THÈSE

présentée à

L'UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE

pour obtenir le titre de

DOCTEUR en INFORMATIQUE

par

David GALINEC

Exécution asynchrone de programmes synchrones par transformations automatiques: application au traitement d'images temps-réel

Thèse qui sera soutenue le 27 janvier 1997, devant la commission d'examen:

Rapporteurs:	Dominique MERY	Université Henri Poincaré, Nancy 1 & IUF
	Edwige PISSALOUX	PSI-La3I, Université de Rouen
Examinateurs:	Mireille CLERBOUT	LIFL, Université de Lille 1
	Marie-Jean COLAÏTIS	THOMSON MULTIMEDIA, Rennes
	Jean-Luc DEKEYSER	LIFL, Université de Lille 1
	Philippe MARQUET	LIFL, Université de Lille 1

UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE
LIFL - URA 369 CNRS - Bât. M3 - UFR IEEA - 59655 VILLENEUVE D'ASCQ CEDEX
Tél: (33) 03 20 43 44 92 - Fax: (33) 03 20 43 65 66 - E_mail: direction@lifl.fr

UNIVERSITE DES SCIENCES
ET TECHNOLOGIES DE LILLE

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M. H. LEFEBVRE, M. PARREAU

PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE, LELONG, LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU, ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

PROFESSEUR EMERITE

M. A. LEBRUN

ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PARREAU, J. LOMBARD, M. MIGEON, J. CORTOIS, A.DUBRULLE

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

M. P. LOUIS

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CHAMLEY Hervé	Géotechnique
M. CONSTANT Eugène	Electronique
M. ESCAIG Bertrand	Physique du solide
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. LABLACHE COMBIER Alain	Chimie
M. LOMBARD Jacques	Sociologie
M. MACKE Bruno	Physique moléculaire et rayonnements atmosphériques

M. TURREL Georges	Spectrochimie infrarouge et raman
M. VANDIJK Hendrik	
Mme VAN ISEGHEM Jeanine	Modélisation,calcul scientifique,statistiques
M. VANDORPE Bernard	Chimie minérale
M. VASSEUR Christian	Automatique
M. VASSEUR Jacques	Biologie
Mme VIANO Marie Claude	
M. WACRENIER Jean Marie	Electronique
M. WARTEL Michel	Chimie inorganique
M. WATERLOT Michel	géologie générale
M. WEICHERT Dieter	Génie mécanique
M. WERNER Georges	Informatique théorique
M. WIGNACOURT Jean Pierre	
M. WOZNIAK Michel	Spectrochimie
Mme ZINN JUSTIN Nicole	Algèbre

M. MIGEON Michel	EUDIL
M. MONTREUIL Jean	Biochimie
M. PARREAU Michel	Analyse
M. TRIDOT Gabriel	Chimie appliquée

PROFESSEURS - 1ère CLASSE

M. BACCHUS Pierre	Astronomie
M. BIAYS Pierre	Géographie
M. BILLARD Jean	Physique du Solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean Pierre	Chimie-Physique
M. BOSCQ Denis	Probabilités
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BRASSELET Jean Paul	Géométrie et topologie
M. BREZINSKI Claude	Analyse numérique
M. BRIDOUX Michel	Chimie Physique
M. BRUYELLE Pierre	Géographie
M. CARREZ Christian	Informatique
M. CELET Paul	Géologie générale
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. CROSNIER Yves	Electronique
Mme DACHARRY Monique	Géographie
M. DAUCHET Max	Informatique
M. DEBOURSE Jean Pierre	Gestion des entreprises
M. DEBRABANT Pierre	Géologie appliquée
M. DECLERCQ Roger	Sciences de gestion
M. DEGAUQUE Pierre	Electronique
M. DESCHEPPER Joseph	Sciences de gestion
Mme DESSAUX Odile	Spectroscopie de la réactivité chimique
M. DHAINAUT André	Biologie animale
Mme DHAINAUT Nicole	Biologie animale
M. DJAFARI Rouhani	Physique
M. DORMARD Serge	Sciences Economiques
M. DOUKHAN Jean Claude	Physique du solide
M. DUBRULLE Alain	Spectroscopie hertzienne
M. DUPOUY Jean Paul	Biologie
M. DYMENT Arthur	Mécanique
M. FOCT Jacques Jacques	Métallurgie
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie structurale
M. FRONTIER Serge	Ecologie numérique
M. GLORIEUX Pierre	Physique moléculaire et rayonnements atmosphériques
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie-Physique
M. GRANELLE Jean Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILBAULT Pierre	Physiologie animale
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. HENRY Jean Pierre	Génie mécanique
M. HERMAN Maurice	Physique spatiale
M. LACOSTE Louis	Biologie Végétale
M. LANGRAND Claude	Probabilités et statistiques

M. LATTEUX Michel	Informatique
M. LAVEINE Jean Pierre	Paléontologie
Mme LECLERCQ Ginette	Catalyse
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique atomique et moléculaire
M. LEROY Jean Marie	Spectrochimie
M. LHENAFF René	Géographie
M. LHOMME Jean	Chimie organique biologique
M. LOUAGE Françis	Electronique
M. LOUCHEUX Claude	Chimie-Physique
M. LUCQUIN Michel	Chimie physique
M. MAILLET Pierre	Sciences Economiques
M. MAROUF Nadir	Sociologie
M. MICHEAU Pierre	Mécanique des fluides
M. PAQUET Jacques	Géologie générale
M. PASZKOWSKI Stéfan	Mathématiques
M. PETIT Francis	Chimie organique
M. PORCHET Maurice	Biologie animale
M. POUZET Pierre	Modélisation - calcul scientifique
M. POVY Lucien	Automatique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. RAMAN Jean Pierre	Sciences de gestion
M. SALMER Georges	Electronique
M. SCHAMPS Joël	Spectroscopie moléculaire
Mme SCHWARZBACH Yvette	Géométrie
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
M. SLIWA Henri	Chimie organique
M. SOMME Jean	Géographie
Melle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. THIEBAULT François	Sciences de la Terre
M. THOMAS Jean Claude	Géométrie - Topologie
M. THUMERELLE Pierre	Démographie - Géographie humaine
M. TILLIEU Jacques	Physique théorique
M. TOULOTTE Jean Marc	Automatique
M. TREANTON Jean René	Sociologie du travail
M. TURRELL Georges	Spectrochimie infrarouge et raman
M. VANEECLOO Nicolas	Sciences Economiques
M. VAST Pierre	Chimie inorganique
M. VERBERT André	Biochimie
M. VERNET Philippe	Génétique
M. VIDAL Pierre	Automatique
M. WALLART Françis	Spectrochimie infrarouge et raman
M. WEINSTEIN Olivier	Analyse économique de la recherche et développement
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ABRAHAM Francis	
M. ALLAMANDO Etienne	Composants électroniques
M. ANDRIES Jean Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BALL Steven	Génétique
M. BART André	Biologie animale
M. BASSERY Louis	Génie des procédés et réactions chimiques
Mme BATTIAU Yvonne	Géographie
M. BAUSIERE Robert	Systèmes électroniques
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique atomique et moléculaire
M. BERNAGE Pascal	Physique atomique, moléculaire et du rayonnement
M. BERTHOUD Arnaud	Sciences Economiques
M. BERTRAND Hugues	Sciences Economiques
M. BERZIN Robert	Analyse
M. BISKUPSKI Gérard	Physique de l'état condensé et cristallographie
M. BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie végétale
M. BOHIN Jean Pierre	Biochimie métabolique et cellulaire
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie civil
M. BOIVIN Jean Claude	Spectrochimie
M. BOUCHER Daniel	Physique
M. BOUQUELET Stéphane	Biologie appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BROCARD Jacques	Chimie
Mme BROUSMICHE Claudine	Paléontologie
M. BUISINE Daniel	Mécanique
M. CAPURON Alfred	Biologie animale
M. CARRE François	Géographie humaine
M. CATTEAU Jean Pierre	Chimie organique
M. CAYATTE Jean Louis	Sciences Economiques
M. CHAPOTON Alain	Electronique
M. CHARET Pierre	Biochimie structurale
M. CHIVE Maurice	Composants électroniques optiques
M. COMYN Gérard	Informatique théorique
Mme CONSTANT Monique	Composants électroniques et optiques
M. COQUERY Jean Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique nucléaire et corpusculaire
M. COUTURIER Daniel	Chimie organique
M. CRAMPON Norbert	Tectolique géodynamique
M. CURGY Jean Jacques	Biologie
M. DANGOISSE Didier	Physique théorique
M. DE PARIS Jean Claude	Analyse
M. DECOSTER Didier	Composants électroniques et optiques
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELAHAYE Jean Paul	Informatique
M. DELORME Pierre	Physiologie animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
Mme DEMUYNCK Claire	Physique atomique, moléculaire et du rayonnement
M. DENEL Jacques	Informatique
M. DEPREZ Gilbert	Physique du solide - christallographie

M. DERIEUX Jean Claude	Microbiologie
M. DERYCKE Alain	Informatique
M. DESCAMPS Marc	Physique de l'état condensé et cristallographie
M. DEVRAINNE Pierre	Chimie minérale
M. DEWAILLY Jean Michel	Géographie humaine
M. DHAMELINCOURT Paul	Chimie physique
M. DI PERSIO Jean	Physique de l'état condensé et cristallographie
M. DUBAR Claude	Sociologie démographique
M. DUBOIS Henri	Spectroscopie hertzienne
M. DUBOIS Jean Jacques	Géographie
M. DUBUS Jean Paul	Spectrométrie des solides
M. DUPONT Christophe	Vie de la firme
M. DUTHOIT Bruno	Génie civil
Mme DUVAL Anne	Algèbre
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FARVACQUE Jean Louis	Physique de l'état condensé et cristallographie
M. FAUQUEMBERGUE Renaud	Composants électroniques
M. FELIX Yves	Mathématiques
M. FERRIERE Jacky	Tectonique - Géodynamique
M. FISCHER Jean Claude	Chimie organique, minérale et analytique
M. FONTAINE Hubert	Dynamique des cristaux
M. FORSE Michel	Sociologie
M. GADREY Jean	Sciences économiques
M. GAMBLIN André	Géographie urbaine, industrielle et démographie
M. GOBLOT Rémi	Algèbre
M. GOURIEROUX Christian	Probabilités et statistiques
M. GREGORY Pierre	I.A.E.
M. GREMY Jean Paul	Sociologie
M. GREVET Patrice	Sciences Economiques
M. GRIMBLOT Jean	Chimie organique
M. GUELTON Michel	Chimie physique
M. GUICHAOUA André	Sociologie
M. HAIMAN Georges	Modélisation,calcul scientifique, statistiques
M. HOUDART René	Physique atomique
M. HUEBSCHMANN Johannes	Mathématiques
M. HUTTNER Marc	Algèbre
M. ISAERT Noël	Physique de l'état condensé et cristallographie
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et statistiques
M. JEAN Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KOENIG Gérard	Sciences de gestion
M. KOSTRUBIEC Benjamin	Géographie
M. KREMBEL Jean	Biochimie
Mme KRIFA Hadjila	Sciences Economiques
M. LANGEVIN Michel	Algèbre
M. LASSALLE Bernard	Embryologie et biologie de la différenciation
M. LE MEHAUTE Alain	Modélisation,calcul scientifique,statistiques
M. LEBFEVRE Yannic	Physique atomique,moléculaire et du rayonnement
M. LECLERCQ Lucien	Chimie physique
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Marc	Composants électroniques et optiques
M. LEFEVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
M. LEGRAND Michel	Astronomie - Météorologie
M. LEGRAND Pierre	Chimie
Mme LEGRAND Solange	Algèbre
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne

M. LE MAROIS Henri	Vie de la firme
M. LEMOINE Yves	Biologie et physiologie végétales
M. LESCURE François	Algèbre
M. LESENNE Jacques	Systèmes électroniques
M. LOCQUENEUX Robert	Physique théorique
Mme LOPES Maria	Mathématiques
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU François	Sciences économiques
M. MAHIEU Jean Marie	Optique - Physique atomique
M. MAIZIERES Christian	Automatique
M. MANSY Jean Louis	Géologie
M. MAURISSON Patrick	Sciences Economiques
M. MERIAUX Michel	EUDIL
M. MERLIN Jean Claude	Chimie
M. MESMACQUE Gérard	Génie mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MOCHE Raymond	Modélisation,calcul scientifique,statistiques
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie organique
M. MORE Marcel	Physique de l'état condensé et cristallographie
M. MORTREUX André	Chimie organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
M. NIAY Pierre	Physique atomique,moléculaire et du rayonnement
M. NICOLE Jacques	Spectrochimie
M. NOTELET Francis	Systèmes électroniques
M. PALAVIT Gérard	Génie chimique
M. PARSY Fernand	Mécanique
M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. PERTUZON Emile	Physiologie animale
M. PETIT Daniel	Biologie des populations et écosystèmes
M. PLIHON Dominique	Sciences Economiques
M. PONSOLLE Louis	Chimie physique
M. POSTAIRE Jack	Informatique industrielle
M. RAMBOUR Serge	Biologie
M. RENARD Jean Pierre	Géographie humaine
M. RENARD Philippe	Sciences de gestion
M. RICHARD Alain	Biologie animale
M. RIETSCH François	Physique des polymères
M. ROBINET Jean Claude	EUDIL
M. ROGALSKI Marc	Analyse
M. ROLLAND Paul	Composants électroniques et optiques
M. ROLLET Philippe	Sciences Economiques
Mme ROUSSEL Isabelle	Géographie physique
M. ROUSSIGNOL Michel	Modélisation,calcul scientifique,statistiques
M. ROY Jean Claude	Psychophysiologie
M. SALERNO Françis	Sciences de gestion
M. SANCHOLLE Michel	Biologie et physiologie végétales
Mme SANDIG Anna Margarette	
M. SAWERYSYN Jean Pierre	Chimie physique
M. STAROSWIECKI Marcel	Informatique
M. STEEN Jean Pierre	Informatique
Mme STELLMACHER Irène	Astronomie - Météorologie
M. STERBOUL François	Informatique
M. TAILLIEZ Roger	Génie alimentaire
M. TANRE Daniel	Géométrie - Topologie
M. THERY Pierre	Systèmes électroniques
Mme TJOTTA Jacqueline	Mathématiques
M. TOURSEL Bernard	Informatique
M. TREANTON Jean René	Sociologie du travail

M. TURREL Georges	Spectrochimie infrarouge et raman
M. VANDIJK Hendrik	
Mme VAN ISEGHEM Jeanine	Modélisation,calcul scientifique,statistiques
M. VANDORPE Bernard	Chimie minérale
M. VASSEUR Christian	Automatique
M. VASSEUR Jacques	Biologie
Mme VIANO Marie Claude	
M. WACRENIER Jean Marie	Electronique
M. WARTEL Michel	Chimie inorganique
M. WATERLOT Michel	géologie générale
M. WEICHERT Dieter	Génie mécanique
M. WERNER Georges	Informatique théorique
M. WIGNACOURT Jean Pierre	
M. WOZNIAK Michel	Spectrochimie
Mme ZINN JUSTIN Nicole	Algèbre

Résumé

Le travail réalisé dans le cadre de cette thèse porte sur la transformation automatique de programmes synchrones en vue de leur implémentation asynchrone sur les architectures hétérogènes distribuées. Dans un contexte applicatif nous nous sommes plus particulièrement intéressés aux systèmes de traitements temps-réel de l'image. La démarche que nous avons suivie se base sur une phase préliminaire de conception et validation des systèmes temps-réel sous l'hypothèse de synchronisme. En pratique, nous avons utilisé un outil existant: le langage SIGNAL. Nous avons développé un ensemble de processus de transformation permettant d'automatiser l'implémentation de ces systèmes sous la forme de processus séquentiels communicants. Les caractéristiques de ce travail sont de garantir a priori la sûreté des programmes en transformant des systèmes prouvés, de répartir intégralement les échanges de données de manière à ne pas recourir à des mécanismes centralisés, de conserver à l'exécution un modèle dicté par la disponibilité des données. Les systèmes ainsi transformés sont constitués d'un ensemble fini de fonctions de calcul dotées d'interfaces de communications; elles constituent les tâches du système. Ces tâches ne communiquent entre-elles que par l'intermédiaire de leurs entrées et sorties. L'émission et la réception de données sont conditionnées par des commandes gardées implémentées dans le corps des interfaces.

Abstract

Work presented in this document concerns a set of automatic transformation processes of synchronous programs in order to implement them on distributed heterogeneous architectures. We particularly took an interest in the area of real-time image processing. Our approach is as follows; real-time systems are designed and validated on the assumption of synchronism. In practice we used an existing tool: the SIGNAL synchronous language. We then developped a set of transformation processes which allow to automatically implement real-time systems in a concurrent sequential processes model. Major features of this work are to guarantee safety properties by transforming proved programs, to distribute data exchanges in order to avoid centralized mecanism requirements, to preserve all dataflow aspects at run-time. Transformed systems are made of a finite set of computing functions with associated communication interfaces, called tasks. Tasks do not communicate except on entry and exit. Data transmissions are restricted by guarded commands which are implemented in the body of the interfaces.

Je tiens à remercier,

Les membres du jury qui ont acceptés de rapporter ce travail.

Les nombreux lecteurs et correcteurs attentifs.

L'ensemble de mes collègues de travail au sein de THOMSON MULTIMEDIA dont la diversité intellectuelle n'a d'égal que l'extrême compétence, pour leur sympathie.

Table des matières

Introduction

Le travail réalisé dans le cadre de cette thèse porte sur l'implémentation des systèmes temps-réel complexes sur les architectures hétérogènes distribuées Notre approche est appliquée au domaine plus spécifique du traitement d'images, mais pourrait être généralisée aux systèmes temps-réel qui manifestent des similarités en terme de traitement et de contrôle. (se reporter également à nos publications [JCR+96, GDM96a, GDM96b, GDM96c, GBDM96, Gal96, Gal95]).

Il s'inscrit dans la dynamique de deux projets: un programme de recherche européen, intitulé ESPRIT BRA PROJECT 8849 SM-IMP[1] [JCR+96] couvrant à la fois la conception d'architectures nouvelles pour le traitement temps-réel de l'image ainsi que la programmation et la mise au point de ces systèmes — et le projet P^3I [CJC+94, CJG+94b, CJG+94a, GBDL+94][2], une plateforme généraliste pour le traitement temps-réel de l'image. Par le truchement de ces deux projets, notre travail se situe donc au croisement de la recherche fondamentale et appliquée.

Les travaux menés conjointement par les différents partenaires ont ainsi abouti à la proposition d'un modèle architectural ouvert dans lequel machines généralistes (de la station de travail aux calculateurs parallèles) et unités cablées sont fédérées sous la forme d'un seul *gros* système parallèle de type MIMD appelé machine hétérogène distribuée. Ce modèle architectural peut être entrevu comme un réseau dont la nature n'a pas été précisément définie et pouvant accueillir différents types d'unités qui sont susceptibles d'être ajoutés ou retirés au cours du temps.

L'exploitation pour des propos industriels d'une plateforme de ce type dans le cadre du projet P^3I, a notamment permis de repositionner les problèmes sur le plan logiciel bien plus que matériel. Ainsi, le besoin crucial de décorréler complètement du modèle architectural sous-jacent, les phases de spécification des problèmes, de conception et de validation des systèmes a été souligné.

Notre propos n'est pas ici d'ajouter une nouvelle pierre au lourd édifice théorique déjà existant, mais au regard des problèmes spécifiques posés par la programmation des systèmes temps-réel complexes, de proposer une méthode d'implémentation directe sur une plateforme hétérogène distribuée.

Le domaine d'application du traitement temps-réel de l'image concerne des secteurs d'activité aussi sensibles que le nucléaire, l'aéronautique, le militaire, etc, qui posent des contraintes fortes en termes de sûreté des programmes, et pour lesquels il est fondamental d'être le plus efficace compte tenu des possibilités de l'architecture sous-jacente.

Un certain nombre de caractéristiques majeures sont liées aux applications temps-réel et à leur contexte d'utilisation [Maf93]:

- Les processus temps-réel maintiennent une interaction permanente avec leur environnement,

1. Le projet SM-IMP est un partenariat de recherche entre sociétés industrielles: THOMSON MULTIMEDIA R&D France — PARSYTEC, et laboratoires universitaires: DIST University of Genoa — DELFT University of Technology — University College London.
2. © THOMSON MULTIMEDIA R&D France.

- La portabilité et la ré-utilisabilité du code développé nécessite que la spécification d'une application soit complètement décorrélée de tout modèle architectural,
- La sûreté des programmes impose la maîtrise d'outils formels permettant de prouver l'existence et l'unicité de solution de tout problème traité,
- Finalement, les applications temps-réel mêlent deux caractéristiques importantes: un ensemble lourd de traitements et un système complexe de contrôle et de commande.

Exécuter des traitements complexes et tenir des performances proches du temps-réel[3] nécessite le plus souvent de s'adresser à des architectures hétérogènes distribuées exhibant des modes de contrôle différents (c.f. taxinomie de FLYNN [Fly66]).

Les langages apparus avec les architectures multi-processeurs et fondés sur le *Communicating Sequential Processes* (CSP) de HOARE [Hoa85], comme par exemple OCCAM [LTD88], ont largement inspiré le modèle de programmation des architectures hétérogènes distribuées. Dès lors une grande expérience a été acquise par l'utilisation de ce modèle qui révèle un ensemble d'atouts majeurs:

- La décomposition d'une application en un ensemble de tâches est une manière intuitive et efficace d'appréhender les problèmes complexes,
- La spécification des programmes n'est pas liée à une architecture particulière,
- L'exécution concurrente est facilitée par l'existence d'un ensemble de tâches à réaliser.
- Ces langages ont l'avantage de posséder les primitives de gestion de processus indispensables à la programmation naturelle des systèmes temps-réel.

Il semble cependant difficile dans le cadre de la programmation asynchrone d'obtenir une définition précise et rigoureuse du comportement temporel des processus spécifiés. Meme les modèles CSP de HOARE, et CCS qui proposent une restriction à un mode de communication synchrone sur *rendez-vous*, mis en œuvre dans des langages comme ADA [RCDS90], OCCAM [LTD88] ou CRSM [Sha92] ne résolvent pas ce problème.

En outre, l'asynchronisme et le non-déterminisme intrinsèque de ces langages compliquent considérablement la mise au point des applications, et rend difficile l'obtention de toute preuve formelle des programmes.

Les langages synchrones ont été introduits très récemment comme une réponse à ces problèmes, en fournissant des primitives *"idéales"*, qui permettent de raisonner comme si le programme réagissait instantanément aux événements externes. Le contexte formel ainsi fourni par les langages synchrones permet à la fois d'exprimer simplement les relations temporelles complexes qui interviennent naturellement dans les problèmes temps-réel, et d'effectuer des preuves automatiques des programmes. Ces modèles posent par contre des difficultés sérieuses en termes d'implémentation.

De manière synthétique, on peut dire que: "*In practice, each style tends to be weak where the other is strong*" [BB91]. L'hypothèse synchrone définit en effet un cadre formel essentiel pour raisonner sur le temps, alors que le modèle asynchrone répond idéalement aux problèmes posés par l'implémentation[4].

3. Les temps de réponse en entrée et sortie du système s'effectuent dans des délais suffisants pour pouvoir être logiquement considérés comme instantanés.

4. Synchronisme et asynchronisme ne se réfèrent pas à une réalité physique, mais à une manière de raisonner sur le temps. Dans le premier cas, communications et traitement sont supposés s'effectuer dans un temps logique nul, alors que dans le second ces délais sont pris en compte.

L'approche nouvelle que nous avons adoptée consiste à tirer parti des deux modèles plutôt que de les considérer comme définitivement antagonistes. Nous proposons donc une démarche visant à conjuguer les avantages respectifs des modèles synchrone et asynchrone. L'essentiel de notre apport réside dans un ensemble de processus formels de transformation permettant de passer automatiquement du squelette synchrone des applications à une implémentation asynchrone sous la forme de processus séquentiels communicants.

Notre objectif est de favoriser l'implémentation automatique de programmes "prouvés" sur les architectures hétèrogènes distribuées moyennement couplées. C'est-à-dire, pour lesquelles les coûts de communication ne peuvent pas être négligés.

La démarche que nous avons adoptée est la suivante: les phases de spécification et conception des systèmes, sont effectuées sous l'hypothèse de synchronisme. Une application temps-réel de traitement de l'image est ainsi décrite comme un ensemble fini de fonctions élémentaires [5] s'exécutant concurremment dans le temps, et ne communiquant que sur leurs entrées et leurs sorties. Les relations temporelles sont exprimées au moyen d'un jeu minimal d'opérateurs empruntés aux langages synchrones. On admet ainsi que l'utilisateur dispose d'une architecture "idéale" sur laquelle les différentes fonctions du programme peuvent s'exécuter dans des délais suffisants (on les suppose bornés) pour tenir le temps-réel. Cette hypothèse vise à décorréler la phase de développement des problèmes liés à l'implémentation.

À partir de ce squelette synchrone de l'application, un processus automatique de vérification des propriétés temporelles est déclenché. En pratique nous avons utilisé un outil existant, il s'agit du *calcul d'horloge* de SIGNAL [BLG90, BB91, LGG90, LGM+91, MCL92, Maf93, Bes92, Ché91] [6].

L'essentiel de notre travail opère sur les résultats produits par cette phase de résolution des contraintes temporelles. Il s'agit d'un ensemble de transformations automatiques portant sur la ré-écriture des équations d'horloge et des dépendances de calcul définies par le *calcul d'horloge*. Ces processus de transformation ont pour but de répartir les échanges de données entre les différentes taches du programme tout en minimisant le volume des flots transmis. Un autre point important est de maintenir les aspects dataflow à l'exécution.

En fin de traitement, expressions d'horloge et dépendances de calcul sont synthétisées sous la forme de dépendances traditionnelles de données qui sont conditionnées au moyen de commandes gardées dans des interfaces de communication ajoutées aux fonctions élémentaires de traitement. Ces interfaces de communication sont destinées à être implémentées via une librairie standard de communication de type *message passing* (i.g. PVM [GBD+94a, GBD+94b], MPI [Uni94, SL94]).

Dans l'optique d'une implémentation automatique de systèmes temps-réel sur les architectures hétérogènes distribuées, les aspects novateurs de notre approche approche sont donc:

- De garantir a priori la sûreté des programmes en implémentant des systèmes déjà prouvés.
- À partir d'une vue monolithique du programme sous la forme d'un graphe de dépendances de calcul et de contrôle nous montrons dans quelle mesure il est possible de répartir complètement entre les différentes tâches l'ensemble des échanges de données, de manière à ce que chaque tâche soit pleinement responsable de ses conditions d'activation. Il en ressort que ces tâches ont gagné une plus grande autonomie qui sied mieux à une implémentation sur une architecture hétérogène distribuée. En particulier, cela permet d'éviter les points de contention qui ne manqueraient pas de surgir si nous utilisions des mécanismes d'échange ou de contrôle centralisés. Cette autonomie correspond en outre à la contrepartie applicative de l'architecture ouverte qui a été définie.

5. La création dynamique de processus est prohibée. Cette condition est cruciale pour assurer la terminaison d'un programme

• De préserver à l'exécution un système purement dicté par la disponibilité des données.

Le document est organisé de la manière suivante:

• CHAPITRE 1: Nous présentons ici les différents concepts liés à la programmation des systèmes complexes de traitement temps-réel de l'image, avec un éclairage particulier porté sur la théorie de la programmation synchrone. Et nous concluons par l'approche que nous préconisons.

• CHAPITRE 2: Ce chapitre ainsi que les deux suivants, décrivent les outils formels que nous avons développés pour produire une implémentation CSP à partir d'un squelette synchrone. Ici nous traitons plus particulièrement la partie contrôle. La compilation SIGNAL permet d'isoler les aspects purement liés au contrôle des applications en produisant un système d'équations d'horloges. Ces expressions d'horloge sont soit directement définies à partir de flots du programme, soit par des formules de calcul sur horloges. À tout instant, le rôle des horloges diffusées dans l'intégralité des nœuds du réseau, et dont le calcul est centralisé, est de valider les dépendances de calcul. Cette gestion centralisée ou semi-centralisée est difficilement acceptable dans un contexte CSP. Nous proposons donc dans cette partie, de transformer toutes les équations d'horloges en expressions normales sur signaux booléens du programme. Cette transformation permettra ultérieurement de distribuer complètement entre les tâches le contrôle des dépendances de calcul.

• CHAPITRE 3: Pour les besoins de la phase de résolution des contraintes temporelles, les dépendances de calcul sont décomposées en un grand nombre d'expressions élémentaires. Ce qui se traduit par la démultiplication des nœuds et des signaux véhiculés dans le graphe synchrone de l'application. Une implémentation CSP efficace suppose de réduire au maximum la masse des signaux échangés par les tâches. Un processus formel d'agrégation des dépendances de calcul est ainsi proposé, afin de connecter directement sorties et entrées des fonctions de calcul.

• CHAPITRE 4: Expressions normalisées d'horloges et expressions de calcul agrégées ne suffisent pas pour permettre une implémentation directe des applications. Encore faut-il disposer d'une méthode qui puisse être automatisée dans l'écriture d'un compilateur. La gestion des mémoires induites par les valeurs retardées pose en ce sens grand nombre de difficultés. Nous proposons ainsi une méthode simple de construction des interfaces de communication des tâches qui vont implémenter les parties contrôle normalisé et dépendances de calcul qui ont été obtenues précédemment.

• CHAPITRE 5: Nous concluons enfin sur notre travail, en rappelant quelles en sont les caractéristiques principales.

• ANNEXE A: Dans ce premier appendice nous présentons une implémentation des interfaces de l'application MPEG-2 version *simple-profile* utilisant la librairie de communication PVM.

• ANNEXE B: Présentation détaillée de la sémantique du langage synchrone SIGNAL sur lequel se fonde notre approche.

• ANNEXE C: Petit aparté technique sur les fichiers produits par le compilateur SIGNAL, et qui nous ont servi à construire les graphes de dépendances conditionnées, ainsi que les arbres d'horloges.

Chapitre I

Contexte de l'étude

Nous nous proposons dans cette partie préliminaire de décrire les particularités du traitement de l'image (SECTION I.1). Puis d'étendre cette présentation aux systèmes temps-réel (SECTION I.2) afin d'identifier les besoins et les mécanismes généraux mis en œuvre dans ce domaine d'activité. Il existe aujourd'hui deux grandes manières d'aborder les problèmes temps-réel: une approche purement synchrone —et une approche purement asynchrone. L'approche synchrone étant relativement récente, nous la décrirons largement dans la SECTION I.3. Puis nous conclurons ce chapitre par une présentation de la démarche que nous avons suivie (SECTION I.4).

I.1 Le traitement d'images

Le traitement d'images est un domaine d'application qui englobe une variété importante de problèmes (SECTION I.1.1). A la base de ces problèmes on trouve le besoin d'acquérir, d'analyser, d'interpréter l'image. Une caractéristique essentielle dans ce domaine d'application est donc le volume énorme des structures élémentaires qui sont manipulées (cf. tableau I.1 emprunté à [Rob94]).

I.1.1 Les niveaux de traitement

Il est communément établi en traitement d'images, d'identifier trois niveaux d'application: bas niveau — niveau intermédiaire — haut niveau. Cette catégorisation est établie sur le *"degré d'intelligence"* des traitements, et par voie de conséquence sur les types de structure et le volume des données consommées, traitées et produites par les algorithmes [DF88, Fau88, Kom90].

Traitements de bas niveau

Les algorithmes de bas niveau sont généralement dédiés à l'amélioration de la qualité des images, et à la simplification de leur contenu. La donnée en entrée du programme est une image de certaine dimension. Les données en sortie sont une ou plusieurs images de même dimension. Les pixels produits peuvent avoir été étiquetés en fonction de propriétés locales de l'image (types de texture, propriétés topologiques, etc) ou modifiés par une transformation locale (opérations morphologiques [Kom90], seuillage, filtrage, rotation, extraction de contour, étiquetage par *Shrinking* et par *Broadcast* [CSS90, HT89], convolution [ZXL89], etc).

Traitements de niveau intermédiaire

Les algorithmes de niveau intermédiaire extraient d'images pré-traitées (provenant du niveau inférieur), des informations sur les objets qui composent la scène (segmentation par *Region growing*, par *transformée de Hough* [KC90], suivi de contour, estimation de mouvement, etc) . Les résultats sortis correspondent à des structures de données différentes de l'image (liste de segments de droite, centre d'inertie, champ de mouvement, etc).

Algorithme concerné	Structure de données	Type de données
Seuillage	Image	de pixels
	Tableau 2D	de caractères
Filtrage	Image	de pixels
	Tableau 2D	d'entiers
F.F.T.	Image	de fréquence
	Tableau 2D	de flottants
Morphologie	Image	binaire
	Tableau 2D	de booléens
Rotation	Image	de déplacements
	Tableau 2D	d'entiers
Labellisation	Image	de réduction binaire
(par *Shrinking*)	Tableau 3D	de booléens
Labellisation	Image	de labels
(par *Broadcast*)	Tableau 2D	d'entiers non signés
Histogramme	Image	de cumul
	Tableau 2D	de vecteurs 1D
Segmentation	Arbre	quaternaire
(par *Region growing*)	Tableau 2D	de tableaux 2D
Segmentation	Image	de paramètres
(par *Transformée de Hough*)	Tableau 3D	d'entiers
Estimation de mouvement	Champs	de mouvement
	Tableau 2D	d'entiers signés

TAB. I.1 - *Structures de données en traitement d'image*

Traitements de haut niveau

Les tâches de haut niveau ont pour objet d'interpréter la scène contenue dans une séquence d'images et font appel aux traitements de niveau inférieur. Basés sur la mise en relation d'objets, ces traitement permettent d'extraire du "sens" des données symboliques passées en entrée, en les associant avec une base de connaissances décrivant les caractéristiques du monde dont l'image est originaire.

I.1.2 Degré du parallélisme

Le grain du parallélisme varie fondamentalement en fonction des différents niveaux de traitement de l'image et du déroulement de l'application. La complexité des traitements évolue de la même manière. La catégorisation des algorithmes de traitement de l'image en trois niveaux illustre l'évolution des besoins en termes de contrôle et la nécessité d'utiliser des architectures hétérogènes exhibant les différents types de parallélisme requis (c.f. tableau I.1.2).

En pratique, l'exécution de certaines fonctions standards de traitement d'image est souvent laissée "entre les mains" d'unités câblées (ex. le *reformateur* de P^3I [CJC+94, CJG+94a, GBDL+94] une machine parallèle hétérogène dédiée au traitement temps-réel de l'image). L'exécution efficace d'applications en traitement d'images est donc l'apanage de machines dédiées à l'électronique totalement ou partiellement spécialisée.

"*In the present state-of-the-art in machine and compiler design, it is not possible to adequately protect the programmer from the need to have knowledge of machine architecture if efficient programs are to be written and tasks are to be interpreted intelligently in the interests both of those specifying the tasks and of those attempting to carry them out.*" [Duf82, p.274].

	Bas-Niveau	Niveau-Intermédiaire	Haut-Niveau
Structures de données	*images*	*listes, vecteurs, arbres*	*complexes*
Éléments de structure	*centaines de milliers*	*centaines*	*dizaine*
Opérations	*élémentaires*	*complexes*	*complexes*
Communications entre éléments de structure	*très régulières voisinage local du pixel*	*irrégulières distantes*	*irrégulières distantes*
Parallélisme exhibé	*SIMD*	*MIMD/SPMD*	*aucun*

TAB. I.2 - *Les besoins du traitement de l'image*

La difficulté porte davantage sur les aspects logiciel que matériel; puisqu'il est toujours possible, moyennant un coût supplémentaire, de concevoir des unités cablées qui réalisent efficacement un certain nombre de fonctions clefs en traitement d'images. Pourtant, les développements sur les machines dédiées continuent à se faire au détriment de la portabilité des applications et même de leur mise à jour lors de modifications éventuelles de l'architecture. Ce qui revient à mésestimer les coûts de développement, d'exploitation et de maintenance du logiciel.

Si on raisonne en termes d'applications, n'importe quelle architecture distribuée hétérogène de type généraliste peut constituer "une bonne" plate-forme de développement. Et selon nous, les optimisations liées à l'architecture ne devraient pas transpirer au niveau global pour remettre en cause l'intégralité du processus de développement; elles doivent se restreindre à quelques nœuds fonctionnels sans qu'il soit nécessaire de repenser tout le système (notions de modularité, d'évolutivité, et d'efficacité).

I.1.3 Mise en œuvre

En pratique, le développement d'applications complexes de traitement de l'image repose traditionnellement sur un découpage en deux niveaux du processus de développement: un niveau fonction et un niveau application. L'utilisateur décrit ainsi dans un premier temps ses algorithmes (niveau fonction), puis la manière selon laquelle ceux-ci vont collaborer pour obtenir le résultat attendu (niveau application).

I.1.4 Niveau fonction

C'est sur le niveau fonctionnel que repose l'essentiel de la partie algorithmique du programme. Une fonction peut ainsi être simplement extraite de la bibliothèque de l'utilisateur, ou augurer le développement de nouvelles sections de code:

1. **Utilisation des fonctions standards de la bibliothèque.** Il s'agit de fonctions classiques telles que: le seuillage, la convolution, la détection de contours, l'histogramme, la transformée de Fourier, etc, pour lesquelles il existe un certain nombre d'algorithmes bien connus.

2. **Développement de fonctions incluses dans la bibliothèque de l'utilisateur.** Elles répondent à des besoins spécifiques de l'utilisateur, mais bien plus souvent à une carence de la bibliothèque standard ou à l'apport d'une nouvelle technologie (algorithmes récents, optimisations liées à l'architecture).

I.1.5 Niveau application

La description du programme au niveau application repose sur la connexion des différentes tâches de traitement. En d'autres termes, les développements menés au niveau application consistent à programmer l'enchainement logique des fonctions de traitement de l'image dont l'évolution dans

le cas des applications temps-réel est liée aux activations provenant de l'environnement (voir SECTION I.2).

Cette phase qui est souvent considérée comme le plus haut niveau d'abstraction, passe généralement par des développements de type graphique (ex. l'EDHN [JCR+96] une interface graphique pour le développement d'application temps-réel de traitement de l'image sur P^3I).

Au niveau application, la nature des connexions entre les fonctions (dépendances de données), induit naturellement un formalisme de type data-flow. Cette particularité est d'ailleurs soulignée par le caractère résolument graphique des interfaces.

Le niveau application correspond donc à la description du graphe de contrôle global de l'application. Il s'agit d'un graphe data-flow dont les opérateurs sont des fonctions de traitement de l'image.

I.1.6 Conclusion

Le modèle traditionnel de développement d'une application complexe en traitement de l'image mélange donc deux approches. Le niveau fonction suggère une approche descendante, qui correspond à une méthode efficace de spécification pour les algorithmes implémentant une fonction unique, de haut niveau et non susceptible d'évoluer. Et l'emploi au niveau application d'une approche montante qui permet de construire une application (au sens large) en réutilisant et en combinant progressivement des éléments existants.

En d'autres termes, développer une application de traitement de l'image c'est définir une bibliothèque de fonctions élémentaires et décrire la manière dont celles-ci collaborent pour obtenir un résultat déterminé.

Les applications temps-réel se caractérisent de surcroît par les flots ou séquences potentiellement infinies qu'elles manipulent. Par exemple, une application temps-réel de l'image peut prendre ses entrées depuis des capteurs tels que caméras, magnétoscopes, etc, et délivrer des sorties vers des acteurs du monde physique tels que moniteurs, robots, etc. Une conséquence immédiate concerne les temps d'acquisition des entrées, aussi bien que les délais de réponse entrées/sorties qui sont strictes. Mais bien d'autres propriétés fondamentales sont liées à la notion de temps qui dans le domaine particulier d'application du traitement temps-réel constitue une contrainte logique de base.

I.2 Le temps-réel

Le traitement temps-réel de l'image concerne des secteurs d'activité aussi sensibles et variés que: l'aéronautique, le nucléaire, les applications militaires, etc. Les caractéristiques principales des systèmes temps-réel sont:

- **des contraintes de sûreté cruciales.** C'est certainement, et de loin, le caractère principal des applications temps-réel. En effet, le dysfonctionnement de tels systèmes peut avoir des conséquences dramatiques. Par conséquent, dans de nombreux cas, la vérification formelle des systèmes temps-réel est fondamentale. Il faut donc pouvoir se doter d'outils et modèles mathématiques permettant de garantir statiquement le déterminisme des applications,

- **un parallélisme intrinsèque.** En effet, il est très souvent commode et même naturel de concevoir de tels systèmes comme un ensemble de composants coopérant et s'exécutant concurremment,

- **des contraintes temporelles strictes.** Ces contraintes concernent aussi bien le rythme d'acquisition des entrées que les temps de réponse entrées/sorties. Il faut être capable d'exprimer ces contraintes dans la spécification du problème, de les prendre en compte dans la conception du système, et de vérifier que l'implémentation les respecte.

Les problèmes qui peuvent apparaître lors de la spécification et la réalisation d'une application temps-réel ne sont généralement pas abordés en informatique classique: prise en compte du temps comme contrainte logique de base, et non comme facteur de performance, et impératifs de sûreté obligeant à considérer l'erreur de fonctionnement comme un phénomène qu'il faut prévoir[Cos91].

I.2.1 Taxinomie

Dans la littérature qui traite du temps-réel, on opère souvent une distinction entre les systèmes temps-réel et les systèmes plus généralement *réactifs* ou *transformationnels*. Cette classification est basée sur la notion de comportement du système.

Système Transformationnel

Un système *transformationnel* est un système qui accepte des entrées, les transforme, et produit des sorties [Dzi90]. Une notion d'interactivité intervient lorsque le processus de calcul (ou transformation) requiert des informations supplémentaires qui peuvent être fournies par l'utilisateur.

Système Réactif

On appellera *réactif*, un système qui maintient une interaction permanente avec son environnement physique [BB91]. Son rôle consiste à réagir instantanément à des entrées provenant de façon répétitive de son environnement en produisant lui-même des sorties vers cet environnement (on parle également de systèmes dirigés par les événements). Ces systèmes seront par conséquent décrits en termes de comportement.

La difficulté suivante est souvent évoquée à propos des systèmes réactifs: les événements peuvent arriver à des instants imprévisibles, et doivent être pris "rapidement" en compte par le système.

Système Temps-réel

Pour notre part, nous avons adopté la distinction faite notamment dans [BB91] qui assimile les systèmes temps-réel à une particularisation des *systèmes réactifs*.

La dénomination *temps-réel* est plutôt réservée aux systèmes réactifs qui sont de surcroît sujets à des contraintes de temps définies extérieurement. Ainsi, un système temps-réel est défini comme un système qui maintient une relation continuelle (en opposition avec événementielle) avec un environnement asynchrone; c'est-à-dire un environnement qui progresse indépendamment du système temps-réel, d'une manière non-coopérante; par conséquent, un système temps-réel est pleinement responsable de la propre synchronisation de ses opérations par rapport à son environnement.

Système Mixte

Un système temps-réel complexe est généralement constitué d'une partie transformationnelle, et d'une partie réactive. Lors de l'exécution d'une application, ces deux parties interagissent. La méthode de conception associée est donc mixte: une partie réactive pour permettre une gestion efficace des événements visant à satisfaire les contraintes temps-réel strictes, et une partie transformationnelle réalisant le traitement des données (exécution d'algorithmes complexes). À cet égard, les applications complexes de traitement temps-réel de l'image appartiennent bien à cet ensemble de problèmes mixtes.

I.2.2 Mise en œuvre

Programmées en premier lieu en utilisant des machines analogiques et des circuits relais, les applications temps-réel bénéficièrent du développement des microprocesseurs et des ordinateurs. Les outils de programmation restaient cependant de bas niveau et spécifiques.

Avec l'évolution rapide des technologies et des besoins, les techniques de programmation bas niveau sont vite devenues inacceptables et inadaptées au développement d'applications complexes, qui posent des contraintes fortes en termes de sécurité.

Pour la résolution de tels problèmes, l'utilisation de langages de haut niveau est indispensable. Il existe *en gros* aujourd'hui deux grandes approches: une approche asynchrone — et une approche synchrone. Toutes deux exhibent des avantages et inconvénients respectifs qui apportent des réponses partielles aux problèmes posés par le traitement temps-réel.

Approche asynchrone

Les langages asynchrones ont essentiellement émergé avec l'apparition des architectures multiprocesseurs. Nous nous référons principalement ici au modèle CSP de HOARE [Hoa85].

Ces langages ont été conçus dès l'origine sur les principes de modularité et de concurrence (ex. ADA [RCDS90]). Ils ont l'avantage de posséder des primitives de gestion de processus indispensables à la programmation naturelle des systèmes temps-réel. Ils permettent en particulier de traiter à la fois les aspects calculatoires et les services de programmation parallèle et d'exécution répartie.

On trouve ainsi des primitives autorisant le démarrage, l'arrêt et la suspension de l'exécution des tâches. Des primitives de synchronisation et de partage de ressources sont également disponibles. De surcroît, la majeure partie de ces langages dispose d'environnements de développement.

En outre, la plupart des applications écrites à l'aide des langages asynchrones ont l'avantage d'être indépendantes des configurations matérielles et des systèmes d'exploitation. En théorie du moins, elles peuvent être exécutées sur des machines monoprocesseur ou sur un réseau de processeurs. Ce qui est appréciable lorsque l'on s'intéresse à la portabilité.

Finalement, beaucoup d'expériences a été acquise dans l'utilisation de ces techniques.

Toutefois, plusieurs inconvénients majeurs sont associés à l'utilisation de ces langages: ils sont tout d'abord intrinsèquement non déterministes. Bien qu'une communication soit vue comme une synchronisation entre deux processus, le temps pris entre la possibilité d'une communication et son accomplissement peut être arbitraire et est imprévisible. Lorsque plusieurs communications doivent être réalisées, leur ordre d'établissement est également aléatoire. Si bien qu'un même programme peut admettre plusieurs exécutions différentes pour une même séquence d'entrée. Ce non-déterminisme est inadapté au contrôle des applications temps-réel.

En outre, il n'est pas possible de réaliser des vérifications automatiques des programmes produits, et par conséquent impossible de garantir statiquement la sûreté des applications.

De plus, ces langages ne proposent aucune structure permettant de traiter le concept de temps autrement que comme simple facteur de performance.

Finalement, les langages de programmation asynchrones, s'ils présentent de nombreux avantages au niveau de l'implémentation des systèmes temps-réel, semblent mal adaptés à la spécification des problèmes où le temps apparait comme contrainte logique de base, et aux raisonnements temporels.

Approche synchrone

Les langages synchrones ont été introduits très récemment en réponse aux besoins soumis par la prise en compte du temps dans la spécification de problèmes où des relations temporelles complexes doivent être modélisées.

Le contexte formel fourni par les langages synchrones permet non seulement de spécifier les relations temporelles complexes qui interviennent naturellement dans les systèmes temps-réel, mais également d'effectuer des preuves automatiques des programmes. Ils ont introduit pour cela un ensemble de primitives "*idéales*", non présentes jusqu'alors dans les langages de programmation classiques, et qui permettent de raisonner sur le temps comme n'importe quel autre domaine de valeur.

Toutefois, même si l'approche synchrone est idéale pour décrire des relations temporelles complexes et raisonner formellement sur le temps, elle pose néanmoins de sérieux problèmes en termes

d'implémentation. A contrario, les langages asynchrones conçus dès le début en termes d'implémentation ne pose pas de problème de ce point de vue.

Nous reprendrons plus en détail dans une partie ultérieure de ce chapitre (c.f. SECTION I.3), les caractéristiques qui sont particulièrement liées à l'hypothèse de synchronisme.

Synthèse des approches

Sur les deux points précédents, on peut finalement conclure par: "*In practice, each style tends to be weak where the other is strong*" [BB91]. L'hypothèse de synchronisme pose en effet un cadre formel essentiel pour raisonner sur le temps, alors que le modèle asynchrone répond idéalement aux problèmes posés par l'implémentation.

Alors que l'on a longtemps considéré ces deux hypothèses comme plus ou moins antagonistes, il est clair qu'une approche mixte combinant les avantages respectifs des modèles synchrones et asynchrones serait de loin particulièrement séduisante.

I.2.3 Modélisation

Les programmes temps-réel (au sens large) consomment des données en provenance de capteurs connectés au monde physique (par exemple des caméras). En réponse, ils doivent construire les commandes de sortie du système (par exemple affichage d'images traitées sur un moniteur).

Les systèmes temps-réel manipulent des valeurs, ou plus exactement des séquences de valeurs qui représentent des suites potentiellement infinies appelées *signaux* dans le langage SIGNAL [BLG90] (ou encore *tissu* dans $\mathbf{8}^{1/2}$ [GS93], ou *flot* dans LUSTRE [HCRP91a], etc). Par exemple, le signal x dénote la séquence infinie $\{x_t\}_{t\geqslant 0}$, où l'index de temps t est attaché à ce signal. Contrairement aux variables dans les langages classiques, les valeurs d'un signal ne sont pas persistantes; c'est-à-dire que l'accès au temps t de la valeur qui a été portée par l'occurrence x_{t-1} du signal x n'est pas direct.

Un signal porte deux types d'informations:

- une information de contrôle liée à sa présence à un instant donné,
- une information de valeur.

L'exécution d'un système temps-réel est rythmée par l'émission et la réception de ces signaux.

Tout signal, en plus des valeurs qu'il prend normalement dans son intervalle de définition, peut être affecté d'une valeur spéciale représentant l'*absence* d'événement à un instant donné. Le symbole utilisé pour noter l'absence d'événement est: $\perp$. L'absence d'événement est une notion commune aux langages LUSTRE et SIGNAL.

À chaque signal est implicitement associée une *horloge* qui définit les instants où le signal est présent (on note généralement: $\top$). La notion d'horloge est d'une certaine façon un moyen de décorréler la fonction de contrôle pure d'un signal de sa fonction de valeur. Deux signaux possédant la même horloge sont dits *synchrones*. L'horloge d'un signal ne fait pas référence à un temps de base (e.g. l'horloge du système), le rôle des horloges est de permettre de parler des relations temporelles existant entre les divers signaux d'un programme.

La préoccupation principale en SIGNAL est que toutes les questions de synchronisation soient complètement réglées à la compilation, de telle manière que la phase d'exécution n'ait pas à se préoccuper des $\perp$'s. Cet objectif est atteint par une représentation statique des relations temporelles exprimées par chaque opérateur (voir ANNEXE B).

I.2.4 Conclusion

Les problèmes temps-réel ont introduit un certain nombre de besoins nouveaux en informatique classique:

- besoin d'exprimer les contraintes temporelles complexes qui existent naturellement entre les différents signaux échangés par les tâches,
- besoin d'outils formels et de modèles mathématiques pour garantir statiquement l'existence et l'unicité de solution à un problème donné.

Ces besoins interviennent pour une part dès la spécification des problèmes: expression des relations temporelles complexes; et pour l'autre part au niveau des outils de compilation: preuve statique des programmes.

Les applications complexes de traitement temps-réel de l'image font partie des systèmes mixtes (c.f. SECTION I.2.1), c'est-à-dire qu'ils sont généralement dotés au niveau fonctionnel d'une lourde partie algorithmique, et au niveau application d'un système complexe de contrôle et de commande.

La méthode de conception associée est donc mixte: développement d'une bibliothèque de tâches séquentielles traditionnelles de traitement d'images au niveau fonctionnel — description de l'enchaînement logique des tâches, ainsi que des relations complexes existant naturellement entre les signaux échangés par celles-ci au niveau application.
Le dernier point sous-entend le besoin d'introduire de nouveaux opérateurs qui permettent d'exprimer simplement les contraintes temporelles dans la spécification des problèmes. Ainsi que des méthodes formelles basées sur ces nouveaux opérateurs qui permettent de raisonner statiquement sur le temps comme n'importe quel autre domaine de valeur. Ces outils sont fournis par l'approche synchrone que nous décrivons dans la section suivante.

Au niveau implémentation, les langages de programmation concurrents constituent une piste intéressante:

- la décomposition d'une application en un ensemble de tâches comme le suggère la manière traditionnelle d'aborder les problèmes en traitement temps-réel de l'image, est inhérente au modèle de programmation concurrent,
- ces techniques sont relativement indépendantes des architectures ce qui facilite l'évolutivité et la ré-utilisabilité du code produit: ces deux propriétés constituent le minimum des services attendus lorsque l'on développe des bibliothèques de fonctions,
- enfin, et surtout, les langages de programmation concurrents fournissent un ensemble de primitives de manipulation de processus essentiels à l'implémentation des systèmes temps-réel.

L'approche que nous avons adoptée vise à combiner les avantages respectifs des modèles synchrones et asynchrones. En spécifiant et vérifiant les systèmes temps-réel sous l'hypothèse de synchronisme, et en fournissant les outils formels permettant d'obtenir automatiquement une implémentation asynchrone traditionnelle.

I.3 L'hypothèse de synchronisme

Les langages synchrones ont été introduits pour faciliter la tâche du programmeur en lui fournissant des primitives "idéales", permettant de raisonner comme si le programme réagissait *instantanément* aux événements externes. Le modèle d'exécution est alors rendu déterministe. Et il est ainsi possible d'effectuer des preuves, des tests, de répéter l'exécution des programmes.

L'approche synchrone permet de raisonner sur le temps comme n'importe quel autre domaine de valeur [LGG90]. Pour cela, elle ne considère l'évolution d'une application que sur une discrétisation du temps physique (ce qui suppose des temps de calcul et de communication bornés); autrement dit, elle réalise une abstraction logique du temps. La notion de synchronisme qualifie ici la manière de raisonner sur le temps, et ne se réfère absolument pas à une réalité physique.

"Ces langages (les langages synchrones) *abandonnent tous l'hypothèse d'asynchronisme qui est à la base des langages classiques, pour la remplacer par une hypothèse de synchronisme plus ou moins forte.* (...) *On suppose ainsi que les sorties sont fournies de manière absolument synchrone aux entrées, donc que leur calcul* ≪ *ne prend pas de temps* ≫" [BCG87, p.306].

L'hypothèse de synchronisme n'est par ailleurs pas neuve: les physiciens savent bien qu'à un niveau assez fin les causes et les effets ne sont pas synchrones; cela ne les empêche pas d'appliquer les équations "instantanées" de NEWTON ou de MAXWELL aux phénomènes macroscopiques. En électricité, la plupart des raisonnements supposent une propagation instantanée du courant électrique. Dans tous les cas, l'hypothèse de synchronisme simplifie considérablement l'étude des phénomènes, pourvu qu'on soit dans son domaine de validité [BCG87].

I.3.1 Propriétés du synchronisme

En pratique, l'hypothèse de synchronisme revient à supposer que le programme réagit assez vite pour percevoir les événements externes en bon ordre. Deux notions importantes en découlent: *instantanéité* et *simultanéité.*

L'instantanéité

La dynamique des systèmes synchrones s'inscrit dans un temps logique pour lequel les tâches élémentaires ont une durée nulle; tout se passe comme si les tâches étaient exécutées sur des processeurs de puissance infinie, elles sont instantanées [LG89].

En pratique, si on considère par exemple un processus élémentaire comportant deux signaux d'entrée et un signal de sortie, la notion d'instantanéité s'exprime logiquement par le fait que: pour tout t, la $t^{\text{ième}}$ occurrence sur la première entrée est évalué avec la $t^{\text{ième}}$ occurrence de la seconde entrée, pour produire une $t^{\text{ième}}$ occurrence en sortie.

La simultanéité

Plusieurs temps logiques peuvent coexister (on parle de temps multiforme), cependant les instants où ces temps coïncident sont tous connus; si deux instants coïncident, ils sont simultanés [LG89].

La notion de simultanéité est liée aux relations de présence et d'absence des signaux les uns par rapport aux autres. Ainsi par exemple, si on considère un couple $(\mathbf{x},\mathbf{y})$ de signaux, les événements possibles sont:

- $(\mathbf{x}_t,\mathbf{y}_t)$ les deux signaux sont *simultanément* présents,
- $(\mathbf{x}_t,\perp)$ le signal $\mathbf{x}_t$ est *présent*, le signal $\mathbf{y}_t$ est *absent*,
- $(\perp,\mathbf{y}_t)$ le signal $\mathbf{x}_t$ est *absent*, le signal $\mathbf{y}_t$ est *présent.*

D'autres caractéristiques importantes du temps liées à la notion d'événement, c'est-à-dire à la relation de présence et d'absence des signaux les uns par rapport aux autres, doivent être pris en considération: Ordre partiel entre les événements — Retard entre événements.

