Langages de description de systèmes logiques : propositions pour une méthode formelle de définition

Dominique Borrione

THESE

présentée à

l' Université Scientifique et Médicale de Grenoble

et à

l' Institut National Polytechnique de Grenoble

pour obtenir le grade de

DOCTEUR ES–SCIENCES
«Informatique»

par

Dominique BORRIONE

LANGAGES DE DESCRIPTION DE SYSTEMES LOGIQUES.

Propositions pour une méthode formelle de définition.

Thèse soutenue le 1er Juillet 1981 devant la commission d'examen.

G. VEILLON	Président
F. ANCEAU P. JORRAND R. PILOTY J. VLIETSTRA	Examinateurs

THESE

présentée à

l' Université Scientifique et Médicale de Grenoble

et à

l' Institut National Polytechnique de Grenoble

pour obtenir le grade de

DOCTEUR ES–SCIENCES
«Informatique»

par

Dominique BORRIONE

LANGAGES DE DESCRIPTION DE SYSTEMES LOGIQUES.

Propositions pour une méthode formelle de définition.

Thèse soutenue le 1er Juillet 1981 devant la commission d'examen.

G. VEILLON	Président
F. ANCEAU P. JORRAND R. PILOTY J. VLIETSTRA	Examinateurs

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

Monsieur Gabriel CAU : Président
Monsieur Joseph ~~REIN~~ BLEIN : Vice-Président

MEMBRES DU CORPS ENSEIGNANT DE L'U.S.M.G.

PROFESSEURS TITULAIRES

MM.	AMBLARD Pierre	Clinique de dermatologie
	ARNAUD Paul	Chimie
	ARVIEU Robert	I.S.N.
	AUBERT Guy	Physique
	AYANT Yves	Physique approfondie
Mme	BARBIER Marie-Jeanne	Electrochimie
MM.	BARBIER Jean-Claude	Physique expérimentale
	BARBIER Reynold	Géologie appliquée
	BARJON Robert	Physique nucléaire
	BARNOUD Fernand	Biosynthèse de la cellulose
	BARRA Jean-René	Statistiques
	BARRIE Joseph	Clinique chirurgicale A
	BEAUDOING André	Clinique de pédiatrie et puériculture
	BELORIZKY Elie	Physique
	BARNARD Alain	Mathématiques pures
Mme	BERTRANDIAS Françoise	Mathématiques pures
MM.	BERTRANDIAS Jean-Paul	Mathématiques pures
	BEZES Henri	Clinique chirurgicale et traumatologie
	BLAMBERT Maurice	Mathématiques pures
	BOLLIET Louis	Informatique (I.U.T. B)
	BONNET Jean-Louis	Clinique ophtalmologie
	BONNET-EYMARD Joseph	Clinique hépato-gastro-entérologie
Mme	BONNIER Marie-Jeanne	Chimie générale
MM.	BOUCHERLE André	Chimie et toxicologie
	BOUCHEZ Robert	Physique nucléaire
	BOUSSARD Jean-Claude	Mathématiques appliquées
	BOUTET DE MONVEL Louis	Mathématiques pures
	BRAVARD Yves	Géographie
	CABANEL Guy	Clinique rhumatologique et hydrologique
	CALAS François	Anatomie
	CARLIER Georges	Biologie végétale
	CARRAZ Gilbert	Biologie animale et pharmacodynamie

.../...

MM.	CAU Gabriel	Médecine légale et toxicologie
	CAUQUIS Georges	Chimie organique
	CHABAUTY Claude	Mathématiques pures
	CHARACHON Robert	Clinique ot-rhino-laryngologique
	CHATEAU Robert	Clinique de neurologie
	CHIBON Pierre	Biologie animale
	COEUR André	Pharmacie chimique et chimie analytique
	COUDERC Pierre	Anatomie pathologique
	DEBELMAS Jacques	Géologie générale
	DEGRANGE Charles	Zoologie
	DELORMAS Pierre	Pneumophtisiologie
	DEPORTES Charles	Chimie minérale
	DESRE Pierre	Métallurgie
	DODU Jacques	Mécanique appliquée (I.U.T. I)
	DOLIQUE Jean-Michel	Physique des plasmas
	DREYFUS Bernard	Thermodynamique
	DUCROS Pierre	Cristallographie
	FONTAINE Jean-Marc	Mathématiques pures
	GAGNAIRE Didier	Chimie physique
	GALVANI Octave	Mathématiques pures
	GASTINEL Noël	Analyse numérique
	GAVEND Michel	Pharmacologie
	GEINDRE Michel	Electroradiologie
	GERBER Robert	Mathématiques pures
	GERMAIN Jean-Pierre	Mécanique
	GIRAUD Pierre	Géologie
	JANIN Bernard	Géographie
	KAHANE André	Physique générale
	KLEIN Joseph	Mathématiques pures
	KOSZUL Jean-Louis	Mathématiques pures
	KRAVTCHENKO Julien	Mécanique
	LACAZE Albert	Thermodynamique
	LACHARME Jean	Biologie végétale
Mme	LAJZEROWICZ Janine	Physique
MM.	LAJZEROWICZ Joseph	Physique
	LATREILLE René	Chirurgie générale
	LATURAZE Jean	Biochimie pharmaceutique
	LAURENT Pierre	Mathématiques appliquées
	LEDRU Jean	Clinique médicale B
	LE ROY Philippe	Mécanique (I.U.T. I)

.../...

MM.	LLIBOUTRY Louis	Géophysique
	LOISEAUX Jean-Marie	Sciences nucléaires
	LONGEQUEUE Jean-Pierre	Physique nucléaire
	LOUP Jean	Géographie
Mlle	LUTZ Elisabeth	Mathématiques pures
MM.	MALINAS Yves	Clinique obstétricale
	MARTIN-NOEL Pierre	Clinique cardiologique
	MAYNARD Roger	Physique du solide
	MAZARE Yves	Clinique Médicale A
	MICHEL Robert	Minéralogie et pétrographie
	MICOUD Max	Clinique maladies infectieuses
	MOURIQUAND Claude	Histologie
	MOUSSA André	Chimie nucléaire
	NEGRE Robert	Mécanique
	NOZIERES Philippe	Spectrométrie physique
	OZENDA Paul	Botanique
	PAYAN Jean-Jacques	Mathématiques pures
	PEBAY-PEYROULA Jean-Claude	Physique
	PERRET Jean	Séméiologie médicale (neurologie)
	RASSAT André	Chimie systématique
	RENARD Michel	Thermodynamique
	REVOL Michel	Urologie
	RINALDI Renaud	Physique
	DE ROUGEMONT Jacques	Neuro-Chirurgie
	SARRAZIN Roger	Clinique chirurgicale B
	SEIGNEURIN Raymond	Microbiologie et hygiène
	SENGEL Philippe	Zoologie
	SIBILLE Robert	Construction mécanique (I.U.T. I)
	SOUTIF Michel	Physique générale
	TANCHE Maurice	Physiologie
	VAILLANT François	Zoologie
	VALENTIN Jacques	Physique nucléaire
Mme	VERAIN Alice	Pharmacie galénique
MM.	VERAIN André	Physique biophysique
	VEYRET Paul	Géographie
	VIGNAIS Pierre	Biochimie médicale

.../...

PROFESSEURS ASSOCIES

MM.	CRABBE Pierre	CERMO
	SUNIER Jules	Physique

PROFESSEURS SANS CHAIRE

Mlle	AGNIUS-DELORS Claudine	Physique pharmaceutique
	ALARY Josette	Chimie analytique
MM.	AMBROISE-THOMAS Pierre	Parasitologie
	ARMAND Gilbert	Géographie
	BENZAKEN Claude	Mathématiques appliquées
	BIAREZ Jean-Pierre	Mécanique
	BILLET Jean	Géographie
	BOUCHET Yves	Anatomie
	BRUGEL Lucien	Energétique (I.U.T. I)
	BUISSON René	Physique (I.U.T. I)
	BUTEL Jean	Orthopédie
	COHEN-ADDAD Jean-Pierre	Spectrométrie physique
	COLOMB Maurice	Biochimie médicale
	CONTE René	Physique (I.U.T. I)
	DELOBEL Claude	M.I.A.G.
	DEPASSEL Roger	Mécanique des fluides
	GAUTRON René	Chimie
	GIDON Paul	Géologie et minéralogie
	GLENAT René	Chimie organique
	GROULADE Joseph	Biochimie médicale
	HACQUES Gérard	Calcul numérique
	HOLLARD Daniel	Hématologie
	HUGONOT Robert	Hygiène et médecine préventive
	IDELMAN Simon	Physiologie animale
	JOLY Jean-René	Mathématiques pures
	JULLIEN Pierre	Mathématiques appliquées
Mme	KAHANE Josette	Physique
MM.	KRAKOWIACK Sacha	Mathématiques appliquées
	KUHN Gérard	Physique (I.U.T. I)
	LUU DUC Cuong	Chimie organique - pharmacie
	MICHOULIER Jean	Physique (I.U.T. I)
Mme	MINIER Colette	Physique (I.U.T. I)

.../...

MM.	PELMONT Jean	Biochimie
	PERRIAUX Jean-Jacques	Géologie et minéralogie
	PFISTER Jean-Claude	Physique du solide
Mlle	PIERY Yvette	Physiologie animale
MM.	RAYNAUD Hervé	M.I.A.G.
	REBECQ Jacques	Biologie (CUS)
	REYMOND Jean-Charles	Chirurgie générale
	RICHARD Lucien	Biologie végétale
Mme	RINAUDO Marguerite	Chimie macromoléculaire
MM.	SARROT-REYNAULD Jean	Géologie
	SIROT Louis	Chirurgie générale
Mme	SOUTIF Jeanne	Physique générale
MM.	STIEGLITZ Paul	Anesthésiologie
	VIALON Pierre	Géologie
	VAN CUTSEM Bernard	Mathématiques appliquées

MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

MM.	ARMAND Yves	Chimie (I.U.T. I)
	BACHELOT Yvan	Endocrinologie
	BARGE Michel	Neuro-chirurgie
	BEGUIN Claude	Chimie organique
Mme	BERIEL Hélène	Pharmacodynamie
MM.	BOST Michel	Pédiatrie
	BOUCHARLAT Jacques	Psychiatrie adultes
Mme	BOUCHE Liane	Mathématiques (CUS)
MM.	BRODEAU François	Mathématiques (I.U.T. B) (Personne étrangère habilitée à être directeur de thèse)
	BERNARD Pierre	Gynécologie
	CHAMBAZ Edmond	Biochimie médicale
	CHAMPETIER Jean	Anatomie et organogénèse
	CHARDON Michel	Géographie
	CHERADAME Hervé	Chimie papetière
	CHIAVERINA Jean	Biologie appliquée (EFP)
	COLIN DE VERDIERE Yves	Mathématiques pures
	CONTAMIN Charles	Chirurgie thoracique et cardio-vasculaire
	CORDONNER Daniel	Néphrologie
	COULOMB Max	Radiologie
	CROUZET Guy	Radiologie

.../...

MM.	CYROT Michel	Physique du solide
	DENIS Bernard	Cardiologie
	DOUCE Roland	Physiologie végétale
	DUSSAUD René	Mathématiques (CUS)
Mme	ETERRADOSSI Jacqueline	Physiologie
MM.	FAURE Jacques	Médecine légale
	FAURE Gilbert	Urologie
	GAUTIER Robert	Chirurgie générale
	GIDON Maurice	Géologie
	GROS Yves	Physique (I.U.T. I)
	GUIGNIER Michel	Thérapeutique
	GUITTON Jacques	Chimie
	HICTER Pierre	Chimie
	JALBERT Pierre	Histologie
	JUNIEN-LAVILLAVROY Claude	O.R.L.
	KOLODIE Lucien	Hématologie
	LE NOC Pierre	Bactériologie-virologie
	MACHE Régis	Physiologie végétale
	MAGNIN Robert	Hygiène et médecine préventive
	MALLION Jean-Michel	Médecine du travail
	MARECHAL Jean	Mécanique (I.U.T. I)
	MARTIN-BOUYER Michel	Chimie (CUS)
	MASSOT Christian	Médecine interne
	NEMOZ Alain	Thermodynamique
	NOUGARET Marcel	Automatique (I.U.T. I)
	PARAMELLE Bernard	Pneumologie
	PECCOUD François	Analyse (I.U.T. B) (Personnalité étrangère habilitée à être directeur de thèse)
	PEFFEN René	Métallurgie (I.U.T. I)
	PERRIER Guy	Géophysique-glaciologie
	PHELIP Xavier	Rhumatologie
	RACHALL Michel	Médecine interne
	RACINET Claude	Gynécologie et obstétrique
	RAMBAUD Pierre	Pédiatrie
	RAPHAEL Bernard	Stomatologie
Mme	RENAUDET Jacqueline	Bactériologie (pharmacie)
MM.	ROBERT Jean-Bernard	Chimie-physique
	ROMIER Guy	Mathématiques (I.U.T. B) (Personnalité étrangère habilitée à être directeur de thèse)
	SAKAROVITCH Michel	Mathématiques appliquées

:../...

MM.	SCHAERER René	Cancérologie
Mme	SEIGLE-MURANDI Françoise	Crytogamie
MM.	STOEBNER Pierre	Anatomie pathologie
	STUTZ Pierre	Mécanique
	VROUSOS Constantin	Radiologie

MAITRES DE CONFERENCES ASSOCIES

MM.	DEVINE Roderick	Spectro Physique
	KANEKO Akira	Mathématiques pures
	JOHNSON Thomas	Mathématiques appliquées
	RAY Tuhina	Physique

MAITRE DE CONFERENCES DELEGUE

M.	ROCHAT Jacques	Hygiène et hydrologie (pharmacie)

Fait à Saint Martin d'Hères, novembre 1977

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Année universitaire 1979-1980

Président : M. Philippe TRAYNARD

Vice-Présidents : M. Georges LESPINARD
M. René PAUTHENET

PROFESSEURS DES UNIVERSITES

MM.	ANCEAU François	Informatique fondamentale et appliquée
	BENOIT Jean	Radioéléctricité
	BESSON Jean	Chimie Minérale
	BLIMAN Samuel	Electronique
	BLOCH Daniel	Physique du Solide - Cristallographie
	BOIS Philippe	Mécanique
	BONNETAIN Lucien	Génie Chimique
	BONNIER Etienne	Métallurgie
	BOUVARD Maurice	Génie Mécanique
	BRISSONNEAU Pierre	Physique des Matériaux
	BUYLE-BODIN Maurice	Electronique
	CHARTIER Germain	Electronique
	CHERADAME Hervé	Chimie Physique Macromoléculaires
Mme	CHERUY Arlette	Automatique
MM.	CHIAVERINA Jean	Biologie, Biochimie, Agronomie
	COHEN Joseph	Electronique
	COUMES André	Electronique
	DURAND Francis	Métallurgie
	DURAND Jean-Louis	Physique Nucléaire et Corpusculaire
	FELICI Noël	Electrotechnique
	FOULARD Claude	Automatique
	GUYOT Pierre	Métallurgie Physique
	IVANES Marcel	Electrotechnique
	JOUBERT Jean-Claude	Physique du Solide - Cristallographie
	LACOUME Jean-Louis	Géographie - Traitement du Signal
	LANCIA Roland	Electronique - Automatique
	LESIEUR Marcel	Mécanique
	LESPINARD Georges	Mécanique
	LONGEQUEUE Jean-Pierre	Physique Nucléaire Corpusculaire
	MOREAU René	Mécanique
	MORET Roger	Physique Nucléaire Corpusculaire
	PARIAUD Jean-Charles	Chimie - Physique
	PAUTHENET René	Physique du Solide - Cristallographie
	PERRET René	Automatique

.../...

MM.	PERRET Robert	Electrotechnique
	PIAU Jean-Michel	Mécanique
	PIERRARD Jean-Marie	Mécanique
	POLOUJADOFF Michel	Electrotechnique
	POUPOT Christian	Electronique - Automatique
	RAMEAU Jean-Jacques	Chimie
	ROBERT André	Chimie Appliquée et des matériaux
	ROBERT François	Analyse numérique
	SABONNADIERE Jean-Claude	Electrotechnique
Mme	SAUCIER Gabrielle	Informatique fondamentale et appliquée
M.	SOHM Jean-Claude	Chimie - Physique
Mme	SCHLENKER Claire	Physique du Solide - Cristallographie
MM.	TRAYNARD Philippe	Chimie - Physique
	VEILLON Gérard	Informatique fondamentale et appliquée
	ZADWORNY François	Electronique

CHERCHEURS DU C.N.R.S. (Directeur et Maître de Recherche)

M.	FRUCHART Robert	Directeur de Recherche
MM.	ANSARA Ibrahïm	Maître de Recherche
	BRONOEL Guy	Maître de Recherche
	CARRE René	Maître de Recherche
	DAVID René	Maître de Recherche
	DRIOLE Jean	Maître de Recherche
	KAMARINOS Georges	Maître de Recherche
	KLEITZ Michel	Maître de Recherche
	LANDAU Ioan-Doré	Maître de Recherche
	MERMET Jean	Maître de Recherche
	MUNIER Jacques	Maître de Recherche

Personnalités habilitées à diriger des travaux de recherche (décision du Conseil Scientifique)

E.N.S.E.E.G.

MM.	ALLIBERT Michel	
	BERNARD Claude	
	CAILLET Marcel	
Mme	CHATILLON Catherine	
MM.	COULON Michel	
	HAMMOU Abdelkader	
	JOUD Jean-Charles	
	RAVAINE Denis	
	SAINFORT	C.E.N.G.

.../...

MM. SARRAZIN Pierre
SOUQUET Jean-Louis
TOUZAIN Philippe
URBAIN Georges — Laboratoire des Ultra-Réfractaires ODEILLO

E.N.S.M.E.E.

MM. BISCONDI Michel
BOOS Jean-Yves
GUILHOT Bernard
KOBILANSKI André
LALAUZE René
LANCELOT Françis
LE COZE Jean
LESBATS Pierre
SOUSTELLE Michel
THEVENOT François
THOMAS Gérard
TRAN MINH Canh
DRIVER Julian
RIEU Jean

E.N.S.E.R.G.

MM. BOREL Joseph
CHEHIKIAN Alain
VIKTOROVITCH Pierre

E.N.S.I.E.G.

MM. BORNARD Guy
DESCHIZEAUX Pierre
GLANGEAUD François
JAUSSAUD Pierre
Mme JOURDAIN Geneviève
MM. LEJEUNE Gérard
PERARD Jacques

E.N.S.H.G.

M. DELHAYE Jean-Marc

E.N.S.I.M.A.G.

