LUSTRE : un langage déclaratif pour le temps réel

Jean-Louis Bergerand

To cite this version:

Jean-Louis Bergerand. LUSTRE : un langage déclaratif pour le temps réel. Génie logiciel [cs.SE]. Institut National Polytechnique de Grenoble - INPG, 1986. Français. NNT : . tel-00320006

HAL Id: tel-00320006

https://theses.hal.science/tel-00320006

Submitted on 10 Sep 2008

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

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

THESE

présentée par

Jean-Louis BERGERAND

pour obtenir le titre de DOCTEUR

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(*arrêté ministériel du 5 juillet 1984*)

Spécialité : INFORMATIQUE

====================

LUSTRE : un langage déclaratif
pour le temps réel

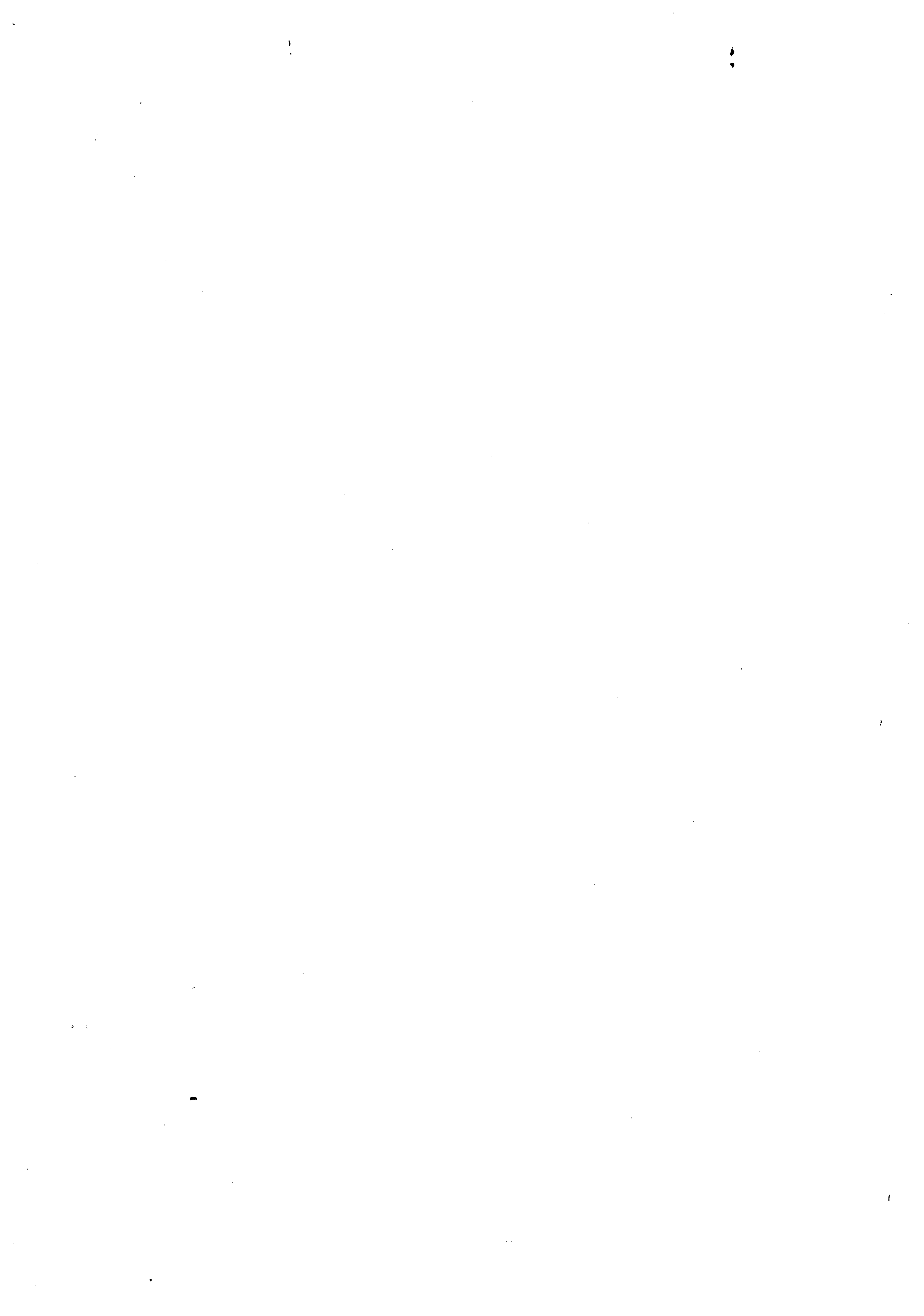
====================

Date de soutenance : 8 Janvier 1986

Composition du jury :

Jean-Pierre BANATRE : président
Gérard BERRY
Gabriele SAUCIER
Paul CASPI
Nicolas HALBWACHS

Thèse préparée au sein du laboratoire Circuits et Systèmes (IMAG)



INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Année universitaire 1982-1983

Président de l'Université : D. BLOCH

Vice-Président : René CARRE
Hervé CHERADAME
Marcel IVANES

PROFESSEURS DES UNIVERSITES :

ANCEAU François	E.N.S.I.M.A.G.
BARRAUD Alain	E.N.S.I.E.G.
BAUDELET Bernard	E.N.S.I.E.G.
BESSON Jean	E.N.S.E.E.G.
BLIMAN Samuel	E.N.S.E.R.G.
BLOCH Daniel	E.N.S.I.E.G.
BOIS Philippe	E.N.S.H.G.
BONNETAIN Lucien	E.N.S.E.E.G.
BONNIER Etienne	E.N.S.E.E.G.
BOUVARD Maurice	E.N.S.H.G.
BRISSONNEAU Pierre	E.N.S.I.E.G.
BUYLE BODIN Maurice	E.N.S.E.R.G.
CAVAIGNAC Jean-François	E.N.S.I.E.G.
CHARTIER Germain	E.N.S.I.E.G.
CHENEVIER Pierre	E.N.S.E.R.G.
CHERADAME Hervé	U.E.R.M.C.P.P.
CHERUY Arlette	E.N.S.I.E.G.
CHIAVERINA Jean	U.E.R.M.C.P.P.
COHEN Joseph	E.N.S.E.R.G.
COUMES André	E.N.S.E.R.G.
DURAND Francis	E.N.S.E.E.G.
DURAND Jean-Louis	E.N.S.I.E.G.
FELICI Noël	E.N.S.I.E.G.
FOULARD Claude	E.N.S.I.E.G.
GENTIL Pierre	E.N.S.E.R.G.
GUERIN Bernard	E.N.S.E.R.G.
GUYOT Pierre	E.N.S.E.E.G.
IVANES Marcel	E.N.S.I.E.G.
JAUSSAUD Pierre	E.N.S.I.E.G.
JOUBERT Jean-Claude	E.N.S.I.E.G.
JOURDAIN Geneviève	E.N.S.I.E.G.
LACOUME Jean-Louis	E.N.S.I.E.G.
LATOMBE Jean-Claude	E.N.S.I.M.A.G.

.../...

LESSIEUR Marcel	E.N.S.H.G.
LESPINARD Georges	E.N.S.H.G.
LONGEQUEUE Jean-Pierre	E.N.S.I.E.G.
MAZARE Guy	E.N.S.I.M.A.G.
MOREAU René	E.N.S.H.G.
MORET Roger	E.N.S.I.E.G.
MOSSIERE Jacques	E.N.S.I.M.A.G.
PARIAUD Jean-Charles	E.N.S.E.E.G.
PAUTHENET René	E.N.S.I.E.G.
PERRET René	E.N.S.I.E.G.
PERRET Robert	E.N.S.I.E.G.
PIAU Jean-Michel	E.N.S.H.G.
POLOUJADOFF Michel	E.N.S.I.E.G.
POUPOT Christian	E.N.S.E.R.G.
RAMEAU Jean-Jacques	E.N.S.E.E.G.
RENAUD Maurice	U.E.R.M.C.P.P.
ROBERT André	U.E.R.M.C.P.P.
ROBERT François	E.N.S.I.M.A.G.
SABONNADIERE Jean-Claude	E.N.S.I.E.G.
SAUCIER Gabrielle	E.N.S.I.M.A.G.
SCHLENKER Claire	E.N.S.I.E.G.
SCHLENKER Michel	E.N.S.I.E.G.
SERMET Pierre	E.N.S.E.R.G.
SILVY Jacques	U.E.R.M.C.P.P.
SOHM Jean-Claude	E.N.S.E.E.G.
SOUQUET Jean-Louis	E.N.S.E.E.G.
VEILLON Gérard	E.N.S.I.M.A.G.
ZADWORNY François	E.N.S.E.R.G.

PROFESSEURS ASSOCIES

BASTIN Georges	E.N.S.H.G.
BERRIL John	E.N.S.H.G.
CARREAU Pierre	E.N.S.H.G.
GANDINI Alessandro	U.E.R.M.C.P.P.
HAYASHI Hirashi	E.N.S.I.E.G.

PROFESSEURS UNIVERSITE DES SCIENCES SOCIALES (Grenoble II)

BOLLIET Louis
Chatelin Françoise

PROFESSEURS E.N.S. Mines de Saint-Etienne

RIEU Jean
SOUSTELLE Michel

CHERCHEURS DU C.N.R.S.

FRUCHART Robert	Directeur de Recherche
VACHAUD Georges	Directeur de Recherche

.../...

ALLIBERT Michel	Maître de Recherche
ANSARA Ibrahim	Maître de Recherche
ARMAND Michel	Maître de Recherche
BINDER Gilbert	
CARRE René	Maître de Recherche
DAVID René	Maître de Recherche
DEPORTES Jacques	
DRIOLE Jean	Maître de Recherche
GIGNOUX Damien	
GIVORD Dominique	
GUELIN Pierre	
HOPFINGER Emil	Maître de Recherche
JOUD Jean-Charles	Maître de Recherche
KAMARINOS Georges	Maître de Recherche
KLEITZ Michel	Maître de Recherche
LANDAU Ioan-Dore	Maître de Recherche
LASJAUNIAS J.C.	
MERMET Jean	Maître de Recherche
MUNIER Jacques	Maître de Recherche
PIAU Monique	
PORTESEIL Jean-Louis	
THOLENCE Jean-Louis	
VERDILLON André	

CHERCHEURS du MINISTERE de la RECHERCHE et de la TECHNOLOGIRE (Directeurs et Maîtres de Recherches, ENS Mines de St. Etienne)

LESBATS Pierre	Directeur de Recherche
BISCONDI Michel	Maître de Recherche
KOBYLANSKI André	Maître de Recherche
LE COZE Jean	Maître de Recherche
LALAUZE René	Maître de Recherche
LANCELOT Francis	Maître de Recherche
THEVENOT François	Maître de Recherche
TRAN MINH Canh	Maître de Recherche

PERSONNALITES HABILITEES à DIRIGER des TRAVAUX de RECHERCHE (Décision du Conseil Scientifique)

ALLIBERT Colette	E.N.S.E.E.G.
BERNARD Claude	E.N.S.E.E.G.
BONNET Rolland	E.N.S.E.E.G.
CAILLET Marcel	E.N.S.E.E.G.
CHATILLON Catherine	E.N.S.E.E.G.
CHATILLON Christian	E.N.S.E.E.G.
COULON Michel	E.N.S.E.E.G.
DIARD Jean-Paul	E.N.S.E.E.G.
EUSTAPOPOULOS Nicolas	E.N.S.E.E.G.
FOSTER Panayotis	E.N.S.E.E.G.

.../...

GALERIE Alain	E.N.S.E.E.G.
HAMMOU Abdelkader	E.N.S.E.E.G.
MALMEJAC Yves	E.N.S.E.E.G. (CENG)
MARTIN GARIN Régina	E.N.S.E.E.G.
NGUYEN TRUONG Bernadette	E.N.S.E.E.G.
RAVAINE Denis	E.N.S.E.E.G.
SAINFORT	E.N.S.E.E.G. (CENG)
SARRAZIN Pierre	E.N.S.E.E.G.
SIMON Jean-Paul	E.N.S.E.E.G.
TOUZAIN Philippe	E.N.S.E.E.G.
URBAIN Georges	E.N.S.E.E.G. (Laboratoire des ultra-réfractaires ODEILLON)
GUILHOT Bernard	E.N.S. Mines Saint Etienne
THOMAS Gérard	E.N.S. Mines Saint Etienne
DRIVER Julien	E.N.S. Mines Saint Etienne
BARIBAUD Michel	E.N.S.E.R.G.
BOREL Joseph	E.N.S.E.R.G.
CHOVET Alain	E.N.S.E.R.G.
CHEHIKIAN Alain	E.N.S.E.R.G.
DOLMAZON Jean-Marc	E.N.S.E.R.G.
HERAULT Jeanny	E.N.S.E.R.G.
MONLLOR Christian	E.N.S.E.R.G.
BORNARD Guy	E.N.S.I.E.G.
DESCHIZEAU Pierre	E.N.S.I.E.G.
GLANGEAUD François	E.N.S.I.E.G.
KOFMAN Walter	E.N.S.I.E.G.
LEJEUNE Gérard	E.N.S.I.E.G.
MAZUER Jean	E.N.S.I.E.G.
PERARD Jacques	E.N.S.I.E.G.
REINISCH Raymond	E.N.S.I.E.G.
ALEMANY Antoine	E.N.S.H.G.
BOIS Daniel	E.N.S.H.G.
DARVE Félix	E.N.S.H.G.
MICHEL Jean-Marie	E.N.S.H.G.
OBLED Charles	E.N.S.H.G.
ROWE Alain	E.N.S.H.G.
VAUCLIN Michel	E.N.S.H.G.
WACK Bernard	E.N.S.H.G.
BERT Didier	E.N.S.I.M.A.G.
CALMET Jacques	E.N.S.I.M.A.G.
COURTIN Jacques	E.N.S.I.M.A.G.
COURTOIS Bernard	E.N.S.I.M.A.G.
DELLA DORA Jean	E.N.S.I.M.A.G.
FONLUPT Jean	E.N.S.I.M.A.G.
SIFAKIS Joseph	E.N.S.I.M.A.G.
CHARUEL Robert	U.E.R.M.C.P.P.
CADET Jean	C.E.N.G.
COEURE Philippe	C.E.N.G. (LETI)

.../...

DELHAYE Jean-Marc	C.E.N.G. (STT)
DUPUY Michel	C.E.N.G. (LETI)
JOUVE Hubert	C.E.N.G. (LETI)
NICOLAU Yvan	C.E.N.G. (LETI)
NIFENECKER Hervé	C.E.N.G.
PERROUD Paul	C.E.N.G.
PEUZIN Jean-Claude	C.E.N.G. (LETI)
TAIEB Maurice	C.E.N.G.
VINCENDON Marc	C.E.N.G.

LABORATOIRES EXTERIEURS

DEMOULIN Eric	C.N.E.T.
DEVINE	C.N.E.T. (R.A.B.)
GERBER Roland	C.N.E.T.
MERCKEL Gérard	C.N.E.T.
PAULEAU Yves	C.N.E.T.
GAUBERT C.	I.N.S.A. Lyon

ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT-ETIENNE

Directeur	: Monsieur M. MERMET
Directeur des Etudes et de la formation	: Monsieur J. LEVASSEUR
Directeur des recherches	: Monsieur J. LEVY
Secrétaire Général	: Mademoiselle M. CLERGUE

Professeurs de 1ère Catégorie

COINDE	Alexandre	Gestion
GOUX	Claude	Métallurgie
LEVY	Jacques	Métallurgie
LOWYS	Jean-Pierre	Physique
MATHON	Albert	Gestion
RIEU	Jean	Mécanique - Résistance des matériaux
SOUSTELLE	Michel	Chimie
FORMERY	Philippe	Mathématiques Appliquées

Professeurs de 2ème catégorie

HABIB	Michel	Informatique
PERRIN	Michel	Géologie
VERCHERY	Georges	Matériaux
TOUCHARD	Bernard	Physique Industrielle

Directeur de recherche

LESBATS	Pierre	Métallurgie

Maîtres de recherche

BISCONDI	Michel	Métallurgie
DAVOINE	Philippe	Géologie
FOURDEUX	Angeline	Métallurgie
KOBYLANSKI	André	Métallurgie
LALAUZE	René	Chimie
LANCELOT	Francis	Chimie
LE COZE	Jean	Métallurgie
THEVENOT	François	Chimie
TRAN MINH	Canh	Chimie

Personnalités habilitées à diriger des travaux de recherche

DRIVER	Julian	Métallurgie
GUILHOT	Bernard	Chimie
THOMAS	Gérard	Chimie

Professeur à l'UER de Sciences de Saint-Etienne

VERGNAUD	Jean-Maurice	Chimie des Matériaux & chimie industrielle

****

REMERCIEMENTS

Le travail présenté dans cet ouvrage est avant tout un travail d'équipe. La définition du langage n'a pas été le fait d'une seule personne mais d'un groupe, et je voudrais remercier ceux qui ont contribué à ce travail, permettant au langage LUSTRE de voir le jour, ainsi qu'à cette thèse.

Que soient donc remerciés Paul Caspi, Nicolas Halbwachs, Daniel et Eric Pilaud.

Ce travail est également le fruit d'une collaboration entre deux laboratoires. Je tiens à remercier Gabriële Saucier, qui m'a accueilli dans son laboratoire, et qui m'y a fourni d'excellentes conditions de travail, ainsi que Joseph Sifakis qui ont permis cette collaboration.

Je remercie Jean-Pierre Banatre qui a accepté de présider le jury

Je remercie enfin Gérard Berry, non seulement pour avoir accepter de participer au jury, mais aussi pour l'accueil qu'il a toujours réservé à LUSTRE ainsi qu'à notre groupe de travail, et pour les conseils qu'il nous a prodigués à chacunes de nos rencontres.

Table des matières

INTRODUCTION 9

CHAPITRE 1 LES PRINCIPAUX LANGAGES TEMPS REEL 13

1. INTRODUCTION 13

2. LES LANGAGES PSEUDO-PARALLELES 14

2.1. Le langage PL/1-TR 15

3. LES LANGAGES ISSUS DE C.S.P. 16

3.1. Les outils temps réels pour ADA 17

4. PROCESSUS SEQUENTIELS SYNCHRONES 20

4.1. Le langage ESTEREL 20

4.1.1. Concepts de base 20

4.1.2. Signaux et évènements 21

4.1.3. Le temps 21

4.1.4. Le contrôle 22

4.1.5. les instructions 22

4.1.6. La communication 23

4.1.7. Diverses constructions 24

4.1.8. Discussion d'ESTEREL 25

5. LES LANGAGES GRAPHIQUES 26

6. LES LANGAGES NON PROCEDURAUX 28

6.1. Le langage LTS 29

6.1.1. Signaux et constantes 29

6.1.2. La notion de temps 29

6.1.3. Les fonctions temporelles 29

6.1.4. Conclusion 30

6.2. Le langage SIGNAL 31

6.2.1. Construction des réseaux statiques 32

6.2.2. Les horloges 34

6.2.3. Générateurs temporels primitifs 35

6.2.4. Les opérateurs du langage 36

6.2.5. Conclusion 38

7. LE LANGAGE LUCID 38

7.1. Variables et opérateurs 39

7.2. Premier exemple 40

7.3. Les opérateurs "latest" et "latest-1" 41

7.4. Structuration des programmes 43

7.4.1. Compute 44

7.4.2. Mapping 44

7.4.3. Produce 45

7.4.4. Function 46

7.5. Conclusion 47

8. CONCLUSION 49

CHAPITRE 2 LE LANGAGE LUSTRE DEFINITION ET PREMIERS EXEMPLES 51

1. PRESENTATION GENERALE 51

1.1. langage déclaratif 51

1.2. langage synchrone 52

1.3. langage structuré 53

1.4. langage temps réel 54

2. EQUATIONS, VARIABLES ET EXPRESSIONS 55

3. LES OPERATEURS DU LANGAGE 56

3.1. Les opérateurs instantanés 56

3.2. Les opérateurs temporels synchrones 57

3.2.1. L'opérateur "pre" 57

3.2.2. L'opérateur "->" 58

3.3. Premier exemple 59

4. STRUCTURATION DES PROGRAMMES 61

4.1. Définitions de noeuds 61

4.2. Appels de noeuds 61

5. LES TYPES 64

5.1. Types simples 64

5.2. Tableaux et structures régulières 66

5.2.1. Les tableaux et leurs restrictions. 66

5.2.2. Structures régulières 66

6. EXEMPLES DE PROGRAMMES SYNCHRONES 68

6.1. Compteur d'évènement 68

6.2. Chiens de garde 69

7. ASYNCHRONISME 70

7.1. Echantillonnage synchrone, horloges 71
7.1.1. L'opérateur "when" 71
7.1.2. L'opérateur "current" 73
7.1.3. Propriétés 74
7.1.4. Echantillonnage et appels de noeuds 76
7.2. Echantillonnage asynchrone : l'opérateur "every" 77
7.3. Simulation de l'opérateur "every" 79
7.4. Paramètres asynchrones 80

8. EXEMPLES ASYNCHRONES 81

8.1. Compteur d'évènement asynchrone 81
8.2. Chien de garde asynchrone 82

9. Conclusion 82

CHAPITRE 3 QUELQUES PROGRAMMES LUSTRE 85

1. DOMAINE TEMPS REEL : un compteur d'essieux 85

2. ALGORITHME SYSTOLIQUE 88

3. AUTOMATIQUE 91

3.1. Automatique logique 91
3.2. Automatique numérique 93

4. SPECIFICATION DE CIRCUIT 96

CHAPITRE 4 SEMANTIQUE DU LANGAGE 99

1. SEMANTIQUE DYNAMIQUE 99

1.1. Le langage de base L0 99
1.2. Domaines syntaxiques 99
1.2.1. Domaine des expressions 100
1.2.2. Domaine des équations 100
1.2.3. Domaine des systèmes d'équations 100
1.2.4. Domaine des programmes 101
1.3. Domaines sémantiques 101
1.3.1. Domaines dérivés 101
1.3.2. Fonctions sémantiques 102
1.4. Sémantique des expressions 103

1.5. Sémantique des équations 105
1.6. Sémantique d'un système d'équations 106
1.7. Sémantique d'un programme 106
1.8. Le langage asynchrone L1 107
1.9.. Le langage structuré L2 108

2. SEMANTIQUE STATIQUE 111

2.1. Domaines syntaxiques 111
2.2. Domaines sémantiques 112
2.3. Fonctions sémantiques 113
2.4. Sémantique du langage L2 - Le calcul d'horloge 114
2.4.1. Sémantique des expressions 114
2.4.2. Sémantique des équations et des systèmes d'équations 116
2.4.3. Sémantique d'un noeud 116
2.4.4. Sémantique d'une déclaration de noeud 117

3. CONCLUSION 117

CHAPITRE 5 TRANSFORMATION DE PROGRAMMES 119

1. PRINCIPE DE SUBSTITUTION 119

2. AXIOMATIQUE DES OPERATEURS TEMPORELS SYNCHRONES 121

3. PROPRIETES DES OPERATEURS 123

3.1. Distributivités 123
3.2. Les booléens et la conditionnelle 123

4. EXEMPLES DE TRANSFORMATIONS 127

4.1. Optimisation d'un compteur 127
4.2. Désynchronisation d'un compteur d'évènements 129

CONCLUSION 133

1. BILAN 133

2. PERSPECTIVES 139

2.1. Compilation 139
2.2. Traitement formel des programmes 140

o o o
o

INTRODUCTION

Cet ouvrage décrit un langage de programmation des systèmes "temps réel". Les domaines d'application principalement visés concernent d'une part, la programmation des systèmes de contrôle de processus industriels, et d'autre part la spécification comme la simulation des systèmes matériels. Les caractéristiques communes de ces domaines sont les suivantes :

- Les systèmes considérés sont généralement non terminants, saisissant leurs entrées et fournissant leurs résultats tout au long de leur exécution - par opposition aux programmes classiques qui commencent avec des données et délivrent leurs résultats aprés terminaison.

- Ce sont des systèmes intrinsèquement parallèles : en ce qui concerne le contrôle de processus, le système de commande interagit avec son environnement (le processus à commander et éventuellement le contrôleur humain); un système matériel est formé de composants interconnectés.

- Ce sont des systèmes dont le comportement dépend du temps : contrairement aux systèmes parallèles de type système d'exploitation, dont le temps de réponse n'est qu'un critère à minimiser, un système de contrôle de processus est généralement soumis à des contraintes temporelles impératives. En effet, le processus physique à commander ne peut attendre que le contrôleur soit prêt à communiquer avec lui. D'autre part, la nécessité de satisfaire ces contraintes temporelles interdit souvent l'usage de techniques de synchronisation indépendantes du temps mais pénalisant le temps d'exécution. Enfin, la correction d'un système matériel dépend des temps de réponse des composants (temps de franchissement d'une porte, de stabilisation d'une bascule ...).

Une autre caractéristique des systèmes de contrôle de processus est l'exigence d'une grande sûreté de fonctionnement.

Dans ces domaines, les outils traditionnels de description des systèmes sont intrinsèquement parallèles (schémas de portes, de relais, schémas analogiques) ou déclaratifs (équations booléennes). Il est paradoxal que leur informatisation provoque le remplacement progressif de ces outils par des langages impératifs et plus ou moins séquentiels (langages séquentiels, processus communicants), au moment même où les langages de programmation évoluent vers un plus grand parallélisme et un style moins procédural.

Au vu de ces remarques, nous avons cherché à définir un langage à parallélisme maximal et implicite, ce qui nous a conduit à adopter une approche "flots de données". Parmi les langages dits à flots de données, le langage LUCID [Ashcroft-Wadge 76] développé par Ashcroft et Wadge nous a paru particulièrement séduisant, par sa nature déclarative, sa sémantique rigoureuse et simple, et la facilité d'y effectuer des manipulations formelles laissant entrevoir des possibilités de preuve et d'optimisation de programmes. Il restait à munir LUCID de capacités "temps réel", c'est à dire, d'une part à y introduire la possibilité d'expliciter le comportement temporel des programmes, et d'autre part à le restreindre afin de pouvoir produire un code efficace. LUSTRE est le résultat de cette étude.

Les caractéristiques principales de LUSTRE sont les suivantes :

- Comme en LUCID, toute variable en LUSTRE représente une suite infinie de valeurs, et les programmes opèrent globalement sur ces suites. LUSTRE est ainsi un langage sans affectation, ce qui facilite la manipulation formelle des programmes grâce à ce que nous appelons le "principe de substitution" : Si une variable X est définie par une équation X = E, alors, partout dans le programme, l'identificateur X peut être remplacé par l'expression E, et inversement.

- LUSTRE est un langage applicatif : Tout programme, comme tout opérateur, est une fonction de suites *(un noeud)*. Les programmes sont structurés en réseaux de noeuds. La déclaration d'un noeud décrit une relation entre ses paramètres d'entrée et de sortie, à l'aide d'un système d'équations. Les noeuds sont instanciés dans un style fonctionnel, qui permet, par le simple jeu du passage de paramètres, de construire des réseaux arbitrairement complexes.

- Le langage est conçu de manière à permettre une interprétation synchrone : chaque variable possède la n-ième valeur de sa suite au n-ième cycle du programme. Ceci permet de décrire des systèmes temps réel, puisque le cycle d'exécution des programmes peut être vu comme une échelle de temps physique. L'interprétation synchrone induit des contraintes sur le type de relations entrée/sortie que l'on peut

exprimer : la sortie d'un programme à un instant donné ne peut dépendre d'entrées futures (causalité), et ne peut dépendre que d'une quantité bornée d'histoire passée (mémoire bornée). Ces contraintes sont à l'origine des principales différences entre LUSTRE et LUCID.

- Cependant le synchronisme strict peut être relâché, au moyen de quelques opérateurs : les échantillonnages synchrone et asynchrone sur condition booléenne (horloges), et la projection (valeur courante).

- Enfin LUSTRE a été conçu pour que la cohérence sémantique des programmes puisse être décidée soit à la compilation, soit durant un prologue de l'exécution. En effet, l'efficacité du code produit est un objectif particulièrement important dans le domaine de la programmation temps réel. En ce qui concerne l'efficacité, le parallélisme maximal inhérent aux programmes LUSTRE est aussi un atout.

Avant d'entreprendre la présentation du langage, il nous a paru souhaitable de faire un survol des principales approches actuelles de la programmation temps réel, afin de bien situer notre travail. Au chapitre 1, nous discuterons successivement des approches suivantes :

- les langages classiques, issus du domaine des systèmes d'exploitation, fondés sur un modèle pseudo-parallèle,

- les langages dérivés du modèle des processus séquentiels communicants,

- les langages à processus séquentiels synchrones,

- l'approche graphique, issue des réseaux de Pétri,

- les langages non procéduraux. En particulier nous rappellerons les principales primitives de LUCID.

Le chapitre 2 présente le langage LUSTRE, dont l'utilisation est illustrée au chapitre 3 par divers exemples (automatismes logiques et numériques, algorithmes systoliques). Le chapitre 4 présente la sémantique formelle de LUSTRE, dans un style dénotationnel.

Les prolongements de ce travail sont multiples : Au chapitre 5, nous parlerons des perspectives de transformation et d'optimisation de programmes.

En guise de conclusion, nous ébaucherons une comparaison entre LUSTRE et le langage SIGNAL. Ces deux langages ont été définis à peu prés en même temps, et à partir des mêmes principes.

La compilation du langage constitue évidemment notre objectif principal à court terme. Nous l'aborderons en discutant de l'architecture du compilateur, du principe du prototype actuel (qui produit du code ESTEREL à partir d'un sous ensemble du langage), et de l'implantation temps réel envisagée.

CHAPITRE I
LES PRINCIPAUX LANGAGES TEMPS REEL

1. INTRODUCTION

Longtemps réservée à la programmation en assembleur, la programmation temps réel a vu, depuis quelques années, l'apparition de langages évolués [Young 82]. Ces langages ont tous été largement influencés par des langages de programmation parallèles, aussi peut-on les classer, de façon grossière, selon le modèle du parallélisme dont ils s'inspirent :

- **Les langages pseudo-parallèles** (FORTRAN-TR, BASIC-TR, PL1-TR) : Il s'agit de langages où l'on peut décrire un ensemble de programmes (tâches) s'exécutant concurremment, et communiquant au moyen d'une mémoire partagée. Grâce à cette notion de tâche, l'ordonnancement des calculs a été reporté sur un noyau de gestion des tâches (moniteur). Le programmeur n'a plus à écrire un programme purement séquentiel, mais il doit donner des informations au moniteur sous forme de priorités.

- **Les langages à processus séquentiels asynchrones** : Issus du modèle CSP [Hoare 78] de Hoare, ces langages évitent l'indéterminisme dû au partage de la mémoire, grâce au concept de communication par rendez-vous. Notons que le principal objectif de CSP était de rendre le comportement des programmes indépendant de leurs temps d'exécution. Nous classerons le langage ESTELLE [Iso 85], et dans une certaine mesure, ADA [ADA 80], bien que ce dernier permette aussi le partage de la mémoire.

- **Les langages graphiques** : Ce sont les langages principalement issus des réseaux de Pétri (Réseaux temporisés, synchronisés, GRAFCET [Afcet 83], Statecharts [Harel 83]). Surtout développés pour la programmation des automates de contrôle, ces langages sont d'une extrême variété, ils sont difficiles à classifier autrement que par leur aspect graphique.

- **Les langages à processus séquentiels synchrones** : Leur origine réside dans le calcul SCCS [Milner 83] de Milner. Dans cette classe, nous présenterons en détail le langage ESTEREL [Berry-Cosserat 84] [Berry-all 83].

- **Les langages non procéduraux** (langages applicatifs, langages à flot de données) : L'absence d'ordre d'exécution explicite dans ces langages permet implicitement un parallélisme maximal, les seules contraintes de séquencement étant dûes à la dépendance entre les données [Ackerman 82]. Ces langages n'ont pas été très étudiés dans le domaine du temps réel, c'est pourquoi nous nous y sommes intéressés.

Dans ce chapitre, nous allons présenter ces différentes approches en essayant de mettre en évidence leurs avantages mais aussi les limitations qui nous ont conduits à définir le langage LUSTRE.

2. LES LANGAGES PSEUDO-PARALLELES

Avant l'introduction des concepts de la programmation parallèle, les langages séquentiels ont permis de produire des langages temps réel par adjonction des concepts de tâches et d'échanges de signaux. Ces mécanismes, issus de la programmation des systèmes d'exploitation, permettent une plus grande souplesse dans l'analyse et la programmation des applications temps réel, notamment par l'introduction d'une certaine modularité, grâce au découpage en tâches ; en revanche, le programmeur perd le contrôle complet de l'ordonnancement. En effet, l'ordonnancement est assuré par le système hôte et le programmeur ne peut fournir que des indications sur la priorité des tâches qu'il a définies. Donc l'information sur les temps d'exécution des différentes tâches est appauvrie par rapport à la connaissance plus précise qu'on en a dans le cas d'un programme purement séquentiel. L'évaluation des temps de réponse sera donc soumise à la connaissance du moniteur d'ordonnancement du système. [Wirth 77]

Parmi les langages les plus courants citons BASIC-TR, FORTRAN-TR. Nous présenterons PL/1-TR qui donne une image assez bonne de ce que sont ces langages.

2.1. Le langage PL/1-TR

Fondements

Il est issu de PL/1 et a été étudié dans l'optique suivante :

- permettre à plusieurs programmes de partager des variables et des fichiers, de façon à ce que tous les objets partagés soient toujours dans un état cohérent, le mécanisme de partage étant étudié pour que le programmeur d'application temps réel puisse assurer que le système ne se bloque pas,

- permettre à un programme de lancer l'exécution d'un autre programme, appelé dans ce cas une *tâche,*

- permettre à une tâche de se tuer elle-même, ou bien de tuer toute autre tâche en laissant le système dans un état bien défini,

- permettre à une tâche de répondre à des interruptions et/ou à des transmissions de données à partir d'un périphérique.

Les états d'une tâche

Les états d'une tâche sont donnés par le graphe d'état suivant :

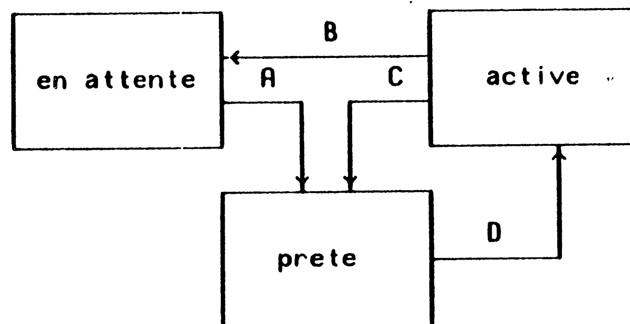


figure 1 : graphe d'états d'une tâche PL/1

Les transitions C et D sont gérées par le moniteur d'ordonnancement du système hôte.