Ordre partiel entre les événements

Pour deux occurrences de temps d'un signal donné, on peut dire que *chronologiquement*, l'un est avant l'autre. Réciproquement, on peut même dire que de l'ordre dans lequel des événements se produisent est dérivé le concept de temps: "*The concept of time* (...) *is derived from the more basic concept of the order in which events occur*" [Lam78, p.558].

Retard entre événements

L'existence d'un ordre partiel entre les événements permet de travailler sur l'historique des valeurs prises par un signal. Ainsi, avec la notion de *retard*, la valeur définie à un instant t d'un signal peut être mémorisée pour être délivrée en sortie à l'occurrence suivante pour laquelle une nouvelle valeur d'entrée est définie.

I.3.2 Mise en œuvre

Sur les principes synchrones, ont été conçus LUSTRE [HCRP91a, HCRP91b, CPHP87], SIGNAL [BB91, LGM+91, BLG90, LGG90] et $8^{1/2}$ [GS93, Gia91] de type "flots de données", ESTEREL [BDS91, BCG87], SL, REACTIVE C [BdS95a][1], REACTIVE OBJECTS [BDS95b], et SML [CLM91] un des nombreux langages de description de circuits, de type impératif, les STATECHARTS [Har87] et STATEMATE [HLN+90] de type graphique, les graphes SDF [Buc93, Lee93, Lee91, LM87b, LM87a] basés sur un formalisme de type réseau de Petri.

Nous introduisons dans la suite quelques-uns des ces langages que nous avons choisis pour leur représentativité. Il ne s'agit en aucun cas d'une liste exhaustive. Outre les références données ci-dessus, un certain nombre de documents de synthèses [Dzi90, Cos91, LM94] nous ont été utiles dans cette présentation.

Les graphes SDF

Les graphes SDF (acronyme pour *Synchronous Data Flow*) [Buc93, Lee93, Lee91, LM87b, LM87a] sont une variante des graphes data-flow traditionnels. Dans un graphe SDF la quantité de données consommée et produite par chaque nœud pour l'ensemble de ses entrées et sorties est fixée à la conception des programmes. Par contre, la nature des éléments produits et consommés n'est pas fixée, et leur granularité peut ainsi varier de quelques pixels à des images entières.

Les éléments qui doivent être consommés par un opérateur sont bufferisés à l'entrée de l'opérateur jusqu'à ce qu'une valeur seuil d'entrée soit atteinte; l'opérateur est alors exécuté. Dans le modèle abstrait, les résultats de l'exécution sont disponibles immédiatement, c'est-à-dire que les éléments en sortie de l'opérateur sont produits sans délai. Le nombre d'éléments nécessaires (valeur seuil) pour déclencher une opération dépend de l'opération et de la granularité des éléments.

Une application exprimée sous forme de graphes SDF peut être à priori décomposée en sous-problèmes, chacun réalisant une opération distincte. La modularité de ces graphes permet donc de décrire une application à différents niveaux d'abstraction. Un graphe SDF est ainsi constitué d'une collection de nœuds connectés. Chaque nœud représente une opération spécifique dont la complexité dépend du niveau de description du bloc fonctionnel associé.

Les nœuds opèrent sur des séquences infinies de valeurs (signaux). Toutes les interactions entre les opérateurs sont spécifiées au moyen de transferts statiques qui correspondent aux arcs du graphe. Le fait que ces transferts doivent être statiquement spécifiés restreint la représentation SDF à une classe limitée d'applications.

Les graphes de "flot de signaux" proposés par LEE et MESSERSCHMITT sont dits synchrones pour exprimer le fait qu'un opérateur est invoqué quand un nombre pré-déterminé de nouvelles valeurs sont disponibles sur toutes ses entrées.

1. SL est un langage synchrone construit à partir de ESTEREL et compilé vers REACTIVE C

Un graphe SDF peut être analysé afin de déterminer si tous ses buffers d'entrée et de sortie sont bornés. Par exemple, lorsque deux nœuds sont reliés par un arc de dépendance, il est fondamental de pouvoir prouver que l'exécution, un nombre fini de fois, de chaque nœud, permette l'échange complet du nombre spécifié de données, sans accumulation de part ou d'autre des nœuds. Cette propriété correspondant à l'existence d'un ordonnancement cyclique borné du graphe est appelée *consistance.*

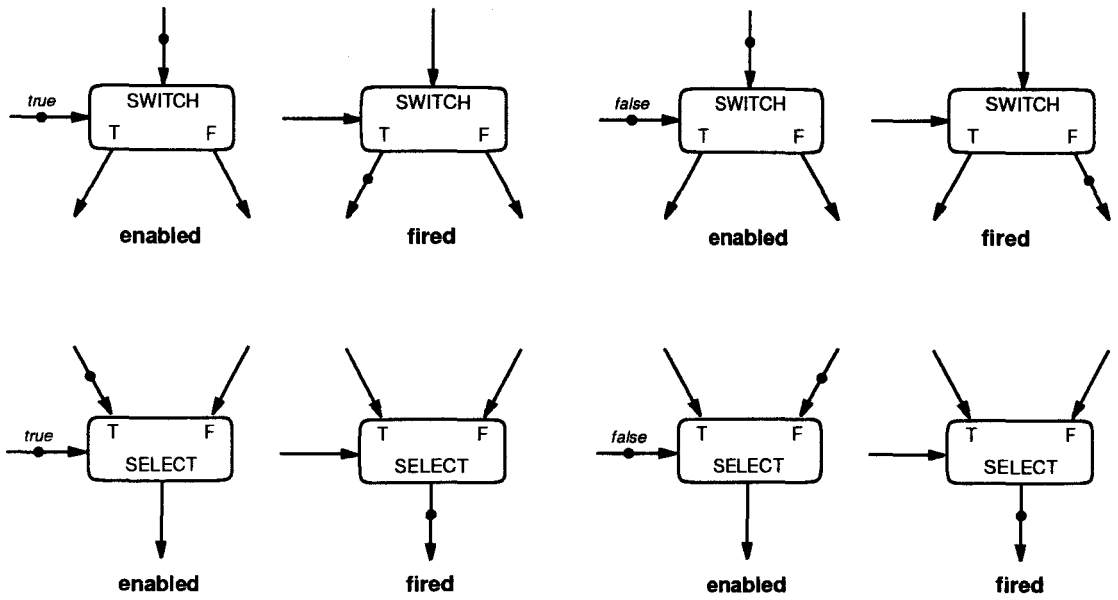


FIG. I.1 - *Les acteurs data-flow dynamiques* `SWITCH` *et* `SELECT`.

Les acteurs data-flow dynamiques principaux des graphes SDF sont présentés en figure I.1. L'acteur `SWITCH` émet un token en sortie de l'un de ses ports (`T` ou `F`) en fonction de la valeur portée par un booléen de contrôle. L'acteur `SELECT` reçoit un token sur l'un de ses ports d'entrée déterminé par la valeur portée par un booléen de contrôle. Un opérateur de retard sur signal est également défini.

Les acteurs dynamiques des graphes SDF n'offrent pas beaucoup de souplesse pour exprimer les relations temporelles complexes qui interviennent naturellement dans les grandes applications temps-réel. L'utilisation de ce modèle se restreint donc à une classe de problèmes simples.

En fait, comme de manière générale pour tous les formalismes basés sur les réseaux de Pétri, les graphes SDF sont mal adaptés aux grandes applications. Ils donnent en effet des objets incompréhensibles au-delà d'une certaine taille.

ESTEREL

ESTEREL [BDS91, BCG87] est un "*langage parallèle impératif qui possède une sémantique mathématique rigoureuse et une implémentation complète*" [BCG87, p.306].

Un programme ESTEREL est constitué d'une collection de *modules* qui peuvent communiquer et échanger de l'information. Cette structure modulaire permet une programmation hiérarchisée. Il existe deux opérateurs de composition de modules: "||" qui rend le parallélisme explicite au niveau de l'utilisateur — et l'opérateur ";" de composition séquentielle permettant d'exécuter des suites d'instructions.

Le concept central d'ESTEREL est celui d'événement: un événement est l'émission d'un signal, ou de plusieurs signaux simultanés, pouvant transporter des valeurs. La gestion de la communication, de la synchronisation et du partage des données est assurée par l'emploi de ces signaux. Ils

sont tous émis instantanément par *broadcast*: toutes les parties du programme reçoivent les memes signaux au même instant (dans la limite de la portée syntaxique de la déclaration pour un signal local).

L'utilisation de la diffusion facilite l'écriture de programmes modulaires à composants réutilisables. Elle a en effet deux avantages: on n'a pas besoin de connaître le ou les destinataires d'un message que l'on émet, et on n'a pas besoin de connaître le ou les auteurs d'un message que l'on reçoit.

Le langage est synchrone, dans le sens où l'émission/réception de signaux est supposée ne pas prendre de temps.

En ESTEREL, un événement peut être formé par l'occurrence d'un seul signal ou de plusieurs signaux simultanés. Les événements vides sont également pris en compte: cas où aucun signal n'est arrivé.

Un signal peut être émis simultanément par plusieurs émetteurs. Ceci ne pose pas de problème pour un signal pur, mais pour un signal portant une valeur, il faut pouvoir définir la valeur du signal lorsque plusieurs émetteurs entrent en collision. La solution adoptée est empruntée à MILNER: à chaque signal est associé une opération associative-commutative dénotée "*".

Il n'y a pas de temps universel unique en ESTEREL: chaque signal définit une "unité de temps", qui est l'intervalle séparant deux émissions successives de ce signal. Autrement dit, chaque signal peut être considéré comme une horloge, dont chaque émission serait un "top". De plus, rien n'exige que deux émissions successives d'un signal donné soient séparées par des délais constants. L'unité de temps définie par chaque signal peut donc être une unité variable.

Les primitives temporelles d'ESTEREL sont extrêmement restreintes:

- Test de présence d'un signal à un instant donné,

  ```
  present <signal> then <inst> else <inst> end
  ```

  le sens en est clair.

- Le chien de garde,

  ```
  do <inst> watching <occ>
  ```

  où le corps `<inst>` est une instruction quelconque, et où l'occurrence `<occ>` a la forme suivante:

  ```
  [<expression>] <signal>
  ```

  Cette instruction sert à donner une limite temporelle à l'exécution de son corps.

Des réactions très complexes peuvent être exprimées dans le langage synchrone ESTEREL, mais il y a un prix à payer pour cette expressivité (c.f. [BdS95a, p.4]): des protocoles coûteux sont nécessaires pour les implémentations distribuées des applications. En particulier, des messages doivent être échangés non seulement pour les signaux présents, mais également pour les absents. Concernant ce point, une des difficultés majeures en programmation synchrone est de supprimer le problème des $\perp$'s (absence de signal à un instant donné) à la compilation, de telle manière que le système d'exécution n'ait pas à s'en préoccuper (voir par exemple SIGNAL).

Le compilateur ESTEREL traduit un programme en un automate fini déterministe. Le programme est dans un premier temps traduit dans un code intermédiaire, puis dans un langage cible (le langage C par exemple).

LUSTRE

LUSTRE [HCRP91a, HCRP91b, CPHP87] est un langage fonctionnel synchrone fondé sur une approche "flot de données" à assignation unique.

Les variables ou expressions en LUSTRE représentent des flots, c'est-à-dire un couple {*suite de valeurs, horloge*}.

Toute variable qui n'est pas une entrée du programme est définie par une équation et une seule.

Un programme LUSTRE est ainsi constitué d'un ensemble de définitions récursives de variables sous forme d'équations: $x_i = exp_i$, où la variable x_i et l'expression exp_i ont même suite de valeurs, et même horloge.

Tout programme ou fragment de programme en LUSTRE a un comportement cyclique, qui définit son *horloge de base*, à partir de laquelle toutes les autres horloges sont dérivées (ce qui permet d'atteindre la notion de temps multiforme). En d'autres termes, toutes les suites de valeurs en LUSTRE sont synchronisées par une "horloge universelle".

Les opérateurs classiques sur les types de base (arithmétiques, booléens, conditionnels), ainsi que les fonctions importées du langage hôte, ne peuvent être appliqués qu'à des opérandes de même horloge.

En plus de ces opérateurs, LUSTRE possède un nombre restreint d'opérateurs "temporels" agissant spécifiquement sur les flots:

- L'opérateur **pre** sert à mémoriser la valeur précédente d'une expression. Les flots **E** et **pre(E)** ont la même horloge. Une valeur particulière *nil* représente une valeur indéfinie, correspondant à l'absence d'initialisation. Par exemple, si $(e_1, e_2, \ldots)$ est la suite des valeurs de l'expression **E**, alors **pre(E)** est une expression dont la suite de valeurs est $(nil, e_1, e_2, \ldots)$.
- L'opérateur $\rightarrow$ sert à définir une valeur initiale: si **E** et **F** sont des expressions sur la même horloge, de suites de valeurs respectives $(e_1, e_2, \ldots)$ et $(f_1, f_2, \ldots)$, alors **E** $\rightarrow$ **F** est une expression de même horloge que **E** et **F**, et dont la suite de valeurs est $(e_1, f_2, \ldots)$.
- L'opérateur **when** sert à "filtrer" une expression sur une horloge plus lente: si **B** est une expression booléenne sur la même horloge que l'expression **E**, alors **E when B** est une expression sur l'horloge définie par **B**, et dont la suite des valeurs est extraite de celle de **E** en ne conservant que les valeurs dont l'indice correspond à des valeurs *vrai* dans la suite des valeurs de **B**.
- L'opérateur **current** sert à "projeter" une expression sur l'horloge immédiatement plus rapide. Soit **E** une expression sur une horloge différente de l'horloge de base, et soit **B** l'expression booléenne ayant engendré cette horloge, alors **current E** est une expression sur la même horloge que **B**, et dont la valeur à chaque instant de cette horloge est la valeur prise par **E** au dernier instant où **B** valait *vrai*.

Par exemple,

X	x_1	x_2	x_3	x_4	x_5	x_6
B	*faux*	*vrai*	*faux*	*faux*	*vrai*	*faux*
Y = X when B		x_2			x_5	
Z = current Y	*nil*	x_2	x_2	x_2	x_5	x_5
W = pre(Y)		*nil*			x_2	
Q = Y $\rightarrow$ W		x_2			x_2	

LUSTRE manipule des fonctions entrée-sortie générales, les nœuds, et les assemble à la manière d'un bloc-diagramme; le résultat est à son tour considéré comme un nœud LUSTRE si le compilateur est capable de vérifier qu'il s'agit bien d'une fonction d'E/S.

L'aspect synchrone apparaît dans la modélisation de la notion d'événement, et dans le fait que les calculs ne provoquent pas de décalage sur l'horloge universelle.

Le compilateur LUSTRE produit un code objet qui est le même que celui produit par ESTEREL, ce qui permet d'utiliser des outils commun.

SIGNAL

SIGNAL [BB91, LGM+91, BLG90, LGG90] est un langage fonctionnel fondé sur une approche flot de données. On considère ici qu'un programme est un système d'équations dynamiques pour lesquelles certains signaux peuvent avoir, à un moment donné, la particularité d'être absents (notion commune à LUSTRE et SIGNAL).

Dans le langage SIGNAL on considère que la notion d'horloge est associée au *signal* et non pas au module de programmation appelé processus. Un processus SIGNAL définit des relations entre signaux exactement à la manière d'un schéma-bloc. Un calcul d'horloges (plus complexe que dans le cas de LUSTRE) permet de savoir sur quelle horloge est cadencée une expression.

La sémantique de SIGNAL est basée sur un noyau minimum d'opérateurs mais suffisant pour exprimer toutes les relations temporelles complexes pouvant intervenir naturellement dans un programme temps-réel. Ces opérateurs, appelés également processus élémentaires se répartissent en deux classes: les processus monochrones qui manipulent un seul index temporel et les processus polychrones qui en manipulent plusieurs.

1. **Processus monochrones.** Ils imposent une synchronisation forte de tous les signaux référencés dans l'expression.

   - **L'extension directe de fonctions à des fonctions agissant sur les flots.** Toute fonction $f()$,

     $$a_{n+1} := f(a_1, \ldots, a_n)$$

     est ainsi implicitement étendue à une fonction opérant sur des flots, telle que $\forall\ t$,

     $$a_{n+1}(t) := f(a_1(t), \ldots, a_n(t))$$

     où les a_i's sont des signaux synchrones du programme, et t dénote le $t^{\text{ème}}$ élément de la séquence a_i.

   - **Le retard sur signal.** Dans les langages synchrones, toute occurrence observée peut être immédiatement prise en compte en raison de l'instantanéité des traitements. Par conséquent, les événements sont implicitement tous fugaces. Ainsi, la conception d'applications où la prise en compte de certains événements doit être retardée, impose l'écriture de tâches complémentaires uniquement chargées de cette mémorisation temporaire: c'est l'opérateur de retard dénoté **$** en SIGNAL. L'expression,

     $$x := y \ \$ \ k$$

     dénote la relation temporelle suivante:

     $$\forall\ t > 1, x(t) := y(t-1), x(1) = k$$

     où t est un index de temps, et k est une valeur initiale. Comme dans le cas de LUSTRE les signaux $x(t)$ et $y(t-1)$ sont synchrones.

2. **Processus polychrones.** On regroupe sous cette dénomination l'ensemble des constructeurs permettant la manipulation de plusieurs horloges au sein d'une même expression. Le langage SIGNAL dispose de deux constructeurs polychrones: l'opérateur **when** d'extraction de sous-séquences de valeurs (ou sous-échantillonnage) —et l'opérateur **default** d'entrelacement de signaux (ou sur-échantillonnage).

   - **L'extraction de sous-séquences de valeurs.** Il s'agit d'un opérateur commun à de nombreux langages synchrones (e.g. LUSTRE, $8^{1/2}$). L'expression,

     $$x := y \ \texttt{when} \ a$$

où a est un signal booléen, affecte à la sous-séquence x, la valeur courante de y lorsque les signaux y et a sont définis, et que a porte la valeur *vrai*.

À la différence du **when** en LUSTRE, les signaux y et a ne sont pas nécessairement synchrones. La sémantique de l'opérateur de sous-échantillonnage en SIGNAL est en ce sens moins contraignante qu'en LUSTRE.

- **Le sur-échantillonnage de valeurs.** L'opérateur **default** en SIGNAL de sur-échantillonnage accroit considérablement la puissance d'expression de SIGNAL par rapport à la majorité des autres langages synchrones (i.e. LUSTRE, ESTEREL, $8^{1/2}$, ...). L'expression,

$$x := u \texttt{ default } v$$

affecte au signal x la valeur courante de u lorsqu'elle est définie, et la valeur courante du signal v lorsque u est indéfini et que v est défini. Une priorité existe donc entre u et v qui rend la sémantique de l'opérateur déterministe.

Une description plus détaillée du langage SIGNAL est donnée en annexe du document.

Statemate et les Statecharts

Les STATECHARTS [Har87] fournissent une représentation graphique et hiérarchisée des automates. Ils forment un des trois langages (les deux autres sont les *activity-charts* et les *module-charts*) utilisés par STATEMATE [HLN+90], un environnement graphique de spécification, d'analyse et de conception de systèmes réactifs complexes.

Les STATECHARTS décrivent le comportement du système, en termes d'états, de transitions et d'événements: un système peut se trouver dans différents états, et subir des transitions d'un état à l'autre suite à l'apparition de certains événements. Ces événements sont des signaux

- qui sont envoyés à l'ensemble du système par *broadcast*,
- qui sont reçus instantanément.

Par exemple, une montre digitale (c.f. [Har87] ou [Dzi90]) peut se trouver dans l'état "*affichage de l'heure*" ou dans l'état "*affichage du chronomètre*", et transiter d'un état à l'autre lorsque l'on appuie sur le bouton ad hoc.

C'est parce que les émissions/réceptions de signaux ne prennent pas de temps, que le modèle des STATECHARTS peut être qualifié de synchrone.

D'autre part, STATEMATE utilise les *activity-charts* pour décrire la fonctionnalité d'un système, c'est-à-dire pour en identifier les fonctions, ainsi que les signaux ou les données circulant à l'intérieur du système, ou entre le système et son environnement. Cette description se fait en termes d'*actions* et d'*activités*. Les actions sont atomiques, par exemple: "*envoyer un signal*", "*lire une donnée*", "*afficher un chiffre sur le cadran d'une montre digitale*", etc. Elles sont hiérarchiquement composées en *activités*, qui sont donc un enchainement de (sous-)activités et/ou d'actions.

Le troisième langage de STATEMATE (les *module-charts*) décrivent la structure du système tel qu'il sera implémenté. Ils identifient ce que seront exactement les modules physiques et comment ils seront connectés via des canaux. Les *activity-charts* et les *module-charts* sont liés: les signaux et les données sont associés aux canaux, les activités sont implantées par des modules physiques.

Les STATECHARTS [Har87] fournissent une représentation graphique et hiérarchisée des automates. Il s'agit plus d'un mécanisme de spécification que d'un langage de programmation. Un état peut être vu comme tel à un niveau supérieur, ou comme un automate complexe à un niveau inférieur. Les états, simples ou structurés sont reliés par des transitions étiquetées par des signaux. Les actions instantanées sont effectuées au niveau des états.

Otto e Mezzo

Le langage $8^{1/2}$ (OTTO E MEZZO) [GS93, Gia91] est à ma connaissance, la seule expérience visant à connecter l'univers du data-parallélisme au monde synchrone. Il propose une approche du data-parallélisme fondée sur le modèle flot de données.

Un programme $8^{1/2}$ est un système d'équations où chaque équation définit un *tissu* (séquence potentiellement infinie de valeurs) à la manière de LUSTRE ou SIGNAL.

La classe des opérateurs temporels en $8^{1/2}$ est relativement sommaire:

- **Un opérateur de délai (notation: `$`).** Utilisé pour décaler ou "retarder" un *stream* tout entier, il suit la même sémantique que l'opérateur **`pred`** de LUSTRE.
- **Un échantillonneur (notation: `when`).** La sémantique de l'opérateur **`when`** d'extraction de sous-séquences de valeurs en $8^{1/2}$ est proche celle de SIGNAL.

Le calcul d'horloge en $8^{1/2}$ est également limité. Son objectif est de remplir les "trous" introduits par l'utilisation des opérateurs temporels. Pour détecter ces "trous", dans la progression relative d'un *stream*, un temps logique global est synthétisé. Ainsi, aux instants où un signal donné est défini ($\top$), on lui associe sa valeur instantanée, dans les autres cas ($\bot$) il prend la valeur qui lui était affectée au top précédent, si celle-ci existe, et une valeur conventionnelle *nil* dans le cas contraire.

Exemple (c.f. [Gia91, p.40]):

Événements	**0**	**1**	**2**	**3**	**4**	**5**	**6**
`X`	**1**	**2**	2	**3**	**4**	4	**5**
`B`	***nil***	*nil*	**FALSE**	**TRUE**	TRUE	**TRUE**	TRUE
`X when B`	*nil*	*nil*	*nil*	**3**	3	**4**	4

On a représenté en gras les instants auxquels les différents signaux sont définis, et en minuscule le remplissage des "trous".

La sémantique temporelle de $8^{1/2}$ accuse un défaut de "propreté". Ainsi par exemple, à la cinquième occurrence de temps (événement `4`), le signal `B` est indéfini, et par conséquent, le signal `X` l'est également. Avec le remplissage des "trous", le signal `B` se trouve affecté de la valeur *vrai* qui n'a plus aucun sens en terme de contrôle.

SML

SML [CLM91] est un langage impératif destiné à la manipulation d'événements. Il est conçu pour la réalisation de circuits. Il est basé sur une horloge universelle correspondant à l'horloge du circuit. Les actions telles que l'affectation ou l'émission prennent une unité de temps, mais plusieurs actions peuvent également être comprimées et réalisées en une seule unité de temps (primitive **`compress`**). Le langage n'est donc pas tout à fait synchrone.

Les programmes sont traduits en automates finis destinés à être intégrés dans du matériel.

Conclusion

Malgré son "jeune âge", l'approche synchrone a déjà engendré nombre de modèles et langages. Même si un certain nombre de divergences sémantiques existent, les langages les plus récents se rejoignent autour de quelques concepts clefs.

Alors que nous venons de présenter un certain nombre de langages et la manière dont ceux-ci apportent isolément leurs solutions aux problèmes temps-réel, il serait sans doute intéressant de synthétiser les différentes approches autour de quelques concepts majeurs, en jetant un regard sur les problèmes que ceux-ci peuvent poser en termes d'implémentation.

I.3.3 Synthèse

Un certain nombre de concepts clefs liés à l'utilisation des langages synchrones et à leur contexte d'implémentation peuvent être synthétisés: l'existence d'un temps de référence vs. un temps multiforme —la granularité des traitements —l'activation multiple —les aspects liés à la concurrence.

Temps de référence vs. temps multiforme

L'existence d'un temps de référence explicite (horloge universelle) restreint considérablement le pouvoir d'expression d'un langage synchrone dédié à la spécification des systèmes temps-réel: les relations complexes pouvant apparaître entre les différents signaux d'un programme sont plus difficiles à appréhender par l'utilisateur. Cette difficulté est notament rencontrée dans le langage SML.

Du point de vue des STATECHARTS, de LUSTRE, de SIGNAL, de $8^{1/2}$ et d'ESTEREL notament, aucun concept d'horloge universelle n'apparait lors de la spécification des relations temporelles. Chaque signal peut être considéré comme une horloge, dont l'unité de temps est l'intervalle séparant deux occurrences de ce signal. Cette propriété est un atout majeur dans la spécification de relations temporelles complexes. Une horloge unique est souvent néanmoins synthétisée lors de la phase de calcul d'horloges pour des propos d'implémentation.

Les langages qui ne font pas explicitement référence à un temps de base, adoptent généralement un mécanisme de diffusion d'horloges. C'est-à-dire que toutes les horloges d'un programme envoient leurs "tops" par *broadcast* à toutes les parties du système (e.g. STATECHARTS, ESTEREL, SL, RC).

Si on a en vue l'implémentation distribuée des systèmes, il est évident que la diffusion d'un grand nombre de signaux d'horloge à tout moment dans le réseau peut être considérablement contraignante.

Granularité

Les langages synchrones se caractérisent généralement par un fin niveau de granularité des actions, et que ce soit par exemple dans le cas de LUSTRE ou d'ESTEREL, les automates produits par le compilateur conduisent à une explosion du nombre des états qui rend impossible toute implémentation répartie des programmes au-delà d'un degré moyen de complexité.

Le langage SIGNAL rencontre le même type de difficultés si l'on considère le nombre importants de nœuds intermédiaires produits par le compilateur: nœuds de calcul d'horloge, nœuds d'expression du signal, nœuds de mémorisation.

En effet, concernant par exemple la mémorisation, dans les langages synchrones toute occurrence observée peut être immédiatement prise en compte en raison de l'instantanéité des traitements. Donc, les événements sont implicitement tous fugaces. Par conséquent, la conception d'applications où la prise en compte de certains événements doit être retardée, impose l'écriture de tâches complémentaires uniquement chargées de cette mémorisation temporaire [RCCE92, p.39].

Le problème peut même se manifester dès la phase de spécification des algorithmes avec les langages basés sur un formalisme de type réseau de Petri qui sont trop peu structurés et donnent des objets incompréhensibles au-delà d'une certaine taille (e.g. les graphes SDF).

Les applications complexes de traitement temps-réel de l'image appartiennent à l'ensemble des problèmes mixtes (c.f. SECTION I.2.1), c'est-à-dire qu'ils sont généralement dotés au niveau fonctionnel d'une lourde partie algorithmique. Les aspects transformationnels complexes en traitement d'image forcent à considérer un haut degré de granularité des opérations.

Multiple activations

Dans une approche distribuée, les coûts occasionnés par les échanges de données ne sont pas nuls. Il est donc fondamental de pouvoir en masquer la latence en occupant au maximum les différentes unités de calcul de l'architecture.

En traitement temps-réel, les structures adressées sont des séquences de valeurs dont il est possible de traiter en parallèle différentes occurrences dans le temps. On entend ainsi par l'expression *multiple activations*, la possibilité pour les unités de l'architecture, à un instant donné, d'effectuer des traitements sur des flots étiquetés par différents index temporels.

Si l'on prend pour exemple les langages synchrones LUSTRE et ESTEREL, le compilateur ne permet de produire que des automates dont l'appel n'exécute qu'une seule transition. En fait, jamais, la question fondamentale du pipelinage dans le temps des unités fonctionnelles n'est abordée dans la littérature portant sur les langages synchrones.

Le pipelinage des applications est même rendu intrinsèquement impossible dans des langages synchrones comme SL, REACTIVE C [BdS95a], ou REACTIVE OBJECTS [BDS95b]. En effet, la *globalité des instants* fait qu'un processus particulier n'est pas autorisé à s'exécuter pour l'instant suivant tant qu'il existe un autre processus qui ne soit pas terminé pour l'instant courant.

Concurrence

Pour programmer un système temps-réel complexe, il est nécessaire de décrire un ensemble de systèmes de processus en interaction constante avec un environnement physique. Ces systèmes sont soumis à des contraintes de temps-réel très strictes. Ils sont conçus comme un ensemble d'entités coopérantes pouvant évoluer en parallèle [Cos91]. Tous les aspects liés à la concurrence sont donc importants.

Il existe en général trois manières d'exprimer le parallélisme fonctionnel exploitable dans une application:

- **De manière explicite.** En laissant à l'utilisateur le soin d'exprimer lui-même le parallélisme.
- **S'en remettre à un outil.** Qui est chargé d'extraire tout le parallélisme disponible dans une application.
- **De manière implicite.** La sémantique opérationnelle du langage permet une exploitation naturelle du parallélisme.

Dans le langage ESTEREL notament, tous les aspect relatifs à la concurrence sont laissés entre les mains des développeurs par l'intermédiaire de l'opérateur de composition de modules "||", qui rend le parallélisme explicite au niveau de l'utilisateur.

En ce qui concerne l'exploitation du parallélisme, l'approche data-flow présente des atouts indéniables. Pourtant, des langages comme LUSTRE, qui sont basés sur une sémantique proche du modèle flot de données, sont en pratique très éloignés d'une implémentation purement data-flow.

Si on reprend par exemple, le modèle de construction des programmes répartis en LUSTRE, il passe curieusement par la construction du programme séquentiel global et la répartition de celui-ci. Alors que selon l'aveu même des auteurs: "*il est beaucoup plus difficile de paralléliser un programme séquentiel, que d'exécuter séquentiellement un programme parallèle*" ([HCRP91a], p.141).

Conclusion

L'étude comparée des différents concepts introduits par la programmation synchrone qui a été réalisée dans cette SECTION jetait un regard sur l'implémentation, où elle soulève un grand nombre de difficultés.

Il faut donc plutôt entrevoir l'approche synchrone comme un moyen efficace, souple et naturel pour spécifier et vérifier les contraintes temporelles complexes qui interviennent naturellement dans les problèmes temps-réel. Et rejeter l'implémentation sur des outils existants, validés et qui ne posent pas de problèmes de ce point de vue.

Il reste à déterminer, dans la "jungle" des langages et modèles existants, les outils synchrones les plus judicieux susceptibles de répondre à nos besoins en traitement temps-réel de l'image.

I.3.4 Vers un squelette synchrone "idéal"

Nous avons résumé dans cette partie les différents éléments nécessaires à la spécification des relations temporelles complexes pouvant intervenir dans les problèmes temps-réel.

Instantanéité et Simultanéité

Les concepts d'instantanéité et de simultanéité qui sont à la base de l'approche synchrone doivent bien entendu être pleinement respectés.

C'est-à-dire, instantanéité des traitements et des communications — coexistence simultanée de plusieurs temps logiques.

La possibilité d'effectuer des preuves formelles, des tests, de répéter l'exécution des programmes en dépend.

Avantages du modèle *flot de données*

Dans un modèle "*flot de données*" [2], tout système est constitué d'un réseau d'opérateurs agissant en parallèle au rythme de leurs entrées [Kah74]. Ce modèle général définit dans le contexte de la programmation parallèle présente un certain nombre d'avantages qui ont été largement évoqués dans les premières sections de ce document.

- **Séquencement implicite.** On vient de le voir, dès que toutes les entrées d'un opérateurs sont présentes, il peut s'exécuter. Ainsi, les seules contraintes de séquencement et de synchronisation sur les actions, sont les contraintes de dépendance entre les données. Le formalisme utilisé exprime donc naturellement le parallélisme. Cette fonctionnalité est préférable plutôt que de laisser à un outil ou encore au programmeur (e.g. l'instruction **PAR** d'OCCAM [LTD88]) le soin d'extraire le parallélisme.

- **Exploitation du parallélisme.** Le séquencement implicite du programme permet potentiellement une exploitation optimale du parallélisme; lors de la phase d'extraction et lors de l'exécution proprement dite. Un ordonnancement minimal des opérations est automatiquement déduit du programme ce qui permet en conséquence une expression maximale du parallélisme.

- **Déterminisme.** Dans un langage asynchrone comme OCCAM, l'expression du parallélisme passe par le non-déterminisme de l'exécution et peut engendrer des interblocages. Pour obtenir le déterminisme des résultats du programme, le programmeur doit exprimer "du séquencement" par des sémaphores, des gardes, ... S'il exprime trop de séquencement, il y a perte de parallélisme exploitable, par contre, s'il exprime trop peu de séquencement, le programme pourra calculer des résultats différents suivant les aléas de l'exécution. Un langage à flot de données ne souffre pas de cet inconvénient: l'expression du séquencement est implicite et correspond à "juste ce qu'il est nécessaire" pour assurer un résultat déterminé.

2. Le lecteur intéressé trouvera dans [Den91] une perspective historique du data-flow.

- **Propreté.** Les formalismes basés sur le modèle flot de données possèdent en général une "*propreté*" mathématique qui fait défaut aux langages impératifs classiques dans lesquels les notions de mémoire et d'affectation peuvent introduire des effets de bord complexes à analyser. Un programme data-flow peut être vu comme un ensemble d'équations mathématiques et toute référence à une variable peut être remplacée par sa définition. Par conséquent, il est plus facile de raisonner formellement sur les programmes. Cela autorise la transformation automatique des programmes, en vue de leur implémentation ou encore de leur optimisation. Enfin, pouvoir considérer un programme comme un système d'équations a pour avantage d'assurer que le programme calculera un résultat, si ce résultat est formellement dérivable du système d'équations.

- **Modularité.** La description sous forme de réseaux d'opérateurs fournit directement une représentation graphique des programmes. En outre, cette représentation facilite la décomposition hiérarchique des applications: un sous-réseau peut être considéré comme un *macro*-opérateur.

- **Flot de données vs. flot de contrôle.** Les systèmes temps-réel sont des systèmes informatiques qui réagissent continûment à leur environnement physique. Ils reçoivent en permanence des flots de données dont l'occurrence dicte le comportement du système et émettent en sortie des flots de données à destination de leur environnement. Dans ces systèmes le flot de données est donc prévalant et induit naturellement une sémantique de type data-flow.

Spécification des relations temporelles

La spécification des relations temporelles passe par la définition d'un nouvel ensemble d'opérateurs qui n'existent naturellement pas en informatique classique. Certains d'entre eux sont communs à de nombreux langages synchrones. Néanmoins, des différences sémantiques importantes peuvent apparaître d'un langage à l'autre.

- **Fonctions agissant sur des flots.** Les fonctions élémentaires de traitement d'images incluses dans les bibliothèques standard et utilisateur, écrites dans un langage hôte vont être amenées dans l'enchaînement logique et temporel du programme à travailler sur les flots.

  Toute fonction doit donc pouvoir être implicitement étendue à une fonction agissant sur des séquences potentiellement infinies de valeurs.

  Les tâches élémentaires de traitement d'image étant des fonctions externes, tous les signaux référencés en entrée de la fonction doivent être présents au moment de l'appel. Ce qui implicitement signifie que les entrées sont synchrones.

  D'autre part, le principe d'instantanéité (l'exécution d'une tâche ne prend pas de temps) induit à son tour que toutes les sorties d'une fonction sont produites instantanément (elles sont synchrones entre elles), elles sont en outre synchrones avec les entrées. On parlera d'opérateur monochrone: un seul index de temps est manipulé.

- **Retard.** La possibilité d'accéder aux valeurs passées d'un signal est fondamentale en traitement d'image, on pourrait par exemple citer le cas de l'algorithme MPEG [GDM96b], de l'estimateur de mouvement, etc.

  Le retard sur signal ne peut cependant être autorisé que de manière explicite. En effet, dans le domaine particulier du traitement d'image notamment, il serait inconcevable de mémoriser des flots potentiellement infinis d'images pour prévenir d'hypothétiques utilisations de ces valeurs.

  La sémantique de l'opérateur de retard est la même dans de nombreux langages synchrones. Elle diffère quelque peu sur la possibilité ou non d'affecter immédiatement une valeur d'initialisation.

Dans tous les langages, cet opérateur est monochrone, c'est-à-dire que signal d'entrée et flot décalé sont synchrones.

- **Extraction de valeurs.** Tout comme l'extension implicite des fonctions de calcul, et le retard sur signal, l'extraction de sous-séquence de valeurs est un opérateur temporel fréquemment rencontré en programmation synchrone (c.f. LUSTRE, SIGNAL, $8^{1/2}$, graphes SDF, ...), il traduit des besoins exprimés en traitement temps-réel.

  Des différences sémantiques importantes se creusent entre les différents langages au sujet du sous-échantillonneur. En fait l'extraction de valeurs appartient à une gamme "supérieure" d'opérateurs plus sensible à définir dans la mesure où il manipule implicitement plusieurs index de temps. On parlera d'opérateur polychrone.

  La forme générale d'une opération de sous-échantillonnage est:

  ```
  X = Y when B
  ```

  où `B` est un flot booléen, et `X` et `Y` sont des signaux de même type, ou de type compatible.

  En LUSTRE par exemple, les signaux `B` et `Y` doivent être synchrones. La valeur de `Y` est affectée à `X` lorsque le signal `B` porte la valeur *vrai*. Le signal ainsi produit possède une horloge inférieure ou égale à celle de `Y` et `B`.

  Par contre en SIGNAL, aucune restriction n'est faite sur l'horloge de `B`, par conséquent `Y` et `B` peuvent avoir des horloges différentes. `X` reçoit ainsi la valeur de `Y` lorsque `Y` et `B` sont définis et que `B` porte de surcroît la valeur *vrai*.

  La sémantique du **when** est plus riche en SIGNAL, dans le sens où elle est beaucoup moins contraignante pour l'utilisateur: celui-ci n'a pas besoin de veiller à ce que les signaux `Y` et `B` soient synchrones. Dans le cas de relations temporelles complexes cela constitue un avantage indéniable. Cette remarque, signifie également que le calcul d'horloge de SIGNAL est beaucoup plus sophistiqué que dans les autres langages synchrones.

- **Sur-échantillonnage de signaux.** L'opération de sur-échantillonnage permet de "mélanger" ou entrelacer différents flots de données. Dans le domaine du traitement temps-réel de l'image, l'intérêt de cette fonctionalité est évident. En TV HD par exemple, il est fréquent de stocker les séquences vidéo sur deux magnétoscopes en répartissant les trames paires et impaires respectivement sur l'un et l'autre. Le flot d'images à traiter est constitué de l'entrelacement des sources vidéos.

  Peu de langages synchrones offrent la possibilité d'exprimer le sur-échantillonnage de signal. Il est évident, qu'un tel opérateur induit des relations temporelles complexes à résoudre.

  Le `current` de LUSTRE par exemple, malgré les apparences, ne remplit pas les fonctionalités attendues. Il correspond plutôt au "remplissage de trous" à la $8^{1/2}$. En effet, dans une expression comme (voir SECTION I.3.2):

  ```
  Y = X when B
  Z = current Y
  ```

  le signal `Z` est synchronisé avec `X` et `B`, et les valeurs de `X` aux instants où `B` est *vrai* sont simplement projetés sur les instants pour lesquels `B` est *faux*. En SIGNAL ce système s'écrirait à l'aide de l'opérateur dérivé `cell`:

  ```
  Y := X when B
  Z := Y cell (not B)
  ```

c'est-à-dire que lorsque le signal `Y` est présent, alors `Z` prend la valeur de `Y`, et dans le cas contraire, si le booléen `B` est *faux*, alors `Z` prend la valeur associée à `Y` la dernière fois que `B` était *vrai*.

L'opérateur de sur-échantillonnage (**default**) en SIGNAL est beaucoup plus intéressant dans la mesure où il permet effectivement de mélanger des signaux différents, sans contraintes a priori sur les horloges. Par exemple,

```
IMAGE := TRAME_PAIRE default TRAME_IMPAIRE
```

un ordre de priorité existe sur les entrées afin de garantir le déterminisme de l'opérateur. Le signal **IMAGE** reçoit ainsi une valeur en fonction de la présence relative des signaux **TRAME_PAIRE** et **TRAME_IMPAIRE**. La souplesse d'utilisation et l'expressivité de l'opérateur de sur-échantillonnage en SIGNAL se paye bien entendu par un processus formel de résolution des contraintes temporelles particulièrement élaboré.

Conclusion

Plusieurs langages synchrones respectant pleinement les principes d'instantanéité et de simultanéité, sont directement basés sur une sémantique flot de données. Même si au niveau de leur implémentation répartie, ils sont loin d'appliquer des principes data-flow purs (e.g. LUSTRE, SIGNAL, $8^{1/2}$, ...).

Des distinctions importantes apparaissent cependant au niveau des opérateurs de traitement du signal. Des opérateurs que nous voulons de très haut niveau et les moins contraignants possible pour l'utilisateur.

La puissance d'expressivité des opérateurs sur les flots est intrinsèquement liée au degré de complexité du processus formel de résolution des contraintes temporelles. Le noyau proposé par le langage synchrone SIGNAL constitue un ensemble minimum et suffisant d'opérateurs permettant d'exprimer de manière naturelle et efficace toutes les relations temporelles complexes qui interviennent dans les grandes applications de traitement temps-réel. Consécutivement, le calcul d'horloge de SIGNAL s'avère être le processus formel de résolution des contraintes temporelles le plus sophistiqué.

Le langage synchrone SIGNAL est donc un bon candidat pour exprimer et résoudre les problèmes temporels posés par les applications complexes de traitement temps-réel de l'image.

I.4 Approche mixte synchrone-asynchrone

Nous avons évoqué dans les sections précédentes, les problèmes posés par une approche purement synchrone ou asynchrone dans la résolution des problèmes temps-réel complexes.

Les notions de synchronisme et d'asynchronisme qualifient ici la manière de raisonner sur le temps, et ne se réfèrent absolument pas à une réalité physique.

Le modèle que nous proposons se base sur l'expérience acquise dans les différentes approches existantes pour tenter d'en tirer le meilleur parti.

Ce travail s'inscrit à la fois dans le cadre du projet SM-IMP et offre une alternative au modèle d'implémentation proposé sur la machine P^3I (c.f. figure I.3).

Les études menées dans SM-IMP concernant le traitement des applications complexes de traitement temps-réel de l'image et du signal, ont abouti à la définition d'une architecture hétérogène distribuée (voir figure I.2).

Cette architecture ressemble à un *"gros"* réseau MIMD, dans lequel chaque unité de traitement peut fonctionner sous un mode de contrôle différent. L'architecture est ouverte et donc susceptible de recevoir de nouvelles cartes.

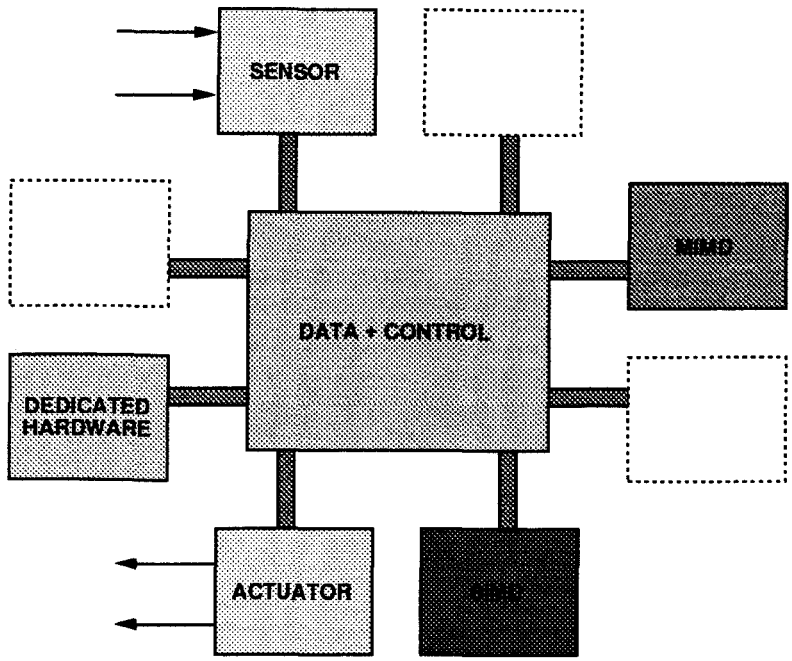


FIG. I.2 - *Modèle d'architecture hétérogène distribuée.*

Différentes hypothèses de couplage des unités de traitement ont été avancées:

- **Couplage intime.** Pour une tâche donnée, les différentes unités coopèrent au niveau de l'instruction élémentaire. Dans la mesure où nous avons choisi de travailler au niveau de la tâche (toute tâche du système est entièrement exécutée sur une seule unité, le traitement ne peut occasionner la moindre communication), ce choix semble impropre. Dans le cas général il serait en effet difficile de masquer la latence des échanges par un aussi fin degré de granularité des opérations.

- **Couplage fort.** Dans ce cas de figure, le coût de la synchronisation est faible. Au niveau de l'implémentation, cette caractéristique autorise aussi bien des systèmes d'échanges de données complètement distribués que centralisés ou semi-centralisés. Le principe est que les surcoûts occasionnés par la centralisation des échanges de données (dépendances de calcul et de contrôle) peuvent être masqués par des traitements, ceci suppose que l'on travaille à un degré supérieur de granularité des opérations. Même si elles existent (ex. P^3I), peu d'architectures permettent de faire une telle hypothèse, et dans un cadre plus général nous ne pouvons faire abstraction du surcoût qui serait occasionné par un processus centralisé ou semi-centralisé des échanges.

- **Couplage moyen.** Nous nous sommes situés dans ce contexte. Ici les coûts occasionnés par les échanges de données sont élevés, ce qui nous amène à tirer au moins deux conclusions:

  1. La granularité des traitements doit être élevée au regard des coûts induits par les communications. Le principe est que la durée des temps de traitement doit au moins être égale à celle des communications afin d'être masqués.
  2. Les échanges de données (calcul et contrôle) ne peuvent pas être centralisés car cela induirait inévitablement des surcoûts. Ainsi, ces échanges doivent être totalement répartis entre les les différentes tâches de manière à ce que chacune d'entre elles soit directement connectée à un sous-ensemble de tâches du système par l'intermédiaire de ses dépendances amont et aval.

- **Couplage faible.** De telles architectures sont conçues pour des applications faiblement coopérantes, ou pour lesquelles les temps de réponse entrée/sortie sont beaucoup moins cruciaux. Par définition, ces architectures ne sont donc pas adaptées au traitement temps-réel.

Dans l'hypothèse d'une architecture *"moyennement"* couplée, notre rôle a été de définir un modèle à la fois de conception, validation et implémentation des systèmes temps-réel de traitement de l'image.

Nous nous sommes largement appuyés dans notre démarche sur l'expérience acquise dans le projet P^3I, une plateforme parallèle, hétérogène, et généraliste pour le traitement vidéo temps-réel développée dans les laboratoires de THOMSON.

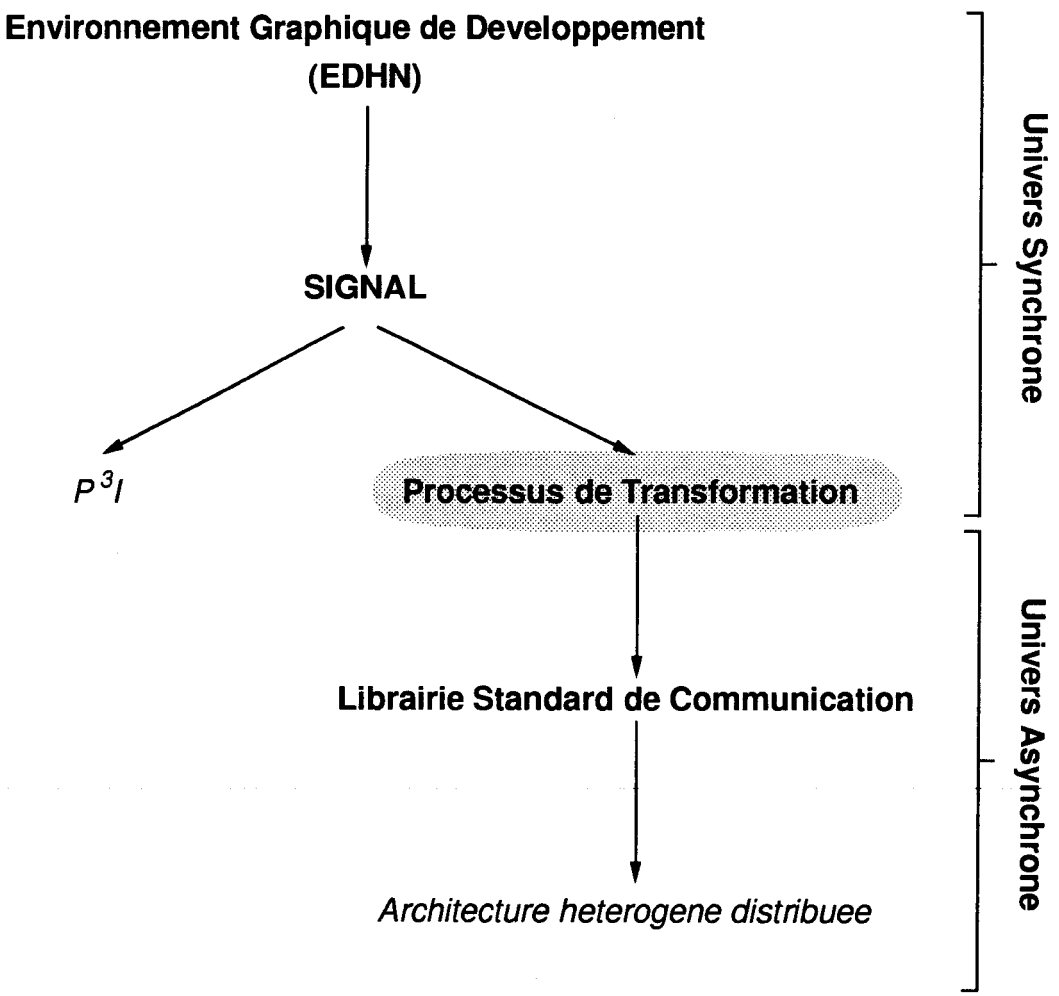


FIG. I.3 - *Mise en parallèle avec l'optique P^3I.*

Ce travail pourrait en effet être mis en parallèle avec ce qui a été réalisé dans le cadre de P^3I (cf. figure I.3): à partir du même schéma de compilation, moyennant une phase formelle de transformation des graphes, nous avons étudié la possibilité d'exécuter des applications vidéo temps-réel sur des réseaux hétérogènes de machines.

I.4.1 L'approche proposée

Nous avons travaillé dans le cadre de cette thèse sur un modèle d'implémentation des applications de traitement temps-réel de l'image sur les architectures hétérogènes distribuées.

À partir d'une méthode haut-niveau de spécification des problèmes sous l'hypothèse de synchronisme, nous proposons un modèle d'implémentation traditionnel sous forme de processus séquentiels concurrents communicants.

Le point fort de notre approche est donc de conjuguer les avantages respectifs des modèles synchrones et asynchrones dans la résolution des problèmes complexes de traitement temps-réel de l'image. C'est-à-dire sous l'hypothèse de synchronisme, permettre à la fois de raisonner simplement et efficacement sur le temps, et garantir statiquement l'existence et l'unicité de solution à tout problème donné. Deuxièmement, permettre des implémentations efficaces sur un grand nombre de plateformes avec une approche asynchrone qui de ce point de vue présente de nombreux atouts.

L'approche que nous préconisons en traitement temps-réel de l'image est donc mixte: une approche synchrone dans la partie conception et validation des applications —et un modèle d'implémentation asynchrone traditionnel.

Conception des systèmes

Une application complexe de traitement temps-réel de l'image se décompose naturellement en un ensemble de tâches à résoudre: seuillage, convolution, mais aussi acquisition et affichage d'images. Celles-ci sont en général intégrées dans des bibliothèques standard ou utilisateur. Le code des fonctions élémentaires de traitement d'images est séquentiel et écrit dans un langage de programmation qui nous est a priori indifférent.

Les aspects temps-réel résident dans la nature des données échangées et traitées par ces fonctions; il s'agit en effet de flots ou séquences potentiellement infinies. Des capteurs tels que caméras, tableaux de contrôle, etc, alimentent le système en flots d'entrées. Des acteurs tels que moniteurs, robots, etc, agissent continûment en sortie sur l'environnement physique (voir figure I.4).

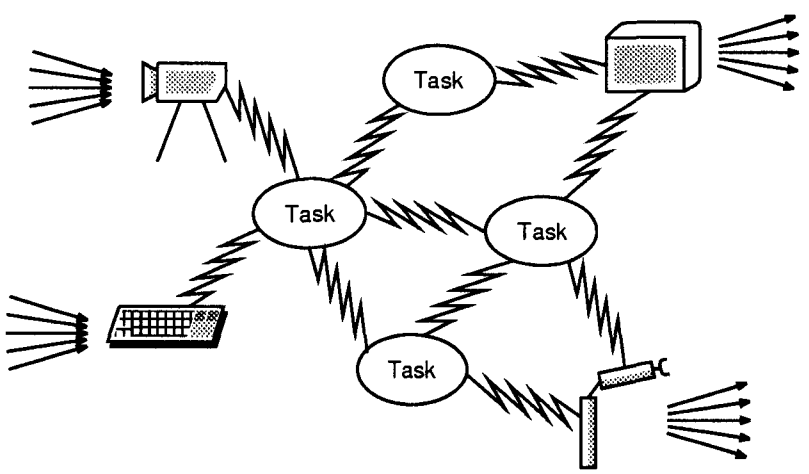


FIG. I.4 - *Spécification des problèmes temps-réel complexes de traitement de l'image.*

L'agencement logique des tâches est déterminé par les dépendances de calcul spécifiées par l'utilisateur selon un mode *data-driven* standard à assignation unique. Nous avons déjà préalable-

ment justifié le choix d'une telle approche. On peut remarquer que tout système est ainsi composé de deux grands types de fonctions: les fonctions en relation directe avec l'environnement physique (*capteurs* et *acteurs*) —les fonctions de calcul pur. Ces dernières manipulent des entrées/sorties à la manière de n'importe quel autre type d'appel de fonction traditionnel.

Les fonctions élémentaires de traitement de l'image ne communiquent ainsi que sur leurs entrées et leurs sorties: signaux du programme. Des tâches particulières, et en pratique tout programme devra au moins en comporter une, peuvent produire des signaux de sortie sans prendre d'entrée: elles correspondent à des fonctions d'acquisition pures (capteurs). Parallèlement, un certain nombre de tâches peuvent recevoir des entrées sans produire de sortie, elles vont correspondre à des acteurs purs.

La notion de temps-réel est également liée aux relations particulières qui peuvent exister entre les différents signaux du programme. Ces relations sont exprimées par des opérateurs spécifiques de traitement du signal: sur- et sous-échantillonnage, retard entre événements, et vont déterminer l'enchaînement temporel des tâches.

L'agencement logique et temporel des tâches élémentaires de traitement de l'image a été dénommé *squelette synchrone* du programme. Il repose donc sur une structuration mathématique des dépendances de calcul et des relations temporelles complexes pouvant exister entre les signaux, exprimées au moyen d'un ensemble d'opérateurs "idéaux" décrits précédemment, en se plaçant sous l'hypothèse de synchronisme.

En pratique, le squelette synchrone constitue un programme SIGNAL, ou du moins peut être implémenté sous la forme d'un programme SIGNAL afin d'en permettre la validation formelle.

Vérification

La phase de validation des programmes est réalisée au moyen du calcul d'horloge de SIGNAL. Le calcul d'horloges est un processus formel de résolution des contraintes temporelles, qui permet de garantir statiquement qu'une solution existe pour tout programme donné, et que celle-ci est unique.

Ce processus opère sur la notion d'horloge implicitement définie par chaque signal d'un programme et qui représente sa fonction de contrôle. Les relations temporelles complexes exprimées par les opérateurs de traitement du signal sont ainsi formalisées pour être résolues par le calcul d'horloges.

Un signal porte à la fois une information de valeur et de contrôle. La phase de compilation SIGNAL, scinde ces deux aspects en définissant un système d'équations d'horloges d'une part, et un ensemble de dépendances de calcul d'autre part (voir figure I.5).

