Spécifications systèmes et synthèses de la communication pour le co-design logiciel/matériel

Jean Marc Daveau

THÈSE DE DOCTORAT

présentée par

Jean-Marc DAVEAU

pour obtenir le grade de **DOCTEUR**

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(arrêté ministériel du 30 mars 1992)

Spécialité : **Microélectronique**

SPÉCIFICATIONS SYSTÈMES ET SYNTHÈSE DE LA COMMUNICATION POUR LE CO-DESIGN LOGICIEL/MATÉRIEL

Date de Soutenance : 19 Décembre 1997

Composition du jury :

Messieurs :	Guy MAZARÉ	*Président*
	Utz BAITINGER	*Rapporteur*
	Przemyslaw BAKOWSKI	*Rapporteur*
	Roland AIRIAU	*Examinateur*
	Stanislaw BUDKOWSKI	*Examinateur*
	Ahmed JERRAYA	*Examinateur*

Thèse préparée au sein du Laboratoire TIMA-INPG
46, Avenue Félix Viallet, 38031 Grenoble Cedex.

Remerciements

Je voudrais d'abord remercier monsieur Bernard Courtois, directeur de recherche au CNRS et directeur du laboratoire TIMA, de m'avoir accueilli dans l'équipe d'architecture des ordinateurs.

Je tiens à exprimer ma profonde reconnaissance à monsieur le professeur Ahmed Jerraya, directeur de recherche au CNRS, pour m'avoir accueilli dans son équipe et sans qui cette thèse n'aurait pas vu le jour. En effet, en plus de la confiance qu'il m'a accordé en acceptant de diriger cette thèse, ses conseils et sa disponibilité furent pour moi d'un grand support tout au long de ces trois années.

Je tiens à remercier monsieur Guy Mazaré, directeur de l'ENSIMAG de m'avoir fait l'honneur de présider mon jury.

Je voudrais aussi exprimer ma gratitude à messieurs Przemyslaw Bakowski, professeur à l'IRESTE à Nantes et Utz Baitinger, professeur à l'université de Stuttgart et directeur de l'Institut des Ordinateurs à Hautes Performances d'avoir accepté la lourde tache de rapporter sur cette thèse. Enfin, pour leur participation au jury je remercie monsieur Stanislaw Budkowski, professeur à l'INT d'Evry et directeur du département réseau et monsieur Roland Airiau, ingénieur au CNET.

Je suis infiniment reconnaissant à Sabiha Arab qui a accepté de relire ma thèse et de corriger les nombreuses fautes d'ortographe. Ses nombreuses remarques m'ont permis d'améliorer la qualité et la lisibilité du manuscript.

Je n'oublie ni la disponibilité, ni la gentillesse des secrétaires, Isabelle Essalhienne, Patricia Scimone, Chantal Bénis, Corine Durand-Viel, et j'en oublie.

Je ne saurais oublier les autres membres de l'équipe SLS Gilberto Marchioro, François Nacabal, Phillipe Guillaume, Rodlophe Suescun, Imed Moussa et tous les autres.

Cette thèse à été réalisée dans le cadre d'un projet CNET-France telecom.

Voyager, c'est bien utile, ça fait travailler l'imagination. Tout le reste n'est que déceptions et fatigues. Notre voyage à nous est entièrement imaginaire. Voilà sa force.

Il va de la vie à la mort. Hommes bêtes, villes et choses, tout est imaginé. C'est un roman, rien qu'une histoire fictive. Littré le dit, qui ne se trompe jamais.

Et puis d'abord tout le monde peut en faire autant. Il suffit de fermer les yeux. C'est de l'autre coté de la vie.

Louis Ferdinant Celine, "*Voyage au bout de la nuit*".

Ils se designaient eux-mêmes : Kaweskars, les Hommes. On les croyait dépourvus de vrai langage, s'exprimant par onomatopées. En réalité, ils avaient une langue très riche où manquaient seulement tragiquement les mots qui expriment le bonheur et la beauté.

Jean Raspail "*Qui se souvient des Hommes.*"

Résumé

Au fur et à mesure que la complexité des systèmes s'accroît, il devient nécessaire de définir de nouvelles méthodes permettant de la gérer. Une des façons de maîtriser cette complexité est d'élever le niveau d'abstraction des spécifications en utilisant des langages de spécification systèmes. D'un autre coté, l'élévation du niveau d'abstraction augmente le fossé entre les concepts utilisés pour la spécification (processus communicants, communication abstraite) et ceux utilisés par les langages de description de matériel. Bien que ces langages soient bien adaptés à la spécification et à la validation de systèmes complexes, les concepts qu'ils manipulent ne sont pas aisément transposables sur ceux des langages de description de matériels. Il est donc nécessaire de définir de nouvelles méthodes permettant une synthèse efficace à partir de spécifications systèmes. Le sujet de cette thèse est la présentation d'une approche de génération de code C et VHDL à partir de spécifications systèmes en SDL. Cette approche résout la principale difficulté rencontrée par les autres approches, à savoir la communication inter-processus. La communication SDL peut être traduite en VHDL en vue de la synthèse. Cela est rendu possible par l'utilisation d'une forme intermédiaire qui supporte un modèle de communication général qui autorise la représentation pour la synthèse de la plupart des schémas de communication. Cette forme intermédiaire permet d'appliquer au système un ensemble d'étapes de raffinement pour obtenir la solution désirée. La principale étape de raffinement, appelée synthèse de la communication, détermine le protocole et les interfaces utilisés par les différents processus pour communiquer. La spécification raffinée peut être traduite en C et VHDL pour être utilisée par des outils du commerce. Nous illustrons la faisabilité de cette approche par une application à un système de télécommunication : le protocole TCP/IP sur ATM.

Abstract

As the system complexity grows there is a need for new methods to handle large system design. One way to manage that complexity is to rise the level of abstraction of the specifications by using system level description languages. On the other side, as the level of abstraction rise the gap between the concepts used for the specifications at the system level (communication channels, interacting processes, data types) and those used for hardware synthesis becomes wider. Although these languages are well suited for the specification and validation of complex real time distributed systems, the concepts manipulated are not easy to map onto hardware description languages. It is thus necessary to defines methods for system level synthesis enabling efficient synthesis from system level specifications. The subject of this thesis is the presentation of a new approach of generation of C and VHDL code from system level specifications in SDL. This approach solves the main problem encountered by previous approach : inter process communications. SDL communication can be translated in VHDL for synthesis. This is achieved by the use of a powerful intermediate form that support the modelling for synthesis of a wide range of communication schemes. This intermediate form allows to apply to the system a set of transformations in order to obtain the desired solution. The main refinement step, called communication synthesis is aimed at fixing the protocol and interface used by the different processes to communicate. The refined specification can be translated in C and VHDL and synthesised by commercial tools. We illustrate the feasibility of this approach through an application to a telecommunication example : the TCP/IP over ATM protocol.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

INTRODUCTION

Dans ce chapitre, les motivations et les objectifs de cette thèse seront exposés. La contribution de cette thèse sera exposée. Finalement un plan de la thèse sera présenté.

1.1 MOTIVATIONS

L'augmentation de complexité des circuits a entrainé le développement de nouvelles méthodes de conception permettant d'accroître le niveau d'abstraction des spécifications en utilisant des langages de description au niveau système. Ce mouvement a été motivé par la complexité toujours croissante des systèmes électroniques et la nécessité d'approches unifiées permettant le développement de systèmes contenant à la fois du logiciel et du matériel. Cette unification des parties logicielles et matérielles au sein d'un langage de spécification unique permet un développement conjoint qui élimine les problèmes d'intégration des différentes parties d'un système. D'un autre coté, l'élévation du niveau d'abstraction des spécifications entraîne une augmentation du fossé entre les concepts utilisés pour les spécifications au niveau système et ceux utilisés pour la synthèse de matériel. Bien que les langages systèmes soient adaptés à la spécification et à la validation de systèmes complexes et temps réel, les concepts qu'ils manipulent ne sont pas facilement représentables dans les langages de description matériel. Il devient donc nécessaire de définir de nouvelles méthodes de synthèse système permettant d'obtenir une synthèse efficace à partir de spécifications systèmes.

Le but de ces travaux est de développer une approche et des outils permettant de maîtriser le niveau d'abstraction de spécifications systèmes en terme de synthèse, c'est à dire de pouvoir obtenir une implémentation matérielle efficace à partir de spécifications systèmes.

1.2 OBJECTIF

Cette thèse s'inscrit dans le projet COSMOS qui est un outil et une méthodologie pour la synthèse de systèmes mixtes logiciel/matériel à partir de spécifications systèmes. Cette méthodologie est plus particulièrement orientée vers la conception de systèmes distribués. L'architecture cible peut être un circuit intégrant plusieurs processeurs programmables ou cablés [69], [113], [141] [145], ou un réseau de processeurs. Le domaine d'application visé est celui des télécommunications. COSMOS utilise le langage SDL qui est utilisé pour la spécification et la validation des protocoles de télécommunication. COSMOS repose sur une forme intermédiaire, appelée SOLAR, qui est utilisée pour réaliser les différentes étapes de la synthèse. La méthodologie implémentée est interactive et basée sur des raffinements incrémentaux de la spécification initiale pour arriver à une architecture finale distribuée composée de processeurs logiciels (code C) et matériels (code VHDL). Cette approche doit permettre :

- une interaction entre le concepteur et les outils afin de pouvoir utiliser l'expérience de celui-ci. Dans cette approche le concepteur commence la synthèse avec une spécification initiale et une solution d'architecture en tête. Il dispose alors d'un ensemble de primitives lui permettant de transformer la spécification en une architecture distribuée correspondant à la solution d'architecture.
- la réutilisation de blocs déjà existants par l'intermédiaire de bibliothèques [86]. Ces blocs peuvent être des unités fonctionnelles, des unités de communication ou des sous systèmes complets. La réutilisation permet d'augmenter la productivité en réutilisant des blocs déjà synthétisés et validés. Ces blocs doivent pouvoir être abstraits pour être intégrés dans la spécification système de façon transparente quel que soit leur niveau d'abstraction.

Cette thèse s'articulera autour des deux points suivants :

- la spécification au niveau système pour la synthèse. Il s'agit d'utiliser le langage SDL pour la spécification de systèmes distribués.
- la synthèse de la communication. L'abstraction des communications dans les langages de spécification système nous impose, si l'on veut obtenir une synthèse efficace, de prévoir une étape de raffinement concernant la communication. On développera une approche interactive basée sur une bibliothèque de communication.

1.3 ETAT DE L'ART

De nombreuses approches de co-design ont été proposées : Codes [22], Castle [27], [28], Chinook [29], Polis [30], Cosyma [47] [70], SpecSyn [54], Vulcan [61] [63], Ptolemy [82], Cobra [88], Comet [87], Parnassus [100], Coware [122], Siera [128], Co-saw [131], Paragon [149]. La plupart des approches de co-design peuvent être classées en fonction de l'architecture cible :

1. les architectures mono-processeurs. Dans ce cas l'architecture cible est composée d'un microprocesseur exécutant le logiciel et d'un ensemble de co-processeurs. Les étapes de codesign sont la décomposition fonctionnelle, l'ordonnancement et la synthèse des interfaces. Polis, Vulcan, Cosyma sont des exemples de cette approche.
2. les architectures multi-processeurs. Dans ce cas, le co-design consiste à distribuer la spécification initiale sur un ensemble de processeurs. Les différentes étapes sont la décomposition fonctionnelle, l'allocation de processeur, l'ordonnacement et la synthèse de la communication. SpecSyn, Ptolemy, Coware, Siera sont des exemples de cette approche.

Toutes ces méthodologies implémentent des méthodes automatiques. Aucune ne permet d'intervention du concepteur, ni dans la phase de partitionnement ou de synthèse de la communication.

Seules quelques approches [28], [30] utilisent comme langage de spécification un langage de spécification pour les systèmes distribués. La plupart des approches utilisent le langage C ou un dérivé, VHDL ou un langage synchrone tel que StateChart ou ESTEREL.

1.4 CONTRIBUTION

La principale contribution de cette thèse est le développement d'une approche interactive de co-design [95] basée sur une spécification exécutable SDL (*Spécification and Description Language*) et la démonstration de la validité de cette approche sur un exemple de télécommunication.

La première partie étudie la transposition des concepts des langages de spécification systèmes en ceux des langages de description matériels. Un outil de traduction a été développé. Il traduit une spécification SDL dans la forme intermédiaire SOLAR. Les concepts SDL sont transposés dans leur équivalent SOLAR et plusieurs transformations de la description SDL sont réalisées. Le modèle de traduction utilisé permet l'utilisation des outils de partitionnement et de synthèse de la communication développés dans COSMOS.

Dans un deuxième temps la forme intermédiaire est raffinée pour transformer la spécification initiale en un modèle proche de l'architecture. Les raffinements abordés dans ce travail concernent

la communication. Un outil de synthèse de la communication a été développé. Il est composé de trois primitives interactives appelées par le concepteur et repose sur une bibliothèque de modèles de communication. Il permet de transformer un système communicant à l'aide de schémas de communication de haut niveau en un ensemble de processeurs interconnectés à travers des bus et des signaux.

Notre méthodologie a été appliquée à un exemple réel du domaine des télécommunications : un réseau ATM (*Asynchronous Transfert Mode*) modulaire. Il s'agira de partir d'une spécification SDL pour générer un prototype virtuel C/VHDL.

1.5 PLAN DE LA THÈSE

Le chapitre 2 détaille la forme intermédiaire utilisée pour la synthèse. Ce modèle, capable de représenter aussi bien les concepts systèmes que les concepts matériels et logiciels est utilisé comme format d'unification des spéfications et comme forme intermédiaire lors de la synthèse.

Le chapitre 3 présente les différents types de langages utilisés pour la spécification système. Tous ne sont pas utilisés pour le codesign mais font l'objet de recherche dans des domaines tel que la spécification et la validation ou la génération de code.

Le chapitre 4 détaille notre approche de la synthèse de la communication. Cette approche vise à réduire le fossé entre les concepts utilisés pour la spécification des communication au niveau système et ceux utilisés pour l'implémentation. Cette approche repose sur une bibliothèque de modèles de communication.

Le chapitre 5 étudie la mise en correspondance des concepts SDL et SOLAR pour la génération de code C/VHDL. Ce chapitre présente une approche qui permet de résoudre le problème de l'abstraction de la communication en SDL lors de la génération de code VHDL synthétisable.

Le chapitre 6 est consacré à l'étude d'une application de notre méthodologie sur un système de télécommunication. Le système sera spécifié en SDL, simulé et validé avant d'être utilisé dans COSMOS. Un prototype virtuel C/VHDL sera généré et pourra etre cosimulé pour vérifier la conformité avec la spécification SDL.

Le chapitre 7 présente les conclusions et perspectives relatives aux travaux menés dans le cadre de cette thèse.

Chapitre 2

MODÈLES POUR LA SYNTHÈSE AU NIVEAU SYSTÈME

Dans ce chapitre nous présentons COSMOS, un environnement de codesign pour la spécification et la synthèse de systèmes mixtes pouvant contenir à la fois du logiciel et du matériel. COSMOS utilise une spécification système pour produire une architecture C/VHDL distribuée. COSMOS repose sur une forme intermédiaire permettant d'unifier des spécifications provenant de langages de description matériel ou logiciel. La forme intermédiaire, appelée SOLAR, sera présentée. Le modèle de description du comportement dans SOLAR est une extension du modèle des machines d'états finis. Les extensions concernent le parallélisme, la hiérarchie, la communication entre machines d'états finis et le traitement des exceptions.

2.1 INTRODUCTION

Dans ce chapitre nous présentons les différentes étapes permettant de transformer une spécification système en une architecture distribuée composée de modules logiciels et matériels. La figure 2.1 représente le flôt COSMOS en partant d'une spécification SDL pour arriver à une architecture finale.

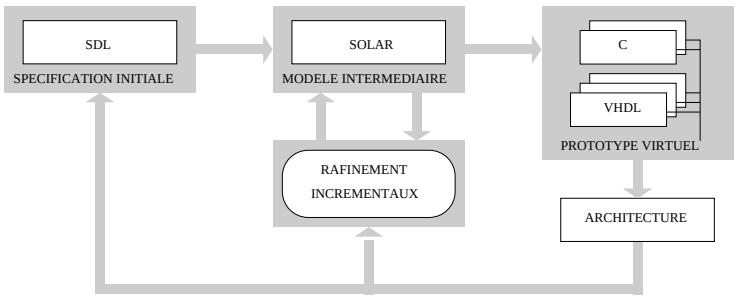


FIG. 2.1: Flôt de conception du système COSMOS

La forme intermédiaire SOLAR sera présentée. Utilisée comme format d'unification de spécifications au niveau système, cette forme intermédiaire doit pouvoir représenter les concepts utilisés dans les langages de description matériels, logiciels et de spécifications systèmes. SOLAR est utilisé comme forme intermédiaire sur laquelle sont effectuées les étapes de raffinement successives.

2.2 LA FORME INTERMÉDIAIRE SOLAR

2.2.1 Introduction

Dans cette section la forme intermédiaire SOLAR est présentée. SOLAR est basé sur le modèle des machines d'états finis étendues communicantes. Ce modèle est suffisamment général pour permettre de représenter les concepts utilisés dans les langages systèmes. D'autres constructeurs ont été ajoutés afin de permettre la spécification de systèmes hiérarchiques et la communication entre processus. La communication entre sous-systèmes peut être effectuée de deux façons différentes :

- La première à travers le concept classique de signal (*net*).
- La seconde à travers le concept de canal de communication (*channel unit*) qui permet de spécifier des protocoles avec différents degrés de complexité.

SOLAR utilise quatre concepts principaux :

2.2.2 La table d'état

Le constructeur de base permettant la description de comportement en SOLAR est la table d'état ou *statetable*. La table d'état étend le concept de machine d'états finis en introduisant le parallélisme, la hiérachie, les transitions globales et les exceptions. Une table d'état peut être constituée d'une combinaison d'états (*state*) ou d'autres tables d'états. Cette combinaison peut être

parallèle, séquentielle ou les deux en même temps. Chaque état peut être hiérarchique et contenir une ou plusieurs tables d'états.

Une table d'état peut comporter des exceptions et des transitions globales. Une exception représente une action à effectuer quel que soit l'état courant. L'état suivant par défaut est utilisé dans le cas ou l'état suivant n'est pas spécifié explicitement. L'état d'initialisation indique l'état de départ de la machine d'états finis. Les transitions globales permettent d'effectuer des transitions entre états indépendamment de leurs positions respectives dans la hiérarchie. Les transitions sont permises entre deux machines d'états finis indépendamment de leur positions respectives dans la hiérarchie. En d'autres termes, les transitions globales peuvent franchir la hiérarchie et les machines d'états finis.

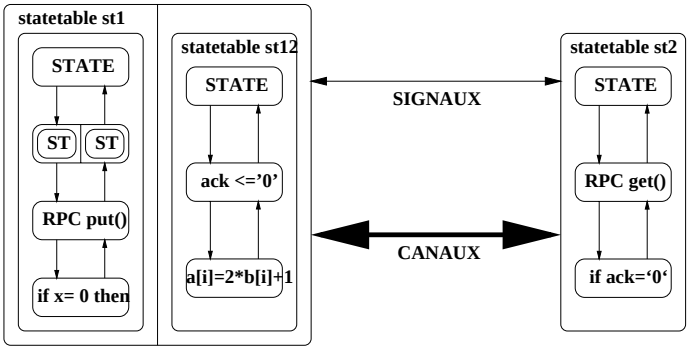


FIG. 2.2: Machines d'états finis hiérarchiques communicantes

La figure 2.2 représente deux automates d'états finis communicants. La première table d'état *st1* est composée de deux machines d'états finis parallèles. Chaque état de l'automate peut contenir un ensemble d'opérations de controle, d'opérations sur des données, des affectations de signaux, des appels à des procédures des communications ou d'autres automates d'états finis. Les deux machines d'états finis *st1* et *st2* sont interconnectées par des signaux et des canaux de communication.

2.2.3 Le canal de communication

SOLAR repose sur un modèle de communication puissant permettant de modéliser la plupart des schémas de communication matériels et système. Au niveau système la communication est effectuée à l'aide de canaux de communication. Un canal offre un ensemble de primitives de communication qui sont appelées par les processus pour communiquer (figure 2.4. Ces primitives sont invoquées par l'intermédiaire d'appel de procédure à distance de ces primitives. Un processus qui désire communiquer invoque une des primitives de communication offerte par le canal. Une fois l'appel de procédure à distance effectué, la communication est réalisée par le canal.

Un canal est composé d'un ensemble de primitives de communication, d'une interface et d'un contrôleur éventuel (figure 2.3). Le contrôleur assure la résolution des conflits d'accès au canal. La complexité du contrôleur peut varier du simple arbitreur de bus à un protocole en couche à une file d'attente *first-in-first-out*. Les primitives lisent et écrivent sur les ports de l'interface pour échanger

les données fournies en paramètre avec le canal. En d'autres termes, les primitives de communication encapsulent le protocole d'accès au canal.

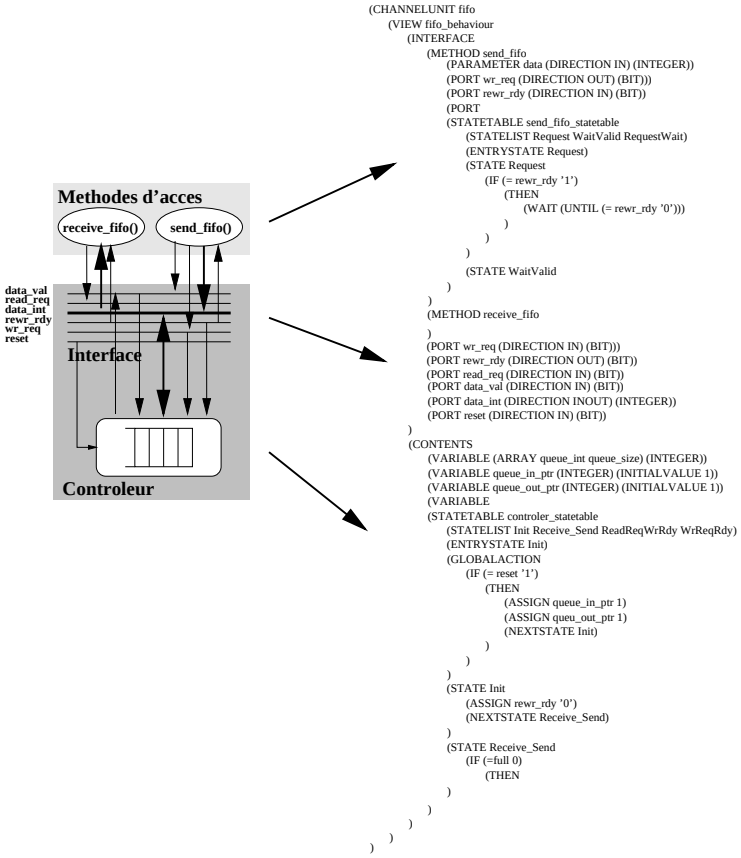


FIG. 2.3: Le canal SOLAR

Le concept de canal de communication est similaire au concept d'objet utilisé dans les langages de programmation. Un canal est un objet capable d'exécuter une communication avec un protocole défini. Un processus qui désire communiquer invoque une des primitives de communication offerte par l'intermédiaire d'un appel de procédure à distance. La primitive est exécutée par le canal indépendamment du processus appelant. La communication est transparente pour le processus appelant.

Le protocole de la communication est distribué entre les primitives de communication et le contrôleur. Lorsqu'il n'y a pas de controleur, le protocole est complètement distribué entre les primitives de communication. Cela peut être le cas d'un protocole *handshake*.

```
(CHANNELUNIT  fifo
   (VIEW  fifo_behaviour
      (INTERFACE
         (PORT wr_req (DIRECTION  IN ) (BIT ))
         (PORT rewr_rdy (DIRECTION  OUT ) (BIT ))
         (PORT read_req (DIRECTION  IN ) (BIT ))
         (PORT data_val (DIRECTION  IN ) (BIT ))
         (PORT data_int (DIRECTION  INOUT ) (INTEGER ))
         (PORT reset (DIRECTION  IN ) (BIT ))
         (METHOD  send_fifo
            (PARAMETER  data  (DIRECTION  IN ) (INTEGER ))
            (PORT wr_req (DIRECTION  OUT ) (BIT ))
```

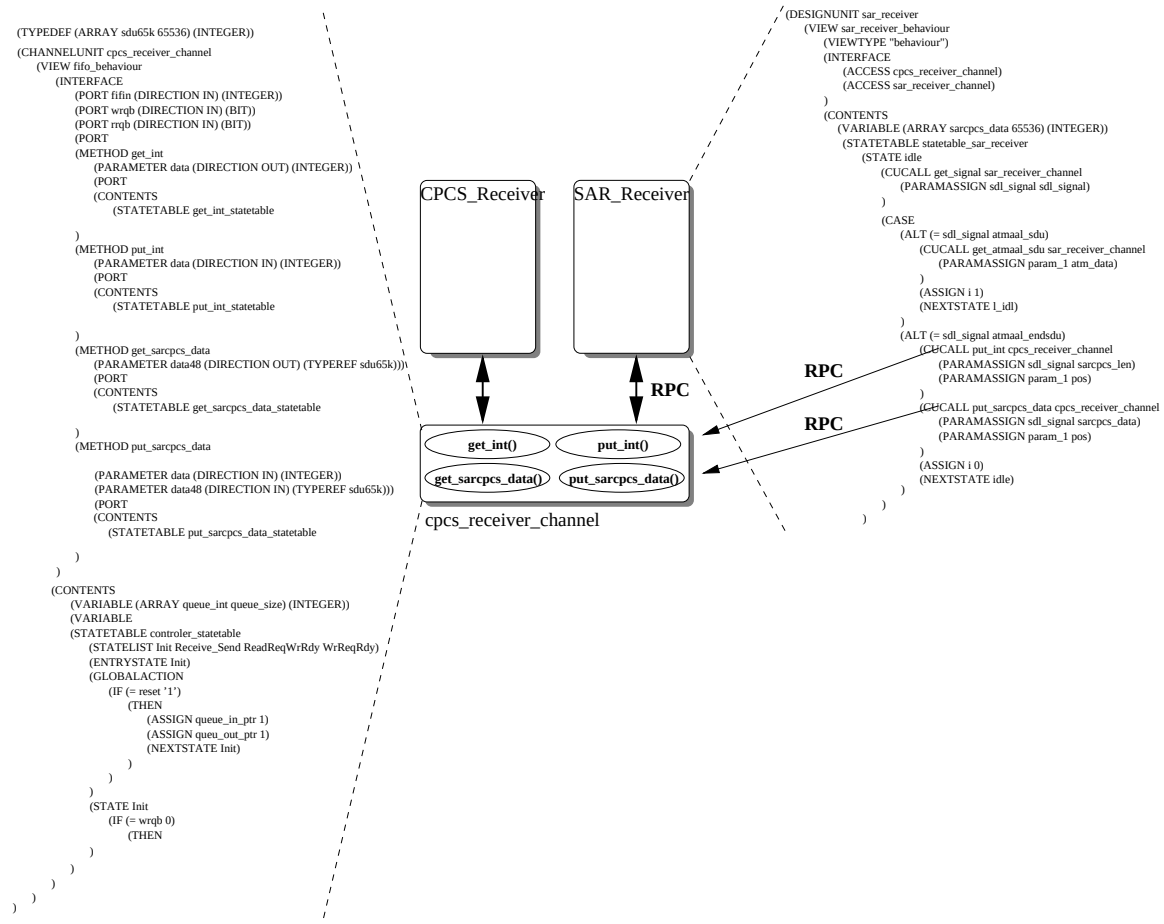


FIG. 2.4: Communication à travers le canal SOLAR

```
(PORT rewr_rdy (DIRECTION  IN ) (BIT ))
(PORT data_val (DIRECTION  OUT ) (BIT ))
(PORT data_int (DIRECTION  OUT ) (INTEGER ))
(STATETABLE  sendInteger_statetable
   (STATELIST  request  waitvalid  requestwait )
   (ENTRYSTATE  request )
   (STATE  request
      (IF (=  rewr_rdy  '1')
         (THEN
            (WAIT  (UNTIL (=  rewr_rdy  '0')))
         )
      )
      (ASSIGN  wr_req  '1')
      (NEXTSTATE   waitvalid )
   )
   (STATE   waitvalid
      (WAIT  (UNTIL (=  rewr_rdy  '1')))
      (ASSIGN  wr_req  '0')
      (ASSIGN  data_int  data )
      (ASSIGN  data_val  '1')
      (NEXTSTATE   requestwait )
   )
   (STATE  requestwait
      (WAIT  (UNTIL (=  rewr_rdy  '0')))
      (ASSIGN  data_val  '0')
   )
   )
)
(METHOD  receive_fifo
   (PARAMETER  data  (DIRECTION  OUT ) (INTEGER ))
   (PORT read_req (DIRECTION  OUT ) (BIT ))
   (PORT rewr_rdy (DIRECTION  IN ) (BIT ))
   (PORT data_int (DIRECTION  IN ) (INTEGER ))
   (STATETABLE  receiveInteger_statetable
      (STATELIST  request  requestwait )
      (ENTRYSTATE  request )
      (STATE  request
         (IF (=  rewr_rdy  '1')
            (THEN
               (WAIT  (UNTIL (=  rewr_rdy  '0')))
            )
         )
         (ASSIGN  read_req  '1')
         (NEXTSTATE   requestwait )
      )
      (STATE  requestwait
         (WAIT  (UNTIL (=  rewr_rdy  '1')))
         (ASSIGN  data_int  data_int )
         (ASSIGN  read_req  '0')
      )
   )
)
```

```
)
(CONTENTS
   (VARIABLE (ARRAY  queue_int  queue_size )(INTEGER ))
   (VARIABLE  queue_in_ptr (INTEGER )(INITIALVALUE  1 ) )
   (VARIABLE  queue_out_ptr (INTEGER )(INITIALVALUE  1 ) )
   (VARIABLE  full (INTEGER )(INITIALVALUE  0 ) )
   (VARIABLE  empty (INTEGER )(INITIALVALUE  1 ) )
   (CONSTANT  queue_size (INTEGER )(INITIALVALUE  16 ) )
   (STATETABLE  controller_statetable
      (STATELIST  Init  Receive_Send  ReadReqWrRdy  WrReqRdy )
      (ENTRYSTATE  Init )
      (GLOBALACTION
         (IF (=  reset  1 )
            (THEN
               (ASSIGN  queue_in_ptr  1 )
               (ASSIGN  queue_out_ptr  1 )
               (NEXTSTATE   Init )
            )
         )
      )
      (STATE  Init
         (ASSIGN  rewr_rdy  '0')
         (NEXTSTATE   Receive_Send )
      )
      (STATE  Receive_Send
         (IF (=  full  0 )
            (THEN
               (IF (= wr_req  '1')
                  (THEN
                     (ASSIGN rewr_rdy '1')
                (WAIT  (UNTIL (= data_val '1')))
                (ASSIGN (MEMBER queue_int queue_in_ptr) data_int))
                     (ASSIGN rewr_rdy '0')
                     (NEXTSTATE   WrReqRdy )
                  )
               )
            )
         )
         (IF (=  empty  0 )
            (THEN
               (IF (= read_req '1')
                  (THEN
                     (ASSIGN data_int (MEMBER queue_int queue_out_ptr)) )
                     (ASSIGN rewr_rdy '1')                          )
                     (NEXTSTATE   ReadReqWrRdy )
                  )
               )
            )
         )
         (NEXTSTATE   Receive_Send )
      )
      (STATE  WrReqRdy
         (WAIT  (UNTIL (= data_val '0')))
         (ASSIGN  queue_in_ptr (+  1  queue_in_ptr ))
         (IF (>  queue_in_ptr  queue_size )
            (THEN
               (ASSIGN   queue_in_ptr  1 )
            )
         )
         (ASSIGN  empty  0 )
         (IF (=  queue_in_ptr  queue_out_ptr )
            (THEN
               (ASSIGN  full  1 )
            )
         )
         (NEXTSTATE   Receive_Send )
      )
      (STATE  ReadReqWrRdy
         (WAIT  (UNTIL (= read_req  '0')))
         (ASSIGN rewr_rdy  indice_r '0')
         (ASSIGN  queue_out_ptr (+  1  queue_out_ptr ))
         (IF (>  queue_out_ptr  queue_size )
            (THEN
               (ASSIGN   queue_out_ptr  1 )
            )
         )
         (ASSIGN  full  0 )
         (IF (=  queue_in_ptr  queue_out_ptr )
            (THEN
               (ASSIGN  empty  1 )
            )
         )
         (NEXTSTATE   Receive_Send )
      )
   )
)
```

FIG. 2.5: Le canal de communication SOLAR

La figure 2.5 représente un canal de communication en SOLAR. L'interface contient les ports nécessaire aux interconnexions ainsi que les procédures de communications (*method*) offertes. Ces procédures accèdent à l'interface du canal avec le protocole requis pour lui transmettre le paramètre entier (*data*). Le contrôleur de communication (*contents*) contient la file d'attente d'écrite sous forme d'un automate d'états finis.

Ce modèle offre une sémantique claire et permet de séparer les aspects de communication et de comportement d'une spécification. Cela permet de spécifier une communication entre processus à l'aide de primitives de haut niveau sans faire d'hypothèse sur la réalisation de cette communication.

2.2.4 L'unité de conception

L'unité de conception en SOLAR permet la structuration d'un système en un ensemble de sous-systèmes interagissants. Chaque sous système interagit avec l'extérieur à travers des frontières bien définies. Les unités de conception peuvent être structurelles ou comportementales. Une unité de conception structurelle peut contenir d'autres unités de conception structurelles ou comportementales, permettant la hiérarchie et/ou un ensemble d'interconnections de signaux et canaux. Une

unité de conception comportementale peut contenir une ou plusieurs tables d'états décrivant le comportement. La communication entre sous-systèmes peut être réalisée de deux façons :

- La première à travers le concept classique de signal, dans lequel un signal transmet ou reçoit des données dans une ou deux directions.
- La seconde à travers le concept de canaux de communication. Le canal de communication permet la spécification de protocole avec divers degrés de complexité.

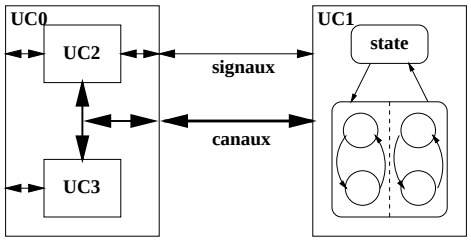


FIG. 2.6: Unité de conception structurelle et comportementale

La figure 2.6 représente un système SOLAR composé de deux unités de conception au niveau le plus haut de la hiérarchie (*UC0* et *UC1*). *UC0* est une unité comportementale comportant une sous structure composée de *UC2*, *UC3*. *UC1* est une unité comportementale contenant une table d'état.

2.2.5 L'unité fonctionnelle

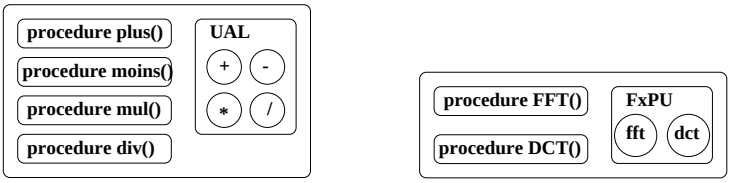
L'unité fonctionnelle SOLAR est l'analogue pour les données du canal pour la communication. Une unité fonctionnelle est un chemin de donnée spécifique capable d'exécuter des opérations sur les données (figure 2.7). Le terme d'unité fonctionnelle [79] regroupe aussi bien celui d'unité arithmétique et logique (+, -, *) (figure 2.7.1) que celui de coprocesseur pouvant exécuter des opérations complexes (/, FFT, DCT) (figure 2.7.2).

Du point de vue conceptuel, une unité fonctionnelle est un objet, au sens des langages de programmation objet, capable d'exécuter une ou plusieurs opérations sur des données. L'unité fonctionnelle offre une ou plusieurs primitives permettant d'accéder aux opérations qu'elle est capable d'exécuter. Le modèle de l'unité fonctionnelle repose sur le principe d'appel de procédure à distance des primitives. L'accès aux opérations d' une unité fonctionnelle s'effectue uniquement en invoquant ces primitives.

Les opérations sont invoquées par l'intermédiaire des procédures qui transmettent les paramètres à l'unité fonctionnelle et renvoient les résultats au processus appelant. ces procédures encapsulent le protocole d'accès à l'unité fonctionnelle rendant son accès transparent au processus appelant. Les opérations sur les données transmises en paramètre sont effectuées par l'unité fonctionnelle.

2.2.6 Modèle d'exécution SOLAR

Il n'y pas de modèle d'éxécution SOLAR à propement parler. SOLAR est une forme intemédiaire utlisée pour la synthèse. Il n'existe pas de simulateur SOLAR. La sémantique d'exécution



2.7.1: Unité fonctionnelle simple

2.7.2: unité fonctionnelle complexe

FIG. 2.7: Unité fonctionnelle

d'un système SOLAR n'est définie que par le modèle de traduction SOLAR → C/VHDL. C'est la traduction d'une instruction SOLAR (*WAIT*) en VHDL (*WAIT*) qui définit le modèle d'exécution de cette instruction SOLAR.

2.3 ÉTAPES DE CONCEPTION DU SYSTÈME COSMOS

COSMOS utilise une approche transformationnelle pour le raffinement d'une spécification système en un modèle C/VHDL. La figure 2.8 représente les différentes étapes de conception dans le système COSMOS. Chaque étape réduit le fossé entre la spécification et la réalisation en fixant certains détails d'implémentation ou en préparant d'autres transformations. La co-synthèse permet par un ensemble de raffinements successifs de transformer la spécification initiale en une solution d'architecture désirée. La première étape est celle de la saisie des spécifications systèmes. Cette spécification est ensuite traduite dans un modèle équivalent SOLAR pour y subir un ensemble d'étapes de raffinement. La seconde étape est le partitionnement, suivie de la synthèse de la communication. Viennent ensuite les étapes de génération de code C/VHDL et de génération d'architecture.

2.3.1 Saisie des spécifications

La première étape est celle de la saisie des spécifications dans un ou plusieurs langages. Actuellement aucun langage ne permet d'englober tous les concepts manipulés dans une application complexe. Différents langages doivent être utilisés en fonction de l'application. Les applications orientées protocole de télécommunication vont utiliser des langages tel que SDL [124], SDL-92 [48], LOTOS [17], [18], ESTELLE. Les applications temps réel utlisent des langages synchrones tel que SIGNAL [64], LUSTRE [65] pour les applications dominées par le flôt de données ou ESTEREL [20], StateChart, [68] pour les applications orientées flôt de contrôle. Certains langages tel que SDL, LOTOS permettent de définir des types de données et opérateurs dans un langage tel que C agissant sur ce données. L'utilisateur dispose ainsi de toute la puissance du langage pour manipuler les données.

Les différentes parties d'un système pouront être décrites chacunes dans le langage le mieux adapté et unifiées dans la forme intermédiaire SOLAR. Cette approche est adoptée dans le cas des systèmes avioniques [120], [121]. Au niveau système, les spécifications peuvent être validées par

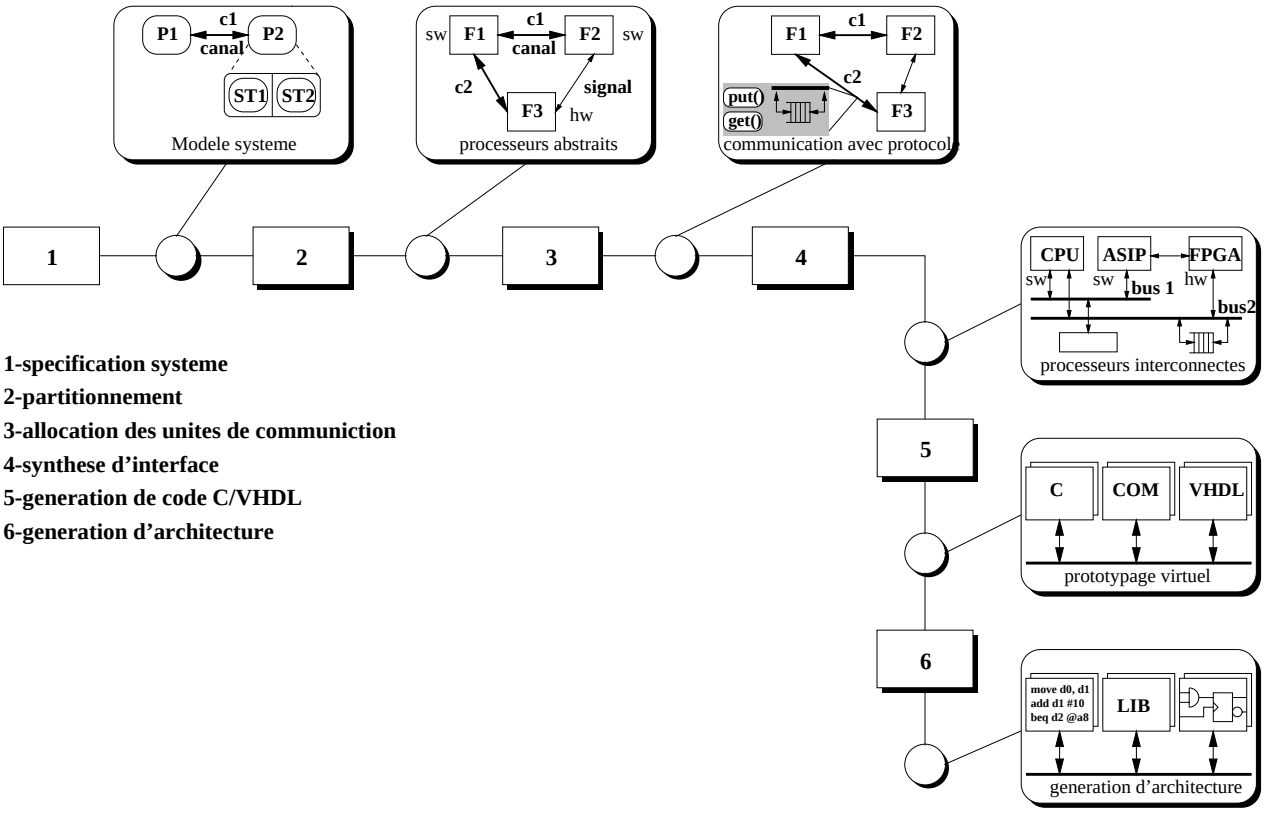


FIG. 2.8: Méthodologie de conception COSMOS

simulation ou de façon formelle. Dans le cas de spécifications multi-langages, celles ci peuvent être validées par co-simulation au niveau système. Une approche de cosimulation au niveau C-VHDL a été développée dans [140]. La validation formelle d'une spécification système multi-langage pose le problème de la composition de formalismes différents au sein d'un même outil de preuve, en particulier la composition de schémas de communication différents. La vérification formelle d'une spécification multi-langage devra faire la preuve d'une communication correcte entre des systèmes décrits dans des langages différents. Le problème qui se posera sera celui de l'interface entre des schémas de communication qui peuvent être extrêmement différents d'un langage à l'autre.

A ce niveau, on s'attache à décrire les fonctionnalités indépendamment de leur implémentation. Les langages systèmes offrent un ensemble de concepts bien définis et sont donc adaptés à la spécification et à la validation de systèmes complexes. Les principaux concepts manipulés sont le comportement et la communication. Leur niveau d'abstraction élevé leur permet de faire abstraction des détails d'implémentation. Une spécification au niveau système se concentre sur la fonctionnalité d'un système et non sa réalisation. C'est l'étape de co-synthèse qui determinera l'implémentation de chaque partie du système. Retarder ainsi les choix d'implémentation permet de n'exclure aucune réalisation valide du système. Une exploration d'architecture permet d'arriver à une solution satisfaisant les contraintes du système.

La validation d'une spécification au niveau système se concentre donc sur la fonctionnalité de celui-ci. Une séparation entre réalisation et fonctionnalité permet de valider cette dernière séparemment. La validation de l'implémentation sera faite après l'étape de co-synthèse par cosimulation C-VHDL. Cette séparation des problèmes permet de traiter chacun d'eux avec les méthodes appropriées (simulation ou preuve formelle).

2.3.2 Partitionnement

Le but du partitionnement au niveau système [134] est de diviser un système en un ensemble de partitions ou chaque partition s'exécutera soit en logiciel soit en matériel. Chacune des différentes partitions contiendra une ou plusieurs fonctions provenant de la spécification initiale. Chaque partition affectée à un réalisation logicielle s'exécutera sur un processeur standard, un DSP (*Digital Signal Processor*) ou un ASIP (*Application specific Integrated Processor*). Les partitions affectées á une réalisation matérielle seront implémentées sous forme de FPGA (*Field Programmable Gate Array*) ou d'ASIC (*Application Specific Integrated Circuit*). Le nombre de partitions générées représente le nombre de processeurs logiciel ou matériel de l'architecture cible. Les questions qui se posent au concepteur sont les suivantes :

- combien de partitions sont nécessaires pour exécuter une description donnée ?
- quelle doit être la réalisation (logicielle ou matérielle) de chaque partition ?
- dans quelle partition exécuter chaque fonction de la spécification ?
- quel est le surcoût de communication introduit par le partitionnement [77] ?
- Quel est le gain en terme de performance, surface et communication obtenu par une opération particulière de partitionnement ?

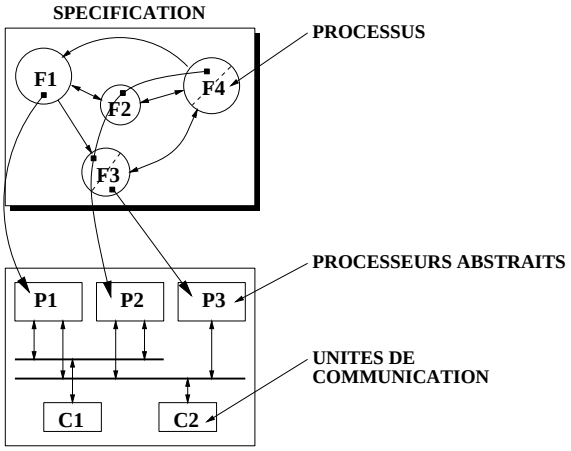


FIG. 2.9: Partitionnement pour les architectures multiprocesseurs

Le partitionnement réalise un compromis entre le nombre de partitions matérielles, le temps d'exécution et les communications entre partitions. En général les partitions qui se trouvent sur

le chemin critique de l'application et qui demandent un temps de réponse court sont réalisées en matériel. Les autres partitions sont réalisées en logiciel. Lors de l'opération de partitionnement, une fonction peut être distribuée sur plusieurs processeurs et plusieurs fonctions peuvent être affectées à un seul processeur. Sur la figure 2.9, la fonction *F3* est divisée entre les processeurs *P2* et *P3*. La fonction *F2*, une partie de *F3* et *F4* sont affectées au processeur *P2*. La fonction *F1* est entièrement affectée au processeur *P1*. *C1* et *C2* désignent des unités de communication.

Il existe actuellement trois approches du partitionnement :

- La première approche [47] commence avec une spécification entièrement logicielle et effectue une migration de certaines parties du code vers le matériel. Les parties critiques du système sont identifiées et réalisées en matériel.
- La seconde [61] commence avec une spécification matérielle. Les parties non critiques sont identifiées et affectées à une implémentation logicielle, réduisant ainsi le coût de l'application.
- La troisième est l'approche système [54]. Elle ne se limite pas à un type de spécification en entrée. Dans cette approche, les différentes fonctionnalités d'une spécification sont affectées à des réalisations logicielles ou matérielles en fonction de contraintes telles que le temps de réponse ou la surface du matériel.

L'architecture cible est une architecture multiprocesseur, chaque processeur pouvant être un processeur logiciel ou matériel. Les méthodes de partitionnement proposées dans [61] [71] supposent une architecture composée d'un processeur, d'un coprocesseur et d'une mémoire partagée. D'autres méthodes telles que [128], [131] prennent en considération un seul processeur avec plusieurs coprocesseurs. Certaines approches sont orientées vers les systèmes dominés par le flot de données tel que le traitement de signal [82], [142].

Le partitionnement dans COSMOS est orienté vers les applications dominées par le flôt de contrôle. Le partitionnement comportemental repose sur le découpage et la fusion de machines d'états finis étendue. La spécification d'entrée est une machine d'états finis étendue composée de machines séquentielles et parallèles. Le résultat du partitionnement est un ensemble de partitions contenant l'automate d'états finis.

L'approche de partitionnement adoptée est interactive en raison de la complexité des systèmes à partitionner. Le problème du partitionnement est NP-complet. Il est donc difficile de réaliser une approche entièrement automatique [7], [85] [109]. Un certain nombre d'heuristiques ont néanmoins été proposées [5], [47], [62], [73], [83], [84], [112], [136]. La décision de ne pas développer une approche automatique a été prise en raison de la difficulté à réaliser des estimations fiables au niveau système [135] indépendantes du domaine d'application. Le but est de combiner à la fois une approche automatique et manuelle. L'approche proposée dans COSMOS [13], [95] suppose que le concepteur commence la synthèse à partir d'une spécification système et une solution d'architecture. L'environnement offre au concepteur un ensemble de primitives de transformation lui permettant d'arriver à la solution d'architecture désirée par un ensemble de raffinements successifs. Tous les raffinements sont réalisés automatiquement. Cela permet une grande accélération du cycle de conception. Néanmoins toutes les décisions sont prises par le concepteur qui utilise son expérience pour converger vers une solution efficace.

Un environnement de type boite à outil est à la disposition du concepteur. Il comprend un ensemble de primitives de partitionnement *move*, *merge*, *split*, *cut*, *flat*.

La primitive *move* permet de déplacer un état ou une table d'état à travers la hiérarchie d'une machine d'états finis. Cette opération permet de déplacer des états en vue d'une réalistation logicielle ou matérielle.
La primitive *merge* fusionne deux machines séquentielles en une seule. Cette opération permet le partage de ressources telles que des unités fonctionnelles.
La primitive *split* transforme une machine d'états finis en deux machines s'exécutant séquentiellement. Cette primitive permet de découper une machine d'états finis en deux pour assigner une partie à une réalisation logicielle et l'autre à une réalisation matérielle.
La primitive *cut* transforme une machine d'états finis parallèle en un ensemble de machines interconnectées.
La primitive *flat* permet de suprimer la hiérarchie dans une machine d'états finis hiérarchique.

2.3.3 Synthèse de la communication

La synthèse da la communication suit généralement le partitionnement. Le partitionnement produit un ensemble de sous-systèmes communicant à travers un réseau de communication composé de canaux offrant des primitives de communication. Chaque sous système aura à communiquer avec un ou plusieurs autres sous-systèmes en utilisant différents protocoles de communication.

Le point de départ de la synthèse de la communication est un ensemble de processus communicant à travers de canaux via des primitives de haut niveau. Rappelons que dans notre modèle, la spécification de la communication reste séparée du reste du système. L'objectif de la synthèse de la communication est de transformer ce système composé de processus communicant à travers des canaux en un ensemble de processeurs interconnectés communicant à travers des bus et des signaux et partageant le contrôle de la communication.

Dans cette approche, la synthèse de la communication est réalisée en deux étapes qui sont l'allocation des unités de communication ou sélection de protocole et la génération d'interface. La sélection de protocole choisit dans la bibliothèque de communication l'ensemble d'unité de communication offrant l'ensemble des primitives requises par les processus pour communiquer. Cette étape fixe le protocole de chaque communication en choisissant une unité de communication avec un protocole défini pour chaque communication. Cette étape détermine aussi la topologie du réseau d'interconnection tel que le nombre de bus. La synthèse d'interface génère les interfaces et détermine la largeur des bus, les mécanismes d'arbitrage d'accès au bus. Notre approche procède en plusieurs étapes qui évitent de prendre des décisions prématurées, lesquelles risquent de restreindre l'espace de solutions. L'idée est de retarder autant que possible les choix concernant la réalisation physique de la communication puisque des décisions prématurées peuvent éliminer prématurement certaines réalisations valides.

Allocation des unités de communication

Cette étape choisit dans la bibliothèque de communication un ensemble d'unités de communication offrant l'ensemble des primitives de communication requises par les processus. La communication entre les processus peut être réalisée par l'un des shémas décrit dans la bibliothèque. Le choix d'une unité de communication ne dépend pas uniquement du protocole implémenté mais

aussi des performances requises. Cette étape est similaire à l'allocation des unités fonctionnelles en synthèse architecturale [55]. Cette étape détermine le protocole utilisé par chaque communication et le nombre d'unités de communication.

Synthèse d'interface

Cette étape sélectionne une réalisation pour chaque unité de communication dans la bibliothèque d'implémentation et génère les interfaces et interconnections correpondantes pour chaque processus. Le protocole implémenté par chaque unité de communication est distribué entre les différents processus et le contrôleur de communication. Le contrôleur est choisi dans la bibliothèque d'implémentation.

2.3.4 Génération de code C/VHDL

L'étape de prototypage virtuel génère une description exécutable C/VHDL pour chaque processeur résultant du partitionnement. Le code généré est acceptable à la fois par les outils de synthèse et de simulation. Le résultat du prototypage virtuel est une architecture composée de processeurs virtuels logiciels (code C) et matériels (code VHDL). Les codes correspondants aux éléments de communication, qui sont soit des primitives de communication soit des controleurs, sont extraits de la bibliothèque de réalisation des canaux.

2.3.5 Co-simulation logicielle/matérielle

A ce niveau de la synthèse, le système est composé de processeur logiciel décrit en C et de processeur matériels décrit en VHDL. Le but de la co-simulation est de simuler un système mixte contenant des parties décrites en C et en VHDL. Indépendamment du protocole utilisé, les parties matérielles communiquent à travers des bus, des mémoires, des E/S (communication matériel/logiciel) alors que les parties logicielles communiquent à travers des primitives d'E/S (logiciel/matériel) ou en utilisant les IPC (Logiciel/logiciel) (figure 2.10). La sélection d'une unité de communication, qu'elle soit logicielle ou matérielle détermine automatiquement l'interface utilisée pendant la communication. C'est la seule information nécessaire pour la co-simulation [140], [141]. Dans l'approche de COSMOS la co-simulation est encapsulée dans des procédures [12] qui cache les détails d'implémentation. Pour les parties logicielles, il est suffisant d'avoir une implémentation des procédures de communication logicielles/matérielle assurant la communication avec le VHDL. Sur le processeur cible, les opérations d'E/S lisent et écrivent sur les ports du circuit d'interface, pour la co-simulation ces procédures feront appel à l'interface C/VHDL. Pour l'implémentation finale, le code assembleur des routines d'E/S sera utilisé. Avec cette approche il est possible d'utiliser le même code source pour la co-simulation et la co-synthèse. La co-simulation ne nécessite aucune information concernant l'implémentation des parties logicielles et matérielles telle que :

- Le processeur cible pour le code C.
- La technologie sur lequel seront implémentés les parties matérielles.

La co-simulation dans COSMOS repose sur les IPC d'Unix. Les parties matérielles seront validées à l'aide d'un simulateur VHDL et les parties logicielles avec un débogeur C. Les deux parties sont

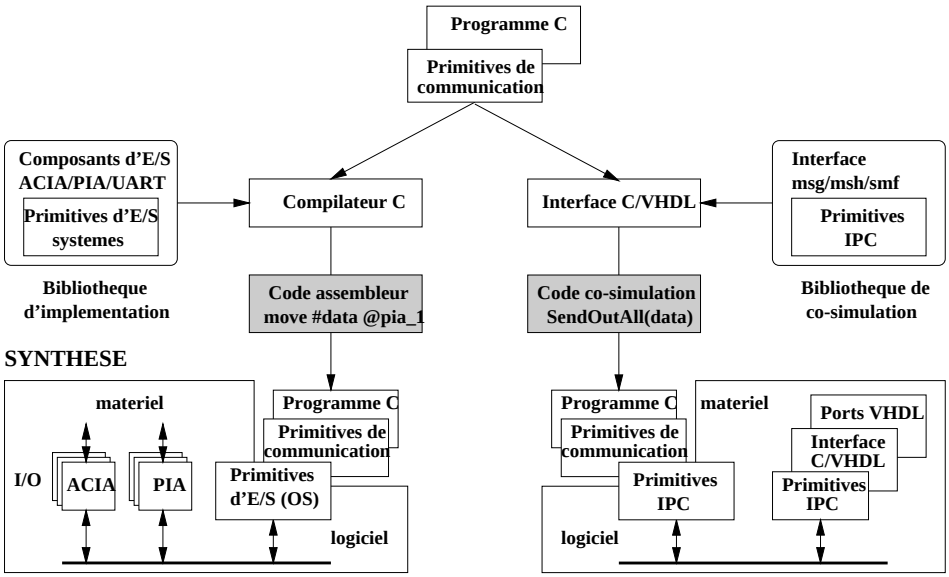


FIG. 2.10: Co-simulation logiciel/matériel

traitées commme des processus Unix communicant à travers les IPC. La figure 2.10 représente l'environnement de co-simulation de COSMOS.

2.3.6 Génération d'architecture

L'étape de génération d'architecture produit une réalisation de la spécification initiale [119]. Cette opération est réalisée en utilisant des complilateurs standard pour les parties logicielles et des outils de synthèse architecturale ou logique pour les parties matérielles. L'architecture générée sera composée de parties logicielles (processeur + code), de parties matérielles (FPGAs, ASICs) et de modules de communication (FIFO, mémoires, arbitreur de bus, IPC, circuit d'E/S, etc.). Cette architecture peut aussi servir au prototypage rapide [24], [88], [128]. Dans [128], l'implémentation est basée sur une bibliothèque de modules logiciels et matériels paramétrables. L'architecture cible proposée dans [82] permet de générer du code pour plusieurs configurations de processeurs tel que des architectures mono-processeurs, ou des architectures parallèles à mémoire partagée ou à passage de message. Un modèle général d'architecture permettant la représentation d'une large gamme d'architectures pour des applications logicielles/matérielles a été défini. Il est représenté en figure 2.11.