MM. COURTIN Jacques
LATOMBE Jean-Claude
LUCAS Michel
VERDILLON André

Je tiens à remercier

Monsieur le Professeur **Gérard VEILLON**, *Directeur de l'E.N.S.I.M.A.G., qui a bien voulu me faire l'honneur de présider le jury de cette thèse. Je n'oublie pas que c'est au sein de son équipe qu'en D.E.A., j'ai effectué mes premiers pas dans la recherche.*

Monsieur le Professeur **François ANCEAU**, *qui s'était déjà intéressé à mon travail de troisième cycle, et avec qui j'ai eu au cours des années des échanges de vues très cordiaux.*

Monsieur **Philippe JORRAND**, *Maître de Recherches au C.N.R.S., qui a accepté de diriger cette thèse, et dont les suggestions m'ont grandement aidée à clarifier le modèle exposé en quatrième partie de ce mémoire.*

Monsieur le Professeur **Robert PILOTY**, *Vice-Président de l'I.F.I.P. et Président du groupe de travail CONLAN, grâce à qui j'ai pu participer à un projet de recherche passionnant. Qu'il reçoive ici l'expression de toute ma reconnaissance pour les discussions nombreuses et animées qui m'ont permis de mettre au point les principales idées présentées dans cet ouvrage, et pour son soutien généreux et répété depuis cinq ans.*

Monsieur **Jakob VLIETSTRA**, *précédent Président du groupe de travail WG 5.2 de l'I.F.I.P., et Conseiller auprès de la Direction du groupe Philips, qui fut le promoteur d'une collaboration fructueuse avec les premiers utilisateurs industriels de nos réalisations, et dont la confiance ne s'est jamais démentie.*

A cause de tout ce qui à présent nous lie, il ne sera pas donné au véritable initiateur de ce travail d'être membre du jury appelé à en juger le résultat. **Jean MERMET** *m'a accueillie dans son équipe, et m'a donné les moyens de mener à bien ma recherche. Sa thèse sur CASSANDRE fut à la base de ma réflexion, et sa renommée fut ma chance de départ. Qu'en lui dédiant cette thèse il me soit permis d'évoquer ce que je lui dois.*

Je tiens également à exprimer ma très vive gratitude à tous ceux avec qui j'ai travaillé, et qui m'ont fait bénéficier de leur expérience ou de leur aide.

J'ai beaucoup appris, au cours des réunions du groupe CONLAN, dont les autres membres sont **Mario BARBACCI, Donald DIETMEYER, Frederick HILL** *et* **Pat SKELLY.** *Je leur dois un éclairage complémentaire sur les problèmes de description des systèmes logiques, beaucoup de patience, et la démonstration qu'un groupe dispersé peut s'attaquer avec succès à un projet d'envergure.*

Parmi mes collègues de l'I.M.A.G., nombreux sont ceux avec qui j'ai eu des discussions qui ont eu un impact sur mon travail. Je voudrais citer notamment **Didier BERT, Michel DELAUNAY, Pedro GUERREIRO** *et* **Paul JACQUET** *de l'équipe Conception et Mise en Oeuvre de Langages,* **Yvon BRESSY, Bertran DAVID, Claude LE FAOU** *et* **Jean-Claude REYNAUD** *de l'équipe Communication Graphique et Méthodologie de la C.A.O.*

Je tiens à souligner tout particulièrement l'apport de **Jean-François . GRABOWIECKI,** *qui a réalisé la première mise en oeuvre de LASSO. Qu'il trouve ici, pour son aide efficace et ses qualités de coeur, l'expression de toute ma reconnaissance.*

Je voudrais aussi remercier tous les membres, présents et passés, de l'équipe Communication Graphique et Méthodologie de la C.A.O., pour l'ambiance amicale qu'ils ont toujours maintenue.

Enfin je tiens à remercier Madame **Josiane CARRY,** *qui a dactylographié cette thèse avec beaucoup de soin et de gentillesse, et en un temps particulièrement court.*

Tout le personnel du Service de Reprographie, qui a réalisé le tirage de cet ouvrage.

A mon mari

LANGAGES DE DESCRIPTION DE SYSTEMES LOGIQUES

PROPOSITIONS POUR UNE METHODE FORMELLE DE DEFINITION

TABLE DES MATIERES

PREMIERE PARTIE : INTRODUCTION 11

Les langages de description de systèmes logiques 13
Niveaux de description 14
La situation actuelle 16
Les outils de vérification 17
Nécessité d'un système intégré d'aide à la conception 19
Notre démarche 20

DEUXIEME PARTIE : UN EXEMPLE : LE LANGAGE LASSO 23

I Présentation de LASSO 25

II La modularité en LASSO 26
II.1 Notions d'entité et d'unité 26
a) Définition externe ou locale d'une entité 26
b) Déclaration d'unité 28
II.2 Structure d'une description d'entité 29
a) L'interface 29
b) Le corps 30
c) Les spécifications 31

III Communications entre unités 32
III.1 Définition de l'interface 32
a) Les données 33
b) Les signaux 33
c) Association signal-donnée 34
III.2 Les instructions de connexion 36
a) Connexion de signaux seuls 38
b) Connexion de signaux à messages 38
c) Connexion à retard 39
d) Connexion multiple 41

IV Synchronisation et contrôle à l'intérieur d'une unité 41
IV.1 Partie opératoire 41
IV.2 Partie contrôle 42
IV.3 Exemple complet de la description d'une mémoire 45
IV.4 Transitions composées 46

V Types et opérateurs primitifs 48
V.1 Types de données et constructeurs primitifs 48
a) booléens 48
b) entiers 49
c) caractères 49
d) structures 49
e) tableaux 49
V.2 Les porteuses du langage 50
a) Les variables 50
b) Les signaux 51
c) Les noms de signaux : nomsig 51

V.3 Expressions 53
V.4 Procédures et fonctions 53
a) Portée et visibilité des variables 53
b) Définition locale ou externe d'une procédure ou d'une fonction
c) Corps de procédure ou de fonction 55

VI Instructions de la partie algorithmique 56
VI.1 L'instruction d'affectation 56
VI.2 L'instruction conditionnelle 56
VI.3 Les instructions d'itération 56
VI.4 Les instructions d'itération conditionnelle 57
VI.5 Appel de procédure ou de fonction 58

VII Les spécifications 58
VII.1 Les prédicats simples 59
VII.2 Expressions de séquence 60
a) Notion de séquence et de vecteur temporel 60
b) Expression de séquence 61
VII.3 Expressions d'évènements 62
a) L'opérateur puis 62
b) L'opérateur commesi 63
c) L'opérateur etdeplus 63
d) L'opérateur lafoisno 64
e) L'opérateur retard 64
f) Priorités des opérateurs d'évènements 64
VII.4 Les prédicats composés 65
a) Détermination des fenêtres de validité 66
b) Détermination des instants de validité 68
c) Ecriture des prédicats composés 69
d) Exemple 71

TROISIEME PARTIE : FORMALISATION DES PRINCIPES COMMUNS AUX LANGAGES DE DESCRIPTION DES SYSTEMES LOGIQUES : L'APPROCHE CONLAN

I Concepts de base et terminologie 75

I.1 Valeurs et porteuses 75

I.2 Parallélisme et séquencement 77

I.3 Notions temporelles 79

I.4 Opérateurs abstraits et opérateurs matériels 80

II Les choix de CONLAN 83

II.1 Objectifs du projet et approche choisie 83

II.2 Notions communes à tous les langages de CONLAN 86

a) Notations 86

b) Modèle du temps 86

c) Notion de segment 87

II.3 Fonctions et Activités 88

a) Fonctions 89

b) Activités 89

c) Opérations statiques, opérations dynamiques 90

d) Passages de paramètres 92

d.1) Désignations 92

d.2) Liaisons 93

d.3) Droits d'accés 94

e) Le qualificatif INTERPRETERà 97

II.4 Les types et les classes 98

a) Domaine d'un type 98

b) Opérations d'un type 100

b.1) Opérations transportées 100

b.2) Opérations définies explicitement 101

b.3) Exemple 102

c) Relations entre types 104

c.1) Sous-type 104

c.2) Type dérivé 104

c.3) Relation de partage de définition et dérivation canonique

c.4) Utilité pratique de ces relations 105

d) Notion de classe 107

II.5 Les descriptions 109
a) Les attributs 111
b) L'interface 112
c) Les objets internes et leurs intercommunications 113
II.6 Segment définissant un langage 115
a) Définition des primitives 115
b) Modifications syntaxiques 117
II.7 Règles portant sur les segments 119
a) Règles d'imbrication des définitions de segments 119
b) Règles de visibilité 120
II.8 Les instructions primitives 121
a) Les prédicats 122
b) Les formes conditionnelles 122
b.1) La forme IF 122
b.2) La forme CASE 123
c) Répétition 123

III Dérivation de BASE CONLAN à partir de CONLAN PRIMITIF 124
III.1 Les types et opérateurs de CONLAN PRIMITIF 124
a) bool 124
b) int 124
c) string 125
d) cellà 125
e) univà 126
f) tupleà 126
g) anyà 127
h) priorité des opérateurs 127
III.2 Principes de construction de BASE CONLAN 128
a) Les types utilitaires 128
b) Dérivation des tableaux 132
c) Dérivation des enregistrements 134
d) Historiques de valeurs 136
e) Porteuses de BASE CONLAN 137

ANNEXE DE LA TROISIEME PARTIE 141

QUATRIEME PARTIE : UN MODELE D'EVALUATION POUR LES LANGAGES DE DESCRIPTION DE SYSTEMES LOGIQUES

I Introduction 153

II Présentation du modèle 156
- II.1 Les agents 157
- II.2 Les messages 158
  - a) Arrivée des messages 159
  - b) Forme des messages 159
- II.3 Notion de contexte 160
- II.4 Présentation intuitive du fonctionnement 161
  - a) Correspondance entre éléments d'une description et éléments du modèle 161
  - b) Modes de transmission des messages 162

III Le modèle d'évaluation 165
- III.1 Les objets 165
- III.2 Les messages 166
  - a) Catégories de messages 166
  - b) Représentation de l'arrivée des messages 169
- III.3 Définition du comportement d'un agent 170
  - a) Squelette de message 170
  - b) Evaluation des noms formels 171
    - b.1) Noms d'agents 171
    - b.2) Paramètres typés 171
  - c) Réception et émission 172
  - d) Formes conditionnelles 173
    - d.1) Test d'un nom formel 174
    - d.2) Test d'un message 174
    - d.3) Règles communes aux deux formes 175
  - e) Instructions spéciales 175
    - e.1) Initialisation 175
    - e.2) Renvoi 175
    - e.3) Arrêt de la tâche 176
    - e.4) Définition de macro 176
    - e.5) Définition de tâche 177
  - f) Exécution d'une tâche 178
    - f.1) Reconnaissance d'un message reçu 178
    - f.2) Traitement des erreurs 179

III.4 Les catégories d'agents et de tâches 180
a) Gestion de tiroirs et magasiniers 181
b) Tâche d'évaluation d'une fonction 183
b.1) Initialisation 184
b.2) Evaluation et liaison des paramètres 184
b.3) Vérification d'assertions sur les paramètres 186
b.4) Embauche des magasiniers locaux 186
b.5) Evaluation du corps de fonction 186
b.6) Evaluation du résultat 187
b.7) Renvoi des agents locaux intérimaires et réponse 187
b.8) Exemple 188
c) Tâche d'évaluation d'une activité 189
c.1) Etape EVAL-EVALUATED 190
c.2) Etape START-DONE 190
c.3) Schéma général d'une tâche d'évaluation d'activité
d) Tâche d'évaluation d'une description 192
d.1) Initialisation 192
d.2) Etape EVAL-EVALUATED 195
d.3) Etape STARD-DONE 195

IV Application à CONLAN 196
IV.1 Les objets et agents primitifs 196
a) L'univers des objets 196
b) Les cellules et les opérations primitives sur les cellules
b.1) Tâche associée à la fonction getà 197
b.2) Tâche associée à la fonction cell-typeà 197
b.3) Tâche associé à la fonction emptyà 197
b.4) Tâche associée à l'activité putà 198
c) Les fonctions sur les types primitifs autres que cellà 198
d) Quantificateurs et sélecteur 199
e) Expressions conditionnelles 201
IV.2 Traduction dans le modèle d'un segment écrit en CONLAN PRIMITIF
a) Traduction d'un segment CONLAN 202
b) Traduction d'un segment CLASS 203
c) Traduction d'un segment TYPE 203
d) Traitement des attributs d'un segment DESCRIPTION, FUNCTION ou ACTIVITY 203
e) Traduction d'un segment FUNCTION ou ACTIVITY 204
f) Traduction d'un segment DESCRIPTION 205

IV.3 Traduction dans le modèle d'un segment écrit en BASE CONLAN 208
a) Modèle du temps et stabilisation 208
b) Compteurs globaux et locaux 209
b.1) Compteurs globaux de temps et de pas d'évaluation 209
b.2) Compteurs locaux de pas d'évaluation 210
c) Production des paramètres des opérations 210
d) Traduction d'un segment FUNCTION ou ACTIVITY 211
e) Evaluation d'une description écrite en BASE CONLAN 215

CINQUIEME PARTIE : APPLICATIONS 217

I Introduction 219

II Définition en CONLAN d'un langage de niveau LASCAR 220
II.1 Présentation 220
II.2 Définition formelle de LASCAR2 225
II.3 Définition en CONLAN d'un langage de niveau CASSANDRE 235

III Définition en CONLAN d'un langage de niveau LASSO 236
III.1 Présentation 236
III.2 Définition formelle de LASSO2 238

IV Une autre méthode de dérivation 253
IV.1 BCL-ETENDU 254
IV.2 LASCAR2 et LASSO2 254
IV.3 Définitions formelles 254

SIXIEME PARTIE : CONCLUSION 265

BIBLIOGRAPHIE 271

PREMIERE PARTIE

INTRODUCTION

LES LANGAGES DE DESCRIPTION DE SYSTEMES LOGIQUES

En 1974, Yaohan Chu, auteur d'un des premiers langages de description de systèmes logiques ayant connu un succès international, faisait la remarque suivante : [Chu.74] :

> Les langages de description du matériel informatique sont, au même titre que le symbolisme mathématique, un moyen d'exprimer des abstractions à l'aide de symboles. A ce titre, ils sont aussi indispensables à la conception des ordinateurs que le symbolisme mathématique est à la résolution des équations différentielles.

C'est dans la perspective de l'aide à la conception des systèmes informatiques que se situe notre travail. Par conception, nous entendons ici bien plus que ce que recouvre le mot anglais "design"; nous englobons les deux sens figurés donnés par le Petit Larousse :

- faculté de comprendre
- production de l'intelligence.

Définis dans ce contexte bien spécialisé, les langages de description de systèmes logiques permettent une précision et une concision hors de portée d'une langue "naturelle" ou d'une représentation graphique. En tant qu'aide à la compréhension, ces deux caractéristiques font de ce mode de description :

1) un support privilégié pour l'enseignement de la structure des ordinateurs, ce dont témoigne le fait que des manuels particulièrement renommés outre Atlantique s'appuient très largement sur un de ces langages [Die.78, Chu.72, HiP.78];

2) un moyen de communiquer entre équipes travaillant sur des sous-ensembles distincts, ou sur des aspects complémentaires d'un même projet;

3) une documentation d'un projet d'autant plus utile qu'elle est susceptible de traitements automatisés, et d'une mise à jour continue.

Notations symboliques d'une relative simplicité, les langages de description de systèmes logiques sont avant tout définis en vue d'une interprétation informatisée. Les outils logiciels qui leur sont associés, aides à la conception au second sens de ce terme, s'orientent autour de deux objectifs principaux :

1) permettre des validations au cours des différentes phases de l'étude (par simulation ou vérifications formelles);

2) automatiser, partiellement ou totalement, la préparation des tests, des documents de fabrication (implantation des circuits, listes de câblage, masques de circuits intégrés) et des données d'entrée pour des machines à commande numérique.

Les programmes de ce second groupe ont une application d'autant plus directe et aisée que l'on se trouve dans une phase proche de la réalisation d'un projet. Souvent très liés à une technologie ou à un processus de fabrication, ils sont largement répandus en milieu industriel.

Nous avons, pour notre part, centré nos travaux autour du premier objectif, nous attachant à offrir au concepteur des langages généraux lui permettant de décrire son projet avec divers degrés d'abstraction.

NIVEAUX DE DESCRIPTION

Il est d'usage en effet de distinguer plusieurs niveaux de description des systèmes logiques, de 4 à 6 selon les auteurs [Mer.73, Bar.75, Shi.79]. Nous considérons quant à nous qu'il existe un nombre a priori aussi grand que l'on veut, dans la mesure où une description peut être progressivement détaillée, et où certaines parties d'un système logique peuvent être définies avec précision, et cohabiter avec d'autres parties plus grossièrement décrites au sein d'un même modèle. Pour les besoins de l'exposé, nous nous rallierons à 6 niveaux typiques, qui sont, du plus global au plus détaillé :

1 - Le niveau architecture système, qui découpe un système en tâches et fonctions à réaliser, sans préciser nécessairement les choix de mise en oeuvre (programmée, microprogrammée ou câblée).

2 - Le niveau architecture matérielle, qui fait apparaître la structuration du système en ses principaux modules matériels (mémoires, processeurs, périphériques, chemins de transfert d'information).

3 - Le niveau instructions et microprogrammes, qui définit l'interprétation du langage machine.

4 - Le niveau transfert de registre, qui décrit la micromachine, et les règles de transfert d'information entre éléments de mémorisation, sans définir tous les chemins de données.

5 - Le niveau circuit combinatoire, qui décrit un système comme un réseau de portes et de bascules, le plus souvent dans des logiques à plusieurs états.

6 - Le niveau circuit électronique, dans lequel les éléments du système sont des résistances, diodes, capacités, transistors.

Le premier niveau correspond à la phase de spécification d'un système logique. Les vérifications à effectuer concernant les problèmes d'interaction et de synchronisation entre tâches, de déterminisme, d'absence de blocage etc ... . Au second niveau, des choix de répartition matérielle et la fixation de paramètres globaux conduisent à des évaluations de performances de l'architecture retenue. Au niveau trois, la logique des microprogrammes est vérifiée; la description des échanges entre modules matériels, et des protocoles de communication, permet de procéder à des évaluations de performances fines, et de faire apparaître d'éventuels conflits d'accès à des ressources matérielles. La logique détaillée est validée au niveau transfert de registre. Les contraintes temporelles et les aléas sont pris en compte aux niveaux transfert de registre, mais surtout circuit combinatoire. C'est aussi au niveau 5 que sont étudiés les problèmes de détection de pannes, et de génération de séquences de test.

Enfin, le niveau circuit électronique est utilisé pour des simulations de fonctionnement et des optimisations, mais de nature différente : contrairement aux 5 premiers niveaux qui permettent une modélisation de valeurs et de phénomènes considérés comme discrets, le niveau 6 modélise des phénomènes continus (courants, tensions).

Nous ne méconnaissons ni l'importance du niveau électronique pour la conception de circuits intégrés, ni la difficulté des problèmes de modélisation qui s'y posent, ni surtout la nécessité de pouvoir traiter des descriptions hybrides, c'est-à-dire faisant cohabiter des modules logiques et des modules analogiques. Mais, d'une part parce que ces problèmes sont abordés par d'autres personnes au sein de l'équipe "Communication Graphique et Méthodologie de la C.A.O.", d'autre part parce que ce niveau fait appel à des techniques de description et de résolution bien spécifiques [Lef.74, LeR.77], nous n'en traiterons pas dans cet ouvrage.

Le domaine que nous avons cherché à couvrir s'étend du niveau 1 au niveau 5. Les systèmes logiques y sont décrits par des modèles discrets, structurés en un nombre de modules fixé statiquement.

Les premiers langages de description, et les premiers simulateurs sont apparus aux niveaux 4 et 5, dans les années soixante. Puis, avec le développement des techniques d'intégration, la complexité croissante des circuits intégrés a rendu trop coûteuse et inefficace la simulation d'un seul composant de type microprocesseur au niveau le plus détaillé, sans même parler d'un système complet; et la nécessité que nous avions déjà soulignée dans [Bor.76] de disposer de plusieurs niveaux de modélisation, d'un ensemble cohérent de langages de description couvrant ces niveaux, et des programmes de traitement associés, est maintenant largement reconnue par la communauté des concepteurs. C'est dans cet optique que nous avons conduit nos travaux.