En pratique, le calcul d'horloges produit un ensemble de fichiers dénotés SYNDEX décrivant un couple: Graphe des Dépendances Conditionnées (GDC), et Arbre d'horloges qui synthétisent pour le système d'exécution l'agencement logique et temporel des dépendances de calcul.

Dans le détail, la résolution de tout système passe par la définition d'un nombre important d'expressions intermédiaires d'une part, et via la création d'un ensemble de dépendances de type *signal-vers-horloge*, et *horloge-vers-horloge* d'autre part.

- Afin de faciliter le processus de résolution du système, les expressions de traitement du signal sont décomposées par le compilateur en expressions élémentaires définissant un ensemble de signaux intermédiaires. Par exemple, la compilation de l'expression [Bes92]

```
X := E when (Y default Z)
```

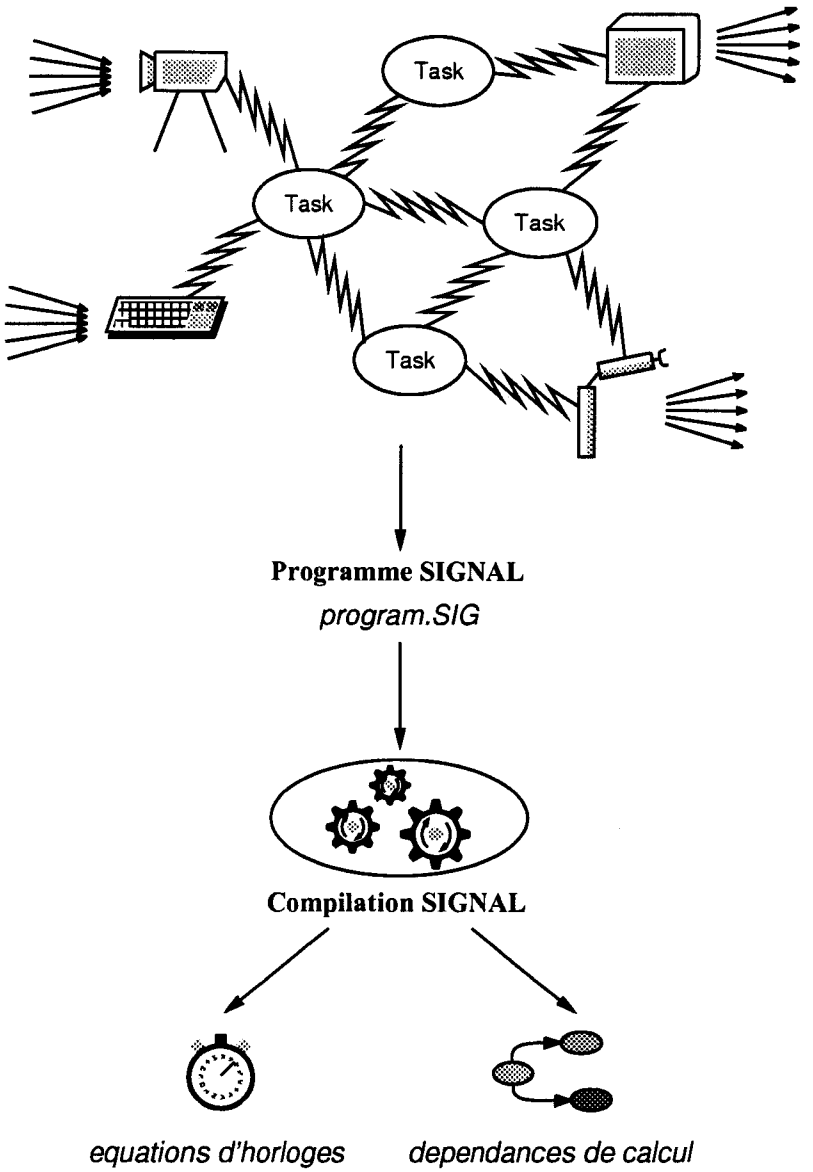


FIG. I.5 - *Calcul d'horloge.*

est obtenue par la compilation des expressions

```
X1  :=  Y default Z
X   :=  E when X1
```

où `X1` est un signal créé pour les besoins de la résolution.

- Les systèmes temps-réel sont définis par un ensemble de fonctions à réaliser, l'agencement logique de ces tâches étant déterminé par des dépendances de calcul. Les applications sont donc entièrement spécifiées par des dépendances de type *signal-vers-signal*. La vérification temporelle des applications est ainsi rejetée sur le compilateur, qui est en charge de synthétiser du contrôle pour le système d'exécution. De nouvelles dépendances de type *signal-vers-horloge*, et *horloge-vers-horloge* sont ainsi créées.

Le contrôle du programme passe dès lors par la manipulation explicite d'un grand nombre d'horloges par le système d'exécution, ainsi que par un nombre non moins conséquent de signaux et nœuds de calcul intermédiaires (calcul d'horloges, nœuds de mémorisation, etc). Une implémentation CSP directe sans transformation du graphe des dépendances conditionnées n'est donc pas envisageable.

Du squelette synchrone vers l'implémentation Asynchrone

Dans le modèle d'implémentation que nous proposons, le système n'est plus constitué que de fonctions de calcul, ou tâches, telles qu'elles sont représentées dans le schéma de la figure I.6.

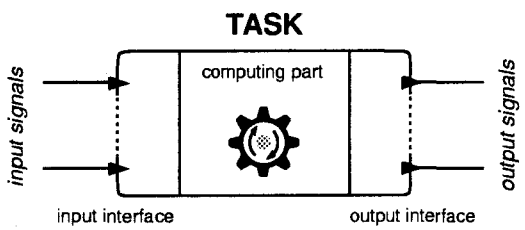


FIG. I.6 - *Une tâche de l'application.*

La fonction de calcul de chaque tâche est enserrée entre une interface d'entrée et une interface de sortie. La première de ces interfaces, ainsi que son nom l'indique, est chargée de recueillir les signaux en amont nécessaires au déclenchement de la fonction de calcul. Tandis que la seconde, transmet les signaux produits en fin de traitement, et nécessaires au déclenchement des tâches en aval.

Dans le système final, les tâches sont connectées les unes aux autres par des liens directs de dépendance. C'est-à-dire, que les seuls signaux circulant dans le réseau sont des flots produits par les différentes fonctions de calcul. Et que seuls les flots participant à la définition d'une entrée de tâche en aval sont conservés (cf. figure I.7). On peut donc dire que les processus de transformation sont appliqués dans une optique *demand-driven*.

Ce qui permet par réciproque de conclure que les expressions définissant des flots en entrée de tâche, et référant des signaux intermédiaires doivent être transformées pour ne faire apparaître que des signaux de sortie en provenance de tâches amont.

Les horloges du systèmes sont quant à elles soit récursivement définies par des formules sur horloges, soit par des expressions booléennes pouvant également référer des signaux intermédiaires.

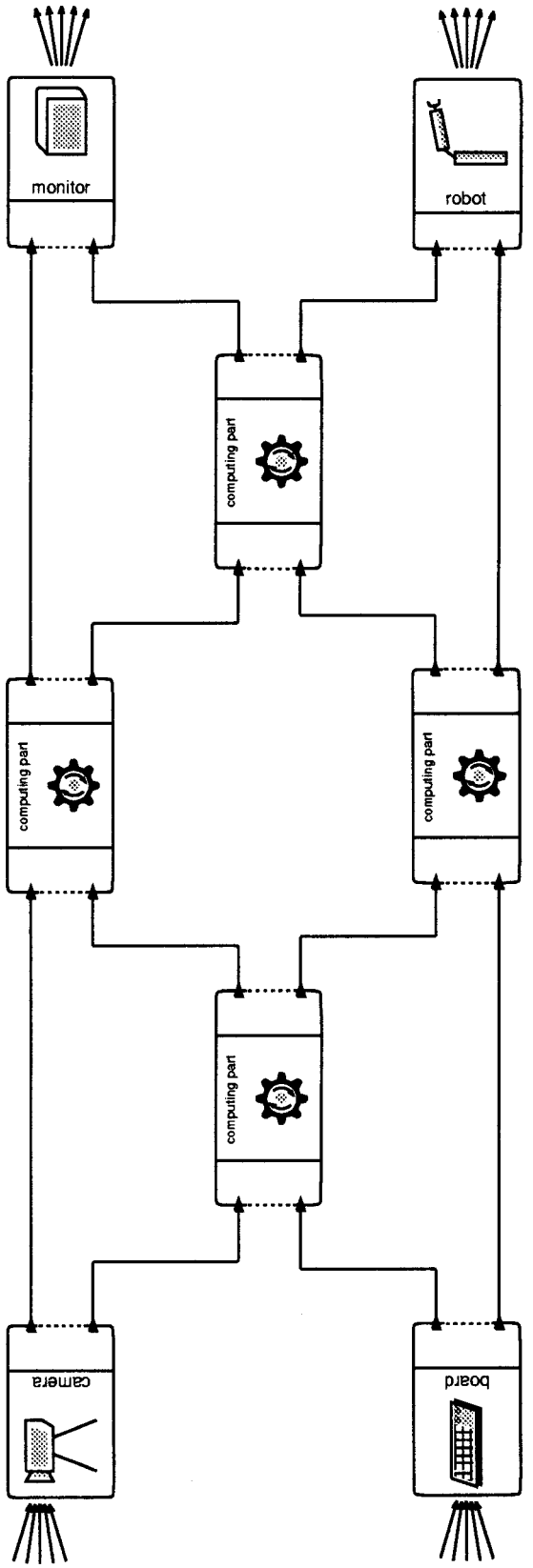


FIG. I.7 - *Implémentation CSP.*

Elles seront donc également transformées de manière à être évaluées par des expressions normales sur signaux de sortie du programme.

Les interfaces d'entrée-sortie des tâches, seront donc généralement constituées de deux parties:

1. Une sous-partie *contrôle*, dans laquelle les différentes horloges validant les dépendances de donnée en amont ou aval sont évaluées à partir de leur expression normale.
2. Une sous-partie *donnée* qui dans le cas d'une interface de sortie, sera en charge de diffuser à destination de chacune de ses tâches consommatrices, l'ensemble des signaux produits par la fonction de calcul de la tâche émettrice.

   Dans le cas d'une interface d'entrée, la sous-partie *donnée* aura pour rôle de définir les flots d'entrée de la fonction de calcul en réceptionnant les signaux référencés dans les expressions de définition correspondant à chacune des entrées.

Ces deux parties de l'interface peuvent chacune entraîner des dépendances de données, ainsi que des opérations de mémorisation.

Le passage du squelette synchrone du programme vers une implémentation asynchrone requiert donc un ensemble de transformations formelles du graphe de dépendance de l'application.

En effet, nous avons cherché dans notre approche à minimiser le nombre de signaux nécessaires à l'exécution d'une application. Avec un tel objectif en vue, nous considérons que le nombre minimum et suffisant de signaux circulant dans le réseau devrait se limiter aux seuls flots produits en sortie par les fonctions de calcul, et servant à définir les entrées d'autres tâches.

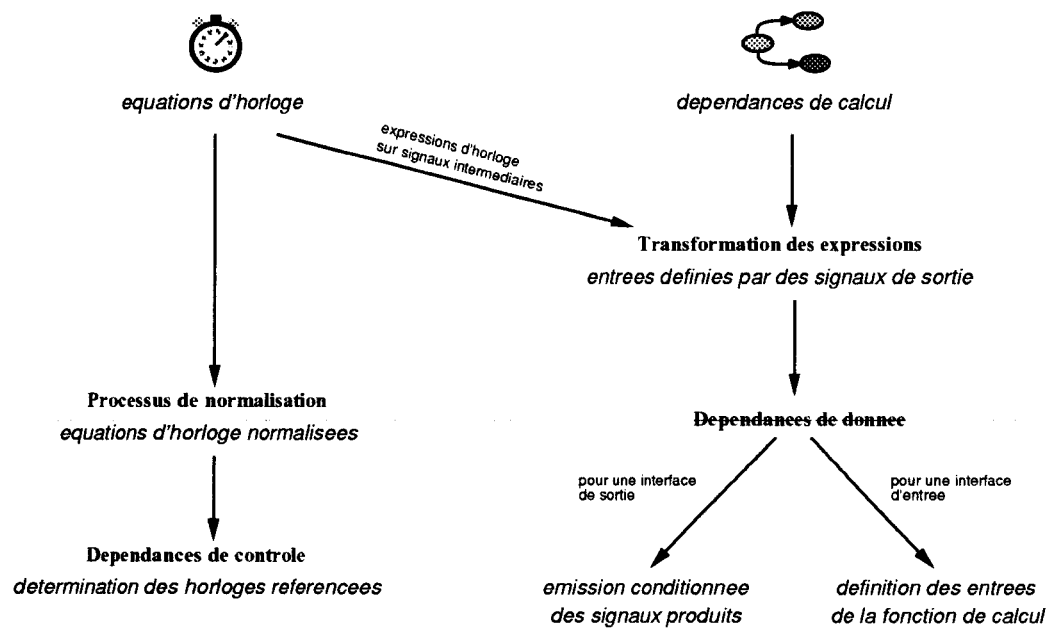


FIG. I.8 - *Des systèmes validés à leur implémentation.*

Nous avons ainsi développé plusieurs processus de ré-écriture (voir figure I.8) qui seront détaillés dans le corps de ce mémoire.

Le premier d'entre-eux, appelé *processus de normalisation* de l'arbre d'horloges vise à transformer toutes les équations récurrentes d'horloge en expressions normales sur signaux de sortie du

programme. L'objectif ainsi visé est de remplacer toute référence aux horloges (horloges et expressions de calcul d'horloges) par des dépendances de données traditionnelles sur signaux de sortie du programme. Il s'agit en d'autres termes de débarrasser le système d'exécution de la gestion contraignante des horloges, en distribuant le contrôle à l'intérieur des interfaces de communication de chaque tâche.

Le second vise à exprimer directement chaque entrée de fonction par des expressions ne référant que des flots produits par d'autres tâches. Ce processus de transformation permet de réduire considérablement le nombre des dépendances de calcul en supprimant les expressions intermédiaires.

Lorsque ces deux processus formels de ré-écriture ont été exécutés, nous sommes en mesure de définir avec précision pour chaque couple de tâches du système quelles sont les dépendances existant, et quelle en est leur fréquence. Ces informations vont définir le contenu des interfaces de communication de chacune des tâches du système.

La phase d'implémentation des interfaces n'est malheureusement pas directe. Il s'agit à partir d'expressions relativement complexes d'en déduire le code qui sera finalement produit dans les interfaces de communication des tâches. Nous nous sommes attelés à résoudre ce problème en proposant des méthodes de complexité raisonnable qui puissent être automatisées au sein d'un compilateur.

Le modèle d'exécution que nous proposons est donc purement dicté par la disponibilité des données. Les dépendances peuvent être à la fois créées pour des besoins de contrôle (expressions normales d'horloge) et pour permettre la définition des entrées d'une fonction de calcul.

Notre schéma d'exécution se confronte donc à un problème traditionnel du dataflow qui est de borner les arcs du graphe. Afin de surmonter cette difficulté, nous avons opté pour un mécanisme d'accusés de réception. Ainsi, dès que toutes les entrées d'une fonction de calcul ont pu être évaluées, et que celle-ci est donc prête à s'exécuter, un message de *"queues vides"* est envoyé aux dépendances amont.

Il faut toutefois noter que les dépendances de données étant contraintes, la fréquence d'émission des tokens d'accusé de réception est également soumise aux horloges de validité des dépendances.

Mise en œuvre

Dans le modèle d'exécution proposé, les tâches communiquent sur leurs entrées et leurs sorties par échanges de messages via une librairie standard de communication (PVM[3] [GBD+94b, GBD+94a] par exemple).

Ces messages sont des signaux référencés dans des expressions de définition d'horloge ou d'entrée de fonction de calcul. Une fois définies, les horloges sont utilisées dans les interfaces de communication des tâches pour contraindre les dépendances de données. Les horloges vont ainsi définir en quelque sorte un ensemble de commandes gardées chargées de valider localement chaque dépendance de donnée.

Considérons par exemple, l'expression suivante:

```
x = y when a
```

où `y`, et `a` sont des signaux émis respectivement par les tâches T_y, et T_a, le signal `a` est un flot de type booléen, et les tâches T_y, et T_a sont supposées synchrones.

Le signal `x` définit l'entrée de la tâche T_x, et la valeur de l'expression `when a` détermine la suite des instants auxquels le signal `y` est utilisé pour définir le signal `x`.

3. PVM est une librairie de communication qui permet à un réseau hétérogène de machines d'être utilisé comme un seul gros calculateur parallèle

Nous avons représenté en figure I.9, l'expression précédente implémentée au moyen de commandes gardées.

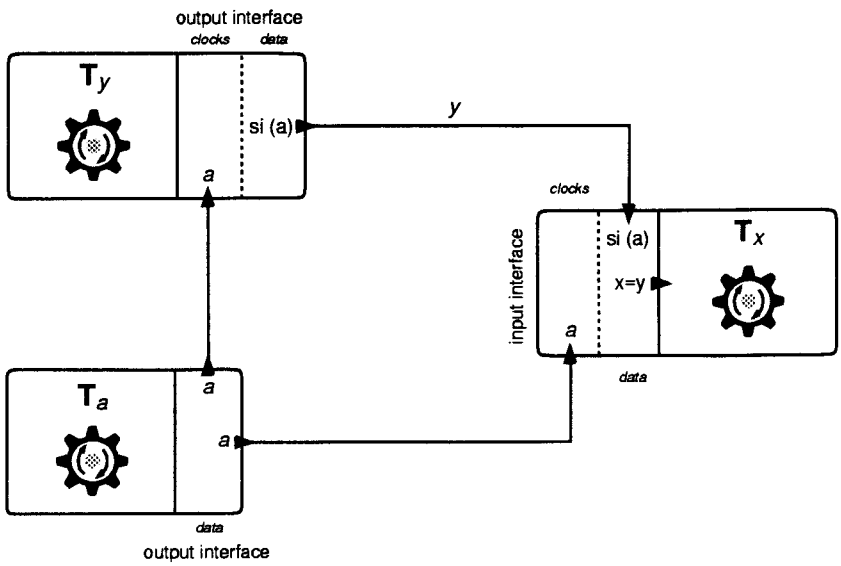


FIG. I.9 - *Implémentation.*

En fin de traitement, la fonction T_y produit le signal **y**. Une dépendance de calcul entre T_y et T_x sur la disponibilité du signal **y** a été établie. Cette dépendance est valide lorsque le flot booléen **a** porte la valeur *vrai*.

Dans sa partie contrôle, l'interface de sortie de T_y est en attente du signal **a** afin de déterminer l'existence de la dépendance entre T_y et T_x pour l'instant courant.

Si la dépendance est validée, le signal **y** est envoyé à T_x afin de définir l'entrée **x** de sa fonction de calcul.

Le rôle des commandes gardées est d'assurer le contrôle temporel de l'application tel qu'il a été synthétisé par le calcul d'horloge. Elles peuvent aussi bien conditionner l'émission que la réception des signaux de données.

En définitive, une fonction de calcul ne peut s'exécuter que lorsque tous ses signaux d'entrée sont présents, et à son achèvement, tous les signaux de sortie ont été produits. Lorsque toutes les entrées d'une tâches sont disponibles, la fonction de calcul associée est exécutée.

L'instanciation des tâches est purement dictée par la disponibilité des données. À la différence des modèles synchrones, aucune interface ou moniteur n'est nécessaire pour contrôler le déroulement du programme: le contrôle de l'application est entièrement réparti entre les tâches par l'intermédiaire de leurs interfaces de communication.

Conclusion

Nous avons validé notre approche en appliquant nos processus de ré-écriture à un ensemble d'exemples théoriques, mais également sur quelques applications concrètes, qui pour partie seront détaillées dans le mémoire.

L'expérimentation de ce travail a été validée sur un réseau hétérogène de machines Unix à l'aide de la librairie de communication PVM (cf. [Gal95]).

Il a bien entendu été impossible de respecter des débits suffisants, ne serait-ce que pour entrevoir la frontière du temps-réel, néanmoins nous avons pu vérifier la cohérence des résultats produits

par les processus de transformation.

I.4.2 Quelques expériences

La confrontation du synchronisme et de l'asynchronisme a déjà été tentée avec le langage asynchrone ELECTRE [RCCE92], mais également sur la base des langages SIGNAL [Maf93, Ché91, LG89] et ESTEREL [Cos91]. Nous évoquons ici ces différentes expérimentations, en soulignant les écueils qu'elles rencontrent.

Le langage asynchrone ELECTRE

Le langage ELECTRE [RCCE92] a été conçu pour exprimer les comportements admissibles des tâches de contrôle d'une application temps-réel. Il s'inscrit donc dans l'ensemble des travaux décrits précédemment, et destinés à proposer une description des comportements événementiels des processus informatiques. ELECTRE se distingue néanmoins des langages précédents par son approche résolument asynchrone au niveau du langage, alors que les langages synchrones reposent quant à eux sur les deux hypothèses fortes de synchronisme: instantanéité et simultanéité des événements.

ELECTRE est un langage comportemental, c'est-à-dire permettant l'expression de l'évolution externe de chacune des tâches de l'application à programmer (démarrage, interruption, reprise, terminaison) en fonction des divers événements susceptibles de les affecter (signaux d'origine matérielle, c'est-à-dire provenant du processus contrôlé, ou signaux d'interaction entre tâches).

Dans tout programme ELECTRE, les tâches sont partitionnées sous la forme de séquences d'instructions ne comportant plus aucun point de synchronisation bloquant, mais pouvant être interrompues par des événements dont, par hypothèse d'asynchronisme, on traitera les occurrences immédiatement. Par conséquent, une occurrence d'événement pouvant survenir entre les instants de début et de fin de traitement: la durée d'exécution ne peut donc pas être considérée comme nulle.

Parmi les opérateurs principaux d'ELECTRE, on trouve la composition séquentielle et parallèle permettant de construire des structures de modules, ainsi que l'opérateur de répétition. La gestion des événements est principalement décrite par des opérateurs de préemption, activation et interruption.

Il est en résumé très difficile et même quasi impossible d'exprimer des relations temporelles complexes en ELECTRE: le concept de simultanéité fait ici cruellement défaut.

La compilation de tout programme ELECTRE produit un système de transitions fini, à la manière de LUSTRE et ESTEREL.

Le langage synchrone ESTEREL

Le besoin de connecter plus étroitement le monde synchrone et asynchrone a déjà été évoqué dans le cadre de la programmation d'applications robotiques [Cos91] développé dans le projet PRISME[4] de l'INRIA[5].

L'approche adoptée dans ce travail comporte certaines similitudes avec la nôtre; il s'agit d'une méthode de programmation au niveau tâche en robotique associant une approche synchrone et une conception orientée objet.

- la modélisation du comportement réactif de la tâche robot est basée sur l'utilisation du langage synchrone ESTEREL,
- les différents composants algorithmiques utilisés dans la synthèse de la loi de commande sont structurés en modèles orientés objets.

4. Programmation des Robots Industriels et des Systèmes Manipulateurs Évolués
5. Institut National de Recherche en Informatique et en Automatique

Cette approche hérite des difficultés posées par le langage synchrone ESTEREL.

- **Parallélisme explicite.** L'instruction **PAR** qui permet d'exécuter en parallèle différentes tâches, correspond en fait à l'opérateur "||" d'ESTEREL,
- **Synchronisation explicite.** Le comportement temporel des programmes est spécifié par des points de synchronisation mettant en œuvre des mécanismes de type "rendez-vous" ou à l'aide de sémaphores[6].

Finalement, l'exploitation du parallélisme est à la charge de l'utilisateur, et il est impossible d'exprimer simplement des relations temporelles complexes.

Implantation de programmes SIGNAL sur multiprocesseur

On trouvera dans [LG89], un ensemble d'outils permettant à un programme SIGNAL compilé d'être structuré en un ensemble de processus séquentiels communicants.

Une partition du GDC est recherchée afin que l'ordonnancement de chaque partie (séquentialisation) puisse être calculé statiquement.

La phase de distribution réalise ainsi l'assignation d'un ensemble de processus à chaque processeur, en requérant éventuellement l'aide de l'utilisateur. Cette répartition statique est figée sur la durée d'exécution du programme.

L'implémentation se compose de deux étapes. La première définit localement une implémentation maximalement séquentielle des processus distribués; la seconde étape infère, à partir de ces implémentations localement définies, un processus de coopération.

En d'autres termes, le problème abordé dans [LG89] est de calculer une partition du GDC telle que chaque sous-graphe puisse être séquentialisé:

"Un GDC est associé à tout programme SIGNAL. Répartir un programme P, *c'est donc partitionner son GDC associé* G *en sous-graphes* $G_1, \ldots, G_n$ *(autant de processeurs) de sorte que l'on ait* $P = P_1 \mid \ldots \mid P_n$, *si* $P_1, \ldots, P_n$ *sont les programmes qu'il est possible d'associer aux sous-graphes* $G_1, \ldots, G_n$ *respectivement"* [LG89, p.105].

Nous avons traité le problème sous un angle différent en rejetant sur le système d'exécution les problèmes liés à l'ordonnancement. Pour celà nous déterminons pour chaque tâche du programme la partie de code nécessaire pour en permettre l'exécution contrôlée. Il serait difficile de qualifier notre approche par rapport à celle existante, disons du moins que cette voie méritait d'être explorée.

Nous pouvons également citer dans cette partie, le système SYNDEX [Sor94, Sor96] d'aide à la répartition et de placement d'une application SIGNAL sur une machine multiprocesseur. Cet outil est à mettre en relation avec les travaux menés dans [LG89, Maf93].

Otto e Mezzo

De part la faiblesse de ses opérateurs temporels, le modèle proposé dans le cadre d'OTTO E MEZZO ne permet pas de résoudre les problèmes complexes que nous abordons.

[6] Un *sémaphore* est un objet dont la représentation concrète est constituée d'un entier sur lequel peuvent opérer deux primitives appelées *P* (Prendre) et *V* (Vider) pour demander ou libérer une ressource. Un sémaphore indique l'état (libre ou occupé) de la ressource à laquelle il est associé

Chapitre II

Normalisation des horloges

À tout signal est implicitement associée une horloge qui en définit précisément les instants de validité. Lors de la spécification des systèmes, les seules dépendances exprimées sont les dépendances de calcul. C'est la phase de résolution des contraintes temporelles (calcul d'horloges) qui rend la manipulation d'horloges explicite pour le système d'exécution. De nouvelles dépendances (les dépendances d'horloges) sont ainsi créées, elles étiquettent chaque dépendance de données par ses instants de validité. Le processus de normalisation des horloges décrit dans ce chapitre, vise à débarrasser le système d'exécution de la gestion contraignante de ces horloges dans un contexte CSP.

Dans une première partie (SECTION II.1) nous détaillons le contexte formel issu du calcul d'horloges en SIGNAL, et nous concluons en introduisant notre processus de normalisation. Dans la section suivante (SECTION II.2), nous illustrons sur une série de transformations, les différentes stratégies de substitution qui pourraient être mises en œuvre, en expliquant dans quelles mesures elles échouent à résoudre notre problème. En relation avec cette partie, nous présentons dans les deux sections suivantes: (II.3 et II.4), les solutions que nous avons mises en place (voir également [GDM96c]). Nous concluons notre discussion en SECTION II.5, et nous donnons un exemple d'application en SECTION II.6 qui illustre un certain nombre de simplifications possibles.

II.1 Calcul d'horloges de SIGNAL

Tout comme en LUSTRE, un programme SIGNAL est structuré sous la forme d'un ensemble de définitions de variables, de type:

$$\mathbf{x}_i = \mathbf{E}_i$$

où la variable $\mathbf{x}_i$ et l'expression $\mathbf{E}_i$ ont même suite de valeurs, et même horloge.

Cette structuration des programmes en LUSTRE et SIGNAL induit deux grands principes:

- **le principe de substitution.** $\mathbf{x}_i$ peut être substitué à $\mathbf{E}_i$, et inversement, partout dans le programme,
- **le principe de définition.** une variable est complètement définie par sa déclaration et l'équation où elle apparaît en membre gauche.

Suivant ces deux principes, un programme s'écrit comme une définition mathématique: l'ordre des équations est indifférent, et l'introduction de variables intermédiaires pour nommer ou décomposer des expressions complexes est sans conséquence aucune.

La phase de compilation repose sur un processus formel de résolution des relations temporelles qui a été rendu particulièrement sophistiqué en SIGNAL, grâce au formalisme mathématique intrinsèque du langage.

Les différents environnements utilisés pour la compilation de programmes SIGNAL sont les environnements de signaux et les environnements d'horloges. On désignera ainsi par:

- $\mathcal{SIG}$, le sous-ensemble fini et dénombrable des signaux du programme,
- $\mathcal{IDH}$, le sous-ensemble fini et dénombrable des identificateurs d'horloge du programme.

Dans sa phase de résolution, le calcul d'horloge de SIGNAL synthétise sous la forme d'un arbre la hiérarchie des horloges du programme, et sous la forme d'un système d'équations les expressions de définition de ces horloges.

Nous présentons ici de manière formelle l'environnement d'horloges résultant du processus de résolution des contraintes temporelles en SIGNAL.

II.1.1 Le système d'équations d'horloges

Chaque horloge définit un temps logique permettant de parler des différentes relations temporelles existant entre cette horloge et les autres horloges du programme. Ceci par l'intermédiaire des valeurs d'absence (notation: $\perp$) et de présence (notation: $\top$) qu'elle peut prendre à tout instant. Les moments où deux instants coïncident, c'est-à-dire la présence simultanée de deux horloges, sont tous connus (voir le concept de *simultanéité*). Le système d'équations d'horloges décrit statiquement l'ensemble des relations temporelles existant entre les différentes horloges du programme.

Une horloge dans le système d'équations peut être définie:

- par extraction directe de valeur,
- récursivement par rapport à d'autres d'horloges.

Il s'agit dans le premier cas d'horloges définies par extraction de valeurs d'un signal booléen. Par exemple, l'expression:

$$h_1 \texttt{ = when b}$$

décrit la relation temporelle suivante,

b	*vrai*	*faux*	*vrai*	...
h_1	$\top$	$\perp$	$\top$	...

L'horloge h_1 est ainsi définie par la suite des instants auxquels le signal booléen b porte la valeur *vrai*. En particulier, lorsque le signal b n'est pas défini, l'horloge h_1 ne l'est pas non plus.

Dans le second cas, les horloges sont définies les unes par rapport aux autres. Par exemple, l'expression:

$$\texttt{x = y}_1 \texttt{ default y}_2$$

définit le système d'équations suivant, où h_x, h_{y_1}, et h_{y_2} sont les horloges respectives des signaux x, $\texttt{y}_1$, et $\texttt{y}_2$, et h' est une horloge intermédiaire:

$$\begin{cases} h_x &= h_{y_1} \vee h' \\ h' &= h_{y_2} \wedge \overline{h_{y_1}} \end{cases}$$

Dans cet exemple (il s'agit d'une opération d'entrelacement de signaux), la phase de résolution se traduit par la décomposition des horloges des arguments et la création d'une expression d'horloge intermédiaire.

Cette décomposition découle directement de l'ordre de priorité placé sur les entrées de l'opérateur d'entrelacement de signaux, afin de garantir sémantiquement son déterminisme. Ainsi, la dépendance entre $\mathbf{x}$ et le couple de signaux $\mathbf{y}_1$, $\mathbf{y}_2$, est remplacée dans le système d'équations d'horloges par une dépendance entre $\mathbf{x}$ et $\mathbf{y}_1$ lorsque le signal $\mathbf{y}_1$ est défini ($\equiv\ h_{y_1} = \top$), et entre $\mathbf{x}$ et $\mathbf{y}_2$ lorsque le signal $\mathbf{y}_2$ est défini ($\equiv\ h_{y_2} = \top$), et que $\mathbf{y}_1$ ne l'est pas ($\equiv\ h_{y_1} = \bot$). Par conséquent, le calcul d'horloges garantit qu'à un instant donné, une seule entrée de l'opérateur peut etre valide. On en déduit la relation temporelle suivante:

h_{y_1}	$\top$	$\bot$	$\top$	...
h_{y_2}	$\top$	$\top$	$\bot$	...
h'	$\bot$	$\top$	$\bot$	...
h_x	$\top$	$\top$	$\top$	...
$\mathbf{x}$	$\mathbf{y}_1$	$\mathbf{y}_2$	$\mathbf{y}_1$	...

En résumé, une horloge peut donc être soit directement définie à partir d'une expression booléenne, on parlera de *contrainte conditionnelle* ou encore *expression contrainte*, soit par composition d'autres horloges, on parlera dans ce cas de *formule* ou *expression non-contrainte*.

Une horloge définie par une expression contrainte sera appelée variable contrainte, et respectivement, une horloge définie par une expression non contrainte sera appelée variable non contrainte.

L'ensemble $\mathcal{IDH}$ des identificateurs d'horloge d'un programme est donc scindé en deux sous-ensembles:

- $\mathcal{VHC}$ qui dénote l'ensemble des variables contraintes de $\mathcal{IDH}$, c'est-à-dire construites directement sur des signaux booléens du programme,
- $\mathcal{VH}$ qui dénote l'ensemble des variables non contraintes de $\mathcal{IDH}$, c'est-à-dire construites récursivement sur d'autres horloges.

II.1.2 Expressions non contraintes

Une expression sur $\mathcal{IDH}$ admet les opérateurs du treillis [Bes92]:

- $h_1 \vee h_2$ est la borne supérieure de h_1 et h_2, cette expression dénote la présence alternative (mais non exclusive) de deux signaux,
- $h_1 \wedge h_2$ est la borne inférieure de h_1 et h_2, elle dénote la présence simultanée de deux signaux,
- l'opérateur de négation $\overline{h_1}$.

L'horloge sans événement est notée $\emptyset$.

Une expression non contrainte, ou formule (notation $\mathcal{F}$), est une expression construite sur le treillis et comporte au plus un opérateur binaire: c'est un mot du langage engendré par la grammaire suivante:

$$\mathcal{F} ::= \mathcal{IDH} \vee \mathcal{IDH} \ ||\ \mathcal{IDH} \wedge \mathcal{IDH} \ ||\ \mathcal{IDH} \wedge \overline{\mathcal{IDH}} \ ||\ \overline{\mathcal{IDH}} \wedge \mathcal{IDH}$$

On dira qu'une horloge h appartient à l'ensemble des variables non contraintes ($h \in \mathcal{VH}$), si l'expression de définition de h contient au moins un argument qui soit lui-même non contraint.

En d'autres termes, une variable non contrainte est une horloge non complètement spécifiée par une expression booléenne sur signaux.

II.1.3 Expressions contraintes

Une variable contrainte est définie par une expression booléenne sur signaux du programme. Formellement, l'expression booléenne **E** est une fonction construite sur un ensemble de signaux synchrones, dont h_E dénote l'horloge des arguments et du résultat.

L'expression `E := a AND b`, par exemple, où **a** et **b** sont des signaux booléens synchrones, définit la relation temporelle suivante:

$$h_E = h_a = h_b$$

où h_a et h_b sont les horloges respectives des signaux **a** et **b**.

Une telle expression engendre implicitement deux nouvelles horloges, qui sont notées conventionnellement: `[E]` et $[\overline{E}]$, et qui dénotent respectivement l'ensemble des instants de h_E où la condition **E** est vérifiée et l'ensemble des instants de h_E où elle n'est pas vérifiée.

Les horloges, `[E]` et $[\overline{E}] \in \mathcal{VHC}$, vérifient les contraintes suivantes:

- $[E] \vee [\overline{E}] = h_E$
- $[E] \wedge [\overline{E}] = \emptyset$

En fait, les variables contraintes `[E]` et $[\overline{E}]$ expriment le sous-échantillonnage de l'horloge h_E par la condition **E**.

Ainsi, par exemple, si on dénote par h_a, et h_b les horloges respectives des signaux **a**, et **b** dans l'expression suivante:

```
a := when b
```

L'horloge h_a associée au signal **a** est définie par l'expression d'horloge:

$$h_a = h_b \wedge (\ b = \texttt{vrai}\ )$$

Autrement dit, h_a dénote l'ensemble des instants auxquels le signal **b** est défini ($\equiv h_b = \top$), et porte la valeur *vrai* ($\equiv b = vrai$). On notera de manière conventionnelle:

$$h_a = [b]$$

En d'autres termes, h_a dénote l'ensemble des instants de h_b où la condition **b** est vérifiée.

Respectivement,

$$\overline{h_a} = [\overline{b}]$$

$\overline{h_a}$ dénote l'ensemble des instants de h_b où la condition **b** n'est pas vérifiée.

II.1.4 Arbre d'horloges

La résolution des contraintes temporelles en SIGNAL passe par la création d'un arbre dans lequel les différentes horloges du programme sont hiérarchisées. Il existe ainsi un ordre partiel effectif entre ces horloges.

Une classification des processus SIGNAL peut être dès lors réalisée: la première classe de processus contient ceux dont le contrôle peut être complètement hiérarchisé à partir d'une horloge maîtresse, appelée mère —la seconde classe contient les processus pour lesquels une telle hiérarchie n'a pu être synthétisée.

Il a été prouvé (c.f. [LG89]) que le contrôle d'un processus de la première classe pouvait être effectué par un automate d'état fini déterministe, ou mieux une hiérarchie de petits automates

d'état fini communiquant entre eux. Les seules informations exigées de l'extérieur sont alors les stimuli de l'horloge mère et bien sûr les valeurs des signaux d'entrée.

En revanche, l'obtention d'un schéma d'exécution déterministe d'un processus de la seconde classe nécessite plus d'information de l'extérieur. Et dans ce cas, un schéma d'exécution pipeliné déterministe et cohérent exigerait l'estampillage à la LAMPORT [Lam78] de toute valeur d'un signal par l'instant auquel il appartient.

Nous ne nous intéresserons donc qu'aux processus de la première classe, également appelés processus *endochrones*, c'est-à-dire ceux dont le contrôle a pu être hiérarchisé dans un seul arbre. La racine de cet arbre, la plus rapide de toutes, est caractérisée par les propriétés suivantes:

- L'horloge racine est l'élément dominant pour l'ordre partiel effectif de l'ensemble des formules placées dans l'arbre.
- L'horloge racine est l'horloge des signaux nécessaires à la définition des sous-échantillonnages par conditions du niveau immédiatement inférieur.

Par conséquent, toutes les horloges sont construites sur des conditions de la racine.

On a représenté en figure II.1, un arbre hypothétique d'horloges dans lequel:

- h_r est l'horloge racine de l'arbre
- $h_a = h_b = h_r$
- $h_c = h_d = h_2$
- $h_1, h_2, h_4, h_5, h_6 \in \mathcal{VHC}$
- $h_3 \in \mathcal{VH}$

où h_a, h_b, h_c, h_d dénote l'horloge de définition respective des signaux **a**, **b**, **c**, et **d**.

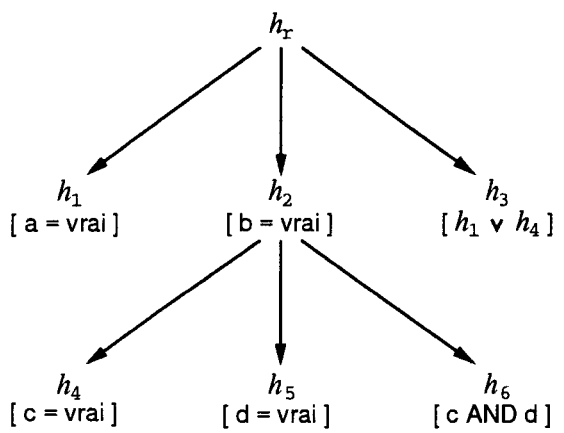


FIG. II.1 - *Arbre d'horloges. Les signaux* **a** *et* **b** *sont émis à l'horloge* $\mathbf{h}_r$. *Les signaux* **c** *et* **d** *sont émis à l'horloge* $\mathbf{h}_2$.

La construction de l'arbre d'horloges durant la phase de compilation d'un programme SIGNAL est progressive, nous distinguerons dans la suite, l'opération de placement d'une variable contrainte, de celle d'une variable non contrainte.

Cas des variables contraintes

Le placement d'une horloge $h \in \mathcal{VHC}$ dans l'arbre dépend des conditions référencées dans son expression de définition: l'horloge h est insérée en tant que dernière fille du nœud correspondant

à l'horloge de son expression de définition.

Par exemple, dans l'arbre hypothétique de la figure II.1, l'horloge h_4 est définie comme extraction du signal booléen c émis à l'horloge h_2. Par conséquent, la variable h_4 est placée sous le nœud définissant h_2.

De la même manière, l'horloge respective des expressions de définition de h_1, et h_2 étant h_r, ces variables sont placées directement sous la racine de l'arbre. En revanche, d étant émis à l'horloge h_2, la variable h_5 est placée sous le nœud h_2.

Cas des variables non contraintes

L'insertion d'une variable non contrainte dans l'arbre (ses opérandes ayant déjà été placés) se fait sous l'horloge la plus grossière contenant l'horloge de ses opérandes. En d'autres termes, une variable non contrainte est placée sous la racine du sous-arbre contenant l'ensemble des horloges référencées dans son expression de définition.

Par exemple, dans l'arbre de la figure II.1, la variable non contrainte h_3 est définie par une expression dont les arguments h_1 et h_4 ont pour nœud commun d'horloge h_r. L'horloge h_3 est donc placée directement sous la racine de l'arbre.

II.1.5 La règle d'inclusion des événements

Par construction, les variables (contraintes ou non) d'un programme sont placées dans l'arbre par inclusion sous l'horloge immédiatement plus rapide. L'inclusion d'événements peut être ainsi interprétée sous la forme d'ensembles (c.f. figure II.2).

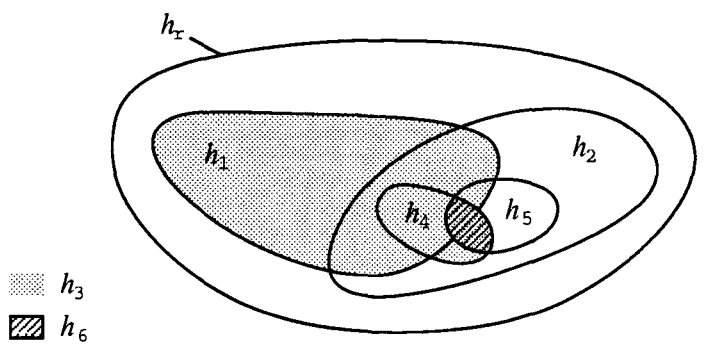


FIG. II.2 - *Graphe d'inclusion des événements.*

A partir du graphe d'inclusion des événements donné en figure II.2, il est plus facile d'appréhender la stratégie de placement des horloges dans l'arbre. On peut ainsi constater intuitivement que les horloges sont placées dans l'ensemble minimal englobant les différents arguments concourant à la définition d'une expression.

Ainsi, par exemple, c'est l'ensemble h_r qui englobe le plus directement les arguments de l'expression de définition de h_3, c'est-à-dire h_1 et h_4. De la même manière, les horloges h_4 et h_5 définissant h_6 sont incluses dans l'ensemble h_2.

II.1.6 Introduction au processus de normalisation

L'ensemble $\mathcal{IDH}$ des horloges d'un programme se compose d'un sous-ensemble $\mathcal{VHC}$ de variables contraintes, et d'un sous-ensemble $\mathcal{VH}$ de variables non contraintes.

Ces sous-ensembles traduisent des dépendances de nature différente: signaux-vers-horloges pour les éléments de $\mathcal{VHC}$, et horloges-vers-horloges pour ceux de $\mathcal{VH}$. Autrement dit, les expressions de définition des variables contraintes sont construites sur les signaux du programme, alors que les variables non contraintes sont composées récursivement à partir d'autres horloges.

Les règles de construction de l'arbre d'horloges en SIGNAL font que finalement toutes les horloges d'un programme sont définies par des règles de composition successives des horloges sur des conditions de la racine.

Le processus de normalisation de l'arbre d'horloges est destiné à remplacer toutes les dépendances de type horloges-vers-horloges par de nouvelles dépendances signaux-vers-horloges.

Une horloge est un terme booléen qui à un instant donné porte la valeur *vrai* lorsqu'elle est définie, et la valeur *faux* dans le cas contraire. Le processus de normalisation s'appuiera donc sur les axiomes du calcul booléen [Wol91] afin de garantir l'équivalence sémantique entre l'arbre d'horloges normalisé et l'arbre originel.

La ré-écriture de fonctions d'interprétation est un processus complexe qui nécessite des heuristiques proposant à la fois des solutions et une démarche qui soient "*raisonnables*". L'expression "*raisonnable*", doit être entendue en termes de complexité du processus de transformation, et d'interprétation des fonctions logiques obtenues. Une démarche formelle est proposée dans cette partie, elle illustre les difficultés posées par la transformation des fonctions d'interprétation ainsi que les solutions apportées.

II.2 La problématique sur un exemple

L'objet de cette approche préliminaire est de donner au lecteur un aperçu du processus de normalisation, ainsi qu'une idée intuitive des problèmes qui doivent être résolus pour obtenir un résultat facilement exploitable, moyennant une méthode de résolution de complexité "*raisonnable*".

Nous avons vu dans la section précédente que le calcul d'horloges produit:

- une hiérarchie d'horloges basée sur la règle d'inclusion d'événements,
- un système d'équations d'horloges donnant à chaque variable de $\mathcal{IDH}$ son expression de définition.

Toutes ces informations sont synthétisées dans l'arbre des horloges.

L'arbre hypothétique d'horloges donné en figure II.3 servira de base à notre analyse.

- h_r est l'horloge racine de l'arbre
- $h_2, h_4, h_5, h_6, h_7 \in \mathcal{VHC}$
- $h_1, h_3, h_8 \in \mathcal{VH}$

Les booléens `a`, `b`, `c`, `d` et `e` sont des signaux booléens du programme,

- le signal `a` est émis à l'horloge h_r
- les signaux `b` et `c` sont émis à l'horloge h_2
- les signaux `d` et `e` sont émis à l'horloge h_4

Le système d'équations correspondant à l'arbre de la figure II.3 est donné ci-après:

$$\begin{aligned}
h_1 &= h_r \wedge \overline{h_3} \\
h_2 &= [a] \\
h_3 &= h_5 \wedge \overline{h_8} \\
h_4 &= [b] \\
h_5 &= [c] \\
h_6 &= [d] \\
h_7 &= [e] \\
h_8 &= h_6 \vee h_7
\end{aligned}$$

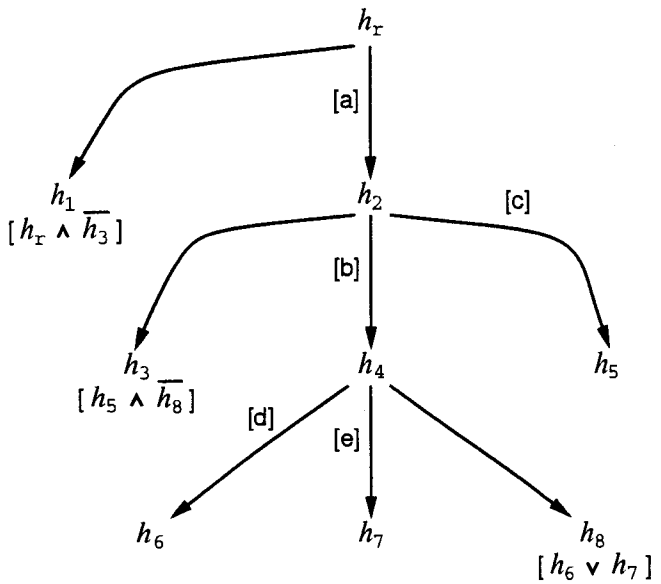


FIG. II.3 - *Arbre d'horloges*

Chaque variable (h_i, $i \in \{ 1, \ldots, 8 \}$) dans le système est définie par une expression unique dans laquelle elle apparaît en membre gauche (voir le principe de définition p.39). Les variables contraintes (h_2, h_4, h_5, h_6, et h_7) sont évaluées par des expressions booléennes. Les variables non contraintes (h_1, h_3, et h_8) sont définies récursivement à partir d'autres horloges dans l'ensemble $\mathcal{IDH}$.

$\mathcal{VH}$	$\mathcal{VHC}$
$h_1 = h_r \wedge \overline{h_3}$	$h_2 = [a]$
$h_3 = h_5 \wedge \overline{h_8}$	$h_4 = [b]$
$h_8 = h_6 \vee h_7$	$h_5 = [c]$
	$h_6 = [d]$
	$h_7 = [e]$

Le processus de normalisation des horloges vise à aboutir à un nouveau système dans lequel toutes les variables sont définies par des expressions contraintes. En d'autres termes, dans le système final, les identificateurs d'horloges (h_i's) ne devraient plus figurer qu'en membre gauche des équations.

La manière intuitive d'aboutir à ce résultat est d'opérer par substitutions successives en dérivant de nouveaux systèmes d'équations (voir le principe de substitution p.39). En pratique malheureusement, la démarche est loin d'être aussi immédiate. Nous décrivons par la suite les différentes étapes de notre raisonnement, en faisant la lumière sur les problèmes rencontrés et les solutions proposées.

II.2.1 Approche directe

On peut dans une première approche, chercher à résoudre directement le système à partir des expressions de définition données dans le système initial d'équations d'horloges. Le processus de

substitution dériverait ainsi successivement:

$\mathcal{VH}$	$\mathcal{VHC}$		$\mathcal{VH}$	$\mathcal{VHC}$	
$h_1 = h_r \wedge \overline{h_3}$ $h_3 = h_5 \wedge \overline{h_8}$ $h_8 = h_6 \vee h_7$	$h_2 = [a]$ $h_4 = [b]$ $h_5 = [c]$ $h_6 = [d]$ $h_7 = [e]$	$\equiv$	$h_1 = h_r \wedge \overline{h_3}$ $h_3 = [c] \wedge \overline{h_8}$	$h_2 = [a]$ $h_4 = [b]$ $h_5 = [c]$ $h_6 = [d]$ $h_7 = [e]$ $h_8 = [d] \vee [e]$	$\equiv$

L'horloge h_8 est donc valide lorsque le signal **d** ou **e** porte la valeur *vrai*.

$\mathcal{VH}$	$\mathcal{VHC}$
$h_1 = h_r \wedge \overline{h_3}$	$h_2 = [a]$ $h_4 = [b]$ $h_5 = [c]$ $h_6 = [d]$ $h_7 = [e]$ $h_8 = [d] \vee [e]$ $h_3 = [c] \wedge (\overline{[d] \vee [e]})$

Par contre, l'expression contrainte de h_3, serait:

$$h_3 = [c] \wedge (\overline{[d] \vee [e]}) = [c] \wedge [\overline{d}] \wedge [\overline{e}]$$

En termes de commandes gardées, cela signifie qu'émettre un signal à l'horloge h_3, revient à attendre les signaux **c**, **d** et **e** pour évaluer l'expression de h_3 et valider ou non la dépendance. Seulement, le signal **c** et le couple **d** et **e** de signaux ne sont pas émis à la même horloge. En particulier, si à un instant donné le signal **b**, qui n'apparaît pas dans l'expression de définition de h_3, porte la valeur *faux*, ou n'est pas défini, alors le couple **d**, **e** de signaux n'est pas émis. Par voie de conséquence, l'interface de communication bâtie sur une telle expression de h_3 risque de se trouver définitivement bloquée en attente de signaux qui n'arriveront jamais.

En fait, l'inexactitude de ce résultat provient du fait que les inclusions d'événements représentées dans l'arbre d'horloges ne sont pas exprimées dans le système d'équations initial.

Maintenir l'équivalence des systèmes d'équations entre deux substitutions, suppose que les expressions soient évaluées dans le même repère temporel.

II.2.2 Résolution à l'horloge racine

Une réponse au problème précédent pourrait être d'exprimer toutes les équations contraintes du système initial dans un repère de temps unique. Dans la mesure où ne nous intéressons qu'aux processus *endochrones*, nous disposons naturellement d'une référence temporelle commune: la racine de l'arbre d'horloges.

Cette horloge, la plus rapide de toutes, est échantillonnée par des expressions booléennes produisant des horloges moins rapides pouvant être à leur tour échantillonnées.

Par exemple, l'horloge h_4 est un échantillonnage de l'horloge h_2, plus rapide, par l'expression booléenne `b` = *vrai*. Ceci traduit la relation temporelle suivante:

h_2	$\top$	$\top$	...
`b`	*vrai*	*faux*	...
h_4	$\top$	$\bot$	...

L'horloge h_2 résulte elle-même de l'échantillonnage de l'horloge racine h_r, par l'expression booléenne `a` = *vrai*. Ce qui se traduirait en termes de relations temporelles par:

h_r	$\top$	$\top$	$\top$	$\top$	...
`a`	*vrai*	*faux*	*vrai*	*faux*	...
h_2	$\top$	$\bot$	$\top$	$\bot$	...
`b`	*vrai*	$\bot$	*faux*	$\bot$	...
h_4	$\top$	$\bot$	$\bot$	$\bot$	...

On peut ainsi définir une expression contrainte de l'horloge h_4, à l'horloge racine, comme une conjonction des conditions `[a]` et `[b]`,

$$h_4 = [a] \wedge [b]$$

On notera par ailleurs que le signal `b` n'est défini que lorsque le signal `a` porte la valeur *vrai*. Dans le cas contraire, `b` n'est pas émis. Cette remarque souligne le fait qu'un ordre partiel est maintenu entre les signaux dans l'expression de définition de h_4: le signal `a` doit être évalué avant `b`.

De manière générale, toute variable contrainte du système d'équations d'horloges peut être exprimée à l'horloge mère par le produit des contraintes se trouvant sur le chemin de son nœud à la racine.

Le système initial peut être ainsi donné sous la forme suivante,

$\mathcal{VH}$	$\mathcal{VHC}$
$h_1 = h_r \wedge \overline{h_3}$	$h_2 = [a]$
$h_3 = h_5 \wedge \overline{h_8}$	$h_4 = [a] \wedge [b]$
$h_8 = h_6 \vee h_7$	$h_5 = [a] \wedge [c]$
	$h_6 = [a] \wedge [b] \wedge [d]$
	$h_7 = [a] \wedge [b] \wedge [e]$

Appliqué au système précédent, le processus de substitution donne:

$\mathcal{VH}$	$\mathcal{VHC}$
$h_1 = h_r \wedge \overline{h_3}$	$h_2 = [a]$
$h_3 = h_5 \wedge \overline{h_8}$	$h_4 = [a] \wedge [b]$
	$h_5 = [a] \wedge [c]$
	$h_6 = [a] \wedge [b] \wedge [d]$
	$h_7 = [a] \wedge [b] \wedge [e]$
	$h_8 = ([a] \wedge [b] \wedge [d]) \vee ([a] \wedge [b] \wedge [e])$

$\mathcal{VH}$	$\mathcal{VHC}$
$h_1 = h_r \wedge \overline{h_3}$	$h_2 = [a]$ $h_4 = [a] \wedge [b]$ $h_5 = [a] \wedge [c]$ $h_6 = [a] \wedge [b] \wedge [d]$ $h_7 = [a] \wedge [b] \wedge [e]$ $h_8 = ([a] \wedge [b] \wedge [d]) \vee ([a] \wedge [b] \wedge [e])$ $h_3 = ([a] \wedge [c]) \wedge \overline{(([a] \wedge [b] \wedge [d]) \vee ([a] \wedge [b] \wedge [e]))}$

$\equiv$

$\mathcal{VH}$	$\mathcal{VHC}$
$\emptyset$	$h_2 = [a]$ $h_4 = [a] \wedge [b]$ $h_5 = [a] \wedge [c]$ $h_6 = [a] \wedge [b] \wedge [d]$ $h_7 = [a] \wedge [b] \wedge [e]$ $h_8 = ([a] \wedge [b] \wedge [d]) \vee ([a] \wedge [b] \wedge [e])$ $h_3 = \overline{([a] \wedge [c]) \wedge \overline{(([a] \wedge [b] \wedge [d]) \vee ([a] \wedge [b] \wedge [e]))}}$ $h_1 = [([a] \wedge [c]) \wedge \overline{(([a] \wedge [b] \wedge [d]) \vee ([a] \wedge [b] \wedge [e]))}]$

Le processus de substitution à l'horloge mère pose deux problèmes essentiels: la complexité des expressions manipulées croit très rapidement — en outre, le résultat obtenu est loin d'être directement exploitable.

En fait, par construction, il s'avère que l'horloge mère est la racine d'une forêt de sous-arbres définissant autant de sous-repères de temps. Dans notre arbre hypothétique d'horloges (c.f. figure II.3), l'horloge h_4, par exemple, est un repère de temps pour toutes les horloges inférieures contenues dans le sous-arbre de racine h_4.