La transition B correspond à l'exécution d'une instruction d'attente.

La transition A correspond à l'occurrence d'un évènement de fin d'attente.

Il est à noter que les transitions entre les états *"prête"* et *"active"* sont complètement liées à l'implantation et que, dans le standard de PL/1-TR, on trouve la remarque suivante :

> *Une implantation devra essayer d'assurer qu'aucune tâche prête n'a une priorité plus élevée qu'une tâche active, mais ceci n'est pas garanti. Le programmeur ne devra donc pas supposer qu'en donnant à une tâche la priorité maximum, celle-ci bloquera toutes les autres.*

Conclusion

De tels langages ont un niveau d'abstraction insuffisant quant à l'expression de la synchronisation des tâches ainsi qu'à l'expression du parallélisme et de la prise en compte du temps. Les avantages simplificateurs de l'introduction de la notion de tâche dans les langages, qui permettent de reporter les problèmes de l'ordonnancement sur le moniteur du système hôte, ont pour contrepartie le fait qu'on ne maîtrise pas de façon complète cet ordonnancement, et que l'on perd certainement en efficacité, voire même en compréhension du comportement temporel du programme.

Puisqu'on ne pourra pas, dans de tels langages, écrire des programmes dont le comportement temporel est très précis, il est inconcevable de programmer des applications temps réel à contraintes temporelles dures et strictes.

3. LES LANGAGES ISSUS DE C.S.P.

CSP a été étudié pour rendre le comportement de programmes parallèles indépendant, en un certain sens, des temps d'exécution des processus qui le composent. Les processus sont totalement asynchrones, ne partagent pas de mémoire et communiquent par un mécanisme de rendez-vous. Seule cette communication synchrone permet aux processus d'échanger des valeurs et de se synchroniser.

CSP ne permet pas de modéliser des systèmes temps réel pour deux raisons fondamentales :

- la correction d'un système temps réel dépend bien entendu du temps d'exécution des processus qui le composent,

- la communication avec l'environnement ne pourra pas être modélisée en termes de rendez-vous, en raison de la symétrie du rendez-vous : en cas d'absence de l'un des interlocuteurs, l'autre processus reste bloqué en attente. Or l'environnement d'un système temps réel ne peut ni ne doit attendre. CSP permet de modéliser des systèmes dans lesquels la synchronisation est mutuelle. Or une application temps réel (au sens où nous l'entendons) ne peut entrer dans ce cadre, car la communication entre le système informatique et le processus réel relève d'une synchronisation unilatérale : les processus du système doivent se synchroniser sur le processus à contrôler, jamais l'inverse.

Il se peut cependant que, dans certaines applications où la vitesse d'évolution du processus à contrôler est faible devant la vitesse d'évolution du système informatique, l'on soit sûr que le processus réel n'arrivera jamais aux rendez-vous avant les processus du système. Dans ce cas, on pourra utiliser CSP pour modéliser l'application. Mais, nous préfèrerons une approche où l'on n'a pas à tenir compte des vitesses relatives d'évolution des processus.

CSP n'est pas un langage de programmation, mais un outil de description et de spécification. Le mécanisme des rendez-vous en CSP implique que les processus se nomment mutuellement pour se synchroniser. Il n'est donc pas possible de créer un environnement de programmation avec une bibliothèque de programmes. Il faut connaître toute l'application pour écrire le programme. Si l'on veut, par exemple, contrôler une section critique à l'aide d'un processus sémaphore, celui-ci doit connaître tous les processus qui sont suceptibles d'entrer en section critique.

Pour créer des langages de programmation réels, la contrainte d'identification mutuelle dans les rendez-vous doit être relâchée. En ADA par exemple, seul l'un des deux processus nomme son partenaire. Autrement dit, il existe des processus qui acceptent des rendez-vous avec tout autre processus. En OCCAM, la communication se fait via des canaux . Elle ne peut être établie qu'entre deux processus ayant nommé le même canal.

3.1. Les outils temps réels pour ADA

Les outils de communication, de synchronisation et les primitives temps réel en ADA sont pour l'essentiel, les suivants [ADA 80] [Leverrand 82] :

- les programmes ont accès à un temps absolu (horloge globale),

- l'instruction **delay n** provoque l'attente *pendant au moins n secondes*,

- la communication entre tâches peut se faire

  . de manière asynchrone, au moyen de variables communes, mais il n'y a aucune assurance que ces variables restent dans un état cohérent dans le cas d'une implantation multi-processeurs.

  . De manière synchrone, par des rendez-vous : un seul des processus intervenant dans le rendez-vous nomme l'autre, avec son point d'entrée, le processus nommé se contente d'effectuer l'instruction **accept** sur ce point d'entrée. Lorsqu'un rendez-vous apparaît dans un système de commandes gardées (instruction **select**) il y a possibilité de spécifier la conduite à tenir en l'absence d'interlocuteur (attente pure, attente limitée dans le temps, rendez-vous immédiat).

- des priorités peuvent être associées aux tâches, mais seulement de manière statique,

- les interruptions sont assimilées à des rendez-vous de priorité supérieure à toutes les tâches du programmes.

Le langage ADA permet d'exprimer facilement les problèmes de synchronisation et de communication que l'on rencontre habituellement en programmation système.

Par l'introduction des interruptions, de la notion de temps et dans une certaine mesure des priorités, les auteurs ont voulu en faire *aussi* un langage temps réel.

ADA est un langage de haut niveau, asynchrone. La manipulation du temps est approximative et l'expression des contraintes temporelles est difficile, voire impossible. En guise de conclusion, nous allons donner deux petits exemples qui montrent combien le caractère asynchrone d'un langage rend délicate la conception de programme.

Exemple 1

Petit exemple de manipulation du temps

```
select
  .
  .
  when CLOCK = midi => .....
  or
  .
  .
end select
```

Cette instruction n'a aucun sens car selon toute probabilité, l'évaluation de l'horloge n'aura jamais lieu exactement à midi.

Exemple 2

Il existe, pour chaque entrée E d'une tâche, un attribut E'COUNT qui représente le nombre de demandes en attente sur cette entrée. COUNT est un cas très particulier de variable partagée, son maniement est donc délicat.

```
select
  .
  .
  when (E'COUNT ≥ 2) => accept E; accept E; .....
  or
  .
  .
end select
```

Même si la garde E'COUNT ≥ 2 est vraie, il se peut très bien qu'à l'exécution de la première instruction il n'existe plus qu'une seule demande de rendez-vous sur l'entrée E, les autres ayant été retirées en raison de l'expiration des délais spécifiés dans les instructions d'appel. Cette commande peut donc très bien se bloquer en attente d'un deuxième rendez-vous, malgré la précaution prise.

Ces exemples inspirent les deux remarques suivantes :

- Il est très difficile de programmer dans un langage asynchrone car le caractère asynchrone introduit une grande part d'indéterminisme.

- Il reste impossible d'exprimer des contraintes temporelles, donc, à plus forte raison, d'en détecter la violation ou d'en assurer le respect.

4. PROCESSUS SEQUENTIELS SYNCHRONES

Le caractère asynchrone des langages est source de difficultés pour le programmeur, comme nous avons essayé de le montrer. Nous allons présenter dans ce paragraphe un langage synchrone. Dans ce cadre la sémantique du langage est simplifiée, puisqu'on ne prend pas en compte la dérive temporelle due à la notion de concurrence.

Dans les langages asynchrones, l'architecture sous-jacente est mono-processeur et la concurrence est modélisée par un entrelacement indéterministe d'actions atomiques. Dans une approche synchrone, il n'est pas question de concurrence ou de quasi-parallélisme, mais de vrai parallélisme. L'architecture sous-jacente est cette fois une infinité de processeurs infiniment rapides.

4.1. Le langage ESTEREL

Le langage ESTEREL a été conçu pour être un véritable langage de programmation temps réel. Le projet, développé à Sophia Antipolis comprend bien entendu la définition du langage, mais aussi, parallèlement, l'étude de sa sémantique formelle. Il est à noter , et c'est là un point important pour ce langage, que la définition d'une sémantique opérationnelle complète a permis de spécifier et de réaliser un compilateur.

Nous présentons dans ce paragraphe quelques notions qui sont à la base du langage, ainsi que les primitives qui définissent son noyau.

4.1.1. Concepts de base

Nous avons vu dans les langages précédents, que la suite d'instructions classiques "**delay** 5 s ; **delay** 3 s" avait une sémantique floue, et que quoi qu'il arrive, l'équivalence avec "**delay** 8 s" était fausse. ESTEREL a été conçu pour que de telles instructions aient une sémantique précise et que l'équivalence soit réalisée. Pour cela deux hypothèses ont été faites :

- les instructions d'attente d'un évènement se terminent *exactement* sur l'occurence de l'évènement,

- Le contrôle et les calculs sont effectués *en un temps nul.*

Ces hypothèses sont regroupées sous le terme *fort synchronisme.* Dans un tel cadre, il est clair que "**delay 5 s ; delay 3 s**" est strictement équivalent à "**delay 8 s ;**" et que le délai associé à cette instruction est *exactement 8 secondes.*

4.1.2. Signaux et évènements

L'unité de base de la communication en ESTEREL est le signal. Un signal possède un nom, et éventuellement un type qui est le type de la valeur qu'il transporte.

Un évènement est une diffusion instantanée d'un ensemble de signaux portant éventuellement des valeurs.

Un évènement n'est pas rémanent, son information est perdue si "personne n'est là pour le recevoir". La communication est du type radio.

Il y aura transmission de l'information si un émetteur émet en présence d'au moins un récepteur. Personne n'est bloqué en émission, ceci permettant de considérer la communication avec l'environnement de la même façon que la communication entre processus.

Il peut y avoir plusieurs émissions simultanées d'un même signal. Si ce signal est porteur d'une valeur, on peut définir une opération de composition sur ces valeurs, qui permettra de calculer la valeur résultante émise. Par exemple si un signal porte une valeur du type ensemble, on peut définir l'union ensembliste comme opération de composition; ainsi la valeur résultante de plusieurs émissions simultanées d'un signal sera l'union des ensembles portés par chaque émission. Si au contraire, plusieurs émissions simultanées d'un signal sont illicites, la loi de composition donnera une valeur "erreur".

4.1.3. Le temps

Dans une application temps réel, le temps n'est en fait rien d'autre que le flot des évènements que le programme reçoit ou génère. La notion de temps en ESTEREL est donc multiforme, puisqu'il y a autant d'horloges qu'il y a d'évènements répétitifs.

L'hypothèse de fort synchronisme conduit naturellement à un modèle discret du temps. Un observateur extérieur au système pourra voir la séquence des évènements d'entrée du système et la séquence des évènements de sortie du programme, appelées respectivement histoire d'entrée et histoire de sortie.

Un programme P reçoit une histoire d'entrée

$$H = E_1 E_2 \ldots\ldots E_n \ldots\ldots$$

et émet une histoire de sortie

$$G = F_1 F_2 \ldots\ldots F_n \ldots\ldots$$

où l'indice n représente le n-ième instant des histoires de P.

4.1.4. Le contrôle

En ESTEREL le contrôle prend un temps nul (hypothèse de fort synchronisme). Il y a trois schémas de contrôle principaux :

- le séquencement : **i; i'**

lorsque l'instruction i est terminée, le contrôle passe *immédiatement* à l'instruction i'.

- la mise en parallèle : **i || i'**

le contrôle est passé simultanément à i et à i'. Cette instruction se termine *dès que* i et i' se terminent.

- le contrôle de bloc : **tag T in .... exit T .... end**

L'instruction **exit** ne peut apparaître que dans ce contexte. Son exécution a pour effet de *terminer instantanément* l'instruction "**tag T in .... end**.

4.1.5. les instructions

- l'instruction vide : **nothing**

Cette instruction ne *fait rien et ne prend aucun temps*

- l'affectation : **X := exp**

X est une variable, et exp une expression simple, pour que l'hypothèse de synchronisme fort soit raisonnable, car l'évaluation de l'expression exp et son affectation à X *prend un temps nul*

- la boucle : **loop i end**

Cette instruction est équivalente à i; i; ....; i; ......

- la conditionnelle : **if C then i else i'**

Construction conditionnelle classique, à ceci près que la condition C est évaluée *en un temps nul* et que suivant sa valeur, le contrôle est passé *instantanément* à i ou à i'.

4.1.6. La communication

- émission d'un signal : **emit s (exp)**

Emission d'un signal s portant la valeur exp. exp est évaluée et s est émis *en un temps nul* .

- réception d'un signal : **do i upto s (x) | do i uptonext s (x)**

i est une instruction, s un signal et x une variable. Dès que le signal s est présent, l'instruction i est *immédiatement* interrompue et x est affectée avec la valeur portée par le signal s. Si i se termine avant l'arrivée de s, on attend le signal et l'affectation termine l'instruction.

Si le signal s est présent quand on exécute l'instruction "**do i upto s (x)**", l'instruction i n'est pas du tout exécutée. En revanche, si le signal est présent quand on exécute l'instruction "**do i uptonext s (x)**", l'instruction i est exécutée jusqu'à l'arrivée du prochain signal s. C'est la seule différence entre ces deux instructions.

Remarques

i) "**do i uptonext s**" est la seule instruction qui requiert explicitement du temps puisqu'elle demande le passage d'une unité de temps relatif à s dans tous les cas.

ii) A l'aide de ces primitives on peut construire des instructions paradoxales par exemple

do emit s upto s

Si s n'est pas présent, on exécute "**emit** s" (qui ne prend aucun temps) alors s est présent donc on n'exécute pas "**emit** s" donc s n'est pas présent et il faut exécuter "**emit** s"...

Ce genre d'incohérence sera détectée à la compilation grâce à l'analyse de causalité des signaux (exprimée dans un graphe de dépendance) et l'on pourra ainsi interdire que l'émission d'un signal dépende de sa reception au même instant.

4.1.7. Diverses constructions

A partir des instructions primitives définies précédemment, on peut construire de nouvelles instructions utiles.

- attente

await s (x) ≡ **do nothing upto** s (x)
awaitnext s (x) ≡ **do nothing uptonext** s (x)

- cadencement

do i **uptoeach** s (x) ≡ **loop do** i **uptonext** s (x) **end**
every s (x) **do** i **end** ≡ **await** s (x); **do** i **uptoeach** s (x)
everynext s (x) **do** i **end** ≡ **awaitnext** s (x); **do** i **uptoeach** s (x)

- **Remarques**

i) Les trois dernières constructions ne diffèrent que par leur comportement au début de leur exécution :

- dans la première, i commence immédiatement,
- dans la deuxième, i commence dès que s est présent,
- dans la troisième, i ne commence que sur la première occurrence de s strictement postérieure au début de l'exécution de l'instruction.

ii) Il est à remarquer également que puisque **await** s (x) ne prend pas de temps, la construction

await s (x); **await** s (x);......; **await** s (x); ≡ **await** s (x)

Ceci choquera le programmeur habitué aux langages classiques. Cependant la même construction à base de **awaitnext** donnera le résultat escompté, c'est-à-dire l'attente d'un nombre donné d'émissions du signal s.

Les deux derniers exemples que nous présentons sont assez représentatifs de la façon d'utiliser ESTEREL, et montrent la puissance de l'hypothèse de fort synchronisme. La programmation en ESTEREL demandera certainement un peu d'habitude, car l'instantanéïté des réactions du programme n'est pas un concept classique, et la structure des programmes ESTEREL demandera beaucoup d'attention.

- test de signaux

```
inpresence s (x) do i end ≡ tag T in
                                 do exit T upto s (x); i
                            end

inabsence s (x) do i end ≡ tag T in
                                 inpresence s (x) do exit T end ; i
                           end
```

- chien de garde

```
do i watching s (x) abnormal i' end ≡ tag T in
                                           do i; exit T upto s (x);
                                           i'
                                      end
```

Si le signal s arrive avant que l'instruction i ne soit terminée (c'est-à-dire, à cause du synchronisme fort, avant que l'instruction **exit** ne soit exécutée) le contrôle passe immédiatement à i' qui s'exécute ; sinon l'instruction se termine avec la terminaison de i (puisque **exit** ne prend pas de temps). Ceci est la définition stricto sensu d'un chien de garde. A ce niveau d'abstraction, on n'a pas à tenir compte d'une quelconque dérive temporelle liée aux contraintes de l'implantation sur une machine réelle. Ceci est tout à fait conforme à la notion de langage de haut niveau.

4.1.8. Discussion d'ESTEREL

Le langage ESTEREL est remarquable pour la rigueur de l'étude dont il a été l'objet. En effet une sémantique opérationnelle complète a été donnée, et a conduit à la spécification et à l'écriture d'un compilateur. La compilation d'un programme ESTEREL produit un automate d'états fini (facilement traductible dans n'importe quel langage de programmation classique) après des vérifications sémantiques importantes sur le programme telles que :

- vérification de l'absence de boucle infinie de durée nulle,
- vérification de la cohérence du système de communication grâce à la définition d'une relation de dépendance entre signaux. La cohérence des signaux se réduit à une analyse de ce graphe de causalité. Il doit être sans boucle.

L'hypothèse de fort synchronisme en ce qui concerne les signaux (émission, reception instantanées) est réalisable puisque la communication ne produit pas de code.

On exprime les contraintes de temps et l'on peut détecter leur violation : c'est la violation du fort synchronisme, c'est-à-dire le cas où un évènement arrive alors que les actions provoquées par le précédent ne sont pas terminées.

Nous émettrons tout de même quelques réserves :

- seule une implantation séquentielle permet la réalisation simple des contraintes du synchronisme fort (traduction en automate d'états fini),
- selon notre expérience du langage, les performances des programmes sont fortement dépendantes du style de programmation. Il s'ensuit qu'une bonne habitude du langage, voire même une certaine connaissance du compilateur, sont nécessaires.

Néanmoins , à notre avis, ESTEREL reste de loin le meilleur langage temps réel opérationnel existant.

5. LES LANGAGES GRAPHIQUES

En France, le principal langage de ce type utilisé en programmation temps réel est le GRAFCET (Graphe de Contrôle Etape Transition). Ce langage, défini par le groupe "Systèmes Logiques" de l'AFCET, a eu du mal à se stabiliser dans une version standard, malgré les efforts réalisés dans ce sens [Moalla 81] [Afcet 83].

Dérivé des réseaux de Petri [Brams 83] [Peterson 77] (dont nous ne parlerons pas ici), ce langage a perdu en rigueur si ce n'est en pouvoir d'expression par rapport à ceux-ci (il est impossible par exemple en GRAFCET de décrire un sémaphore sans faire appel à des variables d'état et à une interprétation).

Un GRAFCET permet, en fait, de décrire la structure de contrôle d'un programme ; les actions associées aux places étant décrites dans un langage séquentiel quelconque.

On ne peut pas structurer un GRAFCET, toutes les variables sont globales à l'ensemble du programme. Il est donc délicat de concevoir une application en GRAFCET de façon modulaire.

Cependant ce langage est assez bien adapté à la programmation des automates industriels dont le domaine d'application concerne surtout les automatismes logiques. Dans ce domaine, il est vrai que les problèmes de conception ne dépassent guère l'étude de la structure de contrôle, et consistent en la synchronisation des différentes actions à faire sur la réception de conditions et d'évènements externes.

Le temps est introduit de façons différentes suivant les constructeurs d'automates programmables. La sémantique temporelle est très fortement liée à la façon d'interprèter le GRAFCET qui n'est pas unifiée. On peut se demander s'il n'y a pas autant de GRAFCET différents qu'il y a d'interpréteurs du langage.

De plus, le succès actuel de l'approche graphique nous paraît discutable. On ne peut s'empêcher de faire référence à l'évolution de la programmation classique dans laquelle on a abandonné l'outil graphique d'analyse (organigramme) au profit d'un outil textuel permettant d'une part la structuration des programmes, et d'autre part une meilleure lisibilité en ce qui concerne les gros programmes. Le domaine de l'automatisme est en pleine évolution grâce à l'introduction massive de microprocesseurs de plus en plus performants, et permettant le contrôle d'automatismes de plus en plus compliqués. Ces problèmes sont très souvent couplés à des contraintes de sûreté. Si, au niveau matériel, l'introduction de redondances dans les systèmes permet, dans une certaine mesure de se prémunir contre des défaillances physiques de l'automate, rien n'est encore prévu au niveau des logiciels. La validation des programmes ne peut être obtenue que par simulation. Il semble donc dommage que l'accent ne soit pas mis sur l'étude de langages de programmation nouveaux permettant, une programmation plus fiable.

6. LES LANGAGES NON PROCEDURAUX

Citons les langages non procéduraux les plus connus en les classifiant sommairement. Cette classification n'a pour but que de définir quelles sont, à notre avis, les caractéristiques qui peuvent paraître intéressantes dans le domaine du temps réel.

Parmi toutes les approches possibles de la programmation, on peut citer par exemple :

- les langages fonctionnels (FP, KRC, LISP pur, ML, LTS [Babiker-all 84]),

- les langages à assignation unique (LAU [Berger-all 82], SAUGE [Lecouffe 81]),

- les langages définitionnels (LUCID [Ashcroft-Wadge 77]),

- les langages orientés flots de données [Ackerman 82] (VAL [Mc Graw 82], SIGNAL [Gautier 85], CAJOLE [Hankin 81]) [Dennis 74],

Ces langages ont été étudiés dans de multiples buts :

- Pour se dégager de l'approche traditionnelle Von Neuman de la programmation [Backus 78],

- Pour s'adapter à de nouvelles architectures de machines plus parallèles [Ackerman 82],

- Pour permettre la manipulation formelle de programmes, dans le but de faire des preuves de correction [Ascheroft-Wadge 77] [Backus 78],

Le domaine du temps réel peut participer d'une telle approche, mais à l'heure actuelle, nous ne connaissons que très peu d'exemples. Seul SIGNAL (orienté flots de données) est conçu pour être temps réel, et, dans une certaine mesure LTS (fondé sur ML) prend en compte des aspects temporels.

Ces nouvelles approches de la programmation ont le souci de permettre une définition simplifiée de la sémantique des langages, et par suite de simplifier la programmation.

Nous présentons brièvement dans ce chapitre le langage LTS, le langage SIGNAL, développé à Rennes et "contemporain" de LUSTRE (puisqu'il n'était pas défini quand nous avons commencé notre étude), ainsi que le langage LUCID. Ce dernier ne prend pas en compte la notion de temps, mais il est important puisqu'il a servi de point de départ à LUSTRE.

6.1. Le langage LTS

LTS est un langage fonctionnel très proche de ML dont il est issu. Le domaine d'application de LTS est la spécification et la simulation de systèmes matériels. Il possède donc un ensemble de primitives spécialement étudiées pour ce domaine, nous ne les présenterons pas.

Un système matériel n'est pas très éloigné conceptuellement d'un système temps réel. Les notions de temps, de délais de réponse, d'évènements et d'histoires y apparaissent. Il était donc intéressant de présenter une partie de ce langage.

6.1.1. Signaux et constantes

Pour la description de matériel au niveau comportemental, on a besoin de décrire l'évolution des variables au cours du temps. On utilise pour cela des entités, appelées signaux, qui sont des fonctions du temps dans un domaine de valeurs. Les constantes seront des signaux constants au cours du temps.

6.1.2. La notion de temps

Le temps en LTS est discret et absolu. Un signal est une fonction de $\mathbb{N}$ dans les valeurs. On peut donc voir un signal comme une séquence de valeurs.

6.1.3. Les fonctions temporelles

Il existe trois fonctions principales de manipulation du temps

- La fonction "**time**"

Elle permet à tout instant de tester l'heure. Elle n'a été introduite que pour permettre d'initialiser des signaux pour le simulateur. Le temps en LTS a une origine : la date 0.

- La fonction "**last**"

Cette fonction permet de faire référence au passé. La valeur de **last** (X) à l'instant n est la valeur de X à l'instant n-1. A la date 0, **last** (X) n'est pas définie puisque ce serait la valeur de X à l'instant -1. Pour résoudre ce problème, les auteurs ont considéré que **last** (X) aurait la valeur de X à la date 0.

- La fonction "**atmostrecent**"

Cette fonction a deux paramètres, un signal quelconque et un signal booléen. Elle renvoie la valeur qu'avait le signal la dernière fois que le booléen était vrai.

```
atmostrecent (X, Y) = if X is
                          true then Y
                          false then atmostrecent (last (X), last (Y))
                      end
```

6.1.4. Conclusion

Le mécanisme des types en LTS est analogue à celui de ML. Il existe un constructeur **where** qui permet de structurer et de hiérarchiser des définitions. La forme récursive pour la définition des fonctions est largement utilisée et rend la programmation élégante. En revanche, la notion de temps est un peu pauvre, il n'existe qu'un seul temps universel et absolu. Si pour la spécification et la simulation de systèmes matériels ceci peut convenir, ce n'est pas le cas dans le domaine du temps réel, où l'on peut avoir besoin de décrire des phénomènes ayant des vitesses d'évolution très différentes, sans pour cela les ramener à une même horloge absolue.

De plus, l'absence de primitive d'itération conduit à une programmation récursive qui peut paraître mal adaptée à des descriptions temporelles car on a l'impression de recalculer le passé, comme dans la fonction **atmostrecent** par exemple. Cette remarque est bien évidemment subjective puisque la forme récursive permet de définir la fonction et l'on peut envisager une implantation efficace qui ne perd pas les traces du passé nececessaires.

LTS est avant tout un langage de spécification et de simulation. Il n'a pas été conçu pour être un langage de programmation temps réel. LTS a été étudié avec le souci d'en faire un langage de spécification précis, et de permettre des manipulations formelles des programmes dans le cadre de la génération de circuits intégrés (compilation de silicium).

6.2. Le langage SIGNAL

Le langage SIGNAL présente plusieurs caractéristiques intéressantes des langages temps réel d'une part, et des langages à flots de données d'autre part.

Un programme SIGNAL comprend

- La spécification d'un réseau statique orienté (bloc diagramme dans le langage des automaticiens) exprimant exactement les dépendances entre les données. Les données circulent sur les arcs du réseau ; aux noeuds du réseau, des actions sont effectuées par des processus (ou boites noires). Le mécanisme de construction des réseaux est inspiré de l'algèbre des réseaux [Milner 79].

- La description de processus agissant aux noeuds du réseau. Les processus élémentaires sont appelés générateurs et sont divisés en deux classes
    - processus arithmétiques (opérateurs classiques +, *, ...)
    - processus temporels (spécifiques à SIGNAL)

- La notion de signal donne une portée temporelle aux règles d'évolution des flots de données.

SIGNAL est présenté comme un langage temps réel, synchrone sans effet de bord, adapté à l'expression et à l'exploitation du parallèlisme.

6.2.1. Construction des réseaux statiques

La construction des réseaux statiques se fait par interconnexion de processus élémentaires (générateurs) grâce au mécanisme classique d'identification par nom. Ceci impose l'utilisation de mécanismes de masquage et renommage des ports des processus. Il y a cinq constructeurs de base.

La composition parallèle

On peut composer deux processus en parallèle si et seulement si l'ensemble des noms des ports de sortie de l'un a une intersection vide avec l'ensemble des noms des ports de sortie de l'autre.

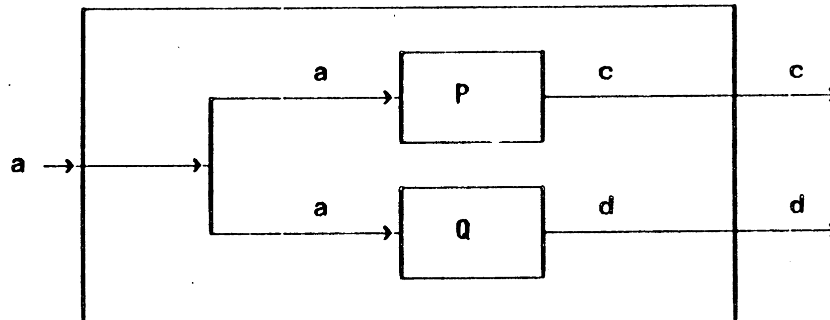


figure 1 : Composition parallèle P & Q

Le bouclage

Etant donnés un processus P, et un nom X. Le bouclage de P par X connecte tous les ports de sortie de nom X au port d'entrée de nom X.

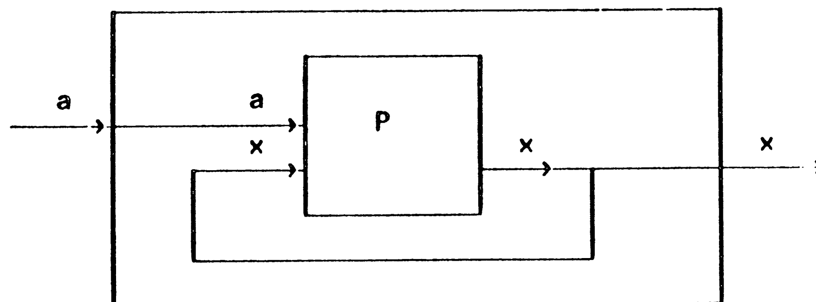


figure 2 : Le bouclage de P par x (P @ x)

La composition

Les ports d'entrées de P sont reliés aux ports de sortie de Q de mêmes noms et inversement. Un port d'entrée connecté à un port de sortie n'est plus visible.

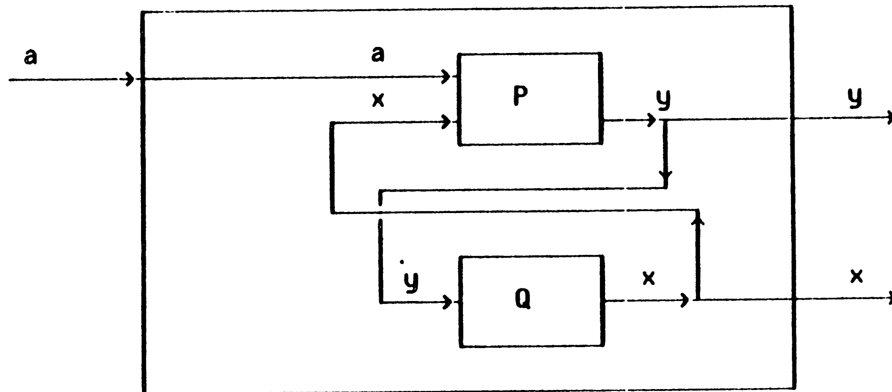


figure 3 : Composition P / Q

La séquence

Les ports de sortie de P sont connectés aux ports d'entrée de Q de mêmes noms. Seuls les ports d'entrée de P ayant un nom différent d'un port de sortie de Q sont visibles en sortie (i.e. de l'extérieur)

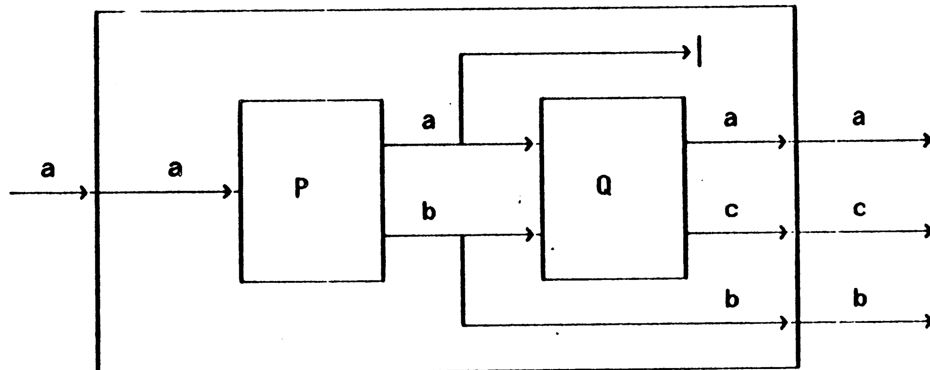


figure 4 : Composition séquentielle P ; Q

Construction de motifs réguliers

Pour un ensemble de processus indexés, on peut définir une connexion générique de la façon suivante:

dopar i until n of P (i) od ≡ P (1) & P (2) & ..... & P (n)
docom i until n of P (i) od ≡ P (1) | P (2) | ..... | P (n)
doseq i until n of P (i) od ≡ P (1) ; P (2) ; ..... ; P (n)

1.2.2. Les horloges

On appelle *flot* une suite de données typées. Les flots circulent sur le réseau statique le long des arcs, et sont transformés par le générateur selon les principes de synchronisation ci-dessous.

On appelle *horloge* une fonction croissante de l'ensemble des entiers naturels (IN) dans un ensemble dénombrable Ω totalement ordonné. A chaque flot est associée une horloge, permettant de spécifier les instants auxquels les valeurs des flots sont disponibles. On appèlera *signal* un couple (flot, horloge).

Il n'existe pas d'horloge absolue, mais pour chaque processus P, on définit sur l'ensemble Ω_P de ses horloges, des relations de dépendance modulo une horloge de référence locale au processus.

Ω est partiellement ordonné. La relation $\omega_1 \preccurlyeq \omega_2$ (où ω_1 et ω_2 sont des horloges) signifie que l'ensemble des instants de ω_1 est contenu dans l'ensemble des instants de ω_2.

$$\text{si } \omega_1 \preccurlyeq \omega_2 \text{ et } \omega_1 \succcurlyeq \omega_2 \text{ on notera } \omega_1 \equiv \omega_2$$

On dira que deux horloges ω_1 et ω_2 sont compatibles et l'on notera

$$\omega_1 \sim \omega_2$$

si et seulement si il existe une horloge ω telle que

$$\omega_1 \preccurlyeq \omega \text{ et } \omega_2 \preccurlyeq \omega$$

On dira que ω domine ω_1 et ω_2. L'ensemble des horloges d'un processus n'est généralement pas un treillis, mais l'ensemble des horloges dominées par une horloge donnée est un treillis. On notera les opérateurs inf et sup respectivement $\vee$ et $\wedge$. L'horloge qui définit uniquement l'instant initial, notée $\perp$, est la borne inférieure de toutes les horloges.

6.2.3. Générateurs temporels primitifs

Le retard : x is y (n) $m init ...

x est une fenêtre de y, de longueur n, retardée de m. Les horloges de x et de y (notées ω_x et ω_y) vérifient la propriété suivante:

$$\omega_x \equiv \omega_y$$

La conditionnelle primitive : x is y if c

Lorsque le signal y et le signal c sont tous les deux disponibles et que le signal c porte la valeur vrai, alors la valeur de y est délivrée sur la sortie x.

$$\text{si } \omega_c \sim \omega_y \text{ alors } \omega_x \equiv \omega_y \wedge \text{true (c)}$$

où true (c) désigne l'horloge dont les instants sont définis par les instants où c est vrai.