Les travaux précédemment menés à l'I.M.A.G., matérialisés par le système CASSANDRE couvraient les niveaux microprogramme, transfert de registre et circuits combinatoires. Partant de cet acquis, nous avons cherché à définir les primitives adaptées à la description des autres niveaux. Le niveau instructions et architecture matérielle détaillée est celui de LASCAR [Bor.76], dont les programmes de traitement sont depuis plus de cinq ans distribués à l'extérieur du Laboratoire, et ont déjà été utilisés pour des projets réels. Le langage LASSO, élaboré plus récemment, se place aux niveaux architecture système et architecture matérielle non détaillée [BoG.79]. Le compilateur et le simulateur de ce langage [BoG.80, Gra.81] en sont encore au stade de prototypes, mais nous ont déjà permis de tester les nouvelles idées du langage, en vue de leur extension aux autres niveaux au sein d'un projet plus vaste [CAD.2]. Le langage LASSO est présenté, de manière volontairement intuitive, dans la deuxième partie de ce mémoire, à titre d'exemple de langage de description de systèmes logiques, et de la manière traditionnelle de les définir.

LA SITUATION ACTUELLE

La situation dans le domaine de la description et de l'aide à la conception des systèmes logiques se caractérise par une grande diversité de langages et de logiciels. Cette prolifération de langages, dont bien peu sont reconnus comme réellement novateurs, est expliquée par Lipovski [Lip.77] comme résultant du succès de la simulation en tant que méthode d'analyse et de vérification, et du fait que

"Quiconque écrit un simulateur se croît autorisé à définir son propre langage".

Le lecteur trouvera dans [CHDL.1-5] l'illustration de cet effet de "tour de Babel". Citons, à chaque niveau, quelques exemples typiques :

- Architecture système : LOGOS, ADLIB, SARA
- Architecture matérielle : ADL, SLIDE, PMS, SAMEN/SAMO
- Instructions, microprogrammes : CAP, LCD, ISP, MIMOLA
- Transfert de registre : CDL, DDL, AHPL, OSM, RTS, ERES, LOGAL
- Circuit combinatoire et réseau de porte : TEGAS, HILO, LOGSIM, EPISODE, SILEX.

A ce grand nombre de langages correspond un nombre égal de combinaisons de primitives et de symbolismes, pour exprimer la syntaxe, les opérateurs, la modularité, le temps, le contrôle, le déroulement parallèle ou séquentiel des instructions etc ... . Le tableau 1 donne un aperçu de cette variété sur un petit nombre d'exemples.

LES OUTILS DE VERIFICATION

La simulation a été la première méthode de vérification d'un projet de système logique, et elle demeure le plus répandue. Selon les niveaux, la simulation permet des vérifications de performance, de bon fonctionnement logique, de bonne synchronisation, d'absence d'aléas dûs à des retards trop importants dans la traversée de certaines parties d'un circuit, de détection de défauts. La simulation exhaustive d'un système logique, même de la complexité d'un seul circuit LSI actuel, est inenvisageable, pour des raisons de coût. Aussi, la simulation, si elle permet de trouver un grand nombre d'erreurs, ne garantit pas que toutes ont pu être détectées. Ainsi, sur l'ordinateur M200 de Hitachi, 10% du nombre total d'erreurs de conception n'ont été découvertes que sur la machine réelle [Ish.80].

Toutefois, dès que se posent des problèmes de tolérance, d'aléas, de performances détaillées, et de façon générale dès que l'intervention des problèmes temporels devient critique, la simulation est la seule technique de vérification applicable avant le test en vraie grandeur [SzT.76]. C'est pourquoi chaque année de nouveaux algorithmes et de nouveaux logiciels sont présentés dans cette spécialité [DAC.75-80].

	Reference	Origine	Modularité	Existence de la notion d'horloge	Asynchrone	Procedural	Primitives de contrôle
ADLIB	Hil.79	PASCAL	TYPE	non	oui	oui	WAITFOR DELAY INHIBIT PERMIT
SLIDE	WaP.79	ISP ALGOL	PROCESS	non	oui	oui	INIT DELAY SIGNAL
ADL	Leu.79	PASCAL concurrent	MODULE	non	oui	oui	MONITOR
PMS	BeN.70		oui	non	non	non	aucune
SAMEN/ SAMO	Sch.77	ALGOL	oui	non	oui	oui	Evaluation Nets
CAP	Ram.79	LOGOS PL/1	?	non	oui	oui	Graphe DELAY
LCD	EGO.77	APL	ITEM	oui	non	non	Condition
ISPS	BBc.78		non	non	non	oui	Condition
CDL	Chu.69 Chu.65		non	oui	non	non	Condition
AHPL	NSH.77	APL	non	oui	non	oui	Etiquette
EPISODE	LET.78		non	non	oui	non	aucune

Tableau 1

D'autres méthodes de vérification ont été proposées ces dernières années. Mais, soit parce qu'elles sont très liées à un niveau de conception, voire à une option de modélisation, soit parce qu'elles sont dépendantes du type de circuit utilisé, aucune n'est à ce jour en mesure de remplacer la simulation. Ainsi, au niveau trans fert de registre, il est possible de détecter statiquement certains conflits, en prouvant pour chaque registre que toutes les conditions qui déterminent son chargement sont, ou ne sont pas en cas d'erreur, deux à deux incompatibles [KSM.79]. Le problème de la validation d'un circuit combinatoire par rapport à sa description fonctionnelle a pu recevoir une solution plus efficace que la simulation exhaustive, dans le cas particulier des réseaux de portes NAND : un programme a été écrit pour vérifier l'équivalence entre le circuit imaginé par le concepteur, et un circuit issu d'une synthèse automatique à partir de sa description fonctionnelle (et de ce fait validé par construction, mais non optimisé) [Rot.77]. A des niveaux moins détaillés, plusieurs auteurs proposent d'appliquer des méthodes de simulation symbolique pour produire automatiquement des assertions sur une description de machine [Oak.79], prouver une relation d'équivalence faible, entre une description fonctionnelle et une description structurelle au niveau transfert de registre [Dar.79], ou vérifier formellement des microprogrammes [CJB.79]; d'autres travaux sur la simulation symbolique sont cités dans [Cov.80]. Cette technique, dont la valeur a été démontrée sur des exemples simples, n'est cependant, à notre connaissance, pas sortie des laboratoires de recherche pour être appliquée à des projets industriels. Enfin, et sans prétendre par ce bref tour d'horizon avoir épuisé le sujet de la vérification, nous ne pouvons passer sous silence les innombrables études menées au niveau le plus abstrait sur les modèles de graphes de contrôle et de graphes de flots de données, ayant pour but de démontrer l'absence d'interblocage, de conflit d'accès, d'indéterminisme etc ... sur les spécifications d'un système logique [Sif.77, Mil.73, Bra.75, ...].

NECESSITE D'UN SYSTEME INTEGRE D'AIDE A LA CONCEPTION

Parallèlement au développement des méthodes de vérification et de synthèse, et souvent en avance sur celles-ci,les réalisations d'outils logiciels automatisant les phases terminales de la conception sont extrêmement nombreuses [DAC.75-80] : génération de séquences de test, optimisation de PLAs, dessin de circuits imprimés, implantation et interconnexion de circuits intégrés, dessin de masques. Malgré la nécessité d'améliorer encore à la fois le résultat produit et les performances de tous ces outils, le problème essentiel qui se pose depuis 5 ans est celui de leur intégration, avec les programmes de vérification, au sein d'un système unique d'aide à la conception, offrant à l'utilisateur :

- une interface standard pour tous les logiciels
- des outils de gestion des différentes versions de son projet
- la possibilité de ne décrire qu'une fois son circuit à un niveau donné
- des outils de vérification de cohérence, et de gestion des informations nécessaires aux différentes phases
- la possibilité d'appliquer, à tous les niveaux de description, l'ensemble des traitements de visualisation, vérification, synthèse etc ... pertinents.

Les réalisations de systèmes CAO dont nous avons connaissance sont loin de répondre à tous les aspects de ce cahier des charges. En général, l'intégration est réalisée au niveau réseau de circuits intégrés, en branchant autour d'un système de gestion de base de données une bibliothèque de modules standard, des outils de placement, d'interconnexion et de tracé, et un simulateur. C'est ce qui est fait par exemple dans le système SCALD, où l'interaction avec le concepteur se fait par l'intermédiaire d'un éditeur graphique [MWW.77]. Le système CAO de Hughes AircraftCompany, plus complet, inclut un simulateur analogique et un programme de tracé de circuits imprimés; l'utilisateur accède à tous les outils par l'intermédiaire d'un moniteur de communication unifiant leurs langages de commande [WoB.79]. D'autres projets analogues, dont nous ne connaissons pas l'état d'avancement actuel sont menés en Norvège [BaA.78], en RFA [Sue.80], au Canada [BoV.77] etc ... .

Tous les systèmes CAO que nous avons évoqués ont pour caractéristique de réaliser une intégration que nous qualifierons d'horizontale, car se plaçant à un niveau, au maximum deux niveaux voisins. Nous pensons qu'un véritable système intégré de CAO des systèmes logiques doit aussi réaliser une intégration verticale, c'est-à-dire être utilisable à tous les niveaux de conception, offrir une méthode uniforme de description, et permettre des traitements sur des descriptions multi-niveaux. C'est sur les problèmes d'intégration verticale qu'ont porté l'essentiel de nos recherches des cinq dernières années.

NOTRE DEMARCHE

Dans ce but, nous avons mené une réflexion théorique visant à dégager les principes communs à la très grande majorité des langages de description de systèmes logiques, afin de définir les primitives d'un ensemble cohérent de langages de description. Ce travail a été conduit au sein du projet international CONLAN, en tant que l'un des six membres du groupe de travail.

L'une des conclusions particulièrement intéressantes et originales auxquelles le groupe est parvenu est que la description d'un système logique fait appel à un sous-ensemble des notions nécessaires à la définition d'un langage de description. La présentation de CONLAN occupe la troisième partie de cet ouvrage; loin d'être une traduction ou un résumé du rapport de définition à paraître, elle a pour but d'en faciliter la lecture en apportant sur certaines primitives un éclairage différent.

La quatrième partie est consacrée à l'exposé d'un modèle d'évaluation permettant de spécifier l'interprétation des primitives d'un langage de description de systèmes logiques. Motivée initialement par la nécessité de lever toute ambiguité dans la définition de CONLAN, et par l'absence de méthode pour définir rigoureusement la sémantique de ce type de langage, l'élaboration de ce modèle a été guidée par une démarche d'essence expérimentale (le modèle est implémentable) et un souci de non ésotérisme. Nous estimons que ce n'est qu'un premier pas vers une formalisation des simulateurs à évènements discrets, et espérons que la critique de cette approche donnera naissance à de meilleurs modèles, laissant à d'autres personnes, ayant le goût des recherches théoriques, le soin de poursuivre dans cette voie.

Nous avons préféré, quant à nous, revenir vers les applications,et vérifier si la méthode de définition CONLAN proposée était réellement utilisable. Nous l'avons donc mise à l'épreuve sur les langages que nous connaissions bien, en redéfinissant en CONLAN la sémantique de CASSANDRE, LASCAR et LASSO. La cinquième partie de ce mémoire propose deux manières de parvenir à ce résultat.

Le travail qui reste à faire est essentiellement de nature pratique, et n'est concevable que dans le cadre d'une équipe importante : mettre en oeuvre le système intégré que nous avons appelé de nos voeux. Nous participons à un tel projet [CAD.2], et nous conclurons en montrant l'impact de nos recherches sur la réalisation en cours.

DEUXIEME PARTIE

UN EXEMPLE : LE LANGAGE LASSO

I PRESENTATION DE LASSO

Un modèle de système logique décrit en LASSO fait apparaître la structure globale du système, c'est-à-dire son découpage en sous-systèmes, a priori considérés comme asynchrones, échangeant des informations. Chaque sous-système est un système logique en lui-même, et peut être à nouveau décomposé, ce qui confère à l'ensemble une structure arborescente d'unités imbriquées (éventuellement réduite à un seul élément). Le comportement des éléments terminaux de cette arborescence, ou de la partie d'un système qui n'a pas été décomposée en unités plus simples, est considéré de manière macroscopique : les fonctions réalisées par ces composants sont décrites, mais non leur mise en oeuvre interne ni leur micro-synchronisation. C'est dans cette optique que LASSO doit servir de couche supérieure à l'ensemble d'outils de simulation CASSANDRE - LASCAR précédemment développés.

Ainsi, par rapport à CASSANDRE et LASCAR, un certain nombre de principes fondamentaux ont été généralisés. La modularité d'une description, caractéristique essentielle de ce type de langage, a été conservée; mais à la notion d'unités connectées par des nappes de signaux booléens, nous avons substitué celle d'unités communicant par des messages plus généraux. La notion d'entité paramétrisable a été introduite, et permet d'engendrer à partir d'une même description, une famille d'unités particulières.
Le contrôle du séquencement des actions à l'intérieur d'une unité, limité dans CASSANDRE et LASCAR à un automate, a été étendu à un schéma de graphe orienté plus général.

Nous avons introduit le concept de spécification d'une entité, permettant d'imposer des contraintes sur son utilisation, et de vérifier, si possible statiquement, sinon à la simulation, le comportement de ses entrées-sorties en fonction du temps. Cet aspect du langage LASSO est sans doute le plus original, car nous n'avons pas connaissance d'un seul langage de description permettant l'expression de spécifications temporelles.

II LA MODULARITE EN LASSO

II.1 Notions d'entité et d'unité

Une description est constituée d'un nombre fixe d'unités interconnectées dont les caractéristiques sont connues statiquement.

Aux niveaux CASSANDRE (Asynchrone et Synchrone) et LASCAR, une unité est la description d'un module matériel, donc c'est une description non paramétrable. Par exemple, un additionneur de 4 bits et un additionneur de 8 bits sont deux modules matériels différents, et doivent être décrits par deux unités différentes.

Au niveau LASSO, plus abstrait, nous avons estimé nécessaire de conserver plus de souplesse dans la déclaration d'un module. Nous distinguons donc la déclaration d'entité, éventuellement paramétrable, de son utilisation en tant qu'unité d'un modèle. Une entité est donc un générateur d'objets; une unité est un objet particulier pour lequel les paramètres éventuels de l'entité qui le définit ont été fixés.

Exemple

On peut décrire un générateur de mémoires par la déclaration :

```
entité MEMOIRE (entier N, A, B);
   :
fin; % MEMOIRE %
```

N, A, B sont des paramètres entier dont la signification peut être la suivante :

N taille de la mémoire en mots
A temps de lecture
B temps d'écriture.

a) Définition externe ou locale d'une entité

Lors de la description du modèle, chaque composant doit être un exemplaire d'une entité qui a été définie auparavant, dans le texte du modèle.

La définition de l'entité génératrice peut être :

- locale à une entité, c'est-à-dire faisant partie du texte de la description et située dans le corps de l'entité dans laquelle le composant est utilisé.

Exemple

```
entité MODELE (...,...); % ceci est un commentaire %
  interface
     :
  corps
  entité A (entier X, Y)
     :
  fin; % entité A %
  unité ALPHA [1 : 4] : A(4, 16);
       % déclaration de 4 exemplaires identiques de A %
   :
   :   % suite de la description du modèle, utilisant les 4 exemplaires de A %
   :
fin; % de l'entité MODELE %
```

- externe au modèle. Elle se trouve dans une bibliothèque d'entités prédéfinies qui peut être chargée par l'utilisateur du système. Cette possibilité permet à l'utilisateur d'avoir à sa disposition des générateurs d'objets (mémoire, processeur, ...) qu'il utilise souvent sans avoir à les réécrire. C'est aussi le moyen de partager des descriptions entre participants à un même projet.

Exemple

```
entité MODELE (..., ...);
  :
  :
corps
  entité externe A, B; % les entités A et B sont définies dans une bibliothè-
       que du système de simulation %
   :
   :   % suite de la description de MODELE déclarant et utilisant des exem-
         plaires de A et B %
fin; % de MODELE %
```

b) Déclaration d'unité

Un composant du modèle se définit comme un exemplaire d'une entité particulière :

Exemple

unité M1 : MEMOIRE (1000, 1, 2);

Déclare une unité mémoire de 1000 mots, et de temps de cycle en lecture et écriture respectivement de 1 et 2 unités de temps.

La déclaration d'une unité peut être comparée au fait de prendre un circuit intégré pour l'inclure dans un système plus important, matérialisé par exemple par une carte. De ce fait, la déclaration d'une unité déclare aussi implicitement ses broches d'entrées-sorties, et tous ses objets locaux.

REGLES D'ECRITURE

. Plusieurs exemplaires d'une même entité peuvent être déclarés en attribuant à chaque exemplaire un identificateur différent.

. Il est possible de déclarer plusieurs unités identiques sous forme de tableau. Ces unités seront alors référencées par le mécanisme d'indexation habituel.

. Lors d'une déclaration d'unité formulée à partir d'une entité paramétrée, il faut fixer tous les paramètres formels. Une unité est donc toujours un objet entièrement défini (ce qui correspond à la réalité de la construction d'une maquette).

En pratique, la structuration d'un modèle en unités peut avoir une motivation logique plutôt que physique, et conduire à un découpage plus fin, ou moins fin, que celui de la réalisation matérielle du système logique décrit.

II.2 Structure d'une description d'entité

La description d'une entité est constituée de trois parties distinctes, qui se succèdent dans un ordre imposé (les crochets entourent ce qui est optionnel) :

entité NOM [(liste de paramètres formels)]
interface
    description des entrées-sorties
corps
    [déclaration des objets internes]
        [description du fonctionnement]
[spécifications
    relations à vérifier sur les paramètres et les entrées-sorties]
fin;

a) L'interface

Elle contient la déclaration des signaux de contrôle et des données par lesquels toute unité engendrée à partir de l'entité est connectée aux autres composants d'un modèle. L'interface constitue le seul moyen de communication entre une unité et son environnement. Les notions de variables globales au sens d'Algol, ou d'accès par notation pointée aux variables internes à un autre module au sens de Simula, sont ici interdites, pour les raisons suivantes :

. De tels accès sont sans signification lorsqu'on décrit des modules matériels.
. On veut pouvoir décrire, vérifier, simuler et tester les unités d'un modèle indépendamment les unes des autres avant de passer à la simulation du modèle tout entier.
. On élimine ainsi de nombreux risques d'erreur.

En LASSO, les communications correspondent à des échanges de messages. Un signal est considéré comme une demande effectuée à une unité ou comme un acquittement de cette unité pour un travail. Nous admettrons qu'un signal maintient sa valeur jusqu'à sa prise en compte par son récepteur, et qu'il est uni-directionnel. Un signal sera donc déclaré comme entrée ou sortie.

A chaque signal de l'interface sont éventuellement associées des données, elles aussi déclarées comme variables d'interface, mais de types généraux et de dimensions quelconques : entiers, caractères, booléens, variables structurées. On ne fait alors aucune hypothèse de réalisation physique.

L'ensemble {signal - données associées} constitue un message, qui sera interprété par l'unité réceptrice.

