interface ajoute à chaque unité le code nécessaire à la communication, en fonction du protocole sélectionné (Figure 42.c).

Deux primitives sont disponibles pour la transformation de la communication :

- ***Map*** (affecter): cette primitive affecte une unité de communication de la bibliothèque physique à chaque canal de communication de haut niveau et génère les interfaces entre les unités interconnectées;
- ***Merge*** (grouper): cette primitive permet de remplacer deux ou plusieurs canaux abstraits par un seul canal abstrait. Dans plusieurs cas, cette opération réduit considérablement le coût de communication. Par exemple, plusieurs canaux représentant des variables partagées peuvent être groupés en un seul canal qui gère ces variables en utilisant une mémoire commune. L'utilisation de cette primitive est conditionnée par la disponibilité d'un protocole dans la bibliothèque capable de réaliser le résultat du groupement.

La Figure 43 montre l'utilisation de la primitive *merge* et *map* dans le cas de notre exemple du répondeur téléphonique. La spécification est composée de deux unités de conception (*Wait_OffHook_s* and *Remote_s*) qui communiquent via quatre canaux abstraits (Figure 43.a). La Figure 43.b montre le résultat de l'application de la primitive *merge*. Les quatre canaux abstraits sont groupés en un seul canal. La Figure 43.c montre le résultat de l'application de la primitive *map*. Dans ce cas le canal abstrait est réalisé en utilisant un protocole du type variable partagée, disponible dans la bibliothèque de protocoles (Figure 43.d).

La prochaine étape de cet exemple est l'affectation de l'option de réalisation logicielle ou matérielle et la génération du code C/VHDL correspondant.

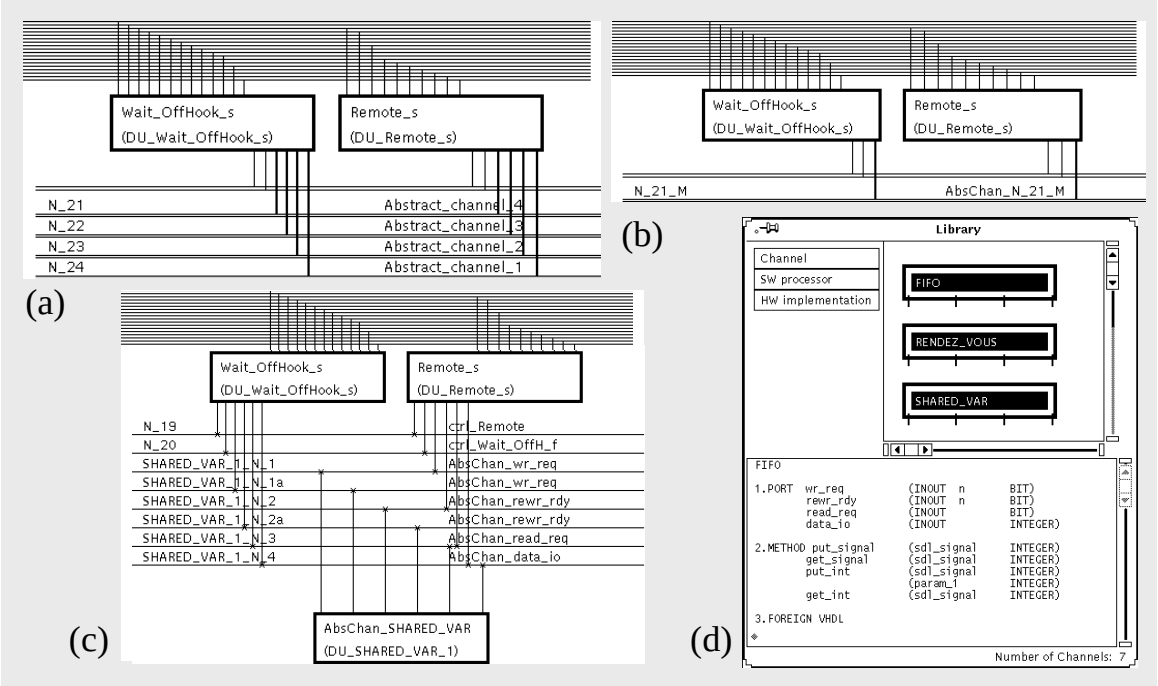


Figure 43: La transformation de la communication du répondeur téléphonique.

4.7 L'application de l'approche transformationnelle

Cette section présente l'application de l'approche transformationnelle à deux exemples de conception conjointe. Le premier exemple correspond à un simple système appelé Send-Receive. Ce système modélise l'échange de données entre deux processus. Le deuxième exemple correspond à une application plus conséquente, un Contrôleur de bras de robot.

Ces deux exemples présentent le flot complet de conception conjointe. Chaque système commence avec une spécification initiale en SDL, et produisent une architecture logicielle/matérielle distribuée. Toutes les transformations appliquées dans ces exemples sont suffisamment rapides pour paraître instantanées lors de l'interaction avec l'utilisateur. Aucune de ces primitives ne prend plus de cinq secondes de temps CPU sur une station de travail du type SPARC 20.

4.7.1 L'exemple Send-Receive

Send-Receive est un système composé de deux processus qui échangent des données. La conception commence avec une spécification initiale en SDL. Cette description initiale, décrite en SDL, est présentée dans la Figure 44.a-e.

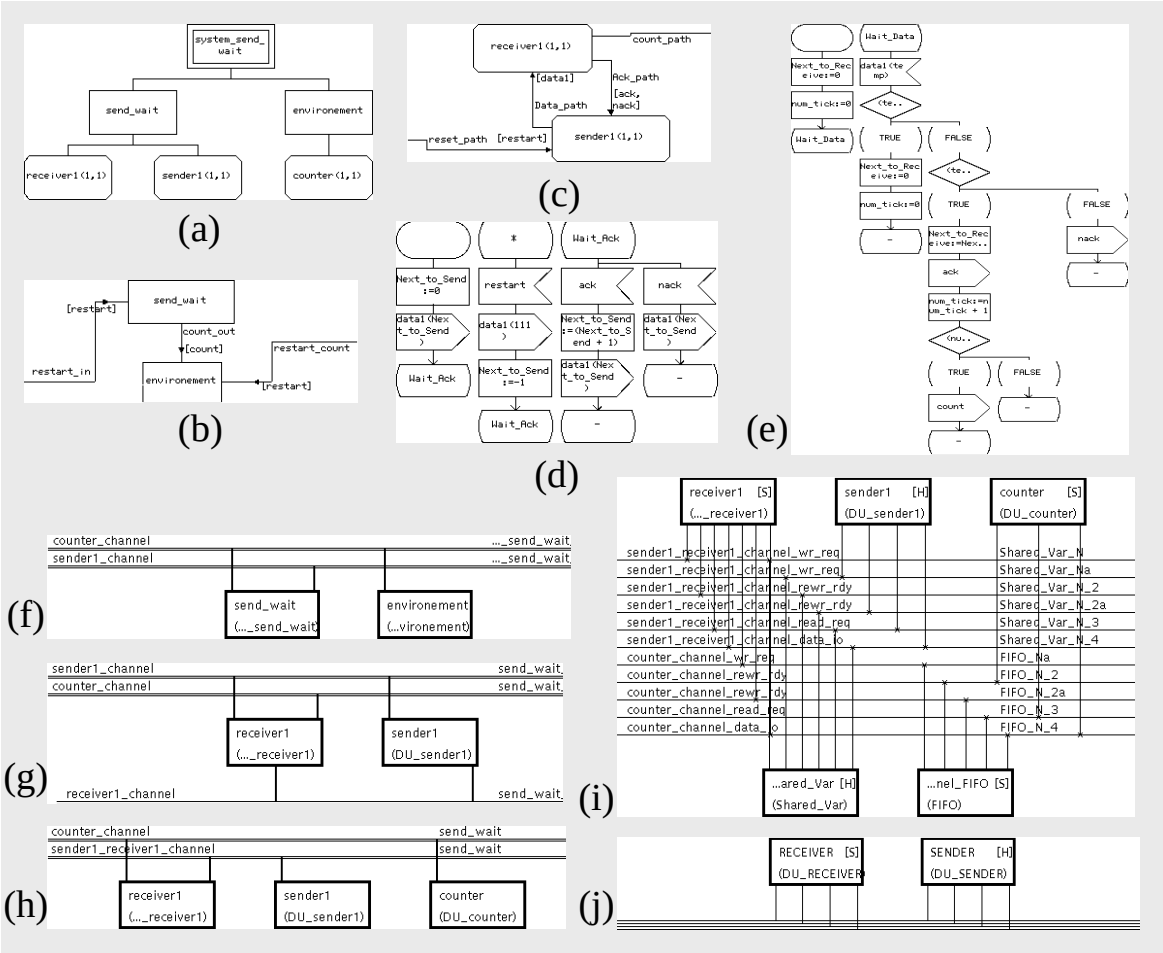


Figure 44: De SDL à C/VHDL.

La Figure 44.a représente la hiérarchie globale de la spécification en SDL. La Figure 44.b montre la première couche du modèle et la Figure 44.c montre l'organisation de la machine *Send_wait*. Cette machine est composée de deux processus qui s'échangent des messages.

La description du comportement des machines *sender1* et *receiver1* est également représentée dans cette figure. Le comportement de ces machines est décrit par des machines d'états finis, où des formes géométriques (boîtes, losanges et cercles) sont utilisées pour représenter des actions (opérations, conditions et états).

Le format intermédiaire Solar est obtenu par la conversion automatique à partir de la spécification initiale en SDL. Pendant cette conversion, les opérations suivantes sont réalisées [Dav97]:

- les processus sont transformés en unités de conception;
- le comportement des processus est transformé en tables d'états;
- les interconnexions entre les processus sont transformées en canaux abstraits.

Le modèle Solar initial est montré dans la Figure 44.f-g. La séquence d'étapes de raffinement de cet exemple est listée ci-dessous :

- Le raffinement commence avec le ***flat*** pour mettre à plat la hiérarchie des processus *send_wait* et *environment*. Cette opération facilite la visualisation et le traitement des interconnexions par étapes suivantes de transformation de la communication;

- L'opération de ***merge* sur les canaux** *sender1_channel* et *receiver1_channel* permet la réduction du coût de communication. Le résultat de ces deux premières transformations est montré dans la Figure 44.h ;
- Des canaux de la bibliothèque sont choisis pendant l'opération d'allocation des unités de communication. Les protocoles *fifo* et *Shared_variable* sont utilisés pour réaliser respectivement les canaux *counter_channel* et *sender1_receiver1_channel* ;
- La primitive ***Map*** remplace les canaux abstraits par le protocole choisit et génère les signaux et l'interface correpondante (Figure 44.i) ;
- Les processus *receiver1*, *counter* et le contrôleur du canal *fifo* sont attribués à une réalisation logicielle, et le processus *sender1* et le contrôleur du canal *Shared_variable* sont attribués à une réalisation matérielle. Cette assignation est identifiée par les symboles [S] et [H] dans la Figure 44.i. La primitive qui réalise cette assignation est le *Map* structurel ;
- Finalement la primitive ***merge* sur la structure** groupe les différentes unités de conception qui vont être exécutées sur le même processeur. Le résultat du groupement est montré dans la Figure 44.j.

Suite à ces transformations, le code C/VHDL peut être généré pour chaque processus. Le comportement des processus logiciels est converti en langage C, et le comportement des processus matériels en VHDL. Ce modèle C/VHDL peut être converti en une architecture logicielle/matérielle en utilisant des outils de synthèse existants.

La taille de la description du système varie beaucoup durant les étapes de la conception. Le Tableau 4 illustre cette évolution. La différence de taille du code entre la spécification Solar générée à partir de la spécification SDL, et la spécification Solar transformée par les raffinements successifs, est dû principalement à l'ajout de la description du protocole qui contrôle l'échange des informations entre les processus.

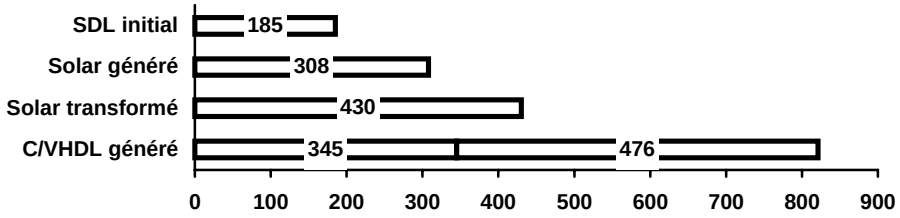


Tableau 4: Evolution de la taille de la spécification du *send_receive*.

4.7.2 L'exemple du contrôleur de bras de robot

Le Controleur de Bras de Robot (*Robot Arm Controller*) est un système qui ajuste les positions et les paramètres de vitesse de dix-huit moteurs commandant le bras d'un robot à trois doigts. Ce traitement est destiné à éviter les problèmes de discontinuité du mouvement des moteurs pendant leurs opérations. Les changements de vitesse des moteurs doivent suivre une courbe souple d'accélération et de décélération pour des raisons mécaniques. Le temps de réponse doit être inférieur à 6 ms (voir Figure 45).

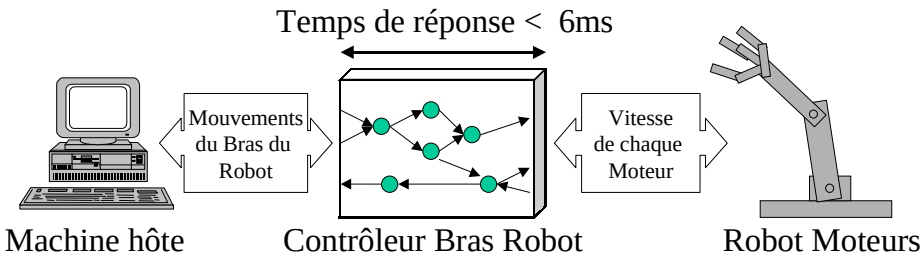


Figure 45: Le contrôleur de bras de robot.

Le diagramme des blocs de base du système et la hiérarchie des blocs en SDL sont représentés dans la Figure 46.a. La Figure 46.b montre le premier niveau de la hiérarchie du modèle SDL. Comme nous pouvons le voir, le système est composé de trois blocs qui communiquent à travers des canaux. La Figure 46.c détaille l'organisation du bloc *AdaptativeSpeedControl*. Celui-ci est composé de trois processus qui communiquent par des routes de signaux (*signalroutes*). Ces figures ont été générées à partir de l'outil ObjectGeode [Obj97] qui offre un environnement graphique de description SDL.

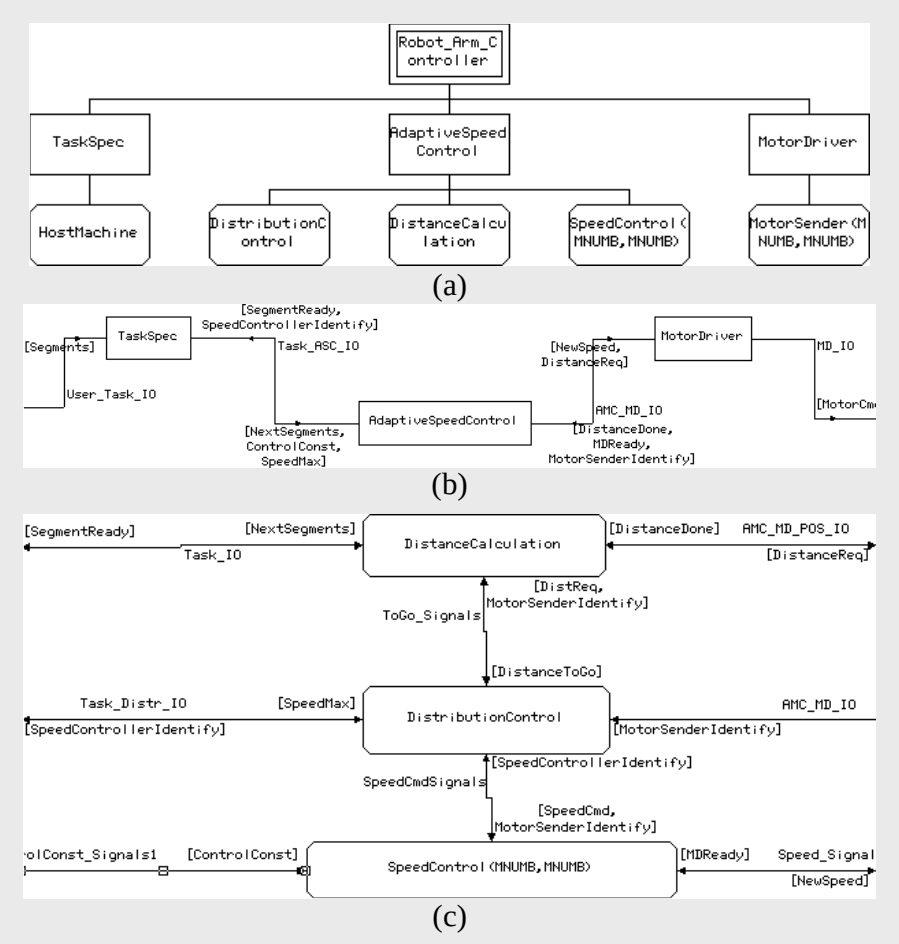


Figure 46: Représentation SDL initiale.

Dans une brève description de la fonctionnalité, le bloc *AdaptativeSpeedControl* reçoit du bloc *TaskSpec* des données concernant le mouvement que le bras du robot doit effectuer. Le processus *AdaptativeSpeedControl* calcule la vitesse nécessaire à chaque moteur, de façon à ce que tous arrivent à la position désirée dans le même espace de temps. Le moteur qui doit parcourir la plus grande distance tourne à la vitesse maximale. La

vitesse de chacun des autres moteurs est adaptée à celle-ci. Ces valeurs de vitesse sont envoyées au processus *MotorDriver* sous forme d'impulsions de contrôle. Le processus *MotorDriver* réalise la conversion des impulsions en signaux de contrôle acceptable par chaque moteur.

Les principaux processus de cette spécification sont :

- *HostMachine*: assure l'interface entre l'utilisateur et le système. Il est responsable du stockage des segments et du calcul des constantes des moteurs ;
- *DistanceCalculation*: garde les informations actualisées sur la distance à parcourir par chaque moteur à partir de la position actuelle du bras ;
- *DistributionControl*: ce processus identifie le moteur qui doit tourner à la vitesse maximale et envoie des commandes en échelle pour les autres contrôleurs ;
- *SpeedControl*: ce processus convertit les valeurs de la vitesse des moteurs en une courbe qui respecte le pire cas d'accélération et de décélération ;
- *MotorSender*: assure l'interface avec les moteurs pas à pas. Ce processus convertit la vitesse des moteurs en impulsions modulées en fréquence.

Les contraintes de réalisation pour ce système sont :

- le temps de réponse du système doit être inférieur ou égal à 6 ms;
- le changement de vitesse des moteurs doit être le plus lisse possible;
- le dépassement de la position finale du bras doit être le plus faible possible.

La simulation du modèle SDL permet la validation de la fonctionnalité du système. La Figure 47 montre le résultat de la simulation. Dans la figure, nous pouvons voir le changement de la vitesse de deux moteurs qui appartiennent au bras du robot. Les courbes représentent la relation vitesse/temps des moteurs.

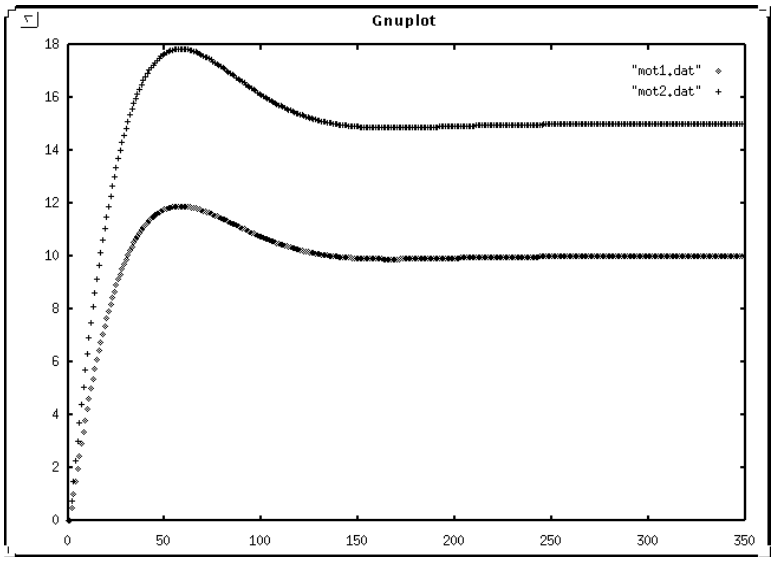


Figure 47: La simulation fonctionnelle du modèle SDL.

Lors du découpage logiciel/matériel, les blocs *SpeedControl* et *MotorSender* seront affectés au matériel et les autres blocs au logiciel. Dans cette réalisation, l'utilisateur a choisi une réalisation matérielle des processus qui ont un calcul intensif afin de respecter les contraintes temps réel. La Figure 48 illustre l'utilisation des primitives de raffinement sur le modèle Solar.

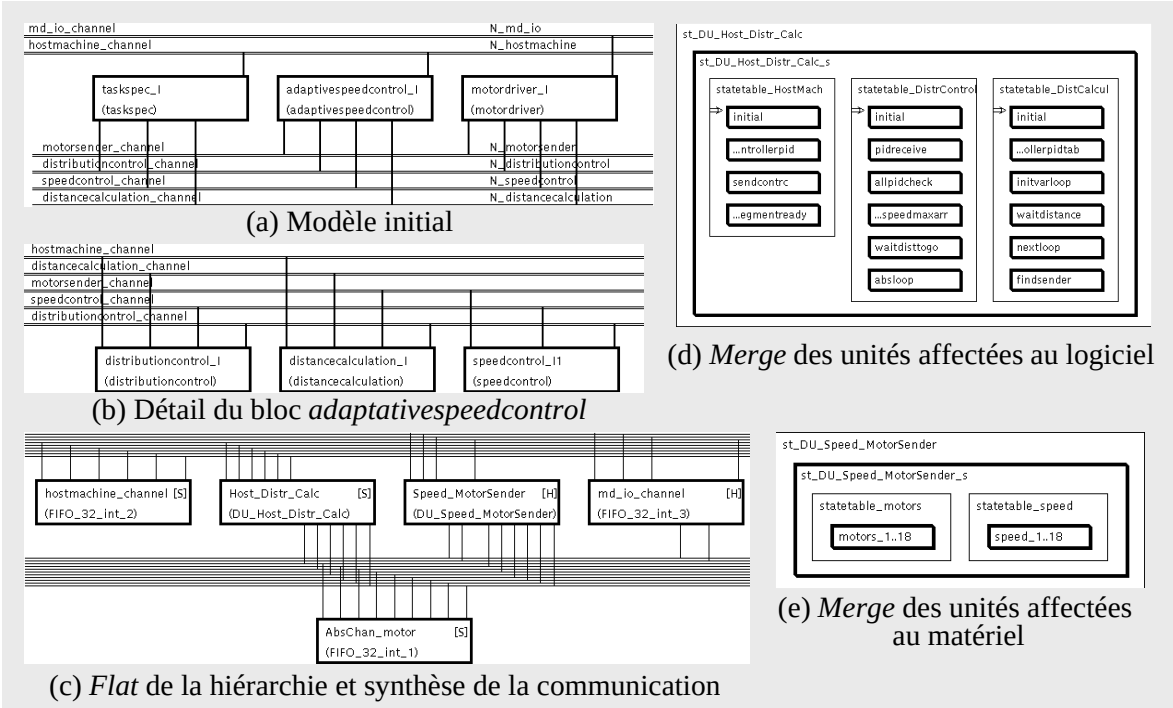


Figure 48: Etapes de raffinement du contrôleur de bras de robot.

Le résultat de ces étapes de raffinement est représenté dans la Figure 48. La Figure 48.a-b montre la structure du modèle initial SDL après la traduction en Solar. La Figure 48.c montre le résultat de l'ensemble des transformations. Les primitives activées sont les suivantes: *flat* et *merge* sur la structure initiale; *merge* et *map* sur les canaux. Les canaux abstraits sont assignés à des protocoles de la bibliothèque. La Figure 48.d-e montre la machine d'états finis résultant de l'application du *merge* structurel.

Les processus *SpeedControl* et *MotorSender* sont alloués 18 fois. Une instance pour chaque moteur est nécessaire afin d'arriver à la performance requise. Sur la Figure 49 nous pouvons voir la structure finale du système et ainsi que des parties du code C et VHDL générés.

Figure 49: Le prototype virtuel du contrôleur de bras de robot.

Dans cet exemple, le nombre de lignes de code des différentes étapes de la conception est indiqué dans le Tableau 5. Le code VHDL généré contient 853 lignes pour chacune des 18 instances du moteur.

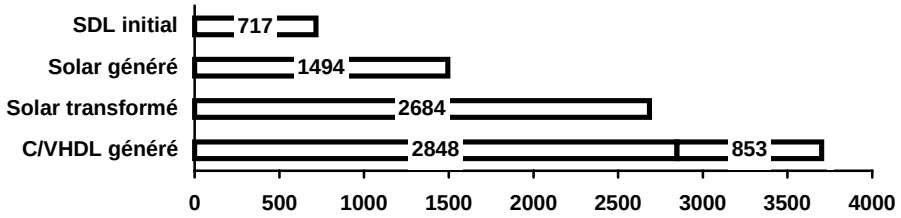


Tableau 5: Evolution de la taille de la spécification du contrôleur de bras de robot.

4.8 Evaluation

Le travail présenté dans ce chapitre correspond à une méthodologie de conception conjoint de systèmes mixtes logiciels/matériels. L'objectif a été de combler l'espace existant entre la spécification au niveau système et la conception d'une architecture logicielle/matérielle. Cette approche permet une réduction du temps de conception et une

exploration facile de l'espace des solutions. Pour arriver à cet objectif, une approche de découpage transformationnel semi-automatique guidé par l'utilisateur a été développée.

Cette méthodologie de conception basée sur des raffinements incrémentaux satisfait aux défis mis en lumière dans la section 1.1 :

1. **L'approche semi-automatique de découpage d'une spécification fonctionnelle**: L'interaction avec l'utilisateur est nécessaire tant que les méthodes d'estimation ne sont pas assez précises pour guider le découpage automatique. L'intervention de l'utilisateur permet la manipulation des systèmes complexes appartenant à une large gamme d'applications;
2. **La prédiction du résultat du découpage**: L'utilisateur sélectionne l'élément qui va être transformé par chaque opération. Le résultat est immédiatement obtenu et affiché par l'interface graphique. De cette façon, l'utilisateur contrôle le flot des opérations ;
3. **L'utilisation de l'expertise de l'utilisateur**: Si l'utilisateur a une solution architecturale en tête, il peut forcer cette solution en utilisant la séquence appropriée de transformations. La solution initiale peut être utilisée par l'utilisateur comme base pour explorer de nouvelles solutions ;
4. **L'Exploration de l'espace des solutions**: L'approche transformationnelle permet à l'utilisateur de guider le flot de conception pour obtenir la solution désirée. Le système Cosmos est suffisamment rapide pour permettre l'exploration de plusieurs solutions, à partir d'une même spécification et ceci dans un temps restreint.

4.9 Conclusion

Dans ce chapitre nous avons présenté une méthodologie de découpage logiciel/matériel réaliste, basée sur l'interaction avec l'utilisateur. Cette méthodologie est capable de manipuler des systèmes distribués. L'utilisateur dispose d'un puissant ensemble de primitives pour le découpage fonctionnel, la réorganisation de la structure et la transformation de la communication.

La conception conjointe commence avec une spécification au niveau système donnée en SDL. Cette spécification initiale est raffinée à travers un processus de compilation guidé par l'utilisateur, avec le but de générer une architecture logicielle/matérielle distribuée, donnée en C/VHDL.

Cette approche a été illustrée sur deux exemples. L'utilisation de cette approche pour la conception d'un système complexe sera aussi présentée dans le chapitre 6. En contraste avec d'autres approches de conception conjointe, l’approche transformationnelle apporte des solutions élégantes aux principaux défis imposés par la conception des systèmes distribués:

- Utiliser l'expertise de l'utilisateur pendant le découpage;
- Permettre à l'utilisateur de comprendre les détails du processus de conception;
- Prendre en considération des solutions partielles;
- Permettre une facile exploration de l'espace des solutions.

Chapitre 5

Estimation de Performances

The trouble with computers is they do everything we tell them to do,
but not what we would like them to do.

Ce chapitre présente le modèle d'estimation de performance développé au cours de ce travail de recherche. L'estimation de performance permet une exploration rapide de l'espace des solutions, car l'utilisateur dispose d'un ensemble d'informations liées à la réalisation. L'utilisateur peut valider chaque réalisation et peut aussi optimiser sa propre stratégie de raffinement de la spécification. Le modèle d'estimation de Cosmos donne des résultats assez satisfaisants, puisqu'il prend en considération les aspects non déterministes de l'architecture, tels que la hiérarchie mémoire et le pipeline des instructions. L'estimation des partitions matérielles utilise des outils existants.

5.1 Introduction

L'utilisateur conçoit des architectures qui doivent satisfaire aux besoins fonctionnels, mais aussi aux restrictions de coût et aux objectifs de performance [Gaj98]. Une fois les besoins fonctionnels établis, lors l'étape de spécification, l'utilisateur doit réaliser son système en optimisant l'architecture. Le choix de la solution optimale dépend de plusieurs critères. Les critères les plus courants concernent les performances et le coût, mais d'autres critères mesurables peuvent être aussi importants pour certains domaines d'application.

Une approche d'estimation du logiciel et du matériel doit utiliser des méthodes complexes, parce qu'il existe un grand espace conceptuel entre le niveau système et le niveau physique qui doit être pris en compte par l'algorithme d'estimation. Cela signifie qu'il est difficile et parfois impossible de prévoir les caractéristiques d'une architecture à partir d'une spécification abstraite du niveau système avec une précision donnée. Un autre facteur à ajouter à la complexité de calcul est la vitesse de calcul. Un algorithme, qui modélise les caractéristiques de l'architecture et qui donne des résultats très proches des valeurs réelles, est beaucoup plus lent qu'un algorithme qui traite les aspects de l'architecture superficiellement.

Le conflit existant entre la vitesse de calcul de l'estimation et la précision du calcul est un facteur critique pour la plus grande partie des systèmes de conception conjointe existants, comme illustre la Figure 50. Certains systèmes mettent en pratique des approches de conception où les décisions de conception sont prises automatiquement par l'algorithme de découpage, à partir des résultats de l'estimation.

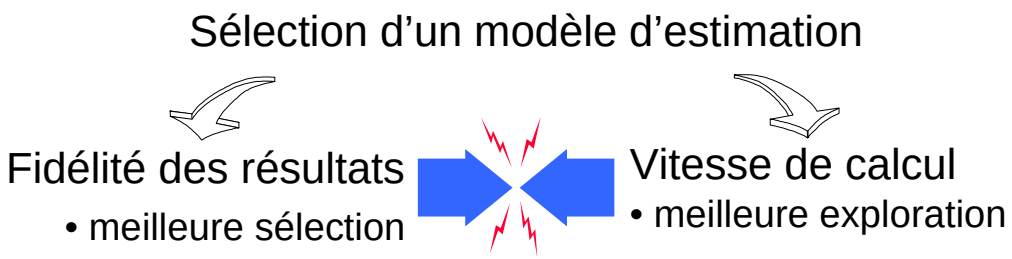


Figure 50: Le conflit entre la vitesse de calcul et la précision.

Une approche d'estimation qui donne des résultats proches des valeurs réelles permet une meilleure sélection de la réalisation. Par contre, cette précision dégrade le temps de réponse, ce qui restreint beaucoup l'exploration de l'espace des solutions. Ce conflit oblige les méthodologies à fixer une stratégie ou une heuristique afin de réduire l'espace des solutions exploré.

L'objectif de l'opération d'estimation est de permettre à l'utilisateur de prendre la plus grande partie des décisions de conception au niveau système, pendant les premières étapes de conception. Ces décisions, basées sur des valeurs fidèles, réduisent le temps de conception. Elles réduisent aussi les retours à des étapes antérieures de la conception, nécessaires pour la correction des erreurs dues à l'adoption de solutions architecturales inadéquates. Le temps de conception croit en fonction du nombre de fois où l'utilisateur doit revenir à des étapes antérieures de la conception jusqu'à l'obtention de l'architecture désirée (boucle de la conception). L'estimation permet aussi la réduction du coût de réalisation, puisqu'il n'est plus nécessaire de surdimensionner l'architecture afin d'être sur de satisfaire les contraintes.

Le Tableau 6 montre la relation existante entre la précision des résultats et la vitesse de calcul pour l'estimation du logiciel et du matériel. Il énumère 5 niveaux d'estimation par ordre croissante de précision permise par ces modèles. Pour chaque solution nous enumèrons les objets pris en compte pour l'estimation du logiciel et du matériel. Le modèle d'estimation, présenté dans cette thèse, obtient une bonne précision, dans la mesure où la modélisation de l'architecture prend en considération plusieurs aspects du côté logiciel et du côté matériel. Le niveau de complexité de ce modèle appartient au quatrième niveau du Tableau 6 (représenté par des caractères épais).

Modèle d'Estimation			Qualité du Résultat	
N	Matériel	Logiciel	Précision	Vitesse
1	Mem	Nombre d'instructions produites	Faible	Rapide
2	Mem+FU	Calcul statique de la fréquence + estimation du temps d'exécution	▼	▲
3	Mem+FU+Reg	Calcul dynamique de la fréquence + estimation du temps d'exécution	▼	▲
4	**Mem+FU+Reg+Mux**	**Calcul dynamique + pipeline + cache + approximation du nombre de cycles**	▼	▲
5	Mem+FU+Reg+Mux +wiring	Synchronisation + communication + estimation du nombre de cycles	Bonne	Lente

Tableau 6: Modèles d'estimation de performance.

Principalement, il y a deux défis imposés aux algorithmes d'estimation de performance:

- Du côté logiciel, la performance dépend du processeur utilisé mais aussi des outils et méthodes de compilation du logiciel. En fait, il est difficile de modéliser les techniques utilisées pour augmenter la vitesse d'exécution du code, comme la hiérarchie de mémoire, le pipeline d'instruction et les optimisations du compilateur;
- Du côté matériel, la difficulté est de modéliser les différentes possibilités de réalisation et les différentes décisions prises par la synthèse d'architecture.

Les différentes sections de ce chapitre illustrent l'approche d'estimation développée au cours de ce travail de recherche et integrée dans le système Cosmos. Cette approche prend en considération les aspects coût et performance pour l'évaluation de l'architecture d'un système. Ces informations sont présentées à l'utilisateur sous la forme d'un feed-back graphique. Les différents aspects d'évaluation d'une architecture, liés aux différents domaines d'application, comme par exemple la sécurité, la consommation et la réutilisation, doivent être traités de façon empirique par l'utilisateur.

5.2 Etat de l'art

La mesure de performance d'une réalisation est passée par plusieurs générations. Chaque génération rend obsolète la technique d'évaluation de performance de la génération précédente [Hen96].

La première génération a représenté la performance par la comparaison entre les réalisations. Le résultat était, par exemple, qu'une réalisation *X* était *n* fois plus rapide qu'une réalisation *Z*. Le problème c'était qu'il n'existait pas une méthodologie fixe pour mesurer la performance des réalisations. La deuxième génération a lié les mesures au

espace de temps nécessaire à l'exécution d'une opération donnée. Par exemple, le temps que la réalisation prenait pour exécuter un ensemble d'instructions. La génération suivante s'est basée sur l'utilisation d'un ensemble de programmes test (*benchmarks*) ou vecteur de données de test.

Actuellement, l'effort d'estimation est dirigé vers la construction d'approches qui modélisent le fonctionnement des éléments de calcul d'une architecture, comme les processeurs standards, les ASICs, et les ASIPs. La caractérisation, suivie par la mesure de performance, considère les composants de l'architecture comme étant des unités isolées, sans prendre en considération le temps de synchronisation et de communication.

Ci-dessous, nous donnons une brève liste des principales approches d'estimation existant pour la conception conjointe:

- SpecSyn [Gaj98, Nar96] réalise une approche d'estimation qui utilise la description comportementale du système et l'allocation des unités fonctionnelles. A partir de ces informations, SpecSyn calcule le nombre de pas de contrôle nécessaire en utilisant la technique d'analyse de flot (*flow-analysis*). Les modules logiciels sont compilés dans un code intermédiaire. La bibliothèque de caractérisation technologique est utilisée pour faire la correspondance entre le code intermédiaire et le processeur cible. SpecSyn utilise un modèle très simple du pipeline et de l'accès à la mémoire;
- Tosca [All97] réalise une approche statique d'estimation du matériel et du logiciel, basé sur la co-simulation et le calcul de la fréquence d'exécution. L'estimation du logiciel est basée sur la caractérisation du processeur et sur la caractérisation du compilateur pour le calcul du temps d'exécution minimal et maximal d'un programme;
- Vulcan [Gup94] modélise le comportement du système par un graphe, où les noeuds représentent les opérations et les arcs représentent les dépendances entre les opérations. Chaque noeud contient une valeur représentant le délai de propagation de l'opération. Le temps d'exécution du graphe est calculé par l'addition du temps d'exécution des noeuds appartenant au chemin d'exécution le plus long. Dans le cas d'un arc, connecté à deux noeuds assignés à des différentes partitions, un délai est associé à l'arc. Ce délai reflète le coût de la communication;
- Aparty [Lag91] utilise comme entrée pour l'estimateur un ensemble de partitions qui correspondent à des opérations. Le temps d'exécution de chaque partition est calculé par le nombre d'étapes de contrôle résultant de l'opération d'ordonnancement.