Le mélange : x is merge (a, b)

La valeur des signaux a et b est délivrée en sortie chaque fois que l'une ou l'autre des deux entrées est disponible. Si les deux entrées sont disponibles simultanément, c'est la valeur portée par a qui est délivrée. Les horloges vérifient les propriétés suivantes.

$$\text{si } \omega_a \sim \omega_b \text{ alors } \omega_x \equiv \omega_a \wedge \omega_b$$

Opérateurs arithmétiques

Tous les opérateurs arithmétiques sont synchrones : les horloges de toutes les entrées et de toutes les sorties sont identiques.

Ces opérateurs primitifs, hormis le retard, ne sont pas dans le langage, mais ils permettent de définir des opérateurs plus puissants offerts au programmeur.

6.2.4. Les opérateurs du langage

Nous présentons les opérateurs du langage et les relations sur les horloges associées

Extension : x extends y at b

Les entrées y et b sont de types quelconques et la sortie x est du même type que y.

L'effet de cet opérateur est le suivant : la dernière valeur du signal y est délivrée sur la sortie x quand une valeur est disponible sur l'une au moins des deux entrées (les valeurs du signal b ne sont pas concernées par ce générateur, seule son horloge est prise en compte).

Les relations temporelles sont les suivantes :

$$\text{si } \omega_y \sim \omega_b \text{ alors } \omega_x \equiv \omega_y \vee \omega_b$$

Extraction : x extracts y at b

Les entrées y et b sont de types quelconques et la sortie x est du même type que y.

L'effet de cet opérateur est le suivant : la valeur courante du signal y est délivrée sur la sortie x quand les entrées y et b sont toutes deux disponibles (les valeurs du signal b ne sont pas concernées par ce générateur, seule son horloge est prise en compte).

Les relations temporelles sont les suivantes :

$$\text{si } \omega_y \sim \omega_b \text{ alors } \omega_x \equiv \omega_y \wedge \omega_b$$

Comptage : n counts y [after b] [from b]

Les entrées y et b sont de types quelconques et la sortie n est de type entier.

L'effet de ces opérateurs est le suivant :

- n **counts** y compte les occurrences de y
- n **counts** y **after** b compte les occurrences de y postérieures
  à la dernière occurrence de b.
- n **counts** y **from** b compte les occurrences de y non antérieures
  à la dernière occurrence de b.

Les relations temporelles sont les suivantes :

$$\omega_n \equiv \omega_y \text{ pour le premier compteur}$$
$$\omega_n \equiv \omega_y \vee \omega_b \text{ pour les deux suivants}$$

La conditionnelle : if c then P else Q fi

C'est un générateur qui prend en paramètres deux processus P et Q pour en créer un nouveau (**if** c **then** P **else** Q **fi**).

Si l'on note respectivement ?P et !P l'ensemble des entrées et l'ensemble des sorties d'un processus P

$$?(\textbf{if } c \textbf{ then } P \textbf{ else } Q \textbf{ fi}) = \{c\} \cup ?P \cup ?Q$$
$$!(\textbf{if } c \textbf{ then } P \textbf{ else } Q \textbf{ fi}) = !P \cup !Q$$

Les sorties de P (respectivement de Q) ne sont disponibles que lorsque le signal c porte la valeur vrai (respectivement faux). Les sorties de P et Q ayant le même nom sont mélangées et émanent de P ou de Q suivant la valeur portée par le signal c.

Les relations temporelles sont les suivantes :

si pour tout $x \in\ !P \cup\ !Q$ on a $\omega_x \sim \omega_c$ alors

- pour tout $y \in\ !P$, $y \notin\ !Q$

$$\Omega_y = \omega_{yP} \wedge \text{true } (c)$$

- pour tout $y \notin\ !P$, $y \in\ !Q$

$$\Omega_y = \omega_{yQ} \wedge \text{false } (c)$$

- pour tout $y \in\ !P \cap\ !Q$

$$\Omega_y = \omega_{yP} \wedge \omega_{yQ} \wedge \omega_c$$

true (c) (respectivement false (c)) désignent les instants où c porte la valeur vrai (respectivement la valeur faux)

Remarque

Dans l'instruction **if** c **then** P **else** Q **fi** on peut omettre (**then** P) ou (**else** Q)

6.2.5. Conclusion

SIGNAL contient les principales fonctionnalités nécessaires à la programmation temps réel. La sémantique des réseaux est définie dans [Gautier 85]. Nous en ferons toutefois quelques critiques.

- Le mécanisme de construction des réseaux nous paraît un peu lourd, le renommage et le masquage des noms des ports, l'encapsulation ont le même pouvoir d'expression et de description qu'un mécanisme de définition de fonctions hierarchisées, et la description du réseau aurait pu être faite par simple jeu de composition fonctionnelle et passage de paramètres, dont la sémantique est peut-être comprise par un ensemble plus large d'utilisateurs potentiels.

- Il n'y a pas de variable horloge, mais tout signal (suite, horloge) peut intervenir en tant qu'horloge. Les règles sur les horloges, moins strictes qu'en LUSTRE, risquent de causer des désagréments au programmeur, dans la mesure où la compilation va essayer de donner un sens à une expression éventuellement erronée.

- Le calcul des horloges n'est pas complètement défini, il est basé sur [Benveniste 85], mais pour l'instant rien n'est fait au niveau du langage.

7. LE LANGAGE LUCID

LUCID a été défini pour permettre à la fois la programmation et la preuve d'algorithmes [Ashcroft-Wadge 77]. Le but recherché est d'avoir un seul formalisme pour la programmation et pour la preuve. L'une des manières pour atteindre ce but consiste à programmer à l'aide d'assertions, plutôt qu'à l'aide d'instructions qui modifient le contenu d'une mémoire. L'idée de base est de décrire la suite des valeurs que prend une variable, plutôt que la suite des actions à réaliser sur cette variable pour construire cette suite. Dans cette optique, les programmes seront des

systèmes d'équations définissant des variables.

7.1. Variables et opérateurs

Toute variable en LUCID est une suite, il faut donc que les constantes soient des suites également. Ainsi 0, par exemple, dénotera la suite constante, partout égale à 0. Pour définir une suite deux possibilités sont offertes ; d'une part, définir une suite directement à partir d'autres suites par une expression :

$$Y_n = U_n + V_n \text{ pour tout } n \in \mathbb{N}$$

d'autre part, définir une suite de façon récurrente :

$$X_0 = 0$$
$$X_{n+1} = X_n + 1 \text{ pour tout } n \geq 1 \in \mathbb{N}$$

LUCID doit permettre d'exprimer les définitions de suites sous ces deux formes. Pour la première forme, on écrira simplement en LUCID

$$Y = U + V$$

Si l'on voit la séquence comme une fonction des entiers naturels $\mathbb{N}$ dans un ensemble de valeurs, en LUCID on exprime donc l'égalité de deux fonctions.

Pour la deuxième forme de définition, on dispose de deux opérateurs, **first** et **next**, qui permettent de définir la suite X par deux équations, de la façon suivante :

$$\textbf{first } X = 0$$
$$\textbf{next } X = X + 1$$

first X est la suite constante égale au premier terme de X

$$\textbf{first } X = \langle X_0, X_0, \ldots, X_0, \ldots \rangle$$

si Z = **next** X alors Z est la suite définie à partir de X par

$$Z = \langle X_1, X_2, \ldots, X_n, \ldots \rangle \text{ c'est-à-dire } Z_n = X_{n+1} \ \forall n \in \mathbb{N}$$

Les variables sont des suites infinies de valeurs. LUCID permet également d'extraire de ces suites des valeurs particulières. On dispose pour cela d'un opérateur particulier noté **as soon as** ou en abrégé **asa** tel que

$$X = Y \textbf{ asa } P$$

définit X constant tel que

- $X = \langle Y_j, Y_j, \ldots, Y_j, \ldots \rangle$ s'il existe j tel que P_j = true et $\forall\, i \in [0, j\text{-}1]\ P_i$ = false
- $X = \langle \bot, \bot, \ldots, \bot, \ldots \rangle$ s'il n'existe pas de tel j

où $\bot$ est un élément particulier de l'ensemble des valeurs qui désigne la valeur indéterminée.

Il existe également un opérateur dérivé, noté **followed by** ou bien en abrégé **fby** qui permet de définir une suite récurrente en une seule équation.

$$X = Y \textbf{ fby } Z$$

est équivalent à

$$\textbf{first } X = Y$$
$$\textbf{next } X = Z$$

X est la suite dont le premier terme est le premier terme de Y et les suivants sont les termes de Z. Par exemple

$$\textbf{first } X = 0$$
$$\textbf{next } X = X + 1$$

peut être désormais remplacé par

$$X = 0 \textbf{ fby } X + 1$$

et définit la suite des entiers naturels. $X = \langle 0, 1, 2, \ldots\ldots \rangle$

7.2. Premier exemple

On veut faire un programme qui extrait la racine carrée entière d'un nombre entier positif, donné en paramètre au programme. Ce programme va être écrit très simplement en LUCID en construisant la suite des carrés des entiers, et en extrayant de la suite des entiers l'élément qui correspond à la racine carrée entière recherchée.

```
n = first input
first i = 0
first j = 1
next i = i + 1                 suite des nombres entiers
next j = j + 2×i + 3           suite des carrés
output = i asa j > n           extraction du résultat
```

input et *output* sont deux mots reservés désignant respectivement l'entrée et la sortie du programme.

i est la suite <0, 1, ..., n-1, ...>

j est la suite <1, 4, ..., n^2, ...>

output sera la suite <k, k, ..., k, ...>

avec k tel que $(n+1)^2 > k \geq n^2$

LUCID a été conçu pour exprimer des itérations. Le programme donné en exemple est un programme itératif. Il vient donc naturellement la possibilité de décrire des itérations imbriquées.

7.3. Les opérateurs "latest" et "latest^{-1}"

LUCID offre la possibilité d'imbriquer des itérations à l'aide de deux nouveaux opérateurs **latest** et **latest**$^{-1}$. L'opérateur **latest** a pour effet de figer l'indice de la suite à laquelle il s'applique et ainsi

$$\text{si } I = < I_0, I_1, \ldots, I_n, \ldots >$$

alors V = **latest** I définit V tel que

$$V = < V_{00}, V_{10}, \ldots, V_{n0}, \ldots, V_{01}, \ldots, V_{02}, \ldots >$$

et $\forall\ (i, j) \in \mathbb{N}^2\ \ V_{ij} = I_i$

On peut voir l'effet de l'opérateur **latest** sur la suite I comme l'addition d'un indice à la suite pour "l'accélérer". On a choisi d'écrire l'indice qui varie le plus vite le plus à gauche.

latest^{-1} est l'opération inverse. Si A_{ij} est une variable telle que

$$A_{ij} = B_j \ \forall\ j \in \mathbb{N}$$

alors **latest**$^{-1}$ est définie et

$$\mathbf{latest}^{-1}\ A_{ij} = B_j$$

Les mécanismes qui permettent de passer d'un niveau d'imbrication à l'autre sont ramenés à une manipulation d'indices.

- addition d'un indice pour définir une boucle de niveau inférieur
- suppression d'un indice pour utiliser à un niveau supérieur des valeurs d'une boucle plus interne.

Exemple

Soit un programme qui détermine si un entier n est premier ou non

```
n = first input
first i = 2
first j = latest i × latest i
next j = j + i
idivn = j eq n asa j ≥ n
next i = i + 1
output = not (latest⁻¹ idivn) asa latest⁻¹ idivn or (i × i ≥ n)
```

Remarques

- **latest** i peut être vu comme un moyen de figer la progression de la suite i pour le calcul de idivn

- **latest**$^{-1}$ est défini puisqu'il s'applique à une suite constante produite par **asa** et donc la perte d'un indice n'entraine pas une perte d'information.

- L'opérateur **first** utilisé dans une boucle interne (comme ici **first** i) permet de définir la valeur de rang 0 pour l'indice le plus rapide.

- L'opérateur **next** agira également sur l'indice le plus rapide.

- Nous pouvons résumer les actions des opérateurs de la façon suivante:

Soit X une variable à "n indices" t1, ..., tn notée $X_{t1, ..., tn}$

$(\textbf{first } X)_{t1, ..., tn} = X_{0, t2, ..., tn}$

$(\textbf{next } X)_{t1, ..., tn} = X_{t1+1, t2, ..., tn}$

$(\textbf{latest } X)_{t1, ..., tn} = X_{t2, ..., tn}$

$(\textbf{latest}^{-1} X)_{t1, ..., tn} = X_{0, t1, ..., tn}$ si $X_{t0, t1, ..., tn} = X_{0, t1, ..., tn}$ pour tout $t0 \in \mathbb{N}$

Soit P une variable booléenne de même niveau que X

$(X \textbf{ asa } P)_{t1, ..., tn} = X_{s, t2, ..., tn}$
s'il existe s tel que $P_{r, t2, ..., tn} =$ *false* pour tout $r < s$
et $P_{s, t2, ..., tn} =$ true

Soit Y une variable de même niveau que X

$(X \textbf{ fby } Y)_{0, ..., tn} = X_{0, t2, ..., tn}$

$(X \textbf{ fby } Y)_{t1+1, ..., tn} = X_{t1, t2, ..., tn}$

et enfin, si C est une constante alors

C = **first** C = **next** C = **latest** C = **latest**$^{-1}$ C = C **fby** C

7.4. Structuration des programmes

Tel que nous venons de le décrire, LUCID permet d'écrire tous les programmes itératifs. Cependant, la lourdeur des notations rend vite un programme tant soit peu complexe complètement inextricable. Les concepteurs de LUCID ont donc ajouté au langage quelques constructions qui, sans ajouter à la puissance d'expression, facilitent considérablement l'écriture des programmes. Ces constructeurs s'appellent des *clauses* et sont au nombre de quatre : **compute, mapping, produce, function**.

7.4.1. Compute

C'est le constructeur le plus simple, il correspond au constructeur **begin-end** habituel d'un langage à structure de bloc. Son rôle est double : d'une part, restreindre la portée des identificateurs définis en son sein, et d'autre part, faciliter la définition de boucles imbriquées.

En effet, toute variable définie dans une clause **compute**, étant locale à la clause, l'identificateur pourra être redéfini à l'extérieur de la clause sans aucun risque. Toute variable X utilisée dans la clause, mais définie à l'extérieur, sera implicitement transformée en **latest** X. Le mot clé *output*, qui représente le résultat de la clause, devra obligatoirement être défini par une expression "**asa**". Toute variable extérieure à la clause, et utilisée à l'intérieur, apparaîtra dans une liste précédée du mot clé **using**.

Exemple

Calcul de la racine carrée approchée à 10^{-5} près d'un entier N.

```
compute R using N
    first U = N + 1
    next U = (U + N / U) / 2
    output = U asa ( (N - U * U) / N) < 1.E-5
end
```

Remarques

- Cette clause définit R ; à l'extérieur de la clause, R sera implicitement utilisé sous la forme $\mathbf{latest}^{-1}$ R.

- De même, la variable N apparaissant dans la liste **using** est implicitement utilisée dans la clause sous la forme **latest** N.

7.4.2. Mapping

La clause **mapping** a un effet tout à fait analogue à celui de la clause **compute**, à la différence près que **mapping** admet des paramètres et, en cela, se rapproche de la notion de fonction.

Exemple

```
mapping ROOT (X) using EPS
    first U = X + 1
    next U = (U + N / U) / 2
    output = U asa ( (N - U * U) / N) < EPS
end
```

Les mêmes remarques que pour **compute** sont à faire ici, seule l'utilisation changera : pour définir R on écrira

R = ROOT (X)

7.4.3. Produce

La clause **produce** est similaire à la clause **compute**. Son utilité principale est de réduire la portée des identificateurs définis dans son corps. On peut faire référence à des variables définies à l'extérieur, mais, à la différence de **compute**, il n'y a pas d'opérations **latest** et $\textbf{latest}^{-1}$ implicites. La variable définie par la clause **produce** sera accessible à l'extérieur telle quelle.

Exemple

```
X = input
produce Y using X
    N = 1 fby N + 1
    first T = first X
    next T = T + next X
    result = T / N
end
output = Y
```

Le mot clé *result* permet de lier la variable Y (définie par la clause **produce**) au résultat qui la définie dans la clause. Le programme précédent prend en entrée une suite de valeurs et émet vers la sortie la suite des moyennes de ces valeurs

Les indices dans la clause sont du même niveau que les indices du programme. La variable T, interne à la clause, est définie par "**first** T = **first** X" et non par "**first** T = **latest** X" comme ce serait le cas dans une clause **compute**.

7.4.4. Function

La clause **function** est la généralisation de la clause **produce**. On peut lui donner des paramètres, elle permet de définir des fonctions de suites.

Exemple

```
N = 1 fby N + 1
function AVG (X)
    I = 1 fby I + 1
    first T = first X
    next T = T + next X
    result = T / I
end
Y = AVG (N) × AVG (N × N)
output = Y
```

L'appel AVG (N) crée les suites

I = <1, 2, 3, 4, 5, ...>
T = <1, 3, 6, 10, 15, ...>
result = <1, 3/2, 2, 5/2, 3, ...>

L'appel AVG (N × N) crée les suites

I = <1, 2, 3, 4, 5, ...>
T = <1, 5, 14, 30, 55, ...>
result = <1, 5/2, 14/3, 15/2, 11, ...>

Ainsi AVG (N) × AVG (N × N) vaudra
<1, 15/4, 28/3, 75/2, 33, ...>

Dans une clause **function**, on pourra faire référence à des variables extérieures à la clause, en les faisant apparaître dans une liste précédée du mot clé **using**, (dans les mêmes conditions que pour toutes les clauses).

L'appel à une fonction va provoquer la création d'un ensemble de suites définies par la clause. Il y aura autant de "copies" indépendantes qu'il y a d'occurrences d'appel à une même fonction.

D'un point de vue opérationnel, on peut voir une fonction comme un processus se déroulant en parallèle avec le programme qui l'a invoquée.

7.5. Conclusion

LUCID n'a pas connu une audience très importante auprès des programmeurs. A notre avis cet échec relatif est dû à deux causes essentielles.

- LUCID est conçu pour être un langage général, et l'approche de la programmation en terme de suites n'est vraiment naturelle que dans un domaine restreint de la programmation, notamment les programmes non terminants.

- LUCID est mis en concurrence directe avec les langages fonctionnels dans lesquels l'expression des itérations se réduit à la définition de fonctions récursives. De plus, la sémantique des boucles imbriquées en terme de suites multi-indicées peut paraître un peu lourde.

Néanmoins, le style déclaratif du langage permet de raisonner sur les programmes et d'en tirer des propriétés dans le but de faire des preuves de correction, et ce sans introduire un autre formalisme. C'est un des aspects les plus intéressants de ce langage.

Les critiques que l'on peut adresser a LUCID sont quasiment d'ordre esthétique.

- D'une part, le choix des opérateurs **next** et **fby** rend délicat la compréhension de la causalité, puisque **next** fait référence à l'indice suivant dans la suite, et **fby** fait référence implicitement à l'indice précédent. Ainsi dans l'équation

$$Y = K \textbf{ fby } Z$$

$$Y_n \text{ dépend de } Z_{n-1}$$

et dans l'équation

$$X = K \textbf{ fby next } Z$$

$$X_n \text{ dépend de } Z_n$$

- D'autre part, les clauses définies dans LUCID ont une structure qui nous paraît peu orthodoxe. Certaines constructions ne sont introduites que pour réduire la portée de certains identificateurs (**compute** et **produce**) les autres sont plus proches de la notion de fonction (**mapping** et **function**), mais toutes possèdent la

construction

using (*liste de variables globales*)

qui permet d'hériter des variables extérieures à la clause.

La différence entre les deux groupes (**compute** et **mapping** d'une part et **produce** et **function** d'autre part) ne se situe qu'au niveau de l'interprètation des variables globales. Dans le premier groupe, ces variables sont figées, c'est-à-dire que dans la clause, les références aux variables globales se font de façon implicite à travers l'opération **latest**, et le resultat, qui doit être une constante du niveau de la clause, est vu à l'extérieur implicitement à travers l'opération $\mathbf{latest}^{-1}$.

Dans le deuxième groupe, les variables globales sont utilisées telles quelles, et la clause est une fonction des variables globales (liste **using**) et éventuellement d'autres paramètres (dans le cas de **function**).

Il semble que ces possibilités de structuration d'un programme auraient pu être unifiées dans une seule forme fonctionnelle qui permettrait de définir des suites en fonction de paramètres. On aurait simplement pour le premier groupe des définitions du type

$$Y = \mathbf{latest}^{-1}\ F\ (\ \mathbf{latest}\ X\ )$$

ou pour le deuxième groupe

$$Y = F\ (\ X\ )$$

sans avoir à séparer les variables globales des autres paramètres de la fonction. Cette approche aurait l'avantage de conserver un style déclaratif à la définition des variables quel que soit leur mode de définition. Ce qui n'est pas le cas dans l'utilisation des clauses **produce** et **compute** dont la forme est par exemple

compute X **using** N ...... *output* = ... **end**

Dans ce cas, X est définie implicitement par

$$X = \mathbf{latest}^{-1}\ \mathit{output}$$

Il semble que ces mécanismes soient un peu compliqués à mettre en oeuvre, et peu lisibles car ils engendrent une multiplication des mots clés pas toujours très parlants. On ne saisit pas d'emblée, par exemple, les nuances qu'il y a entre **compute** et **produce** et entre **mapping** et **function**. De plus **compute** et **produce** ont des connotations impératives qui peuvent surprendre dans le contexte d'un langage à

base mathématique sans affectation.

- Enfin (peut-être surtout) la compilation de LUCID a posé des problèmes non triviaux, qui ont d'ailleurs suscité de nombreux travaux. La principale difficulté réside dans la possibilité de décrire la n-ième valeur d'une suite en fonction de valeurs d'indices supérieurs à n (opérateur **next** en partie droite d'une équation). Cette éventualité rend difficile la détection des boucles de causalité, et complique l'ordonnancement des calculs.

8. CONCLUSION

Nous avons essayé de montrer dans ce chapitre, que les langages temps réel de première génération étaient trop près des langages d'écriture de systèmes d'exploitation, dont les objectifs sont différents puisque les contraintes de temps sont généralement assez lâches et surtout non strictes, au sens où l'environnement peut être bloqué dans l'attente de la disponibilité des ressources internes.

Les langages qui reprennent les concepts de CSP pour la communication et la synchronisation dans la conception de systèmes de processus posent deux types de problèmes. D'une part, un problème philosophique, car CSP a été conçu pour rendre la correction d'un programme indépendante des temps d'exécution des processus qui le composent, et ceci est antinomique avec la notion même d'application temps réel au sens où nous l'entendons. D'autre part, l'aspect asynchrone de ces langages rend la programmation très difficile au niveau temporel. D'autant plus qu'il est impossible de spécifier précisément des contraintes de temps, et par conséquent de s'assurer de leur respect ou d'en detecter la violation.

Il semble qu'une approche synchrone soit plus simple à concevoir et permette d'exprimer et de vérifier le respect des contraintes de temps de façon plus satisfaisante. ESTEREL, en ce sens est un langage très satisfaisant. Cependant, il peut paraître dommage que l'architecture cible de ce langage, soit résolument monoprocesseur.

Evidemment, la tendance multi-processeurs intégrés est encore assez peu marquée, mais il est prévisible qu'à moyenne échéance, des matériels de ce type apparaîssent sur le marché. De plus, l'implantation de systèmes distribués, dans le domaine du contrôle de procédés industriel, commence à se développer. Il est donc

nécessaire de développer des outils de programmation succeptibles de répondre à ces besoins nouveaux.

C'est pour apporter un premier élément de réponse à ces problèmes que nous avons étudié le langage LUSTRE dont les objectifs essentiels sont les suivants :

- Faciliter l'écriture de programmes temps réel, en permettant au programmeur de connaître précisemment le comportement temporel de son programme.

- Faciliter l'approche du parallèlisme grâce au caractère déclaratif du langage, qui permet l'expression implicite du parallèlisme. En effet, les contraintes de l'ordonnancement des différents calculs sont déduites d'une simple analyse de dépendance des données.

- Permettre la preuve de programme grâce à une sémantique mathématique simple du langage, tant au niveau temporel qu'au niveau des calculs, en choisissant d'emblée une approche fonctionnelle de la programmation.

CHAPITRE 2
LE LANGAGE LUSTRE
DEFINITION ET PREMIERS EXEMPLES

1. PRESENTATION GENERALE

LUSTRE a été conçu pour être un langage temps réel, qui permette de programmer des applications dans une forme la plus proche possible des spécifications. Il nous a semblé que les principes de base de LUCID pouvaient être repris pour définir un tel langage. Le domaine du temps réel se prête bien à la spécification des applications en terme d'histoires. Si l'on voit une histoire comme une fonction d'un ensemble de dates dans un domaine de valeurs, et si l'on considère l'ensemble des dates comme un ensemble discret, on se ramène au domaine des suites, dans lequel on interprète l'indice d'un élément comme la date d'apparition de cet élément. LUSTRE est un langage temps réel déclaratif, synchrone, fortement typé et structuré.

1.1. langage déclaratif

Un programme LUSTRE est un ensemble d'équations qui permet de définir les sorties du programme en fonction des entrées. Toutes les variables d'un programme représentent des histoires, c'est-à-dire des fonctions du temps dans un domaine de valeurs. Les variables permettent de manipuler des histoires dans leur globalité, ainsi les histoires de sortie seront définies à partir de propriétés invariantes dans le temps portant sur les histoires d'entrée. Une équation spécifie une synonymie complète entre les histoires apparaissant dans ses deux membres. C'est cette caractéristique qui fournit à LUSTRE, comme à LUCID, ses principales

capacités de manipulation formelle de programmes, nous l'appelons *principe de substitition* : si E est une expression et X est une variable définie par l'équation "X = E", alors, partout dans le programme, X peut être substituée à E et inversement.

1.2. langage synchrone

Les histoires sont construites sur un temps discret. Le modèle du temps est une suite ordonnée d'instants. Un programme est cyclique, le n-ième cycle du programme est le cycle correspondant aux n-ièmes termes des suites représentant les histoires des entrées. Si un programme P calcule la sortie S en fonction de l'entrée E,

$$S = P(E)$$

alors pour tout instant n

$$S(n) = P(E(n)) \quad (1)$$

Autrement dit, à l'instant n l'histoire de la sortie en est à son n-ième pas de temps, et son n-ième terme est le résultat de l'application de la fonction définie par le programme P au n-ième terme de l'histoire de son entrée ; le calcul défini par P est instantané.

En fait, le programme P peut, par le biais de variables locales et d'opérateurs temporels, garder la mémoire des entrées passées. Il serait donc plus exact d'écrire l'équation (1) comme suit:

$$S(n) = P(E(n), E(n-1), \ldots, E(1)) \quad (2)$$

Si, dans cette équation, on voit P comme une relation entre histoires, il est clair que, pour pouvoir être implémentée, cette relation doit satisfaire deux propriétés, qui sont à la base des principales différences entre LUSTRE et LUCID :

La causalité : les sorties d'un programme à un instant donné ne peuvent dépendre du futur de ses entrées.

La mémoire bornée : le nombre de termes de l'histoire des entrées nécessaires à l'élaboration de sorties présentes ou futures, est borné.

Nous allons formaliser ces deux notions. Si x est une suite, et si N est un ensemble d'indices, notons x [N] la suite des termes de x d'indices appartenant à N. Soit R une relation entre suites. Pour tout entier n, notons **window** (x, n) l'ensemble des indices de x nécessaires à l'élaboration des y, tels que xRy, jusqu'à l'indice n :

$$x\,[\mathbf{window}\,(x, n)] = y\,[\mathbf{window}\,(y, n)] \Rightarrow \{\, z\,[1..n] \mid xRz \,\} = \{\, z\,[1..n] \mid yRz \,\}$$

Alors

. La relation R est dite *causale* si et seulement si

$$\forall n \in \mathbb{N},\ \forall x,\ \mathbf{window}\,(x, n) \subseteq [1..n]$$

. La relation R est dite *à mémoire bornée* si et seulement si

$$\exists k \in \mathbb{N} \text{ tel que } \forall n \in \mathbb{N},\ \forall x,\ \mathrm{card}\,(\mathbf{window}\,(x, n)) \leq k$$
$$\forall n_1, n_2 \in \mathbb{N},\ \forall x,\ n_1 \leq n_2 \Rightarrow \mathbf{window}\,(x, n_2) \cap [1..n_1] \subseteq \mathbf{window}\,(x, n_1)$$

Ces propriétés ne sont pas satisfaites par certains opérateurs de LUCID, en particulier **next** (en partie droite d'une équation) et l'opérateur **as soon as**.

Le caractère strictement synchrone du comportement d'un programme LUSTRE pourra toutefois être relâché, grâce à des opérateurs spécifiques de désynchronisation.

1.3. langage structuré

LUSTRE est un langage fonctionnel permettant la définition hiérarchisée de fonctions (noeuds). Comme tout programme, un noeud est un transformateur de suites. La hiérarchisation des définitions permet de restreindre la portée des identificateurs de fonction de façon similaire à un langage à structure de blocs pour la définition de procédures. Cette hiérarchisation n'a pour but que de permettre la conception modulaire des programmes. Par contre, les portées des variables ne suivent pas les règles classiques de la structure de bloc : il n'y a pas de variable globale (pas d'effet de bord ni de phénomène de synonymie).

1.4. langage temps réel

Dans un but de sûreté de programmation, nous avons essayé de doter LUSTRE de possibilités de vérification de cohérence sémantique. Par exemple, toutes les variables seront déclarées et typées. Ceci étant, ces vérifications doivent pouvoir être faites soit à la compilation, soit une fois pour toutes durant un prologue de l'exécution des programmes. En effet, pour un langage temps réel, les performances du code produit sont un critère particulièrement important. Ainsi n'y a-t-il pas de pointeurs en LUSTRE, et l'utilisation des tableaux est-elle soumise à des contraintes strictes.

De plus, le temps d'exécution d'un cycle du programme devra être évaluable (approché par valeur supérieure) ; nous avons donc interdit la définition de fonctions récursives. Nous avons introduit des restrictions syntaxiques qui permettent de vérifier que le programme pourra être exécuté avec une taille de mémoire bornée et connue.

La notion de temps de réponse n'apparaît pas dans le langage puisque la sémantique impose que les calculs soient faits en un temps nul. En fait, un programme LUSTRE a un paramètre implicite qui est la durée maximum d'un cycle, c'est-à-dire la durée de l'intervalle de temps séparant deux instants consécutifs du temps discret. On appellera *base de temps du programme* ce paramètre. Toute implantation du programme dans laquelle la durée d'un cycle est inférieure à la base de temps du programme respectera la sémantique abstraite du langage et donc respectera les délais de réponse. Les délais de réponse sont liés au temps de cycle du programme (c'est une notion que l'on retrouve dans les automates programmables industriels). Si la durée du temps de cycle est trop importante par rapport aux délais de réponse dont on a besoin, l'implantation considérée est inacceptable, il faudra changer de programme ou de machine ....

On pourra cependant relâcher cette contrainte si elle paraît trop forte dans certaines applications grâce à un opérateur particulier du langage qui permet de modifier la base de temps du programme.

Cette démarche est typique de l'approche de la programmation temps réel par un langage synchrone. La sémantique abstraite du langage suppose des calculs infiniment rapides et n'est pas implémentable strictement. Une implantation réelle va introduire une dérive temporelle (dont une borne supérieure est connue dans le cas de LUSTRE). Si cette dérive temporelle est acceptable au regard des constantes

de temps de l'application, on considère que l'implantation est fidèle à la sémantique abstraite, sinon l'implantation est inacceptable [Caspi-all 82]. On peut néanmoins concevoir que l'on veuille conserver cette implantation, dans ce cas, le système doit être capable de déceler toute violation de la sémantique abstraite. En LUSTRE ce pourrait être l'arrivée d'une nouvelle valeur tandis que les calculs engendrés par la précédente ne sont pas terminés. Dans le cas où cela a un sens, une alarme pourra être émise.

2. EQUATIONS, VARIABLES ET EXPRESSIONS

Comme en LUCID, une variable X est une suite de valeurs

$$<X_1, X_2, \ldots, X_n, \ldots>$$

prises dans un domaine D (X) défini par le type de X. Tous les domaines contiennent la *valeur indéfinie* notée *nil*. Une constante est une suite constante.

Une équation permet de définir une variable à l'aide d'une expression. Une équation est de la forme

$$X = E$$

où X est une variable et E une expression, son interprétation est la suivante :

$$\text{pour tout n entier} \quad X_n = E_n$$

ou bien d'un point de vue temporel : à tout instant les valeurs de X et de E sont égales.

Une expression est également une suite, construite à partir de variables, de constantes et des opérateurs du langage.

3. LES OPERATEURS DU LANGAGE

On distingue deux types d'opérateurs, les opérateurs instantanés et les opérateurs temporels.

3.1. Les opérateurs instantanés

Il s'agit des opérateurs sur les termes des suites (sans mémoire). On généralise aux suites les opérateurs classiques en les faisant opérer terme à terme sur les suites. Pour tout opérateur n-aire Ω, l'expression Ω (X1, X2, ..., Xn) définit la suite

$$< \Omega (X1_1, ..., Xn_1), \Omega (X1_2, ..., Xn_2), ..., \Omega (X1_i, ..., Xn_i), ... >$$

Par exemple

l'expression "X+ Y" définit la suite :

$$< X_1 + Y_1, X_2 + Y_2, ..., X_i + Y_i, ... >$$

l'expression "**not** (Y)" définit la suite :

$$< \mathbf{not}\ (Y_1), \mathbf{not}\ (Y_2), ..., \mathbf{not}\ (Y_i), ... >$$

l'expression "**if** C **then** X **else** Y" définit la suite :

$$< \mathbf{if}\ C_1\ \mathbf{then}\ X_1\ \mathbf{else}\ Y_1, \mathbf{if}\ C_2\ \mathbf{then}\ X_2\ \mathbf{else}\ Y_2, ..., \mathbf{if}\ C_i\ \mathbf{then}\ X_i\ \mathbf{else}\ Y_i, ... >$$

Remarques

● Tous les opérateurs classiques existent, ils sont stricts par rapport à *nil*. C'est-à-dire que lorsque l'un de leurs opérandes vaut *nil*, alors le résultat vaut *nil*. Seul l'opérateur conditionnel fait exception à cette règle. Son comportement est le suivant :

if *nil* **then** X **else** Y vaut *nil*
if *true* **then** X **else** Y vaut X quel que soit Y
if *false* **then** X **else** Y vaut Y quel que soit X

• Il peut être utile parfois de pouvoir tester si une variable possède la valeur *nil*. Mais l'opérateur de comparaison "=" est strict comme tous les opérateurs du langage. Il existe néanmoins un prédicat noté **isnil** qui permet de réaliser ce test sans rendre la valeur *nil*

si X est la suite < *nil*, *nil*, 1, 2, 3 >
alors **isnil** (X) est la suite < true, true, false, false, false >

3.2. Les opérateurs temporels synchrones

Les opérateurs temporels du langage sont des opérateurs sur les suites. Les deux choses que l'on veut pouvoir exprimer sont : la référence au passé de la suite, l'initialisation d'une suite.