Exemple

Une mémoire peut recevoir des demandes de lecture (signal LEC accompagné de l'adresse AD du mot à lire) ou d'écriture (signal ECR accompagné de l'adresse AD et de la valeur DATA du mot à écrire). En réponse, la mémoire renvoie respectivement la donnée lue (signal OKLEC accompagné de DATA) ou un simple acquittement (signal OKECR). L'interface de cette mémoire s'écrit :

```
interface
    entier AD, DATA;
    entrée LEC(AD), ECR(AD, DATA);
    sortie OKLEC(DATA), OKECR;
```

b) Corps d'entité

Le corps d'entité contient la déclaration des objets internes, et la description du fonctionnement de l'entité. On y trouve un ou plusieurs des points suivants :

1 - La définition d'entités (locales ou externes).
2 - La définition de fonctions et procédures (locales ou externes).
3 - Si 1) est non vide, la déclaration d'unités internes.
4 - La déclaration des variables et signaux internes.
5 - Si 3) est non vide les instructions de connexion des unités internes.
6 - L'ensemble des instructions permettant de décrire, en fonction des signaux d'entrée, la configuration des signaux de sortie à chaque instant de la simulation.

On peut logiquement classer ces instructions en deux catégories :

. la partie opérative, définissant des actions,
. la partie contrôle, définissant l'enchaînement (parallèle ou séquentiel) de ces actions.

Les points 1 à 4 ci-dessus peuvent être écrits dans un ordre quelconque, à condition de ne pas procéder à des références avant. Ainsi, la définition d'une entité doit précéder la déclaration d'exemplaires de cette entité.

Par contre, si les points 5 et 6 sont tous deux présents, ils doivent se suivre dans cet ordre.

Le cas limite d'un corps d'entité vide est accepté, ce qui permet à l'utilisateur de décrire un système complet avant d'en avoir décrit entièrement tous les composants; toutefois, un tel modèle n'est simulable que lorsque toutes les unités qu'il contient sont des exemplaires d'entités dont le corps renferme au minimum la prise en compte des messages d'entrée (même si leur interprétation est incomplète) et l'émission des messages de sortie (éventuellement en produisant des valeurs par défaut).

c) Spécifications

Cette partie, qui est optionnelle, est destinée à indiquer les propriétés que doit satisfaire tout exemplaire de l'entité pour que la description ait un sens (relations entre paramètres, exclusion de signaux d'interface, etc ...).

Les spécifications servent de référence pour effectuer des vérifications :

- statiquement, à la génération d'une unité, et lorsque l'entité est paramétrée. En effet, il se peut que le programme de description n'ait de sens que si certaines relations entre les paramètres sont vérifiées. Ces relations doivent figurer dans les spécifications, et elles seront testées à chaque création d'une unité, chaque paramètre recevant alors une valeur.

- dynamiquement, au cours de la simulation. Il s'agit alors de vérifier des relations entre les entrées-sorties d'une unité, éventuellement en fonction du temps. Dans l'état actuel des recherches sur ce sujet, on sait rarement démontrer formellement que des spécifications temporelles sont vérifiées par une description. Dans ce cas, l'interpréteur pourra être chargé de vérifier que les résultats de simulation sont compatibles avec les spécifications. Nous pensons qu'un tel mécanisme peut contribuer efficacement à la détection de certaines erreurs.

Les spécifications sont exprimées par des prédicats portant :

- sur des valeurs simples
- sur des séquences de valeurs valables à tout instant ou pendant une certaine durée suivant l'occurrence d'un évènement déterminé.

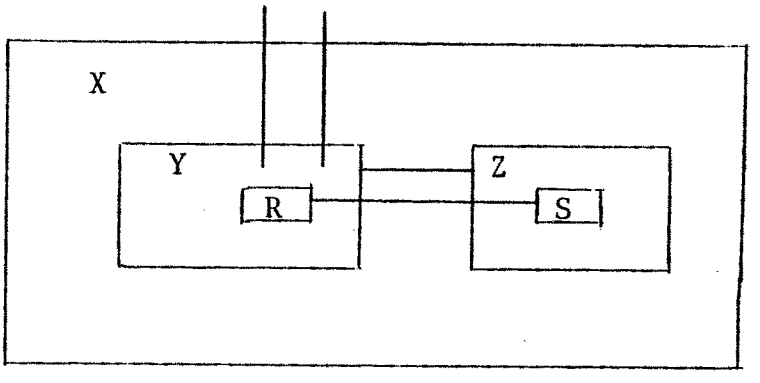
Nous allons à présent expliciter plus en détails chacune des trois parties qui constituent une description d'entité.

III COMMUNICATION ENTRE UNITES

III.1 Définition de l'interface

Les variables d'interface d'une unité peuvent être visualisées comme les "patte d'un circuit intégré. Elles sont donc accessibles à la fois dans l'unité où elles sont définies, et à l'extérieur de cette unité, mais au niveau immédiatement englobant.

Exemple



L'interface de Y est accessible dans Y et dans X, mais non dans R, Z et S. De même, l'interface de R est connu dans R et Y, mais non dans X, Z, S.

Si l'on veut établir une liaison entre Y et Z, celle-ci doit être écrite dans X, qui a accès aux deux interfaces.

Une liaison entre R et S ne peut se faire que si :

- au niveau de Y, il existe une liaison entre les interfaces de Y et R
- au niveau de Z, il existe une liaison entre les interfaces de Z et S
- les variables d'interface correspondantes de Y et Z sont ensuite connectées dans X.

a) Les données

Le mot donnée doit être compris ici avec son sens le plus général, et par opposition au mot contrôle. Ainsi, une donnée peut être un simple paramètre, ou bien un "paquet" transmis dans un réseau, aussi complexe que l'on veut.

Une donnée peut donc être une variable scalaire, tableau, ou structurée, construite à partir des types de base du langage : booléen, caractère et entier. Les constructeurs de tableau et de structure peuvent s'appliquer à des éléments non scalaires.

Exemple

```
entier A, B [1 : 4].
booleen C, D [0 : 3, 0 : 15];
structure (entier X [1 : 2], caract Y) Z [0 : 5].
```

b) Les signaux

Les signaux sont des commandes que l'unité reçoit ou émet vers l'extérieur, ou bien des acquittements d'ordres ou de messages. De manière à faciliter la modélisation des échanges entre unités, et à permettre des vérifications plus aisées de la synchronisation entre unités, les signaux ont été choisis unidirectionnels. Un signal est donc déclaré en entrée d'interface ou en sortie.

Au niveau d'abstraction où nous nous plaçons, une commande ou un acquittement est ou n'est pas validé. Aussi, la valeur d'un signal est-elle un booléen. De plus, un signal est doté d'une propriété de mémorisation : un signal reste validé tant qu'il n'a pas été pris en compte par son récepteur. Enfin, un signal peut être un scalaire ou un tableau.

Exemple

```
entrée X, Y [1 : 3];
sortie Z [0 : 15, 0 : 3, 1 : 2];
```

c) Association signal-donnée

Un signal peut avoir une signification complète par lui-même, auquel cas il est émis seul. Ou bien à un signal (ou un ensemble de signaux), sont associées des données, l'ensemble formant un message qui est envoyé en bloc sur l'interface. Il faut alors préciser, pour chaque signal, quelles données lui sont attachées.

Règle générale d'écriture

1) Les données sont déclarées avant les signaux d'entrée et de sortie
2) L'association de données à un signal scalaire, ou de données communes à tous les éléments d'un tableau de signaux, se fait lors de la déclaration d'entrée ou sortie de ce signal, immédiatement après l'écriture de l'identificateur du signal et de ses dimensions.
3) Si les données ne sont pas communes à tous les éléments d'un signal tableau, il suffit d'en donner la liste des associations individuelles. Cette liste est annoncée par le mot clé avec.

Exemple 1

Reprenons l'interface de l'entité MEMOIRE décrite précédemment :

```
entité MEMOIRE (entier N, A, B);
interface
    entier AD, DATA;
    entrée LEC(AD), ECR(AD, DATA);
    sortie OKLEC(DATA), OKECR;
```

LEC et OKLEC ont chacun une donnée associée, ECR en a deux.

Seul le signal OKECR est indépendant. On voit qu'une même donnée peut être associée à différents signaux, pas forcément de même sens. L'utilisateur a alors la charge de s'assurer (ou de spécifier dans la partie spécification) qu'il n'y a pas de conflit : les signaux ayant une donnée commune doivent être exclusifs deux à deux.

Exemple 2

Redéfinissons une autre mémoire, avec des signaux d'entrée vectoriels. LEC et ECR sont maintenant respectivement DEMANDE[1] et DEMANDE[2]. Les deux signaux d'entrée sont associés aux même données, et dans le cas d'une demande de lecture DATA est ignoré.

L'interface s'écrit alors :

```
entité MEMORIA (entier N, A, B);
interface
    entier AD, DATA;
    entrée DEMANDE[1 : 2] (AD, DATA);
    sortie OKLEC(DATA), OKECR;
```

Exemple 3

Dans l'exemple ci-dessous, chaque élément de la sortie X est associé à des données qui lui sont propres.

```
entité EXEMPLE3
interface
    caract ALPHA[1 : 4];
    booléen BETA[0 : 5]
    sortie X[0 : 2] avec
        X[0] (BETA[0 : 3]),
        X[1] (ALPHA[1 : 2], BETA),
        X[2] (ALPHA[3 : 4],  BETA[2 : 5]);
```

Simplification des écritures répétitives

L'instruction pourtout permet, en faisant varier un indice, d'exprimer de façon compactée des correspondances répétitives entre éléments d'un tableau de signaux et leurs données, lorsqu'il n'y a pas compatibilité dimensionnelle. L'ordre dans lequel l'indice prend ses valeurs successives ne doit pas influer sur le résultat. Il faut comprendre cette instruction comme une macro qui permet de générer automatiquement la liste des correspondances individuelles.

Exemple 4

L'entité MX est un multiplexeur à 8 sorties et 4 entrées, qui reçoit un message et une adresse, et sélectionne en fonction de l'adresse, la sortie sur laquelle le message sera transmis. L'entité MX rajoute au message une information supplémentaire, dépendant de la parité du numéro de sortie.

```
entité MX
interface
    entier AD[1 : 4];
    structure (entier E1, caract E3) DATAIN[1 : 4],
              DATAOUT[1 : 8];
    caract INFO[1 : 4];
    entrée IN[1 : 4] avec
        pourtout I de 1 a 4
        début IN[I] (AD[I], DATAIN[I]) fin;
    sortie OUT[1 : 8] avec
        pourtout I de 1 pas 2 a 7
        début
           OUT[I] (DATAOUT[I]),
        fin,
        pourtout I de 2 pas 2 a 8
        début
           OUT[I] (DATAOUT[I], INFO[I div 2]),
        fin;
```

III.2 Les instructions de connexion

Les différentes unités d'un modèle fonctionnent de manière asynchrone (pas de synchronisation par horloge). Deux unités, qu'elles soient imbriquées l'une dans l'autre, ou qu'elles se trouvent au même niveau d'imbrication dans une même troisième, peuvent communiquer et se synchroniser à l'aide de leurs signaux d'interface, si une liaison a été établie entre ces signaux.

Un ou plusieurs signaux, et leurs données associées, sont positionnés sur l'interface d'une unité par l'exécution de l'instruction valider dans le corps de cette unité. En fonction des liaisons établies, la transmission sera effectuée vers d'autres interfaces, soit instantanément, soit avec un retard indiqué au niveau de la liaison.

Nous appelons instruction de connexion une instruction qui lie, de manière permanente, des signaux et données d'interface d'une unité, interne à une entité,

- soit à des signaux et données d'interface de l'entité qui la contient,
- soit à des signaux et données d'interface d'une autre unité interne de même niveau d'imbrication.

. Les instructions de connexion apparaissent toujours dans le corps de l'entité englobante, laquelle a accès aux interfaces des unités qu'elle contient (et seulement à leurs interfaces) par notation dite pointée.

. Les connexions peuvent correspondre à l'établissement de liens physiques (équivalents à des fils), ou plus généralement décrire l'existence d'un chemin par lequel transitent des messages.

. Une instruction de connexion est distincte d'une affectation. Lors d'une affectation, la variable en partie gauche du symbole ":=" reçoit la valeur de l'expression en partie droite, quelle que soit cette valeur si elle est du type requis, et à des instants déterminés par une condition d'activation externe à l'affectation elle-même.
Lorsqu'il s'agit d'une connexion, c'est la validité du signal en partie droite qui détermine le transfert d'information vers la partie gauche. Pour cette raison, la connexion est écrite à l'aide d'un symbole particulier : "↚".

L'interprétation de la connexion est la suivante : lorsque le signal en partie droite du symbole "↚" est validé, le signal en partie gauche est validé, soit immédiatement si aucun retard n'est précisé, soit après le nombre d'unités de temps précisé. Le signal en partie droite est automatiquement invalidé par l'activation de la connexion. Si les signaux connectés ont des données associées, la transmission de leurs valeurs à partir de l'interface émettrice est déclenchée par la validation du signal en partie droite; leur arrivée sur l'interface réceptrice se fait en même temps que la validation du signal en partie gauche.

Une connexion avec retard peut être ré-activée alors que le message précédent n'est pas encore parvenu à destination (effet "pipe-line"). Par contre, toute tentative pour valider en partie gauche du symbole "←" un signal déjà valide est détectée comme une erreur de synchronisation : le message précédent n'a pas été pris en compte par l'unité réceptrice.

a) Connexion de signaux seuls

Elle permet d'exprimer le fait qu'une unité envoie uniquement une interruption ou un signal d'acquittement à une ou plusieurs unités.

Dans l'exemple suivant :

P. ECRIT ← MEM.OKECR;

est l'émission du signal d'acquittement d'écriture envoyé par la mémoire MEM au processeur P; ECRIT est un signal d'entrée de P.

b) Connexion de signaux à messages

L'envoi d'un message ne peut se faire que s'il est accompagné d'un envoi de signal. Par défaut, la seule connexion des signaux véhiculant un message implique la connexion de leurs données associées.

Dans l'exemple suivant, la connexion :

MEM.ECR ← P.ECRIRE;

exprime une demande d'écriture envoyée par le processeur P à la mémoire MEM. ECRIRE est un signal de sortie de P. Le message est ici composé (voir les interfaces de MEMOIRE et de PROCESSEUR) de deux parties :

- une partie adresse mémoire où doit être écrite la donnée
- une partie donnée à écrire.

Les messages déclarés dans les interfaces de l'unité émettrice et de l'unité réceptrice doivent avoir le même nombre d'éléments, et leurs éléments de même rang doivent être deux à deux de même type.

Dans notre exemple :

P.ADRESSE et MEM.AD
P.DONNEE et MEM.DATA

sont implicitement connectés.

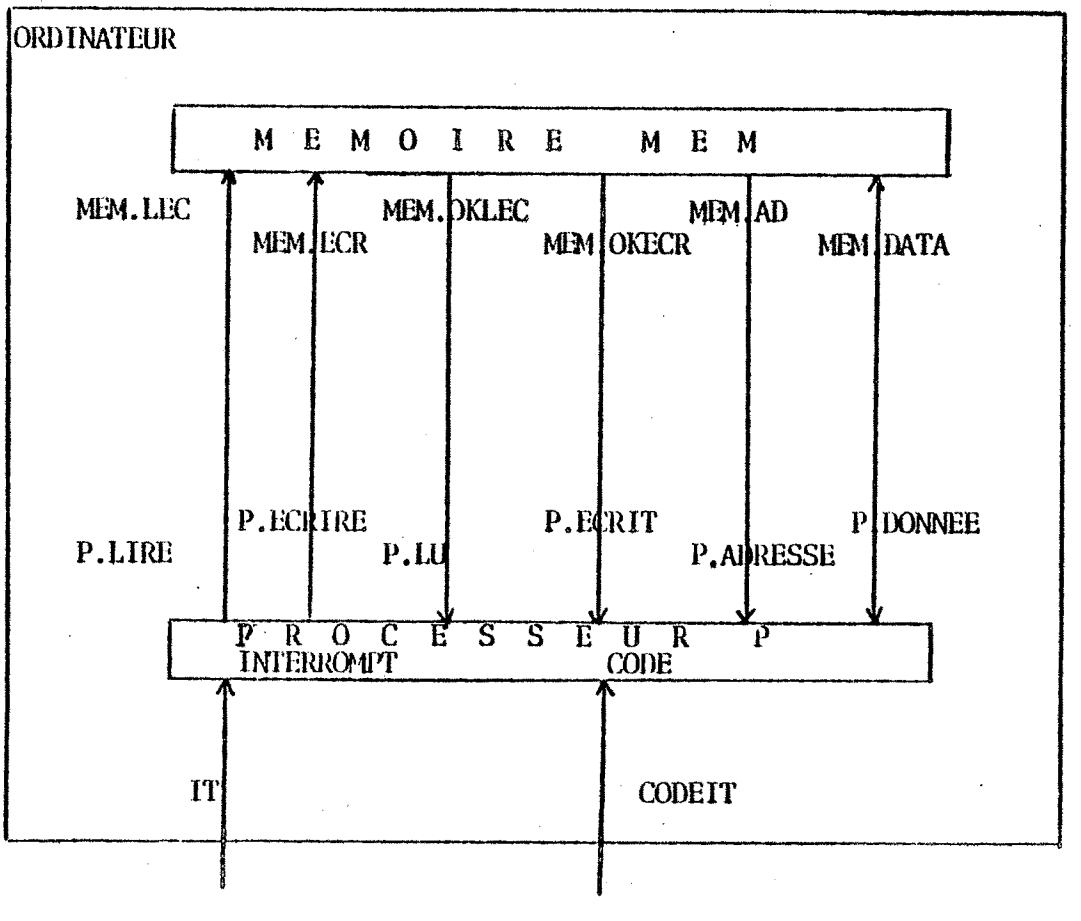
c) Connexion à retard

Dans les deux exemples précédents, la transmission du signal était immédiate. Il est cependant possible qu'un signal et des données associées (si elles existent), ne puissent être disponibles avant un certain temps correspondant au temps de transfert d'une interface vers une autre. C'est ce qui est exprimé dans la dernière connexion de l'exemple ci-dessous, qui illustre en outre la connexion d'un signal entre unité englobante et unité englobée :

P.INTERROMP ← IT retard(1);

Exemple

L'entité ORDINATEUR est constituée de deux unités interconnectées, une MEMOIRE et un PROCESSEUR.



```
entité MEMOIRE(entier N, A, B);
interface
    entier AD, DATA;
    entrée  LEC(AD), ECR(AD, DATA);
    sortie OKLEC(DATA), OKECR;
corps
  :
fin; % MEMOIRE %

entité PROCESSEUR;
interface
    entier ADRESSE, DONNEE, CODE;
    entrée LU(DONNEE),ECRIT, INTERROMP(CODE);
    sortie LIRE(ADRESSE), ECRIRE(ADRESSE, DONNEE);
corps
  :
fin; % PROCESSEUR %

entité ORDINATEUR;
interface
    entier CODEIT; % code interruption externe %
    entrée IT(CODEIT); % interruption externe %
corps
    entité externe MEMOIRE, PROCESSEUR;
    unité MEM : MEMOIRE(1024, 6, 8);
    unité P : PROCESSEUR;
      :
      % connexionx permanentes %
    MEM.LEC ← P.LIRE; % demande lecture mémoire %
    MEM.ECR ← P.ECRIRE; % demande écriture mémoire %
    P.LU ← MEM.OKLEC; % acquittement lecture %
    P.ECRIT ← MEM.OKECR; % acquittement écriture %
    P. INTERROMPT ← IT retard(1); % envoi interruption externe %
fin; % entité ORDINATEUR %
```

d) Connexion multiple

Une seule instruction de connexion multiple permet de condenser l'écriture d'une suite de connexions simples se rapportant à une même unité.