Aucun des modèles d'estimation présentés, ou connus par l'auteur, prend en considération les aspects dynamiques d'exécution des systèmes (comme les instructions de boucles et les instructions conditionnelles) et les caractéristiques des architectures modernes.

Le modèle d'estimation de Cosmos permet la validation d'une réalisation, plutôt que de servir comme base à un algorithme de découpage automatique. Cette caractéristique est liée au fait que Cosmos met en pratique une approche semi-automatique de découpage. De cette façon, le modèle d'estimation de Cosmos bénéficie d'un laps de temps plus flexible pour le calcul de l'estimation, en comparaison avec d'autres modèles. Cosmos renvoie à l'utilisateur des valeurs approximatives sur les caractéristiques de la réalisation pour qu'il interprète les résultats d'une façon plus réaliste qu'un simple algorithme automatique.

5.3 Les étapes de l'estimation

La méthode la plus précise pour l'obtention du temps d'exécution d'une réalisation est basée sur la synthèse suivie de l'émulation. Du côté logiciel, l'utilisateur peut compiler la spécification sur le processeur cible et mesurer dynamiquement ou statiquement la performance. Du côté matériel, la méthode la plus précise est basée sur la synthèse et le prototypage du matériel. Il est clair que cette méthode est très coûteuse, qu'elle consomme un grand de temps de conception et qu'elle n'est pas viable même pour un petit sous ensemble de réalisations.

Par exemple, la compilation d'un module logiciel sur plusieurs processeurs nécessite une infrastructure couteuse, composée de plusieurs compilateurs et de processeurs cibles, qui ne sont pas toujours disponibles. De cette façon, uniquement pour la détermination de performance des modules logiciels, même les systèmes les plus simples ont un grand coût de conception lié à l'achat des compilateurs et des processeurs cibles. La solution à ce problème est d'essayer de modéliser les étapes de la synthèse avec une approche automatique d'estimation du logiciel et du matériel au niveau système.

Dans Cosmos, l'estimation au niveau système est divisée en deux étapes, comme l'illustre la Figure 51:

- L'**analyse et l'annotation de la spécification** ou le **calcul de la fréquence d'exécution**: Cette étape ajoute des informations à la spécification initiale. Il s'agit de la fréquence d'exécution de chaque instruction;
- La **modélisation de l'architecture,** divisée en **transposition** et **estimation**: Cette étape a lieu après le découpage. Elle fait la correspondance entre le comportement du système et les composants de l'architecture et réalise le calcul des performances de la réalisation.

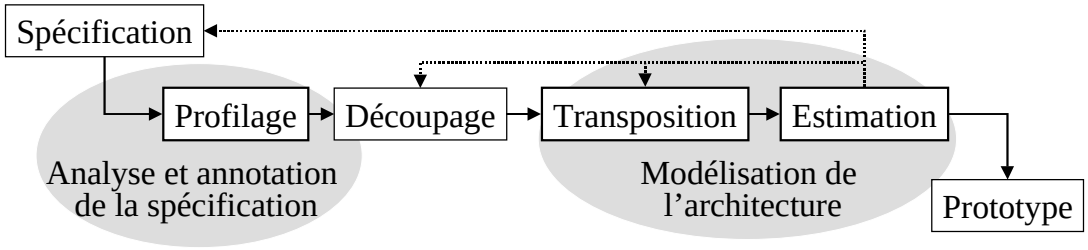


Figure 51: Etapes traditionnelles d'estimation de performance.

Les sections suivantes introduisent les techniques d'analyse de la spécification, de calcul du coût de la réalisation et de calcul de performance. Ces thèmes sont fondamentaux pour comprendre l'approche d'estimation de Cosmos, décrite dans la section 5.6.

5.4 Le temps d'exécution d'une spécification

Un principe important que l'utilisateur doit avoir en tête pendant le raffinement d'une réalisation est que les fonctions, ou instructions, ne s'exécutent pas de façon homogène. Généralement le temps d'exécution d'une spécification est consommé par un petit ensemble de fonctions tandis que le reste du code est rarement exécuté. Par exemple, les instructions responsables de l'initialisation sont exécutées un petit nombre de fois. Par

contre, les calculs à l'intérieur des boucles imbriquées peuvent prendre une grande partie du temps d'exécution du système.

L'évaluation des différentes possibilités de réalisation doit prendre en considération le cas le plus fréquent au détriment du cas le plus rare. Ce principe s'applique aussi lorsqu'il faut déterminer comment utiliser des ressources matérielles, car l'accélération obtenue est plus grande si la ressource est fréquemment utilisée.

La section 5.6.1 présente une approche dynamique de calcul capable de détecter les parties de la spécification qui ont une grande fréquence d'exécution. Le gain en performance, occasionné par l'amélioration d'une fraction du code, est donné par la loi d'Ahdahl (Equation 3).

$$Aceleration_{totale} = \frac{1}{(1 - Fraction) + \frac{Fraction}{Accélération_{bloc}}}$$

Equation 3: Le raffinement et le calcul du gain en performance (loi d'Ahdahl [Hen96]).

Cette équation mesure l'accélération d'une réalisation occasionnée par l'accélération d'une fraction du code. Supposons par exemple, qu'une fonction subisse un facteur accélérateur de 10 mais qu'elle ne représente que 7% du temps total d'exécution du système. Dans ce cas, l'accélération globale n'est que de 6%. Cette équation peut aussi être utile pour mesurer l'impact de l'utilisation d'un nouveau dispositif dans l'architecture.

Le concepteur peut utiliser la loi d'Ahdahl, en conjonction avec le résultat du calcul de la fréquence d'exécution pour décider sur quelle partie du code il doit concentrer ses efforts pendant le raffinement. Le calcul de l'accélération du temps total d'exécution d'un système, a partir du calcul de l'accélération d'une partie du code, n'est pas précis puisque d'autres aspects, comme la synchronisation et la communication, ne sont pas considérés.

5.5 La performance

Le calcul de performance est lié au temps de réponse du système à un stimulus externe, ou au temps d'exécution d'un bloc, d'une fonction ou d'une partie de la fonctionnalité du système. Normalement, l'utilisateur est intéressé par le calcul du temps écoulé entre le début et la fin d'un calcul, comme illustre la Figure 52. Ainsi, l'utilisateur peut comparer deux ou plusieurs réalisations avec les mêmes critères.

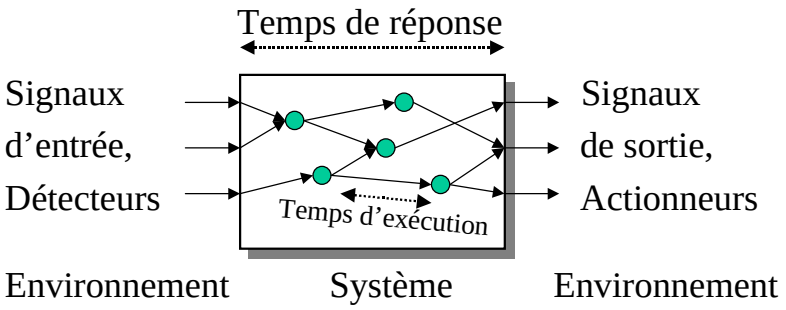


Figure 52: Performance d'un système.

Le moyen le plus précis pour mesurer la performance est, pendant l'étape de prototypage, mettre le système en exécution avec un ensemble de données d'entrée

représentatives et réellement mesurer la performance. Un simulateur du jeu d'instructions peut être utile quand le processeur n'est pas disponible. Avec le résultat de l'exécution ou de la simulation l'utilisateur peut faire la relation entre la performance des différentes réalisations et leur coût de production. Le problème est que, normalement, l'utilisateur de systèmes veut estimer la performance d'une réalisation virtuelle, i.e. une réalisation qui est encore dans la phase de synthèse. L'utilisateur ne dispose pas d'une réalisation pour faire de telles mesures réelles. D'un autre côté, la réalisation des solutions possibles, même d'un sous-ensemble, dans le seul but de mesurer ses performances est inconcevable.

5.5.1 Le calcul de la fréquence d'exécution des instructions

La disponibilité du code devant être exécuté sur un processeur logiciel n'est pas suffisante pour l'estimation de performance. Le processus d'estimation a besoin d'avoir des informations soit sur le flot d'exécution de la spécification, soit sur la fréquence d'exécution de chaque groupe d'instruction.

Il existe plusieurs façons de tracer le flot d'exécution d'une spécification, parmi lesquelles:

- L'annotation de la spécification suivie par la simulation au niveau système. Pendant la simulation le flot d'exécution est sauvegardé. Cette approche a besoin d'un simulateur capable d'exécuter la spécification initiale, lequel n'est pas toujours disponible;
- La génération d'une réalisation entièrement logicielle suivie par la compilation et l'annotation du code. Le code généré est exécuté et les informations concernant le flot d'exécution sont enregistrées;
- L'analyse statique de la spécification avec la prévision du flot de contrôle.

Fondamentalement, l'objectif de ces trois solutions est de détecter les changements dans le flot de contrôle occasionnés par les instructions conditionnelles et les boucles. Quant aux deux premières solutions, la détection se base sur l'exécution de la spécification. Ces solutions sont fiables mais la qualité du résultat dépend de la représentativité des données d'entrée. La deuxième solution réalise la traduction de la spécification vers le jeu d'instruction d'un processeur existant (le processeur cible n'est pas nécessaire).

5.5.2 Le calcul du temps d'exécution du logiciel

Les processeurs sont construits avec une horloge fonctionnant à une fréquence constante. La vitesse d'un processeur est donnée par la fréquence de l'horloge (MHz) ou par la durée du cycle d'horloge (ns). De cette façon, le temps d'exécution du code logiciel peut être représenté par l'Equation 4.

$$TempsExécution = NombreCycles \times TempsCycle = \frac{NombreCycles}{FréquenceHorloge}$$

Equation 4: Calcul du temps d'exécution du code logiciel.

Ce modèle simplifié d'estimation du temps d'exécution peut utiliser le nombre moyen de cycles d'horloge nécessaires à l'exécution d'une instruction. Cette valeur peut être obtenue par des mesures sur des exemples représentatifs ou à travers l'étude de l'architecture du processeur. Le nombre moyen de cycles d'horloge par instruction est le résultat du jeu d'instruction et des caractéristiques de réalisation de chaque processeur. Le

temps d'exécution d'une réalisation peut être calculé en utilisant l'Equation 5. Cette équation se base sur le nombre d'instructions exécutées, sur le nombre moyen de cycles par instruction et sur le temps du cycle d'horloge.

$$TempsExécution = \frac{NombreInstruction \times CyclesParInstruction}{FréquenceHorloge}$$

Equation 5: Calcul du temps d'exécution à partir du temps moyen par instruction.

Cette formulation du temps d'exécution ne prend pas en considération les caractéristiques détaillées du jeu d'instruction du processeur, ni les caractéristiques indéterministes d'exécution, comme le pipeline d'instructions et la hiérarchie de mémoire. Un modèle d'estimation plus complet sera présenté dans la section 5.6.3.

Le calcul de performance se base sur trois aspects :

- la **fréquence d'horloge** est résultat de la technologie matérielle et de la structure du circuit ;
- le **nombre de cycles par instruction** est résultat de l'architecture du jeu d'instruction et de l'architecture du circuit ;
- le **nombre d'instructions exécutées** est résultat de l'architecture du jeu d'instruction et de la technologie des compilateurs.

Les architectures et les compilateurs modernes utilisent des techniques permettant accroître les performances de ces trois éléments, nécessitant l'utilisation de techniques d'estimation chaque fois plus complexes. Une façon simple d'améliorer le résultat de l'estimation est de baser le calcul sur le temps moyen d'exécution de chaque instruction ou groupe d'instructions séparément. Cette méthode de calcul donne de meilleurs résultats pour les spécifications qui ont une tendance à utiliser un sous-ensemble de l'ensemble des instructions du processeur, par exemple, des applications réalisant des calculs extensifs. L'Equation 6 présente le calcul basé sur le temps moyen d'exécution de chaque instruction séparément.

$$TempsExécution = \sum_{i=1}^{n} \left(CyclesParInstruction_i \times NombreInstructions_i \times TempsCycle \right)$$

Equation 6: Calcul du temps d'exécution basé sur le temps moyen par instruction.

5.6 Le modèle d'estimation Cosmos

Le système Cosmos permet une conversion facile d'une spécification du niveau système en une architecture mixte logicielle/matérielle, comme illustre la Figure 53 (à gauche). Les principales étapes de cette méthodologie sont le découpage et la synthèse de la communication. L'ajout de techniques d'estimation dans ce flot a nécessité la création de nouvelles étapes de conception, comme illustre la Figure 53 (à droite). Ces nouvelles étapes ne réalisent pas des raffinements de la spécification, comme les étapes précédentes, mais aident le concepteur à optimiser la réalisation. Le calcul dynamique de la fréquence d'exécution, le choix de l'architecture et les estimations du côté logiciel et matériel ont pour objectif d'améliorer l'exploration de l'espace des solutions.

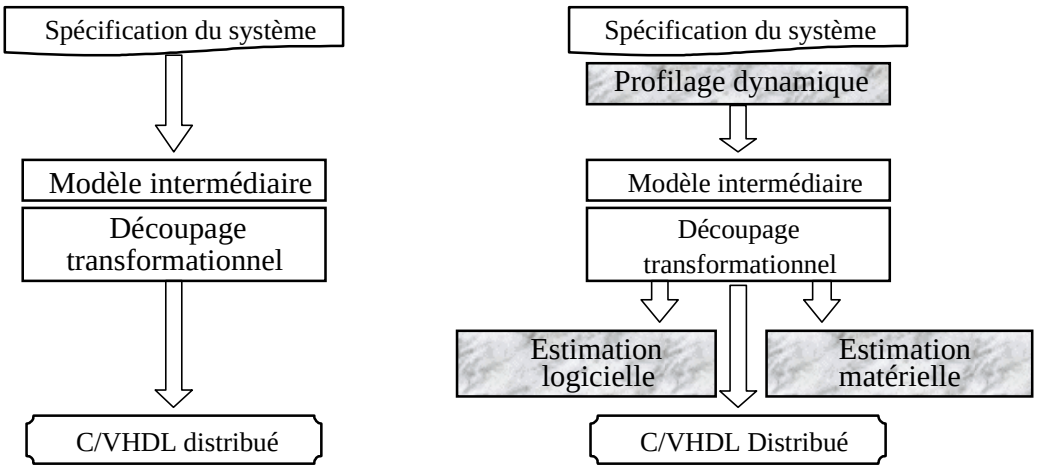


Figure 53: Etapes de la conception avec Cosmos.

Ces nouvelles étapes sont:

1. Le **calcul dynamique de la fréquence d'exécution** réalise l'analyse de la spécification d'entrée. L'analyse détecte les 'points chauds' de la spécification, les blocs d'instruction les plus fréquemment exécutés. L'utilisateur peut concentrer son attention sur ces points chauds pendant le découpage et le raffinement de la spécification;
2. L'**estimation du logiciel** utilise les informations du calcul de la fréquence d'exécution et du découpage pour calculer la performance de la réalisation. Des informations additionnelles, comme la taille du code et la taille des données, sont aussi générées pour valider l'allocation des mémoires;
3. L'**estimation du matériel** est basée sur la synthèse comportementale et l'estimation RTL. Cette technique est décrite par [Din96].

5.6.1 Le calcul dynamique de la fréquence d'exécution des instructions

L'objectif du calcul de la fréquence d'exécution des instructions (profiling) est de mesurer la fréquence d'exécution de chaque bloc d'instruction de la spécification. Cette mesure est importante pour le calcul du temps d'exécution de la spécification, puisque normalement les instructions de la spécification ne s'exécutent pas de façon homogène.

Normalement, les spécifications obéissent à la propriété de localité de référence [Hen96]. La localité de référence signifie que les programmes ont tendance à réutiliser les données et les instructions qu'ils ont utilisées récemment. Les programmes passent une grande partie du temps total d'exécution sur une petite partie du code. Il est donc important de déterminer les parties du programme les plus couramment exécutées. Ces 'points chauds' sont de bons candidats pour une réalisation matérielle, quand le temps d'exécution impose des restrictions.

Nous distinguons deux types de calcul de la fréquence d'exécution :

- le **calcul statique** analyse statiquement le code source pour prédire le flot de contrôle de la spécification;
- le **calcul dynamique** utilise l'exécution ou la simulation de la spécification pour la saisie des informations relatives à la fréquence d'exécution de chaque bloc.

L'approche de calcul statique de la fréquence d'exécution est utilisée par la plupart des systèmes de conception conjointe existants [San96, Gon94]. L'analyse statique d'une

spécification ne permet pas d'estimer la fréquence d'exécution dynamiques des boucles et des différentes branches des instructions conditionnelles. Cette approche impose les restrictions suivantes (voir Figure 54):

- le nombre de fois que le corps d'une boucle est exécuté doit être constant ou défini manuellement par l'utilisateur à partir de l'analyse du code;
- la probabilité d'exécution des branches d'une instruction conditionnelle doit être prédéfinie ou définie manuellement par l'utilisateur.

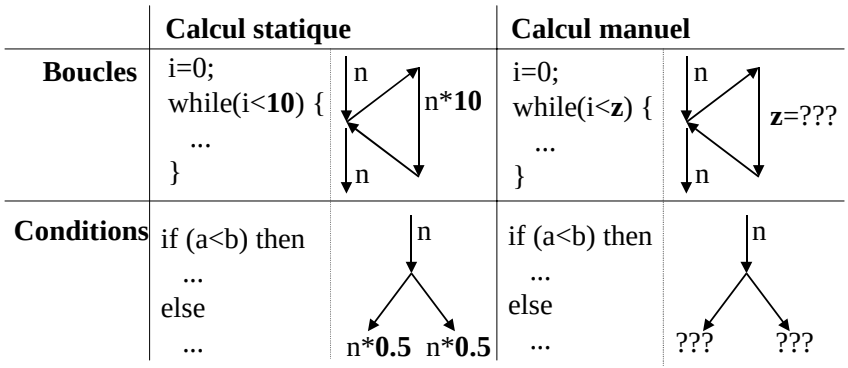


Figure 54: Traitement statique des boucles et des conditions.

Ces restrictions ne sont pas pénalisantes pour les applications de type flots de données. Par contre, pour les applications de contrôle le profilage statique s'avère insuffisant.

Le calcul dynamique de la fréquence d'exécution surmonte facilement les restrictions de l'approche statique, car il se base sur l'exécution ou sur la simulation de la spécification avec une sauvegarde des informations liées au flot d'exécution. Dans ce cas, plus le vecteur de test d'entrée est expressif, plus les annotations relatives aux fréquences et aux probabilités d'exécution seront représentatives.

L'algorithme de calcul de la fréquence d'exécution utilisé par Cosmos est basé sur le calcul dynamique de la fréquence d'exécution. Cette approche est divisée en trois étapes :

1. L'analyse de la spécification et l'**addition des indicateurs de trace** aux endroits critiques de la spécification (voir section 5.6.1.2);
2. La **simulation dynamique** de la spécification initiale, où les indicateurs de trace sont alimentés et le fichier de trace généré ;
3. L'**annotation de la spécification** avec les informations obtenues à partir du fichier de trace.

Cette technique de calcul dynamique de la fréquence d'exécution utilise trois outils, une pour chaque étape:

- l'outil *PuTF* (*Put Trace Flags*) ajoute les indicateurs de trace à la spécification;
- le simulateur au niveau système (*GeodeSim*);
- l'outil *GeTV* (*Get Trace Values*) qui réalise la lecture des valeurs de trace et l'annotation de la spécification.

La syntaxe d'activation de ces outils est présentée dans l'appendice B. Cette approche de calcul de la fréquence d'exécution est rapide, puisqu'elle se base sur la simulation au niveau fonctionnel.

5.6.1.1 Les outils de calcul dynamique de la fréquence d'exécution

L'utilisateur dispose de deux outils internes à l'environnement Cosmos qui, en conjonction avec le simulateur SDL, réalisent l'approche de calcul dynamique de la fréquence d'exécution présentée dans cette thèse. Le calcul de la fréquence d'exécution est nécessaire, dans le flot de conception, après l'entrée de la spécification initiale ou après des changements de la spécification.

La Figure 55 montre le flot d'activation des outils de calcul de la fréquence d'exécution. L'outil PuTF commence en ajoutant des indicateurs de trace à la spécification initiale, décrite en SDL. Les indicateurs sauvegardent le nombre de fois que le flot d'exécution traverse la partie du code associée à l'indicateur, pendant la simulation. L'outil GeodeSim [Obj97] est utilisé pour la simulation au niveau système de la spécification SDL. L'utilisateur doit utiliser des données d'entrée au simulateur qui soient représentatives de l'exécution du système. Plus la simulation est longue et représentative, meilleur sera le résultat obtenu par le calcul dynamique. Pendant la simulation, l'utilisateur doit s'assurer que tous les chemins d'exécution de la spécification sont parcourus. Le résultat de la simulation est la création d'un fichier de trace.

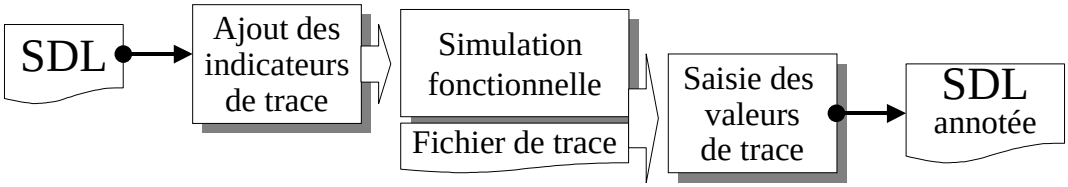


Figure 55: Manipulation des indicateurs de trace.

Il est apparu que la simulation fonctionnelle est rapide et facile à suivre. La simulation SDL est environ 15 fois plus rapide que la simulation VHDL et 30 fois plus rapide que la co-simulation C/VHDL. Le temps épargné par la simulation fonctionnelle et l'analyse des résultats peut être dédié au choix du vecteur de test d'entrée de la simulation.

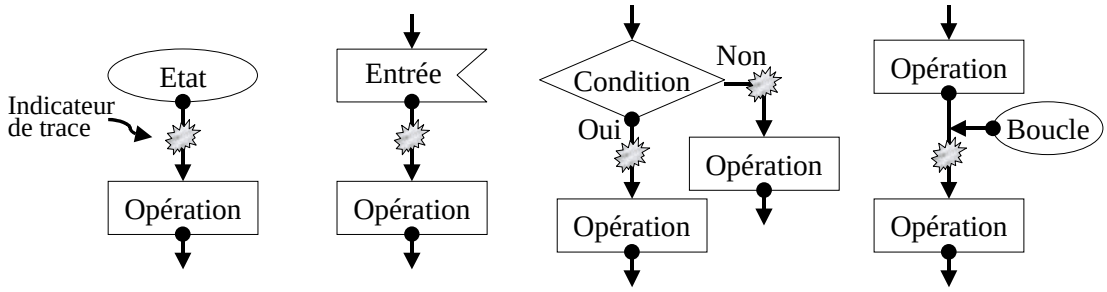
Après la simulation fonctionnelle, vient la récupération des valeurs de trace. Pour cette opération l'outil GeTV réalise la lecture du fichier de trace généré par le simulateur et calcule la fréquence d'exécution et la probabilité d'exécution de chaque bloc d'instruction.

5.6.1.2 Les indicateurs de trace

L'approche de calcul dynamique de la fréquence d'exécution est basée sur l'utilisation des indicateurs de trace. Un indicateur de trace est un compteur local ajouté à la spécification du système. Chaque bloc d'instruction contient un indicateur de trace qui est incrémenté chaque fois que le flot d'exécution active le bloc.

Les instructions du comportement du système sont divisées en deux types : les **instructions de contrôle** et les **instructions opératoires**. Les instructions de contrôle sont celles qui peuvent changer le flot d'exécution du programme. Des exemples d'instructions de contrôle sont les conditions, les boucles et les instructions de changement d'état. Ces instructions ont besoin d'un indicateur de trace pour détecter les changements possibles du flot d'exécution. La Figure 56 montre les positions, dans la représentation SDL, où les indicateurs de trace sont nécessaires.

Les instructions opératoires sont groupées en blocs de base. Un bloc de base est un ensemble d'instructions qui a un point d'entrée et un point de sortie. Un bloc de base est considéré comme une seule entité par l'algorithme de calcul de fréquence d'exécution.



(a) Activation d'un état (b) Entrée de données (c) Conditions (d) Boucles

Figure 56 : Localisation des indicateurs de trace dans SDL.

La Figure 57 montre une partie d'une spécification en SDL qui a subi l'addition des indicateurs de trace, avec l'utilisation de l'outil *PuTF*. De façon plus détaillée, un indicateur de trace est une fonction, activée par le simulateur fonctionnel, capable d'enregistrer le contexte actuel du flot d'exécution. La syntaxe de l'indicateur de trace est la suivante :

```
CALL writeln('-> <module> <ligne>') ;
```

`Writeln` correspond à la fonction qui enregistre ces paramètres dans le fichier de trace ; `<module>` indique le module SDL en question, puisqu'une spécification peut être partagée en plusieurs modules (ou fichiers) ; `<ligne>` est le numéro de la ligne où l'indicateur se trouve, dans le module SDL.

```
SYSTEM moteur;
...
 BLOCK PID;
...
  START ;                          CALL writeln('=> 1 175');
     OUTPUT pid_pc_values;                 } bloc 1
     TASK ind1 := 0;
     NEXTSTATE moteur;
  STATE moteur;
     INPUT pc_pid_value(temp);     CALL writeln('=> 1 193');
     TASK ind1 := ind1+1;                  } bloc 2
     TASK memoire(ind1):=temp;
     DECISION ind1;
     ( 7 ):                        CALL writeln('=> 1 206');
      OUTPUT pid_pc_values;                } bloc 3
      NEXTSTATE -;
     ( <14 ):                      CALL writeln('=> 1 215');
      NEXTSTATE -;                         } bloc 4
     ( 14 ):                       CALL writeln('=> 1 221');
      OUTPUT pid_moteur1_data(0.0);
      OUTPUT pid_moteur2_data(0.0);
      OUTPUT pid_moteur1_read;             } bloc ...
      NEXTSTATE calcul1;
     ENDDECISION;
  ENDSTATE;
  STATE calcul1;                                          Indicateurs
     INPUT moteur1_pid(u2);        CALL writeln('=> 1 245');  de trace
     TASK u1:=memoire(1),
     u3:=memoire(2),
     u4:=memoire(3),
     u5:=memoire(4);
     TASK u6:=memoire(5),
     u7:=memoire(6),
     u8:=memoire(7);
     TASK u9:=u1-u2,
     u10:=u9*u6,
     u11:=u4*u7,                          } bloc n-1
     u12:=u3-u11,
     u13:=u5*u8;
     TASK u14:=u10+u12+u13;
     OUTPUT pid_moteur1_data(u14);
     TASK memoire(4):=u4,
     memoire(3):=u9,
     memoire(2):=u14;
     OUTPUT pid_moteur2_read;
     NEXTSTATE calcul2;
  ENDSTATE;
  STATE calcul2;
     INPUT moteur2_pid(u2);        CALL writeln('=> 1 292');
     TASK u1:=memoire(8),
     u3:=memoire(9),
     u4:=memoire(10),
     u5:=memoire(11);
     TASK u6:=memoire(12),
     u7:=memoire(13),
     u8:=memoire(14);
     TASK u9:=u1-u2,
     u10:=u9*u6,
     u11:=u4*u7,                          } bloc n
     u12:=u3-u11,
     u13:=u5*u8;
     TASK u14:=u10+u12+u13;
     OUTPUT pid_moteur2_data(u14);
     TASK memoire(11):=u4,
     memoire(10):=u9,
     memoire(9):=u14;
     OUTPUT pid_moteur1_read;
     NEXTSTATE calcul1;
  ENDSTATE;
 ENDBLOCK;
...
ENDSYSTEM;
```

Figure 57: Spécification SDL et indicateurs de trace.

La Figure 58 montre le résultat du calcul de la fréquence d'exécution sur un exemple de spécification. Sont représentés un état SDL et son graphe de flot de contrôle avec des annotations. Dans la représentation SDL, à gauche, nous pouvons identifier les instructions séquentielles et les instructions de flot de contrôle. Le graphe de flot de contrôle, à droite, contient des arcs qui représentent les blocs d'instructions séquentielles. Chaque arc contient des informations sur la fréquence d'exécution du bloc. Les noeuds représentent les instructions de contrôle du flot d'exécution.

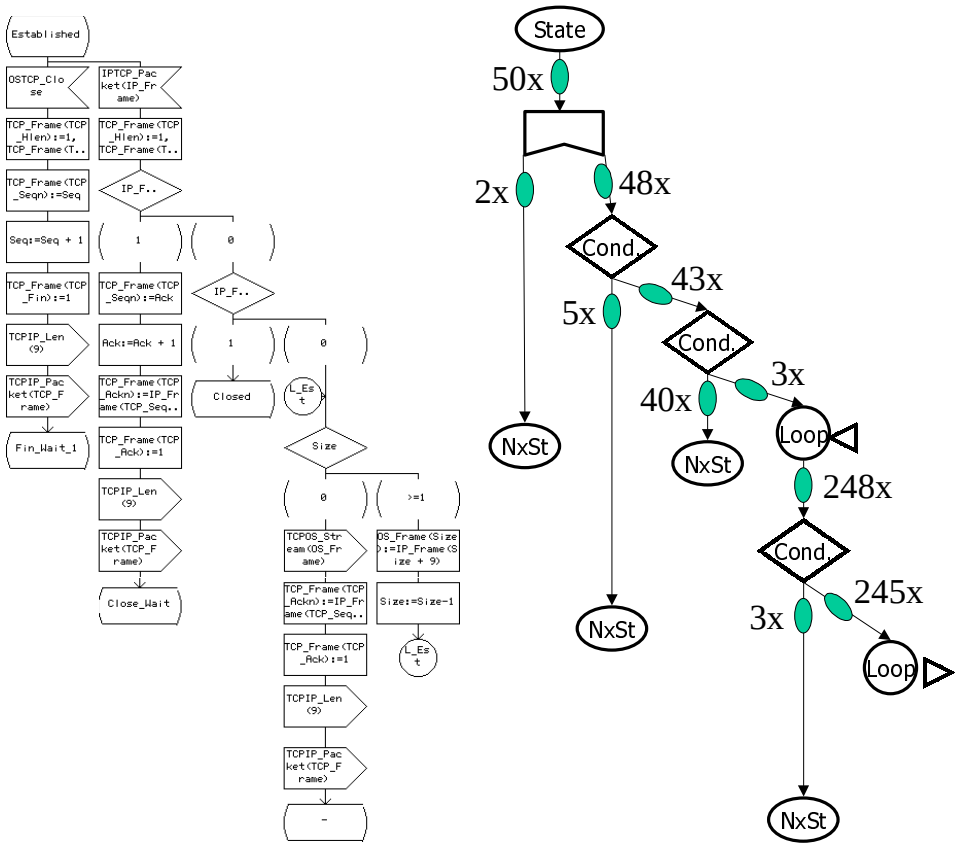


Figure 58 : Représentation d'un état SDL et son GFC annoté.

Les annotations relatives aux valeurs de trace sont ajoutées au fichier source SDL sous la forme de commentaires. Les commentaires ajoutés aux lignes ont la syntaxe suivante : `COMMENT #frequency 999` ou `COMMENT #probability 99%`. La spécification Solar générée, à partir de la spécification SDL annotée, contient les fréquences et les probabilités modélées par la commande *property*: `(PROPERTY frequency 999)` ou `(PROPERTY probality 99)`.

5.6.2 Les bibliothèques de caractérisation

Les bibliothèques caractérisent les composants utilisées dans l'architecture. Chaque composant a ses caractéristiques, restrictions et comportement décrits dans une les bibliothèques suivantes: la bibliothèque des canaux de communication, la bibliothèque des processeurs, la bibliothèque des réalisations matérielles et la bibliothèque des mémoires. L'appendice C présente une pseudo syntaxe des bibliothèques.

Par exemple, un canal de communication est caractérisé par son interface, les méthodes d'accès et le protocole. Un processeur logiciel est caractérisé par la fréquence d'horloge, le nombre de registres, le nombre d'étages du pipeline, la taille et le temps d'exécution de chaque instruction. Un ASIC est caractérisé par les contraintes de réalisation et par les caractéristiques des cellules de la partie opératrice. Une mémoire est caractérisée par la capacité en octets et par le temps d'accès.

5.6.3 L'estimation du logiciel

L'estimation du temps d'exécution des modules logiciels sur des processeurs programmables a pour objectif de calculer des informations sur le temps d'exécution de chaque module, la taille du code et la taille des données.

Pour l'estimation du nombre de cycles d'horloge nécessaires à l'exécution d'une instruction, sur un processeur cible, nous avons choisi de distinguer le calcul entre le temps pris par les processeurs et le temps pris par les systèmes de mémoire. Cette séparation est utile car les méthodes d'évaluation de ces composants sont différentes. Le temps pris par une mémoire est basée sur une valeur pré-calculé, qui reflète le temps moyen d'accès. Le calcul de cette valeur doit prendre en considération les paramètres de la mémoire, l'utilisation d'une mémoire cache et le domaine de l'application.

Le temps d'exécution lié au processeur dépend des ses caractéristiques et des caractéristiques de la réalisation. Cette dernière peut être calculée à partir des annotations résultant de l'étape de calcul de la fréquence d'exécution. La Figure 59 illustre le flot d'estimation des modules logiciels. La première étape est la conversion des modules destinés au logiciel dans un code générique, plus proche de l'architecture des processeurs. Cette conversion est considérée comme une compilation très simplifiée. La seconde étape est le choix du processeur. Lors de cette étape, la bibliothèque des processeurs est utilisé. La dernière étape est l'application du modèle de calcul d'estimation et la mise en forme des résultats pour l'utilisateur. Ces étapes sont détaillées dans les sections qui suivent.

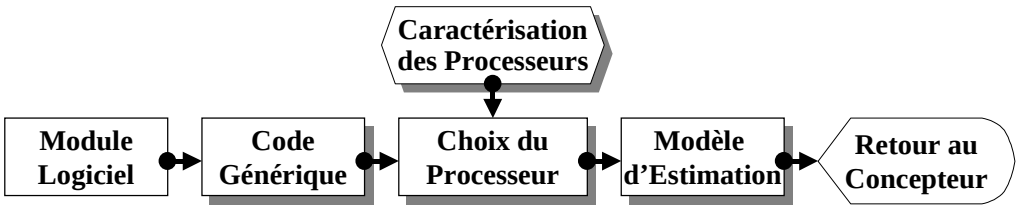


Figure 59: Flot d'estimation des modules logiciels.

5.6.3.1 La compilation du code logiciel

La compilation, une simple traduction de la spécification logicielle, est indispensable pour que l'algorithme d'estimation puisse faire la correspondance entre le comportement du module logiciel avec l'architecture du processeur cible.

Nous distinguons deux approches de compilation de la spécification [Gon95]:

- L'utilisation des **compilateurs dédiés**, qui compilent le code vers le jeu d'instruction d'un processeur existant;
- L'utilisation d'un **compilateur générique,** qui réalise la traduction du code vers un format générique pour un processeur virtuel.

Si l'on dispose d'un compilateur capable de générer du code pour le processeur en question, la première option est possible. Cette solution donne des résultats plus précis, une fois que les optimisations de compilation sont prises en compte.

Si l'on ne dispose pas d'un compilateur dédié au jeu d'instruction de chaque processeur cible, l'utilisation d'un compilateur générique peut être adoptée. Le code généré par le compilateur générique ne contient pas d'optimisations, comme les optimisations globales et locales, les optimisations d'allocation des registres, l'élimination des sous-expressions communes et les optimisations dépendantes du processeur. Mais l'utilisateur peut utiliser des stratégies pour réduire la différence de l'effet des

optimisations sur le résultat final. Une solution est d'avoir un facteur d'optimisation appartenant à chaque processeur cible. Dans ce cas, après l'obtention du résultat de l'estimation, le facteur d'optimisation est utilisé pour approcher ce résultat de la valeur réelle.

La Figure 60 présente ces deux modèles d'estimation: le **modèle à base de processeurs spécifiques** et le **modèle à base de processeurs génériques** [Gaj94b]. Avec le modèle à base de processeurs spécifiques, à gauche de la figure, l'estimation débute par la compilation des modules logiciels en code assembleur du processeur cible. Avec le code, la performance peut être obtenue par l'exécution ou par l'estimation statique. La taille du code et des données peut facilement être obtenue.