3.2.1. L'opérateur "pre"

On peut voir l'opérateur **pre** comme un opérateur de retard, ou de mémorisation. Il est défini comme suit :

si X $= < X_1, X_2, \ldots\ldots, X_n, \ldots\ldots >$
pre (X) $= < nil, X_1, \ldots\ldots, X_{n-1}, X_n, \ldots\ldots >$

La valeur de **pre** (X) à l'instant n est la valeur de X à l'instant précédent c'est-à-dire à l'instant n-1. Le premier élément de la suite **pre** (X) est *nil*, cette valeur correspond à la valeur qu'avait X avant son initialisation, il est donc logique que ce soit une valeur indéterminée.

3.2.2. L'opérateur "->"

Cet opérateur appelé *"suivi de"* permet d'initialiser une suite. Il est défini comme suit :

si X est la suite $<X_1, X_2, \ldots\ldots, X_n, \ldots\ldots>$
et Y est la suite $<Y_1, Y_2, \ldots\ldots, Y_n, \ldots\ldots>$ l'équation

$$Z = X \to Y$$

définit Z comme la suite $<X_1, Y_2, \ldots\ldots, Y_n, \ldots\ldots>$

On remarque que Z est presque partout égal à Y sauf pour le premier terme où il est égal au premier terme de X.

Remarques

- L'opérateur -> a la propriété suivante :

$$X \to (Y \to Z) = (X \to Y) \to Z = X \to Z$$

Autrement dit, "->" ne permet pas de construire des suites élément par élément. C'est la construction "-> **pre**" qui permet cette opération.

En effet :

$$X = A \to \mathbf{pre}\ (B \to \mathbf{pre}\ (C \to \mathbf{pre}\ (D)))$$

définit Y tel que :

$$Y = <A_1, B_2, C_3, D_4, \ldots\ldots, D_n, \ldots.>$$

- Notons la différence entre notre opérateur -> et l'opérateur **fby** de LUCID. Les équivalences suivantes sont évidentes:

X fby Y (en LUCID) est équivalent à X -> **pre** (Y) (en LUSTRE)
X **fby next** (Y) (en LUCID) est équivalent à X -> Y (en LUSTRE)

Cette différence tient au fait, mentionné plus haut, que l'opérateur next, n'étant pas causal, a dû être remplacé par pre, mais qu'il nous fallait cependant pouvoir exprimer la construction causale **fby next**. De plus, à notre avis, ces opérateurs sont plus naturels lorsqu'on pense en termes d'invariants : par exemple, l'équation LUSTRE "X=0->Y" rend visible que X est presque toujours égal à Y, ce

qui est moins évident de l'équation LUCID "X=0 **fby next** (Y)".

• A l'aide des opérateurs -> et **pre**, on peut définir des suites récurrentes. Par exemple l'équation

X = 1 -> **pre** (X) + 1

définit la suite X_1, X_2, ..., X_n, ... telle que

$$X_n = \begin{cases} 1 \text{ , si } n=1 \\ X_{n-1} + 1 \text{ , si } n>1 \end{cases}$$

c'est-à-dire : X_n = n pour tout entier n

3.3. Premier exemple

Un dispositif ayant deux entrées et une sortie, toutes booléennes, doit réaliser la fonction suivante :

- La sortie est vraie initialement.

- La sortie est vraie chaque fois que les deux entrées sont passées simultanément de faux à vrai, et ce trois fois de suite. C'est-à-dire chaque fois que le nombre de passages simultanés des deux entrées de faux à vrai est un multiple de 3.

Le programme LUSTRE sera de la forme :

```
node EXEMPLE1 (ENTREE1, ENTREE2 : bool)
                          returns (SORTIE : bool);
    var
        -- declarations des variables locales
    let
        -- système d'équations
    tel.
```

Il faut définir la variable SORTIE. Elle est vraie initialement puis devient vraie chaque fois que le compteur modulo 3 d'une certaine condition vaut 0. L'équation définissant SORTIE sera de la forme :

SORTIE = true -> (COMPTEUR = 0)

On a introduit une variable de travail COMPTEUR qui permet de compter modulo 3 les occurences d'une condition. L'équation définissant COMPTEUR sera de la forme :

COMPTEUR = 0 -> if CONDITION then (pre (COMPTEUR) + 1) mod 3 else pre (COMPTEUR);

La variable CONDITION représente le passage simultané de la valeur vrai à la valeur faux des deux entrées. CONDITION est la conjonction de deux autres variables que l'on notera FRONT1 et FRONT2. L'équation définissant CONDITION a donc la forme :

CONDITION = FRONT1 **and** FRONT2

Il ne reste plus qu'à exprimer les deux variables FRONT1 et FRONT2 qui sont directement dépendantes des entrées. Le passage de la valeur faux à la valeur vrai des entrées est exprimé de la façon suivante : l'entrée est à vrai à l'instant n et elle était à faux à l'instant n-1. D'où :

FRONT1 = ENTREE1 **and not pre** (ENTREE1);
FRONT2 = ENTREE2 **and not pre** (ENTREE2);

Le programme est donc le suivant :

```
node EXEMPLE1 (ENTREE1, ENTREE2 : bool)
                        returns (SORTIE : bool);
    var
       -- declarations des variables locales
       COMPTEUR : int;
       CONDITION, FRONT1, FRONT2 : bool
    let
       -- système d'équations
       SORTIE = true -> (COMPTEUR = 0)
       COMPTEUR = 0 -> if CONDITION
                       then (pre (COMPTEUR)+1) mod 3
                       else pre (COMPTEUR);
       CONDITION = FRONT1 and FRONT2
       FRONT1 = ENTREE1 and not pre (ENTREE1);
       FRONT2 = ENTREE2 and not pre (ENTREE2);
    tel.
```

4. STRUCTURATION DES PROGRAMMES

Dans l'exemple précédent, les expressions de FRONT1 et FRONT2 ne diffèrent que par les opérandes. LUSTRE permet de définir des fonctions appelées *noeuds* qui sont des sous-programmes. Dans l'exemple précédent, on peut introduire un noeud "FRONT" défini de la façon suivante :

```
node FRONT (C : bool) returns (H : bool);
    let
        H = C and not pre (C)
    tel;
```

D'aprés cette définition, si COND est une expression booléenne, alors FRONT (COND) est strictement synonyme de "COND **and not pre** (COND). Dans l'exemple précédent, on aurait écrit :

```
FRONT1 = FRONT (ENTREE1);
FRONT2 = FRONT (ENTREE2);
```

4.1. Définitions de noeuds

Un noeud est une fonction. Un programme LUSTRE est un noeud, et il peut contenir d'autres noeuds internes. Les déclarations de noeuds doivent intervenir dans la section des déclarations locales d'un noeud (illustration figure 1).

La définition des noeuds est hiérarchisée, selon la structure de bloc. Pour être suffisamment général, un noeud pourra rendre plusieurs résultats.

4.2. Appels de noeuds

Les appels de noeuds se font dans un style fonctionnel, en considérant qu'une fonction peut rendre un tuple de valeurs dans le cas d'un noeud à plusieurs résultats.

Si N est l'identificateur d'un noeud déclaré avec l'en-tête :

node N (E_1 : τ_1; ...; E_n : τ_n) **returns** (S_1 : θ_1; ...; S_m : θ_m);

Si $E_1, \ldots, E_n$ sont des expressions de types respectifs $\tau_1, \ldots, \tau_n$ alors N ($E_1, \ldots, E_n$) est un m-uplet d'expressions de type ($\theta_1, \ldots, \theta_m$) et l'on pourra écrire

$$(X_1, \ldots, X_m) = N\,(E_1, \ldots, E_n)$$

pour définir les variables $X_1, \ldots, X_m$ de types respectifs $\theta_1, \ldots, \theta_m$.

---

node NOEUD-PRINCIPAL (*liste des paramètres d'entrée*)
return (*liste des paramètres de sortie*);

node NOEUD-INTERNE (*liste des paramètres d'entrée*)
returns (*liste des paramètres de sortie*);
var
Déclarations de variables locales au noeud interne
let
Système d'équations
tel;

var
Déclarations de variables locales au noeud principal
let
Système d'équations
tel.

---

figure 1 : structuration d'un programme.

Exemple

Considérons le noeud F déclaré de la façon suivante :

```
node F (X : int) returns (Y, Z : int) ;
    let
        Y = X + pre (X);
        Z = X - pre (X);
    tel;
```

Un appel à ce noeud pourrait être de la forme

(U, V) = F (T + W);

où U, V, et T sont des variables entières.

On peut voir tout opérateur comme un noeud et ainsi, associer un réseau à chaque expression. Par exemple l'équation suivante

X = 1 -> **pre** (X) + 1; **(1)**

peut être représentée par le réseau suivant

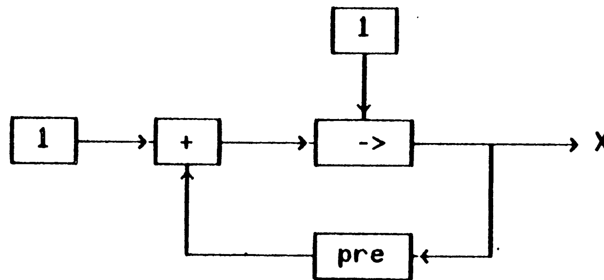


figure 2 : reseau associé à l'équation (1)

De même, on peut associer un réseau au noeud F

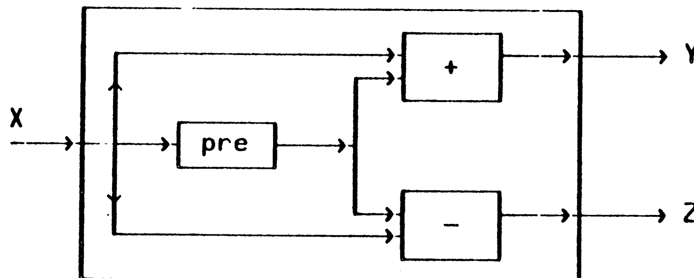


figure 3 : reseau associé au noeud F

On peut donner au langage LUSTRE une interprétation flots de données, mais son caractère synchrone en fait un langage à flots de données particulier : les opérateurs (noeuds du réseau) communiquent sans file d'attente.

5. LES TYPES

5.1. Types simples

LUSTRE possède tous les types simples standards (booléens, entiers, réels, caractères, chaînes, scalaires). Les types construits sont les tableaux (dont nous parlerons plus loin) et les tuples. Ceux-ci sont des types implicites, au sens où l'on ne peut déclarer une variable de type tuple, mais où certains opérateurs (notamment les noeuds à plusieurs sorties) retournent des tuples.

Les opérateurs temporels (pre et ->) sont polymorphes (ils acceptent des opérandes de n'importe quel type, y compris les tuples). Il en est de même des opérateurs conditionnels (en ce qui concerne leurs branches). On a souvent besoin de définir, à l'aide de noeuds, de nouveaux opérateurs temporels, qui sont, logiquement, polymorphes. Plutôt que d'utiliser un mécanisme sophistiqué de polymorphismes, à la manière de ML, nous nous sommes contentés (dans cette première version du langage) d'introduire l'identificateur de type **any**, qui permet de faire référence à n'importe quel type.

Par exemple, soit un noeud PRE3 qui retourne la valeur de son paramètre "vieille de trois cycles" (quel que soit le type de ce paramètre):

```
node PRE3 (X : any) returns (Y : any);
    let
        Y = pre (pre (pre (X)))
    tel;
```

ou encore le noeud DOUBLE-INIT qui permet d'initialiser une suite sur ses deux premiers termes :

```
node DOUBLE-INIT (X1, X2, X3, : any) returns (Y : any);
    var
        X : any;
    let
        X = X2 -> pre (X3);
        Y = X1 -> pre (X);
    tel
```

Une équation de la forme

$$Z = \text{DOUBLE-INIT}\ (1, 2, U)$$

définit la suite $Z = <1, 2, U_3, ..., U_n, ...>$

Une équation de la forme

$$W = \text{DOUBLE-INIT}\ (\text{false}, \text{false}, Z<T)$$

définit la suite $W = <\text{false}, \text{false}, Z_3, ..., Z_n, ...>$

Ces deux équations sont correctes sous condition que Z, U et T soient d'un type numérique et que W soit du type booléen.

Le type polymorphe **any** est un paramètre formel du noeud. Toutes les occurences du type **any** dans une déclaration de noeud feront référence au même type.

Tout noeud polymorphe peut accepter des n-uplets de variables en paramètre, par exemple :

$$(X_1, X_2, X_3) = \text{PRE3}\ (Y_1, Y_2, Y_3)$$

est correct (avec la définition précédente) et est équivalent à :

$$X_1 = \text{PRE3}\ (Y_1);$$
$$X_2 = \text{PRE3}\ (Y_2);$$
$$X_3 = \text{PRE3}\ (Y_3);$$

5.2. Tableaux et structures régulières

5.2.1. Les tableaux et leurs restrictions.

Le type tableau existe en LUSTRE, il permet de définir des *variables tableau.* On ne peut pas référencer un élément de tableau par un indice variable, ce pour éviter la génération de tests dynamiques sur la valeur de la variable.

Un tableau n'est complètement défini que lorsque tous ses éléments sont eux-mêmes définis. Le compilateur devra s'assurer que cette condition est bien remplie.

Un élément de tableau peut être défini soit séparément, comme une variable ordinaire, soit à l'intérieur d'une boucle **for**, qui permet de définir globalement tout ou partie d'un tableau, selon la syntaxe suivante:

for i **in** *intervalle* **let** *système d'équations* **tel**

Par exemple un tableau d'entiers T de dimension 1..N peut être défini par le système d'équations suivant:

```
T[1] = 1;
for i in [2..N]
    let
        T[i] = 2 * T[i-1] + i
    tel
```

i n'est pas une variable, c'est un index. Il n'a pas à être défini ni déclaré. C'est l'instruction **for in let tel** qui tient lieu de définition et de déclaration de i. Sa portée est limitée au système d'équations défini dans le constructeur **for**.

5.2.2. Structures régulières

Le constructeur **for** permet de décrire des "réseaux" de noeuds réguliers. Par exemple le réseau RES de la figure 4, peut être décrit en LUSTRE de la façon suivante :

```
node RES ( X : TX; Y : TY) returns ( X' : TX; Y' : TY);

   node CELL ( X : TX; Y : TY) returns ( X' : TX; Y' : TY);
      let
         X' = f (X, Y); Y' = g (X, Y);
      tel

   var
      AX : array [1..N] of TX;
      AY : array [1..N] of TY;
   let
      AX[0] = X; AY[0] = Y;
      for i in [1..N]
         let
            (AX[i], AY[i]) = CELL (AX[i-1], AY[i-1]);
         tel
      X' = AX[N]; Y' = AY[N];
   tel;
```

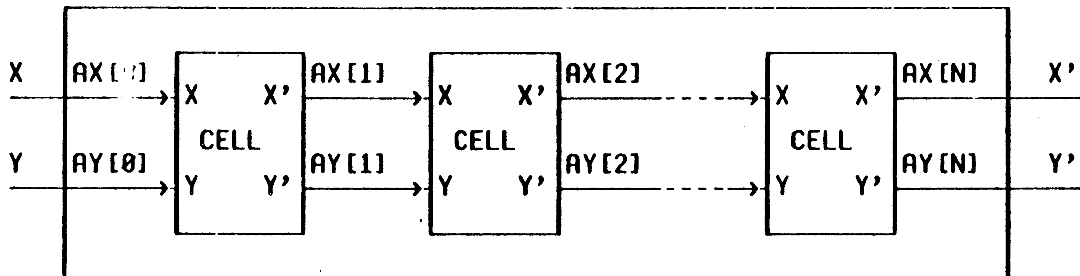


figure 4 : le réseau RES

Ce genre de construction sera très largement utilisé dans le domaine de la programmation d'algorithmes systoliques [Kung 82] dont nous donnerons un exemple dans le chapitre 3.

Le constructeur **for** et la structuration des programmes en noeuds permettront de construire les réseaux les plus complexes par simple jeu de passage de paramètres.

Les restrictions sur l'utilisation des tableaux ont été introduites pour éviter les tests dynamiques (appartenance d'un index à l'intervalle de déclaration, vérification que tout élément du tableau est défini une fois et une seule). Ces restrictions sont compatibles avec l'objectif essentiel de l'introduction des tableaux dans le langage, qui est la représentation de réseaux réguliers de noeuds.

6. EXEMPLES DE PROGRAMMES SYNCHRONES

Avant de poursuivre l'exposé, nous présentons quelques exemples simples de programmes synchrones pour illustrer les possibilités du langage.

6.1. Compteur d'évènement

Nous voulons écrire un programme comptant le nombre d'occurrences d'un évènement arrivées depuis l'occurrence d'un évènement de remise à 0. Les évènements sont vus comme des paramètres booléens dont l'interprétation naturelle est : l'évènement se produit chaque fois que la valeur du paramètre est "true".

Nous écrirons le programme suivant :

```
node COMPTEUR-EVENEMENT (evenement, remise-a-zero : bool)
                    returns (compteur : int);
let
      compteur = 0 -> if remise-a-zero then 0
                    else if evenement then pre (compteur)+1
                    else pre (compteur)
tel;
```

L'appel "COMPTEUR-EVENEMENT (true, false)" définit la suite des entiers. En effet

COMPTEUR-EVENEMENT (true, false) = X

où

X = 0 -> **if** false **then** 0 **else** (**if** true **then** **pre** (X)+1 **else** pre (X))

On peut simplifier cette expression, et l'on obtient finalement

X = 0 -> **pre** (X)+1 .

6.2. Chiens de garde

Un des problèmes classiques de la programmation temps réel est le chien de garde. Nous donnons deux exemples de chien de garde.

Le premier est un programme qui reçoit trois évènements : armer, desarmer, date-limite; il doit émettre une alarme chaque fois qu'il a reçu date-limite alors que le chien de garde était armé. La variable alarme est définie par la conjonction de l'entrée date-limite et d'une condition "le chien de garde est armé". Cette dernière condition est fausse initialement, elle est mise à jour par les entrées armer et desarmer.

```
node CHIEN-DE-GARDE-1 (armer, desarmer, date-limite : bool)
                                returns (alarme : bool);
var est-arme : bool;
let
    alarme = date-limite and est-arme;
    est-arme = false -> if armer then true
                        else if desarmer then false
                        else pre (est-arme)
tel;
```

Le deuxième chien de garde est spécifié comme suit.

Le programme doit envoyer une alarme lorsque le chien de garde reste armé pendant un délai donné, compté en nombre de cycles.

Nous n'avons qu'à appeler le chien de garde précédent avec une condition date-limite qui exprime ce délai. Pour réaliser cette condition, nous allons utiliser un noeud DEPUIS qui compte le nombre de cycles depuis la dernière fois que son entrée booléenne a été vraie :

```
node DEPUIS (evenement : bool) returns (nb-cycles : int);
let
    nb-cycles = 0 -> if evenement then 0
                     else pre (nb-cycles)+1
tel;
```

remarquons au passage que DEPUIS (evenement)
est equivalent à COMPTEUR-EVENEMENT (true, evenement).

Le chien de garde est maintenant :

```
node CHIEN-DE-GARDE-2 (armer, desarmer : bool; const delai : int)
                         returns (alarme : bool);

    node DEPUIS (evenement : bool) returns (nb-cycles : int);
    let
       nb-cycles = 0 -> if evenement then 0
                        else pre (nb-cycles) + 1
    tel;

let
   alarme = CHIEN-DE-GARDE-1 (armer, desarmer,
                        DEPUIS(armer) > delai)
tel;
```

Remarque

Lorsqu'il est défini à l'intérieur du noeud CHIEN-DE-GARDE-2, le noeud DEPUIS est masqué pour tout autre noeud, et l'on peut redéfinir un autre noeud DEPUIS sans risque d'interférences.

7. ASYNCHRONISME

Jusqu'à présent, nous n'avons présenté que l'aspect purement synchrone de LUSTRE : Le comportement d'un programme est cyclique et l'on décrit la valeur de chaque variable à chaque cycle. Dans cette partie, nous allons donner la possibilité de définir des variables évoluant à différentes "vitesses", c'est à dire dont "l'horloge" de renouvellement est une sous-suite du cycle de base du programme. Cette possibilité présente divers avantages :

- Elle permet d'abréger la programmation, en n'indiquant que les changements effectifs de valeur d'une variable.

- Elle permet d'optimiser les programmes, d'une part en indiquant explicitement au compilateur les calculs utiles, et d'autre part en lui laissant un degré de liberté quant au choix d'une politique d'ordonnancement efficace.

- Elle augmente le pouvoir d'abstraction de la notion de noeud : On va pouvoir définir un noeud sans connaître son rythme d'activation. Le temps, dans un programme LUSTRE, est défini localement à un noeud, par le rythme d'arrivée de

ses entrées. Les entrées d'un noeud sont des séquences, et le n-ième terme de la séquence arrive, par définition, à l'instant n du temps du noeud. Ainsi pourra-t-on définir l'horloge de base d'un noeud lors de son appel, simplement en lui fournissant des paramètres convenablement cadencés.

- Surtout, elle facilite la programmation des systèmes asynchrones.

En guise d'introduction, considérons les deux exemples suivants :

- Soit une variable X qui doit être égale à l'expression E chaque fois que la variable booléenne B est vraie, et qui reste stable lorsque B est fausse. A l'aide des opérateurs précédemment décrits, on peut écrire:

Y = **if** B **then** X **else** **pre** (X)

Cependant, il faudra une certaine intelligence au compilateur pour s'apercevoir que la variable X peut n'être mise à jour que lorsque B est vraie. De plus, pour des raisons de modularité, il serait souhaitable de pouvoir ne décrire que les valeurs significatives de X - c'est-à-dire la séquence des valeurs de E lorsque B vaut vrai - indépendemment de la condition de filtrage B (nous parlerons dorénavant d'*horloge*). Ceci sera possible grâce à l'opérateur **when**.

- Considérons maintenant un système qui reçoit une entrée X - par exemple échantillonnée périodiquement par un capteur - et qui doit, entre autres choses, calculer un coefficient de correction C. Supposons que C varie peu, et n'ait à être mis à jour qu'une fois toutes les dix réceptions de X. Une réalisation efficace d'un tel système devrait pouvoir profiter du fait que le calcul de C peut être fait dans n'importe lequel des dix cycles, et même que ce calcul peut être étalé sur plusieurs cycles. Cette liberté d'ordonnancement sera introduite par l'opérateur **every**.

7.1. Echantillonnage synchrone, horloges

7.1.1. L'opérateur "when"

L'expression "EXP **when** COND" définit la suite des valeurs de EXP aux instants où COND (variable booléenne) vaut true. La n-ième valeur de "EXP **when** COND" est la valeur de EXP à l'instant où la valeur de COND est true pour la n-ième fois. La vitesse de "EXP **when** COND" est caractérisée par les

valeurs de la suite définie par COND.

X = EXP **when** COND

définit X telle que

- X n'est plus synchrone avec EXP et COND
- X est caractérisée par la suite des valeurs de EXP et la suite des valeurs de COND

On dira que X *est sur l'horloge* COND

Exemple

Nous illustrons l'effet de l'opérateur "**when**" dans le tableau suivant (ou tt et ff représentent respectivement les valeurs **true** et **false**)

COND	=	tt	ff	tt	tt	ff	tt	ff	ff	tt
EXP	=	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9
EXP **when** COND	=	e_1		e_3	e_4		e_6			e_9

tableau 1 : effet de l'opérateur "when"

Remarques

Dans l'exemple ci-dessus, l'expression "EXP **when** COND" est calculée selon l'horloge COND, ce qui signifie que la seule notion de temps connue pour cette expression est la suite des cycles où COND vaut **true**. Par suite, la question "que vaut EXP **when** COND quand COND a la valeur **false**" n'a pas de sens.

Deux variables peuvent maintenant décrire la même suite do valeurs sans pour autant être égales. Une variable est caractérisée non plus seulement par sa suite de valeurs, mais aussi par son horloge. A toute variable est associée syntaxiquement une horloge.

- Toute variable booléenne est susceptible d'être utilisée comme horloge. La constante **true** sera appelée *horloge de base* du noeud (X **when true** = X). Toute constante est à l'horloge de base du noeud où elle apparaît.

• Il peut y avoir coexistence dans un système d'équations de variables ayant des horloges différentes. Le caractère synchrone du langage LUSTRE interdit de faire agir des opérateurs sur des variables ayant des horloges différentes. En effet le calcul d'une expression comme

A + (B when C)

A, B, C étant sur l'horloge de base, violerait soit la causalité, soit la propriété de mémoire bornée.

• Afin de pouvoir vérifier statiquement que les opérandes d'un opérateur sont sur la même horloge, l'association des horloges aux variables et aux expressions est syntaxique. *Deux variables booléennes égales ne définissent pas la même horloge.*

7.1.2. L'opérateur "current"

Pour pouvoir opérer sur des variables d'horloges différentes, LUSTRE propose un opérateur noté **current** dont l'effet est de "projeter" une expression d'horloge C sur une expression dont l'horloge sera l'horloge de C.

par exemple :

soit COND une variable booléenne sur l'horloge de base du noeud, et EXP une expression également sur l'horloge de base du noeud, alors :

X = EXP **when** COND

définit X sur l'horloge COND

Y = **current** (X)

définit Y sur l'horloge de base tel que

Y = **if isnil** (COND) **then pre** (Y) **else if** COND **then** EXP **else pre** (Y)

Exemple

Nous illustrons dans le tableau suivant l'effet de la combinaison des opérateurs **when** et **current**.

COND	=	tt	ff	tt	tt	ff	tt	ff	ff	tt
EXP	=	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9
EXP **when** COND	=	e_1		e_3	e_4		e_6			e_9
current (EXP **when** COND)	=	e_1	e_1	e_3	e_4	e_4	e_6	e_6	e_6	e_9

tableau 2 : effet de l'opérateur "current"

L'opérateur **current** permet d'opérer sur des variables asynchrones, puisque si X et X' sont des variables d'horloges respectives C et C', et si C et C' sont sur la même horloge, alors

current (X) * **current** (X')

est une expression correcte quel que soit l'opérateur *.

7.1.3. Propriétés

- L'opérateur **when** est distributif par rapport à tous les opérateurs instantanés du langage

  (A + B) **when** C = A **when** C + B **when** C
  (**if** COND **then** A **else** B) **when** C =
  if (COND **when** C) **then** (A **when** C) **else** (B **when** C)
  not (A) **when** C = **not** (A **when** C)

- **when** n'est pas distributif par rapport aux opérateurs temporels.

  (A -> B) **when** C ≠ A **when** C -> B **when** C
  pre (A) **when** C ≠ **pre** (A **when** C)

- L'opérateur **current** est distributif par rapport à tous les opérateurs sur les valeurs du langage

  current (A + B) = **current** (A) + **current** (B)
  current (**if** COND **then** A **else** B) =
  if current (COND) **then current** (A) **else current** (B)
  current (**not** (A)) = **not** (**current** (A))

- **current** n'est pas distributif par rapport aux opérateurs temporels.

  current (A -> B) ≠ current (A) -> B current (B)

  pre (current (A)) ≠ pre (current (A))

- Nous pouvons donner des contre-exemples pour illustrer ces propositions dans les deux tableaux suivants

C	=	tt	ff	tt	tt	ff	tt	ff	ff	tt
EXP	=	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9
pre (EXP)	=	nil	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8
pre (EXP) when C	=	nil		e_2	e_3		e_5			e_8
pre (EXP when C)	=	nil		e_1	e_3		e_4			e_6

tableau 3 : illustration du comportement de l'opérateur "when"

C	=	tt	ff	tt	tt	ff	tt	ff	ff	tt
EXP	=	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9
X = EXP when C	=	e_1		e_3	e_4		e_6			e_9
pre (X)	=	nil		e_1	e_3		e_4			e_6
current (pre (X))	=	nil	nil	e_1	e_3	e_3	e_4	e_4	e_4	e_6
pre (current (X))	=	nil	e_1	e_1	e_3	e_4	e_4	e_6	e_6	e_6

tableau 4 : illustration du comportement de l'opérateur "current"

7.1.4. Echantillonnage et appels de noeuds

Lors d'un appel de noeud, on peut avoir à filtrer les entrées ou les sorties du noeud. Sauf pour des noeuds instantanés, c'est à dire des noeuds dont le système d'équations ne comporte pas d'opérateurs temporels, le filtrage des entrées et le filtrage des sorties donneront des résultats différents.

```
node COMPTEUR (INIT, INCREMENT : int) returns (N : int);
    let
            N = INIT -> pre (N) + INCREMENT
    tel
```

Ce noeud permet de définir une suite d'entiers sur la même horloge que la liste des paramètres qu'il reçoit. L'équation

$$X_1 = \text{COMPTEUR } (0, 1) \textbf{ when } C$$

définit la suite des numéros de cycle où C vaut **true**. Le compteur "tourne" sur le rythme de l'horloge de base et l'on regarde ses sorties sur l'horloge C. Cette construction permet de filtrer les sorties du noeud COMPTEUR.

L'équation

$$X_2 = \text{COMPTEUR } ((0, 1) \textbf{ when } C)$$

définit la suite des entiers au rythme de l'horloge C, puisque les paramètres du noeud sont sur l'horloge C. Nous pouvons illustrer ces différents résultats dans le tableau suivant :

C	=	tt	ff	tt	tt	ff	tt	ff	ff	tt
X_1	=	1		3	4		6			9
X_2	=	1		2	3		4			5
current (X_1)	=	1	1	3	4	4	6	6	6	9
current (X_2)	=	1	1	2	3	3	4	4	4	5

tableau 5 : les différents filtrages.

Remarquons que **current** (X_1) donnera à chaque instant, l'indice (sur l'horloge de C) de la dernière fois où la condition C a été égale à **true**, et **current** (X_2) donnera à chaque instant le nombre de fois que C a été égale à **true** jusqu'à cet instant.

7.2. Echantillonnage asynchrone : l'opérateur "every"

L'opérateur **when** ne permet de définir une horloge que comme une sous-suite du cycle de base du noeud. L'opérateur **every** va nous permettre de modifier aussi la longueur des cycles.

"EXP **every** C" est une expression dont la suite de valeurs est définie de manière indéterministe commme suit :

la n-ième valeur de "EXP **every** C" est l'une des valeurs de EXP présentes entre le n-ième *front montant* de C et le n-ième *front descendant* de C, où les fronts sont définis de la façon suivante :

L'instant i est un front montant de C si et seulement si :

$$C_{i-1} \neq \text{true et } C_i = \text{true}$$

L'instant i est un front descendant de C si et seulement si :

$$C_{i-1} = \text{true et } C_i \neq \text{true}$$

Nous illustrons sur un exemple l'effet de l'opérateur every dans le tableau suivant

C	=	tt	ff	ff	tt	tt	tt	ff	ff	tt
EXP	=	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9
		<	>		<			>		
		1-er cycle			**2-ème cycle**					

tableau 6 : cycles définis par l'opérateur "every"

alors, X = EXP **every** C sera tel que

$X_1 =$ e_1 à l'instant 1 ou 2
ou e_2 à l'instant 2

$X_2 =$ e_4 à l'instant 4, 5, 6 ou 7
ou e_5 à l'instant 5, 6 ou 7
ou e_6 à l'instant 6 ou 7
ou e_7 à l'instant 7

Remarques

- L'opérateur **every** permet d'associer une horloge à une variable, tout comme **when**. Bien sûr, il ne sera pas possible d'opérer directement sur une expression "**every** C" et une expression "**when** C". Car, si la même horloge est associée aux deux expressions, l'association est de nature différente, et l'on n'a aucune garantie du respect de la causalité entre ces expressions. Par contre il est licite d'opérer sur deux expressions "**every** C".

- L'opérateur **every** ne se distribue sur aucun opérateur du langage, ni ne commute avec eux. En effet

(X + Y) **every** C = Z **every** C
où Z = X + Y

(X + Y) **every** C est un échantillonnage indéterministe d'une expression calculée de façon synchrone.

X **every** C + Y **every** C

peut être calculée à partir de termes d'indices différents et donc

X **every** C + Y **every** C ≠ (X + Y) **every** C

- En particulier,

X **every** C + X **every** C ≠ 2*X **every** C

Ceci met en défaut le principe de substitution pour une expression contenant l'opérateur **every**.

Y = X **every** C
Z = Y + Y

Ces deux équations ne peuvent être remplacées par l'équation suivante

Z = X **every** C + X **every** C

L'opérateur "every" ne vérifie pas le principe général de substitution

Le fait que le front descendant de l'horloge C soit inclus dans le cycle de "E **every** C" peut surprendre. En fait cette façon de décrire les cycles présente plusieurs avantages :

- Elle permet de définir très simplement le début et la fin des cycles éventuellement adjacents mais sans recouvrement.

- Du point de vue de l'implantation, elle permet au moniteur de savoir qu'une tâche devient urgente.

Remarque

Nous pouvons maintenant donner le programme correspondant aux spécifications données dans l'introduction des opérateurs asynchrones :

```
node COEF (X: real) returns (C: real);
var n, dix : int; C: bool;
let
   C = F (X every dix);
   dix = (n < 9);
   n = 0 -> (pre (n) + 1) mod 10
tel;
```

7.3. Simulation de l'opérateur "every"

On pourra simuler le comportement de l'opérateur **every** à l'aide de l'opérateur **when** et d'un tirage aléatoire. En effet, dans l'expression EXP **every** C, tout se passe comme si l'on avait EXP **when** C', où C' est une condition qui est vraie une fois et une seule pendant que C est vraie. L'instant où C' est vraie est choisi aléatoirement. Nous donnons ici un système d'équations LUSTRE qui réalise cette simulation.