Dans l'exemple précédent, une écriture plus compacte des connexions pourrait être :

```
MEM(entrée P.LEC, P.ECR; sortie P.OKLEC, P.OKECR);
P.IT ← IT retard(1);
```

ou bien encore

```
P(entrée MEM.OKLEC, MEM.OKECR, IT retard(1);
  sortie MEM.LEC, MEM.ECR);
```

IV SYNCHRONISATION ET CONTROLE A L'INTERIEUR D'UNE UNITE

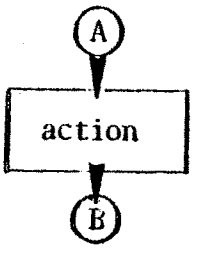
La partie fonctionnelle d'une entité écrite en LASSO s'exprime comme un ensemble d'opérations sous condition, dont le déroulement (séquentiel ou parallèle) et la synchronisation sont modélisés par un graphe de contrôle.

IV.1 Partie opératoire

La partie opératoire est représentée par des actions, portions d'algorithmes formant un tout ininterruptible, et ayant une certaine durée exprimée par un délai. L'exécution d'une action est déclenchée par un signal, et la fin de son exécution valide un autre signal. Si la primitive délai est absente, l'action est supposée instantannée. Plusieurs actions lancées en parallèle ont un déroulement asynchrone.

Exemple

```
? A ? début
        :   liste d'insctructions
        :   décrivant l'action
        :
    fin délai(X) valider(B);
? B ? ......
```



Les actions sont décrites à l'aide d'un langage algorithmique dérivé de PASCAL.

IV.2 Partie contrôle

La partie contrôle sert à valider les signaux de sortie de l'unité et les signaux déclenchant les actions, et à synchroniser les actions internes à l'unité. On peut aisément représenter le contrôle par un graphe de transition dans lequel les places représentent les signaux et les boîtes les transitions; les termes "place" et "transition" ont ici la signification qui leur est habituellement donnée dans la littérature consacrée aux réseaux de Pétri et autres graphes de contrôle [Bae.73].

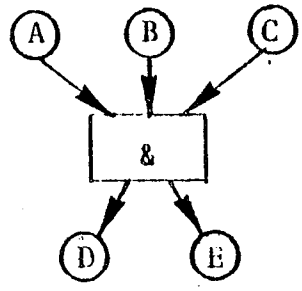
Le fonctionnement modélisé est le suivant : à un instant donné, toutes les transitions actives sont évaluées en parallèle. Plusieurs cas de non déterminisme peuvent se produire :

- deux actions parallèles modifient une même variable en lui affectant des valeurs différentes
- deux transitions simultanément actives partagent un même signal d'entrée, de sorte que l'évaluation de l'une rend l'autre inactive.
- deux transitions simultanément actives partagent un même signal de sortie, de sorte que l'évaluation de l'une bloque l'autre.

Un fonctionnement non déterministe dans un système logique étant le plus souvent l'indication d'une erreur de conception, tous ces cas sont détectés et un message d'anomalie est envoyé à l'utilisateur.

En LASSO, une transition peut être instantannée, ou bien avoir une durée non nulle exprimée par un délai.

Nous avons choisi les primitives suivantes dont la représentation mais non la sémantique est inspirée de celles de DIGITEST II [Ram.75] et de LOGOS [Ros.72].

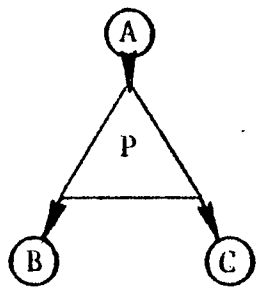


Si tous les signaux en sortie de la transition sont à 0 et si tous les signaux en entrée de la transition sont à 1, ces derniers sont tous remis à 0 et tous les signaux en sortie sont mis à 1.
Dans le cas contraire, la transition n'est pas effectuée (attente).
Le nombre de signaux en entrée et en sortie est quelconque.

? A & B & C ? délai(X) valider(D, E)

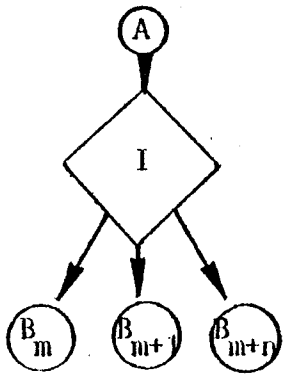
Cas particuliers : S'il y a une seule entrée et plusieurs sorties, cela correspond à l'activation de plusieurs branches en parallèle, souvent appelée FORK.

S'il y a plusieurs entrées et une seule sortie, cela correspond au point de synchronisation de plusieurs branches parallèles, souvent appelée JOIN.



P est une expression booléenne calculée à partir de variables internes ou d'interface (tests de valeurs).
Si les signaux B et C sont à 0 et si le signal A est à 1, P est calculé.
Si P est vrai, B est validé, sinon C est validé. Dans les deux cas, A est remis à 0.

? A ? selon P délai(X) valider(B, C)



I est une expression entière scalaire appartenant à l'intervalle fermé [m, n].
Le nombre de sorties doit être égal à n-m+1.

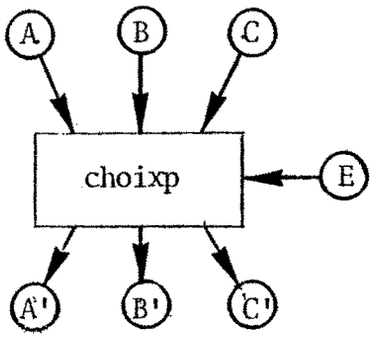
? A ? index I de m à n délai(X) valider(B_m, B_{m+1}, ..., B_{m+n});

Si toutes les sorties sont à 0 et si A a la valeur 1, I est calculé et sert à indexer la place de sortie mise à 1. A est remis à 0.

Si m et n sont omis, m a par défaut la valeur 0 et n, le nombre de sorties diminué de 1.

Exemple

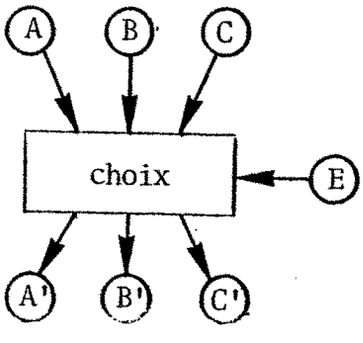
? A ? index I valider(B, C, D) est équivalent à :
? A ? index I de 0 à 2 valider(B, C, D).



Cette transition correspond au test de plusieurs signaux dont la priorité va décroissant de la gauche vers la droite, en vue de choisir le signal valide de plus forte priorité.

? E ? choixp(A, B, C) délai(X) valider(A', B', C');

Si toutes les sorties sont à 0 et si E a la valeur 1, la transition sélectionne, parmi toutes ses autres entrées à 1 celle de plus forte priorité. La sortie de même rang est mise à 1, et E et l'entrée sélectionnée sont remis à 0.

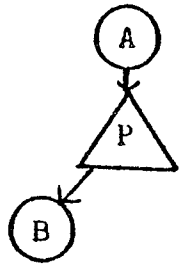


Cette transition ressemble à la précédente, en ce sens qu'à chaque entrée (exceptée l'entrée E qui a un rôle particulier) correspond une place de sortie. Mais ici, les entrées ont une égale priorité, et une seule à la fois peut être valide (dans le cas contraire il y a erreur). Si toutes les sorties sont à 0, si E vaut 1 et si l'une, et une seule, des autres entrées vaut 1, la sortie correspondante prend la valeur 1 et les entrées sont remises à 0.

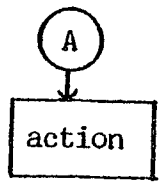
? E ? choix(A, B, C) délai(X)
valider(A', B', C');

Cas limite

On peut vouloir exprimer qu'une branche du graphe de contrôle se termine sans valider aucun signal, ou que l'effet de certains signaux n'est pas pris en compte (modèle incomplet par exemple). Ce fonctionnement se traduit dans le langage par l'absence de certains, ou de tous signaux en sortie de transition, les paramètres de l'instruction valider étant vides.



? A ? selon P valider(B,);
Si P est faux, A est détruit.
Cette transition décrit le fonctionnement d'une porte "et".



? A ? début
:
fin; valider ( );
Action sans suite.

IV.3 Exemple complet de la description d'une mémoire

```
entité MEMOIRE(entier N, A, B);
interface
    entier AD, DATA;
    entrée  LEC(AD), ECR(AD, DATA);
    sortie OKLEC(DATA), OKECR;
corps
    entier TABLE[1 : N]; % la mémoire contient N entiers %
    signal PRET init 1;
    ? PRET ? choix(LEC, ECR) valider(READ, WRITE);
             % état initial : aiguillage en fonction des signaux d'entrée %

    ? READ ? début % action de lecture, durée A unités de temps %
                DATA := TABLE[AD];
             fin délai(A) valider(FINREAD);
    ? WRITE ? début % action d'écriture, durée B unités de temps %
                TABLE[AD] := DATA;
             fin délai(B) valider(FINWRITE);
    ? FINREAD ? valider(OKLEC, PRET);
    ? FINWRITE ? valider(OKECR, PRET);
```

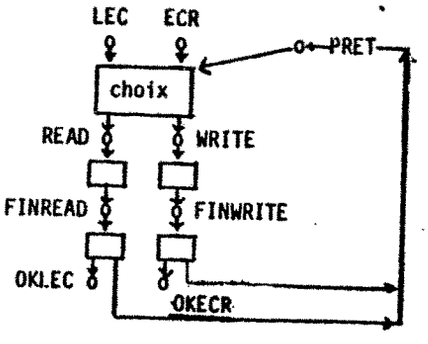
spécification

A < B; % temps de lecture inférieur au temps d'écriture %

∿(LEC & ECR); % on ne peut avoir en même temps une demande de lecture et une demande d'écriture %

fin; % MEMOIRE %

Le schéma de contrôle sous forme de graphe de transition est le suivant :



IV.4 Transitions composées

Une des conclusions d'une étude d'évaluation de LASSO pour la description fonctionnelle de circuits intégrés [Dub.79] était qu'une description respectant rigoureusement les principes de séparation de la partie contrôle et de la partie opératoire devenait rapidement trop lourde. "L'introduction de nombreux signaux intermédiaires, n'ayant aucune existence réelle dans le circuit à simuler, est un défaut majeur aux yeux d'utilisateurs épris de règles simples d'écriture, et de descriptions réalistes".

La justification essentielle d'un outil de CAO étant son utilité pratique, laquelle passe nécessairement par son acceptation auprès des concepteurs auxquels il est destiné, nous avons étendu la définition des transitions, de manière à compacter, en une seule instruction, l'utilisation de 2 ou plusieurs primitives, dont au plus une action. Le type de modélisation que permet cette écriture n'est pas sans rappeler par certains côtés les "macro evaluation-nets" [NoN.73], et la distinction entre action opératoire et primitive de contrôle se perd. Le choix entre rigueur formelle et simplicité d'écriture est donc laissé à l'utilisateur.

Une transition composée permet au maximum de combiner :

- une synchronisation de branches parallèles
- une action
- la validation de plusieurs signaux en sortie, ou d'une liste de signaux parmi plusieurs listes possibles.

Cette combinaison est obtenue par :

- l'extension à une conjonction de signaux du signal de déclenchement de n'importe quelle transition
- l'extension à une liste de signaux de chaque sortie de transition
- l'association d'une action opératoire à chaque transition.

La forme générale d'une transition composée est donc :

```
? A1 & A2 & ... An ? début ... fin délai(X)
        transition_primitive
        valider((S11, S12, ... S1i), (S21, S22, ... S2j), ...)
```

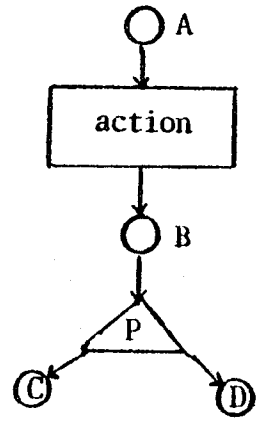
Exemple 1

Association d'une action et d'une décision

```
? A ? début ... fin
      valider(B);
? B ? selon P valider(C, D);
```

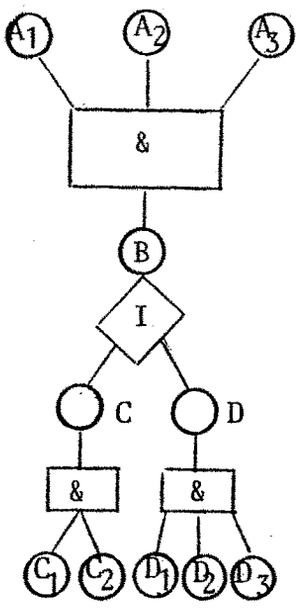
se simplifie en

```
? A ? début ... fin
      selon P valider(C, D);
```



Exemple 2

Association d'une synchronisation,d'une décision et du lancement de plusieurs branches en parallèle.




```
? A1 & A2 & A3 ? valider(B);
? B ? index I valider(C, D);
? C ? valider(C1, D2);
? D ? valider(D1, D2, D3);
```

se simplifie en

```
? A1 & A2 & A3 ? index I
      valider((C1, C2), (D1, D2, D3));
```

V TYPES ET OPERATEURS PRIMITIFS

Dans ce paragraphe, nous distinguerons la notion de type de données, considéré comme un ensemble de valeurs et d'opérations sur ces valeurs, de la notion de type de porteuse, considéré comme un ensemble d'objets contenant une valeur et d'opérations de modification de cette valeur. Ces deux concepts, qui seront formalisés dans le chapitre suivant de ce mémoire, sont ici illustrés de manière informelle.

V.1 Types de données et constructeurs primitifs

Les types de données de base du langage LASSO sont identiques à ceux des langages de programmation classiques, et leur sémantique est bien connue. Nous les évoquons rapidement pour en fixer la syntaxe. Dans tous les cas, les opérateurs sont définis sur des scalaires comme sur des tableaux. Lorsqu'un opérateur unaire porte sur un tableau, l'opération est effectuée sur chaque élément du tableau. Lorsqu'un opérateur binaire est appliqué à deux tableaux, ceux-ci doivent être compatibles, c'est-à-dire avoir le même nombre de dimensions, leurs dimensions ayant deux à deux la même étendue : l'opération est alors effectuée entre éléments de même rang.

a) Booléens

constantes : 0 1
opérateurs : ∿ négation
& et
| ou
= < ≤ > ≥ ∿= comparaisons

b) Entiers

constantes : entiers positifs et négatifs (nombre maximum dépendant de la machine).
opérateurs : + - * div mod abs arithmétiques
= < ≤ > ≥ ∿= comparaisons

c) Caractères

constantes : "A", "B", ... "Z"
opérateurs : = ∿= comparaisons

d) Structures

type composé à l'aide des trois précédents, par définition de champs.

Exemple

structure(entier A, caractère B)

Opérateurs : . sélection de champ
= ∿= comparaisons

Les comparaisons se font champ par champ, entre structures composées de champs deux à deux de mêmes noms et de mêmes types. Si tous les champs ont la même valeur, le résultat est 1, sinon 0.

e) Tableaux

Les tableaux ont un nombre quelconque de dimensions. Pour chaque dimension, les bornes doivent être des entiers positifs.

L'étendue d'une dimension est donnée par borne supérieure - borne inférieure + 1, c'est-à-dire que l'indexation parcourt tous les entiers compris entre les deux bornes, les "trous" n'étant pas autorisés.
Les dimensions sont indiquées entre [ ].
Opérateurs : sélection d'un élément
sélection d'un sous-tableau
/<opérateur> réduction de la première dimension par rapport à l'opérateur (cf APL)
concaténation selon la première dimension.

Exemple

```
A[1 : 2, 0 : 6, 7 : 9]
A[1, 2, 7]          sélection d'un élément
A[1, 2 : 5, 7]      sélection d'un vecteur
A[, 2 : 4,]         sélection d'un sous-tableau
```

Lorsqu'une dimension est prise dans sa totalité, l'indication des bornes peut être omise, mais l'existence de cette dimension doit être marquée par une virgule.

Remarque

Il est possible de définir des tableaux de structures et des structures dont un ou plusieurs champs sont des tableaux.

V.2 Les porteuses du langage

Le concept de porteuse est une généralisation de la notion de variable qui sera entièrement définie au paragraphe I.1 de la troisième partie. Admettons en première approximation qu'une porteuse est un objet contenant une valeur, et caractérisé par les opérations par lesquelles cette valeur peut être lue ou modifiée.

On distingue en LASSO trois catégories de porteuses : les variables, les signaux et les noms de signaux.

Toutes les porteuses doivent être déclarées, sauf éventuellement les signaux scalaires internes à une entité, pour lesquels leur première apparition dans une primitive de contrôle peut servir de déclaration (afin d'alléger la description, aucune ambiguité n'étant possible). Les porteuses peuvent être initialisées lors de leur déclaration, à l'aide de la directive init.

a) Les variables

Les variables ont en LASSO la même signification que dans un langage de programmation :

- aucune restriction de type n'est associée à la notion de variable
- une variable peut prendre un nombre arbitraire de valeurs à une date simulée fixée
- une variable garde sa valeur jusqu'à ce que celle-ci soit changée.

La valeur d'une variable est modifiée par l'opérateur d'affectation :=, dans le corps d'une action. Toute variable doit être déclarée avec le type des valeurs qu'elle peut prendre.

Exemple

entier A, B[1 : 4]; booléen T[0 : 7] init 11000010;
structure(booléen X[0 : 1], caractère CHAINE[1 : 12]) Z;

b) Les signaux

Un signal ne peut prendre que des valeurs booléennes : 0, 1. Un signal garde sa valeur jusqu'à ce que celle-ci soit inversée : il passe de 0 à 1 et de 1 à 0; mais s'il contient 1, il ne peut pas recevoir 1 à nouveau. Un signal change de valeur sous l'effet de deux circonstances :

- la connexion +
- la transition d'une primitive de contrôle.

Un signal ne peut pas être en partie gauche d'une affectation.

Nous avons vu qu'on distingue 3 catégories de signaux : les signaux d'interface déclarés entrée ou sortie, et les signaux internes déclarés signal dans le corps de l'entité.

Exemple

signal ALPHA[1 : 3, 2 : 4]

c) Les noms de signaux : nomsig

Cette catégorie de porteuse a pour rôle de permettre une plus grande souplesse dans l'écriture du graphe de contrôle. La notion de nomsig en LASSO est assez proche de celle d'état en CASSANDRE et LASCAR.

Un nomsig peut prendre comme valeur un identificateur de signal.
Cette valeur peut être modifiée par une affectation dans le corps d'une action. Mais un nomsig peut être placé en entrée ou en sortie de transition : tout se passe alors, à chaque instant, comme si c'était le signal dont il contient l'identificateur qui était en entrée ou en sortie de cette transition. On est alors proche de la notion de dérepérage d'Algol 68 [VaW.76].

Exemple 1

```
entité PROCESSEUR;
  :
corps
  :
structure(entier A, caractère B) MSG;
nomsig OPERATION[0 : 3] init(PLUS, MOINS, MULT, DIVISE);
        % les 4 éléments du tableau OPERATION sont initialisés %
entier CODEOP; % valeur du code opération servant d'index dans le tableau
                 OPERATION %
% suivant la valeur du code opération ou exécute l'une des 4 transitions
  PLUS ou MOINS ou MULT ou DIV %
? DECODE ? INDEX CODEOP VALIDER(OPERATION);
? PLUS ? .....% addition %
? MOINS ? ....% soustraction %
? MULT ? .....% multiplication %
? DIV ? ......% division %
  :
  fin;        % entité processeur %
```

Exemple 2

```
signal A, B, C, D, E;
nomsig N;
1  ? A ? début ... N := C; ... fin valider(B);
2  ? B & E ? valider(D);
3  ? D ? début ... fin valider(N);
```

Si la transition 3 est effectuée après la transition 1, tout se passe comme si on avait écrit :

? D ? début ... fin valider(C);

V.3 Expressions

Une expression de type T (T = entier, caractère ou booléen) est une combinaison de variables et de constantes de type T, scalaires ou tableau, d'appels de fonction à résultat de type T et d'opérateurs définis sur T.