Par voie de conséquence, lorsqu'une substitution s'opère dans un sous-arbre, la racine de ce sous-arbre constitue un temps de référence pour cette opération, sans qu'il soit besoin d'exprimer systématiquement tous les calculs à l'horloge mère. Nous étudions cette propriété dans la section suivante.

II.3 Expression à multiples fréquences

L'arbre d'horloges définit une hiérarchie d'horloges basée sur l'inclusion d'événements. On peut ainsi déduire pour une meme horloge $h \in \mathcal{VHC}$, une expression contrainte à différentes fréquences. Par exemple, on peut donner une expression de l'horloge h_7 aux fréquences respectives h_4, h_2 et h_r:

$$
\begin{aligned}
h_7 &\xrightarrow{h_4} [e] \\
h_7 &\xrightarrow{h_2} [b] \wedge [e] \\
h_7 &\xrightarrow{h_r} [a] \wedge [b] \wedge [e]
\end{aligned}
$$

La première équation donne une expression de h_7 à h_4, la seconde à h_2 et la dernière à h_r.

II.3.1 Généralisation

De manière générale, on a représenté en figure II.4 une inclusion multiple d'horloges:

$$h_r = h_0 \supseteq h_1 \supseteq \cdots \supseteq h_i \supseteq h_{i+1} \supseteq \cdots \supseteq h_n$$

Avec $h_{i+1} = \texttt{[E}_i\texttt{]}$, où $\texttt{E}_i$ est une expression booléenne sur signaux synchrones d'horloge h_i. C'est-à-dire que h_{i+1} dénote l'ensemble des instants de h_i pour lesquels la condition $\texttt{E}_i$ est vérifiée.

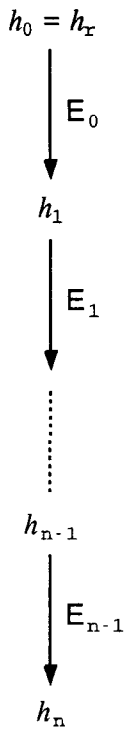


FIG. II.4 - *Inclusion multiple d'horloges*

L'expression contrainte de h_j à l'horloge h_i, $\texttt{0} \leqslant \texttt{i} < \texttt{j} \leqslant \texttt{n}$ est donnée par:

$$h_j \xrightarrow{h_i} \bigwedge_{k=i}^{j-1} [E_k]$$

La décomposition en multiples fréquences des expressions contraintes va nous permettre de simplifier grandement le processus de normalisation des formules en effectuant le calcul à une horloge "*plus fine*" que celle de la racine.

II.3.2 Le processus de substitution à multiples fréquences

Par construction, le placement d'une formule dans l'arbre d'horloges est effectué sous le nœud correspondant à la borne supérieure de ses arguments. Par exemple, dans l'arbre de la figure II.3, l'horloge h_3 est placée sous le nœud h_2. En fait, h_3 est simplement placée sous la racine du premier sous-arbre commun à h_5 et h_8.

Si on généralise, dans le schéma de la figure II.5, on a représenté un couple d'horloges h_m et h_n dont la racine du premier sous-arbre commun est dénotée h_l. Une formule construite sur h_m et h_n sera donc placée directement sous h_l.

Pour obtenir une expression correcte de la formule construite sur h_m et h_n, il est suffisant de travailler à l'horloge du premier sous-arbre commun, en l'occurrence h_l. La définition contrainte de la formule ainsi obtenue est exprimée à l'horloge h_l.

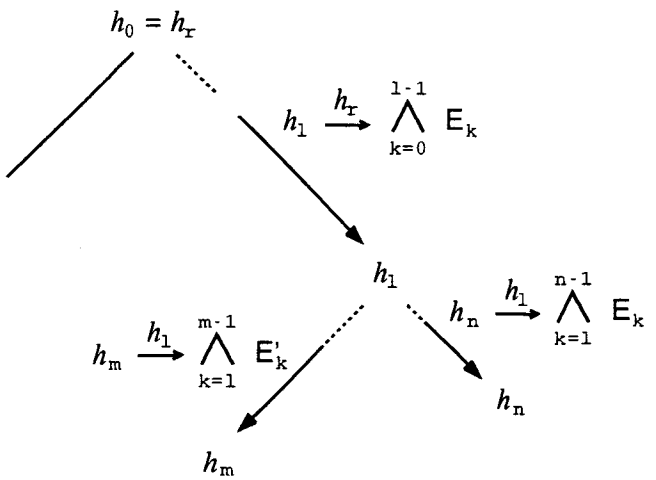


FIG. II.5 - *Expressions à fréquences multiples*

En effet, une expression d'horloge, ou formule (notation $\mathcal{F}$), est une expression construite sur le treillis et comporte au plus un opérateur binaire: c'est un mot du langage engendré par la grammaire suivante [Bes92]:

$$\mathcal{F} := \mathcal{IDH} \vee \mathcal{IDH} \;||\; \mathcal{IDH} \wedge \mathcal{IDH} \;||\; \mathcal{IDH} \wedge \overline{\mathcal{IDH}} \;||\; \overline{\mathcal{IDH}} \wedge \mathcal{IDH}$$

Si on étudie séparément chacun des quatre mots pouvant être engendrée par la grammaire:

1. $\mathcal{F} := \mathcal{IDH} \vee \mathcal{IDH}$

La normalisation de la formule $\mathcal{F} = h_m \vee h_n$ à l'horloge mère h_r, donnerait:

$$\begin{cases} h_m \xrightarrow{h_r} \bigwedge_{k=0}^{m-1} [E'_k] \\ h_n \xrightarrow{h_r} \bigwedge_{k=0}^{n-1} [E_k] \end{cases} \equiv \begin{cases} h_m \xrightarrow{h_r} \bigwedge_{k=0}^{l-1} [E_k] \wedge \bigwedge_{k=l}^{m-1} [E'_k] \\ h_n \xrightarrow{h_r} \bigwedge_{k=0}^{l-1} [E_k] \wedge \bigwedge_{k=l}^{n-1} [E_k] \end{cases}$$

$$\begin{aligned} \mathcal{F} &= h_m \vee h_n \\ \equiv\ \mathcal{F} &\xrightarrow{h_r} (\bigwedge_{k=0}^{l-1} [E_k] \wedge \bigwedge_{k=l}^{m-1} [E'_k]) \vee (\bigwedge_{k=0}^{l-1} [E_k] \wedge \bigwedge_{k=l}^{n-1} [E_k]) \\ \equiv\ \mathcal{F} &\xrightarrow{h_r} (\underbrace{\bigwedge_{k=l}^{m-1} [E'_k]}_{h_m \text{ à } h_l} \vee \underbrace{\bigwedge_{k=l}^{n-1} [E_k]}_{h_n \text{ à } h_l}) \wedge \bigwedge_{k=0}^{l-1} [E_k] \end{aligned}$$

$$\mathcal{F} \xrightarrow{h_l} \underset{(\text{à } h_l)}{h_m} \vee \underset{(\text{à } h_l)}{h_n}$$

Par conséquent,

$$\mathcal{F} \xrightarrow{h_l} \bigwedge_{k=l}^{m-1} [E'_k] \vee \bigwedge_{k=l}^{n-1} [E_k]$$

2. $\mathcal{F} := \mathcal{IDH} \wedge \mathcal{IDH}$

On montre de la même façon, pour une formule $\mathcal{F} = h_m \wedge h_n$.

$$\begin{array}{rcll} & \mathcal{F} & = & h_m \wedge h_n \\ \equiv & \mathcal{F} & \xrightarrow{h_r} & (\bigwedge_{k=0}^{l-1}[E_k] \wedge \bigwedge_{k=l}^{m-1}[E'_k]) \wedge (\bigwedge_{k=0}^{l-1}[E_k] \wedge \bigwedge_{k=l}^{n-1}[E_k]) \\ \equiv & \mathcal{F} & \xrightarrow{h_r} & (\bigwedge_{k=l}^{m-1}[E'_k] \wedge \bigwedge_{k=l}^{n-1}[E_k]) \wedge \bigwedge_{k=0}^{l-1}[E_k] \end{array}$$

Par conséquent,

$$\mathcal{F} \xrightarrow{h_l} \bigwedge_{k=l}^{m-1}[E'_k] \wedge \bigwedge_{k=l}^{n-1}[E_k]$$

3. $\mathcal{F} := \mathcal{IDH} \wedge \overline{\mathcal{IDH}}$

Afin de simplifier le calcul de normalisation de la formule $\mathcal{F} = h_m \wedge \overline{h_n}$, on pose,

- $\mathrm{M} = \bigwedge_{k=l}^{m-1}[E'_k]$
- $\mathrm{N} = \bigwedge_{k=l}^{n-1}[E_k]$
- $\mathrm{L} = \bigwedge_{k=0}^{l-1}[E_k]$

on a donc,

$$\begin{array}{rcll} & \mathcal{F} & = & h_m \wedge \overline{h_n} \\ \equiv & \mathcal{F} & \xrightarrow{h_r} & (\,\mathrm{M} \wedge \mathrm{L}\,) \wedge \overline{(\,\mathrm{N} \wedge \mathrm{L}\,)} \\ \equiv & \mathcal{F} & \xrightarrow{h_r} & \mathrm{M} \wedge (\,\mathrm{L} \wedge \overline{(\,\mathrm{N} \wedge \mathrm{L}\,)}\,) \\ \equiv & \mathcal{F} & \xrightarrow{h_r} & \mathrm{M} \wedge (\,\mathrm{L} \wedge (\,\overline{\mathrm{N}} \vee \overline{\mathrm{L}}\,)\,) \\ \equiv & \mathcal{F} & \xrightarrow{h_r} & \mathrm{M} \wedge ((\,\mathrm{L} \wedge \overline{\mathrm{N}}\,) \vee (\underbrace{\mathrm{L} \wedge \overline{\mathrm{L}}}_{\emptyset})\,) \\ \equiv & \mathcal{F} & \xrightarrow{h_r} & (\,\mathrm{M} \wedge \overline{\mathrm{N}}\,) \wedge \mathrm{L} \end{array}$$

Par conséquent,

$$\mathcal{F} \xrightarrow{h_l} \bigwedge_{k=l}^{m-1}[E'_k] \wedge \overline{\bigwedge_{k=l}^{n-1}[E_k]}$$

4. $\mathcal{F} := \overline{\mathcal{IDH}} \wedge \mathcal{IDH}$

La commutativité de l'opérateur $\wedge$ ramène l'étude de ce cas au cas précédent. Par conséquent,

$$\mathcal{F} \xrightarrow{h_l} \overline{\bigwedge_{k=l}^{m-1}[E'_k]} \wedge \bigwedge_{k=l}^{n-1}[E_k]$$

II.3.3 Application

On peut ainsi simplifier le processus de normalisation de l'arbre d'horloges donné en figure II.3:

$\mathcal{VH}$	$\mathcal{VHC}$
$h_1 \xrightarrow{h_r} h_r \wedge \overline{h_3}$	$h_2 \xrightarrow{h_r} [a]$
$h_3 \xrightarrow{h_2} h_5 \wedge \overline{h_8}$	$h_4 \xrightarrow{h_2} [b]$
$h_8 \xrightarrow{h_4} h_6 \vee h_7$	$h_5 \xrightarrow{h_2} [c]$
	$h_6 \xrightarrow{h_4} [d]$
	$h_7 \xrightarrow{h_4} [e]$

$\equiv$

$\mathcal{VH}$	$\mathcal{VHC}$
$h_1 \xrightarrow{h_r} h_r \wedge \overline{h_3}$	$h_2 \xrightarrow{h_r} [a]$
$h_3 \xrightarrow{h_2} h_5 \wedge \overline{h_8}$	$h_4 \xrightarrow{h_2} [b]$
	$h_5 \xrightarrow{h_2} [c]$
	$h_6 \xrightarrow{h_4} [d]$
	$h_7 \xrightarrow{h_4} [e]$
	$h_8 \xrightarrow{h_4} [d] \vee [e]$

La normalisation de l'horloge h_8 est directe, puisque ses arguments h_6, $h_7 \in \mathcal{VHC}$, et que les expressions de h_6 et h_7 à h_4 sont connues.

Puisque qu'une expression contrainte de h_8 a été déterminée, la normalisation de h_3 est désormais possible, car tous ses arguments sont maintenant évalués dans $\mathcal{VHC}$. Toutefois, la normalisation de h_3 requiert une expression de h_5 et h_8 à h_2. Une telle expression existe pour h_5 dans le système d'équations, mais nous avons par contre besoin d'évaluer h_8 à h_2.

$$h_8 \xrightarrow{h_2} [b] \wedge ([d] \vee [e])$$

Par conséquent

$\mathcal{VH}$	$\mathcal{VHC}$
$h_1 \xrightarrow{h_r} h_r \wedge \overline{h_3}$	$h_2 \xrightarrow{h_r} [a]$
	$h_4 \xrightarrow{h_2} [b]$
	$h_5 \xrightarrow{h_2} [c]$
	$h_6 \xrightarrow{h_4} [d]$
	$h_7 \xrightarrow{h_4} [e]$
	$h_8 \xrightarrow{h_4} [d] \vee [e]$
	$h_3 \xrightarrow{h_2} [c] \wedge \overline{(([d] \vee [e]) \wedge [b])}$

$\equiv$

La normalisation de h_1 réclame une expression contrainte de h_3 à l'horloge mère:

$$h_3 \xrightarrow{h_r} [a] \wedge ([c] \wedge \overline{(([d] \vee [e]) \wedge [b])})$$

ce qui permet de conclure par le système suivant:

$$
\equiv \quad
\begin{array}{c|l}
\mathcal{VH} & \mathcal{VHC} \\
\hline
\emptyset & h_2 \xrightarrow{h_r} [a] \\
 & h_4 \xrightarrow{h_2} [b] \\
 & h_5 \xrightarrow{h_2} [c] \\
 & h_6 \xrightarrow{h_4} [d] \\
 & h_7 \xrightarrow{h_4} [e] \\
 & h_8 \xrightarrow{h_4} [d] \vee [e] \\
 & h_3 \xrightarrow{h_2} [c] \wedge \overline{(([d] \vee [e]) \wedge [b])} \\
 & h_1 \xrightarrow{h_r} [a] \wedge ([c] \wedge \overline{(([d] \vee [e]) \wedge [b])})
\end{array}
$$

Si les opérations intermédiaires, ainsi que les résultats finaux ont été en partie simplifiés, le processus de substitution, de même que quelques expressions d'horloge demeurent par trop complexes. Des problèmes sont manifestement posés par l'opération de négation logique: elle engendre des expressions et calculs difficiles. Nous nous intéressons plus particulièrement à ce problème dans la section suivante.

II.4 Fréquence complémentaire

La notion de fréquence complémentaire dénote en fait la négation logique d'une horloge: $\overline{\mathcal{IDH}}$. Nous avons différencié dans cette étude, le cas des variables contraintes et non contraintes.

II.4.1 Variables contraintes

On a déjà vu qu'une expression booléenne `E` construite sur un ensemble de signaux synchrones, et dont h_E dénote l'horloge de ses arguments et de son résultat, engendre deux nouvelles horloges notées `[E]` et `[$\overline{\text{E}}$]`. Ces horloges définissent respectivement l'ensemble des instants de h_E pour lesquels la condition `E` est vérifiée, et l'ensemble des instants de h_E pour lesquels elle n'est pas vérifiée.

On a représenté en figure II.6, les branches contraintes de l'arbre de la figure II.3 accompagnées de leurs expressions complémentaires.

Rappel: Le signal `a` est émis à l'horloge h_r, les signaux `b` et `c` à l'horloge h_2, et les signaux `d` et `e` à l'horloge h_4.

Les fréquences complémentaires s'expriment de la manière suivante:

$$
h_2 = [a] \equiv \begin{cases} h_2 \xrightarrow{h_r} [a] \\ \overline{h_2} \xrightarrow{h_r} [\overline{a}] \end{cases}
\qquad
h_4 = [b] \equiv \begin{cases} h_4 \xrightarrow{h_2} [b] \\ \overline{h_4} \xrightarrow{h_2} [\overline{b}] \end{cases}
$$

$$
h_5 = [c] \equiv \begin{cases} h_5 \xrightarrow{h_2} [c] \\ \overline{h_5} \xrightarrow{h_2} [\overline{c}] \end{cases}
\qquad
h_6 = [d] \equiv \begin{cases} h_6 \xrightarrow{h_4} [d] \\ \overline{h_6} \xrightarrow{h_4} [\overline{d}] \end{cases}
$$

$$
h_7 = [e] \equiv \begin{cases} h_7 \xrightarrow{h_4} [e] \\ \overline{h_7} \xrightarrow{h_4} [\overline{e}] \end{cases}
$$

Nous nous intéressons dans cette section aux manières d'exprimer la négation d'une variable contrainte à une horloge qui n'est pas son horloge de définition.

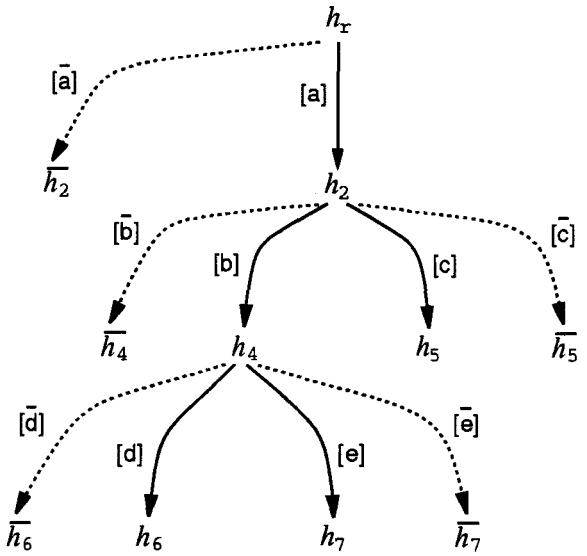


FIG. II.6 - *Expressions complémentaires*

Par exemple, dans l'arbre de la figure II.6, on a l'inclusion suivante d'horloges: $h_2 \supseteq h_4 \supseteq h_6$. L'expression de h_6 à l'horloge h_2 dénote la relation temporelle suivante:

h_2	$\top$	$\top$	$\top$	...
[b]	*vrai*	*faux*	*vrai*	...
h_4	$\top$	$\bot$	$\top$	...
[d]	*vrai*	$\bot$	*faux*	...
h_6	$\top$	$\bot$	$\bot$	...

Autrement dit, le signal **b** est émis à condition que son horloge de définition h_2 soit valide (notation: $\top$). L'horloge h_4 est définie par la suite des instants auxquels la condition **b** est vérifiée. Le signal **d** est émis à l'horloge h_4, et la suite des instants auxquels il prend la valeur *vrai* définit la variable h_6.

En d'autres termes, exprimer la variable contrainte h_6 à l'horloge h_2, c'est attendre en tout premier lieu le signal **b**, pour en tester la valeur, puis si et seulement si il porte la valeur *vrai*, alors le signal **d** peut être à son tour attendu pour être évalué. On constate notamment que si **b** est *faux*, le signal **d** ne sera jamais émis. Donc,

$$h_6 \xrightarrow{h_2} [b] \wedge [d]$$

L'horloge $\overline{h_6}$ dénote l'expression complémentaire de h_6, c'est-à-dire:

$$\overline{h_6} \xrightarrow{h_2} \overline{[b] \wedge [d]}$$

Les signaux **b** et **d** n'étant pas émis à la même horloge, une reformulation de cette expression est nécessaire, de manière à dissocier les deux conditions:

$$\overline{[b] \wedge [d]} = [\bar{b}] \vee [\bar{d}]$$

Cette égalité traduit la relation temporelle suivante à l'horloge h_2:

h_2	$\top$	$\top$	$\top$	...
[b]	*vrai*	*faux*	*vrai*	...
h_4	$\top$	$\bot$	$\top$	...
[d]	*vrai*	$\bot$	*faux*	...
h_6	$\top$	$\bot$	$\bot$	...
$\overline{h_6}$	$\bot$	$\top$	$\top$	...

Connaissant l'expression contrainte de l'horloge h_6 à la fréquence h_2, il est possible d'en déduire une expression contrainte de $\overline{h_6}$, sans autre calcul, par un simple parcours de l'arbre d'horloge.

$$\overline{h_6} \xrightarrow{h_2} [\bar{b}] \vee ([b] \wedge [\bar{d}])$$

Les expressions $\begin{cases} \overline{h_6} \xrightarrow{h_2} [\bar{b}] \vee [\bar{d}] \\ \overline{h_6} \xrightarrow{h_2} [\bar{b}] \vee ([b] \wedge [\bar{d}]) \end{cases}$ sont équivalentes lorsque $h_2 \supseteq h_4 \supseteq h_6$.

Généralisation

Dans l'arbre d'horloges donné en figure II.5, l'expression à l'horloge h_l de la variable contrainte h_n est égale à:

$$h_n \xrightarrow{h_l} \bigwedge_{k=l}^{n-1} [E_k] = [E_l] \wedge [E_{l+1}] \wedge [E_{l+2}] \wedge \ldots \wedge [E_{n-1}]$$

Par conséquent,

$$\begin{array}{lll} \overline{h_n} \xrightarrow{h_l} & [\overline{E_l}] & \vee \\ & ([E_l] \wedge [\overline{E_{l+1}}]) & \vee \\ & ([E_l] \wedge [E_{l+1}] \wedge [\overline{E_{l+2}}]) & \vee \\ & \ldots & \vee \\ & ([E_l] \wedge [E_{l+1}] \wedge \ldots \wedge [E_{n-2}] \wedge [\overline{E_{n-1}}]) & \end{array}$$

De manière plus synthétique, l'expression complémentaire d'une variable contrainte h_n à l'horloge h_l est donnée par la relation suivante:

$$\overline{h_n} \xrightarrow{h_l} \bigvee_{k=l}^{n-1} \left([\overline{E_k}] \bigwedge_{\substack{i=l \\ k>l}}^{k-1} [E_i] \right)$$

II.4.2 Variables non contraintes

La variable non contrainte h_1 dans l'arbre de la figure II.3, illustre toute la difficulté du problème à résoudre. Nous avions en effet obtenu à la fin du calcul précédent (c.f. page 54), l'expression contrainte suivante pour h_1:

$$h_1 \xrightarrow{h_r} \overline{[a] \wedge ([c] \wedge \overline{(([d] \vee [e]) \wedge [b])})}$$

Cette variable dans le système initial d'équations était définie par la négation de la formule h_3. Les opérations de substitution nous ont donné une expression contrainte de h_3 à h_2:

$$h_3 \xrightarrow{h_2} [c] \wedge \overline{(([d] \vee [e]) \wedge [b])}$$

La normalisation de h_1 requiert la définition de $\overline{h_3}$ à l'horloge h_r. Puisque nous disposons d'une expression contrainte de la formule h_3, nous pourrions lui appliquer le résultat précédent, et définir ainsi $\overline{h_3}$:

$$\overline{h_3} \xrightarrow{h_r} [\overline{a}] \vee ([a] \wedge \overline{[c] \wedge \overline{([[d] \vee [e]) \wedge [b]]}})$$

Le problème posé par de telles expressions (e.g. h_3, h_1), outre leur complexité intrinsèque, est qu'elles requièrent une phase supplémentaire de ré-écriture. Sur cet exemple, on comprend aisément qu'il serait déraisonnable d'attendre d'un compilateur qu'il puisse manipuler efficacement de telles expressions. En effet, les formules peuvent être ainsi récursivement composées.

L'approche plus pragmatique que nous avons adoptée passe par la décomposition des expressions non contraintes, et la création de variables intermédiaires. Cette technique est illustrée sur le système suivant, dans lequel "o" dénote une opération logique binaire: "$\wedge$" ou "$\vee$".

$\mathcal{VH}$	$\mathcal{VHC}$		$\mathcal{VH}$	$\mathcal{VHC}$	
$h_{\alpha_1} = h_{\alpha_2} \wedge \overline{h_{\alpha_3}}$ $h_{\alpha_3} = h_{\alpha_4}$ o h_{α_5}	$h_{\alpha_2} = [E_2]$ $h_{\alpha_4} = [E_4]$ $h_{\alpha_5} = [E_5]$	$\equiv$	$h_{\alpha_1} = h_{\alpha_2} \wedge \overline{h_{\alpha_3}}$ $\overline{h_{\alpha_3}} = \overline{h_{\alpha_4}}$ $\overline{\text{o}}$ $\overline{h_{\alpha_5}}$	$h_{\alpha_2} = [E_2]$ $h_{\alpha_4} = [E_4]$ $h_{\alpha_5} = [E_5]$ $h_{\alpha_3} = [E_4]$ o $[E_5]$	$\equiv$

Suivant le principe de décomposition des formules, la variable $\overline{h_{\alpha_3}}$ dans l'expression de définition de h_{α_1} a conditionné la création d'une formule intermédiaire à partir de l'expression de définition de h_{α_3}:

$$\overline{h_{\alpha_3}} = \overline{h_{\alpha_4}} \ \overline{\text{o}} \ \overline{h_{\alpha_5}}$$

Parallèlement, nous avons vu qu'une variable ($\in \mathcal{VH}$) peut être normalisée lorsque les arguments contenus dans son membre droit sont tous définis par une expression contrainte. Par exemple, dans le système initial, les arguments de définition de la formule h_{α_3} sont des variables contraintes. Une expression normalisée de h_{α_3} est donc calculable.

$\mathcal{VH}$	$\mathcal{VHC}$		$\mathcal{VH}$	$\mathcal{VHC}$
$h_{\alpha_1} = h_{\alpha_2} \wedge \overline{h_{\alpha_3}}$	$h_{\alpha_2} = [E_2]$ $h_{\alpha_4} = [E_4]$ $h_{\alpha_5} = [E_5]$ $\overline{h_{\alpha_4}} = \overline{[E_4]}$ $\overline{h_{\alpha_5}} = \overline{[E_5]}$ $h_{\alpha_3} = [E_4]$ o $[E_5]$ $\overline{h_{\alpha_3}} = \overline{[E_4]}$ $\overline{\text{o}}$ $\overline{[E_5]}$	$\equiv$	$\emptyset$	$h_{\alpha_2} = [E_2]$ $h_{\alpha_4} = [E_4]$ $h_{\alpha_5} = [E_5]$ $\overline{h_{\alpha_4}} = \overline{[E_4]}$ $\overline{h_{\alpha_5}} = \overline{[E_5]}$ $h_{\alpha_3} = [E_4]$ o $[E_5]$ $\overline{h_{\alpha_3}} = \overline{[E_4]}$ $\overline{\text{o}}$ $\overline{[E_5]}$ $h_{\alpha_1} = [E_2] \wedge (\overline{[E_4]} \ \overline{\text{o}} \ \overline{[E_5]})$

Les variables $\overline{h_{\alpha_4}}$ et $\overline{h_{\alpha_5}}$, ont à leur tour conditionné la création d'expressions intermédiaires à partir de respectivement, h_{α_4} et h_{α_5}.

Les variables $\overline{h_{\alpha_4}}$ et $\overline{h_{\alpha_5}}$, dans la formule $\overline{h_{\alpha_3}}$ ont ainsi été substituées par leurs expressions contraintes, permettant la normalisation de $\overline{h_{\alpha_3}}$.

Une expression contrainte de $\overline{h_{\alpha_3}}$ ayant été obtenue, la normalisation de h_{α_1} a pu à son tour être entreprise.

En fin de résolution, les variables intermédiaires sont supprimées, et le système final est ainsi réduit à:

$\mathcal{VH}$	$\mathcal{VHC}$
$\emptyset$	$h_{\alpha_1} = [E_2] \wedge ([\overline{E_4}] \,\bar{\circ}\, [\overline{E_5}])$ $h_{\alpha_3} = [E_4] \circ [E_5]$ $h_{\alpha_2} = [E_2]$ $h_{\alpha_4} = [E_4]$ $h_{\alpha_5} = [E_5]$

II.4.3 Application

Nous reprenons notre exemple (c.f. figure II.3), dont le système initial était:

$\mathcal{VH}$	$\mathcal{VHC}$
$h_1 \xrightarrow{h_r} h_r \wedge \overline{h_3}$	$h_2 \xrightarrow{h_r} [a]$
$h_3 \xrightarrow{h_2} h_5 \wedge \overline{h_8}$	$h_4 \xrightarrow{h_2} [b]$
$h_8 \xrightarrow{h_4} h_6 \vee h_7$	$h_5 \xrightarrow{h_2} [c]$
	$h_6 \xrightarrow{h_4} [d]$
	$h_7 \xrightarrow{h_4} [e]$

À titre d'illustration, nous allons détailler les opérations de ré-écriture de la formule h_1:

$$h_1 \xrightarrow{h_r} h_r \wedge \overline{h_3}$$

Cette expression conditionne la création d'une expression intermédiaire dénotée $\overline{h_3}$,

$$\begin{cases} h_3 \xrightarrow{h_2} h_5 \wedge \overline{h_8} \\ \overline{h_3} \xrightarrow{h_2} \overline{h_5} \vee h_8 \quad \text{règle de DE MORGAN} \end{cases}$$

Récursivement, l'expression non contrainte de $\overline{h_3}$ entraîne la création d'une horloge intermédiaire $\overline{h_5}$,

$$\begin{cases} h_5 \xrightarrow{h_2} [c] \\ \overline{h_5} \xrightarrow{h_2} [\bar{c}] \end{cases}$$

L'ensemble des expressions intermédiaires décrit donc en quelque sorte un nouvel arbre d'horloges qui est représenté en figure II.7.

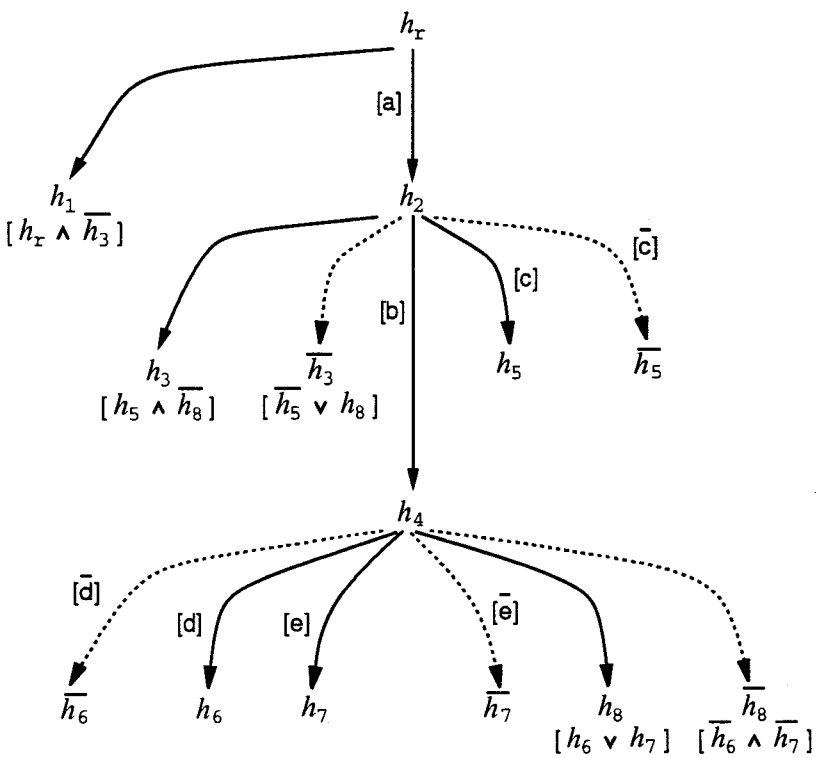


FIG. II.7 - *Arbre d'horloges intermédiaire.*

À partir de l'arbre d'horloges complété de ses expressions intermédiaires, le processus de normalisation peut être entrepris.

$\mathcal{VH}$	$\mathcal{VHC}$
$h_1 \xrightarrow{h_r} h_r \wedge \overline{h_3}$	$h_2 \xrightarrow{h_r} [a]$
$h_3 \xrightarrow{h_2} h_5 \wedge \overline{h_8}$	$h_4 \xrightarrow{h_2} [b]$
$h_8 \xrightarrow{h_4} h_6 \vee h_7$	$h_5 \xrightarrow{h_2} [c]$
$\overline{h_3} \xrightarrow{h_2} \overline{h_5} \vee h_8$	$h_6 \xrightarrow{h_4} [d]$
$\overline{h_8} \xrightarrow{h_4} \overline{h_6} \wedge \overline{h_7}$	$h_7 \xrightarrow{h_4} [e]$
	$\overline{h_5} \xrightarrow{h_2} [\overline{c}]$
	$\overline{h_6} \xrightarrow{h_4} [\overline{d}]$
	$\overline{h_7} \xrightarrow{h_4} [\overline{e}]$

$\equiv$

$\mathcal{VH}$	$\mathcal{VHC}$
$h_1 \xrightarrow{h_r} h_r \wedge \overline{h_3}$	$h_2 \xrightarrow{h_r} [a]$
$h_3 \xrightarrow{h_2} h_5 \wedge \overline{h_8}$	$h_4 \xrightarrow{h_2} [b]$
$\overline{h_3} \xrightarrow{h_0} \overline{h_5} \vee h_8$	$h_5 \xrightarrow{h_2} [c]$
	$h_6 \xrightarrow{h_4} [d]$
	$h_7 \xrightarrow{h_4} [e]$
	$\overline{h_5} \xrightarrow{h_2} [\overline{c}]$
	$\overline{h_6} \xrightarrow{h_4} [\overline{d}]$
	$\overline{h_7} \xrightarrow{h_4} [\overline{e}]$
	$\overline{h_8} \xrightarrow{h_4} [\overline{d}] \wedge [\overline{e}]$
	$h_8 \xrightarrow{h_4} [d] \vee [e]$

$\equiv$

$\mathcal{VH}$	$\mathcal{VHC}$
$h_1 \xrightarrow{h_r} h_r \wedge \overline{h_3}$	$h_2 \xrightarrow{h_r} [a]$
	$h_4 \xrightarrow{h_2} [b]$
	$h_5 \xrightarrow{h_2} [c]$
	$h_6 \xrightarrow{h_4} [d]$
	$h_7 \xrightarrow{h_4} [e]$
	$\overline{h_5} \xrightarrow{h_2} [\overline{c}]$
	$\overline{h_6} \xrightarrow{h_4} [\overline{d}]$
	$\overline{h_7} \xrightarrow{h_4} [\overline{e}]$
	$\overline{h_8} \xrightarrow{h_4} [\overline{d}] \wedge [\overline{e}]$
	$h_8 \xrightarrow{h_4} [d] \vee [e]$
	$h_3 \xrightarrow{h_2} [c] \wedge ([\overline{b}] \vee ([b] \wedge [\overline{d}] \wedge [\overline{e}]))$
	$\overline{h_3} \xrightarrow{h_0} [\overline{c}] \vee ([b] \wedge ([d] \vee [e]))$

$$
\equiv \quad
\begin{array}{c|l}
\mathcal{VH} & \mathcal{VHC} \\
\hline
\emptyset & h_2 \xrightarrow{h_r} [a] \\
 & h_4 \xrightarrow{h_2} [b] \\
 & h_5 \xrightarrow{h_2} [c] \\
 & h_6 \xrightarrow{h_4} [d] \\
 & h_7 \xrightarrow{h_4} [e] \\
 & \overline{h_5} \xrightarrow{h_2} [\overline{c}] \\
 & \overline{h_6} \xrightarrow{h_4} [\overline{d}] \\
 & \overline{h_7} \xrightarrow{h_4} [\overline{e}] \\
 & \overline{h_8} \xrightarrow{h_4} [\overline{d}] \wedge [\overline{e}] \\
 & h_8 \xrightarrow{h_4} [d] \vee [e] \\
 & h_3 \xrightarrow{h_2} [c] \wedge ([\overline{b}] \vee ([b] \wedge [\overline{d}] \wedge [\overline{e}])) \\
 & \overline{h_3} \xrightarrow{h_0} [\overline{c}] \vee ([b] \wedge ([d] \vee [e])) \\
 & h_1 \xrightarrow{h_r} [\overline{a}] \vee ([a] \wedge ([\overline{c}] \vee ([b] \wedge ([d] \vee [e]))))
\end{array}
$$

Le résultat final de la phase de normalisation donne donc, après suppression des horloges intermédiaires:

$$
\equiv \quad
\begin{array}{c|l}
\mathcal{VH} & \mathcal{VHC} \\
\hline
\emptyset & h_1 \xrightarrow{h_r} [\overline{a}] \vee ([a] \wedge ([\overline{c}] \vee ([b] \wedge ([d] \vee [e])))) \\
 & h_2 \xrightarrow{h_r} [a] \\
 & h_3 \xrightarrow{h_2} [c] \wedge ([\overline{b}] \vee ([b] \wedge [\overline{d}] \wedge [\overline{e}])) \\
 & h_4 \xrightarrow{h_2} [b] \\
 & h_5 \xrightarrow{h_2} [c] \\
 & h_6 \xrightarrow{h_4} [d] \\
 & h_7 \xrightarrow{h_4} [e] \\
 & h_8 \xrightarrow{h_4} [d] \vee [e]
\end{array}
$$

II.5 Conclusion

Le processus de ré-écriture que nous venons de décrire, peut être qualifié de complexité raisonnable. Il s'effectue par des opérations simple, et manipule des expressions relativement élémentaires.

En outre, le système final obtenu, peut être directement interprété sous forme de commandes gardées. En effet, prenons par exemple l'expression de h_1 dans le système précédent:

$$h_1 \xrightarrow{h_r} [\overline{a}] \vee ([a] \wedge ([\overline{c}] \vee ([b] \wedge ([d] \vee [e]))))$$

Un ordre partiel existe entre les conditions référencées dans les expressions contraintes. Cette hiérarchie se traduit par l'horloge de définition associée à chaque condition. Nous avons fait figurer dans l'expression de h_1, les différentes horloges auxquelles sont émises les différentes conditions:

$$h_1 \xrightarrow{h_r} \overset{h_r}{[\overline{a}]} \vee ([a] \wedge ([\overline{c}] \vee ([b] \wedge ([d] \vee [e])))) \quad \overset{h_r}{[a]} \quad \overset{h_2}{[\overline{c}]} \quad \overset{h_2}{[b]} \quad \overset{h_4}{[d]} \quad \overset{h_4}{[e]}$$

La hiérarchie des dépendances peut également être représentée plus clairement sur la base d'un arbre (c.f. figure II.8), dans lequel chaque chemin de la racine à une feuille est une expression validant l'horloge h_1. Une strate (ensemble de fils d'un même nœud) définit en largeur une somme de contraintes, et en profondeur, un produit.

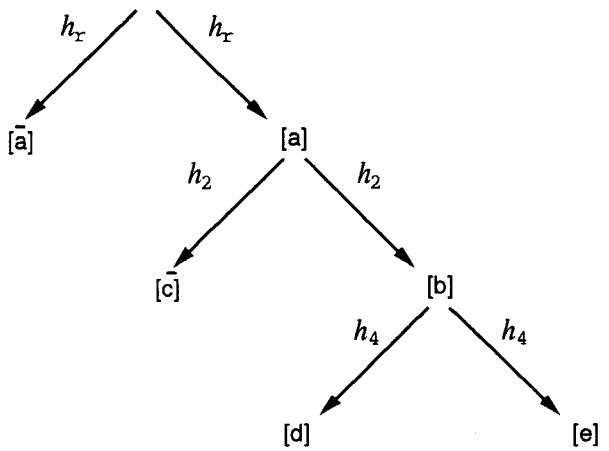


FIG. II.8 - *Cascade des dépendances induites par l'horloge* h_1.

En termes de commandes gardées, l'ordre partiel existant entre les différentes conditions est interprété par une série d'imbrications, dans laquelle chaque horloge traduit l'attente d'un nouveau flot de données. On pourrait par exemple représenter l'horloge h_1 comme une fonction renvoyant une valeur instantanée (un `TICK`) $\top$ ou $\bot$.

```
                TICK function h1(void)
                {
hr                wait_for(a);
                  if (ā)
                    return ⊤;
                  else {
h2                  wait_for(b,c);
                    if (c̄)
                      return ⊤;
                    if (b) {
h4                    wait_for(d,e);
                      if (d)
                        return ⊤;
                      if (e)
                        return ⊤;
                    }
                  }
                  return ⊥;
                }
```

Les expressions contraintes que nous avons utilisées jusqu'à présent dans nos exemples sont simples. Elles étaient définies par le sous-échantillonnage de signaux booléens produits par des fonctions de calcul. C'est-à-dire par des expressions du type:

$$h_1 = [x]$$

où le signal **x** est un flot booléen produit par une tâche du système. Dans le cas général, le flot **x**, est comme tout signal, défini par une expression du type:

$$\mathbf{x} := (s_1 \texttt{ when } hu_x^{s_1}) \texttt{ default } \ldots \texttt{ default } (s_n \texttt{ when } hu_x^{s_n}) \texttt{ default } (x\$1 \texttt{ when } h_{x\$})$$

où,

- les $\mathbf{s}_i$'s peuvent être pour une part des signaux de sortie de tâches, et pour l'autre part des signaux intermédiaires eux-même respectivement définis par des expressions,
- $hu_x^{s_i}$ est appelée *horloge d'utilisation* du signal $\mathbf{s}_i$ pour définir **x**,
- les horloges d'utilisation s'excluent mutuellement,
- $h_{x\$}$ peut être nulle, et est exclusive de toutes les $hu_x^{s_i}$.

Les expressions $\mathbf{s}_j$'s définissant des signaux intermédiaires sont construites sur un ensemble de signaux $(\sigma_1, \ldots, \sigma_n)$; chacun de ces signaux peut apparaître dans l'expression de définition de plusieurs $\mathbf{s}_j$'s.

Par conséquent, dans le cas général, les signaux référencés dans l'expression **x** peuvent aussi bien être des signaux de sortie que des signaux intermédiaires. Notre objectif étant d'établir des dépendances directes entre producteurs et consommateurs de signaux, il importe donc de transformer ces expressions de manière à se débarrasser de toute référence à des signaux intermédiaires.

Le processus de transformation que nous appliquons est commun à celui utilisé pour définir les entrées d'une tâche qui sera décrit dans le chapitre suivant. Nous ne nous attarderons donc pas à le détailler ici.

On peut résumer ce chapitre (voir également la figure II.9) en disant que l'ensemble des horloges d'un programme se compose de variables contraintes et non contraintes. Parmi les variables contraintes, un certain nombre d'entre elles ne sont pas construites directement sur des signaux de sortie. Le processus de transformation commun à celui utilisé pour définir les entrées de tâche leur est appliqué. Il en résulte de nouvelles expressions ne contenant plus de signaux intermédiaires.

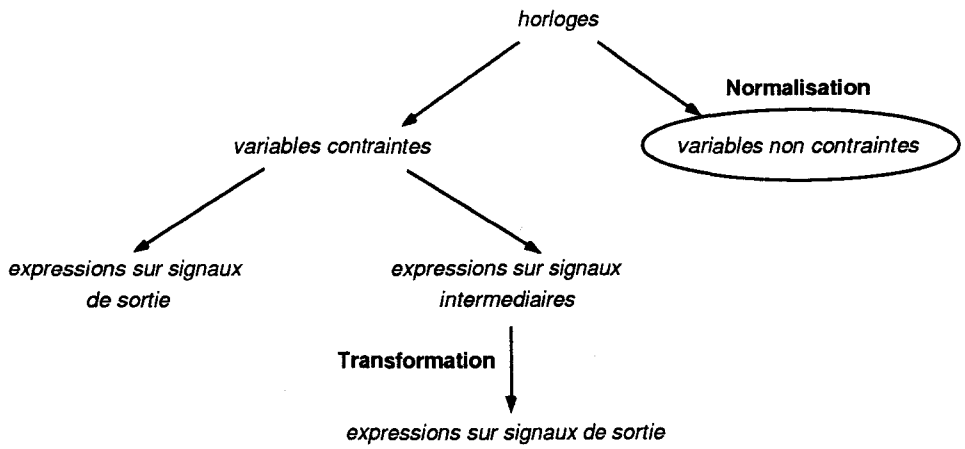


FIG. II.9 - *Le processus de normalisation.*

II.6 Application: Un codeur de type MPEG

Nous proposons d'illustrer notre démarche sur la base d'une étude pratique: l'implémentation d'un codeur vidéo temps-réel de type MPEG-2 [Tud95] (c.f. figure II.10). Cet exemple a également été donné en démonstration lors de la conférence internationale ICIP'96 [GDM96b] sur le traitement d'image.

Le codeur transforme des séquences potentiellement infinies d'images provenant par exemple d'une caméra, en flots codés. Deux types d'images sont traitées,

- les trames de type `I` (*"Intra" pictures*), qui sont compressées directement sans faire référence à d'autres images *Intra Frame Coding* (IFC),

- les trames de type `P` (*"Predictive" pictures*), qui sont compressées en se référant à l'image (de type *I* ou *P*) précédente: *Mono-Directional Motion Estimator coding* (MDMEC).

Les images de type `B` (*"Bidirectionnaly-predictive" pictures*) qui se présentent traditionnellement dans les séquences vidéo en MPEG-2, ne sont pas traitées ici. Cette simplification correspond au *"simple profile"* utilisé dans les applications ne tolérant pas de retard important, telles que les applications de vidéo conférence. En effet, le codage des images de type `B` nécessite une phase supplémentaire de réordonnancement des séquences vidéo.

Les séquences d'images ne sont pas définies de manière standard, si bien que les trames peuvent apparaître dans n'importe quel ordre. Une séquence peut ainsi ressembler à:

Occurrences	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Type d'image	I	I	I	P	P	P	I	P	I	P

Le comportement temporel de ce programme est schématisé dans le "chronogramme" de la figure II.11, sur le flot hypothétique d'entrée donné précédemment. La présence d'un signal à une occurrence de temps donnée y est figurée par un point noir, accompagné d'une valeur (I_i, P_i, Ic_i, ...) lorsque celle-ci s'avère pertinente.

Le programme est ainsi régulé par la tâche `STDI` qui alimente le système par un flot continu d'entrées: signaux `IMAGE` et `I_type`. Tous les autres signaux du programme sont échantillonnés sur la base de ce couple d'entrées.

La valeur instantanée portée par le signal booléen `I_type` détermine la création des flots `I` et `P`: lorsque ce signal est *vrai*, le flot `I` est défini et `P` ne l'est pas — inversement, lorsque le signal `I_type` porte la valeur *faux*, le flot `P` est défini tandis que `I` ne l'est pas.

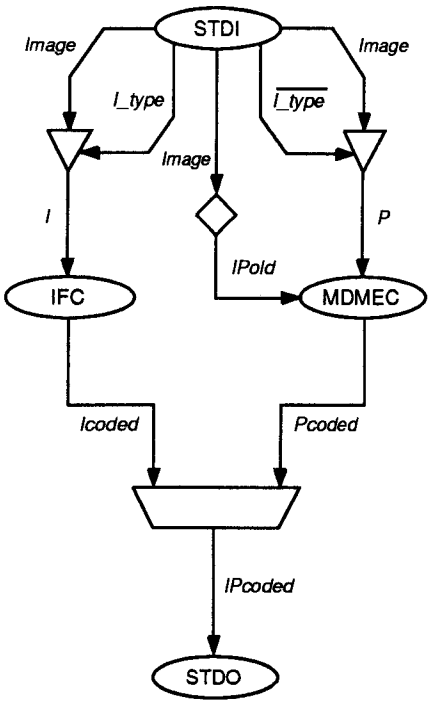
Le flot `IPold` maintient la valeur du signal `IMAGE` pour l'occurrence temporelle suivante. Conformément à la sémantique de l'opérateur de retard en SIGNAL, les séquences `IPold` et `IMAGE` sont synchrones. À la première occurrence de temps, le signal `IPold` n'est pas défini, il pourrait toutefois l'être par une valeur d'initialisation.

Les flots `Icoded` et `Pcoded` produits par les tâches `IFC` et `MDMEC` sont respectivement synchrones avec les signaux `I` et `P`.

Le signal `IPcoded` est cadencé à la même horloge que les signaux `IMAGE` et `I_type`. Il correspond en effet à l'entrelacement des trames `I` et `P` codées.

II.6.1 Processus de normalisation

Le graphe d'horloges de ce système est représenté dans le schéma de la figure II.12. Le processus de normalistion qui lui est appliqué donne:



(1)

FIG. II.10 - *Codeur vidéo temps-réel de type MPEG. Implémentation sur la machine P^3I.*

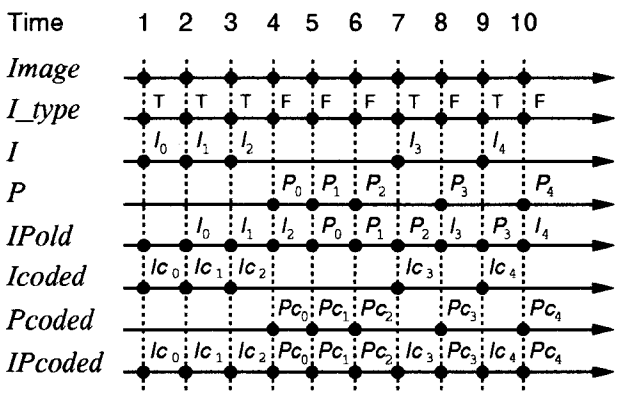


FIG. II.11 - *Flot hypothétique d'entrée.*

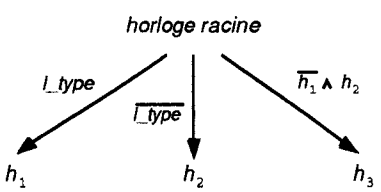


FIG. II.12 - *Arbre d'horloges.*

$\mathcal{VH}$	$\mathcal{VHC}$
$h_3 \xrightarrow{h_{root}} h_2 \wedge \overline{h_1}$	$h_1 \xrightarrow{h_{root}} [\texttt{I\_type}]$ $h_2 \xrightarrow{h_{root}} \overline{[\texttt{I\_type}]}$

$\equiv$

$\mathcal{VH}$	$\mathcal{VHC}$
$h_3 \xrightarrow{h_{root}} h_2 \wedge \overline{h_1}$	$h_1 \xrightarrow{h_{root}} [\texttt{I\_type}]$ $h_2 \xrightarrow{h_{root}} \overline{[\texttt{I\_type}]}$ $\overline{h_1} \xrightarrow{h_{root}} \overline{[\texttt{I\_type}]} = h_2$

Par conséquent,

$$h_3 \xrightarrow{h_{root}} h_2 \wedge h_2 \equiv h_3 \xrightarrow{h_{root}} h_2$$

L'égalité précédente permet de remplacer dans le graphe, toute référence à l'horloge h_3, par l'horloge h_2. Des simplifications ultérieures seront occasionnées par cette substitution.

On obtient finalement,

$\mathcal{VH}$	$\mathcal{VHC}$
$\emptyset$	$h_1 \xrightarrow{h_{root}} [\texttt{I\_type}]$ $h_2 \xrightarrow{h_{root}} \overline{[\texttt{I\_type}]}$ $h_3 \equiv h_2$

Une fois normalisés, les nœuds de calcul d'horloge du GDC n'ont plus de raison d'être, et peuvent être supprimés du graphe d'application (voir figure II.13).

II.6.2 Simplifications

Des simplifications dans le graphe d'horloges peuvent être effectuées après le processus de normalisation. Un premier type de simplification a déjà été mis en évidence dans l'exemple précédent du codeur de type MPEG, il s'agit de la mise en équivalence des variables h_2 et h_3 (voir p.66). Cette propriété déduite de la normalisation de la variable non contrainte h_3 a permis de réduire le nombre d'horloges nécessaires à la synchronisation du système.

Un second type de simplification est également illustré sur cet exemple. Il repose sur la mise en équivalence des horloges d'utilisation et de définition d'un signal donné.

Dans le graphe de la figure II.14, par exemple, nous avons mis en évidence les nœuds **WHEN_1** et **WHEN_2**.

Les horloges respectives d'activation, h_1 et h_2, des tâches **IFC** et **MDMEC** ont été également reportées sur le graphe.

On rappellera qu'une tâche ne peut être activée que lorsque toutes ses entrées sont présentes à un instant donné. Ce qui suppose la synchronisation implicite de toutes les entrées d'une tâche. Dans le cas de **IFC**, la suite de ses instants d'activation est définie par l'horloge h_1. Le principe d'instantanéité suppose en outre, que les sorties d'une tâche sont toutes synchrones avec ses entrées. En d'autres termes, si l'horloge d'activation de la tâche **IFC** est h_1, alors, son entrée **I**, et sa sortie **Icoded** sont émises à cette même horloge.

Sachant que **Icoded** est émis à la fréquence h_1, l'expression:

$$\texttt{Icoded'} = \texttt{Icoded when } h_1$$

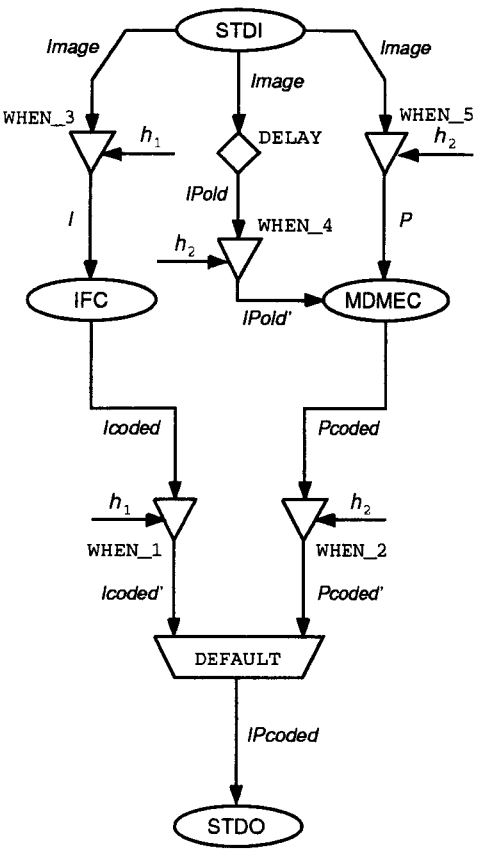


FIG. II.13 - *GDC débarrassé des nœuds de calcul d'horloge.*

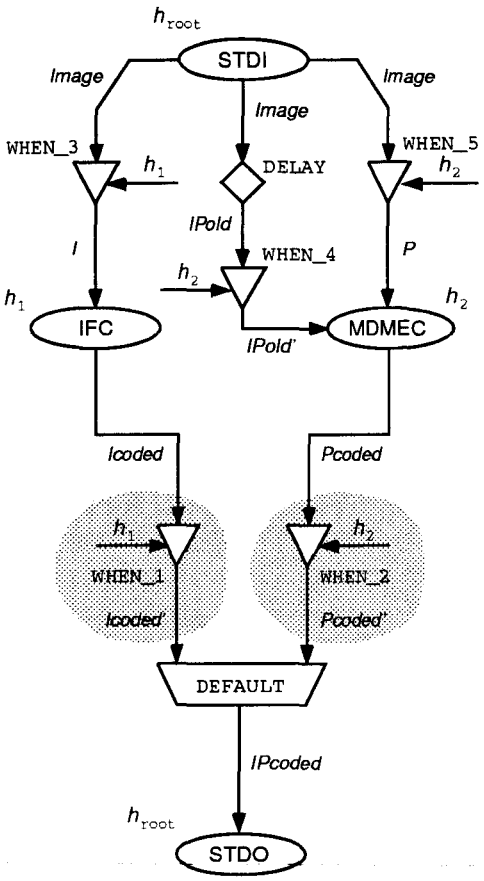


FIG. II.14 - *Graphe des Dépendances Conditionnées.*

définie par le nœud **WHEN_1**, où h_1 est l'horloge d'utilisation du signal **Icoded** pour définir **Icoded'**, est équivalente à:

Icoded' = **Icoded**

Le signal **Icoded** peut ainsi être substitué au signal **Icoded'**, et l'expression de définition de **Icoded'** (c'est-à-dire le nœud **WHEN_1**) supprimée dans la mesure où le signal **Icoded'** n'est plus utilisé.

Des conclusions identiques peuvent être tirées pour le nœud **WHEN_2**:

Pcoded' = **Pcoded when** h_2
$\equiv$ **Pcoded'** = **Pcoded**

Le graphe ainsi simplifié est donné en figure II.15.