Ce modèle d'architecture sert de plateforme sur laquelle l'architecture mixte logicielle/matérielle est transposée. Les modules de communication sont extraits de la bibliothèque. Le modèle d'architecture proposé est suffisamment général pour représenter une large classe de systèmes mixtes logiciels/matériels, y compris des architectures distribuées utilisant plusieurs modèles de communication. Une architecture typique est représentée en figure 2.12. Elle contient plusieurs modules logiciels, matériels et de communication. Les modules de communication se comportent comme des

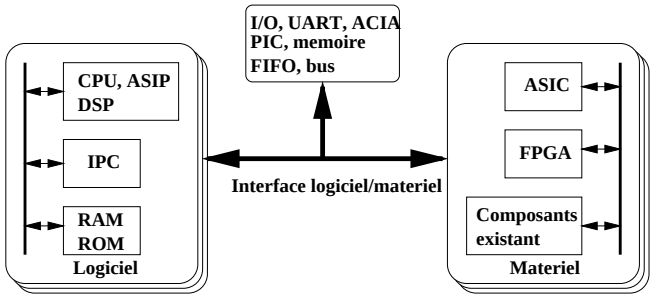


FIG. 2.11: Modèle d'architecture logicielle/matérielle

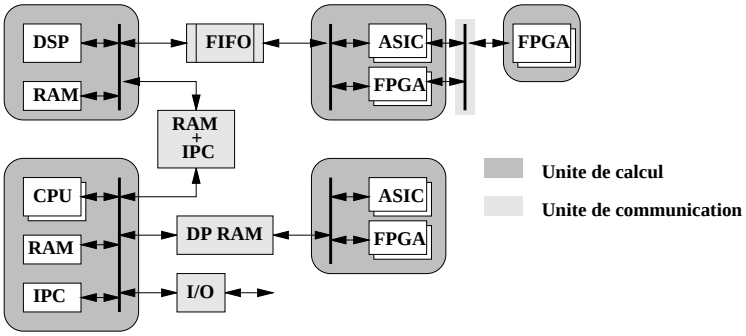


FIG. 2.12: Plateforme logicielle/matérielle

serveurs de communication offrant les primitives de communication requises.

2.4 CONCLUSION

Nous avons présenté dans ce chapitre les principales étapes d'une approche de co-design pour les systèmes mixtes logiciels/matériels basée sur la forme intermédiaire SOLAR. Cette approche part d'une spécification système en SDL pour aboutir à une architecture distribuée composée de modules logiciels (processeur + code C) et matériels (VHDL). Le point clé de cette approche est l'utilistation d'un modèle de communication abstrait et général. La séparation entre unités de calcul et unités de communication permet la réutilisation de modèles de communication existant. Une bibliotèque de communication permet au concepteur de choisir le protocole de communication adapté à son application. Dans la mesure où aucune restriction n'est imposée au modèle de communication, cette approche de co-design peut être appliquée à une large gamme d'applications.

Chapitre 3

LANGAGE DE SPÉCIFICATIONS POUR LA SYNTHÈSE SYSTÈME

Les langages sont utilisés durant la principale phase de conception d'un système : la phase de spécification. Un langage spécifique est choisi en fonction du domaine d'application, de son pouvoir d'expression, et de la disponibilité de méthodes et d'outils autour de ce langage. Dans ce chapitre nous étudierons les différents types de langage pour les spécifications systèmes. Nous présenterons plus particulièrement trois langages de spécification pour les systèmes distribués :

- ESTELLE
- LOTOS
- SDL

3.1 INTRODUCTION

Les langages sont habituellement classés par modèle d'exécution. Les langages de spécifications systèmes reposent principalement sur quatre principes [102] de base qui sont *la concurrence*, *la hiérarchie*, *la communication* et la *synchronisation*.

3.1.1 Concurrence

La concurrence est caractérisée par le grain du parallélisme. Celle-ci peut être au niveau de l'instruction (opérations parallèles) ou au niveau du processus. Au niveau du processus, le parallélisme spécifie le séquencement du flôt d'opérations de la spécification. Les machines d'états finis concurrentes sont un exemple de concurrence de flôt de contrôle. Le parallélisme d'instructions est orienté flôt de données. L'ordre d'exécution des opérations est déterminé par les dépendances de données.

3.1.2 Hiérarchie

La hiérarchie permet de décomposer un système en sous-systèmes afin de gérer sa complexité. La hiérarchie peut être structurelle ou comportementale.

- la hiérarchie comportementale
  La hiérarchie comportementale permet de décomposer un comportement en sous-comportements qui peuvent être séquentiels ou concurrents. Les procédures et les états hiérarchiques sont des exemples de hiérarchie comportementale. Le séquencement des différents comportements est permis par la détection de terminaison et la possibilité d'activation et de désactivation d'un comportement par un autre. Le traitement des exceptions consiste à réagir à un évènement extérieur en suspendant ou en terminant les actions en cours et en transférant le contrôle à une autre partie du comportement. Le traitement des exceptions peut être considéré comme une forme de hiérarchie comportementale dans la mesure où il consiste à désactiver le comportement courant pour entrer dans un état de traitement de l'exception.
- la hiérarchie structurelle
  La hiérarchie structurelle permet de décomposer un système en sous-systèmes interconnectés interagissant à travers des frontières bien définies. La communication peut être spécifiée à l'aide de signaux ou avec un niveau d'abstraction plus haut en utilisant des canaux encapsulant des protocoles de communication complexes.

3.1.3 Communication

La communication permet l'échange de données entre des modules concurrents. La communication inter-processus repose sur deux modèles distincts : *le passage de messages* et *la mémoire partagée* [2].

- le passage de messages
  Dans le modèle de communication par passage de messages, les processus échangent des données par l'intermédiaire de messages convoyés par des canaux de communication.

– la mémoire partagée
  Dans le modèle de communication par mémoire partagée, le transfert de données se fait par l'intermédiaire d'un médium partagé (mémoire) dans lequel les messages peuvent être écrits et lus par les processus. La communication par mémoire partagée s'appuie sur la notion de variable partagée par plusieurs processus. Des mécanismes d'exclusion mutuelle ou de gestion de section critique (*sémaphores*) sont alors mis en jeu pour assurer l'accès atomique et la cohérence des données. Le rendez-vous permet à une tâche de solliciter un rendez-vous avec une autre tâche qui va l'accepter. Cette opération est synchrone lors de la réalisation du rendez-vous mais elle possède un caractère asynchrone par le fait qu'on ne maitrise pas le moment de sa réalisation.

Les appels de procédures à distance sont un modèle hybride qui permet la modélisation des modèles de communication par passage de messages et par mémoire partagée. Dans ce modèle, la communication est effectuée par l'intermédiaire de procédures de communication. Celles ci sont invoquées par les processus communicants et correspondent à des opérations qui seront effectuées par une unité de communication distante. Cette encapsulation permet de séparer les parties comportement et communication d'une spécification.

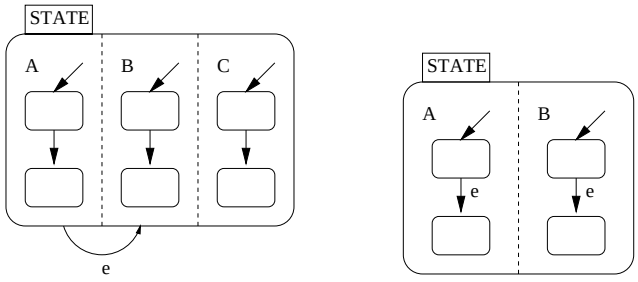
3.1.4 Synchronisation

La synchronisation permet la coordonnination des différents comportements et de la communication. Synchronisation et communication sont deux concepts étroitement liés. On peut distinguer deux types de synchronisation suivant qu'elle repose sur une communication par variable partagée ou par passage de message [1]. La synchronisation par initialisation et par événement commun sont des cas particuliers de la synchronisation par passage de messages, la synchronisation par détection d'état est un cas particulier de la synchronisation par variable partagée. Les différentes méthodes de synchronisation sont illustrées en figure 3.1 et 3.2.

– synchronisation par initialisation
  La synchronisation par initialisation permet d'initialiser tous les sous-états d'un état lorsque celui-ci devient actif. Sur la figure 3.1.1 l'évènement e synchronise les trois tables d'états A, B, C dans leur état d'initialisation.

– synchronisation par événement commun
  Les différents comportements sont en attente sur un même événement et ne continueront leur exécution que lorsque cet événement aura eu lieu. L'instruction *wait* permet la synchronisation par événement commun. Sur la figure 3.1.2 l'événement e synchronise les tables d'états A et B dans leur états *A2* et *B2* respectivement.

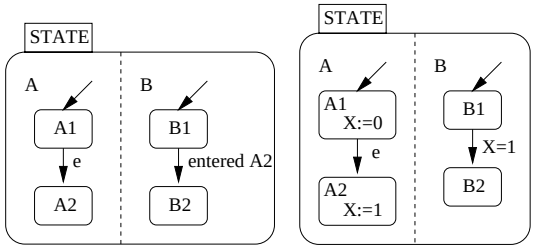
– synchronisation par passage de messages
  La synchronisation par message dépend du protocole de communication implémenté entre les comportements. Dans le cas d'un protocole synchrone (rendez-vous) un des processus concurrents sera bloqué jusqu'à ce que l'autre soit prêt à émettre ou à recevoir les données. Aucune synchronisation n'est réalisée lorsque les différents comportements sont découplés par une file de messages ou une mémoire partagée. Dans ce cas, la synchronisation entre les



3.1.1: Synchronisation par initialisation

3.1.2: Synchronisation par évènement commun

FIG. 3.1: Méthode de synchronisation par passage de message



3.2.1: Synchronisation par détection d'état

3.2.2: Synchronisation par variable commune

FIG. 3.2: Méthode de synchronisation par variable partagée

comportements peut être réalisée par le passage de messages ou par l'intermédiaire de variables partagées.

– synchronisation par détection d'état
La transition d'un état vers un autre d'un comportement déclenche une transition dans un autre comportement. Sur la figure 3.2.1 la transition de l'état *A1* vers *A2* provoque la transition vers létat *B2* de la table d'état *B*

– synchronisation par variable partagée
Une transition d'un état vers un autre d'un comportement est déclenchée par la valeur d'une variable partagée qui est mise à jour par un autre comportement. L'instruction *wait until* permet la synchronisation par variable partagée. Sur La figure 3.2.2, l'affectation de la variable *X* dans l'état *A2* synchronise la table d'état *B* dans l'état *B2*.

3.1.5 Les langages de spécification

Il existe différents types de langages issus de différentes communautés qui répondent à des critères particuliers selon les domaines d'application :

- Les langages de programmation.
- Les langages de description de matériel.
- Les langages pour les systèmes temps réels.
- Les langages de spécification de protocoles.
- Les langages fonctionnels.

Il n'y a pas de langage de spécification universel qui couvre toutes les applications. Un langage de spécification est sélectionné en fonction de l'application. Des langages tel que SDL, StateChart, ESTELLE sont bien adaptés à la description sous forme d'automate d'états finis (flôt de contrôle), d'autres tel que SIGNAL ou LUSTRE sont orientés flôt de données alors que des langages tel que C se prètent bien aux descriptions algorithmiques.

3.2 LANGAGE DE SPÉCIFICATION POUR LES SYSTÈMES RÉACTIF TEMPS RÉEL

Les systèmes réactifs-temps réel regroupent des systèmes aussi variés que :

- les systèmes embarqués.
- Les systèmes de contrôle/commande.
- Les protocoles de télécommunication.

Le terme *réactif* qualifie le comportement de ces systèmes alors que le terme *temps réel* est relatif aux contraintes qui leur sont imposés. Le terme réactif designe des systèmes qui réagissent à des événements d'entrée en générant des sorties. Le terme temps réel impose au système un temps maximum de réaction entre la variation des entrées et la réaction correspondante. L'importance prise par ce type de systèmes a entrainé le développement de langages tels que les langages synchrones.

3.2.1 L'approche synchrone

l'approche synchrone [10] est basée sur un petit nombre de concepts et repose sur un fondement mathématique. Le terme synchrone fait référence à la définition d'instants d'activation. Ces instants définissent les moments d'activation du système. Les réactions du système sont provoquées par ces signaux provenant de l'environnement qui y réagit de façon synchrone. En dehors de tout stimuli le système est inactif. La première hypothèse de l'approche synchrone est de considérer des système idéaux qui réagissent instantanément en produisant leurs sorties de façon synchrone aux entrées. Cette hypothèse est effectivement valable si toute opération a en réalité un temps d'exécution borné. Les aspects temporels prennent en compte l'ordre des événements, la simultanéïté mais pas le délai entre événements dans la mesure où le système a un temps de réaction nul. Celui-ci est en fait considéré comme une contrainte d'implémentation que le système devra satisfaire. Les principales caractéristiques des langages synchrones sont :

- la réactivité
  Le modèle réactif considère des systèmes communicants qui interagissent continuellement avec leur environnement. Un système réactif est une fonction qui produit une séquence de signaux de sortie à partir d'une séquence d'événements d'entrée. La vie d'un système synchrone est divisée en instants qui sont les moments où le système est activé. Chaque variation des entrées provoque une réaction du système qui met à jour ses sorties. Il existe de nombreuses primitives temporelles dans les langages synchrones permettant de manipuler les exceptions, les delais, les interruptions d'exécution.
- l'atomicité des réactions
  le modèle synchrone suppose que les réactions sont instantanées, c'est à dire que les variations des entrées et des sorties sont synchrones. Une autre manière d'exprimer l'hypothèse synchrone consiste à dire que toutes les actions internes effectuées par le système s'exécutent en un temps nul et de façon atomique. Une réaction du système ne peut pas interférer avec d'autres réactions : le système ne peut pas réagir à une nouvelle solicitation s'il est déja en train de réagir. Les réations du système ne peuvent pas être superposées. Cette hyptothèse permet de faire abstraction du temps physique.
- communication par diffusion instantanée
  Dans les langages synchrones, le modèle de communication repose sur la diffusion instantanée ou *broadcast*. Dans ce modèle, un message est diffusé à tout le système en même temps et sa récèption est synchrone à son émission. A un instant donné seule certaines parties du système vont être réceptives à ces messages.
- le parallélisme
  La notion de parallélisme est différente selon les langages. Dans des langages orientés flôt de données tels que LUSTRE ou SIGNAL, le parallélisme est à grain fin et il est exprimé de manière implicite. Dans le cas d'ESTEREL, le parallélisme est explicite et s'exprime au niveau des tâches.

3.2.2 Différents langages synchrones

Il y a principalement deux forme équivalentes de modélisation synchrone :

- le formalisme à base de machines d'états finis (langage impératif)
  Ce formalisme est bien adapté à la description de systèmes de contrôle. Il décrit un système en terme d'états, de transitions et d'événements. ESTEREL [20] et StateChart [68] en sont des exemples. StateChart utilise un modèle d'automates d'états finis étendus avec la hiérarchie, le parallélisme et la communication.
- le formalisme des horloges multiples (langage équationnel)
  Ce formalisme est bien adapté aux problèmes dominés par le flôt de données. Cette appellation provient du modèle mathématique de flôt de données à plusieurs horloges sur lequel est basé la sémantique de ces langages. Ce flôt de données est composé d'un ensemble de valeurs ainsi que d'une horloge qui donne la séquence de dates associées à ces données. Chaque flot possède sa propre horloge. LUSTRE [64], SIGNAL [51], [65] font partie de cette catégorie. Dans ce formalisme, le comportement du système est décrit comme un ensemble d'opérations effectuées

sur un flôt de données. LUSTRE est bien adapté à la description d'automates programmables et SIGNAL aux applications purement flôt de données.

Les langages synchrones sont principalement utilisés en remplacement des langage classiques (C, assembleur) pour le développment d'applications embarqués et temps réel [66]. Plusieurs approches de génération de matériel sont rapportées dans la littérature. Dans [101], une approche de génération d'architectures parallèles régulières à partir de ALPHA est présentée. Dans [40], StateChart est transposé sur des automates d'états finis ou sur une architecture ASIP [25]. [14] présente une approche de réalisation matérielle de spécifications ESTEREL et [3] une approche de co-design. Peu d'approches de co-design à partir de ces langages sont rapportées dans la littérature. Dans [8], une approche de co-design utilisant SIGNAL pour décrire les parties contrôle et ALPHA pour les parties de calcul est présentée. [24] propose une approche de co-design basée sur StateChart et [5], [88] une approche de partitionnement logiciel/matériel basée sur le langage UNITY.

3.3 LANGAGES FORMELS

Les langages formels reposent sur des fondements mathématiques qui permettent d'effecuter des vérifications formelles sur les systèmes décrits. Ils sont divisés en deux grandes catégories qui sont les langages fonctionnels et les algèbres de processus.

3.3.1 Langages fonctionnels

Les langages fonctionnels sont basés sur la théorie des ensembles et la logique des prédicats. Il permettent de décrire les propriétés fonctionnelles d'un système en termes d'états et de prédicats. Z [114], [127] permet de décomposer une spécification en modules, appelés schémas, qui décrivent à la fois les aspect statiques et dynamiques du système. Ces schémas peuvent être utilisés pour décrire le système en terme d'états et comment ceux ci sont affectés lorsque le système effectue des opérations.

Le langage Z a été utilisé pour la spécification et la validation de l'unité flottante du Transputer T800 [4]. L'approche adoptée a été la suivante :

- description du format flottant IEEE en Z.
- décomposition de la spécification en unités élémentaires en Z.
- réécriture en OCCAM des unités élémentaires satisfaisant à la spécification décomposée.
- optimisation des performances des programmes OCCAM à l'aide d'outils de transformation semi-automatiques.
- traduction des programmes OCCAM en microcode.

3.3.2 Algèbres de processus

Les algèbres de processus permettent de décrire un système comme un ensemble de processus communicants. La plupart des travaux concernent les langages tel que CSP [72] ou CCS [99]. Ils ont menés à des langages tel que OCCAM ou UNITY [5].

3.4 LANGAGES ALGORITHMIQUES

Les langages algorithmiques sont aussi qualifiés de langages asynchrones par opposition aux langages synchrones. Les langages algorithmiques regroupent tous les langages utilisés en informatique pour la programmation des ordinateurs.

3.4.1 langages mono-flôts

La plupart des langages de programmation ont été utilisés pour la description de matériel. Bien que ces langages offrent de nombreuses possibilités pour les spécifications au niveau système, il manque généralement des concepts tels que le parallélisme, la communication ou le temps. Certaines approches ont étendues des langages tel que C pour y ajouter des concepts matériels et de communication (HardwareC [61], [100] ou C^x [47], C ou C++ [27], [145], [149]). Elles consistent en la définition de fonctions ou de classes (dans les langages objets) permettant d'exprimer le parallélisme (*process*) et la synchronisation (*wait*) et la communication (*send*, *receive*). L'utilisation de fonction prédéfinies pour la communication (*send*, *receive*) permet d'utiliser une bibliothèque lors de la synthèse.

3.4.2 Langages multi-flôts

Les langages de programmation parallèle sont plus adaptés aux spécifications systèmes dans la mesure où ils intègrent la notion de concurrence. Ces langages disposent de constructeur permettant d'exprimer le parallélisme au niveau du processus ou de la tâche. En ADA, JAVA, l'entité qui permet d'exprimer le parallélisme est la tâche. En OCCAM, le parallélisme est exprimé par le constructeur *par* du langage. Ce constructeur permet de lancer plusieurs instances du même processus avec des paramètres différents. Contrairement à la majorité des langages où le parallélisme s'exprime au niveau du processus, OCCAM permet d'exprimer le parallélisme avec une granularité plus fine allant du processus jusqu'au niveau de l'instruction. En OCCAM il n'existe pas de constructeur permettant de suspendre un processus ou de traiter des exceptions. Une approche utilisant le langage OCCAM est décrite dans [6].

De nombreuses extensions parallèles du langage C ou C++ ont été développées. Ces extentions peuvent être classées en deux catégories :

- les extentions *task parallel*
  Elles utilisent la notion de *coroutine* ou de *thread* pour exprimer le parallélisme au niveau du processus. Le modèle de parallélisme est *multiple-instruction-multiple-data.*
- les extentions *data parallel*
  Le parallélisme est introduit au niveau des données. Des constructeurs tel que *forall* permettent d'effectuer en parallèle des opérations sur des données indépendantes. (parallélisme *single-instruction-multiple-data*).

Parmis les langages parallèles, certains comme OCCAM utilisent le passage de messages. Les messages entre les processus concurrents sont acheminés par des canaux de communication point à point. ADA offre deux schémas de communication : la mémoire partagée, le rendez-vous. JAVA

offre une communication inter-threads basée sur des objets partagés. Ce modèle est très proche de celui utilisé dans COSMOS.
Dans les langages *task parallel* la communication et la synchronisation est effectuée à l'aide de fonction provenant d'une bibliothèque. Cette approche rend les langages *task parallel*, et les extentions du langage C telle que HardwareC ou C^x très proches. Dans les langages *data parallel* la communication est implicite. Certaines approches orientées objet offrent une communication par objets partagés.

3.4.3 Langage de description matériel

Les deux langages de description matériels les plus utilisés sont VHDL et VERILOG. Les langages de description matériels supportent les concepts de parallélisme et de communication interprocessus et se rapprochent de la catégorie des langages multi-flôts. Ces langages ne supportent pas directement des concepts système tel que les machines d'états finis ou la communication de haut niveau qui requièrent des descriptions verbeuses.

Plusieurs approches utilisent un langage de description matériel pour décrire les parties matérielles d'un système en conjonction avec un langage logiciel (C) pour décrire les parties logicielles [87], [100]. Une approche utilisant VHDL comme langage de spécification système est rapportée dans [46]. De nombreuses extensions orientées objet et système de VHDL ont été proposées [108], [125], [129].

3.4.4 Langages systèmes

Le langage SpecCharts [107], [137], [138] a été développé pour la spécification et la synthèse au niveau système. SpecChart combine les différents concepts systèmes dans un même langage. SpecChart repose essentiellement sur les machines d'états finis hiérarchiques communicantes. Celles-ci combinent le parallélisme, la hiérarchie des états dans les machines d'états finis. La communication en SpecChart repose sur les signaux et supporte la notion de protocole de communication qui est une procédure encapsulant un protocole de communication particulier. Ce niveau d'abstraction de la communication est néanmoins trop faible pour lui permettre de modéliser des schémas de communication complexe tel que le passage de messages.

3.5 LANGAGES DE SPÉCIFICATION POUR LES TÉLÉCOMMUNICATIONS

Les *techniques de description formelle*, ou *FDT* dérivent des travaux effectués sur les *langages de spécifications formelles* et les méthodes de développement rigoureuses de logiciel. Les langages de spécifications formelles reposent sur des concepts mathématiques qui perme-ttent de vérifier les propriétés des systèmes décrits. Différentes approches ont été développées. Parmi elles, les automates d'états finis et les types de données abstraits sont devenus très populaires. Les techniques de description formelle ont été appliquées en particulier au domaine des télécommunications ou les protocoles sont complexes et évoluent rapidement. Les techniques de description formelle permettent :

– d'offrir un ensemble de concepts bien définis.

- des spécifications claires, concises et non ambigües.
- de vérifier si les spécifications sont complètes et correctes.
- de déterminer la conformance entre la spécification et l'implémentation.

Une spécification formelle se concentre sur les propriétés du système décrit plutôt que sur les détails d'implémentation. Cette spécification formelle sert de base pour une réalisation, elle doit donc faire abstraction des détails d'implémentation afin :

- de retarder les décisions concernant la réalisation et
- de permettre de n'exclure aucune implémentation valide.

3.5.1 ESTELLE

ESTELLE [26] est un langage de spécification pour les télécommunications et les systèmes distribués, en particulier les protocoles de communication utilisant les modèles en couche de l'OSI. ESTELLE est standardisé par l'ISO [75]. Une spécification ESTELLE décrit la structure et le comportement d'un système. La sémantique d'ESTELLE est définie de façon formelle, ce qui permet de disposer d'outils de validation. ESTELLE repose sur les machines d'états finis étendues communicantes.

La structure hiérarchique d'un système est décrite par un ensemble de modules interco-nnectés. Le comportement d'un module est décrit par un automate d'états finis. Les modules ESTELLE agissent en exécutant des transitions d'un état à l'autre. Exécuter une transition consiste à consommer un message, effectuer des actions et changer d'état. Les modules ESTELLE peuvent être non déterministes, ce qui signifie que plusieurs transitions peuvent être tirables dans un état donné pour une entrée donnée. Une seule d'entre elles sera alors exécutée. Une implémentation basée sur une spécification ESTELLE devra résoudre ce non déterminisme d'une façon quelconque.

Différents modules sont interconnectés par des canaux appelés liens. Les canaux sont connectés aux modules par des points d'intéractions (IP) qui représentent l'interface du module. Seules les connections point à point sont autorisées mais celles-ci peuvent être configurées à l'exécution. Les messages (aussi appelé interactions) reçus par les points d'intéractions sont stockés dans des files d'attente qui peuvent être privées ou communes aux points d'intéractions (figure 3.3).

Un module peut être structuré en sous modules (appelés modules fils) permettant de créer une hiérarchie. Les transitions des sous modules peuvent s'exécuter soit en parallèle soit en série. Les transitions d'un module père sont exécutées en priorité par rapport à celle de ses modules fils. Cela exclut le parallélisme entre des modules situés à des niveaux différents dans une hiérachie de modules.

ESTELLE repose sur le modèle des machines d'états finis étendues. Chaque état est composé d'un ensemble de transitions gardées par un ensemble de conditions :

- l'état courant de l'automate.
- le message en tête d'une des files d'attente.
- la valeur de variables.
- le status de timers.

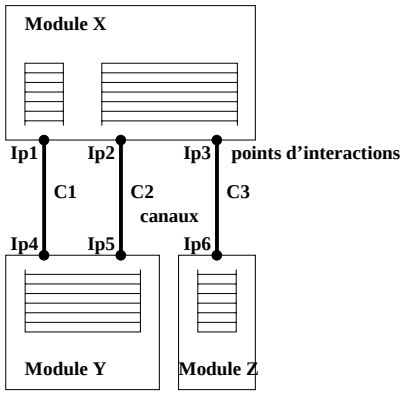


FIG. 3.3: Modules et canaux ESTELLE

– la priorité de la transition.

Les transitions qui ne requièrent l'arrivée d'aucun message sont appelées transitions spontanées. A chaque transition est associée une priorité, implicitement ou explicitement. Pour qu'une transition puisse être exécutée, aucune transition de priorité plus élevée ne doit être tirable. Lorsque les conditions de garde d'une transition sont valides alors celle-ci est exécutée. Si plusieurs transitions sont tirables à un instant donné, alors une d'entre elles est choisie de façon non déterministe. La figure 3.4 représente un automate avec un choix non déterministe dans l'état *disconnected*. A la réception d'un signal *ConReq* dans la file d'attente *ip1*, une des deux transitions possibles est exécutée. Les actions d'une transition s'exécutent en un temps nul et de façon atomique.

```
STATE disconnected, called, calling, connected;

INITIALIZE TO disconnected
  BEGIN
  END;

TRANS

FROM disconnected TO SAME
  WHEN ip1.ConReq
    BEGIN  { refus de la connection }
      OUPUT ip1.DisInd; { emission d'un message }
      { actions de la transition }
    END;
FROM disconnected TO calling
    WHEN ip1.ConReq
      BEGIN { acceptation de la connection }
        OUPUT ip2.IConReqInd;  { emission d'un message }
        { actions da la transition }
      END;
FROM calling TO connected
  WHEN ip2.IConReqInd
    BEGIN
      { actions de la transition }
    END;
FROM connected TO disconnected
  WHEN ip1.DatReq(UserData)
    BEGIN
      { actions de la transition }
    END;
```

FIG. 3.4: Spécification d'un automate ESTELLE

3.5.2 LOTOS

LOTOS a été développé à partir des théories bien établies de CCS (*Calculus of Communicating Systems*, [99]) et CSP (*Communicating Sequential Process*, [72]). LOTOS [18], [93], [132] repose

principalement sur trois concepts qui sont les processus, la synchronisation et le comportement pour modéliser les systèmes. LOTOS est standardisé par l'ISO [76]

Le comportement d'un processus LOTOS est décrit comme une séquence d'événements observables de l'environnement. Celle-ci définit la séquence ordonnée d'événements de synchronisation et d'actions effectuées par le processus. Cette séquence peut être représentée sous forme d'un arbre. Les noeuds représentent l'état du système qui sont des points de synchronisation. Les arcs représentent les transitions entre états et sont etiquetés avec les actions à effectuer.

Les différents processus d'un système s'exécutent en parallèle et se synchronisent à travers des canaux de communication appelés portes. La figure 3.5 représente un système LOTOS. Le processus S interagit avec l'environnement à travers les portes a, b, c, d. Ces portes représentent l'interface du processus avec son environnement. Le processus S est la composition des processus P et Q. Le processus P interagit avec l'environnement à travers les portes a, b d et Q à travers les portes b et c. Les processus P et Q interagissent à travers la porte b en synchronisation partielle.

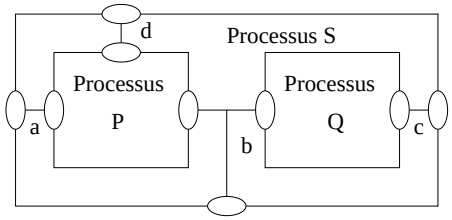


FIG. 3.5: Processus LOTOS

Différents processus peuvent être composés en parallèle pour décrire le comportement d'un système. Trois opérateurs de composition sont définis :

- entrelacement : les processus s'exécutent en parallèle sans synchronisation. Dans ce modèle, les processus évoluent indépendamment sans interagir.
- synchronisation totale : les processus s'exécutent en parallèle et se synchronisent sur toutes les portes.
- synchronisation partielle : les processus se synchronisent sur un ensemble de portes.

Il n'y pas de réel parallélisme entre différents processus LOTOS. Les opérateurs de composition génèrent la séquence d'événements séquentiels équivalente à l'exécution parallèle des processus composés. Celle-ci est donc le produit cartésien des comportements composés, c'est à dire une alternative entre tous les entrelacements possibles des différents comportement composés. L'exécution parallèle de deux événements est définie comme une situation d'alternative où l'un des événements arrive en premier et l'autre ensuite ou vice versa. La figure 3.6 montre le résultat de l'entrelacement de deux processus. a, b, c représentent des portes sur lesquelles les processus attendent des événements. Dans le cas de l'entrelacement il n'y a aucune synchronisation entre les deux comportements, il en résulte que le comportement équivalent est le produit des deux arbres de comportement. Les événements sont atomiques en LOTOS et s'exécutent en un temps nul.

La figure 3.7 représente une composition de processus avec synchronisation sur la porte a.

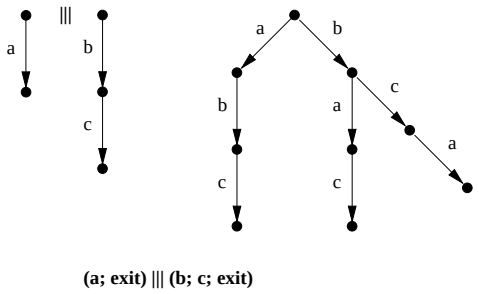


FIG. 3.6: Entrelacement de processus LOTOS

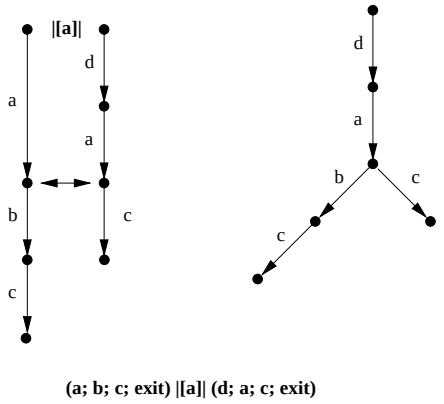


FIG. 3.7: Synchronisation de processus LOTOS

LOTOS implémente un schéma de communication synchrone qui repose sur le modèle du rendez-vous multiple. Une communication ne peut avoir lieu que si tous les processus qui y participent sont prêts.

Plusieurs types de rendez-vous sont possibles :

- synchronisation : les processus se synchronisent à travers une porte avant de continuer leur exécution. Dans ce rendez-vous, il n'y a pas d'échange de données.
- correspondance de valeur (*value matching*) : les processus se synchronisent en émettant une valeur commune à travers une porte.
- passage de valeur (*value passing*) : un processus émet une donnée à un ou plusieurs autres processus qui sont en attente sur la porte.
- négociation et génération de valeur (*value negociation and generation*) : les processus se synchronisent en déterminant une valeur compatible avec toutes les valeurs offertes.

La table 3.1 résume les différents modèles de synchronisation LOTOS. Lors d'une synchronisation les différents processus peuvent :

- offrir une donnée x à travers une porte g ($g\,!\,x$).
- recevoir une donnée x à travers une porte g ($g\,?\,x$).

communication	processus 1	processus 2	condition de synchronisation	résultat
synchronisation	|[g]|	|[g]|	*aucune*	
value matching	g ! E_1	g ! E_2	$Value(E_1) = Value(E_2)$	
value passing	g ! E_1	g ? x : T_2	$Type(E_1) = T_2$	$x = E_1$
value generation	g ? x : T_1	g ? y : T_2	$T_1 = T_2$	$x = y \in T_1$

TAB. 3.1: Différents rendez-vous LOTOS

Le non déterminisme en LOTOS apparait dans le comportement et dans la communication. Lors d'une synchronisation avec génération de valeur, celle-ci est déterminée aléatoirement parmi l'ensemble des valeurs possibles.

3.6 SPÉCIFICATIONS SYSTÈME AVEC SDL

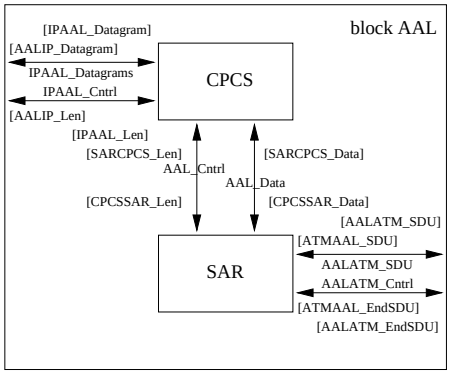
Le langage SDL (Specification and Description Language) [9], [48] est dédié à la modélisation et à la simulation des systèmes temps réel, distribués et de télécommunication et est standardisé par l'ITU [151]. Un système décrit en SDL est composé d'un ensemble de processus concurrents communicant à travers des signaux. SDL supporte différents concepts permettant la description de systèmes tels que la structure, le comportement et la communication. Il existe deux formats SDL, un graphique et l'autre textuel.

3.6.1 Structure

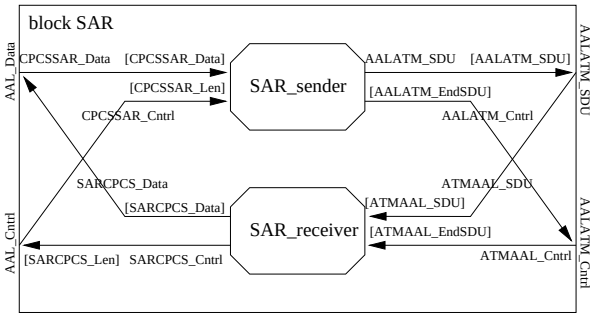
L'entité la plus haute de la hiérarchie est appelée système (*system*). La structure statique d'un système est décrite en utilisant une hiérarchie de blocs (*block*). Un bloc peut contenir d'autres blocs, résultant en une structure hiérarchique ou bien un ensemble de processus (*process*) dans le cas d'un bloc feuille.

- Une instance de système contient un ensemble d'instances de blocs.
- Une instance de bloc peut contenir d'autres instances de blocs ou un ensemble d'instances de processus.
- Une instance de processus peut contenir un automate d'états finis ou un ensemble de *services* exécutés séquentiellement.
- Un service est un automate d'états finis.
- Une procédure est un automate d'états finis.

Les interfaces de communication des différents processus sont interconnectées entre elles à l'aide de canaux de communication (figure 3.8.2). Les différents processus d'un même bloc sont connectés entre eux et jusqu'à la frontière du bloc par des *routes*. Les blocs sont connectés entre eux par des *canaux*. Deux instances dans différentes parties d'un système sont connectées entre elles à travers les interfaces des blocs qui les englobent : deux processus appartenant à des blocs différents sont reliés



3.8.1: Système SDL



3.8.2: Bloc SDL

FIG. 3.8: Description SDL graphique

à leur blocs respectifs par des routes et les blocs sont reliés entre eux par des canaux. Les canaux et les routes sont des vecteurs de signaux utilisés par les processus pour communiquer. Les signaux échangés par les processus suivent un chemin composé de routes et de canaux du processus émetteur au processus receveur. Sur le système représenté en figure 3.8.2, le processus *SAR_ sender* communique avec les frontières du bloc *SAR* à travers les routes *CPCSSAR_ Data*, *CPCSSAR_ Cntrl*, *AALATM_ SDU* et *AALATM_ EndSDU*. Ces routes sont connectées respectivement aux canaux *AAL_ DATA*, *AAL_ Cntrl*, *AALATM_ SDU*, *AALATM_ Cntrl*. Sur la figure 3.8.1, les deux blocs *SAR* et *CPCS* communiquent à travers les canaux *AAL_ Data* et *AAL_ Cntrl* et aux frontières à travers *AALIP_ Datagrams*, *AALIP_ Cntrl*, *AALATM_ SDU*, *AALATM_ Cntrl*.

Création et destruction de processus

Les processus SDL sont soit créés statiquement à l'initialisation du système soit dynamiquement lors de l'exécution. Pour chaque processus il est possible de définir des paramètres transmis au processus lors de l'instanciation ainsi que le nombre minimal et maximal d'instances qui peuvent être créés. Lorsque le nombre minimal d'instances possibles d'un processus est supérieur à zéro, celles ci sont crées à l'initialisation du système.

Un processus qui vient d'être créé se retrouve dans l'état initial *start*. Il peut alors s'exécuter. Un processus peut se terminer avec l'instruction *stop* et devient inactif. Durant l'exécution d'un système SDL, il est possible de créer dynamiquement de nouvelles instances de processus en utilisant l'instruction *create*. Chaque instance de processus possède une adresse unique, appelée *Pid* (*process identifier*), qui l'identifie parmi les autres processus. Chaque processus peut accéder à son *Pid* à travers la variable *self*. Cette adresse peut être utilisée pour communiquer avec une instance particulière d'un processus. La figure 3.9 montre un exemple de création statique d'instances multiples du processus. Trois instances du processus *P1* sont créées à l'initialisation du système SDL. Chaque instance de *P1* aura un *Pid*) différent, ce qui permettra à *P2* de les identifier.

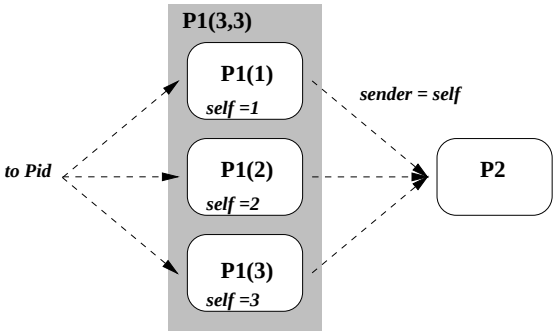


FIG. 3.9: Instances multiples de processus SDL

3.6.2 Communication

En SDL, la communication est basée sur le modèle de passage asynchrone de messages entre les processus. La communication est effectuée par l'intermédiaire de signaux. Les signaux SDL sont typés et peuvent transporter un ensemble de paramètres.

Les signaux sont transférés entre les processus par les routes et les canaux. Les routes sont directements connectées aux processus et finissent à la frontière des blocs où elles sont connectées aux canaux. les canaux connectent les blocs entre eux. A la frontière d'un bloc, un canal peut être connecté à plusieurs autres canaux ou routes. Si les processus sont contenus dans le même bloc alors les routes suffisent à acheminer les signaux. Si les processus sont situés dans des blocs différents, les signaux doivent emprunter des canaux. Les principales différences entre les routes et les canaux sont :

- une communication à travers une route s'effectue en un temp nul alors qu'une communication à travers un canal subit un retard non deterministe. Aucune hypothèse ne peut être faite concernant le délais et l'ordre d'arrivée de signaux utilisant deux chemins différents ne peut pas être prédit.
- un canal peut effectuer une opération de routage sur un signal. Il peut router un signal à travers différents canaux (routes) connectés à la frontière du bloc en fonction du processus destinataire.
- les routes acheminent les signaux aux processus destinataires. Une route ne peut pas être connectée à plus d'un canal.

```
BLOCK AAL_Layer;
SIGNAL  CPCSSAR_Data(DATA), CPCSSAR_Len(INTEGER),
        SARCPCS_Data(DATA), SARCPCS_Len(INTEGER);
  SUBSTRUCTURE;
    CHANNEL AAL_Data
      FROM CPCS TO SAR WITH CPCSSAR_Data;
      FROM SAR TO CPCS WITH SARCPCS_Data;
    ENDCHANNEL AAL_Data;
    CHANNEL AAL_Cntrl
      FROM CPCS TO SAR WITH CPCSSAR_Len;
      FROM SAR TO CPCS WITH SARCPCS_Len;
ENDCHANNEL AAL_Cntrl;
CHANNEL IPAAL_Datagrams
  FROM CPCS TO ENV WITH AALIP_Datagram;
  FROM ENV TO CPCS WITH IPAAL_Datagram;
ENDCHANNEL IPAAL_Datagrams;
CHANNEL IPAAL_Cntrl
  FROM CPCS TO ENV WITH AALIP_Len;
  FROM ENV TO CPCS WITH IPAAL_Len;
ENDCHANNEL IPAAL_Cntrl;
CHANNEL AALATM_SDU
  FROM SAR TO ENV WITH AALATM_SDU;
```

```
      FROM ENV TO SAR WITH ATMAAL_SDU;
    ENDCHANNEL AALATM_SDU;
    CHANNEL AALATM_Cntrl
      FROM SAR TO ENV WITH AALATM_EndSDU;
      FROM ENV TO SAR WITH ATMAAL_EndSDU;
    ENDCHANNEL AALATM_Cntrl;
    CONNECT AALATM_Cntrl AND AALATM_Cntrl;
    CONNECT IPAAL_Cntrl AND IPAAL_Cntrl;
    CONNECT AALATM_SDUs AND AALATM_SDU;
    CONNECT IPAAL_Datagrams AND IPAAL_Datagrams;

    BLOCK CPCS REFERENCED;
    BLOCK SAR REFERENCED;
  ENDSUBSTRUCTURE;
ENDBLOCK AAL_Layer;

BLOCK SAR;
  SIGNALROUTE SARCPCS_Data
    FROM SAR_Receiver TO ENV WITH SARCPCS_Data;
  SIGNALROUTE SARCPCS_Cntrl
    FROM SAR_Receiver TO ENV WITH SARCPCS_Len;
  SIGNALROUTE CPCSSAR_Data
    FROM ENV TO SAR_Sender WITH CPCSSAR_Data;
  SIGNALROUTE CPCSSAR_Cntrl
    FROM ENV TO SAR_Sender WITH CPCSSAR_Len;
  SIGNALROUTE ATMAAL_SDU
    FROM ENV TO SAR_Receiver WITH ATMAAL_SDU;
  SIGNALROUTE ATMAAL_Cntrl
    FROM ENV TO SAR_Receiver WITH ATMAAL_EndSDU;
  SIGNALROUTE AALATM_SDU
    FROM SAR_Sender TO ENV WITH AALATM_SDU;
  SIGNALROUTE AALATM_Cntrl
    FROM SAR_Sender TO ENV WITH AALATM_EndSDU;
  CONNECT AALATM_Cntrl AND ATMAAL_Cntrl,AALATM_Cntrl;
  CONNECT AALATM_SDU AND ATMAAL_SDU,AALATM_SDU;
  CONNECT AAL_Cntrl AND SARCPCS_Cntrl,CPCSSAR_Cntrl;
  CONNECT AAL_Data AND SARCPCS_Data,CPCSSAR_Data;

  PROCESS SAR_Sender REFERENCED;
  PROCESS SAR_Receiver REFERENCED;
ENDBLOCK SAR;
```

FIG. 3.10: Spécification textuelle SDL

Les canaux et les routes peuvent être soit mono soit bidirectionnels. Si plusieurs signaux sont transférés sur un même canal ou route, leur ordre est préservé. Lors de leur acheminement par les canaux et les routes, les signaux ne sont pas autorisés à se doubler. Malgré tout, aucun ordre d'arrivée dans une file d'attente ne peut être prédit pour des signaux ayant suivi des canaux et des routes différents.

La figure 3.8 représente la spécification de la structure et de la communication d'un système en SDL. La définition textuelle correspondante est donnée en figure 3.10.

3.6.3 Comportement

Le comportement d'un système est décrit comme un ensemble de processus concurrents communicants (figure 3.11). Un processus est décrit comme un automate d'états finis qui communique avec les autres processus de façon asynchrone à travers des signaux. Chaque processus possède en entrée une file d'attente infinie dans laquelle les signaux sont stockés à leur arrivée. Les signaux sont extraits de la file d'attente par le processus dans leur ordre d'arrivée. En d'autres termes, les signaux sont bufferisés dans un ordre *first-in-first-out*. Ce modèle de communication asynchronne faiblement couplé semble être un bon modèle pour la description de systèmes distribués.

États et transitions

Chaque processus est composé d'un ensemble d'états (*state*) et de transitions entre deux états (*nextstate*). Dans un état donné de l'automate, un processus est en attente sur un ensemble de signaux (*input*). L'arrivée d'un signal attendu dans la file d'attente valide une transition et le processus peut alors exécuter un ensemble d'opérations (*task*) telle que la manipulation de variables, des appels de procédure (*call*) et l'émission de signaux (*output*). Le signal reçu détermine la transition

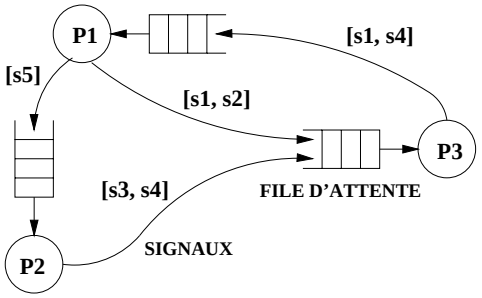


FIG. 3.11: Modèle de spécification SDL

à exécuter. Lorsqu'un signal a initié une transition il est retiré de la file d'attente. Si aucune transition ne correspond au signal en tête de la file alors celui-ci est simplement consommé sans changement d'état de l'automate (transition implicite). Les transitions implicites ont pour but d'éviter qu'un processus ne reste infiniment bloqué dans le même état. Un système qui exécute des transitions implicites peut être considéré comme sous-spécifié puisque certains cas d'exécution n'ont pas éte prévus. Une transition mentionne le nom du signal et un ensemble de variables dans lequel seront placés les paramètres du signal. La variable prédéfinie *sender* recevra implicitement l'adresse du processus émetteur du dernier signal extrait de la file d'attente. Une transition peut être validée quel que soit le signal extrait de la file d'attente en utilisant *input **. Dans ce cas l'astérisque remplace tous les signaux succeptibles d'être reçus par le processus.

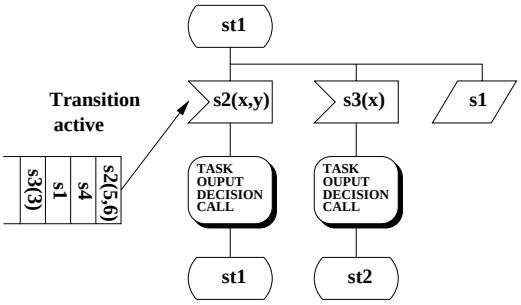
Un signal peut être retenu dans la file d'attente et son traitement peut être reporté dans un autre état à l'aide de l'instruction *save*. Cela permet d'affecter des priorités dans le traitement des signaux.

La *condition d'autorisation* permet d'ajouter une condition booléenne supplémentaire au déclenchement d'une transition. Une transition validée ne sera exécutée que si la condition d'autorisation est vraie. Sinon le signal sera sauvegardé et l'automate restera dans le même état.

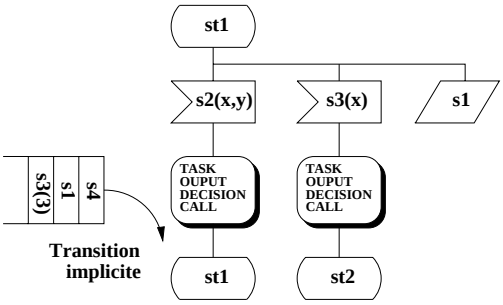
Les *signaux continus* permettent de définir des transitions gardées par des conditions booléenes. Lorsque la condition est vraie la transition est exécutée. Un signal continu ne peut déclencher une transition que lorsque la file d'attente est vide. Si plusieurs signaux continus sont associés à un état, l'ordre d'évaluation des priorités peut être fixé en associant une priorité à chacun d'eux.

La figure 3.12.1 représente un exemple de spécification graphique de processus SDL. Dans l'état *st1* le processus est en attente sur les signaux *s2* et *s3* et le signal *s1* est sauvé. Le signal *s2* a deux paramètres, le signal *s3* un et le signal *s1* aucun. Un exemple d'exécution est donné en figure 3.12. La figure 3.12.2 représente une transition implicite. Un signal extrait de la file d'attente dans un état donné (*st1*) et qui ne déclenche aucune transition est consommé sans changement d'état de l'automate. La figure 3.12.3 représente la sauvegarde du signal *s1*. Dans l'état *st2* (figure 3.12.4), la transition gardée par le signal *s4* est associée à une condition d'autorisation $z=7$, z étant une variable définie dans le processus. La transition ne pourra être exécutée que si cette condition est remplie. Cet état comporte aussi un signal continu gardé par l'équation booléene *b1=false*.

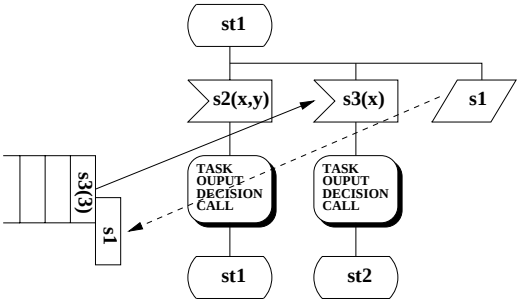
La figure 3.13.1 représente une spécification graphique d'un processus SDL et la figure 3.13.2 représente la description textuelle correspondante. L'état *start* est l'état d'initialisation. *input* représente



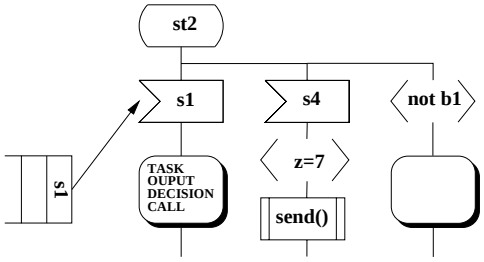
3.12.1: Automate d'états finis SDL



3.12.2: Transition implicite



3.12.3: Sauvegarde d'un signal



3.12.4: Signaux continus et condition d'autorisation

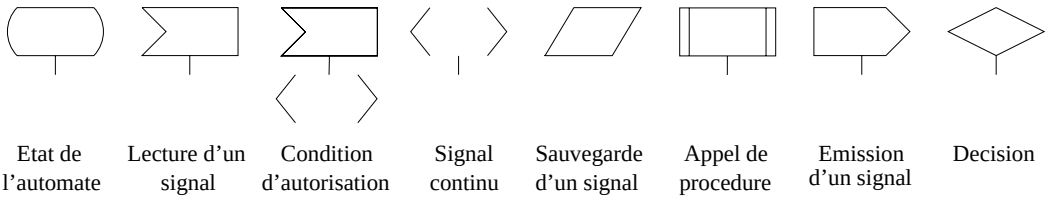
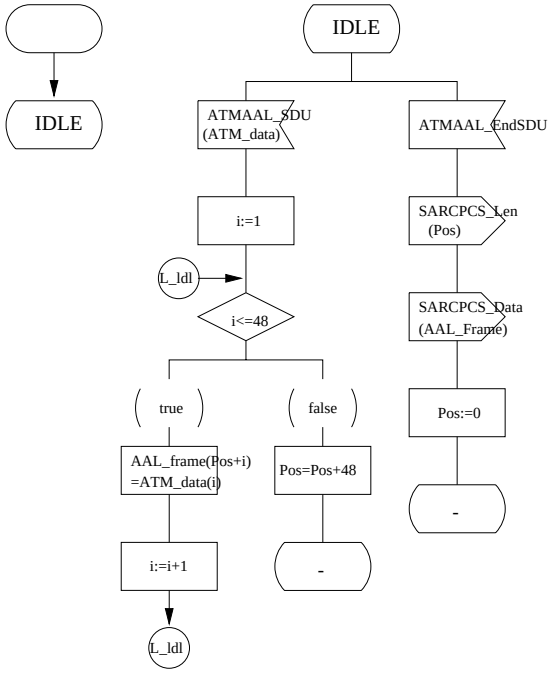


FIG. 3.12: Modèle d'exécution SDL



3.13.1: Description graphique de processus SDL

```
PROCESS SAR_Receiver;

    DCL Pos INTEGER :=0;
    DCL I INTEGER;
    DCL AAL_Frame DATA;
    DCL ATM_Data ATM_DATA;

  START;
    NEXTSTATE IDLE;

  STATE IDLE;
    INPUT ATMAAL_SDU(ATM_Data);
      TASK I:=1;
    L_Idl:
      DECISION I<= 48;
      ( true ):
        TASK AAL_Frame(Pos+I):=ATM_Data(I);
        TASK I:=I+1;
        JOIN L_Idl;
      ( false ):
        TASK Pos:=Pos+48;
         NEXTSTATE -;
      ENDDECISION;
    INPUT ATMAAL_EndSDU;
      OUTPUT SARCPCS_Len(POS);
      OUTPUT SARCPCS_Data(AAL_Frame);
      TASK Pos:=0;
      NEXTSTATE -;
  ENDSTATE;
ENDPROCESS;
```

3.13.2: Description textuelle de processus SDL

FIG. 3.13: Spécification de processus SDL

la condition de garde d'une transition. Cette transition sera exécutée lorsque le signal correspondant sera extrait de la file d'attente. *task* représente une opération à effectuer lorsque la transition est exécutée et *output* émet un signal avec ses paramètres éventuels.

Variables

En SDL les variables sont propres à un processus et ne peuvent pas être modifiées par d'autres processus. La synchronisation inter-processus est effectuée à l'aide de signaux uniquement. La table 3.2 résume les types de données disponibles en SDL.

Types de données abstraits

SDL autorise la définition de nouveau type de données et d'opérateurs agissant sur ces données. Les opérateurs peuvent être implémentés de différentes façon :

- de façon axiomatique par un ensemble d'équations.
- de façon algorithmique.
- dans un langage extérieur tel que C.

Type	Littéraux	Opérateurs
Integer	$-\infty, .. -1, 0, 1, ..$	$+, -, *, /, =, <, \leq, >, \geq$, float, fix
Real	$-\infty, ..37.2..$	$+, -, *, /, =, <, \leq, >, \geq$
Booleen	true, false	not, and, or xor
Time	as real	$+, -, <, \leq, >, \geq$
Duration	as real	+, -,>, *, /
Caractère	caractère entre ' '	$<, \leq, >, \geq$
chaine de caratère	caractère entre ' '	mkstring, length, first, last, extract, modify, substring
Pid	Null	aucun

TAB. 3.2: Type de données SDL

L'implémentation d'opérateurs en C permet de bénéficier de la puissance de ce langage, en particulier pour la manipulation de données. Certaines opérations sur des données qui ne peuvent pas être spécifiées en SDL peuvent l'être en utilisant les types de données abstraits et les opérateurs.

Procédures

SDL permet la définition de procédures dans un processus. Une procédure peut définir ses propres variables locales mais ne peut pas les exporter (voir section 3.6.4). Une procédure utilise la file d'attente du processus où elle est définie pour déclencher des transitions.

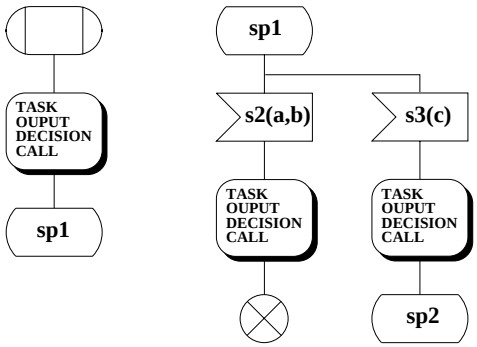


FIG. 3.14: Procédure en SDL

La figure 3.14 représente la procédure *send* appelée en figure 3.12.4 (l'état *sp2* n'est pas représenté).

Services

Le comportement d'un processus peut être spécifié comme une ensemble de *services* s'exécutant séquentiellement. Chaque service est defini séparemment comme un automate d'états finis mais un seul service s'exécute à un moment donné. Lorsqu'un service atteint un nouvel état, le service capable de consommer le signal en tête de la file d'attente devient actif et s'exécute. Les différentes transitions des services sont donc entrelacées et il n'y a aucun risque d'interférence entre les différents services. Les différents services d'un processus partagent la file d'attente, les variables et les expressions *self* et *sender* du processus dans lequel ils sont définis.

Exception

SDL permet de définir des exceptions. Il s'agit d'actions à effectuer quel que soit l'état courant (*state* ∗) et qui nécessitent un traitement particulier. Certains états dans lesquels une exception ne devrait pas être prise en compte peuvent être spécifiés à l'aide de *state ∗()*. Cela permet de définir des exceptions facilement sans avoir à spécifier la transition correspondante dans chaque état de l'automate. Il est aussi possible de spécifier à l'automate de rester dans l'état où l'exception s'est déclenchée en utilisant *nextstate* -. La figure 3.15 représente une spécification d'exception pour l'automate de la figure 3.12. Le signal *error* provoque un traitement spécial et laisse l'automate dans l'état courant. Le signal *reset* remet l'automate dans un état de départ connu (*st1*).

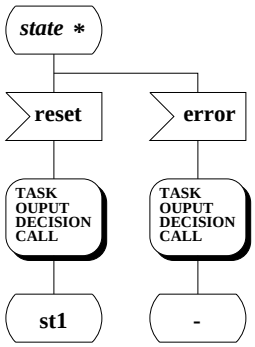


FIG. 3.15: Exception en SDL

3.6.4 Communication inter-processus

SDL supporte trois types de communications inter-processus :

- les signaux.
- les variables partagées.
- les appels de procédure à distance (SDL92).