La priorité des opérateurs est donnée par le tableau ci-dessous (les chiffres croissants indiquent une plus forte priorité) :

0	:= ←
1	|
2	&
3	< > ≤ ≥ = ∿=
4	+ -
5	* mod div
6	↑ retard
7	‖
8	abs /<opérateur> ∿ (opérateurs unaires)

V.4 Procédures et fonctions

Une action peut être décrite par un algorithme de complexité arbitraire. Il peut donc être tout à fait nécessaire de faire appel à des procédures ou à des fonctions pour rendre plus lisible et plus compacte l'écriture de la partie algorithmique d'une description.

Procédures et fonctions ont ici le même sens que celui qui leur est donné dans les langages de programmation de type Algol.

a) Portée et visibilité des variables

Pour des raisons de cohérence vis-à-vis du modèle de synchronisation qui est à la base du langage, une procédure ne doit pas manipuler de porteuses de catégories signal ou nomsig. Elle ne peut s'appliquer qu'à des constantes et des variables.

Une procédure définie dans une entité a accès à toutes les variables déclarées dans cette entité. Mais de même que de l'intérieur d'une entité on ne connaît pas les composants d'une entité englobante, une procédure définie à l'intérieur d'une entité n'a pas accès aux variables d'une entité englobante. Les règles concernant les variables locales à une procédure sont identiques à celles qui s'appliquent dans les langages de programmation : ces variables sont inaccessibles de l'extérieur. Cela signifie en particulier que l'utilisateur ne peut pas modifier ces variables, à partir du langage de commande, en cours de simulation. Dans le cas où une variable locale à une procédure porte le même identificateur qu'une variable définie au niveau de l'entité, toute référence à cet identificateur dans la procédure affectera la variable locale.

Les règles que nous venons de donner s'appliquent de la même manière aux fonctions.

b) Définition locale ou externe des procédures et des fonctions

Une version industrielle du langage LASSO offrira un certain nombre de procédures et fonctions prédéfinies (lecture, écriture, tirage de nombres aléatoires, etc ...). Ces fonctions seront considérées comme définies par défaut dans toute entité d'une description LASSO. En ce qui concerne les procédures et fonctions écrites par l'utilisateur, elles doivent être définies dans toute entité où elles sont utilisées, ou bien déclarées externes. Une déclaration minimale doit comporter le nom de la procédure et les noms et types de ses paramètres formels éventuels : la porté d'une telle déclaration est limitée à l'entité où elle apparaît.

- définition locale :

Le texte source suit l'entête de la fonction ou procédure au moment de sa déclaration. Dans ce cas, il s'agit d'une définition locale et la fonction (ou la procédure) ne peut être utilisée que dans l'entité où elle est définie.

Exemple

```
entité A;
  :
corps
  :
procédure P(entier X, Y)
  début
   :    % définition du corps de la procédure %
  fin; % de P %
 :
fin; % de A %
```

- définition externe

Il est possible d'utiliser des fonctions ou procédures précompilées dont la définition se trouve dans une bibliothèque. Dans ce cas, le mot-clé externe permet de limiter la déclaration à la seule écriture de l'entête. Tout se passe comme si le corps de la procédure était automatiquement inséré dans le texte de l'entité. Une procédure externe peut être utilisée dans plusieurs entités.

Exemple

```
entité B;
 :
corps
 :
fonction externe Q(entier X, Y)
  :       résultat booléen R;
fin; % de B %
```

c) Corps de procédure ou de fonction

Le corps d'une procédure est délimité par les mots-clés début et fin.
Il est composé de déclarations de variables internes et d'instructions identiques à celles qui sont utilisées dans l'écriture des actions (voir paragraphe suivant). On peut en particulier à l'intérieur d'une procédure ap-

peler une autre procédure déclarée dans la même entité.

A l'intérieur d'une procédure, les instructions sont exécutées séquentiellement, comme dans un langage de programmation algorithmique de type ALGOL.

VI INSTRUCTIONS DE LA PARTIE ALGORITHMIQUE

Les actions sont décrites par une suite d'instructions séquentielles. Ces instructions sont semblables à celles que l'on peut trouver dans un langage algorithmique classique (PASCAL, ALGOL W, etc ...).

Ainsi on introduit :

VI.1 L'instruction d'affectation (symbole :=), permettant d'affecter à une variable en partie gauche la valeur d'une expression de même type et de dimensions compatibles figurant en partie droite.

VI.2 L'instruction conditionnelle

si <cond> alors <liste d'instructions> sinon
<liste d'instructions> finsi;

La partie sinon étant facultative, l'instruction conditionnelle peut se réduire à :

si <cond> alors <liste d'instructions> finsi;

La condition doit être une expression booléenne scalaire.

VI.3 Les instructions d'itération

Nous distinguons l'instruction pour de l'instruction pourtout.

pour V de E1 pas E2 a E3
début <liste d'instructions> fin

V est un identificateur de type entier scalaire qui prend successivement toutes les valeurs comprises entre E1 et E3 avec un incrément E2 (l'incrément par

défaut de 1). E1 et E3 sont des entiers positifs, E2 est un entier strictement positif.

Si E1 > E3, l'itération n'est pas exécutée. L'utilisation de l'instruction pour doit être limitée aux cas où l'ordre dans lequel l'index de boucle prend ses valeurs influe sur le résultat.
Exemple : trouver le premier élément d'un tableau vérifiant une certaine condition.

L'instruction pourtout a le même formalisme d'écriture, et les conventions sur E1, E2 et E3 sont identiques

```
pourtout V de E1 pas E2 a E3
début <liste d'instructions> fin
```

Dans ce cas, V peut prendre dans n'importe quel ordre toutes les valeurs de l'ensemble {E1, E1 + E2, E1 + 2E2, ..., E1 + pE2} avec E1 + (P+1) E2 > E3.
L'instruction pourtout sert à répéter, autant de fois que l'index de boucle prend de valeurs différentes, la liste d'instructions qu'elle contrôle.

Exemple : compter le nombre d'éléments d'un tableau vérifiant une certaine condition.

VI.4 Les instructions d'itération conditionnelles

Les itérations conditionnelles peuvent s'exprimer sous deux formes :

```
tantque <condition> faire
début <liste d'instructions> fin
```

La liste d'instructions sera exécutée tant que la condition booléenne est vraie.

```
répéter <liste d'instructions>
   :
jusqu'à <condition>
```

La liste d'instructions sera exécutée tant que la condition booléenne est fausse.

VI.5 Appel de procédure ou de fonction

Tout appel à une procédure (ou fonction) se fait par l'apparition de l'identificateur de la procédure (ou fonction) dans la partie algorithmique d'une description LASSO, ou dans le corps d'une autre procédure (ou fonction), suivi éventuellement par la liste des paramètres effectifs. Une correspondance est alors établie entre paramètres formels et paramètres effectifs par leur position respective dans les deux listes. Les paramètres formels apparaissant lors de la déclaration de la procédure sont uniquement des paramètres par valeur; ceci signifie que chaque paramètre formel déclaré est une variable locale (à la procédure ou fonction) qui sera initialisée lors de l'appel avec la valeur du paramètre effectif correspondant.

Exemple

```
entité C;
  :
corps
  entier I, J;
  :
  procédure externe P(entier X, Y);
        % déclaration de la procédure P et de ses paramètres formels X, Y %
? A ? début            % d'une action %
        :
        I := J ↑ I;
        P(I, J);  % appel procédure avec paramètres effectifs I, J %
      fin valider(B);
? B ?
...
fin; % de C %
```

VII LES SPECIFICATIONS

Les spécifications sont en LASSO le moyen d'exprimer :

1) des contraintes sur les paramètres d'une entité, pour que la description ait un sens.

2) des contraintes sur l'utilisation de l'entité, notamment en indiquant des séquences, ou des combinaisons de valeurs interdites ou au contraire imposées sur les signaux d'interface pour assurer qu'une entité est utilisée dans l'environnement pour lequel elle a été conçue.

3) des relations entre entrées et sorties, éventuellement en fonction du temps, ce qui permet d'exprimer de manière non algorithmique les fonctions réalisées par une entité, indépendamment des choix de mise en oeuvre qui sont décrits dans le corps de l'entité.

Sous ces deux derniers aspects, les spécifications d'une entité décrite en LASSO sont très semblables aux spécifications qu'un constructeur de circuits intégrés indique dans son catalogue. De plus, elles peuvent servir à effectuer des vérifications sur la description interne du modèle.

Les spécifications sont écrites dans un langage de prédicats.

VII.1 Les prédicats simples

Les prédicats simples sont des expressions booléennes indépendantes du temps. Ces expressions peuvent être :

- des comparaisons sur les paramètres d'une entité
- des comparaisons sur des données d'interface
- des expressions logiques sur des signaux d'entrée et de sortie d'interface.

Une ou plusieurs expressions booléennes peuvent être combinées à l'aide des opérateurs booléens habituels (négation, et, ou).

En outre, pour des questions de lisibilité, l'implication peut être écrite sous la forme d'une expression conditionnelle dans laquelle la partie sinon est optionnelle et vaut vrai par défaut.

$\sim A \mid B$ s'écrit aussi $A \rightarrow B$
ou bien
si A alors B finsi

Exemple 1

```
entité HORLOGE(entier PERIODE, PHASE);
interface
sortie Z;
corps
? DEP ? délai(PHASE) valider(SUITE, Z);
? SUITE ? délai(PERIODE) valider(SUITE, Z);
spécification
0 ≤ PHASE < PERIODE
fin % HORLOGE %
```

Dans cet exemple, la spécification indique quelles contraintes doivent vérifier les paramètres pour que la description soit conforme à la définition d'une horloge.

Exemple 2

Reprenons l'exemple de la mémoire du paragraphe IV.3. Les spécifications peuvent permettre d'imposer des restrictions sur l'interface, pour qu'un fonctionnement correct soit assuré.

```
entité MEMOIRE(entier N, A, B);
  :
spécification
 ∿(LEC & ECR) % pas de demandes simultanées de lecture et d'écriture %
AD ≤ N % adresse transmise non supérieure à la taille de la mémoire %
fin;
```

VII.2 Expressions de séquences

a) Notion de séquence et de vecteur temporel

Une séquence est une suite de valeurs que peut prendre une porteuse au cours du temps, à raison d'une valeur par unité de temps.

La séquence s'exprime à l'aide d'un vecteur de valeurs constantes d'un type déterminé, auquel on ajoute la valeur "." qui signifie "valeur indifférente".

Exemple

0 1 1 0 0 1 1 est une séquence de 7 valeurs booléennes
(-4, 52, 3, ., -1) est une séquence de 5 valeurs entières, la quatrième valeur étant indifférente.

L'écriture des séquences peut être abrégée, dans le cas de répétitions d'une même valeur, par l'utilisation d'expressions régulières.

Exemple

0(4) . (5) 1 (2) équivaut à
0 0 0 0 .....1 1

Nous appellerons vecteur temporel l'indication d'une suite de valeurs consécutives prises par une porteuse au cours d'instants passés consécutifs. Si $\underline{t}$ désigne l'instant courant, I et J sont deux expressions entières positives ou nulles telles que $I \geq J$, et A est l'identificateur d'une porteuse, le vecteur temporel

$$A\{\underline{t} - I : \underline{t} - J\}$$

a pour résultat la séquence de valeurs qu'a prise A au cours des dates $\underline{t} - I, \underline{t} - I + 1, \ldots \underline{t} - J$

b) Expressions de séquence

La combinaison de séquences, de vecteurs temporels et d'opérateurs de relation forme des expressions booléennes. Ces expressions, appelées expressions de séquences, obéissent aux même règles de compatibilité de types et de dimensions que les expressions provenant de la comparaison de deux tableaux.

Exemple

Soient A et B deux signaux prenant les valeurs suivantes au cours du temps :

dates	0	1	2	3	4	5	6	7	8	9
A	0	0	0	1	1	0	0	1	0	1
B	1	1	1	1	1	0	0	0	1	1

A l'instant $\underline{t} = 9$, on a les expressions de séquence suivantes :

Expression	Valeur
A{t - 4 : t} = 0 0 1 0 1	vrai
A{t - 8 : t - 5}= 0 . . 1	vrai
A{t - 8 : t - 5} = B{t - 3 : t}	vrai
B{t - 4 : t} = 0 0 0	erreur
B{t - 4 : t} = 0(5)	faux

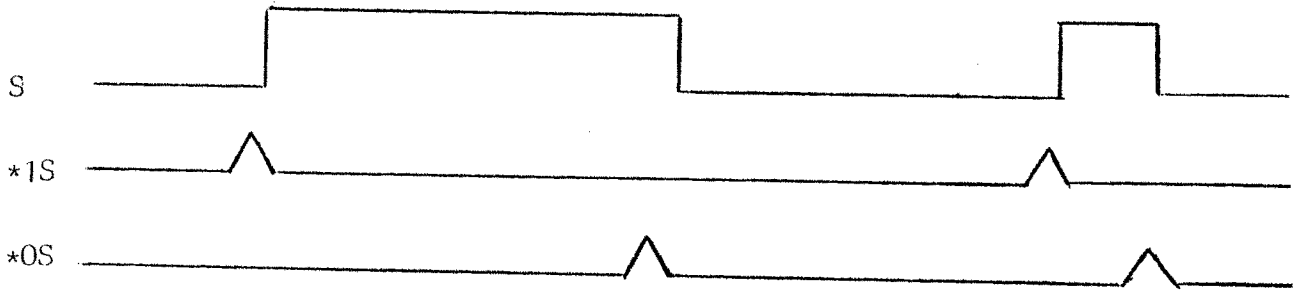
Cas particuliers

Un vecteur temporel dans lequel le mot clé t n'apparaît pas fait référence à des dates absolues, comptées à partir de la date 0.

A{2 : 7} = 0 1 1 0 0 1 a la valeur vrai dans l'exemple ci-dessus.

VII.3 Expressions d'évènements

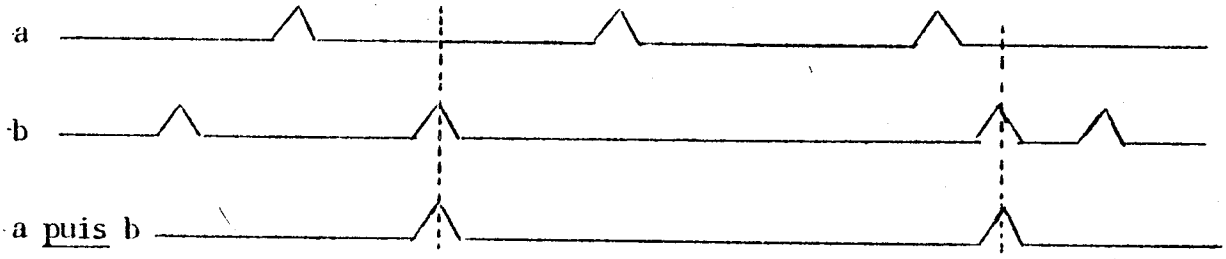
Un évènement élémentaire est le passage de 0 à 1 (noté *1) ou le passage de 1 à 0 (noté *0) de la valeur d'un signal. Un évènement a une durée nulle.



On appelle expression d'évènements, l'évènement résultant de la combinaison de plusieurs évènements élémentaires. 5 opérateurs ont été définis, admettant des évènements comme opérandes.

a) L'opérateur puis

Cet opérateur exprime la séquence de deux évènements; le résultat est un évènement qui se produit chaque fois qu'il y a eu au moins une occurrence du 1er opérande, suivie d'une occurrence du 2ème opérande.

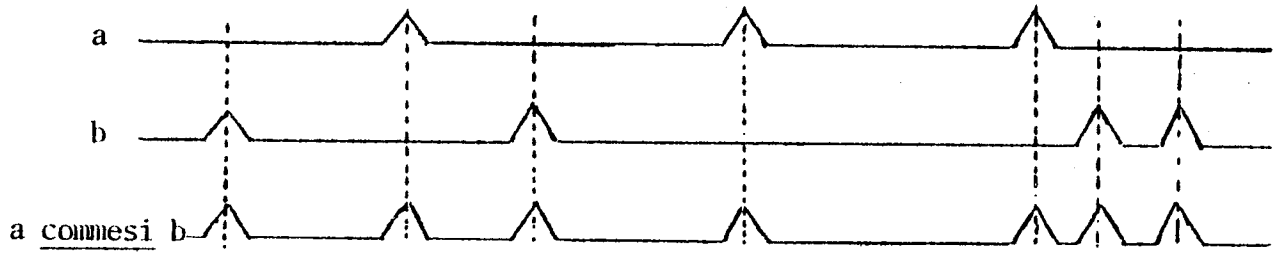


Remarque :

On ne mémorise pas le nombre de validations de l'évènement a, rencontrées avant la validation de l'évènement b.

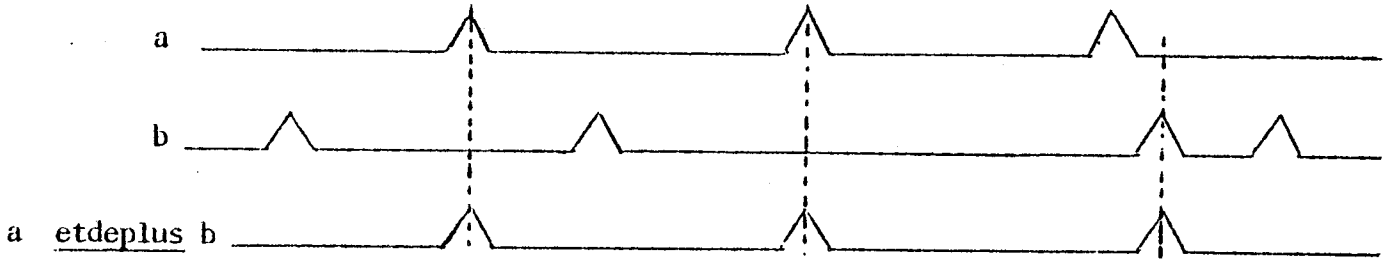
b) L'opérateur commesi

Cet opérateur engendre un évènement chaque fois que l'un ou l'autre des deux opérandes de produit; c'est en fait l'union ensembliste des évènements décrits par les deux opérandes.



c) L'opérateur etdeplus

Cet opérateur exprime la succession de deux évènements, sans considérer l'ordre dans lequel ces deux évènements sont validés, comme le montre le chronogramme suivant :



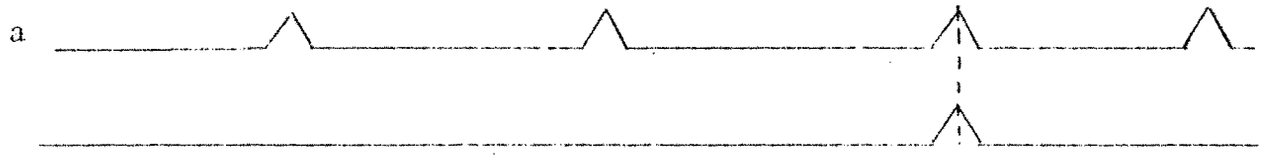
Remarque :

a etdeplus b ≠ (a puis b) commesi (b puis a)

Ceci est dû au fait que, lors de l'évaluation de l'expression a etdeplus b, on ne mémorise pas le dernière évènement validé.

d) L'opérateur lafoisno

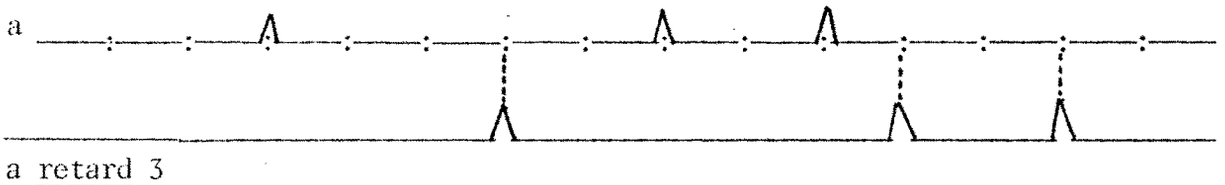
Cet opérateur entre un entier strictement positif i et un évènement sélectionne la ième occurrence de cet évènement.



lafoisno 3 que a

e) L'opérateur retard

Cet opérateur entre un évènement et un entier positif i engendre un évènement i unités de temps après chaque occurrence de son premier opérande.



a retard 3

f) Priorités des opérateurs d'évènements

2 puis, etdeplus, commesi
6 retard
8 lafoisno i que (traité comme un opérateur unaire vis-à-vis de l'évènement).