Supposons que l'on ait à calculer

Y = E every C

Le système d'équations suivant simule cette équation.

C' = **if** A-FAIRE **then if** FRONT (**not** (C)) **then** true **else alea**
  else false
A-FAIRE = **if** FRONT (C) **then** true **else not** (FAIT)
FAIT = false -> **if pre** (C') **then** true
  else if FRONT (**not** (C)) **then** false **else pre** (FAIT)
Y = E **when** C'

alea est une suite booléenne aléatoire, on peut la voir également comme un oracle (entrée supplémentaire du programme). Il est à remarquer également que l'expression **alea** ne vérifie pas le principe de substitution, sa nature est à rapprocher de celle d'un noeud sans paramètre, deux occurrences différentes de l'expression **alea** donneront deux suites aléatoires distinctes.

7.4. Paramètres asynchrones

Dans certaines applications, il peut être utile de décrire des noeuds dont les paramètres ne sont pas tous sur la même horloge. LUSTRE permet de telles définitions, mais impose des contraintes pour rendre possible la vérification statique de la cohérence des horloges.

Soit par exemple un noeud déclaré de la façon suivante :

node N (H, H' : **bool**; V_1 : Tv_1; ....; V_n : Tv_n;
  (X_1 : Tx_1; X_2 : Tx_2; ....; X_n : Tx_n) **every** H';
  (G : **bool**; Y_1 : Ty_1; ....; Y_n : Ty_n) **when** H;
  (Z_1 : Tz_1; Z_2 : Tz_2; ....; Z_n : Tz_n) **when** G);
   returns ....

Les variables booléennes H et H' sont en fait des horloges que l'on passe en paramètre. Elles sont à l'horloge de base du noeud, ainsi que les variables V_i.

Les variables X_i sont des variables échantillonnées de façon indeterministe, leur horloge est "every-H'". Les variables Y_i sont sur l'horloge "when-H".

Enfin les variables Z_i sont sur l'horloge "when-G", où G est une entrée booléenne déjà déclarée.

Cette déclaration permet de mettre en évidence différents groupes de variables qui n'ont pas le même rythme, par simple passage d'une horloge en paramètre, et association statique d'une horloge formelle aux paramètres formels (mots clés **when** et **every**).

Si un noeud définit une sortie qui n'est pas sur l'horloge de base ni sur une horloge définie en entrée, nous imposerons que le noeud émette également le booléen représentant l'horloge de la sortie, afin de pouvoir associer une horloge à tout résultat du noeud. Nous développerons le calcul d'horloge dans le chapitre consacré à la sémantique statique du langage.

8. EXEMPLES ASYNCHRONES

Nous terminons ce chapitre en présentant des versions asynchrones des deux exemples synchrones déjà présentés.

8.1. Compteur d'évènement asynchrone

Si nous voulons activer le compteur seulement lorsqu'il faut modifier sa valeur, c'est-à-dire, à l'instant initial, et sur chaque occurrence de l'évènement "evenement" et de l'évènement "remise-a-zero". On peut définir une horloge "mise-a-jour" par :

mise-a-jour = true -> (evenement **or** remise-a-zero);

On pourra définir le nombre des occurrences de "evenement" par

compteur =
COMPTEUR-EVENEMENT ((evenement, remise-a-zero) **when** mise-a-jour);

Dans ce contexte, on s'apperçoit que la branche "**else pre** (compteur)" n'est jamais selectionnée, puisque l'évènement à compter, une fois filtré, est toujours vrai selon l'horloge mise-a-jour. On peut donc redéfinir un noeud COMPTEUR-EVENEMENT-2 en tenant compte de ces remarques :

```
node COMPTEUR-EVENEMENT-2 (evenement, remise-a-zero : bool)
                returns (compteur : int);
        node COMPTEUR (remise-a-zero : bool)
                        returns (n : int);
        let
                n = 0 -> if remise-a-zero then 0
                        else then pre (n) + 1
        tel;
let
compteur = COMPTEUR ((evenement, remise-a-zero) when mise-a-jour);
mise-a-jour = true -> (evenement or remise-a-zero);
tel;
```

8.2. Chien de garde asynchrone

Dans la deuxième version du chien de garde, le délai était compté en nombre de cycles de base. Supposons maintenant que nous voulions compter ce délai en nombre d'occurrences d'un évènement "evt". Nous allons utiliser le deuxième chien de garde (CHIEN-DE-GARDE-2), en filtrant convenablement ses paramètres.

```
node CHIEN-DE-GARDE-3 (armer, desarmer, evt : bool; const delai : int)
                returns (alarme : bool)
var horloge : bool;
let
   alarme = CHIEN-DE-GARDE-2 ((armer, desarmer, delai) when horloge );
   horloge = armer or desarmer or evt;
tel
```

9. Conclusion

Nous avons défini pour chaque noeud LUSTRE, une horloge de base. mais nous n'avons pas défini ce qu'était l'interface entre un programme LUSTRE et son environnement et notamment, nous n'avons pas précisé ce qu'était l'horloge de base pour le noeud principal. Nous voyons plusieurs possibilités.

- L'horloge de base d'un programme peut être une horloge physique, dans ce cas les calculs sont rithmés par cette horloge, les entrées du programme sont scrutées (échantillonnées) à la fréquence de l'horloge physique (interruptions périodiques).

Ainsi, la précision temporelle avec laquelle le programme perçoit l'environnement est strictement connue et déterminée par la fréquence de l'horloge. Cette horloge est "réglable" pour être ajustée aux caractéristiques de l'application (granularité du temps discret).

- L'horloge de base d'un programme peut être aussi simplement définie par le temps de chaque cycle. Le programme va scruter de lui-même les entrées dès qu'il a terminé un cycle de calcul (la date étant éventuellement elle-même une entrée). C'est une scrutation sans notion d'interruption.

- Enfin l'horloge de base d'un programme peut être définie par les instants où il apparaît une entrée significative (comme en ESTEREL). Les interruptions sont provoquées par l'arrivée d'une entrée.

Il est à remarquer que l'opérateur **every** n'a un sens que dans le cadre d'une implantation réalisée avec des interruptions. Il faut en effet que la notion de période d'inactivité soit présente dans la réalisation pour que l'utilisation du temps libre ait un sens.

Le choix du mécanisme de construction de l'horloge de base, va dépendre fortement de l'application à programmer. Pour une application très réactive, c'est-à-dire une application réagissant à des évènements externes, il sera préférable d'envisager une implantation du troisième type. Pour une application du traitement du signal, on aura tendance à préférer une implantation par échantillonnage du premier type. Enfin pour une application dans laquelle les contraintes de temps sont très larges par rapport au constantes de temps de la machine (temps d'exécution), on pourra se contenter d'une implantation du deuxième type.

CHAPITRE 3
QUELQUES PROGRAMMES LUSTRE

Nous présentons dans ce chapitre quelques exemples de programmes LUSTRE, pris dans différents domaines qui relèvent de la programmation de systèmes temporisés. Ainsi nous présenterons un exemple temps réel classique (compteur d'essieux), un exemple de programmation d'un algorithme systolique (produit de convolution), des exemples de l'automatique industrielle (logique et numérique) et enfin nous utiliserons quelques uns de ces programme comme spécification de circuit, pour illustrer une méthode simple de déduction d'un schéma logique à partir d'une spécification en LUSTRE.

1. DOMAINE TEMPS REEL : un compteur d'èssieux

Une partie d'un système de régulation ferroviaire doit réaliser les fonctions suivantes :

Une voie est divisée en cantons. A la frontière entre deux cantons deux pédales sont disposées de telle sorte qu'elles se recouvrent comme il est indiqué sur la figure suivante :

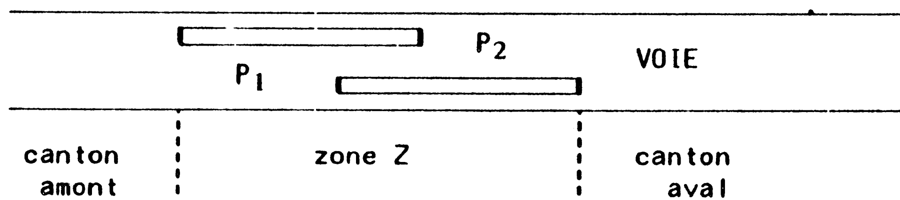


figure 1 : le système de pédales

Ce système de capteurs permet de détecter le passage d'un essieu et le sens de ce passage. Un essieu sera passé entièrement lorsque l'on aura détecté l'enfoncement d'une pédale suivi de l'enfoncement des deux, suivi de l'enfoncement de

l'autre pédale. C'est pour ne pas confondre le demi-tour d'un essieu avec son passage effectif, dans le cas de l'oscillation d'un essieu entre les deux pédales, qu'elles se recouvrent sur la zone Z.

Dans le programme LUSTRE, on représentera les capteurs par des variables booléennes p_1 et p_2 (p_i est vrai lorsque qu'il y a un essieu sur une pedale P_i). Le système doit sortir en permanence les deux variables booléennes s_1 et s_2, et un entier n, tels que :

- La sortie s_1 (respectivement s_2) sera vraie si l'essieu traverse le canton d'amont en aval (respectivement d'aval en amont).
- La sortie n représente, à chaque instant, le nombre d'essieux qui sont passés du canton amont au canton aval, diminué du nombre d'essieux qui sont passés dans le sens inverse.

Des parasites peuvent se produire sur les capteurs qui détectent l'enfoncement des pédales. Ces parasites provoquent le passage à vrai de ces capteurs, mais la durée de maintient de la valeur parasite sera inférieure à 10 millisecondes, alors que la présence effective d'un essieu sur une pédale provoque la mise à vrai du capteur pendant une durée supérieure à 20 millisecondes. On pourra donc se protéger contre ces parasites par un simple filtrage des entrées

Nous allons tout d'abord décrire le programme sans tenir compte des parasites. Ce programme reçoit en entrée les variables p_1 et p_2 et émet les sorties s_1, s_2 et n. Les sorties s_i peuvent être décrites simplement : s_i est vraie si la variable p_i passe de vrai à faux et que la variable p_j est vraie (pour $i=1, 2$ et $j=3-i$).

Nous avons donc à écrire la condition "p_i passe de faux à vrai" ($i=1, 2$). Nous définissons pour cela un noeud appelé FRONT.

```
node FRONT (cond : bool) returns (front : bool);
let
    front = false -> cond and not pre (cond)
tel;
```

pour définir n il faut savoir d'où vient l'essieu qui entre, on a donc besoin d'une variable locale pour conserver cette information.

```
node ESSIEUX (p1, p2 : bool) returns (s1, s2 : bool; n : int);
var dernier-entre : {amont, aval};
let
   s1 = FRONT (not p1) and not p2;
   s2 = FRONT (not p2) and not p1;
   n = 0 -> if isnil (dernier-entre) then pre (n)
            else if dernier-entre=amont and s2 then pre (n)+1
            else if dernier-entre=aval and s1 then pre (n)-1
            else pre (n);
   dernier-entre = if FRONT (p1) and not p2 then amont
             else if FRONT (p2) and not p1 then aval
             else pre (dernier-entre);
tel;
```

pour éliminer les parasites, nous appèlerons ce noeud après avoir convenablement filtré ses entrées. Nous utilisons pour cela un noeud noté FILTRE défini par :

```
node FILTRE (base, p : bool; const delai : int) returns (q : bool);

   node DEPUIS (evenement : bool) returns (nb-cycles : int);
   let
         nb-cycles = 0 -> if evenement then 0 else pre (nb-cycles)+1
   tel;

let
   q = DEPUIS ((not p) when base) >= delai
tel;
```

Le paramètre base représente la base de temps sur laquelle on veut filtrer le paramètre booléen p.

En supposant qu'il existe une entrée donnant la milliseconde, (dont la précision est liée à l'implantation), et moyennant les définitions précédentes, le programme principal est le suivant :

```
node COMPTEUR-D-ESSIEUX (milliseconde, p1, p2 : bool)
                    returns (s1, s2 : bool; n : int);

var p1-filtre, p2-filtre : bool;
let
       p1-filtre = FILTRE (milliseconde, p1, 10);
       p2-filtre = FILTRE (milliseconde, p2, 10);
       (s1, s2, n) = ESSIEUX (p1-filtre, p2-filtre)
tel;
```

2. ALGORITHME SYSTOLIQUE

A partir du produit de convolution, nous reprenons la démarche décrite dans [Quinton 83], appliquée en LUSTRE, pour arriver à un programme systolique.

A chaque étape de transformation du sytème d'équations, on associe un système d'équations LUSTRE ou pseudo-LUSTRE, pour obtenir en dernier lieu un programme LUSTRE.

- Ce programme permettra de simuler l'algorithme du produit de convolution.

- Il sera une spécification du circuit systolique à construire.

Dans [Quinton 83], la démarche comporte trois phases :

- réécriture des équations du problème sous forme d'un système d'équations récurrentes uniformes,

- définition de la fonction temporelle spécifiant le cadencement des calculs en fonction des vitesses de propagation des données,

- définition d'architectures systoliques par application de fonctions d'allocation des calculs aux processeurs.

L'équation de départ est la suivante :

$$y(i) = \sum_{k=0}^{K} w(k)\, x(i-k) \qquad (1)$$

ou bien en LUSTRE :

$$y = \sum_{k=0}^{K} pre^{k}(x) * w(k)$$

x et y sont des suites infinies, l'indice i en LUSTRE est donc sous-entendu.

première étape

Elle consiste à transformer le système d'équations de manière à mettre en évidence

- les calculs élémentaires
- le flux des données.

L'équation (1) peut être mise sous la forme d'un système d'équations récurrentes uniformes

$$\begin{aligned} &Y(i, k) = y(i) \\ &Y(i, k) = Y(i, k-1) + W(i, k) * X(i-1, k-1) \\ &W(i, k) = W(i-1, k) = w(k) \\ &X(i, k) = X(i-1, k-1) \\ &X(i, -1) = x(i) \end{aligned} \qquad (2)$$

ce système correspond aux équations suivantes en LUSTRE

```
X [-1] = x;
Y [-1] = 0;
y = Y [K];
for k in [0..K]
let
        Y [k] = Y [k-1] + W [k] * pre (X [k-1]);
        X [k] = pre (X [k-1]);
tel;
```

Ce système d'équations n'est pas systolique, puisque Y [k] dépend de Y [k-1]

deuxième étape

Il faut faire le choix d'une fonction de cadencement, pour rendre le système systolique, on fait le changement de variable suivant

$X'(i, k) = X(i-k, k)$
$Y'(i, k) = Y(i-k, k)$

soit en LUSTRE

X' [k] = **pre**^{k} (X [k]);
Y' [k] = **pre**^{k} (Y [k]);

Le système d'équations LUSTRE devient

```
X' [-1] = x;
Y' [-1] = 0;
y = Y' [K];
for k in [0..K]
let
        Y' [k] = pre (Y' [k-1]) + W [k] * pre (pre (X' [k-1]));
        X' [k] = pre (pre (X' [k-1]));
tel;
```

troisième étape

On fait choix d'une fonction d'allocation des calculs aux processeurs, ce qui revient à déterminer la structure d'une cellule de base. La fonction d'allocation a(i, k) = k définit une structure ayant K cellules identiques connectées de la façon suivante :

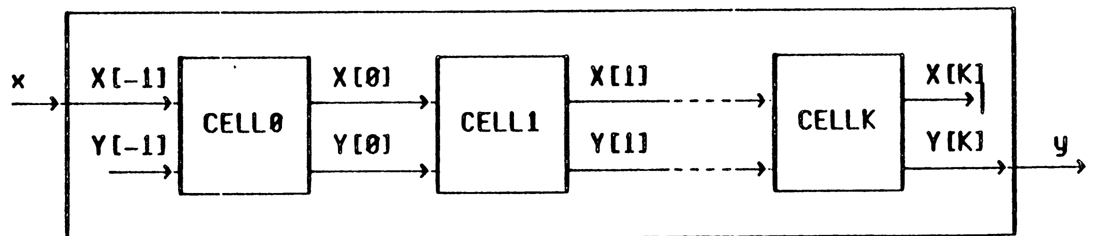


figure 2 : le réseau systolique

Avec CELL définie en LUSTRE par :

```
node CELL (const Wk : real; Xc, Yc : real) returns (Xs, Ys : real);
let
        Ys = pre (Yc) + Wk * pre (pre (Xc));
        Xs = pre (pre (Xc));
tel;
```

Ce réseau correspond au programme LUSTRE final

```
node CONVOLUTION (x : real) returns y : real;
   const W : array [0..K] of real;
   var X, Y : array [-1..K] of real;
   let
        X [-1] = x;
        Y [-1] = 0;
        y = Y [K];
        for k in [0..K]
                let
                (X [k], Y [k]) = CELL (W [k], X [k-1], Y [k-1]);
                tel;
   tel;
```

3. AUTOMATIQUE

3.1. Automatique logique

Dans ce domaine, les langages à relais des automaticiens font place peu à peu au GRAFCET. Les équations, et le parallélisme inhérent aux constructions logiques, ont été remplacés par une approche séquentielle et impérative de la programmation avec l'apparition des microprocesseurs. LUSTRE permet par son style déclaratif d'avoir une démarche qui se rapproche de la pensée des automaticiens.

L'exemple suivant illustre l'utilisation du langage LUSTRE dans ce domaine. Nous donnons également le GRAFCET déduit des spécifications pour montrer la distance qui apparaît entre les spécifications initiales, et le résultat obtenu dans un langage impératif. Il est à remarquer, de plus, que pour obtenir le programme LUSTRE correspondant, il a suffit de formaliser les spécifications sous forme d'équations.

Un dispositif de sécurité est greffé sur l'interface de commande d'un procédé afin d'inhiber toute commande sur cette interface en cas de détection d'anomalie. La présence d'une anomalie est signalée par le procédé au moyen d'un signal AL.

Ce dispositif a deux modes de fonctionnement : CONTROLE et SERVICE. En mode CONTROLE, la détection du signal AL a pour effet, outre l'inhibition des commandes, d'allumer un voyant V ; ce voyant restera allumé jusqu'à la disparition de AL. En mode SERVICE, la détection du signal AL active une sirène interne et fait clignoter le voyant V à la fréquence d'un signal H ; ceci jusqu'à la disparition de AL.

A la mise sous tension, ce dispositif est en mode CONTROLE. Une touche SELECT (interrupteur à une position) permet de commander les changements de mode : chaque nouvelle action sur cette touche fait changer de mode.

Un changement de mode peut être effectué à tout moment. Toutefois, en présence du signal AL, le comportement du dispositif n'est pas modifié. Il reste déterminé par le mode actif à l'instant de détection du signal AL. Le mode de fonctionnement est visualisé par une lampe témoin rouge TR, allumée en mode service.

Le GRAFCET correspondant tiré de [Moalla 81] est présenté figure 3

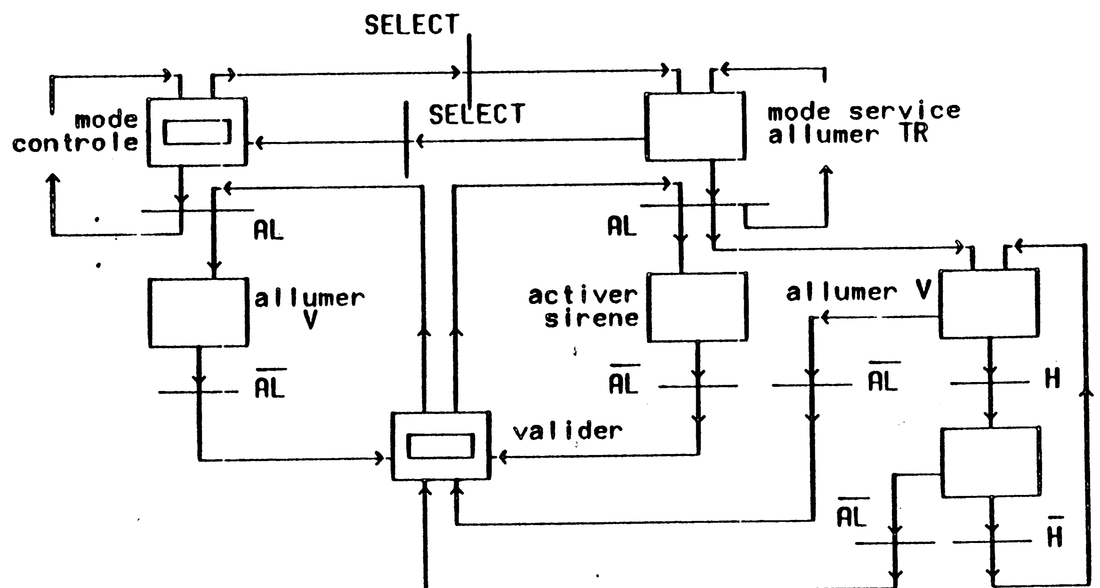


figure 3.: GRAFCET du dispositif de contrôle

Nous donnons le programme LUSTRE correspondant :

```
node SUPERVISEUR (AL, H, SELECT : bool)
                returns (SIRENE, V, TR, VALID : bool);
var CONTROLE, SERVICE : bool;
let
        CONTROLE = true -> if (SELECT and not (AL))
                    then not pre (CONTROLE)
                    else pre (CONTROLE);
        SERVICE = not (CONTROLE);
        TR = SERVICE;
        VALID = not (AL);
        SIRENE = SERVICE and AL;
        V = AL and (CONTROLE or (SERVICE and H));
tel
```

On remarque dans cet exemple très simple, combien la possibilité de programmer en invariant simplifie l'expression des programmes. Le caractère impératif du GRAFCET rend fastidieuses des compositions simples d'expressions booléennes. Il suffit de regarder dans la figure 3, le nombre des transitions étiquetées par AL ou son complémentaire pour s'en convaincre.

3.2. Automatique numérique

Depuis toujours les automaticiens raisonnent en termes de suite et de "transformée en z". Il y a peu de langages dans ce domaine, qui permettent d'exprimer un programme simplement à partir des spécifications qui sont généralement données sous forme d'un système d'équations faisant intervenir la transformée en z. Il se trouve que ce genre de spécifications sont très simples à exprimer en LUSTRE et sous une forme très peu modifiée.

La commande automatique de systèmes continus par calculateur digital (systèmes de commande numérique échantillonnée) fait grand usage, pour représenter à la fois les signaux et les opérateurs sur ces signaux, de la transformée en z. Celle-ci consiste à associer à un signal numérique échantillonné (c'est-à-dire à une suite de valeurs $s = (s_0, s_1, \ldots, s_n, \ldots)$ sa transformée en z, fonction complexe de variable complexe définie par :

$$S(z) = \sum_{n \geq 0} \frac{s_n}{z^n}$$

ou plus simplement

$$S(z) = \sum \frac{s_n}{z^n}$$

en adoptant la convention $s_n = 0$, pour $n < 0$

Un filtre à une entrée et une sortie (ou opérateur linéaire stationnaire de dimension d'état finie) correspond à l'équation récurrente

$$s_n = \sum_{i=0,k} b_i\, e_{n-i} - \sum_{i=1,k} a_i\, s_{n-i}$$

Il est facile de voir que la transformée de la suite $(\ldots, s_{n-i}, \ldots)$ est

$$\frac{S(z)}{z^i}$$

de sorte que la transformée en z de l'équation précédente peut s'écrire

$$S(z) = \sum_{i=0,k} b_i \frac{E(z)}{z^i} - \sum_{i=1,k} a_i \frac{S(z)}{z^i}$$

ou encore

$$S(z) = \frac{\sum_{i=0,k} \frac{b_i}{z^i}}{1 + \sum_{i=1,k} \frac{a_i}{z^i}} E(z)$$

et finalement

$$S(z) = \frac{\sum_{i=0,k} b_i\, z^{k-i}}{z^k + \sum_{i=1,k} a_i\, z^{k-i}} E(z)$$

Le terme

$$H(z) = \frac{\sum_{i=0,k} b_i\, z^{k-i}}{z^k + \sum_{i=1,k} a_i\, z^{k-i}}$$

s'appelle la fonction de transfert échantillonnée du filtre. Un telle représentation est agréable pour l'étude des propriétés mathématiques de ces filtres. En outre, il est fréquent que l'automaticien décrive l'algorithme de commande sous forme d'un

schéma d'interconnexion de filtres représentés par leur fonctions de transfert (blocs diagrammes).

Par exemple, une commande PID (proportionnelle, intégrale, dérivée) pourra être représentée par le schéma :

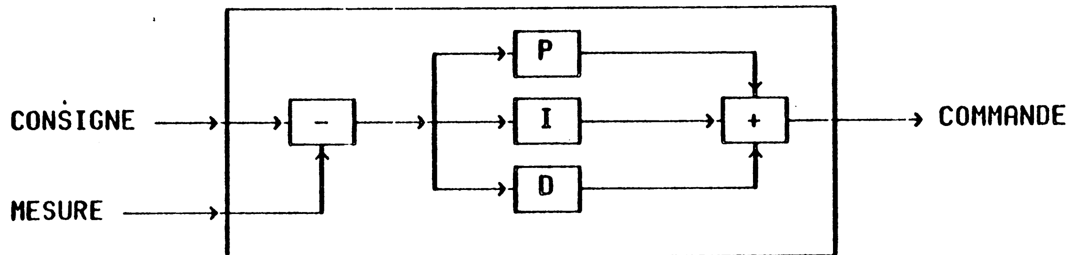


figure 4 : le schéma d'un PID

La traduction de tels schémas en programmes LUSTRE est très simple. En effet, chaque bloc se traduit par un noeud LUSTRE qui est obtenu immédiatement en remarquant que l'équivalent LUSTRE de

$$\frac{S(z)}{z}$$

est

$$0 \text{ -> pre } (s)$$

A la fonction de transfert H(z) correspond le noeud LUSTRE :

```
node FILTRE (e : real) returns s : real;
const K : int;
        a, b : array [0..K] of real;
var x : array [1..K] of real;
let
        x [K] = b [K] * e - a [K] * s;
        s = b [0] * e + (0 -> pre ( x [1]));
        for k in [0..K-1]
        let
                x [k] = b [k] * e - a [k] * s + (0 -> pre (x [k+1]));
        tel;
tel;
```

Le schéma correspondant est décrit par la figure 5.

La traduction de ce schéma dans le noeud LUSTRE est pratiquement de nature syntaxique et pourrait être générée ou vérifiée automatiquement.

Une fois obtenus les noeuds correspondants à chaque bloc, on obtient directement à partir du schéma d'interconnexion la structure d'appel de ces noeuds.

Ces considérations illustrant bien, à notre avis, la grande adéquation que LUSTRE présente vis à vis d'outils courants de l'automatique comme la transformée en z. Mais cette adéquation ne s'arrête pas là, car il n'y a pas plus de difficultés pour programmer en LUSTRE des systèmes stationnaires (coefficients a_i, b_i non constants) où non linéaires ($\sqrt{s}$, ou si c alors s sinon s').

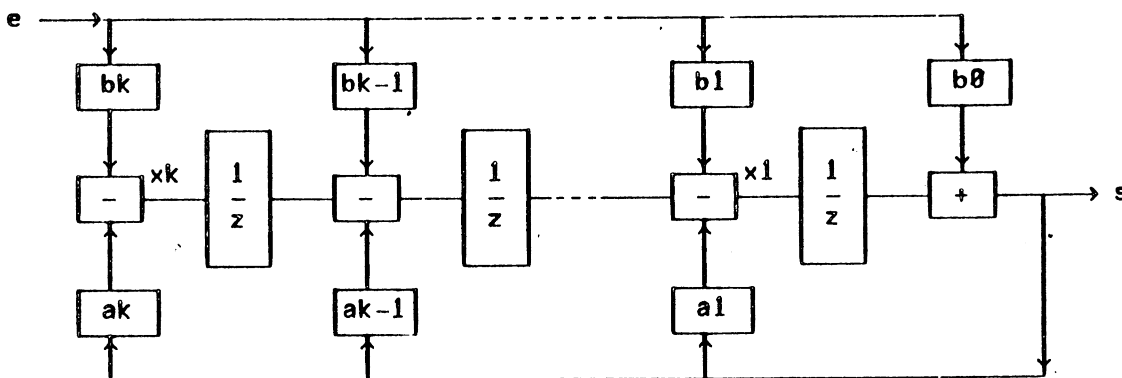


figure 5 : le diagramme du filtre

4. SPECIFICATION DE CIRCUIT

Un circuit est un cas particulier de système temporisé. On peut étudier ses propriétés à partir d'un modèle discret du temps. C'est ce que nous faisons en LUSTRE, d'autant que les liens qu'il existe entre le comportement d'un système matériel et un réseau à flots de données sont très étroits. LUSTRE semble donc se prêter à la spécification, à la simulation et à la conception de circuits.

Nous terminerons ce chapitre d'exemples par la production d'un circuit simple à partir d'un programme LUSTRE. Plutôt que de donner de nouvelles spécifications, nous prenons un programme LUSTRE déjà écrit.

Conception d'une cellule du réseau systolique

Comme dans [Quinton 83], nous terminons par le circuit qui permet le calcul élémentaire dans une cellule.

```
node CELL (const Wk : real; Xe, Ye : real) returns (Xs, Ys : real);
let
        Ys = pre (Ye) + Wk * pre (pre (Xe));
        Xs = pre (pre (Xe));
tel;
```

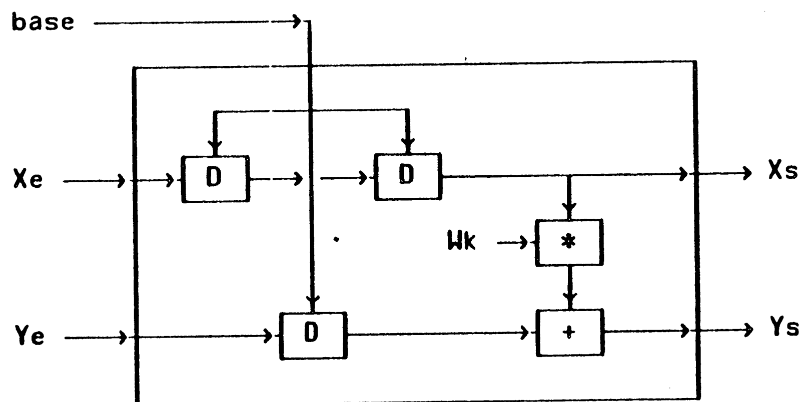


figure 6 : circuit associé à une cellule

Où D représente symboliquement une bascule D, * représente un mutiplieur, et + un additionneur.

Il existe, pour tout programme LUSTRE, une traduction systématique triviale qui permet de passer à un circuit. Cette voie n'est pas encore très explorée, mais nous semble très prometteuse. Un DEA sur ce sujet et en cours, il devra nous permettre (entre autre) d'étudier les relations qu'il existe entre les transformations d'un programme LUSTRE (que nous présentons dans le chapitre 5) et l'effet induit sur le circuit. A titre d'exemple citons simplement la transformation suivante :

Y = **if** C **then** X **else pre** (Y);
devient
Y = **current** (X **when** C);

Cette équivalence est démontrée au chapitre 5.

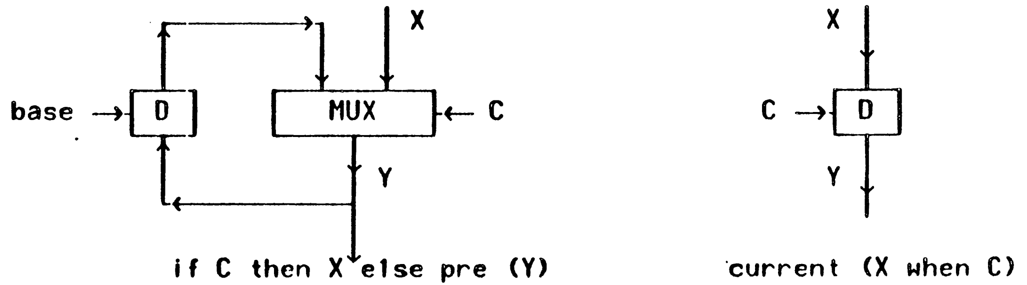


figure 7 : transformation de circuit

Où MUX représente symboliquement un multiplexeur à deux entrées.

Les exemples présentés dans ce chapitre, illustrent l'utilisation du langage, et permettent de mettre en évidence les différents domaines que l'on peut aborder tant au niveau de la programmation (applications temps réel classiques, automatismes industriels) qu'au niveau de la spécification ou de la simulation (production de circuits systoliques, conception de circuits). D'une manière générale, nous pensons que LUSTRE peut être utilisé dans la plupart des applications où apparaît une notion de temps.

CHAPITRE 4
SEMANTIQUE DU LANGAGE

Nous présentons dans ce chapitre les sémantiques dénotationnelles statique et dynamique d'un sous-ensemble du langage représentatif des principales caractéritiques de LUSTRE. Ce chapitre est écrit en deux parties. La première est consacrée à la sémantique dynamique, et la seconde à la sémantique statique. Cet ordre peut surprendre le lecteur, nous l'avons choisi pour des raisons de lisibilité.

Nous avons écarté volontairement l'opérateur **every** du langage, sa sémantique peut être obtenue par simulation grâce l'opérateur **when** et à un oracle.

1. SEMANTIQUE DYNAMIQUE

Nous présentons la sémantique dynamique de LUSTRE sur des programmes bien formés vis à vis de la sémantique statique. Nous partons d'un sous-ensemble réduit du langage que nous enrichissons au fur et à mesure de l'exposé pour arriver à un sous-ensemble représentatif du langage.

1.1. Le langage de base L0

Nous définissons tout d'abord les différents domaines dont nous avons besoin pour présenter la sémantique dynamique du langage L0.

1.2. Domaines syntaxiques

Nous définissons les domaines syntaxiques de base qui permettront de construire les expressions du langage.

. *ID* : domaine des identificateurs du langage (id ∈ *ID*)
. *INT* : domaine des dénotations des entiers du langage (i ∈ *INT*)
. *BOOL* : domaine des dénotations des booléens du langage (l ∈ *BOOL*)
. *VAL* = *INT* ∪ *BOOL* (k ∈ *VAL*)
. *OP* : domaine des opérateurs du langage (op ∈ *OP*) *OP* = *UOP* ∪ *BOP* où
. *UOP* : domaine des opérateurs unaires du langage (uop ∈ *UOP*)
. *BOP* : domaine des opérateurs binaires du langage (bop ∈ *BOP*)

1.2.1. Domaine des expressions

On note EXP_0 le domaine syntaxique des expressions (e ∈ EXP_0) défini par la syntaxe abstraite suivante

$$e ::= k \mid id \mid e_1 \text{ bop } e_2 \mid \text{uop } e \mid \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \mid \textbf{pre } (e_1) \mid e_1 \text{ -> } e_2 \mid e_1, \ldots, e_n$$

Remarque

Cette syntaxe abstraite définit des expressions synchrones (**when, current** et **every** n'apparaissent pas), construites à partir de constantes et d'identificateurs simples (pas de tableaux ni d'enregistrements) et sans appels de noeuds.