Signaux

Chaque processus peut, lors de l'exécution d'une transition émettre des signaux (*ouput*) qui iront valider les transitions d'autres processus. Un signal peut transporter un ensemble de paramètres qui correspondent au données échangées. Un processus peut se mettre en attente d'un signal particulier en utilisant l'instruction *input*. Lorsque le signal sera consommé par le processus destinataire, les variables spécifiées recevront les valeurs des paramètres du signal.

Un signal transporte toujours implicitement l'adresse du processus émetteur, l'adresse du processus destinataire si celle-ci est explicitement spécifiée ainsi qu'un ensemble éventuel de paramètres. Lorsque le signal est extrait de la file d'attente, l'adresse du processus émetteur et placée dans la variable prédéfinie *sender*. L'adresse du destinataire est utilisée dans le cas où celui-ci ne peut pas être déterminé statiquement et l'adresse de l'émetteur peut être utilisée pour répondre à un signal.

Pour chaque signal émit par un processus, il ne doit y avoir qu'une destination. La destination peut être spécifiée :

- explicitement en précisant l'adresse (*Pid*) du destinataire (*output to [Pid]*, *ouput to sender*).
- implicitement si le signal n'a qu'un chemin possible à travers un ensemble de routes et de canaux.

Un signal ne peut être émis d'un processus vers un autre que s'il existe un chemin de communication (route et/ou canaux) entre les deux processus. Il est possible de restreindre le chemin de communication utilisé par un signal en indiquant la route ou le canal emprunté par le signal (*output via*).

Variables importées et exportées

SDL supporte la notion de variable partagée par l'intermédiaire des variables *importées* et *exportées*. Une variable déclarée exportée sera visible des autres processus qui pourront importer sa valeur. Lorsqu'un processus exécute une expression *import*, la valeur de la copie implicite de la variable exportée par l'expression *export* sera retounée. La valeur de la copie implicite disponible est la valeur de la variable à l'instant où l'expression *export* est executée.

Procédures importées et exportées

SDL92 offre le concept de procédure exportée comme une extension du concept de variable exportée. Une procédure exportée est visible par des processus autres que celui où elle est définie et pourra être appelée par les processus qui l'auront importée. Cet appel correspond à un appel de procédure à distance ou *Remote Procedure Call* [2], [15]. Un appel de procédure distante (importée) est transformé en une émission de signaux implicites pour le passage et le retour des paramètres. Le processus appelant enverra un signal contenant les paramètres de la procédure et le processus serveur retournera un signal contenant le resultat de l'exécution de la procédure. Ces signaux transiteront dans les files d'attentes des processus serveur (appel de la procédure) et appelant (retour de la procédure) comme des signaux standard. Lors de l'appel de la procédure, le processus appelant se placera dans un état d'attente implicite jusqu'à ce que celle ait fini de s'exécuter (3.16). Dans un état quelconque du processus serveur, la réception du signal d'appel de procédure à distance déclenche l'exécution de celle ci et l'envoie du résultat (3.17). Il est possible de spécifier qu'un appel de procédure à distance ne sera pas servi dans un état de l'automate en effectuant une sauvegarde (identique au save d'un signal) de l'appel de procédure. Lorsque plusieurs processus utilisent un même serveur, le processus appelant sera identifié par son *Pid*.

La figure 3.16 représente un appel de procédure à distance du coté du processus appelant et son modèle d'exécution SDL. La figure 3.17 représente le coté serveur. La figure 3.17.2 représente la transition implicite correspondant au traitement de la procédure par le processus serveur.

3.6.5 Timers

Les dépendances temporelles peuvent être modélisées à l'aide de *timers*. Un processus peut armer un timer et recevoir un signal de timeout lorsque celui ci est arrivé à expiration. Le signal de timeout

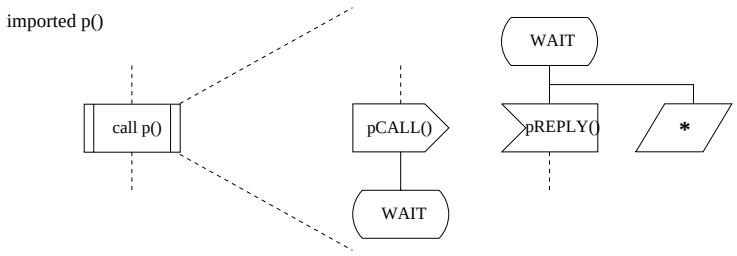
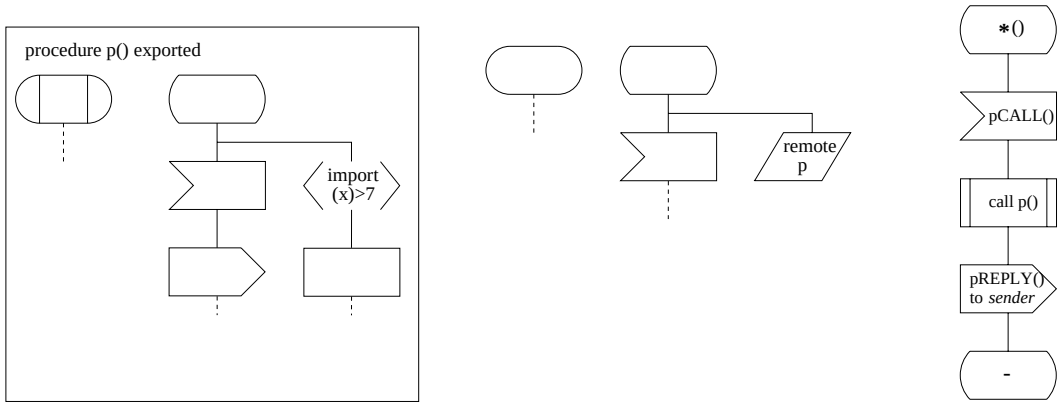


FIG. 3.16: Appel de procédure à distance en SDL : processus appelant



3.17.1: Procédure exportée

3.17.2: transition implicite

FIG. 3.17: Modèle d'exécution SDL des appels de procédure à distance : processus serveur

arrivera dans la file d'attente du processus et pourra être utilisé pour valider des transitions. Un processus peut :

- armer un timer avec une date d'expiration.
- arrêter un timer et le rendre inactif.
- tester l'état d'un timer(actif/inactif). Un timer est actif à partir du moment où il décompte et ce jusqu'à ce que son message de timeout soit consommé par le processus.

En SDL une référence de temps absolu est disponible à travers la variable *now* et peut être utilisée pour déterminer la date d'expiration d'un timer.

3.6.6 Non déterminisme

SDL permet d'exprimer le non déterminisme à l'aide des transitions spontanées et de l'expression *any*. Le non déterminisme peut être utilisé pour modéliser un comportement non fiable ou pour introduire des erreurs dans un système.

Les transition spontanées (*input none*) sont des transitions qui peuvent être activées de façon aléatoire sans qu'un signal de la file d'attente ne soit consommé. L'activation de cette transition est non déterministe et rend l'état instable.

Le constructeur *any* dénote une valeur indéterminée d'un type particulier. La valeur retournée par l'expression *any* est determinée au hasard parmi toutes les valeurs possibles du type. *Any* peut être utilisé :

- pour générer une valeur aléatoire d'un type donné (*i :=any(INTEGER)*).
- dans une alternative (*decision*) pour la rendre non déterministe (*decision any*).
- dans une transition pour la rendre non déterministe *input any*. Il ne faut pas confondre une transition *input any* qui génère un signal quelconque et exécute la transition de *input ** qui valide la transition quel que soit le signal en tête de la file d'attente.

3.7 CONCLUSION

Dans ce chapitre nous avons présenté les principales familles de langages. Chaque famille a été developpée pour répondre à des problèmes particuliers tels que les systèmes temps réels ou les protocoles de télécommunication. Ces langages incluent tous les concepts de concurrence, hiérarchie, communication et synchronisation et peuvent être utilisés pour les spécifications au niveau système.

Parmi ces langages, certains font partie des langages de spécifications formelles. Ceux-ci permettent de développer des spécifications claires et non ambigües et d'effectuer une vérification des propriétés du système décrit. Les langages de spécifications formelles permettent de décrire les propriétés des systèmes en faisant abstraction de la réalisation. Cela les rend adaptés à la spécification au niveau système. Les techniques de description formelle ont été appliquées au domaine des télécommunications. Trois langages (ESTELLE, LOTOS, SDL) dédiés à la spécification de protocoles de télécommunication et reposant sur les *FDT* ont été présentés. Le langage SDL est bien adapté à la spécification et à la validation des systèmes distribués. Son point faible réside dans la manipulation des données. Celui-ci est en partie compensé par la possibilité de définir des types de données abstraits dont les opérateurs sont décrits en C. SDL a été présenté en détail. Ce langage est utilisé pour la spécification au niveau système dans le système COSMOS.

Chapitre 4

SÉLECTION DE PROTOCOLE ET SYNTHÈSE D'INTERFACE POUR LA SYNTHÈSE SYSTÈME

La principale difficulté rencontrée lors de la synthèse à partir de spécifications systèmes est l'interprétation des schémas de communication de haut niveau. Le but de ce chapitre est de présenter une approche de la synthèse de la communication permettant de transposer une communication de haut niveau sur un schéma matériel. Cette nouvelle approche formule la synthèse de la communication comme un problème d'allocation. Dans l'approche proposée, la synthèse de la communication permet de transformer un système composé de processus qui communiquent en utilisant des primitives de haut niveau à travers des canaux abstraits en un ensemble de processus interconnectés par des bus et des signaux et partageant le contrôle de la communication. Cette approche traite à la fois la sélection de protocole et la géneration d'interface et est basée sur l'allocation d'unités de communication. Elle permet une large exploration de l'espace des solutions grâce à une sélection du protocole de communication et une synthèse du réseau d'interconnexion. Nous illustrerons l'utilité de cet approche pour allouer différents protocoles au sein d'un même système. La synthèse de la communication permet de résoudre les problèmes dû à l'abstraction des communications systèmes en choisissant une réalisation efficace pour chaque communication.

4.1 INTRODUCTION

Récemment, un nouveau niveau d'abstraction, dénomé niveau système est apparu dans le domaine de la synthèse [21] [47] [60] [61] [82] [131]. Ce mouvement a été motivé par la complexité toujours croissante des systèmes électroniques et la nécessité d'approches unifiées permettant le développement de systèmes contenant à la fois du logiciel et de matériel.

Lorsqu'on élève le niveau des spécifications, des problèmes auparavant inexistants apparaissent [54] [146]. Au niveau système les principaux concepts sont le comportement et la communication [102]. Ces deux concepts ont apporté de nouveaux problèmes tels que le partitionnement et la synthèse des communications. Le but du partitionnement est de découper la fonctionnalité d'un système en un ensemble de partitions où chaque partition peut être exécutée soit en logiciel (processeur + code) soit en matériel (ASIC) [134]. Le problème de la synthèse de la communication [12], qui apparait après le partitionnement au niveau système, est de déterminer les protocoles et les interfaces requises par les différents sous-systèmes pour communiquer.

4.1.1 Problème de la synthèse de la communication

Lors de la conception de systèmes distribués embarqués, la synthèse de la communication devient primordiale car les différents sous-systèmes communiquent inévitablement. Différents schémas et protocoles de communication peuvent être requis, de même que différentes topologies d'interconnexion. La topologie d'interconnexion et le protocole de communication influencent grandement les performances d'un système et peuvent mener à des solutions infaisables si le concepteur sous-estime le débit des communications. Les décisions basées sur des débits moyens de communication tendent à oublier les débits crête et les délais de communication dû au partage de ressources de communication qui peuvent dégrader les performances. En conséquence une large exploration de l'espace des solutions est nécessaire afin d'obtenir une solution satisfaisant aux contraintes.

4.1.2 Travaux antérieurs

La plupart des travaux effectués sur la synthèse de la communication pour le codesign s'est concentré sur la synthèse d'interface en assumant un réseau d'interconnexion fixe [47] [60]. Seuls quelques travaux ont abordé la synthèse de réseau de communication [52], [58], [150]. L'approche proposée dans [52] part d'un réseau de processeurs interconnectés par des bus et d'éventuelles mémoires. La synthèse de réseau consiste à réduire le nombre de bus nécessaires en fusionnant et ordonnancant certaines communications sur un même bus. Dans [58], la synthèse du réseau est guidée par la distribution des variables (privées ou partagées) dans la mémoire (locale ou globale). Le but est de décider du placement des variables en mémoire et de fixer le protocole de communication permettant d'accéder à celle-ci. Dans [150], Yen transpose une description composée de plusieurs processus sur une architecture composée d'éléments de calcul (*PE*) interconnectés par des bus point à point. Un nouvel élément de calcul et un bus sont crées lorsqu'il n'est plus possible d'assigner un processus à un élément de calcul déjà existant ou une communication sur un bus sans violer les contraintes. Cette approche résoud le problème de l'allocation des processeurs et des communications en même temps. Elle détermine le nombre de bus et le protocole de chacun, les messages véhiculés

par chaque bus et ordonnance les communications. Dans [29], [128] Chou et Srivastava utilisent un ensemble prédéfini de modèles d'interconnexion lors de la synthèse de la communication. De nombreux travaux sur la sélection de protocole dans le domaine de la synthèse logicielle pour les systèmes distribués est disponible [123]. De nombreux travaux ont abordé le problème de la synthèse d'interface [44], [43], [91], [96], [103], [104], [111], [118] et [143]. Dans [44], Ecker présente une méthode de transformation et d'optimisation de protocoles. Dans [103], Narayan aborde le problème de la génération d'interface entre deux modules matériel d'une spécification partitionnée. Le but est d'optimiser l'utilisation du bus en entrelacant différentes communications point à point dessus. Dans [91] [104], Lin et Nayaran considèrent le problème de la génération et de l'adaptation d'interface avec conversion automatique de protocole avec une ou deux interfaces fixées. Une modélisation object de la bibliothèque de communication est présentée dans [139]. Madsen [96] considère le problème de l'adaptation d'interface entre une interface fixe et un medium de communication choisi durant le partitionnement. Un modèle d'état qui permet de décrire à la fois les propriétes fonctionnelles et temporelles d'une interface est décrit par Ravn dans [118]. Un autre modèle utilisant des graphes de transitions permettant la spécification d'interface synchrones et asynchrones complexes est décrit par Vanbekbergen dans [143]. Des approches où la communication est réalisée avec une mémoire partagée sont détaillées dans [71] [59] et [29]. Dans [71] le problème de l'interfaçage entre une mémoire et un coprocesseur ou un processeur d'entrée sortie est adressé. Dans [59], Gupta aborde aussi le problème d'interface entre un processeur (logiciel) et un coprocesseur (matériel). Dans cette approche la communication peut être réalisée à travers une mémoire ou à travers un bus entre le processeur et le coprocesseur. Différents modèles de communication sont disponibles (bloquant, non bloquant). Cette approche adresse principalement l'interfaçage logiciel/matériel. Dans [128], Srivastava commence après le partitionnement avec un graphe de processus, un modèle d'architecture cible et transpose les communications sur les ressources physiques de communication disponibles. Un seul modèle de communication est disponible, le fifo à lecture et écriture simple. Quand les ressources de communication ne supportent pas directement ce protocole, il est émulé. Ce travail vise principalement le domaine des systèmes hétérogènes distribués temps réel.

4.1.3 Contribution

Les principaux objectifs de notre méthode de synthèse de la communication sont :

- de pouvoir choisir entre plusieurs schéma de communication
- de pouvoir modéliser la communication indépendament du comportement. La spécification du système doit être indépendante de la spécification de la communication afin de permettre des changements dans le modèle de communication sans modification de la spécification du système.
- de pouvoir choisir entre plusieurs protocoles de communication dans une bibliothèque.
- d'avoir une méthode de synthèse de la communication basée sur une fonction coût et des contraintes.

Ce chapitre décrit une nouvelle approche de synthèse de la communication. Cette tâche est formulée comme un problème d'allocation qui choisit dans une bibliothèque un ensemble d'unités de communication qui implémente les communications requises par les differents processus. A notre

connaissance, aucune approche ne présente la synthèse de la communication comme un problème d'allocation. La principale contribution de ce chapitre est de présenter la synthèse de la communication comme un problème d'allocation. Celui ci peut être résolu de façon automatique ou interactive.

Comparé aux approches classiques de synthèse de la communication, les principaux avantages de notre méthode sont :

- une synthèse complète de la communication :
  - sélection de protocole et synthèse de réseau de communication.
  - synthèse d'interface.
- une large exploration de l'espace des solutions à travers la sélection de protocole et la synthèse de réseau de communication.
- La formulation de la synthèse de la communication comme un problème d'allocation permettant l'utilisation de nombreux algorithmes pour le résoudre.

Les limitations de notre approche sont :

- la nécessité qu'a l'utilisateur de fournir une bibliothèque de communication. Il n'est pas possible d'utiliser un protocole qui n'est pas décrit dans la bibliothèque.
- la nécessité d'estimations réalistes [150] [135], permettant de guider la sélection de protocole et la synthèse de réseau.

Dans les paragraphes suivant nous présentons notre méthode de synthèse de la communication. Le paragraphe suivant présente notre modèle de communication. Le paragraphe 4.2.4 présente le concept d'unité de communication. Le problème de la synthèse de la communication est détaillé au paragraphe 4.3. Les différentes étapes de la synthèse de la communication sont détaillées aux paragraphes 4.3.1 et 4.3.2. Dans le paragrahe 4.3.3 nous proposerons une approche de la synthèse de la communication comme un problème d'allocation des unités de communication. Le paragraphe 4.4 décrit une approche transformationnelle de la synthèse de la communication basée sur des primitives d'allocation d'unités de communication et de synthèse d'interface.

4.2 MODÈLE DE COMMUNICATION

Dans cette section nous décrivons un paradigme qui autorise la modélisation pour la synthèse d'un grand nombre de schéma de communication système.

4.2.1 Modèle de communication

Nous utiliserons la modélisation de la communication présentée dans [78].

Conceptuellement un canal est un objet capable d'exécuter une communication avec un protocole défini. Il offre un ensemble de primitives de communication qui sont utilisées par les processus pour communiquer. Une primitive de communication est une procédure standard utilisée pour la communication entre processus. Un processus qui désire communiquer effectue un appel à distance d'une primitive de communication en lui transmettant en argument les données à émettre ou à recevoir. La communication est effectée par la procédure. Cela permet d'encapsuler les détails de la

communication en ne laissant apparaitre que les données transmises. L'accès au canal est controlé par cet ensemble de primitives qui constitue la seule partie visible du canal et repose sur l'appel de procédure à distance [2], [15] de ces primitives. Un processus qui désire communiquer à travers un canal invoque une des primitives de communication offerte par l'intermédiaire d'un appel de procédure à distance. Une fois l'appel de procédure à distance effectué, la communication est exécutée indépendemment du processus appelant par le canal de communication. La communication est totalement transparente pour le processus appelant. Cela permet au processus de communiquer par des schémas de haut niveau en ne faisant aucune hypothèse sur la réalisation de la communication.

Il n'y a pas d'ensemble prédéfini de primitives de communication. Elle sont définies comme des procédures standards offertes par le canal de communication. Chaque application peut requérir un ensemble différent de primitives de communication. Il devient possible d'avoir plusieurs implémentations différentes d'une même primitive de communication.

L'utilisation d'appels de procédure à distance permet de séparer la spécification de la communication du reste du système. Ces modèles de communication peuvent être décrits séparément et les détails d'implémentation sont cachés. Dans notre approche, les détails d'implémentation et le protocole sont cachés dans une bibliothèque de composants de communication.

4.2.2 Système communicant et système interconnecté

Un système comunicant est défini comme un ensemble de processus communicant à travers des schémas de communication de haut niveau. Les détails du protocole de communication sont encapsulés dans des primitives de communication qui sont utilisées par les processus. Le but étant de pouvoir décrire des systèmes communicant sans spécifier les détails de cette communication. La figure 4.1 représente un système communicant. Partant d'un système communicant, plusieurs

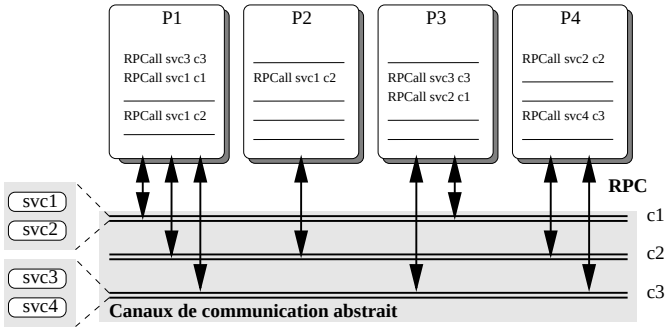


FIG. 4.1: Système communicant

réalisations de la communication utilisant différents protocoles sont possibles.

Un système interconnecté est un système composé de processeurs interconnectés à travers des bus et des signaux. Les processeurs communiquent à travers des shémas de bas niveau tels que des affectations de signaux avec un protocole bien définis. La figure 4.14 représente un système interconnecté correpondant au système de la figure 4.1. Partant d'un système communicant plusieurs étapes de raffinement sont nécessaires pour obtenir un système interconnecté. Elles sont décrites en

section 4.3.

Au niveau système, dans un langage tel que SDL, le niveau d'abstraction fait que l'on a affaire à des systèmes communicants. A l'inverse, les langages de description matériels tels que VHDL ou Verilog manipulent des systèmes interconnectés. Le but de la synthèse de la communication est de transformer un système communicant en un système interconnecté.

4.2.3 Notion de canal abstrait

Afin de représenter les schémas de communications systèmes sans faire d'hypothèses contraignantes sur leur réalisation nous utiliserons le concept de canal abstrait. Lors de la traduction d'une spécification pour la synthèse il nous est nécessaire de représenter la communication présente dans la forme intermédiaire SOLAR. Néanmoins si l'on représente cette communication avec des canaux de communications (*channel unit*) cela revient à choisir un protocole et une réalisation pour chaque communication, ce que l'on veut éviter. Fixer prématurément un protocole et une réalisation pour une communication peut évincer des réalisations plus efficaces.

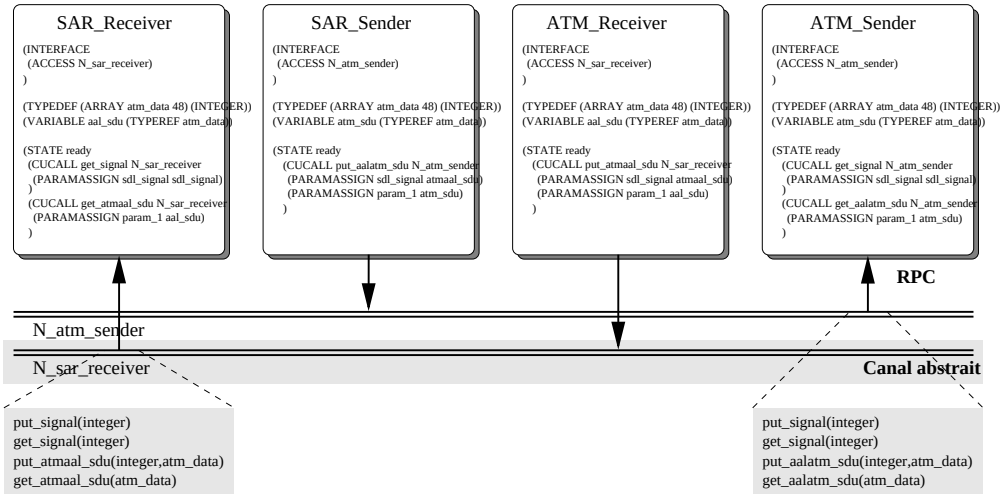


FIG. 4.2: Spécification de la communication à l'aide de canaux abstraits

Un canal abstrait est un canal de communication qui spécifie les primitives de communication requises par les processus qui l'utilisent mais pas son protocole ni sa réalisation. Il offre l'ensemble des primitives de communication qui sont utilisées par les processus pour communiquer. Un processus qui désire communiquer à travers un canal fait un appel de procédure à distance d'une primitive de communication de ce canal. Une fois l'appel de procédure à distance effectué, la communication est exécutée indépendemment du processus appelant par le canal de communication. Cela permet de spécifier des schémas de communication de haut niveau en ne faisant aucune hypothèse sur leur réalisation. Au niveau système, un système est donc représenté par un ensemble de processus communicants à travers des canaux abstraits (figure 4.2). Il s'agit d'un système communicant.

La figure 4.2 représente un ensemble de processus communicant à travers un réseau de canaux abstraits. Le canal abstrait *N_sar_receiver* offre les primitives de communication *put_signal()*,

get_ signal(), *put_ atmaal_ sdu()* et *get_ atmaal_ sdu()*. Ces primitives sont utilisées par les processus *sar_ receiver* et *atm_ receiver* pour communiquer. Le canal abstrait *N_ atm_ sender* offre les primitives de communication *put_ signal()*, *get_ signal()*, *put_ aalatm_ sdu()* et *get_ aalatm_ sdu()*.

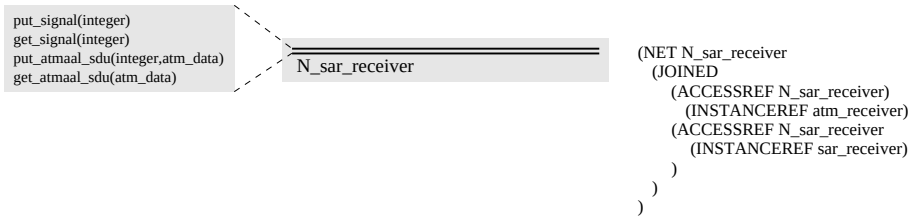


FIG. 4.3: Concept de canal abstrait et sa représentation SOLAR

En SOLAR la représentation du canal abstrait est le *net* (figure 4.3). Les primitives offertes par le canal abstrait ne sont pas explicitement spécifiées dans celui-ci mais elles sont déduites des processus qui l'utilisent. Un canal abstrait a donc la particularité d'offrir toutes les primitives de communications requises.

4.2.4 Modélisation des unités de communication

Nous utiliserons la modélisation de la communication présentée dans [78]. Cette modélisation est rappelée en section 2.2.3. Nous définissons une unité de communication comme l'abstraction d'un composant physique. Les unités de communication sont sélectionnées dans la bibliothèque et instanciées lors de la synthèse de la communication.

D'un point de vue conceptuel, une unité de communication est un objet capable d'exécuter une ou plusieurs primitives de communication implémentant un protocole spécifique. Une unité de communication est composée d'un ensemble de primitives de communication, un éventuel contrôleur de communication, une interface et un ensemble d'interconnexions.

Le contrôleur détermine le protocole de la communication, assure la synchronisation de la communication et l'absence de conflits d'accès. Il offre un ensemble de ressources de communication (bus, arbitres de bus, mémoire) qui sont utilisées par les différentes primitives de communication. La complexité du contrôleur peut varier d'une simple file d'attente *first-in-first-out* à un protocole complexe en couches. Pour certain protocole de communication tel que le *handshake*, le contrôleur est complètement distribué entre les primitives de communication. Les primitives de communication intéragissent avec le contrôleur pour effectuer la communication.

Les primitives de communication représentent une abstraction du canal de communication et encapsulent le protocole d'accès. Les primitives de communication prennent un ensemble de paramètres qui correspondent aux données échangées et accèdent à l'unité de communication avec le protocole requis. Une fois les données transmises à l'unité de communication, celle-ci effectue la communication. Les primitives de communication encapsulent le protocole d'accès au canal et font le lien entre les paramètres échangés lors d'une communication et les signaux réalisant le protocole d'accès au canal (figure 4.4). Chaque primitive de communication contient un ensemble de ports qui correspondent à la réalisation du protocole. Les primitives de communication déterminent le protocole

d'échange des données entre les processus et l'unité de communication. La partie paramètres d'une primitive de communication correspond à la partie spécification de la communication et constitue la partie visible de la primitive. Elle est relative aux systèmes communicants. La partie ports, correspondant à la partie réalisation de la communication est relative aux systèmes interconnectés. Du point de vue de la spécification du système, cette partie des primitives est cachée.

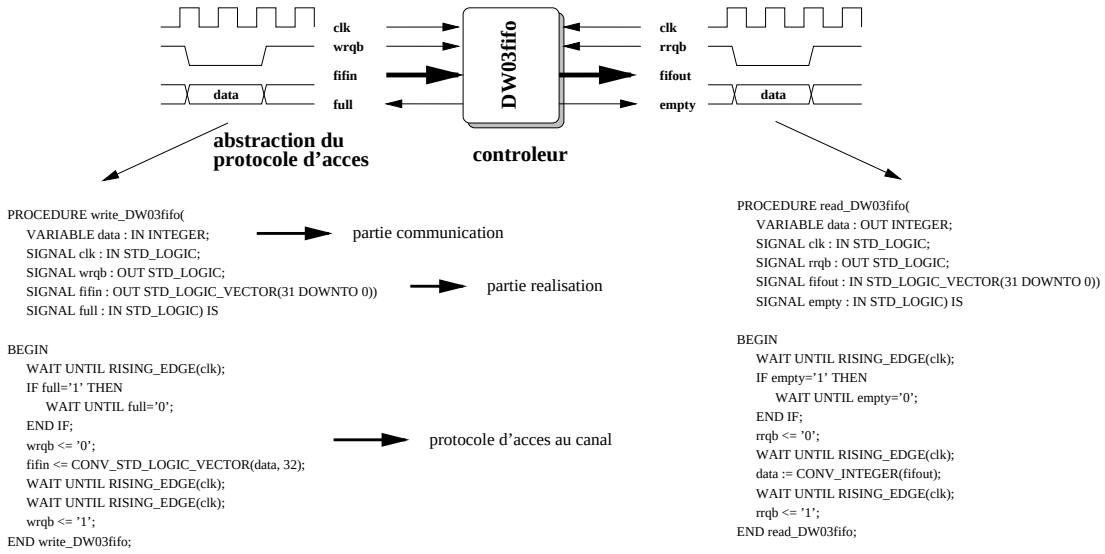


FIG. 4.4: Primitives de communication et abstraction d'un canal de communication

Une unité de communication contient aussi une interface sur laquelle les primitives de communication lisent et écrivent. Cette interface représente les ports et les interconnexions nécessaires aux différents processus utilisant cette unité de communication. Tous les accès à l'interface d'une unité de communication sont faits à travers les primitives de communication.

Ce schéma modulaire permet de cacher les détails d'implémentation dans une bibliothèque C/VHDL (figure 4.5) où une unité de communication peut avoir différentes implémentations en fonction de l'architecture cible (logiciel/matériel). Cette approche permet la réutilisation de bibliothèques de communication déjà existantes. Il est seulement nécessaire d'abstraire les canaux pour pouvoir les réutiliser. L'opération d'abstraction consiste en l'écriture de primitives de communication C/VHDL permettant d'accéder au canal de communication. Ainsi réalisée l'abstraction de la communication permet une spécification modulaire autorisant le traitement de la communication indépendamment du reste du système.

La figure 4.4 représente le processus d'abstraction d'une unité de communication (une file d'attente *first-in-first-out*) à partir de son protocole d'accès. Les deux procédures permettent de lire et d'écrire un paramètre entier dans la file d'attente.

```
LIBRARY IEEE;
        USE IEEE.std_logic_1164.ALL;
        USE IEEE.std_logic_arith.ALL;
        USE IEEE.std_logic_unsigned.ALL;
```

```
PACKAGE package_DW03fifo IS

PROCEDURE write_DW03fifo(
        VARIABLE data : IN INTEGER;
        SIGNAL clk : IN STD_LOGIC;
```

```
        SIGNAL wrqb : OUT STD_LOGIC;
        SIGNAL fifin : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
        SIGNAL full : IN STD_LOGIC);

PROCEDURE read_DW03fifo(
        VARIABLE data : OUT INTEGER;
        SIGNAL clk : IN STD_LOGIC;
        SIGNAL rrqb : OUT STD_LOGIC;
        SIGNAL fifout : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        SIGNAL empty : IN STD_LOGIC);
END package_DW03fifo;

COMPONENT DW03fifo
GENERIC (data_width, depth, level : INTEGER);
PORT( fifin : IN STD_LOGIC_VECTOR(data_width-1 DOWNTO 0);
wrqb : IN STD_LOGIC;
rrqb : IN STD_LOGIC;
csb : IN STD_LOGIC;
clk : IN STD_LOGIC;
reset : IN STD_LOGIC;
fifout : OUT STD_LOGIC_VECTOR(data_width-1 DOWNTO 0);
full :  OUT STD_LOGIC;
empty : OUT STD_LOGIC;
threshold : OUT STD_LOGIC);
END COMPONENT;

PACKAGE BODY package_DW03fifo IS

PROCEDURE write_DW03fifo(
        VARIABLE data : IN INTEGER;
        SIGNAL clk : IN STD_LOGIC;
        SIGNAL wrqb : OUT STD_LOGIC;
        SIGNAL fifin : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
        SIGNAL full : IN STD_LOGIC) IS

BEGIN
  WAIT UNTIL RISING_EDGE(clk);
  IF full ='1' THEN
      WAIT UNTIL full='0';
  END IF;
  wrqb <='0';
  fifin <= CONV_STD_LOGIC_VECTOR(data, 32);
  WAIT UNTIL RISING_EDGE(clk);
  WAIT UNTIL RISING_EDGE(clk);
  wrqb <='1';
END write_DW03fifo;

PROCEDURE read_DW03fifo(
        VARIABLE data : OUT INTEGER;
        SIGNAL clk : IN STD_LOGIC;
        SIGNAL rrqb : OUT STD_LOGIC;
        SIGNAL fifout : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        SIGNAL empty : IN STD_LOGIC) IS

BEGIN
  WAIT UNTIL RISING_EDGE(clk);
  IF empty='1' THEN
      WAIT UNTIL empty='0';
  END IF;
  rrqb <='0';
  WAIT UNTIL RISING_EDGE(clk);
  data:=CONV_INTEGER(fifout);
  WAIT UNTIL RISING_EDGE(clk);
  rrqb <='1';
END read_DW03fifo;

END pk_DW03fifo;
```

FIG. 4.5: Abstraction d'une unité de communication VHDL possédant un contrôleur de communication

La figure 4.5 représente deux procédures VHDL permettant de lire (procédure *read_DW03fifo()*) et d'écrire (procédure *write_DW03fifo()*) dans le fifo *DW03_fifo_s_sf* de la bibliothèque designware de Synopsys. Dans ce cas le contrôleur de communication est le *fifo* et le protocole offert est *first-in-first-out*. La figure 4.6 représente une unité de communication ne possédant pas de contrôleur de communication. Le protocole de communication *handshake* est distribué entre les deux procédures *send_handshake()* et *receive_handshake()*. Chacune des quatres procédures de communication accepte un paramètre de type entier (paramètre formel *data*). Ce paramètre correspond à la donnée transférée sur le canal.

```
LIBRARY IEEE;
        USE IEEE.std_logic_1164.ALL;
        USE IEEE.std_logic_arith.ALL;
        USE IEEE.STD_logic_unsigned.ALL;

PACKAGE package_handshake IS

PROCEDURE send_handshake(
        VARIABLE data : IN INTEGER;
        SIGNAL clk : IN STD_LOGIC;
        SIGNAL srdyqb : OUT STD_LOGIC;
        SIGNAL dbus : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
        SIGNAL rrdyqb : IN STD_LOGIC);

PROCEDURE receive_handshake(
        VARIABLE data : OUT INTEGER;
        SIGNAL clk : IN STD_LOGIC;
        SIGNAL srdyqb : IN STD_LOGIC;
        SIGNAL dbus : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        SIGNAL rrdyqb : OUT STD_LOGIC);

END package_handshake;

PACKAGE BODY package_handshake IS

PROCEDURE send_handshake(
        VARIABLE data : IN INTEGER;
        SIGNAL clk : IN STD_LOGIC;
        SIGNAL srdyqb : OUT STD_LOGIC;
        SIGNAL dbus : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
        SIGNAL rrdyqb : IN STD_LOGIC) IS

BEGIN
  WAIT UNTIL RISING_EDGE(clk);
  dbus <= CONV_STD_LOGIC_VECTOR(INTEGER(data), 32);
  srdyqb <='0';
  LOOP
      WAIT UNTIL RISING_EDGE(clk);
      EXIT WHEN rrdyqb ='0';
  END LOOP;
```

```
  srdyqb <='1';
END send_handshake;

PROCEDURE receive_handshake(
        VARIABLE data : OUT INTEGER;
        SIGNAL clk : IN STD_LOGIC;
        SIGNAL srdyqb : IN STD_LOGIC;
        SIGNAL dbus : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        SIGNAL rrdyqb : OUT STD_LOGIC) IS

  VARIABLE i : INTEGER;

BEGIN
  WAIT UNTIL RISING_EDGE(clk);
  rrdyqb <='0';
  LOOP
      WAIT UNTIL RISING_EDGE(clk);
      EXIT WHEN (srdyqb ='0');
  END LOOP;
  i:=CONV_INTEGER(dbus);
  rrdyqb <='1';
  data:=i;
END receive_handshake;

END package_handshake;
```

FIG. 4.6: Abstraction d'une unité de communication VHDL sans contrôleur de communication

4.3 SYNTHÈSE DE LA COMMUNICATION

La synthèse de la communication a pour but de transformer un système composé de processus communicant à l'aide de primitives de haut niveau à travers des canaux abstraits en un ensemble de processeurs interconnectés communicant à travers des signaux et partageant le contrôle de la communication. A partir d'une spécification système, deux étapes sont nécessaires (figure 4.7). La première vise à choisir le protocole de chaque communication et à fixer la topologie du réseau d'interconnexion. Cette étape est nommée sélection de protocole ou allocation d'unités de communication. La seconde, appelée synthèse d'interface, génère et adapte les interfaces des différents processus au réseau de communication choisi. Dans cette approche la synthèse de la communication est réalisée en deux étapes. L'avantage de procéder en deux étapes est d'avoir un contrôle plus fin sur l'activité de raffinement. Cette synthèse procède par étapes successives qui évitent de prendre des décisions très tôt, lequelles risquent d'écarter plusieurs alternatives de réalisation. Le flôt de la synthèse de la communication est représenté en figure 4.7.

La synthèse de la communication utilise une bibliothèque de communication et une bibliothèque d'implémentation. La bibliothèque de communication est utilisée lors de l'allocation des unités de communication et est décrite en SOLAR (figure 4.8). Elle contient le protocole et le prototype des primitives de communication offertes par chaque canal. Les ports utilisés par les primitives de communication ainsi que l'interface du contrôleur de communication sont décrits dans ce fichier d'abstraction. Le corps des primitives et du contrôleur sont décrits dans la bibliothèque C/VHDL. La bibliothèque de communication est utilisée lors de l'allocation des unités de communication pour déterminer si le canal alloué de la bibliothèque est capable de fournir toutes les primitives requises par le canal abstrait. La synthèse d'interface utilise une bibliothèque d'implémentation C/VHDL des unités de communication. Cette bibliothèque contient la réalisation des canaux de communication (corps des primitives de communication et du contrôleur) en C ou VHDL. Une bibliothèque de communication est représentée en figure 4.8. La bibliothèque VHDL correspondante est représentée en figure 4.5. La bibliothèque de communication est une abstraction pour la synthèse de la bibliothèque C/VHDL.

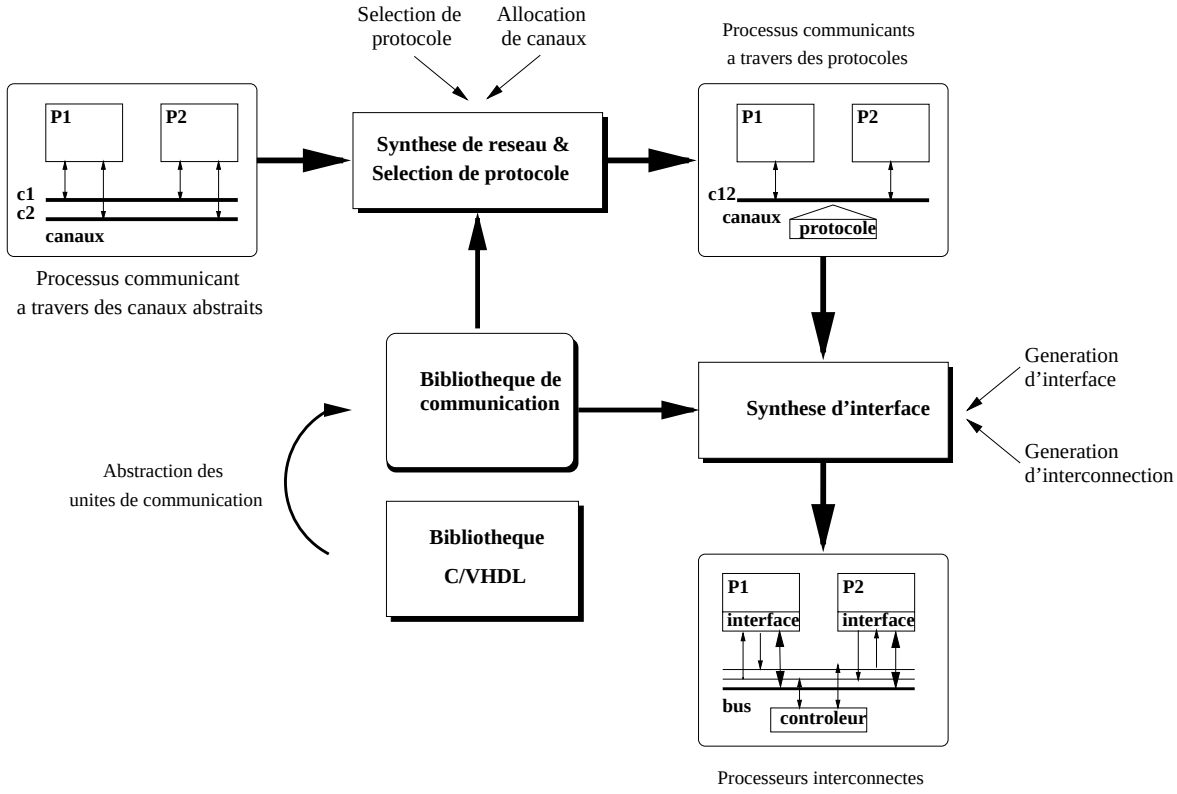


FIG. 4.7: Flot de la synthèse de la communication

```
(SOLAR  Library_of_Communication_Protocols
  (CHANNELUNIT  DW03fifo (PROPERTY protocol fifo )
    (VIEW  DW03fifo_behaviour
      (LIBRARY VHDL "IEEE" )
      (LIBRARY VHDL "DW03" )
      (LIBRARY VHDL "DWARE" )
      (LIBRARY VHDL "WORK.package_DW03fifo.ALL" )
      (INTERFACE
        (PARAMETER data_width (INTEGER) (INITIALVALUE 32))
        (PARAMETER depth (INTEGER) (INITIALVALUE 64))
        (PARAMETER level (INTEGER) (INITIALVALUE 63))
        (PORT fifin (DIRECTION  IN )
                    (TYPEREF std_logic_vector(31 downto 0 )))
        (PORT wrqb (DIRECTION  IN ) (TYPEREF std_logic ))
        (PORT rrqb (DIRECTION  IN ) (TYPEREF std_logic ))
        (PORT csb (DIRECTION  IN ) (TYPEREF std_logic ))
        (PORT clk (DIRECTION  IN ) (TYPEREF std_logic ))
        (PORT reset (DIRECTION  IN ) (TYPEREF std_logic ))
        (PORT fifout (DIRECTION  OUT )
                     (TYPEREF std_logic_vector(31 downto 0 )))
        (PORT full (DIRECTION  OUT ) (TYPEREF std_logic ))
        (PORT empty (DIRECTION  OUT ) (TYPEREF std_logic ))
        (PORT threshold (DIRECTION  OUT ) (TYPEREF std_logic ))

        (METHOD  write_DW03fifo
          (FOREIGN VHDL "package_DW03fifo" )
          (PARAMETER  data (DIRECTION  IN ) (INTEGER ))
          (PORT clk (DIRECTION  IN ) (TYPEREF std_logic ))
          (PORT wrqb (DIRECTION  OUT ) (TYPEREF std_logic ))
          (PORT fifin (DIRECTION  OUT )
                      (TYPEREF std_logic_vector(31 downto 0 )))
          (PORT full (DIRECTION  IN ) (TYPEREF std_logic ))
          (FOREIGN VHDL "WORK.package_DW03fifo.ALL")
        )
        (METHOD  read_DW03fifo
          (FOREIGN VHDL "package_DW03fifo" )
          (PARAMETER  data (DIRECTION  OUT ) (INTEGER ))
          (PORT clk (DIRECTION  IN ) (TYPEREF std_logic ))
          (PORT rrqb (DIRECTION  OUT ) (TYPEREF std_logic ))
          (PORT fifout (DIRECTION  IN )
                       (TYPEREF std_logic_vector(31 downto 0 )))
          (PORT empty (DIRECTION  IN ) (TYPEREF std_logic ))
          (FOREIGN VHDL "WORK.package_DW03fifo.ALL")
        )
      )
      (CONTENTS
        (FOREIGN VHDL "DW03.DW03_fifo_s_sf_cfg_sim"
        )
      )
    )
```

FIG. 4.8: Bibliothèque de communication SOLAR

4.3.1 Sélection de protocole et allocation des unités de communication

L'allocation d'unités de communication prend en entrée un ensemble de processus communicant à travers des canaux abstraits (figure 4.1) et une bibliothèque d'unités de communication (figure 4.9). Ces unités de communication sont une abstraction de composants physiques. Cette étape

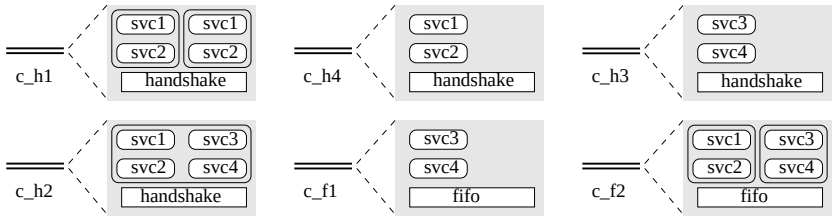


FIG. 4.9: Bibliothèque de communication

choisit dans la bibliothèque un ensemble d'unités de communication capables d'exécuter toutes les primitives de communication requises par les processus. Le choix d'une unité de communication particulière ne dépend pas seulement de la communication à exécuter (données à transférer) mais aussi des performances requises et de la réalisation des processus concernés. Cette étape fixe le protocole de chaque primitive de communication en choisissant une unité de communication avec un protocole défini pour chaque canal abstrait. Elle fixe aussi la topologie du réseau d'interconnexion en déterminant le nombre d'unités de communication alloués et les canaux abstraits exécutés sur chacun d'entre eux.

Les canaux abstraits spécifient les primitives de communication requises mais ne spécifient pas la réalisation du canal. Plusieurs canaux abstraits peuvent donc être réalisés par une seule unité de communication si celle-ci fournit toutes les primitives requises et est capable d'exécuter plusieurs communications indépendantes. Cette opération est dénommée *fusion* de canaux abstraits. Tous les canaux fusionnés s'exécuterons avec le même protocole mais les différentes communications, correspondant aux différents canaux abstraits fusionnés, resteront indépendantes. La figure 4.10 représente la fusion de deux canaux abstraits *c1* et *c3* de la figure 4.1 sur l'unité de communication *c_h1*. L'unité de communication *c_h4* implémente la communication offerte par le canal abstrait *c2*.

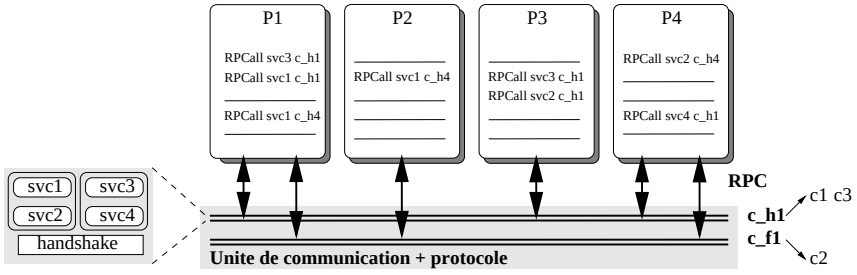


FIG. 4.10: Fusion de canaux abstraits sur une unité de communication

Fusionner plusieurs canaux abstraits sur une seule unité de communication permet de partager

un même medium de communication entre plusieurs communications abstraites. Les différents canaux abstraits seront multiplexés par l'unité de communication. L'opération de fusion permet de réduire les interfaces et les interconnexions nécessaires en allouant une seule ressource de communication pour plusieurs communications abstraites.

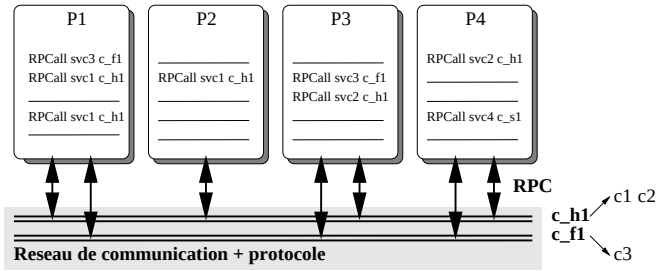


FIG. 4.11: Système après allocation des unités de communication

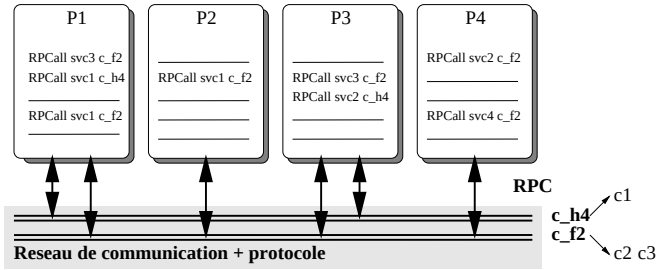


FIG. 4.12: Alternative d'allocation

Un exemple d'allocation d'unités de communication pour le système de la figure 4.1 est donné en figure 4.11. En utilisant la bibliothèque de communication donnée en figure 4.9, l'unité de communication *c_h1* a été alloué pour exécuter la communication offerte par les canaux abstraits *c1* et *c2*. l'unité de communication *c_h1* est capable d'exécuter deux communications indépendantes utilisant les services *svc1* et *svc2*. L'unité de communication c_f1 a été allouée pour le canal abstrait *c3*. Une autre solution aurait été de fusionner les canaux *c2* et it c3 et d'allouer *c_f2* pour exécuter cette communication. *c_h4* aurait put être alloué pour *c1*. Cette solution est représentée figure 4.12.

4.3.2 Synthèse d'interface

La synthèse d'interface choisit dans la bibliothèque d'implémentation (figure 4.13) une réalisation pour chaque unité de communication et génère les interfaces et interconnexions requises par les processus utilisant les unités de communication (figure 4.14). L'implémentation de chaque unité de communication sera choisie dans la bibliothèque d'implémentation en fonction des taux de transfert, de la mémoire requise et de la taille des bus et de l'interface. La bibliothèque peut contenir plusieurs implémentations de la même unité de communication. Les interfaces des différents processus sont adaptées en fonction de l'implémentation choisie et interconnectées. Une approche pour déterminer

la taille d'un bus qui implémente un ensemble de canaux est présentée dans [50] and [103] ou pour interfacer des protocoles incompatibles dans [104]. Vahid [133], propose une approche de synthèse d'interface permettant de réduire les interfaces en partageant les ports d'entrée/sortie entre différentes fonctionnalités. Dans [143] une approche de synthèse automatique d'interface à partir d'un graphe de transitions est présenté.

Le résultat de la synthèse d'interface est un ensemble de processeurs interconnectés communicant à travers des bus et des signaux et d'éventuels contrôleurs de communication provenant de la bibliothèque d'implémentation tels que des arbitres de bus, des files d'attente. Avec cette approche, il est possible de transposer une spécification de communication sur n'importe quel protocole, du simple *rendez-vous* à un protocole plus complexe.

La synthèse d'interface réalise la génération des interfaces de chaque processus utilisant le canal, ajoute l'eventuel contrôleur de communication et génère les interconnexions correspondantes. Conceptuellement les primitives de communication sont extraites du canal pour être incluses dans chaque processeur les utilisant. Les procédures de communication deviennent locales et les appels de procédure à distance sont remplacés par des appels locaux. Les procédures de communication lisent et écrivent alors sur les ports de chaque processeur. Le corps des primitives de communication est extrait de la bibliothèque d'implémentation C/VHDL et est inclus dans chaque processus utilisant le canal.

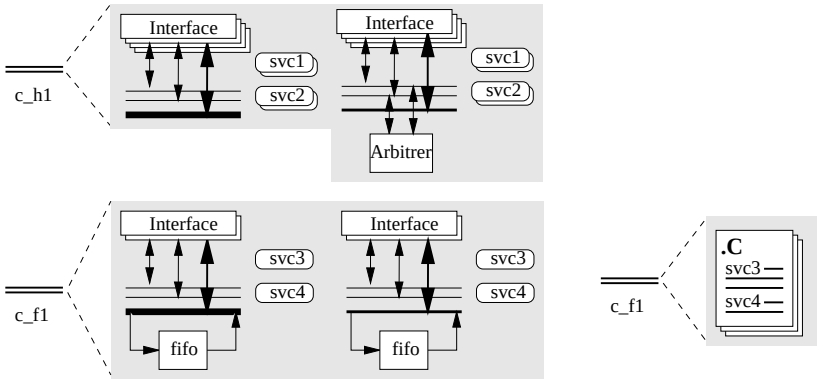


FIG. 4.13: Bibliothèque d'implémentation

Partant du système de la figure 4.11, le résultat de la synthèse d'interface est détaillé en figure 4.14. L'unité de communication *c_h1* a deux réalisations possibles, l'une avec un arbitre de bus externe pour arbitrer l'accès au bus, l'autre avec l'arbitreur distribué dans les interfaces. Chacune des deux implémentations peut être sélectionnée.

4.3.3 Formulation de la synthèse de la communication comme un problème d'allocation

La synthèse de la communication peut être formulée comme un problème d'allocation visant à déterminer le nombre et la nature des unités de communication nécessaires pour implémenter un réseau abstrait de communication. Etant donné :

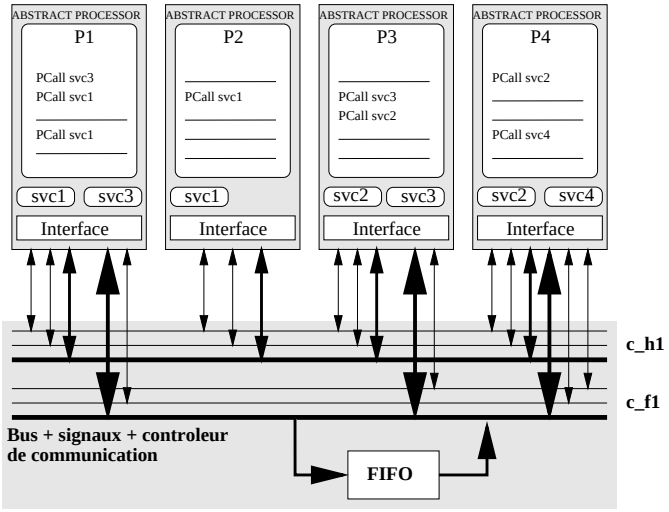


FIG. 4.14: Système après la synthèse d'interface

1. un ensemble de processus communicant à travers des primitives de haut niveau.
2. une bibliothèque d'unités de communication implémentant des primitives de communication avec un protocole spécifique.

L'objectif est d'allouer et d'instancier un ensemble d'unités de communication pour implémenter le réseau abstrait. L'allocation d'unités de communication pour des canaux abstraits est liée au compromis de performance/surface classique. Le choix d'une unité de communication ne depend pas seulement des canaux abstraits exécutés dessus mais aussi des performances requises, de la taille des interfaces et de la réalisation des processus communicants. Toutes ces contraintes peuvent être réunies dans une fonction coût réduite par l'algorithme d'allocation. Cette étape est similaire à l'allocation/binding des unités fonctionnelle en synthèse de haut niveau [55] [74]. La plupart des algorithmes d'allocation utilisés en synthèse de haut niveau peuvent être utilisés pour résoudre le problème de la synthèse des communications [98]. Ce schéma permet d'allouer plusieurs protocoles différents dans le même système.

L'algorithme d'allocation doit instancier un ensemble d'unité de communication de la bibliothèque en minimisant une fonction coût et en respectant un ensemble de contraintes. La fonction coût peut contenir des paramètres tels que :

– le nombre d'unités de communication allouées.

Les contraintes peuvent concerner :

– le taux de transfert maximal requis.
– le taux de transfert moyen disponible.
– le temps de latence d'une communication.
– la taille de buffer nécessaire.
– le protocole de la communication.

Ce problème peut être formulé comme un problème de clique (sous graphe complet) [32] et résolu par des algorithmes connus [98]. Ce problème étant NP complet [32], seule des heuristiques donnant des solutions approchées peuvent être développées.

Soit une ensemble d'unités de communication appartenant à une bibliothèque et un ensemble de canaux abstraits spécifiant des communications. On peut construire un graphe de compatibilité de la façon suivante (figure 4.15) :

- un sommet *canal abstrait* (sommet blanc) et un sommet *unité de communication* (sommet gris) sont compatibles (reliés par un arc) si l'unité de communication fournit toutes les primitives requises par le canal abstrait.
- deux sommets *canaux abstraits* sont compatibles (traits pointillés) si ceux ci peuvent être fusionnés sur une même unité de communication.

L'algorithme d'allocation consiste alors à trouver le nombre minimum de cliques (sous graphe complet) couvrant le graphe de compatibilité et respectant les containtes fixées. Chaque clique possède un sommet *unité de communication* correspondant à l'élément de bibliothèque alloué et un ou plusieurs sommets *canaux abstraits* correspondant au canaux fusionnés. Une clique devra posséder un nombre de sommets *canaux abstraits* inférieur ou égal au nombre maximum de communications indépendantes acceptables sur le sommet *unité de communication*.

La figure 4.15 résume la formulation de la synthèse de la communication comme un problème d'allocation. La bibliothèque contient trois unités de communication, *CU_1*, *CU_2*, *CU_3* pouvant exécuter respectivement 1, 2 et 2 communications indépendantes et trois canaux abstrait *C_1*, *C_2*, *C_3* offrant des primitives *send()/rcv()* et *put()/get()*.