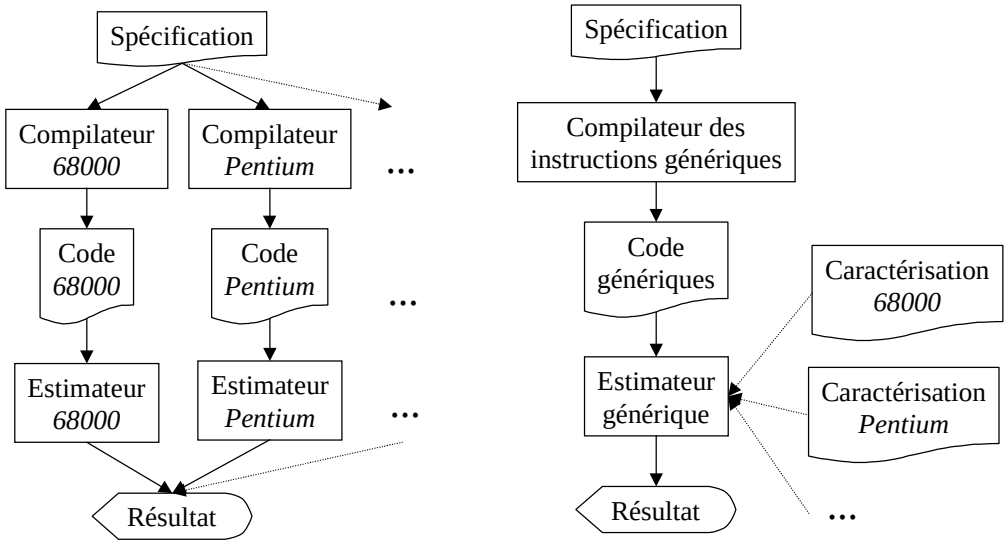


Figure 60: Estimation spécifique et estimation générique.

Le modèle à base de processeurs génériques, à droite de la figure, utilise un compilateur unique qui génère du code générique (voir section 5.6.3.2). L'estimateur calcule les valeurs de performance basé sur le code générique et sur un fichier de technologie contenant les caractéristiques des processeurs cibles. Ces fichiers de technologies contiennent, entre autres, des informations sur le nombre de cycles d'horloge nécessaires à l'exécution de chaque instruction générique.

Le fait d'isoler le compilateur de l'architecture du processeur cible a plusieurs avantages et inconvénients. Le principal inconvénient est que normalement il existe une étroite relation entre le jeu d'instructions du processeur et l'architecture du compilateur. Nous pouvons dire que le jeu d'instructions des processeurs modernes doit prendre en considération les caractéristiques des compilateurs. Cette relation fait que l'estimation fondée sur le modèle à base de processeurs spécifiques est plus précise. L'avantage du modèle à base de processeur générique est que l'utilisation d'un compilateur unique, qui modélise les principales caractéristiques des compilateurs existants, permet la mise en pratique d'une approche d'estimation flexible. L'adaptation de l'estimateur à un nouveau processeur est directe. Cette adaptation consiste simplement en la création du fichier de technologie pour le nouveau processeur. Le modèle générique permet aussi l'estimation de nouveaux processeurs avant que les compilateurs ne soient disponibles. Finalement, le modèle générique est portable, il peut être exécuté sur n'importe quelle machine, et non seulement sur celle qui contient le compilateur cible.

Le compilateur générique de Cosmos produit un ensemble d'instructions de plus bas niveau similaires à la sortie d'un compilateur traditionnel. La Figure 61 représente une

partie du code Solar, appartenant à une instruction conditionnelle, et le code générique correspondant.

```
(IF (> x y)
  (THEN
       (ASSIGN x 0)
  )
  (ELSE
       (ASSIGN x (+ x y ) )
  )
)
```

```
(1)          MOV DIS INT
(2)          MOV DIS INT
(3)          BEQZ
(4)          MOV IMM INT
(5)          JMP
(6)else:     ADD INT
(7)          MOV DIS INT
(8)end:
```

Figure 61: Traduction Solar vers le format générique.

5.6.3.2 Le code générique

Le code générique est une représentation de bas niveau, plus proche des langages des processeurs. L'objectif de l'utilisation d'un code générique est de permettre la mise en correspondance de la spécification initiale avec l'architecture du processeur cible, comme illustre la Figure 62. La spécification initiale est traduite dans un modèle générique de bas niveau. Ces instructions génériques sont mise en correspondance avec les données contenues dans le fichier de caractérisation du processeur cible.

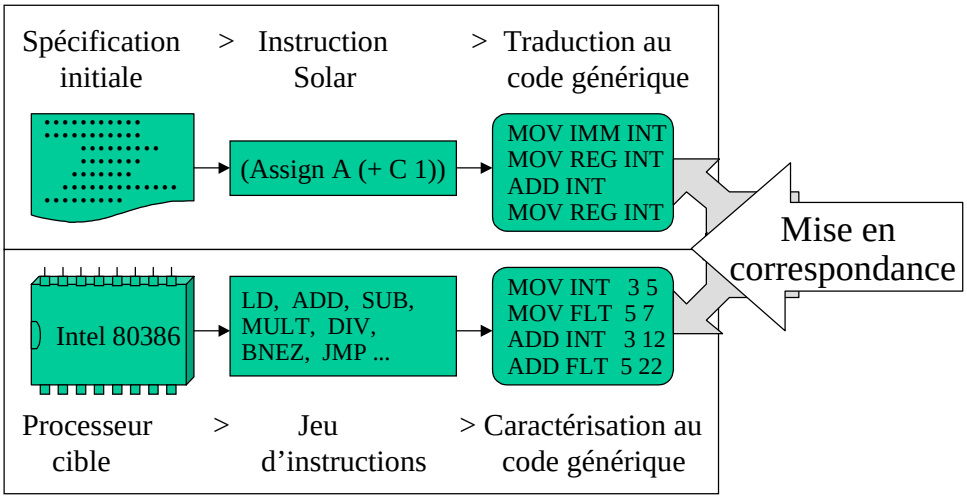


Figure 62: Recoupement entre la spécification initiale et le processeur cible.

Le code générique doit avoir une architecture qui puisse facilement être adaptée au code des processeurs existants [Vah95]. Le jeu d'instructions génériques de Cosmos est réduit, il contient neuf instructions, quatre modes d'adressage et trois types de données. Le Tableau 7 décrit ce jeu d'instructions.

Code-op	**Cas d'utilisation**
	Transferts de données
MOV	Chargement/rangement d'une registre donnée
	Arithmétiques et logiques
ADD/SUB	Addition/soustraction
MULT	Multiplication
DIV	Division
AND/OR	Et/ou
	Contrôle
JMP	Saut
BEQZ	Branchement conditionnel

JAL	Saut et lien avec sauvegarde du CP
RFE	Retour d'exception vers le programme

Tableau 7: Instructions génériques de Cosmos.

Les modes d'adressage, acceptés par les instructions de transferts des données, les instructions arithmétiques et les logiques sont présentés dans le Tableau 8.

Code-op	**Mode d'adressage**	**Représentation**	**Cas d'utilisation**
REG	Registre	Reg[Rx]	La valeur est dans le registre Rx
IMM	Immédiat	#val	Utile pour les constantes
DIS	Déplacement	Mem[posic+Rx]	Accès aux variables
IND	Indexé	Mem[Reg[Rx]+Reg[Ry]]	Utile pour l'adressage des tableaux

Tableau 8: Modes d'adressage des instructions génériques.

Les types de données du modèle générique sont décrits dans le Tableau 9.

Code-op	**Type de données**
INT	Des entiers
FLO	Des flottants
BIN	Des binaires

Tableau 9: Type de données des instructions génériques.

La Figure 63 présente une pseudo syntaxe du code générique de Cosmos. Dans cette représentation, les symboles terminaux sont représentés en caractères gras. Ces symboles sont responsables de la formation du code. Les symboles non-terminaux sont représentés en caractères normaux. Ces symboles spécifient le langage et la structure hiérarchique de la représentation. Les axiomes contiennent les caractères '<' et '>'. Dans cette représentation, quand une règle de production termine avec une symbole ↻, cela signifie une répétition possible de la règle.

```
Program      : (GENERICPROGRAM <name> Code )

Code         : (CODE <name> Data Procedure Instruction ) ↻

Data         : (VARIABLE Datatype ) ↻
             | (CONSTANT Datatype ) ↻

Procedure    : (PROCEDURE <name> Data Instruction ) ↻

Instruction :  (MOV Addressmode Addressmode ) ↻
             | (ADD/SUB Datatype Addressmode1 Addressmode2 ) ↻
             | (MULT Datatype Addressmode Addressmode ) ↻
             | (DIV Datatype Addressmode Addressmode) ↻
             | (AND/OR Addressmode ) ↻
             | (JMP ) ↻
             | (BEQZ Addressmode ) ↻
             | (JAL ) ↻
             | (RFE ) ↻
             | (SOURCELINE <value> ) ↻

Addressmode  : REG
             | IMM
             | DIS
             | IND
```

```
Datatype      : INT
              | FLO
              | BIN

<name>        : 0-9 a-z A-Z _ ↻
<value>       : 0-9 ↻
```

Figure 63: Pseudo syntaxe du code générique de Cosmos.

5.6.3.3 La caractérisation des processeurs

La Figure 64 contient la description d'un processeur, l'*Intel 8086*, dans le format de la bibliothèque Cosmos. La syntaxe du fichier de caractérisation est présentée dans l'appendice C. Le processeur représenté est simple, mais des processeurs modernes peuvent être caractérisés de la même façon.

```
(PROCESSORLIBRARY     Default

(PROCESSOR                    Intel_8086
  (CLOCKFREQUENCY             8)
  (BUSDATAWIDTH               16)
  (BUSADRESSWIDTH             16)
  (REGISTERS                  8)
  (PINS               40)
  (PIPELINESTAGE              4)
  (OPTIMIZATIONRATE           0.05)

  (INSTRUCTIONS       ; Instr_size   Exec_time(integer) Exec_time(float)
    (MOV              2              2                     5)
    (ADD/SUB          3              3                     7)
    (MULT             6              13                    24)
    (DIV              8              15                    40)
    (AND/OR           2              2                     2)
    (JMP              3              3)
    (BEQZ             4              4)
    (JAL              7              4)
    (RFE              7              2) )

  (ADDRESSINGMODES    ; Instr_size Exec_time
    (REG              0              0)
    (IMM       2              2)
    (DIS       3              3)
    (IND              5              4) )

  (DATATYPES          ; Instr_size
    (INT              16)
    (FLO              32)
    (BOO              1) ) )
)                             ;End PROCESSOR_LIBRARY Default
```

Figure 64: Fichier de caractérisation du processeur *Intel 8086*.

La définition du nombre de registres du processeur est importante pour le calcul du temps d'exécution. Par exemple, si un processeur contient un grand nombre de registres, le calcul du temps d'exécution d'une opération algébrique peut prendre en considération la disponibilité des registres pour sauvegarder les valeurs temporaires résultant des étapes de l'opération. Dans le cas d'un processeur qui contient un nombre réduit de registres, les valeurs temporaires doivent être sauvegardées dans la mémoire, avec l'addition d'un temps supplémentaire à l'exécution.

Par exemple, le temps d'exécution de l'expression $(a+b)*(c+d)$ dépend, entre autres, du nombre de registres disponibles pour garder les résultats partiels ($a+b$ et $c+d$). Si le processeur dispose d'un nombre suffisant de registres, le traducteur Solar-code générique et l'estimateur doivent prendre en compte le fait que la multiplication est effectuée sans l'accès supplémentaire à la mémoire.

La valeur `OPTIMIZATIONRATE` représente le facteur d'optimisation du compilateur. Cette valeur d'approximation est utilisée car il est très difficile de modéliser les techniques avancées de compilation et d'optimisation. La valeur du facteur d'optimisation, pour chaque processeur reflète les techniques d'allocation de registres, l'optimisation des boucles et les optimisations globales. Cette valeur est normalement calculée à partir des programmes exemples (benchmarks). Dans ce cas, la comparaison est faite entre la performance du code optimisé retourné par le compilateur et code non optimisé. Des mesures relatives au degré d'optimisation pour des processeurs simples sont disponibles dans la littérature [Gon94].

5.6.3.4 La modélisation du pipeline d'instruction

Cette section présente la technique de modélisation du pipeline d'instruction utilisée dans Cosmos. Le pipeline est une technique de réalisation dans laquelle plusieurs instructions se retrouvent, dans un temps donné, en cours d'exécution. Le pipeline permet d'accroître le nombre d'instructions que le processeur exécute par unité de temps.

La structure du pipeline est composée d'un nombre fixe d'étages, aussi appelé la profondeur du pipeline. Chaque étage exécute une partie de l'instruction. Le modèle d'estimation Cosmos suppose que le processeur réalise un pipeline idéal, où les étages du pipeline sont équilibrés. De cette façon, nous supposons que le temps d'exécution d'un bloc de base est obtenu par le temps d'exécution du bloc sans pipeline divisé par le nombre d'étages du pipeline, comme montre l'Equation 7.

$$CyclesBlocPipeliné = \frac{CyclesBloc}{EtagesPipeline} + PénalitéBranchement$$

Equation 7: Modélisation du pipeline des instructions appartenant à un bloc.

Comme il a été dit dans la section 5.6.1, un bloc de base est une séquence d'instructions linéaire, sans branchement, à l'exception du début et de la fin. Notre modèle suppose que le pipeline est toujours plein pendant l'exécution du bloc. Nous pouvons voir que l'accélération d'exécution d'un bloc est directement proportionnelle au nombre d'étages du pipeline exécutés par le processeur. Le nombre d'étages du pipeline pour chaque processeur est obtenu directement de la bibliothèque de caractérisation des processeurs. Dans l'équation, la *pénalité de branchement* est liée à la pénalité de suspension du pipeline occasionnée par le changement du flot de contrôle existant à la fin de chaque bloc.

Ce modèle est simple et il ne prend pas en considération la **dépendance des données** dans les blocs des instructions (aléas de données), ni le **conflit des ressources matérielles** (aléas structurel), mais il modélise partiellement le changement du **flot de contrôle** existant entre les blocs (aléas de contrôle). La modélisation complète des aléas de contrôle doit prendre en considération les aspects d'optimisation de branchement et de prédiction dynamique. Les aléas peuvent générer une suspension du pipeline, comme arrive avec la mémoire cache lorsque les données ne sont pas disponibles. Une dépendance des données signifie qu'une instruction dépend du résultat de l'instruction précédente, qui peut être encore en cours d'exécution.

5.6.3.5 La modélisation de la hiérarchie de mémoire

Les processeurs modernes utilisent la hiérarchie de mémoire (cache) pour accélérer l'exécution des programmes. Pour chaque accès à la mémoire, en lecture ou en

écriture, si l'adresse demandée est présent dans le cache (succès d'accès au cache) la donnée est disponible à une grande vitesse. Le temps d'accès à la mémoire cache est normalement dix fois plus rapide que le temps d'accès à des mémoires traditionnelles et peut atteindre deux fois le temps d'accès d'un registre interne d'un processeur [Gee93]. Quand l'élément demandé ne se trouve pas dans le cache (échec d'accès au cache), un bloc de données de taille fixe, contenant le mot demandé est lu dans la mémoire principale et placé dans le cache. Mais cette opération consomme du temps, et l'exécution du processeur est suspendue jusqu'à ce que la donnée soit disponible.

L'approche d'estimation de Cosmos modélise la hiérarchie de mémoire pour calculer le temps que le processeur utilise pour accéder aux données et aux instructions contenues dans la mémoire pendant l'exécution du code. Le temps d'attente du processeur pour accéder à la donnée ou à l'instruction est calculé par l'Equation 8.

$$TempsAccès = TempsRéussi + (TauxEchec \times Pénalité)$$

Equation 8: Modélisation des l'accès à la mémoire avec cache.

Dans cette équation, *TempsRéussi* est le temps nécessaire pour accéder à une donnée déjà présente dans le cache. Cette valeur est fixe et fait partie de la caractérisation de la mémoire, dans la bibliothèque. Le *TauxEchec* est une valeur prédéfinie qui représente la probabilité de ne pas trouver l'information désirée dans le cache. La valeur *Pénalité* représente le temps nécessaire pour rafraîchir le cache (défaut de cache). *Pénalité* est une valeur pré-calculée obtenue à partir de facteurs comme la taille du cache, le temps d'accès à la mémoire principale et le temps de cycle de la mémoire. Le temps de cycle de la mémoire est le temps minimum entre deux accès. Le temps d'accès obtenu par l'Equation 8 représente le temps moyen entre la demande de la donnée et sa disponibilité.

La valeur liée au taux d'échec peut être calculée par avance pour l'accès aux données et aux instructions. Dans notre modèle, la valeur *TauxEchec* pour les données et les instructions est obtenue à partir d'une table, décrite dans la bibliothèque de mémoire du système. Cette table lie la taille du cache aux valeurs *TauxEchec* approximatives. Ces valeurs, dans la version actuelle de Cosmos, ont été obtenues à partir de [Smi93] et sont basées sur la taille et la vitesse de l'accès à la mémoire.

Une étude plus approfondie est nécessaire pour que le calcul du *TauxEchec* prenne en considération d'autres aspects comme la taille du code ou le domaine d'application, la structure hiérarchique du cache et son organisation (correspondance directe, pleinement associatif, associatif par bloc). La valeur *TauxEchec* moyenne pour l'accès aux données et aux instructions est respectivement 0,08 et 0,007. Ces valeurs signifient que en moyenne, pour des applications variées, 8% des accès aux données et seulement 0,7% des accès aux instructions génèrent un défaut de cache. Cette différence est due au principe de localité (voir section 5.4).

5.6.3.6 La performance du logiciel

Une bonne mesure de performance d'un système est une tâche difficile, car elle dépend de la modélisation détaillé du processeur cible et de la connaissance du flot d'exécution des instructions. Pour des processeurs très simples il est possible de calculer la performance de la réalisation à partir d'une table de valeurs qui contient le temps d'exécution de chaque instruction sur le processeur cible. Le temps d'exécution total du processus est calculé par la multiplication de la fréquence d'exécution de chaque instruction par le temps d'exécution de l'instruction.

L'approche de calcul de performance du logiciel, développé dans Cosmos, utilise cette solution simple, enrichie par la modélisation des techniques d'accélération d'exécution des programmes. Cette modélisation est nécessaire car les processeurs modernes exécutent des instructions avec un nombre de cycles varié, qui dépend de l'état du processeur au moment où l'instruction est exécutée.

Certaines instructions peuvent avoir un temps d'exécution variable et donc difficile à modéliser. Par exemple, le temps d'exécution de l'instruction de saut conditionnel (*code-op BEQZ* dans le code générique) dépend du résultat de la condition.

5.6.3.7 La taille de la mémoire

L'algorithme d'estimation de performance est capable de calculer la taille minimal de la mémoire nécessaire à une réalisation. Cette valeur est liée à la taille du code qui réalise le comportement et la taille des données, comme les variables et les constantes. L'Equation 9 présente le modèle de calcul de la taille de la mémoire, mesurée en nombre d'octets. Cette information est utile pendant l'allocation des mémoires.

$$TailleMem(P) = \sum_{lli_i \in P} \left(TInstr_{lli_i} + TAdresse_{lli_i} \right) + \sum_{d_i \in P} TailleDonnée_{d_i}$$

Equation 9: Modèle de calcul de la taille de la mémoire.

Pour calculer la taille de chaque d'instruction générique nous utilisons deux composantes: la taille du code de l'instruction (*TInstr*) et la taille associée au mode d'adressage de l'instruction (*TAddress*). Ces informations sont contenues dans la bibliothèque de processeurs. Le calcul de la taille des données est basé sur la taille des variables et des constantes du comportement. Ce modèle assume que les variables et les constantes ont un temps d'existence égal au temps d'exécution du processus. La taille de chaque type de donnée est aussi spécifiée dans la bibliothèque de processeurs.

5.6.4 L'estimation du matériel

Cette section présente l'approche d'estimation de la partie matérielle de la réalisation. L'estimation du matériel dans la méthodologie Cosmos est basée sur la synthèse comportementale, suivie de l'estimation au niveau transfert des registres (RTL). La synthèse comportementale est nécessaire pour l'estimation car nous ne disposons pas d'informations suffisantes sur la réalisation, au niveau système, pour permettre une estimation représentative [Hen95]. Au niveau système il manque des informations primordiales sur l'ordonnancement des opérations et sur l'allocation des ressources matérielles. Le niveau transfert de registres permet une estimation assez significative, capable de valider la réalisation ou de guider l'utilisateur vers de nouvelles alternatives. L'étape de synthèse comportementale dans Cosmos utilise le compilateur d'architecture Amical [Kis96]. La Figure 65 illustre les étapes de conception liées à l'estimation du matériel. La synthèse comportementale utilise la bibliothèque de caractérisation technologique.

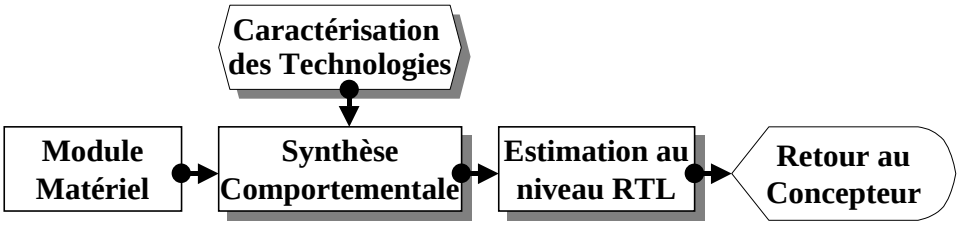


Figure 65: Flot d'estimation des modules matériels.

L'estimation, basée sur la synthèse comportementale de la partie matérielle ne provoque pas une augmentation exagérée du temps de conception. La première raison est que, dans la méthodologie Cosmos, l'estimation est utilisée pour valider une réalisation, contrairement aux approches automatiques où l'estimation est utilisée par l'algorithme de découpage. La deuxième raison est que, actuellement, l'utilisateur dispose d'un système intégré contenant les outils Cosmos et les outils Amical au sein d'un seul environnement de synthèse. Cet environnement est appelé Music (*MUltilaguage codeSIgn for system on Chip*). Dans cet environnement, les processus matériels peuvent être synthétisés automatiquement après le découpage et la vérification des caractéristiques de la réalisation peut être faite dans le même environnement de travail.

L'estimation du matériel avec Amical donne à l'utilisateur une information permanente sur l'évolution du processus de synthèse. Les résultats calculés par l'estimateur comprennent les mesures de surface, de vitesse et de consommation du circuit.

Ce modèle d'estimation utilise en entrée trois types de spécifications :

- Une **spécification comportementale,** générée automatiquement par Cosmos et décrite en Solar ou VHDL. Une flexibilité dans la spécification permet la description de circuits complexes, avec des instructions du type boucles, branchements, appels de fonctions et appels de procédures;
- Une **bibliothèque de fonctions externes** (*Functional units*), qui contient des unités fonctionnelles standards et des unités fonctionnelles privées définies par l'utilisateur. La bibliothèque décrit les caractéristiques de chaque unité fonctionnelle. Une unité fonctionnelle peut réaliser plusieurs opérations, comme des opérations standards (addition, multiplication, logiques) mais aussi des opérations complexes programmées par l'utilisateur (unité d'entrée/sortie, mémoire cache). Les unités fonctionnelles sont utilisées dans la spécification comportementale et offrent la possibilité d'utiliser des circuits réalisés préalablement;
- Un **fichier de technologie,** qui spécifie des contraintes, les tailles des composants, les paramètres de consommation, et les paramètres de délai maximal des composants.

5.6.5 L'estimation de la communication

Cette section présente l'approche d'estimation du temps lié à la communication dans un système. Le temps de communication, dans cette approche, ne prend pas en considération les temps d'attente liés à la synchronisation des processeurs. Nous supposerons que les processeurs et les données sont toujours disponibles pour la communication. C'est une grande simplification du modèle de communication.

La mesure du temps de communication permet à l'utilisateur de vérifier l'impact des décisions de raffinement sur la communication. Par exemple, l'utilisateur peut vérifier la relation entre le temps d'exécution d'un bloc du comportement et le temps pris par la

communication, ou il peut identifier des accès aux canaux de communication occasionnés par un découpage non optimisé du comportement d'un bloc.

Le taux des données échangées entre les processeurs peut être facilement calculée, à partir de la taille des données et la fréquence d'accès au canal. L'Equation 10 calcule la quantité d'informations échangées par un processus pendant son exécution en multipliant la taille de chaque message (*TailleMess*) par la fréquence d'exécution de l'instruction qui active la communication (*FreqExéc*).

$$Données(P) = \sum_{Ic_i \in P} \left(TailleMess(Ic_i) * FreqExéc(Ic_i) \right)$$

Equation 10: Modèle de calcul de la quantité des données échangées.

L'algorithme de calcul du temps de communication utilise le résultat du calcul dynamique de la fréquence d'exécution et les informations disponibles dans la description de chaque protocole. Les protocoles de communication de la bibliothèque contiennent les paramètres suivant:

- le **coût de réalisation** du protocole est lié au coût intrinsèque de réalisation (complexité, surface, capacité);
- le **temps de transmission** des données correspond au temps nécessaire à une transmission simple sur le canal;
- le **débit moyen** du canal correspond au taux de transmission des données sur le canal.

Equation 11 présente le calcul du temps de communication d'un processus, résultant de la somme des temps requis à l'envoi de chaque message.

$$TempsComm(P) = \sum_{I_i \in P} \left(Comm(I_i.Prot, I_i.Donnée) * FreqExéc(I_i) \right)$$

$$Comm(prot, donnée) = ProcEmiss + prot.TempsStab + \frac{donnée.Taille}{prot.Débit} + ProcRecep$$

Equation 11: Modèle du temps de communication.

Le calcul du temps de communication prend en considération plusieurs aspects:

- le **temps d'émission et de réception** (*ProcEmiss* et *ProcRecep*), qui fait référence au temps pris par la procédure d'émission pour injecter les données sur le réseau et le temps pris par la procédure de réception pour retirer les données du réseau;
- le **temps de transmission** (*prot.TempsStab*);
- le **temps du transfert**, qui est le temps nécessaire au message pour traverser le réseau. Cette valeur est égale à la division de la taille du message (*donnée.Taille*) par le débit du réseau d'interconnexion (*prot.Débit*). Le débit de transmission de chaque protocole est spécifié dans la bibliothèque de communication.

5.7 Résultats préliminaires

Cette section présente les résultats préliminaires obtenus à partir des outils d'estimation de Cosmos. Ces résultats débutent le processus de validation de l'approche d'estimation développé dans ce travail de thèse. Les résultats générés par les outils se

référents à la fréquence d'exécution d'une réalisation donnée et sont comparés avec des valeurs d'exécution observées pendant le profilage (voir section 6.7).

L'exemple traité, la conception d'une carte d'interconnexion au réseau ATM, est détaillée dans le chapitre 6. Les valeurs relatives au temps d'exécution de cet exemple, pour une architecture initiale composée de quatre processeurs logiciels, sont représentées dans le Tableau 10. Dans cette réalisation l'architecture utilise des processeurs de type Intel Pentium. Le temps d'exécution, représenté dans le tableau, fait référence au nombre de cycles d'horloge nécessaires au traitement d'une cellule ATM.

Intel Pentium	Estimation Manuelle (Cycles)	Estimation Automatique (Cycles)	Préci-sion %
TCP	3.900	4.339	+11
IP	5.645	6.124	+8
AAL	10.550	12.037	+14
ATM	4.335	5.076	+17

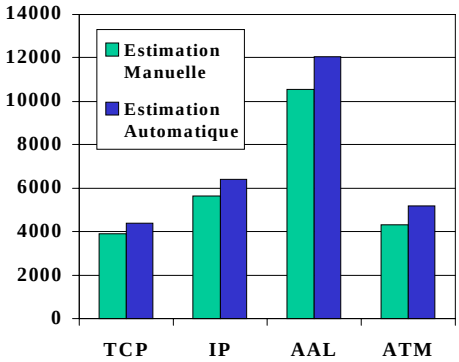


Tableau 10: Résultats préliminaires.

L'estimation réalisée par le concepteur donne pour la couche TCP environ 3,9 kilo cycles par cellule ATM. Le nombre de cycles calculé par l'algorithme d'estimation est de 4,3 kilo cycles, ce qui donne une différence de précision proche de 11%. Bien que l'on ne dispose pas de mesures réelles de performance nous pouvons estimer que ce résultat est très positif. En fait, l'estimation manuelle a été réalisée par le concepteur et a nécessitée plusieurs heures de travail. Tandis que, l'estimation automatique est réalisée de manière preste instantanée.

5.8 Conclusion

Le découpage logiciel/matériel génère une réalisation au niveau comportemental qui doit satisfaire aux contraintes de conception telles que la performance et la surface. Des mesures de qualité sont nécessaires pour appuyer la conception conjointe de deux manières:

- Les mesures sont nécessaires pour déterminer la qualité de la réalisation finale. Elles permettent la comparaison des caractéristiques de la réalisation avec des contraintes données;
- Les mesures ont une grande importance pour guider la synthèse à la recherche de la meilleure solution de réalisation, en améliorant la stratégie de raffinement de l'utilisateur.

L'utilisateur a besoin d'explorer un grand espace de solutions pendant la conception. Normalement l'architecture initiale proposée par l'utilisateur doit subir plusieurs étapes de raffinement pour être adaptée aux contraintes. De la même façon, des solutions architecturales différentes de celles prévues peuvent surgir pendant le raffinement. L'exploration nécessite des estimations rapides et fiables guidand les prises de décisions. Les informations calculées par les algorithmes présentées dans ce chapitre sont:

- le temps d'exécution, la taille du code et la taille des données (côté logiciel);
- le cycle d'horloge, le temps d'exécution et la surface (côté matériel);
- le temps de communication (procédures logicielles/matérielles et protocoles).

L'inclusion de ces trois modèles d'estimation dans la méthodologie Cosmos doit permettre un gain considérable de temps de conception. Ces algorithmes travaillent au sein d'un même environnement et l'utilisateur n'a plus besoin d'utiliser des moyens externes coûteux en terme de temps, ou des solutions informelles.

Le conflit existant entre la vitesse de calcul de l'estimation et la précision de calcul, qui est un facteur critique pour la plus grande partie des systèmes de conception conjointe existants, ne pose pas de problèmes dans Cosmos. L'approche semi-automatique de découpage de Cosmos peut, en effet, consacrer un temps plus grand au calcul de l'estimation que d'autres approches. Cette caractéristique est liée au fait que Cosmos utilise l'estimation pour valider une réalisation, plutôt que de servir comme base à l'algorithme de découpage automatique. Cosmos fournit à l'utilisateur des valeurs approximatives sur les caractéristiques de la réalisation pour que celui ci les interprète d'une façon plus réaliste qu'un simple algorithme.

Le point clef de l'opération d'estimation est de permettre à l'utilisateur de prendre la plus grande partie des décisions de conception au niveau système, pendant les premières étapes de conception. Ces décisions réduisent le temps de conception puisqu'elles réduisent les retours à des étapes antérieures, nécessaires à la correction des erreurs ou à l'exploration de solutions architecturales inadéquates. Nous constatons que le temps de conception est normalement le résultat du nombre de fois que l'activité de synthèse est nécessaire, cela veut dire, le nombre de fois que l'utilisateur doit revenir à des étapes antérieures jusqu'à l'obtention de l'architecture désirée (boucles de la conception). L'estimation réduit aussi le coût de réalisation en exploitant l'architecture optimale, une fois que l'utilisateur n'est plus obligé d'adopter des solutions coûteuses afin de respecter les restrictions imposées.

L'estimation du logiciel utilise l'approche de calcul de la fréquence d'exécution suivie de la modélisation du processeur. Les caractéristiques des architectures modernes, comme la hiérarchie de mémoire et le pipeline des instructions sont prises en compte. L'estimation du matériel est basée sur la synthèse comportementale, ce qui donne des résultats précis en terme de surface et de délai des blocs RTL. L'estimation de la taille des mémoires requise et du temps de communication utilisent des approches simples.

Le modèle d'estimation de Cosmos est assez puissant, mais il nécessite plusieurs améliorations, principalement du côté logiciel:

- la caractérisation des processeurs modernes afin de compléter la bibliothèque de processeurs de Cosmos. Cette caractérisation nécessite encore un effort considérable pour permettre la génération des résultats plus fiables;
- un travail plus approfondi sur la compilation Solar et la génération de code générique, ce qui peut augmenter largement la fidélité des résultats;
- une étude des approches d'optimisation qui font partie des outils de synthèse et des compilateurs de code. Ces approches d'optimisation peuvent être modélisés par l'algorithme d'estimation. Les premiers résultats ont montré que l'absence de la prise en compte des optimisations résulte en une surestimation des caractéristiques du système;
- et, finalement, l'introduction de nouveaux algorithmes d'estimation, comme par exemple le calcul de la consommation d'énergie.

Chapitre 6

Résultats et Evaluation

Everything should be made as simple as possible,
but not simpler.

Ce chapitre présente l'application de l'approche de découpage transformationnel de Cosmos à un système issus du domaine des télécommunications: une carte d'interface du réseau ATM. La carte ATM est destinée à lier des programmes d'application à un réseau, en utilisant la couche physique. L'objectif de ce projet est de produire des cartes ATM pour différents domaines d'application, qui demandent différents débits de transmission. Cette application en vraie grandeur a permis la validation de la chaîne de conception Cosmos et a aussi permis l'identification des faiblesses et des robustesses de la méthodologie.

6.1 La carte d'interface ATM

Ce chapitre présente la conception conjointe d'une carte d'interconnexion au réseau ATM (*Asynchronous Transfert Mode* [Pry95]). La carte ATM est destinée à lier des programmes d'application à un réseau ATM, en utilisant la couche physique. Chaque carte est modélisée par une pile composée de quatre couches de protocoles: TCP, IP, AAL et ATM. La Figure 66 illustre les quatre couches de la carte ATM, où les couches échangent des données de différents types et utilisent plusieurs formats d'encapsulation.

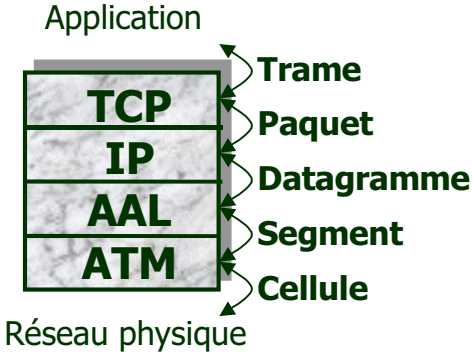


Figure 66: La carte d'interconnexion du réseau ATM.

La conception débute avec la spécification, en SDL, suivie par la simulation fonctionnelle au niveau système des quatre couches du protocole. Cosmos est utilisé pour réaliser les étapes de découpage, de synthèse de la communication et de génération du code C/VHDL. Finalement, la réalisation C/VHDL est co-simulée pour valider les étapes de raffinement du système.

Le but des étapes de raffinement réalisées est de pouvoir, à partir d'une description système, générer plusieurs réalisations, en fonction des performances requises des différents domaines d'application visés (transmission de données, vidéo, téléphonie, vidéophonie). L'étape de découpage, dans cet exemple, a pour objectif d'identifier quelles couches seront réalisées en logiciel et quelles seront réalisées en matériel. L'étape de synthèse de la communication établit les protocoles adéquats pour la communication entre les couches, en fonction des débits de transfert requis.

La spécification de la carte ATM, présentée dans ce chapitre, contient des simplification du modèle ATM, comme par exemple:

- Le modèle prend en compte des systèmes communicants à travers des connexions statiques. Les algorithmes de routage, de congestion, de contrôle de trafic et de gestion des ressources ne sont pas réalisés;
- La correction des erreurs n'est pas réalisée. Cela élimine la détection et la gestion des erreurs à tous les niveaux. En conséquence, l'*Internet Control Message Protocol* (ICMP) n'est pas présent et la retransmission des trames perdues est assurée par la couche de transport;
- Le modèle ne prend pas en compte le désordonnancement des trames. En conséquence, le mécanisme des fenêtres glissantes (*sliding window*) n'est pas réalisé.

Même avec ces simplifications, la partie restante de la carte ATM constitue un système très complexe. La Figure 67 illustre un système composé de deux cartes du réseau ATM qui communiquent via un modèle de couche physique. Ce système a permis le test du comportement des cartes ATM, par la simulation au niveau système et au niveau comportemental.

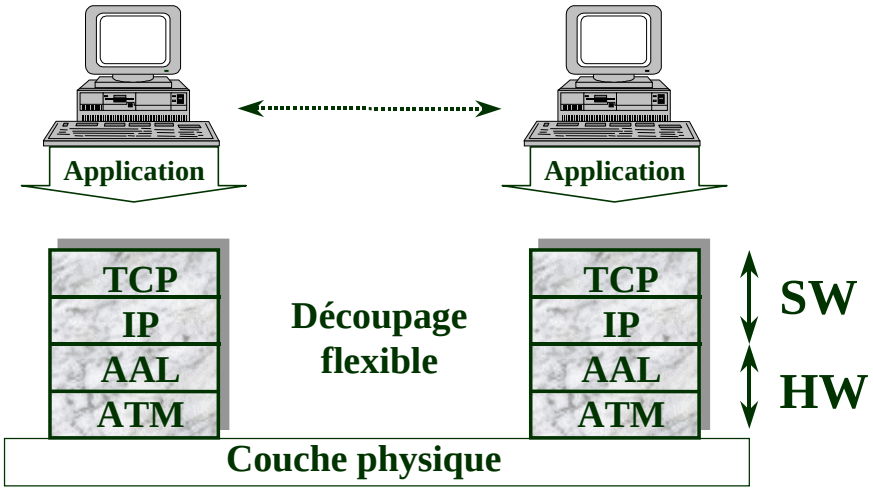


Figure 67: Modèle général de l'application.