1.2.2. Domaine des équations

On note *EQU* le domaine syntaxique des équations (q ∈ *EQU*) défini par la syntaxe abstraite suivante

$$q ::= (id_1, \ldots, id_n) = e$$

1.2.3. Domaine des systèmes d'équations

On note *SEQ* le domaine syntaxique des systèmes d'équations (Sq ∈ *SEQ*) défini par la syntaxe abstraite suivante

$$Sq ::= \textbf{let } q_1; \ldots; q_n \textbf{ tel}$$

I.2.4. Domaine des programmes

On note *PRG* le domaine syntaxique des programmes (prog $\in$ *PRG*) défini par la syntaxe abstraite suivante

prog ::= **in** id_1; ..., id_n ; **out** id_{n+1}; ..., id_m ; **local** id_{m+1}; ..., id_k ; Sq

I.3. Domaines sémantiques

Nous définissons les domaines sémantiques de base qui permettront de construire les domaines sémantiques dérivés.

. **Z** : {nil} $\cup$ $\mathbb{Z}$ (entiers relatifs)
. **B** : {nil} $\cup$ {tt, ff} (booléens)
. **V** = **Z** $\cup$ **B**
. **UOP** : domaine des opérateurs unaires sur les suites
. **BOP** : domaine des opérateurs binaires sur les suites
. **OP** = **UOP** $\cup$ **BOP**

I.3.1. Domaines dérivés

- $\mathbf{V}^{\infty}$ = $\mathbf{V}^* \cup \mathbf{V}^{\omega}$
  est l'ensemble des suites finies ou infinies d'éléments de **V**
  avec $\mathbf{V}^* = \emptyset + \mathbf{V} \times \mathbf{V}^*$ et $\mathbf{V}^{\omega} = \mathbf{V} \times \mathbf{V}^{\omega}$

- S = $\{ \sigma \in \mathbf{V}^{\infty} \mid \sigma (0) = \text{nil} \}$

σ (0) désigne le premier élément de la suite σ, *S* est donc un sous ensemble de $\mathbf{V}^{\infty}$ qui ne contient pas la suite vide.

Pour tout $\sigma \in \mathbf{S}$, lg (σ) est la longueür de la suite σ et σ (i) est le i-ème terme de σ (i < lg (σ)) et l'on définit sur **S** la relation d'ordre suivante :

$$\sigma \preccurlyeq \sigma' \Leftrightarrow \lg(\sigma) \leq \lg(\sigma')$$
et
$$\text{pour tout } i \in [0, \lg(\sigma)-1] \; \sigma(i) = \sigma'(i)$$

Cette relation d'ordre fait de **S** un ensemble ordonné inductif (cpo) dont les éléments maximaux sont les suites infinies.

L'élément minimum de **S** est la suite de longueur 1 ($\perp$ = <nil>).

On confondra toujours dans la suite de l'exposé les dénotations des éléments de **V** et les dénotations des suites de **S** constantes. On notera

k la suite <nil, k, k, ... , k, ... >
nil la suite <nil, nil, ... , nil, ... >

- **EV** est l'ensemble des environnements défini par

$$\mathbf{EV} = ID \rightarrow \mathbf{S}$$

EV est un cpo dont la relation d'ordre est induite par la relation d'ordre sur **S**

Soient ε_1 et ε_2 deux éléments de **EV**

$$\varepsilon_1 \preccurlyeq \varepsilon_2 \Leftrightarrow \varepsilon_1(id) \preccurlyeq \varepsilon_2(id) \text{ pour tout id} \in ID$$

1.3.2. Fonctions sémantiques

On définit une fonction sémantique de chaque domaine syntaxique dans un domaine sémantique, nous utiliserons les règles d'écriture suivantes:

DOM	:	domaine syntaxique
DOM	:	domaine sémantique
Dom	:	fonction sémantique associée $\in$ *DOM* $\rightarrow$ **DOM**

1.4. Sémantique des expressions

Le domaine sémantique d'une expression **EXP** est défini par

$$\mathbf{EXP} = \mathbf{EV} \rightarrow \mathbf{S}^*$$

La sémantique d'une expression est une fonction des environnements dans les séquences.

Notations

Dans tout ce qui suit, nous adopterons les notations suivantes :

k dénote indifféremment un élément de *VAL*, de **V** ou de **S**
ε dénote un élément de **EV**
c, c_1, c_2 dénotent des éléments de EXP_0
n dénote un élément de $\mathbb{N}$
id dénote un élément de *ID*

Soit op un opérateur du langage $\in$ { +, -, $\leq$, **and**, **not**, etc... } et soit *op* l'opérateur mathématique associé. On peut distinguer de plus

- l'opérateur correspondant sur les éléments des suites défini par

$$Op(v_1, \ldots, v_n) = op(v_1, \ldots, v_n)$$

si $v_i \neq$ nil pour tout i = 1, 2, ..., n
nil sinon
puisque les opérateurs sur **V** sont tous stricts.

- l'opérateur correspondant sur les suites défini par

$$\mathbf{op}\ (\sigma_1, \ldots, \sigma_n) = \lambda\ n.\ Op\ (\sigma_1\ (n), \ldots, \sigma_n\ (n))$$

On confondra toutes ces notations en op, le contexte d'apparition permettra de déterminer sans ambiguïté quelle est la nature de l'opérateur rencontré.

La sémantique d'une expression est une suite, le langage synchrone ne permet de définir que des suites infinies que l'on peut voir comme des fonctions totales de $\mathbb{N}$ dans **V**. Néanmoins, nous traiterons les suites dans le cas général des suites finies ou infinies (les fonctions de $\mathbb{N}$ dans **V** sont éventuellement partielles) : le domaine de définition étant caractérisé par la longueur de la suite. Le langage asynchrone fera apparaître des suites finies ; nous définissons à ce niveau les longueurs des suites,

pour ne pas avoir à réécrire la sémantique des expressions synchrones lorsque nous aborderons le langage asynchrone.

Constantes

$$\mathrm{Exp}_0\ (k)\ (\varepsilon) = \lambda n.\ k$$
$$\mathrm{lg}\ (\mathrm{Exp}_0\ (k)\ (\varepsilon)) = \infty$$

Identificateurs de variables

$$\mathrm{Exp}_0\ (id)\ (\varepsilon) = \varepsilon\ (id)$$
$$\mathrm{lg}\ (\mathrm{Exp}_0\ (id)\ (\varepsilon)) = \mathrm{lg}\ (\varepsilon\ (id))$$

Opérations binaires

$$\mathrm{Exp}_0\ (c_1\ \mathrm{bop}\ c_2)\ (\varepsilon) = \mathrm{Exp}_0\ (c_1)\ (\varepsilon)\ \mathrm{bop}\ \mathrm{Exp}_0\ (c_2)\ (\varepsilon)$$
$$\mathrm{lg}\ (\mathrm{Exp}_0\ (c_1\ \mathrm{bop}\ c_2)\ (\varepsilon)) = \inf\ (\mathrm{lg}\ (\mathrm{Exp}_0\ (c_1)\ (\varepsilon)),\ \mathrm{lg}\ (\mathrm{Exp}_0\ (c_2)\ (\varepsilon)))$$

Opérations unaires

$$\mathrm{Exp}_0\ (\mathrm{uop}\ (c))\ (\varepsilon) = \mathrm{uop}\ (\mathrm{Exp}_0\ (c)\ (\varepsilon))$$
$$\mathrm{lg}\ (\mathrm{Exp}_0\ (\mathrm{uop}\ (c))\ (\varepsilon)) = \mathrm{lg}\ (\mathrm{Exp}_0\ (c)\ (\varepsilon))$$

Opérations conditionnelles

$$\mathrm{Exp}_0\ (\textbf{if}\ c\ \textbf{then}\ c_1\ \textbf{else}\ c_2)\ (\varepsilon) =$$
$$\lambda n.\ \text{si}\ (\mathrm{Exp}_0\ (c)\ (\varepsilon)\ (n) = \text{nil})\ \text{alors nil}$$
$$\text{sinon si}\ (\mathrm{Exp}_0\ (c)\ (\varepsilon)\ (n))\ \text{alors}\ \mathrm{Exp}_0\ (c_1)\ (\varepsilon)\ (n)$$
$$\text{sinon}\ \mathrm{Exp}_0\ (c_2)\ (\varepsilon)\ (n)$$

$$\mathrm{lg}\ (\mathrm{Exp}_0\ (\textbf{if}\ c\ \textbf{then}\ c_1\ \textbf{else}\ c_2)\ (\varepsilon)) =$$
$$\inf\ (\mathrm{lg}\ (\mathrm{Exp}_0\ (c)\ (\varepsilon)),\ \mathrm{lg}\ (\mathrm{Exp}_0\ (c_1)\ (\varepsilon)),\ \mathrm{lg}\ (\mathrm{Exp}_0\ (c_2)\ (\varepsilon)))$$

Opérations temporelles

Exp_0 **(pre (c)) (ε) = Exp_0 (c) (ε) ∘ Dec**
lg (Exp_0 (pre (c)) (ε)) = lg (Exp_0 (c) (ε)) + 1

Où Dec = λn. si n ≤ 1 alors 0 sinon n-1 et ∘ dénote la composition fonctionnelle.

Exp_0 (c_1 -> c_2) (ε) =
λn. si (n ≤ 1) alors Exp_0 (c_1) (ε) (n) sinon Exp_0 (c_2) (ε) (n)
lg (Exp_0 (c_1 -> c_2) (ε)) =
si (lg (Exp_0 (c_1) (ε) ≥ 2) alors sup (2, lg (Exp_0 (c_2) (ε))) sinon 1

Tuplage

Exp_0 (c_1, c_2, ..., c_n) (ε)
= [Exp_0 (c_1) (ε), Exp_0 (c_2) (ε), ..., Exp_0 (c_n) (ε)]

Où [ ] dénote l'opération de tuplage.

La sémantique des expressions synchrones est très simple, elle consiste pour l'essentiel en la description de l'effet des opérateurs du langage sur les séquences.

1.5. Sémantique des équations

Le domaine sémantique d'une équation EQU est défini par

EQU = EV → EV

La sémantique d'une équation est une fonction des environnements dans les environnements.

En effet une équation permet de définir une ou plusieurs variables, c'est donc un enrichissement de l'environnement. Equ est définie par

Equ (id_1, ..., id_n = c) (ε) =
ε [π_1 (Exp_0 (c) (ε)), ..., π_n (Exp_0 (c) (ε))/id_1, ..., id_n]

π_i est l'opérateur de projection selon la i-ème composante
et f [y_0 / y] = λx. si (x = y) alors y_0 sinon f (x)

Nous supposons que l'équation est bien formée dans ε, c'est-à-dire que l'équation est correcte vis à vis de la sémantique statique (toutes les variables qui interviennent dans l'équation sont déclarées et l'expression e retourne un tuple de n suites définissant les variables et dont les types sont cohérents avec les déclarations de ces variables)

1.6. Sémantique d'un système d'équations

Le domaine sémantique d'un système d'équations **SEQ** est défini par

$$\mathbf{SEQ} = \mathbf{EV} \rightarrow \mathbf{EV}$$

La sémantique d'un système d'équations est une fonction des environnements dans les environnements.

$$\text{Seq}\ (\textbf{let}\ q_1, \ldots q_n\ \textbf{tel}) = \text{Equ}\ (q_1) \circ \ldots \circ \text{Equ}\ (q_n)$$

Nous ne définissons la sémantique que d'un système d'équations bien formé vis à vis de la sémantique statique de la même manière que pour les équations. C'est-à-dire un système d'équations bien formées dans lequel il n'y a pas de boucles de causalité.

1.7. Sémantique d'un programme

Le domaine sémantique d'un programme **PRG** est défini par

$$\mathbf{PRG} = \mathbf{S}^* \rightarrow \mathbf{S}^*$$

La sémantique d'un programme est une fonction des tuples de suites dans les tuples de suites. Cette définition reflète le fait qu'un programme LUSTRE est un transformateur d'histoires.

prog ::= **in** id_1; ..., id_n ; **out** id_{n+1}; ..., id_m ; **local** id_{m+1}; ..., id_k ; Sq

$$Prg\ (prog) = [\lambda\sigma_1, \ldots, \sigma_n.\ env_{prog}\ (id_i)]_{i\ =\ n+1..m}$$
$$env_{prog} = \mu_\varepsilon.(Seq\ (Sq)\ (\varepsilon\ [\sigma_1, \ldots, \sigma_n/id_1, \ldots, id_n]))$$

$\mu_\varepsilon.f\ (\varepsilon)$ définit le plus petit point fixe de f
et $[f\ (x_i)]_{i\ =\ 1..n} = [f\ (x_1), \ldots, f\ (x_n)]$

On ne définit la sémantique que pour un programme bien formé. C'est-à-dire un programme dont le système d'équations et bien formé et cohérent avec les déclarations. Un système d'équations est dit cohérent avec les déclarations

- si toute variable locale et toute variable de sortie est définie par une équation du système une fois et un seule et conformément à son type,

- si aucune équation ne définit une variable d'entrée,

- et enfin si toute variable apparaîssant dans le système d'équations a été déclarée une fois et une seule.

1.8. Le langage asynchrone L1

On enrichit le langage L0 des opérateurs **when** et $\mathbf{current_1}$. Les domaines syntaxiques différents de EXP_0 restent inchangés, ainsi que les domaines sémantiques associés. Le domaine syntaxique EXP_1 remplacera le domaine syntaxique EXP_0 . EXP_1 est défini à partir de EXP_0 par adjonction de deux règles de production.

$$e ::= e_1\ \mathbf{when}\ id \mid \mathbf{current_1}\ (e_1, id)$$

On définit $Exp_1 : EXP_1 \rightarrow (E \rightarrow S^*)$ telle que

$$Exp_1\ (e) = Exp_0\ (e)\ \text{si}\ e \in EXP_0$$

$$Exp_1\ (e_1\ \mathbf{when}\ id)\ (\varepsilon) = Exp_1\ (e_1)\ (\varepsilon) \circ \mathbf{rank}\ (\varepsilon\ (id))$$
$$lg\ (Exp_1\ (e_1\ \mathbf{when}\ id)\ (\varepsilon)) = \mathbf{count}\ (\varepsilon\ (id))\ (\infty)$$

$$Exp_1\ (\mathbf{current_1}\ (e_1, id))\ (\varepsilon) = Exp_1\ (e_1)\ (\varepsilon) \circ \mathbf{count}\ (\varepsilon\ (id))$$
$$lg\ (Exp_1\ (\mathbf{current_1}\ (e_1, id))\ (\varepsilon)) = inf\ (lg\ (Exp_1\ (e_1)\ (\varepsilon)), lg\ (\varepsilon\ (id)))$$

où **rank** et **count** sont deux fonctions définies par :

$$\textbf{count}\ (\sigma) = \lambda n.\ \text{card}\ \{\ m \leq n,\ \text{et}\ m \leq \text{lg}\ (\sigma)\ |\ \sigma\ (m) = tt\ \}$$
$$\textbf{rank}\ (\sigma) = \lambda n.\ \text{inf}\ \{\ m\ |\ \textbf{count}\ (\sigma)\ (m) \geq n\ \}$$

rank et **count** sont deux fonctions de **S** dans ($\mathbb{N} \rightarrow \mathbb{N}$)

Remarque

L'opérateur de projection $\textbf{current}_1$ du langage L1 est plus général que l'opérateur **current** du langage complet, ceci a pour conséquence qu'un programme dans le langage L1 ne sera pas forcément causal. Nous avons fait ce choix pour alléger l'exposé, puisque la sémantique de l'opérateur **current** est liée au calcul d'horloge. Ainsi nous avons pu séparer le calcul d'horloge (qui fait partie de la sémantique statique) de la sémantique dynamique. L'opérateur **current** sera introduit complètement dans la deuxième partie du chapitre.

1.9. Le langage structuré L2

Nous introduisons maintenant la possibilité de définir des noeuds dans un programme, ainsi que l'appel des noeuds dans les expressions. Un programme est donc maintenant augmenté d'un ensemble de définitions hierarchisées de noeuds. Nous supposerons dans toute la suite que les définitions de noeuds sont bien formées. Le programme aura passé correctement l'analyse de la sémantique statique, c'est-à-dire que la portée des noeuds est cohérente avec leur utilisation, qu'aucun noeud ne fait partie d'une boucle de récursivité, et que tous les appels sont conformes aux déclarations des noeuds quant au nombre de paramètres, leur type etc... .

Nous introduisons un domaine syntaxique supplémentaire noté *DND* qui est l'ensemble des déclarations de noeuds. La fonction sémantique associée, notée Dnd est une fonction de l'ensemble des environnements de déclarations de noeuds dans lui même. En effet, de même que la définition d'une variable dans un noeud enrichit l'environnement des définitions de variables, une déclaration de noeud enrichit l'environnement des déclarations de noeuds.

Soit **EN** l'environnement des déclarations de noeuds.

$$\mathbf{EN} = ID \rightarrow (\mathbf{S}^* \rightarrow \mathbf{S}^*)$$

la fonction sémantique Dnd est donc un élément de

$$DND \rightarrow (\mathbf{EN} \rightarrow \mathbf{EN})$$

La syntaxe abstraite du langage est modifiée :

prog ::= **in** $id_1, \ldots, id_n$ **out** $id_{n+1}, \ldots, id_m$; dnd ;
local $id_{m+1}, \ldots, id_k$; **let** Sq **tel**
dnd ::= **vide** | **node** id_1 $prog_1$; ...; **node** id_n $prog_n$

La sémantique d'un programme est inchangée

$$\text{Prg} : PRG \rightarrow (\mathbf{S}^* \rightarrow \mathbf{S}^*)$$

La sémantique d'une déclaration de noeud est une fonction qui associe un transformateur d'histoires à un identificateur dans un contexte donné. La sémantique d'une déclaration est définie comme suit :

dnd ::= **vide** | **node** id_1 $prog_1$; ...; **node** id_n $prog_n$
Dnd (vide) $(\varepsilon_n) = \varepsilon_n$
Dnd (dnd) $(\varepsilon_n) = \varepsilon_n$ [Prg $(prog_1)$, ..., Prg $(prog_n)/id_1, \ldots, id_n$]

Les domaines sémantiques autres que EXP_1 sont inchangés. EXP_2 est défini à partir de EXP_1 par adjonction d'une règle de production :

$$e ::= id\ (e)$$

Les domaines sémantiques sont transformés puisque l'on a modifié l'environnement. On définit **E** comme le produit cartésien de l'environnement des variables et l'environnement des noeuds.

$$\mathbf{E} = \mathbf{EN} \times \mathbf{EV}$$

si $\varepsilon \in \mathbf{E}$ alors $\varepsilon = (\varepsilon_n, \varepsilon_v)$

les nouveaux domaines sémantiques sont les suivants:

. **PRG** $= S^* \to S^*$
. **DND** $= E \to E$
. **SEQ** $= E \to E$
. **EQU** $= E \to E$
. **EXP₂** $= E \to S^*$

Les fonctions sémantiques ont changé elles aussi et l'on peut les définir ainsi:

$$Dnd_2 = (Dnd\ ,\ 1_{EV})$$

où Dnd est la fonction définie précédemment et 1_F désigne la fonction identité sur l'ensemble F

$$Seq_2 = (1_{EN}\ ,\ Seq)$$
$$Equ_2 = (1_{EN}\ ,\ Equ)$$

pour tout $e \in EXP_1$

$$Exp_2\ (e)\ (\varepsilon) = (\varepsilon_n\ ,\ Exp_1\ (e)\ (\varepsilon))$$

et

$$Exp_2\ (id\ (e))\ (\varepsilon_n\ ,\ \varepsilon_v) = [\varepsilon_n\ ,\ \varepsilon_n\ (id)\ (\pi_2\ (Exp_2\ (e)\ (\varepsilon_n\ ,\ \varepsilon_v)))]$$

L'adjonction d'appels de noeuds modifie essentiellement la structure des domaines, les fonctions restent très simples, et un appel de noeud n'est que l'application de la définition de la fonction liée au nom du noeud (ε_n (id)) à l'expression en paramètre (Exp_2 (e) (ε_n, ε_v)).

Sémantique d'un programme L2

prog ::= **in** $id_1, \ldots, id_n$ **out** $id_{n+1}, \ldots, id_m$; dnd ;
local $id_{m+1}, \ldots, id_k$; **let** Sq **tel**

$$Prg\ (prog) = [\lambda\sigma_1, \ldots, \sigma_n.\ env_{prog}\ (id_i)]_{i\ =\ n+1..m}$$

$$env_{prog} = \mu_{\varepsilon}.\ [Dnd_2\ (dnd)\ (\varepsilon'),\ Seq_2\ (Sq)\ (\varepsilon')]$$

$$\varepsilon' = [\pi_1\ (\varepsilon),\ \pi_2\ (\pi_2\ (\varepsilon\ [\sigma_1, \ldots, \sigma_n/id_1, \ldots, id_n]))]$$

Dans l'expression de la sémantique d'un programme, on a simplement tenu compte de la nouvelle structure des domaines sémantiques du langage L2 par rapport au langage L0 et L1.

2. SEMANTIQUE STATIQUE

Nous ne présentons que la partie la sémantique statique qui concerne les horloges. Une sémantique statique est en cours de définition, elle est donnée à l'aide de règles de réécriture "à la Plotkin" [Plotin 81] et sera présentée dans [Buors 86].

. Nous avons donné au sous-ensemble L2 du langage une sémantique "à la Kahn" [Kahn 83]. Ce sous-ensemble ne vérifie pas la contrainte de mémoire bornée, puisque l'on n'a pas imposé aux suites d'être synchrones. La sémantique dynamique que nous avons présentée, n'est réalisable que si l'on introduit pour chaque opérateur une file d'attente infinie pour ses opérandes.

Nous voulons que LUSTRE soit un langage qui permette de définir des programmes exécutables avec une mémoire finie, bornée (puisque la taille des files d'attente doit être réduite à 1). Pour cela, nous avons introduit des contraintes de synchronisme entre les variables d'un programme (celles-ci n'étant plus uniquement des suites mais des couples (suite, horloge)). On ne pourra composer entre elles que des suites ayant leur n-ième valeur strictement en même temps.

On veut pouvoir vérifier ces contraintes, le plus simplement possible, et surtout statiquement (à la compilation) ; c'est pourquoi nous avons introduit une définition syntaxique des horloges :

Une horloge est

- soit un identificateur de variable booléenne

- soit la constante "tt" (appelée dans ce contexte *horloge de base*)

2.1. Domaines syntaxiques

Les domaines syntaxiques sont les domaines du langage L2 que nous rappelons brièvement :

Domaine des expressions

$$e ::= k \mid id \mid e_1 \textbf{ bop } e_2 \mid \textbf{uop } e \mid \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \mid \textbf{pre } (e) \mid e_1 \text{ -> } e_2 \mid e_1, \ldots, e_n \mid id\ (e) \mid e \textbf{ when } id \mid \textbf{current } (e)$$

Domaine des équations

$$q ::= (id_1, \ldots, id_n) = e$$

Domaine des systèmes d'équations

$$Sq ::= \textbf{let } q_1; \ldots; q_n \textbf{ tel}$$

Domaine des programmes

$$prog ::= \textbf{in } id_1, \ldots, id_n \textbf{ out } id_{n+1}, \ldots, id_m \text{ ; dnd ; } \textbf{local } id_{m+1}, \ldots, id_k \text{ ; } \textbf{let } Sq \textbf{ tel}$$

$$dnd ::= \textbf{vide} \mid \textbf{node } id_1\ prog_1; \ldots; \textbf{node } id_n\ prog_n$$

2.2. Domaines sémantiques

Une horloge est un identificateur de variable booléenne ou bien la constante tt. Nous définissons l'ensemble *HOR* des horloges structuré en treillis plat

- $HOR = \{tt\} \cup IDBOOL \cup \{\top, \bot\}$

où *IDBOOL* est l'ensemble des identificateurs de booléens, $\bot$ et $\top$ sont respectivement les éléments minimum et maximum du treillis.

La relation d'ordre sur *HOR* est notée $\sqsubseteq$, elle est définie classiquement par :

$$\text{soient } h_1 \text{ et } h_2 \in HOR$$
$$h_1 \sqsubseteq h_2 \Leftrightarrow h_1 = \bot \text{ ou } h_2 = \top \text{ ou } h_1 = h_2$$

- On définit l'ensemble **HV** des environnements d'horloges qui associe une horloge à chaque identificateur de variable.

$$\mathbf{HV} = ID \rightarrow HOR$$

L'ordre sur **HV** est l'ordre induit par l'ordre sur *HOR*. Nous noterons ω_v un élément de **HV**.

- On définit l'ensemble **HN** des environnements de transformateurs d'horloges qui associe un transformateur d'horloges à chaque identificateur de noeud.

$$\mathbf{HN} = ID \rightarrow HOR^* \rightarrow HOR^*$$

Nous noterons ω_n un élément de **HV**.

- $\mathbf{H} = \mathbf{HN} \times \mathbf{HV}$ on notera $\omega = (\omega_n, \omega_v)$ un élément de **H**

2.3. Fonctions sémantiques

A toute expression, on associe une fonction des environnements d'horloges dans les horloges

$$\text{Hexp} : EXP \rightarrow \mathbf{HEXP} \text{ avec } \mathbf{HEXP} = \mathbf{H} \rightarrow HOR^*$$

A tout système d'équations et à toute équation, on associe un transformateur d'environnements d'horloges

$$\text{Hseq} : SEQ \rightarrow \mathbf{HSEQ} \text{ avec } \mathbf{HSEQ} = \mathbf{H} \rightarrow \mathbf{H}$$
$$\text{Hequ} : EQU \rightarrow \mathbf{HEQU} \text{ avec } \mathbf{HEQU} = \mathbf{H} \rightarrow \mathbf{H}$$

A toute déclaration de noeud, on associe une fonction de l'ensemble des environnements de transformateurs d'horloges dans lui-même.

$$\text{Hdnd} : DND \rightarrow \mathbf{HDND} \text{ avec } \mathbf{HDND} = \mathbf{H} \rightarrow \mathbf{H}$$

A tout noeud, on associe un transformateur d'horloge

$$\text{Hprg} : PRG \rightarrow \mathbf{HPRG} \text{ avec } \mathbf{HPRG} = HOR^* \rightarrow HOR^*$$

Ces structures sont similaires à celles définies pour la sémantique dynamique.

2.4. Sémantique du langage L2 - Le calcul d'horloge

2.4.1. Sémantique des expressions

Le domaine sémantique d'une expression **HEXP** est défini par

$$\mathbf{HEXP} = \mathbf{H} \rightarrow HOR^*$$

Une expression ne modifiera jamais l'environnement des transformateurs d'horloges, et elle ne l'utilisera que dans le cas d'un appel de noeud.

Constantes

L'horloge d'une constante est toujours l'horloge de base du noeud où elle apparaît

$$\text{Hexp}\ (k)\ (\omega_n, \omega_v) = (\omega_n, tt)$$

Identificateurs de variables

$$\text{Hexp}\ (id)\ (\omega_n, \omega_v) = (\omega_n, \omega_v\ (id))$$

Opérations synchrones

Cette catégorie contient les opérateurs instantanés et les opérateurs temporels synchrones (exception faite donc de **when** et **current**). Puisque ces opérations doivent être synchrones, il faut donc que leurs opérandes aient la même horloge, dans ce cas l'horloge du résultat est égale à l'horloge des opérandes.

$$\begin{aligned}&\text{Hexp}\ (op\ (e_1, \ldots, e_n))\ (\omega) = \\ &\qquad (\pi_1\ (\omega), \sup\ (\pi_2\ (\text{Hexp}\ (e_1)\ (\omega)), \ldots, \pi_2\ (\text{Hexp}\ (e_n))\ (\omega)))\end{aligned}$$

Remarque

L'opération sup (h_1, h_2) donnera T si $h_1 \neq h_2$ donc T sera interprété comme l'horloge erreur.

Tuplage

L'opération de tuplage peut être vue comme le cas particulier d'un opérateur d'arité variable. Mais le tuplage autorise la manipulation d'expressions non synchrones. L'horloge d'un tuple d'expressions sera donc le tuple des horloges des expressions qui le composent. Soit

$$Hexp\ (c_1, c_2, \ldots, c_n)\ (\omega) = [Hexp\ (c_1)\ (\omega), Hexp\ (c_2)\ (\omega), \ldots, Hexp\ (c_n)\ (\omega)]$$

Appels de noeuds

Un noeud définit un transformateur d'horloge, un appel de noeud effectue cette transformation

$$Hexp\ (id\ (e))\ (\omega_n, \omega_v) = (\omega_n, \omega_n\ (id)\ (Hexp\ (e)\ (\omega_n, \omega_v)))$$

Filtrage

L'opérateur **when** est le seul opérateur qui prend explicitement en paramètre une horloge.

$$Hexp\ (e\ \textbf{when}\ id)\ (\omega_n, \omega_v) = \begin{cases} (\omega_n, id) \text{ si } sup\ (\pi_2\ (Hexp\ (e)\ (\omega_n, \omega_v), \omega_v\ (id)) \neq T \\ (\omega_n, T) \text{ sinon} \end{cases}$$

Le filtrage doit être fait à partir de suites synchrones c'est pourquoi il y a une condition dans le calcul de l'horloge.

Projection

Puisque nous disposons de la définition des horloges, nous pouvons introduire l'opérateur **current** du langage. Nous le définissons à partir de **current**$_I$.

Dans l'environnement ω

$$\textbf{current}\ (e) = \textbf{current}_I\ (e, \pi_2\ (Hexp\ (e)\ (\omega)))$$

La sémantique dynamique de **current** est définie par **current**$_I$ (e) (cf. partie précédente). Nous définissons l'horloge d'une expression **current**

$$Hexp\ (\textbf{current}\ (e)\ (\omega_n, \omega_v)) = \begin{cases} (\omega_n, T) \text{ si } \pi_2\ (Hexp\ (e)\ (\omega)) = T \\ (\omega_n, \bot) \text{ si } \pi_2\ (Hexp\ (e)\ (\omega)) = \bot \\ Hexp\ (e)\ (\omega_n, \omega_v) \text{ si } \pi_2\ (Hexp\ (e)\ (\omega)) = tt \\ (\omega_n, \omega_v\ (\pi_2\ (Hexp\ (e)\ (\omega)))) \text{ sinon} \end{cases}$$

2.4.2. Sémantique des équations et des systèmes d'équations

Le domaine sémantique d'une équation **HEQU** est défini par

$$\mathbf{HEQU} = \mathbf{H} \to \mathbf{H}$$

Le domaine sémantique des équations est l'ensemble des transformateurs de l'environnement **H**

$$\begin{aligned} &Hequ\ (id_1, \ldots, id_n = e)\ (\omega_n, \omega_v) = \\ &\qquad (\omega_n, \\ &\qquad \omega_v\ [\pi_1\ (\pi_2\ (Exp\ (e)\ (\omega))), \ldots, \pi_n\ (\pi_2((Exp\ (e)\ (\omega)))/id_1, \ldots, id_n]) \\ &où\ \omega = (\omega_n, \omega_v) \end{aligned}$$

Le domaine sémantique d'une équation **HSEQ** est défini par

$$\mathbf{HSEQ} = \mathbf{H} \to \mathbf{H}$$

Le domaine sémantique des équations est l'ensemble des transformateurs de l'environnement **H**

$$Hseq\ (\mathbf{let}\ q_1, \ldots q_n\ \mathbf{tel}) = Hequ\ (q_1) \circ \ldots \circ Hequ\ (q_n)$$

2.4.3. Sémantique d'un noeud

Le domaine sémantique d'un noeud **HPRG** est définie par

$$\mathbf{HPRG} = HOR^* \to HOR^*$$

Un noeud est un transformateur d'horloges.

$$\begin{aligned} prog \quad ::= &\ \mathbf{in}\ id_1, \ldots, id_n\ \mathbf{out}\ id_{n+1}, \ldots, id_m\ ;\ dnd\ ; \\ &\ \mathbf{local}\ id_{m+1}, \ldots, id_k\ ;\ \mathbf{let}\ Sq\ \mathbf{tel} \end{aligned}$$

$$Hprg\ (prog) = [\lambda h_1, \ldots, h_n.\ henv_{prog}\ (id_i)]_{i\ =\ n+1..m}$$

$$henv_{prog} = \mu_\omega.[Hdnd\ (dnd)\ (\omega'), Hseq\ (Sq)\ (\omega')]$$

où

$$\omega' = [\pi_1\ (\omega), \pi_2\ (\pi_2\ (\omega)\ [h_1, \ldots, h_n/id_1, \ldots, id_n])]$$

2.4.4. Sémantique d'une déclaration de noeud

Le domaine sémantique **HDND** d'une déclaration de noeud est défini par

$$\mathbf{HDND} = \mathbf{H} \rightarrow \mathbf{H}$$

dnd ::= **vide | node** $\mathbf{id_1}$ $\mathbf{prog_1}$; ...; **node** $\mathbf{id_n}$ $\mathbf{prog_n}$

Hdnd (vide) $(\omega) = \omega$
Hdnd (dnd) $(\omega_n, \omega_v) = [\omega_n', \omega_v]$

$\omega_n' = \omega_n\ [\pi_1\ (\mathrm{Hprg}\ (prog_1)\ (\omega), \ldots, \pi_1\ (\mathrm{Hprg}\ (prog_n)\ (\omega)/id_1, \ldots, id_n]$
où $\omega = (\omega_n, \omega_v)$

Cette fonction permet d'associer un transformateur d'horloge à un identificateur de noeud.

3. CONCLUSION

La sémantique du langage présentée dans ce chapitre a été volontairement simplifiée pour ne présenter que les traits particuliers du langage. Nous n'avons pas traité les déclarations, le typage, la portée des identificateurs de noeuds, et toutes les vérifications qui y sont liées, car ceci relève des techniques habituelles utilisées dans la définition de la sémantique des langages classiques.

La sémantique dynamique est très simple puisque le langage est fonctionnel, et donc sans effet de bord. La sémantique d'un programme, considéré comme un transformateur de suites, est le plus petit point fixe défini par son système d'équations.