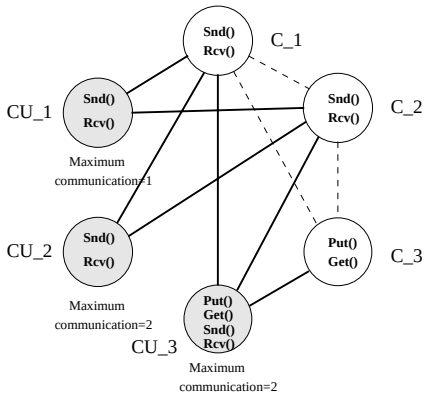
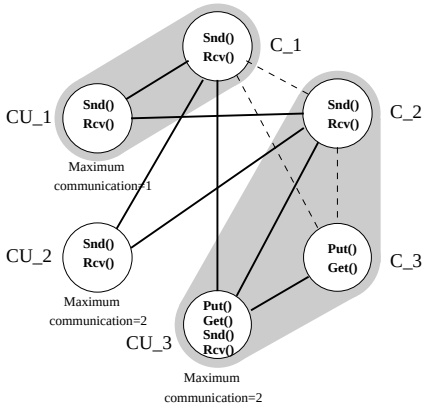


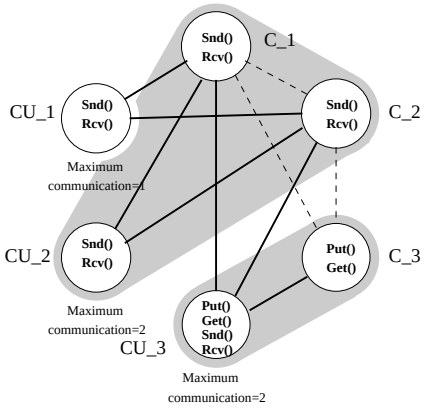
FIG. 4.15: Graphe de compatibilité pour l'allocation

La figure 4.16 donne deux exemples de partitionnement en cliques possibles. Différentes solutions peuvent être obtenues en faisant varier les contraintes.

Une seconde formulation de la synthèse de la communication peut être faite sous forme d'un arbre de recherche. L'algorithme consiste à construire un arbre énumérant toute les solutions possibles d'allocation. Cet arbre de décision énumère pour chaque canal abstrait, l'ensemble des unités de communication de la bibliothèque qui peuvent être alloués. Les nœuds de l'arbre sont les canaux



4.16.1: Solution d'allocation



4.16.2: Solution d'allocation

FIG. 4.16: Allocation d'unités de communication à l'aide de cliques

abstraits. Chaque nœud aura autant d'arcs sortant qu'il y a d'unités de communication pouvant le réaliser. Les nœuds feuilles sont vides. Chaque chemin de la racine jusqu'à une feuille est une solution possible. Cette approche ainsi que l'algorithme sont détaillé dans [33].

La seconde partie de l'algorithme effectue un parcours en profondeur de l'arbre afin de trouver une solution optimale. Lors du parcours de l'arbre, l'algorithme essaye de fusionner plusieurs canaux abstrait sur une même instance d'une unité de communication déjà allouée. Une fusion est possible lorsque plusieurs arcs d'un chemin désignent la même unité de communication. Si la fusion n'est pas possible ou viole les contraintes, une autre instance de l'unité de communication est créé. La technique du *branch and bound* peut être utilisée afin de limiter le nombre de parcours en profondeur de l'arbre effectués.

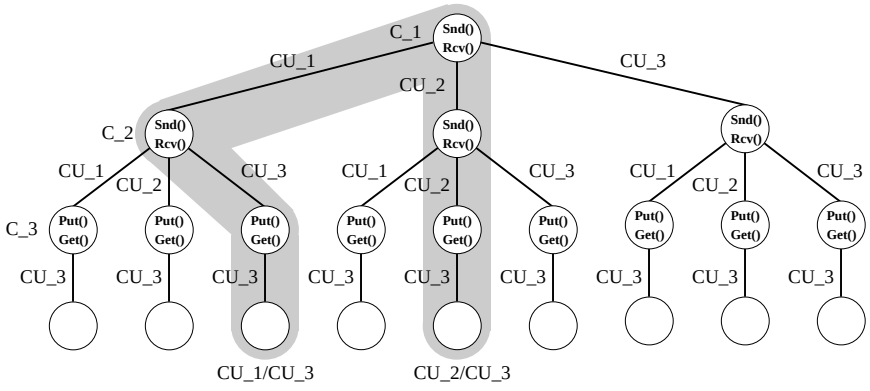


FIG. 4.17: Arbre de décision pour l'allocation

La figure 4.17 représente un arbre de décision correspondant au graphe de la figure 4.15. Les deux chemin en grisés représentent le deux solution représentées en figure 4.16.

4.4 PRIMITIVES DE SYNTHÈSE DE LA COMMUNICATION INTERACTIVE

Notre approche interactive de la synthèse système repose sur l'utilisation de primitives permettant de transformer par étapes successives un système, pour arriver à une réalisation [95]. Dans la section 4.3 nous avons mis en évidence trois opérations nécessaires à la synthèse de la communication :

- l'allocation d'unité de communication doit permettre de choisir dans la bibliothèque un canal capable d'exécuter les primitives de communication requises.
- la fusion de canal doit permettre de réunir deux canaux abstraits pour les allouer sur une même unité de communication.
- la synthèse d'interface doit générer les interfaces et interconnexions correspondantes à l'unité de communication choisie.

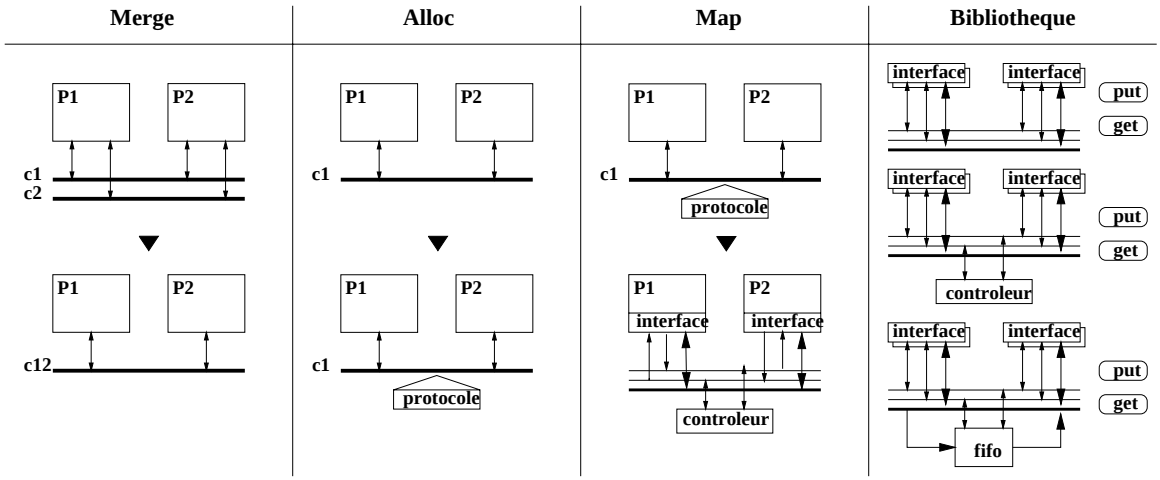


FIG. 4.18: Primitives pour la synthèse de la communication

Notre approche interactive de la synthèse de la communication reposera sur trois primitives (*alloc*, *merge*, *map*) permettant de réaliser les étapes d'allocation, de synthèse de réseau d'interconnexion et de synthèse d'interface. Les primitives *alloc* et *merge* permettent de réaliser la sélection de protocole et la synthèse de réseau d'interconnexion et la primitive *map* la synthèse d'interface. La figure 4.18 montre l'organisation générale de la synthèse de la communication interactive à l'aide des primitives *merge*, *alloc*, et *map*.

4.4.1 Merge

La primitive *merge* permet de fusionner plusieurs canaux abstraits en un seul canal abstrait (figure 4.19) afin de les allouer sur la même unité de communication. Toutes les communications fusionnées restent indépendantes mais toutes les primitives de communication sont offertes par un seul canal abstrait. Pour cela un identificateur de communication logique est généré et un paramètre est rajouté dans chaque primitive du canal abstrait pour identifier la communication. Toutes les primitives seront exécutées avec le même protocole par une seule unité de communication. La primitive *merge* permet de réduire le nombre de canaux de communication nécessaires en les partageant entre différentes communications. Elle permet de réduire les interfaces et les interconnexions et de partager les ressources entre plusieurs communications abstraites. La figure 4.19 représente le système de la figure 4.2 après un *merge* des canaux *N_ sar_ receiver* et *N_ sar_ sender*.

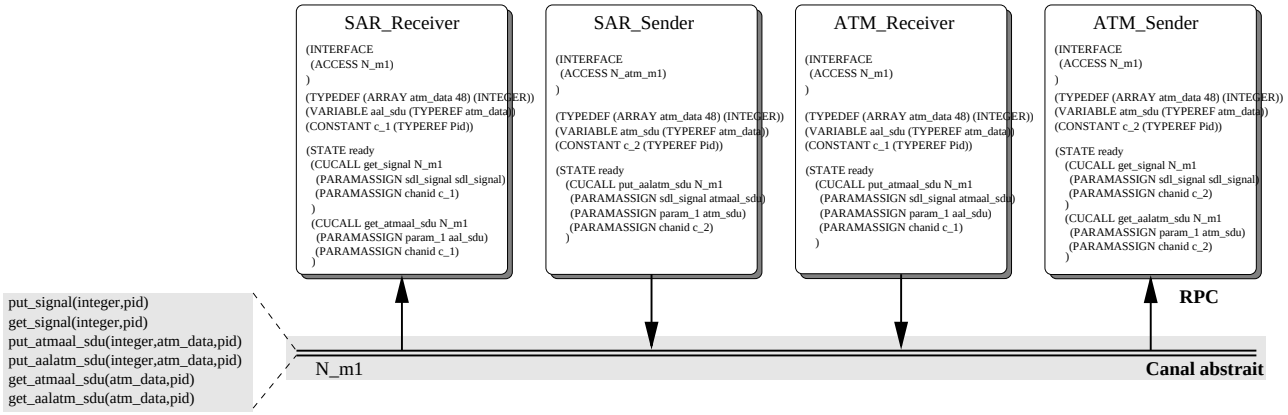


FIG. 4.19: Fusion de canaux abstraits

4.4.2 Alloc

La primitive *alloc* permet de lier un ou plusieurs canaux abstraits à un canal de la bibliothèque (étape de sélection de protocole).

La primitive d'allocation met en correspondance les primitives du canal abstrait avec celles offertes par l'unité de communication. Elle vérifie que l'unité de communication selectionnée dans la bibliothèque est capable d'exécuter la communication requise par le canal abstrait, c'est à dire qu'à chaque primitive offerte par le canal abstrait correspond une primitive de l'unité de communication. Pour cela elle effectue la comparaison des prototypes des primitives offertes par l'unité de communication avec ceux des primitives offertes par le canal abstrait. Il s'agit de mettre en correspondance les paramètres formels des procédures décrites dans la bibliothèque et les paramètres effectifs échangés par les processus. A plusieurs primitives offertes par le canal abstrait peuvent correspondre une même primitive de l'unité de communication allouée. Cela peut être le cas lorsque l'on a fusionné des canaux abstrait. Cette opération est interactive et peut requérir l'intervention du concepteur lorsque la correspondance n'est pas directe. L'allocation réalise deux opérations :

– la mise en correspondance des prototypes des primitives de communication offertes par les canaux abstraits avec ceux des primitives (*method*) de l'unité de communication choisie.

– la génération des identificateurs de communication logique. Si N canaux abstraits sont implémentés par une unité de communication $\log_2 N$ signaux seront nécessaires pour identifier la communication logique. Chaque communication abstraite fusionnée se voit assigner un identificateur unique permettant au canal de l'identifier.

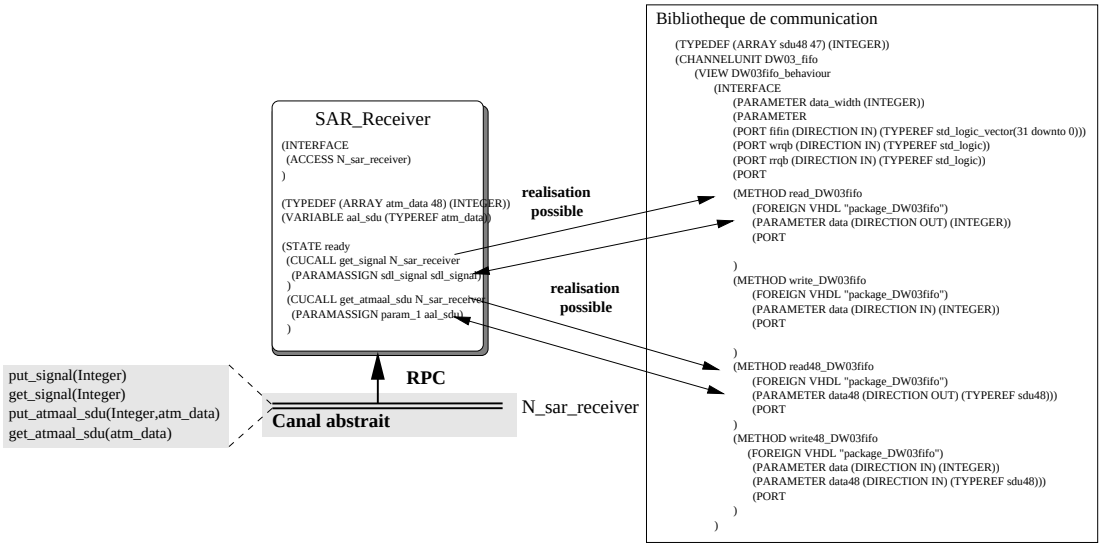


FIG. 4.20: Allocation d'une unité de communication pour un canal abstrait

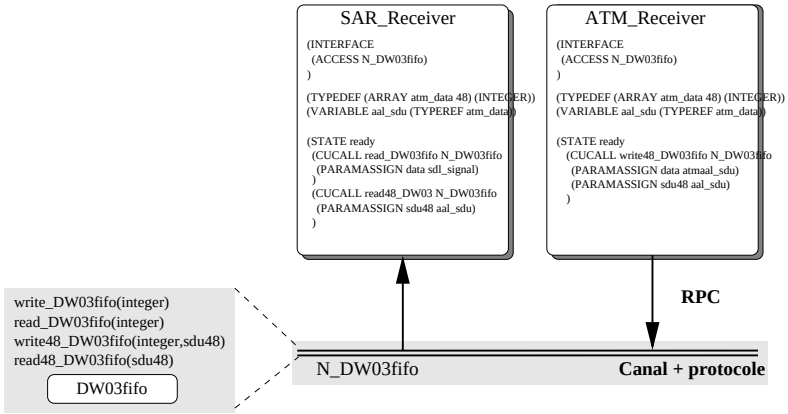


FIG. 4.21: Système communicant à travers des protocoles

Lorsque l'on alloue une unité de communication pour plusieurs canaux abstraits une opération de fusion des canaux abstraits est implicitement effectuée par la primitive *alloc*. La primitive *alloc*

utilise la bibliothèque de communication où sont décrit le prototype des primitives de communication et le protocole offert par chaque unité de communication (figure 4.8).

La figure 4.21 montre le résultat de l'allocation de l'unité *DW03fifo* pour le canal abstrait *N_ sar_ receiver* de la figure 4.20. Après l'allocation, les processus communiquent à l'aide de primitives *write_ DW03fifo()*, *read_ DW03fifo()*, *write48_ DW03fifo()*, *read48_ DW03fifo()* offerte par l'unité de communication avec un protocole *first-in-first-out*. Ces primitives réalisent les communications abstraites spécifiées respectivement par les primitives *put_ signal()*, *get_ signal()*, *put_ aalatm_ sdu()* et *get_ aalatm_ sdu()*.

4.4.3 Map

La primitive *map* transforme un ensemble de processus communicant à travers des canaux en un ensemble de processeur interconnectés par des signaux (étape de synthèse d'interface). La synthèse d'interface réalise quatre opérations :

- la génération des interfaces pour chaque processus accédant au canal. Les ports utilisés par les primitives de communication ainsi que l'interface du contrôleur de communication sont décrits dans la bibliothèque de communication.
- l'instanciation d'éventuels contrôleurs de communication, buffers.
- la génération de la netlist d'interconnexion reliant les interfaces entre elles et au contrôleur de communication.
- l'expansion en ligne des primitives de communication dans chaque processus. Le corps des primitives et du contrôleur sont décrits dans la bibliothèque C/VHDL.

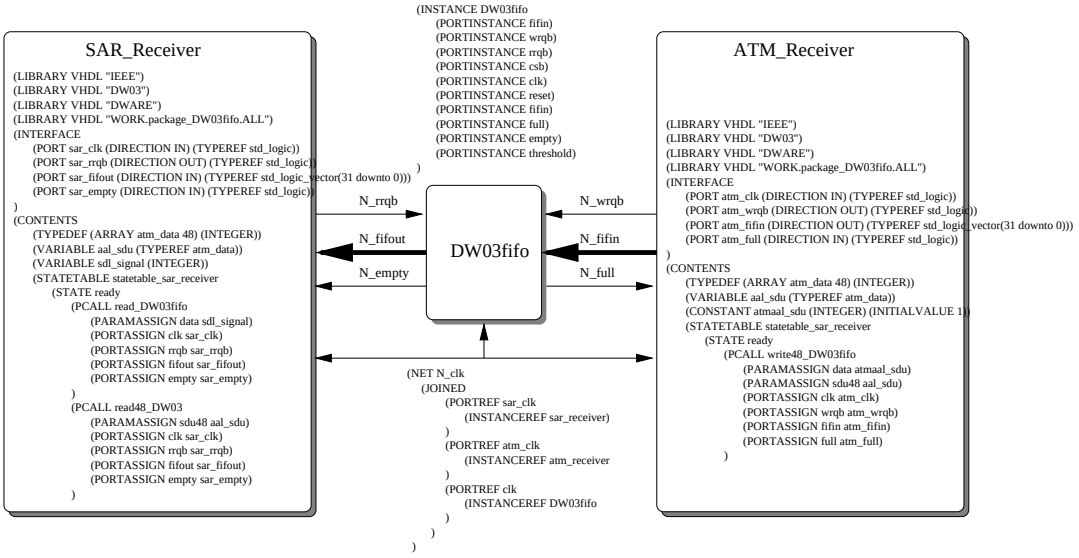


FIG. 4.22: Système interconnecté communicant à travers des bus et des signaux

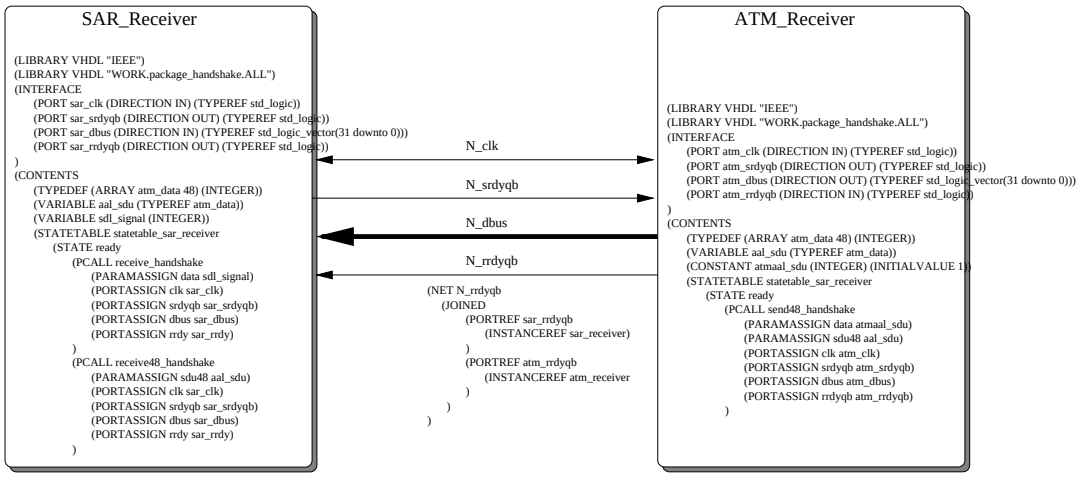


FIG. 4.23: Alternative de sélection de protocole/génération d'interface

Conceptuellement les procédures de communication sont extraites du canal pour être incluses dans les processus utilisant le canal. Les procédures de communication deviennent locales et les appels de procédure à distance (*cucall*) sont replacés par des appels locaux (*pcall*). Le corps des primitives de communication et du contrôleur sont décrits dans la bibliothèque C/VHDL (figure 4.5 et 4.6). Ces fichiers sont inclus dans le fichier source au moment de la compilation du code C et VHDL grâce aux directives *include* du C et *use library* de VHDL.

La figure 4.22 représente le système de la figure 4.20 après l'application de la primitive *map*. Seuls les ports necessaires à chaque primitive de communication sont inclus dans l'interface de chaque processus. Ceux-ci sont connectés soit aux ports d'autres processus (figure 4.23) soit au port du contrôleur de communication (figure 4.22).

Le Map des paramètres *Pid*

Les paramètres de type *Pid* ont un statut particulier dans les primitives de communication. Ils peuvent être soit des paramètres de communication (transférés au processus destinataire) soit des paramètres réalisation (utilisés par le canal) (figure 4.4).

Un paramètre de type *Pid* peut être envoyé par un processus qui est une instance multiple ou qui fait un appel de procédure à distance pour s'identifier et prend la valeur de *self*. Le processus destinataire récuperera cette valeur dans le paramètre *sender* et l'utilisera pour répondre au processus. Dans ce cas ce paramètre fait partie des données tranférées par le canal et appartient à la partie communication. Un processus peut aussi transmettre un paramètre de type *Pid* à un canal pour identifier le canal du processus destinataire. Cela peut être le cas lorsqu'un processus envoie un message à une instance multiple (paramètre *to*), lorsqu'un serveur de procédures distante renvoie le résultat de l'exécution d'un appel de procédure à distance ou lorsque plusieurs canaux abstraits ont été fusionnés. Dans ce cas ce paramètre n'est pas transmis mais est utilisé par le canal pour identifier le sous-canal destinataire (figure 5.4). Il fait partie de la partie réalisation (figure 4.4).

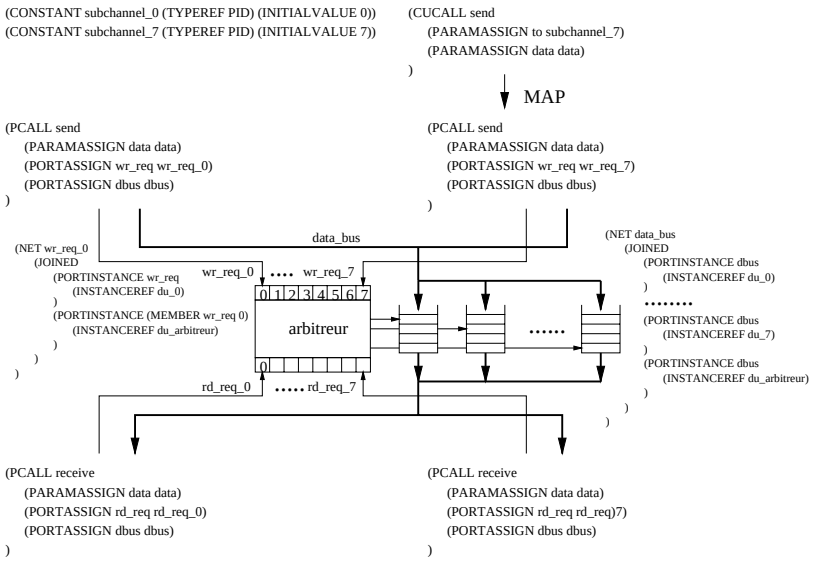


FIG. 4.24: Réalisation cablée du paramètre *to Pid*

La figure 4.24 représente le *map* des paramètres *Pid* losrque le destinataire peut être identifié statiquement. Cela est le cas lorsque plusieurs canaux abstrait ont été fusionnés. Le paramètre *Pid* a été supprimé des paramètres de la procédure et remplacé par une réalisation cablée.

La figure 4.25 représente une réalisation possible d'un canal *RPC* et du mécanisme d'identification du processus appelant.

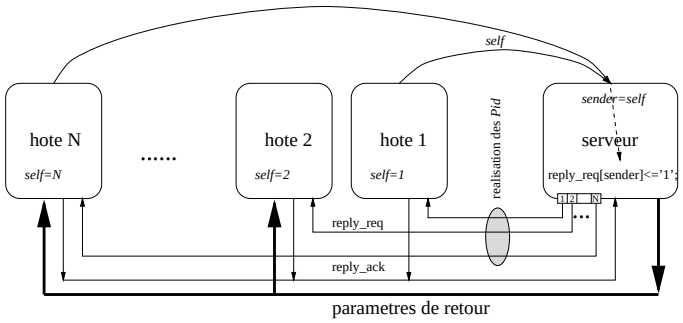


FIG. 4.25: Réalisation du canal RPC

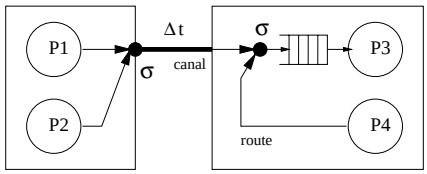
Considérations sur la réalisation des canaux SDL

En SDL, un signal émis par un processus sera toujours reçu par son destinataire. Aucun signal ne peut être retenu indéfiniment dans le réseau de communication. Si un signal traverse des canaux, il subit un retard aléatoire mais fini. Un signal qui traverse uniquement une route (reliant deux processus d'un même bloc) ne subit aucun délai. De même lorsque plusieurs signaux arrivent au

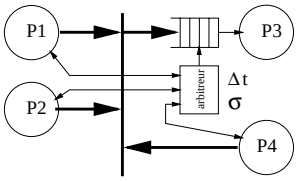
même instant dans une file d'attente, leur ordre d'écriture dans la file est aléatoire, mais tous sont assurés d'y accéder. En d'autres termes, le réseau de communication SDL se comporte équitablement pour tous les signaux.

Lors de la réalisation des files d'attentes SDL en VHDL, il conviendra de s'assurer que le mécanisme de résolution des conflits d'accès au canal est équitable et qu'aucun processus ne va se voir refuser l'accès indéfiniment (famine). Par contre l'accès à un canal pourra être retardé (en cas de conflit) si ce délai reste fini. Les différences entre le modèle de communication SDL et VHDL sont représentées en figure 4.26. Les signaux subissent des retards (Δt) ou des ordonnancements dans les files d'attentes ou canaux SDL (σ). Ceux ci sont aléatoires en SDL, et deviennent déterministes en VHDL.

En SDL le délai de la communication est introduit par les canaux et l'ordonnancement des signaux par les canaux et les files d'attente. Dans la réalisation VHDL, le délai et l'ordonnacement sont dû à l'arbitreur. On pourra remarquer le délai, nul, introduit par les routes SDL ne l'est plus en VHDL. L'hyptothèse du zéro délai utilisée dans les langages de spécifications, et valable, dans une certaine mesure pour le logiciel, ne l'est plus dans le cas d'une réalisation matérielle.



4.26.1: Non déterminisme dans le modèle de communication SDL



4.26.2: Réalisation VHDL

FIG. 4.26: Réalisation de la communication SDL en VHDL

4.5 CONCLUSION

Nous avons présenté une approche dans laquelle la synthèse de la communication est abordée comme un problème d'allocation. Cette approche permet une large exploration de l'espace des solutions grâce à une bibliothèque de communication. Ce problème peut être résolu par la plupart des algorithmes utilisés en synthèse de haut niveau. Notre approche interactive est basée sur trois primitives *merge*, *alloc*, *map* permettant de réaliser les étapes d'allocation, de synthèse de réseau d'interconnexion et de synthèse d'interface. Le point clé de cette approche est l'utilisation d'un modèle de communication abstrait et général permettant la représentation pour la synthèse de la plupart des modèles de communication. La séparation entre unités de calcul et de communication permet la réutilisation de modèles de communication existants. Une bibliothèque de communication offre un large choix de modèles de communication et permet au concepteur de sélectionner un protocole adapté à son application. Dans la mesure où aucune restriction n'est imposée au modèle de communication, cette approche peut être appliquée à une large gamme d'applications.

Chapitre 5

GÉNÉRATION DE CODE C-VHDL À PARTIR DE SPÉCIFICATIONS SDL

Dans ce chapitre nous allons présenter une approche de génération de code C et VHDL à partir de spécifications systèmes en SDL. Nous étudierons les principales différences entre les langages SDL et VHDL qui pourraient poser des problèmes de traduction. Nous nous attacherons à mettre en correspondance les concepts SDL et les concepts SOLAR et à montrer comment l'utilisation d'une forme intermédiaire permet d'obtenir une synthèse efficace. La principale difficulté de traduction réside dans l'abstraction de la communication au niveau système.

5.1 INTRODUCTION

L'augmentation de complexité des circuits a entrainé le développement de nouvelles méthodes adaptées à ces systèmes complexes. Une façon de gérer cette complexité est d'accroître le niveau d'abstraction des spécifications en utilisant des langages de description système. D'un autre coté, l'élévation du niveau d'abstraction des spécifications entraine une augmentation du fossé entre les concepts utilisés pour les spécifications au niveau système et ceux utilisés pour la synthèse de matériel. Bien que les langages systèmes soient adaptés à la spécification et à la validation de systèmes complexes et temps réel, les concepts qu'ils manipulent ne sont pas facilement représentables dans les langages de description matériels. Il devient donc nécessaire de définir de nouvelles méthodes de synthèse système permettant d'obtenir une synthèse efficace à partir de spécifications système.

Les langages de spécifications systèmes offrent des concepts et des méthodes bien adaptés à la description de systèmes complexes. Il offrent des méthodes formelles permettant de vérifier et de valider le comportement d'un système. Comme une spécification système sert de base pour la réalisation, elle doit faire abstraction des détails d'implémentation afin de retarder autant que possible les choix concernant la réalisation pour n'exclure aucune alternative possible. Plusieurs étapes de raffinement sont alors nécessaires pour obtenir une implémentation. Chaque étape choisit la réalisation d'un ensemble de détails réduisant ainsi le fossé entre spécification et réalisation. De nombreux concepts offerts par les langages de spécifications systèmes (machines d'états finies, communication de haut niveau, exceptions) ne sont pas facilement représentés dans les langages de description matériel et demandent une description verbeuse. Une traduction automatique est donc une méthode efficace et évitant les erreurs pour obtenir une implémentation à partir d'une spécification système.

5.1.1 Objectifs

Lors de la traduction d'une spécification système pour l'implémentation, le principal problème concerne l'implémentation de la communication. Les modèles de communication système sont abstraits et généraux, une traduction directe de ces modèles aboutit inévitablement à une solution irréalisable. Pour obtenir une solution efficace différents protocoles de communication peuvent être nécessaires de même que différentes topologies d'interconnections. Le protocole de communication et la topologie d'interconnection peuvent grandement influencer les performances d'un système et requièrent donc une grande exploration de l'espace des solutions.

Dans ce chapitre nous présentons une nouvelle approche de génération de code C et VHDL qui résoud les problèmes de transposition des concepts systèmes sur les concepts matériels. Les principaux objectifs de notre méthode sont :

1. d'avoir une génération automatique de code C et VHDL à partir d'une spécification système en SDL. Les concepts qui n'ont pas de sémantique matérielle efficace ne seront pas supportés. Le code produit devra être acceptable par les outils de synthèse et de simulation (code VHDL) ainsi que les compilateurs standard (code C).
2. d'avoir une réalisation efficace des modèles de communications systèmes.
3. de pouvoir choisir entre plusieurs protocoles de communication dans une bibliothèque.

4. de pouvoir modéliser la communication indépendament du comportement. La spécification du système devrait être indépendante de la spécification de la communication afin de permettre des changements dans le modèle de communication sans modification de la spécification du système.

En plus cette approche devra être compatible avec les autres outils de synthèse système tel que le partitionnement logiciel/matériel.

5.1.2 Travaux antérieurs

De nombreux travaux sur la synthèse système ont rapportés dans la littérature [23], [29], [47], [54], [61], [106] mais peu d'entre eux traitent de la synthèse système à partir de spécifications SDL [19], [57], [116]. Bien que SDL [9], [48] soit largement utilisé dans le domaine du logiciel et des télécommunications [16], il ne l'est pas parmi les concepteurs de matériel [117].

De nombreuses approches ont étendu des langages mono-flôt pour supporter des concepts matériels et de communication. La plupart d'entre eux ont utilisé le langage C tel que HardwareC [61] ou C^x [47]. [149] utilise une spécification en langage C++ et un ensemble de classes prédéfinies encapsulant les communications et les processeurs. D'autres approches [27], [87], [100] partent d'une spécification pré-partitionnée dont les parties logicielles sont décrite en C/C++ et les parties matérielles en VHDL/Verilog. Une autre approche est de créer un nouveau langage de spécifications systèmes tel que SpecChart [102], [137], [138].

D'autres approches utilisent des langages de spécifications systèmes synchrones [10], [67] tels que ESTEREL [14], [3], [20], [30], SIGNAL [64], ALPHA, LUSTRE [65] ou StateChart [24], [25],[40], [68]. StateChart est un langage orienté flôt de contrôle et est transposé sur des automates d'états finis. Dans [101], une approche de génération d'architectures parallèles régulières à partir de ALPHA est présentée. Dans [8] une approche de codesign utilisant SIGNAL pour décrire les parties contrôle et ALPHA pour les parties de calcul est présentée. Le domaine visé est celui des systèmes temps réel réactifs.

Peu d'approches ont utilisées des langages de spécifications de systèmes distribués tel que SDL [19], [57], [94], LOTOS [28], [39], [49], [88] ou ESTELLE [148]. Cela est principalement dû aux fossé existant entre les concepts manipulés par ces langages et ceux des langages de description matériel. Ces langages offrent des concepts tel que le parallélisme ou un modèle de communication abstrait. Ce fossé peut être réduit progressivement par l'utilisation de raffinements successifs [12], [89]. Malgré tout, l'essentiel des travaux se sont concentrés sur des approches de traduction directe. Seul [53] [58] utilise SpecChart comme forme intermédiaire, bien que ce langage ne permette pas de modéliser les protocoles de communication complexe tel que le passage de messages. Le raffinement de la spécification est nécessaire car le niveau d'abstraction des spécifications système est souvent trop élevé pour être exécuté directement de façon efficace. Le but de cette phase est de produire un système exécutable qui satisfasse aux spécifications initiales et offre des performances acceptables. Cette phase de transformations peut être vue comme une étape de compilation interactive guidée par le concepteur. L'interaction humaine est nécessaire car de tels algorithmes sont en dehors des possibilités des compilateurs actuels [89].

Dans le domaine des langages de spécification de systèmes distribués, [39] présente la génération

de code VHDL à partir de spécifications LOTOS. En LOTOS [18], [93], [132], la communication et la synchronisation entre processus est effectuée à travers des canaux appelés portes. Le modèle de communication est le *rendez-vous*. Il n'y a pas de file de messages entre les processus. Lorsqu'un processus est prêt à communiquer, il offre un ensemble de données à une porte. La communication aura lieu lorsque tous les processus qui désirent communiquer seront prêts. Les trois schéma de *rendez-vous* disponibles en LOTOS sont détaillés dans le paragraphe 3.5.2. En LOTOS, les différents processus s'executent en (pseudo) parallélisme et se synchronisent à travers des portes. Tous les processus impliqués dans une communication sont bloqués jusqu'à ce que la communication ait eu lieu. Le modèle de communication de LOTOS implique un mécanisme de synchronisation complexe. Il est difficile à implémenter en matériel et des restrictions sont faites pour la synthèse. Dans [39], les processus LOTOS sont traduits en automates d'états finis VHDL et chaque schéma de communication a une seule implémentation basée sur un module de synchronisation prédéfini extrait d'une bibliothèque VHDL. Öberg [110] présente une approche de synthèse matérielle de protocoles de communication orientés données à partir d'un langage proche de LOTOS.

ESTELLE possède un schéma de communication puissant. Chaque processus peut disposer de plusieurs files d'attentes. Un processus envoie un message à une file déterminée d'un autre processus à travers des liens de communication. Chaque lien peut convoyer plusieurs types de messages. En ESTELLE le comportement est specifié à l'aide de machine d'états finis qui peuvent être non déterministes. Dans un état de l'automate, la transition à exécuter est choisie au hasard parmi toutes les transitions tirables. Dans l'approche décrite dans [148], chaque message est traduit comme un type VHDL et une déclaration de file d'attente est générée pour chaque file qui stocke des messages de type différent. Cette approche supporte aussi les aspects dynamiques de la communication ESTELLE en utilisant des élements spécifiques de bibliothèque.

[19], [57] et [116] génèrent du code VHDL à partir de spécifications SDL. Dans l'approche proposée dans [116], des restrictions sont faites concernant le modèle de communication. La traduction en VHDL n'associe pas de file d'attente à chaque processus dans laquelle les messages arrivants sont stockés (voir section 3.6). Chaque signal SDL est traduit comme un signal VHDL changeant le modèle de communication du passage de message au signal (voir section 5.2). Chaque signal peut avoir au plus un paramètre. Bien que cette approche facilite la synthèse, seulement une implémentation du modèle de communication SDL est possible. La traduction ne respecte le modèle d'exécution SDL que lorsque chaque état de l'automate d'états finis n'a qu'une transition. Dans [19], l'algorithme de traduction simplifie le modèle de communication pour obtenir du VHDL synthétisable. Trois protocoles de communication sont disponibles dans la bibliothèque :

1. une file d'attente à une position partagée par tous les signaux.
2. une file d'attente à une position pour chaque signal. Dans ce cas un processus peut avoir plusieurs file d'attente.
3. une file d'attente à N positions partagée par tous les signaux.

Chacun des trois protocoles peut être utilisé et de nouveaux protocoles peuvent être ajoutés dans la bibliothèque. Suivant la même approche que Pulkkinen [116], il assigne un signal VHDL à chaque signal SDL. Lorsqu'il utilise le second protocole, l'algorithme requiert l'assistance du concepteur pour assigner une priorité à chaque signal. Cette approche ne respecte le modèle d'exécution SDL que dans la mesure où les signaux sont consommés dans l'ordre des priorités et non plus dans leur

ordre d'arrivée dans la file. Dans [57], Glunz présente une approche plus puissante de traduction SDL vers VHDL. Le modèle de communication peut être changé à l'aide d'une bibliothèque de protocole mais cette approche ne supporte qu'un sous-ensemble de VHDL. Les transitions dans les procédures et les labels ne sont pas supportés et les états avec différentes transitions déclenchées par des signaux ayant un nombre différent de paramètres ne semble pas être supportés.

A notre connaissance aucune des précédentes approches résoud le principal problème de traduction relatif à la communication. La contribution majeure de ce chapitre est de présenter une approche où la synthèse est réalisée à l'aide d'une forme intermédiaire qui permet un ensemble de transformations et d'étapes de raffinement sur le système. Cela permet de supporter un large sous-ensemble de SDL et de résoudre le problème de l'abstraction des communications.

Dans les paragraphes suivants, nous présentons notre approche de la synthèse système à partir de spécifications SDL. Dans le paragraphe 5.2 nous étudierons les principales différences entre les concepts SDL et VHDL pouvant poser des problèmes de traduction. La section 5.3 détaille la traduction de spécifications SDL et donne le sous-ensemble supporté. Ensuite nous présentons quelques résultats avant de conclure.

5.2 DISTANCE ENTRE LES CONCEPTS SDL ET VHDL

Dans cette section nous étudions les principales différences entre les concepts SDL et VHDL qui pourraient générer des problèmes de traduction.

5.2.1 Modèle d'exécution

Chaque langage repose sur un modèle d'exécution qui spécifie le flôt de données ou le flôt de contrôle et la synchronisation [67]. SDL et VHDL sont tous deux orientés flôt de contrôle.

En SDL, chaque processus évolue independamment en fonction des signaux qu'il reçoit. l'exécution de processus parallèles peut être non déterministe. Par exemple lorsqu'un processus reçoit au même instant deux signaux provenants de deux processus leur ordre dans la file d'attente ne peut pas être prédit. Dans un système SDL, une seule transition est exécutée à la fois, ce qui signifie qu'un seul automate du système n'est actif à un instant donné. Lorsque plusieurs transitions correspondant à plusieurs processus peuvent être exécutées alors un choix non déterministe d'une transtition parmis toutes celles exécutables est effectué.

En VHDL, l'exécution des processus parallèles est ordonnancée par le simulateur. Chaque $\triangle$ cycle tous les processus sont évalués. Les affectations de signaux sont instantannées et déterministes dans la mesure où les conflits sont résolus à l'aide de fonctions de résolution. En VHDL93 le non déterminisme peut être introduit à l'aide des variables partagées.

Lors du passage d'une spécification SDL à une implémentation en VHDL, les signaux doivent être ordonnés dans la file d'attente. SDL ne fait pas d'hypothèse sur l'ordre des signaux dans la file et tout ordonnacement sera valide.

5.2.2 Abstraction de la communication

La communication inter-processus repose essentiellement sur deux modèles : le passage de messages et la mémoire partagée [1]. SDL utilise le premier modèle alors que VHDL utilise le dernier. Dans le premier cas, les processus communiquent à travers des messages envoyés avec un protocole spécifique. Dans le modèle à mémoire partagée les processus communiquent en échangeant des données à travers une mémoire commune. En VHDL la communication peut aussi s'effectuer à l'aide de signaux.

La communication SDL encapsule totalement le protocole de communication. La modélisation de la communication SDL en VHDL requiert une expansion du protocole de communication et de la file d'attente qui sont implicites en SDL.

5.2.3 Description du comportement

Le modéle de comportement spécifie comment décrire le comportement d'un système.

En SDL, le constructeur de base est l'automate d'états finis. SDL offre aussi des possibilités d'instanciation et d'adressage dynamique de processus.

En VHDL, le nombre d'instances de composant est fixe et le comportement est spécifié à l'aide de processus sous forme algorithmique. De plus VHDL n'offre pas de facilité de contrôle tel que *les exceptions*, *le reset*. La représentation de ces concepts en VHDL demande une implémentation verbeuse [102].

5.2.4 Le temps

En SDL, une référence de temps globale est accessible à travers un serveur de temps global. Ce serveur contrôle un ensemble de timers offrant des primitives *set* et *reset*. Les timers peuvent être utilisés pour synchroniser différents processus sur un événement extérieur. En SDL, les differents timers sont indépendant et ne sont pas synchronisés par une horloge globale. Le temps global est accessible à chaque processus à travers la variable *now*.

En VHDL, la référence de temps accessible est le temps global de la simulation. Les processus peuvent utiliser ce temps pour la synchronisation et l'ordonancement des affectations de signaux (clauses *for* et *after*). Le temps de simulation global est accessible à travers la variable *now* bien que celle-ci ne soit pas supportée par la synthèse. La traduction des timers SDL implique des choix concernant leur réalisation.

5.2.5 Les type de données

SDL offre la possibilité de définir des types de données abstraits et des opérateurs agissant sur ces types. Les opérateurs peuvent être définis sous forme algorithmique en C ou en SDL. VHDL n'offre que des types de données simples mais les types de données abstraits SDL peuvent être traduit en VHDL sans modification du programme original.

Comme il est montré dans le tableau 5.1, SDL supporte la plupart des concepts système tels qu'il sont définis dans [138]. Nous avons aussi inclus la synchronisation et la communication inter-processus qui sont importantes dans les systèmes distribués hétérogènes. la principale restriction

language	hierarchie	parallélisme	comportement	temps
SDL	structurelle	un niveau de processus	automate d'états finis	timer
VHDL	structurelle	instruction concurrentes, processus	algorithme	clause AFTER

language	exception	synchronisation	communication
SDL	oui	signaux globaux	passage de message, protocole FIFO
VHDL	non	instruction WAIT événement commun	mémoire partagée signaux

TAB. 5.1: Concepts VHDL et SDL supportés

vient de la hiérarchie comportementale qui n'est pas supportée et d'un seul modèle de communication disponible. VHDL ne supporte pas directement tous ces concepts (machine d'états finis, communication de haut niveau, exception) mais ils peuvent être implémentés indirectement. Une comparaison plus complète de plusieurs langages de spécification au niveau système peut être trouvée dans [102].

5.3 SYNTHÈSE DE LOGICIEL/MATÉRIEL À PARTIR DE SPÉCIFICATIONS SDL

Cette section détaille les étapes de raffinement et les modèles utilisés par COSMOS pour transformer un modèle SDL en VHDL. Cette approche supporte un large sous-ensemble de SDL. Seul les concepts qui n'ont pas d'équivalent en matériel sont exclus de ce sous-ensemble.

5.3.1 Restriction pour la synthèse de matériel

SDL offre un modèle de communication abstrait et général qui n'est pas adapté à la synthèse de matériel. Cela est principalement dû au fait que les signaux peuvent être routés à travers les canaux. En d'autres termes, la destination d'un signal peut être déterminée dynamiquement par l'adresse du destinataire. En SDL, l'adressage dynamique est principalement destiné à être utilisé avec la création dynamique de processus. Cet aspect est très orienté logiciel et est difficile à conceptualiser pour le matériel. En conséquence, nous pouvons restreindre le modèle de communication SDL pour la synthèse de matériel sans perdre toutefois sa généralité et son abstraction. Les restrictions apportées à SDL concerneront les aspects dynamiques tels que la création de processus et l'adressage dynamique.

Les aspects non supportés incluent : la création dynamique de processus, le type *Pid* (supporté avec les instances multiples de processus), les sous-structure de canaux, le non déterminisme (*input any*, *input none*, *decision any*).

5.3.2 Sous-ensemble supporté

Un large sous-ensemble de SDL est supporté :

- *system*, *block*, *process*, instances multiples.
- *state*, *state **, *state *()*, *save*, *save **, signaux continus, condition d'autorisation.
- *input*, *output*, signaux avec paramètres, *task*, *label*, *join*, *nextstate*, *stop*, *decision*, appel de procédure (*call*).
- variables importées et exportées, procédures importées et exportées (appels de procédures à distance).

Certains mécanismes tel que le *save* peuvent être couteux à implémenter en matériel bien qu'ils fassent partie du sous-ensemble supporté.

5.4 CORRESPONDANCE ENTRE LES MODÈLES SDL ET C/VHDL

Dans cette section nous présentons les modèles de traduction des principaux concepts SDL.

5.4.1 Structure

Comme il est mentionné précédemment, les aspects dynamiques de SDL ne sont pas considérés pour la synthèse de matériel. Pour éviter tout problème de routage et afin d'obtenir une communication efficace, nous nous restreindrons au cas où le destinataire d'un processus peut être déterminé statiquement. La structure de communication SDL (routes et canaux) sera alors mise à plat. Un signal émis par un processus et qui traverse un ensemble de canaux doit avoir un récepteur unique parmi les processus connectés à ces canaux. Dans ce cas, les canaux ne font qu'acheminer les signaux d'une frontière d'un bloc à l'autre. Aucune décision de routage n'a besoin d'être prise dans la mesure où il n'y a qu'un chemin possible pour un signal à travers un ensemble de canaux. Les canaux et les routes ne seront donc pas représentés dans le système final. Un processus qui émet un signal l'écrira directement dans la file d'attente du destinataire sans traverser de nombreux canaux. La mise à plat de la communication élimine le surcoût dû à la traversée de canaux intermédiaires. Au niveau des blocs, on respectera la hiérarchie de la spécification initiale. Chaque bloc sera traduit comme une unité de conception structurelle (*design unit*) et les processus comme des unités de conception comportementales. Les interfaces d'accès (*access*) aux différents canaux abstraits (*net*) et la netlist d'interconnection (*joined*) des canaux seront générés.

5.4.2 Communication

En SDL, chaque processus possède une file d'attente implicite qui stocke les messages. Pour la réalisation, cette file d'attente devra être explicitée. Nous associerons donc un canal abstrait (*net*) à chaque processus (figure 5.2). Ce canal abstrait tiendra le rôle de la file d'attente et offrira les primitives de communication requises. Chaque signal SDL sera traduit comme un couple (identificateur de signal de type entier, paramètres du signal). A chaque signal SDL sera associe deux primitives de communication offertes par le canal abstrait permettant :

- de lire les paramètres d'un signal.
- d'écrire un signal ainsi que ces paramètres.

Chaque canal abstrait offrira en plus une primitive permettant de lire un identificateur de signal. Il est nécessaire de pouvoir lire l'identificateur de signal seul pour déterminer la transition à exécuter et les paramètres du signal à lire. Les instructions *input* seront traduites comme des appels aux primitives de communication lisant dans un canal et *ouput* sera traduit comme un appel aux primitives de communication écrivant dans le canal.

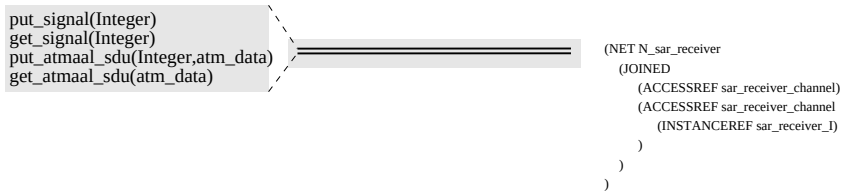


FIG. 5.1: Modélisation de la communication SDL à l'aide de canaux abstraits

La figure 5.1 représente le canal abstrait qui sera associé au processus SDL de la figure 3.13 lors de la traduction dans la forme intermédiaire SOLAR. Les procédures *get_ signal()* et *put_ signal()* permettent de lire et d'écrire un signal dans le canal. La procédure *get_ atmaal_ sdu()* (respectivement *put_ atmaal_ sdu()*) permet de lire les paramètres du signal $ATMAAL\_SDU$ (respectivement d'écrire le signal $ATMAAL\_SDU$ et ses paramètres) dans le canal. Le signal $ATMAAL\_EndSDU$ n'a pas de paramètre. Il est donc lu et écrit dans un canal à l'aide des procédures *get_ signal()* et *put_ signal()*.

Durant l'étape de synthèse de la communication, une unité de communication capable d'exécuter le schéma de communication requis sera sélectionnée de la bibliotèque. Cette approche permet de sélectionner dans la bibliothèque une unité de communication qui offre une implémentation efficace de la communication requise. Bien que SDL n'offre qu'un modèle de communication, différents protocoles peuvent être sélectionnés dans la bibliothèque pour un canal abstrait (paragraphe 4.3.1).

La figure 5.2 représente le bloc SDL de la figure 3.8.2 pour la synthèse. Chaque processus SDL est traduit comme une unité de conception comportementale contenant l'automate et un canal abstrait qui offre les primitives de communication permettant d'émettre et de recevoir des signaux. Chaque processus lira des signaux dans sa propre file d'attente et écrira dans la file des autres processus. Une spécification SDL est donc représentée par un ensemble de processus et de canaux abstraits. Comme il a été mentionné auparavant les routes et les canaux du modèle SDL n'apparaissent pas.

5.4.3 Comportement

Chaque processus SDL sera traduit dans la table d'état SOLAR correspondante.

Transition

Chaque état de l'automate SDL sera reproduit dans un état de l'automate C ou VHDL. Les états *state ** et *state *()* seront expansés en ligne et leurs transitions seront ajoutées aux transitions des états concernés. En SDL, la transition à exécuter est déterminée par le signal en tête de la file. On commencera donc par lire le signal pour effectuer le choix de la transition à exécuter. Une transition SDL sera traduite selon le modèle suivant :

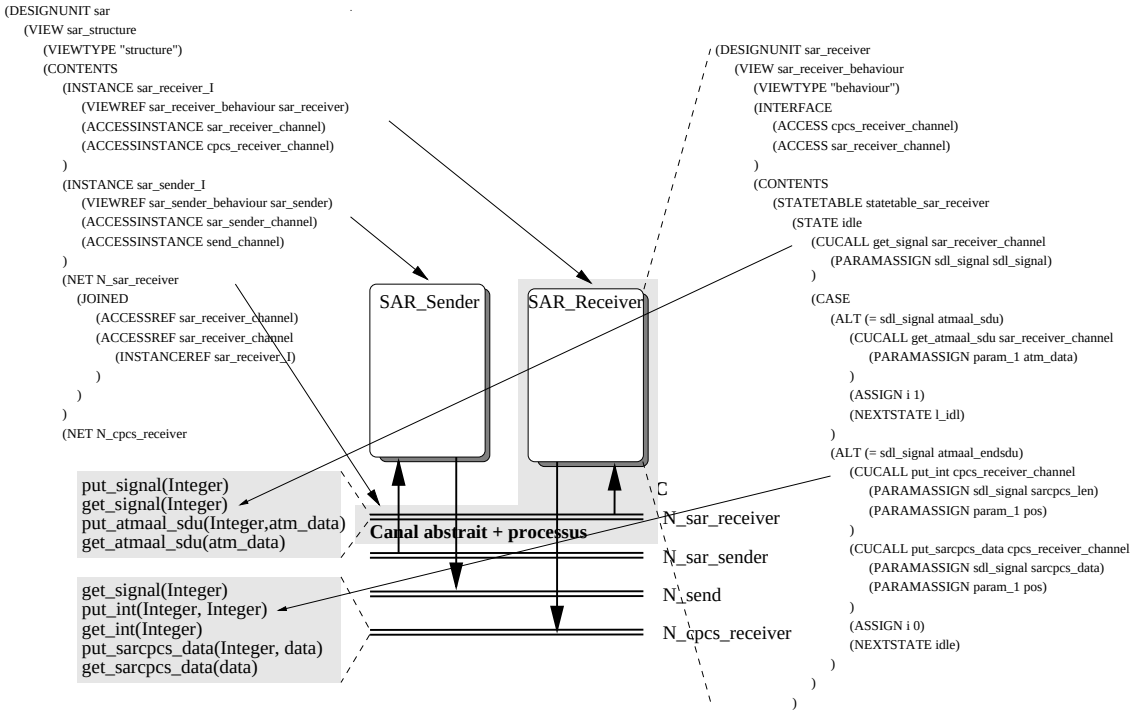


FIG. 5.2: Modélisation de la communication SDL à l'aide de canaux abstraits

– lecture du signal dans le canal.

– choix de la transition à exécuter en fonction du signal reçu.

– lecture des paramètres du signal dans le canal.

– exécution des actions de la transition.

Le modèle d'exécution SDL impose que si dans un état, la réception de certains signaux n'est pas prévue, une transition implicite sera effectuée. Pour respecter le modèle d'exécution SDL, les transitions implicites seront expansées en ligne. Dans chaque état de l'automate la réception de tous les signaux sera prévue. Ces transitions seront composées d'un retour de l'automate dans l'état courant.

Lorsqu'un signal est succeptible d'être sauvé on fera appel à une procédure de communication spéciale pour lire le signal. Celui-ci pourra être sauvegardé en utilisant une procédure de communication appropriée. Dans le cas d'un *save* *, la sauvegarde sera effectuée à la place des transitions implicites.

Dans le cas d'une condition d'autorisation (*input provided*), la transition est effectuée si la condition est vraie sinon le signal est sauvé. On ajoutera donc la condition d'autorisation dans la clause gardant la transition.

Les signaux continus (*provided*) peuvent déclencher des transitions lorsque la file d'attente est vide. Lorsque dans un état, on se trouvera en présence de signaux continus on commencera par tester le canal. Si celui-ci est vide, on évaluera les conditions des signaux continus dans l'ordre des priorités.

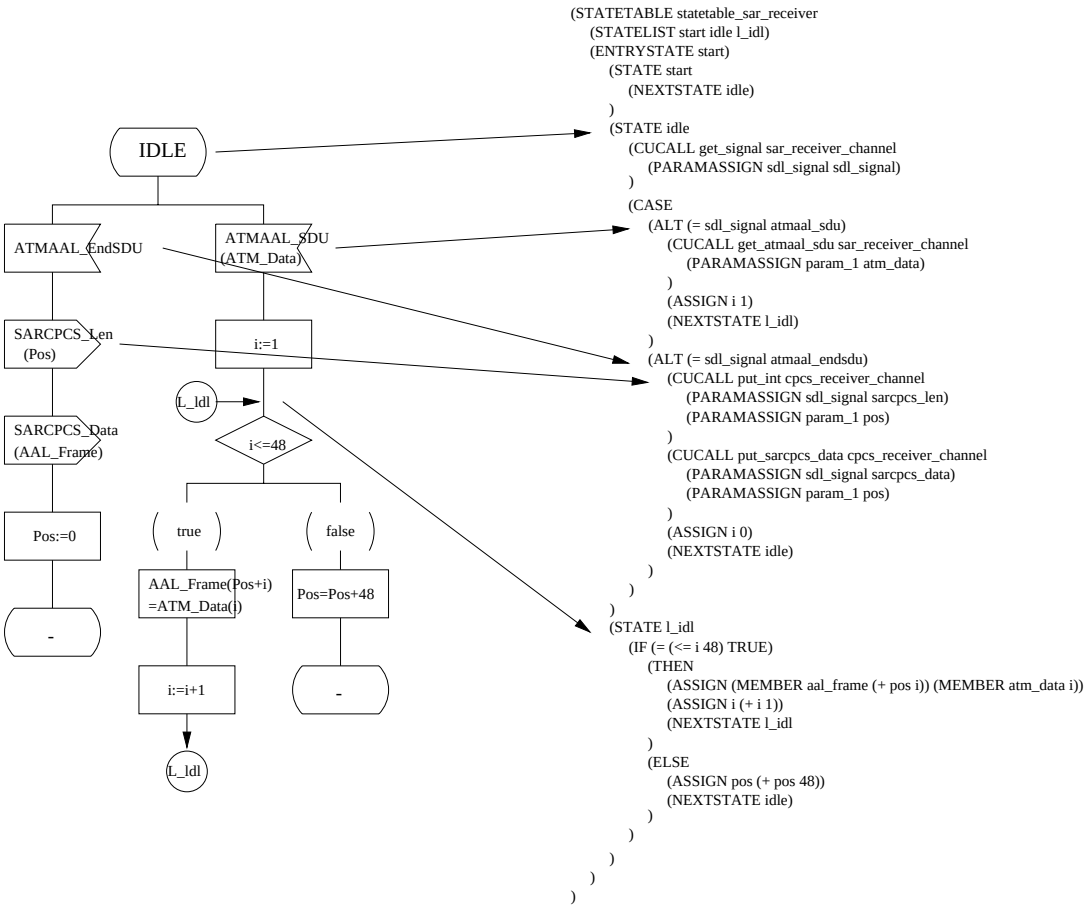


FIG. 5.3: Traduction SDL à SOLAR

Dans le cas de signaux continus, le canal abstrait devra offrir une procédure de communication permettant de tester son état.