La spécification du réseau ATM, dans l'équipe Cosmos, est en constante évolution. Le travail actuel consiste en l'ajout des types de données abstraits (*Abstract Data Types - ADTs*) à la spécification initiale. Cet ajout a pour objectif de surmonter les limitations de SDL dans le traitement des champs de bits et dans la conversion des formats.

6.2 La couche TCP

La couche TCP (*Transport Control Protocol*) a pour but d'offrir une communication fiable entre deux machines de vitesses différentes, en utilisant les services de la couche IP. Comme dans de nombreux protocoles, TCP utilise des temporisateurs de retransmission pour garantir la fiabilité. Contrairement à d'autres protocoles, TCP est élaborée pour fonctionner même si des datagrammes sont perdus, retardés, dupliqués, délivrés en désordre, tronqués ou corrompus. De plus, TCP autorise les machines qui communiquent à réinitialiser et à rétablir des connexions à tout moment, sans confondre les connexions précédemment ouvertes.

La structure d'une trame TCP est représentée dans la Figure 68 (à gauche), et la structure simplifiée, utilisée dans ce travail, est représentée à droite. La simplification de la trame du modèle TCP rend plus facile sa réalisation. Les champs *Reserved*, *Window*, *Checksum*, *Urgent pointer* et *Option* ne sont pas réalisés. Un entête de taille fixe (Hlen = 4) est utilisé. TCP a la capacité d'ouvrir une connexion avec un numéro de séquence aléatoire (*Sequence number*), mais pour des raisons de simplification de l'algorithme de génération de nombres aléatoires, nous ouvrirons systématiquement une connexion avec un numéro de séquence égal à 1. Le champ *Window* n'est pas utilisé, et la taille de la fenêtre est fixée à 1. Le champ Code est composé de six bits: *Urg* (*urgent pointer* valide), *Ack* (*acknowledge* valide), *Psh* (la trame doit être poussée), *Rst* (*reset* de la connexion), *Syn* (synchronisation des numéros de séquence), et *Fin* (fermeture de la connexion).

0	4	10	16	24	31
Source port				Destination port	
Sequence number					
Acknowledge number					
Hlen	Reversed	Code	Window		
Checksum				Urgent pointer	
Options					Padding
Data					
Data...					

0	4	10	16	24	31
Source port				Destination port	
Sequence number					
Acknowledge number					
Hlen	Code	Padding			
Data					
Data...					

Figure 68: Structure d'une entête de la trame TCP et le modèle simplifié.

Le protocole TCP est décrit sous la forme d'un automate d'états finis, représenté Figure 69. Il a été développé en tenant compte des simplifications apportées au modèle. L'automate commence dans l'état *Closed*. Les transitions sont étiquetées avec des signaux de garde suivies des signaux émis lors de l'exécution (en italique) de la transition.

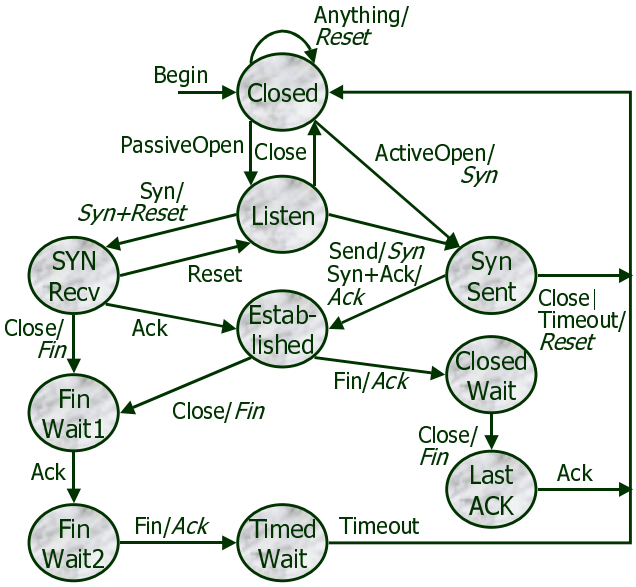


Figure 69: Automate du protocole TCP.

6.2.1 Etablissement et fermeture d'une connexion

La couche TCP est responsable de l'ouverture et de la fermeture des connexions entre les applications. L'ouverture et la fermeture d'une connexion se fait par un protocole du type *handshake* à trois phases, comme le montre la Figure 70. Une application peut demander l'ouverture d'une connexion par la commande *ActiveOpen* ou se mettre à l'écoute du réseau et attendre une connexion par la commande *PassiveOpen*.

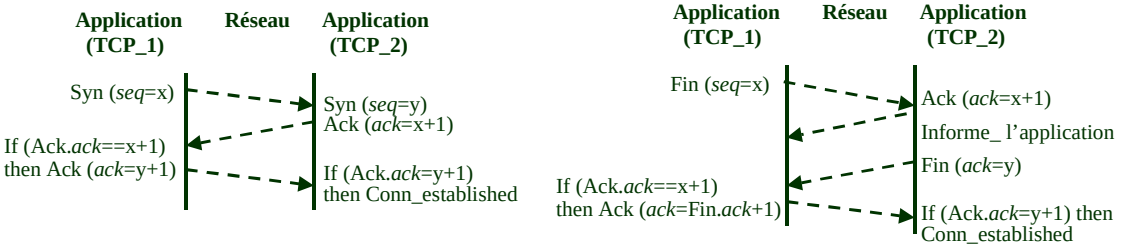


Figure 70: Ouverture et fermeture d'une connexion par protocole *handshake*.

Lorsque l'automate atteint l'état *Established* l'échange de paquets de données peut avoir lieu. Cet état est hiérarchique et est décrit lui-même par un automate. L'état *Established* contient trois sous-états qui sont représentés par leurs transitions, sur la Figure 71. Du fait que SDL ne supporte pas la hiérarchie comportementale, l'état *Established* a été mis à plat pendant la spécification.

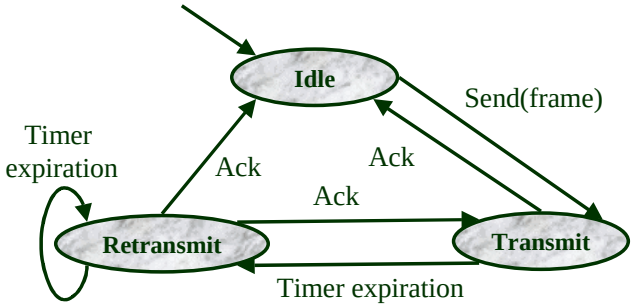


Figure 71: Automate de l'état *Established*.

Lorsque l'automate entre dans l'état hiérarchique *Established*, il active l'état *idle*. L'automate passe à l'état *transmit* à la réception d'une commande *send* provenant de l'application ou si un acquittement est prêt à être envoyé en réponse à une trame reçue. Le processus génère et envoie la trame, active un temporisateur et retourne à l'état *idle*. Si le temporisateur de retransmission expire, l'automate rentre dans l'état *retransmit* et effectue la retransmission de la dernière trame envoyée et non acquittée. Dans le cas de la réception multiple d'une même trame, celle-ci est éliminée, mais un signal d'acquittement est envoyé pour informer le processus émetteur que la trame a été reçue. Lorsque toutes les trames ont été envoyées ou acquittées, l'automate retourne dans l'état *idle*. Ce protocole de transfert des trames est assez simple. Un protocole de transfert beaucoup plus efficace peut être réalisé en utilisant le concept de fenêtre glissante [Puj95].

6.3 La couche IP

Le principal service offert par la couche IP (*Internet Protocol*) est le transport non connecté. Le protocole *internet* spécifie le format des paquets *internet*, appelés datagrammes, qui sont constitués d'un entête et de données. L'entête d'un datagramme contient des informations telles que les adresses source et destination, le contrôle de fragmentation, la précédence et le *checksum*, qui permet la correction d'erreurs. La Figure 72 détaille la structure d'entête du datagramme IP et la structure simplifiée.

0	4	8	16	19	31
Vers	Hlen	Service	Total length		
Identification			Flags	Frag.Offset	
Time to live		Protocol	Header checksum		
Source address					
Destination address					
IP options				Padding	
Data					
Data...					

0	4	8	16	31
Vers	Hlen	Protocol	Total length	
Source address				
Destination address				
Data				
Data...				

Figure 72: Structure d'un entête de la trame IP et le modèle simplifié.

Le champ *protocole* contient le type de protocole qui utilise les services de la couche IP. Pour un service TCP, cette valeur vaut <<6>>. D'autres utilisateurs du service TCP, tel que UDP (*User Datagram Protocol*), correspondent à d'autres valeurs du champ

protocole. Le champ *vers* contient le numéro de la version du protocole utilisé. Le numéro de la version actuelle est <<4>>. Le champ *total length* donne la longueur totale du datagramme (entête + données). Le champ *Hlen* (*header length*) est fixé à <<3>> et indique la longueur de l'entête en mots de 32 bits.

Lors de l'émission d'un datagramme, la couche IP ajoute l'entête IP avant d'envoyer la trame vers l'interface réseau représentée par les couches AAL/ATM. Lors de la réception d'un datagramme, la couche IP doit décider si le datagramme est valide ou non. Si le datagramme n'est pas valide, il est simplement éliminé. De même, s'il contient l'adresse IP d'une autre machine, il ne sera pas réémis sur le réseau mais éliminé. L'entête du datagramme est retiré avant que celui-ci soit envoyé vers la couche TCP.

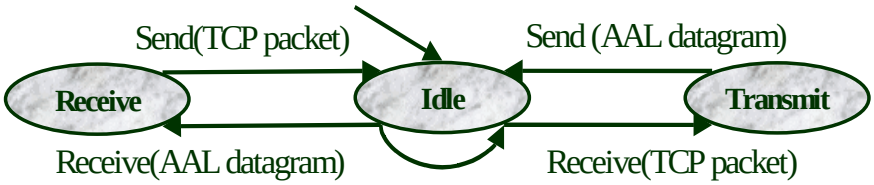


Figure 73: Automate de la couche IP.

L'automate de la couche IP est représenté Figure 73. Si un datagramme est reçu par la couche AAL, l'automate rentre dans l'état *Receive*. Si le paquet est reçu par la couche TCP, l'automate rentre dans l'état *Transmit*. Lorsque le traitement et l'envoi du segment à la couche concernée sont effectués, l'automate retourne dans l'état *Idle*.

6.4 La couche AAL

La couche AAL (*ATM Adaptation Layer*) a la fonction d'améliorer la qualité du service fourni par la couche ATM. Il peut s'agir de services de l'utilisateur ou des fonctions de contrôle comme la signalisation et la gestion. La couche AAL se compose de deux sous-couches: la sous-couche de convergence (CS ou *Convergence Sublayer*) et la sous-couche de segmentation et de réassemblage (SAR ou *Segmentation And Reassembly*). La sous-couche de convergence exécute des fonctions telles que l'identification des messages. La sous-couche SAR a pour objectif principal la segmentation des informations provenant de la couche supérieure en cellules ATM, ainsi que l'opération inverse, c'est-à-dire le réassemblage du contenu des cellules de la couche ATM en segments pour fournir à la couche supérieure [Kya95].

Un datagramme IP ne rentre pas dans une cellule ATM (de 53 octets) et doit donc être segmenté. Cette fonctionnalité sera assurée par la couche AAL. Il existe plusieurs formats de réalisation différents pour la couche AAL. Dans notre modèle le format <<AAL 5>> est réalisé. Ce format supporte les protocoles sans connexion et avec peu d'*overhead*, ce qui permet une réalisation rapide.

<<AAL 5>> offre deux modes de transfert: assuré et non assuré. Le mode de transfert assuré effectue la correction d'erreurs et la retransmission des segments perdus ou corrompus, ainsi que le contrôle de flux. Le mode non assuré n'effectue pas la correction d'erreurs et la retransmission des segments perdus ou corrompus. A la réception d'un segment erroné, détecté par le <<CRC 32 bits>>, le segment est simplement écarté. Il incombe aux couches supérieures de demander la retransmission.

6.4.1 La sous-couche CS

La sous-couche CS accepte des paquets de taille variable entre 1 et 65535 octets. La taille des paquets est rendue multiple de 48 octets par l'ajout de 0-47 octets de bourrage (champ *Padding*). Une en-queue de 8 octets de la couche de convergence est ajoutée à la fin de la cellule. La Figure 74 représente le format d'un segment CS.

1-65535 octets	0-47 octets	1 oct	1 oct	2 oct	4 octets
Data CPCS-PDU	Padding	UU	CPI	Length	CRC

Figure 74: Format du segment de la couche CS.

Le champ UU (*User to User Indication*) est réservé au transfert d'informations utilisateur. Le champ CPI (*Commom Part Indicator*) a plusieurs fonctions dont celle d'aligner l'en-queue sur 64 bits. Le champ *Length* donne la longueur de la charge utile du segment (données + bourrage). Le champ CRC (*Cyclic Redundancy Check*) contient un CRC de 32 bits utilisé pour détecter les trames corrompues. Il est calculé sur la totalité du segment. La trame est abandonnée si une erreur est détectée. La retransmission devra être redemandée par la couche TCP qui en sera informée par l'expiration de son temporisateur.

6.4.2 La sous-couche SAR

La sous-couche SAR accepte les segments de longueur variable (multiple de 48 octets) de la sous-couche CS et génère des cellules identiques aux cellules de la couche ATM. La délimitation des cellules s'effectue à l'aide du bit utilisateur du champ PTI (*Payload Type Identifier*) de l'entête de la couche ATM. Ce bit indique si une cellule ATM contient le début, la suite, ou la dernière cellule d'un segment. Ce bit a la valeur 0 pour le début ou la suite des cellules d'un segment et a la valeur 1 pour la dernière cellule, comme l'illustre la Figure 75.

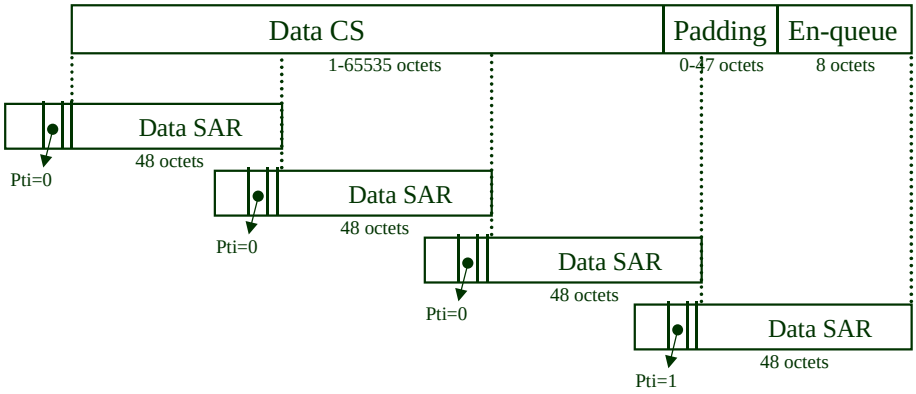


Figure 75: Flot de données de la couche SAR.

6.4.3 L'automate AAL

L'automate de la couche AAL a été divisé en deux processus concurrents, un pour l'émission et l'autre pour la réception. La Figure 76 décrit l'automate de réception de la couche AAL. Il commence dans l'état *Idle*. A la réception d'un segment de la couche ATM, l'automate entre dans l'état *Active* si le segment est indiqué comme étant le début ou la suite d'un segment (Pti=0). Dans l'état *Active*, l'automate réassemble le datagramme jusqu'à la réception du segment de fin (Pti=1). Le contrôle du CRC est effectué avant le

retour à l'état *Idle*. Si le segment tient dans une seule cellule ATM (Pti=1), l'automate traite le segment et reste dans l'état *Idle*. Dans l'automate, *Oversized* signifie le débordement de la taille du segment (> 65535 octets) et *Error* signifie que le segment est corrompu.

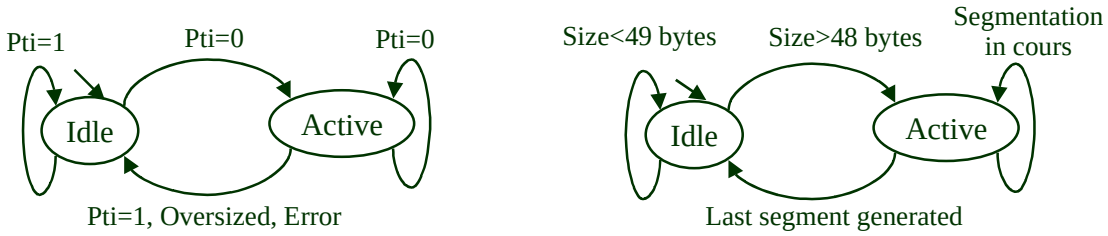


Figure 76: Automates d'émission et de réception de la couche AAL.

A la réception d'un datagramme, l'automate d'émission entre dans l'état *Active* si la segmentation est nécessaire, et retourne dans l'état *Idle* lorsque celle-ci se termine. La segmentation est nécessaire lorsque le datagramme a une taille supérieure à 48 octets.

6.5 La couche ATM

La couche ATM est indépendante du support physique et assure les fonctions suivantes: l'ajout (respectivement suppression) de l'entête pour les cellules provenant (à destination) de la couche d'adaptation; le contrôle de flux; le multiplexage et démultiplexage des cellules; la traduction d'adresses; l'ouverture et la fermeture des connexions; et la gestion de la qualité de service et de la bande passante requise.

La couche ATM reçoit des paquets de taille fixe (48 octets) de la couche AAL et génère des cellules ATM (53 octets) en ajoutant un entête de 5 octets. Afin de simplifier notre modèle, toutes les parties de signalisation, de gestion OAM (*Operation And Management*) et de gestion de flot sont réduites au minimum dans cette version.

L'entête des cellules ATM est représenté Figure 77. Le champ GFC (*Generic Flow Control*) est utilisé pour le contrôle d'accès dans les réseaux locaux. Le champ VPI/VCI (*Virtual Path Identifier/Virtual Channel Identifier*) est utilisé pour le routage des cellules ATM. Le champ PT (*Payload Type Identifier*) est utilisé pour l'identification du champ de données de la cellule. Il est constitué de 3 bits: le premier bit contient l'indicateur de début/suite (bit=0) ou de fin (bit=1) d'un segment, ce bit est affecté par la couche SAR; le deuxième bit contient des informations relatives à la congestion du réseau; et le troisième bit identifie les cellules de gestion du réseau. Le champ CLP (*Cell Loss Priority*) détermine la priorité de la cellule. Les cellules de basse priorité sont écartées en premier lors de la congestion du réseau. Le champ HEC (*Header Error Control*) est utilisé pour détecter des erreurs dans l'entête ATM au niveau physique.

7 4	1 0	
Generic Flow Control (GFC)	Virtual Path Identifier (VPI)	
VPI	Virtual Channel Identifier (VCI)	
VCI		
VCI	Payload Type (PT)	CLP
Header Error Control (HEC)		

Figure 77: Format de l'entête ATM.

6.6 Modélisation en SDL

Le réseau ATM a été spécifié en SDL à l'aide de l'outil ObjectGeode [Obj97]. La structure générale du modèle est représentée dans la Figure 78. La spécification complète du modèle SDL de l'ATM est assez complexe (1k lignes de code SDL).

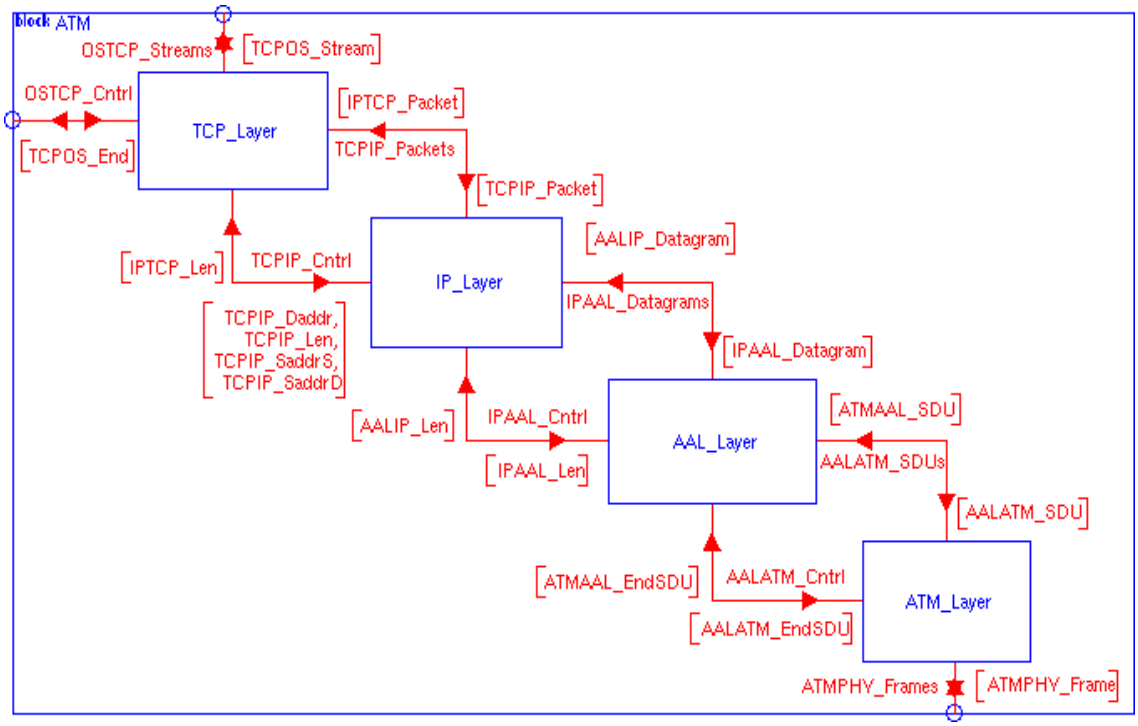


Figure 78: Structure de la description SDL de l'ATM.

Les principales difficultés rencontrées concernent la représentation des trames et leur segmentation. SDL étant un langage fortement typé, il n'est pas possible de définir des types pour chaque paquet (paquet TCP, IP, AAL, ATM), car la segmentation en paquet de la couche inférieure n'est pas directe. Cette restriction nous a amené à opter pour l'inclusion de code C dans la spécification SDL à travers des ADTs (*Abstract Data Types*). Les ADTs ajoutés réalisent le paquetage des données et d'autres opérations difficiles à modéliser en SDL, comme la manipulation de bits et les opérations arithmétiques complexes.

Le comportement de chaque couche est représenté par un automate d'états finis. L'automate d'états est modélisé en SDL par une machine d'états finis. La Figure 79 représente une partie de l'état *Established* où des éléments géométriques sont utilisés pour représenter des états, des conditions et des opérations. Le modèle complet du protocole TCP est trop grand pour être visible dans la figure.

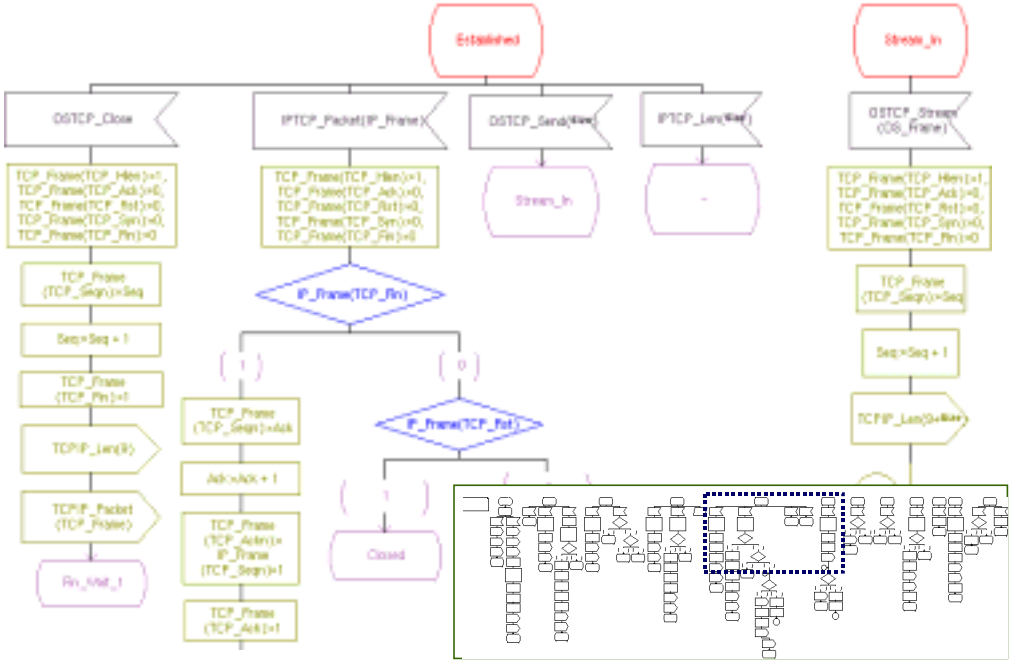


Figure 79: Comportement de la couche TCP.

Après la spécification, l'utilisateur peut valider la fonctionnalité du système, avec un outil de simulation au niveau système. La Figure 80 illustre la simulation SDL de l'ATM. Pour permettre la simulation, nous avons modélisé un système fermé composé de deux instances ATM interconnectées par un modèle du réseau physique. Chaque instance ATM est connectée à une application qui commande l'ouverture de la communication et le transfert des données. Sur la figure nous pouvons apercevoir les trames échangées lors de l'ouverture d'une connexion entre deux processus ATM. Après la validation, la spécification est prête pour le découpage transformationnel et la génération des différentes architectures.

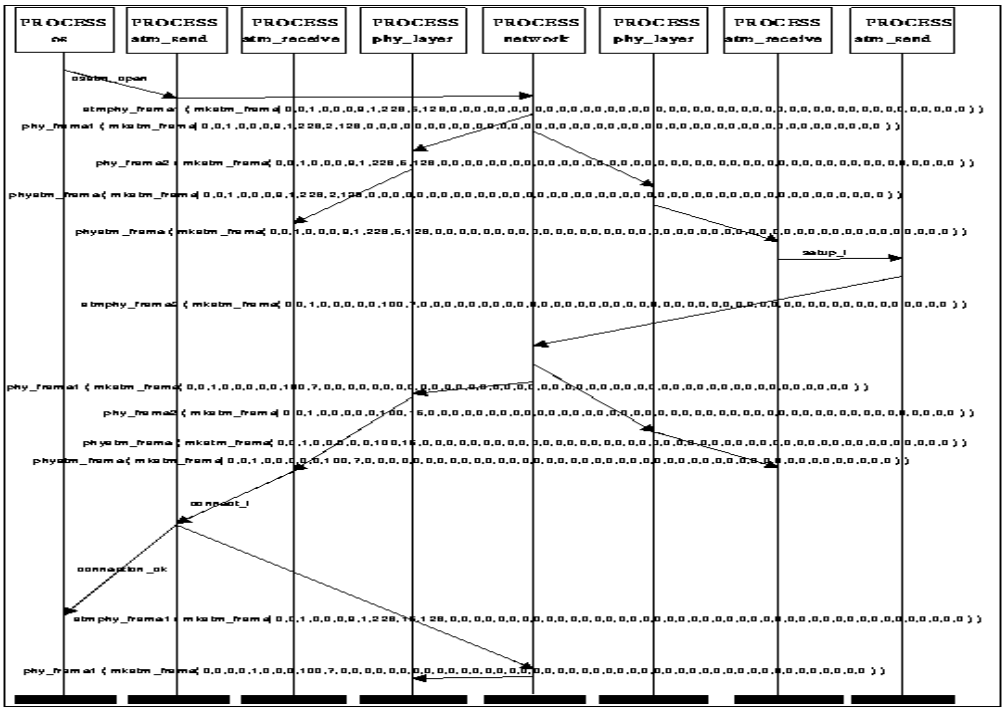


Figure 80: Simulation fonctionnelle de l'ATM en SDL.

6.6.1 Synthèse

La première étape de la synthèse concerne la traduction de la spécification SDL en Solar. La Figure 81 montre le format Solar graphique de la carte réseau ATM. Dans la figure, la structure du système apparait représentée sous forme de boîtes, où chaque boîte, correspondant à une unité de conception Solar, représente une couche du protocole. Les canaux de communication SDL sont traduits en Solar par des canaux de communication abstraits, représentés par des lignes horizontales.

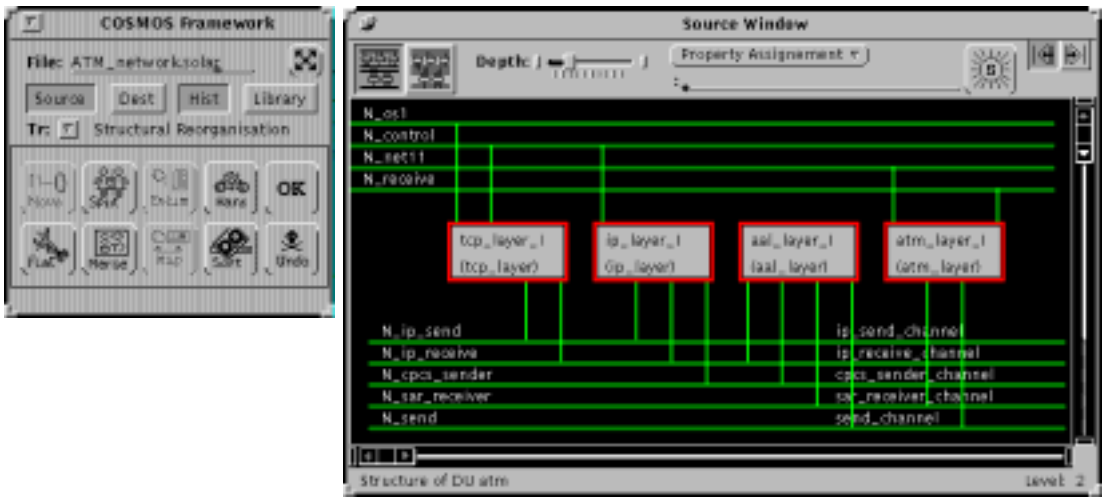


Figure 81: Modèle Solar de la carte ATM.

Dans une première réalisation, nous allons transposer les trois premières unités de conception (TCP, IP et AAL) sur un processeur logiciel, et la dernière unité (ATM) sur un processeur matériel. Cette opération est réalisée en utilisant l'ensemble de primitives de raffinement (voir section 4.3). Les primitives de transformation de la structure permettent le groupement des unités qui vont être attribuées au processeur logiciel. Les primitives de communication ont été utilisées pour réaliser les canaux abstraits.

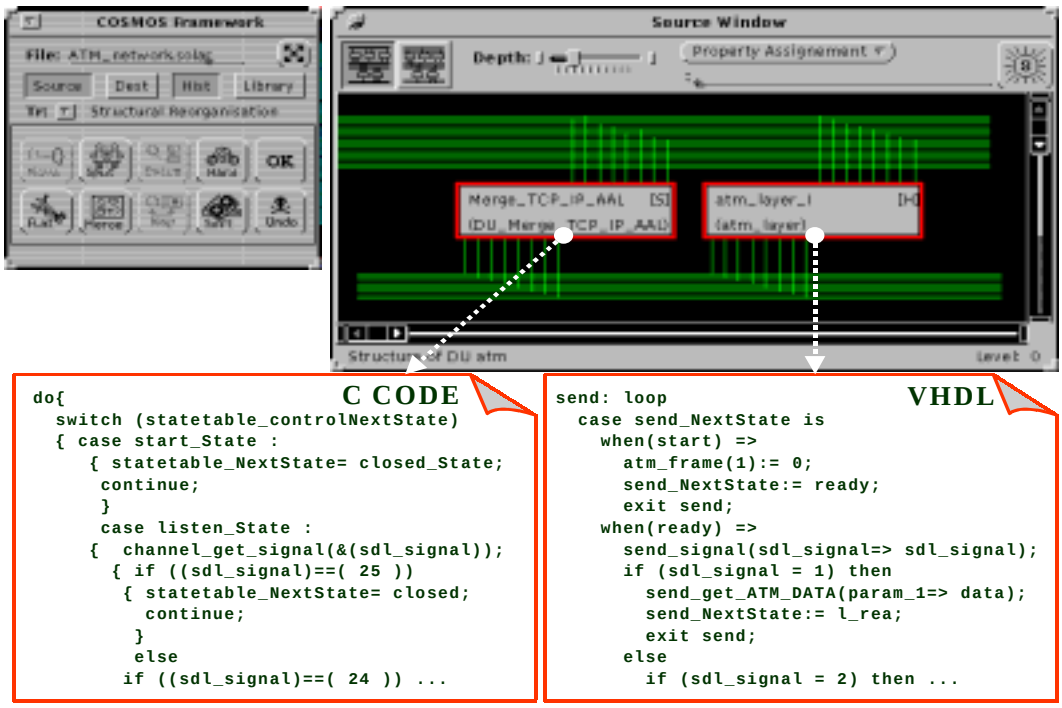


Figure 82: Code C/VHDL généré.

Le résultat des transformations est un autre modèle Solar composé de deux unités, qui correspondent aux processeurs alloués, comme montre la Figure 82. L'étape finale de la synthèse système est la génération du modèle C/VHDL utilisé par la co-simulation.

Plusieurs alternatives de découpage sont possibles en fonction des performances requises. Les alternatives explorées se sont limitées à l'affectation logicielle ou matérielle d'une couche et à l'affectation de plusieurs couches sur le même processeur (logiciel ou matériel). Le découpage est guidé par les performances requises et les estimations [Mar97]. La Figure 83 représente les alternatives de découpage et les performances obtenues pour chaque réalisation.

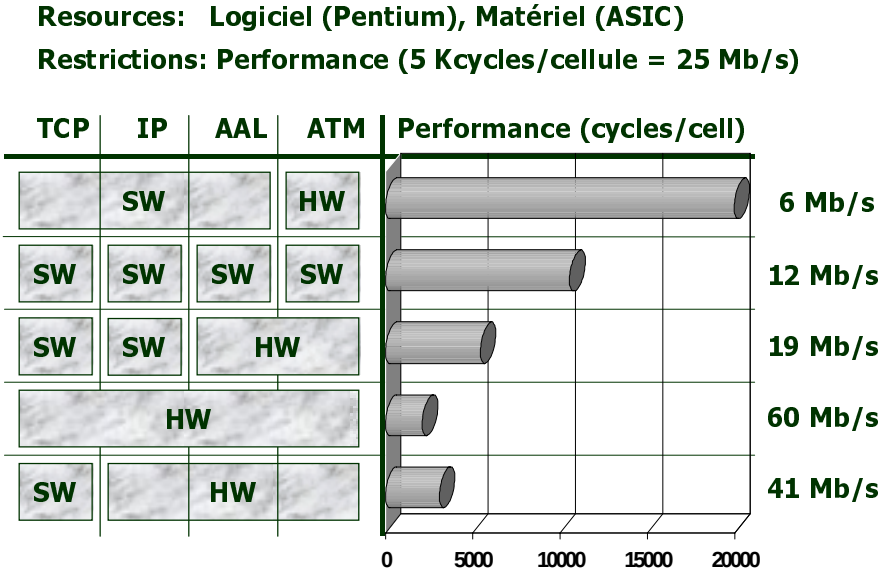


Figure 83: Alternatives de découpage logiciel/matériel de l'ATM.

Ces différentes réalisations ont été générées automatiquement à partir de la même spécification initiale. Toutes les solutions utilisent un processeur Pentium pour la partie logicielle et un ASIC pour la partie matérielle. Pour chaque solution la performance estimée est basée sur la co-simulation et sur les techniques d'estimation manuelles du logiciel. La première réalisation est composée d'un processeur logiciel qui exécute les trois premières couches. Cette réalisation ne respecte pas la contrainte de performance. La deuxième solution utilise un processeur logiciel pour chaque couche de l'ATM. La troisième solution utilise 7 K cycles pour le calcul d'une cellule ATM. La quatrième solution a une performance élevée, puisque c'est une réalisation entièrement matérielle. Et, finalement, la dernière solution réalise la couche TCP sur un processeur logiciel.

Pendant cette conception, la bibliothèque de communication a été étendue afin de supporter les schémas de communication requis par les différents processus SDL. De nouvelles primitives de communication permettant de transférer les types de données utilisés dans la spécification SDL ont été ajoutées aux unités de communication déjà existantes (protocole *fifo* et *handshake*).

6.7 Résultats

Le Tableau 11 résume les principaux résultats de la co-synthèse de la carte ATM en terme de lignes de code et de temps de simulation. Premièrement, la spécification SDL initiale est presque 10 fois plus petite que le modèle C/VHDL généré. Cette différence est

principalement due au raffinement de la communication. La simulation du modèle SDL est 15 fois plus rapide que la simulation du modèle VHDL, et 30 fois plus rapide que la co-simulation du modèle C/VHDL. Cette différence est liée au modèle de communication. Cette valeur peut varier selon les protocoles de communication utilisés. Dans le cas de l'ATM, la communication entre les différents blocs implique le transfert de grandes quantités de données et l'utilisation de protocoles tel que le *fifo* ou le rendez-vous.

Style de description				Comportement (lignes)	Communication (lignes)	Simulation (minutes)
TCP	IP	AAL	ATM			
SDL				794	103	1
VHDL				7.210	5.382	15
C	C	C	VHDL	≅	≅	30

Tableau 11: Résultat de synthèse de la carte ATM.