La sémantique statique ne pose pas de problème particulier au niveau des horloges. La complexité du calcul d'horloge est de l'ordre de la complexité de la synthèse des types, le point fixe, qui permet de définir l'ensemble des horloges ou de détecter une erreurs sera obtenu très vite (en quelques itérations) à partir d'un environnement dans lequel toutes les horloges sont égales à $\perp$. On aurait pu supprimer le calcul de ce point fixe soit en ordonnant les équations pour que le calcul puisse être fait directement, soit en obligeant le programmeur à préciser les horloges des variables à la déclaration.

Enfin, l'introduction de contraintes syntaxiques pour la définition de la synchronisation, a beaucoup simplifié le calcul d'horloge d'un programme. le fait que le calcul d'horloge puisse être fait de façon statique, nous paraît un atout important pour la facilité de la mise au point des programmes, et de leur preuve.

CHAPITRE 5

TRANSFORMATION DE PROGRAMMES

Le caractère mathématique du langage LUSTRE offre des possibilités multiples de transformation formelle de programmes, à sémantique constante, comme en LUCID [Richier 82]. Ces transformations sont intéressantes soit pour prouver l'equivalence de deux programmes, soit dans un but d'optimisation (par le programmeur ou même par le compilateur). Ce domaine n'ayant pas encore été étudié de façon très approfondie, nous nous contenterons dans ce chapitre de donner et d'illustrer les règles de transformation les plus immédiates.

1. PRINCIPE DE SUBSTITUTION

Cette règle est fondamentale en LUSTRE : *on peut remplacer toute occurrence d'une variable par sa définition.* Ce principe n'est infirmé que par les expressions contenant l'opérateur **every** ou le générateur aléatoire **alea**.

X = Z+T; Y = f (X) ≡ X = Z+T; Y = f (Z+T)

Si l'on suppose maintenant que f est le noeud défini de la façon suivante

```
node f (X : int) returns (N : int);
  let
    N = 1 -> pre (N) + X;
  tel;
```

On peut remplacer f (Z+T) par sa définition dans l'équation qui définit Y, soit

Y = 1 -> **pre** (Y) + (Z+T);

Elimination et introduction de variables locales

Il est toujours possible de remplacer une expression quelconque dans le système d'équations par une variable locale que l'on introduit, et dont la définition sera cette expression. Cette possibilité n'est pas particulière à LUSTRE, mais son

exploitation est facilitée, dans un langage déclaratif, par le fait que l'ordre des définitions est sans signification.

Par exemple, si dans un sytème d'équations, on rencontre l'équation suivante

$$X = \textbf{if } (A>B+C) \textbf{ then } X_1+Y_1 \textbf{ else } X_2+Y_2 \qquad (1)$$

Si l'on introduit les variables locales LC, LX_1, LX_2

$$\begin{aligned} LC &= A>B+C; \\ LX_1 &= X_1+Y_1; \\ LX_2 &= X_2+Y_2; \\ X &= \textbf{if } LC \textbf{ then } LX_1 \textbf{ else } LX_2 \end{aligned} \qquad (2)$$

Il est toujours légal de remplacer l'équation (1) par le système d'équations (2)

La transformation inverse est possible également, à condition que toutes les variables qui disparaîssent soient des variables locales. On pourra alors remplacer toutes les occurrences de ces variables par leur définition. L'élimination d'une variable locale ne sera possible que si le processus de substitution de la variable par sa définition se termine. En d'autres termes, cette élimination sera impossible pour toute variable locale récurrente.

Exemple

X = 1 -> **pre** (X) + 1;
Y = Z + X;

X est une variable locale qui ne pourra pas être éliminée. En effet, en remplaçant X par sa définition dans Y, on obtient

Y = Z + (1 -> **pre** (X) + 1) ;

2. AXIOMATIQUE DES OPERATEURS TEMPORELS SYNCHRONES

Nous donnons ci-dessous un système d'axiomes complet des opérateurs **pre** et ->. Les axiomes sont au nombre de cinq :

1. X = X -> X
2. (X -> Y) -> Z = X -> Z
3. X -> (Y -> Z) = X -> Z
4. **pre** (C) = nil -> C si et seulement si C est une constante
5. **pre** (X) = nil -> **pre** (X)

Validité

Nous pouvons démontrer ces axiomes relativement à la sémantique : (nous confondons dans ce qui suit les dénotations des identificateurs de varibles et les dénotations de leur sémantique)

- X = X -> X

X -> X
= λn. si n $\leq$ 1 alors X sinon X
= X

- (X -> Y) -> Z = X -> Z

X -> (Y -> Z)
= λn. si n $\leq$ 1 alors X sinon (si n$\leq$ 1 alors Y sinon Z)
= λn. si n $\leq$ 1 alors X sinon Z
= X -> Z

- X -> (Y -> Z) = X -> Z

(X -> Y) -> Z
= λn. si n $\leq$ 1 alors (si n$\leq$ 1 alors X sinon Y) sinon Z
= λn. si n $\leq$ 1 alors X sinon Z
= X -> Z

- **pre** (C) = nil -> C

pre (C)
= λn. C $\circ$ **dec** (n)
= λn. C (si n$\leq$ 1 alors 0 sinon n-1)
= λn. si n$\leq$ 1 alors C (0) sinon C (n-1)
= λn. si n$\leq$ 1 alors nil sinon C (n-1)
= λn. si n$\leq$ 1 alors nil sinon C (n) (puisque C est constant)
= nil -> C

- **pre** (X) = nil -> **pre** (X)

pre (X)

$$\begin{aligned}
&= \lambda n.\ X \circ \mathbf{dec}\ (n) \\
&= \lambda n.\ X\ (\text{si } n \leq 1 \text{ alors } 0 \text{ sinon } n\text{-}1) \\
&= \lambda n.\ \text{si } n \leq 1 \text{ alors } X\ (0) \text{ sinon } X\ (n\text{-}1) \\
&= \lambda n.\ \text{si } n \leq 1 \text{ alors nil sinon } X\ (n\text{-}1) \\
&= \lambda n.\ \text{si } n \leq 1 \text{ alors nil sinon } X\ (\text{si } n \leq 1 \text{ alors } 0 \text{ sinon } n\text{-}1) \\
&= \lambda n.\ \text{si } n \leq 1 \text{ alors nil sinon } X \circ \mathbf{dec}\ (n) \\
&= \text{nil} \rightarrow \mathbf{pre}\ (X)
\end{aligned}$$

Complétude

On considère l'algèbre (Const, **pre**, ->), où Const est un alphabet, interprété comme l'ensemble des constantes. Nous allons montrer que les axiomes permettent de mettre tout terme de cette algèbre sous la forme:

$$C_0 \ \mathbf{fby}\ C_1 \ \mathbf{fby}\ C_2 \ \mathbf{fby} \ldots \mathbf{fby}\ C_n$$

où X **fby** Y = X -> **pre** (Y) et où n est soit un entier, soit ∞

L'ensemble des termes de cette forme est trivialement isomorphe à l'ensemble des suites d'élements de Const.

La démonstration se fait par induction sur la complexité des termes: Soit X un terme, alors

- Si X = C ∈ Const, X est déjà sous la forme voulue;
- Si X = **pre** (Y), X se réécrit en nil **fby** Y, d'après l'axiome (5);
- Si X = Y -> Z alors

d'une part

- si Y est une constante, c'est la constante C_0 de la forme cherchée
- si Y = **pre** (W) alors X = nil -> Z , d'après les axiomes (5) et (2)
- si Y = W -> T alors X = W -> Z , d'après l'axiome (2) et l'hypothèse d'induction s'applique

et d'autre part

- si Z ∈ Const, alors
  X = Y -> (nil -> C) d'après l'axiome (3)
  = Y **fby** C d'après l'axiome (4), et c'est la forme cherchée
- si Z = **pre** (W), alors X = Y **fby** W , et c'est la forme cherchée
- si Z = W -> T, alors X = Y -> T d'après l'axiome (3), et l'on applique l'hypothèse d'induction

3. PROPRIETES DES OPERATEURS

Nous ne traitons pas le cas de l'opérateur **every** qui ne possède pas de propriétés intéressantes.

3.1. Distributivités

Soit **op** un opérateur instantané n-aire du langage.

Propriété 1 : -> distribue sur **op**

$$\mathbf{op}\ (E_1, \ldots, E_n) \text{ -> } \mathbf{op}\ (F_1, \ldots, F_n) = \mathbf{op}\ (E_1\text{->}F_1, \ldots, E_n\text{->}F_n)$$

Propriété 2 : **pre** distribue sur **op**

$$\mathbf{pre}\ (\mathbf{op}\ (E_1, \ldots, E_n)) = \mathbf{op}\ (\ \mathbf{pre}\ (E_1)\ , \ldots, \mathbf{pre}\ (E_n))$$

Propriété 3 : **when** distribue sur **op**

$$\mathbf{op}\ (E_1, \ldots, E_n)\ \mathbf{when}\ C = \mathbf{op}\ ((E_1\ \mathbf{when}\ C)\ , \ldots, (E_n\ \mathbf{when}\ C))$$

Propriété 4 : **current** distribue sur **op**

$$\mathbf{current}\ (\mathbf{op}\ (E_1, \ldots, E_n)) = \mathbf{op}\ (\ \mathbf{current}\ (E_1)\ , \ldots, \mathbf{current}\ (E_n))$$

On ne démontre pas ces propriétés qui se déduisent directement de la définition des opérateurs. Des démonstrations analogues ont été faites pour les axiomes dans le paragraphe précédent.

3.2. Les booléens et la conditionnelle

Les propriétés classiques de la conditionnelle sont les suivantes:

if C **then** true **else** false = C

if C **then** X **else** Y = **if not** (C) **then** Y **else** X

Malheureusement la propriété suivante n'est pas vérifiée :

if C **then** X **else** X = X

Cette propriété est fausse puisque lorsque C vaut nil l'expression **if** C **then** X **else** X vaut nil quelle que soit la valeur de X.

Cet état de chose est un peu fâcheux car les relations entre une expression conditionnelle portant sur des booléens et une expression booléenne ne tiennent plus. Par exemple

(A **and** B) **or** (**not** (A) **and** C) ≠ **if** A **then** B **else** C

puisque lorsque B ou C valent nil, l'expression booléenne vaut nil quelle que soit la valeur de A, alors que l'expression conditionnelle va absorber la valeur nil qand elle est présente sur la branche non selectionnée

La propriété qui lie des expressions de ce type est la suivante

(A **and** B) **or** (**not** (A) **and** C)
= **if isnil** (B **or** C) **then** nil **else if** A **then** B **else** C

Si maintenant on suppose qu'aucune variable ne rend la valeur nil, alors "tout rentre dans l'ordre" et les propriétés tiennent. Il serait donc souhaitable d'interdire l'apparition dynamique de la valeur nil. Afin de pouvoir vérifier statiquement cette propriété, nous envisageons d'imposer de nouvelles contraintes dans la sémantique statique. La valeur nil apparaît dans deux cas : lorsque l'opérateur **pre** n'est pas précédé d'un -> (ce cas est facilement détectable) et lorsque l'opérateur **current** est appliqué avant que l'horloge de son opérande ait éte vraie pour la première fois. Ce dernier cas ne peut être facilement écarté que si l'on impose que toute horloge soit vraie au premier cycle.

Si l'on admet cette propriété, on peut, à présent, donner des propriétés plus agréables des opérateurs du langage

Propriétés 5

if C **then** true **else** false = C (*)

if C **then** X **else** Y = **if not** (C) **then** Y **else** X (*)

if C **then** X **else** X = X

if true **then** X **else** Y = X (*)

if false **then** X **else** Y = Y (*)

(A **and** B) **or** (**not** (A) **and** B) = **if** A **then** B **else** C

Les propriétés marquées d'un astérisque sont toujours vérifiées, avec ou sans la suppression de la valeur nil.

Propriété 6

if C_1->C_2 **then** X_1->X_2 **else** Y_1->Y_2 =
if C_1 **then** X_1 **else** Y_1 -> **if** C_2 **then** X_2 **else** Y_2 (*)

En particulier on peut démontrer que

if C_1->C_2 **then** X **else** Y
= **if** C_1 **then** X **else** Y -> **if** C_2 **then** X **else** Y (*)

if true->false **then** X **else** Y = X -> Y (*)

On ne démontre pas ces propriétés : on peut les établir immédiatement partir de la définition des opérateurs.

Propriété 7

Cette propriété est très importante pour la transformation de programmes synchrones en programmes asynchrones.

current (X **when** C) = **if** C **then** X **else pre** (**current** (X **when** C))

On voit qu'elle va permettre de transformer une équation de la forme Y = **if** C **then** X **else pre** (Y) en Y = **current** (X **when** C) et inversement

Démonstration

Il faut démontrer que les sémantiques des deux expressions sont identiques soit

current (X **when** C) = λn. X $\circ$ **rank** (C) $\circ$ **count** (C) d'une part

if C **then** X **else** **pre** (**current** (X **when** C)) =

λn. si C (n) alors X (n) sinon X $\circ$ **rank** (C) $\circ$ **count** (C) $\circ$ **dec**

d'autre part.

Lemme

λn. **rank** (C) $\circ$ **count** (C) = λn. si C (n) alors n sinon **rank** (C) $\circ$ **count** (C) $\circ$ **dec**

Rapelons tout d'abord les définitions de **rank** et **count** :

count (C) (n) = **card** { m ≤ n | C (m) = true }
rank (C) (n) = **inf** { m | **count** (C) (m) ≥ n }

En les composant, on obtient :

rank (C) $\circ$ **count** (C) (n) = **inf** { m | **count** (C) (m) = **count** (C) (n) }
= **inf** { m | **card** { i ≤ m | C (i) = true } = **card** { i ≤ n | C (i) = true

Si C (n) est égal à false, on remarque que

card { m ≤ n | C (m) = true } = **card** { m ≤ n-1 | C (m) = true } si n ≥ 1
card { m ≤ 0 | C (m) = true } = 0 sinon

soit si C (n) = false
rank (C) $\circ$ **count** (C) (n) = **rank** (C) $\circ$ **count** (C) $\circ$ **dec**

Si C (n) est égal à true, on remarque que

pour tout m < n
card { i ≤ m | C (i) = true } < **card** { i ≤ n | C (i) = true }

donc pour que
card { i ≤ m | C (i) = true } ≥ **card** { i ≤ n | C (i) = true }
il faut et il suffit que m ≥ n

et il vient
rank (C) $\circ$ **count** (C) (n) = si C (n) alors **inf** { m | m ≥ n }
sinon **rank** (C) $\circ$ **count** (C) $\circ$ **dec**
soit
rank (C) $\circ$ **count** (C) (n) = si C (n) alors n sinon **rank** (C) $\circ$ **count** (C) $\circ$ **dec**

De ce lemme on tire directement que

```
current (X when C)
        = λn. X ∘ rank (C) ∘ count (C)
        = λ n. X (si C (n) alors n sinon rank (C) ∘ count (C) ∘ dec)
        = λ n. si C (n) alors X (n) sinon X ∘ rank (C) ∘ count (C) ∘ dec
        = if C then X else pre (current (X when C)))
```

□

4. EXEMPLES DE TRANSFORMATIONS

Nous présentons deux petits exemples de transformations de programmes simples, mais qui illustrent assez bien les possibilités du langage.

4.1. Optimisation d'un compteur

```
node N (C1, C2 : bool) returns (N : int);
   var CPT1, CPT2 :int;
   let
      CPT1 = 0 -> if C1 then pre (CPT1) + 1 else pre (CPT1);
      CPT2 = 0 -> if C2 then pre (CPT2) + 1 else pre (CPT2);
      N = CPT1 - CPT2;
   tel
```

Dans cet exemple, on remarque que les compteurs CPT1 et CPT2 vont provoquer fatalement un débordement si C1 et C2 sont vraies "suffisamment" souvent.

Pour éviter cela, on pourrait définir des compteurs modulo une borne, et gérer les dépassements, mais ce serait une solution un peu lourde.

Nous allons plutôt modifier formellement ce programme afin de calculer directement la différence des deux compteurs.

On peut exprimer N différemment, en remplaçant les variables CPT1 et CPT2 par leur définition.

```
N = (0 -> if C1 then pre (CPT1) + 1 else pre (CPT1))
    - (0 -> if C2 then pre (CPT2) + 1 else pre (CPT2));
```

et en simplifiant les expressions

```
N = 0 -> (if C1 then pre (CPT1) + 1 else pre (CPT1))
      - (if C2 then pre (CPT2) + 1 else pre (CPT2));
```

ou bien encore en faisant apparaître l'expression **pre (CPT1) - pre (CPT2)** chaque fois que c'est possible

```
N = 0 -> if C1 and C2
           then pre (CPT1) - pre (CPT2) + 1 - 1
           else
             if C1 and not (C2) then pre (CPT1) - pre (CPT2) + 1
           else
             if not (C1) and C2 then pre (CPT1) - pre (CPT2) - 1
           else pre (CPT1) - pre (CPT2);
```

et en remplaçant **pre (CPT1) - pre (CPT2)** par **pre (CPT1 - CPT2)** c'est à dire **pre (N)** on obtient

```
N = 0 -> if C1 and C2
           then pre (N)
           else
             if C1 and not (C2) then pre (N) + 1
           else
             if not (C1) and C2 then pre (N) - 1
           else pre (N);
```

Soit encore en regroupant les deux branches qui donnent **pre (N)**

```
N = 0 -> if (C1 and C2) or (not (C1) and not (C2)) then pre (N)
           else
             if C1 then pre (N) + 1 else pre (N) - 1
```

Ainsi on a pu éliminer les deux compteurs et définir directement leur différence. Enfin en remarquant que

(C1 and C2) or (not (C1) and not (C2)) $\equiv$ **(C1 = C2)**

Le programme devient

```
node N (C1, C2 : bool) returns (N : int);
   let
      N = 0 -> if (C1 = C2) then pre (N)
                else
                   if C1 then pre (N) + 1 else pre (N) - 1
   tel
```

4.2. Désynchronisation d'un compteur d'évènements

Le deuxième exemple est un exemple de désynchronisation. Nous reprenons le programme compteur d'évènements donné dans le chapitre consacré à la définition du langage.

```
node COMPTEUR-EVENEMENT (evenement, remise-a-zero : bool)
                   returns (compteur : int);
let
      compteur = 0 -> if remise-a-zero then 0
                   else if evenement then pre (compteur)+1
                   else pre (compteur)
tel;
```

Nous allons essayer de désynchroniser ce compteur, de sorte à ne calculer que les termes significatifs de la suite compteur. C'est à dire fabriquer une suite telle que deux éléments successifs soient toujours différents. Nous allons pour cela, transformer l'expression qui définit la variable compteur pour obtenir une équation de la forme

compteur = **if** condition **then** expression **else pre** (compteur)

Lorsqu'elle sera sous cette forme, la définition de compteur pourra être transformée en :

compteur = **current** (expression **when** condition)

La première étape pour aboutir à cette forme est l'élimination de l'opérateur -> à l'aide de la règle de transformation

- E1 -> E2 ≡ **if** (true->false) **then** E1 **else** E2

et l'on obtient

```
compteur = if (true->false) then 0
           else if remise-a-zero then 0
           else if evenement then pre (compteur) + 1
           else pre (compteur)
```

Il faut maintenant que la branche conditionnelle qui contient l'expression **pre** (compteur) soit la partie **else** de l'expression conditionnelle la plus extérieure. Pour réaliser cette transformation, nous supposons qu'aucune des conditions ne s'évalue à nil et nous utilisons la règle suivante deux fois

- **if** C1 **then** E1 **else** (**if** C2 **then** E2 **else** E3) ≡
  if C1 **or** C2 **then** (**if** C1 **then** E1 **else** E2) **else** E3

Ce qui donne l'expression ci-dessous

```
compteur = if (true->false) or remise-a-zero or evenement then
              (if true->false then 0
               else if remise-a-zero then 0
               else pre (compteur) + 1)
           else pre (compteur)
```

ou encore

```
compteur = if condition then
              (if true->false then 0
               else if remise-a-zero then 0
               else pre (compteur) + 1)
           else pre (compteur)
condition = (true->false) or remise-a-zero or evenement
```

L'expression qui définit la variable compteur est sous la forme que l'on voulait, et donc on peut effectuer la dernière transformation, soit :

```
compteur = current (
               (if true->false then 0
                else
                       if remise-a-zero then 0
                       else pre (compteur)+1))
                when condition )
condition = (true->false) or remise-a-zero or evenement
```

Il est à remarquer que l'on peut alléger l'écriture des conditions en appliquant de nouveau la règle

- **if** (true->false) **then** E1 **else** E2 ≡ E1->E2

```
compteur = current (
               (0 -> if remise-a-zero then 0 else pre (compteur)+1))
                when condition )
```

Enfin en remarquant que

- (true->false) **or** C ≡ true->C

On obtient

```
condition = true-> (remise-a-zero or evenement)
```

et le programme définitif

```
node COMPTEUR-EVENEMENT (evenement, remise-a-zero : bool)
                  returns (compteur : int);
var condition : bool;
let
      compteur = current (
                   (0 -> if remise-a-zero then 0
                         else pre (compteur)+1)
                 when condition);
     condition = true-> (remise-a-zero or evenement)
tel;
```

Une telle écriture de la définition de la variable permet de déterminer clairement les instants où l'on doit calculer une nouvelle valeur du compteur, c'est bien évidemment lorsqu'il y a une occurrence de l'évènement ou bien une occurrence de la remise à zéro. Ceci n'apparaîssait pas explicitement dans l'expression initiale, alors que dans l'expression finale, cette propriété est mise en évidence. Il est evidemment plus facile à un compilateur de produire du code efficace à partir d'une expression de la dernière forme plutôt que de la forme initiale.

CONCLUSION

1. BILAN

Afin de tirer un bilan de ce travail, il faut comparer LUSTRE aux différents langages temps réel mentionnés dans le chapitre 1. Tout d'abord, la comparaison avec les langages pseudo-parallèles se passe de tout commentaire, puisque ces langages ne permettent pas d'exprimer des contraintes de temps. En ce qui concerne les langages à processus asynchrones, les avantages du modèle synchrone (en ce qui concerne la simplicité et la rigueur) commencent à être reconnus dans le domaine du temps réel ; nous espérons en avoir apporté une démonstration supplémentaire. De même, vis à vis des langages graphiques, nous sommes convaincus que de bonnes primitives de structuration des programmes apportent une lisibilité comparable à celle des langages graphiques, dans le cas des "petits" programmes, et que cette lisibilité se détériore beaucoup moins vite lorsque la complexité des programmes croît. Cette conclusion ne nous empêche pas d'envisager un éditeur graphique permettant de visualiser, à un niveau de détail variable, l'exécution d'un programme LUSTRE sur son réseau de noeuds.

La comparaison avec les langages synchrones est plus délicate (et peut-être plus subjective). Nous parlerons successivement d'ESTEREL, de LTS et de SIGNAL.

ESTEREL a le mérite d'avoir adopté une approche formelle, tant dans la définition du langage que dans son implémentation, et d'avoir "inventé" le modèle synchrone. Une autre qualité d'ESTEREL, que nous n'avons pu préserver dans LUSTRE, est le caractère multiforme du temps (absence d'horloge de base). Cette différence est due au fait que le mode de fonctionnement implicite d'ESTEREL est fondé sur des interruptions (l'environnement interrompt le programme pour communiquer avec lui) alors qu'en LUSTRE, un programme est supposé scruter périodiquement son environnement. L'adéquation de chacune de ces approches dépend évidemment du domaine d'application. Il reste qu'à notre avis, le

parallélisme explicite imposé par la nature impérative du langage peut être jugé difficile à appréhender. Ceci dit, on peut penser que cette réserve dépend elle aussi du domaine d'application, certains programmes se concevant très naturellement en termes de processus parallèles, impératifs et synchrones. Pour terminer, une des critiques que nous faisions à ESTEREL (ou, du moins à son compilateur actuel) est l'extrême dépendance entre le style de programmation et les performances du code produit. Les transformations formelles de programmes LUSTRE nous font espérer de génerer du code dont les performances dépendent moins du style externe de programmation.

Les principes de LTS sont très proches des notres, mais LTS n'a pas été conçu pour la programmation temps réel. En particulier, les définitions récursives rendent imprédictible le temps d'exécution des programmes. LTS ne permet pas l'asynchronisme.

Il faut mener un peu plus profondément la comparaison avec SIGNAL, qui est un langage contemporain de LUSTRE, fondé sur les mêmes idées et avec les mêmes objectifs. Une des différences entre les deux langages concerne les des contraintes imposées sur les horloges, SIGNAL autorisant, à priori, un programme à recevoir des entrées complètement asynchrones. Ceci étant, il semble qu'il soit impossible d'opérer sur de telles variables et, d'autre part, la signification des opérations sur des variables d'horloges reliées mais différentes n'est pas évidente, ce qui présente des risques d'erreur. La puissance d'expression de LUSTRE est la même que celle de SIGNAL, mais nous avons préféré imposer des contraintes strictes sur les horloges afin de rendre évidente la signification des expressions asynchrones. Notre principale critique de SIGNAL concerne la forme externe du langage, et notamment l'arsenal compliqué des primitives de structuration. A titre d'illustration, nous allons traiter en SIGNAL et en LUSTRE un exemple donné dans [Gautier 85]. Cet exemple est du domaine du traitement du signal et a été choisi pour illustrer l'adéquation de SIGNAL à la résolution de ce genre de problèmes.

Les solutions SIGNAL et LUSTRE sont analogues, les démarches pour y parvenir également. La différence va résider uniquement dans la forme des programmes obtenus.

L'exemple est un algorithme de prédiction linéaire en treillis. Cet algorithme (*Adaptative Gradient Algorithm*) est décrit dans [Makhoul 78] il est donné par les formules suivantes

$$e_t(n+1) = e_t(n) - k_t(n+1) * f_{t-1}(n)$$
$$f_t(n+1) = f_{t-1}(n) - k_t(n+1) * e_t(n)$$
$$e_t(0) = f_t(0) =_\Delta X_t$$
$$k_t(n+1) = cor(k_{t-1}(n+1), e_t(n), f_{t-1}(n))$$
$$k_0(n) =_\Delta 0$$

La fonction **cor**, non spécifiée ici, est un estimateur de la corrélation entre les signaux $e_t(n)$ et $f_{t-1}(n)$.

Solution SIGNAL : Désignons par $zf_t(n+1)$ et $zk_t(n)$ les signaux retardés respectivement $f_{t-1}(n)$ et $k_{t-1}(n)$ (les coefficients k étant considérés comme un signal interne) on obtient les équations :

$$e_t(n+1) = e_t(n) - k_t(n+1) * zf_t(n+1)$$
$$f_t(n+1) = f_t(n+1) - k_t(n+1) * e_t(n)$$
$$zf_t(n+1) = f_{t-1}(n)$$
$$e_t(0) = f_t(0) = X_t$$
$$k_t(n+1) = cor(zk_t(n+1), e_t(n), zf_t(n+1))$$
$$zk_t(n+1) = k_{t-1}(n+1)$$
$$k_0(n) = 0$$

Ces équations peuvent être directement exprimées en SIGNAL, en utilisant, pour chaque valeur de n, les expressions arithmétiques et temporelles suivantes :

$$e := e - k * zf \qquad (R_1)$$
$$f := zf - k * e \qquad (R_2)$$

R_1 est une expression arithmétique qui décrit un processus possédant les entrées e, k, zf et la sortie e (distincte de l'entrée de même nom).

Ces deux processus R_1 et R_2 peuvent alors être superposés (entrées e, k, zf diffusées) en utilisant l'opérateur de composition parallèle.

$$(e := e - k * zf \ \& \ f := zf - k * e) \qquad (R_{12})$$

Ce qui donne le réseau suivant :

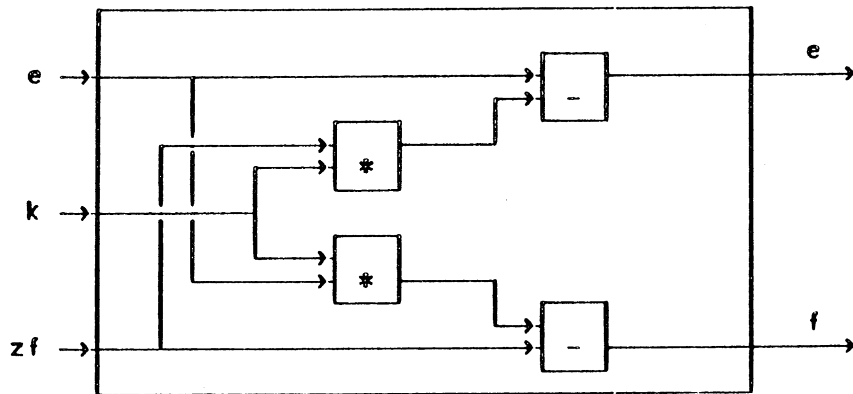


figure 5 : le réseau R_{12}

Le calcul du coefficient k est exprimé par la composition de deux processus :

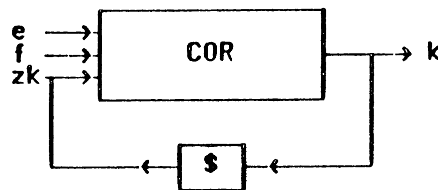


figure 6

Le calcul peut alors être décrit par l'expression suivante :

(zk **is** k **$** 1 | cor) / kz

(où l'opération / désigne le masquage d'un signal de sortie.)

finalement, le système d'équations est décrit par la mise en séquence en SIGNAL des expressions suivantes

> zf **is** f **$** 1;
> (zk **is** k **$** 1 | cor) / kz;
> (e := e - k * zf & f := zf - k * e)

Dans lesquelles on ne laisse visible que les ports de nom e et f. Ce qui donne le réseau suivant :

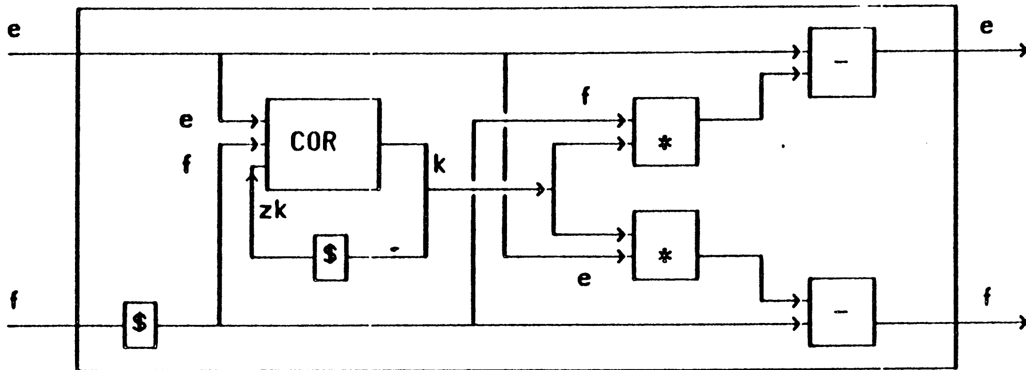


figure 7 : le réseau R_{12}

On décrit l'algorithme en SIGNAL par une déclaration de processus (appelé ORTHOG) utilisant la structure de construction répétitive séquentielle et comportant une partie déclarative décrivant l'élément construit ci-dessus (appelé LATTICE). (Remarque: "f : e" est le renommage de f en e)

Soit le programme suivant :

```
ORTHOG ( { ? real e, ! real f }
        par int n)
doseq i until n of LATTICE od ? f : e / e
where
        process LATTICE ( { ? e, f ! e, f } )
        LATTICE =
        zf is f $ 1;
        (zk is k $ 1 | cor) / kz;
        (e := e - k * zf & f := zf - k * e)
        where
                process signal cor { ? e, zf, zk ! k }
        end % LATTICE %
end % ORTHOG %
```

Cette déclaration correspond au réseau suivant :

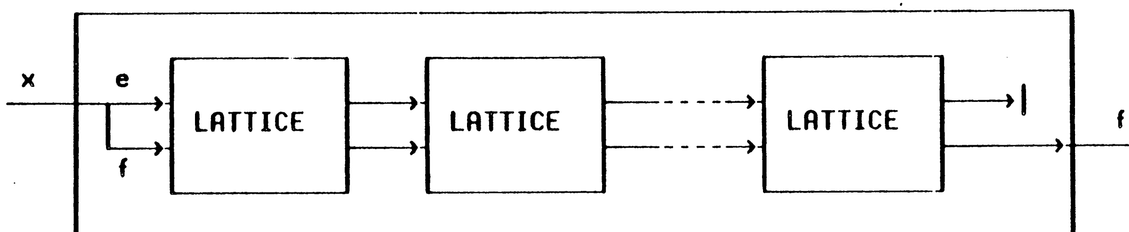


figure 8 : Réseau général

Solution LUSTRE : On part du même système d'équations, que l'on va transformer en un programme LUSTRE par des modifications très simples.

Première étape : supprimer l'indice t (qui représente le temps) en remplaçant toutes les variables qui apparaîssent sous la forme X_{t-1} par **pre** (X), et toutes les définitions qui font référence à l'instant initial par une expression unique en utilisant l'opérateur ->. Ce qui donne le résultat suivant :

Pour tout entier n ∈ [0, N]

```
e (n+1) = e (n) - k (n+1) * pre (f (n))
f (n+1) = pre (f (n)) - k (n+1) * e (n)
e (0) = f (0) = X
k (n+1) = 0 -> cor (pre (k (n+1)), e (n), pre (f (n)))
```

Deuxième étape : écrire le système précédent en respectant la syntaxe du langage.

```
e [0] = X;
f [0] = X;
for n in [0..N-1]
let
        e [n+1] = e [n] - k [n+1] * pre (f [n]);
        f [n+1] = pre (f [n]) - k [n+1] * e [n];
        k [n+1] = 0 -> cor (pre (k [n+1]), e [n], pre (f [n]));
tel;
```

Sachant que le résultat est toujours le terme f [N], on déduit le programme LUSTRE complet suivant :

```
node ORTHOG (const N : int ; X : real) returns (orthog : real) ;
var e, f, k : array [0..N] of real ;
        node cor (zk, e, zf : real) returns (k : int) ;
        let
                -- description de la fonction
        tel;
let
        orthog = f [N];
        e [0] = X;
        f [0] = X;
        for n in [0..N-1]
        let
                e [n+1] = e [n] - k [n+1] * pre (f [n]);
                f [n+1] = pre (f [n]) - k [n+1] * e [n];
                k [n+1] = 0 -> cor (pre (k [n+1]), e [n], pre (f [n]));
        tel;
tel.
```

On peut remarquer que ce programme est une traduction presque directe du système d'équations dont on est parti. C'est un des buts recherchés par LUSTRE que de permettre la production de programmes très proches des spécifications. C'est un des gages de la correction d'un programme. On ne peut pas en dire autant du programme SIGNAL équivalent. Il est à noter que pour tous les problèmes dont la spécification est donnée sous forme d'un système d'équations, la traduction en LUSTRE sera immédiate. Dans le domaine du traitement du signal, il semble être courant que les problèmes soient spécifiés ainsi, et l'on peut se demander si un langage déclaratif tel que LUSTRE ne serait pas plus adéquat pour la programmation de ces applications. La forme de SIGNAL a certainement été choisie par rapport aux habitudes de programmation des automaticiens, alors que nous avons voulu créer LUSTRE en tenant compte de la formulation des spécifications. Cette différence provient peut-être du fait que SIGNAL est un langage créé sur commande, et était soumis à un cahier des charges, alors que LUSTRE a vu le jour à partir d'un modèle de spécification des systèmes temporisés [Halbwachs 84] [Caspi-Halbwachs 86].