Labels

Les labels SDL seront traduits comme des états de l'automate C ou VHDL. A chaque label on associera un nouvel état et une instruction de saut dans cet état. L'instruction SDL *join* sera traduite par une instruction de saut dans cet état.

Paramètres *to* et *sender*

Comme il a été mentionné au paragraphe 5.4.1, les possibilités d'adressage dynamique de SDL doivent être restreintes pour la synthèse. Les possibilités d'adressage dynamique ne sont autorisée que dans le cas des instances multiples de processus et dans le cas des appels de procédure à distance. Dans tous les autres cas, le destinataire d'un processus devra pouvoir être identifié statiquement.

Le mécanisme d'identification des processus, implicite en SDL, sera explicité.

Le constructeur SDL *to* sera traduit comme un paramètre de la procédure de communication. Celui-ci sera utilisé pour identifier le canal du processus destinataire (voir paragraphe 4.4.3). En SDL, chaque message contient l'adresse (*Pid*) du processus émetteur. Celle-ci est implicitement placée dans la variable *sender* à la réception d'un message. Cette adresse n'est nécessaire que lorsque l'émetteur du message ne peut pas être déterminé statiquement. La variable *sender* est alors utilisée par le processus récepteur pour renvoyer un message (*output to sender*). Cela est le cas lorsqu'un processus qui est une instance multiple envoie un message, ou lorsqu'un processus fait un appel de procédure à distance. Dans ces deux cas, un paramètre supplémentaire *sender* sera ajouté aux procédures de communication (figure 5.4).

La figure 5.4 représente une instance multiple de processus (*P1*) et le canal associé. Celui-ci contient trois file d'attente, une par processus. La sélection d'une des trois files d'attente se fait grâce au paramètre *to* de la primitive de communication. Lorsqu'un des processus *P1* envoie un message à *P2*, il s'identifie en affectant le paramètre *sender* à *self* dans la procédure de communication.

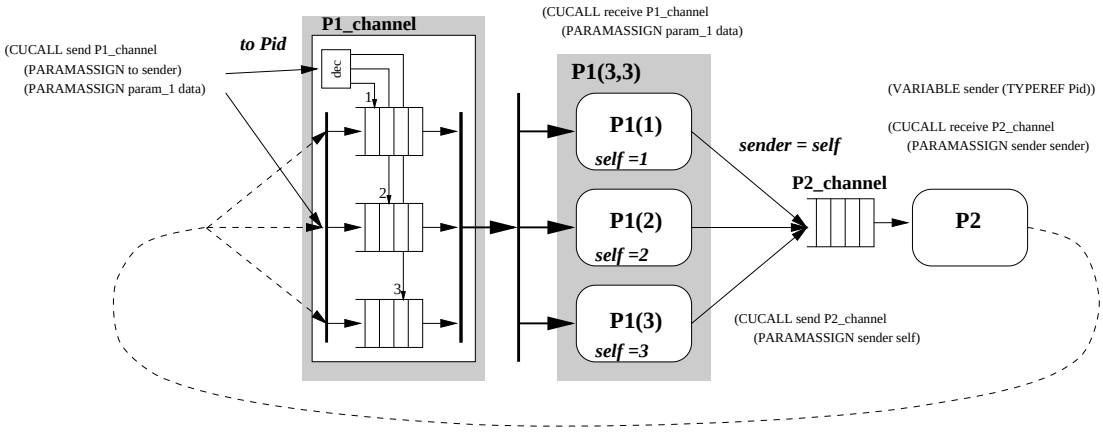


FIG. 5.4: Paramètre *to* et *sender* des primitives de communication

Variables importées et exportées

A chaque variable exportée on associera un canal abstrait qui contiendra la variable partagée (figure 5.5). Ce canal offrira les primitives de communication suivantes :

- export() pour écrire la variable partagée.
- import() pour lire la variable partagée.

La figure 5.3 représente la traduction dans la forme SDL du processus de la figure 3.13. L'automate SDL est traduit dans une table d'états SOLAR correspondante. Dans un état de la table d'états on commence par lire le signal en tête de la file d'attente à l'aide de *get_ signal()* avant de déterminer la transition à exécuter. La procédure *get_ atmaal_ sdu()*) permet de lire les paramètres du signal SDL *ATMAAL_ SDU*.

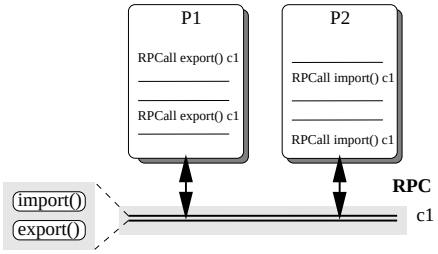


FIG. 5.5: Canal abstrait pour les variables importées et exportées

Procédures importées et exportées

La principale difficultée rencontrée lors de la modélisation des *remote procedure call* SDL en SOLAR est la présence de *save ** dans l'état d'attente (figure 5.6) du processus appelant. Dans cet état, le processus hôte se met en attente du résultat de la procédure et devient aveugle à tous les autres signaux présents dans la file d'attente. Si l'on achemine les paramètres de retour par la file d'attente de l'hôte, cela obligera le canal à disposer d'un mécanisme de *save **, ce qui peut être très lourd à implémenter. De plus, le processus serveur devra connaitre explicitement le processus appelant pour pouvoir aller écrire dans son canal, ce qui alourdi encore le traitement des appels de procédure à distance.

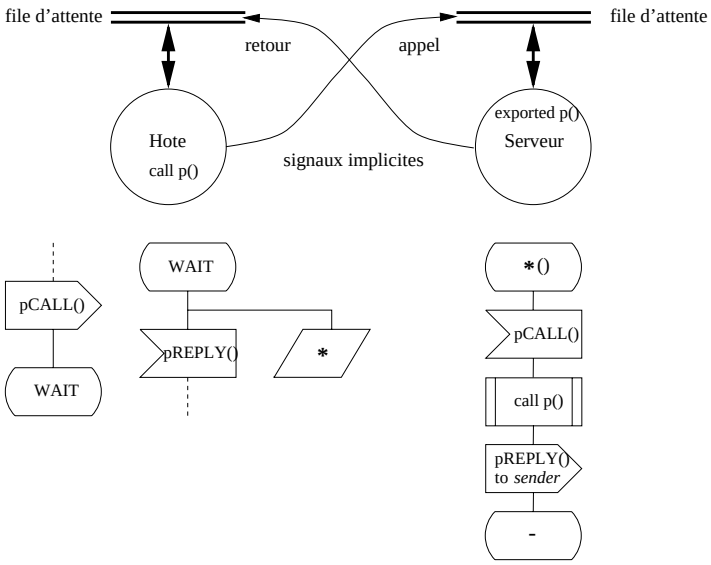


FIG. 5.6: Modèle d'exécution SDL des appels de procédure à distance

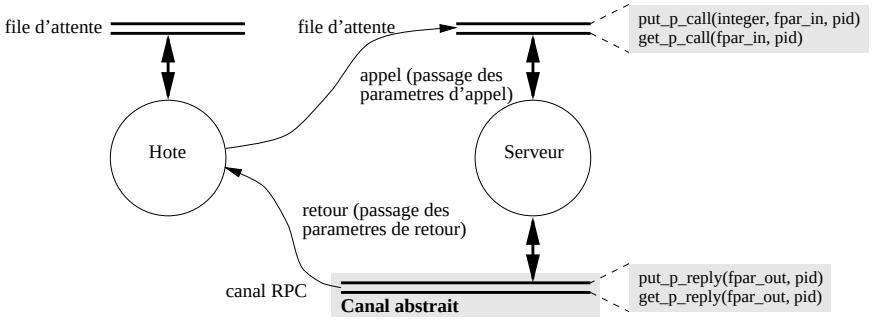
Notre approche de modélisation des appels de procédure à distance SDL pour la synthèse repose sur l'utilisation d'un canal spécifique pour l'exécution des *remote procedure call* (figure 5.7). A chaque processus serveur de procédure sera associé un canal utilisé pour le retour des paramètres. Un processus qui fait un appel de procédure à distance ira écrire la requête dans la file d'attente du

serveur ainsi que les paramètres d'appel de la procédure et se mettra en attente des résultats sur le canal *RPC*. Les différents envois de signaux d'appel et de retour de procédure, qui sont implicites en SDL, seront explicités. Un fois le signal extrait de la file d'attente et la procédure exécutée, le processus serveur retournera les paramètres résultats sur le canal *RPC*.

Le canal du processus serveur devra offrir deux primitives de communication permettant de lire et d'écrire une requête de *RPC* et les paramètres d'entrée de la procédure. Le canal *RPC* offrira, pour chaque procédure SDL exportée, deux primitives de communication permettant de lire et d'écrire les paramètres de retour de la procédure.

Plusieurs processus pouvant être en attente d'un résultat d'appel de procédure à distance sur le canal *RPC*, il est nécessaire de pouvoir identifier l'émetteur de la requête. Un processus qui effectue un appel de procédure à distance enverra dans sa requête son *Pid* (*self*) en paramètre. Ce paramètre sera utilisé par le serveur de procédures pour renvoyer les résultats au processus hôte (*PARAMASSIGN to sender*) sans qu'il y ait de confusion possible entre différentes requêtes.

Ce modèle de traduction nous permet d'éviter d'avoir à gérer le *save* et utilise un mécanisme d'identification du processus appelant identique au cas des instances multiples (paramètre *Pid*). La réalisation des canaux *RPC* utilisant un paramètre *Pid* pour l'identification du processus destinataire est détaillée au paragraphe 4.4.3.




```
(CUCALL put_p_call serveur_channel
    (PARAMASSIGN sender self)
    (PARAMASSIGN ... param1_in)
    (PARAMASSIGN ... param2_in)
)
(NEXTSTATE wait_p)
)

(STATE wait_p
    (CUCALL get_p_reply rpc_channel
        (PARAMASSIGN ... param_out)
    )
```

```
(STATE idle
    (CUCALL get_signal serveur_channel
        (PARAMASSIGN sdl_signal sdl_signal)
    )
    (CASE
        (ALT (= sdl_signal p_call)
            (CUCALL get_p_call serveur_channel
                (PARAMASSIGN sender sender)
                (PARAMASSIGN ... param1_in)
                (PARAMASSIGN ... param2_in)
            )
            (PCALL p
                (PARAMASSIGN fpar1_in param1_in)
                (PARAMASSIGN fpar2_in param2_in)
                (PARAMASSIGN fpar_out param_out)
            )
            (CUCALL put_p_reply rpc_channel
                (PARAMASSIGN to sender)
                (PARAMASSIGN ... param_out)
            )
            (NEXTSTATE idle)
        )
    (ALT (= sdl_signal ...
```

FIG. 5.7: Modélisation des appels de procédure à distance SDL en SOLAR

Durant l'étape de partitionnement, les machines d'états pourront être découpées et fusionnées afin d'obtenir la solution désirée. Cette étape peut générer des communications additionnelles telles que les variables partagées. Tous les détails d'implémentation de la communication seront fixés par l'étape de synthèse de la communication.

5.4.4 Génération du code C/VHDL

L'étape de génération de code traduit le code intermédiaire SOLAR en C/VHDL. Pour effectuer cette traduction il est nécessaire d'avoir effectué les étapes de synthèse de la communication et d'affectation logiciel/matériel de chaque processeur. Chaque table d'état SOLAR sera traduite dans l'automate d'états finis correspondant. Dans le cas de génération de code C l'automate SDL sera traduit comme une fonction C. Afin de pouvoir cosimuler le programme C avec le reste du système celui-ci sera encapsulé dans l'entité VHDL correspondant au processus [140]. Les procédures de communication feront l'interface entre le programme C et les ports de l'entité VHDL [141]. Dans le cas de la génération de code VHDL, l'automate SDL sera traduit comme un processus VHDL (*process*) contenant l'automate d'états finis SDL.

La table 5.2 résume les concepts de traduction d'une spécification SDL pour la co-simulation en C/VHDL :

- un système sera traduit comme une entité VHDL.
- un bloc sera traduit comme une entité VHDL structurelle et instancié (composant VHDL).
- un processus sera traduit comme une entité VHDL et sera instancié (composant VHDL). L'architecture de l'entité sera composée d'un processus C dans le cas d'une implémentation logicielle et par une architecture VHDL dans le cas d'une implémentation matérielle.

spécification	**implémentation**	
SDL	**VHDL**	**C**
système	entité/architecture	
bloc	entité/architecture + composant	
processus	entité/architecture + processus VHDL	entité/architecture + programme C
input, *output*	appels de procédures de communication écrivant sur les ports de l'entité	

TAB. 5.2: Génération de code C/VHDL pour la co-simulation

La figure 5.8 donne le code VHDL généré pour l'exemple de la figure 5.3. La figure 5.9 représente les primitives de communication permettant de lire et d'écrire le signal *ATMAAL_SDU*. Toutes les instructions *input* et *output* ont été remplacées par des appels à ces procédures. La file d'attente n'est pas détaillée, c'est un fifo standard. Les primitives de communication lisent et écrivent sur les ports du fifo.

```
entity sar_receiver is
   Generic
   (
     IPCKEY: INTEGER:= 1
   );
   Port (
     CLK: IN BIT;
     RST: IN BIT;
     N_sar_receiver_rewr_rdy: IN BIT;
     N_sar_receiver_read_req: OUT BIT;
     N_sar_receiver_data_int: INOUT RESOLVED_INTEGER;
     N_sar_receiver_data_real: INOUT RESOLVED_REAL;
     N_cpcs_receiver_wr_req: OUT BIT;
     N_cpcs_receiver_rewr_rdy: IN BIT;
     N_cpcs_receiver_data_val: OUT BIT;
     N_cpcs_receiver_data_int: INOUT RESOLVED_INTEGER;
     N_cpcs_receiver_data_real: INOUT RESOLVED_REAL
   );
end sar_receiver;
architecture sar_receiver_behaviour of sar_receiver is
begin
  process
    constant atmaal_sdu: INTEGER:= 5;
    constant atmaal_endsdu: INTEGER:= 6;
    constant sarcpcs_data: INTEGER:= 9;
    constant sarcpcs_len: INTEGER:= 10;
    variable sdl_signal: INTEGER;
    variable pos: INTEGER:= 0;
    variable i: INTEGER;
    variable aal_frame: data;
    variable atm_data: atm_data;
    variable PCALL: INTEGER:= 1;
    procedure sar_receiver_get_signal(sdl_signal: OUT INTEGER );
    procedure sar_receiver_get_ATM_DATA(
        param_1: OUT INTEGER_VECTOR(48 downto 1) );
    procedure cpcs_receiver_put_int(sdl_signal: IN INTEGER;
        param_1: IN INTEGER );
    procedure cpcs_receiver_put_DATA(sdl_signal: IN INTEGER;
        param_1: IN INTEGER_VECTOR(150 downto 1) );
    type statetable_StateType is (start, idle, l_idl);
    variable statetable_NextState:statetable_StateType:= start;
  begin
    wait until (rising_edge(CLK)) OR (RST='1');
    if (RST='1') then
      statetable_NextState:= start;
      wait until (rising_edge(CLK));
    end if;
    StateTable_sar_receiver: loop
      case statetable_NextState is
        when(start) =>
          statetable_NextState:= idle;
          exit StateTable_sar_receiver;
        when(idle) =>
          sar_receiver_get_signal(sdl_signal=> sdl_signal);
          if (sdl_signal = 5) then
            sar_receiver_get_ATM_DATA(param_1=> atm_data);
            i:= 1;
            statetable_NextState:= l_idl;
            exit StateTable_sar_receiver;
          else
            if (sdl_signal = 6) then
              cpcs_receiver_put_int(sdl_signal=> 10,param_1=> pos);
              cpcs_receiver_put_DATA(sdl_signal=> 9,
                  param_1=> aal_frame);
              pos:= 0;
              statetable_NextState:= idle;
              exit StateTable_sar_receiver;
            else
            end if;
          end if;
        when(l_idl) =>
          if (i <= 48) then
            aal_frame((pos + i)):= atm_data(i);
            i:= (i + 1);
            statetable_NextState:= l_idl;
            exit StateTable_sar_receiver;
          else
            pos:= (pos + 48);
            statetable_NextState:= idle;
            exit StateTable_sar_receiver;
          end if;
      end case;
      exit StateTable_sar_receiver;
    end loop StateTable_sar_receiver;
  end process;
end sar_receiver_behaviour;
```

FIG. 5.8: Code VHDL généré pour le processus *sar_receiver*

```
procedure sar_receiver_get_ATM_DATA(param_1: OUT
      INTEGER_VECTOR(48 downto 1)) is
  variable indice: INTEGER:= 1;
  variable get_ATM_DATAREADY: INTEGER:= 0;
  type get_ATM_DATA_StateType is (get_ATM_DATAIdle, request, waitrdy);
  variable get_ATM_DATA_NextState: get_ATM_DATA_StateType:= request;
begin
  PCALL:= 1;
  while (PCALL = 1) loop
    StateTable_get_ATM_DATA: loop
      case get_ATM_DATA_NextState is
        when(request) =>
          if (N_sar_receiver_rewr_rdy = '1') then
            if NOT(N_sar_receiver_rewr_rdy = '0') then
              wait until ((N_sar_receiver_rewr_rdy = '0'));
            end if;
          end if;
          N_sar_receiver_read_req<= '1';
          if NOT(N_sar_receiver_rewr_rdy = '1') then
            wait until ((N_sar_receiver_rewr_rdy = '1'));
          end if;
          param_1(indice):= N_sar_receiver_data_int;
          N_sar_receiver_data_int<= INTEGER_LOW;
          N_sar_receiver_data_real<= REAL_LOW;
          N_sar_receiver_read_req<= '0';
          if (indice < 48) then
            indice:= (1 + indice);
            get_ATM_DATA_NextState:= request;
            exit StateTable_get_ATM_DATA;
          else
            get_ATM_DATA_NextState:= waitrdy;
            exit StateTable_get_ATM_DATA;
          end if;
        when(waitrdy) =>
          get_ATM_DATA_NextState:= get_ATM_DATAIdle;
        when OTHERS =>
          PCALL:= 0;
          get_ATM_DATA_NextState:= request;
      end case;
      exit StateTable_get_ATM_DATA;
    end loop StateTable_get_ATM_DATA;
    if (PCALL=1) then
      wait until (rising_edge(CLK));
    end if;
  end loop;
end sar_receiver_get_ATM_DATA;

procedure sar_receiver_put_ATM_DATA(sdl_signal: IN INTEGER;
      param_1: IN INTEGER_VECTOR(48 downto 1) ) is
  variable indice: INTEGER:= 1;
  variable put_ATM_DATAREADY: INTEGER:= 0;
  type put_ATM_DATA_StateType is (put_ATM_DATAIdle, request,
      waitvalid, requestwait, paramwait);
  variable put_ATM_DATA_NextState: put_ATM_DATA_StateType:= request;
begin
  PCALL:= 1;
  while (PCALL = 1) loop
    StateTable_put_ATM_DATA: loop
      case put_ATM_DATA_NextState is
        when(request) =>
```

```
  if (N_sar_receiver_rewr_rdy = '1') then
    if NOT(N_sar_receiver_rewr_rdy = '0') then
      wait until ((N_sar_receiver_rewr_rdy = '0'));
    end if;
  end if;
  N_sar_receiver_wr_req<= '1';
  put_ATM_DATA_NextState:= waitvalid;
  exit StateTable_put_ATM_DATA;
when(waitvalid) =>
  if NOT(N_sar_receiver_rewr_rdy = '1') then
    wait until ((N_sar_receiver_rewr_rdy = '1'));
  end if;
  N_sar_receiver_wr_req<= '0';
  N_sar_receiver_data_int<= sdl_signal;
  N_sar_receiver_data_real<= REAL_LOW;
  N_sar_receiver_data_val<= '1';
  put_ATM_DATA_NextState:= requestwait;
  exit StateTable_put_ATM_DATA;
when(requestwait) =>
  if NOT(N_sar_receiver_rewr_rdy = '0') then
    wait until ((N_sar_receiver_rewr_rdy = '0'));
  end if;
  N_sar_receiver_data_int<= INTEGER_LOW;
  N_sar_receiver_data_real<= REAL_LOW;
  N_sar_receiver_data_val<= '0';
  N_sar_receiver_wr_req<= '1';
  if NOT(N_sar_receiver_rewr_rdy = '1') then
    wait until ((N_sar_receiver_rewr_rdy = '1'));
  end if;
  N_sar_receiver_wr_req<= '0';
```

```
  N_sar_receiver_data_int<= param_1(indice);
  N_sar_receiver_data_real<= REAL_LOW;
  N_sar_receiver_data_val<= '1';
  if (indice < 48) then
    indice:= (1 + indice);
    put_ATM_DATA_NextState:= requestwait;
    exit StateTable_put_ATM_DATA;
  else
    put_ATM_DATA_NextState:= paramwait;
    exit StateTable_put_ATM_DATA;
  end if;
when(paramwait) =>
  if NOT(N_sar_receiver_rewr_rdy = '0') then
    wait until ((N_sar_receiver_rewr_rdy = '0'));
  end if;
  N_sar_receiver_data_int<= INTEGER_LOW;
  N_sar_receiver_data_real<= REAL_LOW;
  N_sar_receiver_data_val<= '0';
  put_ATM_DATA_NextState:= put_ATM_DATAIdle;
when OTHERS =>
  PCALL:= 0;
  put_ATM_DATA_NextState:= request;
end case;
exit StateTable_put_ATM_DATA;
end loop StateTable_put_ATM_DATA;
if (PCALL=1) then
  wait until (rising_edge(CLK));
end if;
end loop;
end sar_receiver_put_ATM_DATA;
```

FIG. 5.9: Implémentation des primitives de communication en VHDL

5.5 RÉSULTATS

Nous présentons ici quelques résultats concernant la génération de code VHDL à partir de spécification systèmes en SDL.

Nous pouvons voir sur la table 5.3 l'augmentation de la taille du code entre une spécification SDL et son implémentation en VHDL. La communication qui ne représente que 10-20% d'une spécification SDL représente plus de 50% de l'implémentation VHDL. Cela est principalement dû à l'abstraction de la communication en SDL. La taille du modèle VHDL est environ huit fois celle du modèle SDL.

design	**complexite**	**lignes SDL**		**lignes VHDL**		**augmentation**
		compor tement	**commu nication**	**compor tement**	**commu nication**	
contrôleur pid	4 processus 34 états 33 transitions	331	73 (22 %)	2403	1194 (50 %)	726 %
contrôleur logique floue	9 processus 16 états 29 transitions	560	88 (15 %)	4765	2856 (60%)	850 %
TCP/IP sur ATM	9 processus 20 états 40 transitions	794	103 (13 %)	7210	5382 (75%)	908 %

TAB. 5.3: Résultat de la traduction SDL à VHDL

En SDL, émettre ou recevoir un signal ne demande qu'une instruction. Le protocole de la communication, l'acheminement du signal et la file d'attente sont implicites. Lors de l'implémentation de

la spécification, la communication devient explicite requiérant un protocole spécifique, un contrôleur de communication et des bus. L'augmentation dépend beaucoup de la complexité de la communication qui est complètement ocultée en SDL. Dans le cas de TCP/IP sur ATM les communications sont plus complexes que dans le cas du contrôleur PID. Cette complexité dépend du protocole de la communication mais aussi des données transférées sur le canal.

constructeur SDL		**implémentation**	**note**
processus	*start*	état d'initialistation de la table d'états (*ENTRYSTATE*)	
	state	état de l'automate + lecture de l'identificateur de signal dans la file d'attente (*CUCALL*)	
	*state *, state *()*	expansion en ligne dans tous les états concernés	
	transition	choix multiple sur le signal (*CASE-ALT*)	
	input	lecture des paramètres du signal	
	input , , , ;	duplication de la transition pour chaque signal	
	save, *save **, *save *()*	Lecture non destructive de l'identificateur de signal + sauvegarde du signal	
	input provided	lecture non destructive de l'identificateur de signal, test de la condition puis sauvegarde du signal ou exécution de la transition	
	provided	test du canal puis test des signaux continus dans l'ordre des priorités	
	output	écriture du signal et de ses paramètres Le canal destinataire doit pouvoir être identifié statiquement	
	task	opération équivalante	
	call	appel de procédure local (*PCALL)*	
	call (procédure importée)	envoi de la requête et des paramètres dans la file d'attente du serveur + saut dans l'état d'attente, lecture des paramètres de retour dans le canal *RPC*	
	decision, *else*	(*IF-THEN-ELSE*) ou (*CASE-ALT*)	
	label	nouvel état de l'automate (*STATE*) + saut dans cet état (*NEXTSTATE*)	
	join	saut dans l'état correspondant au label	
	nextstate	saut dans cet état (*NEXTSTATE*)	
	stop	création d'un état *stop* + rebouclage dans cet état	

constructeur SDL		**implémentation**	**note**
structure	*system*	unité de conception structurelle	
	bloc	unité de conception structurelle	
	process	unité de conception comportementale + 1 canal abstrait + 1 table d'états	
	process(x,x)	instances multiples + 1 canal abstrait	
	service	table d'états	
communication	*channel*	non représenté	
	signalroute	non représenté	
	connect	non représenté	
	signal	identificateur de signal + 3 primitives de communication attachées au canal abstrait	
type de données	*dcl*, *synonym*	variable, constantes	
	newtype, *litterals*	type, type énuméré	
	ADT + opérateurs	unités fonctionnelles + opérations	
variable partagées	*import*, *export*	canal abstrait + 2 primitives permettant de lire/écrire dans le canal	
procédures exportées	*imported*, *exported*	1 canal abstrait par serveur + 2 primitives permettant de lire/écrire les paramètres de retour	
adressage	*pid*	utilisation restreinte aux instances multiples de processus et *RPC*	
	to	paramètre supplémentaire de la procédure de communication	
	self	défini seulement pour les processus instance multiple ou utilisant les appels de procédure à distance	
	sender	lors de la lecture du signal, prend la valeur du paramètre *to*	

TAB. 5.4: Sous ensemble SDL supporté

5.6 CONCLUSION

Dans ce chapitre nous avons présenté une approche dans laquelle le langage SDL (Specification and Description Language) peut être utilisé comme langage de spécification système pour générer une implémentation en C/VHDL. Nous avons donné les modèles de traduction vers la forme intermédiaire SOLAR pour les principaux concepts SDL. Notre approche supporte un large sous-ensemble de SDL, c'est à dire tous les concepts qui peuvent être utilisés pour la spécification de matériel. Le code VHDL généré est acceptable par les outils de simulation et de synthèse comportementale standard. Cette approche résoud la principale difficultée rencontrée lors de la traduction d'une

spécification SDL pour la synthèse, à savoir l'abstraction de la communication SDL. Une implémentation efficace des schémas de communication système est possible à travers une bibliothèque de communication. Celle-ci contient une réalisation des primitives de communication nécessaire.

Chapitre 6

APPLICATION À UN SYSTÈME DE TÉLÉCOMMUNICATION

Dans ce chapitre nous allons appliquer notre méthode de codesign à un système de télecommunication : l'ATM. Nous partirons d'une spécification exécutable en SDL qui sera simulée. Celle-ci sera utilisée par l'outil COSMOS pour aboutir à un prototype virtuel C/VHDL. Dans un premier temps nous utiliserons une spécification simplifée de L'ATM. Elle sera enrichie par la suite lorsqu'une première cosimulation de code C/VHDL aura eu lieu. Cette application nous permettra de valider notre chaine et de mettre en évidence les forces et les faiblesses de SDL pour la spécification système [38].

6.1 INTRODUCTION

Ce chapitre présente le codesign d'une application de télécommunication ATM (*Asynchronous Transfert Mode*) [115]. Nous mettons en avant les problèmes rencontrés lors de la spécification système en SDL et les limitations de notre modèle qui en ont résulté. La spécification SDL sera co-simulée au niveau système (co-simulation SDL-SDL) et validée avant d'être utilisée dans l'outil COSMOS. Nous lui appliquerons les étapes de partitionnement, de synthèse de la communication et de génération de code C/VHDL. Cette description sera ensuite co-simulée (co-simulation C/VHDL) pour vérifier sa fonctionnalité. D'autres approches se sont concentrées sur des problèmes spécifiques lors de la synthèse système du protocole ATM tels que la gestion de la mémoire [126].

L'étape de partitionnement déterminera les couches à réaliser en logiciel ou en matériel en fonction des performances requises. L'étape de synthèse de la communication choisira les protocoles de communication entre les processus en fonction des taux de transfert requis. Le but étant de pouvoir, à partir d'une même description système, adapter la réalisation en fonction des performances requises (transmition de données, vidéo, téléphonie, vidéophonie).

6.2 STRUCTURE GÉNÉRALE DU MODÈLE

Nous implémenterons le protocole TCP/IP au dessus de l'ATM sous forme du modèle classique en couches (figure 6.1). Des simplifications majeures ont été faites afin d'obtenir rapidement un modèle simulable. Elle sont les suivantes :

- Le modèle prendra en compte deux systèmes communicant à travers une connection unique. Les algorithmes de routage, de congestion, de contrôle de traffic et de gestion de ressources ne seront pas implémentés.
- La correction d'erreurs ne sera pas implémentée. Cela élimine la correction et la gestion des erreurs à tous les niveaux. En consequence le *Internet Control Message Protocol* (ICMP) ne sera pas présent. La retransmission des trames perdues sera assurée par la couche *transport*.
- Le modèle ne prendra pas en compte le désordonnancement des trames. En consequence le méchanisme de fenêtre glissante (*sliding window*) ne sera pas implémenté. La fenêtre aura une taille fixe de 1.

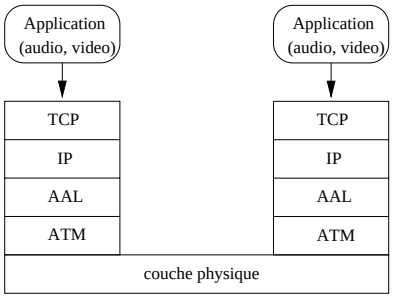


FIG. 6.1: Modèle général de l'application

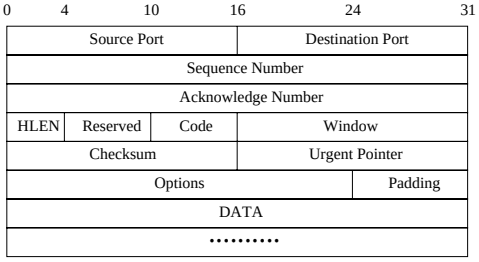
6.3 MODÉLISATION PROTOCOLE TCP/IP

Conceptuellement le protocole TCP/IP [31] offre un ensemble de trois services à l'utilisateur qui sont :

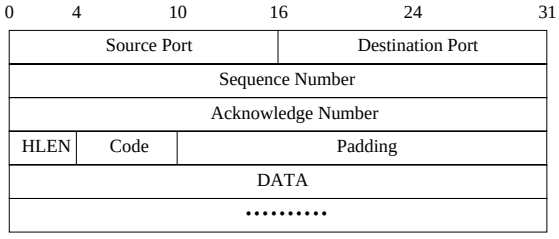
- un service de communication entre applications.
- un service de connection fiable.
- un service de transport des packets en mode non connecté.

6.3.1 La couche TCP

La couche TCP (*Transport Control Protocol*) offre une communication fiable entre deux machines de vitesses différentes en utilisant les services de la couche IP. Comme de nombreux autres protocoles, TCP utilise les timers de retransmission pour assurer la fiabilité. Contrairement à d'autres protocoles, TCP est étudié pour fonctionner même si les datagrammes sont perdus, retardés, dupliqués, délivrés en désordre, tronqués ou corrompus. De plus, TCP autorise les machines qui communiquent à rebooter et réétablir des connections à tout moment sans confondre les connections précédemment ouvertes de celles nouvellement ouvertes.



6.2.1: Structure d'une entête TCP



6.2.2: Structure de l'entête utilisée

FIG. 6.2: Format d'une trame TCP

La structure d'une trame TCP est représentée en figure 6.2.1 et celle utilisée dans notre modèle en figure 6.2.2. Le modèle TCP a été simplifié afin de pouvoir le rendre facilement implémentable. Les champs *Reserved*, *Window*, *Checksum*, *Urgent Pointer* et *Option* ne sont pas implémentés. Une entête de taille fixe (champ *header length* = 4) sera utilisée. TCP a la capacité d'ouvrir une connection avec un numéro de séquence aléatoire (champ *sequence number*). Cette option requérant un générateur de nombre aléatoire, nous ouvrirons systématiquement une connection avec un numéro de séquence égal à 1. Le champ *Window* ne sera pas utilisé, la taille de la fenêtre étant fixée à 1. Le champ *Code* est composé de six bits détaillés dans la table 6.1. Tous les bits du champ *code* sauf *PSH* et *URG* seront utilisés.

Le protocole TCP est décrit sous forme d'un automate d'états finis. Celui-ci a été modifié en tenant compte des simplifications apportées au modèle (paragraphe 6.2). Il est représenté en figure

NOM	Signification lorsque le bit vaut 1
URG	champ *Urgent Pointer* valide
ACK	champ *Acknowledge* valide
PSH	cette trame doit être poussée
RST	reset de la connection
SYN	synchronisation des numéros de séquence
FIN	demande de fermeture de la connection

TAB. 6.1: Signification du champ *code*

6.3. Chaque application commence dans l'état *Closed*. Les transitions sont étiquetées avec les signaux de garde (en italique) suivi des signaux émis lors de son exécution.

Etablissement et fermeture d'un connection

C'est la couche TCP qui ouvre et ferme une connection entre applications. L'établissement d'une connection se fait par un protocole de *handshake* à trois phases (figure 6.4.1) alors que la fermeture se fait par un protocole *handshake* à trois phases modifié (figure 6.4.2). Une application peut demander l'ouverture d'une connection par la commande *active open* ou se mettre à l'écoute du réseau et attendre une connection par la commande *passive open*.

Lorsque la connection est établie

Lorsque l'automate atteint l'état *Established* l'échange de paquet de données peut avoir lieu. Ce processus est décrit lui-même comme un automate. L'état *Established* est donc hiérarchique et contient trois sous-états qui sont représentés avec leurs transitions en figure 6.5. SDL ne supportant pas la hiérarchie comportementale, l'état *Established* a été mis à plat.

La figure 6.5 décrit l'état *established*. Les transitions correspondantes sont détaillées ci dessous :

1. *Send*.
2. toutes les données/acquittement envoyés.
3. expiration du timer de retransmission.
4. acquittement reçu.
5. toutes les données/acquittement envoyés.
6. expiration du timer de retransmission.

Lorsque l'automate entre dans l'état hiérarchique *Established*, il entre dans l'état *idle*. L'automate entre dans l'état *transmit* à la réception d'une commande *send* provenant de l'application ou si un *acknowlegement* est prêt à être envoyé en réponse à une trame reçue. Le processus génère et envoie la trame, arme un timer puis retourne dans l'état *idle*. Si le timer de retransmission expire, l'automate entre dans l'état *retransmit* et effectue la retransmission de la dernière trame envoyée non acquittée. Dans le cas de réception multiple d'une même trame, celle-ci est éliminée mais un *acknowlegement* est envoyé pour informer le processus émetteur que la trame a été reçue. Lorsque toutes les trames

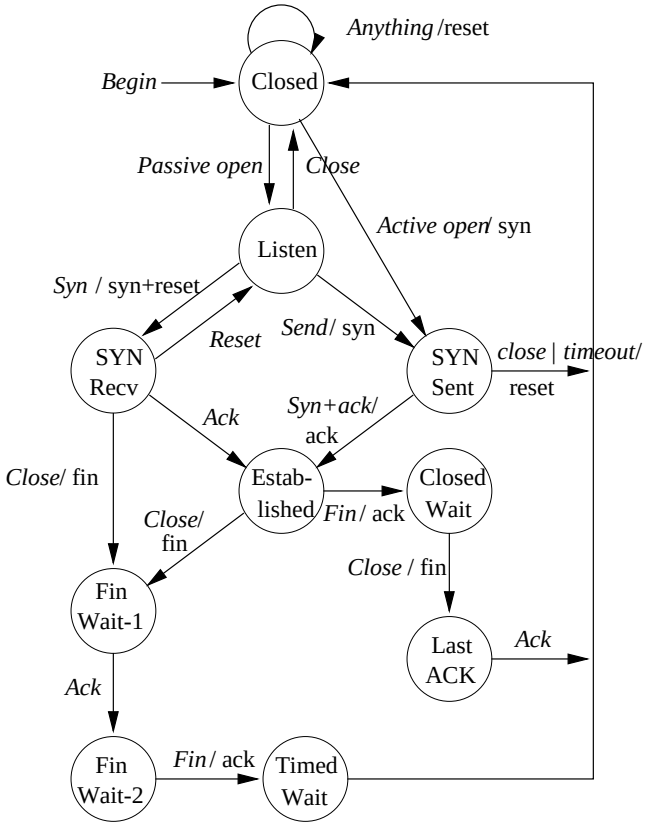


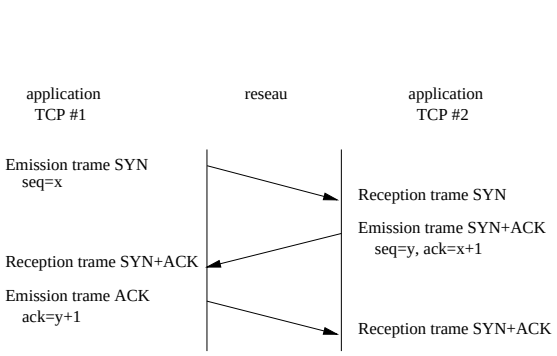
FIG. 6.3: Automate du protocole TCP

ont été envoyées ou acquittées l'automate retourne dans l'état *idle*. Un protocole d'acquittement simple (figure 6.6) est utilisé pour le transfert des trames. Un protocole de transfert beaucoup plus efficace peut être implémenté en utilisant le concept de fenêtre glissante. Ce protocole requiérant l'implémentation du contrôle de congestion, il n'a pas été implémenté dans un premier temps. Une description SDL de ce protocole peut être trouvée dans [132].

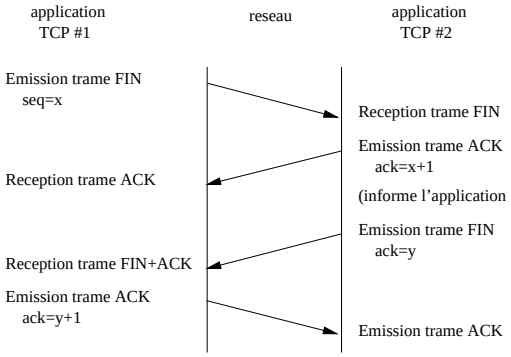
6.3.2 La couche IP

Le principal service offert par la couche IP (Internet Protocol) est un service de transport non connecté et non fiable. Le protocol internet spécifie le format des packets internet, appelés datagrammes, qui sont constitué d'une 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 6.7.1 détaille une entête IP. Afin de permettre une comparaison avec notre modèle, celui-ci est représenté en figure 6.7.2.

Les champs ont été réduits au minimum nécessaire, tel que les adresses de *source* et de *destination*. Le champ *protocole* contient le type de protocole utilisant les services de la couche IP. Pour un service TCP, cette valeur vaut 6. D'autres utilisateurs du service TCP tel que UDP (User



6.4.1: Ouverture d'une connection TCP par protocole *handshake*



6.4.2: Fermeture d'une connection TCP

FIG. 6.4: Ouverture et fermeture d'une connection TCP

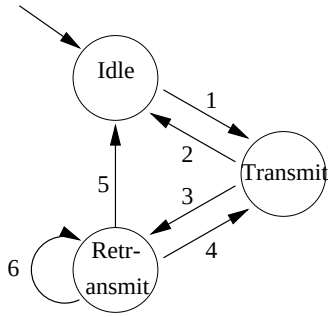


FIG. 6.5: Automate de l'état *Established*

Datagram Protocol), ICMP correspondent à d'autres valeurs du champ protocole. Le champ *vers* contient le numéro de 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 *header length* (HLEN) est fixé à trois et n'est donc pas nécessaire, mais il est utilisé pour obtenir une entête de longueur multiple de 32 bits. Le champ HLEN 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 qui est la couche 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 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 retirée avant que celui-ci soit envoyé vers la couche TCP.

L'automate IP

L'automate IP est représenté en figure 6.8. Les transitions, détaillées ci-dessous, correspondent à la réception ou à l'émission de trames.

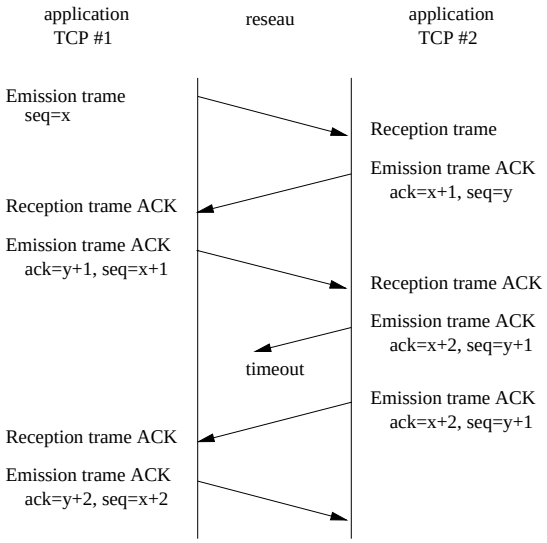
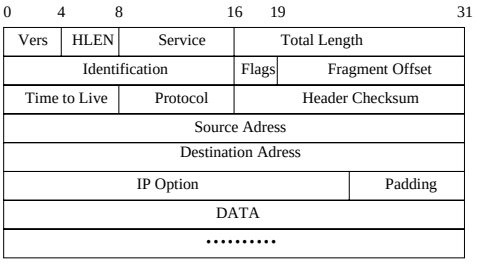


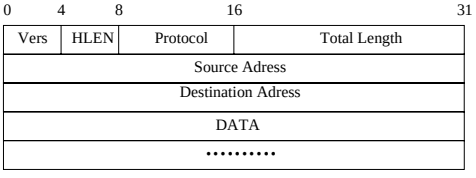
FIG. 6.6: Protocole bidirectionnel d'échange avec acquittement et retransmission

1. réception d'un segment de la couche AAL.
2. réception d'un segment de la couche TCP.
3. segment envoyé à la couche TCP ou éliminé.
4. datagramme envoyé à la couche AAL.

Si un segment est reçu de la couche AAL, l'automate entre dans l'état *Receive*. Si le segment est reçu de la couche TCP, l'automate entre 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*.



6.7.1: Structure d'une entête IP



6.7.2: Structure de l'entête utilisée

FIG. 6.7: Format d'un datagramme IP

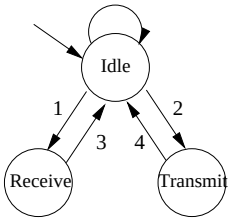


FIG. 6.8: Automate de la couche IP

6.4 LA COUCHE AAL

La couche AAL (*ATM Adaptation Layer*) améliore la qualité du service fourni par la couche ATM au niveau requis par un service spécifique. Il peut s'agir de services utilisateur, de fonction de contrôle telle que la signalisation et de 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 réassemblage (SAR ou *Segmentation And Reassembly*).

La sous-couche de convergence exécute des fonctions telles que l'identification des messages, etc. Pour certains types d'AAL la sous-couche CS est encore décomposée en sous-couche CPSP (*Common Part Convergence Sublayer*) et SSCS (*Service Specific Convergence Sublayer*).

La sous-couche SAR a pour objectif principal la segmentation des informations provenant de la couche supérieure dans la taille d'une cellule ATM, ainsi que l'opération inverse, le réassemblage du contenu des cellules de la couche ATM en segments à fournir à la couche supérieure.

Une description détaillée de la couche d'adaptation pour les protocoles AAL 3/4 et AAL 5 peut être trouvée dans [90].

Choix d'une AAL

Un datagramme IP ne tient pas dans une cellule ATM de 53 octets et doit donc être segmenté. Cette fonctionnalité sera assurée par la couche AAL. Pour notre modèle, il a été décidé d'implémenter AAL 5. AAL 5 supporte les protocoles orientés sans connection, à peu d'overhead et permet une implémentation rapide.

6.4.1 La couche AAL 5

La couche AAL 5 supporte deux modes d'interface avec la couche supérieure. Le mode *message* et le mode *stream*. Dans le mode *message* une AAL-SDU (*AAL-Service Data Unit*) est passée à la sous-couche SAR en une SAR-SDU exactement. Dans le mode *stream* une AAL-SDU peut être passée comme une ou plusieurs SAR-SDU. Le mode *message* étant le plus simple c'est celui qui sera implémenté dans notre modèle. Si la sous-couche SSCS n'est pas implémentée, les AAL-SDU seront identiques au CPCS-SDU.

AAL 5 offre deux modes de transfert : assuré et non assuré. Le mode de transfert assuré effectue une correction d'erreur et retransmet les trames CS-PDU perdues ou corrompues ainsi qu'un contrôle de flux. Le mode non assuré n'effectue pas de correction d'erreurs et de retransmission des trames

perdues ou corrompues. A la réception d'une trame erronée, détectée par le CRC 32 bits, la CS-PDU sera simplement écartée. Il incombe aux couches supérieures de demander la retransmission.

Contrairement à la couche AAl 3/4 la couche AAL 5 ne permet pas de multiplexer plusieurs SAR-PDU sur une même connection virtuelle ATM (champ *MID*). AAL 5 n'offre que des connections point à point.

La sous-couche convergence

La sous-couche CPCS accepte des paquets (*CPCS-SDU*) de taille variable entre 1 et 65535 octets. Celle-ci est rendue multiple de 48 octet 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 6.9 représente le format d'une cellule CPCS-PDU (*CPCS-Protocol Data Unit*). L'en-queue d'une CPCS-PDU a le format suivant :

- Le champ *Padding* est utilisé pour rendre la taille (données + en-queue) d'une CPCS multiple de 48 octets par ajout de 0-47 octet(s) de bourrage.
- Le champ *User to User Indication* (UU) est réservé pour le transfert d'informations utilisateur. Il sera mis à zéro.
- Le champ *Commom Part Indicator* (CPI) a plusieurs fonctions dont celle d'aligner l'en-queue sur 64 bits. Les autres fonctions ne sont pas encore définies. Il sera donc mis à zero.
- Le champ *Length* donne la longueur de la charge utile (données + bourrage) de la CPCS-PDU. Une CPCS-PDU avec un champ longueur nul indique que le transfert de la CPCS-PDU courante est avorté.
- Le champ *Cyclic Redundancy Check* (CRC) contient un CRC de 32 bits qui est utilisé pour détecter les trames corrompues. Celui-ci est calculé sur la totalité de la CPCS-PDU. Si une erreur est détectée, la trame sera abandonnée. Toute retransmission devra être redemandée par la couche TCP qui en sera informée par l'expiration de son timer.

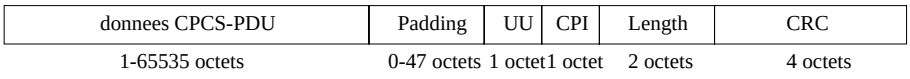


FIG. 6.9: Format d'un segment CPCS-PDU

La couche SSCS est vide pour les protocoles non connectés et ne sera donc pas implémentée dans notre modèle.

La sous-couche SAR

La sous-couche SAR accepte les SAR-SDU (*SAR-Service Data Unit*) de longueur variable, multiple de 48 octet de la sous-couche CPCS et génère des SAR-PDU (*SAR-Protocol Data Unit*) de 48 octets identiques aux cellules de la couche ATM. La délimitation du SAR-SDU s'effectue à l'aide du bit utilisateur du champ PTI (*Payload Type Identifier*) de l'entête de la couche ATM qui indique si une SAR-PDU contient le debut ou la suite ou bien la dernière cellule d'une SAR-SDU. Ce bit

est à 0 pour le début ou la suite des cellules SAR-PDU d'une SAR-SDU et à 1 dans la dernière SAR-PDU (figure 6.10).

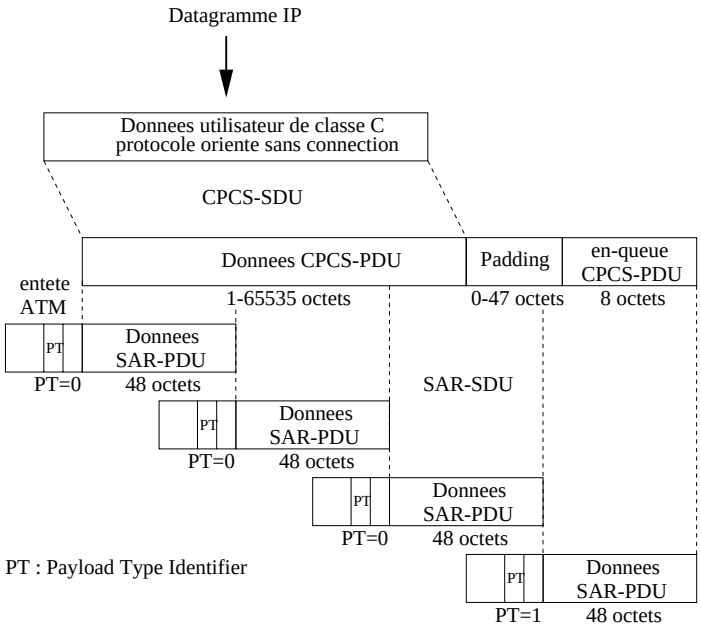


FIG. 6.10: Flôt de données de la couche AAL 5

6.4.2 L'automate AAL

L'automate de la couche AAL a été divisé en deux processus concurrents, un pour l'émission et un pour la réception.

L'automate de réception

La figure 6.11 décrit l'automate de réception de la couche AAL. Celui-ci commence dans l'état *Idle*. A la réception d'un segment de la couche ATM, il entre dans l'état *Active* si le segment est indiqué comme étant le début ou la suite d'une CPCS-PDU (PT=0), réassemble le datagramme jusqu'à la réception du segment de fin (PTI=1) et effectue le contrôle du CRC avant de retourner dans l'état *Idle*. Si le segment tient dans une seule cellule ATM (PT=1) l'automate traite le segment et reste dans l'état *Idle*. Dans le cas d'un segment érroné celui-ci est simplement écarté.

Les transitions de la figure 6.11 sont les suivantes :

1. segment ATM, PTI=1 (fin de CPCS-PDU).
2. segment ATM, PTI=0 (début ou suite de CPCS-PDU).

– Oversized : la taille de la CPCS-PDU dépasse 65535 octets.

– NoErrors : la CPCS-PDU n'est pas corrompue.

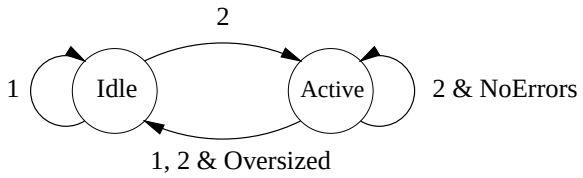


FIG. 6.11: Automate de réception de la couche AAL

L'automate d'émission

A la réception d'un datagramme, l'automate entre dans l'état *Active* si la segmentation est nécessaire, et retourne dans l'état *Idle* lorsque celle-ci se termine.

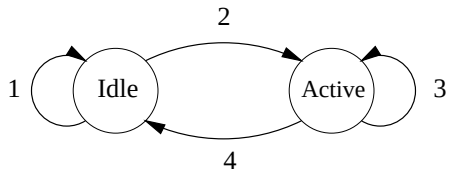


FIG. 6.12: Automate d'émission de la couche AAL

L'automate d'émission de la couche AAL est décrit en figure 6.12 et les transitions correspondantes sont détaillées ci-dessous :

1. datagramme IP de taille inférieure à 40 octets.
2. datagramme IP de taille supérieure à 40 octets.
3. segmentation en cours
4. dernier segment généré.

6.5 LA COUCHE ATM

La couche ATM est indépendante du support physique et assure les fonctions suivantes :

- ajout (respectivement suppression) de l'entête pour les cellules provenant (respectivement à destination) de la couche d'adaptation.
- contrôle de flux sur l'UNI (*User Network Interface*) (champ *GFC* de l'entête).
- multiplexage et démultiplexage des cellules et traduction d'adresses (champ VPI/VCI).
- ouverture et fermeture des connections ATM (affectation des champs VCI et/ou VPI de l'entête), gestion de la qualité de service (bit CLP) et de la bande passante requise.

La couche ATM reçoit des paquets de taille fixe (48 octets) de la couche AAL et génère de cellules ATM de taille 53 octets en y ajoutant une entête de 5 octets.

Afin de simplifier notre modèle, toute la partie signalisation, gestion OAM (*Operation And Management*) et gestion de flôt seront réduites au minimum dans un premier temps.

6.5.1 L'entête ATM

Il y a deux formats définis par l'ITU pour l'entête ATM. Nous utiliserons l'entête UNI (*User Network Interface*) représenté en figure 6.13.

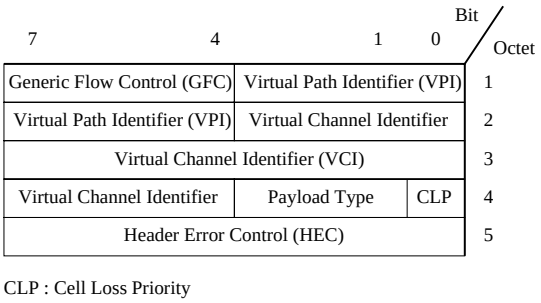


FIG. 6.13: Format de l'entête ATM

- Le champ GFC *Generic Flow Control* est utilisé pour le contrôle d'accès dans les réseaux locaux. Il sera ignoré dans notre modèle.
- Le champ VPI/VCI (*Virtual Path Identifier/Virtual Channel Identifier*) est utilisé pour le routage des cellules ATM. Les adresses IP sont d'abord traduites à l'aide d'une AAL-CEI (*AAL connection End Indicator*), d'une ATM-CEI et finalement d'une ATM-LI. Cette Traduction est effectuée à l'aide de tables de traduction. Dans notre modèle, ces tables ne seront pas implémentées.
- Le champ PT (*Payload Type Identifier*) est utilisé pour l'identification du champ données de la cellule. Il est constitué de 3 bits. Le premier bit (bit 2 de l'octet) contient l'indicateur de début/suite (bit=0) ou fin (bit=1) d'une SAR-PDU. Ce bit est affecté par la couche SAR (paragraphe 6.4.1). Le bit 3 contient des informations relatives à la congestion du réseau et le bit 4 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. Dans notre modèle, celui-ci sera mis à 0.
- le champ HEC (*Header Error Control*) est utilisé pour détecter des erreurs dans l'entête ATM au niveau physique. La valeur du champ n'est donc pas utilisée ou calculée au niveau de la couche ATM et celui-ci est donc simplement mis à 0. Au niveau de la couche physique le champ HEC est calculé et placé dans l'entête.

L'automate ATM

Dans le cas des deux automates, seulement un état est nécessaire (figure 6.14). A l'émission (respectivement réception) d'un paquet l'entête est soit générée et ajouté (respectivement enlevée).

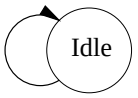


FIG. 6.14: Automate d'émission et de réception de la couche ATM

6.6 MODÉLISATION EN SDL

6.6.1 Spécification en SDL

L'ATM a été spécifié en SDL à l'aide de l'outil GEODE. La structure générale du modèle est représentée figure 6.15. Le modèle SDL a été simulé afin de détecter les erreurs fonctionnelles avant l'étape de synthèse. La totalité du modèle SDL de l'ATM est donnée en annexe A.

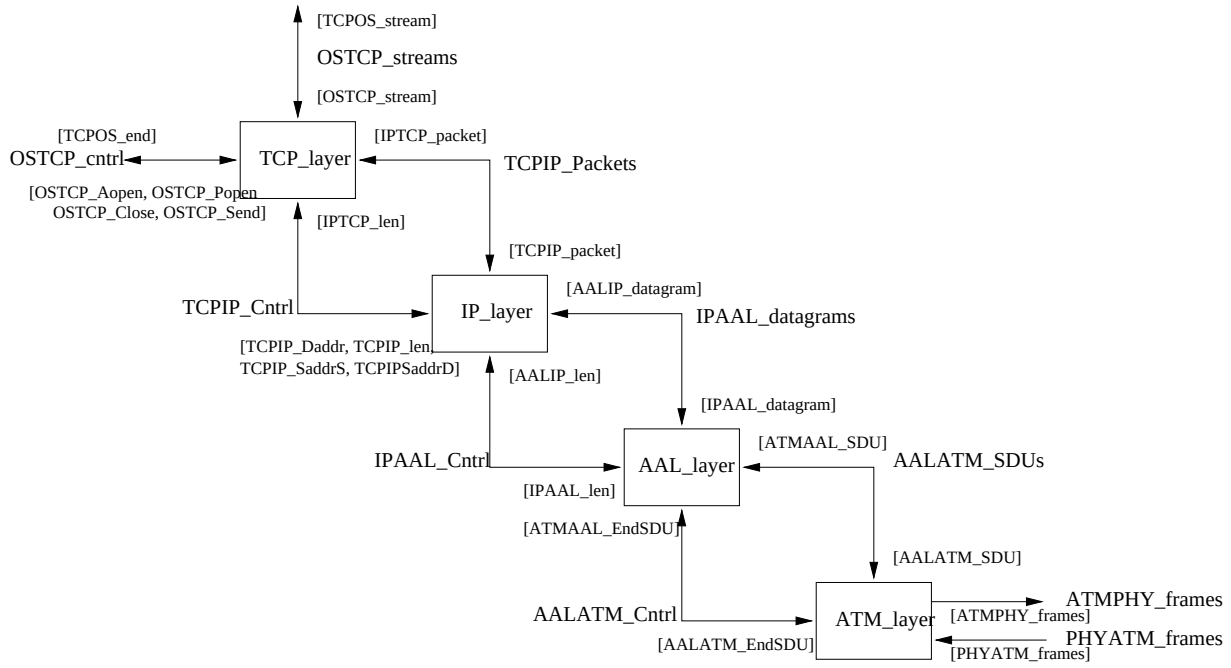


FIG. 6.15: Structure de la description SDL de l'ATM

Les principales difficultées rencontrées ont concerné la représentation des trames et leur segmentation. SDL étant un langage fortement typé, il n'a pas été possible de définir des types pour chaque paquet (paquet TCP, IP, AAL, ATM), la segmentation d'un paquet en paquets de la couche inférieure n'étant pas possible. Une possibilité aurait été de définir des types de données abstraits (ADT) pour chaque paquet et des opérateurs permettant de découper et d'assembler ces paquets. Les ADT nétant pas supportés par le traducteur SDL/SOLAR, il a été décidé de coder tous les champs sous forme d'entier. Cette solution aurait nécessité un traducteur C/SOLAR qui n'est que partiellement implémenté. Un paquet est donc représenté par un tableau d'entiers qui peut être

facilement découpé et réassemblé par les différentes couches. Cette solution a l'inconvénient de représenter des champs de taille et de type différents avec un type unique. Certains entiers représenterons en fait des octets, des demi-mots ou des mots de 32 bits. Néanmoins cette représentation unique des éléments d'un paquet permet une manipulation aisée.