La Figure 84 illustre la simulation VHDL de deux instances de la carte réseau ATM. Les états des différents automates depuis l'ouverture jusqu'à la fermeture de la connexion sont représentés.

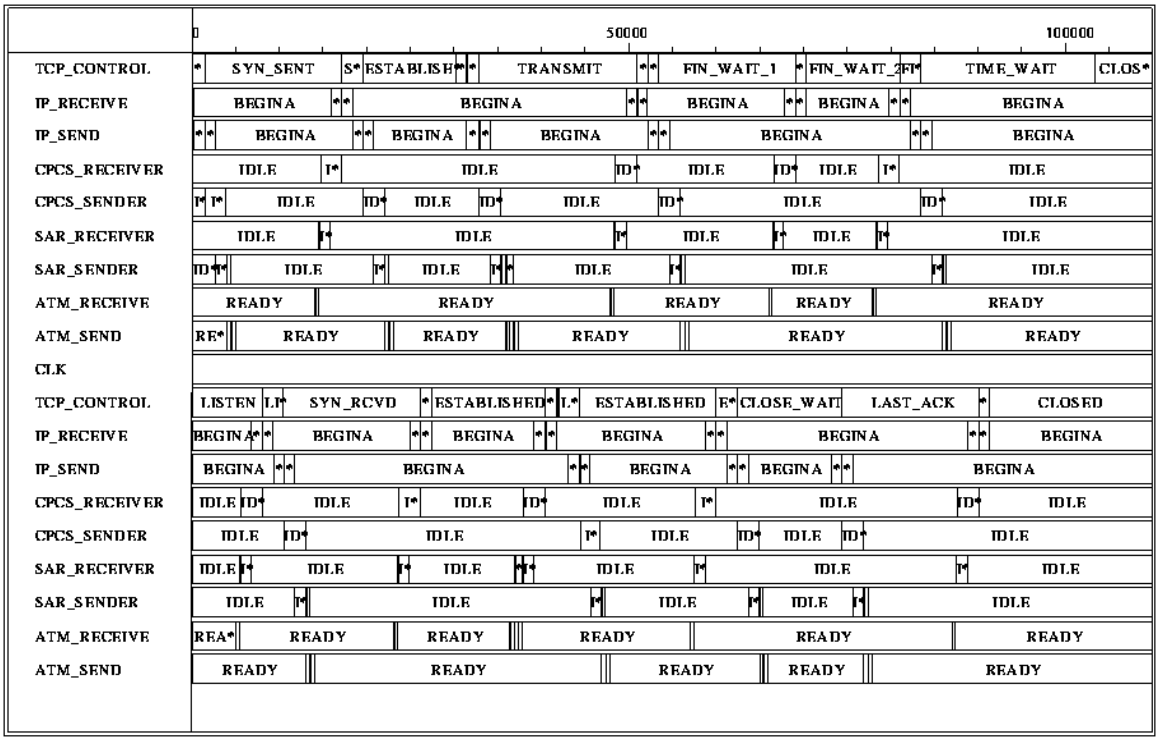


Figure 84: Simulation VHDL de l'ATM

6.8 Conclusion

Ce chapitre a présenté l'application de la méthodologie de conception conjointe Cosmos à un exemple complet du domaine des télécommunications: une carte du réseau ATM. La conception a commencé à partir d'une spécification abstraite de haut niveau pour arriver à la génération de plusieurs réalisations, correspondantes à différentes architectures et performances. Les réalisations résultantes ont des performances liées aux contraintes de conception. Cette conception a démontré qu'il est possible d'obtenir une réalisation C/VHDL simulable, synthétisable et adaptée aux contraintes de coût et de performance, à partir d'une spécification au niveau système donnée en SDL.

Les tailles du code SDL de la spécification initiale et du code C/VHDL des la réalisations varient considérablement. Nous avons constaté que l'ajout des informations, par les étapes de raffinement, occasionne une augmentation moyenne du code de 10 fois (en nombre de lignes). Cela veut dire que la taille de la spécification au niveau système est 10 fois plus petite, et proportionnellement plus facile à comprendre et à traiter, que la taille de la spécification comportementale. De la même façon, les temps de simulation et de co-simulation varient beaucoup. Ces différences montrent clairement le bénéfice de l'utilisation d'une spécification au niveau système.

Pendant cette conception, nous avons pu constater que SDL est adapté à la description de protocoles de communication sous forme de processus échangeant des messages. La description de flot de contrôle n'est pas aisée dans la mesure où l'on ne dispose pas d'instructions de haut niveau telles que *for* ou *while*. De même, SDL n'est pas adapté à la manipulation de données malgré l'existence des types de données abstraits. Le langage est fortement typé et supporte mal les conversions de types. Il ne supporte pas aisément les champs de bits, l'affectation d'un bit particulier ou la concaténation.

En SDL, il n'est pas possible de spécifier le parallélisme à un niveau autre que le processus, ce qui peut amener une augmentation du nombre de processus et des communications inter-processus. Un processus ne peut posséder qu'une file d'attente. Si plusieurs processus communiquent avec un même processus, ils devront tous utiliser le même schéma de communication, ce qui peut ne pas être optimal.

Ces restrictions ont mené à la simplification du modèle de la carte du réseau ATM utilisé dans les premières versions et de nombreux détails du protocole TCP/IP et ATM ont été omis. En particulier au niveau de la couche ATM, où la description de concepts du matériel est difficile en SDL.

La version actuelle de la carte du réseau ATM est assez complète, grâce à l'inclusion des types de données abstraits (ADTs). Les ADTs permettent l'utilisation du code C intégré à la description SDL et permettent donc la manipulation et la conversion des différents types de données. La manipulation aisée des types de données a permis l'encapsulation de données (frames, paquets, datagrammes, segments, cellules) des différentes couches.

Conclusion

The reasonable man adapts himself to the world;
the unreasonable one persists in trying to adapt the world to himself.
Therefore all progress depends on the unreasonable man.

Ce travail de thèse a développé une nouvelle méthodologie de conception conjointe logicielle/matérielle, basée sur une approche transformationnelle et interactive de découpage, capable de manipuler des systèmes distribués ainsi que des architectures multiprocesseurs. Cette approche transformationnelle permet à l'utilisateur de réaliser le lien entre une spécification au niveau système et une architecture logicielle/matérielle avec un temps de conception réduit et avec une exploration rapide de l'espace des solutions.

L'approche transformationnelle est très pragmatique, ce qui facilite son application dans le domaine industriel. Elle diffère en plusieurs aspects des approches existantes, parce qu'elle met en pratique une méthodologie semi-automatique de découpage logiciel/matériel. Cette méthodologie est basée sur des algorithmes simples de raffinement de la spécification et utilise l'intervention de l'utilisateur afin de prendre des décisions pendant la conception. Finalement, cette approche transformationnelle manipule des architectures flexibles, composées de processeurs qui communiquent à travers un réseau complexe.

Ce travail fait partie du projet Cosmos et s'intègre dans un travail de groupe. Les principaux résultats spécifiques à ce travail de thèse sont listés dans ce qui suit:

7.1 Environnement de conception Cosmos

Au début de ce travail de recherche, Cosmos était une méthodologie de conception composée d'outils et d'algorithmes isolés. La construction d'une interface graphique a permis la réalisation d'un environnement unifié pour l'exploration de l'espace des solutions et synthèse. Celui ci facilite aussi la présentation de la méthodologie au public extérieur. Dans l'état actuel (version 3.2), l'environnement Cosmos exécute le flot complet de conception de la méthodologie.

Cosmos a reçu plusieurs enrichissements au cours de ce travail, comme: la création d'un environnement qui intègre les outils de Cosmos; la création des versions fermes du système (environnement et outils); la construction des outils de calcul de la fréquence d'exécution, d'estimation de performance du logiciel et de découpage; l'exécution d'exemples complets de conception, à partir de la spécification jusqu'à l'architecture; la définition et l'écriture des protocoles de la bibliothèque de communication; et la définition et la mise en pratique d'une approche d'estimation de performance;

Ces nouvelles caractéristiques ont permis:

- de rendre plus facile l'apprentissage et la compréhension de la méthodologie Cosmos;
- une visualisation claire et facile de la structure hiérarchique et du comportement du format Solar ;
- une exploration facile de l'espace des solutions. La bonne adéquation de l'environnement à la méthodologie permet à l'utilisateur d'essayer plusieurs alternatives de réalisation ;
- de mieux guider le flot de conception. L'utilisateur a le contrôle sur le flot de raffinement, ce qui lui permet de guider le processus jusqu'à la réalisation souhaitée;
- de présenter de façon claire les résultats des opérations invoquées par l'utilisateur. La présentation des résultats de l'estimation de performance est directe et compréhensible par l'utilisateur.
- une vérification facile du fonctionnement des outils. Les erreurs relatives aux descriptions générées sont facilement identifiées.

7.2 Format intermédiaire Solar

Les objectifs initiaux de la construction du modèle Solar étaient:

- l'identification des structures de données permettant l'accommodation des besoins nécessaires aux étapes de co-spécification, de conception conjointe et de co-synthèse;
- de permettre la construction d'outils de synthèse au niveau système basés sur un format de représentation unique;
- d'utiliser ce modèle pour la conception de systèmes logiciels/matériels complexes et distribués;
- de fournir des caractéristiques qui permettent une modélisation orientée langage et orientée architecture.

Pendant le déroulement de cette thèse, Solar a été validé et a reçu plusieurs enrichissements, comme: l'unification du format utilisé par les différents outils du

système; la jonction de nouveaux objets à la demande des nouveaux algorithmes; et la mise à jour de plusieurs classes existantes.

Cette opération de maturation a été possible grâce à plusieurs facteurs: l'utilisation du format intermédiaire sur plusieurs exemples; la construction de nouveaux outils manipulant les caractéristiques du modèle Solar qui n'avaient pas été testées auparavant; l'intégration des outils en un environnement unique; et finalement, l'interface graphique Solar qui a donné une nouvelle identité au format intermédiaire.

7.3 Découpage transformationnel

Le découpage est en train de devenir le goulet d'étranglement du processus de développement des systèmes électroniques complexes. Dans une méthodologie de conception traditionnelle, l'utilisateur réalise le découpage logiciel/matériel à un stade prématuré du cycle de développement. Les différentes parties du système sont conçues par différents groupes. L'intégration de ces différentes parties amène généralement à une détection tardive des erreurs. La propagation des erreurs à des étapes de conception postérieures génère une augmentation du coût de conception et une augmentation du temps nécessaire à l'étape d'intégration. Ce découpage prématuré restreint les possibilités d'investigation d'un meilleur découpage. L'utilisateur est obligé de surdimensionner les différentes parties du système dans le but de réduire les risques d'intégration de dernière minute.

L'approche transformationnelle présentée dans ce travail est très pragmatique et se base sur l'interaction de l'utilisateur. L'utilisateur dispose d'un puissant ensemble de primitives pour le découpage fonctionnel, la réorganisation de la structure et la transformation de la communication. Cette approche apporte des solutions élégantes aux principaux défis imposés au découpage: l'utilisation de l'expertise de l'utilisateur pendant le découpage; permettre à l'utilisateur de comprendre les détails du processus de conception conjointe; prendre en considération des solutions partielles; permettre une exploration facile de l'espace des solutions.

7.4 Estimation de performance

L'utilisateur doit concevoir des architectures satisfaisant aux besoins fonctionnels, mais aussi aux restrictions de coût et aux objectifs de performance. Une fois établis les besoins fonctionnels, pendant l'étape de spécification, l'utilisateur doit concevoir son système en essayant d'optimiser son architecture. Le choix de la conception optimale dépend de plusieurs critères. Les critères les plus courants concernent les performances et le coût, mais d'autres critères mesurables peuvent être aussi importants pour certains domaines d'application. De la même façon, pendant le raffinement de la spécification, l'utilisateur doit avoir constamment en tête l'impact des décisions de conception sur les caractéristiques de la réalisation.

Le modèle d'estimation, présenté dans cette thèse, a une bonne précision, dans la mesure où la modélisation de l'architecture prend en considération plusieurs aspects du côté logiciel comme du côté matériel. Il y a deux principaux défis imposés par les algorithmes d'estimation de performance: du côté logiciel, c'est la difficulté à modéliser les techniques utilisées pour augmenter la vitesse d'exécution du code; et du côté matériel,

c'est la difficulté à modéliser les différentes possibilités de réalisation et les différentes décisions prises pendant la synthèse.

Les informations calculées par les algorithmes d'estimation sont: le temps d'exécution, la taille du code et des données (côté logiciel); le cycle d'horloge, le temps d'exécution et la surface (côté matériel); et le temps de communication (procédures logicielles/matérielles et protocoles).

L'inclusion de ces trois modèles d'estimation à la méthodologie Cosmos a permis un gain considérable de temps de conception. Ces algorithmes travaillent au sein d'un même environnement, et l'utilisateur n'a plus besoin d'utiliser des moyens externes, coûteux en terme de temps, ou des solutions informelles. Finalement, le conflit existant entre la vitesse de calcul de l'estimation et la précision de calcul, qui est un facteur critique pour la plus grande partie des systèmes de conception conjointe existants, ne pose pas de problèmes dans Cosmos. L'approche semi-automatique de découpage de Cosmos peut, en effet, consacrer un temps plus grand au calcul d'estimation, que d'autres méthodes. Cette caractéristique est liée au fait que Cosmos utilise l'estimation pour valider une réalisation, plutôt que pour servir de base à l'algorithme de découpage. Cosmos fournit à l'utilisateur des valeurs approximatives sur les caractéristiques de la réalisation pour que l'utilisateur interprète les résultats d'une façon plus réaliste qu'un simple algorithme.

7.5 Application et évaluation

L'évaluation de l'approche transformationnelle à été possible grâce à l'utilisation de plusieurs exemples. Dans cette thèse sont décrites les étapes de conception et les leçons apprises par l'utilisation de Cosmos sur une carte d'interconnexion au réseau ATM. La carte ATM est destinée à interfacer des programmes d'application à un réseau, en utilisant le protocole ATM. Chaque carte est modélisée par une pile composée de quatre couches de protocoles (TCP, IP, AAL et ATM) qui échangent des données de différents types et utilisent différents formats d'encapsulation.

Pendant le déroulement de la conception, nous avons pu identifier certaines limitations du langage SDL pour la spécification des systèmes complexes, et aussi mesurer les conséquences de ces limitations sur la spécification des fonctionnalités de la carte du réseau ATM.

Nous avons constaté que l'ajout d'informations, par les étapes de raffinement, occasionne une augmentation moyenne du code d'un facteur 10 en nombre de lignes. Cela veut dire que la taille de la spécification au niveau système est 10 fois plus petite, et proportionnellement plus facile à comprendre et à traiter, que la taille de la spécification comportementale. De la même façon, le temps de simulation et de co-simulation varient beaucoup. Ces différences montrent clairement le bénéfice de l'utilisation d'une spécification au niveau système.

Pendant cette conception, nous avons aussi pu constater que SDL est adapté à la description de protocoles de communication sous forme de processus échangeant des messages. La description de flot de contrôle, quand à elle, n'est pas aisée dans la mesure où l'on ne dispose pas d'instructions de haut niveau telles que *for* ou *while*. SDL n'est pas adapté à la manipulation de données malgré l'existence des types de données abstraits. Le langage est fortement typé et supporte mal les conversions de types.

7.6 Conclusion

Il y a quatre points forts résultant des décisions prises au début de ce travail, qui contribuent à mettre Cosmos en évidence sur la scène des outils de conception conjointe:

1. L'utilisation de SDL comme langage de spécification. Ce langage est standardisé par l'ITU (International Telecommunication Union) et est en constante croissance d'utilisation pour la spécification de systèmes;
2. L'utilisation d'une forme intermédiaire Solar, capable de modéliser les aspects langage et architecture de la réalisation au sein d'un format unifié;
3. l'approche de découpage semi-automatique qui se présente comme la solution la plus pragmatique pour traiter les différents aspects et besoins des domaines d'application, sans restreindre l'exploration de l'espace des solutions;
4. la synthèse de la communication qui utilise des protocoles prédéfinis.

Cette thèse décrit le résultat de quatre ans de travail sur la méthodologie et l'environnement Cosmos. Pendant cette période nous avons vu apparaître plusieurs méthodologies de conception conjointe. Malheureusement, jusqu'à maintenant, aucune de ces méthodologies n'est devenue un standard et la conception conjointe continue à imposer ses défis.

Pour conclure, la méthodologie Cosmos n'est pas le résultat du travail d'une seule personne, mais d'un grand nombre de chercheurs. Chaque thèse ou stage clarifie des aspects de la conception conjointe qui étaient jusqu'à présent obscurs, et permet l'avancement de la méthodologie. Cosmos est une approche de conception pragmatique, simple et efficace, et elle est déjà utilisée dans plusieurs centres de recherche. L'avenir semble être prometteur et son transfert dans l'industrie semble être faisable.

La version actuelle du système est stable et nous encourageons vivement le lecteur à le tester et à l'utiliser. La demande des outils, pour la recherche et l'enseignement, peut être faite par simple émail à l'adresse suivante: cosmos@verdon.imag.fr.

Abréviations

AAL : Couche d'adaptation ATM (*ATM Adaption Layer*);

ANSI : Institute national américain des standards (*American National Standards Institute*);

API : Interface de programmation des applications (*Application Programing Interface*);

ASIC : Circuit intégré d'application spécifique (*Application Specific Integrated Circuit*);

ASIP : Processeur intégré dédié à l'application (*Application Specific Integrated Processor*);

ATM : Protocole de communication asynchrone dhaut débit (*Asynchronous Transfer Mode*);

C : Langage de programmation;

CAO : Conception assistée par l'ordinateur;

CASE : Ingénierie des systèmes aidée par l'ordinateur (*Computer Aided System Engineering*);

CASHE : Ingénierie des systèmes logiciel/matériel aidée par l'ordinateur (*Computer Aided Software/Hardware Engineering*);

CCITT : Organisme de normalisation des protocoles de communication (*Consultative Committee on International Telephony and Telegraphy*);

CFG : Graphe de flot de contrôle (*Control Flow Graph*);

CMOS : Technologie de fabrication des circuits Intégrés;

CODES: Outil de conception conjointe logicielle/matérielle développé par Siemens;

COSMOS: Outil de conception conjointe logicielle/matérielle développé par le laboratoire TIMA;

COSYMA: Outil de conception conjointe logicielle/matérielle développé par l'université de Braunschweig;

CPU : Unité centrale de calcul (Central Processing Unit);

CSP : Langage de description de processus séquentiels communicants (*Communicating Sequential Processes*);

CU : Unité de communication Solar (*Channel Unit*);

DAG : Graphe acyclique direct (*Direct Acyclic Graph*);

DFG : Graphe de flot de donnée (*Data Flow Graph*);

DU : Unité de conception Solar (*Design Unit*);

DSP : Processeur de traitement de signal digital (*Digital Signal Processor*);

EDIF : Format d'échange pour la conception des systèmes électroniques (*Electronic Design Interchange Format*);

EFSM : Machine d'états finis étendue (*Extended Finite State Machine*);

ESTEREL: Langage parallèle synchrone ayant une sémantique mathématique;

ESTELLE: Langage pour la spécification des protocoles de communication;

FPGA : Circuit programmable (*Field Programmable Gate Array*);

MEF : Machine d'états finis;

FSMD : Machine d'états finis avec chemin de données (*Finite state machine with datapath*);

INPG: Institut Nationale Polytechnique de Grenoble;

IEEE : Organisation de normalisation dans le domaine électronique (*Institut of Electrical, Electronic and Engineers*);

ILP : Programmation linéaire en nombre entiers (*Integer Linear Program*);

I/O: Entrée/sortie (*Input/Output*);

IP : Protocole internet (*Internet Protocol*);

IPC : Communication inter-processus (*Inter Process Communication*);

ISO : Organisme de standardisation internationale (*International Standards Organization*);

LOTOS: Langage de description formel de protocoles et de systèmes distribués;

LSI : Circuit intégré à large échelle (*Large Scale Integrated circuit*);

MHz : Unité de mesure de fréquence (*Mega Hertz*);

MIPS: Million d'instructions par seconde;

MPU : Unité multiprocesseur (*Multi processor Unit*);

MSC: Diagramme de séquence de messages (*Message Sequence Chart*);

MUSIC: Conception conjointe multi-langage pour des systèmes en circuit (*MUltilagage codesign for system on Chip*);

PE : Elément de calcul (*Processing Element*);

PID : Descripteur d'identification de processus (*Process Identification Descriptor*);

PTOLEMY: Environnement de simulation et conception logicielle/matérielle de l'université de Berkeley;

RISC : Ordinateur à jeu d'instruction réduit (*Reduced Instruction Set Computer*);

RPC : Appel de procédure à distance (*Remote Procedure Call*);

RTL : Niveau transfert des registres (*Register Transfer Level*);

SDL : Langage de spécification et description (*Specification and Description Language*);

SLS: Synthèse au niveau système (*System Level Synthesis*);

SOLAR: Format intermédiaire de description du système Cosmos;

ST : Table d'états (*State Table*);

STATECHARTS: Langage synchrone, permettant de décrire des MEFs hiérarchiques;

TCP : Protocole de transmission asynchrone (*Transmission Control Protocol*);

TIMA: Techniques de l'Informatique et de la Microélectronique pour l'Architecture d'Ordinateur;

TOSCA: Outil de conception logicielle/matérielle d'Italtel;

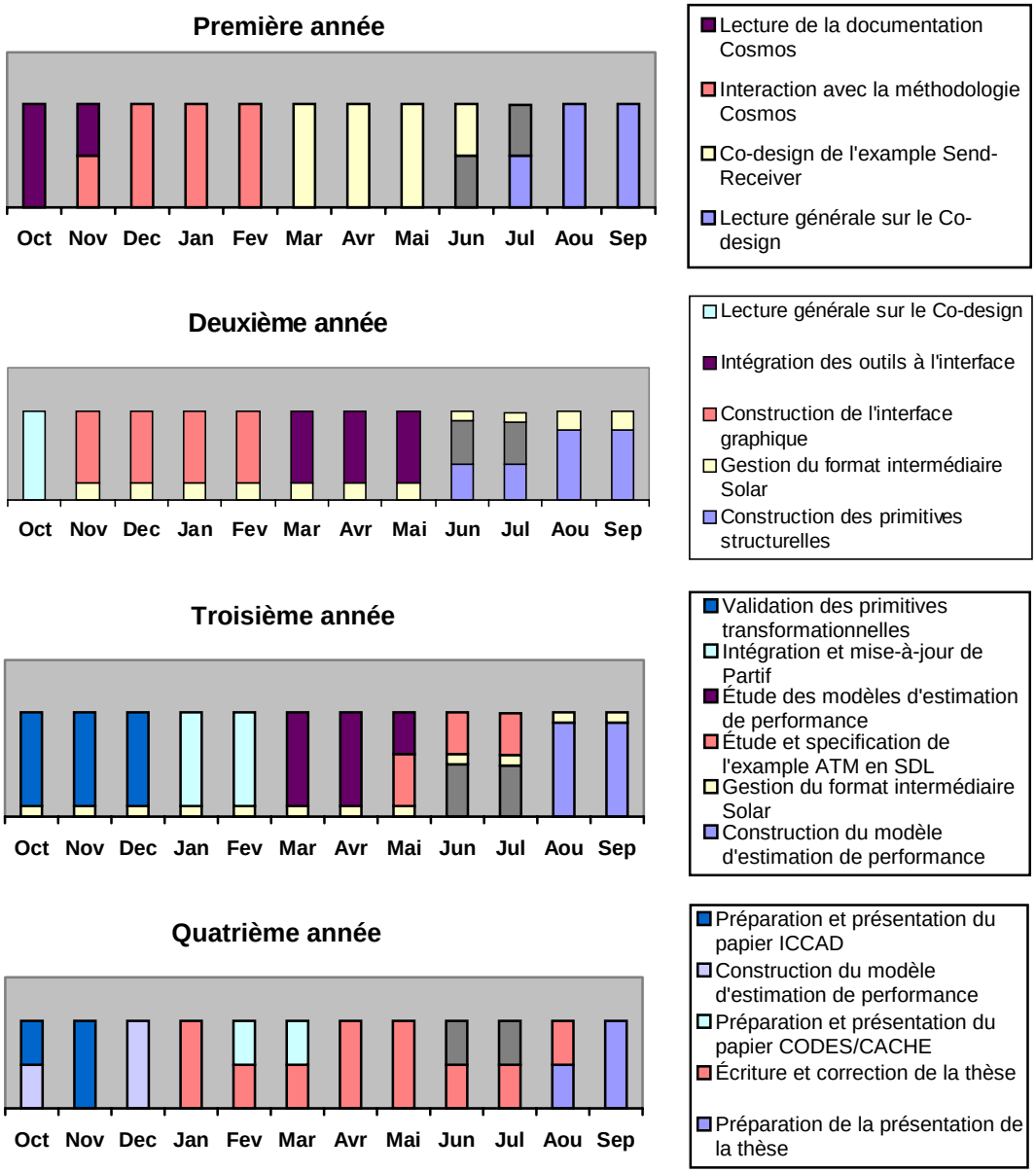
UNIX : Système d'exploitation;

VERILOG: Langage de description du matériel;

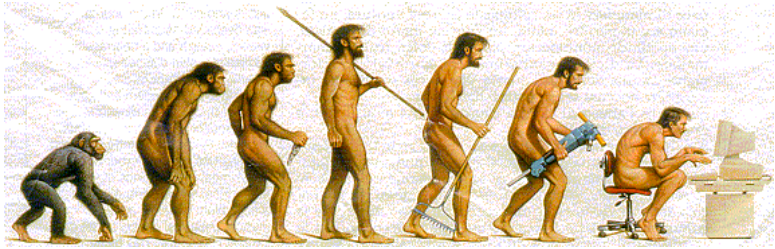
VHDL : Langage standard de description du matériel (*VHSIC Hardware Description Language*);

VLSI : Circuit à très grande échelle d'intégration (*Very Large Scale Integration circuit*).

Calendrier de la Thèse



Appendice



Appendice A

Primitives Structurelles

We never make mistakes,
we only learn great lessons.

Cet appendice présente les algorithmes des primitives de transformation structurelles (voir section 4.6). Ces primitives réalisent le raffinement de la structure du système. Le but de ces primitives est de grouper les blocs de comportement qui vont être exécutés sur un même processeur de l'architecture cible.

Primitive *Split* Structurelle

La primitive *Split* structurelle transforme le comportement d'une unité de conception parallèle en un ensemble d'unités interconnectes. Chaque état parallèle devient une nouvelle unité de conception comportementale. Les principales transformations de la description sont: l'adaptation des interfaces des unités, la copie des constantes globales au comportement des nouvelles unités, la conversion des variables globales en canaux abstraits, et la transformation des accès des variables globales en appels de procédures de communication.

```
Split(Appel d'Instance _IUC)
| _UC est l'unité de conception qui contient l'appel d'instance _IUC;
| Si _IUC est un appel d'instance à une unité comportementale _UCC;
| | Si _UCC contient des états parallèles _EP dans le premier niveau de la
hiérarchie;
| | | Pour chaque état parallèle _EPi de _UCC;
| | | | Créer une unité de conception comportementale _UCi;
| | | | Ajouter un appel à l'instance UCi dans les appels d'instances de _UC;
| | | | Copier le comportement _EPi dans _UCi;
| | | | Pour chaque porte _Pi d'interface de _UCC;
| | | | | Si la porte _Pi de l'interface de _UCC est utilisée dans _EPi;
```

```
| | | | | | Créer une porte _Pi dans l'interface de _UCi;
| | | | Pour chaque constante partagée _Ci de _EPi;
| | | | | Déclarer la constante _Ci dans _EPi;
| | | | Pour chaque variable partagée _Vi de _EPi;
| | | | | Si un canal relatif à la variable _Vi n'a pas encore été ajouté dans
_UC;
| | | | | | Ajouter un canal _N dans le réseau de _UC;
| | | | | Si un accès au canal _N n'a pas été encore ajouté dans _UC;
| | | | | | Ajouter l'accès au canal _N dans le réseau de _UC;
| | | | | Pour chaque instruction _Ii de _EPI;
| | | | | | Si l'instruction _Ii fait une lecture à la variable _Vi;
| | | | | | | Changer la lecture par l'appel à une méthode de lecture du canal
_N;
| | | | | | Si l'instruction _Ii fait une écriture à la variable _Vi;
| | | | | | | Changer l'écriture par l'appel à une méthode d'écriture du canal
_N;
| | Enlever de _UC l'appel d'instance _IUC;
| | Si l'unité _UCC n'est pas une instance d'autres unités;
| | | Enlever l'unité _UCC de la spécification;
```

Primitive *Merge* Structurelle

La primitive *Merge* structurelle groupe des unités de conception en un nouveau niveau de hiérarchie. Ces unités deviennent des instances de cette nouvelle unité de conception structurelle. La principale transformation de la description est l'adaptation des ports et des canaux de communication à cette nouvelle structure. Si le port ou le canal est utilisé seulement par les unités de conception qui ont subi l'opération de *Merge*, cette interconnexion est déplacée dans la nouvelle hiérarchie. Si la porte ou le canal est aussi utilisé par des unités externes au groupe qui a subi le *Merge*, des nouvelles interconnections avec les deux niveaux de hiérarchie sont créées.

```
Merge(Appel d'Instance _IUC1, Appel d'Instance _IUC2, ...)
| _UC est l'unité de conception qui contient les appels d'instances _IUCi;
| Créer une unité de conception structurelle _UCS;
| Pour chaque instance _IUCi de la commande Merge();
| | Déplacer l'appel d'instance _IUCi de _UC à _UCS;
| | Copier les portes et canaux d'interface de _IUCi à _UCS;
| Pour chaque signal ou canal _ERi du réseau de _UC;
| | Vérifier si _ERi est lié à des instances _IUCi;
| | Si tous les accès de _ERi sont liés aux instances _UICi;
| | | Déplacer le signal ou canal _ERi de _UC à _UCS;
| | | Déplacer l'accès _ACI de l'interface de _UC à l'interface de _UCS;
| | Si des accès _AERi de _ERi sont liés aux instances _UICi;
| | | Copier _ERi de _UC à _UCS sous le nom de _ERNi;
| | | Faire le lien de _ERi avec l'interface de _UC;
| | | Pour chaque accès _ACi de _ERi;
| | | | Si l'accès _ACi est connecté à des instances _IUCi;
| | | | | Enlever l'accès _ACi;
| | | Faire le lien de _ERNi avec l'interface de _UCS;
| | | Pour chaque accès _ACNi de _ERNi;
| | | | Si l'accès _ACNi est connecté à des instances _IUCi;
| | | | | Enlever l'accès _ACNi;
| | | Ajouter l'accès _ACI à l'interface de _UCSi;
| Ajouter l'appel d'instance à l'unité _UCS dans la liste des instances de _UC;
```

Primitive *Cluster* Structurelle

La primitive *Cluster* structurelle transforme une unité de conception, composée d'un ensemble d'unités comportementales, en une nouvelle unité comportementale. Le comportement de l'unité générée contient un ensemble d'états parallèles représentant le comportement des unités qui ont subi l'opération de *Cluster*. La principale transformation de la description est la conversion des signaux internes de l'unité qui a subi le *Cluster* en variables internes au comportement de l'unité générée.

```
Cluster(Appel d'Instance _IUC)
| _UC est l'unité de conception qui contient l'appel d'instance _IUC;
| _UCS est l'unité de conception qui _IUC fait appel;
| Créer une unité de conception comportementale _UCC;
| Créer un état parallèle _EP dans l'unité _UCC;
| Copier l'interface de _UCS à _UCC;
| Pour chaque instance _IUCi de _UCS;
| | Copier le comportement de _IUCi dans un état parallèle de _EPl sous le nom
de _COi;
| | Copier les variables et constantes de _IUCi a _UCC;
| Pour chaque signal ou canal _Ni du réseau de _UCS;
| | Si _Ni est interne à l'unité _UCS;
| | | Créer une variable globale _VP dans _EP du type _Ni;
| | | Pour chaque état _COi;
| | | | Remplacer les accès _Ni par des accès à _VP;
| Remplacer l'appel d'instance _IUC par l'appel à l'unité _UCC dans _UC;
| Si l'unité _UCS n'est pas une instance d'une autre unité;
| | Enlever l'unité _UCS de la spécification;
```

Primitive *Move* Structurelle

La primitive *Move* structurelle déplace une unité de conception dans la hiérarchie de la description. La principale transformation de la description est l'adaptation des interfaces et du réseau d'interconnexion à cette nouvelle structure.

```
Move(Appel d'Instance _IUCS, Appel d'Instance _IUCD)
| _UC est l'unité de conception qui contient l'appel d'instance _IUCS;
| _UD est l'unité de conception qui contient l'appel d'instance _IUCD;
| Déplacer l'appel d'instance de _UC à _UD;
| Pour chaque signal ou canal _Ni du réseau de _UC;
| | Pour chaque accès _ACi de _Ni;
| | | Si _ACi est lié à l'unité _UCS;
| | | | Enlever l'accès _ACi;
| | | | Si _Ni n'est pas connecté à l'interface de _UC;
| | | | | Connecter _Ni à l'interface de _UC;
| | | | Pour chaque niveau de la hiérarchie _NHi entre _UC et _UD;
| | | | | Créer une copie de _Ni dans le niveau _NHi;
| | | | Créer une copie de _Ni dans le réseau de _UD;
| | | | Créer un accès entre _Ni et _UCD;
```

Primitive *Flat* Structurelle

La primitive *Flat* structurelle met à plat la hiérarchie d'une unité de conception. La principale transformation de description est l'adaptation des interfaces et du réseau d'interconnexion à cette nouvelle structure.

```
Flat(Appel d'Instance _IUC)
| _UC est l'unité de conception qui contient l'appel d'instance _IUC;
| _UCS est l'unité de conception qui _IUC fait appel;
| Si _UCS est une unité de conception structurelle;
| | Copier tous les appels d'instance de _UCS à _UC;
| | Copier les constantes et les variables de _UCS à _UC;
| | Pour chaque signal ou canal _Ni du réseau de _UCS;
| | | Si _Ni est interne à _UCS;
| | | | Copier _Ni au réseau de _UC;
| | Pour chaque signal ou canal _Ni du réseau de _UC;
| | | Pour chaque accès _ACi de _Ni;
| | | | Si _ACi est lié à l'unité _UCS;
| | | | | Pour chaque signal ou canal _NNi du réseau de _UCS;
| | | | | | Si _NNi est lié à _ACi;
| | | | | | | Copier tous les accès de _NNi à _Ni
| | | | | Effacer l'accès _ACi;
| | Enlever l'appel d'instance _IUC de _UC;
| | Si l'unité _UCS n'est pas une instance d'une autre unité;
| | | Enlever l'unité _UCS de la spécification;
```

Primitive *Map* Structurel

La primitive *Map* Structurel réalise l'assignation des propriétés relatives à la réalisation matérielle ou logicielle.

```
Map(Appel d'Instance _IUC, réalisation _I)
| _UC est l'unité de conception qui contient l'appel d'instance _IUC;
| Si _I est du type matériel;
| | Ajouter la propriété qui indique réalisation matérielle;
| Si _I est du type logiciel;
| | Ajouter la propriété qui indique réalisation logicielle;
```

Appendice B

Paramètres d'Activation des Outils du Système

Cet appendice présente les outils qui font partie de la méthodologie Cosmos. Cosmos utilise des outils internes, développés par le groupe, et des outils externes, développés dans le domaine industriel. Les outils internes sont: Cosmos, Sas, Partif, Estim, PuTF, GeTV et S2cv. Les outils externes font partie des environnements: ObjectGeode, Gnu et Synopsys.

L'utilisation de l'environnement graphique Cosmos réalise automatiquement, de façon transparente à l'utilisateur, les appels aux outils. Mais l'utilisateur a la possibilité d'activer manuellement les outils en dehors de l'environnement graphique. Cette activation manuelle est utile dans les cas suivants:

- quand l'utilisateur veut réaliser automatiquement les transformations avec l'utilisation d'un fichier de commande du type *Unix* (*shell script*), comme illustre l'appendice D;
- quand les outils sont utilisés sur une machine non compatible avec l'interface graphique;
- quand l'utilisateur est en train de développer un nouvel outil qui n'est pas encore intégré dans l'environnement.

Outils internes

1. Outil COSMOS

L'outil COSMOS représente l'environnement graphique du système. L'environnement graphique contient l'interface Solar et permet la transformation rapide d'une spécification décrite en SDL en une architecture distribuée C/VHDL.

- Dernière version: 2.3
- Fichier d'entrée : SDL (.pr) ou Solar (.solar)
- Fichier de sortie : code C (.c) et code VHDL (.vhdl)
- Activation : COSMOS <nom du fichier SDL ou Solar>

2. Outil SAS

L'outil SAS (SDL à Solar) réalise la conversion du format SDL au format Solar.

- Dernière version: 1.2
- Fichier d'entrée : SDL (.pr)
- Fichier de sortie : Solar (.solar)
- Activation : SAS <nom du fichier SDL>

3. Outil PARTIF

L'outil PARTIF réalise le découpage matériel/logiciel de la spécification du système.