2. PERSPECTIVES

2.1. Compilation

Notre objectif primordial, à court terme, est évidemment l'écriture de compilateurs LUSTRE. Un petit prototype a déja été réalisé, qui produit du code ESTEREL selon le principe suivant : à chaque équation LUSTRE est associé un module ESTEREL, qui attend les signaux et les valeurs nécessaires à l'évaluation de la partie droite de l'équation, puis évalue le résultat de cette partie droite et émet un signal porteur du résultat. Une optimisation importante consiste de plus à coder les booléens sous forme de signaux purs. Malgré son caractère expérimental (seul un sous-ensemble de LUSTRE est implémenté), ce prototype nous a permis de constater que les programmes que nous lui avons soumis ne contenaient qu'un nombre minime d'erreurs, ce qui est encourageant quant au mode de programmation autorisé en LUSTRE. Un véritable compilateur est en cours de réalisation, qui produira directement du code C. Ces programmes font un large usage des outils développés autour de CEYX/LE_LISP, à Sophia Antipolis, outils que nous avons pu apprécier. Une part importante du travail restant à faire concerne la sémantique

statique, qui est en cours de formalisation sous forme de règles de réécriture conditionnelles. Le programme de vérification statique s'appuiera sur cette formalisation, soit à l'aide de MENTOR/TYPOL [Clément 85], soit directement en CEYX.

D'autres implémentations doivent être envisagées: Il faut intégrer le code généré à un moniteur temps réel, rythmant les cycles du programme selon une horloge physique, et réalisant l'ordonnancement des tâches non impératives (variables every). Le moniteur devra également être capable de détecter tout débordement du temps de cycle. Enfin, la production de code pour des architectures parallèles doit être étudiée.

2.2. Traitement formel des programmes

Les idées sur l'utilisation de LUSTRE comme modèle formel, que nous avons esquissées dans le chapitre 5 doivent être développées. Plusieurs angles d'approche, et divers types d'application sont à l'étude:

- Manipulations algébriques, telles celles que nous avons illustrées, et dont l'application principale concerne l'optimisation des programmes.

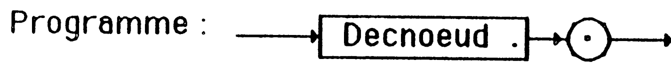
- Traduction de tout ou partie de LUSTRE en logique temporelle linéaire, qui fournirait d'une part, des méthodes automatiques de preuve de programme, et d'autre part une interface plus agréable pour écrire des formules de logique temporelle.

En ce qui concerne les applications, nous étudions actuellement, d'une part l'utilisation de LUSTRE pour décrire et construire des systèmes matériels, et d'autre part la synthèse de programmes LUSTRE à partir des spécifications de l'environnement.

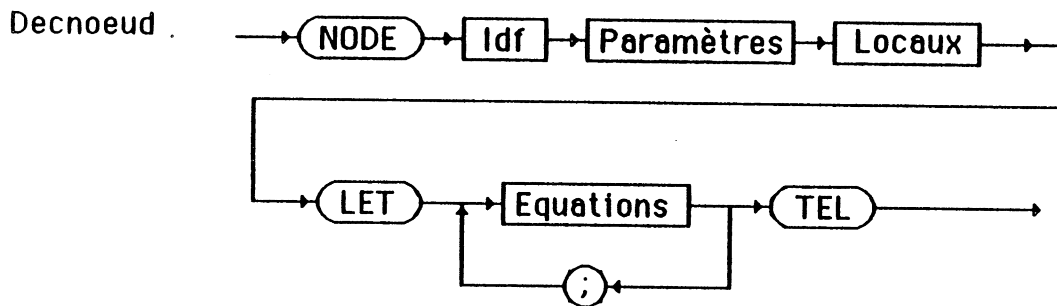
ANNEXE

Carte syntaxique du langage LUSTRE

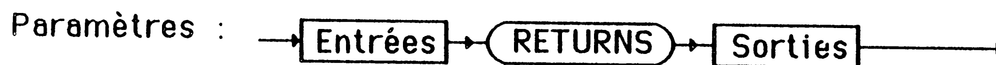
Programme :



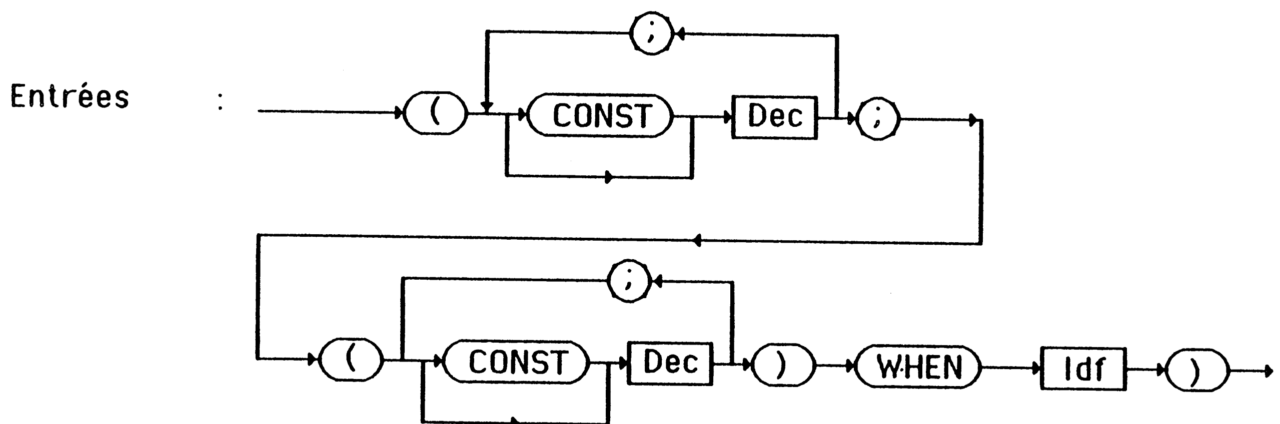
Decnoeud .



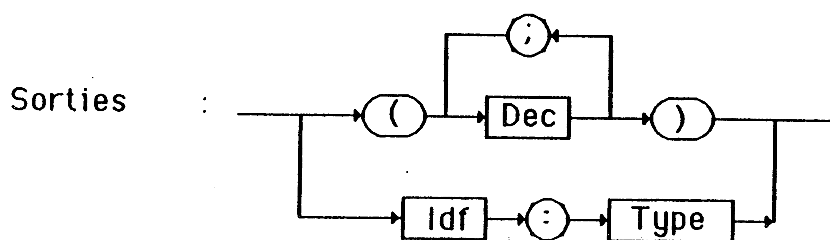
Paramètres :

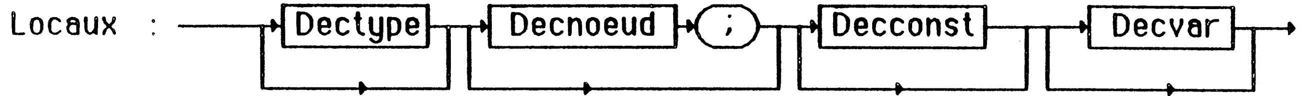


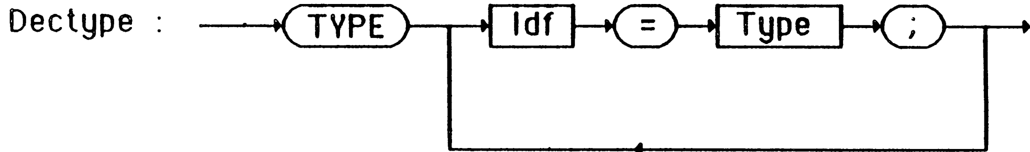
Entrées :

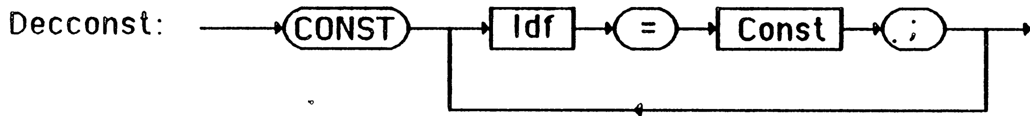


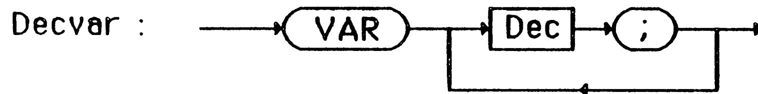
Sorties :

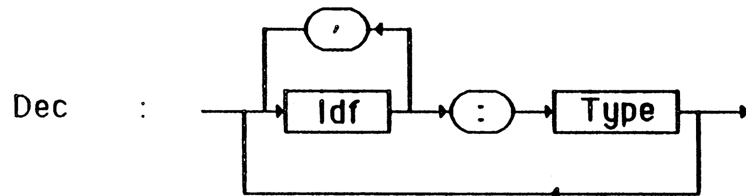


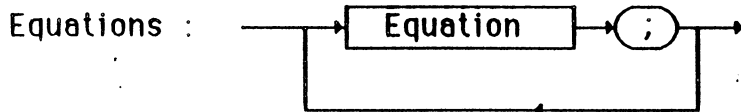

Locaux :
Dectype
Decnoeud
;
Decconst
Decvar

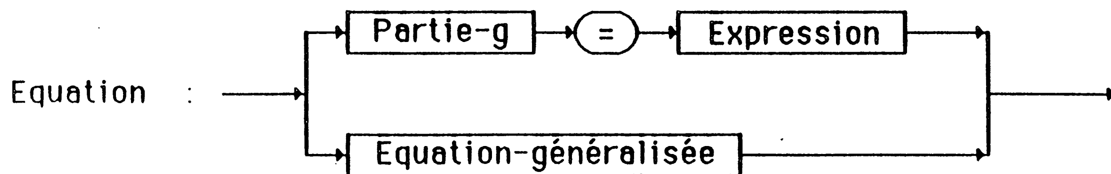

Dectype :
TYPE
Idf
=
Type
;

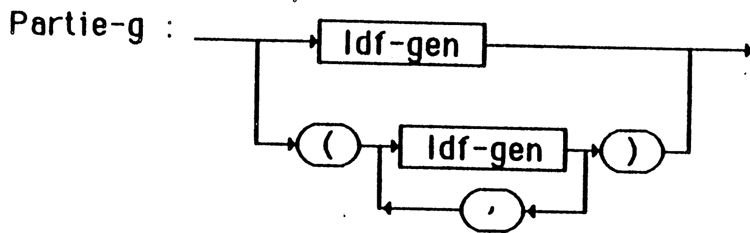

Decconst:
CONST
Idf
=
Const
;

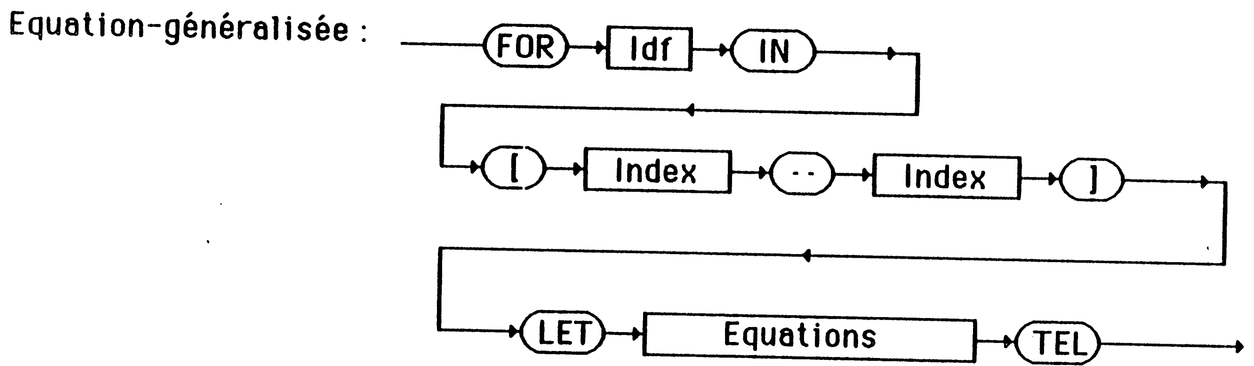

Decvar :
VAR
Dec
;

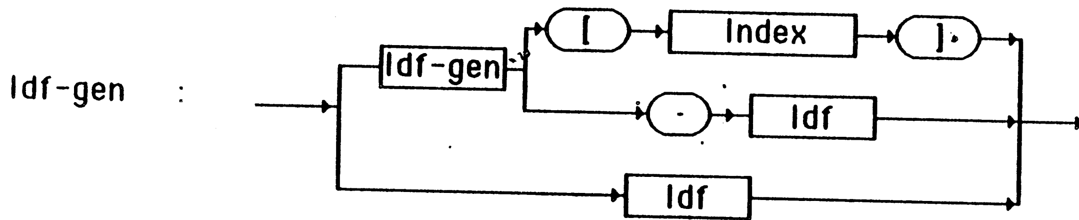

Dec :
,
Idf
:
Type

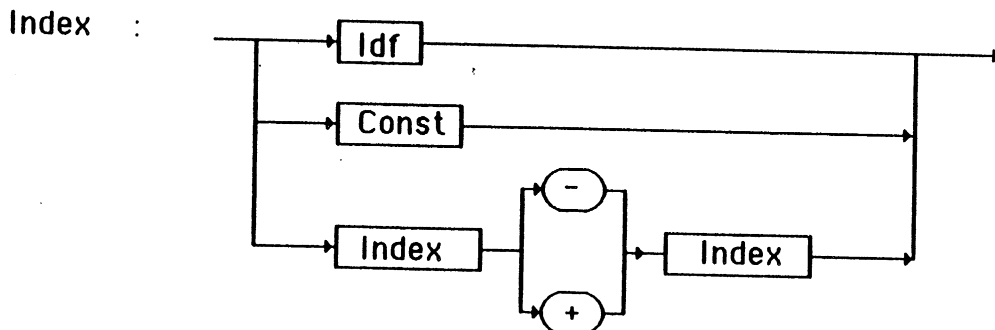

Equations :
Equation
;

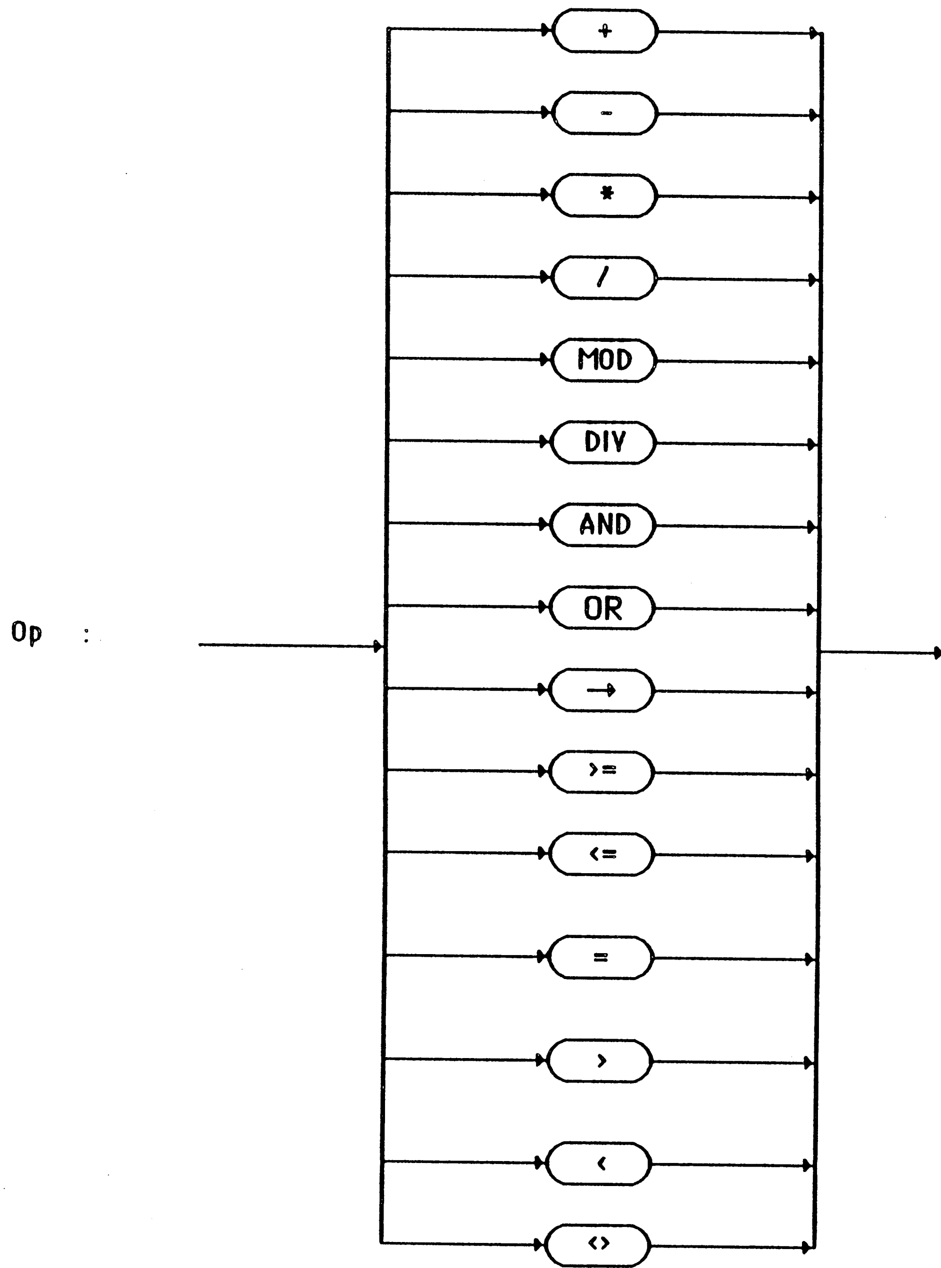

Equation :
Partie-g
=
Expression
Equation-généralisée

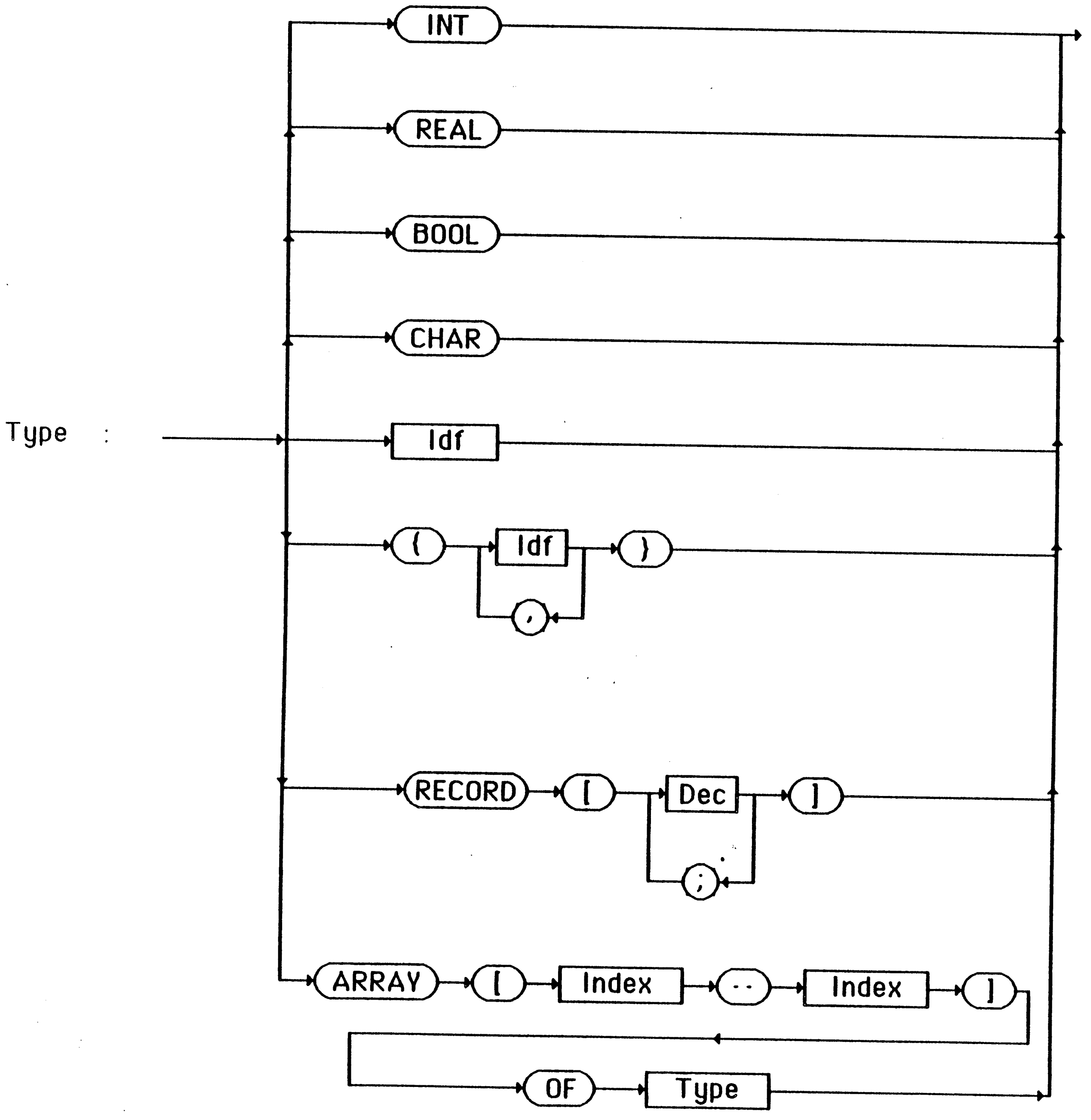

Partie-g :
Idf-gen
(
Idf-gen
)
,

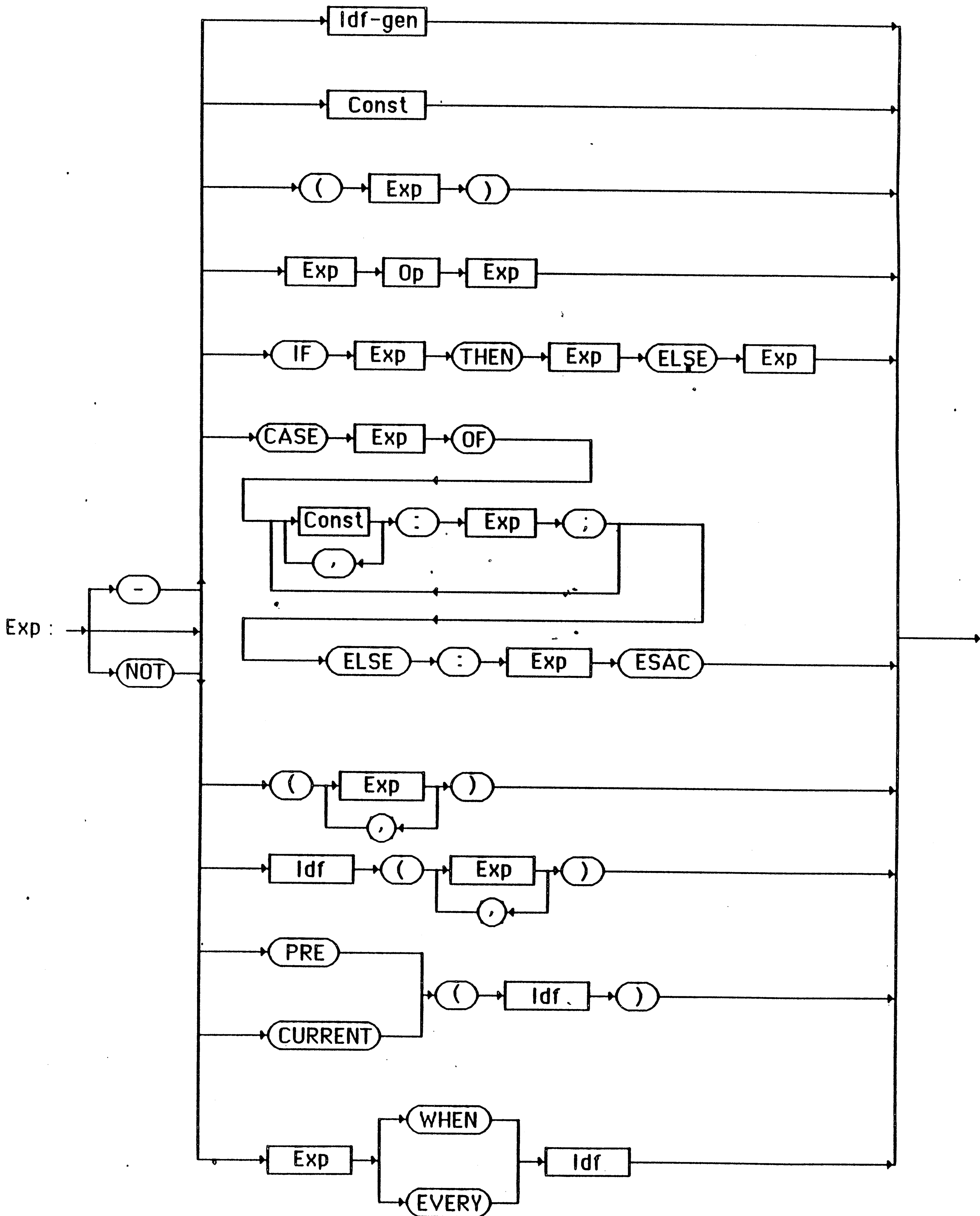

Equation-généralisée :
FOR
Idf
IN
[
Index
..
Index
]
LET
Equations
TEL


Idf-gen :
Idf-gen
[
Index
]
.
Idf
Idf


Index :
Idf
Const
Index
-
+
Index


Op :
+
-
*
/
MOD
DIV
AND
OR
→
>=
<=
=
>
<
<>


Type :
INT
REAL
BOOL
CHAR
Idf
(
Idf
)
,
RECORD
[
Dec
]
;
ARRAY
[
Index
..
Index
]
OF
Type


Exp :
-
NOT
Idf-gen
Const
(
Exp
)
Exp
Op
Exp
IF
Exp
THEN
Exp
ELSE
Exp
CASE
Exp
OF
Const
:
Exp
;
,
ELSE
:
Exp
ESAC
(
Exp
)
,
Idf
(
Exp
)
,
PRE
CURRENT
(
Idf
)
Exp
WHEN
EVERY
Idf

BIBLIOGRAPHIE

[Ackerman 79]

Ackerman W.B.: "Data flow languages". Proc. AFIPS Conf., Arlington, 1979.

[ADA 80]

"Reference manual for the ADA programming language", CII Honeywell 1980.

[Afcet 83]

Groupe Afcet logique : "Pour une représentation normalisée du cahier des charges d'un automatisme logique", Automatique et Informatique industrielles n° 61-62 1977.

[Ashcroft-Wadge 76]

Ashcroft E.A., Wadge W.W.: " LUCID: A formal system for writing and proving programs". SIAM Journal on Computing, 1976.

[Ashcroft-Wadge 77]

Ashcroft E.A., Wadge W.W.: "LUCID: A non procedural language with iteration". CACM, vol. 20, n° 7, juillet 77.

[Austry-Boudol 84]

D. Austry, G. Boudol: "Algèbre de processus et synchronisation". TCS 30 , avril 84.

[Babiker-all 84]

Babiker S.A., Fleming R.A., R.E.Milne: "A tutorial for LTS", RR n° 225.84.1, Standard Telecommunication Laboratories, 1984.

[Backus 78]

Backus J.: "Can Programming be liberated from Von Neumann style? A functional style and its algebra of programs". CACM, vol.21, n° 8, août 78.

[Benveniste 85]

Benveniste A. : "A model to analyse the causality in synchronous real-time system", rapport INRIA n° 411, mai 1985.

[Berger-all 82]

Berger P., Comte D., Hifdi N., Pelois B., Syre J.C.: "Le système LAU : Un multiprocesseur à assignation unique". TSI vol 1, n° 1 janvier-février 1982.

[Bergerand-all 85a]

Bergerand J.L., Caspi P., Halbwachs N.: "Specification and formal validation of distributed systems : the real-time approach", IEE International conference Control 85, Cambridge juillet 85.

[Bergerand-all 85b]

Bergerand J.L., Caspi P., Pilaud D., Pilaud E., Halbwachs N.: "Outline of a real-time data flow language", IEE International symposium on real-time, San Diego décembre 85.

[Berry-all 83]

Berry g., Moisan S., Rigault J.P.: "ESTEREL : Towards a synchronous and semantically sound high level language for real-time applications", Proceedings of th IEEE real-time symposium 1983.

[Berry-Cosserat 84]

Berry G., Cosserat L.: "The ESTEREL synchronous programming language and its mathematical semantics". RR n° 327, INRIA, Sophia Antipolis (France), septembre 84.

[Brams 83]

Brams G.W.: "Reseaux de Petri, Theorie et pratique", Masson ed., Paris, 1983.

[Caspi-Halbwachs 86]

Caspi P., Halbwachs N.: "A functional model for describing and reasoning about time behaviour of computing systems". à paraître dans ACTA-INFORMATICA.

[Caspi-all 82]

Caspi P., Halbwachs N., Pilaud D.: "Implantation fidèle de langages temps reèl". RR n° 315, IMAG, Grenoble, 1982.

[Clément 85]

Clément D., Despeyroux J., Despeyroux T., Hascoet L., Kahn G.: "Natural semantics on the computer", Rapport INRIA n° 416 1985.

[Dennis 74]

Dennis J.B.: "First version of a data flow procedure language". Proc. Colloque sur la programmation, LNCS n° 19, 1974.

[Faustini 82]

A.A. Faustini: "The equivalence of an operational and a denotational semantics of pure data-flow". PhD Thesis, University of Warwick, 1982.

[Halbwachs 84]

Halbwachs N.: "Modelisation et analyse des systemes informatiques temporises". These d'état, Université de Grenoble, juin 84.

[Hankin-Glaser 81]

Hankin C.L., Glaser H.W.: "The data flow programming language CAJOLE. An informal introduction". SIGPLAN Notices 16(7), juillet 81.

[Hoare 74]

Hoare C.A.R.: "Communicating sequential processes". CACM 17(10), 1974.

[Iso 85]

Iso, FDT, CCITT : "ESTELLE : A formal description tecnique based on an extended'state transition model", output from Joint CITT and ISO TC 97/SC 21/WG 16-1 FDT-B, Paris 5-13 février, 1985.

[Kahn 74]

G. Kahn: "The semantics of a simple language for parallel processing". Proc. IFIP Congress, 1974.

[Kung 82]

Kung H.T.: "why systolic architectures ?", Computer 15-1, janvier 82.

[Lecouffe 81]

Lecouffe P.: "SAUGE un langage et un schéma de contrôle pour machine prallèle", bulletin du groupe GROPLAN de l'afcet n° 12 1981.

[Leguernic-all 85]

LeGuernic P., Benveniste A., Bournai P., Gautier T.: "SIGNAL: A data flow oriented language for signal processing". RR n° 246, IRISA, Rennes (France), janvier 85.

[Leverrand 82]

LeVerrand D.: "Le langage ADA, manuel d'évaluation", Afcet-DUNOD (France) 1982.

[McGraw 82]

McGraw J.R.: "The VAL lannguage: Description and analysis". ACM TOPLAS 4(1), janvier 82.

[Makhoul 78]

Makhoul J. : "A class of all-zero lattice digital filters : properties and applications", IEE trans. on Acoust. Speech Signal Processing, vol. 26, août 78.

[Milner 79]

R. Milner: "Flowgraphs and flow algebras", JACM n° 26 vol. 4, octobre 1979

[Milner 80]

R. Milner: "A calculus of communicating system", LNCS n° 92 Springer-Verlag 1980.

[Milner 83]

R. Milner: "Calculi for synchrony and asynchrony". TCS vol. 25, n° 3, juillet 83.

[Moalla-all 78]

Moalla M., Pulou J., Sifakis J.: "Reseaux de Petri synchronises", RAIRO Automatique, vol.12, n° 2, 1978.

[Moalla 81]

Moalla M.: "Spécification et conception sûre d'automatismes discrets complexes basée sur l'utilisation du GRAFCET et des réseaux de Petri" Thèse d'état, université de Grenoble, juillet 81.

[Peterson 77]

Peterson J.L.: "Petri nets", ACM Computing Surveys, vol.9, n° 3, septembre 77.

[Plotkin 81]

Plotkin G.D.: "A structural approach to operational semantics", DAIMI FN 19, Aarhus univ., Denmark, septembre 1981.

[Quinton 83]

Quinton P.: "The sytematic design of systolic arrays", Publication interne IRISA (France) n° 193, mars 83.

[Richier 82]

J.L. Richier: " Preuves de programmes dans un langage sans assignation: LUCID". Thèse de troisième cycle, Universite Pierre et Marie Curie, juin 82.

[Wadge 79]

Wadge W.W. : "An extensional treatment of dataflow deadlock", Proceedings of conference on semantics of concurrent computation, Evian, LNCS Springer-Verlag n° 70 1979.

[Wirth 77]

Wirth N.: "Toward a discipline of real time programming" CACM 20-8 août 78

[Young 82]

Young S.J.: "Real-time languages : design and development" Elis Horwood editeur 1982

INPG département des études doctorales

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 15 titre III de l'arrêté du 5 juillet 1984 relatif aux études doctorales

VU les rapports de présentation de Messieurs

. J.P BANATRE, Professeur
. G. BERRY, Maître de recherche

Monsieur Jean-Louis BERGERAND

est autorisé à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité "Informatique".

Fait à Grenoble, le 18 décembre 1985

Le Président de l'I.N.P.-G

D. BLOCH
Président
de l'Institut National Polytechnique
de Grenoble

Le Vice-Président
de l'I.N.P.-G.