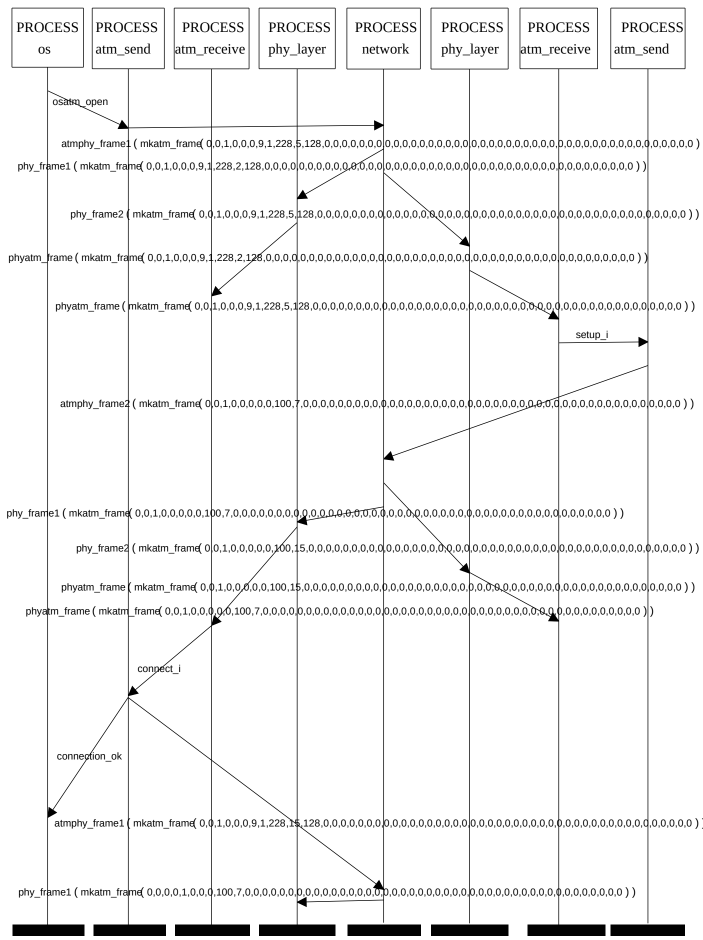


FIG. 6.16: Simulation fonctionnelle de l'ATM en SDL

Les figures 6.16 et 6.17 représentent des simulations SDL de l'ATM. La figure 6.16 représente les trames échangées lors de l'ouverture d'une connection ATM entre deux processus et le réseau. La partie supérieure de la figure 6.17 représente un échange d'une trame TCP et l'arrivée de la trame d'acquittement. On peut y voir la segmentation de la trame TCP/IP par le processus SAR. La partie inférieure représente la fermeture de la connection conformèment à la figure 6.4.2.

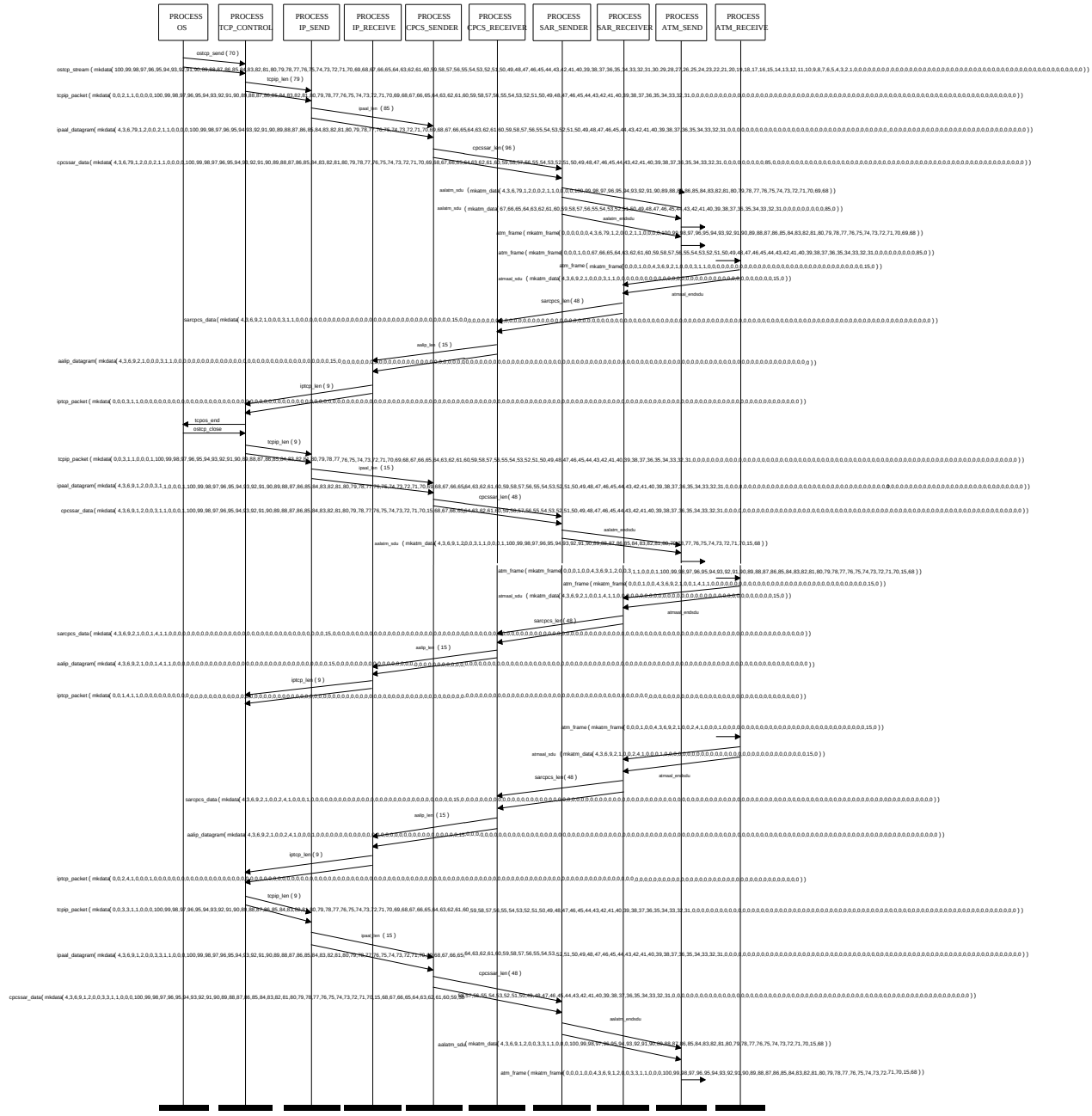


FIG. 6.17: Simulation fonctionnelle de l'ATM en SDL

6.6.2 Transformation du modèle SDL en une architecture C/VHDL

La première étape de la synthèse concerne la traduction de la spécification SDL en SOLAR. Les principaux concepts de cette étape ont été détaillés au chapitre 5.

La première étape de la synthèse concerne l'allocation des processeurs logiciels et matériels. Plusieurs alternatives de découpage sont possibles en fonction de performances requises (figure 6.18). 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 [95]. La figure 6.18 représente les alternatives de partitionnement et les performances estimées pour chacune d'elle.

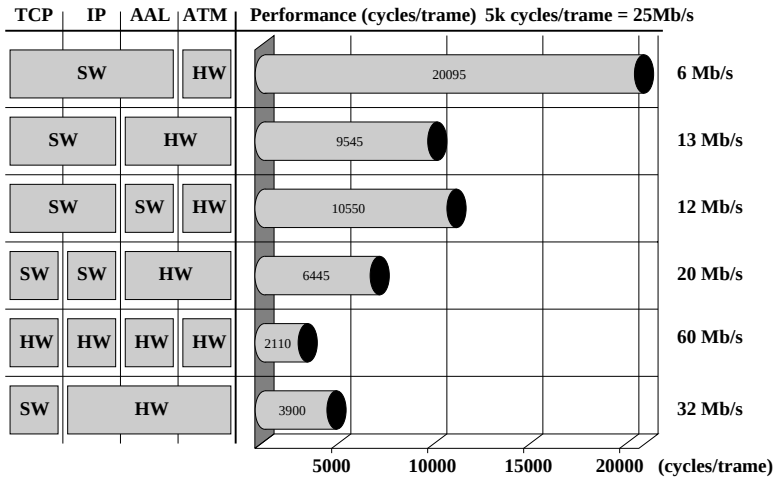


FIG. 6.18: Alternatives de partitionnement logiciel/matériel de l'ATM

Deux alternatives d'affectation logiciel/matériel ont été effectuées :

- une réalisation totalement matérielle. Chaque couche de l'ATM est affectée à un processeur matériel. Cette solution permet des performances élevées ($\simeq$60Mb/s) mais a le coût le plus élevé.
- une réalisation mixte dans laquelle les couches TCP-IP sont logicielles et les couches AAL-ATM matérielles. Chaque couche logicielle est affectée à un processeur distinct. Dans cette alternative de partitionnement il est possible de réduire le coût du matériel en affectant les couches AAL et ATM au même processeur matériel dans la mesure où les performances ($\simeq$20Mb/s) sont limitées par la réalisation de la couche IP.

La bibliothèque de communication a été étendue afin de supporter les schémas de communication requis par les différents processus SDL. Il n'a pas été nécessaire d'ajouter de nouvelles unités de communication dans la bibliothèque. 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*). Les ajouts de primitives de communication ont concerné les données suivantes :

- une trame TCP ou IP.
- une charge de cellule ATM.
- une cellule ATM.

6.6.3 Evaluation de SDL pour la synthèse système

La table 6.2 résume les principaux concepts utilisés pour la spécification au niveau système et la capacité de SDL à les manipuler.

calcul	flôt de donnée	flôt de contrôle	communication	comportement
peu d'opérateurs arithmétiques, fortement typé, types de données abstraits (C)	non	trois instructions (*decision*, *join*, *label*), pas d'instructions de haut niveau (*for*, *while*)	passage de messages, communication de haut niveau (*input*, *output*), 1 seule file d'attente par processus, 1 seul protocole (*fifo*)	EFSM transitions déclenchées par les signaux uniquement + signaux continus, parallélisme limité au processus, 1 fsm par processus, pas de hiérarchie comportementale

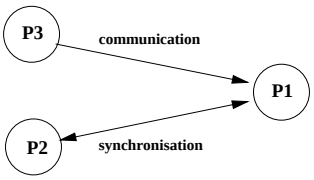
TAB. 6.2: Evaluation de SDL pour la synthèse système

SDL est adapté à la description de protocoles de communication sous forme de processus échangeant des messages. La description de flôt 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*. SDL n'est pas du tout adapté à la manipulation de données malgré l'existance des types de données abstraits. Le langage est fortement typé et supporte mal les conversions de types. Il ne supporte pas aisement les champs de bits, l'affectation d'un bit particulier et la concaténation ou la possibilité de voir un champ de bits comme un entier. Ce point particulier a posé de nombreux problèmes lors de la spécification de l'ATM.

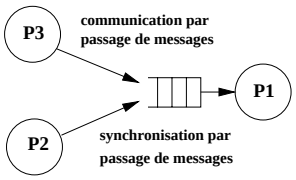
Il n'est pas possible de spécifier le parallélisme à un niveau autre que le processus, ce qui peut amener à augmenter le nombre de processus et les communications inter-processus. Un processus ne peut posséder qu'une file d'attente. Si plusieurs processus communiquent avec un même processus, ils devront tous utiliser le même schéma de communication, ce qui peut ne pas être optimal. Ce cas est représenté en figure 6.19. Les processus *P1* et *P3* communiquent par passage de messages alors que les processus *P1* et *P2* se synchronisent sans échanger de données. Lors de la réalisation de cette communication, la synchronisation entre *P1* et *P2* sera effectuée par passage de message.

6.6.4 Résultats

Le tableau 6.3 résume les principaux résultats de la co-synthèse de l'ATM en terme de lignes de code et de temps de simulation. Comme il a été vu au chapitre 5, l'augmentation de la taille du code entre une spécification SDL et sa réalisation C/VHDL est d'environ huit fois. Cette valeur peut varier selon les protocoles de communication utilisés pour implémenter les communications.



6.19.1: Communication et synchronisation inter-processus



6.19.2: Synchronisation par passage de message

FIG. 6.19: Limitation de la communication en SDL

langage	lignes		temps de simulation	augmentation
	comportement	communication		
SDL	794	103	1 min	
VHDL	7210	5382	30 min	908 %
C/VHDL	$\simeq$	$\simeq$	180 min	$\simeq$

TAB. 6.3: Résultat de synthèse de l'ATM

Dans le cas de l'ATM, la communication entre les différents blocs implique le transfert de grandes quantités de données et les protocoles de communication sont plus complexes.

La figure 6.20 représente une trace de simulation de deux ATM. Les états des différents automates depuis l'ouverture jusqu'à la fermeture de la connection sont représentés. la figure 6.21 represente un détail de la figure précedente.

La figure 6.22 représente la réalisation de la communication du processus *sar_sender*. Celui-ci lit les messages dans sa file d'attente, génère des segments de 48 octets et les écrit dans le canal de processus *atm_send*. La figure 6.23 représente le protocole d'échange et les données transférées entre les processus au niveau du cycle d'horloge correspondant à la figure 6.22. Le protocole d'échange implémenté est le protocole *first-in-first-out*.

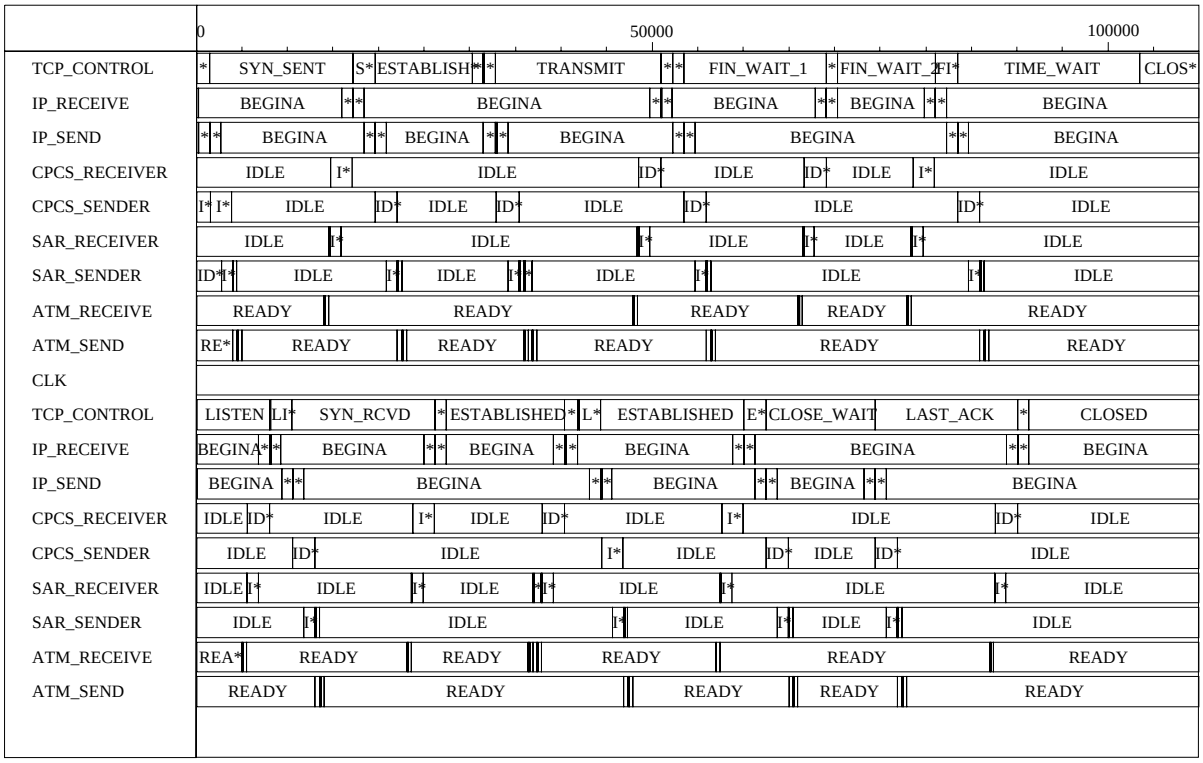


FIG. 6.20: Simulation VHDL de l'ATM

6.7 CONCLUSION

Dans ce chapitre nous avons appliqué notre méthodologie de co-synthèse à un exemple complet du domaine des télécommunications. A partir d'un spécification abstraite de haut niveau, une réalisation C/VHDL a été obtenue. Cette réalisation a été co-simulée à l'aide de simulateur VHDL et de débogueur C du commerce. Cette application démontre qu'il est possible d'obtenir une réalisation C/VHDL simulable et synthétisable à partir d'une spécification système en SDL. Néanmoins 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 matériel est difficile en SDL. La section 6.6.3 a mis en évidence des faiblesses de SDL qui peuvent résulter en des réalisations moins efficaces.

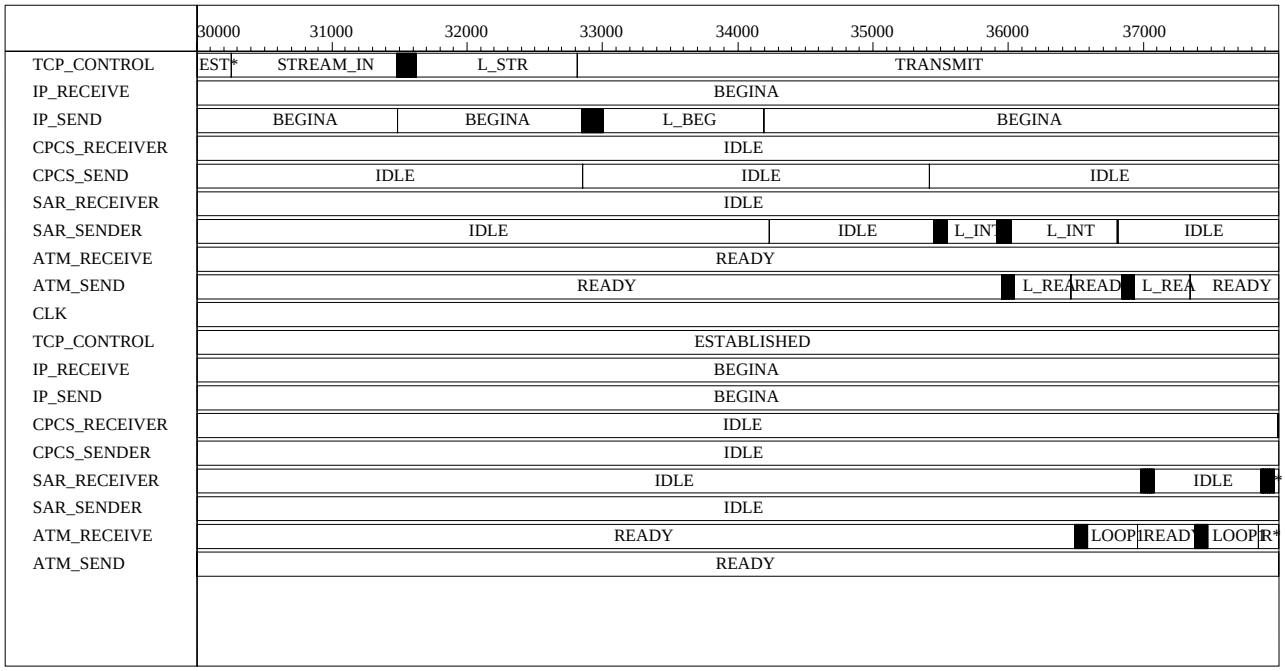


FIG. 6.21: Détail de la simulation VHDL de l'ATM

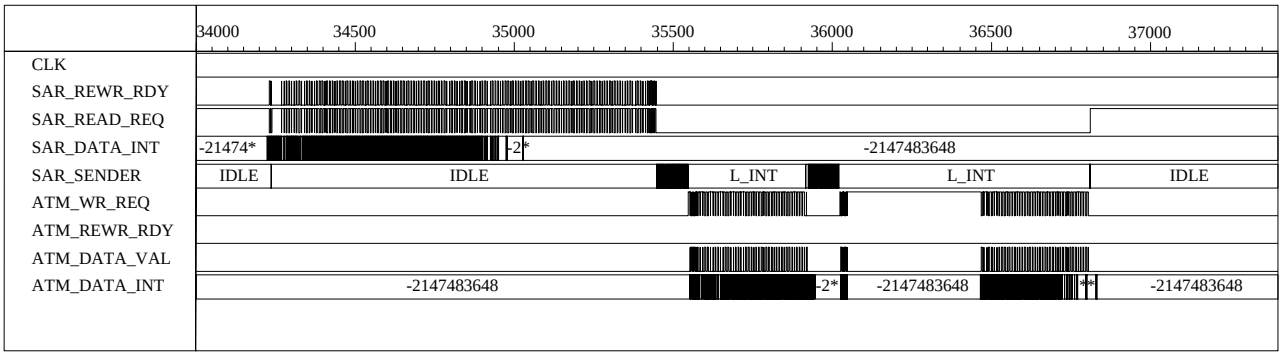


FIG. 6.22: Réalisation de la communication SDL en VHDL

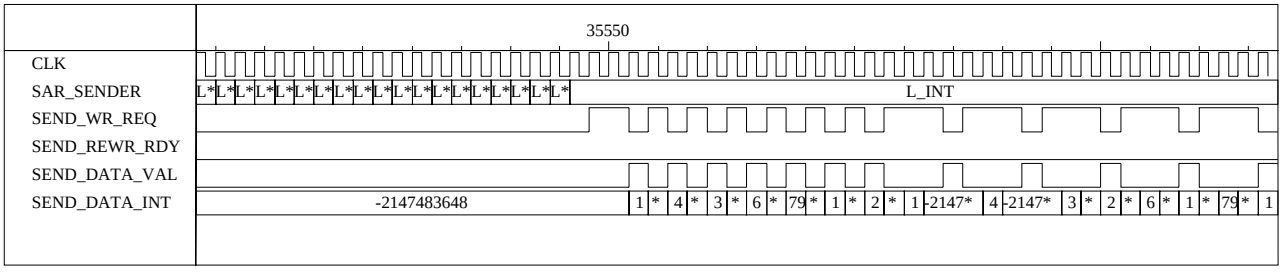


FIG. 6.23: Protocole de communication entre les processus *sar_sender* et *atm_sender*

Chapitre 7

CONCLUSION ET PERSPECTIVES

7.1 CONCLUSION

Dans cette thèse nous avons abordé la synthèse logiciel/matériel à partir de spécifications systèmes. Différents langages peuvent être utilisés pour la spécification système. Nous nous sommes concentrés sur les langages du domaine des télécommunications : SDL, ESTELLE et LOTOS. Ces langages sont dédiés à la spécification et à la vérification de protocoles de communication. SDL étant bien adapté à la description de systèmes distribués communicants il est utilisé comme langage de spécification système dans notre approche.

Le niveau d'abstraction de ce langage est beaucoup plus élevé que celui des langages de description matériel, en particulier en ce qui concerne la communication. Une spécification SDL ne donne aucune indication sur la réalisation de la communication et des différentes fonctionnalités. Nous avons étudié les principales différences entre SDL et VHDL. Cette étude montre qu'une traduction directe de SDL vers VHDL ne permet pas d'obtenir un réalisation efficace. Par contre le haut niveau d'abstraction de ces spécifications peut être réduit par l'application d'étapes de raffinements successifs. La principale étape de raffinement, appelée synthèse de la communication, détermine le protocole et les interfaces utilisés par les différents processus pour communiquer. La modélisation de la synthèse de la communication comme un problème d'allocation nous a permis de proposer une nouvelle solution basée sur une bibliothèque de communication. Cette approche transformationnelle, basée sur des raffinements incrémentaux permet d'obtenir une réalisation efficace à partir d'une spécification fonctionnelle abstraite.

Nous avons proposé une modélisation pour la synthèse du schéma de communication SDL. Cette modélisation repose sur le canal SOLAR (*channel unit*). Cette forme intermédiaire supporte un modèle de communication général qui autorise la représentation pour la synthèse de la plupart des schémas de communication systèmes et matériels. Une réalisation adaptée aux performances requises peut être sélectionné dans une bibliothèque d'unités de communication.

L'utilisation d'une forme intermédiaire sur laquelle sont appliquées les étapes de raffinement permet de s'abstraire du langage de spécification. Les étapes de raffinement sont effectuées sur la forme intermédiaire. Ainsi l'approche proposée dans cette thèse est indépendante du langage de spécification et peut être appliquée à d'autres langages. Le modèle de communication de SOLAR est suffisamment puissant pour représenter les modèles de communication de ESTELLE ou LOTOS qui sont les deux autres langages dediés aus systèmes de télécommunication que nous avons présentés.

Un large sous-ensemble de SDL pour la synthèse est supporté, c'est à dire tous les concepts qui ont une représentation matérielle. La communication SDL peut être traduite en SDL pour la synthèse.

Le chapitre 2 a permis de présenter notre approche de la synthèse système. La forme intermédiaire SOLAR a été présentée. Ce modèle sert de représentation intermédiaire sur laquelle la co-synthèse est réalisée. Le modèle de représentation SOLAR repose sur les machines d'états finis avec une communication de haut niveau. Il autorise la modélisation pour la synthèse la majorité des schémas de communication de haut niveau. Les différentes étapes de la co-synthèse avec COSMOS ont été présentées. Dans notre approche transformationnelle, la co-synthèse est réalisée par raffinements successifs où chaque étape fixe certains détails de réalisation ou prépare une autre étape de raffinement.

Le chapitre 3 a présenté les principaux concepts et familles de langages utilisés pour la spécification au niveau système. Les langages de spécifications peuvent être classés par domaine d'application (systèmes temps réel, traitement du signal, télécommunications) et sont en général bien adaptés à un domaine particulier. COSMOS propose une approche multi-langage du co-design. Les différentes parties d'un système peuvent ainsi être décrites dans le langage le plus approprié. Elles sont ensuite unifiées dans la forme intermédiaire SOLAR pour y subir les étapes de co-synthèse. Trois langages issus du domaine des télécommunications ont été presentés. Le langage SDL est un langage de spécification dédié aux systèmes de télécommunications. Il permet la spécification et la validation de protocoles de communication ou de systèmes distribués. Ce langage est utilisé comme langage de spécification système dans COSMOS. Il a été présenté en détail dans ce chapitre.

Le chapitre 4 a détaillé notre approche de la synthèse de la communication. Cette approche vise à réduire le fossé entre les concepts utilisés dans les communications au niveau système et ceux utilisés pour la réalisation. Elle permet de transformer un système communicant par des schémas de haut niveau en un ensemble de processeurs interconnectés et communicant à travers des protocoles définis. Une nouvelle approche interactive et basée sur des bibliothèques de communication, a été développée. Elle permet une réalisation efficace des schémas de communication systèmes.

Dans le chapitre 5 nous avons étudié la mise en correspondance des concepts SDL et SOLAR pour la génération de code C/VHDL. Les principaux problèmes de la mise en correspondance ont été identifiés. Le principal problème posé lors de cette étape est le haut niveau d'abstraction des communications en SDL. Nous avons proposé une modélisation de la communication SDL en SOLAR compatible avec notre approche de la synthèse de la communication. Elle permet la traduction de la communication SDL en VHDL pour la synthèse.

Le dernier chapitre a présenté une application de la méthodologie présentée à un exemple du domaine des télécommunications. Le protocole TCP/IP sur ATM a été spécifié en SDL pour être co-synthétisé dans COSMOS. Le code C/VHDL obtenu a pu être co-simulé à l'aide d'outil standard. Ce chapitre se termine par une évaluation de SDL comme langage de spécification système.

Dans cette thèse, nous avons montré comment le langage SDL peut être utilisé pour la spécification et la synthèse système. L'étape de synthèse de la communication permet d'obtenir une synthèse efficace à partir de spécifications système abstraites. Cette approche ne dépend pas du langage de spécification et peut être appliquée à d'autres langages. A travers un exemple d'application, nous avons démontré la validité de l'approche, en particulier en ce qui concerne la qualité du code produit.

7.2 PERSPECTIVES

Ce travail de thèse a concerné la co-synthèse à partir de spécifications SDL, néanmoins d'autres langages sont envisageables, surtout au vu de l'évaluation de SDL faite au chapitre 6. L'approche développée pour la synthèse de la communication restera valable, pourvu qu'une modélisation adéquate du schéma de communication du langage utilisé soit trouvée. Le langage C semble un bon candidat, d'autant plus qu'il est utilisé en SDL pour la description des opérateurs des types de données abstraits. Parmi les extensions possibles du travail réalisé concernant SDL, le sous-ensemble supporté peut être étendu notamment :

– aux types de données abstraits

– aux appels de procédure à distance (SDL92)

L'ajout des types de données abstraits au sous-ensemble supporté nécessite le développement d'un traducteur C→SOLAR. Cela permettra d'avoir un second langage de spécification dans COSMOS.

La synthèse d'interface peut être optimisée pour réduire le nombre d'interconnections. Il serait intéressant de développer des estimateurs au niveau système. Ceux ci pourraient guider le concepteur lors de l'étape de synthèse d'interface. En effet cette étape détermine des paramètres tels que la taille des bus en fonction des performances requises.

De nombreux travaux peuvent être réalisés autour de l'exemple de l'ATM : d'autres alternatives de partitionnement peuvent être explorées, des estimateurs au niveau système peuvent être développés. L'exemple de l'ATM peut encore être étendu pour offrir plus de fonctionnalités telles que :

– le désordonnancement des trames TCP.

– un système de fenêtre glissante pour la réception des trames TCP.

– l'utilisation les types de données abstraits pour implémenter la correction d'erreur.

– l'implémentation du protocole *ARP* (*adress resolution protocol*) au niveau IP.

– la gestion de flux aux niveaux ATM (*CBR*, *VBR*, *ABR*).

– l'extension des fonctions de gestion du réseau (flux *OAM*) au niveau ATM.