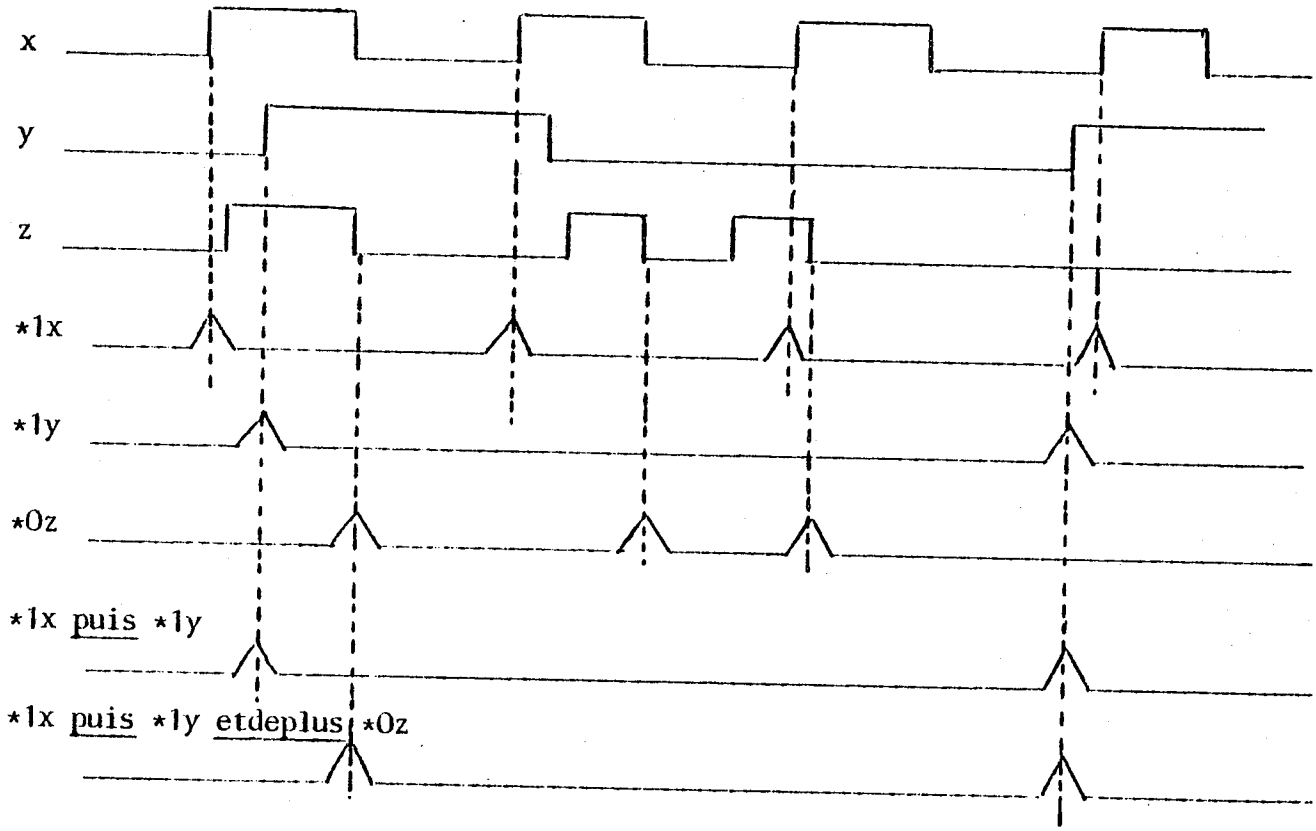
Les 3 opérateurs puis, etdeplus, commesi ont la même priorité. Si aucun parenthésage ne force l'évaluation, les expressions d'évènements sont évaluées de gauche à droite. Dans la suite, on désignera par évènement soit un évènement élémentaire, soit une expression d'évènements.

Exemple

Si x, y et z sont trois signaux d'une unité, l'évènement

*1x puis *1y etdeplus *0z

est évalué selon le diagramme suivant :



VII.4 Les prédicats composés

Les prédicats composés servent à exprimer des vérifications que l'on veut faire :

- à l'occurrence d'un évènement (unique ou répétitif)
- sur des valeurs instantannées, ou sur des vecteurs temporels.

L'occurrence de l'évènement conditionne la validité d'un ou plusieurs prédicats à vérifier. En l'absence de l'évènement condition, le prédicat composé a globalement la valeur vrai.

La détermination des évènements et de leurs dates d'occurrence s'exprime à l'aide d'instructions conditionnelles particulières, qui sont un sous-ensemble, adpaté à notre application, du langage préfixe élaboré dans le cadre d'une modélisation des phénomènes continus [Mer.78]. A la suite d'une pré-étude visant à construire formellement le langage préfixe [Bor.78], ces instructions ont été sélectionnées [BaL.79], puis affinées et mises en oeuvre [Gra.81] pour conditionner des prises de mesures automatiques sur une description écrite en LASSO.

Notion "ev" un évènement, "d" unedate fixée statiquement et exprimée par un entier positif. Sous sa forme la plus complexe, un prédicat composé est constitué de 3 parties imbriquées :

1 - une partie (optionnelle) définissant un sous-ensemble, unique ou répétitif, de la demi-droite temporelle, à l'intérieur duquel les vérifications doivent avoir lieu : c'est la fenêtre de validité des prédicats à vérifier.

2 - une partie (optionnelle) définissant une ou plusieurs dates, à l'intérieur de la fenêtre de validité, auxquelles les vérifications doivent être effectuées : ce sont les instants de validité des prédicats à vérifier.

3 - un ou plusieurs prédicats, qui peuvent être :

. des prédicats simples
. des expressions de séquence.

Si la première partie est omise, la fenêtre de validité est égale à la demi-droite des temps. Si la seconde partie est omise, les instants de validité couvrent la fenêtre de validité toute entière. Si les deux premières parties sont omises, on est en présence de prédicats simples (voir VII.1) ou d'expressions de séquences dont la validité est permanente.

a) Détermination des fenêtres de validité

a.1) Origine autre que la date 0

depuis d_1 définit la demi-droite $[d, \infty[$
alors ... fin

depuis ev_1
alors .... fin — définit la demi-droite $[date(ev_1), \infty[$

a.2) Limitation dans le temps

jusqua d_2
alors ... fin — définit le segment $[0, d[$

jusqua ev_2
alors ... fin — définit le segment $[0, date(ev_2)[$

a.3) Segment unique d'origine non nulle

depuis $\left\{\begin{matrix} d_1 \\ ev_1 \end{matrix}\right\}$ jusqua $\left\{\begin{matrix} d_2 \\ ev_2 \end{matrix}\right\}$ alors ... fin

définit le segment [date1, date2[
où, selon les cas :

date1 = d_1 ou date1 = $date(ev_1)$
date2 = d_2 ou date2 = $date(ev_2)$

Si i désigne un entier strictement positif,

depuis $\left\{\begin{matrix} d_1 \\ ev_1 \end{matrix}\right\}$ pendant i alors ... fin

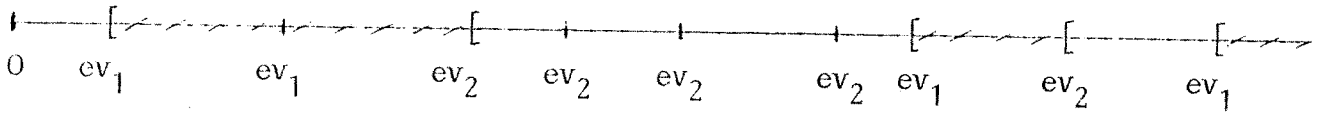
définit le segment [date1, date1+i[

Dans les 3 paragraphes ci-dessus, si ev_1 se produit plusieurs fois, seule la 1ère occurrence est considérée. De même on ne prend en compte que la 1ère occurrence de ev_2 qui suit date1.

a.4) Segments multiples

répéter depuis ev_1 jusqua ev_2 alors ... fin

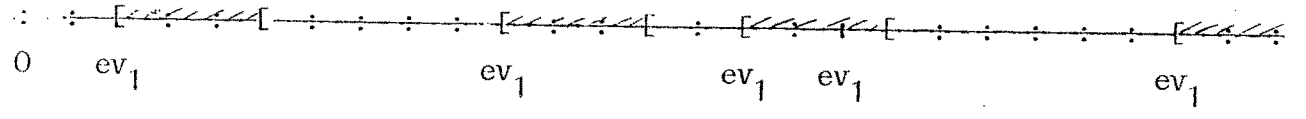
définit l'union de tous les segments de la demi-droite des temps séparant une occurrence de ev_1 de la prochaine occurrence de ev_2.



Si i désigne un entier strictement positif,

répéter depuis ev_1 pendant i alors ... fin

définit à partir de la première occurrence de ev_1 un ensemble disjoint de segments de longueur i, ayant pour origine les occurrences de ev_1 non inclues déjà dans un segment.



b) Détermination des instants de validité

b.1) L'instruction en

en $\left\{\begin{matrix} d \\ ev \end{matrix}\right\}$ alors prédicats fin

ev doit être un évènement unique. Cette instruction valide les prédicats à la date d (resp. à l'occurrence de ev).

Exemple

en 12 alors X = 'DEPART' fin,
en lafoisno 5 que *0 INITIALISE alors ... fin

b.2) L'instruction dèsque

dèsque ev alors prédicats fin

Cette instruction valide les prédicats lors de la première occurrence de ev. C'est une abréviation de

en lafoisno 1 que ev alors prédicats fin

qui est particulièrement utile pour exprimer les conditions qui doivent être satisfaites à la fin de la phase d'initialisation d'un circuit.

Exemple

dèsque *1 START puis *1 PRET alors
X = 0, Y = 1 fin

b.3) L'instruction chaque

chaque ev alors prédicats fin

Cette instruction valide les prédicats de la partie alors à chaque occurrence de l'évènement ev.

b.4) L'instruction tousles

tousles i alors prédicats fin

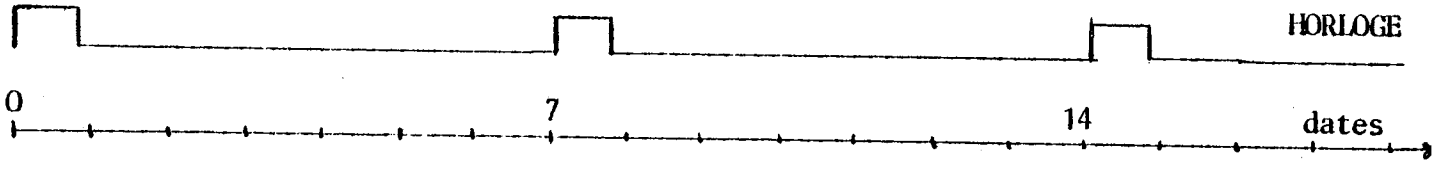
i est une constante entière strictement positive. Cette instruction valide les prédicats de la partie alors aux instants

t, t+i, t+2i, ...

où t est soit déterminé par une condition depuis englobante, soit vaut 0 par défaut.

Exemple

tousles 7 alors HORLOGE (t-6 : t) = 10(6) fin



c) Ecriture des prédicats composés

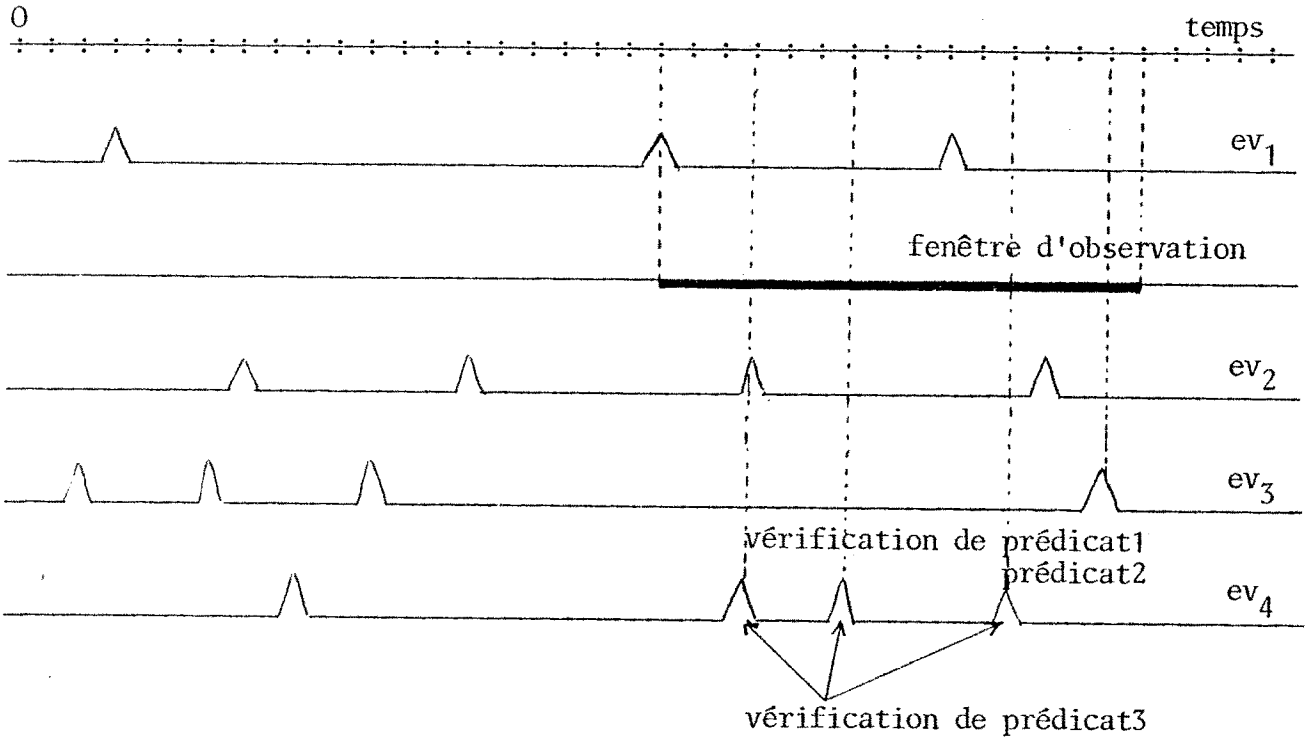
Les prédicats composés s'écrivent comme des instructions conditionnelles admettant au plus deux niveaux d'imbrication : les fenêtres de validité et les instants de validité. Un ou plusieurs prédicats peuvent être placés

sous la portée du même instant de validité. Un ou plusieurs prédicats et/ou instants de validité peuvent être placés sous la portée de la même fenêtre.

Nous avons volontairement restreint le nombre de niveaux d'imbrication, et imposé un ordre entre instants et fenêtres de validité, afin de rester maîtres de la signification de toutes les écritures autorisées (un langage plus général aboutissait à des expressions tellement complexes qu'il n'était pas, parmi les auteurs du langage, deux personnes pour leur donner la même interprétation).

Bien que l'éventail des possibilités d'expression de conditions temporelles soit réduit par rapport au langage préfixe complet, les prédicats composés ainsi définis permettent d'exprimer des conditions de vérification assez complexes, comme le montrent les exemples ci-dessous :

depuis lafoisno 2 que ev_1 pendant 15 alors
  dèsque ev_2 puis ev_3 alors
    prédicat1, prédicat2, fin,
  chaque ev_4 alors prédicat3 fin
fin



d) Exemple

L'exemple qui suit illustre comment peuvent être spécifiées les fonctions et le temps de réponse d'une unité arithmétique et logique.

```
entité UAL
interface
entier SELECTION;
bool A[1 : 16], B[1 : 16], S[1 : 16];
entrée CTL(A, B, SELECTION);
sortie R(S);
corps
 :
spécification
chaque *1 CTL puis *1 R alors
  (SELECTION = 0 → S =  ∿A),
  (SELECTION = 1 → S =  ∿(A | B)),
     :
  (SELECTION = 15 → S = A) fin;
 % spécification des 16 fonctions de l'UAL.
 cf. par exemple le boîtier INTEL SN54 181 %
chaque *1 CTL retard 48 alors
   R(t - 8 : t)  ∿= 0(9) fin;
 % validation de R au maximum 48 unités de temps après réception du signal
 d'entrée et au minimum 40 unités de temps %
fin % UAL %
```

TROISIEME PARTIE

FORMALISATION DES PRINCIPES COMMUNS AUX LANGAGES DE DESCRIPTION DES SYSTÈMES LOGIQUES :

L'APPROCHE CONLAN

AVERTISSEMENT

Les chapitres II et III de cette troisième partie sont le fruit d'un travail collectif, mené depuis cinq ans dans le cadre du projet CONLAN.

L'auteur a participé à la définition et à la mise au point de chacun des aspects présentés dans ce mémoire, participation matérialisée par une quarantaine de memoranda techniques internes au groupe de travail. La propriété intellectuelle des résultats est détenue collectivement par le groupe de travail CONLAN.

I CONCEPTS DE BASE ET TERMINOLOGIE

En programmation, un certain nombre de concepts se sont dégagés au fil des années (notions de variable, de pointeur, de procédure etc ...); tous ne sont pas simultanément présents dans un langage de programmation particulier, mais tous sont pertinents pour qui s'intéresse à l'activité de programmation et à la réflexion sur les langages qui sous-tendent cette activité.

De manière analogue, les langages de description de systèmes logiques reposent dans leur ensemble sur des notions primitives dont certaines sont communes avec les langages de programmation, d'autres sont soit beaucoup plus générales soit nettement distinctes. Ces notions ont cependant été longues à se dégager, car dans la présentation qu'ils font de leur langage, les auteurs mettent toujours l'accent sur l'utilisation des primitives qu'ils introduisent pour représenter telle caractéristique des systèmes logiques, négligeant presque toujours, sauf pour les primitives de contrôle, d'en expliciter les fondements sémantiques. Les premiers efforts de synthèse cherchant à dégager les concepts de base des langages de description datent de moins de dix ans [Bar.75, Jos.77], précédant la première publication des conclusions de l'étude collective menée dans ce but [PBB.80 a-d]. La suite de ce paragraphe a pour objet de présenter les notions primitives qui distinguent les langages de description des langages de programmation.