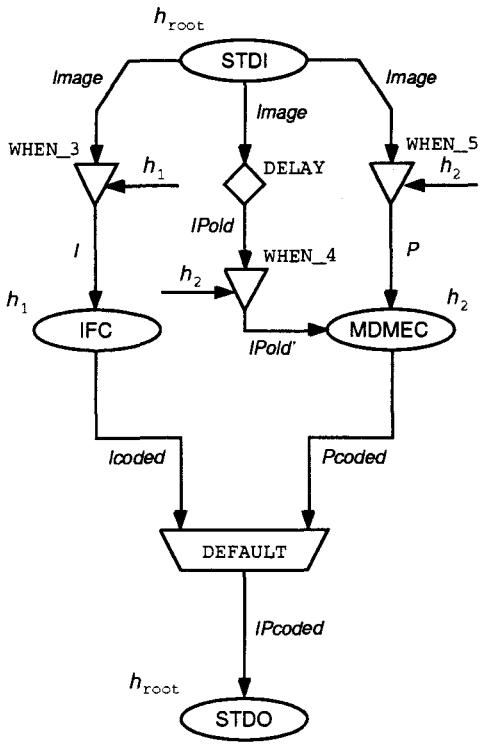


FIG. II.15 - *MPEG-2 version "simple profile".*

Chapitre III

Transformation des expressions

Un grand nombre d'expressions intermédiaires sont créées pour les besoins de la compilation en SIGNAL. Pour des propos d'implémentation il est important de chercher tous les moyens permettant de réduire la masse des signaux circulant dans le système. Le processus de transformation des dépendances de calcul que nous présentons dans ce chapitre répond à ce problème. Il détermine pour chaque entrée d'une tâche (sur un modèle *demand-driven*), une expression ne référençant que des signaux produits par des fonctions de calculs en amont.

Nous présentons dans la première partie de ce chapitre (SECTION III.1), le contexte formel dans lequel nous nous situons. Les notions d'horloge d'utilisation et d'expression de définition d'un signal sont en particulier abordées. Nous introduisons ensuite (SECTION III.2) un ensemble de notations qui permettront de décrire le processus de transformation des expressions en SECTION III.3.

III.1 Expressions de définition de signaux

La clarification des relations temporelles par le calcul d'horloge entraîne une décomposition en expressions intermédiaires des opérations de traitement du signal. Il s'ensuit qu'un grand nombre de signaux intermédiaires apparaissent dans le graphe des dépendances conditionnées.

L'agrégation de signaux n'est pas nouvelle puisqu'elle a déjà été présentée comme optimisation dans [MCL92, Ché91]. Il s'agissait en effet de réduire l'espace mémoire requis par l'inflation du nombre des signaux. Par exemple,

```
x := y when c
```

On dénote par hd_x et hd_y l'horloge de disponibilité respective des signaux **x** et **y**. La sémantique de l'opération d'extraction de signal induit la relation temporelle suivante,

$$hd_x \leq hd_y$$

Les zones de représentation de **x** et **y** peuvent être confondues: $hd_x = hu_x^y$, où hu_x^y est l'horloge d'utilisation du signal y pour définir x. hu_x^y est valide lorsque hd_y est elle-même valide, et que le signal booléen **c** est défini avec la valeur *vrai*.

La réduction des expressions intermédiaires telle qu'elle a été réalisée jusqu'à présent n'est cependant effectuée que partiellement dans la mesure où les nœuds de mémorisation sont considérés comme des tâches. Prenons en effet un système comme celui de la figure III.1, la présence du nœud de mémorisation donnerait deux expressions réduites.

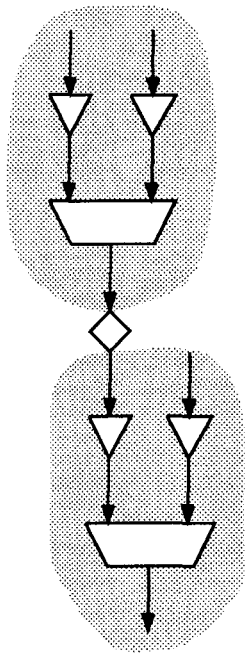


FIG. III.1 - *Réduction d'expression sur nœud mémoire.*

III.1.1 Horloge d'utilisation

La complexité des relations temporelles pouvant exister dans un programme temps-réel fait que les dépendances de données ne sont pas forcément valides à tout instant. Par exemple, si on considère l'expression `x = y when b`, les instants de validité dans le temps de la dépendance existant entre le signal `x` et le signal `y` sont liés à la valeur portée par le signal booléen `b`. Ainsi, cette dépendance de donnée n'est valide qu'aux instants pour lesquels les signaux `y` et `b` sont définis, et uniquement lorsque le booléen `b` porte la valeur *vrai*. Ce qui peut se traduire en termes d'horloges par:

$$hd_x = hd_y \wedge [\texttt{b}]$$

Autrement dit, le signal y est utilisé pour définir le signal x à l'horloge $hd_y \wedge [\texttt{b}]$. On parle d'*horloge d'utilisation* pour définir un signal.

Plus généralement, différents signaux peuvent concourir à la définition d'un flot. Un résultat important du calcul d'horloges est de déterminer de manière exclusive les différents instants auxquels chaque signal dans une expression participe à la définition d'un nouveau flot. Il s'agit ici de garantir le déterminisme des traitements. Ainsi par exemple, l'opération d'entrelacement de signaux:

$$\texttt{x = y}_1 \texttt{ default y}_2$$

sera résolue par une expression du type:

$$\texttt{x = (y}_1 \texttt{ when } h_{y_1} \texttt{) default (y}_2 \texttt{ when (} \overline{h_{y_1}} \wedge h_{y_2} \texttt{))}$$

C'est-à-dire que le signal $\texttt{y}_1$ participe à la définition du signal `x` à l'horloge h_{y_1}, et que le signal $\texttt{y}_2$ est utilisé pour définir le signal `x` à l'horloge $\overline{h_{y_1}} \wedge h_{y_2}$.

Dans cet exemple, les horloges respectives d'utilisation des signaux $\texttt{y}_1$ et $\texttt{y}_2$ pour définir `x` sont:

$$\begin{aligned} hu_x^{y_1} &= h_{y_1} \\ hu_x^{y_2} &= \overline{h_{y_1}} \wedge h_{y_2} \end{aligned}$$

III.1.2 Expression de définition de signaux

De manière générale dans le système des expressions de définition des flots, un signal **x** est défini par une expression du type:

$$\mathbf{x} := (s_1 \texttt{ when } hu_x^{s_1}) \texttt{ default } \ldots \texttt{ default } (s_n \texttt{ when } hu_x^{s_n}) \texttt{ default } (x\$1 \texttt{ when } h_{x\$})$$

où,

- les $\mathbf{s}_i$'s peuvent être pour une part des signaux de sortie de tâches, et pour l'autre part des signaux intermédiaires eux-mêmes respectivement définis par des expressions,
- $hu_x^{s_i}$ est appelée *horloge d'utilisation* du signal $\mathbf{s}_i$ pour définir **x**,
- les horloges d'utilisation s'excluent mutuellement,
- $h_{x\$}$ peut être nulle, et est exclusive de toutes les $hu_x^{s_i}$.

Les expressions $\mathbf{s}_j$'s définissant des signaux intermédiaires sont construites sur un ensemble de signaux $(\sigma_1, \ldots, \sigma_n)$; chacun de ces signaux peut apparaître dans l'expression de définition de plusieurs $\mathbf{s}_j$'s.

III.2 Formalisme

L'objectif visé par le processus de transformation des dépendances de calcul est de donner pour chaque entrée de tache, une expression ne référençant que des flots produits en amont par d'autres fonctions de calcul.

Ainsi, soit **T** une tâche, ou fonction de calcul, comportant a entrées, on dénote par $\mathbf{e}_\alpha$ l'expression de définition d'une des a entrées de la tâche **T** (cf. figure III.2).

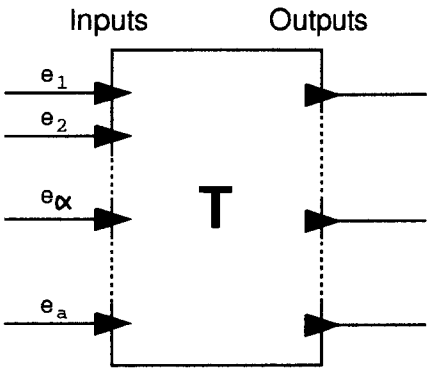


FIG. III.2 - *Tâche* **T**.

Les $\mathbf{e}_\alpha$ sont synchrones, et on a déjà vu que de manière générale un signal $\mathbf{e}_\alpha$ est entièrement défini par une expression du type:

$$\mathbf{e}_\alpha := (\mathbf{s}_1 \texttt{ when } hu_{e_\alpha}^{s_1}) \texttt{ default } \ldots \texttt{ default } (\mathbf{s}_n \texttt{ when } hu_{e_\alpha}^{s_n}) \texttt{ default } (\mathbf{e}_\alpha\$ \texttt{ when } h_{e_\alpha\$})$$

On définit par par $\mathcal{E}_{e_\alpha}$ l'ensemble des signaux référencés dans l'expression $\mathbf{e}_\alpha$. On a ainsi,

$$\mathcal{E}_{e_\alpha} = \{ \mathbf{s}_1, \ldots, \mathbf{s}_n \}$$

Tout signal $\mathbf{s}_j$ peut apparaître dans la définition de différents signaux d'entrée (différents $\mathbf{e}_\alpha$'s). On a donc,

$$\bigcap_{\alpha=1}^{a} \mathcal{E}_{e_\alpha} \text{ peut être non vide}$$

On dénote par $\mathcal{T}_{s_i}$ la tâche émettrice du signal $\mathbf{s}_i$. Et par $\mathcal{E}_T$, $\mathcal{S}_T$ l'ensemble respectif des entrées, et des sorties d'une tâche T donnée.

On réfère également par $\mathcal{S}$ l'ensemble fini des signaux produits en sortie de tâche, par $\mathcal{E}$ l'ensemble fini des signaux en entrée de tâche, et par $\mathcal{I}$ l'ensemble fini des signaux intermédiaires pour la totalité du système. On a,

$$\mathcal{S} \cup \mathcal{E} \cup \mathcal{I} = \mathcal{SIG}$$

En d'autres termes, l'ensemble $\mathcal{S}$ représente l'union des sous-ensembles $\mathcal{S}_{T_\theta}$ pour toutes les tâches T_θ du système. L'intersection des $\mathcal{S}_{T_\theta}$'s est vide (règle d'assignation unique). Et symétriquement, l'ensemble $\mathcal{E}$ représente l'union des sous-ensembles $\mathcal{E}_{T_\theta}$ pour toutes les tâches T_θ du système. L'intersection des $\mathcal{E}_{T_\theta}$'s peut être non-vide.

Dans notre exemple du codeur MPEG dit *"simple profile"*, ces différents ensembles contiennent les listes suivantes d'éléments:

- $\mathcal{SIG}$ = { `Image, I, P, IPold, IPold', Icoded, Pcoded, IPcoded` }
- $\mathcal{S}$ = { `Image, Icoded, Pcoded` }
- $\mathcal{E}$ = { `I, P, IPold', IPcoded` }
- $\mathcal{I}$ = { `IPold` }

Trois signaux de sortie sont présents dans le système, ils sont respectivement émis par,

- $\mathcal{T}_{Image}$ = `STDI`
- $\mathcal{T}_{Icoded}$ = `IFC`
- $\mathcal{T}_{Pcoded}$ = `MDMEC`

Les expressions définissant les dépendances de calcul sont données par le système suivant:

```
1- I       = Image when h1
2- P       = Image when h2
3- IPold   = Image$1
4- IPold'  = IPold when h2
5- IPcoded = Icoded default Pcoded
```

Les expressions définissant des entrées de tâches (`I, P, IPold', IPcoded`) sont données par les expressions `1, 2, 4,` et `5`.

Les expressions de définition des entrées étant relativement complexes à appréhender lorsqu'elles ont été transformées pour ne plus référencer de signaux intermédiaires, nous commencerons dans une première partie par commenter un cas de figure plus simple: $\mathcal{SIG} = \mathcal{E} \cup \mathcal{S}$.

III.2.1 Absence de signaux intermédiaires

Dans ce cas de figure, l'ensemble $\mathcal{SIG}$ ne comporte plus de signaux intermédiaires. C'est-à-dire que chaque $\mathbf{s}_i$ référencé dans une expression $\mathbf{e}_\alpha$ peut être associé à un signal produit en sortie de tâche. Et chaque expression $\mathbf{e}_\alpha$ à une entrée de tâche.

Le processus de transformation des dépendances de calcul vise à obtenir des expressions de ce type pour tout élément de $\mathcal{E}$.

Un signal de sortie $\mathbf{s}_i$ référencé dans une expression $\mathbf{e}_\alpha$ définissant une entrée est attendu dans le cas général à une horloge inférieure à celle de son horloge de définition (c'est-à-dire à une horloge inférieure à celle à laquelle il est disponible).

Dans l'expression générale de définition d'une entrée,

$\mathbf{e}_\alpha$:= ($\mathbf{s}_1$ **when** $hu^{s_1}_{e_\alpha}$) **default** ... **default** ($\mathbf{s}_n$ **when** $hu^{s_n}_{e_\alpha}$) **default** ($\mathbf{e}_\alpha$**$** **when** $h_{e_\alpha\$}$)

le signal $\mathbf{s}_i$ est utilisé à l'horloge $hu^{s_i}_{e_\alpha}$ pour définir $\mathbf{e}_\alpha$.

En d'autres termes, le signal $\mathbf{s}_i$ produit par $\mathcal{T}_{s_i}$ doit être transmis à la tâche référençant l'entrée e_α lorsque $hu^{s_i}_{e_\alpha}$ est valide.

Par exemple, dans l'expression de définition du signal `I` de notre codeur MPEG dit *"simple profile"*, le signal **Image** produit à l'horloge h_{root} par la tâche **STDI** est utilisé pour définir le signal `I` à l'horloge h_1. Ce signal **Image** est ainsi transmis à la fonction **IFC** lorsque l'horloge h_1 est valide.

Un signal $\mathbf{s}_i$ donné peut apparaître dans l'expression de définition de différentes entrées d'une même tâche (différents $\mathbf{e}_\alpha$'s). On a alors,

$$\mathcal{D}^{s_i}_T = \bigvee_{\substack{\alpha=1 \\ s_i \in \mathcal{E}_{e_\alpha}}}^{a} hu^{s_i}_{e_\alpha}$$

L'horloge $\mathcal{D}^{s_i}_T$ détermine la fréquence de la dépendance $\mathbf{s}_i$ existant entre la tâche amont émettrice du signal $\mathbf{s}_i$ et la tâche `T` qui référence ce flot dans une ou plusieurs de ses expressions d'entrée.

Afin de clarifier nos propos, nous avons représenté sur le schéma de la figure III.3, les différentes relations temporelles que nous venons de déterminer.

Dépendances de calcul

Une tâche `T` donnée comporte a entrées dénotées $\mathbf{e}_1, \ldots, \mathbf{e}_a$ (figure III.3). Chaque entrée $\mathbf{e}_\alpha$ est définie par un ensemble de signaux dénotés $\mathbf{s}^\alpha_\nu$, et éventuellement une mémoire $\mathbf{e}_\alpha$**$**. La dernière valeur affectée à l'expression $\mathbf{e}_\alpha$, lorsque celle-ci est référencée dans son expression de définition (c'est-à-dire $h_{e_\alpha\$}$ n'est pas nulle), est supposée être maintenue localement. Par conséquent, la mémoire ne consommant pas de signaux extérieurs à la tâche, elle ne crée pas de dépendance.

Dans ce schéma nous avons étudié le cas d'un signal `s`, produit par une tâche $\mathbf{T}_s$, et qui est utilisé pour définir une ou plusieurs entrées $\mathbf{e}_\alpha$'s de `T`.

Le signal `s` est par exemple référencé dans l'expression définissant l'entrée $\mathbf{e}_1$, dans celle de $\mathbf{e}_\alpha$, et dans l'expression définissant l'entrée $\mathbf{e}_a$. Les instants respectifs de validité de ces dépendances sont dénotés respectivement $hu^s_{e_1}$, $hu^s_{e_\alpha}$, et $hu^s_{e_a}$. L'horloge globale $\mathcal{D}^s_T$, de la dépendance `s` entre les tâches $\mathbf{T}_s$ et `T` correspond à la somme de ces horloges d'utilisation.

Les horloges $hu^s_{e_1}$, $hu^s_{e_\alpha}$, $hu^s_{e_a}$, d'utilisation du signal `s` pour définir les entrées de la tâche `T` ne sont pas systématiquement valides aux mêmes instants. La seule chose qui puisse être affirmée c'est que lorsque $\mathcal{D}^s_T$ est valide, au moins l'une de ces horloges l'est également.

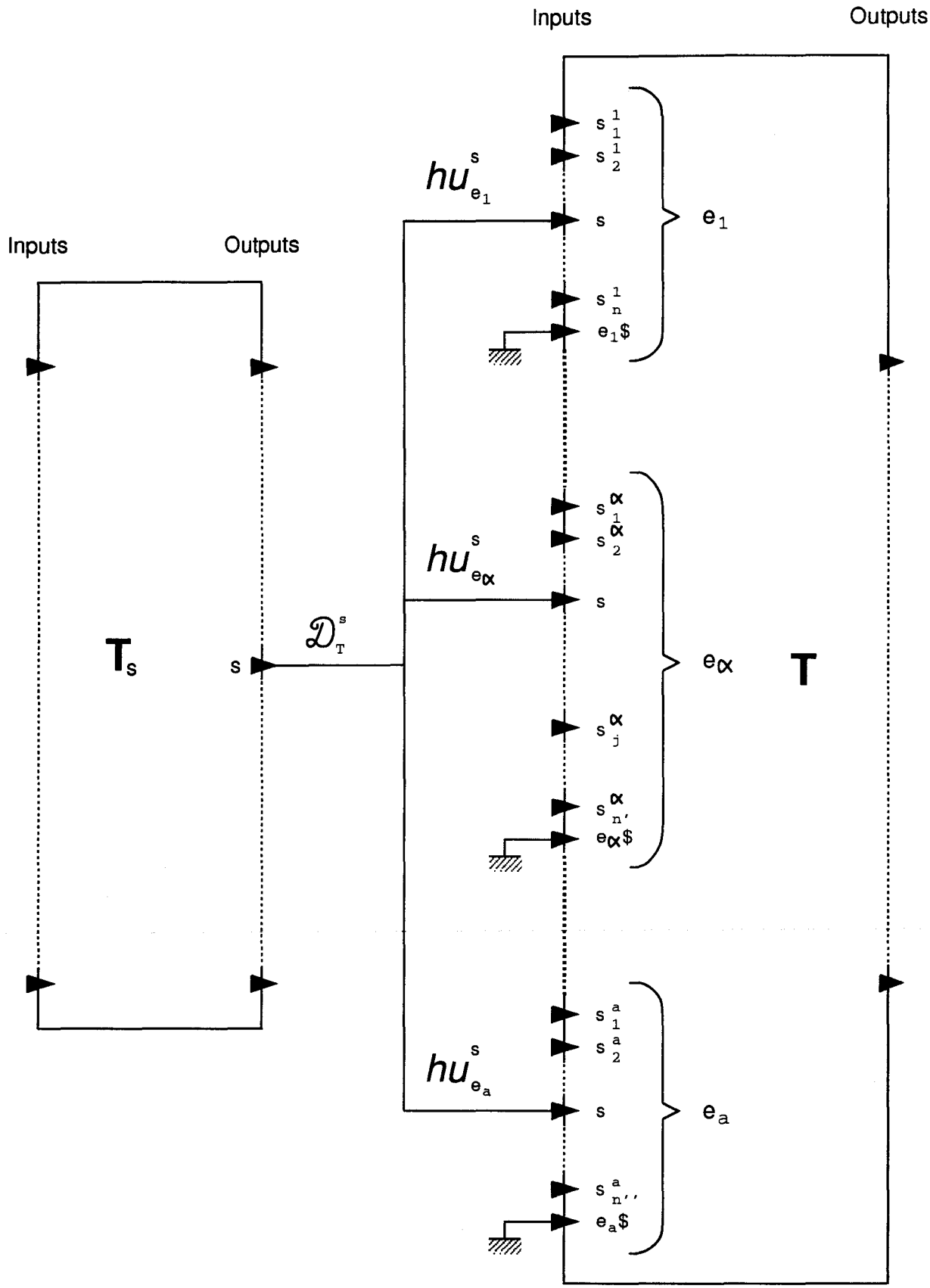


FIG. III.3 - *Dépendance* s *entre une tâche* T_s *amont, et une tâche* T *aval.*

Généralisation à toutes les entrées d'une tâche

Pour une tâche donnée T comportant a entrées dénotées $\mathtt{e}_1, \ldots, \mathtt{e}_\alpha, \ldots, \mathtt{e}_a$ (voir figure III.4), on a appelé $\mathcal{E}_{e_\alpha}$ l'ensemble fini des signaux référencés dans l'expression définissant l'entrée e_α. L'ensemble des dépendances amont d'une tâche T se compose donc de la réunion de tous les $\mathcal{E}_{e_\alpha}$'s pour chaque entrée $\mathtt{e}_\alpha$ de la tâche.

$$\bigcup_{\alpha=1}^{n} \mathcal{E}_{e_\alpha} = \{ \mathtt{S}_1, \mathtt{S}_2, \ldots, \mathtt{S}_\sigma, \ldots, \mathtt{S}_\Sigma \}$$

Chacun de ces signaux ($\{ \mathtt{S}_1, \mathtt{S}_2, \ldots, \mathtt{S}_\sigma, \ldots, \mathtt{S}_\Sigma \}$) dénote un lien de dépendance entre la tâche productrice du signal et la tâche T. Nous avons déterminé dans le paragraphe précédent, les instants de validité d'une dépendance $\mathtt{s}_j$ entre sa tâche productrice et une de ses tâches consommatrice (T). Cette horloge a été dénommée $\mathcal{D}_T^{s_j}$.

Ce résultat peut être généralisé pour tous les signaux référencés dans des expressions d'entrée d'une tache:

$$\forall\, s_\sigma \in \bigcup_{\alpha=1}^{n} \mathcal{E}_{e_\alpha},\ \mathcal{D}_T^{s_\sigma} = \bigvee_{\substack{\alpha=1 \\ s_\sigma \in \mathcal{E}_{e_\alpha}}}^{a} hu_{e_\alpha}^{s_\sigma}$$

On peut donc déterminer pour l'ensemble des signaux référencés dans les expressions définissant les entrées d'une tâche T, les horloges auxquelles ceux-ci doivent être transmis, lorsqu'une expression sur signaux de sorties a pu être déterminée pour chaque entrée $\mathtt{e}_\alpha$.

III.3 Règles de transformation

Nous donnons ici les mécanismes généraux permettant d'exprimer toute entrée de tâche en fonction de signaux de sortie. Avant d'en donner une description formelle (SECTION III.3.1), nous introduisons le principe de substitution tel qu'il est opéré. Nous abordons ensuite le point plus délicat de la récursion temporelle (SECTION III.3.2) en illustrant le bouclage du processus qu'il peut induire. Les solutions apportées à ce problème dans la littérature sont ensuite présentées et leur application au processus de transformation est explicitée.

Dans le cas général, les $\mathtt{s}_i$'s appartiennent pour partie à l'ensemble $\mathcal{S}$, et pour l'autre à l'ensemble $\mathcal{I}$.

Afin de faire disparaître les signaux intermédiaires nous procédons par substitutions successives en remplaçant chaque référence à un signal intermédiaire par son expression de définition. On procède ainsi récursivement jusqu'à ce que tous les signaux référencés en amont soient des flots de sortie.

Afin de simplifier l'écriture, nous introduisons une nouvelle notation:

$$\mathtt{e}_\alpha := \left[(\mathtt{E}_1 \texttt{ when } hu_{e_\alpha}^{E_1}) \texttt{ default } \ldots \texttt{ default } (\mathtt{E}_n \texttt{ when } hu_{e_\alpha}^{E_n}) \right]_{hu_{e_\alpha}\$}$$

Où $hu_{e_\alpha}\$$ représente l'horloge à laquelle la valeur retardée de l'expression est utilisée pour définir le signal.

Considérons une entrée $\mathtt{e}_\alpha$ quelconque,

$$\mathtt{e}_\alpha := \left[(\mathtt{s}_1 \texttt{ when } hu_{e_\alpha}^{s_1}) \texttt{ default } \ldots \texttt{ default } (\mathtt{s}_i \texttt{ when } hu_{e_\alpha}^{s_i}) \texttt{ default } \ldots \texttt{ default } (\mathtt{s}_n \texttt{ when } hu_{e_\alpha}^{s_n}) \right]_{hu_{e_\alpha}\$}$$

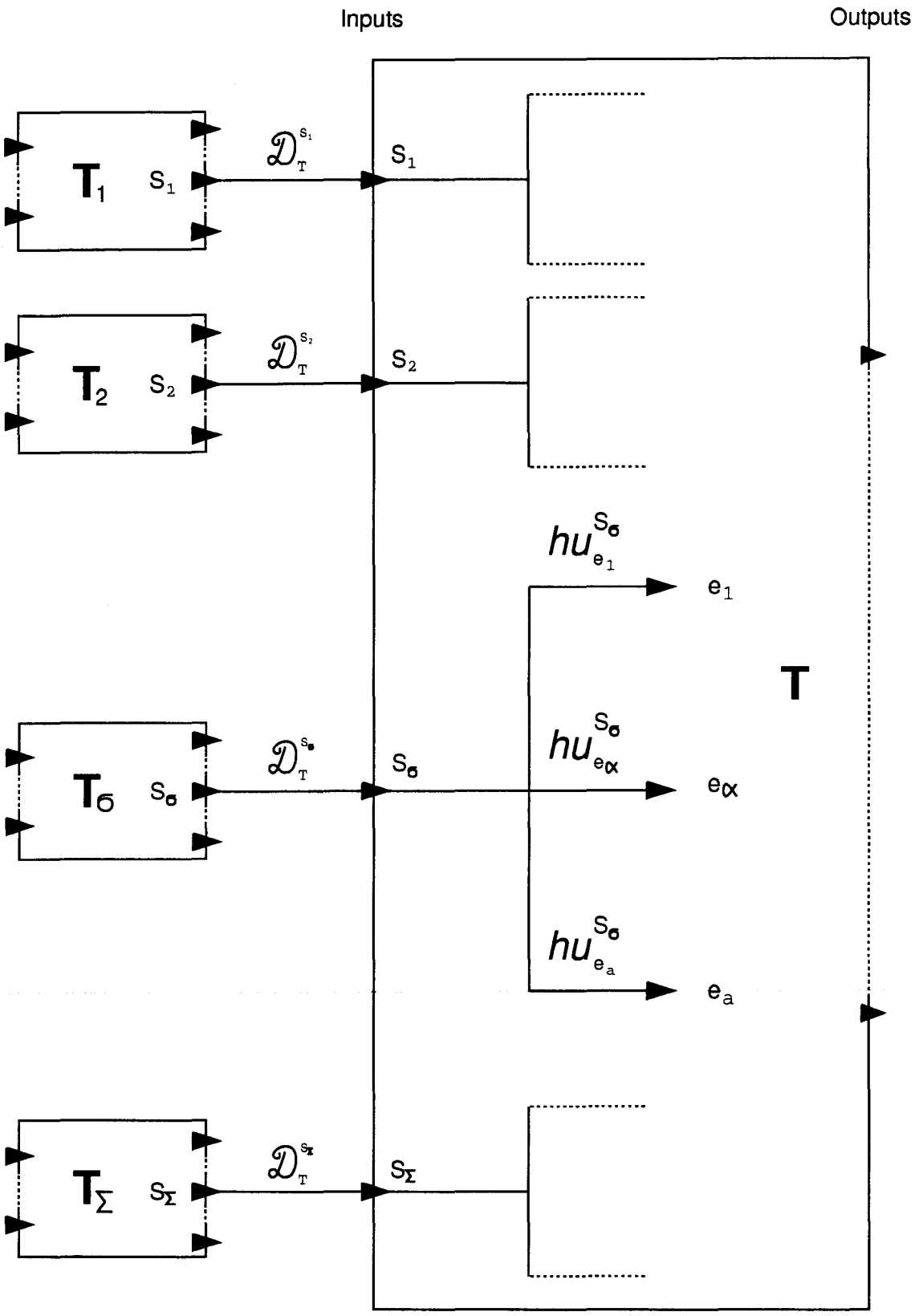


FIG. III.4 - *Ensemble des dépendances en entrée d'une tâche donnée* T.

Où, $\mathbf{s}_i$ est un signal intermédiaire défini par l'expression suivante:

$$\mathbf{s}_i := \left[(\mathbf{s'}_1 \texttt{ when } hu_{s_i}^{s'_1}) \texttt{ default } \ldots \texttt{ default } (\mathbf{s'}_{n'} \texttt{ when } hu_{s_i}^{s'_{n'}})\right]_{hu_{s_i}\$}$$

L'expression de définition du signal intermédiaire $\mathbf{s}_i$ peut être ainsi être substituée à sa référence dans l'expression définissant $\mathbf{e}_\alpha$.

$$\begin{aligned}\mathbf{e}_\alpha := \; & \left[(\mathbf{s}_1 \texttt{ when } hu_{e_\alpha}^{s_1}) \texttt{ default } \ldots \texttt{ default }\right. \\ & \left(\left[(\mathbf{s'}_1 \texttt{ when } hu_{s_i}^{s'_1}) \texttt{ default } \ldots \texttt{ default } (\mathbf{s'}_{n'} \texttt{ when } hu_{s_i}^{s'_{n'}})\right]_{hu_{s_i}\$} \texttt{ when } hu_{e_\alpha}^{s_1}\right) \\ & \left.\texttt{default } \ldots \texttt{ default } (\mathbf{s}_n \texttt{ when } hu_{e_\alpha}^{s_n})\right]_{hu_{e_\alpha}\$}\end{aligned}$$

À chaque signal intermédiaire référencé dans l'expression de définition de l'entrée $\mathbf{e}_\alpha$ on peut ainsi lui substituer son expression de définition.

Le développement de l'expression $\mathbf{s}_i$ dans $\mathbf{e}_\alpha$ fait apparaître un nouvel ensemble de signaux qui sera ajouté à l'ensemble des signaux déjà référencés dans l'expression de définition de l'entrée $\mathbf{e}_\alpha$.

On poursuit ainsi récursivement le processus de substitution jusqu'à ce que l'ensemble des signaux référencés dans $\mathbf{e}_\alpha$ corresponde à des flots appartenant à l'ensemble $\mathcal{S}$ (cf. figure III.5).

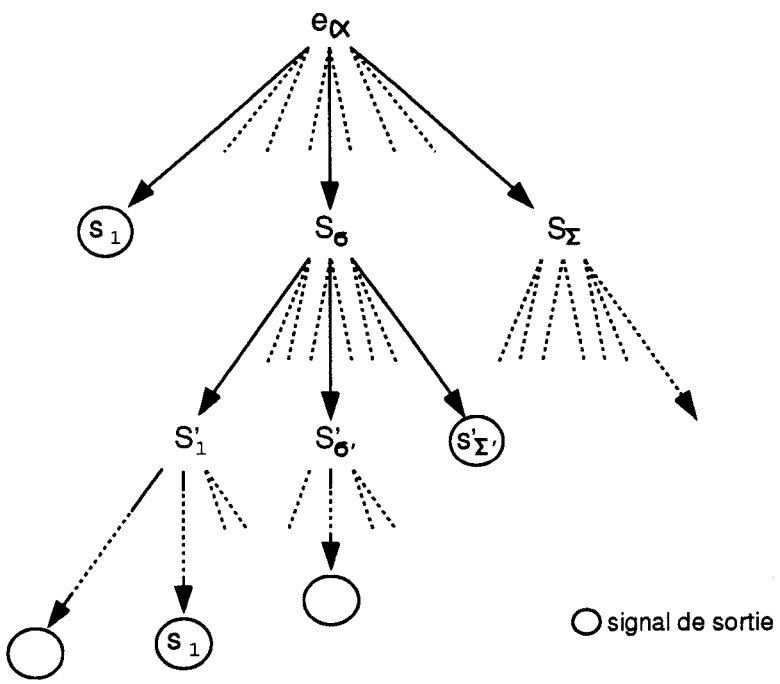


FIG. III.5 - *Détermination des flots référencés par l'entrée* $\mathbf{e}_\alpha$.

III.3.1 Processus de transformation

De manière générale, on dénote par $\mathcal{ES}_{e_\alpha}$, et $\mathcal{EI}_{e_\alpha}$ les sous-ensembles finis respectifs des signaux de sortie et des signaux intermédiaires référencés dans l'expression de $\mathbf{e}_\alpha$.

$$\mathcal{E}_{e_\alpha} = \mathcal{ES}_{e_\alpha} \cup \mathcal{EI}_{e_\alpha}$$

Une entrée $\mathbf{e}_\alpha$, définit un système $\{ \mathbf{e}_\alpha, \mathcal{ES}_{e_\alpha}, \mathcal{EI}_{e_\alpha} \}$ qui va subir un ensemble de transformations du type:

$$\frac{\{ \mathbf{e}_\alpha, \mathcal{ES}_{e_\alpha}, \mathcal{EI}_{e_\alpha} \}}{\mathcal{EI}_{e_\alpha} \neq \emptyset, \mathbf{e}_s \in \mathcal{EI}_{e_\alpha}} \longrightarrow \{ \mathbf{e'}_\alpha, \mathcal{ES'}_{e_\alpha}, \mathcal{EI'}_{e_\alpha} \}$$

Où,

- $\mathbf{e'}_\alpha$ est une expression équivalente à $\mathbf{e}_\alpha$, dans laquelle toute référence au signal intermédiaire $\mathbf{e}_s$ est remplacée par l'expression de définition de ce dernier.
- $\mathcal{ES'}_{e_\alpha} = \mathcal{ES}_{e_\alpha} \bigcup \mathcal{ES}_{e_s}$. À l'ensemble fini des signaux de sortie référencés dans l'expression $\mathbf{e}_\alpha$, on ajoute ceux qui apparaissent dans $\mathbf{e}_s$ et qui ne sont pas déjà référencés dans $\mathcal{ES}_{e_\alpha}$.
- $\mathcal{EI'}_{e_\alpha} = \{ \mathcal{EI}_{e_\alpha} - \{ \mathbf{e}_s \} \} \bigcup \mathcal{EI}_{e_s}$. La référence au signal $\mathbf{e}_s$ est ôtée de $\mathcal{EI}_{e_\alpha}$, et on ajoute à cet ensemble, la liste des signaux intermédiaires qui apparaissent dans $\mathbf{e}_s$ et qui ne sont pas déjà référencés dans $\mathcal{EI}_{e_\alpha}$.

Le processus de transformation, ou ré-écriture du système $\{ \mathbf{e}_\alpha, \mathcal{ES}_{e_\alpha}, \mathcal{EI}_{e_\alpha} \}$ s'achève lorsque $\mathcal{EI}_{e_\alpha} = \emptyset$. Autrement dit, tous les signaux référencés dans l'expression de définition de l'entrée $\mathbf{e}_\alpha$ sont des flots produits par des fonctions de calcul du programme. Chacun de ces flots définit un lien de dépendance entre sa tâche productrice et la tâche $\mathtt{T}_{e_\alpha}$ qui est une de ses consommatrices.

III.3.2 Récursion temporelle

Les seules formes récursives qui peuvent apparaître dans le GDC produit par SIGNAL sont les récursions temporelles. Elles peuvent ainsi conduire à un *"bouclage"* du processus de ré-écriture.

Nous illustrons nos propos sur un petit exemple représenté en figure III.6. Le graphe originel du programme et le GDC produit sont représentés côte-à-côte.

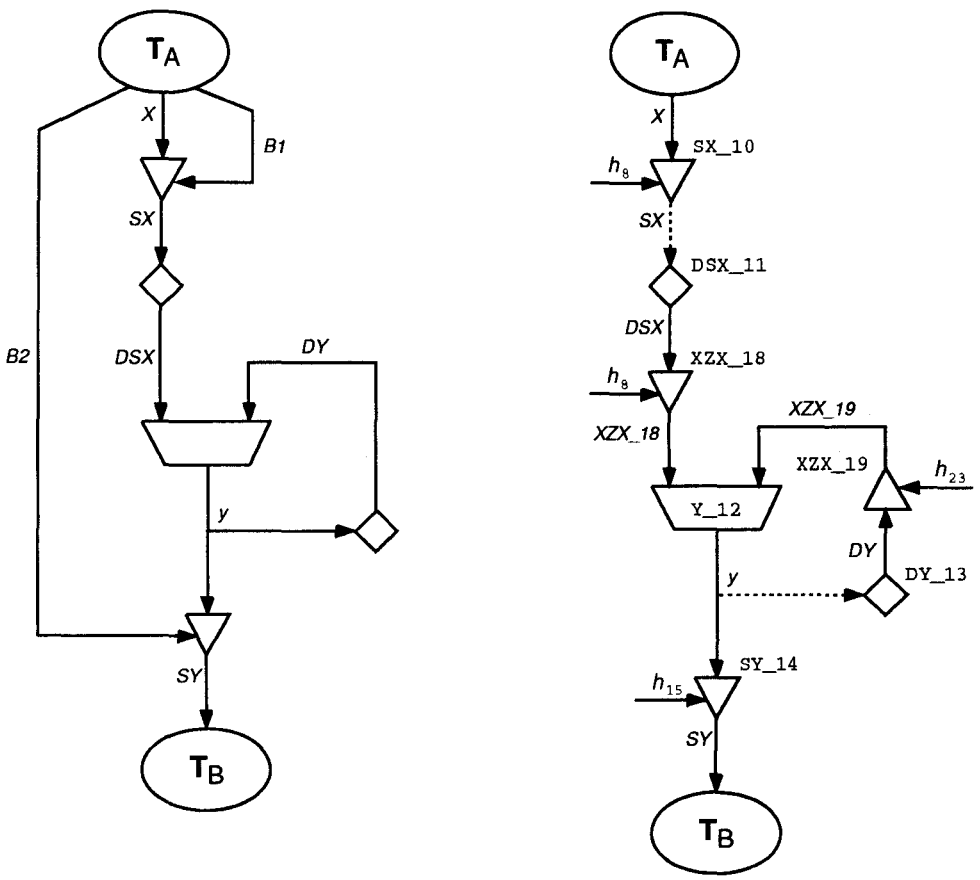
Si nous interprétions sans plus de précaution le système de dépendances induit par le GDC, nous obtiendrions un ensemble d'équations de la forme:

```
1-  SY      =  Y when h15
2-  Y       =  (XZX_18 when h8) default (XZX_19 when h23)
3-  XZX_18  =  DSX when h8
4-  DSX     =  SX$ when h8
5-  SX      =  X when h8
6-  XZX_19  =  DY when h23
7-  DY      =  Y$ when h21
```

Dans cet exemple, ne figurent que deux tâches de calcul: $\mathtt{T}_A$ et $\mathtt{T}_B$. L'ensemble $\mathcal{S}$ des signaux produits par le système comporte trois éléments: $\mathcal{S} = \{$ X, B1, B2 $\}$. L'ensemble des signaux d'entrée du système est quant à lui réduit à un singleton: $\mathcal{E} = \{$ SY $\}$.

Le processus de ré-écriture va donc se réduire à la transformation de l'expression de définition de l'entrée SY.

```
a:          SY  =  Y when h15
b:  2 → a   SY  =  ((XZX_18 when h8) default (XZX_19 when h23)) when h15
c:  3 → b   SY  =  ((DSX when h8) default (XZX_19 when h23)) when h15
d:  4 → c   SY  =  ((SX$ when h8) default (XZX_19 when h23)) when h15
e:  5 → d   SY  =  (((X when h8)$ when h8) default (XZX_19 when h23)) when h15
f:  6 → e   SY  =  (((X when h8)$ when h8) default (DY when h23)) when h15
g:  7 → f   SY  =  ((X when h8)$ when h8) default ((Y$ when h8) when h23)) when h15
```



(1) (2)

FIG. III.6 - *Exemple d'application. (1) Graphe du programme. (2) Graphe des dépendances conditionnées.*

L'expression (g) nous reporte une nouvelle fois à la substitution du signal Y, et nous condamne donc à recommencer éternellement le même processus de transformation. Le bouclage du processus est provoqué par l'utilisation récursive de la valeur retardée du signal Y. Ce problème a déjà été résolu par le passé, et nous allons en donner une interprétation dans notre processus de transformation.

III.3.3 Dépendances entre un signal et son signal retardé

Des optimisations par analyse des dépendances entre un signal et son signal retardé ont déjà été explicitées dans la littérature (par exemple dans [Ché91]). Elles apportent des solutions à notre problème.

Nous nous contenterons d'évoquer succinctement les grands principes de l'optimisation proposée, et nous renvoyons le lecteur intéressé aux documents concernés pour en obtenir tous les développements.

"Une action mémoire peut être vue comme deux sommets distincts, un sans prédécesseur et un autre sans successeur, si bien qu'un graphe ne possédant de cycles qu'au travers de ses nœuds mémoire peut être considéré comme acyclique" [Sor94].

Il est possible de déterminer par simple parcours du graphe, le ou les chemins existant entre **x$** le signal retardé et le signal **x**. Pour chaque chemin, **x** est défini par **x$** aux instants définis par le produit des horloges de ce chemin. L'horloge globale d'utilisation du signal **x$** pour définir **x** est donnée par la somme de ces produits.

C'est ainsi que dans notre exemple, il existe un chemin entre Y$, dénoté DY, et Y (voir figure III.7).

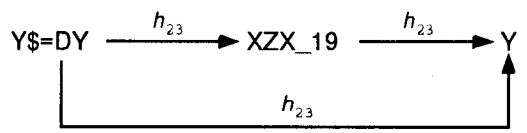


FIG. III.7 - *Chemin entre* Y$ *et* Y.

On a donc une expression de Y qui est en fait égale à:

$$\texttt{Y} = \texttt{XZX\_18 default (Y\$ when } h_{23}) = [\texttt{XZX\_18}]_{h_{23}}$$

Cette substitution opérée, le chemin menant de DY, c'est-à-dire Y$, à Y peut être coupé:

$$\begin{array}{lllll} 1\text{-} & \texttt{SY} & = & \texttt{Y when } h_{15} \\ 2\text{-} & \texttt{Y} & = & [\,\texttt{XZX\_18 when } h_8\,]_{h_{23}} \\ 3\text{-} & \texttt{XZX\_18} & = & \texttt{DSX when } h_8 \\ 4\text{-} & \texttt{DSX} & = & \texttt{SX\$ when } h_8 \\ 5\text{-} & \texttt{SX} & = & \texttt{X when } h_8 \end{array}$$

Si l'on reprend le processus de substitution avec la nouvelle expression de Y qui vient d'être déterminée, on obtient:

$$\begin{array}{llllll} \text{a:} & & \texttt{SY} & = & \texttt{Y when } h_{15} \\ \text{b:} & 2 \rightarrow \text{a} & \texttt{SY} & = & [\,\texttt{XZX\_18 when } h_8\,]_{h_{23}} \texttt{ when } h_{15} \\ \text{c:} & 3 \rightarrow \text{b} & \texttt{SY} & = & [\,\texttt{DSX when } h_8\,]_{h_{23}} \texttt{ when } h_{15} \\ \text{d:} & 4 \rightarrow \text{c} & \texttt{SY} & = & [\,\texttt{SX\$ when } h_8\,]_{h_{23}} \texttt{ when } h_{15} \\ \text{e:} & 5 \rightarrow \text{d} & \texttt{SY} & = & [\,(\,\texttt{X when } h_8\,)\texttt{\$ when } h_8\,]_{h_{23}} \texttt{ when } h_{15} \end{array}$$

Le résultat obtenu est bien celui que nous attendions. Une expression de l'entrée **SY** ne contenant que des signaux de sortie a pu être dérivée.

III.3.4 Plusieurs chemins

Prenons un système plus complexe afin d'illustrer la méthode de substitution opérée dans le cas de récursions temporelles définissant plusieurs chemins (cf. figure III.8).

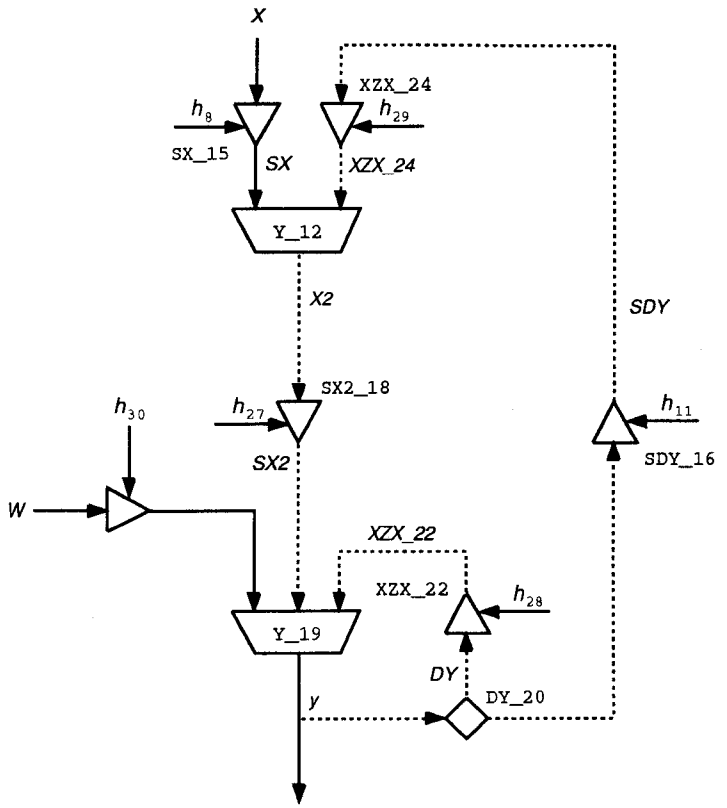


FIG. III.8 - *Exemple de récursions temporelles.*

Dans cet exemple, le signal **Y** est *"doublement"* défini par sa valeur retardée. Il existe en effet deux chemins de **Y$** à **Y**. Le premier définit une dépendance à h_{28}, et le second à $(h_{11} \wedge h_{29} \wedge h_{27})$. L'expression de **Y** sera donc:

$$\mathtt{Y} = [\ (\mathtt{W\ when}\ h_{30})\ \mathtt{default}\ (\mathtt{X2\ when}\ h_{27})\ ]_{h_{28} \vee (h_{11} \wedge h_{29} \wedge h_{27})}$$

On peut représenter ce système par le couple d'équations:

$$\begin{array}{lllll} 1\text{-} & \mathtt{Y} & = & [\ (\mathtt{W\ when}\ h_{30})\ \mathtt{default}\ (\mathtt{X2\ when}\ h_{27})\ ]_{h_{28} \vee (h_{11} \wedge h_{29} \wedge h_{27})} \\ 2\text{-} & \mathtt{X2} & = & (\mathtt{X\ when}\ h_8)\ \mathtt{default}\ (\mathtt{Y\$\ when}\ (h_{29} \wedge h_{11})) \end{array}$$

L'expression entre crochets de l'équation définissant le signal **Y** représente la fonction de calcul de la mémoire **Y$**. Toute référence à la valeur **Y$** dans le système peut être simplement remplacée par cette expression retardée. Ainsi dans notre système:

$$\mathtt{X2} = (\mathtt{X\ when}\ h_8)\ \mathtt{default}\ ([(\mathtt{W\ when}\ h_{30})\ \mathtt{default}\ (\mathtt{X2\ when}\ h_{27})]\$\ \text{when}\ (h_{29} \wedge h_{11}))$$

De manière générale, pour un signal e défini de la manière suivante:

$$\mathtt{e} = (\mathtt{E}_1 \ \mathtt{when}\ hu_e^{E_1})\ \mathtt{default}\ \ldots\ \mathtt{default}\ (\mathtt{E}_n\ \mathtt{when}\ hu_e^{E_n})\ \mathtt{default}\ (\mathtt{y\$}\ \mathtt{when}\ hu_e^{y\$})$$

où,

$$\mathtt{y} = \left[\ (\mathtt{Y}_1\ \mathtt{when}\ hu_y^{Y_1})\ \mathtt{default}\ \ldots\ \mathtt{default}\ (\mathtt{Y}_m\ \mathtt{when}\ hu_y^{Y_m})\ \right]_{hu_{y\$}}$$

et hu_b^a est l'horloge d'utilisation du signal a pour définir b. L'expression e peut être ré-écrite en:

$$\begin{aligned}\mathtt{e} \ = \ & (\mathtt{E}_1\ \mathtt{when}\ hu_e^{E_1})\ \mathtt{default}\ \ldots\ \mathtt{default}\ (\mathtt{E}_n\ \mathtt{when}\ hu_e^{E_n})\ \mathtt{default} \\ & \Big(\ (\ (\mathtt{Y}_1\ \mathtt{when}\ hu_y^{Y_1})\ \mathtt{default}\ldots\mathtt{default}\ (\mathtt{Y}_m\ \mathtt{when}\ hu_y^{Y_m})\ )\ \$\ \mathtt{when}\ hu_e^{y\$}\ \Big)\end{aligned}$$

La récursion sur X2 dénote la participation des autres termes de l'expression X2 à la définition de Y. Il s'agit des autres termes car tous les chemins menant de Y$ à Y ont déjà été explorés. Ce qui donne, par substitution:

$$\mathtt{X2} = (\mathtt{X}\ \mathtt{when}\ h_8)\ \mathtt{default}\ ([(\mathtt{W}\ \mathtt{when}\ h_{30})\ \mathtt{default}\ ((\mathtt{X}\ \mathtt{when}\ h_8)\ \mathtt{when}\ h_{27})]\$\ \mathtt{when}\ (h_{29} \wedge h_{11}))$$

De manière générale, pour un signal e défini de la manière suivante:

$$\begin{aligned}\mathtt{e} \ = \ & (\mathtt{E}_1\ \mathtt{when}\ hu_e^{E_1})\ \mathtt{default}\ \ldots\ \mathtt{default}\ (\mathtt{E}_n\ \mathtt{when}\ hu_e^{E_n})\ \mathtt{default} \\ & \Big(\ (\ (\mathtt{Y}_1\ \mathtt{when}\ hu_y^{Y_1})\ \mathtt{default}\ldots\mathtt{default}\ (\mathtt{e}\ \mathtt{when}\ hu_y^{e})\ )\ \$\ \mathtt{when}\ hu_e^{y\$}\ \Big)\end{aligned}$$

où,

$$\mathtt{y} = \left[\ (\mathtt{Y}_1\ \mathtt{when}\ hu_y^{Y_1})\ \mathtt{default}\ \ldots\ \mathtt{default}\ (\mathtt{Y}_m\ \mathtt{when}\ hu_y^{Y_m})\ \mathtt{default}\ (\mathtt{e}\ \mathtt{when}\ hu_y^{e})\ \right]_{hu_{y\$}}$$

et hu_b^a est l'horloge d'utilisation du signal a pour définir b. L'expression e peut être ré-écrite en:

$$\begin{aligned}\mathtt{e} \ = \ & (\mathtt{E}_1\ \mathtt{when}\ hu_e^{E_1})\ \mathtt{default}\ \ldots\ \mathtt{default}\ (\mathtt{E}_n\ \mathtt{when}\ hu_e^{E_n})\ \mathtt{default} \\ & \Big(\ (\ (\mathtt{Y}_1\ \mathtt{when}\ hu_y^{Y_1})\ \mathtt{default}\ldots\mathtt{default}\ (\ (\ (\mathtt{E}_1\ \mathtt{when}\ hu_e^{E_1})\ \mathtt{default}\ldots \\ & \mathtt{default}\ (\mathtt{E}_n\ \mathtt{when}\ hu_e^{E_n})\ )\ \mathtt{when}\ hu_y^{e}\ )\ )\ \$\ \mathtt{when}\ hu_e^{y\$}\ \Big)\end{aligned}$$

Ces deux opérations de substitution reviennent à couper les chemins existant entre Y et Y$. Elles peuvent s'interpréter par le schéma de la figure III.9.

III.4 Horloges contraintes par des signaux intermédiaires

Supposons qu'une tâche T réfère une horloge contrainte par un flot intermédiaire dans une ou plusieurs de ses expressions d'entrée. La définition de cette horloge requiert une expression de définition pour ce signal intermédiaire. Elle est obtenue en appliquant à ce flot le processus de transformation au même titre que les entrées d'une tâche.

Dans ce cas particulier, un certain nombre de signaux intermédiaires seront donc traités comme des entrées de tâches.

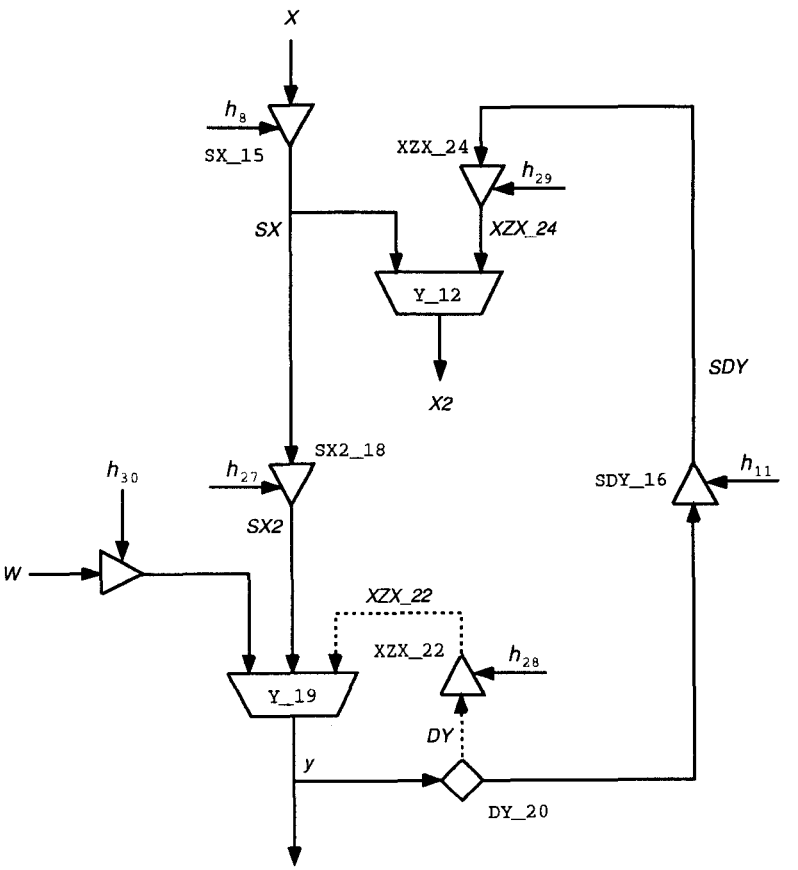


FIG. III.9 - *Graphe sans récursions temporelles.*

Chapitre IV

Implémentation des interfaces

La phase de définition des interfaces de communication qui vont enserrer les fonctions de calcul de l'application se base sur les résultats produits par les processus de normalisation et de transformation des expressions.

À ce stade de la ré-écriture, nous disposons à la fois d'un arbre d'horloges normalisé et d'une expression sur signaux de sortie (c'est-à-dire $\in \mathcal{S}$) pour chaque entrée des fonctions de calcul du système.

Nous avons deux types d'interfaces à définir:

1. La partie contrôle, chargée de définir les horloges contraignant les dépendances de donnée.
2. La partie donnée, qui est chargée de transmettre des flots produits localement (interface de sortie), et de définir les entrées de la fonction locale de calcul (interface d'entrée).

Notre approche étant basée sur le mode dataflow dit *demand-driven*, les interfaces de sortie des tâches sont entièrement définies par les expressions d'entrée des fonctions de calcul en aval.

Par exemple, considérons l'expression

$$\mathtt{x := y\ when}\ hu_x^y$$

où le signal **y** est émis par une tâche $\mathtt{T}_y$, **x** définit l'entrée d'une tâche $\mathtt{T}_x$, et hu_x^y est l'horloge d'utilisation du signal **y** pour définir **x**.
L'interface de sortie de $\mathtt{T}_y$ est en charge de transmettre le signal **y** à l'interface d'entrée de $\mathtt{T}_x$ parce que ce signal est requis dans l'expression de définition de l'entrée **x**.

Les horloges nécessitées par les interfaces de contrôle en amont et aval de chaque dépendance sont également déterminées par les expressions d'entrée des fonctions de calcul en aval.

IV.1 Dépendances de calcul

Pour toute entrée $\mathtt{e}_\alpha$ d'une fonction de calcul donnée, nous sommes capables de donner une expression sur signaux de sortie.

L'ensemble des entrées d'une tâche va déterminer les dépendances de calcul amont, ainsi que les horloges référencées localement par la tâche.

En pratique, l'implémentation d'une expression d'entrée est effectuée à partir d'un arbre. Cet arbre, dont la racine est le flot d'entrée, correspond simplement à un moyen commode d'en représenter l'expression de définition.