Annexe A

Description SDL de l'ATM

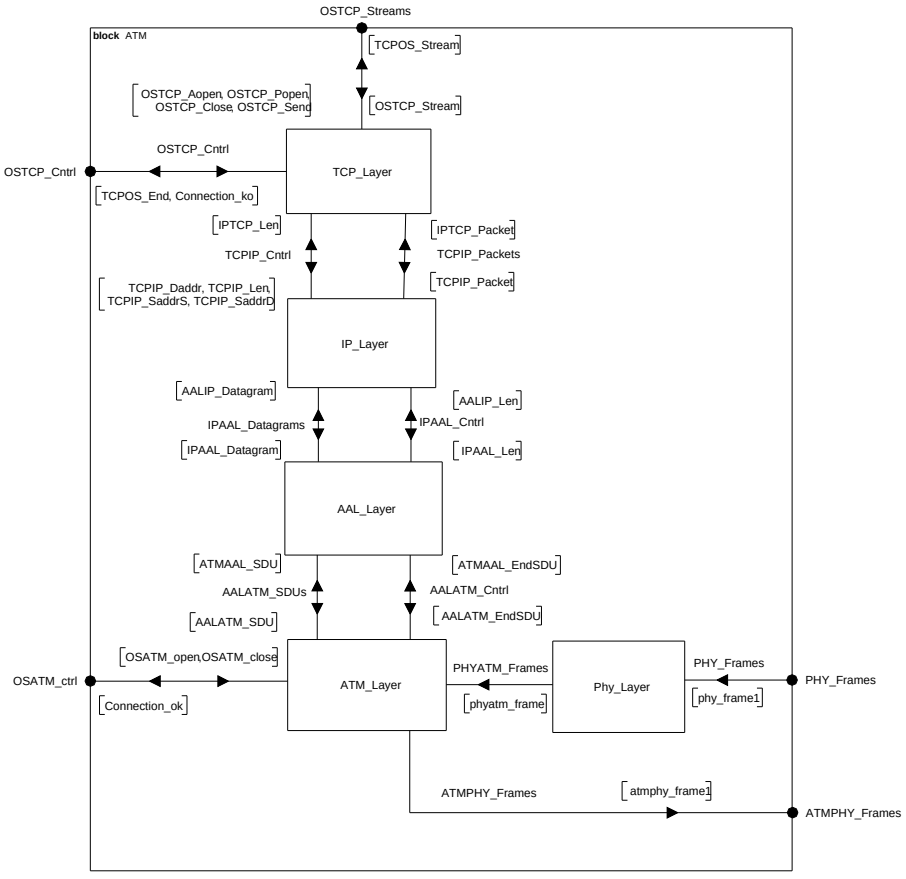


FIG. A.1: Modèle général de l'ATM

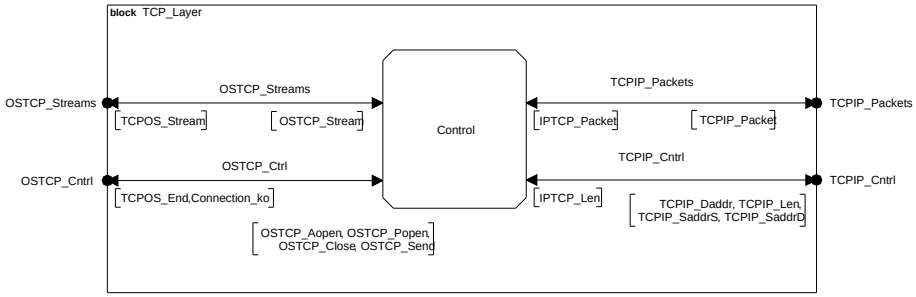


FIG. A.2: Structure de la couche TCP

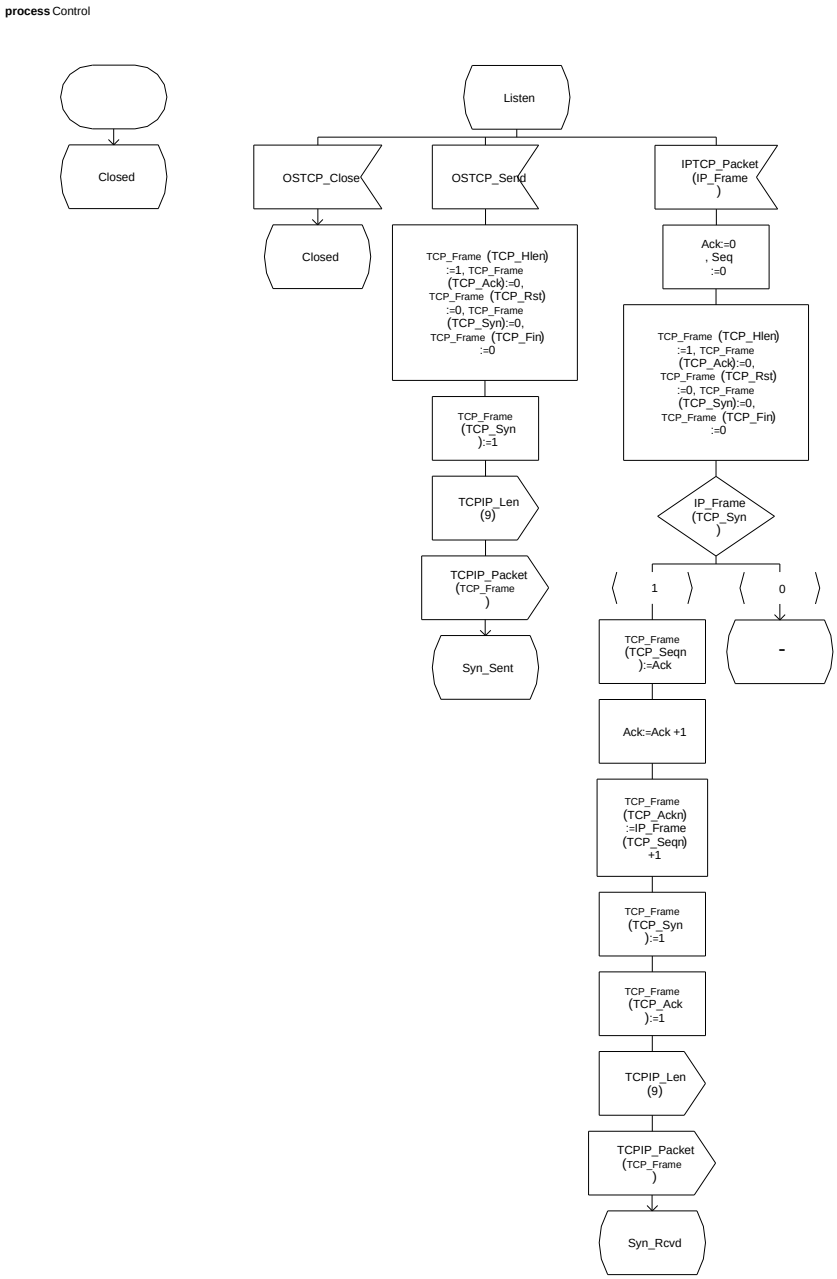


FIG. A.3: Automate de la couche TCP

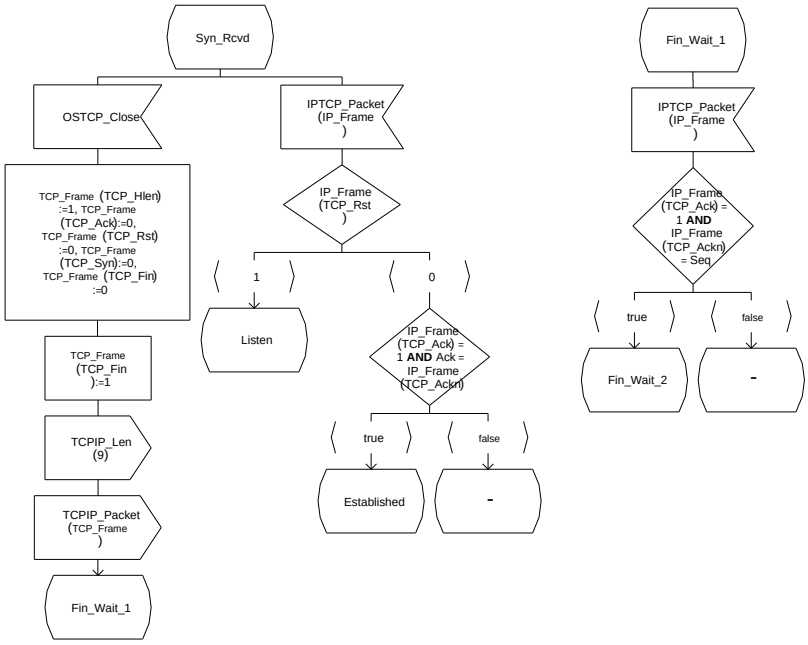


FIG. A.4: Automate de la couche TCP

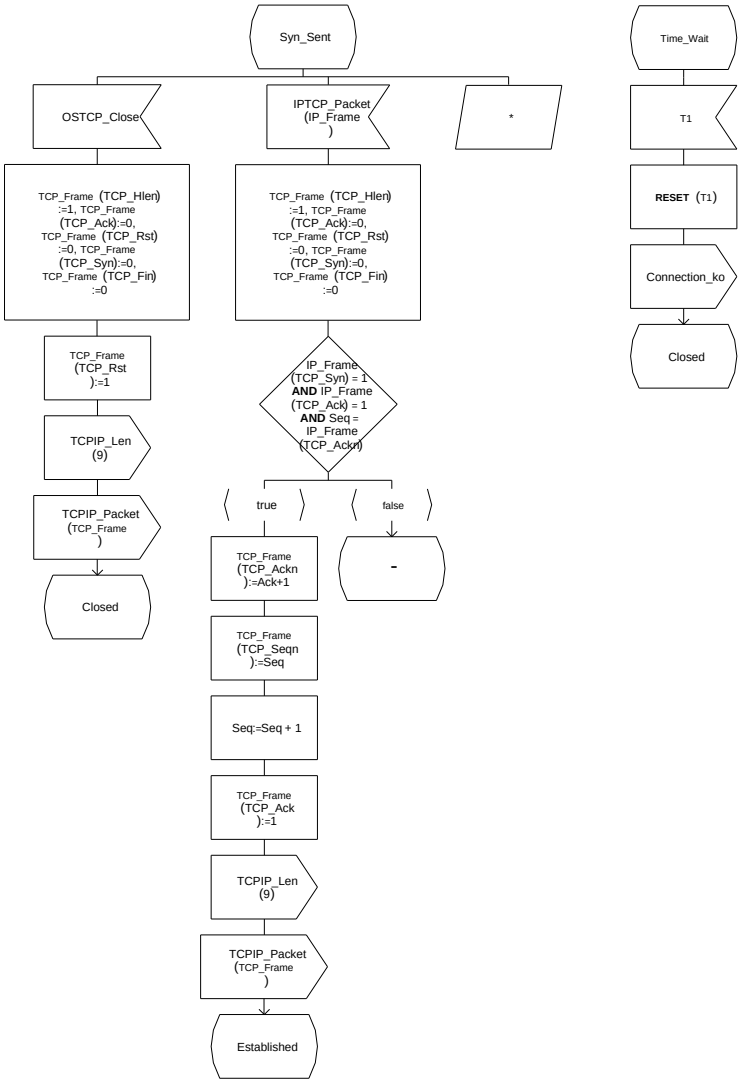


FIG. A.5: Automate de la couche TCP

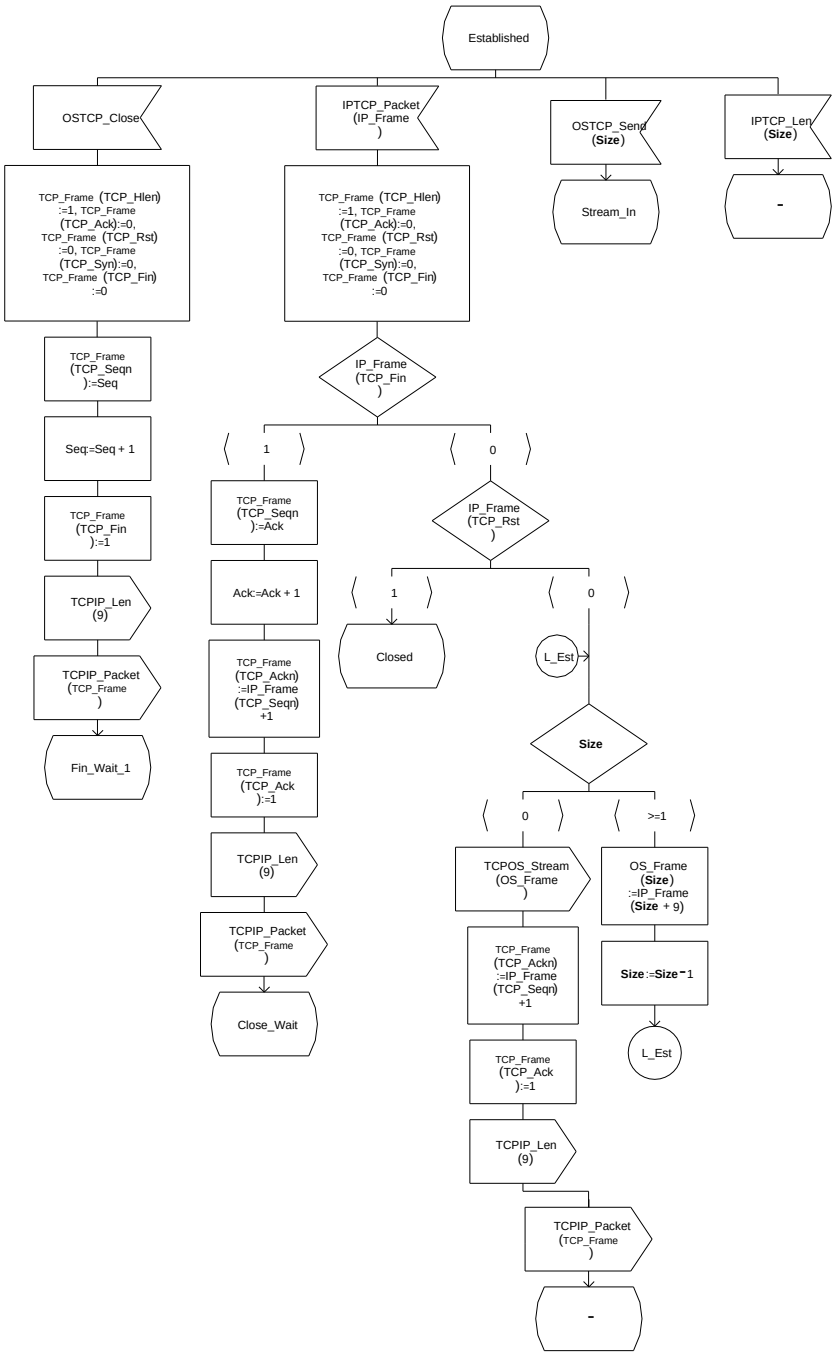


FIG. A.6: Automate de la couche TCP

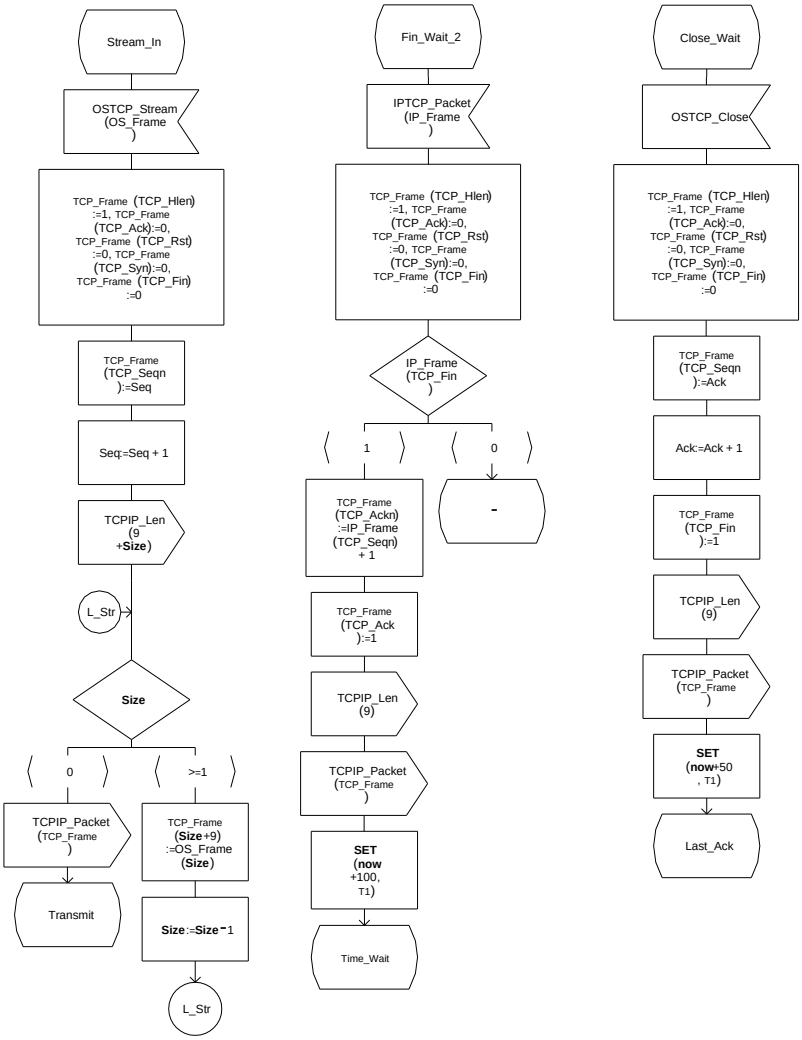


FIG. A.7: Automate de la couche TCP

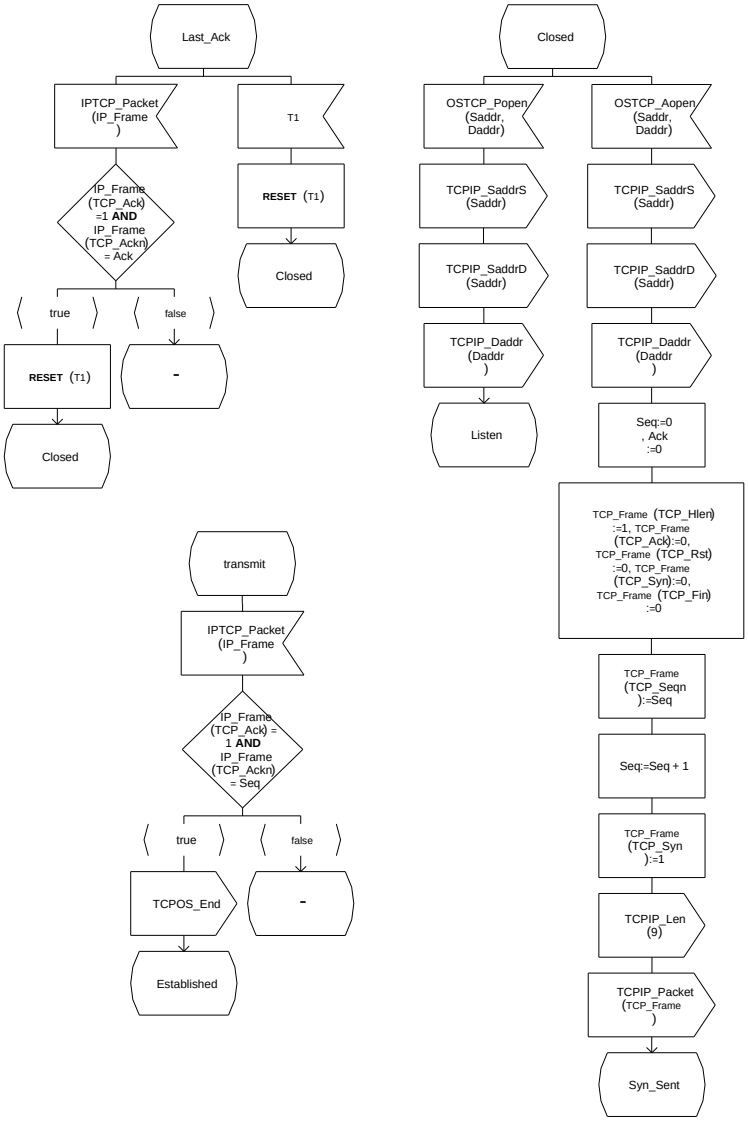


FIG. A.8: Automate de la couche TCP

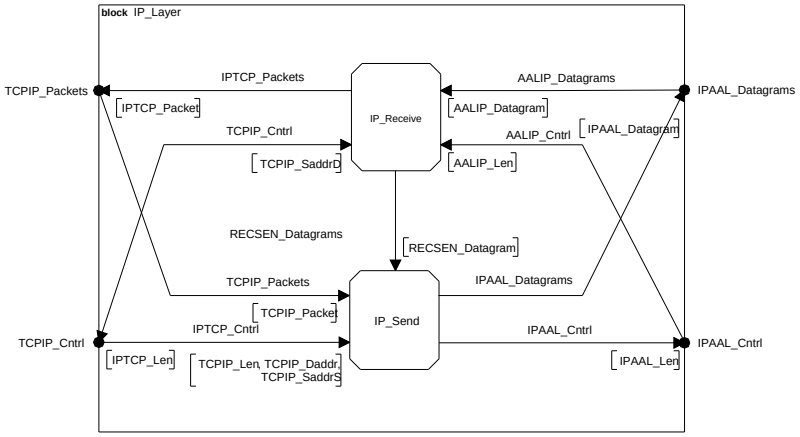


FIG. A.9: Structure de la couche IP

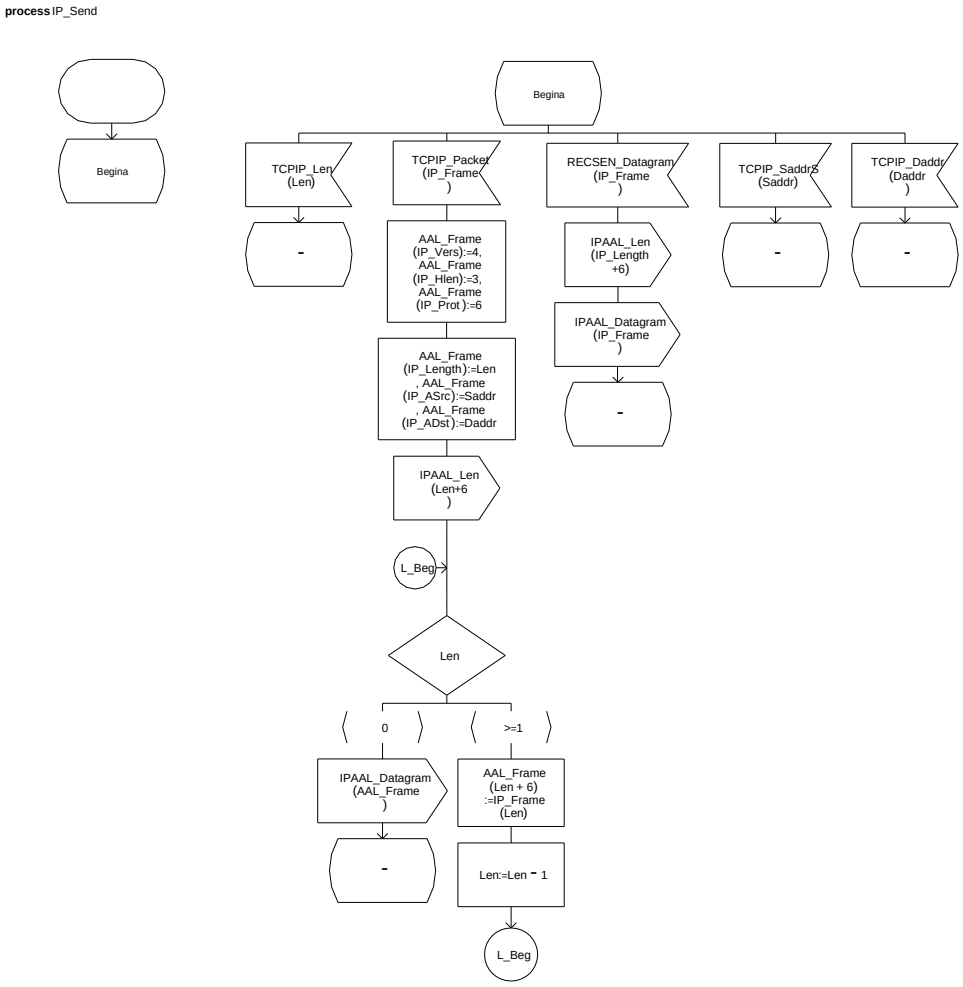


FIG. A.10: Automate d'émission la couche IP

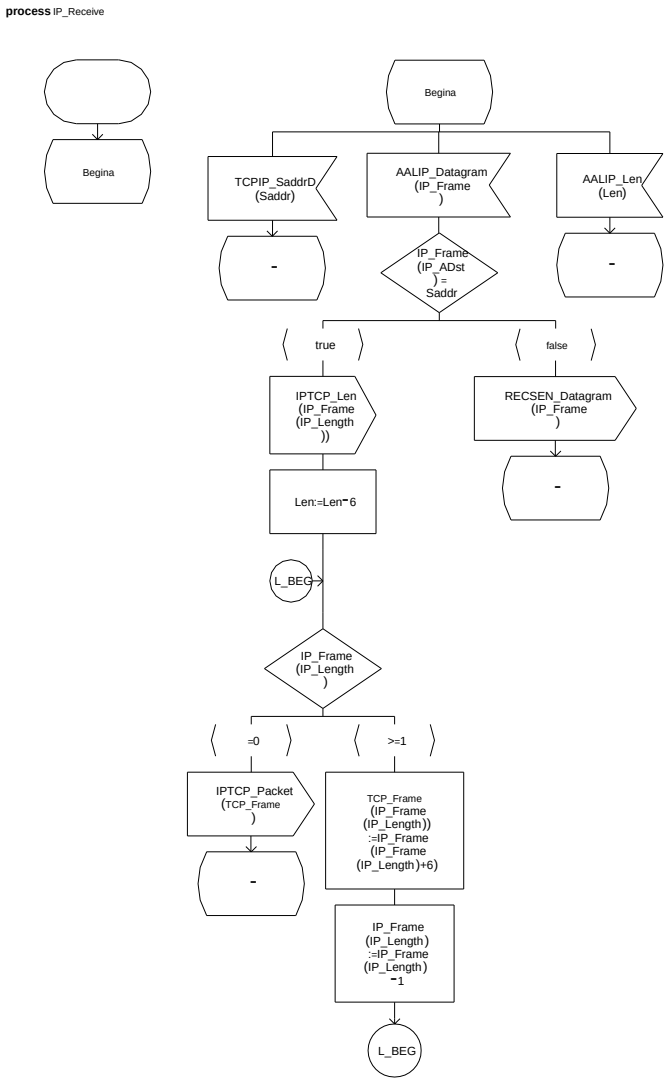


FIG. A.11: Automate de réception la couche IP

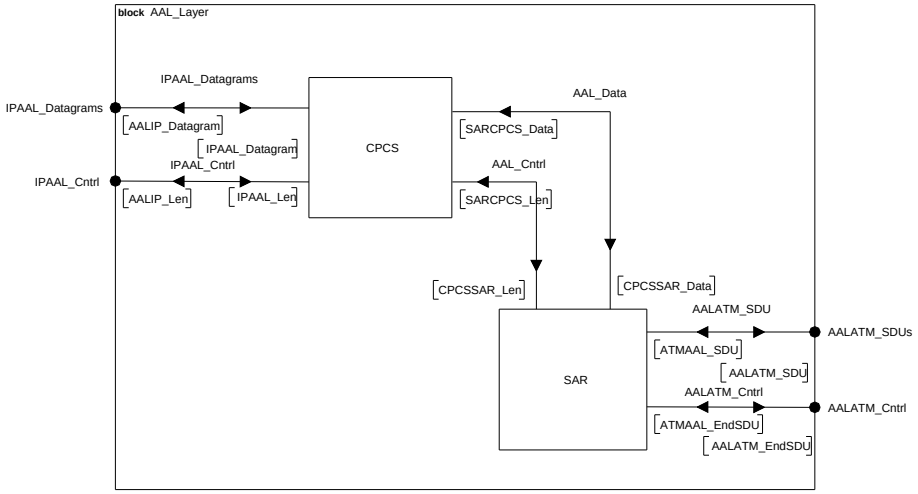


FIG. A.12: Structure de la couche AAL

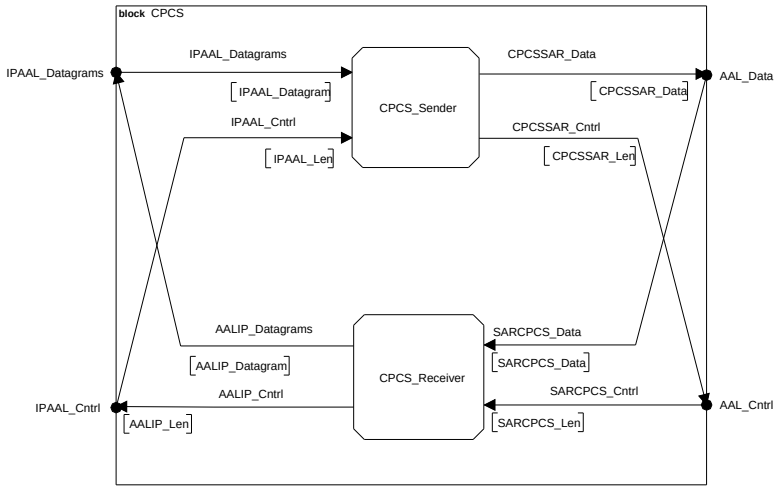


FIG. A.13: Structure de la couche CPCS

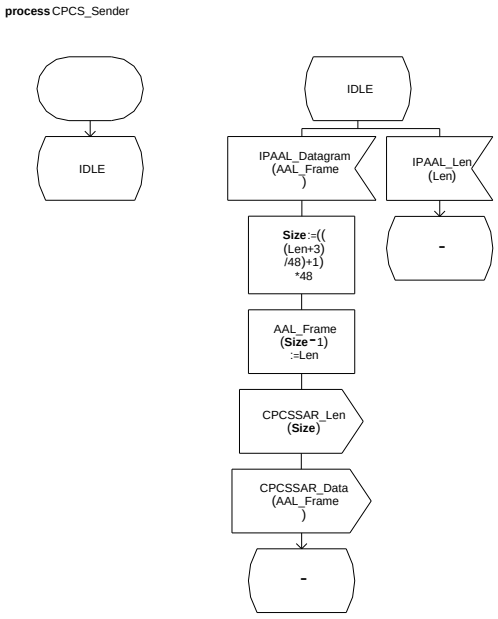


FIG. A.14: Automate d'émission la couche CPCS

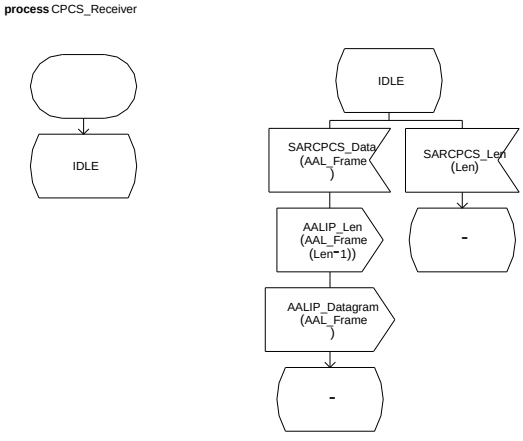


FIG. A.15: Automate de réception la couche CPCS

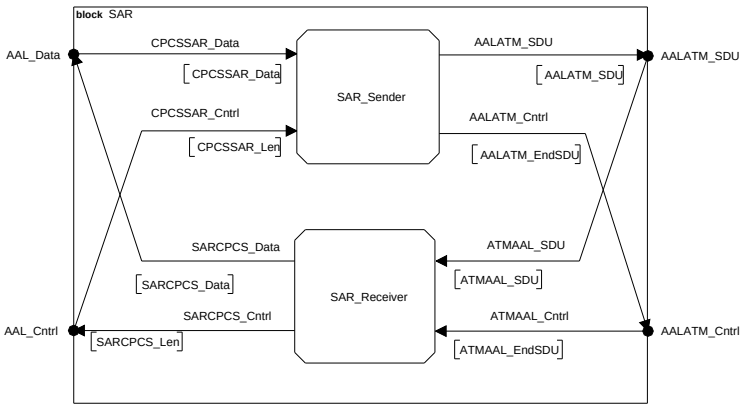


FIG. A.16: Structure de la couche SAR

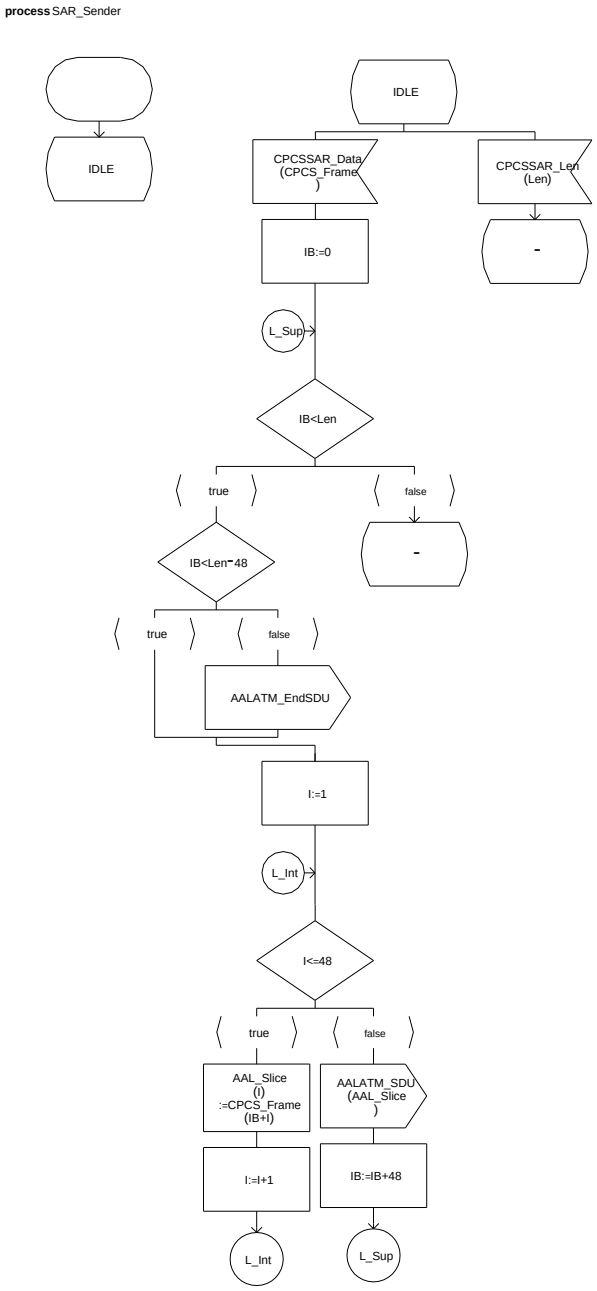


FIG. A.17: Automate d'émission la couche SAR

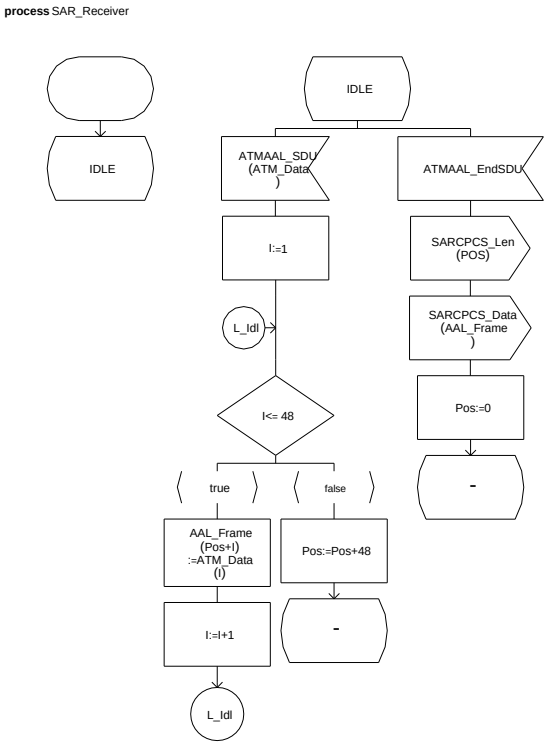


FIG. A.18: Automate de réception la couche SAR

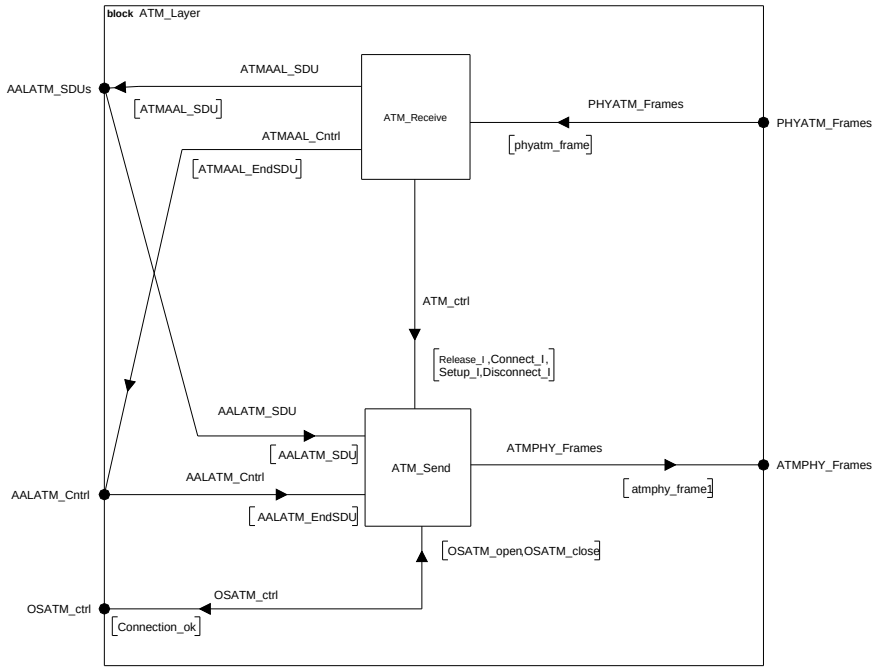


FIG. A.19: Structure de la couche ATM

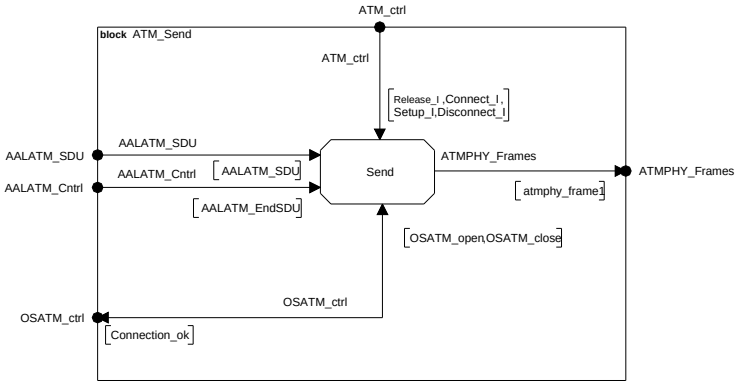


FIG. A.20: Structure du bloc d'émission de la couche ATM

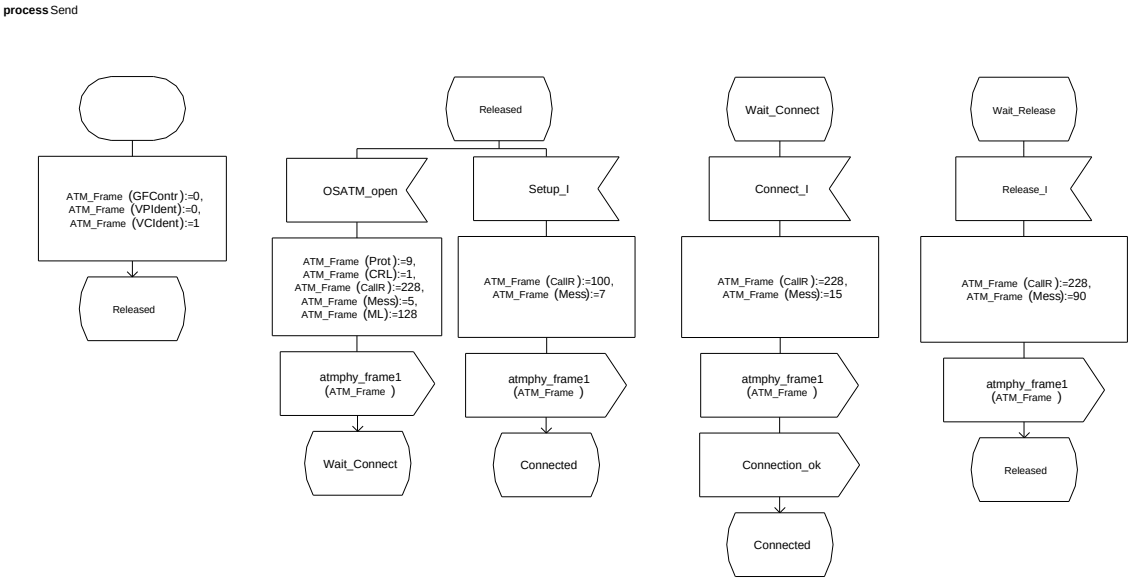


FIG. A.21: Automate d'émission la couche ATM

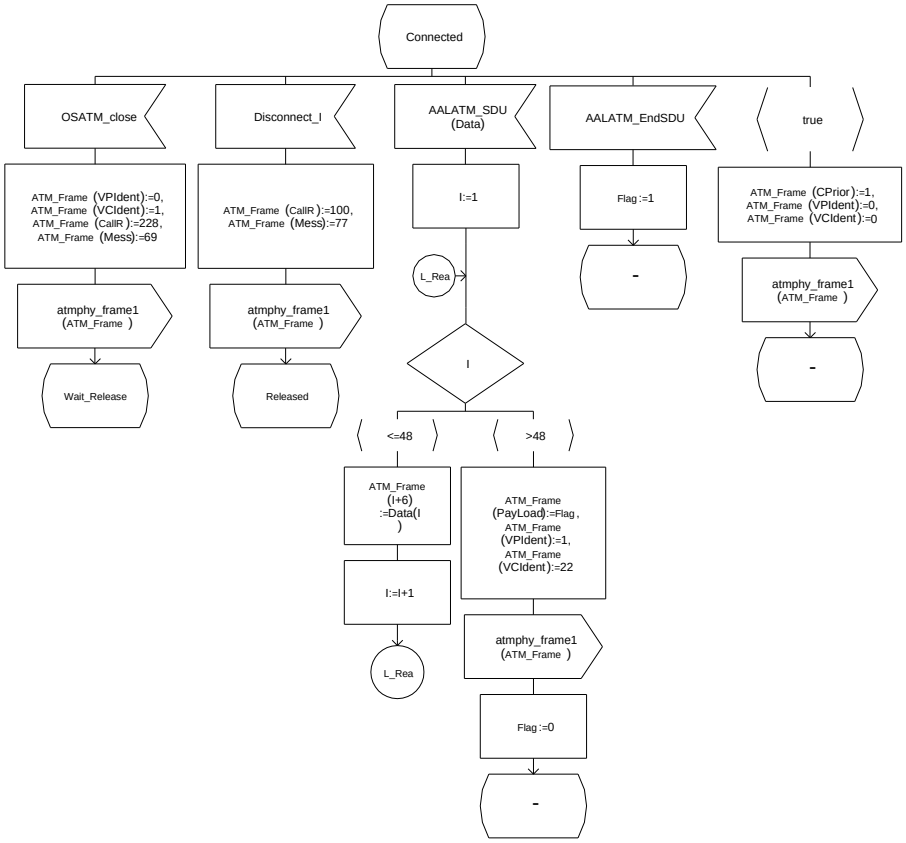


FIG. A.22: Automate d'émission la couche ATM

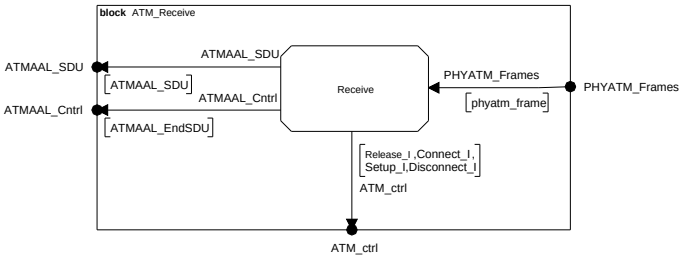


FIG. A.23: Structure du bloc de réception de la couche ATM

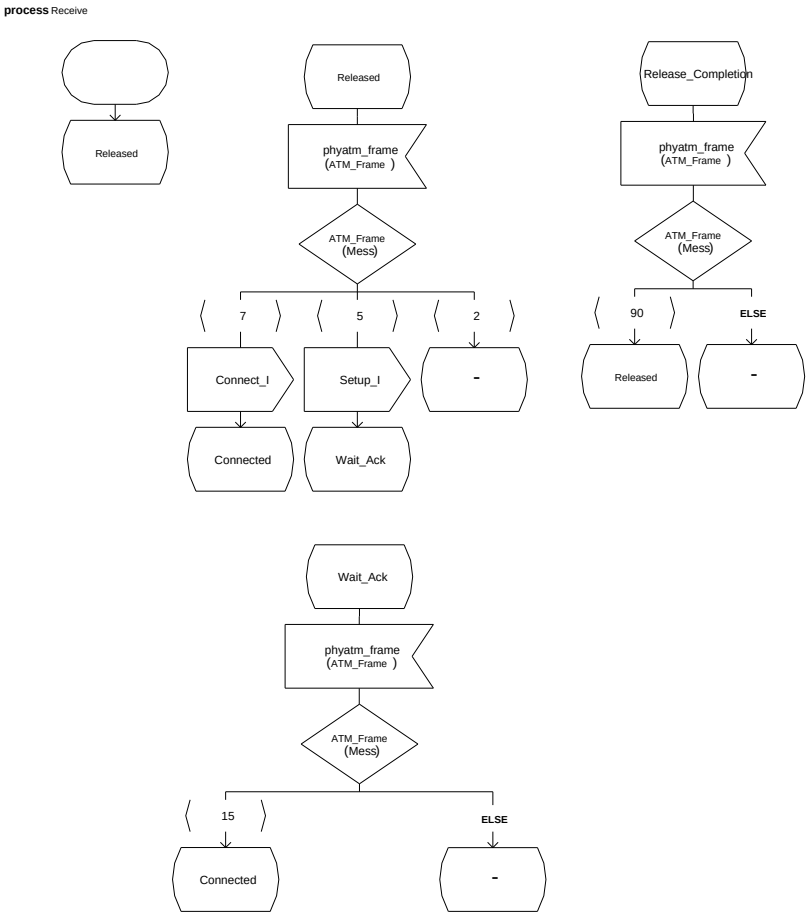


FIG. A.24: Automate de réception la couche ATM

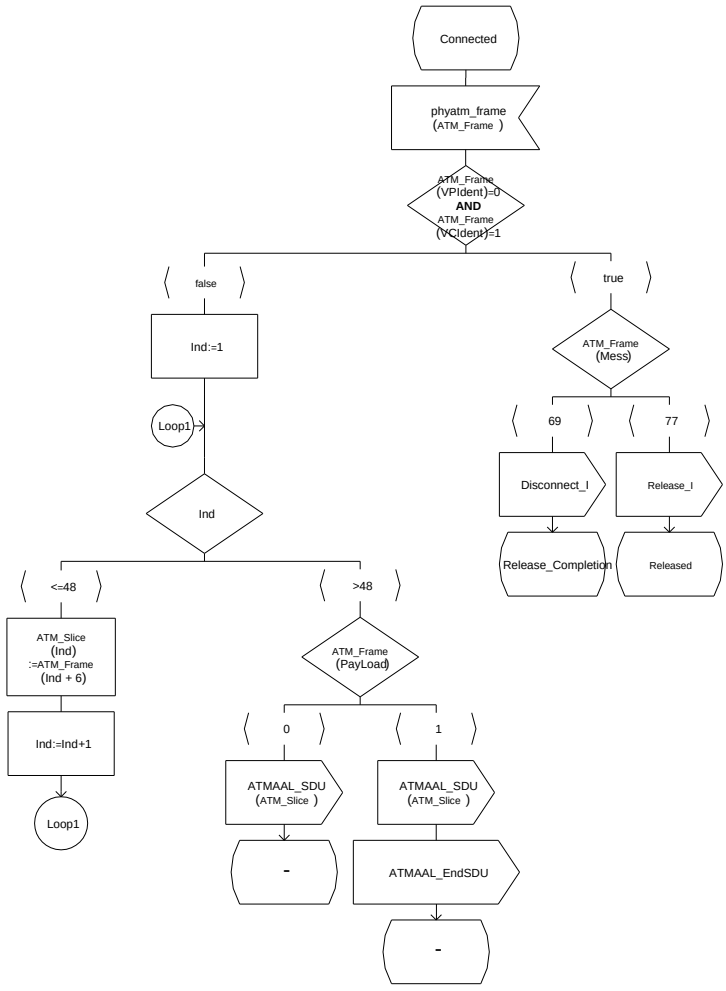


FIG. A.25: Automate de réception la couche ATM

Annexe B

Primitive ALLOC

B.1 Syntaxe

La primitive *alloc* permet d'effectuer l'allocation manuelle d'unités de communication. La syntaxe d'appel est la suivante :

```
alloc [fichier.solar] [channel unit] [net]+
```

La primitives *alloc* effectue deux opérations :

- la fusion des différents canaux concernés.
- La vérification que le canal alloué est capable d'effectuer toutes les primitives de communication requises.

B.1.1 Fusion de canaux

Lorsqu'une unité de communication est allouée pour exécuter les communications offertes par plusieurs canaux abstrait, ceux ci sont préalablement fusionnés. L'opération de fusion effectue les transformations suivantes :

- la fusion des *net* et *access* concernés.
- la génération de constantes permettant d'identifier les sous-canaux. Chaque appel à une primitives de communication d'un des canaux (*cucall*) se voit ajouter un paramètre permettant d'adresser le sous-canal désiré. Ces paramètres et constantes sont de type *pid* qui est définit comme un sous-type du type entier.

A ce paramètre sera ajouté une propriéte [1] *subchannel static* indiquant que ce paramètre sert à identifier le sous-canal accédé. La propriété *subchannel dynamic* est utilisée lorsque la valeur du paramètre n'est pas constante et peut varier à l'exécution. Cela est le cas lorsque le paramètre est généré à l'aide du constructeur *to* de SDL.

[1] *property* SOLAR

B.1.2 Mise en correspondance des paramètres formels et effectifs

La seconde opération effectuée par *alloc* est la mise en correspondance des paramètres formels, indiqués dans les *methods* du *channel unit* alloué, et effectifs (*cucall* de la description). Cette opération remplace chaque *cucall* à une primitive de communication abstraite (définie nulle part) par un *cucall* à une *method* offerte par le *channel unit* alloué. Cette mise en correspondance est basée sur :

1. la correspondance des types et de la direction entre paramètres formels et effectifs.
2. l'ordre d'apparition des paramètres dans la *method*. Un paramètre d'appels du *cucall* est mis en correspondance avec le premier paramètre formel de la *method* du même type lorsque plusieurs correspondance ont été trouvée.

Lorsque plusieurs *method* du *channel unit* peuvent effectuer le *cucall* c'est le concepteur qui déterminera laquelle utiliser. A l'inverse, plusieurs primitives de communication abstraite peuvent être exécutées par une même *method* du *channel unit* alloué.

B.2 Code SOLAR généré

```
(SOLAR  Library_of_Communication_Protocols
   (CHANNELUNIT  DW03fifo
      (VIEW  DW03fifo_behaviour
         (LIBRARY VHDL "IEEE")
         (LIBRARY VHDL "DW03")
         (LIBRARY VHDL "DWARE")
         (LIBRARY VHDL "WORK.package_DW03fifo.ALL")
         (INTERFACE
            (PARAMETER data_width (INTEGER) (INITIALVALUE 32))
            (PARAMETER depth (INTEGER) (INITIALVALUE 128))
            (PARAMETER level (INTEGER) (INITIALVALUE 127))
            (PARAMETER nport (INTEGER) (INITIALVALUE 8))
            (PORT (ARRAY fifin data_width) (DIRECTION IN) (TYPEREF STD_LOGIC))
            (PORT (ARRAY wrqb nport) (DIRECTION IN) (TYPEREF STD_LOGIC))
            (PORT (ARRAY rrqb nport) (DIRECTION IN) (TYPEREF STD_LOGIC))
            (PORT csb (DIRECTION IN) (TYPEREF STD_LOGIC))
            (PORT clk (DIRECTION IN) (TYPEREF STD_LOGIC))
            (PORT reset (DIRECTION IN) (TYPEREF STD_LOGIC))
            (PORT (ARRAY fifout data_width) (DIRECTION OUT) (TYPEREF STD_LOGIC))
            (PORT full (DIRECTION OUT) (TYPEREF STD_LOGIC))
            (PORT empty (DIRECTION OUT) (TYPEREF STD_LOGIC))
            (PORT threshold (DIRECTION OUT) (TYPEREF STD_LOGIC))
            (PORT (ARRAY req nport) (DIRECTION IN) (TYPEREF STD_LOGIC))
            (PORT (ARRAY gnt nport) (DIRECTION OUT) (TYPEREF STD_LOGIC))
            (PORT done (DIRECTION IN) (TYPEREF STD_LOGIC))
            (METHOD  DW03put_int
               (PORT (ARRAY fifin data_width) (DIRECTION OUT)
                  (TYPEREF STD_LOGIC))
               (PORT wrqb (DIRECTION OUT) (TYPEREF STD_LOGIC) )
               (PORT clk (DIRECTION IN) (TYPEREF STD_LOGIC))
               (PORT full (DIRECTION OUT) (TYPEREF STD_LOGIC))
               (PORT req (DIRECTION OUT) (TYPEREF STD_LOGIC)
                  (PROPERTY process private))
               (PORT gnt (DIRECTION IN) (TYPEREF STD_LOGIC)
                  (PROPERTY process private))
               (PORT done (DIRECTION OUT) (TYPEREF STD_LOGIC))
               (PARAMETER  sdl (DIRECTION IN) (INTEGER ))
               (PARAMETER  data (DIRECTION IN) (INTEGER ))
               (PARAMETER   id (DIRECTION IN) (TYPEREF PID )
                  (PROPERTY subchannel static))
               (FOREIGN VHDL "package_DW03fifo")
            )
            (METHOD  DW03get_int
               (PORT rrqb (DIRECTION OUT) (TYPEREF STD_LOGIC)
                  (PROPERTY process private))
               (PORT clk (DIRECTION IN) (TYPEREF STD_LOGIC))
               (PORT (ARRAY fifout data_width) (DIRECTION IN)
                  (TYPEREF STD_LOGIC))
               (PORT empty (DIRECTION IN) (TYPEREF STD_LOGIC))
               (PARAMETER  data (DIRECTION OUT) (INTEGER ))
               (PARAMETER   id (DIRECTION IN) (TYPEREF PID )
                  (PROPERTY subchannel static))
               (FOREIGN VHDL "package_DW03fifo")
            )
            (METHOD  DW03put_real
               (PORT (ARRAY fifin data_width) (DIRECTION OUT)
                  (TYPEREF STD_LOGIC))
               (PORT wrqb (DIRECTION OUT) (TYPEREF STD_LOGIC)
                  (PROPERTY process private))
               (PORT clk (DIRECTION IN) (TYPEREF STD_LOGIC))
               (PORT full (DIRECTION OUT) (TYPEREF STD_LOGIC))
               (PORT req (DIRECTION OUT) (TYPEREF STD_LOGIC)
                  (PROPERTY process private))
               (PORT gnt (DIRECTION IN) (TYPEREF STD_LOGIC)
                  (PROPERTY process private))
               (PORT done (DIRECTION OUT) (TYPEREF STD_LOGIC))
               (PARAMETER  sdl (DIRECTION IN) (INTEGER ))
               (PARAMETER  data (DIRECTION IN) (REAL ))
               (PARAMETER   id (DIRECTION IN) (TYPEREF PID )
                  (PROPERTY subchannel static))
               (FOREIGN VHDL "package_DW03fifo")
            )
            (METHOD  DW03get_real
               (PORT rrqb (DIRECTION OUT) (TYPEREF STD_LOGIC)
                  (PROPERTY process private))
               (PORT clk (DIRECTION IN) (TYPEREF STD_LOGIC))
               (PORT (ARRAY fifout data_width) (DIRECTION IN)
                  (TYPEREF STD_LOGIC))
               (PORT empty (DIRECTION IN) (TYPEREF STD_LOGIC))
               (PARAMETER  data (DIRECTION OUT) (REAL ) )
               (PARAMETER   id (DIRECTION IN) (TYPEREF PID )
                  (PROPERTY subchannel static))
               (FOREIGN VHDL "package_DW03fifo")
            )
         )
         (CONTENTS
            (FOREIGN VHDL "package_DW03fifo")
         )
      )
   )
)
```

FIG. B.1: Bibliothèque SOLAR d'unité de communication

La figure B.3 donne le résultat de l'application de la ligne de commande `alloc controleur.solar DW03fifo N_algo1_pid N_algo2_pid` sur le système représenté figure B.2.

```
(DESIGNUNIT  moteur
(VIEW  moteur_structure
      (VIEWTYPE "structure")
      (CONTENTS
         (INSTANCE  pid_I
            (VIEWREF  pid_structure  pid )
            (ACCESSINSTANCE  software_channel )
            (ACCESSINSTANCE  algo2_channel )
            (ACCESSINSTANCE  algo1_channel )
            (ACCESSINSTANCE  control_channel )
         )
         (INSTANCE  moteur1_I
            (VIEWREF  moteur1_structure  moteur1 )

            (ACCESSINSTANCE  control_channel )
            (ACCESSINSTANCE  algo1_channel )
         )
         (INSTANCE  moteur2_I
            (VIEWREF  moteur2_structure  moteur2 )

            (ACCESSINSTANCE  control_channel )
            (ACCESSINSTANCE  algo2_channel )
         )
         (INSTANCE  pc_I
            (VIEWREF  pc_structure  pc )

            (ACCESSINSTANCE  control_channel )
            (ACCESSINSTANCE  software_channel )
         )
         (NET  N_software_moteur
            (JOINED
               (ACCESSREF  software_channel
                  (INSTANCEREF  pid_I )
               )
               (ACCESSREF  software_channel
                  (INSTANCEREF  pc_I )
               )
            )
         )
         (NET  N_algo1_moteur
            (JOINED
               (ACCESSREF  algo1_channel
                  (INSTANCEREF  pid_I )
               )
               (ACCESSREF  algo1_channel
                  (INSTANCEREF  moteur1_I )
               )
            )
         )
         (NET  N_control_moteur
            (JOINED
               (ACCESSREF  control_channel
                  (INSTANCEREF  pc_I )
               )
               (ACCESSREF  control_channel
                  (INSTANCEREF  moteur1_I )
               )
               (ACCESSREF  control_channel
                  (INSTANCEREF  moteur2_I )
               )
               (ACCESSREF  control_channel
                  (INSTANCEREF  pid_I )
               )
            )
         )
         (NET  N_algo2_moteur
            (JOINED
               (ACCESSREF  algo2_channel
                  (INSTANCEREF  pid_I )
               )
               (ACCESSREF  algo2_channel
                  (INSTANCEREF  moteur2_I )
               )
            )
         )
      )
   )
)
(DESIGNUNIT  pid
   (VIEW  pid_structure
      (VIEWTYPE "structure")
      (INTERFACE
         (ACCESS  software_channel)
         (ACCESS  algo2_channel)
         (ACCESS  algo1_channel)
         (ACCESS  control_channel)
      )
      (CONTENTS
         (INSTANCE  control_I
            (VIEWREF  control_behaviour  control )
            (ACCESSINSTANCE  algo1_channel )
            (ACCESSINSTANCE  algo2_channel )
            (ACCESSINSTANCE  software_channel )
            (ACCESSINSTANCE  control_channel )
         )
         (NET  N_software_pid
            (JOINED
               (ACCESSREF  software_channel
                  (INSTANCEREF  control_I )
               )
               (ACCESSREF  software_channel
               )
            )
         )
         (NET  N_algo2_pid
            (JOINED
               (ACCESSREF  algo2_channel
                  (INSTANCEREF  control_I )
               )
               (ACCESSREF  algo2_channel
               )
            )
         )
         (NET  N_algo1_pid
            (JOINED
               (ACCESSREF  algo1_channel
                  (INSTANCEREF  control_I )
               )
               (ACCESSREF  algo1_channel
               )
            )
         )
         (NET  N_control_pid
            (JOINED
               (ACCESSREF  control_channel
               )
               (ACCESSREF  control_channel
                  (INSTANCEREF  control_I )
               )
            )
         )
      )
   )
)
(DESIGNUNIT  control
   (VIEW  control_behaviour
      (VIEWTYPE "behaviour")
      (INTERFACE
         (ACCESS  algo1_channel )
         (ACCESS  algo2_channel )
```

```
   (ACCESS  software_channel )
   (ACCESS  control_channel )
)
(CONTENTS
   (VARIABLE  sdl_signal (INTEGER ))
   (VARIABLE  memoire (TYPEREF  memory ) )
   (VARIABLE  u1 (REAL ) )
   (VARIABLE  u2 (REAL ) )
   (VARIABLE  u3 (REAL ) )
   (VARIABLE  u4 (REAL ) )
   (VARIABLE  u5 (REAL ) )
   (VARIABLE  u6 (REAL ) )
   (VARIABLE  u7 (REAL ) )
   (VARIABLE  u8 (REAL ) )
   (VARIABLE  u9 (REAL ) )
   (VARIABLE  u10 (REAL ) )
   (VARIABLE  u11 (REAL ) )
   (VARIABLE  u12 (REAL ) )
   (VARIABLE  u13 (REAL ) )
   (VARIABLE  u14 (REAL ) )
   (VARIABLE  temp (REAL ) )
   (TYPEDEF (ARRAY  memory  14 )(REAL ) )
   (TYPEDEF  mindexint (TYPEREF  natural ) )
   (CONSTANT  pc_to_pid (INTEGER )(INITIALVALUE  2 ) )
   (CONSTANT  moteur2_to_pid (INTEGER )(INITIALVALUE  5 ) )
   (CONSTANT  moteur1_to_pid (INTEGER )(INITIALVALUE  8 ) )
   (CONSTANT  pid_to_pc (INTEGER )(INITIALVALUE  1 ) )
   (CONSTANT  pid_to_moteur2_read (INTEGER )(INITIALVALUE  3 ) )
   (CONSTANT  pid_to_moteur2_data (INTEGER )(INITIALVALUE  4 ) )
   (CONSTANT  pid_to_moteur1_read (INTEGER )(INITIALVALUE  6 ) )
   (CONSTANT  pid_to_moteur1_data (INTEGER )(INITIALVALUE  7 ) )
   (STATETABLE  statetable_control
      (STATELIST  start  attente_donnee_pc_1  attente_donnee_pc_2
         attente_donnee_pc_3  attente_donnee_pc_4  attente_donnee_pc_5
         attente_donnee_pc_6  attente_donnee_pc_7  attente_donnee_pc_8
         attente_donnee_pc_9  attente_donnee_pc_10  attente_donnee_pc_11
         attente_donnee_pc_12  attente_donnee_pc_13  attente_donnee_pc_14
         calcul1  calcul2 )
      (ENTRYSTATE  start )
      (STATE  start
            (PROPERTY  sdlline  69 )
         (CUCALL  put_real  algo1_channel
            (PARAMETERASSIGN  sdl_signal  pid_to_moteur1_data
               (DIRECTION  IN ) (INTEGER ))
            (PARAMETERASSIGN  param_1  0.000000
               (DIRECTION  IN ) (REAL ) )
         )
         (CUCALL  put_real  algo2_channel
            (PARAMETERASSIGN  sdl_signal  pid_to_moteur2_data
               (DIRECTION  IN ) (INTEGER ))
            (PARAMETERASSIGN  param_1  0.000000
               (DIRECTION  IN ) (REAL ) )
         )
         (NEXTSTATE   attente_donnee_pc_1 )
      )
      (STATE  attente_donnee_pc_1
            (PROPERTY  sdlline  75 )
         (CUCALL  get_signal  control_channel
            (PARAMETERASSIGN  sdl_signal  sdl_signal
               (DIRECTION  OUT ) (INTEGER ))
         )
         (CASE
            (ALT (=  sdl_signal  pc_to_pid )
                  (PROPERTY  sdlline  76 )
               (CUCALL  get_real  control_channel
                  (PARAMETERASSIGN  param_1  temp
                     (DIRECTION  OUT ) (REAL ) )
               )
               (ASSIGN (MEMBER  memoire  1 ) temp )
               (CUCALL  put_signal  software_channel
                  (PARAMETERASSIGN  sdl_signal  pid_to_pc
                     (DIRECTION  IN ) (INTEGER ))
               )
               (NEXTSTATE   attente_donnee_pc_2 )
            )
         )
      )

      ;STATE  attente_donnee_pc_2 TO attente_donnee_pc_13

      (STATE  attente_donnee_pc_14
            (PROPERTY  sdlline  179 )
         (CUCALL  get_signal  control_channel
            (PARAMETERASSIGN  sdl_signal  sdl_signal
               (DIRECTION  OUT ) (INTEGER ))
         )
         (CASE
            (ALT (=  sdl_signal  pc_to_pid )
                  (PROPERTY  sdlline  180 )
               (CUCALL  get_real  control_channel
                  (PARAMETERASSIGN  param_1  temp
                     (DIRECTION  OUT ) (REAL ) )
               )
               (ASSIGN (MEMBER  memoire  14 ) temp )
               (CUCALL  put_signal  algo1_channel
                 (PARAMETERASSIGN  sdl_signal  pid_to_moteur1_read
                    (DIRECTION  IN ) (INTEGER ))
               )
               (NEXTSTATE   calcul1 )
            )
         )
      )
      (STATE  calcul1
            (PROPERTY  sdlline  187 )
         (CUCALL  get_signal  control_channel
            (PARAMETERASSIGN  sdl_signal  sdl_signal
               (DIRECTION  OUT ) (INTEGER ))
         )
         (CASE
            (ALT (=  sdl_signal  moteur1_to_pid )
                  (PROPERTY  sdlline  188 )
               (CUCALL  get_real  control_channel
                  (PARAMETERASSIGN  param_1  u2
                     (DIRECTION  OUT ) (REAL ) )
               )
               (ASSIGN  u1 (MEMBER  memoire  1 ))
               (ASSIGN  u3 (MEMBER  memoire  2 ))
               (ASSIGN  u4 (MEMBER  memoire  3 ))
               (ASSIGN  u5 (MEMBER  memoire  4 ))
               (ASSIGN  u6 (MEMBER  memoire  5 ))
               (ASSIGN  u7 (MEMBER  memoire  6 ))
               (ASSIGN  u8 (MEMBER  memoire  7 ))
               (ASSIGN  u9 (-  u1  u2 ))
               (ASSIGN  u10 (*  u9  u6 ))
               (ASSIGN  u11 (*  u4  u7 ))
               (ASSIGN  u12 (-  u3  u11 ))
               (ASSIGN  u13 (*  u5  u8 ))
               (ASSIGN  u14 (+ (+  u10  u12 ) u13 ))
               (CUCALL  put_real  algo1_channel
                  (PARAMETERASSIGN  sdl_signal  pid_to_moteur1_data
                     (DIRECTION  IN ) (INTEGER ))
                  (PARAMETERASSIGN  param_1  u14
                       (DIRECTION  IN ) (REAL ) )
               )
               (ASSIGN (MEMBER  memoire  4 ) u4 )
               (ASSIGN (MEMBER  memoire  3 ) u9 )
               (ASSIGN (MEMBER  memoire  2 ) u14 )
               (CUCALL  put_signal  algo2_channel
                  (PARAMETERASSIGN  sdl_signal  pid_to_moteur2_read
                     (DIRECTION  IN ) (INTEGER ))
               )
               (NEXTSTATE   calcul2 )
            )
         )
      )
      (STATE  calcul2
            (PROPERTY  sdlline  211 )
         (CUCALL  get_signal  control_channel
            (PARAMETERASSIGN  sdl_signal  sdl_signal
               (DIRECTION  OUT ) (INTEGER ))
         )
         (CASE
            (ALT (=  sdl_signal  moteur2_to_pid )
                  (PROPERTY  sdlline  212 )
               (CUCALL  get_real  control_channel
                  (PARAMETERASSIGN  param_1  u2
                     (DIRECTION  OUT ) (REAL ) )
               )
               (ASSIGN  u1 (MEMBER  memoire  8 ))
               (ASSIGN  u3 (MEMBER  memoire  9 ))
               (ASSIGN  u4 (MEMBER  memoire  10 ))
               (ASSIGN  u5 (MEMBER  memoire  11 ))
               (ASSIGN  u6 (MEMBER  memoire  12 ))
               (ASSIGN  u7 (MEMBER  memoire  13 ))
```

```
                (ASSIGN  u8 (MEMBER  memoire  14 ))
                (ASSIGN  u9 (-  u1  u2 ))
                (ASSIGN  u10 (*  u9  u6 ))
                (ASSIGN  u11 (*  u4  u7 ))
                (ASSIGN  u12 (-  u3  u11 ))
                (ASSIGN  u13 (*  u5  u8 ))
                (ASSIGN  u14 (+ (+  u10  u12 ) u13 ))
                (CUCALL  put_real  algo2_channel
                   (PARAMETERASSIGN  sdl_signal  pid_to_moteur2_data
                      (DIRECTION  IN ) (INTEGER ))
                   (PARAMETERASSIGN  param_1  u14
                      (DIRECTION  IN ) (REAL ) )
                )
                (ASSIGN (MEMBER  memoire  11 ) u4 )
                                                        (ASSIGN (MEMBER  memoire  10 ) u9 )
                                                        (ASSIGN (MEMBER  memoire  9 ) u14 )
                                                        (CUCALL  put_signal  algo1_channel
                                                           (PARAMETERASSIGN  sdl_signal  pid_to_moteur1_read
                                                              (DIRECTION  IN ) (INTEGER ))
                                                        )
                                                        (NEXTSTATE   calcul1 )
                                                     )
                                                  )
                                               )
                                            )
                                         )
                                      )
```

FIG. B.2: Système SOLAR avant l'allocation du canal *DW03fifo*

```
(DESIGNUNIT  moteur
   (VIEW  moteur_structure
      (VIEWTYPE "structure")
      (CONTENTS
         (TYPEDEF  PID (INTEGER ))
         (CONSTANT  N_algo1_pid (TYPEREF  PID ) (INITIALVALUE  0 ) )
         (CONSTANT  N_algo2_pid (TYPEREF  PID ) (INITIALVALUE  1 ) )
         (INSTANCE  pid_I
            (VIEWREF  pid_structure  pid )
            (ACCESSINSTANCE  software_channel )
            (ACCESSINSTANCE  control_channel )
            (ACCESSINSTANCE  DW03fifo_1_channel )
         )
         (INSTANCE  moteur1_I
            (VIEWREF  moteur1_structure  moteur1 )
            (ACCESSINSTANCE  control_channel )
            (ACCESSINSTANCE  DW03fifo_1_channel )
         )
         (INSTANCE  moteur2_I
            (VIEWREF  moteur2_structure  moteur2 )
            (ACCESSINSTANCE  control_channel )
            (ACCESSINSTANCE  DW03fifo_1_channel )
         )
         (INSTANCE  pc_I
            (VIEWREF  pc_structure  pc )
            (ACCESSINSTANCE  control_channel )
            (ACCESSINSTANCE  software_channel )
         )
         (NET  N_software_moteur
            (JOINED
               (ACCESSREF  software_channel
                  (INSTANCEREF  pid_I )
               )
               (ACCESSREF  software_channel
                  (INSTANCEREF  pc_I )
               )
            )
         )
         (NET  N_control_moteur
            (JOINED
               (ACCESSREF  control_channel
                  (INSTANCEREF  pc_I )
               )
               (ACCESSREF  control_channel
                  (INSTANCEREF  moteur1_I )
               )
               (ACCESSREF  control_channel
                  (INSTANCEREF  moteur2_I )
               )
               (ACCESSREF  control_channel
                  (INSTANCEREF  pid_I )
               )
            )
         )
         (NET  DW03fifo_1_moteur
            (JOINED
               (ACCESSREF  DW03fifo_1_channel
                                                        (INSTANCEREF  pid_I )
                                                     )
                                                     (ACCESSREF  DW03fifo_1_channel
                                                        (INSTANCEREF  moteur1_I )
                                                     )
                                                     (ACCESSREF  DW03fifo_1_channel
                                                        (INSTANCEREF  moteur2_I )
                                                     )
                                                  )
                                               )
                                            )
                                         )
                                      )
                                   )
                                )
                                (DESIGNUNIT  pid
                                   (VIEW  pid_structure
                                      (VIEWTYPE "structure")
                                      (INTERFACE
                                         (ACCESS  software_channel)
                                         (ACCESS  control_channel)
                                         (ACCESS  DW03fifo_1_channel)
                                      )
                                      (CONTENTS
                                         (INSTANCE  control_I
                                            (VIEWREF  control_behaviour  control )
                                            (ACCESSINSTANCE  software_channel )
                                            (ACCESSINSTANCE  control_channel )
                                            (ACCESSINSTANCE  DW03fifo_1_channel )
                                            (PARAMETERASSIGN  algo1_channel  N_algo1_pid )
                                            (PARAMETERASSIGN  algo2_channel  N_algo2_pid )
                                         )
                                         (NET  N_software_pid
                                            (JOINED
                                               (ACCESSREF  software_channel
                                                  (INSTANCEREF  control_I )
                                               )
                                               (ACCESSREF  software_channel
                                               )
                                            )
                                         )
                                         (NET  N_control_pid
                                            (JOINED
                                               (ACCESSREF  control_channel
                                               )
                                               (ACCESSREF  control_channel
                                                  (INSTANCEREF  control_I )
                                               )
                                            )
                                         )
                                         (NET  DW03fifo_1_pid
                                            (JOINED
                                               (ACCESSREF  DW03fifo_1_channel
                                                  (INSTANCEREF  control_I )
                                               )
                                               (ACCESSREF  DW03fifo_1_channel
                                               )
                                            )
                                         )
```

```
      )
   )
)
(DESIGNUNIT  control
   (VIEW  control_behaviour
      (VIEWTYPE "behaviour")
      (INTERFACE
         (PARAMETER  algo1_channel (TYPEREF  PID ) )
         (PARAMETER  algo2_channel (TYPEREF  PID ) )
         (ACCESS  software_channel )
         (ACCESS  control_channel )
         (ACCESS  DW03fifo_1_channel )
      )
      (CONTENTS
         (VARIABLE  sdl_signal (INTEGER ))
         (VARIABLE  memoire (TYPEREF  memory ) )
         (VARIABLE  u1 (REAL ) )
         (VARIABLE  u2 (REAL ) )
         (VARIABLE  u3 (REAL ) )
         (VARIABLE  u4 (REAL ) )
         (VARIABLE  u5 (REAL ) )
         (VARIABLE  u6 (REAL ) )
         (VARIABLE  u7 (REAL ) )
         (VARIABLE  u8 (REAL ) )
         (VARIABLE  u9 (REAL ) )
         (VARIABLE  u10 (REAL ) )
         (VARIABLE  u11 (REAL ) )
         (VARIABLE  u12 (REAL ) )
         (VARIABLE  u13 (REAL ) )
         (VARIABLE  u14 (REAL ) )
         (VARIABLE  temp (REAL ) )
         (TYPEDEF (ARRAY  memory  14 )(REAL ) )
         (TYPEDEF  mindexint (TYPEREF  natural ) )
         (CONSTANT  pc_to_pid (INTEGER )(INITIALVALUE  2 ) )
         (CONSTANT  moteur2_to_pid (INTEGER )(INITIALVALUE  5 ) )
         (CONSTANT  moteur1_to_pid (INTEGER )(INITIALVALUE  8 ) )
         (CONSTANT  pid_to_pc (INTEGER )(INITIALVALUE  1 ) )
         (CONSTANT  pid_to_moteur2_read (INTEGER )(INITIALVALUE  3 ) )
         (CONSTANT  pid_to_moteur2_data (INTEGER )(INITIALVALUE  4 ) )
         (CONSTANT  pid_to_moteur1_read (INTEGER )(INITIALVALUE  6 ) )
         (CONSTANT  pid_to_moteur1_data (INTEGER )(INITIALVALUE  7 ) )
         (STATETABLE  statetable_control
            (STATELIST  start  attente_donnee_pc_1  attente_donnee_pc_2
               attente_donnee_pc_3  attente_donnee_pc_4  attente_donnee_pc_5
               attente_donnee_pc_6  attente_donnee_pc_7  attente_donnee_pc_8
               attente_donnee_pc_9  attente_donnee_pc_10  attente_donnee_pc_11
               attente_donnee_pc_12  attente_donnee_pc_13  attente_donnee_pc_14
               calcul1  calcul2 )
            (ENTRYSTATE  start )
            (STATE  start
                  (PROPERTY  sdlline  69 )
               (CUCALL  DW03put_real  DW03fifo_1_channel
                  (PARAMETERASSIGN  sdl  pid_to_moteur1_data
                     (DIRECTION  IN ) (INTEGER ))
                  (PARAMETERASSIGN  data  0.000000
                     (DIRECTION  IN ) (REAL ) )
                  (PARAMETERASSIGN  id  algo1_channel
                     (DIRECTION  IN ) (TYPEREF  PID )
                        (PROPERTY  subchannel  static ))
               )
               (CUCALL  DW03put_real  DW03fifo_1_channel
                  (PARAMETERASSIGN  sdl  pid_to_moteur2_data
                     (DIRECTION  IN ) (INTEGER ))
                  (PARAMETERASSIGN  data  0.000000
                     (DIRECTION  IN ) (REAL ) )
                  (PARAMETERASSIGN  id  algo2_channel
                     (DIRECTION  IN ) (TYPEREF  PID )
                        (PROPERTY  subchannel  static ))
               )
               (NEXTSTATE   attente_donnee_pc_1 )
            )
            (STATE  attente_donnee_pc_1
                  (PROPERTY  sdlline  75 )
               (CUCALL  get_signal  control_channel
                  (PARAMETERASSIGN  sdl_signal  sdl_signal
                     (DIRECTION  OUT ) (INTEGER ))
               )
               (CASE
                  (ALT (=  sdl_signal  pc_to_pid )
                     (CUCALL  get_real  control_channel
                        (PARAMETERASSIGN  param_1  temp
                           (DIRECTION  OUT ) (REAL ) )
                     )
                     (ASSIGN (MEMBER  memoire  1 ) temp )
                     (CUCALL  put_signal  software_channel
                        (PARAMETERASSIGN  sdl_signal  pid_to_pc
                           (DIRECTION  IN ) (INTEGER ))
                     )
                     (NEXTSTATE   attente_donnee_pc_2 )
                  )
               )
            )

            ;STATE  attente_donnee_pc_2 TO attente_donnee_pc_13

            (STATE  attente_donnee_pc_14
                  (PROPERTY  sdlline  179 )
               (CUCALL  get_signal  control_channel
                  (PARAMETERASSIGN  sdl_signal  sdl_signal
                     (DIRECTION  OUT ) (INTEGER ))
               )
               (CASE
                  (ALT (=  sdl_signal  pc_to_pid )
                     (CUCALL  get_real  control_channel
                        (PARAMETERASSIGN  param_1  temp
                           (DIRECTION  OUT ) (REAL ) )
                     )
                     (ASSIGN (MEMBER  memoire  14 ) temp )
                     (CUCALL  DW03put_signal  DW03fifo_1_channel
                        (PARAMETERASSIGN  sdl  pid_to_moteur1_read
                           (DIRECTION  IN ) (INTEGER ))
                        (PARAMETERASSIGN  id  algo1_channel
                           (DIRECTION  IN ) (TYPEREF  PID )
                              (PROPERTY  subchannel  static ))
                     )
                     (NEXTSTATE   calcul1 )
                  )
               )
            )
            (STATE  calcul1
                  (PROPERTY  sdlline  187 )
               (CUCALL  get_signal  control_channel
                  (PARAMETERASSIGN  sdl_signal  sdl_signal
                     (DIRECTION  OUT ) (INTEGER ))
               )
               (CASE
                  (ALT (=  sdl_signal  moteur1_to_pid )
                     (CUCALL  get_real  control_channel
                        (PARAMETERASSIGN  param_1  u2
                           (DIRECTION  OUT ) (REAL ) )
                     )
                     (ASSIGN  u1 (MEMBER  memoire  1 ))
                     (ASSIGN  u3 (MEMBER  memoire  2 ))
                     (ASSIGN  u4 (MEMBER  memoire  3 ))
                     (ASSIGN  u5 (MEMBER  memoire  4 ))
                     (ASSIGN  u6 (MEMBER  memoire  5 ))
                     (ASSIGN  u7 (MEMBER  memoire  6 ))
                     (ASSIGN  u8 (MEMBER  memoire  7 ))
                     (ASSIGN  u9 (-  u1  u2 ))
                     (ASSIGN  u10 (*  u9  u6 ))
                     (ASSIGN  u11 (*  u4  u7 ))
                     (ASSIGN  u12 (-  u3  u11 ))
                     (ASSIGN  u13 (*  u5  u8 ))
                     (ASSIGN  u14 (+ (+  u10  u12 ) u13 ))
                     (CUCALL  DW03put_real  DW03fifo_1_channel
                        (PARAMETERASSIGN  sdl  pid_to_moteur1_data
                           (DIRECTION  IN ) (INTEGER ))
                        (PARAMETERASSIGN  data  u14
                           (DIRECTION  IN ) (REAL ) )
                        (PARAMETERASSIGN  id  algo1_channel
                           (DIRECTION  IN ) (TYPEREF  PID )
                              (PROPERTY  subchannel  static ))
                     )
                     (ASSIGN (MEMBER  memoire  4 ) u4 )
                     (ASSIGN (MEMBER  memoire  3 ) u9 )
                     (ASSIGN (MEMBER  memoire  2 ) u14 )
                     (CUCALL  DW03put_signal  DW03fifo_1_channel
                        (PARAMETERASSIGN  sdl  pid_to_moteur2_read
                           (DIRECTION  IN ) (INTEGER ))
                        (PARAMETERASSIGN  id  algo2_channel
                           (DIRECTION  IN ) (TYPEREF  PID )
                              (PROPERTY  subchannel  static ))
                     )
                     (NEXTSTATE   calcul2 )
```

```
      )
   )
)
(STATE  calcul2
     (PROPERTY  sdlline  211 )
   (CUCALL  get_signal  control_channel
      (PARAMETERASSIGN  sdl_signal  sdl_signal
         (DIRECTION  OUT ) (INTEGER ))
   )
   (CASE
      (ALT (=  sdl_signal  moteur2_to_pid )
         (CUCALL  get_real  control_channel
            (PARAMETERASSIGN  param_1  u2
               (DIRECTION  OUT ) (REAL ) )
         )
         (ASSIGN  u1 (MEMBER  memoire  8 ))
         (ASSIGN  u3 (MEMBER  memoire  9 ))
         (ASSIGN  u4 (MEMBER  memoire  10 ))
         (ASSIGN  u5 (MEMBER  memoire  11 ))
         (ASSIGN  u6 (MEMBER  memoire  12 ))
         (ASSIGN  u7 (MEMBER  memoire  13 ))
         (ASSIGN  u8 (MEMBER  memoire  14 ))
         (ASSIGN  u9 (-  u1  u2 ))
         (ASSIGN  u10 (*  u9  u6 ))
         (ASSIGN  u11 (*  u4  u7 ))
         (ASSIGN  u12 (-  u3  u11 ))
         (ASSIGN  u13 (*  u5  u8 ))
         (ASSIGN  u14 (+ (+  u10  u12 ) u13 ))
```

```
      (CUCALL  DW03put_real  DW03fifo_1_channel
         (PARAMETERASSIGN  sdl  pid_to_moteur2_data
            (DIRECTION  IN ) (INTEGER ))
         (PARAMETERASSIGN  data  u14
            (DIRECTION  IN ) (REAL ) )
         (PARAMETERASSIGN  id  algo2_channel
            (DIRECTION  IN ) (TYPEREF  PID )
               (PROPERTY  subchannel  static ))
      )
      (ASSIGN (MEMBER  memoire  11 ) u4 )
      (ASSIGN (MEMBER  memoire  10 ) u9 )
      (ASSIGN (MEMBER  memoire  9 ) u14 )
      (CUCALL  DW03put_signal  DW03fifo_1_channel
         (PARAMETERASSIGN  sdl  pid_to_moteur1_read
            (DIRECTION  IN ) (INTEGER ))
         (PARAMETERASSIGN  id  algo1_channel
            (DIRECTION  IN ) (TYPEREF  PID )
               (PROPERTY  subchannel  static ))
      )
      (NEXTSTATE   calcul1 )
   )
)
)
)
)
)
)
)
```

FIG. B.3: Système SOLAR après l'allocation du canal *DW03fifo*

La figure B.3 représente le système de la figure B.2 après l'allocation du canal *DW03fifo* de la bibliothèque de la figure B.1 pour les canaux abstrait (*net*) *N_ algo1_ moteur* et *N_ algo2_ moteur*. Un canal *DW03fifo_ 1_ moteur* a été crée ainsi que deux contantes *N_ algo1_ pid* et *N_ algo2_ pid* correspondant au deux canaux fusionnés. Ces constantes sont passées comme paramètre dans l'interface de chaque *design unit* accédant au canal et servent à identifier le deux sous-canal adressé. Elles ont été rajoutées dans chaque *cucall* comme paramètre avec la propriété *subchannel static*. Les primitives de communication abstraite *get_ signal* et *get_ int* sont exécutées par la même *method* : *DW03get_ int*.