- Dernière version: 2.0
- Fichier d'entrée : Solar (.solar) et bibliothèque de canal (channel.lib)
- Fichier de sortie : Solar (.solar)
- Activation : PARTIF <nom du fichier Solar d'entrée> <nom du fichier Solar de sortie> <code du domaine> <primitive> <paramètres>

Transformations:

- Décomposition Fonctionnelle
  - Domaine : Tables d'états
  - Code du domaine : FD
  - Primitives et paramètres:
    FLAT <nom de la machine>
    SPLIT <nom de la machine>
    MOVE <nom de la machine source> <nom de la machine destination>
    MERGE <nom de la machine> <nom de la machine>
    OK
- Réorganisation de la Structure
  - Domaine : Unités de conception
  - Code du domaine: SR
  - Primitives et paramètres:
    FLAT <nom de l'unité de conception>

SPLIT <nom de l'unité de conception>
MERGE <nom de l'unité de conception> <nom de l'unité de conception>...
SOFT <nom de l'unité de conception>
HARD <nom de l'unité de conception>
OK

- Synthèse de la Communication
  - Domaine: Unités de communication
  - Code du domaine: CS
  - Primitives et paramètres:
    MAP <nom du canal> <nom du protocole>
    MERGE <nom du canal> <nom des canaux>
    OK

4. Outil ESTIM

L'outil ESTIM réalise le calcul de performance et de la taille de la réalisation.

- Dernière version: 1.0
- Fichier d'entrée : Solar (.solar), bibliothèque de processeurs (sw.lib), bibliothèque de caractérisation des réalisations (hw.lib) et bibliothèque de mémoires (memory.lib)
- Fichier de sortie : Solar (.solar)
- Activation : ESTIM <nom du fichier d'entrée> <nom du fichier de sortie> <primitive> <paramètres>
  - Primitives et paramètres:
    MAP <nom de la machine>
    SOFT <nom de l'unité de conception>
    HARD <nom de l'unité de conception>
    OK

5. Outil S2CV

L'outil S2CV (Solar *to* C/Vhdl) réalise la génération des modèles C et VHDL de la réalisation.

- Dernière version: 3.03
- Fichier d'entrée : Solar (.solar)
- Fichier de sortie : code C (.c), code VHDL (.vhdl), fichier d'inclusion (.h) et fichier de configuration (.vci)
- Activation : s2cv -f <nom du fichier Solar>

6. Outil PuTF

L'outil PuTF (*Put Trace Flags*) réalise l'addition des indicateurs de trace à la spécification initiale, décrite en SDL.

- Dernière version: 1.0
- Fichier d'entrée : SDL (.pr)
- Fichier de sortie : SDL (.pr)
- Activation : PuTF <nom du fichier SDL> <nom des fichiers SDL>

7. Outil GeTV

L'outil GeTV (*Get Trace Values*) récupère les valeurs de trace générées par la simulation fonctionnelle pendant le prototypage, et réalise l'annotation de la spécification SDL.

- Dernière version: 1.0
- Fichier d'entrée : SDL (.pr)
- Fichier de sortie : SDL (.pr)
- Activation : GeTV <nom du fichier SDL> <nom des fichiers SDL>

Outils Externes

1. Outil geodedit

L'outil GEODEDIT (Verilog [Obj97]) est un éditeur graphique du langage SDL.

- Dernière version: 3.01
- Fichier de sortie : SDL (.pr)
- Activation : geodedit

2. Outil gsmcomp

L'outil GSMCOMP (Verilog) est le compilateur du langage SDL. Le compilateur permet la génération du code C et la détection des erreurs syntaxiques et sémantiques de la spécification.

- Dernière version: 2.22
- Fichier d'entrée : SDL (.pr)
- Fichier de sortie : fichier de simulation (.sym)
- Activation : gsmcomp <fichier SDL>

3. Outil geodesim

L'outil GEODESIM (Verilog) est le simulateur fonctionnel du langage SDL. Cet outil permet la validation de la spécification en utilisant des techniques formelles de vérification.

- Dernière version: 2.22
- Fichier d'entrée : fichier de simulation (.sym)
- Fichier de sortie : graphe de flot de message (.msc)
- Activation : geodesim <fichier de simulation>

4. Outil gcc

L'outil GCC (Gnu) est le compilateur du langage C, utilisé pendant les étapes de co-simulation et de prototypage.

- Fichier d'entrée : code C (.c)
- Fichier de sortie : code exécutable
- Activation : gcc <code C>

5. Outil xxgdb

L'outil XXGDB (gnu) est le debugger du code C utilisé pendant l'étape de co-simulation.

- Fichier d'entrée : code exécutable
- Activation : xxgdb <code exécutable>

6. Outil vhdlan

L'outil VHDLAN (Synopsis) est le compilateur du code VHDL.

- Fichier d'entrée : code VHDL (.vhdl)
- Fichier de sortie : fichier de simulation (.sym)
- Activation : vhdlan <code VHDL>

7. Outil vhdldbx

L'outil VHDLDBX (Synopsis) est le débougueur de la spécification VHDL.

- Fichier d'entrée : fichier de simulation (.sym)
- Activation : vhdldbx <fichier de simulation>

Exemple de fichier de commandes

La liste ci-dessous contient un exemple de fichier de commandes qui fait le lien entre une spécification SDL et une architecture C/VHDL. L'exemple utilisé est le *Robot Arm Controller* (section 4.8.2). Chaque commande décrit les fichiers sources, le fichier destination et les paramètres auxiliaires. Dans cet exemple, les outils d'estimation et les outils de calcul de la fréquence d'exécution ne sont pas utilisés, car ils nécessitent l'intervention de l'utilisateur.

Fichier de commandes

```
# SAS controleur
# ; Conversion des formats SDL à Solar du fichier contrôleur
# PARTIF controleur ctrl_1 SR FLAT ALL
# ; Met à plat la structure hiérarchique du système
# PARTIF ctrl_1 ctrl_2 CS MERGE N_software N_control
# ; Groupement des canaux abstraits N_software et N_control
# PARTIF ctrl_2 ctrl_3 CS MAP N_merge1 FIFO
# ; Allocation et mappage du protocole FIFO au canal N_merge1
# PARTIF ctrl_3 ctrl_4 CS MAP N_algo1 RENDEZ_VOUS
# PARTIF ctrl_4 ctrl_5 CS MAP N_algo2 RENDEZ_VOUS
# ; Allocation et mappage du protocole RENDEZ_VOUS aux canaux
# PARTIF ctrl_5 ctrl_6 SR MERGE pc_I algo1_I algo2_I
# PARTIF ctrl_6 ctrl_7 SR MERGE pid_I moteur1_I moteur2_I
# ; Groupement des unités qui vont être exécutées sur un processeur
# PARTIF ctrl_6 ctrl_7 SR SOFTWARE merge1_I
# PARTIF ctrl_9 ctrl_10 SR HARDWARE merge2_I
# ; Assignation des options de réalisation logicielle/matérielle
```

```
# S2CV –f ctrl_10
# ; génération des fichiers C/VHDL
```

Appendice C

Bibliothèques du Système Cosmos

Cet appendice présente la syntaxe des bibliothèques du système Cosmos. Cosmos utilise quatre bibliothèques internes:

- La **bibliothèque des canaux de communication** qui contient les protocoles de communication utilisés pendant l'étape de synthèse de la communication;
- La **bibliothèque de caractérisation des processeurs logiciels** utilisée pendant les étapes de transposition de l'architecture et d'estimation du logiciel;
- La **bibliothèque de caractérisation des réalisations matèrielles** utilisée pendant les étapes de transposition de l'architecture, de synthèse comportementale et d'estimation RTL;
- La **bibliothèque de caractérisation des mémoires** utilisée pendant les étapes de transposition de l'architecture et d'estimation du logiciel.

Bibliothèque des canaux de communication

La syntaxe de la bibliothèque de communication obéit à la syntaxe Solar. La bibliothèque est composée d'une classe principale (classe Solar) et d'un ensemble de canaux de communication. Chaque canal est représenté par son interface, ses méthodes d'accès et son protocole de communication, défini dans la table d'états. Les principaux éléments de cette bibliothèque sont:

```
Communicationlibrary:(SOLAR <name> Protocol)

Protocol: (CHANNELUNIT <name> Description) ↻

Description: (VIEW <name> Interface Contents) ↻

Interface: (INTERFACE Ports Methods)

Ports: (PORT Signal Direction Type) ↻

Methods: (METHOD <name> MethodContents) ↻

Contents: (CONTENTS Variables Constants Instances Net StateTable)
```

Bibliothèque des processeurs logiciels

Chaque élément de la bibliothèque de caractérisation des processeurs est décrit par ses caractéristiques générales, la modélisation de ses instructions génériques, la modélisation de ses modes d'adressage et de ses types de données. La modélisation des instructions génériques fait la relation entre les instructions génériques du modèle Cosmos et l'ensemble des instructions qui font partie de l'architecture du processeur. Cette bibliothèque a la forme suivante:

```
Processorlibrary:(PROCESSORSLIBRARY <name> Processor)

Processor: (PROCESSOR <name> Characterization) ↻

Characterization:
       (CLOCKFREQUENCY <value>)
       (BUSDATAWIDTH <value>)
       (BUSADRESSWIDTH <value>)
       (REGISTERS <value>)
       (PINS <value>)
       (PIPELINESTAGES <value>)
       (OPTIMIZATIONRATE <value>)
       (INSTRUCTIONS Instruction)
       (ADDRESSINGMODES Addressmode)
       (DATATYPES Datatypes)

Instructions :
       (MOV <size> <temps> <temps>)
       (ADD/SUB <size> <temps> <temps>)
       (MULT <size> <temps> <temps>)
       (DIV <size> <temps> <temps>)
       (AND/OR <size> <temps> <temps>)
       (JMP <size> <temps> <temps>)
       (BEQZ <size> <temps> <temps>)
       (JAL <size> <temps> <temps>)
       (RFE <size> <temps> <temps>)

addressmode:
       (REGISTER <size> <temps>)
       (IMMEDIATE <size> <temps> )
       (DISPLACEMENT <size> <temps>)
       (INDEXED <size> <temps>)

datatype:
       (INTEGER <size>)
       (FLOAT <size>)
       (BOOLEAN <size)
```

```
<temps>: value
<size> : value
<name> : 0-9 a-z A-Z _ ↻
<value>: 0-9 ↻
```

Bibliothèque des réalisations matérielles

La bibliothèque de caractérisation des technologies matérielles a la forme suivante [Din96]:

```
Technologylibrary :(TECHNOLOGY_FILE Parameter Constraint Factor)

Parameter: (PARAMETER DataPathCells) ↻

DataPathCells :
       (CONSTANT_REGISTER Param_values)
       (FLAG_REGISTER Param_values)
       (VARIABLE_REGISTER Param_values)
       (EXTERNAL_REGISTER Param_values)
       (SWITCH Param_values)
       (MUX Param_values)
       (BUS (HEIGHT <value>)(POWER <value>)(MAX_DELAY <value>))

Param_values:
       (WIDTH <value>)(HEIGHT <value>)(AREA <value>)(POWER <value>)
       (MAX_DELAY <value>)

Constraint :
       (CONSTRAINT (MAX_MUX <par>)(MAX_MICRO <par>)(MAX_BUS_CHANNEL <par>)
       (MAX_FU <par>)(MAX_SWITCH <par>)(MAX_WIDTH <par>)(MAX_WEIGHT <par>)
       (MAX_AREA <par>)(MAX_POWER <par>)) ↻

par: <value> (WEIGHT <value>)

Factor : (FACTOR (FDATAPATH <value>)(FCONTROLLER <value>)
         (FCIRCUIT <value>)(TR2AREA <value>)(SWPOWER <value>)) ↻

<value>      : 0-9.0-9 ↻
```

Bibliothèque de caractérisation des mémoires

La bibliothèque de caractérisation des mémoires a la forme suivante:

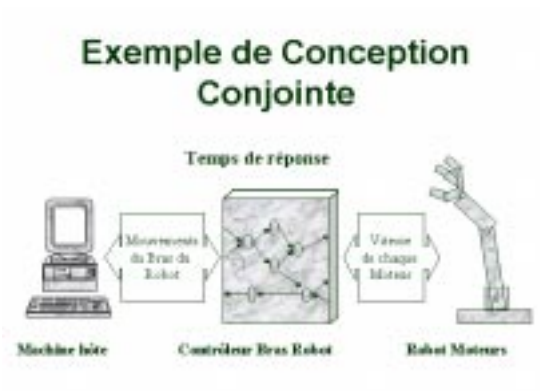
```
Memorylibrary:(MEMORYLIBRARY <name> Memory)

Memory: (MEMORY <name> Characterization) ↻

Characterization:
       (CAPACITY <value>)
       (ACCESSTIME <value>)
       (WORDLENGTH <value>)
       (COST <value>)
```

Appendice D

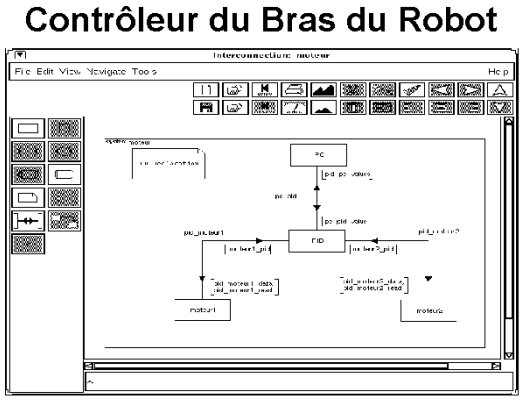
Exemple de Conception Conjointe

Cet appendice présente les étapes de conception conjointe appliquées à un exemple: le contrôleur de bras de robot. L'exemple commence avec une spécification au niveau système, parcourt les étapes de conception de la méthodologie Cosmos et arrive à une architecture C/VHDL distribuée. Ci-dessous nous trouvons le commentaire de chaque étape du raffinement suivi par les figures correspondantes.

Commentaire des Etapes:



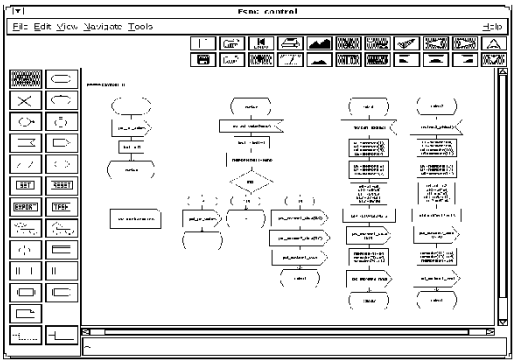
1. Le Contrôleur de Bras de Robot est un système qui reçoit, d'une machine hôte, les informations liées au mouvement que le bras du robot doit exécuter. Ce système ajuste la variation de vitesse de chaque moteur de façon à ce que le mouvement du bras soit souple et que les contraintes physiques d'accélération et de freinage soient respectées;



2. La figure montre la structure des processus qui composent le contrôleur de bras du robot. L'utilisateur dispose de

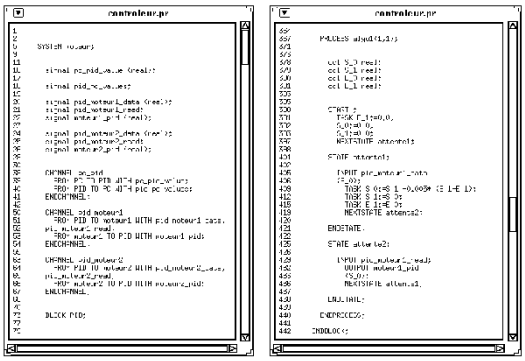
l'environnement *Geode* pour réaliser l'entrée de la spécification fonctionnelle en SDL. Avec *Geode* il est relativement facile de décrire le système graphiquement et de valider sa fonctionnalité. La figure représente le contrôleur, décrit par quatre processus interconnectés. Le processus *PC* réalise l'interface entre le contrôleur et la machine hôte. Le processus *PID* exécute un algorithme de contrôle *PID*. Les processus *moteur1* et *moteur2* réalisent l'interface entre le contrôleur et les moteurs du bras de robot. La version complète du contrôleur est composée de seize instances du processus moteur, au lieu des deux instances présentées dans cet exemple. Cette simplification est destinée à améliorer la clarté de l'exemple. Si nous regardons la machine d'états, qui décrit le comportement du processus PID, nous apercevons un ensemble d'états et de commandes représentés graphiquement;

Comportement du Processus PID



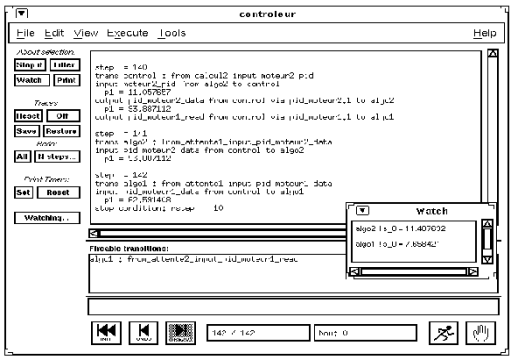
3. Dans cette représentation, chaque symbole graphique représente une commande, comme par exemple l'envoi ou la réception de messages, des conditions ou des boucles. Sur la figure, l'état initial (à gauche) envoie un message, réalise une commande et passe le contrôle à un nouvel état. Cet état attend l'arrivée d'un message pour activer la séquence de commandes qui suit. Tous les éléments qui forment la représentation graphique ont une traduction textuelle en SDL;

Format SDL Textuel



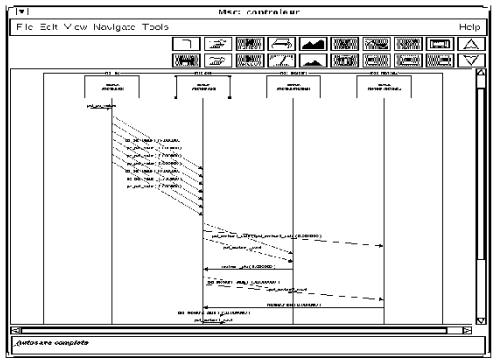
4. Cette figure représente la spécification du système décrit en format *SDL* textuel. Le parser SDL de Geode dispose d'une API permettant à des outils externes un traitement facile d'une spécification SDL;

Simulation Fonctionnelle en SDL



5. La validation fonctionnelle de la spécification utilise le simulateur GeodeSim. Le simulateur exécute pas à pas la spécification et permet la vérification du comportement du système, tel que l'état, des variables, les échanges de messages. L'échange de messages entre les processus peut être représenté de façon plus claire à travers les diagrammes de séquence de messages (*MSC*);

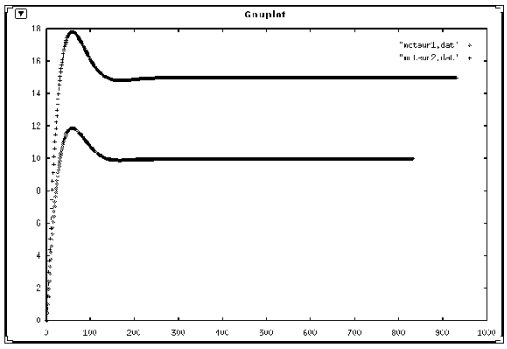
Diagramme Echange Messages



6. Les *MSC* permettent de représenter l'échange de messages entre processus. Sur

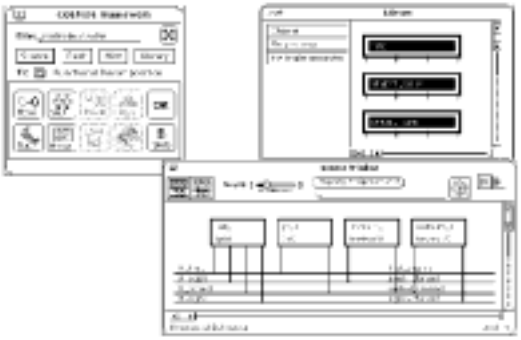
la figure, les processus sont représentés par des boîtes. L'historique de simulation est représenté par les lignes verticales et les messages échangés par des flèches diagonales. Chaque message exhibe les données associées;

Résultat de la Simulation Fonctionnelle



7. Cette courbe représente graphiquement les informations relatives à la vitesse de chaque moteur en fonction du temps, générées par le simulateur. La vitesse du moteur 1 s'est stabilisée en 15 et la vitesse du moteur 2 en 10 pas de simulation. Les vitesses obéissent à la courbe d'accélération produite par le processus PID.

Environnement de Conception Conjointe

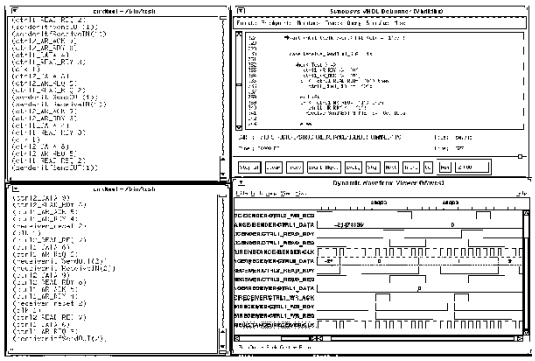


8. La première étape est la conversion du modèle SDL en Solar. La représentation graphique Solar du contrôleur de moteur apparait sur la figure. Les quatre processus de la description d'origine, interconnectés par des canaux de communication, sont représentés. Il est possible de parcourir la structure de la spécification jusqu'à un niveau feuille, qui décrit le comportement des processus. Pour le raffinement de la spécification l'utilisateur dispose de trois ensembles de transformations: la décomposition de la fonctionnalité, la réorganisation de la structure et la transformation de la communication. Dans cet exemple, le raffinement commence par la mise à plat de la hiérarchie de la structure en utilisant la primitive Flat Structurelle. L'utilisateur peut, à ce moment, attribuer une réalisation aux canaux abstraits. En premier lieu, l'utilisateur choisit le canal à réaliser, par exemple le canal assurant la communication entre les processus PC et PID. Il peut ensuite choisir dans la bibliothèque de communication un protocole adapté à ce canal. Ce choix peut être effectué manuellement ou automatiquement. Le résultat de la synthèse de la communication est un système composé par des processus qui communiquent par des signaux bien définis. A la suite du raffinement vient la génération des codes C et VHDL.

Génération du Code C/VHDL

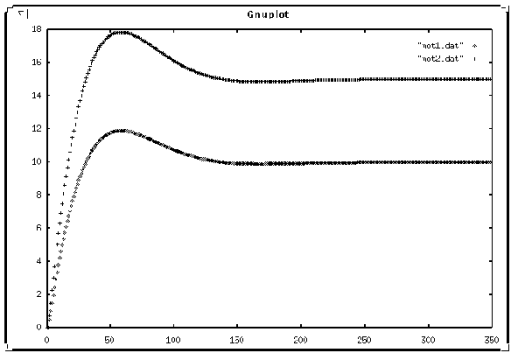
9. Les codes C et VHDL réalisent exactement le comportement qui a été spécifié initialement, plus les transformations réalisées. Par exemple, le code correspondant au modèle d'échange de messages de la spécification initiale a été remplacé par des appels de procédures. Ces procédures réalisent les échanges de signaux du protocole. A ce moment vient la co-simulation C/VHDL permettant valider les transformations réalisées.

Co-Simulation C/VHDL



10. Grâce au simulateur VHDL de synopsys il est possible d'exécuter pas à pas la description matérielle et de la même façon, avec le débugger de code C, le logiciel. La co-simulation valide le système à un niveau plus détaillé que le niveau système, mais elle nécessite une attention plus grande et un temps de simulation plus long.

Résultat de la Co-Simulation



11. La figure représente graphiquement les informations de vitesse obtenues par co-simulation. On constate que la courbe obtenue est identique à celle donnée par la simulation système.

Appendice E

Structure du Format Intermédiaire Solar

Cet appendice présente, sous forme graphique, les principales classes qui composent la structure du format intermédiaire Solar. Cette représentation met en évidence les relations existant entre les classes et elle est extrêmement utile pour la compréhension de la structure Solar. L'utilisateur a fréquemment besoin d'accéder à la structure des classes pendant la programmation des outils qui manipulent le format intermédiaire. Le format d'exhibition des classes, présenté dans cet appendice, détaille les aspects d'héritage, de relation et les méthodes disponibles.

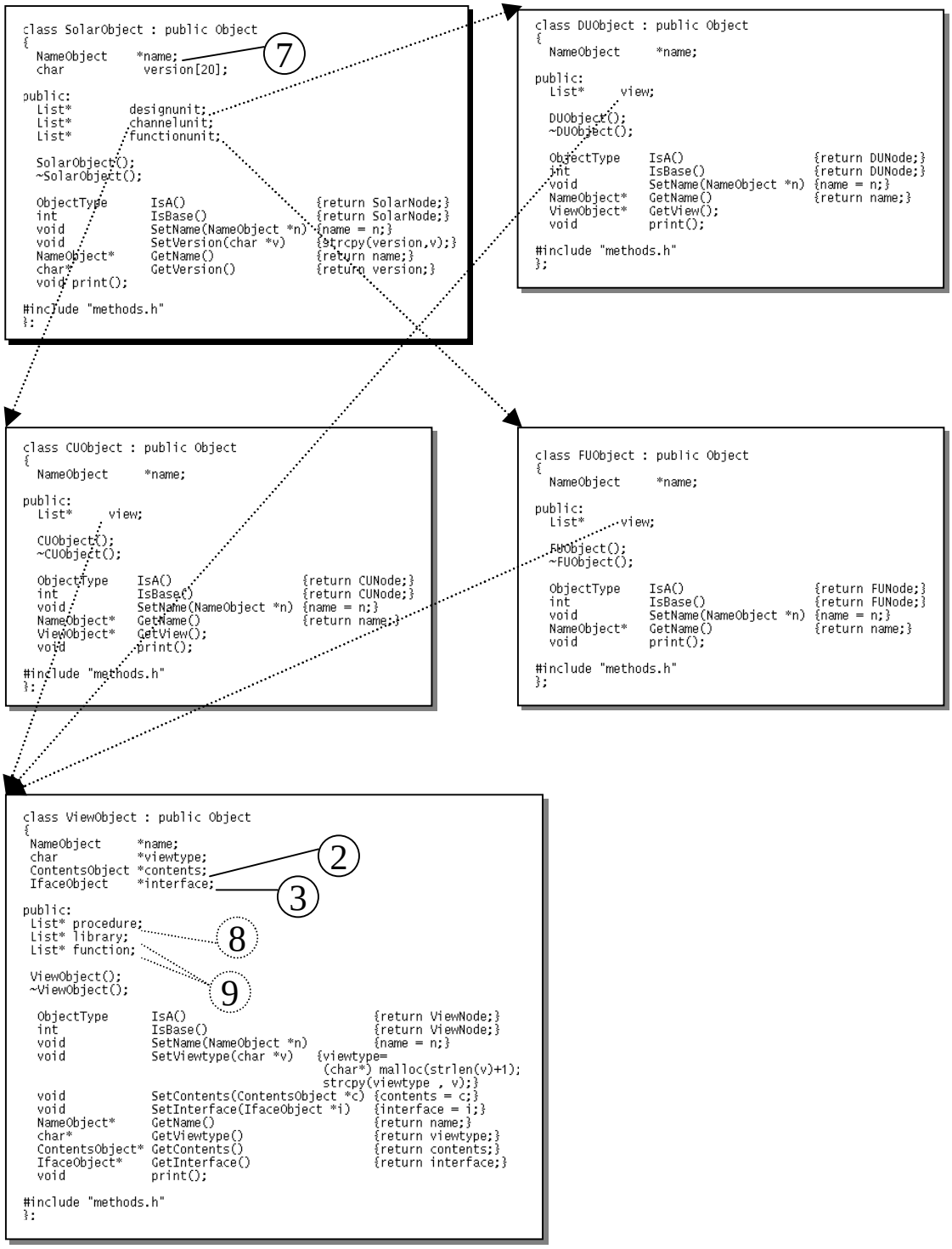
La représentation graphique montre la structure des classes à partir de la racine de la hiérarchie (SolarObject) jusqu'au comportement (STObject et StateObject). Les classes qui décrivent les instructions du comportement des unités (par exemple : IfObject, AssignObject, PcallObject) ne font pas partie de cet appendice puisqu'elles ont une structure simple et facile à comprendre.

La structure de données Solar est basée sur une approche orientée objets. L'orientation objet a été choisie afin de permettre une adaptation facile aux différents outils du système et pour faciliter les futures extensions de Solar. À chaque entité du modèle Solar correspond une classe qui groupe les informations liées à l'entité.

Dans les pages qui suivent, la représentation de la hiérarchie des classes Solar obéit aux régles suivantes :

- chaque boîte contient la description d'une classe Solar (i.e. un objet);
- quand un attribut d'une classe fait référence à une autre classe Solar, une flèche est utilisée pour indiquer où se trouve cette classe référencée;

- quand un attribut d'une classe fait référence à une liste de classes (ListObject), une flèche pointille est utilisée pour indiquer où se trouve la classe élément de la liste ;
- quand la classe référencée se trouve en dehors de la page actuelle, la page qui contient la référence est indiquée sous le format d'un connecteur.



(1)

```
class ContentsObject : public Object
{
public:
  List*   foreign;
  List*   statetable;
  List*   variable;
  List*   constant;
  List*   instance;
  List*   net;
  List*   typedefi;

  ContentsObject();
  ~ContentsObject();

  ObjectType IsA()    {return ContentsNode;}
  int        IsBase() {return ContentsNode;}
  void       print();

#include "methods.h"
};
```

4

```
class ForeignObject : public Object
{
  NameObject *language;
  char  filename[50];

public:

  ForeignObject();
  ~ForeignObject();

  ObjectType      IsA()                      {return ForeignNode;}
  int             IsBase()                   {return ForeignNode;}
  void            SetLanguage(NameObject *n){language = n;}
  void            SetFilename(char *v)       {strcpy(filename,v);}
  NameObject*     GetLangage()               {return language;}
  char*           GetFilename()              {return filename;}
  void print();

#include "methods.h"
};
```

```
class InstanceObject : public Object
{
  NameObject      *name;
  ViewrefObject   *viewref;
 public:
  List  *portinst;
  List  *accinst;
  List  *paramass;

  InstanceObject();
  ~InstanceObject();

  ObjectType      IsA()                      {return InstanceNode;}
  int             IsBase()                   {return InstanceNode;}
  void            SetName(NameObject *n)     {name = n;}
  void            SetViewref(ViewrefObject *v){viewref = v;}
  NameObject*     GetName()                  {return name;}
  ViewrefObject*  GetViewref()               {return viewref;}
  void            print();

#include "methods.h"
};
```

5

9

8

```
class NetObject : public Object
{
  NameObject     *name;
  JoinedObject   *joined;

public:
  NetObject();
  ~NetObject();

  ObjectType    IsA()                        {return NetNode;}
  int           IsBase()                     {return NetNode;}
  void          SetName(NameObject *n)       {name = n;}
  void          SetJoined(JoinedObject *j){joined = j;}
  NameObject*   GetName()                    {return name;}
  JoinedObject* GetJoined()                  {return joined;}
  void print();

#include "methods.h"
};
```

5

```
class VarObject : public Object
{
  ValueObject    *value;
  TypedvalObject *typedval;
  InitvalObject  *initialvalue;

public:

  VarObject();
  ~VarObject();

  ObjectType IsA()    {return VarNode;}
  int IsBase() {return VarNode;}
  void            SetValue(ValueObject *v)   {value = v;}
  void            SetTval(TypedvalObject *t) {typedval = t;}
  void            SetInitVal(InitvalObject *o) { initialvalue = o;}
  ValueObject*    GetValue()                 {return value;}
  TypedvalObject* GetTval()                  {return typedval;}
  InitvalObject * GetInitVal()               {return initialvalue;}
  void print();

#include "methods.h"
};
```

```
class ConstObject : public Object
{
  ValueObject    *value;
  TypedvalObject *typedval;
  InitvalObject  *initialvalue;

public:
  ConstObject();
  ~ConstObject();

  ObjectType IsA()    {return ConstNode;}
  int IsBase() {return ConstNode;}
  void            SetValue(ValueObject *v)   {value = v;}
  void            SetTval(TypedvalObject *t) {typedval = t;}
  void            SetInitVal(InitvalObject *o) { initialvalue = o;}
  ValueObject*    GetValue()                 {return value;}
  TypedvalObject* GetTval()                  {return typedval;}
  InitvalObject * GetInitVal()               {return initialvalue;}
  void print();

#include "methods.h"
};
```

```
class InitvalObject : public Object
{
  ValueObject    *value;

public:

  InitvalObject() ;
  ~InitvalObject() { };

  ObjectType IsA()    {return InitvalNode;}
  int IsBase()        {return InitvalNode;}
  void            SetValue(ValueObject *v) {value = v;}
   ValueObject*  GetValue()                {return value;}
   void print();

#include "methods.h"
};
```

```
class TypedvalObject : public Object
{
public:

  TypedvalObject() {}
  ~TypedvalObject() {}

  ObjectType   IsA()    {return TypedvalNode;}
  int          IsBase() {return TypedvalNode;}
  virtual void print()  { }

#include "methods.h"
};
```

(2)

```
class IfaceObject : public Object
{

public:
  List*   access;
  List*   port;
  List*   method;
  List*   parameter;
  List*   operation;

  IfaceObject();
  ~IfaceObject();

  ObjectType IsA()    {return IfaceNode;}
  int        IsBase() {return IfaceNode;}
  void print();

#include "methods.h"
};
```

```
class AccessObject : public Object
{
  NameObject     *name;
  ViewrefObject  *viewref;

public:
  AccessObject();
  ~AccessObject();

  ObjectType      IsA()                        {return AccessNode;}
  int             IsBase()                     {return AccessNode;}
  void            SetName(NameObject *n)       {name = n;}
  void            SetViewref(ViewrefObject *v){viewref = v;}
  NameObject*     GetName()                    {return name;}
  ViewrefObject*  GetViewref()                 {return viewref;}
  void print();

#include "methods.h"
};
```

```
class MethodObject : public Object
{
  NameObject      *name;

public:
  List*   statetable;
  List*   foreign;
  List*   parameter;
  List*   port;

  MethodObject();
  ~MethodObject();

  ObjectType      IsA()                        {return MethodNode;}
  int             IsBase()                     {return MethodNode;}
  void            SetName(NameObject *n){name = n;}
  NameObject*     GetName()                    {return name;}
  void print();

#include "methods.h"
};
```

```
class ParamObject : public Object
{
  ValueObject    *name;
  TypedvalObject *typedval;
  DirObject      *direction;

public:
  ParamObject();
  ~ParamObject();

  ObjectType      IsA()                        {return ParamNode;}
  int             IsBase()                     {return ParamNode;}
  void            SetName(ValueObject *n)      {name = n;}
  void            SetTval(TypedvalObject *t){typedval = t;}
  ValueObject*    GetName()                    {return name;}
  TypedvalObject* GetTval()                    {return typedval;}
  void SetDir(DirObject *d)                    {direction = d;}
  DirObject* GetDir()                          {return direction;}
  void print();

#include "methods.h"
};
```

```
class PortObject : public Object
{
  ValueObject    *name;
  TypedvalObject *typedval;
  DirObject      *direction;

public:
  PortObject();
  ~PortObject();

  ObjectType      IsA()                        {return PortNode;}
  int             IsBase()                     {return PortNode;}
  void            SetName(ValueObject *n)      {name = n;}
  void            SetTval(TypedvalObject *t){typedval = t;}
  void            SetDir(DirObject *d)         {direction = d;}
  ValueObject*    GetName()                    {return name;}
  TypedvalObject* GetTval()                    {return typedval;}
  DirObject*      GetDir()                     {return direction;}
  void print();

#include "methods.h"
};
```

```
class OpnObject : public Object
{
  STObject        *statetable;
  ParamObject     *parameter;

public:
  OpnObject();
  ~OpnObject();

  ObjectType      IsA()                        {return OpnNode;}
  int             IsBase()                     {return OpnNode;}
  void            SetST(STObject *s)           {statetable = s;}
  void            SetParam(ParamObject *p){parameter = p;}
  STObject*       GetST()                      {return statetable;}
  ParamObject*    GetParam()                   {return parameter;}
  void print();

#include "methods.h"
};
```

```
class DirObject : public Object
{
  NameObject  *name;

public:
  DirObject() {name = 0;}
  ~DirObject();

  ObjectType      IsA()                        {return DirNode;}
  int             IsBase()                     {return DirNode;}
  void            SetName(NameObject *n){name = n;}
  NameObject*     GetName()                    {return name;}
  void print();

#include "methods.h"
};
```

(3)

```
class STObject : public Object
{
  NameObject       *name;
  RststObject      *resetstate;
  GlobactObject    *globalaction;
  DefnxtstObject   *defnextstate;
  STNmListObject   *STNames;
  EntStListObject  *entrystate;          //list of entry states

public:
  List*      Block;
  List*      variable;
  List*      constant;
  List*      typedefi;

  STObject();
  ~STObject();

  ObjectType       IsA()                          {return STNode;}
  int              IsBase()                       {return StateNode;}
  void             SetName(NameObject *n)         {name = n;}
  void             SetResetSt(RststObject *s)     {resetstate = s;}
  void             SetGlobAct(GlobactObject *g)   {globalaction = g;}
  void             SetDefNxtSt(DefnxtstObject *d) {defnextstate = d;}
  void             SetSTNames(STNmListObject *s)  {STNames = s;}
  void             SetEntSt(EntStListObject *e)   {entrystate = e;}
  NameObject*      GetName()     {return name;}
  RststObject*     GetResetSt()  {return resetstate;}
  GlobactObject*   GetGloAct()   {return globalaction;}
  DefnxtstObject*  GetDefNxtSt() {return defnextstate;}
  STNmListObject*  GetSTNames()  {return STNames;}
  EntStListObject* GetEntSt()    {return entrystate;}
  void print();

#include "methods.h"
};
```