IV.1.1 Construction de l'arbre d'implémentation

Pour une entrée $\mathbf{e}_\alpha$ donnée d'une fonction de calcul, $\mathbf{e}_\alpha$ est la racine d'un arbre. Les différents éléments de l'expression sur signaux de sortie de $\mathbf{e}_\alpha$ sont placés dans l'arbre en respectant les règles suivantes:

1. Une expression $\mathbf{exp}_1$ définie par le sous-échantillonnage d'une expression $\mathbf{exp}_2$ à l'horloge hu, provoque le placement de l'expression $\mathbf{exp}_2$ sous le nœud $\mathbf{exp}_1$, la dépendance étant étiquetée par l'horloge hu.

2. Une expression **exp** définie par l'entrelacement d'un ensemble d'expressions $\mathbf{exp}_i$, aux horloges respectives d'utilisation hu_i, provoque le placement des expressions $\mathbf{exp}_i$ sous le nœud **exp**, les dépendances étant étiquetées par les hu_i.

3. Une expression $\mathbf{exp}_1$ définie par la valeur retardée d'une expression $\mathbf{exp}_2$ à l'horloge hu, provoque la création d'un nœuds de type *mémoire* sous $\mathbf{exp}_2$, la dépendance étant étiquetée hu.

4. Un nœud intermédiaire est créé pour chaque expression entre crochets. Deux nœuds fils lui sont associés: un premier nœud qui est la racine du sous-arbre définissant l'expression, et un second représentant la valeur retardée de l'expression (nœud de type *mémoire*). Sur la branche reliant le nœud père au nœud mémoire on fait figurer l'horloge d'utilisation de la valeur retardée de l'expression.

Ces différents points sont illustrés par les schémas de la figure IV.1.

Certains nœuds particuliers de l'arbre sont appelés *terminaux*.

Nœuds terminaux

L'ensemble des nœuds terminaux d'un arbre est constitué des nœuds "mémoire" et des signaux de sortie. On notera que les feuilles de l'arbre d'implémentation sont toujours des nœuds terminaux.

Dans la pratique, les nœuds terminaux correspondent aux flots qui détermineront effectivement les valeurs affectées aux entrées de la fonction de calcul.

- Les nœuds terminaux de type "signaux de sortie" sont acquis instantanément par les dépendances amont.

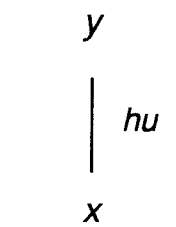
- Les nœud terminaux de type "mémoire" sont maintenus et mis à jour localement.

L'ensemble des nœuds de type "mémoire" définit les besoins requis en terme de stockage. Un nœud mémoire est conventionnellement noté **mem$x$\$** [1], il fait référence au contenu de la mémoire dénotée **mem$x$**.

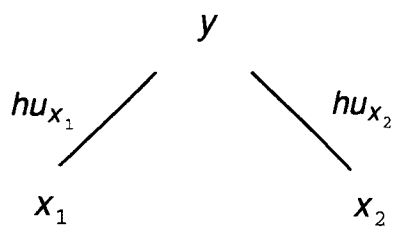
Exemple

Nous avons traité à titre d'exemple, l'expression $\mathbf{y}_1$ représentée en figure IV.2.

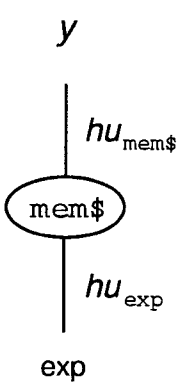
1. Remarque: la notation **mem$x$\$** ne se réfère absolument pas à l'opérateur de retard en SIGNAL, elle dénote simplement une case mémoire.



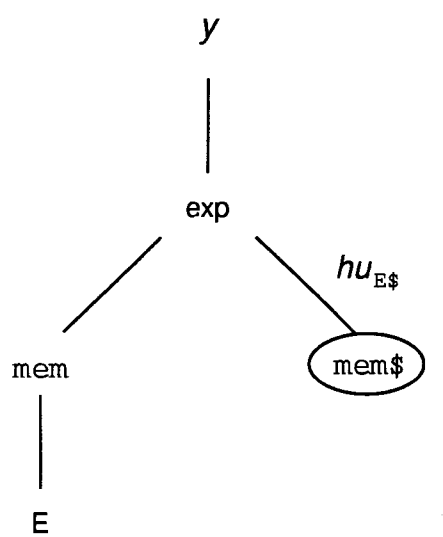
$y = x$ when hu



$y = (x_1$ when $hu_{x_1})$ default
$(x_2$ when $hu_{x_2})$



$y = (\text{exp}$ when $hu_{exp})\$$ when $hu_{mem\$}$



$y = [E]_{hu_{E\$}}$

FIG. IV.1 - *Éléments de construction de l'arbre d'implémentation.*

$$y_1 = \Big[\Big(\underbrace{\Big(\underbrace{\Big[\underbrace{(\underbrace{(x_1 \text{ when } h_1)\$}_{mem4\$} \text{ when } h_2) \text{ default } (x_2 \text{ when } h_3)}_{mem3}\Big]_{h_4}}_{mem2\$} \text{ when } h_5\Big) \$ \text{ when } h_6}_{exp2} \text{ default } \Big(\underbrace{\Big(\underbrace{\Big[\underbrace{(\underbrace{(x_3 \text{ when } h_7)\$}_{mem6\$} \text{ when } h_8) \text{ default } (x_4 \text{ when } h_9)}_{mem5}\Big]_{h_{10}}}_{exp2} \text{ default } (x_1 \text{ when } h_{11})\Big)}_{exp2} \text{ when } h_{12}\Big)\Big]_{h_{13}}$$

L'arbre d'implémentation construit à partir de l'expression de $\mathbf{y}_1$ est donné en figure IV.3.

IV.1.2 Parcours de l'arbre d'implémentation

Les règles de parcours de l'arbre d'implémentation permettent d'une part d'implémenter l'expression de définition de l'entrée racine de l'arbre et d'autre part de définir les expressions de remise à jour des mémoires.

Implémentation d'une entrée

Le parcours en profondeur de l'arbre définit un produit de contraintes à respecter, alors que le parcours en largeur de l'arbre définit une somme de contraintes.

L'implémentation de l'expression d'entrée est réalisée en parcourant l'arbre en profondeur du haut vers le bas et en largeur. On arrête de descendre dans une branche dès qu'un nœud terminal est rencontré.

Prenons l'arbre de définition de l'entrée $\mathbf{y}_1$, en figure IV.3. Si nous parcourons l'arbre de $\mathbf{y}_1$, nous obtenons:

```
SI (h13) y1 = mem1;
SINON SI (h6) y1 = mem2;
      SINON SI (h12) {
                SI (h11) y1 = x1;
                SINON SI (h10) y1 = mem5;
                SINON SI (h9) y1 = x4;
                SINON SI (h8) y1 = mem6; }
```

Implémentation des mises à jour "mémoire"

La gestion des "mémoires" du programme est intrinsèquement liée au choix d'implémentation que nous faisons: les valeurs retardées sont mémorisées localement par chacune de leurs tâches consommatrices.

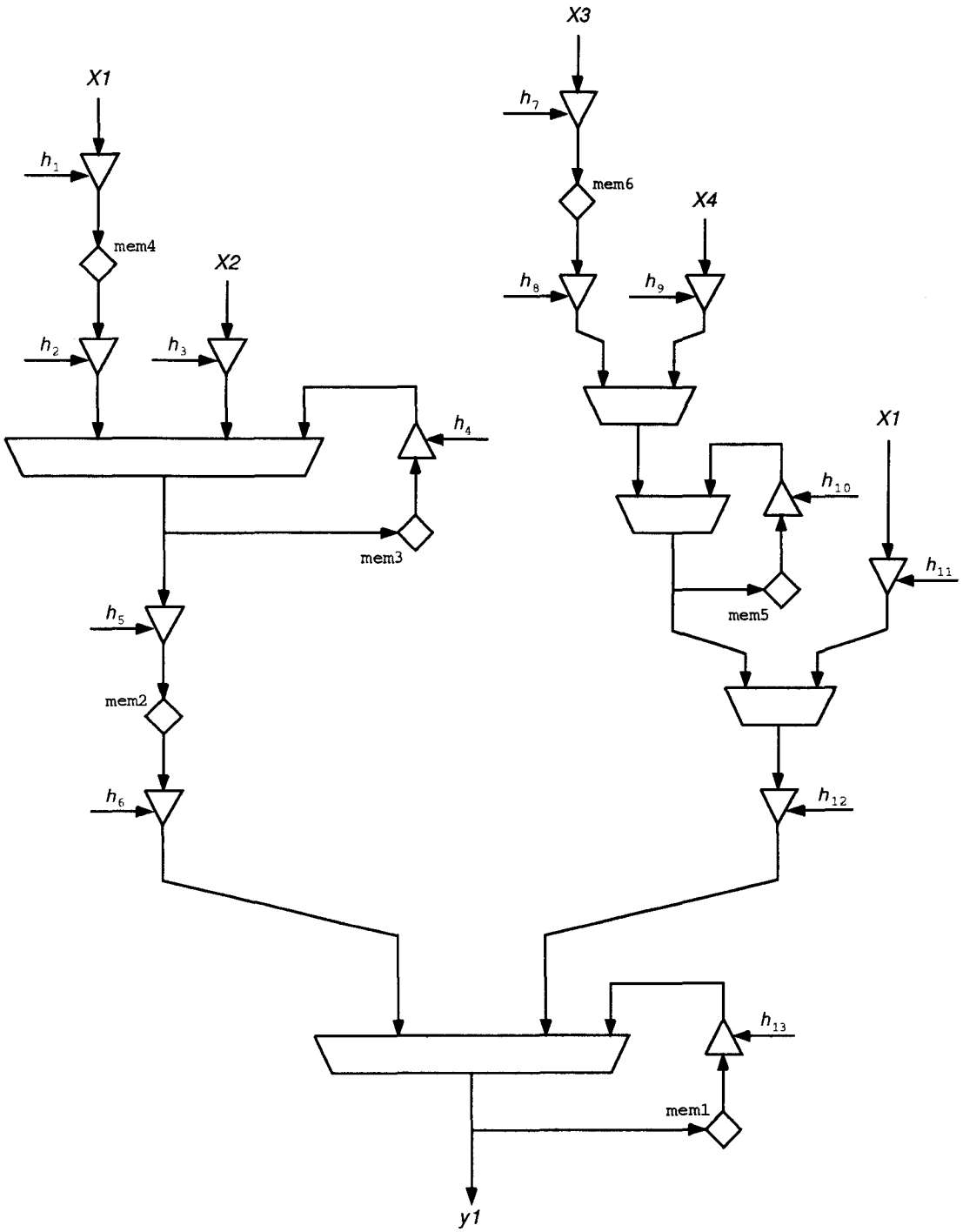


FIG. IV.2 - *Graphe de l'expression* $\mathbf{y}_1$.

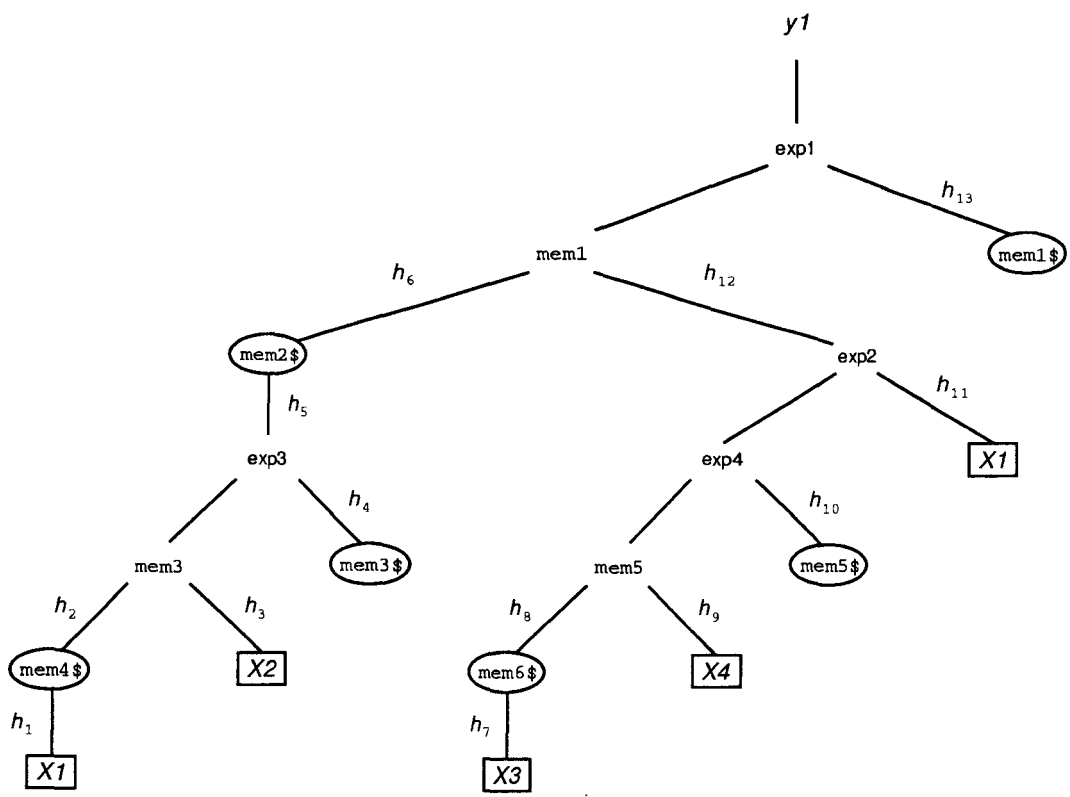


FIG. IV.3 - *Arbre d'implémentation de l'expression* $\mathbf{y}_1$.

Par conséquence les dépendances de calcul entre tâches sont exclusivement liées aux valeurs instantanées transmises dans le système. Ce qui signifie en d'autres termes que l'horloge d'utilisation d'une expression retardée pour définir une entrée ne joue aucun rôle dans la fréquence des dépendances entre tâches. Ces horloges ne seront donc pas mises en œuvre dans le calcul des dépendances.

Prenons par exemple, l'expression de définition d'une entrée SY. On dénote par T_A la tâche productrice du signal X et par T_B une tâche prenant le flot SY en entrée:

$$\texttt{SY} = [\ (\ \texttt{X when}\ h_8\ )\ \$\ \texttt{when}\ h_8\ ]_{h_{23}}\ \ \texttt{when}\ h_{15}$$

- (X when h_8) dénote une dépendance de calcul entre T_A et T_B, valide à l'horloge h_8. En d'autres termes la valeur instantanée du signal X qui vient d'être produite par T_A doit être transmise à la tâche T_B lorsque l'horloge h_8 est valide.
- Par contre, à l'horloge ($h_{23} \wedge h_{15}$), c'est la valeur retardée de l'expression (X when h_8) \$ qui est utilisée pour définir SY. Et cette valeur est détenue localement par la tâche T_B, elle n'augure donc pas de dépendance supplémentaire.

On remarque donc sur cet exemple, que la dépendance de calcul X entre T_A et T_B, valide à l'horloge h_8, sert à définir le contenu d'une mémoire qui sera effectivement utilisée à une horloge inférieure.

À partir de l'expression de définition de l'entrée SY que nous avons dérivée, nous pouvons donc déduire l'existence d'une dépendance de calcul de fréquence h_8 entre le consommateur du signal SY et le producteur du flot X.

Ce résultat, ajouté à l'expression de SY peut être directement interprété sous forme de commandes gardées, qui pourront être ultérieurement implémentées au moyen d'une librairie standard de communication.

$$\texttt{SY} = \left[\underbrace{\underbrace{(\ \texttt{X when}\ h_8\ )}_{mem1}\ \$\ \texttt{when}\ h_8}_{mem2} \right]_{h_{23}}\ \ \texttt{when}\ h_{15}$$

L'expression de définition de l'entrée SY nécessite l'implémentation de deux mémoires dénotées $mem1$ et $mem2$. L'horloge de mise à jour de la mémoire $mem1$ est égale à l'horloge de l'expression définissant son contenu, à savoir h_8. L'horloge $h_8 \wedge h_{15}$ définit les instants d'utilisation de $mem1$ pour définir SY, tandis que l'horloge h_8 dénote la suite des instants auxquels $mem1$ est utilisée pour définir la mémoire $mem2$. L'horloge d'utilisation de la mémoire $mem2$ est fixée par l'expression $h_{23} \wedge h_{15}$

Interface de sortie de T_A:
SI (h_8) émettre X vers T_A;

Interface d'entrée de T_B:
SI ($h_8 \wedge h_{15}$) SY = $mem1$;
SINON SI ($h_{23} \wedge h_{15}$) SY = $mem2$;
** Remise à jour des mémoires ***
SI (h_8) $mem2$ = $mem1$;
SI (h_8) $mem1$ = X;

Les "mémoires" ainsi définies dans les arbres d'implémentation permettent de déterminer avec précision pour chaque tâche du programme les besoins locaux en terme de stockage.

Les nœuds "mémoire" sont créés de deux manières: par des expressions définissant des récursions temporelles (nœuds **mem1$**, **mem3$**, **mem5$** dans l'exemple de la figure IV.3) — par des expressions qui sont simplement retardées sans impliquer de récursion temporelles (nœuds **mem2$**, **mem4$**, **mem6$** dans l'exemple de la figure IV.3).

Dans le premier cas, l'expression définissant la mémoire est constituée par le sous-arbre dont la racine est le nœud frère. Dans le second cas, l'expression définissant la mémoire est constituée par le sous-arbre dont le nœud mémoire est la racine.

De manière générale, si un nœud mémoire n'a pas de fils, il est défini par le sous-arbre dont le nœud frère est racine. Dans le cas contraire, il est défini par le sous-arbre dont il est la racine.

L'implémentation de la mise-à-jour des mémoires suit les mêmes règles de parcours que dans le cas de l'implémentation d'une entrée. À la différence que la racine de l'arbre à parcourir devient le nœud frère où le nœud mémoire lui-même.

Considérons à nouveau l'arbre de définition de l'entrée $\mathbf{y}_1$, en figure IV.3. Si nous parcourons l'arbre de $\mathbf{y}_1$, pour chacune des 6 mémoires:

Remise à jour de la mémoire mem1:
SI (h_6) mem1 = mem2;
SINON SI (h_{12})
SI (h_{11}) mem1 = $\mathbf{x}_1$;
SINON SI (h_{10}) mem1 = mem5;
SINON SI (h_9) mem1 = $\mathbf{x}_4$;
SINON SI (h_8) mem1 = mem6;

Remise à jour de la mémoire mem2:
SI (h_5)
SI (h_4) mem2 = mem3;
SINON SI (h_2) mem2 = mem4;
SINON SI (h_3) mem2 = $\mathbf{x}_2$;

Remise à jour de la mémoire mem3:
SI (h_2) mem3 = mem4;
SINON SI (h_3) mem3 = $\mathbf{x}_2$;

Remise à jour de la mémoire mem4:
SI (h_1) mem4 = $\mathbf{x}_1$;

Remise à jour de la mémoire mem5:
SI (h_8) mem5 = mem6;
SINON SI (h_9) mem5 = $\mathbf{x}_4$;

Remise à jour de la mémoire mem6:
SI (h_7) mem6 = $\mathbf{x}_3$;

On notera par ailleurs que l'ordre dans lequel les mémoires sont mises à jour est important. Par exemple, si **mem2** était définie avant **mem1**, la mise à jour de **mem1** serait erronée, car **mem1** référence **mem2** dans son expression de définition.

IV.2 Partage des mémoires

Les n entrées d'une tâche T, dénotées { y_1, ..., y_n } sont donc définies par une forêt d'arbres dont les racines sont les y_i.

Supposons qu'une tâche T donnée possède deux entrées y_1, y_2, y_1 étant définie par l'arbre de la figure IV.3, et y_2 par l'arbre de la figure IV.5, qui correspond au système IV.4.

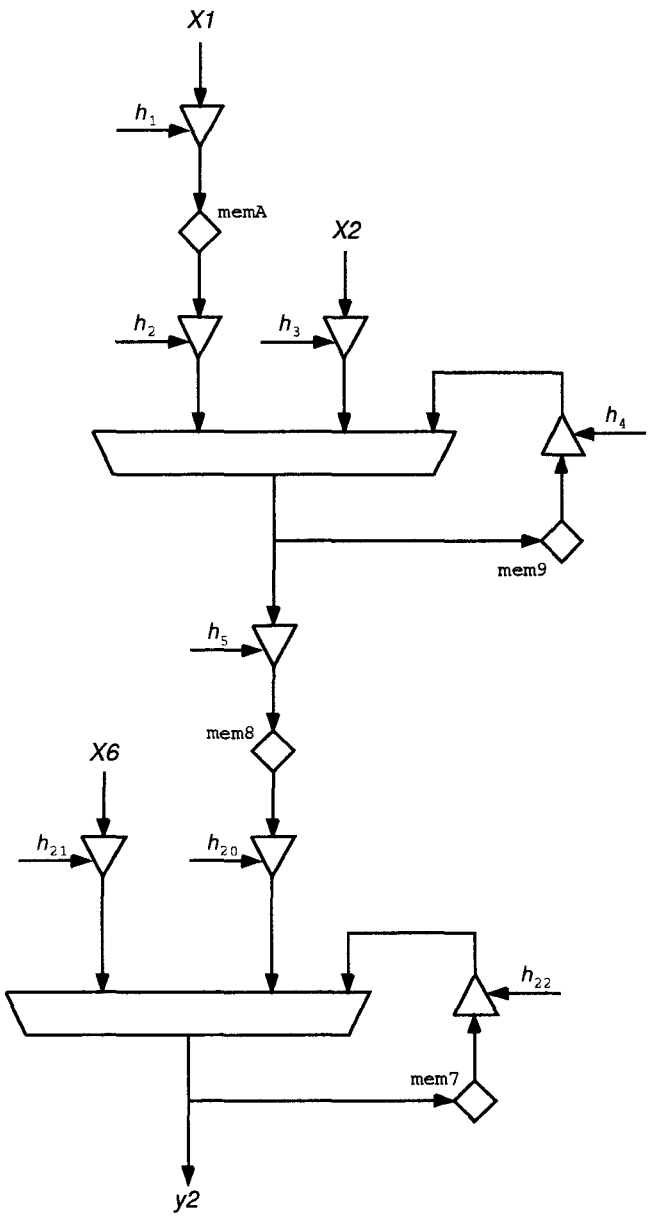


FIG. IV.4 - *Graphe de l'expression* y_2.

L'entrée y_2 ajoute 4 nouvelles mémoires à la tâche T. On remarquera cependant qu'une partie de l'arbre y_2 coïncide avec l'arbre y_1. La réunion de ces deux arbres peut apporter un gain très appréciable en terme de gestion mémoire. Car en effet, 3 des mémoires définies par l'entrée y_2

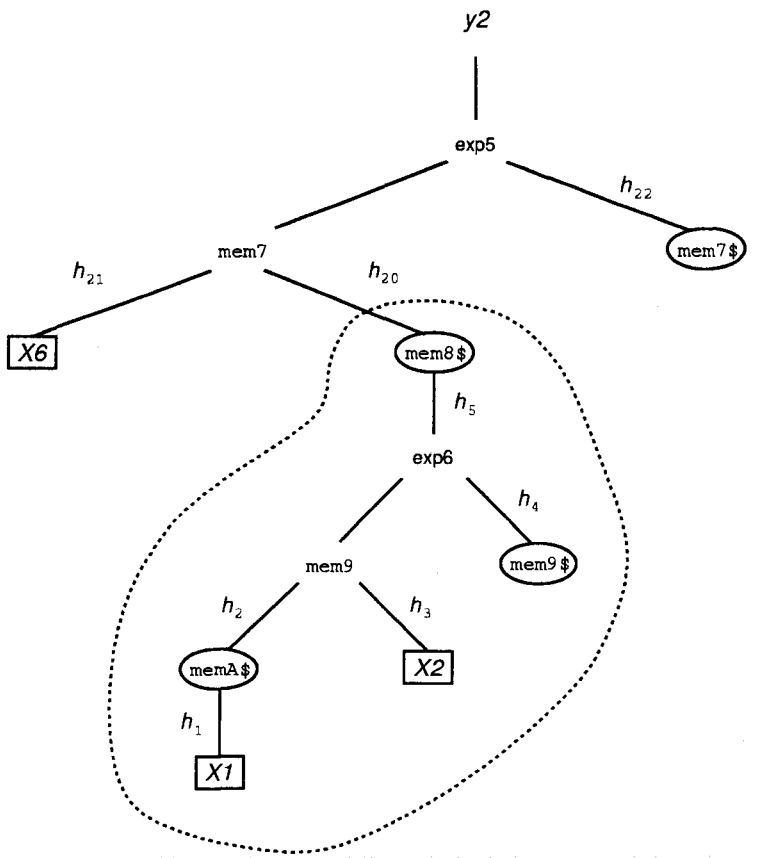


FIG. IV.5 - *Arbre d'implémentation de l'expression* $\mathbf{y}_2$.

(**mem8**, **mem9**, **memA**) sont également utilisées par l'entrée $\mathbf{y}_1$.

Deux sous-arbres peuvent être ainsi fusionnés s'ils coïncident non seulement au niveau des nœuds, mais également au niveau des horloges.

Nous présentons en figure IV.6 la réunion de ces deux arbres.

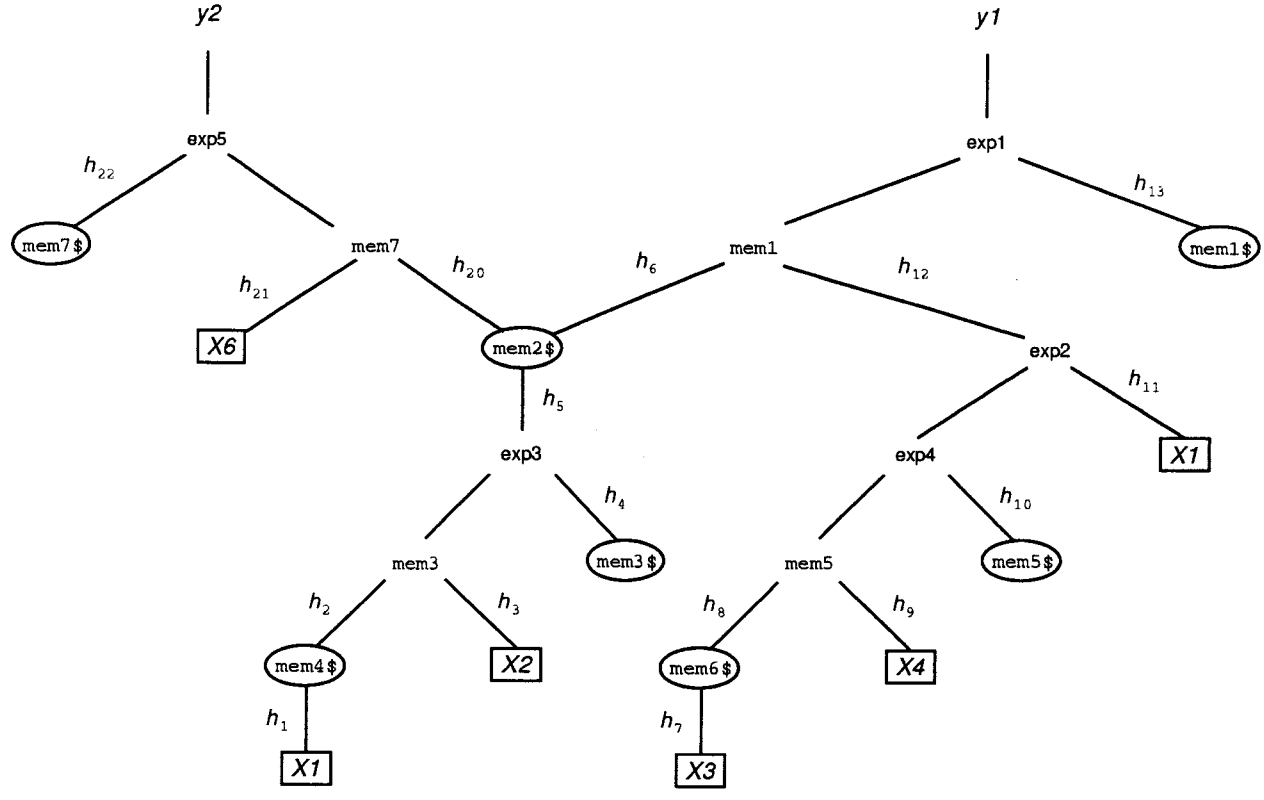


FIG. IV.6 - *Réunion des arbres d'implémentation* $\mathbf{y}_1$ *et* $\mathbf{y}_2$.

En définitive, l'implémentation des n entrées d'une fonction de calcul sera définie à partir d'une forêt d'arbres. Cette forêt sera constituée d'au moins un arbre et d'au plus n, un arbre pouvant définir plusieurs entrées.

IV.3 Partie contrôle

Avant de démarrer l'étude de cette partie, faisons un point sur la chaîne de transformations des expressions (voir figure IV.7). Nous avons commencé par scinder les aspects contrôle et donnée pures. La partie contrôle traitant des dépendances d'horloge, alors que la partie donnée détermine les dépendances de calcul induites par la demande de signaux.

- Les aspects contrôle reposent sur le processus de normalisation de l'arbre d'horloge. À chaque horloge référencée dans le système, celui-ci associe une expression booléenne sur signaux. Ces signaux sont soit des sorties produites par des fonctions de calcul, soit des signaux intermédiaires. Dans ce dernier cas, une expression sur signaux de sortie est donnée par le processus de transformation des entrées.

- La partie donnée vise par un processus de transformation, à donner une expression sur signaux de sortie à chaque entrée de fonction de calcul, ainsi qu'à tout signal intermédiaire référencé dans l'ensemble des équations booléennes d'horloge de l'arbre normalisé.

Nous trouvons dans la forêt des arbres d'implémentation, la liste des horloges nécessaires au contrôle des dépendances de données. Ces dépendances sont de deux types:

1. Les dépendances de calcul instantanées, c'est-à-dire toute relation liant la valeur courante d'un signal de sortie avec une entrée de fonction de calcul.

2. Les dépendances mémoire, c'est-à-dire toute relation mettant à jour une mémoire locale par un signal de sortie.

Dans l'arbre d'implémentation de l'entrée $\mathbf{y}_1$ (voir figure IV.3), nous trouvons des exemples de ces deux types de relation: la valeur instantanée du signal $\mathbf{x}_1$ est à la fois référencée dans l'expression définissant l'entrée $\mathbf{y}_1$, mais aussi dans la définition de la mémoire **mem4$**.

Les horloges associées à chacune de ces dépendances sont obtenues par simple parcours des arbres d'implémentation.

IV.3.1 Expression normale d'horloge sur signaux intermédiaires

Un signal intermédiaire référencé dans une expression normale d'horloge donne lieu à la création d'un arbre d'implémentation. Ces arbres ont la particularité d'être nécessaires au contrôle des arbres d'implémentation des entrées, dans la mesure où ils définissent des horloges. Ils seront donc implémentés avant toute autre partie de code.

Nous présentons dans le schéma de la figure IV.8, un exemple de signal intermédiaire définissant une horloge (le signal `B`). Ce signal définit également une dépendance de calcul en entrée de $\mathtt{T}_C$.

Le graphe des dépendances de calcul associé à ce programme est donné en figure IV.9.

Normalisation des horloges

La compilation de ce programme produit le système d'équations d'horloges suivant:

$$
\begin{aligned}
h_8 &= \overline{[\mathrm{B1}]} \\
h_9 &= [\mathrm{B1}] \\
h_{11} &= [\mathrm{B2}] \\
h_{23} &= [\mathrm{B}] \\
h_{30} &= h_8 \wedge h_{11} \\
h_{31} &= h_9 \vee h_{11} \\
h_{32} &= \overline{h_{23}} \\
h_{33} &= \overline{h_{31}} \\
h_{34} &= h_{30} \wedge \overline{h_9}
\end{aligned}
$$

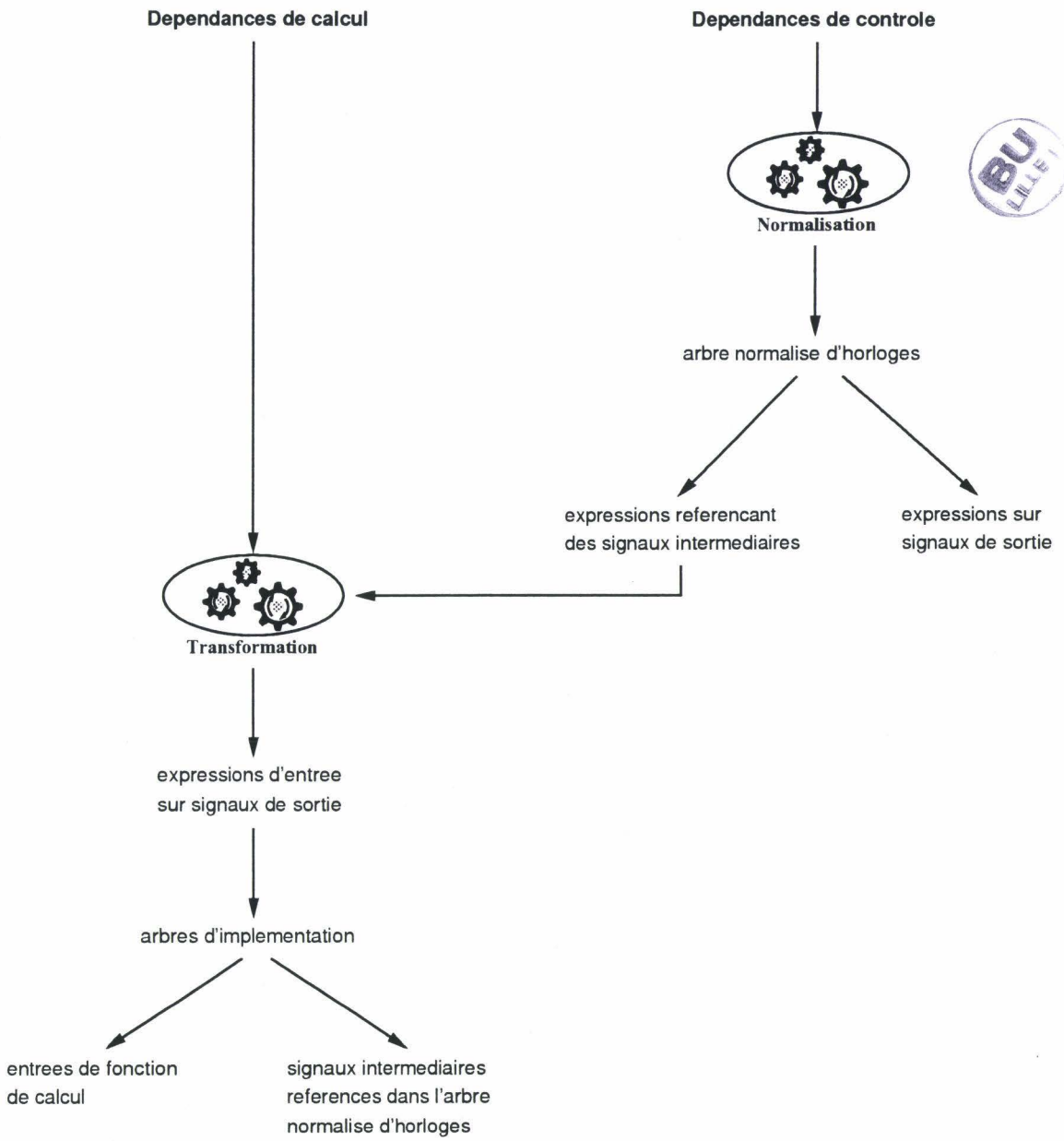


FIG. IV.7 - *Chaîne de transformation des dépendances de contrôle et de calcul.*

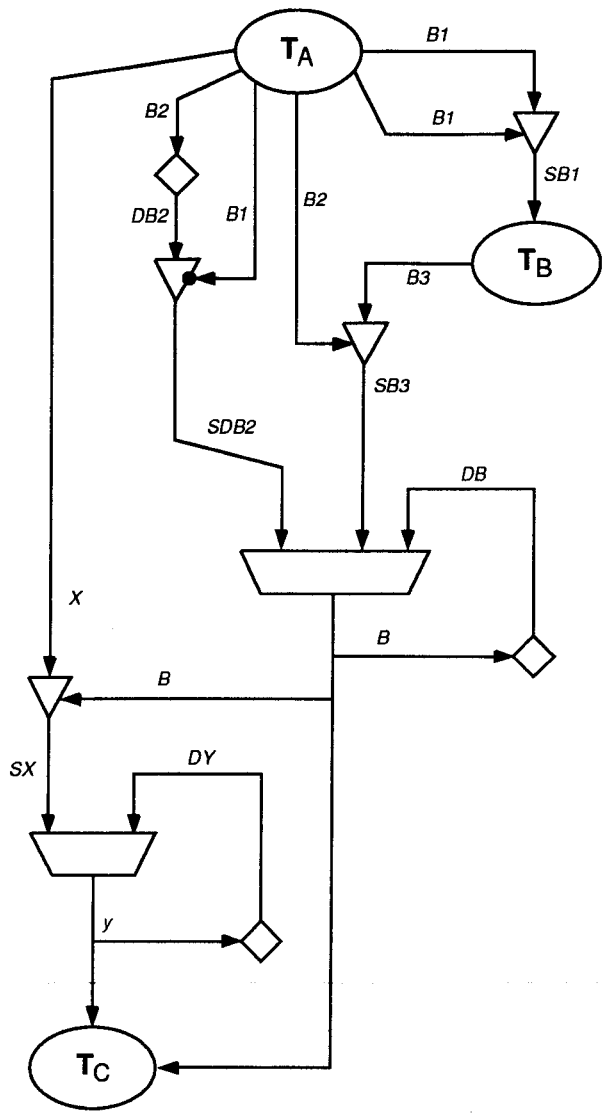


FIG. IV.8 - *Exemple.*

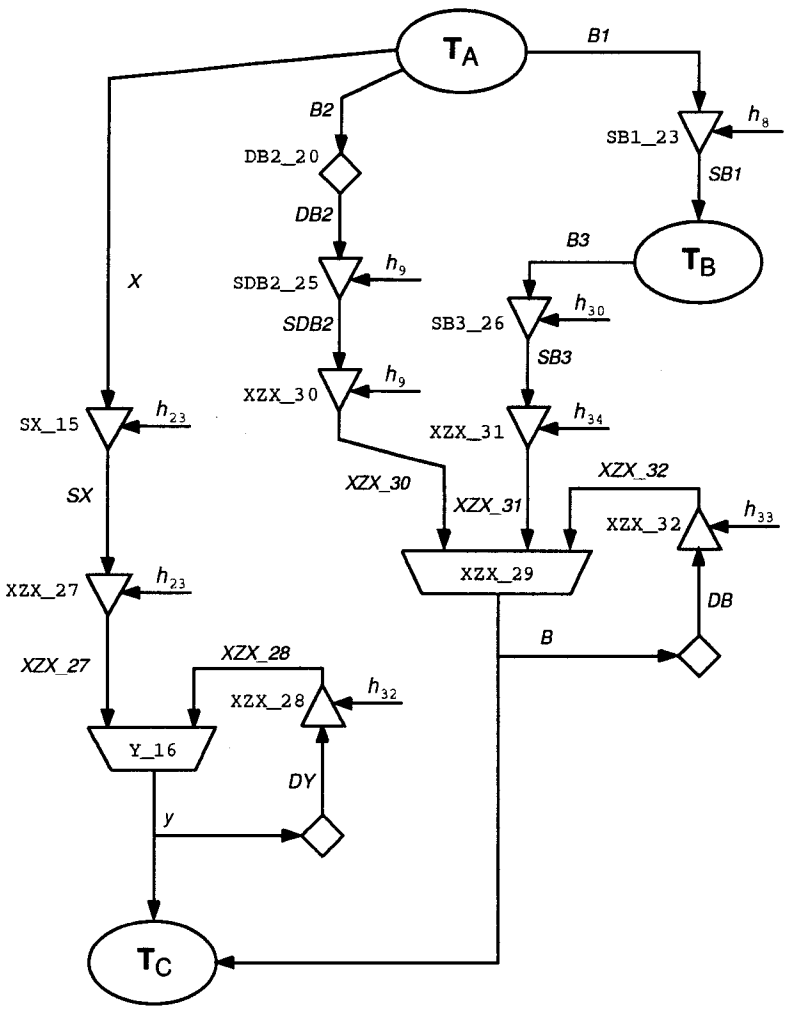


FIG. IV.9 - *Dépendances de calcul.*

Ce qui donne donc,

$\mathcal{VH}$	$\mathcal{VHC}$
$\emptyset$	$h_8 \xrightarrow{h_{23}} [B1]$
	$h_9 \xrightarrow{h_{23}} \overline{[B1]}$
	$h_{11} \xrightarrow{h_{23}} [B2]$
	$h_{23} \xrightarrow{h_{23}} [B]$
	$h_{30} \xrightarrow{h_{23}} [B1] \wedge [B2]$
	$h_{31} \xrightarrow{h_{23}} \overline{[B1]} \wedge [B2]$
	$h_{32} \xrightarrow{h_{23}} \overline{[B]}$
	$h_{33} \xrightarrow{h_{23}} [B1] \vee [B2]$
	$h_{34} = h_{30}$

Détermination des équations d'entrée

L'obtention des entrées de la tâche T_B est directe. Nous ne nous attarderons donc pas dessus. La tâche T_C prend deux entrées: $\mathcal{E}_{T_C} = \{$ `Y`, `B` $\}$. Les dépendances de calcul sont données par le système suivant:

1- Y = [XZX_27 when h_{23}]$_{h_{32}}$
2- XZX_27 = SX when h_{23}
3- SX = X when h_{23}
4- B = [(XZX_30 when h_9) default (XZX_31 when h_{30})]$_{h_{33}}$
5- XZX_30 = SDB2 when h_9
6- SDB2 = DB2 when h_9
7- DB2 = B2$
8- XZX_31 = SB3 when h_{30}
9- SB3 = B3 when h_8
10- SB1 = B1 when h_8

Transformation de l'entrée `Y`:

a: $2 \rightarrow 1$ Y = [SX when h_{23}]$_{h_{32}}$
b: $3 \rightarrow$ a Y = [X when h_{23}]$_{h_{32}}$

Transformation de l'entrée `B`:

c: $\{ 5, 8 \} \rightarrow 4$ B = [(SDB2 when h_9) default (SB3 when h_{30})]$_{h_{33}}$
d: $\{ 6, 9 \} \rightarrow$ c B = [(DB2 when h_9) default (B3 when ($h_{30} \wedge h_8$))]$_{h_{33}}$
e: $7 \rightarrow$ d B = [(B2$ when h_9) default (B3 when ($h_{30} \wedge h_8$))]$_{h_{33}}$

Implémentation

Des simplifications sont très souvent possibles dans les expressions d'horloge en fin de transformation. Par exemple, on a:

$$h_{30} \wedge h_8 = h_{30}$$

Nous ne nous sommes pas préoccupé davantage de ce point, dans le cadre de notre travail, dans la mesure où ce type de simplifications ne change en rien le nombre de dépendances. En effet, que nous traitions l'expression $h_{30} \wedge h_8$ ou plus simplement h_{30}, nous obtenons toujours

une dépendance sur les signaux **B1** et **B2**.

Les arbres d'implémentation correspondant aux entrées **B** et **Y** sont donnés en figure IV.10. Les horloges h_{23} et h_{32} font référence au signal **B**. Il importe donc que ce flot soit défini avant de traiter l'entrée **Y**. Dans notre exemple, le signal **B** est explicitement demandé en entrée de la tâche T_C, néanmoins, si cela n'avait pas été le cas, la référence aux horloges h_{23} et h_{32} dans l'arbre **Y** aurait suffit à commander l'implémentation de **B**.

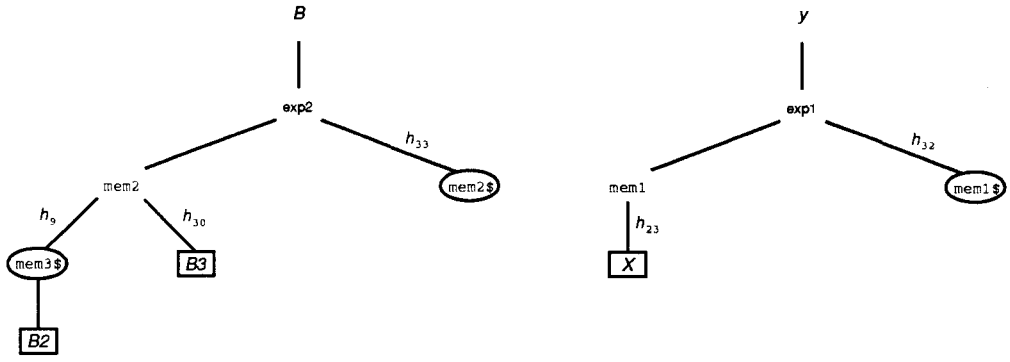


FIG. IV.10 - *Arbres d'implémentation.*

Nous donnons ci-après, le code correspondant aux deux flots d'entrée.

Définition du flot B:	`SI (`h_{33}`) B = mem2;`
	`SINON SI (`h_9`) B = mem3;`
	`SINON SI (`h_{30}`) B = B3;`
Définition du flot Y:	`SI (`h_{32}`) Y = mem1;`
	`SINON SI (`h_{23}`) Y = X;`
Remise à jour de la mémoire mem2:	`SI (`h_9`) mem2 = mem3;`
	`SINON SI (`h_{30}`) mem2 = B3;`
Remise à jour de la mémoire mem3:	`mem3 =B2;`
Remise à jour de la mémoire mem1:	`SI (`h_{23}`) mem1 = X;`

En pratique, l'expression de définition du signal **B** serait implémentée dans la partie contrôle de l'interface. Les horloges dont l'expression normale fait référence à ce signal sont définies après la définition de **B**.

IV.3.2 Détermination de la fréquence des dépendances

Les règles de parcours des arbres d'implémentation sont simples:

1. Pour une tâche donnée, on détermine l'ensemble fini des signaux de sortie référencés dans la forêt des arbres d'implémentation. On parcourt pour cela le feuillage.
2. Pour chaque signal de sortie, on examine dans la forêt des arbres d'implémentation, tous les chemins menant d'une feuille référençant ce signal vers la racine de l'arbre.
3. La *"remontée"* d'une branche s'achève lorsqu'on a atteint la racine d'un arbre, ou un nœud terminal (qui correspondra en l'occurrence à un nœud mémoire). Dans le premier cas, la dépendance est instantanée, dans le second cas, il s'agit d'une dépendance de remise à jour mémoire.

4. Pour chaque branche étudiée, la dépendance est contrainte par le produit des horloges rencontrées sur le chemin.

5. Pour un signal de sortie donné, la fréquence de la dépendance entre la tâche productrice du signal, et sa consommatrice est définie par la somme des produits calculés sur chaque chemin.

Par exemple, si on examine la fréquence de la dépendance entre la tâche productrice du signal $\mathbf{x}_1$ et la fonction de calcul prenant le signal $\mathbf{y}_1$ en entrée (cf. figure IV.3),

$$\begin{array}{lll} \text{Chemin 1-} & \mathbf{x}_1 \xrightarrow{h_1} & \texttt{mem4\$} \\ \text{Chemin 2-} & \mathbf{x}_1 \xrightarrow{h_{11}} \texttt{exp2} \xrightarrow{h_{12}} \texttt{mem1} \longrightarrow \texttt{exp1} \longrightarrow & \mathbf{y}_1 \end{array}$$

Il existe donc entre la tâche émettrice du signal $\mathbf{x}_1$ et la fonction de calcul `T` prenant le signal $\mathbf{y}_1$ en entrée, une dépendance de fréquence,

$$\mathcal{D}_T^{x_1} = h_1 \vee (h_{11} \wedge h_{12})$$

En d'autres termes, le signal $\mathbf{x}_1$ est transmis à la tâche `T` lorsque l'expression $\mathcal{D}_T^{x_1}$ est valide. Cette dépendance sera implémentée dans l'interface de sortie de la tâche émettrice du signal $\mathbf{x}_1$.

Symétriquement, l'expression $\mathcal{D}_T^{x_1}$ donne la fréquence à laquelle le signal $\mathbf{x}_1$ est reçu sur le port d'entrée correspondant de la tâche `T`.

IV.3.3 Horloges

Dans les arbres d'implémentation, référence est faite à un certain nombre d'horloges. Nous appelons $\mathcal{EH}$ l'ensemble fini de ces horloges. À chaque élément de $\mathcal{EH}$, on associe un arbre représentant son expression normale à l'horloge maîtresse.

Forêt des formes normales

Ces arbres sont construits à partir de l'arbre normalisé des horloges. Un arbre a pour racine un élément de $\mathcal{EH}$, et les chemins de la racine à chacune des feuilles de l'arbre détermine les conditions de validité de cette horloge.

Les nœuds de l'arbre sont des expressions booléennes référençant des signaux de sortie ou des signaux intermédiaires pour lesquels un arbre d'implémentation a été construit.

Le parcours de l'arbre en profondeur dénote un produit de contraintes. Alors que le parcours en largeur dénote une somme de contraintes.

Les fils d'un nœud sont des signaux booléens synchrones émis sous la condition exprimée par le chemin menant de la racine de l'arbre au nœud père.

La validation d'une horloge est réalisée lorsque les signaux booléens référencés dans l'arbre normal ont permis de valider au moins l'un de ces chemins.

Promenade dans la forêt des formes normales

On dénote par $\mathcal{A}$, l'ensemble des arbres normaux pour un ensemble $\mathcal{EH}$ d'horloges. La cardinalité de ces deux ensembles est bien entendue la même. On dénote également par $\mathcal{SA}$, l'ensemble des signaux référencés dans la forêt $\mathcal{A}$ des formes normales pour une même strate de fréquence. Tous les signaux référencés dans une même strate de fréquence sont synchrones.

La validation des horloges de $\mathcal{EH}$ est réalisée de la manière suivante:

1. Condition initiale: On dénote respectivement par $\mathcal{EVH}$, et $\mathcal{EIH}$ l'ensemble des horloges validées et invalidées. Initialement, les ensembles $\mathcal{EVH}$ et $\mathcal{EIH}$ sont vides.

$$\mathcal{EH}, \mathcal{EVH} = \emptyset, \mathcal{EIH} = \emptyset$$

   La première strate de fréquence, $\mathcal{SA}$ correspond à l'ensemble des signaux de sortie émis à l'horloge racine. On passe d'une strate de fréquence à une autre en largeur d'abord, puis en profondeur.

2. Attente des signaux référencés dans la strate $\mathcal{SA}$ courante.

3. En fonction des instants de vérité des expressions de $\mathcal{A}$ référençant des éléments de $\mathcal{SA}$, un certain nombre d'horloges sont validées ou invalidées.

$$\frac{\{\ \mathcal{EH}, \mathcal{EVH}\ \}}{h \in \mathcal{EH},\ \mathrm{E}(\mathcal{SA}) \text{ valide un chemin de } h} \longrightarrow \{\ \mathcal{EH}', \mathcal{EVH}'\ \}$$

   - $\mathcal{EH}' = \mathcal{EH} - \{\ h\ \}$
   - $\mathcal{EVH}' = \mathcal{EVH} + \{\ h\ \}$

$$\frac{\{\ \mathcal{EH}, \mathcal{EIH}\ \}}{h \in \mathcal{EH},\ \mathrm{E}(\mathcal{SA}) \text{ invalide le dernier chemin possible de } h} \longrightarrow \{\ \mathcal{EH}', \mathcal{EIH}'\ \}$$

   - $\mathcal{EH}' = \mathcal{EH} - \{\ h\ \}$
   - $\mathcal{EIH}' = \mathcal{EIH} + \{\ h\ \}$

   De manière plus générale, des chemins dans les arbres normaux peuvent être coupés.

4. Changement de strate si $\mathcal{EH} \neq \emptyset$

Exemple 1

Prenons par exemple, l'arbre que nous avons étudié dans le second chapitre (SECTION II.2, voir également la figure IV.11), et supposons que $\mathcal{EH} = \{\ h_1, h_5, h_8\ \}$.

Nous avions obtenu pour cet arbre le système suivant:

$\mathcal{VH}$	$\mathcal{VHC}$
$\emptyset$	$h_1 \xrightarrow{h_r} [\overline{a}] \vee (\ [a] \wedge (\ [\overline{c}] \vee (\ [b] \wedge ([d] \vee [e])\ )\ )\ )$
	$h_2 \xrightarrow{h_r} [a]$
	$h_3 \xrightarrow{h_2} [c] \wedge ([\overline{b}] \vee ([b] \wedge [\overline{d}] \wedge [\overline{e}]))$
	$h_4 \xrightarrow{h_2} [b]$
	$h_5 \xrightarrow{h_2} [c]$
	$h_6 \xrightarrow{h_4} [d]$
	$h_7 \xrightarrow{h_4} [e]$
	$h_8 \xrightarrow{h_4} [d] \vee [e]$

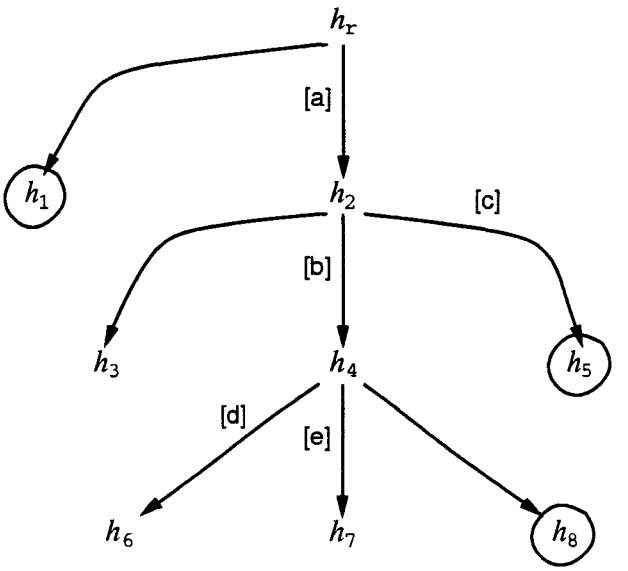


FIG. IV.11 - *Arbres d'implémentation.*

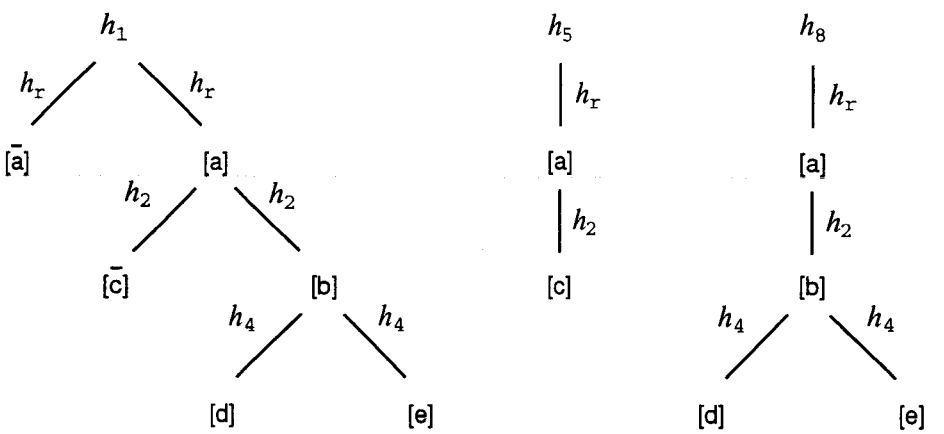


FIG. IV.12 - *Expressions normales d'horloge.*

La forêt des formes normales associée à l'ensemble $\mathcal{EH}$ est représentée en figure IV.12.

En pratique, le processus de validation des horloges { h_1, h_5, h_8 } est implémenté dans la partie contrôle des tâches concernées par la portion de pseudo-code suivante, où la fonction **get**(*liste_de_signaux*) dénote l'attente de signaux. Pour les signaux localement disponibles, la fonction **get()** n'occasionnera pas d'opération de communication.

Initialisation: Toutes les horloges sont invalidées

```
Strate h_r:     get(a);
                  si (a) {
Strate h_2:         get(b,c);
                    si (c)
                      h_5 valide;
                    sinon
                      h_1 valide;
                    si (b) {
Strate h_4:           get(d,e);
                      si (d || e) {
                        h_1 valide;
                        h_8 valide;
                      }
                    }
                  }
                  sinon
                    h_1 valide;
```

Supposons qu'à l'instant courant, le flot booléen **a** porte la valeur *faux*. Dans ce cas, seule l'horloge h_1 est validée, tandis que toutes les autres restent invalidées sans que le reste du code soit exécuté.