Annexe C

Primitive MAP

C.1 Syntaxe

La primitive map permet d'effectuer la synthèse d'interface sur un système décrit en SOLAR. La syntaxe d'appel est la suivante :

```
map [fichier.solar] [channel unit] [net]
options :
    -nc
    -du [instance externe] -proc [method]+
    -lib [bibliotheque]
```

Les options de *map* sont les suivantes :

- `<-nc>` permet de ne pas inclure le contrôleur de communication. Cette option permet de générer l'interface externe pour le contrôleur de communication d'un système communicant avec son environnement. Les conditions d'application de cette option sont les suivantes :
  1. le map doit être appliqué au plus haut niveau de la hiérarchie du système.
  2. le système doit disposer d'un *access* à l'environnement dans son interface (le *net* expansé doit être connecté à l'interface du *design unit*).
- `<-du>` permet de spécifier le nom d'une instance d'un *design unit* externe accédant au canal. Cette option doit etre utilisée si le *channel unit* transposé contient des ports *process private* [1] dans son interface.
- `<-proc>` permet de spécifier des procédures externes accédant au canal et de générer l'interface nécessaire.
  Les conditions d'applications de ces deux options sont les suivantes :
  1. Le système doit posséder un accès *access* à son environnement (le *net* expansé doit être connecté à l'interface du *design unit*).

[1] voir paragraphe C.2

– `<-lib>` permet de spécifier le nom de la bibliothèque de communication. La bibliothèque par défaut est `./.cu`. La variable d'environnement `COSMOS_LIBRARY_PATH` permet de spécifier le répertoire par défaut de la bibliothèque de communication.

Les restrictions de *map* sont les suivantes :

– seules les instances multiples de *design unit* comportemental sont supportés avec les ports de type *process private*. Les instances multiples de *design unit* structurels ne sont supportés qu'avec les ports de type *process shared*.

– Les ports de type *subchannel private* permettant de transposer des canaux fusionnés ne sont pas supportés actuellement.

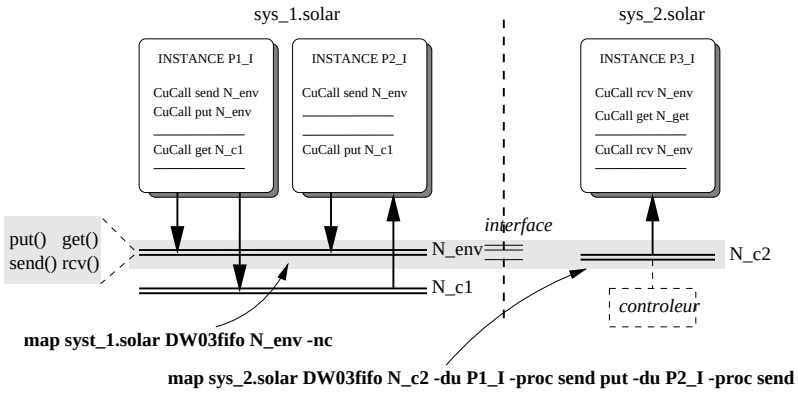


FIG. C.1: Map avec contrôleur externe

La figure C.1 représente le *map* d'un canal (*N_ env*) connecté à l'environnement. Le système *syst_ 1* est connecté à l'environnement à travers le canal *N_ env*. L'option `-nc` de *map* permet de générer l'interface pour un contrôleur de communication externe au système (situé dans *syst_ 2*). Dans le système *syst_ 2* l'option `-du P1_I -proc send put` permet de générer l'interface avec l'environnement permettant au *design unit P1_ I* d'accéder au canal et l'option `-du P2_I -proc send` celle correspondant au *design unit P2_ I*.

C.2 Bibliothèque de communication

La bibliothèque SOLAR contient les prototypes SOLAR des primitives de communication et l'interface de l'éventuel contrôleur de communication. A chaque port, peuvent être attribués deux propriétés [2] indiquant le type de port de l'interface [3] :

– la propriété "*process*".
Elle permet d'indiquer si le port est privé (*process private*) ou partagé (*process shared*) par chaque *design unit* accédant au canal. Un bus de données est généralement de type "*process*

[2] *property* SOLAR
[3] Cette propriété doit être indiquée dans l'interface de chaque procédure de communication du canal

shared" alors que des signaux de contrôle peuvent être partagés ou privés. En fonction de la valeur de la propriété, les ports sont interconnectés de la façon suivante :

1. port "*process private*" : un port [4] dans l'interface du *design unit* et un *net* sont crées pour chaque processus accédant au canal. A un port de type "*process private*" d'une primitive de communication doit correspondre un port de type tableau à une dimension dans l'interface du canal. Chaque *net* généré sera connecté à un élément du tableau.
2. port "*process shared*" : un port dans l'interface du *design unit* est crée et connecté à un *net* partagé par tous les processus accédant au canal. Par défaut, un port est de type "*process shared*".

– la propriété "*subchannel*".
Elle permet d'indiquer si le port est privé (*subchannel private*) ou partagé (*subchannel shared*) par les sous-canaux du canal. Un bus de données est généralement de type "*subchannel shared*" alors que des signaux de contrôle peuvent être partagés ou privés. Les ports sont interconnectés de la façon suivante :

1. port *subchannel private* : un port [5] et un *net* sont crées pour chaque sous-canal accèdé par un *design unit*. A un port de type "*subchannel private*" d'une primitive de communication doit correspondre un port de type tableau à une dimension dans l'interface du canal.
2. port *subchannel shared* : un port dans l'interface du *design unit* est crée et connecté à un *net* partagé par tous les sous-canaux du canal. Par défaut, un port est de type "*subchannel shared*".

Le tableau C.1 résume les différentes interconnexions possibles. La figure C.2 illustre les différentes

subchannel	**process**	
	shared	**private**
shared	port unique	tableau unidimensionnel indexé par le N^o du processus
private	tableau unidimensionnel indexé par le N^o du sous-canal	tableau bidimensionnel indexé par le N^o du processus par le N^o du sous-canal

TAB. C.1: Interconnexions de la primitive *map*

interconnexions générées par *map*. Le numéros de sous-canal sont générés par *alloc* [6] et utilisés par *map*. Les numéros de processus sont générés par *map*. La figure C.2.2 représente le *map* d'un canal contenant deux sous-canaux. Le processus *P1* accède au sous canal 0 et les processus *P2* et *P3* au sous canal 1. La figure C.2.1 représente le cas ou les signaux sont partagés par les sous-canaux.

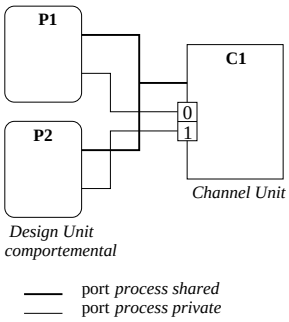
La figure C.3 représente une bibliothèque de communication SOLAR contenant deux *channel unit* :

– une unité de communication réalisant un protocole *fifo* contenant un contrôleur externe (DW03_fifo).
– un protocole *handshake* sans contrôleur externe (HANDSHAKE).

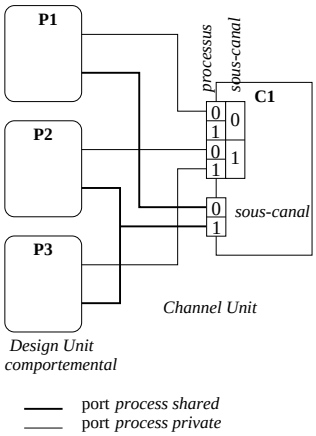
[4] les ports "*process private/shared*" sont partagés par les différentes primitives de communication d'un même *design unit* accédant au même canal

[5] les ports de type *subchannel private* ne sont pas supportés dans la version actuelle de la primitive *map*

[6] paramètre *subchannel static*



C.2.1: Port *subchannel shared*



C.2.2: Port *subchannel private*

FIG. C.2: Différentes possibilités d'interconnexion de *map*

```
(SOLAR  Library_of_Communication_Protocols
   (CHANNELUNIT  DW03fifo
      (VIEW  DW03fifo_behaviour
         (LIBRARY VHDL "IEEE")
         (LIBRARY VHDL "DW03")
         (LIBRARY VHDL "DWARE")
         (LIBRARY VHDL "WORK.package_DW03fifo.ALL")
         (INTERFACE
            (PARAMETER data_width (INTEGER) (INITIALVALUE 32))
            (PARAMETER depth (INTEGER) (INITIALVALUE 128))
            (PARAMETER level (INTEGER) (INITIALVALUE 127))
            (PARAMETER nport (INTEGER) (INITIALVALUE 8))
            (PORT (ARRAY fifin data_width) (DIRECTION IN) (TYPEREF STD_LOGIC))
            (PORT (ARRAY wrqb nport) (DIRECTION IN) (TYPEREF STD_LOGIC))
            (PORT (ARRAY rrqb nport) (DIRECTION IN) (TYPEREF STD_LOGIC))
            (PORT csb (DIRECTION IN) (TYPEREF STD_LOGIC))
            (PORT clk (DIRECTION IN) (TYPEREF STD_LOGIC))
            (PORT reset (DIRECTION IN) (TYPEREF STD_LOGIC))
            (PORT (ARRAY fifout data_width) (DIRECTION OUT) (TYPEREF STD_LOGIC))
            (PORT full (DIRECTION OUT) (TYPEREF STD_LOGIC))
            (PORT empty (DIRECTION OUT) (TYPEREF STD_LOGIC))
            (PORT threshold (DIRECTION OUT) (TYPEREF STD_LOGIC))
            (PORT (ARRAY req nport) (DIRECTION IN) (TYPEREF STD_LOGIC))
            (PORT (ARRAY gnt nport) (DIRECTION OUT) (TYPEREF STD_LOGIC))
            (PORT done (DIRECTION IN) (TYPEREF STD_LOGIC))
            (METHOD  DW03put_int
               (PORT (ARRAY fifin data_width) (DIRECTION OUT)
                  (TYPEREF STD_LOGIC))
               (PORT wrqb (DIRECTION OUT) (TYPEREF STD_LOGIC) )
               (PORT clk (DIRECTION IN) (TYPEREF STD_LOGIC))
               (PORT full (DIRECTION OUT) (TYPEREF STD_LOGIC))
               (PORT req (DIRECTION OUT) (TYPEREF STD_LOGIC)
                  (PROPERTY process private))
               (PORT gnt (DIRECTION IN) (TYPEREF STD_LOGIC)
                  (PROPERTY process private))
               (PORT done (DIRECTION OUT) (TYPEREF STD_LOGIC))
               (PARAMETER  sdl (DIRECTION IN) (INTEGER ))
               (PARAMETER  data (DIRECTION IN) (INTEGER ))
               (FOREIGN VHDL "package_DW03fifo")
            )
            (METHOD  DW03get_int
               (PORT rrqb (DIRECTION OUT) (TYPEREF STD_LOGIC)
                  (PROPERTY process private))
               (PORT clk (DIRECTION IN) (TYPEREF STD_LOGIC))
               (PORT (ARRAY fifout data_width) (DIRECTION IN)
                  (TYPEREF STD_LOGIC))
               (PORT empty (DIRECTION IN) (TYPEREF STD_LOGIC))
               (PARAMETER  data (DIRECTION OUT) (INTEGER ))
               (FOREIGN VHDL "package_DW03fifo")
            )
            (METHOD  DW03put_real
               (PORT (ARRAY fifin data_width) (DIRECTION OUT)
                  (TYPEREF STD_LOGIC))
               (PORT wrqb (DIRECTION OUT) (TYPEREF STD_LOGIC)
                  (PROPERTY process private))
               (PORT clk (DIRECTION IN) (TYPEREF STD_LOGIC))
               (PORT full (DIRECTION OUT) (TYPEREF STD_LOGIC))
               (PORT req (DIRECTION OUT) (TYPEREF STD_LOGIC)
                  (PROPERTY process private))
               (PORT gnt (DIRECTION IN) (TYPEREF STD_LOGIC)
                  (PROPERTY process private))
               (PORT done (DIRECTION OUT) (TYPEREF STD_LOGIC))
               (PARAMETER  sdl (DIRECTION IN) (INTEGER ))
               (PARAMETER  data (DIRECTION IN) (REAL ))
               (FOREIGN VHDL "package_DW03fifo")
            )
            (METHOD  DW03get_real
               (PORT rrqb (DIRECTION OUT) (TYPEREF STD_LOGIC)
                  (PROPERTY process private))
               (PORT clk (DIRECTION IN) (TYPEREF STD_LOGIC))
               (PORT (ARRAY fifout data_width) (DIRECTION IN)
                  (TYPEREF STD_LOGIC))
               (PORT empty (DIRECTION IN) (TYPEREF STD_LOGIC))
               (PARAMETER  data (DIRECTION OUT) (REAL ) )
               (FOREIGN VHDL "package_DW03fifo")
            )
         )
         (CONTENTS
            (FOREIGN VHDL "package_DW03fifo")
         )
      )
   )
   (CHANNELUNIT  HANDSHAKE
      (VIEW  HANDSHAKE_behaviour
         (LIBRARY VHDL "IEEE")
         (LIBRARY VHDL "WORK.package_handshake.ALL")
         (INTERFACE
            (PARAMETER data_width (INTEGER) (INITIALVALUE 32))
            (PARAMETER nport (INTEGER) (INITIALVALUE 8))
            (METHOD  HANDput_int
```

```
            (PORT req  (DIRECTION  OUT ) (BIT )
               (PROPERTY  process  private ))
            (PORT gnt  (DIRECTION  IN ) (BIT )
               (PROPERTY  process  private ))
            (PORT (ARRAY data data_width) (DIRECTION  INOUT )
               (TYPEREF  STD_LOGIC ) )
            (PARAMETER  sdl_signal (INTEGER ))
            (PARAMETER  param_1 (INTEGER ))
            (FOREIGN VHDL "package_hanshake")
         )
         (METHOD  HANDget_int
            (PORT (ARRAY req  nport) (DIRECTION  IN ) (BIT ))
            (PORT (ARRAY req  nport) (DIRECTION  OUT ) (BIT ))
            (PORT (ARRAY data data_width) (DIRECTION  INOUT )
               (TYPEREF  STD_LOGIC ) )
            (PARAMETER  param_1 (INTEGER ))
            (FOREIGN VHDL "package_hanshake")
         )
         (METHOD  HANDput_real
            (PORT req  (DIRECTION  OUT ) (BIT )
               (PROPERTY  process  private ))
            (PORT gnt  (DIRECTION  IN ) (BIT )
               (PROPERTY  process  private ))
            (PORT (ARRAY data data_width) (DIRECTION  INOUT )
               (TYPEREF  STD_LOGIC ) )
            (PARAMETER  sdl_signal (INTEGER ))
            (PARAMETER  param_1 (REAL ) )
            (FOREIGN VHDL "package_hanshake")
         )
         (METHOD  HANDget_real
            (PORT (ARRAY req  nport) (DIRECTION  IN ) (BIT ))
            (PORT (ARRAY req  nport) (DIRECTION  OUT ) (BIT ))
            (PORT (ARRAY data data_width) (DIRECTION  INOUT )
               (TYPEREF  STD_LOGIC ) )
            (PARAMETER  param_1 (REAL ) )
            (FOREIGN VHDL "package_hanshake")
         )
      )
      (CONTENTS
      )
   )
)
```

FIG. C.3: Bibliothèque SOLAR d'unités de communication

C.3 Code SOLAR généré

La figure C.3 donne un exemple de bibliothèque de communication et la figure C.4 une partie d'un système SOLAR.

```
(DESIGNUNIT  moteur1
   (VIEW  moteur1_structure
      (VIEWTYPE "structure")
      (INTERFACE
         (ACCESS  control_channel )
         (ACCESS  algo1_channel )
      )
      (CONTENTS
         (INSTANCE  algo1_I
            (VIEWREF  algo1_behaviour  algo1 )
            (ACCESSINSTANCE  control_channel )
            (ACCESSINSTANCE  algo1_channel )
         )
         (NET  N_control_moteur1
            (JOINED
               (ACCESSREF  control_channel
                  (INSTANCEREF  algo1_I )
               )
               (ACCESSREF  control_channel
               )
            )
         )
         (NET  N_algo1_moteur1
            (JOINED
               (ACCESSREF  algo1_channel
               )
               (ACCESSREF  algo1_channel
                  (INSTANCEREF  algo1_I )
               )
            )
         )
      )
   )
)
(DESIGNUNIT  algo1
   (VIEW  algo1_behaviour
      (VIEWTYPE "behaviour")
      (INTERFACE
         (ACCESS  control_channel)
         (ACCESS  algo1_channel)
      )
      (CONTENTS
         (VARIABLE  sdl_signal (INTEGER ))
         (VARIABLE  s_0 (REAL ) )
         (VARIABLE  s_1 (REAL ) )
         (VARIABLE  e_0 (REAL ) )
         (VARIABLE  e_1 (REAL ) )
         (CONSTANT  pid_moteur1_read (INTEGER )(INITIALVALUE  4 ) )
         (CONSTANT  pid_moteur1_data (INTEGER )(INITIALVALUE  5 ) )
         (CONSTANT  moteur1_pid (INTEGER )(INITIALVALUE  6 ) )
         (STATETABLE  statetable_algo1
            (STATELIST  start  attente1  attente2 )
            (ENTRYSTATE  start )
            (STATE  start
                  (PROPERTY  sdlline  525 )
               (ASSIGN  e_1  0.000000)
               (ASSIGN  s_0  0.000000)
               (ASSIGN  s_1  0.000000)
               (NEXTSTATE   attente1 )
            )
            (STATE  attente1
                  (PROPERTY  sdlline  538 )
               (CUCALL  DW03get_int  algo1_channel
                     (PARAMETERASSIGN  data  sdl_signal
                        (DIRECTION  OUT ) (INTEGER ))
               )
               (CASE
                  (ALT (=  sdl_signal  pid_moteur1_data )
                     (CUCALL  DW03get_real  algo1_channel
                        (PARAMETERASSIGN  data  e_0
                           (DIRECTION  OUT ) (REAL ))
                     )
                     (ASSIGN  s_0 (-  s_1 (*  0.005000(-  s_1  e_1 ))))
                     (ASSIGN  s_1  s_0 )
                     (ASSIGN  e_1  e_0 )
                     (NEXTSTATE   attente2 )
                        (PROPERTY  sdlline  542 )
                  )
```

```
            )
         )
         (STATE  attente2
               (PROPERTY  sdlline  561 )
            (CUCALL  DW03get_int  algo1_channel
               (PARAMETERASSIGN  data  sdl_signal
                  (DIRECTION  OUT ) (INTEGER ))
            )
            (CASE
               (ALT (=  sdl_signal  pid_moteur1_read )
                  (CUCALL  put_real  control_channel
                     (PARAMETERASSIGN  sdl_signal  moteur1_pid
                        (DIRECTION  IN ) (INTEGER ))
                     (PARAMETERASSIGN  param_1  s_0
                        (DIRECTION  IN ) (REAL ))
                  )
                  (NEXTSTATE  attente1 )
                     (PROPERTY  sdlline  565 )
               )
            )
         )
      )
   )
```

FIG. C.4: Système SOLAR avant la synthèse d'interface

La figure C.5 donne le résultat de l'application de la ligne de commande `map controleur.solar DW03fifo N_algo1_moteur1`

```
(DESIGNUNIT  moteur1
   (VIEW  moteur1_structure
      (VIEWTYPE "structure")
      (INTERFACE
         (PORT (ARRAY  N_algo1_moteur1_fifin  32 )(DIRECTION  IN )
            (TYPEREF  STD_LOGIC ) )
         (PORT  N_algo1_moteur1_control_I_wrqb (DIRECTION  IN )
            (TYPEREF  STD_LOGIC ) )
         (PORT  N_algo1_moteur1_clk (DIRECTION  OUT )
            (TYPEREF  STD_LOGIC ) )
         (PORT  N_algo1_moteur1_full (DIRECTION  IN )
            (TYPEREF  STD_LOGIC ) )
         (PORT  N_algo1_moteur1_control_I_req (DIRECTION  IN )
            (TYPEREF  STD_LOGIC ) )
         (PORT  N_algo1_moteur1_control_I_gnt (DIRECTION  OUT )
            (TYPEREF  STD_LOGIC ) )
         (PORT  N_algo1_moteur1_done (DIRECTION  IN )
            (TYPEREF  STD_LOGIC ) )
         (ACCESS  control_channel)
      )
      (CONTENTS
         (INSTANCE  algo1_I
            (VIEWREF  algo1_behaviour  algo1 )
            (PORTINSTANCE  N_algo1_moteur1_algo1_I_rrqb )
            (PORTINSTANCE  N_algo1_moteur1_clk )
            (PORTINSTANCE  N_algo1_moteur1_fifout )
            (PORTINSTANCE  N_algo1_moteur1_empty )
            (ACCESSINSTANCE  control_channel )
          )
         (INSTANCE  N_algo1_moteur1_DW03fifo
            (VIEWREF  DW03fifo_behaviour  DW03fifo )
            (PORTINSTANCE  fifin )
            (PORTINSTANCE  wrqb )
            (PORTINSTANCE  rrqb )
            (PORTINSTANCE  clk )
            (PORTINSTANCE  reset )
            (PORTINSTANCE  fifout )
            (PORTINSTANCE  full )
            (PORTINSTANCE  empty )
            (PORTINSTANCE  threshold )
            (PORTINSTANCE  req )
            (PORTINSTANCE  gnt )
            (PORTINSTANCE  done )
            (PARAMETERASSIGN  data_width  32 )
            (PARAMETERASSIGN  depth  128 )
            (PARAMETERASSIGN  level  127 )
            (PARAMETERASSIGN  nport  8 )
         )
         (NET  N_control_moteur1
            (JOINED
               (ACCESSREF  control_channel
                  (INSTANCEREF  algo1_I )
               )
               (ACCESSREF  control_channel
                  )
               )
            )
            (NET  N_algo1_moteur1_fifin
               (JOINED
                  (PORTREF  N_algo1_moteur1_fifin
                  )
                  (PORTREF  fifin
                     (INSTANCEREF  N_algo1_moteur1_DW03fifo )
                  )
               )
            )
            (NET  N_algo1_moteur1_control_I_wrqb
               (JOINED
                  (PORTREF  N_algo1_moteur1_control_I_wrqb
                  )
                  (PORTREF (MEMBER  wrqb  1 )
                     (INSTANCEREF  N_algo1_moteur1_DW03fifo )
                  )
               )
            )
            (NET  N_algo1_moteur1_clk
               (JOINED
                  (PORTREF  N_algo1_moteur1_clk
                  )
                  (PORTREF  N_algo1_moteur1_clk
                     (INSTANCEREF  algo1_I )
                  )
                  (PORTREF  clk
                     (INSTANCEREF  N_algo1_moteur1_DW03fifo )
                  )
               )
            )
            (NET  N_algo1_moteur1_full
               (JOINED
                  (PORTREF  N_algo1_moteur1_full
                  )
                  (PORTREF  full
                     (INSTANCEREF  N_algo1_moteur1_DW03fifo )
                  )
               )
            )
            (NET  N_algo1_moteur1_control_I_req
               (JOINED
                  (PORTREF  N_algo1_moteur1_control_I_req
                  )
                  (PORTREF (MEMBER  req  1 )
                     (INSTANCEREF  N_algo1_moteur1_DW03fifo )
                  )
               )
            )
            (NET  N_algo1_moteur1_control_I_gnt
               (JOINED
                  (PORTREF  N_algo1_moteur1_control_I_gnt
```

```
            )
            (PORTREF (MEMBER  gnt  1 )
               (INSTANCEREF  N_algo1_moteur1_DW03fifo )
            )
         )
      )
      (NET  N_algo1_moteur1_done
         (JOINED
            (PORTREF  N_algo1_moteur1_done
            )
            (PORTREF  done
               (INSTANCEREF  N_algo1_moteur1_DW03fifo )
            )
         )
      )
      (NET  N_algo1_moteur1_algo1_I_rrqb
         (JOINED
            (PORTREF  N_algo1_moteur1_algo1_I_rrqb
               (INSTANCEREF  algo1_I )
            )
            (PORTREF (MEMBER  rrqb  0 )
               (INSTANCEREF  N_algo1_moteur1_DW03fifo )
            )
         )
      )
      (NET  N_algo1_moteur1_fifout
         (JOINED
            (PORTREF  N_algo1_moteur1_fifout
               (INSTANCEREF  algo1_I )
            )
            (PORTREF  fifout
               (INSTANCEREF  N_algo1_moteur1_DW03fifo )
            )
         )
      )
      (NET  N_algo1_moteur1_empty
         (JOINED
            (PORTREF  N_algo1_moteur1_empty
               (INSTANCEREF  algo1_I )
            )
            (PORTREF  empty
               (INSTANCEREF  N_algo1_moteur1_DW03fifo )
            )
         )
      )
   )
)
(DESIGNUNIT  algo1
   (VIEW  algo1_behaviour
      (VIEWTYPE "behaviour")
      (LIBRARY  VHDL  "IEEE" )
      (LIBRARY  VHDL  "DW03" )
      (LIBRARY  VHDL  "DWARE" )
      (LIBRARY  VHDL  "WORK.package_DW03fifo.ALL" )
      (INTERFACE
         (PORT  N_algo1_moteur1_rrqb (DIRECTION  OUT )
            (TYPEREF  STD_LOGIC ) )
         (PORT  N_algo1_moteur1_clk (DIRECTION  IN )
            (TYPEREF  STD_LOGIC ) )
         (PORT (ARRAY  N_algo1_moteur1_fifout  32 )(DIRECTION  IN )
            (TYPEREF  STD_LOGIC ) )
         (PORT  N_algo1_moteur1_empty (DIRECTION  IN )
            (TYPEREF  STD_LOGIC ) )
         (ACCESS  control_channel)
      )
      (CONTENTS
         (VARIABLE  sdl_signal (INTEGER ))
         (VARIABLE  s_0 (REAL ) )
         (VARIABLE  s_1 (REAL ) )
         (VARIABLE  e_0 (REAL ) )
         (VARIABLE  e_1 (REAL ) )
         (CONSTANT  pid_moteur1_read (INTEGER )(INITIALVALUE  4 ) )
         (CONSTANT  pid_moteur1_data (INTEGER )(INITIALVALUE  5 ) )
         (CONSTANT  moteur1_pid (INTEGER )(INITIALVALUE  6 ) )
         (STATETABLE  statetable_algo1
            (STATELIST  start  attente1  attente2 )
            (ENTRYSTATE  start )
            (STATE  start
                  (PROPERTY  sdlline  525 )
               (ASSIGN  e_1  0.000000)
               (ASSIGN  s_0  0.000000)
               (ASSIGN  s_1  0.000000)
               (NEXTSTATE   attente1 )
            )
            (STATE  attente1
                  (PROPERTY  sdlline  538 )
               (PCALL  DW03get_int
                  (PARAMETERASSIGN  data  sdl_signal
                     (DIRECTION  OUT ) (INTEGER ))
                  (PORTINSTANCE  N_algo1_moteur1_rrqb )
                  (PORTINSTANCE  N_algo1_moteur1_clk )
                  (PORTINSTANCE  N_algo1_moteur1_fifout )
                  (PORTINSTANCE  N_algo1_moteur1_empty )
               )
               (CASE
                  (ALT (=  sdl_signal  pid_moteur1_data )
                        (PROPERTY  sdlline  542 )
                     (PCALL  DW03get_real
                        (PARAMETERASSIGN  data  e_0
                           (DIRECTION  OUT ) (REAL ) )
                        (PORTINSTANCE  N_algo1_moteur1_rrqb )
                        (PORTINSTANCE  N_algo1_moteur1_clk )
                        (PORTINSTANCE  N_algo1_moteur1_fifout )
                        (PORTINSTANCE  N_algo1_moteur1_empty )
                     )
                     (ASSIGN  s_0 (-  s_1 (*  0.005000(-  s_1  e_1 ))))
                     (ASSIGN  s_1  s_0 )
                     (ASSIGN  e_1  e_0 )
                     (NEXTSTATE   attente2 )
                  )
               )
            )
            (STATE  attente2
                  (PROPERTY  sdlline  561 )
               (PCALL  DW03get_int
                  (PARAMETERASSIGN  data  sdl_signal
                     (DIRECTION  OUT ) (INTEGER ))
                  (PORTINSTANCE  N_algo1_moteur1_rrqb )
                  (PORTINSTANCE  N_algo1_moteur1_clk )
                  (PORTINSTANCE  N_algo1_moteur1_fifout )
                  (PORTINSTANCE  N_algo1_moteur1_empty )
               )
               (CASE
                  (ALT (=  sdl_signal  pid_moteur1_read )
                        (PROPERTY  sdlline  565 )
                     (CUCALL  put_real  control_channel
                        (PARAMETERASSIGN  sdl_signal  moteur1_pid
                           (DIRECTION  IN ) (INTEGER ))
                        (PARAMETERASSIGN  param_1  s_0
                           (DIRECTION  IN ) (REAL ))
                     )
                     (NEXTSTATE   attente1 )
                  )
               )
            )
         )
      )
   )
)
```

FIG. C.5: Système SOLAR après la synthèse d'interface

Bibliographie

[1] G.R. Andrews and F.B. Schneider, *Concepts and Notations for Concurrent Programming*, Computing Survey, Vol 15, No. 1, pp. 3-42, March 1983.

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

[3] F. Balarin, M. Chiodo *et all*, *Hardware/Software Co-Design of Embedded Systems, The POLIS Approach*, Kluwer Academic Publishers, 1997.

[4] G. Barrett, *Formal Methods Applied to a Floating Point Number System*, IEEE Transactions on Software Engineering, pp. 611-617, May 1989.

[5] E. Barros, W. Rosenstiel and X. Xiong, *A Method for Partitionning UNITY Language in Hardware and Software*, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 220-225, September 1994.

[6] E. Barros and 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.

[7] 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.

[8] M. Belhadj, T. Gautier, P. Le Guernic and 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.

[9] F. Belina, D. Hogrefe and A. Sarma, *SDL with Applications from Protocol Specification*, Prentice Hall Internationnal, 1991.

[10] A. Benveniste and G. Berry, *The Synchronous Approach to Reactive and Real-Time Systems*, Proceedings of the IEEE, Vol. 79, No. 9, pp. 1270-1282, September 1991.

[11] T. Ben Ismail, J. M. Daveau, K. O'Brien and A.A. Jerraya, *A System Level Communication Synthesis Approach for Hardware/Software Systems*, Microprocessors and Microsystems, Vol. 20, pp. 149-157, 1996.

[12] T. Ben Ismail and 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.

[13] T. Ben Ismail G. F. Marchioro and A. A. Jerraya, *Partitionning of VLSI Systems from a High Level Specification*, Techniques et Sciences Informatiques, Vol. 15, No. 8, pp. 1131-1165, 1996.

[14] G. Berry, *Hardware Implementation of Pure ESTEREL*, Proceedings of the ACM Workshop on Formal Methods in VLSI Design, January 1991.

[15] A.D. Birrell and B.J. Nelson, *Implementing remote procedure call*, ACM Transactions on Computer Systems, Vol. 2, No. 1, pp. 39-59, February 1984.

[16] G.V. Bochmann, *Specification Languages for Communication Protocols*, Proceedings of the Conference on Hardware Description Languages, pp. 365-382, April 1993.

[17] T. Bolognesi and E. Brinksma, *Introduction to the ISO Specification language LOTOS*, Computer Networks and ISDN Systems, Vol 14, pp. 25-29, 1987.

[18] T. Bolognesi, E. Najm and P. A. J. Tilanus, *G-LOTOS : a Graphical Language for Concurrent Systems*, Computer Networks and ISDN Systems, Vol 23, pp. 1101-1127, 1994.

[19] I.S. Bonatti and R.J.O. Figuerido, *An Algorithm for the translation of SDL into Synthesizable VHDL*, Current Issue in Electronic Modelling, Vol 3, pp. 99-110, August 1995.

[20] F. Boussinot and R. De Simone, *The ESTEREL Language*, Proceedings of the IEEE, Vol. 79, No. 9, pp. 1293-1304, September 1991.

[21] K. Buchenrieder, *A Prototyping Environnement for Control Oriented Hardware/Software Systems Using State-Charts, Activity-Charts and FPGA*, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 60-65, September 1994.

[22] K. Buchenrieder, A. Sedlmeier and C. Veith, *Hw/Sw Codesign with PRAM using CODES*, Proceedings of the Conference on Hardware Description Languages, pp. 55-68, April 1993.

[23] K. Buchenrieder and C. Veith, *CODES : A Practical Concurent Design environment*, Proceedings of the Internationnal Workshop on Hardware/Software Codesign, October 1992.

[24] K. Buchenrieder and C. Veith, *A Prototyping Environnement 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.

[25] K. Buchenrieder, A. Pyttel and C. Veith, *Mapping StateCharts Models onto an FPGA Based ASIP Architecture*, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 184-189, September 1996.

[26] S. Budkowski and P. Dembinski, *An Introduction to ESTELLE : A Specification language for distributed systems*, Computer Networks and ISDN Systems, Vol 13, No. 2, pp. 2-23, 1987.

[27] R. Camposano and J. Wilberg, *Embedded System Design*, Design Automation for Embedded Systems, Vol. 1, No. 1-2, pp. 5-50, January 1996.

[28] C. Carreras, J.C. López, M.L. López, C. Delgado Kloos, N. Martínez and L. sánchez, *A Codesign Methodology Based on Formal Specifications and High Level Estimation*, Proceedings of the 7th CODES/CASHE Workshop, pp. 28-35, March 1996.

[29] P. H. Chou, R. B. Ortega and G. Borriello, *The Chinook Hardware/Software Co-Synthesis System*, Proceedings of the 8th International Symposium on System Synthesis, pp. 22-27, September 1995.

[30] 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.

[31] D. Comer, *TCP/IP, Architecture, Protocole, Application*, Collection iaa, InterEditions, 1996.

[32] T. Cormem, C. Leiserson, R. Rivest, *Introduction à l'Algorithmique*, Dunod, 1994.

[33] J. M. Daveau, T. Ben-Ismail and 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.

[34] J. M. Daveau, G.F. Marchioro and A. A. Jerraya, *Partition et Synthèse de la Communication pour les Systèmes Mixtes Logiciel/Matériel*, Actes des Séminaires Action Scientifique Codesign, No. 10, pp. 23-31, Novembre 1996.

[35] J. M. Daveau, G.F. Marchioro, T. Ben-Ismail and A. A. Jerraya, *COSMOS : An SDL Based Hardware/Software Codesign Environnement*, Current Issue in Electronic Modelling, Vol. 8, pp. 59-88, December 1996.

[36] J. M. Daveau, G.F. Marchioro, T. Ben-Ismail and A. A. Jerraya, *Protocol Selection and Interface Generation for Hw/Sw Codesign*, IEEE Transactions on VLSI systems, Vol. 5, No. 1, pp. 136-144, March 1997.

[37] J. M. Daveau, G.F. Marchioro, C. A. Valderrama and A. A. Jerraya, *VHDL generation from SDL specifications*, Proceedings of the IFIP Conference on Hardware Description Languages and their Application, pp. 182-201, April 1997.

[38] J. M. Daveau, G.F. Marchioro and A. A. Jerraya, *Hardware/ Software Codesign of an ATM Network Interface Card : a Case Study*, Proceedings of the Sixth International Workshop on Hardware/Software Codesign, March 1998.

[39] C. Delgado Kloos, A. Marín López, T. de Miguel Moro and T. Robles Valladares, *From Lotos to VHDL*, Current Issue in Electronic Modelling, Vol. 3, pp. 111-140, September 1995.

[40] D. Drusinsky and D. Harel, *Using StateCharts for Hardware Description and Synthesis*, IEEE Transactions on Computer-Aided Design, Vol. 8, No. 7, pp. 798-807, July 1989.

[41] N.D. Dutt, J. Hwee Cho and T. Hadley, *A User Interface for VHDL Behavioural Modelling*, Proceedings of the Conference on Hardware Description Languages, pp. 375-393, April 1991.

[42] W. Ecker, *Using VHDL for HW/SW Co-Specification*, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 500-505, September 1993.

[43] W. Ecker, *Semi Dynamic Scheduling of Synchronisation Mechanisms*, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 374-379, September 1995.

[44] W. Ecker, M. Glesner and A. Vombach, *Protocol merging : A VHDL Based Method for Clock Cycle Minimising and Protocol Preserving Scheduling of IO Operations*, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 624-629, September 1994.

[45] W. Ecker and M. Huber, *VHDL Based Communication and Synchronization Synthesis*, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 458-462, September 1995.

[46] P. Eles, Z. Peng and 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.

[47] R. Ernst, J. Henkel and T. Benner, *Hardware/Software Co-Synthesis for Microcontrollers*, IEEE Design & Test of Computers, Vol. 10 No. 4, pp. 64-75, December 1993.

[48] O. Færgemand and A. Olsen, *Introduction to SDL-92*, Computer Networks and ISDN Systems, Vol 26, pp. 1143-1167, 1994.

[49] M. Faci and L. Logrippo, *Specifying Hardware Systems in LOTOS*, Proceedings of the Conference on Hardware Description Languages, pp. 305-312, April 1993.

[50] D. Filo, D. Ku, C. N. Coelho and G. de Micheli, *Interface Optimisation for Concurrent Systems Under Timing Constraints*, IEEE Transactions on VLSI, Vol. 1, pp. 268-281, September 1993.

[51] T. Gautier and P. Le Guernic, *Signal, A Declarative Language for Synchronous Programming of Real Time Systems*, Computer Science, Formal Languages and Computer Architecture, No. 274, 1987.

[52] M. Gasteier and M. Glesner, *Bus Based Communication Synthesis on System Level*, Proceedings of Fourth International on Hardware/Software Codesign, pp. 65-70, March 1996.

[53] D. Gajski, F. Vahid and S. Narayan, *A System Design Methodology : Executable Specification Refinement*, Proceedings of the European Design and Test Conference, pp. 458-463, February 1994.

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

[55] C. H. Gebotys, *Optimal Scheduling and Allocation of Embedded VLSI Chips*, Proceedings of the IEEE Design Automation Conference, pp. 116-120, June 1992.

[56] R. Gerndt, *A Case Study in Co-Design of Communication Controller*, Proceedings of the 7th CODES/CASHE Workshop, pp. 104-110, March 1996.

[57] W. Glunz, T. Kruse, T. Rossel and D. Monjau, *Integrating SDL and VHDL for System Level Specification*, Proceedings of the Conference on Hardware Description Languages, pp. 175-192, April 1993.

[58] J. Gong, D. Gajski and S. Bakshi, *Model Refinement for Hardware-Software Codesign*, Proceedings of the European Design and Test Conference, pp. 270-274, March 1996.

[59] R. K. Gupta, C. N. Coelho and 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.

[60] R. K. Gupta, C.N. Coelho and G. de Michelli, *Program Implementation Schemes for Hardware Software Systems*, IEEE Design & Test of Computers, Vol. 27, No. 1, pp. 48-55, January 1994.

[61] R. K. Gupta and G. de Michelli, *Hardware/Software Cosynthesis for Digital Systems*, IEEE Design & Test of Computers, Vol. 10 No. 4, pp. 29-41, December 1993.

[62] R. K. Gupta and 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.

[63] R. K. Gupta and 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.

[64] P. Le Guernic, T. Gautier, M. Le Borgne and C. Lemaire, *Programming Real-Time Applications with SIGNAL*, Proceedings of the IEEE, Vol. 79, No. 9, pp. 1321-1336, September 1991.

[65] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud, *The Synchronous Data Flow Programming Language LUSTRE*, Proceedings of the IEEE, Vol. 79, No. 9, pp. 1305-1320, September 1991.

[66] N. Halbwachs, F. Lagnier and C. Ratel, *Programming and Verifying Real-Time Systems by means of the Synchronous Data-Flow Language LUSTRE*, IEEE Transactions on Software Engineering, Vol. 18, No. 9, September 1992.

[67] N. Halbwachs, *Synchronous programming of reactive systems*, Kluwer Academic Publishers, 1993.

[68] D. Harel, *Statechart : A Visual Formalism for Complex systems*, Science of Programming, Vol 8, pp. 231-274, 1987.

[69] M. Harrand, *A Single Chip Videophone Video Encoder/Decoder*, Proceedings of IEEE International Solid State Circuits Conference, pp. 292-293, February 1995.

[70] J. Henkel, T. Benner R. Ernst, W. Ye, N. Serafimov and 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.

[71] J. Henkel, R. Ernst, U. Holtman and T. Benner, *Adaptation of Partitioning and High Level synthesis in Hardware/Software Co-Synthesis*, Proceedings of IEEE International Conference on Computer Aided Design, pp. 96-100, November 1994.

[72] C. A. R. Hoare, *Communicating Sequential Processes*, Communication of the ACM, Vol 21, No. 8, pp. 666-677, August 1978.

[73] J. Hou and W. Wolf, *Process Partitionning for Distributed Embedded Systems*, Proceedings of the 7th CODES/CASHE Workshop, pp. 70-76, March 1996.

[74] C. Y. Huang, Y. S. Chen, Y. L. Lin and Y. C. Hsu, *Data Path Allocation Based on Bipartite Weighted Matching*, Proceedings of the IEEE Design Automation Conference, pp. 499-504, June 1990.

[75] International Organisation for Standardization, *ESTELLE, A Formal Description Technique Based on an Extended State Transition Model*, ISO/IEC 9074, 1991.

[76] International Organisation for Standardization, *LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observationnal behaviour*, ISO/IEC 8807, 1989.

[77] A. Jantsch, P. Ellervee, J. Öberg, A. Hemani and H. Tenhunen, *Hardware/Software Partitionning and Minimizing Memory Interface Traffic*, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 226-231, February 1994.

[78] A. A. Jerraya and K. O'Brien, *SOLAR : An Intermediate Format for System-Level Modelling and Synthesis*, in "Computer Aided Software/Hardware Engineering", J. Rozenblit, K. Buchenrieder (eds), IEEE Press, Piscataway, N. J., pp. 147-175, 1994.

[79] A. A. Jerraya, H. Ding, P. Kission, M. Rahmouni, *Behavioural Synthesis and Component Reuse with VHDL*, Kluwer Academic publishers, 1997.

[80] A. Jirachiefpattana and R. Lai, *A Rapid Protocol Prototyping Development System*, Proceedings of 6th International Workshop on Rapid System Prototyping, pp. 118-124, June 1995.

[81] S. Kahlert, J. U. Knäbchen and D. Monjau, *Design and Analysis of Heterogeneous Systems Using Graphically Oriented Methods : A Case Study*, Proceedings of SASIMI, pp. 23-32, October 1993.

[82] A. Kalavade and 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.

[83] A. Kalavade and E.A. Lee, *A Global Criticality/Local Phase Driven Algorithm for the Constrained Hardware/Software Partitionning Problem*, Proceedings of the Third International Workshop on Hardware/Software Codesign, pp. 42-48, September 1994.

[84] A. Kalavade and E.A. Lee, *The Extended Partitionning Problem : Hardware/Software Mapping and Implementation-Bin Selection*, Proceedings of sixth International Workshop on Rapid System Prototyping, pp. 12-18, June 1995.

[85] S. A. Khan and V. K. Madisetti, *System Partitionning of MCMs for Low Power*, IEEE Design & Test of Computers, Vol. 12, No. 1, pp. 41-52, Spring 1995.

[86] P. Kission, H. Ding and A. A. Jerraya, *VHDL Based Design Methodology for Hierarchy and Component Reuse at the Behavioural Level*, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 470-475, September 1995.

[87] M. J. Knieser and C. A. Papachristou, *COMET : A Hardware-Software Codesign Methodology*, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 178-183, September 1996.

[88] G. Koch, U. Kebschull and W. Rosenstiel, *A Prototyping Environmement for Hardware/Software Codesign in the COBRA Project*, Proceedings of the Third International Workshop on Hardware/Software Codesign, pp. 10-16, September 1994.

[89] C. Krueger, *Software reuse*, ACM computer survey, Vol. 24, No. 2, pp. 131-183, June 1992.

[90] O. Kyas, *ATM Networks*, Internationnal Thomson Publishing, 1995.

[91] B. Lin and S. Vercauteren, *Synthesis of Concurrent System Interface Modules with Automatic Protocol Conversion Generation*, Proceedings of IEEE International Conference on Computer Aided Design, pp. 395-399, November 1994.

[92] B. Lin, S. Vercauteren and H. De Man, *Embedded Architecture Co-Synthesis and System Integration*, Proceedings of the 7th CODES/CASHE Workshop, pp. 2-9, March 1996.

[93] L. Logrippo, M. Faci and M. Haj-Hussein, *An Introduction to LOTOS : Learning by Examples*, Computer Networks and ISDN Systems, Vol 23, pp. 325-342, 1992.

[94] B. Lutter, W. Glunz and F.J. Ramming, *Using VHDL for Simulation of SDL Specifications*, Proceedings of the European Design Automation Conference with Euro-VHDL (EuroDAC), pp. 630-635, September 1992.

[95] G. F. Marchioro, J. M. Daveau and A. A. Jerraya, *Transformational Partitionning for Co-Design of Multiprocessors Systems*, Proceedings of IEEE International Conference on Computer Aided Design, pp. 508-515, November 1997.

[96] J. Madsen and B. Hald, *An Approach to Interface Synthesis*, Proceedings of the 8th International Symposium on System Synthesis, pp. 16-21, September 1995.

[97] A.J. Martin, *Synthesis of Asynchronous VLSI Circuits*, Formal Methods for VLSI Design, J. Staunstrup (eds), North Holland, 1990.

[98] P. Michel, U. Lauther and P. Duzy, *The Synthesis Approach to Digital System Design*, Kluwer Academic Publishers, 1992.

[99] R. Milner, *Calculi for Synchrony and Asynchrony*, Theoritical Computer Science, Vol 23, pp 267-310, 1983.

[100] 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.

[101] P. Le Moenner, L. Perraudeau, P. Quinton, S. Rajopadhye and T. Risset, *Generating Regular Arithmetic Circuit with ALPHAHARD*, Proceedings of International Conference on Massively Parallel Computing Systems, May 1996.

[102] S. Narayan and D. Gajski, *Features Supporting System-Level Specification in HDLs*, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 540-545, September 1993.

[103] S. Narayan and D. Gajski, *Synthesis of System-Level Bus Interfaces*, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 395-399, February 1994.

[104] S. Narayan and D. Gajski, *Interfacing Incompatible Protocols Using Interface Process Generation*, Proceedings of the IEEE Design Automation Conference, pp. 468-473, June 1995.

[105] S. Narayan and D. Gajski, *Rapid Performance Estimation For System Design*, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 206-211, September 1996.

[106] S. Narayan, F. Vahid and D. Gajski, *Translating System Specifications to VHDL*, Proceedings of the European Design Automation Conference (EDAC), pp. 391-394, February 1991.

[107] S. Narayan, F. Vahid and D. Gajski, *System Specification and Synthesis with the SpecCharts Language*, Proceedings of the Internationnal Conference on Computer Aided Design (ICCAD), pp. 266-269, November 1991.

[108] W. Nebel and G. Schumacher, *Object Oriented Hardware Modelling - Where to apply and what are the objects*, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 428-433, September 1996.

[109] R. Niemann and P. Marwedel, *Hardware/Software Partitionning using Integer Programming*, Proceedings of the European Design and Test Conference, pp. 473-479, March 1996.

[110] J. Öberg, A. Kumar and A. Hemani, *Grammar-Based Hardware Synthesis of Data Communication Protocols*, Proceedings of Fourth International on Hardware/Software Codesign, pp. 14-19, March 1996.

[111] R. B. Ortega and G. Borriello, *Communication Synthesis for Embedded Systems with Global Considerations*, Proceedings of Fifth International on Hardware/Software Codesign, pp. 69-73, March 1997.

[112] Z. Peng and K. Kuchcinski, *An Algorithm for Partitionning of Application Specific Systems*, Proceedings of the European Design Automation Conference (EDAC), pp. 316-321, February 1993.

[113] P. Paulin, J. Fréhel, M. Harrand, E. Berrebi, C. Liem, F. Naçabal and J. C. Herluison, *High Level Synthesis and Codesign Methods : An Application to a Videophone Codec*, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 444-451, September 1995.

[114] B. Potter, J. Sinclair and D. Till, *An Introduction to Formal Specification and Z*, Prentice Hall Internationnal Series in Computer Science, CAR Hoare Series Editor, 1991.

[115] M. De Pryker, *ATM, Mode de Transfert Asynchrone*, Masson / Prentice Hall, 1995.

[116] O. Pulkkinen and K. Kronlöf, *Integration of SDL and VHDL for High Level Digital Design*, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 624-629, September 1992.

[117] B. Rahardjo and R. D. Mc Leod, *Design and Analysis of a Conterflow Pipeline Processor in SDL*, Journal of Microelectronic Systems integration, Vol 4, No. 1, pp. 33-43, 1996.

[118] A. P. Ravn and J. Staunstrup, *Interface Models*, Proceedings of the IEEE CODES/CASHE workshop, pp. 157-164, September 1994.

[119] N. L. Rethman and P. A. Wilsey, *RAPID, : A Tool for Hardware/Software Tradeoff Analysis*, Proceedings of the Conference on Hardware Description Languages, April 1993.

[120] M. Romdhani, P. Chambert, A. Jeffroy, P. de Chazelles and A. A. Jerraya, *Composing ActivityCharts/StateCharts, SDL and SAO Specifications for Codesign in Avionics*, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 585-590, September 1995.

[121] M. Romdhani R. P. Hautbois, A. Jeffroy, P. de Chazelles and A. A. Jerraya, *Evaluation and Composition of Specification Languages, an Industrial Point of View*, Proceedings of the IFIP Conference on Hardware Description Languages, pp. 519-523, September 1995.

[122] K. V. Rompaey, D Verkest, I. Bolsens and H. De Man, *CoWare, A Design Environnement for Heterogeneous Hardware/Software Systems*, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 252-257, September 1996.

[123] K. Salah and R. Probert, *A Service-Based Method for the Synthesis of Communication Protocols*, International Journal of Mini and Microcomputers, Vol. 12, No. 3, pp. 97-103, April 1990.

[124] R. Saracco and 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.

[125] G. Schumacher and W. Nebel, *Inheritance Concept for Signals in Object Oriented Extention to VHDL*, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 428-435, September 1995.

[126] P. Slock, S. Wuytack, F.Catthoor and G. de Jong, *Fast end Extensive System-Level Memory Exploration for ATM Applications*, Proceedings of the 10th International Symposium on System Synthesis, pp 74-81, September 1997.

[127] J. M. Spivey, *An Introduction to Z Formal Specifications*, Software Engineering Journal, pp. 40-50, January 1989.

[128] M. B. Srivastava and R. W. 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, Vol. 14, No. 6, pp. 676-693, June 1995.

[129] S. Swamy, A. Molin, and B. Covnot, *OO-VHDL Object Oriented Extention to VHDL*, IEEE Computer, Vol. 28, No. 10, pp. 18-26, October 1995.

[130] A. Takach and W. Wolf, *Scheduling Constraint Generation for communicating Processes*, IEEE Transactions on VLSI Systems, Vol. 3, No. 2, pp. 215-230 June 1995.

[131] D. E. Thomas, J.K. Adams and H. schmit, *A Model and Methodology for Hardware/Software Codesign*, IEEE Design & Test of Computers, Vol. 10 No. 4, pp. 6-15, December 1993.

[132] K. J. Turner, *Using Formal Description Techniques, An Introduction to Estelle, Lotos and Sdl*, John Wiley & Sons, 1993.

[133] F. Vahid, *Port Calling : A Transformation for Reducing I/O during Multi-Package Functionnal Partitionning*, Proceedings of the 10th International Symposium on System Synthesis, pp 107-112, September 1997.

[134] F. Vahid and D. Gajski, *Specification Partitioning For System Design*, Proceedings of the IEEE Design Automation Conference, pp. 219-224, June 1992.

[135] F. Vahid and D. Gajski, *Closeness Metrics for System Level Functional Partitioning*, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 328-333, September 1995.

[136] F. Vahid, J. Gong and D. Gajski, *A Binary Constraint Search Algorithm for Minimizing Hardware during Hardware/Software Partitioning*, Proceedings of the European Design Automation Conference with Euro-VHDL, pp. 214-219, September 1994.

[137] F. Vahid, S. Narayan and D. Gajski, *SpecCharts : A Language for System Level Synthesis*, Proceedings of the Conference on Hardware Description Languages, pp. 145-154, April 1991.

[138] F. Vahid, S. Narayan and 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.

[139] F. Vahid and L. Tauro, *An Object Oriented Communication Library for Hardware Software Codesign*, Proceedings of Fifth International on Hardware/Software Codesign, pp. 81-85, March 1997.

[140] C. A. Valderrama, A. Changuel, P.V. Raghavan, M. Abid, T. Ben Ismail and A.A. Jerraya, *A Unified Model for Cosimulation and Cosynthesis of Mixed Hardware/Software Systems*, Proceedings of the European Design and Test Conference, pp. 180-184, March 1995.

[141] C. A. Valderrama, F. Naçabal, P. Paulin and A. A. Jerraya, *Automatic Generation of Interface for Distributed C-VHDL Cosimulation of Embedded Systems : An Industrial Experience*, Proceedings of IEEE International Workshop on Rapid System Prototyping, pp. 72-77, June 1996.

[142] J. Vanhoof, K. Vanrompaey, I. Bolsens, G. Goosens and H. de Man, High-Level Synthesis for Real Time Digital Signal Processing, Kluwer Academic Publishers, 1993.

[143] P. Vanbekbergen, C. Ykman-Couvreur, B. Lin and H. de Man, *A Generalised Signal Transition Graph Model for Specification of Complex Interfaces*, Proceedings of the European Design & Test Conference, pp. 378-384, February 1994.

[144] S. Vercauteren, B. Lin and 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.

[145] J. Wilberg, R. Camposano and 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.

[146] W. Wolf, *Hardware/Software Co-Design of Embedded Systems*, Proceedings of the IEEE, Vol 82, No. 7, pp. 967-989, 1994.

[147] J. Wytrebwicz and S. Budkowski, *Communication Protocols Implemented in Hardware : VHDL generation from Estelle*, Proceedings of VHDL Forum for CAD in Europe, pp. 29-40, 1996.

[148] J. Wytrebwicz and S. Budkowski, *Communication Protocols Implemented in Hardware : VHDL Generation from Estelle*, Current Issue in Electronic Modelling, Vol 3, pp. 77-98, September 1995.

[149] X. Xiong, P. Gutberlet and W. Rosenstiel, *Automatic Generation of Interprocess Communication in the PARAGON System*, Proceedings of IEEE International Workshop on Rapid System Prototyping, pp. 24-29, June 1996.

[150] T. Yen and W. Wolf, *Communication Synthesis for Distributed Embedded Systems*, Proceedings of the International Conference on Computer Aided Design, pp. 288-294, November 1995

[151] ITU-T *Z.100 Functionnal Specification and Description Language*, Recommandation Z.100 - Z.104, March 1993.

Résumé

Au fur et à mesure que la complexité des systèmes s'accroît, il devient nécessaire de définir de nouvelles méthodes permettant de la gérer. Une des façons de maîtriser cette complexité est d'élever le niveau d'abstraction des spécifications en utilisant des langages de spécification systèmes. D'un autre coté, l'élévation du niveau d'abstraction augmente le fossé entre les concepts utilisés pour la spécification (processus communicants, communication abstraite) et ceux utilisés par les langages de description de matériel. Bien que ces langages soient bien adaptés à la spécification et à la validation de systèmes complexes, les concepts qu'ils manipulent ne sont pas aisément transposables sur ceux des langages de description de matériels. Il est donc nécessaire de définir de nouvelles méthodes permettant une synthèse efficace à partir de spécifications systèmes. Le sujet de cette thèse est la présentation d'une approche de génération de code C et VHDL à partir de spécifications systèmes en SDL. Cette approche résout la principale difficulté rencontrée par les autres approches, à savoir la communication inter-processus. La communication SDL peut être traduite en VHDL en vue de la synthèse. Cela est rendu possible par l'utilisation d'une forme intermédiaire qui supporte un modèle de communication général qui autorise la représentation pour la synthèse de la plupart des schémas de communication. Cette forme intermédiaire permet d'appliquer au système un ensemble d'étapes de raffinement pour obtenir la solution désirée. La principale étape de raffinement, appelée synthèse de la communication, détermine le protocole et les interfaces utilisés par les différents processus pour communiquer. La spécification raffinée peut être traduite en C et VHDL pour être utilisée par des outils du commerce. Nous illustrons la faisabilité de cette approche par une application à un système de télécommunication : le protocole TCP/IP sur ATM.

Mots clés

Spécifications systèmes, synthèse système, SDL, synthèse de la communication, synthèse d'interfaces, VHDL.

Abstract

As the system complexity grows there is a need for new methods to handle large system design. One way to manage that complexity is to rise the level of abstraction of the specifications by using system level description languages. On the other side, as the level of abstraction rise the gap between the concepts used for the specifications at the system level (communication channels, interacting processes, data types) and those used for hardware synthesis becomes wider. Although these languages are well suited for the specification and validation of complex real time distributed systems, the concepts manipulated are not easy to map onto hardware description languages. It is thus necessary to defines methods for system level synthesis enabling efficient synthesis from system level specifications. The subject of this thesis is the presentation of a new approach of generation of C and VHDL code from system level specifications in SDL. This approach solves the main problem encountered by previous approach : inter process communications. SDL communication can be translated in VHDL for synthesis. This is achieved by the use of a powerful intermediate form that support the modelling for synthesis of a wide range of communication schemes. This intermediate form allows to apply to the system a set of transformations in order to obtain the desired solution. The main refinement step, called communication synthesis is aimed at fixing the protocol and interface used by the different processes to communicate. The refined specification can be translated in C and VHDL and synthesised by commercial tools. We illustrate the feasibility of this approach through an application to a telecommunication example : the TCP/IP over ATM protocol.

Key words

System specifications, system synthesis, SDL, communication synthesis, interface synthesis, VHDL.