6

2

```
class GlobactObject : public ActionObject
{
public:
  List*    simpleaction;
  GlobactObject();
  ~GlobactObject();

  ObjectType IsA()    {return GlobactNode;}
  int        IsBase() {return ActionNode;}
  void       print();

#include "methods.h"
};
```

6

```
class STNmListObject : public Object
{
public:
  List*  statenames;

  STNmListObject() {statenames = new List(ValueNode);}
  ~STNmListObject();

  ObjectType IsA()    {return STNmListNode;}
  int        IsBase() {return STNmListNode;}
  void print();

#include "methods.h"
};
```

```
class RststObject : public Object
{
  StaterefObject  *stateref;

public:
  RststObject() {stateref = 0;}
  ~RststObject();

  ObjectType      IsA()                          {return RststNode;}
  int             IsBase()                       {return RststNode;}
  void            SetStateRef(StaterefObject *s){stateref = s;}
  StaterefObject* GetStateRef()                  {return stateref;}
  void print();

#include "methods.h"
};
```

```
class EntStListObject : public Object
{
public:
  List*  statenames;

  EntStListObject() {statenames = new List(ValueNode);}
  ~EntStListObject();

  ObjectType IsA()    {return EntStListNode;}
  int        IsBase() {return EntStListNode;}
  void print();

#include "methods.h"
};
```

```
class DefnxtstObject : public Object
{
  StaterefObject  *stateref;

public:
  DefnxtstObject() {stateref = 0;}
  ~DefnxtstObject();

  ObjectType      IsA()                          {return DefnxtstNode;}
  int             IsBase()                       {return DefnxtstNode;}
  void            SetStateRef(StaterefObject *s){stateref = s;}
  StaterefObject* GetStateRef()                  {return stateref;}
  void print();

#include "methods.h"
};
```

```
class StaterefObject : public Object
{
  NameObject     *nameref;
  StaterefObject *stateref;

public:

  StaterefObject() {nameref = 0; stateref = 0;}
  ~StaterefObject();

  ObjectType      IsA()                          {return StaterefNode;}
  int             IsBase()                       {return StaterefNode;}
  void            SetNameRef(NameObject *n)      {nameref = n;}
  void            SetStateRef(StaterefObject *s){stateref = s;}
  NameObject*     GetNameRef()                   {return nameref;}
  StaterefObject* GetStateRef()                  {return stateref;}
  void print();

#include "methods.h"
};
```

(4)

```
class JoinedObject : public Object
{
public:
  List*    portref;
  List*    accessref;

  JoinedObject();
  ~JoinedObject();

  ObjectType    IsA()        {return JoinedNode;}
  int           IsBase()     {return JoinedNode;}
  int           localnet();
  int           localnet_generic();
  PortrefObject *GetExternPort();
  PortrefObject *GetExternPort_generic();
  void          print();

#include "methods.h"
};
```

```
class AccessrefObject : public Object
{
  NameObject    *name;
  ViewrefObject *viewref;
  InstrefObject *instref;

public:
  AccessrefObject();
  ~AccessrefObject();

  ObjectType      IsA()                       {return AccessrefNode;}
  int             IsBase()                    {return AccessrefNode;}
  void            SetName(NameObject *n)      {name = n;}
  void            SetViewref(ViewrefObject *v) {viewref = v;}
  void            SetInstref(InstrefObject *i) {instref = i;}
  NameObject*     GetName()                   {return name;}
  ViewrefObject*  GetViewref()                {return viewref;}
  InstrefObject*  GetInstref()                {return instref;}
  void print();

#include "methods.h"
};
```

```
class PortrefObject : public Object
{
  ValueObject    *value;
  InstrefObject  *instref;
  ViewrefObject  *viewref;

public:
  PortrefObject();
  ~PortrefObject();

  ObjectType      IsA()                       {return PortrefNode;}
  int             IsBase()                    {return PortrefNode;}
  void            SetValue(ValueObject *n)    {value = n;}
  void            SetInstref(InstrefObject *i) {instref = i;}
  void            SetViewref(ViewrefObject *v) {viewref = v;}
  ValueObject*    GetValue()                  {return value;}
  InstrefObject*  GetInstref()                {return instref;}
  ViewrefObject*  GetViewref()                {return viewref;}
  void print();

#ifdef PATVERSION
  #include "pat.h"
#endif

#ifdef SGRAPHVERSION
  #include "sgraph.h"
#endif

#ifdef S2CVVERSION
  #include "s2cv.h"
#endif

#ifdef PARTIFVERSION
  #include "partif.h"
#endif
};
```

```
class InstrefObject : public Object
{
  NameObject     *name;
  InstrefObject  *instref;
  ViewrefObject  *viewref;

public:
  InstrefObject();
  ~InstrefObject();

  ObjectType      IsA()                       {return InstrefNode;}
  int             IsBase()                    {return InstrefNode;}
  void            SetName(NameObject *n)      {name = n;}
  void            SetInstref(InstrefObject *i) {instref = i;}
  void            SetViewref(ViewrefObject *v) {viewref = v;}
  NameObject*     GetName()                   {return name;}
  InstrefObject*  GetInstref()                {return instref;}
  ViewrefObject*  GetViewref()                {return viewref;}
  void print();

#include "methods.h"
};
```

```
class ViewrefObject : public Object
{
  NameObject    *name;
  NameObject    *unitname;
  NameObject    *libname;

public:
  ViewrefObject();
  ~ViewrefObject();

  ObjectType      IsA()                 {return ViewrefNode;}
  int             IsBase()              {return ViewrefNode;}
  void            SetName(NameObject *n){name = n;}
  void            SetUnit(NameObject *n){unitname = n;}
  void            SetLib(NameObject *n) {libname = n;}
  NameObject*     GetName()             {return name;}
  NameObject*     GetUnit()             {return unitname;}
  NameObject*     GetLib()              {return libname;}
  void print();

#include "methods.h"
};
```

(5)

```
class StateObject : public Object
{
  NameObject     *name;

public:
  List*  action;

  StateObject();
  ~StateObject();

  ObjectType  IsA()                 {return StateNode;}
  int         IsBase()              {return StateNode;}
  void        SetName(NameObject *n){name = n;}
  NameObject* GetName()             {return name;}
  void print();

#include "methods.h"
};
```

```
class ParstateObject : public ActionObject
{
  GlobactObject  *globalaction;

public:
  List*          Block;

  ParstateObject();
  ~ParstateObject();

  ObjectType      IsA()                        {return ParstateNode;}
  int             IsBase()                     {return ActionNode;}
  void            SetGlobAct(GlobactObject *g) {globalaction = g;}
  GlobactObject*  GetGlobAct()                 {return globalaction;}
  void print();

#include "methods.h"
};
```

```
class ParactObject : public ActionObject
{
  GlobactObject   *globalaction;

public:
  List*            action;
  List*            state;

  ParactObject();
  ~ParactObject();

  ObjectType      IsA()    {return ParactNode;}
  int             IsBase() {return ActionNode;}
  void            SetGlobAct(GlobactObject *g){globalaction = g;}
  GlobactObject*  GetGlobAct()                 {return globalaction;}
  void            print();

#include "methods.h"
};
```

```
class GlobactObject : public ActionObject
{
public:
  List*    simpleaction;

  GlobactObject();
  ~GlobactObject();

  ObjectType IsA()    {return GlobactNode;}
  int        IsBase() {return ActionNode;}
  void       print();

#include "methods.h"
};
```

```
class SeqactObject : public ActionObject
{
  GlobactObject   *globalaction;

public:
  List*    action;

  SeqactObject();
  ~SeqactObject();

  ObjectType      IsA()                        {return SeqactNode;}
  int             IsBase()                     {return ActionNode;}
  void            SetGlobAct(GlobactObject *g){globalaction = g;}
  GlobactObject*  GetGlobAct()                 {return globalaction;}
  void print();

#include "methods.h"
};
```

(6)

```
class Object
{
public:
  Object   *parent;
  List*    property;
  List*    toolinfo;
  int      nodenum;

  Object();
  ~Object();

  virtual ObjectType IsA()    {return BadNode;}
  virtual int        IsBase() {return ObjectNode;}
  virtual void       print()  { }

#ifdef PATVERSION
  #include "pat_virtual.h"
#endif

#ifdef S2CVVERSION
  #include "s2cv.h"
#endif

#ifdef SGRAPHVERSION
  #include "sgraph-virtual.h"
#endif

#ifdef PARTIFVERSION
  #include "partif-virtual.h"
#endif
};
```

```
class ToolinfoObject : public Object
{
  NameObject  *name;
public:

  ToolinfoObject();
  ~ToolinfoObject();

  ObjectType  IsA()                 {return ToolinfoNode;}
  int         IsBase()              {return ToolinfoNode;}
  void        SetName(NameObject *n){name = n;}
  NameObject* GetName()             {return name;}
};
```

```
class NameObject : public ValueObject
{
public:
  int           index;     //location in symbol table
  ObjectType    idtype;

  NameObject() {}
  ~NameObject() {}

  ObjectType IsA()                {return NameNode;}
  int        IsBase()             {return ValueNode;}
  void       SetIndex(int i)      {index = i;}
  void       SetID(ObjectType i)  {idtype = i;}
  int        GetIndex()           {return index;}
  NameObject* GetName()           {return this;}
  ObjectType GetID()              {return idtype;}
  void       AddToSymbolTable(char *Name);
  int        GetSymbolTableIndex(char *Name);
  int        CreateSymbolTableIndex(char *Name);
  char    *  GetString()        { return(SymbolTable[index]); }
  void       print();

#include "methods.h"
};
```

```
class PropertyObject : public Object
{
  NameObject     *name;
  ValueObject    *valueref;
  OwnerObject    *owner;

public:
  PropertyObject();
  ~PropertyObject();

  ObjectType   IsA()                    {return PropertyNode;}
  int          IsBase()                 {return PropertyNode;}
  void         SetName(NameObject *n)   {name = n;}
  void         SetVal(ValueObject *v)   {valueref = v;}
  void         SetOwner(OwnerObject *o) {owner = o;}
  NameObject*  GetName()                {return name;}
  ValueObject* GetVal()                 {return valueref;}
  OwnerObject* GetOwner()               {return owner;}
  void         print();

#include "methods.h"
};
```

```
class ValueObject : public Object
{
public:
  ValueObject() {}
  ~ValueObject() {}

  virtual ObjectType IsA()    {return ValueNode;}
  int     IsBase()            {return ValueNode;}
  virtual int GetIndex(){ return 0; }
  virtual NameObject* GetName(){ return 0; }
  virtual void print()        { }

#ifdef PATVERSION
  #include "pat-virtual.h"
#endif

#ifdef SGRAPHVERSION
  #include "sgraph-virtual.h"
#endif

#ifdef S2CVVERSION
  #include "s2cv.h"
#endif

#ifdef PARTIFVERSION
  #include "partif-virtual.h"
#endif
};
```

```
class OwnerObject : public Object
{
public:
  List      *name;

  OwnerObject();
  ~OwnerObject();

  ObjectType   IsA()     {return OwnerNode;}
  int          IsBase()  {return OwnerNode;}
  void         print();
};
```

(7)

```
class List
{
  ObjectType    ListType;
public:
  Link*  head;

  List(ObjectType o) {ListType = o; head = 0;}
  List(ObjectType o, Object *a) {ListType=o; head=new Link(a);
head->succ=head;}
  ~List();

  void    append(Object *a);
  void    push(Object *a);
  void    concatList(List* c);
  void    adjustParent(Object *o);
  void    concatOwner(PropertyObject *o);
  void    EmptyList();
  int     Unique(Object *u);
  int     length();
  int     Belong(NameObject *b);
  int     Belong(ProcedureObject *b);
  int     Belong(PropertyObject *c);
  int     Belong(PortObject *p);
  int     Belong(AccessObject *p);
  int     Belong(AccinstObject *a);
  int     Belong(CUObject *c);
  int     BelongVar(NameObject *b);
  int     BelongCU(NameObject *b);
  Object* RemoveObj(Object *obj);
  Object* RemoveHead();
  Object* RemoveTail();
  void    RemoveElemt(Object *r);
  void    RemoveElemt(NameObject *n);
  void    RemoveElemt(PropertyObject *p);
  void    ReplaceElemt(NameObject *n1, NameObject *n2);
  void    ReplaceElemt(CUCallObject *c1, PCallObject *p2);
  void    ReplaceElemt(Object *o1,Object *o2);
  Object* ExtractHead();
  AccessObject* ExtractAccess(NameObject *elt);
  Link* GetNextLink(Link *link);
  Link* GetObjectLink(Object* object);
  void    print();
  void    ConcatList(List* c);
  void    DupList(List* c);

#include "methods.h"
};
```

```
class Link
{
  friend class List;

public:
  Link*  succ;
  Link*  pred;
  Object *e;

  Link(Object *a) {e = a; succ = 0; pred = 0;}
  ~Link();
};
```

```
class ProcedureObject : public Object
{
  NameObject      *name;

public:
  List*   statetable;
  List*   parameter;

  ProcedureObject();
  ~ProcedureObject();

  ObjectType  IsA()                  {return ProcedureNode;}
  int         IsBase()               {return ProcedureNode;}
  void        SetName(NameObject *n){name = n;}
  NameObject* GetName()              {return name;}
  void print();

#include "methods.h"
};
```

4

```
class AccinstObject : public Object
{
  NameObject      *name;

public:
  AccinstObject();
  ~AccinstObject();

  ObjectType  IsA()                  {return AccinstNode;}
  int         IsBase()               {return AccinstNode;}
  void        SetName(NameObject *n){name = n;}
  NameObject* GetName()              {return name;}
  void        print();

#include "methods.h"
};
```

```
class ParamassObject : public SimpactObject
{
  ValueObject *valnameref;
  ValueObject *expr;
  DirObject *direction;
  TypedvalObject *typedval;

public:

  ParamassObject() ;
  ~ParamassObject();

  ObjectType   IsA()                        {return ParamassNode;}
  int          IsBase()                     {return ParamassNode;}
  void         SetValNameRef(ValueObject *n){valnameref = n;}
  void         SetExpr(ValueObject *e)      {expr = e;}
  void         SetDir(DirObject *e)         {direction = e;}
  void         SetType(TypedvalObject *e)   {typedval = e;}
  ValueObject* GetValNameRef()              {return valnameref;}
  ValueObject* GetExpr()                    {return expr;}
  DirObject*   GetDir()                     {return direction;}
  TypedvalObject* GetType()                 {return typedval;}
  void         print();

#include "methods.h"
};
```

3

2

(8)

```
class FunctionObject : public Object
{
  NameObject      *name;
  TypedvalObject  *typedval;
  ReturnObject    *freturn;

public:
  List*   statetable;
  List*   parameter;

  FunctionObject();
  ~FunctionObject();

  ObjectType  IsA()                  {return FunctionNode;}
  int         IsBase()               {return FunctionNode;}
  void        SetName(NameObject *n){name = n;}
  void        SetType(TypedvalObject *t){typedval = t;}
  void        SetReturn(ReturnObject *r){freturn = r;}
  NameObject* GetName()              {return name;}
  TypedvalObject* GetType()          {return typedval;}
  ReturnObject* GetReturn()          {return freturn;}
  void print();

#include "methods.h"
} ;
```

②

```
class ReturnObject : public SimpactObject
{
  ValueObject      *nameref;

public:

  ReturnObject();
  ~ReturnObject();

  ObjectType   IsA()                      {return ReturnNode;}
  int          IsBase()                   {return ActionNode;}
  void         SetValueRef(ValueObject *n){nameref = n;}
  ValueObject*  GetValueRef()              {return nameref;}
  void         print();

#include "methods.h"
};
```

```
class PortinstObject : public Object
{
  ValueObject     *name;

public:

  PortinstObject();
  ~PortinstObject();

  ObjectType  IsA()                    {return PortinstNode;}
  int         IsBase()                 {return PortinstNode;}
  void        SetName(ValueObject *n){name = n;}
  ValueObject* GetName()               {return name;}
  void        print();

#include "methods.h"
};
```

```
class LibraryObject : public Object
{
  NameObject *language;
  char  filename[50];

public:

  LibraryObject();
  ~LibraryObject();

  ObjectType      IsA()                       {return LibraryNode;}
  int             IsBase()                    {return LibraryNode;}
  void            SetLanguage(NameObject *n) {language = n;}
  void            SetFilename(char *v)        {strcpy(filename,v);}
  NameObject*     GetLangage()                {return language;}
  char*           GetFilename()               {return filename;}
  void print();

#include "methods.h"
};
```

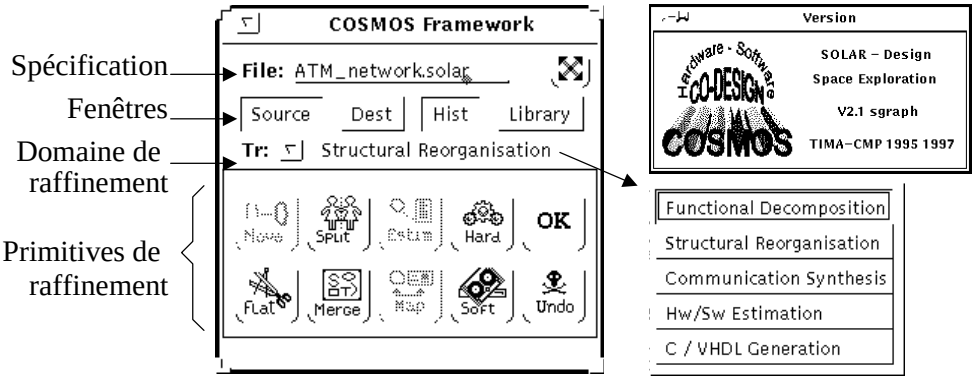
(9)

Appendice E

Fenêtres de l'Environnement Cosmos

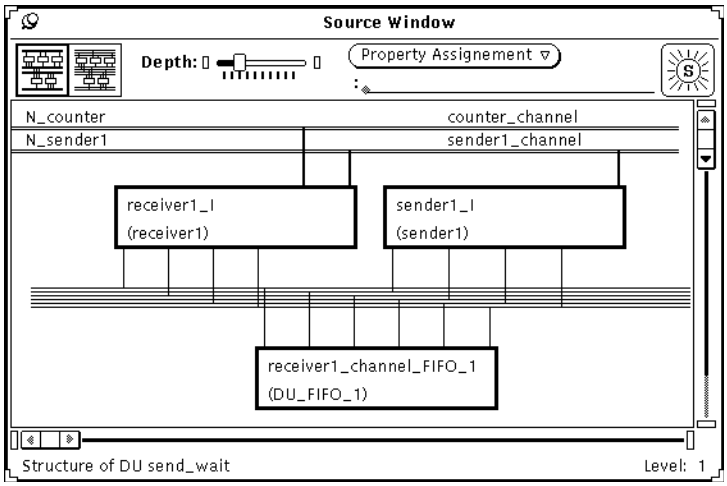
La fenêtre *Principale*

La fenêtre *Principale* de l'environnement Cosmos présente l'ensemble des primitives de raffinement. L'application de chaque primitive dépend du contexte d'utilisation de la transformation. Les contextes d'utilisation sont: la décomposition fonctionnelle, la réorganisation de la structure, la synthèse de la communication et l'estimation de performance. Les commandes et les primitives, qui ne sont pas disponibles dans le contexte de la transformation choisie, sont présentées en mode masqué. La fenêtre *Version* montre le logo du système et la version des outils. Cette fenêtre est présentée pendant quelques secondes après chaque activation de l'environnement Cosmos.



Les fenêtres *Source* et *Destination*

Les fenêtres *Source* et *Destination* présentent la structure et le comportement de la spécification Solar en format graphique. Ces deux fenêtres ont le même format de présentation. La différence est que la fenêtre *Source* présente la spécification avant l'application de la primitive de transformation. Le résultat de la transformation, aussi appelé réalisation, est présenté dans la fenêtre *Destination*. De cette façon l'utilisateur peut valider visuellement le résultat de la transformation pour accepter ou annuler l'opération (*undo*). Si la réalisation est validée, une nouvelle version de la spécification est créée, représentant cette réalisation.



Les différentes versions d'une spécification correspondent à de différents fichiers (sur le format <nom_fichier>_<numéro_version>.solar). La création de nouvelles versions de la spécification forme l'historique de développement. L'historique permet à l'utilisateur le retour à des étapes antérieures du raffinement et l'essai des nouvelles stratégies de raffinement. Le format de présentation de la spécification est décrit dans la section 3.4.4.

La fenêtre *Solar textuel*

La fenêtre *Solar textuel* affiche le code Solar, dans une forme textuelle, de la partie du système visible dans les fenêtres *Source* ou *Destination*. L'utilisateur peut visualiser, avec plus de détails que sur le format graphique, le comportement des unités de conception et le résultat des raffinements. Quand l'unité de conception feuille de la hiérarchie est présentée dans les fenêtres *Source* ou *Destination*, la description Solar textuelle affichée représente le système complet. Le changement de la description dans la fenêtre *Solar textuel* est possible, mais cela n'est pas conseillé puisque l'utilisateur peut facilement introduire des erreurs syntaxiques et sémantiques dans la spécification.

```
Description SOLAR – (Source)
(VIEW  FIFO_behaviour
   (VIEWTYPE "behaviour")
   (INTERFACE
      (PORT (ARRAY  MsgCount_channel_wr_req  2 )(DIRECTION  IN ) (BIT ))
      (PORT (ARRAY  MsgCount_channel_rewr_rdy  2 )(DIRECTION  OUT ) (BIT ))
      (PORT (ARRAY  MsgCount_channel_read_req  2 )(DIRECTION  IN ) (BIT ))
      (PORT  MsgCount_channel_data_int (DIRECTION  INOUT ))
      (PORT  MsgCount_channel_data_real (DIRECTION  INOUT ))
   )
   (CONTENTS
      (VARIABLE (ARRAY  queue_int  queue_size )(INTEGER ))
      (VARIABLE (ARRAY  queue_real  queue_size )(REAL ) )
      (VARIABLE  queue_in_ptr (INTEGER )(INITIALVALUE  1 ) )
      (VARIABLE  temp1 (INTEGER )(INITIALVALUE  1 ) )
      (VARIABLE  temp2 (INTEGER )(INITIALVALUE  1 ) )
      (VARIABLE  queue_out_ptr (INTEGER )(INITIALVALUE  1 ) )
      (VARIABLE  indice (INTEGER )(INITIALVALUE  1 ) )
```

La fenêtre *Historique de développement*

La fenêtre *Historique de développement* trace l'évolution du processus de conception. Les informations présentées dans cette fenêtre permettent à l'utilisateur de se rappeler des étapes de raffinement réalisées ou de réaliser automatiquement un ensemble de transformations sur la spécification. Les informations suivantes font partie de cette fenêtre: les transformations ou opérations réalisées, le domaine d'application des transformations, les paramètres et l'acceptation ou l'annulation de la transformation.

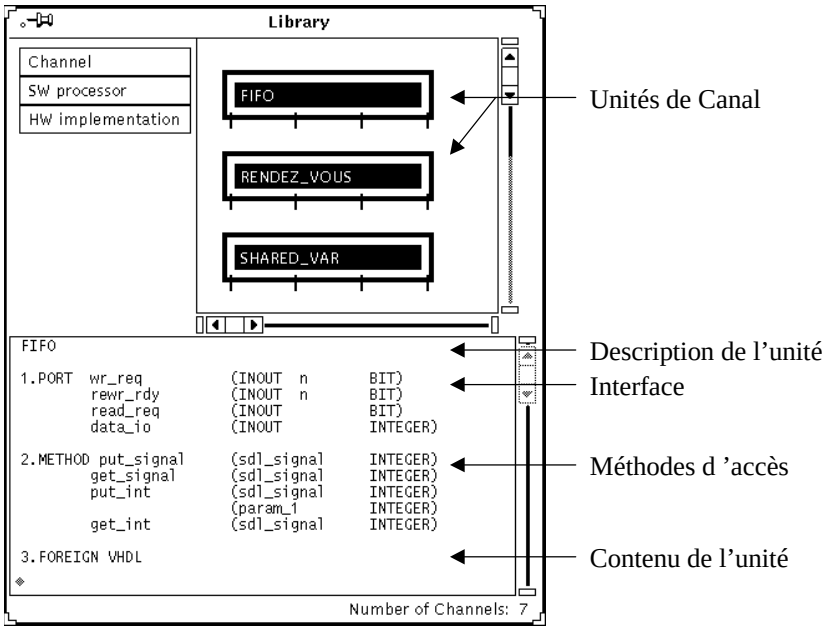
Le principal avantage de cette fenêtre est de permettre à l'utilisateur de sauvegarder l'historique de développement et de l'appliquer après un changement de la spécification initiale. L'utilisateur peut activer les transformations sauvegardées dans l'historique pour refaire automatiquement les transformations. Chaque historique de développement sauvegardé peut contenir les étapes de raffinement utilisées pour arriver à une réalisation. L'application d'une séquence de transformations sur une spécification initiale permet d'effectuer automatiquement le lien entre une spécification système et une architecture cible. De la même façon, l'utilisateur peut éditer les informations liées à l'historique pour générer différentes solutions architecturales.

Transformation History

Step	Primitive	Type	Parameters	Confirmation
0–	ATM.pr	SDL–SOLAR file conversion		OK
1–	ATM.solar	SOLAR file loaded		OK
2–	FLAT	SR	atm_I	OK
3–	MERGE	SR	tcp_layer_I ip_layer_I aal_layer_I	OK
4–	MAP	CS	sar_receiver_channel RENDEZ_VOUS	OK
5–	MAP	CS	send_channel RENDEZ_VOUS	OK
6–	MAP	CS	control_channel FIFO	OK

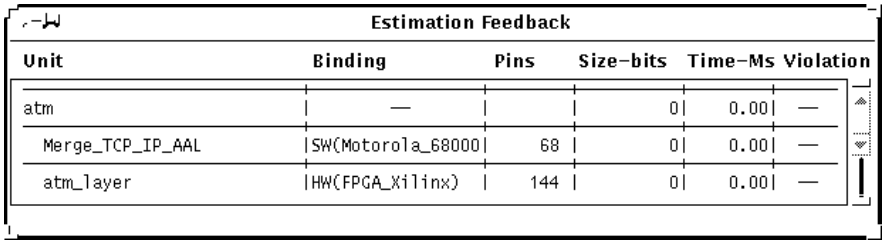
La fenêtre de *Bibliothèques*

Cette fenêtre présente les composants des bibliothèques de Cosmos. La version actuelle du système distingue entre quatre classes de composants: les protocoles de communication, les processeurs logiciels, les réalisations matérielles et les mémoires. Cette fenêtre fournit aussi des informations sur les composants, comme aide pendant l'étape d'allocation manuelle.



La fenêtre d'Estimation globale

La fenêtre d'*Estimation globale* présente la structure du système où chaque unité représente un composant de l'architecture. Les informations présentées, pour chaque unité, sont les suivantes: l'option d'allocation, le nombre de ports (*pins*) du composant, la taille du code logiciel ou du circuit matériel, le temps d'exécution et l'indicateur de violation. Le champs indicateur de violation indique la violation de contraintes de la conception. Ce champ réalise la comparaison entre les restrictions de la conception et les résultats de l'algorithme d'estimation.



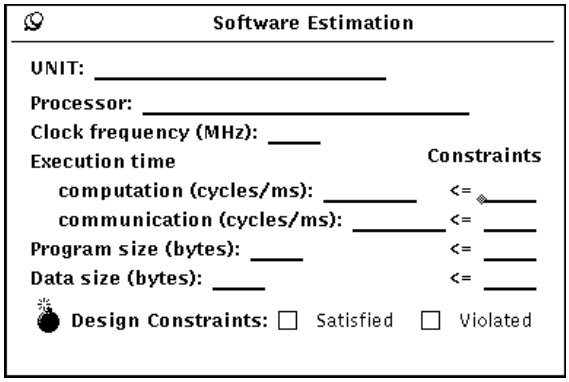
Les fenêtres d'Estimation détaillée

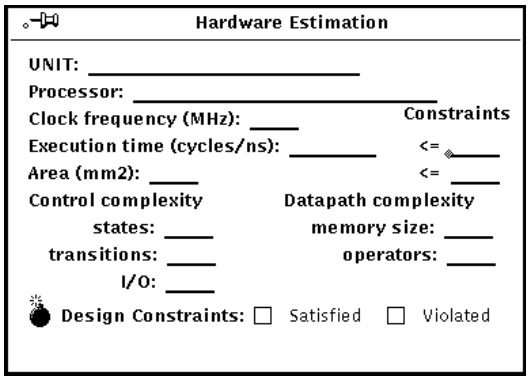
Les fenêtres d'*Estimation détaillée* du logiciel et du matériel donnent des informations complètes sur le résultat de l'estimation. L'utilisateur sélectionne une unité ou une procédure de la spécification et l'environnement présente toutes les valeurs d'estimation calculées, en respectant l'option de réalisation logicielle ou matérielle. Ces informations aident l'utilisateur à prendre des décisions de conception.

La fenêtre d'estimation du logiciel affiche: le nom de l'unité ou de la procédure sélectionnée, le processeur alloué, la fréquence de l'horloge du processeur, le temps d'exécution du processus, le temps d'exécution des procédures de communication, la taille du code logiciel et la taille des données. Le valeur liée au temps de communication prend en considération simplement le temps utilisé par le protocole et les procédures de

communication. Cette valeur ne prend pas en considération le temps d'attente ou de synchronisation.

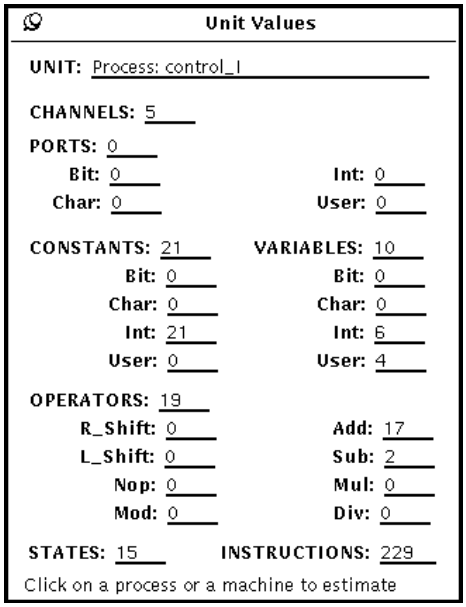
La fenêtre d'estimation du matériel affiche: le nom de l'unité ou de la procédure sélectionnée, la réalisation choisie, la fréquence de l'horloge, le temps d'exécution, la taille du circuit, la complexité de la partie contrôle et de la partie chemin de données. Les champs liés aux restrictions (*contraints*) permettent à l'utilisateur d'appliquer ou de charger des valeurs des contraintes du système.





La fenêtre des *Paramètres d'unités de conception*

Cette fenêtre présente des informations relatives aux unités de conception. Ces informations donnent à l'utilisateur un aperçu de la complexité de l'unité sélectionnée, comme: son interface, ses données, ses opérations et la taille de la description.



Bibliographie

For a job well done,
first sharp your tools.

Publications Personnelles

[Mar98] G.F. Marchioro, J.M. Daveau, T.Ben Ismail, A.A. Jerraya, **Transformational Partitioning for Co-design**, IEE Proceedings – Computer and Digital Techniques, United Kingdom, 1998.

[Mar97] G.F. Marchioro, J.M. Daveau, A.A. Jerraya, **Transformational Partitioning for Co-design of Multiprocessor Systems**, IEEE/ACM International Conference on Computer Aided Design, ICCAD'97. San Jose, CA. pp. 508-515. 9-13 November, 1997.

[Mar97a] G.F. Marchioro, J.M. Daveau, A.A. Jerraya **Co-design of Multiprocessor Systems.** XXIV Software/Hardware Integration Seminars - Semish 97. pp. 13-24. 2-8 August 1997. Brasilia, DF.

[Mar97b] G.F. Marchioro, F. Hessel, **Hardware/Software Co-design**. XVI Computer Science up-to-date Journey - JAI'97. XVII Congress of Brazilien Computer Society. pp. 449-494. 2-8 August 1997. Brasilia, DF.

[Mar97c] G.F. Marchioro, A.A. Jerraya, **Conception Conjointe Logicielle/matérielle Basée en Transformations Interactives**, Colloque CAO de circuits integres et systemes. 15-17 Janvier 1997. Villard de Lans, France.

[Mar97d] G.F. Marchioro, C. Valderrama, M. Romdhani, J.M. Daveau, F. Hessel, A.A. Jerraya, **COSMOS Framework v2.2 – User's Manual**, Internal Report, TIMA/INPG, Grenoble, France, May 1997.

[Mar95] G.F. Marchioro, T.Ben Ismail, J.M. Daveau, **Interactive Co-design Environment**, Internal Report, TIMA/INPG, Grenoble, France, November 1995.

[Mar92] G.F. Marchioro, A.A. Suzim, **System of Integrated Tools for IC Design**, XI Conferência Latino Americana de Informatica – CLEI, July 1992, Antofagasta – Chile.

[Mar92a] G.F. Marchioro, A.A. Suzim, **CAD tool integration for IC Design**, IPESI - Eletro Eletrônica Revista Brasileira de Eletrônica - Ano XII - Nov/Dez 1992, December 1992, São Paulo, Brazil.

[Mar92c] G.F. Marchioro, A.A. Suzim, **SILEX – CAD Tool for IC Design**, VII Sociedade Brasileira de Microeletrônica – SBMICRO, July 1992, São Paulo, Brazil.

[Mar90] G.F. Marchioro, A.A. Suzim, **Hierarchical Symbolic Editor for Integrated Circuits**, X Conferência Latino Americana de Informatica – CLEI, October 1990, Assunção - Paraguai .

[Mar90a] G.F. Marchioro, A.A. Suzim. **CHARRUA – A Symbolic Editor for IC Design**, V Sociedade Brasileira de Microeletrônica – SBMICRO, July 1990, Campinas, Brazil.

[Mar90b] G.F. Marchioro, A.A. Suzim, **ESQUELETO – A Schematic Editor**, V Sociedade Brasileira de Microeletrônica – SBMICRO, July de 1990, Campinas, Brazil.

[Mar89] G.F. Marchioro, A.A. Suzim, **Implementation of a Symbolic Editor for Integrated Circuits,** IV Sociedade Brasileira de Microeletrônica - SBMICRO, July de 1989, Porto Alegre, Brazil.

Co-Auteur de Publications

[Jer98] A.A. Jerraya, G.F. Marchioro, **Partition et Synthèse de la Communication pour les Systemes Logiciel/Matériel**, Co-design: Conception Conjointe Logiciel-Matériel, Eyrolles, C.I.T. Comete, pp. 87-123. Paris - France.

[Zer98] N.E. Zergainoh, G.F. Marchioro, A.A. Jerraya, **Hw/Sw Codesign of an ATM Network Interface card starting from a System Level Spécification.** ISSSE'98 International Symposium on Signals, Systems, and Electronics. October 1998, Pisa Italy.

[Abi98] M. Abid, T.Ben Ismail, A. Changuel, C.A. Valderrama, M. Romdhani, G.F. Marchioro, J.M. Daveau, A.A. Jerraya, **Design of Embedded Systems using a Hardware/Software Co-Design Methodology**. Integrated Computer-Aided Engineering (ICAE'98), 1998.

[Dav98] J.M. Daveau, G.F. Marchioro, A.A. Jerraya, **Hardware/Software Co-design of an ATM Network Interface Card: a Case Study**, Codes/CASHE'98, Washington, March 1998.

[Dav97] J.M. Daveau, G.F. Marchioro, C. Valderrama, A.A. Jerraya, **VHDL generation from SDL specification,** XIII IFIP Conference on Computer Hardware Description Languages (CHDL97), pp 182-201, April 1997, Toledo, Spain.

[Dav96] J.M. Daveau, G.F. Marchioro, A.A. Jerraya, **Partition et synthèse de la communication pour les systèmes mixtes logiciel/matériel**. Codesign - Actes des Séminaires Action Scientifique. november 1996. n. 10. pp. 23-31. France Telecom - Meylan.

[Dav96a] J.M. Daveau, G.F. Marchioro, T.Ben Ismail, A.A. Jerraya, **Cosmos: an SDL Based Hardware/Software Codesign Environment.** Hardware /Software Co-design and Co-verification. CIEM - Current Issues In Electronic Modeling. Kluwer Academic Publishers, vol 8, pp. 59-88. MA, USA, 1997.

[Dav96b] J.M. Daveau, T.Ben Ismail, G.F. Marchioro, A.A.Jerraya**, Protocol Selection and Interface Generation for HW-SW Codesign**, IEEE Transactions on VLSI Systems, Special issue on Design Automation of complex integrated systems, September 1996.

[Ism96] T.Ben Ismail, G.F. Marchioro, A.A. Jerraya, **Partitioning of VLSI Systems from a High Level Specification.** Technique et Science Informatiques (TSI). Volume 15 - n 8. pp. 1131-1165. Hermes (eds), 1996.

[Ism96a] T.Ben Ismail, G.F. Marchioro, A.A. Jerraya, **Découpage de Systèmes VLSI a partir d'une Spécification de haut niveau**, Technique et Science Informatiques (TSI), Hermes (eds), 1996.