Exemple 2

Reprenons le système décrit en figures IV.8 et IV.9. Cet exemple fait apparaître un signal intermédiaire dans l'expression normale associée aux horloges h_{23} et h_{32}. Il est à ce titre intéressant.

L'arbre d'horloges correspondant à ce système est donné en figure IV.13.

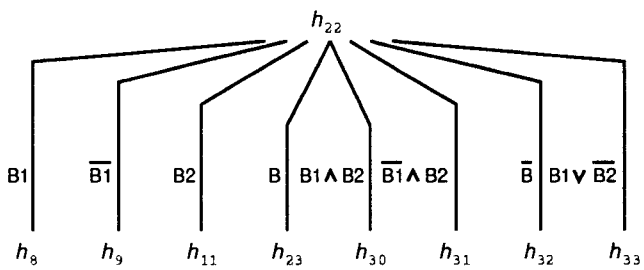


FIG. IV.13 - *Arbre d'horloges.*

L'ensemble $\mathcal{EH}_B$ des horloges référencées par l'expression **B** est égal à: { h_9, h_{30}, h_{33} }. Ces horloges sont donc nécessaires pour définir **B**.

L'ensemble $\mathcal{EH}_Y$ des horloges référencées par l'expression `Y` est égal à: { h_{23}, h_{32} }. Toutes ces horloges référencent le signal `B`.

Partie contrôle:

Strate h_{22}:
`get(B1,B2);`
`si (!B1)` h_9 valide;
`si (B1 && B2)` h_{30} valide;
`si (B1 || !B2)` h_{33} valide;

Définition des ports d'entrée: `si (`$\mathcal{D}^{B3}$`) get(B3);`

Implémentation du flot `B`

Strate h_{22}:
`si (B)` h_{23} valide;
`si (!B)` h_{32} valide;

Partie donnée:

Définition des ports d'entrée: `si (`$\mathcal{D}^{X}$`) get(X);`

Implémentation du flot `Y`

La section *Définition des ports d'entrée* mérite un petit commentaire. Nous avons déjà montré comment obtenir la fréquence d'une dépendance pour un signal de sortie donné référencé dans la forêt des arbres d'implémentation.

L'arbre `B` référence le signal `B3` à l'horloge h_{30}, l'arbre `Y` référence le signal `X` à l'horloge h_{23}, nous avons donc pour ce cas simple:

$$\mathcal{D}^{B3} = h_{30}, \mathcal{D}^{X} = h_{23}$$

Émission des signaux

Dans notre approche *demand-driven*, ce sont les même arbres de contrôle et de calcul définissant les entrées d'une tâche qui vont déterminer le corps des interfaces de sortie en amont.

Des dépendances amont sont créées pour tout signal référencé dans un arbre de contrôle ou de calcul. L'émission de ces signaux est contraint par leur fréquence d'utilisation en aval.

Nous avons déjà déterminé dans le cas des arbres d'implémentation l'horloge à laquelle une dépendance est globalement valide pour un signal donné (voir SECTION IV.3.2). Par exemple, dans le système défini par le schéma de la figure IV.8, nous avons déterminé les dépendances suivantes:

$$\mathcal{D}^{B3} = h_{30}, \mathcal{D}^{X} = h_{23}$$

En termes d'implémentation cela signifie que le signal `B3` sera transmis à la fonction $\mathtt{T}_C$ à l'horloge h_{30}, tandis que le signal `X` sera transmis à l'horloge h_{23}. L'émission contrainte de ces signaux suppose donc de définir les horloges h_{30} dans l'interface de sortie de la tâche productrice du signal `B3`, et h_{23} dans l'interface de sortie de la tâche produisant le signal `X` (voir figure IV.14). L'implémentation des horloges h_{23} et h_{30} dans les différentes interfaces de sortie induisant de nouvelles dépendances en amont.

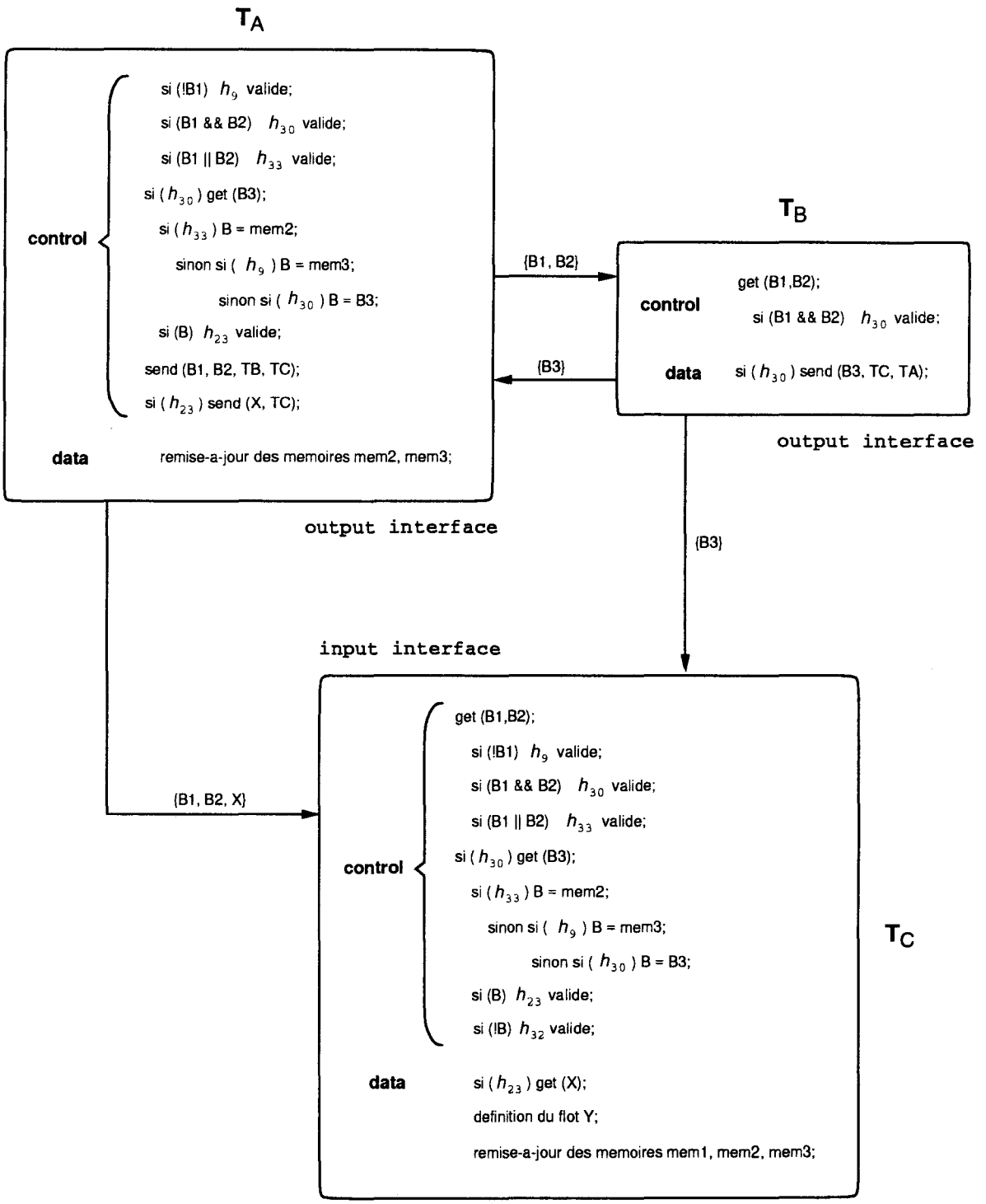


FIG. IV.14 - *Implémentation des interfaces induites par la définition de l'entrée* Y.

On remarquera que bien souvent des optimisations sont ponctuellement envisageables. Par exemple, la tâche $\mathtt{T}_B$ est active à l'horloge h_8, c'est-à-dire lorsque le flot booléen `B1` porte la valeur *vrai*. Autrement dit, le flot `B3` n'est produit que lorsque `B1` est *vrai*. Il est donc possible de simplifier le code de l'interface de sortie de `B1` (voir figure IV.15).

Cette optimisation a en outre permis de réduire le nombre de dépendances entre $\mathtt{T}_A$ et $\mathtt{T}_B$.

L'automatisation des optimisations nécessiterait d'être étudiée plus formellement. Nous ne traitons pas ce problème dans le cadre du document, nous nous contentons de montrer sur un exemple l'intérêt qui peut en être tiré.

IV.4 Accusés de réception

Un problème traditionnel posé par le modèle data-flow est de fournir des mécanismes permettant de borner les queues implémentant les arcs de données [Buc93].

Dans notre cas, nous avons opté pour un système d'accusé de réception à la manière d'ATAMM [SMS93] (*Algorithm To Architecture Mapping Model*)[2].

Les systèmes temps-réel manipulent des flots *infinis* portant à la fois une information de contrôle et de valeur. Les dépendances de données dans de tels systèmes sont soumis à des contraintes qui à un instant donné peuvent valider ou non une dépendance. Les instants de validité des dépendances sont parfaitement connus à la compilation.

Nous savons déterminer pour chaque signal référencé dans la forêt des expressions d'entrée, la fréquence de la dépendance qu'elle entraîne. Cette valeur a été dénotée $\mathcal{D}_T^s$, pour une dépendance de fréquence $\mathcal{D}_T^s$ entre la tâche productrice du signal `s` et sa consommatrice `T`.

Appelons $\mathtt{T}_\Sigma$ la tâche productrice du signal `s`. Un token d'accusé réception doit être émis de `T` vers $\mathtt{T}_\Sigma$ lorsque la donnée `s` a été transmise, afin de débloquer la fonction de calcul de $\mathtt{T}_\Sigma$ et lui permettre ainsi de produire de nouvelles sorties.

Si nous savons que la donnée `s` est transmise aux instants pour lesquels l'horloge $\mathcal{D}_T^s$ est valide, on peut conclure qu'un accusé de réception doit être retourné à $\mathtt{T}_\Sigma$ avec la même fréquence.

Supposons maintenant, qu'il existe un ensemble de dépendances $\{ \mathtt{s}_1, \ldots, \mathtt{s}_\sigma, \ldots, \mathtt{s}_n \}$ entre $\mathtt{T}_\Sigma$ et `T` (voir figure IV.16). On peut dire qu'il existe une dépendance de fréquence $\mathcal{D}_T^{T_\Sigma}$ entre $\mathtt{T}_\Sigma$ et `T` définie de la manière suivante:

$$\mathcal{D}_T^{T_\Sigma} = \bigvee_{\substack{s_\sigma \in \mathcal{E}_T \\ s_\sigma \in \mathcal{S}_{T_\Sigma}}} \mathcal{D}_T^{s_\sigma}$$

où,

- $\mathcal{E}_T$ est l'ensemble des signaux de sortie référencés dans les arbres d'implémentation, et le code de contrôle de la tâche `T`,
- $\mathcal{S}_{T_\Sigma}$ est l'ensemble des signaux de sortie produits par la fonction de calcul $\mathtt{T}_\Sigma$,
- $\mathcal{D}_T^{s_\sigma}$ est l'horloge de la dépendance pour le signal $\mathtt{s}_\sigma$ entre les tâches $\mathtt{T}_\Sigma$ et `T`.

L'accusé de réception qui sera envoyé de $\mathtt{T}_\Sigma$ et `T`, validant tous les liens de dépendance existant entre ces deux tâches, sera donc émis à l'horloge $\mathcal{D}_T^{T_\Sigma}$.

2. ATAMM est un modèle de graphe dataflow qui décrit l'exécution temps-réel des algorithmes de contrôle itératif et de traitement du signal sur les architectures dataflow multiprocesseurs

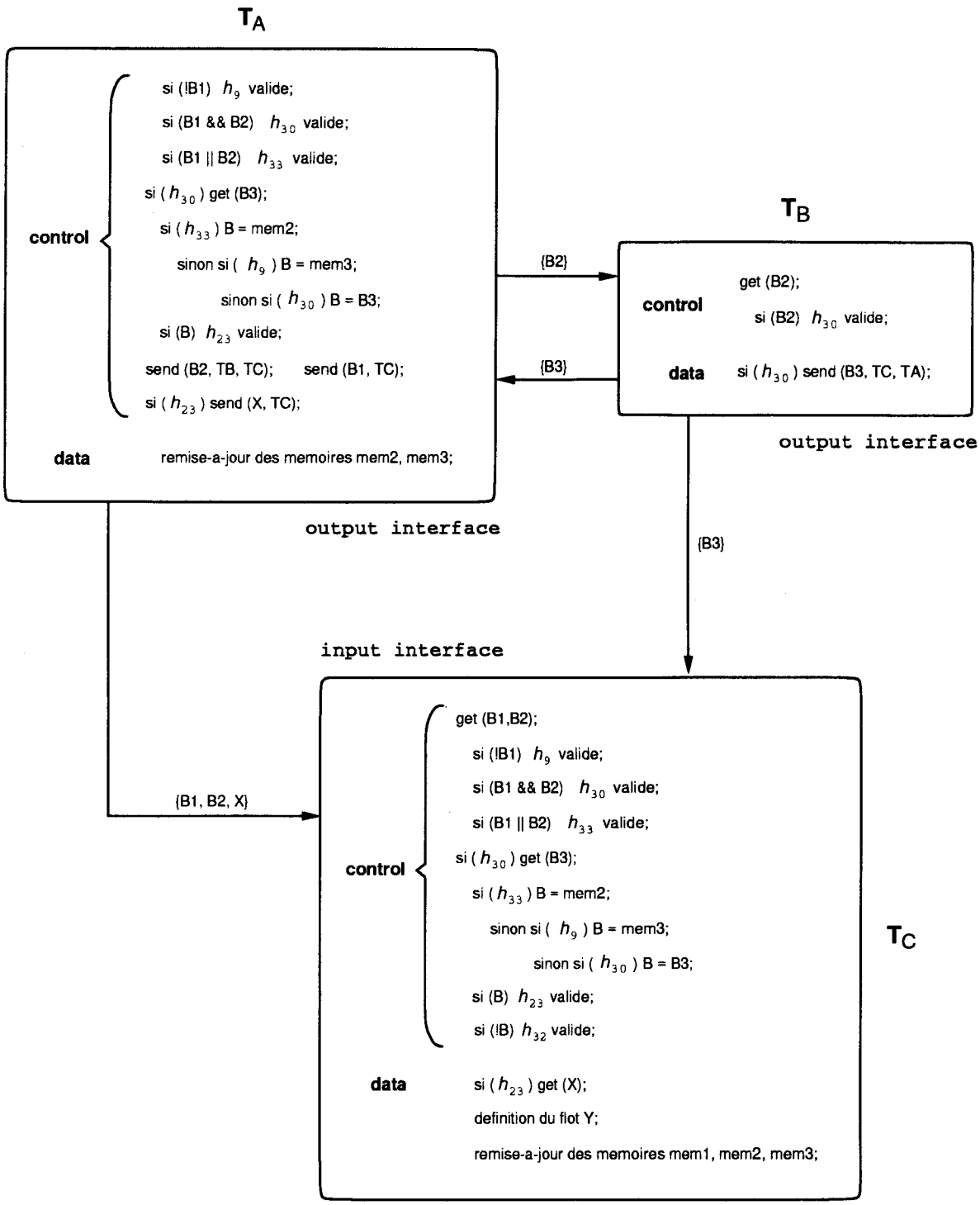


FIG. IV.15 - *Implémentation optimisée des interfaces induites par la définition de l'entrée* Y.

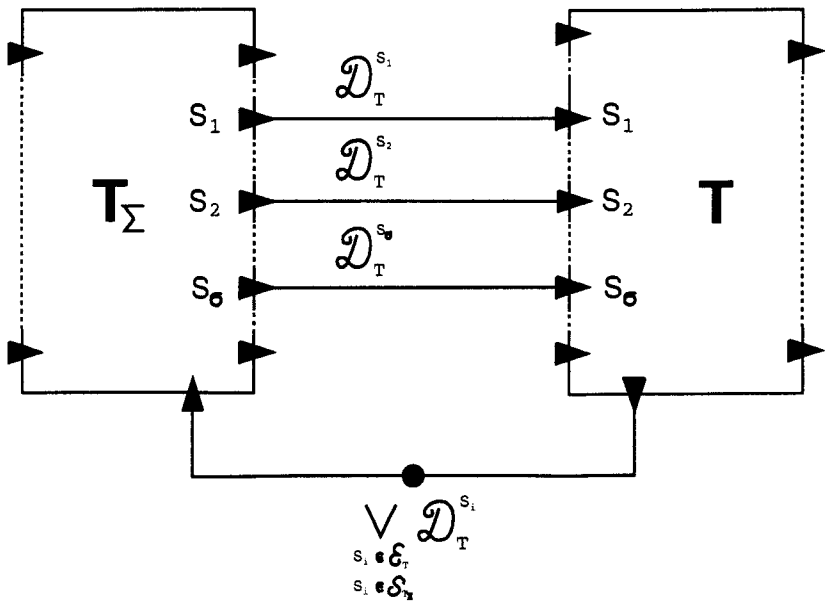


FIG. IV.16 - *Accusés réception.*

IV.5 Conclusion

À partir des formes normales déduites du processus de normalisation des arbres d'horloges et des expressions sur signaux de sortie, il est possible de définir les interfaces de communication des différentes tâches d'un système.

L'ensemble des signaux du système nécessaires au déclenchement des fonctions de calculs décrivent dans leur expression transformée les dépendances de calcul et de contrôle qui seront implémentées dans les interfaces d'entrée des tâches.

Ces mêmes expressions conditionnent également l'implémentation des interfaces de sortie des tâches amont.

Nous avons décrit sous forme d'arbres normaux et d'implémentation l'automatisation de ce processus. Ces derniers définissent en outre les besoins mémoires de chaque tâche. Le pseudo-code que nous avons dès lors utilisé dans nos exemples, peut être remplacé par des appels à une librairie de communication.

Chapitre V

Conclusion

Nous nous sommes intéressés dans le cadre de ce travail à l'implémentation des systèmes complexes de traitement temps-réel de l'image sur une machine hétérogène distribuée moyennement couplée. C'est-à-dire un réseau d'unités de traitement hétérogènes pouvant être reconfiguré pour accueillir de nouvelles unités fonctionnelles, et pour lequel les coûts de communication ne peuvent pas être ignorés. Les caractéristiques du modèle architectural qui nous a été imposé dans le cadre du projet SM-IMP, ont eu des répercussions importantes sur le modèle d'implémentation des applications:

- L'absence d'hypothèse sur le système d'échanges de données de l'architecture nous a contraint à éviter tout mécanisme centralisé ou semi-centralisé de gestion des dépendances de calcul et de contrôle au niveau de l'implémentation.
- La flexibilité de l'architecture nous a par ailleurs encouragé à donner le maximum d'autonomie aux tâches.

Notre approche combine les avantages respectifs des modèles synchrones et asynchrones: la phase de conception et validation des systèmes temps-réel est effectuée dans un contexte synchrone —l'implémentation proprement dîte étant réalisée selon un modèle CSP traditionnel.

Sous les hypothèses fortes d'*instantanéité* et de *simultanéité* décrites dans la théorie de la programmation synchrone, il est en effet possible:

- d'exprimer simplement les relations temporelles complexes qui apparaissent naturellement dans les grandes applications de traitement temps-réel de l'image,
- et d'effectuer des preuves automatiques des programmes.

Alors que le modèle CSP a été conçu directement en terme d'implémentation pour programmer les architectures distribuées que nous visons.

Le passage du squelette synchrone des applications vers l'implémentation asynchrone est réalisé au moyen d'un ensemble de processus formels que nous avons développés:

- **Le processus de normalisation.** Traitant des aspects de contrôle pur, le processus de normalisation associe à chaque horloge du programme une expression booléenne sur signaux.
- **Le processus de transformation.** Appliqué aux flots référencés en entrée des fonctions de calcul, et aux signaux intermédiaires apparaissant dans les formes normalisées d'horloges, il donne une expression sur signaux de sortie provenant de tâches amont.
- **Les arbres d'implémentation.** Définissent à la fois le code nécessaire à l'émission-réception des dépendances de calcul, ainsi qu'à la gestion des mémorisations.

- **Les arbres normaux.** Ils déterminent dans une partie préliminaire de contrôle des interfaces de communication des tâches, les expressions permettant de définir les instants de validité des dépendances de calcul.

Nous avons validé notre approche en générant *"à la main"* du code PVM à partir des arbres d'implémentation produits sur différents exemples. Nous présentons l'un d'entre eux en ANNEXE A.

Afin de permettre l'implémentation automatique de systèmes temps-réel sur les architectures hétérogènes distribués, les aspects novateurs de notre approche résident dans:

- l'implémentation de systèmes prouvés sous l'hypothèse de synchronisme afin de pas avoir à garantir a posteriori dans un contexte plus difficile la sûreté des programmes,
- la répartition complète des échanges de données entre les tâches en évitant ainsi de recourir à des mécanismes plus ou moins centralisés,
- le maintient d'un modèle d'exécution dicté par la disponibilité des données.

En définitive, à partir d'une vue unique du programme qui correspond à une perception plus agréable pour le développeur des problèmes liés à la conception des systèmes, nous répartissons complètement entre les différentes tâches du programme la totalité des échanges de données (calcul et contrôle). Cette répartition est effectuée de manière à ce que chaque tâche soit directement connectée à un sous-ensemble d'autres tâches du système par l'intermédiaire de ses dépendances en amont et aval.
L'autonomie qui est ainsi obtenue par les tâches correspond sur le plan matériel à la flexibilité attendue de l'architecture.

Le domaine applicatif que nous avons étudié dans le cadre du projet SM-IMP est symptomatique de la classe de systèmes temps-réel concernés par ce travail. Il s'agit en effet de systèmes dit mixtes présentant les particularités suivantes:

- Ils sont dotés d'une lourde partie algorithmique et non coopérante au niveau fonction qui dénote un haut niveau de granularité des opérations ou tâches élémentaires.
- Au niveau application, l'agencement logique et temporel de ces tâches est soumis à un système complexe de contrôle et de commande.

De manière générale, notre approche concerne tous les systèmes temps-réel mixtes dont les aspects transformationnels sont suffisamment importants pour justifier le masquage des coûts engendrés par les échanges de données. Dans ce cas, leur implémentation sur une architecture hétérogène distribuée moyennement couplée peut être entrevue.

Annexe A

Application

Nous présentons à titre d'illustration dans cette SECTION, l'implémentation via PVM de l'application MPEG-2 dans sa version *"simple-profile"* (voir figure A.1). Nous avons déjà largement décrit cette application dans le chapitre II, nous y renvoyons donc le lecteur.

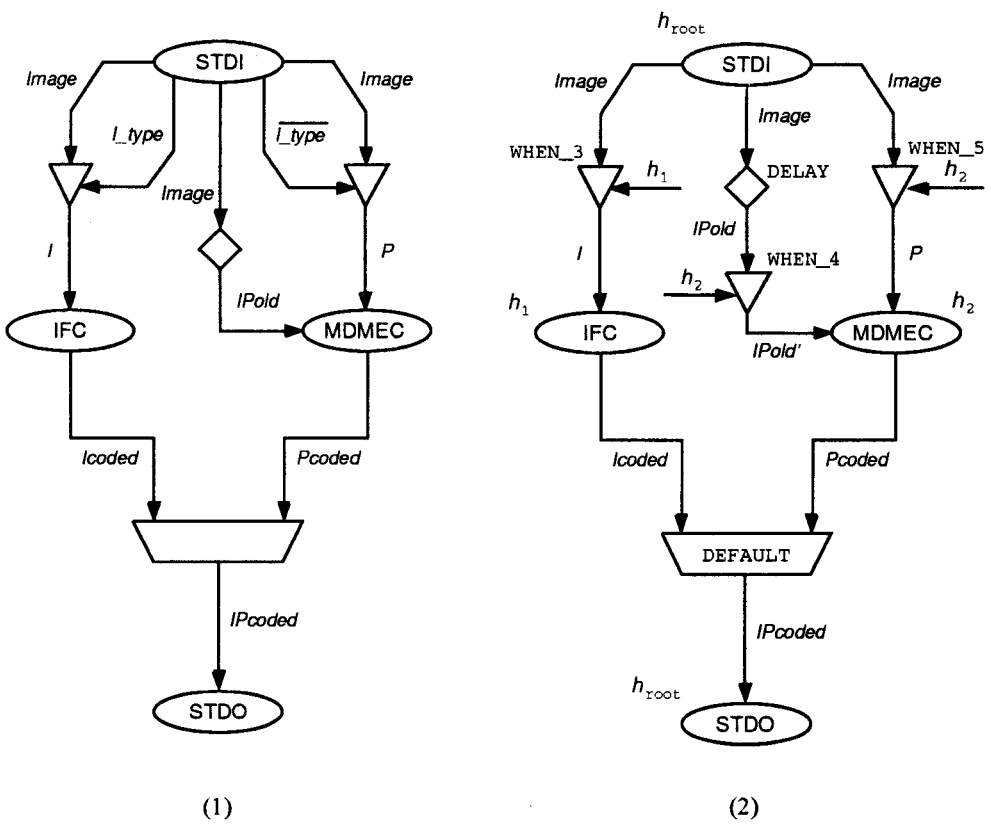


FIG. A.1 - *MPEG-2 encodeur "simple-profile". (1) Graphe data-flow de l'application. (2) Graphe des dépendances de calcul.*

Résolution du système

La résolution de ce système passe tout d'abord par la compilation SIGNAL de l'application. Le code SIGNAL correspondant à ce programme est détaillé dans une première sous-partie (SECTION A). Nous montrons ensuite de quelle manière le résultat de la compilation est extrait d'un ensemble de fichiers générés par le compilateur (SECTION A). Enfin les interfaces produites sont décrites (SECTION A).

Le code SIGNAL

Le code SIGNAL correspondant à l'implémentation du codeur est donné en page 117 (le lecteur peut se référer à [BCG+93] pour obtenir la syntaxe complète du langage).

Les tâches `STDI` (STanDard Input), `IFC`, `MDMEC` et `STDO` (STanDard Output) correspondent à des fonctions externes du langage, implicitement étendues par le modèle pour opérer sur des flots (instructions `17` à `32`).

On suppose que les séquences d'images (dénotées `IMAGE`), ainsi qu'un flot booléen de contrôle dénoté `I_type`, sont produits par la fonction `STDI` de gestion des entrées standard. Le booléen `I_type` précise pour chaque nouvelle trame captée si elle est de type I ou P, en prenant la valeur *vrai* dans le premier cas, et *faux* dans le second. Le fait que les signaux `IMAGE` et `I_type` soient issus de la même tâche (instruction `5`), garantit leur synchronisation implicite.

En fonction du signal de contrôle, deux flots distincts d'images sont extraits: la sous-séquence des trames de type I (Instruction `6`), et la sous-séquence de trames de type P (Instruction `7`).

La mémorisation de la trame précédente est effectuée par la variable `IPold` (instruction `8`). Cette mémorisation est rendue nécessaire pour toute image courante, dans la mesure où le codage de la trame suivante peut éventuellement requérir cette valeur (si la trame suivante est de type P).

La fonction `IFC` prend une trame de type I en entrée et produit une image codée (dénotée `Icoded` dans le source du programme) en sortie (Instruction `9`).

L'algorithme de codage `MDMEC` (Instruction `10`) prend quant à lui deux entrées: la trame de type P courante, et l'image précédente (dénotée `IPold`). Pour les besoins du calcul d'horloge, la trame `IPold` est explicitement synchronisée avec la sous-séquence de type P extraite du flot d'images (`IPold when event(P)`). Cette instruction traduit la nécessité en SIGNAL que toutes les entrées d'une tâche soient parfaitement synchrones.

Finalement, l'interface de sortie `STDO` traite un seul flot d'images codées, qui à un instant donné peut être de manière exclusive, de type I ou P (Instruction `11`).

```
1  -  process CODER=
2  -        { ?
3  -          ! }
4  -
5  -        (| {IMAGE, I_type} := STDI() {}
6  -         | I               := IMAGE when I_type
7  -         | P               := IMAGE when not I_type
8  -         | IPold           := IMAGE$1
9  -         | {Icoded}        := IFC(){I}
10 -         | {Pcoded}        := MDMEC(){IPold when event(P), P}
11 -         |                    STDO(){Icoded default Pcoded}
12 -         |)
13 -        where
14 -             integer IMAGE, I, P, I_coded, P_coded, IPold;
15 -             logical I_type
16 -
17 -        function STDI=
18 -               { ?
19 -                 ! integer IMAGE;
20 -                   logical I_type };
21 -
22 -        function IFC=
23 -               { ? integer IMAGE
24 -                 ! integer IMAGE_coded };
25 -
26 -        function MDMEC=
27 -               { ? integer IMAGE_ref, IMAGE_src
28 -                 ! integer IMAGE_coded          };
29 -
30 -        function STDO=
31 -               { ? integer IMAGE_coded
32 -                 !                      }
33 -  end %CODER%
```

Résultats de la compilation

Le calcul d'horloge effectué par le compilateur SIGNAL produit différents résultats dont un ensemble de trois fichiers dénotés SYNDEX. Ces fichiers donnent à la fois une description du Graphe des Dépendances Conditionnées mais aussi l'arbre d'horloges associés au programme. Dans un souci de clarté, les identificateurs utilisés dans les fichiers SYNDEX ont été modifiés pour correspondre aux signaux originels du programme.

Graphe des Dépendances Conditionnées

La description du Graphe des Dépendances Conditionnées réside en grande partie dans les deux premiers fichiers SYNDEX. Le fichier SYNDEX 1 répertorie les différents nœuds du graphe, alors que le fichier SYNDEX 2 décrit les dépendances de données existant entre ces nœuds.

Syndex 1 On trouve trois types de nœuds dans le fichier SYNDEX 1: les nœuds de calcul d'horloges — les fonctions externes du programmes — les opérations de traitement du signal.

```
1  -  %----------
2  -  % SYNDEX 1
3  -  %----------
4  -
5  -  (hinput H_ROOT h_root event !h_root)
6  -  (htt H_1 logical ?i_type !h_1)
7  -  (hff H_2 logical ?i_type !h_2)
8  -  (hsub H_3 event ?h_1 ?h_2 !h_3)
9  -
10 -  (function STDI integer !image, logical !i_type)
11 -  (function STDO integer ?ipcoded)
12 -  (function IFC integer ?i,integer !icoded)
13 -  (function MDMEC integer ?ipold',integer ?p,
14 -       integer !pcoded)
15 -  (memory DELAY integer ?image $1 !ipold)
16 -  (when WHEN_1 event ?h_1, integer ?icoded !icoded')
17 -  (when WHEN_2 event ?h_3, integer ?pcoded !pcoded')
18 -  (default DEFAULT integer ?icoded' ?pcoded' !ipcoded)
19 -  (when WHEN_3 event ?h_1, integer ?image !i)
20 -  (when WHEN_4 event ?h_2, integer ?ipold !ipold')
21 -  (when WHEN_5 event ?h_2, integer ?image !p)
```

1. **Les nœuds de calcul d'horloges.** Dans notre exemple, ils figurent entre les lignes 5 à 8 du fichier SYNDEX 1. Plusieurs mots clefs sont associés à la spécification d'un nœud de calcul d'horloges:

   - `hinput` produit l'horloge d'entrée ou racine du programme,
   - `htt` est équivalent à l'instruction `event` du langage, et produit une horloge valide lorsque le signal booléen en entrée est défini et porte la valeur *vrai*,
   - `hff` prend un signal booléen en entrée et produit une horloge valide lorsque ce flot porte la valeur *faux*..
   - `hsub` effectue la "soustraction" d'un couple d'horloges en entrée,

   Le symbole "?" précède la spécification d'un signal d'entrée, et le symbole "!" d'un signal de sortie.

2. **Les fonctions externes du programmes.** Elles sont introduites par l'identificateur `function` (cf. lignes 10 à 14).

3. **Les opérations de traitement du signal.** On retrouve les opérations standards: extraction d'une séquence de valeurs (nœud `when`) — entrelacement de signaux (nœud `default`) — retard sur signal (nœud `memory`).

Syndex 2 Les connexions du Graphe des Dépendances Conditionnées sont décrites dans le fichier `Syndex 2`. On trouvera par ailleurs dans le fichier `Syndex 1` (lignes 1 à 21) la description des différents nœuds du graphe. L'instruction `connect` du SYNDEX 2 précise l'existence d'une dépendance de donnée dénotée `signal` entre un `NOEUD_AMONT` et un `NOEUD_AVAL`.

Pour une meilleure clarté, les identificateurs de nœuds du graphe sont notés en majuscules, cependant que les signaux échangés par ces nœuds sont spécifiés en minuscules.

Le Graphe des Dépendances Conditionnées est ainsi décrit par une succession de connexions entre nœuds du graphe (voir SYNDEX 2 du codeur d'images page 119).

```
1  -  %----------
2  -  % SYNDEX 2
3  -  %----------
4  -
5  -  (connect STDI/i_type H_1/i_type)
6  -  (connect STDI/i_type H_2/i_type)
7  -  (connect H_1/h_1 H_3/h_1)
8  -  (connect H_2/h_2 H_3/h_3)
9  -  (connect STDI/image DELAY/image)
10 -  (connect IFC/icoded WHEN_1/icoded)
11 -  (connect H_1/h_1 WHEN_1/h_1)
12 -  (connect MDMEC/pcoded WHEN_2/pcoded)
13 -  (connect H_3/h_3 WHEN_2/h_3)
14 -  (connect WHEN_1/icoded' DEFAULT/icoded')
15 -  (connect WHEN_2/pcoded' DEFAULT/pcoded')
16 -  (connect DEFAULT/ipcoded STDO/ipcoded)
17 -  (connect STDI/image WHEN_3/image)
18 -  (connect H_1/h_1 WHEN_3/h_1)
19 -  (connect WHEN_3/i IFC/i)
20 -  (connect DELAY/ipold WHEN_4/ipold)
21 -  (connect H_2/h_2 WHEN_4/h_2)
22 -  (connect STDI/image WHEN_5/image)
23 -  (connect H_2/h_2 WHEN_5/h_2)
24 -  (connect WHEN_4/ipold' MDMEC/ipold')
25 -  (connect WHEN_5/p MDMEC/p)
26 -
27 -  (execroots  H_ROOT)
```

Le Graphe des Dépendances Conditionnées correspondant au fichier SYNDEX 2 est donné en figure A.2.

Les nœuds du GDC représentés par des ovales blancs correspondent aux fonctions externes du programme (`STDI`, `STDO`, `IFC`, et `MDMEC`). Les nœuds de calcul d'horloges sont figurés par des ovales grisés (`H_1`, `H_2`, et `H_3`). Enfin, les opérateurs de traitement du signal (extraction: `WHEN_`i, entrelacement: `DEFAULT` et retard: `DELAY`) sont respectivement représentés par des triangles, losanges, et trapèzes.

Dans le GDC, toutes les dépendances de données sont conditionnées par leurs horloges respectives de validité. Le calcul d'horloge de SIGNAL a ainsi permis de déterminer l'horloge exclusive d'utilisation de chacun des signaux du programme. Nous avons représenté en figure A.3 les contraintes temporelles définies par le calcul d'horloge et portant sur la définition de chacun des signaux.

Trois horloges dénotées h_1, h_2, et h_3 ont été sous-échantillonnées à partir de l'horloge mère du programme: h_{root}.

Arbre d'horloges

La seconde construction importante issue du calcul d'horloges est l'arbre d'horloges. Sur l'exemple du codeur d'images temps-réel, trois horloges (h_1, h_2, et h_3) ont été synthétisées en plus de l'horloge racine du programme (h_{root}).

Nous ne nous intéressons qu'aux programmes endochrones, c'est-à-dire ceux pour lesquels un arbre unique d'horloges peut être synthétisé. Les nœuds sans arc entrant (il s'agira le plus souvent

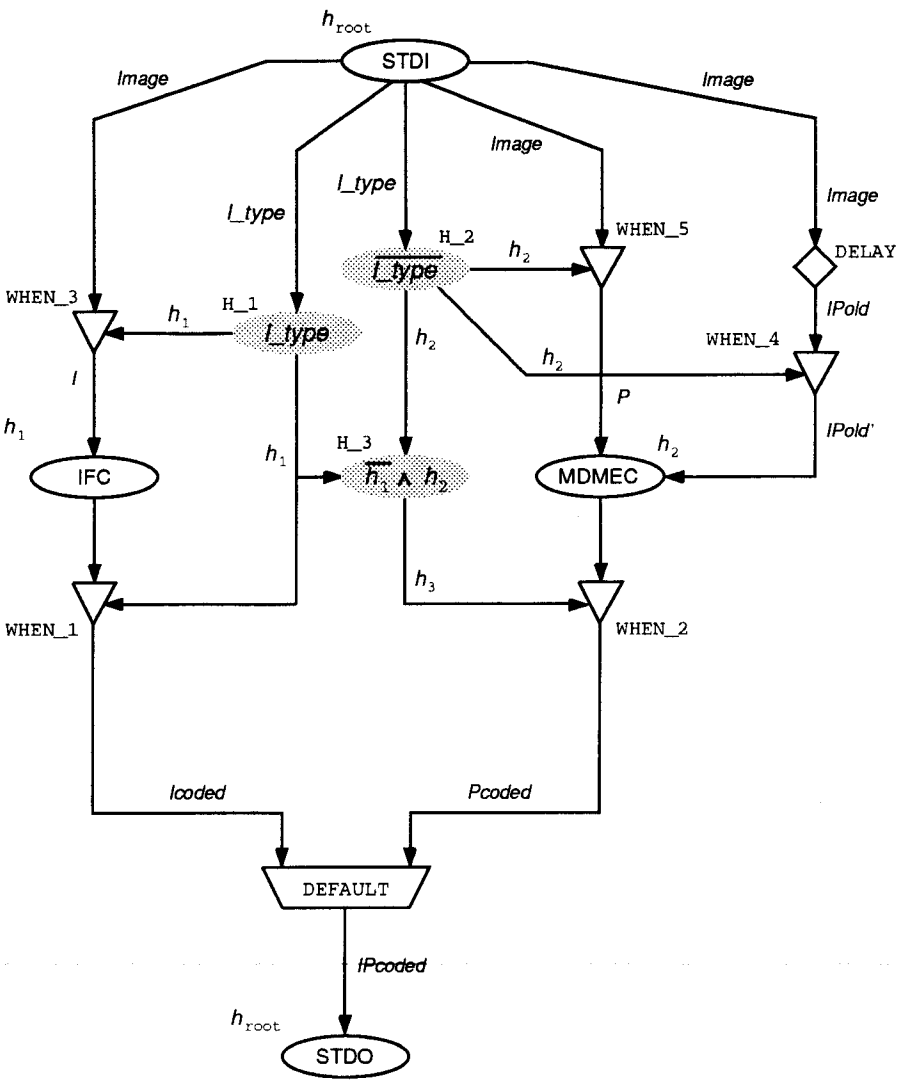


FIG. A.2 - *Graphe des Dépendances Conditionnées.*

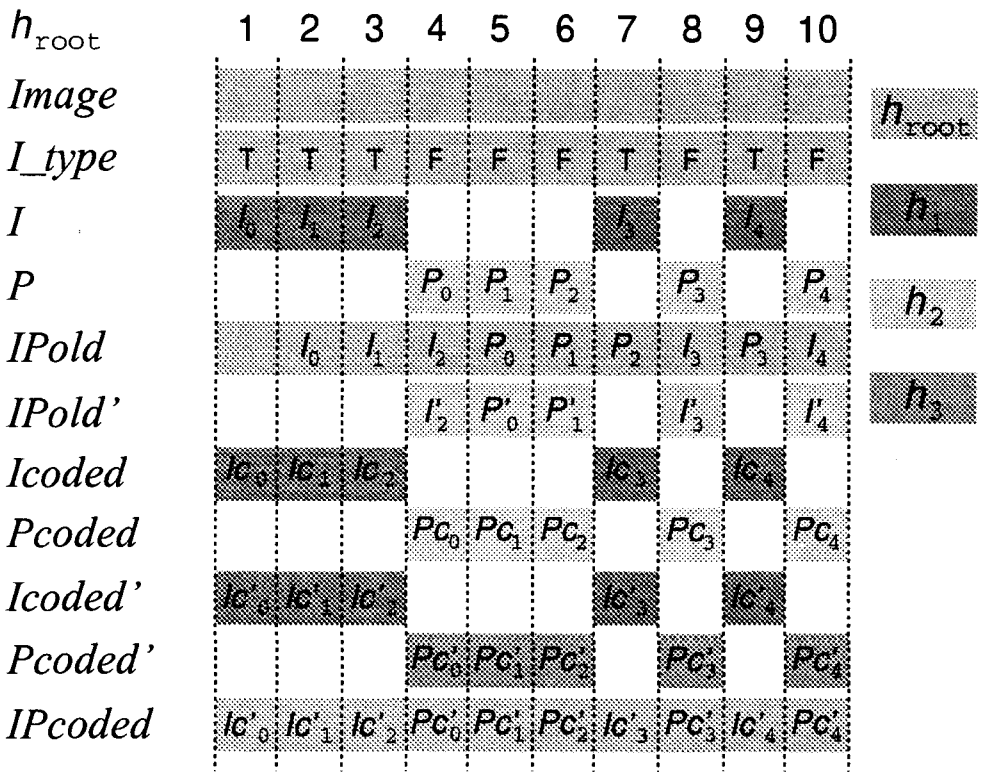


FIG. A.3 - *Horloge de validité des différents signaux du GDC sur un flot hypothétique d'entrées.*

de tâches d'acquisition, comme par exemple des caméras, magnétoscopes, etc) sont implicitement activés à l'horloge racine de l'arbre d'horloges (cf. `STDI`).

En conjonction avec les informations données dans le fichier SYNDEX 1 (lignes 5 à 9), les données contenues dans le fichier SYNDEX 3 permettent de reconstituer l'arbre des horloges.

Le fichier SYNDEX 3 spécifie les différentes inclusions d'événements qui ont été calculées par le compilateur SIGNAL. Par exemple sont cadencés à l'horloge h_{root} émise par le nœud `H_ROOT` les nœuds suivants: `STDI`, `DELAY`, `DEFAULT`, `H_1`, `H_2`, `H_3`, et `STDO`.

```
1 - %----------
2 - % SYNDEX 3
3 - %----------
4 -
5 - (exec H_3/h_3 WHEN_2)
6 - (exec H_2/h_2 WHEN_5 MDMEC WHEN_4)
7 - (exec H_1/h_1 WHEN_3 IFC WHEN_1)
8 - (exec H_ROOT/h_root STDI DELAY DEFAULT H_1 H_2 H_3 STDO)
```

Ces informations sont représentées en figure A.4 sous la formes de sous-ensembles regroupant les tâches synchrones du programme avec leur horloge d'utilisation. Le programme étant endochrone, ces sous-ensembles sont tous inclus dans l'horloge racine.

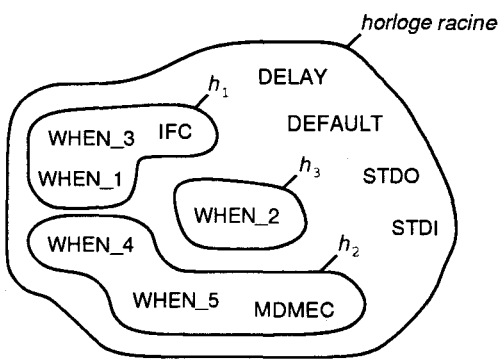


FIG. A.4 - *Inclusion des événements.*

Les informations disposées dans les fichiers SYNDEX 3 et SYNDEX 1 permettent donc de construire l'arbre d'horloges donné en figure A.5, et de lui appliquer le processus de normalisation.

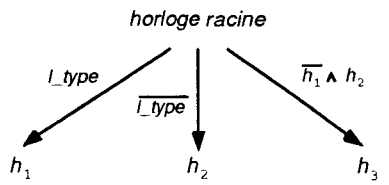


FIG. A.5 - *Arbre d'horloges.*

Transformations

Les transformation portent sur l'arbre des horloges (processus de normalisation), et sur les dépendances de calcul (transformation des expressions).

Normalisation de l'arbre des horloges

Nous avons déjà détaillé dans le chapitre 1, le résultat obtenu après la normalisation des horloges de l'application:

$\mathcal{VH}$	$\mathcal{VHC}$
$\emptyset$	$h_1 \xrightarrow{h_{root}}$ [I_type] $h_2 \xrightarrow{h_{root}}$ [I_type] $h_3 \equiv h_2$

Transformation des expressions

Les expressions définissant les dépendances de calcul sont données par le système suivant:

```
1- I       =  Image when h1
2- P       =  Image when h2
3- IPold   =  Image$1
4- IPold'  =  IPold when h2
5- IPcoded =  Icoded default Pcoded
```

avec,

- $\mathcal{S}$ = { Image, Icoded, Pcoded }
- $\mathcal{E}$ = { I, P, IPold', IPcoded }
- $\mathcal{I}$ = { IPold }

Le processus de transformation est ici trivial.

Implémentation des interfaces d'entrée

Des interfaces d'entrée sont définies pour toutes les tâches du système qui possèdent des arc entrant: IFC, MDMEC, et STDO.

IFC. La fonction de calcul IFC est définie par un seul flot, $\mathcal{E}_{IFC}$ = { I }.

Partie contrôle

```
get(I_type);
si (I_type) h1 valide;
```

Partie donnée

Définition des ports d'entrée: `si (h1) get(Image);`

Définition du flot I: `si (h1) I = Image;`

MDMEC. La fonction de calcul IFC est définie par deux flots, $\mathcal{E}_{MDMEC}$ = { P, IPold' }. Les arbres d'implémentation de ces entrées sont donnés en figure A.7.

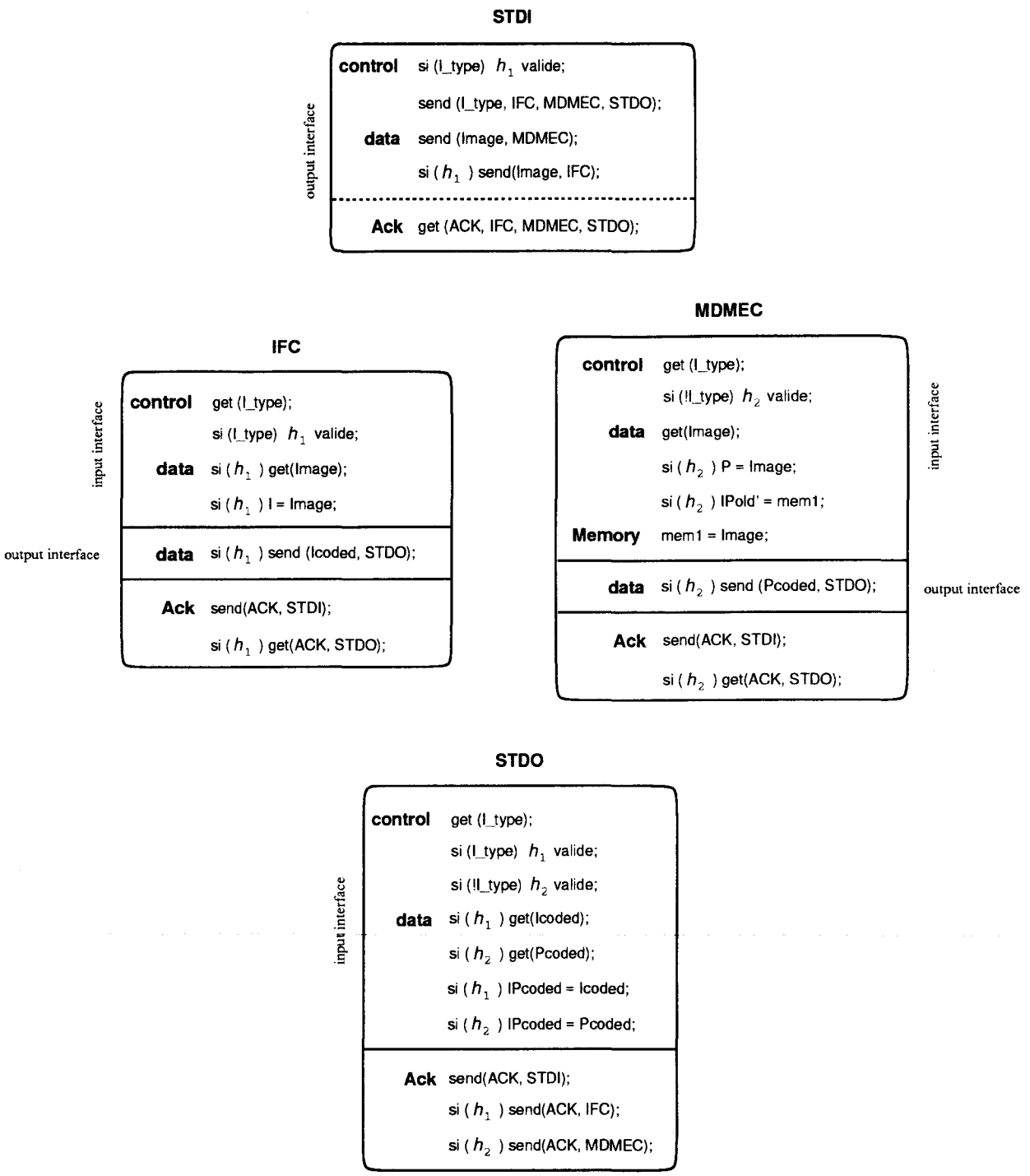


FIG. A.6 - *MPEG-2 encodeur "simple-profile". Implémentation des interfaces.*

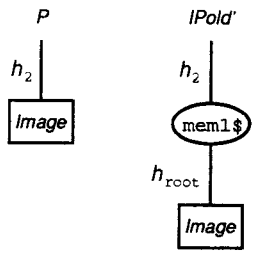


FIG. A.7 - *Arbres d'implémentation des entrées de MDMEC.*

Une dépendance de donnée de fréquence $\mathcal{D}^{Image}_{MDMEC} = h_2 \vee h_{root}$ existe sur le signal **Image** entre les tâches **MDMEC** et **STDI**.

Partie contrôle

`get(I_type);`
`si (!I_type)` h_2 valide;

Partie donnée

Définition des ports d'entrée: `get(Image);`

Définition du flot `P`: `si (`h_2`) P = Image;`
Définition du flot `IPold'`: `si (`h_2`) IPold' = mem1;`

Remise à jour de la mémoire **mem1**: `mem1 = Image;`

STDO. La fonction de calcul IFC est définie par un seul flot, $\mathcal{E}_{IFC} = \{$ **IPcoded** $\}$. Le résultat est également trivial:

Partie contrôle

`get(I_type);`
`si (I_type)` h_1 valide;
`si (!I_type)` h_2 valide;

Partie donnée

Définition des ports d'entrée: `si (`h_1`) get(Icoded);`
`si (`h_2`) get(Pcoded);`

Définition du flot `IPcoded`: `si (`h_1`) IPcoded = Icoded;`
`si (`h_2`) IPcoded = Pcoded;`

Implémentation des interfaces de sortie

Les interfaces de sortie sont implémentées en fonction des flots référencés dans les différentes expressions d'entrée qui viennent d'etre définies.

IFC. Une seule dépendance de fréquence $\mathcal{D}^{Icoded}_{STDO} = h_1$ a été définie entre les tâches **IFC** et **STDO**.

`si (`h_1`) send(Icoded, STDO)`

MDMEC. Une dépendance de fréquence $\mathcal{D}^{Pcoded}_{STDO} = h_2$ a été définie entre les tâches **MDMEC** et **STDO**.

`si (`h_2`) send(Pcoded, STDO)`

STDI. Le signal `I_type` est requis par les autres tâches de l'application, ce signal est produit à l'horloge racine. Nous avons donc en sortie de `STDI`:

```
send(I_type, IFC, MDMEC, STDO)
```

Le signal `Image` est quant à lui référencé dans les expressions d'entrée de `IFC` et `MDMEC` aux horloges respectives $\mathcal{D}^{Image}_{IFC} = h_1$, et $\mathcal{D}^{Image}_{MDMEC} = h_{root}$:

```
send(Image, MDMEC)
if (h1) send(Image, IFC)
```

La définition de l'horloge h_1 est requise dans l'interface de contrôle de sortie de `STDI`:

`si (I_type)` h_1 valide;

Accusés de réception

Les fréquences de dépendances suivantes peuvent être déterminées à partir des interfaces qui ont été implémentées:

$$\begin{array}{lcl} \mathcal{D}^{STDI}_{IFC} & = & h_{root} \\ \mathcal{D}^{STDI}_{MDMEC} & = & h_{root} \\ \mathcal{D}^{STDI}_{STDO} & = & h_{root} \\ \mathcal{D}^{IFC}_{STDO} & = & h_1 \\ \mathcal{D}^{MDMEC}_{STDO} & = & h_2 \end{array}$$

Implémentation sous PVM

Une implémentation de l'application utilisant la librairie de communication PVM est proposée dans cette SECTION. Le code PVM a été produit à partir d'une version optimisée des interfaces donnée en figure A.8. Une simplification notable du code de contrôle de la tâche `IFC` s'est en effet avérée possible.

L'implémentation sous PVM de l'application est décomposée en deux parties:

1. Un processus maître est généré. Celui-ci est simplement chargé d'instancier des différentes tâches de calcul de l'application. Dès que c'est fait il disparait.
2. L'application proprement dite est composée d'autant de tâches que de fonctions de calcul. Celles-ci s'exécutent indéfiniment tant que le système est approvisionné en flots d'entrée.

```
/*
 * MASTER PROGRAM
*/

#include <pvm3.h>
#include <stdio.h>

#include "coder-tags.h"

main()
{
  int tids[4];
  int info;

  pvm_catchout(stdout);
```

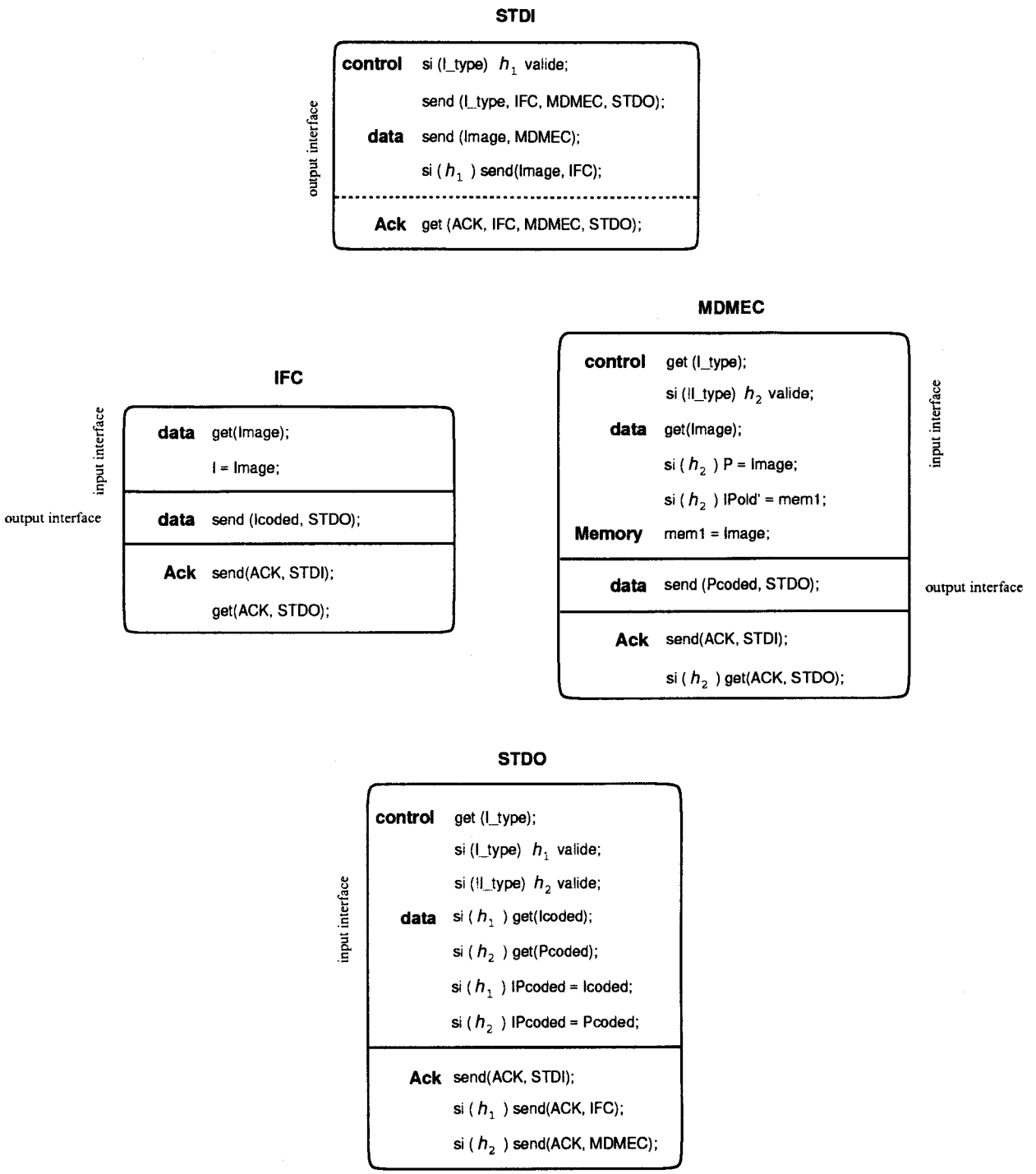


FIG. A.8 - *MPEG-2 encodeur "simple-profile". Implémentation optimisée des interfaces.*

```
  if ( ( pvm_spawn("coder-STDI",  (char**)0, 0, "", 1, &tids[0]) == 1 ) &&
       ( pvm_spawn("coder-MDMEC", (char**)0, 0, "", 1, &tids[1]) == 1 ) &&
       ( pvm_spawn("coder-IFC",   (char**)0, 0, "", 1, &tids[2]) == 1 ) &&
       ( pvm_spawn("coder-STDO",  (char**)0, 0, "", 1, &tids[3]) == 1 )    )
    {
      printf("[MASTER/%d] Tasks spawned\n", pvm_mytid());
      printf("[MASTER/%d] STDI  --> %d\n", pvm_mytid(), tids[0]);
      printf("[MASTER/%d] MDMEC --> %d\n", pvm_mytid(), tids[1]);
      printf("[MASTER/%d] IFC   --> %d\n", pvm_mytid(), tids[2]);
      printf("[MASTER/%d] STDO  --> %d\n", pvm_mytid(), tids[3]);

      /* STDI dependencies */
      pvm_initsend(PvmDataRaw);
      pvm_pkint(&tids[1], 3, 1);
      pvm_send(tids[0], DD, info);

      /* MDMEC dependencies */
      pvm_initsend(PvmDataRaw);
      pvm_pkint(&tids[0], 1, 1);
      pvm_send(tids[1], UD, info);

      pvm_initsend(PvmDataRaw);
      pvm_pkint(&tids[3], 1, 1);
      pvm_send(tids[1], DD, info);

      /* IFC dependencies */
      pvm_initsend(PvmDataRaw);
      pvm_pkint(&tids[0], 1, 1);
      pvm_send(tids[2], UD, info);

      pvm_initsend(PvmDataRaw);
      pvm_pkint(&tids[3], 1, 1);
      pvm_send(tids[2], DD, info);

      /* STDO dependencies */
      pvm_initsend(PvmDataRaw);
      pvm_pkint(&tids[0], 3, 1);
      pvm_send(tids[3], UD, info);

    }
  else
      printf("[MASTER/%d] Can't start tasks\n", pvm_mytid());

  pvm_exit(0);
}

/*
 * STandard Input PROGRAM
*/

#include <pvm3.h>
#include <stdio.h>
```

```
#include "coder-tags.h"

main()
{
  int  downstream[3];
  int  info, i;

  pvm_catchout(stdout);

  /* get downstream dependencies */
  pvm_recv(pvm_parent(), DD);
  pvm_upkint(downstream, 3, 1);
  /* 0 ----> MDMEC
   * 1 ----> IFC
   * 2 ----> STDO
  */

  while (1)
    {
      /* CALL FOR STDI EXTERNAL FUNCTION  */
      /* return values for: image, i_type */

      /* send "image" to MDMEC */
      pvm_initsend(PvmDataRaw);
      pvm_pkint(&image, 1, 1);
      pvm_send(downstream[0], SIGNAL_IMAGE, info);

      /* send "i_type" to MDMEC */
      pvm_initsend(PvmDataRaw);
      pvm_pkint(&i_type, 1, 1);
      pvm_send(downstream[0], SIGNAL_I_TYPE, info);

      /* send "i_type" to STDO */
      pvm_initsend(PvmDataRaw);
      pvm_pkint(&i_type, 1, 1);
      pvm_send(downstream[2], SIGNAL_I_TYPE, info);

      if ( i_type )
        {
          /* send "image" to IFC */
          pvm_initsend(PvmDataRaw);
          pvm_pkint(&image, 1, 1);
          pvm_send(downstream[1], SIGNAL_IMAGE, info);

          /* waiting for acknowledgment */
          pvm_recv(downstream[1], ACK_IFC_2_STDI);
        }

      /* waiting for acknowledgment */
      pvm_recv(downstream[0], ACK_MDMEC_2_STDI);

      /* waiting for acknowledgment */
      pvm_recv(downstream[2], ACK_STDO_2_STDI);
    }
```

```
  pvm_exit(0);
}

/*
 * Mono-Dimensional Motion Estimator Coding PROGRAM
*/

#include <pvm3.h>
#include <stdio.h>

#include "coder-tags.h"

main()
{
  int  upstream;
  int  downstream;
  int  info, i;
  int  p_num=0;
  int  i_num=0;
  int  ip_num;

  pvm_catchout(stdout);

  /* get upstream dependencies */
  pvm_recv(pvm_parent(), UD);
  pvm_upkint(&upstream, 1, 1);

  /* get downstream dependencies */
  pvm_recv(pvm_parent(), DD);
  pvm_upkint(&downstream, 1, 1);


  while (1)
    {
      /* get "image" from STDI */
      pvm_recv(upstream, SIGNAL_IMAGE);
      pvm_upkint(&image, 1, 1);

      /* get "i_type" from STDI */
      pvm_recv(upstream, SIGNAL_I_TYPE);
      pvm_upkint(&i_type, 1, 1);

      if !( i_type )
        {
          /* CALL FOR MDMEC(ip_old,image) EXTERNAL FUNCTION */
          /* return value for: p_coded                      */

          /* send pcoded to STDO */
          pvm_initsend(PvmDataRaw);
          pvm_pkint(&p_coded, 1, 1);
          pvm_send(downstream, SIGNAL_P_CODED, info);

          /* waiting for acknowledgment */
```

```
          pvm_recv(downstream, ACK_STDO_2_MDMEC);
        }

      /* replace in memory */
      ip_old = image;

      /* send acknowledgement to STDI */
      pvm_initsend(PvmDataRaw);
      pvm_send(upstream, ACK_MDMEC_2_STDI, info);
    }

  pvm_exit(0);
}

/*
 * Intra-Frame Coding PROGRAM
*/

#include <pvm3.h>
#include <stdio.h>

#include "coder-tags.h"

main()
{
  int  upstream;
  int  downstream;
  int  info, i;

  pvm_catchout(stdout);

  /* get upstream dependencies */
  pvm_recv(pvm_parent(), UD);
  pvm_upkint(&upstream, 1, 1);

  /* get downstream dependencies */
  pvm_recv(pvm_parent(), DD);
  pvm_upkint(&downstream, 1, 1);

  while (1)
    {
      /* get "image" from STDI */
      pvm_recv(upstream, SIGNAL_IMAGE);
      pvm_upkint(&image, 1, 1);

      /* CALL FOR IFC(image) EXTERNAL FUNCTION */
      /* return value for: pcoded              */

      /* send "i_coded" to STDO */
      pvm_initsend(PvmDataRaw);
      pvm_pkint(&i_coded, 1, 1);
      pvm_send(downstream, SIGNAL_I_CODED, info);

      /* send acknowledgement to STDI */
```

```
      pvm_initsend(PvmDataRaw);
      pvm_send(upstream, ACK_IFC_2_STDI, info);

      /* waiting for acknowledgment */
      pvm_recv(downstream, ACK_STDO_2_IFC);
    }

  pvm_exit(0);
}

/*
 * STandard Output PROGRAM
*/

#include <pvm3.h>
#include <stdio.h>

#include "coder-tags.h"

main()
{
  int  upstream[3];
  int  info, i;

  pvm_catchout(stdout);

  /* get upstream dependencies */
  pvm_recv(pvm_parent(), UD);
  pvm_upkint(upstream, 3, 1);
  /* 0 ----> STDI
   * 1 ----> MDMEC
   * 2 ----> IFC
  */

  while (1)
    {
      /* get "i_type" from STDI */
      pvm_recv(upstream[0], SIGNAL_I_TYPE);
      pvm_upkint(&i_type, 1, 1);

      if (i_type )
        {
          /* get "i_coded" from IFC */
          pvm_recv(upstream[2], SIGNAL_I_CODED);
          pvm_upkint(&image_coded, 1, 1);

          /* send acknowledgement to IFC */
          pvm_initsend(PvmDataRaw);
          pvm_send(upstream[2], ACK_STDO_2_IFC, info);
        }
      else
        {
          /* get "p_coded" from MDMEC */
          pvm_recv(upstream[1], SIGNAL_P_CODED);
```

```
            pvm_upkint(&image_coded, 1, 1);

            /* send acknowledgement to IFC */
            pvm_initsend(PvmDataRaw);
            pvm_send(upstream[1], ACK_STDO_2_MDMEC, info);
          }

        /* send acknowledgement to STDI */
        pvm_initsend(PvmDataRaw);
        pvm_send(upstream[0], ACK_STDO_2_STDI, info);

        /* CALL FOR STDO(image_coded) EXTERNAL FUNCTION */
      }


    pvm_exit(0);
}
```

Annexe B

SIGNAL

Les fondements de SIGNAL sont proches du concept de flot, tel qu'il est défini dans la sémantiques des langages data-flow [LGG90] à assignation unique. Le langage SIGNAL [BLG90] est construit sur un noyau minimal d'opérateurs, et complet dans le sens où il suffit à la spécification des applications temps-réel complexes. Ce qui permet de définir simplement la sémantique formelle du langage et de dériver facilement de nouveaux opérateurs.

Le langage noyau est constitué: d'expressions sur signaux constituant les processus élémentaires, et d'expressions sur processus.

Processus élémentaires

Tout signal de nom `x`, hormis les entrées du système, est défini par une expression sur signaux ou constantes `E` notée `x := E`. Les variables de nom y_i apparaissant dans `E` sont les entrées de cette expression qui constitue un *processus élémentaire*; la variable `x` est la sortie de ce même processus élémentaire.

Ainsi par exemple, le signal `x` de la figure B.1 est définit par une expression `E` dont les entrées sont un couple (y_1, y_2) de signaux.

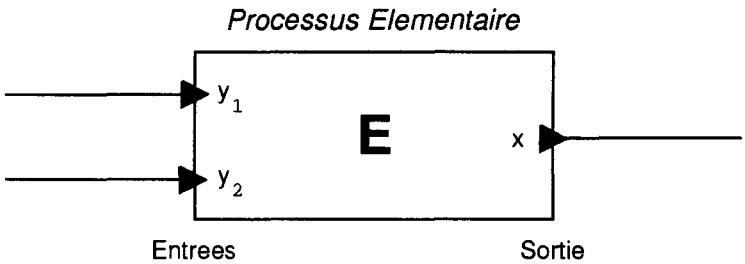


FIG. B.1 - *Processus élémentaire*

Un processus élémentaire exprime, d'une part, une propriété relative aux valeurs des signaux: les deux suites `x` et `E` sont identiques, et d'autre part, une propriété relative à la présence de ces valeurs: l'horloge associée à `x` et l'horloge associée à `E` sont identiques.

Cependant, même si l'horloge globale de l'expression `E` est identique à celle de `x`, les horloges des signaux en entrée de l'expression (c'est-à-dire les y_i's) peuvent par contre être différentes.

Ainsi, les processus élémentaires sont divisés en deux classes selon la nature des horloges manipulées en entrée d'une expression:

- *Les processus monochrones*, qui manipulent un seul index temporel: les horloges associées

aux signaux en entrée de l'expression sont toutes identiques,

- *Les processus polychrones*, qui nécessitent la manipulation de plusieurs index temporels: les horloges des signaux d'entrée sont différentes.

Processus monochrones

A tout instant, la $t^{\text{ième}}$ occurrence de l'élément en sortie du processus est produit à partir de la $t^{\text{ième}}$ occurrence de ses arguments. Les signaux en entrée et sortie de ces processus sont donc toujours parfaitement synchrones.

Processus monochrones statiques

Ils correspondent à l'extension directe de fonctions à des fonctions agissant sur des flots (e.g. opérations arithmétiques et booléennes). Par exemple [LGG90], l'expression:

$$\mathtt{x := f(y_1, \ldots, y_n)}$$

définit un processus élémentaire tel que, $\forall\ t$,

$$\mathtt{x(t) := f(y_1(t), \ldots, y_n(t))}$$

où `x(t)` dénote la $t^{\text{ième}}$ occurrence dans le temps de la séquence représentée par **x**.

Ces fonctions imposent une synchronisation forte: à tout instant, le $t^{\text{ième}}$ élément du résultat est défini par les $t^{\text{ième}}$ élément de ses arguments. Par conséquent, les signaux référencés par les $\mathtt{y}_i$'s doivent être présents simultanément.

Processus monochrones dynamiques

On dénote par processus monochrone dynamique, la possibilité de faire référence aux valeurs passées d'un signal. En effet, dans un langage synchrone, toute occurrence observée peut être immédiatement prise en compte en raison de l'instantanéité des traitements. Par conséquent, les événements sont implicitement tous fugaces. Ainsi, la conception d'applications où la prise en compte de certains événements doit être retardée, impose l'écriture de tâches complémentaires uniquement chargées de cette mémorisation temporaire: c'est l'opérateur de retard dénoté **$** en SIGNAL. Par exemple, l'expression,

$$\mathtt{x := y\ \$\ 1}$$

où `1` représente une valeur d'initialisation, exprime la relation temporelle suivante:

$$\mathtt{x(t) := y(t-1)}$$

Les horloges de **x** et de **y** sont identiques; autrement dit, les signaux **x** et **y** sont synchrones. Ainsi, par exemple:

signal original (y)	1	-4	$\perp$	$\perp$	4	2	$\perp$	...
signal retardé (x)		1	$\perp$	$\perp$	-4	4	$\perp$	...

L'opérateur de retard impose donc également le synchronisme strict des signaux référencés.

Processus polychrones

Les constructions du langage que nous avons vu pour le moment ne mettent en jeu que des signaux synchrones. On a regroupé ici sous le terme *processus polychrones* l'ensemble des constructeurs permettant la manipulation de plusieurs horloges au sein d'une même expression. Le langage SIGNAL dispose de deux constructeurs polychrones: l'opérateur **when** d'extraction de sous-séquences de valeurs (ou sous-échantillonnage) — et l'opérateur **default** d'entrelacement de signaux (ou sur-échantillonnage).

Extraction de sous-séquences

L'opérateur de sous-échantillonnage, permet d'extraire une sous-séquence de valeurs d'un signal. Dans l'exemple ci-dessous, le signal **x** est défini et prend la valeur du signal **y** quand celui-ci est présent, et que le signal booléen **b** est défini avec la valeur *vrai*:

```
x := y when b
```

cette expression exprime la relation temporelle suivante:

y	1	-4	$\perp$	$\perp$	4	2	$\perp$	...
b	vrai	faux	vrai	faux	$\perp$	vrai	$\perp$	...
x	1	$\perp$	$\perp$	$\perp$	$\perp$	2	$\perp$	...

Les signaux **y**, **b** et **x** ne sont pas synchrones: l'horloge du signal **x** est *moins rapide* que celle de **y** et **b**. Autrement dit, le signal **x** est beaucoup moins souvent défini dans le temps que les signaux **y** et **b**.

Entrelacement de signaux

L'opérateur de sur-échantillonnage, exprime l'entrelacement de signaux. Afin de garantir le déterminisme de cet opérateur, un ordre de priorité existe sur les entrées. Par exemple, l'expression suivante dénote l'entrelacement des signaux $\mathbf{y}_1$ et $\mathbf{y}_2$:

```
x := y1 default y2
```

le signal **x** prend la valeur $\mathbf{y}_1$ lorsque ce signal est défini (quel que soit $\mathbf{y}_2$). Lorsque $\mathbf{y}_1$ est indéfini, alors **x** prend la valeur de $\mathbf{y}_2$ si ce signal est présent. L'opérateur de sur-échantillonnage exprime ainsi la relation temporelle suivante:

y_1	1	-4	$\perp$	$\perp$	4	2	$\perp$	...
y_2	$\perp$	-3	$\perp$	6	$\perp$	3	9	...
x	1	-4	$\perp$	6	4	2	9	...

Les signaux $\mathbf{y}_1$, $\mathbf{y}_2$ et **x** ne sont pas synchrones, l'horloge du signal **x** est *plus rapide* que celle de $\mathbf{y}_1$ et $\mathbf{y}_2$. Autrement dit: le signal **x** est défini bien plus fréquemment dans le temps que ne le sont les signaux $\mathbf{y}_1$ et $\mathbf{y}_2$.

Opérateurs sur processus

Un programme SIGNAL est constitué d'un ensemble de processus composés à partir des processus élémentaires. Les opérations sur processus sont constituées d'un ensemble d'opérateurs de manipulation de noms: masquage, changement de nom, ... et par l'opérateur de composition de processus. Nous nous sommes attachés à décrire succinctement ce dernier, le lecteur intéressé pourra se référer au document: ([Bes92],pp.22-25) pour obtenir une plus ample information.

Les processus peuvent être composés à l'aide d'un opérateur associatif et commutatif, noté "|". Par exemple:

```
P | Q
```

dénote la création d'un nouveau processus composé à partir des processus **P** et **Q**.

Au niveau formel, la composition de processus représente l'union des systèmes d'équations de signaux associés à chacun des processus.

Les processus composés communiquent par l'intermédiaire de leurs signaux communs. Ainsi, dans l'exemple du processus `P | Q`, les sorties du processus `P` apparaissant en entrées du processus `Q` sont identifiés pour servir de liens de connexion. Les entrées qui restent dans `Q`, c'est-à-dire n'apparaissant pas en sortie de `P` sont confondues avec les entrées de `P` pour former l'interface d'entrée du processus `P | Q`. Symétriquement, les sorties de `P` n'apparaissant pas en entrée de `Q` sont confondues avec les sorties de `Q` pour former l'interface de sortie de `P | Q`.

Opérateurs dérivés du noyau

Les 5 opérateurs du noyau de SIGNAL que nous venons de commenter:

1. `x := f(`y_1`, ... ,` y_n`)` l'extension directe de fonctions aux fonctions agissant sur des flots,
2. `x := y $ 1` le retard sur signal,
3. `x := y when b` l'extraction de valeur d'un signal,
4. `x :=` y_1 `default` y_2 l'entrelacement de signaux,
5. `P | Q` la composition de processus.

ont permis d'en dériver un certain nombre de nouveaux. Nous les évoquons ici à titre d'information, mais l'étude des opérateurs de base suffit pleinement à spécifier le comportement temporel des applications.

Extraction d'horloge

L'opérateur **event** appliqué sur un signal permet d'en extraire l'horloge. Par exemple:

```
h := event y
```

h est un signal booléen émis à la fréquence de `y`, c'est-à-dire que h est défini aux même instants que `y`, et porte toujours la valeur **vrai**. L'opérateur **event** exprime la relation temporelle suivante:

y	1	-4	$\perp$	$\perp$	4	2	$\perp$	...
h	vrai	vrai	$\perp$	$\perp$	vrai	vrai	$\perp$	...

Une autre façon d'extraire une horloge, est d'utiliser une écriture simplifiée de l'opérateur de sous-échantillonnage:

```
h := when b
```

cette instruction est équivalente à,

```
h := (event b) when b
```

h est un signal booléen ne portant que la valeur **vrai**, émis à la fréquence de `b` sous la condition que la valeur booléenne `b` soit **vrai** elle même.

Synchronisation

La synchronisation explicite de signaux est possible au travers l'utilisation de l'opérateur **synchro**. Ainsi, l'instruction

```
synchro { x, y }
```

force la synchronisation des signaux `x` et `y`.

Valeurs passées d'un signal

L'opérateur de retard du noyau (c.f.§B) réfère la valeur passée d'un signal. À partir de cet opérateur, on a défini un constructeur dérivé (**window**) permettant de *remonter plus loin dans le temps.*

```
x := y window n
```

Le signal **x** est un vecteur construit à partir des **n** valeurs antérieures du signal **y**. **n** est une valeur entière strictement positive.

Mémoire synchronisée

La mémoire synchronisée (**cell**) est un processus dérivé des opérateurs de sous-, et sur- échantillonnage et retard.

```
x := y cell b
```

cette instruction est équivalente au processus suivant:

```
(| synchro { x, ( event y ) default ( when b ) }
 | zx := x $ 1
 | x := y default zx
 |)
```

Lorsque le signal **y** est présent, alors **x** prend la valeur de **y**. Dans le cas contraire, si le signal booléen **b** est défini avec la valeur **vrai**, alors **x** prend la valeur retardée de **y**.

Principe de modularité

Un processus peut comporter la définition de sous-processus: ce sont des processus locaux à la manière du langage PASCAL, ou réaliser des appels à des processus externes.

Processus locaux

Un processus local peut lui-même comporter des définitions de sous-processus *encore plus locaux*, etc .... L'appel d'un processus local s'accompagne du passage de ses paramètres effectifs, et provoque la duplication et l'insertion du système d'équations qu'il dénote en remplaçant les paramètres formels par les paramètres effectifs.

Processus externes

Le langage offre également la possibilité d'appeler des processus externes dont l'interface (paramètres formels d'entrée et de sortie du processus externe) doit être déclarée dans chaque module appelant. Cette déclaration se limite à l'interface du processus et ne peut pas comporter de déclarations de nouveaux paramètres statiques formels.

Compilation de programmes SIGNAL

La compilation SIGNAL vise essentiellement à assurer le respect de la synchronisation des calculs exprimés dans le langage. Il est pour cela nécessaire de respecter d'une part l'ordre induit par les dépendances entre les données, et d'autre part de calculer les relations existant entre

les différentes horloges [Bes92] du programme. La phase de compilation doit ainsi répondre aux questions suivantes:

- Les relations d'horloges et les dépendances de calcul spécifiées dans un processus donné permettent-elles d'en donner un schéma d'exécution valide pour tout instant (i.e. indépendamment des valeurs manipulées) ?
- Si un tel schéma existe, est-il déterministe ?

Les réponses à ces différentes questions passe par une modélisation de tout programme SIGNAL. Cette modélisation comporte deux aspects [BLG90, Bes92]:

- *La modélisation des relations temporelles* a pour but de synthétiser les propriétés du temps et de prouver leur consistance. Ceci permet d'une part, de détecter les erreurs de synchronisation et, d'autre part, d'organiser le contrôle du processus programmé.
- *La synthèse des informations relatives aux dépendances de calcul.* De par l'orientation data-flow du langage, un programme est indépendant de l'ordre d'écriture des instructions qui le composent. Les noms de signaux induisent des dépendances entre les calculs.

Les dépendances de calcul d'une part, et les informations de contrôle d'autre part définissent un graphe appelé *Graphe des Dépendances Conditionnées (GDC)* dont la particularité est que les arcs sont affectés d'une condition (horloge d'utilisation du signal) dont la valeur à chaque instant détermine l'effectivité de la dépendance.

Modélisation des relations temporelles

On a déjà vu que les relations temporelles sont exprimées en termes d'absence ($\perp$) et de présence ($\top$) relative des signaux les uns par rapport aux autres, lorsque ceux-ci correspondent à des valeurs non-booléennes.

Les signaux booléens répondent quant à eux à une logique tri-valuée. Ainsi à un instant donné, un signal booléen peu se trouver dans l'un des trois états suivants:

1. le signal n'est pas défini,
2. le signal est défini et porte la valeur *vrai*
3. le signal est défini et porte la valeur *faux*.

Cette logique bi-, et tri-, valuée peut être appréhendée de deux manières:

- Par la modélisation des relations temporelles dans le corps commutatif $\mathcal{Z}/3\mathcal{Z}$ des entiers modulo 3. Ce qui s'exprime de la manière suivante:

  Pour un signal booléen:

$$\begin{array}{lllll} b & = & true & \leftrightarrow & b = +1 \\ b & = & false & \leftrightarrow & b = -1 \\ b & = & \perp & \leftrightarrow & b = 0 \end{array}$$

  Ainsi quel que soit le signal **x** (booléen ou non):

$$\begin{array}{lllll} x & = & \top & \leftrightarrow & x^2 = 1 \\ x & = & \perp & \leftrightarrow & x^2 = 0 \end{array}$$

  Le lecteur intéressé trouvera notamment dans [BLG90] une description théorique de ce formalisme.

- Par la modélisation des relations temporelles dans le treillis booléen associé à une hiérarchie de variables en imposant des contraintes entre variables booléennes. C'est cette optique qui a été adoptée dans la mise en œuvre du compilateur SIGNAL, et que nous reprenons. On référencera dans la suite par h_x l'horloge de définition du signal **x**. Ainsi par exemple, les opérateurs du noyau expriment les égalités d'horloge suivantes:

Opérateurs du noyau	*Équations d'horloge*
`x = f(y1, ..., yn)`	$h_x = h_{y_1} = \ldots = h_{y_n}$
`x = y $ 1`	$h_x = h_y$
`x = y when b`	$h_x = h_y \wedge h_b \wedge (\mathrm{b} = \mathrm{VRAI})$
`x = y1 default y2`	$h_x = h_{y_1} \vee h_{y_2}$

N.B.: ($\wedge$, $\vee$) représentent les opérateurs logiques **AND** et **OR**.

Graphe des dépendances conditionnées

Parallèlement à la mise en équations des relations temporelles, le compilateur doit synthétiser les informations relatives aux dépendances de calcul.

Outre les dépendances directes entre signaux induites par les calculs, le compilateur synthétise un certain nombre de dépendances supplémentaires:

- *Signaux vers horloges*: un signal peut définir une horloge. Par exemple, l'opérateur **event** prend un signal pour définir une horloge,
- *Horloges vers signaux*: tout signal d'un programme est conditionné par l'horloge d'utilisation dont il dépend. Par exemple,

```
x := y when b
```

  on a vu (c.f. p.138) que la condition **when b** correspondait à l'extraction d'une horloge. La valeur du signal **x** dépend de cette horloge.

- *Horloges vers horloges*: la modélisation des relations temporelles passe par la création d'un certain nombre d'horloges composées à partir d'autres horloges. Par exemple, l'entrelacement de signaux (c.f. p.137):

```
x := y1 default y2
```

  nécessite le calcul de l'horloge exclusive de chacun des arguments du **default**, à partir de l'horloge respective des signaux y_1 et y_2, afin de garantir le déterminisme du résultat.

Exemple

La présentation du langage SIGNAL est synthétisé sur le petit exemple suivant, la syntaxe complète du langage est donnée dans [BCG+93]:

```
1  -  process PGMCELL=
2  -        { ?
3  -          ! }
4  -
5  -        (| {X, B}    := T1() {}
6  -         | {Y}       := SUBPGMCELL() {X, B}
7  -         | T2()          {Y}
8  -         |)
9  -        where
10 -             integer X, Y;
11 -             logical B
12 -
13 -        process SUBPGMCELL=
14 -              { ? integer X;
15 -                  logical B
16 -                ! integer Y }
17 -              (| synchro      { Y, B }
18 -               | ZY         := Y $ 1
19 -               | Y          := ( X when B ) default ZY
20 -               |)
21 -              where
22 -                   integer ZY
23 -              end %SUBPGMCELL%;
24 -
25 -        function T1=
26 -               { ?
27 -                 ! integer X;
28 -                   logical B };
29 -
30 -        function T2=
31 -               { ? integer Y
32 -                 !           }
33 -
34 -  end %PGMCELL%
```

Le processus principal PGMCELL (instructions 1-34) est constitué d'un sous-processus local: SUBPGMCELL (instructions 13-23), et de deux appels à des fonctions externes au programme: T1 (instructions 25-28) et T2 (instructions 30-32).

Le processus local (SUBPGMCELL) définit une mémoire synchronisée dont la sémantique est légèrement différente de celle de l'opérateur cell dérivé du noyau (c.f. p.139): le signal y prend la valeur x lorsque le signal x est défini et que le signal booléen b porte la valeur vrai, dans le cas contraire y conserve son ancienne valeur.

Pour une meilleure compréhension, nous avons représenté cet exemple sous la forme simplifiée d'un graphe data-flow (c.f. figure B.2). Le signal zx qui a été introduit dans ce graphe correspond à l'expression intermédiaire:

```
zx := x when b
```

c'est-à-dire l'extraction de la valeur x sous la condition b (c.f. p.137).

Le comportement du programme peut être décrit en quelques mots: la fonction T_1 qui ne prend pas d'entrée, émet deux signaux en fin de traitement (x et b). T_1 comme T_2 sont des fonctions implicitement étendues pour agir sur des flots (c.f. p.136), par conséquent les signaux x et b, en sortie

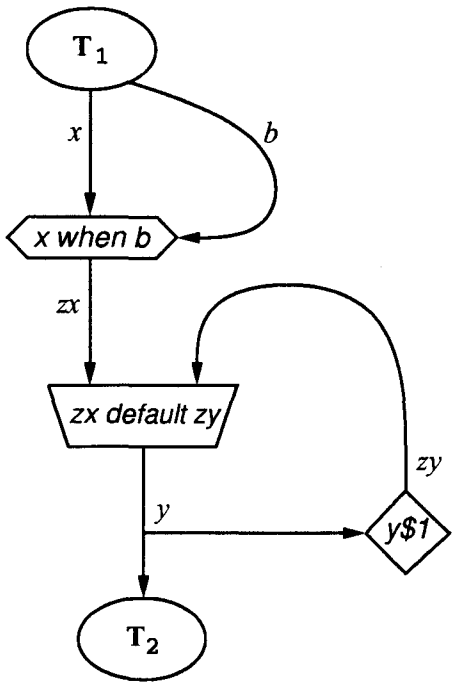


FIG. B.2 - *Représentation data-flow de* **PGMCELL**

de la même tâche (T_1), sont synchrones. Le signal intermédiaire **zx** prend la valeur de **x** lorsque **b** est **vrai**. Le signal **y** prend la valeur **zx** lorsque le signal **zx** est défini, dans le cas contraire **y** conserve son ancienne valeur.

Sur un flot hypothétique de valeurs d'entrée pour les signaux **x** et **b**, nous avons représenté le comportement temporel du programme (c.f. figure B.3).

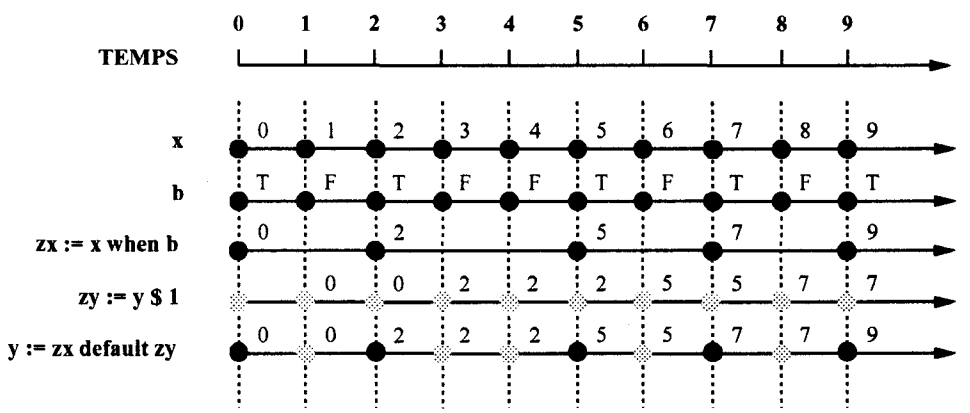


FIG. B.3 - *Chronogramme*

Le Graphe des Dépendances Conditionnées

Si le graphe data-flow donné en figure B.2 est un moyen commode de représenter les dépendances existant entre les données, il est impropre à modéliser le comportement temporel du programme temps-réel. Un tel graphe d'application est en particulier totalement non-déterministe.

La résolution des contraintes temporelles posées par le programme temps-réel fait apparaître un certain nombre de processus intermédiaires nécessités par la manipulation des différentes horloges.

Le graphe des dépendances conditionnées produit par le compilateur SIGNAL est montré dans la figure B.4. Les nœuds de calcul d'horloge ont été représentés en grisé, les fonctions externes dans un ovale, et les expressions sur signaux dans des formes géométriques similaires à celle utilisées généralement en data-flow synchronisé [Buc93, Lee93, Lee91, LM87b, LM87a]. Les signaux du type h_i, $i \in \{1, 2, r\}$ correspondent à des horloges.

Les différents types de dépendances pouvant être induites par la résolution des relations temporelles (cf. p.141) apparaissent plus clairement dans ce graphe: signaux vers horloges, horloges vers signaux et horloges vers horloges.

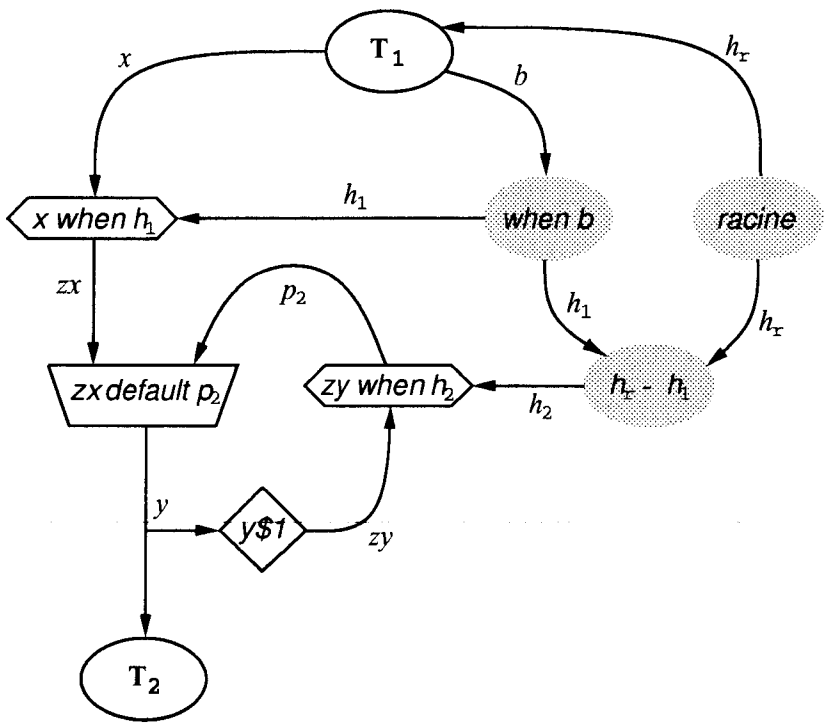


FIG. B.4 - *Graphe des dépendances conditionnées. (GDC)*

Ainsi, la construction du graphe des dépendances conditionnées (GDC) passe par la création d'un certain nombre de nœuds intermédiaires. Par exemple, les nœuds **when** et **default** du graphe data-flow (c.f. figure B.2) sont décomposés en un ensemble d'instructions plus élémentaires dans

le GDC:

Représentation data-flow	*GDC*
`zx := x when b`	`h1 := when b` `zx := x when h1`

`y := zx default zy`	`h2 := hr - h1`
	`p2 := zy when h2`
	`y := zx default p2`

La particularité du GDC est de faire figurer explicitement toutes les dépendances d'horloge induites par les relations temporelles intervenant dans le programme temps-réel. Ce développement étant nécessité par la phase formelle de résolution des contraintes temporelles de l'application.

Les relations temporelles

On réfère dans la suite par h_x, h_b, h_{zx}, h_{p_2}, h_y, et h_{zy}, l'horloge respective des signaux **x**, **b**, **zx**, **p2**, **y** et **zy**. Le nœud **racine** émet l'horloge maîtresse du programme, c'est-à-dire l'horloge la plus rapide notée h_r. La hiérarchisation des horloges en SIGNAL fait que toutes les horloges d'un programme sont incluses dans cette horloge maîtresse. Les différentes relations temporelles qui ont été synthétisées dans la phase de résolution sont:

- $h_x = h_b = h_r$. Cette équation traduit le fait que **T1** est une fonction implicitement étendue pour opérer sur des flots, par conséquent, tous ses arguments sont synchrones (c.f. p.136).
- $h_1 = h_b \wedge (\ b = VRAI\ )$. Autrement dit, l'horloge h_1 est valide lorsque le signal **b** est présent, et qu'il porte la valeur **VRAI**.
- $hzx = h_x \wedge h_1$. Cette égalité traduit en terme d'horloges, l'extraction de signal.
- $h_2 = h_r - h_1$. L'horloge h_2 est la complémentaire de l'horloge h_1 dans h_r. C'est-à-dire que h_2 est définie comme valide, lorsque h_1 ne l'est pas, et que h_r est valide. Cette propriété suit très précisément la règle d'inclusion multiple d'horloges en SIGNAL, elle peut se représenter sous une forme ensembliste comme cela a été explicité en figure B.5.

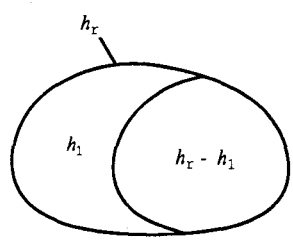


Figure B.5: Inclusion d'horloges. L'horloge h_r - h_1 est l'horloge complémentaire de h_1, h_1 étant incluse dans h_r.

- $h_{p_2} = h_{zy} \wedge h_2$. En d'autres termes, l'horloge h_{p_2} est valide lorsque le couple d'horloges h_{zy} et h_2 sont valides simultanément.
- $h_y = h_{zx} \vee h_{p_2}$. Cette égalité traduit l'entrelacement de signaux, c'est-à-dire que l'horloge h_y est valide lorsque l'horloge h_{zx} ou h_{p_2} est valide.
- $h_{zy} = h_y$. Nous avons déjà vu qu'un signal et sa valeur retardée restent synchrones.

Annexe C

Les fichiers Syndex

Le lecteur trouvera une description plus complète et exhaustive de la syntaxe des instructions SYNDEX dans [LS93].

Syndex 1

Le premier fichier SYNDEX décrit les différents nœuds du graphe. On trouvera des nœuds de calcul d'horloge, des nœuds correspondant aux fonctions externes, et des nœuds de traitement du signal.

Les nœuds d'horloge

Les nœuds d'horloge traduisent pour une part, des dépendances de type signal-vers-horloge, et d'autre part horloge-vers-horloge. C'est dans ce dernier type de nœuds que s'élaborent les formules de calcul d'horloge.

Dépendances signal-vers-horloge

Un flot booléen définit une horloge en fonction de ses instants de vérités.

1. **htt**

   L'horloge produite est valide aux instants pour lesquels, le flot booléen pris en entrée porte la valeur *vrai*.

   Exemple:

   `(htt rH_10_H logical ?eB1_14 !H_10_H)`, le nœud de sous-échantillonage `rH_10_H` produit à partir de son entrée booléenne `eB1_14` l'horloge `H_10_H`.

   `H_10_H = when eB1_14`

2. **hff**

   L'horloge produite est valide aux instants pour lesquels, le flot booléen pris en entrée porte la valeur *faux*.

   Exemple:

(hff rH_9_H logical ?eB1_18 !H_9_H), le nœud de sous-échantillonage rH_9_H produit à partir de son entrée booléenne eB1_18 l'horloge H_9_H.

$$H\_9\_H = \texttt{when not } eB1\_18$$

Dépendances horloge-vers-horloge

Ces nœuds sont directement issus du calcul sur les horloges.

1. **hinput**

   Définie l'horloge maîtresse du programme. Dans un contexte endochrone, une horloge unique peut être synthétisée.

   Exemple:

   (hinput rH_7_H H_7_H event !H_7_H), le nœud rH_7_H sans entrée, émet l'horloge maitresse H_7_H du système.

2. **hsub**

   Cette fonction définit une opération de soustraction entre deux horloges prises en entrée.

   Exemple:

   (hsub rH_22_H event ?H_15_H ?H_7_H !H_22_H), le nœud rH_22_H est un nœud de soustraction d'horloges tel que:

$$H\_22\_H = H\_7\_H \wedge \overline{H\_15\_H}$$

3. **hadd**

   Symétriquement, une fonction d'addition d'horloges existe. Le signal produit correspond à la *"somme"* des instants de validité d'un couple d'horloges pris en entrée.

   Exemple:

   (hadd rH_21_H event ?H_8_H ?H_15_H !H_21_H), le nœud rH_21_H est un nœud d'addition d'horloges qui dénote la relation temporelle suivante:

$$H\_21\_H = H\_8\_H \vee H\_15\_H$$

4. **hmul**

   Exemple:

   (hmul rH_30_H event ?H_8_H ?H_11_H !H_30_H), le nœud rH_30_h est un nœud de multiplication d'horloges prises en entrée, qui dénote la relation temporelle suivante:

$$H\_30\_H = H\_8\_H \wedge H\_11\_H$$

Les nœuds signal-vers-signal

1. **function**

   Dans le graphe de contrôle, les fonctions de calcul sont définies par leurs entréees et sorties.

   Exemple:

   `(function B3_24 TB 10 logical ?e1SB1_23, logical !B3_24)`, la fonction de calcul `TB`, dénomée `B3_24` dans le graphe, prend une entrée dénotée `e1SB1_23` et produit une sortie dénotée `B3_24`.

2. **when**

   Le nœud `when` sous-échantillonne un flot d'entrée à partir d'une horloge qui définie ses instants de validité dans le temps.

   Exemple:

   `(when XZX_27 event ?H_23_H, integer ?eSX_15 !XZX_27)`, par exemple le flot `eSX_15` est utilisé à l'horloge `H_23_H` pour définir la séquence `XZX_27`.

   ```
   XZX_27 = eSX_15 when H_23_H
   ```

3. **default**

   Le nœud `default` entrelace deux flots pris en entrée pour en produire un nouveau.

   Exemple:

   `(default Y_16 integer ?eXZX_27 ?eXZX_28 !Y_16)`, le flot `Y_16` est défini par le sur-échantillonnage des entrées `eXZX_27` et `?eXZX_28`. Le calcul d'horloge nous garantie que l'horloge respective de disponibilité de chacune des entrées sont exclusives l'une de l'autre.

   ```
   Y_16 = eXZX_27 when eXZX_28
   ```

4. **memory**

   Le nœud `memory` est induit par le besoin de mémoriser la valeur instantannée d'un signal lorsque sa valeur retardée est utilisée dans une expression.

   Exemple:

   `(memory DY_17 integer ?eY_16 $1 !DY_17)`, la valeur instantannée du signal `eY_16` est mémorisée à l'occurrence courante pour être délivrée à la suivante.

   ```
   DY_17 = eY_16$
   ```

Syndex 2

Description des connections existant dans le Graphe des Dépendances Conditionnées entre les différents noeuds de l'application. L'instruction `connect` du SYNDEX 2 précise l'existence d'une dépendance de donnée dénotée `signal` entre un `NOEUD_AMONT` et un `NOEUD_AVAL`:

(connect `NOEUD_AMONT/signal NOEUD_AVAL/signal`)

Ce qui peut également se traduire en terme de graphe par le schéma donné en figure C.1.

Exemple:

`(connect Z_13/B1_14 rH_10_H/eB1_14)`, le noeud `Z_13` émet sur sa sortie `B1_14` un signal sur l'entrée `eB1_14` de la tâche `rH_10_h`.

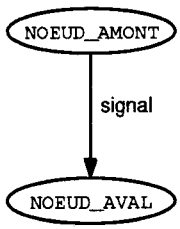


FIG. C.1 - *Sémantique du* connect.

Syndex 3

Précise l'horloge des différents noeuds, en fait on associe à chaque nœud l'horloge de son signal d'entrée le plus rapide. Les entrées des nœuds ne sont pas obligatoirement synchrones entre elles.

Exemple:

`(exec rH_14_H/H_14_H K_10 CW_19)`, l'horloge `H_14_H` émise par le noeud `rH_14_H` est associée aux nœuds `K_10` et `CW_19`.

Liste de nos publications

[Gal95] David Galinec. An unified language for real-time image processing. In *Proceedings of the Second European PVM Users'Group Meeting, shortpapers*, Lyon, France, September 1995. ENS.

[Gal96] David Galinec. Modèle d'exécution asynchrone des applications temps-réel complexes. In *Actes du colloque Automatique, Génie informatique, et Image*, pages 271–274. AFCET/SEE, June 1996.

[GBDM96] D. Galinec, F. Battini, J.-L. Dekeyser, and P. Marquet. An asynchronous run-time model for a synchronous approach of distributed real-time problems. In *Proceedings of the Telecommunication, Distribution and Parallelism International Conference*, June 1996. Hermès.

[GDM96a] D. Galinec, J.-L. Dekeyser, and P. Marquet. Distributing signal and real time image processing applications. In *Proceedings of the Computer-Aided Design International Conference in Russia*, Gelengick, Russia, September 1996.

[GDM96b] D. Galinec, J.-L. Dekeyser, and P. Marquet. Mixed synchronous-asynchronous approach for real-time processing: A MPEG-like coder. In *Proceedings of the IEEE International Conference on Image Processing*, volume II, pages 121–124, Lausanne, Switzerland, September 1996. IEEE Signal Processing Society.

[GDM96c] D. Galinec, J.-L. Dekeyser, and P. Marquet. A mixed synchronous-asynchronous approach for digital signal processing. In *Proceedings of the Third IEEE/CIE International Conference on Signal Processing*, volume I, pages 158–161, Beijing, China, October 1996. IEEE press.

[JCR+96] J.-L. Jumpertz, B. Chéron, F. Roudier, D. Galinec, and F. Battini. Implementation of the application programming model of SM-IMP. ESPRIT BRA project 8849 SM-IMP Deliverable Bn 4, Thomson Multimedia, Corporate Research Rennes, Avenue de Belle Fontaine, 35510 Cesson-Sévigné, FRANCE, January 1996.

Bibliographie

[BB91] Albert Benveniste and Gérard Berry. The Synchronous Approach to Reactive and Real-Time Systems. *Special section of proceedings of the IEEE: Another look at real-time programming*, 79(9):1270–1282, September 1991.

[BCG87] Gérard Berry, Philippe Couronné, and Georges Gonthier. Programmation synchrone des systèmes réactifs: le langage ESTEREL. *Technique et Science Informatiques*, 6(4):305–316, 1987. DUNOD/AFCET.

[BCG+93] P. Bournai, B. Chéron, T. Gautier, B. Houssais, and P. Le Guernic. SIGNAL manual. Rapport de Recherche 1969, INRIA-Rennes, September 1993.

[BDS91] F. Boussinot and R. De Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[BdS95a] F. Boussinot and R. de Simone. The SL synchronous language. Rapport de Recherche 2510, INRIA, http://ftp.inria.fr:./INRIA/publication/publi-ps.gz/RR/RR-2510.ps.gz, March 1995.

[BDS95b] F. Boussinot, G. Doumenc, and J.-B. Stefani. Reactive Objects. Rapport de Recherche 2664, INRIA, October 1995.

[Bes92] Loïc Besnard. *Compilation de* SIGNAL*: horloges, dépendances, environnement.* PhD thesis, IRISA-INRIA, FRANCE, September 1992. No 759.

[BLG90] Albert Benveniste and Paul Le Guernic. Hybrid Dynamical Systems Theory and the SIGNAL Language. *IEEE Transactions on Automatic Control*, 35(5):535–546, May 1990.

[Buc93] Joseph Tobin Buck. Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model. Memorandum UCB/ERL M93/69, Electronics Research Laboratory, University of California, Berkeley, CA 94720, USA, September 1993.

[Ché91] Bruno Chéron. *Transformations syntaxiques de programmes* SIGNAL. PhD thesis, Université de Rennes 1, Institut de Formation Supérieure en Informatique et Communication, IRISA-INRIA, Campus de Beaulieu, 35042 Rennes CEDEX, FRANCE, September 1991. No d'ordre: 670.

[CJC+94] M.-J. Colaïtis, J.-L. Jumpertz, B. Chéron, F. Battini, et al. P^3I: a Multi-paradigm Real-time Video Engine. In Springer, editor, *Image Processing for Broadcast and Video Production*, pages 52–71. Workshops in computing, British Computer Society, November 1994.

[CJG+94a] M.-J. Colaïtis, J.-L. Jumpertz, B. Guérin, B. Chéron, F. Battini, et al. Multi-paradigm Processor Pool Architectures. Technical report, Thomson Broadcast Systems, Centre d'Etude de Rennes, Avenue de Belle Fontaine, 35510 Cesson-Sévigné, FRANCE, February 1994.

[CJG+94b] M.-J. Colaïtis, J.-L. Jumpertz, B. Guérin, B. Chéron, et al. A Memory Management Sheme for Massively Parallel Video Real-time Processing. In IOS Press, editor, *Transputer 94*, 1994. Amsterdam.

[CLM91] Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. A language for compositional specification and verification of finite state hardware controllers. *Proceedings of the IEEE*, 79(9):1283–1292, September 1991.

[Cos91] Eve Coste-Manière. *Synchronisme et asynchronisme dans la programmation des sytèmes robotiques: apport du langage* ESTEREL *et de concepts objets.* PhD thesis, École des Mines de Paris, Juillet 1991.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J-.A. Plaice. LUSTRE: A declarative language for programming synchronous systems. *14^{th} ACM Symposium on Principles of Programming Languages*, pages 178–188, January 1987.

[CSS90] R. E. Cypher, J. L. C. Sanz, and L. Snyder. Algorithms for image component labelling on simd mesh-connected computers. *IEEE Transactions on Computers*, 39(2):276–281, February 1990.

[Den91] Jack B. Dennis. The evolution of "static" data-flow architecture. In *Advanced Topics in Data-Flow Computing*, chapter 2, pages 35–91. Prentice Hall, 1991.

[DF88] M.J.B. Duff and T.J. Fountain. Enhancing the two-dimensional mesh. Department of Physics and Astronomy - University College London, Gower Street, London, 1988.

[Duf82] M.J.B. Duff. Parallel algorithms and their influence on the specification of application problems. In *Multicomputers and Image Processing Algorithms and Programs*, pages 261–274. Academic Press, 1982.

[Dzi90] Daniel Dzierzgowski. Quatre exemples de langages ou environnements pour le développement de programmes où le temps intervient. *Technique et Science Informatiques*, 9(4):289–312, 1990. (article de synthèse).

[Fau88] Olivier Faugeras. Les machines de vision. *La Recherche*, 19(204):1334–1346, November 1988.

[Fly66] M. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[GBD+94a] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, USA, May 1994.

[GBD+94b] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.* Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, USA, mit press edition, 1994.

[GBDL+94] B. Guérin, F. Battini, B. De Lescure, M. Picart, and F. Roudier. P3I, programming model and applications. SM-IMP (8849) LER/2, Thomson Broadcast Systems, Centre d'Etude de Rennes, Avenue de Belle Fontaine, 35510 Cesson-Sévigné, FRANCE, January 1994.

[Gia91] Jean-Louis Giavitto. *$8^{1/2}$: un modèle MSIMD pour la simulation massivement parallèle.* PhD thesis, Université de Paris-Sud, Centre d'Orsay, December 1991. No d'ordre: 1872.

[GS93] Jean-Louis Giavitto and Jean-Paul Sansonnet. $8^{1/2}$: data-parallélisme et data-flow. *Technique et Science Informatiques*, 12(5):621–647, 1993. AFCET/HERMES.

[Har87] David Harel. STATECHARTS: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987. North-Holland.

[HCRP91a] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. Programmation et vérification des sytèmes réactifs: le langage LUSTRE. *Technique et Science Informatiques*, 10(2):139–158, 1991. DUNOD/AFCET.

[HCRP91b] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1319, September 1991.

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, et al. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

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

[HT89] S. Hambrusch and L. TeWinkel. A study of connected component labelling algorithms on the MPP. In III, editor, *International Conference on Parallel Processing*, pages 477–483, 1989. Amsterdam.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In North-Holland, editor, *Information Processing 74.* IFIP 74, 1974.

[KC90] C.S. Kannan and Henry Y.H. Chuang. Fast hough transform on a mesh connected processor array. In *Information Processing Letters*, volume 33, pages 243–248. North-Holland, 1990.

[Kom90] Erwin Ronald Komen. *Low-level Image Processing Architectures: Compared For Some Non-linear Recursive Neighbourhood Operations.* PhD thesis, Technische Universiteit Delft, 1990.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of ACM*, 21(7):558–565, July 1978.

[Lee91] Edward A. Lee. Consistency in data flow graphs. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):223–235, April 1991.

[Lee93] Edward A. Lee. Multidimensional streams rooted in dataflow. In North-Holland, editor, *Working Conference on Architectures and Compilation Techniques for fine and Medium Grain Parallelism*, New York, January 1993. IFIP.

[LG89] Bernard Le Goff. *Inférence de Contrôle Hiérarchique; Application au Temps-Réel.* PhD thesis, Université de Rennes 1, Institut de Formation Supérieure en Informatique et Communication, IRISA-INRIA, Campus de Beaulieu, 35042 Rennes CEDEX, FRANCE, June 1989. No d'ordre: 277.

[LGG90] Paul Le Guernic and Thierry Gautier. Dataflow to von neumann: the SIGNAL approach. Publication interne 531, Institut de Recherche en Informatique et Systèmes Aléatoires, IRISA-INRIA, Campus de Beaulieu, 35042 Rennes CEDEX, FRANCE, April 1990.

[LGM+91] Paul Le Guernic, Le Borgne Michel, et al. Programming real-time applications with SIGNAL. *Procedings of the IEEE*, 79(9):1321–1336, September 1991. (general paper on SIGNAL).

[LM87a] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, January 1987.

[LM87b] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[LM94] Dominique Lavenier and Roderick McConnell. A component model for synchronous vlsi system design. Rapport de Recherche 2285, INRIA, May 1994.

[LS93] C. Lavarenne and Y. Sorel. Syntaxe des instructions SynDEx. Document interne, INRIA, September 1993.

[LTD88] INMOS LTD, editor. *occam 2 Reference Manual.* International series in computer science. Prentice Hall, 1988.

[Maf93] Olivier Maffeïs. *Ordonnancements de graphes de flots synchrones; application à la mise en œuvre de* SIGNAL. PhD thesis, Université de Rennes 1, Institut de Formation Supérieure en Informatique et Communication, IRISA-INRIA, Campus de Beaulieu, 35042 Rennes CEDEX, FRANCE, January 1993.

[MCL92] O. Maffeïs, B. Chéron, and P. Le Guernic. Transformations du graphe des programmes SIGNAL. Rapport de Recherche 1574, INRIA-Rennes, January 1992.

[RCCE92] O. Roux, D. Creusot, F. Cassez, and J.-P. Elloy. Le langage réactif asynchrone ELECTRE. *Technique et Science Informatiques*, 11(5):35–66, 1992. AFCET/HERMES.

[RCDS90] B. Raithier, J-.F. Caillet, and P. De Seze. IDEFIX: A tool for debugging ada tasks in a real-time environment. *Technique et Science Informatiques*, 9(2):150–156, 1990.

[Rob94] Yves Robin. *Programmation et compilation sur architecture SIMD à mémoire distribuée dans un contexte de traitement d'image temps-réel vidéo.* PhD thesis, Université de Rennes 1, Institut de Formation Supérieure en Informatique et Communication, November 1994. No d'ordre: 1266.

[Sha92] Alan C. Shaw. Communicating Real-Time State Machines. *IEEE Transactions on Software Engineering*, 18(9):805–816, September 1992.

[SL94] Anthony Skjellum and Ewing Lusk. Early Applications in the Message-Passing Interface (MPI). Submitted to Special Issue if *International Journal of Supercomputing Applications*, June 1994.

[SMS93] Som Sukhamoy, Roland R. Mielke, and John W. Stoughton. Prediction of performance and processor requirements in real-time data flow architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(11):1205–1216, November 1993.

[Sor94] Y. Sorel. Massively parallel computing systems with real time constraints: The algorithm architecture adequation methodology. In *Massively Parallel Computing Systems, the Challenge of General-Purpose and Special-Purpose Computing*, May 1994.

[Sor96] Y. Sorel. Real-time embedded image processing applications using the A^3 methodology. In *Proceedings of the IEEE International Conference on Image Processing*, volume 2, pages 145–148, Lausanne, Switzerland, September 1996. IEEE Signal Processing Society.

[Tud95] Phil Tudor. Mpeg-2 Video Compression Tutorial. Technical report, The Institution of Electrical Engineers, London, UK, 1995.

[Uni94] University of Tennessee, Knoxville, Tennessee, USA. *MPI: A Message-Passing Interface Standard*, 1994.

[Wol91] Pierre Wolper. *Introduction à la calculabilité.* IIA, 1991.

[ZXL89] Fang Zhixi, Li Xiabo, and M. Ni Lionel. On the communication complexity of generalized 2-D convolution on array processors. *IEEE Transactions on Computers*, 38(2):184–193, February 1989.