[Jer97] A.A. Jerraya, M. Romdhani, C.A. Valderrama, Ph.Le Marrec, F. Hessel, G.F. Marchioro, J.M. Daveau. **Languages for System-Level Specification and Design**. Hardware/Software Co-Design: Principles and Practice. Kluwer Academic Publishers. pp. 209-236. 1997. Boston/Dordrecht/London

[Rom97] M. Romdhani, G.F. Marchioro, **Solar Reference Manual**, Internal Report, TIMA/INPG, Grenoble, France, Mars 1997.

[Val97] C.A. Valderrama, M. Romdhani, J.M. Daveau, G.F. Marchioro, A. Changuel, A.A. Jerraya. **Cosmos: A Transformational Co-Design Tool for Multiprocessor Architectures** Hardware/Software Co-Design: Principles and Practice. Kluwer Academic Publishers. pp. 281-330. 1997. Boston/Dordrecht/London.

Bibliographie Référencée

[Abi96] M. Abid, A. Changuel, A.A. Jerraya, **Exploration of Hardware/Software Design Space Through a Codesign of Robot Arm Controller**, In Proceedings of the European Design Automation Conference with EURO-VHDL 96, Geneva, Switzerland, pp. 42-47, September 1996.

[All97] A. Allara, S. Filipponi, W. Fornaciari, F. Salice, D. Sciuto, **A Flexible Model for Evaluating the Behavior of Hardware/Software System**, 5th International Workshop on Hardware/Software Co-Design Codes/Cache'97, Braunschweig, Germany, March 1997.

[Alo93] A. Alomary, T. Nakata, Y. Honma, **An ASIP Instruction Set Optimization Algorithm with Functional Module Sharing Constraint**, In ICCAD 1993. pp. 526-532, 1993.

[And91] Andrews, **Concurrent Programming, Principles and Practice**, Benjamin/Cummings (eds), Redwood City, Calif., pp. 484-494, 1991.

[Bar94] E. Barros, W. Rosentiel, X. Xiong, **A Method for Partitioning UNITY Language in Hardware and Software**. Proceedings of the European Design Automation Conf. (Euro-*DAC*), 1994.

[Bel91] F. Belina, D. Hogrefe, A. Sarma, **SDL with Applications from Protocol Specification**, Prentice Hall Internationnal, ISBN 0-13-785890-6, 1991.

[Bir84] A.D. Birrell, B.J. Nelson, **Implementing remote procedure call**, ACM Transactions on Computer Systems, Vol. 2, No. 1, pp. 39-59, February 1984.

[Boc93] G.V. Bochmann, **Specification Languages for Communication Protocols**, Proceedings of the Conference on Hardware Description Languages CHDL'93, 1993, pp. 365-382.

[Boo97] B. Grady, **Analyse & conception orientées objets,** Addison-Wesley, 1997.

[Buc94] K. Buchenrieder, C. Veith, **A Prototyping Environment for Control Oriented Hardware/Software Systems Using StateCharts, ActivityCharts and FPGA**, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 60-65, September 1994.

[Cam89] R. Camposano, R.M. Tablet, **Design Representation for the Synthesis of behavioural VHDL models**, XIII IFIP Conference on Computer Hardware Description Languages CHDL, 1989.

[Cam93] P. Camurati, F. Corno, P. Prinetto, C. Bayon, B. Soulas, **Inter-process Communications for System-Level Design**, International Workshop on Computer-Aided Hardware-Software Codesign, Cambridge, October 1993.

[Cam94] P. Camurati, F. Corno, P. Prinetto, C. Bayol, B. Soulas, **System-Level Modeling and Verification: A Comprehensive Design Methodology**, Proceedings of the European Design & Test Conference (EDAC-ETC-EuroASIC), Paris, France, IEEE CS Press, February 1994.

[Chi94] M. Chiodo et all, **Hardware-software codesign of embedded systems**, In IEEE MICRO, pp. 26-36, August 1994.

[Chi96] M. Chiodo, D. Engels, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, K. suzuki and 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, January 1996.

[Dav95] J.M. Daveau, T.Ben Ismail, A.A. Jerraya, **Synthesis of System Level Communication by an Allocation Based Approach**, Proceedings of the 8th International Symposium on System Synthesis, pp 150-155, September 1995.

[Dav97b] J.M. Daveau, **Spécifications Systèmes et Synthèse de la Communication pour le Co-Design Logiciel/Matériel**, Thèse de Doctorat INPG, TIMA Laboratory, 19 Decembre 1997.

[Din96] H. Ding, **Synthese Architecturale Interactive et Flexible**, PhD thesis, INPG, 1996.

[Ern93] R. Ernst, J. Henkel, T. Benner, **Hardware-software co-synthesis for microcontrollers**. IEEE Design & Test of Computers, 10(0), December 1993.

[Ern95] 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.

[Fae94] O. Faergemand, A. Olsen, **Introduction to SDL-92**, Computer Networks and ISDN Systems, Vol 26, pp. 1143-1167, 1994.

[Fel97] B. Felice, **Hardware-Software Co-Design of Embedded Systems – The Polis Approach**, Kluwer Academic Publishers, 1997.

[Fre97] L. Freund, D. Dupont, M. Israël, **Interface Optimization During Hardware-Software Partitioning**, 5th International Workshop on Hardware/Software Co-Design Codes/Cache'97, Braunschweig, Germany, pp. 75-79, March 1997.

[Gaj94] D. Gajski, F. Vahid, S. Narayan, **A System Design Methodology: Executable-Specification Refinement**, Proceedings of the European Design & Test Conference (E*DAC*-ETC-EuroASIC), Paris, France, IEEE CS Press, pp. 458-463, February 1994.

[Gaj94b] D. Gajski, F. Vahid, S. Narayan, J. Gong, **Specification and Design of Embedded Systems**, Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1994.

[Gaj94c] D. Gajski, L. Ramachandran, P. Fung, F. Vahid, S. Narayan, **Towards Achieving an 100-hour Design Cycle**, European Design Automation Conference, Grenoble, France, 1994.

[Gaj95] D. Gajski, F. Vahid, **Specification and Design of Embedded Systems**, IEEE Design & Test of Computers, pp. 53-67, Spring 1995.

[Gaj98] D. Gajski, J. Gong, F. Vahid, S. Narayan, **The SpecSyn Design Process and Human Interface**, Technical Report #93-3 http://www.ele.kth.se/ESD /courses/2B1421/specs_des.html.

[Gee93] J.D. Gee, M.D. Hill, D.N. Pnevmatikatos, A.J. Smith. **Cache performance of the SPEC92 benchmark suite**, IEEE Micro 13:4, 17-27, 1993.

[Gon94] J. Gong, D. Gajski, S. Narayan**, Software Estimation from Executable Specifications**, Proceedings of the European Design & Automation Conference (*EuroDAC*), IEEE CS Press, Grenoble, France, September 1994.

[Gon95] J. Gong, D. Gajski, S.Narayan, **Software estimation from system-level specifications**, in Proceedings of the European Conference on Design Automation (EDAC), 1995.

[Gon96] J. Gong, D. Gajski, S. Bakshi**, Model Refinement for Hardware-Software Codesign**, Proceedings of the European Design and Test Conference, pp. 270-274, March 1996.

[Gup93] R.K. Gupta, G.De Micheli, **Hardware-Software Cosynthesis using Reprogrammable Components**, IEEE Design & Test of Computers, Vol 10, No.3, pp 29-41, September 1993.

[Gup94] R.K. Gupta, C.N. Coelho, G.De Michelli, **Program Implementation Schemes for Hardware Software Systems**, IEEE Design & Test of Computers, Vol. 27, No. 1, pp. 48-55, January 1994.

[Gup94b] R.K. Gupta, G.De Micheli, **Partitioning of functional models of synchronous digital systems**, Proceedings of the European Conference on Design Automation E*DAC*, 1994.

[Hal93] N. Halbwachs, **Synchronous programming of reactive systems**, Kluwer Academic Publishers, ISBN 0-7923-9311-2, 1993.

[Har95] W. Hardt, R. Camposano, **Specification Analysis for HW/SW - Partitioning**, In 3. GI/ITG Workshop, Passau, Germany, March 1995.

[Har87] D. Harel, **Statecharts: a Visual Formalism for Complex Systems**, Science of Computer Programming 8, North-Holland, pp.231-274, 1987.

[Hen94] J. Henkel, T. Benner R. Ernst, W. Ye, N. Serafimov, G. Glawe, **COSYMA: A Software Oriented Approach to Hardware/Software Codesign**, The Journal of Computer and Software Engineering, Vol 2, No 3, pp. 293-314, 1994.

[Hen94b] J. Henkel, R. Ernst, U. Holtman, T. Benner, **Adaptation of Partitioning and High Level synthesis in Hardware/Software Co-Synthesis**, Proceedings of the IEEE International Conference on Computer Aided Design, pp. 96-100, November 1994.

[Hen95] J. Henkel, R. Ernst, **A Path-Based Technique for Estimating Hardware Runtime in Hw/Sw Cosynthesis**, 8th Intl. Symposium on System Synthesis (SSS), pp. 116-121, 1995.

[Hen96] J.L. Hennessy, D.A. Patterson, **Architecture des ordinateurs: un approche quantitative**, Thomson Publishing, 1996.

[Her94] D. Herrmann, J. Henkel, R. Ernst. **An Approach to the Adaptation of Estimated Cost Parameters in the COSYMA System**, in Third International Workshop on Hardware/Software Codesign. Grenoble, France: IEEE Computer Society Press, 100-107, 1994.

[Hou96] J. Houand, W. Wolf, **Process Partitioning for Distributed Embedded Systems**, Proceedings of the 7th CODES/CASHE Workshop, pp. 70-76, March 1996.

[Ism95] 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.

[Ism95b] T.Ben Ismail, A.A. Jerraya, **Synthesis Steps and Design Models for CoDesign**, IEEE Computer, special issue on rapid-prototyping of microelectronic systems, Vol. 28, No. 2, pp. 44-52, February 1995.

[Ism96] T.Ben Ismail, **Synthèse au Niveau Système et Conception de Systèmes Mixtes Logiciels/Matériels**, Thèse de Doctorat INPG, TIMA Laboratory, Janvier 1996.

[Jer94] A.A. Jerraya, K. O'Brien, ***Solar*: An Intermediate Format for System-Level Modeling and Synthesis**, in "Computer Aided Software/Hardware Engineering", J.Rozenblit, K.Buchenrieder (eds), IEEE Press, Piscataway, N.J., pp 147-175, 1994.

[Jer97] A.A. Jerraya, H. Ding, P. Kission, M. Rahmouni, **Behavioral Synthesis and Component Reuse with VHDL**, Kluwer Academic publishers, 1997.

[Kal94] A. Kalavade, E.A. Lee, **A Global Criticality/Local Phase Driven Algorithm for the Constrained Hardware/Software Partitioning Problem**, Proceedings of the Third International Workshop on Hardware/Software Codesign, pp. 42-48, September 1994.

[Kal95] 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.

[Kis96] P. Kission, **Exploitation de la hierarchie et de la ré-utilisation de blocs existants par la synthèse de haut niveau**, PhD thesis, INPG, 1996.

[Kru92] C.W. Krueger, **Software Reuse**, ACM Computing Surveys, vol. 24, no. 2, pp. 131-183, June 1992.

[KuD88] D.C. Ku, G.De Micheli, **HardwareC - A Language for Hardware Design**, Computer Systems Lab, Stanford University, Technical Report N. CSL-TR-88-362, August 1988.

[Kya95] O. Kyas, **ATM Networks**, International Thomson Publishing, 1995.

[Lag91] 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.

[Lan95] 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.

[Mad95] J. Madsen, B. Hald, **An Approach to Interface Synthesis**, Proceedings of the 8th International Symposium on System Synthesis, pp. 16-21, September 1995.

[Mad96] J. Madsen, J. Brage, **Codesign Analysis of a Computer Graphics Application**, Journal: Design Automation of Embedded Systems, vol.1, no.1-2, January 1996.

[Mad96b] J. Madsen, J. Grode, P.V. Knudsen, M.E. Petersen, A. Haxthausen. **Lycos: the lyngby co-synthesis system**. Design Automation for Embedded Systems, 1996.

[Mar95] P. Marwedel, G. Goessens, **Code Generation for Embedded Processors (DSP)**, Kluwer Academic Publishers, 1995.

[Mic94] G.De Micheli. **Computer-Aided Hardware-Software Codesign**. IEEE Micro, Vol. 14, No. 4, pp. 10-16, Sep. 1994

[Mic97] G.De Micheli, M. Sami, **Hardware/Software Co-Design**, NATO ASI, Series E: Applied Sciences- Vol. 310, Kluwer Academic Publishers, Dordrecht/Boston/London, pp. 467, 1997.

[Moo96] V.J. Mooney, C.N. Coelho, T. Sakamoto and G. De Micheli**, Synthesis From Mixed Specifications**, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 114-119, September 1996.

[Nar95] S. Narayan, D. Gajski, **Interfacing Incompatible Protocols Using Interface Process Generation**, Proceedings of the IEEE Design Automation Conference, pp. 468-473, June 1995.

[Nar96] S. Narayan, D. Gajski, **Rapid Performance Estimation For System Design**, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 206-211, September 1996.

[Obj97] **ObjectGeode**, http://www.verilogusa.com/og/og.html.

[Pau94] 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), Kluwer Academic Publishers, 1994.

[Pau96] P. Paulin, J. Frehel, M. Harrand, E. Berrebi, C. Liem, F. Nacabul, J. Herluison, **High-Level Synthesis and Codesign Methods: An Application to a Videophone Codec**, In Proceedings of the European Design Automation Conference with EURO-VHDL 95, Brighton, Great Britain, pp. 444-451, September 1996.

[Pol98] **Polis**, http://www-cad.eecs.berkeley.edu/~szollar/Research/EmbSysJava/polis.html.

[Pra92] S. Prakash, A.C. Parker, **SOS: synthesis of application-specific heterogeneous multiprocessor systems**. Journal of Parallel and Distributed Computing, 16, 1992.

[Pry95] M.De Pryker, **ATM, Mode de Transfert Asynchrone**, Masson / Prentice Hall, ISBN 2-225-84874-X, 1995.

[Puj95] G. Pujolle, **Les Réseaux**, Editions Eyrolles, 1995.

[Rom95] M. Romdhani, R.P. Hautbois, A. Jeffroy, P. Chazelles, A.A. Jerraya, **Evaluation and Composition of Specification Languages**, an Industrial Point of View, Proceedings of the IFIP Conf. Hardware Description Languages (CHDL), Japan, pp. 519-523, September 1995.

[Rom96] K.V. Rompaey, D. Verkest, I. Bolsens, H. DeMan, CoWare, **A Design Environment for Heterogeneous Hardware/Software Systems**, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 252-257, September 1996.

[Rou97] F. Rousseau, **Etude du partitionnement logiciel/matériel dans la conception de systèmes électroniques: application aux systèmes de télécommunications orientés traitement de données**, Thèse de Doctorat à l'université de Evry, Juillet 1997.

[Sar97] R. Saracco, P.A.J. Tilanus, **CCITT SDL: an Overview of the Language and its Applications**, Computer Networks and ISDN Systems, Special issue on CCITT SDL Vol 13 No 2, pp 65-74, 1987.

[Smi91] T.E. Smith, D.E. Setliff, **Towards an Automatic Synthesis System for Real-Time Software**, In: Proc. Of 12th Real-Time Systems Symposium, pp. 34-42, 1991.

[Sri95] 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.

[The94] 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.

[Vah92] F. Vahid, D.D. Gajski, **Specification Partitioning for System Design**, Proceedings of the 29th Design Automation Conference (*DAC*), IEEE Press, June 1992.

[Vah94] F. Vahid, J. Gong, D.D. Gajski, **A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning**. In Proceedings of the European Design Automation Conference, pp. 214-219, 1994.

[Vah95] F. Vahid, D.D. Gajski, **Closeness metrics for system-level functional partitioning**, Proceedings of the European Conference on Design Automation EDAC, 1995.

[Xcl95] **Xclass'95**, http://www.terraware.net/ftp/pub/Mirrors/FVWM95/xclass.html.

[Wol94] W. Wolf, **Hardware-Software Co-Design of Embedded Systems**, Proceedings of the IEEE, vol 82. no 7, pp. 967-989, July 1994.

[Wol95] W. Wolf, **Object-Oriented Co-Synthesis of Distributed Embedded Systems**, in Proceedings of CHDL'95, IFIP, 1995.

Bibliographie Générale

[Aho91] A. Aho, R. Sethi, J. Ullman, **Compilateurs Principes, techniques et outils**, InterEditions, Paris, 1991.

[Aho93] A. Aho, J. Ullman, **Concepts fondamentaux de l'informatique**, Dunod, Paris, 1993.

[And83] G.R. Andrews, F.B. Schneider, **Concepts and notation for Concurrent Programming**, Computing Survey 15(1), March, 1983.

[Aka96] H. Akaboshi, T. Ishihara, H. Yasuura, **An Approximate Estimation of Power for Processor Architecture Design**, Proc. 3rd Asian Pacific Conf. Hardware Description Languages(APCHDL'96), Bangalore, pp.23-27, Jan. 1996.

[Axe97] J. Axelsson**, Analysis and Synthesis of Heterogeneous Real-Time Systems**, Thesis, Linköping University, Sweden, 1997.

[Bal97] A. Balboni, W. Fornaciari, D. Sciuto**, Partitioning of Hw/Sw/ Embedded Systems: a Metrics-Based Approach**, Integrated Computer-Aided Engineering, 1997.

[Bar94b] E. Barros, A. Sampaio, **Towards Provably Correct Hardware/Software Partitionning Using Occam**, Proceedings of the Third International Workshop on Hardware/Software Codesign, pp. 210-217, September 1994.

[Bee93] Benner, T., J. Henkel, R. Ernst. **Internal Representation of Embedded Hardware- /Software-Systems**, in Second IFIP International Workshop on Hardware/Software Codesign (Codes/CASHE'93). Innsbruck-Igls, Austria, May 24-27. 1993.

[Bel94] M. Belhadj, T. Gautier, P.Le Guernic, P. Quinton, **Towards a Multi-Formalism Framework for Architectural Synthesis : the ASAR Project**, Proceedings of the Third International Workshop on Hardware/Software Codesign, pp. 25-32, September 1994.

[Ben91] A. Benveniste, G. Berry, **The Synchronous Approach to Reactive and Real-time Systems**, Proceedings of the IEEE, Vol. 79, No. 9, pp. 1270-1282, September 1991.

[Ben96] A. Bender, **MILP based Task Mapping for Heterogeneous Multiprocessor Systems**, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 190-197, September 1996.

[Bou91] F. Boussinot, R.De Simone, **The ESTEREL Language**, Proceedings of the IEEE, Vol. 79, No. 9, pp. 1293-1304, September 1991.

[Buc92] K. Buchenrieder, C. Veith, **CODES: A Practical Concurent Design environment**, Proceedings of the Internationnal Workshop on Hardware/Software Codesign, October 1992.

[Buc93] K. Buchenrieder, A. Sedlmeier, C. Veith, **Hw/Sw Codesign with PRAM using CODES**, Proceedings of the Conference on Hardware Description Languages, April 1993.

[Bud87] S. Budkowski, P. Dembinski, **An Introduction ESTELLE : A Specification language for distributed systems**, Computer Networks and ISDN Systems, Vol 13, No. 2, pp. 2-23, 1987.

[BuH94] J. Buck, S. Ha, E. Lee, **Ptolemy: A framework for simulating and prototyping heterogeneous systems**, International Journal of Computer Simulation, January 1994.

[Cal92] J.P. Calvez, **Spécification et Conception des Systèmes: une méthodologie**, Masson, 1992.

[Cal94] J.P. Calvez, D. Isidoro. **A Codesign Experience with the MCSE Methology**, in Third International Workshop on Hardware/Software Codesign. Grenoble, France: IEEE Computer Society Press, 140- 147. 1994.

[Cam96] R. Camposano, J. Wilberg, **Embedded System Design, Design Automation for Embedded Systems**, Vol. 1, No. 1-2, pp. 5-50, January 1996.

[Car94] C. Carreras, J.C. Lopez, M.L. Lopez, M. Miranda, O. Nieto. **Estimation Methodology for Hardware-Software Codesign based on Intermediate Analytic Models**, 21st Workshop on CAD Aids for VLSI in Europe (CAVE), Sesimbra (Portugal), May 24-27, 1994.

[Car96] C. Carreras, J.C. Lopez, M.L. Lopez, C. Delgado-Kloos, N. Martinez, L. Sanchez. **A Co-Design Methodology Based on Formal Specification and High-Level Estimation**, Proc. Workshop on HW/SW Co-Design CODES/CASHE'96, pp. 28-35, Pittsburgh, PA (USA), Mar. 18-20, 1996.

[Cha96] A. Changuel, R. Rolland, A.A. Jerraya, **Design of an Adaptative Motors Controller based on Fuzzy Logic using Behavioral Synthesis**, In Proceedings of the European Design Automation Conference with EURO-VHDL 96, Geneva, Switzerland, pp. 48-52, September 1996.

[Cho95] P. H. Chou, R. B. Ortega, G. Borriello, **The Chinook Hardware/Software Co-Synthesis System**, Proceedings of the 8th International Symposium on System Synthesis, pp. 22-27, September 1995.

[Com96] D. Comer, **TCP/IP, Architecture, Protocole, Application**, Collection iaa, InterEditions, 1996.

[Cor94] T. Cormem, C. Leiserson, R. Rivest, **Introduction à l'Algorithmique**, Dunod, ISBN 2-10-003128-7, 1994.

[Dav92] J.W. Davidson, J.R. Rabung, D.B. Whalley, **Relating Static and Dynamic Machine Code Measurements**, in IEE Transactions on Computers, Vol 41, n 4, April 1992.

[DeM95] 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.

[Dru89] D. Drusinsky, D. Harel, **Using StateCharts for Hardware Description and Synthesis**, IEEE Transactions on Computer-Aided Design, Vol. 8, No. 7, pp. 798-807, July 1989.

[Dut91] N.D. Dutt, J. Hwee Cho, T. Hadley, **A User Interface for VHDL Behavioural Modelling**, Proceedings of the Conference on Hardware Description Languages, pp. 375-393, April 1991.

[Eck93] W. Ecker, **Using VHDL for HW/SW Co-Specification**, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 500-505, September 1993.

[Ele94] P. Eles, Z. Peng, A. Doboli, **VHDL System Level Specification and Partitionning in a Hardware/Software Co-Synthesis environment**, Proceedings of the Third International Workshop on Hardware/Software Codesign, pp. 49-53, September 1994.

[Fft97] **FFT Benchmark**, http://theory.lcs.mit.edu/~fftw/benchmark/benchmark.html.

[Fil93] D. Filo, D. Ku, C. N. Coelho, G. de Micheli, **Interface Optimization for Concurrent Systems Under Timing Constraints**, IEEE Transactions on VLSI, Vol. 1, pp. 268-281, September 1993.

[Gau87] T. Gautier, P. Le Guernic, Signal, **A Declarative Language for Synchronous Programming of Real Time Systems**, Computer Science, Formal Languages and Computer Architecture, No 274, 1987.

[Geb92] C. H. Gebotys, **Optimal Scheduling and Allocation of Embedded VLSI Chips**, Proceedings of the IEEE Design Automation Conference, pp. 116-120, June 1992.

[Goe91] R. Goettke, E. Brehm, **START - A Tool for Performance and Reliability Evaluation of Hardware / Software Designs**, MCC Technical Report, Technical Report MCC-CAD-156-91, 1991.

[Gre94] P. Green, P. Rushton, R. Beggs. **An Example of Applying the Codesgn Method MOOSE**, in Third International Workshop on Hardware/Software Co-design, Grenoble, France. IEEE Computer Society Press, 65-72. 1994.

[Gup92] R.K. Gupta, C. N. Coelho, G. de Michelli, **Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components**, Proceedings of the IEEE Design Automation Conference, pp. 225-230, June 1992.

[Gup94c] R.K. Gupta, G. de Michelli, **Constrained Software Generation for Hardware-Software Systems**, Proceedings of the Third International Workshop on Hardware/Software Codesign, pp. 56-63, September 1994.

[Gup96] R.K. Gupta, G. de Michelli**, A Co-synthesis Approach to Embedded System Design Automation**, Design Automation for Embedded Systems, Vol. 1, No. 1-2, pp. 69-120, January 1996.

[Gup97] R.K. Gupta, G. DeMicheli, **Specification and Analysis of Timing Constraints for Embedded Systems**, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol 16, No. 3, March 1997.

[Gue91] P. Le Guernic, T. Gautier, M. Le Borgne, C. Lemaire**, Programming Real-Time Applications with SIGNAL**, Proceedings of the IEEE, Vol. 79, No. 9, pp. 1321-1336, September 1991.

[Hoa78] C. A. R. Hoare, **Communicating Sequential Processes**, Communication of the ACM, Vol 21, No 8, pp. 666-677, August 1978.

[Hul95] H. Hulgaard, S.M.Burns, T, Amon, G. Borriello, **An algorithm for extract bounds on the time separation of events in concurrent systems**, IEEE Transactions on Computer, vol 44, pp. 1306-1317, November 1995.

[HuY93] Y. Hu, A. Ghouse, B.S. Carlson, **Lower Bounds on the Iteration Time and the Number of Resources for Functional Pipelined DataFlow Graphs**, Proc. ICCD'93, pp 21-24, 1993.

[Ism93] T. Ben Ismail, J. M. Daveau, K. O'Brien, A.A. Jerraya, A System Level **Communication Synthesis Approach for Hardware/Software Systems**, Microprocessors and Microsystems, Vol. 20, pp. 149-157, 1996.

[Itu93] ITU-T, **Z.100 Functional Specification and Description Language**, Recommendation Z.100 - Z.104, March 1993.

[Jai92] R. Jain, A.C. Parker, N. Park, **Predicting System-Level Area and Delay for Pipelined and Non-pipelined Designs**, IEEE Trans. CAD, Vol 11, No 8, pp 955-965, August 1992.

[Jan94] A. Jantsch, P. Ellervee, J. Oberg, A. Hemani, H. Tenhunen, **Hardware/Software Partitioning and Minimizing Memory Interface Traffic**, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 226-231, February 1994.

[Kah93] S. Kahlert, J. U. Knabchen, D. Monjau, **Design and Analysis of Heterogeneous Systems Using Graphically Oriented Methods : A Case Study**, Proceedings of SASIMI, pp. 23-32, October 1993.

[Kal93] A. Kalavade, E.A. Lee, **A Hardware/Software Codesign Methodology for DSP applications**, IEEE Design & Test of Computers, Vol. 10, No. 4, pp. 16-28, December 1993.

[Kha95] S. A. Khan, V. K. Madisetti, **System Partitioning of MCMs for Low Power**, IEEE Design & Test of Computers, Vol. 12, No. 1, pp. 41-52, Spring 1995.

[Kis95] P. Kission, H. Ding, A. A. Jerraya, **VHDL Based Design Methodology for Hierarchy and Component Reuse at the Behavioral Level**, Proceedings of the European Design Automation Conference with Euro-VHDL, September 1995.

[Kni96] M. J. Knieser, C. A. Papachristou, **COMET : A Hardware-Software Codesign Methodology**, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 178-183, September 1996.

[Koc94] G. Koch, U. Kebschull, W. Rosenstiel, **A Prototyping Environment for Hardware/Software Codesign in the COBRA Project**, Proceedings of the Third International Workshop on Hardware/Software Codesign, pp. 10-16, September 1994.

[Kum93] S. Kumar, J.H. Aylor, B.W. Johnson, W.A. Wulf, **Exploring Hardware/Software Abstractions & Alternatives for Codesign**, in International Workshop on Hardware-Software Co-Design. Cambridge, Massachusetts, USA., October 7-8: Workshop Handouts, p. 15. 1993.

[Kum96] S. Kumar, J. Aylor, B.W. Johnson, Wm. A. Wulf, **The Codesign of Embedded Systems**, Kluwer Academic Publishers, 1996.

[Lap92] P.A.Laplante, **Real-Time Systems Design and Analysis - an engineer's handbook**, IEEE Computer Society Press, New York, 1992.

[Lav94] L. Lavagno, M. Chiodo, P. Giusto, **Hardware-Software Co-design of Embedded Systems**, IEEE Micro, Vol 14, No 4, pp. 26-36, August 1994.

[Lav96] L. Lavagno, A. Sangiovani-Vincentelli, H. Hsieh**, Embedded System Codesign: Synthesis and Verification**, Kluwer Academic Publishers, Boston, MA, pp. 213-242, 1996.

[Lin96] B. Lin, S. Vercauteren, H. De Man, **Embedded Architecture Co-Synthesis and System Integration**, Proceedings of the 7th CODES/CASHE Workshop, pp. 2-9, March 1996.

[LiY95] Y.S. Li, S. Malik, **Performance analysis of embedded software using implicit path enumeration**, Proceedings Design Automation Conference, pp. 456-461, June 1995.

[Lon96] P. Longuet, **Java: votre guide complet de développement**, Sybex, 1996.

[Mic92] P. Michel, U. Lauther, P. Duzy, **The Synthesis Approach to Digital System Design**, Kluwer Academic Publishers, 1992.

[Mic95] G. De Micheli, M Sami, **Hardware/Sofware Co-Design**, Kluwer Academic Publishers, 1995.

[Nac93] A. Nachef, **Modélisation des systèmes distribués**, Technique et Science Informatiques, Vol 12, No. 2, pp. 163-192, 1993.

[Neb96] W. Nebel, G. Schumacher, **Object Oriented Hardware Modeling - Where to apply and what are the objects**, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 428-433, September 1996.

[Nie96] R. Niemann, P. Marwedel**, Hardware/Software Partitioning using Integer Programming**, Proceedings of the European Design and Test Conference, pp. 473-479, March 1996.

[Ort97] R.B. Ortega, G. Borriello, **Communication Synthesis for Embedded Systems with Global Considerations**, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 69-73, September 1997.

[Par93] C.Y. Park, **Predicting program execution times by analyzing static and dynamic program paths**, Real-Time System, vol 5, No 1, pp. 31-62, March 1993.

[Pds97] **Performance Database Server**, http://performance.netlib.org/performance/html/PDStop.html.

[Pen91] Z. Peng, J. Fagerstrsm, K. Kuchcinski, **A Unified Approach to Evaluation and Design of Hardware/Software Systems**, MCC Technical Report, Technical Report MCC-CAD-156-91, 1991.

[Pen93] Z. Peng, K. Kuchcinski, **An Algorithm for Partitioning of Application Specific Systems**, Proceedings of the European Design Automation Conference (EDAC), pp. 316-321, February 1993.

[Pus89] P. Puschner, C. Koza, **Calculating the Maximum Execution Time of Real-Time Programs**, The Journal of Real-Time Systems, Vol. 1, pp. 159-176, 1989.

[Ram92] C.Ramachandran, F.J. Kurdahi, D. Gajski, V. Chaiyakul, A. Wu, **Accurate Layout Area and Delay Modeling for System Level Design**, Proc. ICCAD'92, Nov. 1992.

[Ram94] K. Ramamritham, J. A. Stankovic, **Scheduling algorithms and operating systems support for real-time systems**. Proceedings of the IEEE, 82(1), January 1994.

[Rim92] M. Rim, R. Jain, **Estimating Lower-Bound Performance of Schedules Using a Relaxation Technique**, Proc. ICCD'92, pp. 290-294, October 1992.

[Sha93] A. Sharma, R. Jain, **Estimating Architectural Resources and Perfornce for High-Level Synthesis Applications**, IEEE Trans. VLSI Systems, vol 1, No 2, pp. 175-190, June 1993.

[Sha94] L. Sha, R. Rajkumar, S. S. Sathaye, **Generalized rate-monotonic scheduling theory: A framework for developing real-time systems**. Proceedings of the IEEE, 82(1), January 1994.

[Shi94] K. G. Shin, P. Ramanathan, **Real-time computing: A new discipline of computer science and engineering**. Proceedings of the IEEE, 82(1), January 1994.

[Sta95] J. Staunstrup, **Toward a Common Model of Software and Hardware Components**, in Codesign: Computer-Aided Software/Hardware Engineering, K. IEEE Press: Piscataway, NJ, pp. 117-127, 1995.

[Sta97] J. Staunstrup, W. Wolf, **Hardware/Software Co-Design: Principles and Practice**, Kluwer Academic Publishers, 1997.

[Sun98] **Sun Microsystems**, http://www.sun.com/.

[Suz96] K. Suzuki, A.S. Vincentelli, **Efficient Software Performance Estimation Methods for Hardware/Software Co-design**, (DAC'96) 33rd Design Automation Conference, Las Vegas, USA, 1996.

[Tho93] D.Thomas, J.Adams, H.Schmit, **A model and methodology for hardware-software codesign**, IEEE Design & Test of Computers, September 1993.

[Tim93] A.H. Timmer, M.J.M. Heijligers, J.A.G. Jess, **Fast System-Level Area-Delay Curve Prediction**, Proc. 1st APCHDL, pp. 198-207, 1993.

[Vah94b] F. Vahid, D.D. Gajski, **A New Method for Rapid Hardware Estimation during Hardware/Software Functional Partitioning**, UC Irvine ICS Technical Report, Technical Report TR 92-29, 1994.

[Vah95b] F. Vahid, S. Narayan, D. Gajski, **SpecCharts: A VHDL Front End for Embedded Systems**, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 14, No. 6, pp. 694-706, June 1995.

[Ver96] S. Vercauteren, B. Lin, H. de Man, **Constructing Application Specific Heterogeneous Embedded Architecture from Custom HW/SW Application**, Proceedings of the IEEE Design Automation Conference, pp. 521-526, June 1996.

[Vos94] M. Voss, **Towards a Theory for Hardware/Software Codesign**, in Third International Workshop on Hardware/Software Codesign. Grenoble, France, IEEE Computer Society Press, 173- 180. 1994.

[YeE93] W. Ye, R. Ernst, Th. Benner, J. Henkel. **Fast Timing Analysis for Hardware-Software Co-Synthesis**. In Proceedings of the ICCD'93, pp. 452-457, October 1993.

[Yen95] Ti-Yen, W. Wolf, **Sensitivity-Driven Co-Synthesis of Distributed Embedded Systems**, Eighth International Symposium on System Synthesis, Cannes France, pp. 4-9, September 13-15, 1995.

[Yen95b] T. Yen, W. Wolf, **Performance estimation for real-time distributed embedded systems**, In Proceedings, IEEE International Conference on Computer Design, 1995.

[Wil94] J. Wilberg, R. Camposano, W. Rosenstiel, **Design Flow for Hardware/Software Co-Synthesis of a Video Compression System**, Proceedings of the Third International Workshop on Hardware/Software Codesign, pp. 73-80, September 1994.

[Wol93] W. Wolf, J.C. Martinez, **C Program Performance Estimation for Embedded Systems Architecture Sizing**, in International Workshop on Hardware-Software Co-Design. Cambridge, Massachusetts, USA., October 7-8: Workshop Handouts, p. 12. 1993.

[Wol94b] W. Wolf, **Architectural co-synthesis of distributed embedded computing systems**. IEEE Transactions on VLSI Systems, 1994.

[Woo92] N. Woo, W. Wolf, A. Dunlop**. Compilation of a single specification into hardware and software**. In Int'l Workshop on Hardware-Software Codesign, Oct. 1992.

Résumé

Ce travail de thèse développe une nouvelle méthodologie pour la conception conjointe du logiciel et du matériel. Cette méthodologie met en pratique une approche transformationnelle de découpage capable de manipuler des systèmes distribués et des architectures multiprocesseurs. Cette approche semi-automatique de découpage réalise le lien entre une spécification au niveau système et une architecture logicielle/matérielle de manière rapide permettant une exploration rapide de l'espace des solutions.

Le principal domaine d'application de ce travail est la conception des architectures distribuées, composées de processeurs logiciels (e.g. programmes) et processeurs matériels (e.g. ASICs). Par rapport à l'approche classique de conception, utilisée dans la plus grande partie des systèmes de conception conjointe existants, cette approche supporte des architectures flexibles composées de processeurs qui communiquent en utilisant un réseau de communication complexe. La principale contribution de ce travail est l'application de l'approche transformationnelle à la conception conjointe logicielle/matérielle d'architectures multiprocesseurs.

Mots-clés : Conception conjointe logicielle/matérielle, découpage transformationnel, estimation de performance, exploration de l'espace des solutions, raffinement de spécifications système.

Abstract

This thesis develops a new methodology for hardware/software co-design. It introduces a transformational partitioning approach capable of handling distributed systems and multiprocessor target architectures. This semi-automatic partitioning approach allows to link a system-level specification to a hardware/software architecture in a short design time with fast exploration of the design space.

The main application domain of this work is the design of distributed architectures composed of software processors (e.g. Instruction-level programs) and hardware processors (e.g. ASICs). In contrast to the classical co-design approach implemented by most of existing co-design systems, this approach deals with flexible architectures composed of processors communicating through a complex network. The main contribution of this work is the application of the transformational approach to hardware/software co-design of multi-processor architecture.

Keywords : Hardware/software co-design, transformational partitioning, design space exploration, performance estimation.

Format Papier: ISBN 2_913329_04_7

Format Electronique: ISBN 2_913329_05_5