I.1 Valeurs et porteuses

Un texte écrit dans un langage de description définit un système à partir de relations entre des objets. Les objets les plus simples, et généralement pré-définis pour chaque langage, sont des valeurs : entiers, booléens, caractères, ternaires etc ... . D'autres objets, représentant des éléments matériels d'un système logique (registres, mémoires, fils de connexion etc ...) contiennent une valeur; nous les appellerons porteuses.

Les objets sont regroupés en fonction de propriétés, considérées comme caractéristiques de l'ensemble auquel ils appartiennent. Un type est un ensemble d'objets auquel est attribué un identificateur, et sur lequel sont définies des opérations. Ainsi, le type bool est l'ensemble (0, 1) sur lequel sont définis les opérateurs $\sim$ (non), | (ou), & (et), et les opérateurs de relation : =, $\sim$=, <, $\leq$, >, $\geq$.

Un très grand nombre de langages de description ont pour but de permettre l'écriture de descriptions simulables, faisant apparaître la structure physique d'un système, et permettant d'étudier l'évolution de ses éléments au cours du temps.

Aussi la notion de porteuse doit-elle être précisée. Une porteuse est un objet capable de contenir une valeur, d'un type déterminé, à chaque instant. La manière par laquelle une porteuse peut changer de valeur, ou conserver sa valeur, est fixée lors de sa déclaration par le type de porteuse auquel elle appartient. Ainsi, les notions de flip-flop, fil de connexion, horloge sont-elles représentées par des types de porteuses différents. La notion de variable, au sens des langages de programmation, n'est qu'un cas particulier de porteuse.

Un type de porteuse doit donc, pour être entièrement défini, comporter les informations suivantes :

- le type des valeurs que contiennent ses éléments,
- la valeur initiale, ou la valeur par défaut, en l'absence d'affectation de valeur,
- sous quelles conditions et par quel(s) opérateur(s) la valeur d'un de ses éléments est modifiée,
- quelles sont les propriétés de rémanence de cette valeur.

Anticipant sur le paragraphe II.2, considérons que chaque type de porteuse générique a deux paramètres : "t" le type de valeurs du contenu, et "d" la valeur par défaut qui est de type t. Les types ci-dessous sont caractéristiques des langages de description de systèmes logiques (la liste n'est pas exhaustive) :

TERMINAL(t, d)	fil de connexion, broche
	valeur fugitive
	contenu changé instantanément par connexion
REGISTRE(t, d)	registre, flip-flop, latch
	valeur mémorisée
	contenu changé par chargement, sur front montant (ou descendant) de condition
VARIABLE(t, d)	variable abstraite
	valeur mémorisée
	contenu changé instantanément par affectation.

En fait, la plupart des langages offrent un nombre réduit de types de valeurs et de porteuses primitifs. Et le niveau d'un langage particulier est en grande partie déterminé par ces types primitifs.

Exemple

Au niveau réseau de portes, on considère par exemple des fils à valeurs ternaires, la valeur par défaut étant indéterminée.

SIGTERN = TERMINAL(TERN, U)

Au niveau transfert de registre, les valeurs considérées sont booléennes, à la fois pour les registres et les fils de connexion.

SIG0 = TERMINAL(BOOL, 0)
SIG1 = TERMINAL(BOOL, 1)
REG0 = REGISTRE(BOOL, 0)
REG1 = REGISTRE(BOOL, 1)

A un niveau plus abstrait, on peut vouloir manipuler des valeurs entières, et non plus des fils de connexion mais des variables.

IREG = REGISTRE(INT, 0)
IVAR = VARIABLE(INT, 0)
BVAR0 = VARIABLE(BOOL, 0)

I.2 Parallélisme et séquencement

Compte tenu de la nature même des systèmes logiques, dans lesquels le déroulement simultané d'actions parallèles est la règle quel que soit le niveau d'observation, tous les langages de description offrent des primitives pour exprimer le parallélisme.

Suivant la terminologie de Barbacci [Bar.75], à présent couramment acceptée et référencée, on distingue entre langages procéduraux et non-procéduraux.

Un langage est dit non-procédural si l'ordre d'éxécution des instructions est indépendant de leur ordre d'écriture. Dans un langage de description

de systèmes logiques non-procédural, toutes les instructions, élémentaires ou composées, décrivent des action parallèles. Ainsi, l'ordre lexicographique des instructions n'a aucune signification autre que de convenance ou de lisibilité. Des groupes d'instructions peuvent être préfixés par une condition sous forme d'expression de contrôle (CDL, RTS, PHPL, ...) ou de nom d'état (CASSANDRE, LASCAR, DDL, ...). Les validations et invalidations successives de ces préfixes assurent le séquencement des actions qui en dépendent.

Un langage est dit procédural si l'ordre d'écriture des instructions influe sur l'ordre dans lequel elles sont exécutées. Un tel langage fournit des primitives pour décrire l'enchaînement des instructions, ou modifier l'enchaînement pris par défaut. En ce qui concerne les langages de description de systèmes logiques procéduraux, deux approches sont particulièrement typiques :

1 - le parallélisme et la séquentialité des instructions sont spécifiés par des séparateurs différents et un parenthésage adéquat (ISP);

2 - une action est décrite par une liste d'instructions, élémentaires ou composées, s'exécutant séquentiellement selon l'ordre dans lequel elles ont été écrites, le parallélisme et la séquentialité entre les actions sont exprimés par un graphe de contrôle distinct (DIGITEST II devenu CAP, LASSO, ...).

On voit donc que ce sont les langages de description procéduraux qui sont les plus proches des langages de programmation. Ainsi peut-on trouver des analogies entre les instructions de ISP (ou SLIDE) et les instructions collatérales d'ALGOL 68, si l'on s'abstrait des nombreuses différences syntaxiques, et si l'on inverse entre les deux langages la prédominance des séparateurs de parallélisme et de séquentialité. La ressemblance entre Pascal Concurrent [Bri.75] et ADL [Leu.79] est encore plus marquée, les deux langages incluant la notion de moniteur [Hoa.74]. De nombreux langages de description procéduraux sont en fait des extensions de langages de programmation classiques : CAP est une extension de PL/1, ADLIB est un sur-ensemble de Pascal fortement influencé par Simula, ... la liste exhaustive serait très longue.

Toutefois, une différence essentielle existe entre une majorité de langages de description procéduraux, et les langages de programmation comportant des primitives d'expression du parallélisme (on parle plus souvent dans ce deu-

xième cas de processus concurrents), malgré leurs similitudes syntaxiques. Cette différence tient précisément à la sémantique du parallélisme que l'on exprime. Dans les seconds, le programmeur est responsable d'utiliser des mécanismes complémentaires (moniteurs, sémaphores) pour assurer le déterminisme du résultat de l'exécution d'un programme subdivisé en processus concurrents. Dans les premiers, le concepteur exprime à l'aide des mêmes primitives l'activation d'actions parallèles ou séquentielles, et les protections d'accès aux objets partagés, des aides logicielles étant censées être fournies pour vérifier les propriétés attendues de sa description (déterminisme, absence de blocage, etc ...).

Cependant, avec les études sur la sémantique des programmes concurrents [Kah.79], qui visent à décrire les programmes pour pouvoir appliquer des méthodes de démonstration et de vérification sur ces descriptions, les deux activités de conception de programmes et de conception de systèmes logiques deviennent indiscernables, et utilisent les mêmes techniques.

I.3 Notions temporelles

La plupart des langages de description de systèmes logiques ayant été conçus en vue de permettre la simulation des circuits décrits, la majorité d'entre eux disposent de primitives d'expression du temps. Le temps intervient sous trois formes essentielles, qui caractérisent assez bien le niveau de modélisation.

Aux niveaux les plus abstraits, où un système est considéré comme un réseau de modules asynchrones complexes, le temps apparaît sous la forme de durées d'exécutions de processus ou d'actions. On retrouve à ces niveaux des instructions semblables à celles des langages généraux de simulation de type SIMULA, par exemple DELAY en LASSO, SLIDE, ADLIB.

Les niveaux intermédiaires (microprogramme, transfert de registre) sont ceux où l'on décrit les systèmes synchrones. La synchronisation est assurée par un ou plusieurs signaux de contrôle périodiques, appelés horloges, déclarés dans la description. Certains langages permettent de spécifier la valeur en secondes de la période de chaque horloge (DDL), d'autres non (CASSANDRE, CDL, ...). Le concepteur, lorsqu'il décrit un système synchrone, fait l'hypothèse que toute action représentée, par exemple tout chargement de registre, a une durée inférieure à la période de l'horloge dont elle dépend.

Le temps n'intervient plus explicitement dans la description du comportement d'un tel circuit, mais implicitement, par l'intermédiaire de l'écriture des conditions d'horloge, qui déclenchent les actions.

Aux niveaux les plus fins (circuits combinatoires, réseaux de portes), où l'on veut par exemple valider l'hypothèse de stabilisation du circuit en un cycle d'horloge, on retrouve des descriptions asynchrones. On s'intéresse alors aux temps de traversée des portes, ou des différentes couches d'opérateurs élémentaires, qui ont pour effet de retarder la transmission des signaux. Il est alors classique d'appliquer un opérateur de retard aux porteuses de la description, dont le résultat est la valeur de la porteuse un certain temps auparavant (PHPL, CASSANDRE Asynchrone, etc ...).

Lorsqu'un concepteur simule une description dans un but de vérification, et tout particulièrement lorsqu'il simule une description asynchrone, il s'intéresse moins aux valeurs instantanées des porteuses qu'à leurs suites de valeurs aux cours du temps. Nous avons vu que LASSO permet d'écrire des assertions sur des suites de valeurs. SLIDE, qui fut conçu à peu près en même temps, donne aussi le moyen de référencer des suites de valeurs, mais dans le corps de la description. L'historique des valeurs des porteuses d'une description nous semble être la notion centrale, commune à tous les niveaux de description et à tous les modes d'expression du temps simulé. Les phénomènes représentés dans les systèmes logiques étant de nature discrète, nous considérerons toujours dans la suite une discrétisation du temps : le concepteur exprime donc ses retards et ses délais en nombre entiers d'une unité de temps qu'il a choisie; l'historique des valeurs d'une porteuse est une séquence indexée par les entiers naturels.

I.4 Opérateurs abstraits et opérateurs matériels

A propos de la description d'un système logique, il est courant d'opposer les notions de structure et de comportement. Une description structurèlle indique la décomposition d'un système en éléments matériels ou en sous-systèmes, et leurs interconnexions.Une description de comportement, à l'inverse, spécifie l'évolution de l'état du système au cours du temps, et les relations entre ses entrées-sorties, sans indiquer comment ce comportement est matériellement réalisé. Les langages PMS et ISP [BeN.70] sont un exemple extrême de séparation de ces deux aspects. PMS ne permet de décrire que l'architecture globale d'un ordinateur, à partir de composants de base pré-

définis (mémoire, processeur, liens de données et de contrôle, transducteurs etc ...), sans qu'il soit possible de préciser sur ces composants autre chose que des attributs très généraux (par exemple : taille d'une mémoire en mots, temps moyen d'exécution d'une instruction pour un processeur). A l'inverse, ISP met l'accent sur le comportement d'un processeur qui exécute une séquence d'instructions, et sur l'évolution des mots d'état du processeur, alors que les circuits combinatoires, les chemins de données, les échanges avec la mémoire etc ... ne sont pas représentés.

Cette séparation entre description de structure et de comportement a été systématisée dans certains systèmes d'aide à la conception. A Stanford University, le simulateur SABLE [Hiv.79] traite des descriptions exprimées à l'aide de deux langages distincts : ADLIB [Hil.79] pour le comportement de chaque type de module, SDL [VaC.77] pour l'interconnexion des modules. Aux niveaux les plus détaillés, les concepteurs décrivent le plus souvent des réseaux de portes ou de composants primitifs dont les équations logiques sont pré-définies [LET.78, SzT.76], toute introduction d'un nouveau composant nécessitant soit l'écriture d'un sous-programme dans le langage d'écriture du simulateur [LET.78], soit l'intervention de l'équipe de CAO pour étendre la bibliothèque des composants primitifs.

Notre approche a été tout à fait à l'opposé de celle que nous venons d'évoquer. Les langages CASSANDRE, LASCAR et LASSO permettent des descriptions à la fois de structure et de comportement, l'utilisateur étant maître de décider de la finesse de décomposition de son système, et de définir au sein du langage choisi les modules qu'il considère comme primitifs pour son application. Cette vision, qui est celle de la plupart des constructeurs d'outils généraux, a été reprise dans CONLAN, et sera développée au paragraphe II.

C'est dans cette optique que nous envisageons la notion de description fonctionnelle, définie dans [Bar.75] comme intermédiaire entre une description structurelle et une description de comportement, en ce sens que les composants matériels d'un système logique sont représentés, mais que les opérateurs peuvent ou non correspondre à des opérateurs matériels. Par contre, le parallélisme et les aspects temporels sont pris en considération.

La synthèse d'un système logique basée sur des outils d'aide à la conception peut être caractérisée comme le passage d'une description fonctionnelle à une description structurelle, les deux descriptions devant définir le même comportement. Différentes méthodes de synthèse automatique ont été proposées, parmi lesquelles on peut citer le compilateur de hardware de CASSANDRE [BDF.75], et la méthode des macro-modules bâtie sur ISP [BaS.75]. Un logiciel de synthèse automatique est toujours construit à partir d'une hypothèse sur la réalisation des opérateurs qui apparaissent dans une description, lorsque la définition du langage ne précise rien à ce sujet. Les deux exemples cités ci-dessus partent d'hypothèses différentes. Pour le compilateur de hardware de CASSANDRE, à toute occurrence d'un opérateur dans une description correspond un élément matériel, qui est produit pendant le processus de synthèse; le concepteur a la charge ensuite d'optimiser son circuit, en supprimant le matériel pléthorique lorsque des partages sont possibles. Dans le système RT-CAD, pour lequel le langage ISPS ne donne aucune indication temporelle, on peut demander au logiciel de synthèse de proposer un maximum de parallélisme, ou à l'inverse un maximum de partage du matériel, les deux circuits résultants offrant alors des performances bien différentes. A notre connaissance, aucun de ces deux outils n'a dépassé le stade de prototype. Leur étude a toutefois l'intérêt de souligner l'importance d'une définition précise de la sémantique attachée aux invocations d'opérateurs. Pour pouvoir prétendre couvrir plusieurs stades de conception, un langage doit offrir deux catégories de primitives pour décrire les opérateurs : celles qui seront associées aux fonctions dont la traduction (matérielle ou logicielle) ou le nombre d'exemplaires n'est pas encore fixé, et celles qui représentent des fonctions pour lesquelles le nombre et la réalisation sont déjà prévus. En LASCAR et en LASSO par exemple, les appels de procédure font partie de la première catégorie, les déclarations et connexions d'unités sont dans la seconde.

II LES CHOIX DE CONLAN

II. 1 Objectifs du projet et approche choisie

Les objectifs suivants ont été retenus par le groupe de travail pour guider la définition du langage CONLAN :

- CONLAN doit permettre de décrire des systèmes logiques à différents niveaux de détail : réseaux de portes, circuits séquentiels et combinatoires, micro-programmes, processeurs complets, systèmes multi-processeurs.

- Il doit être possible, pour tout système, de décrire sa structure et son comportement.

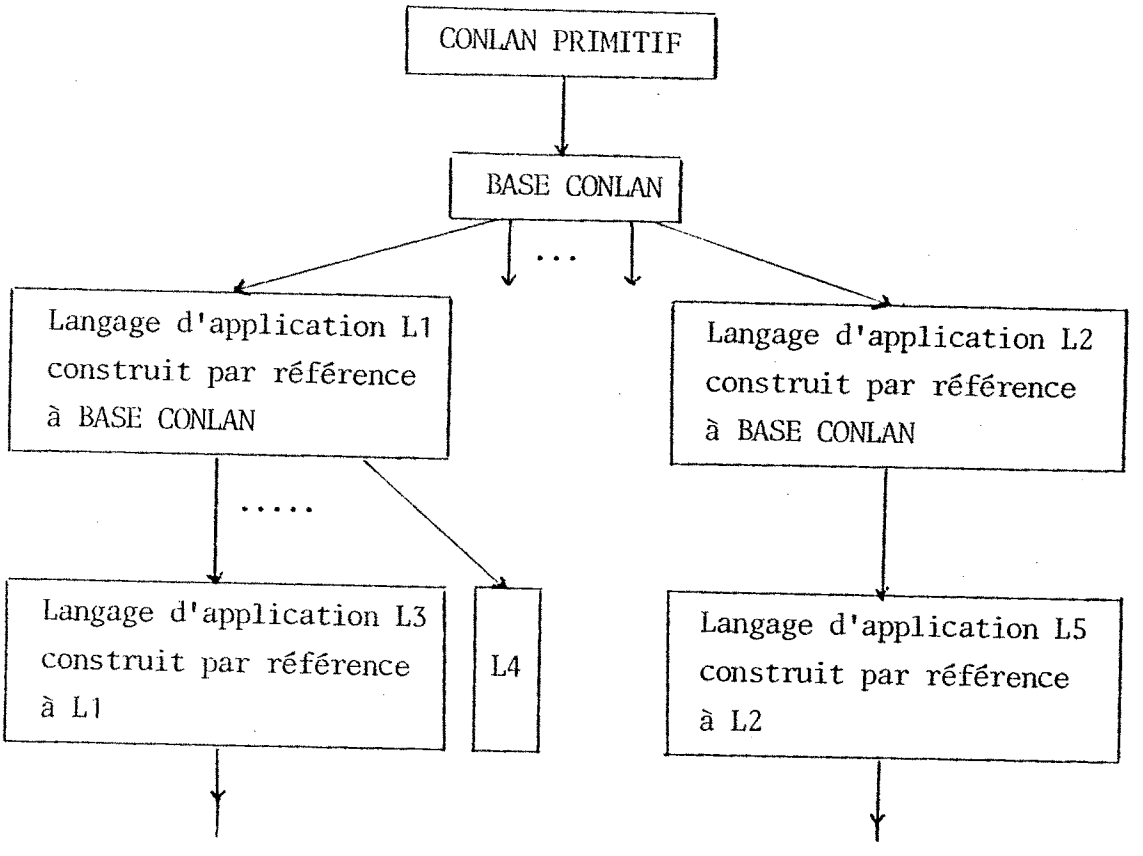
- La syntaxe et la sémantique de CONLAN doivent :

  . permettre des description non ambigües,
  . permettre une segmentation des descriptions,
  . servir de base à un système de conception assistée par ordinateur : détections d'erreurs, simulation de fonctionnement, simulation de pannes, documentation, synthèse, schémas de câblage, etc ...
  . être raisonnablement uniformes pour tous les niveaux de description.

Un langage unique ne pouvant satisfaire simultanément tous ces objectifs, il a été choisi de définir une famille de langages, dont chaque élément sera particularisé pour un niveau de description. Tous les langages possèderont un noyau syntaxique commun, et seront construits à l'aide d'un procédé uniforme d'extension, à partir d'un ensemble commun de notions primitives.

Par ailleurs, les différents langages seront obtenus non seulement par la définition de nouveaux objets, considérés comme primitifs à un certain niveau, à partir d'objets existant au niveau inférieur, mais aussi par la suppression de certains objets d'un niveau à l'autre, de manière à ne conserver, pour chacun, que les notions utiles, et contenir chaque langage dans des limites de complexité raisonnables.

Ainsi, tous les langages de la famille CONLAN sont définis à partir d'un langage de base appelé BASE CONLAN. L'ensemble de l'édifice repose donc sur un nombre limité de notions primitives, et sur des mécanismes constructifs puissants. En fait, BASE CONLAN n'est pas minimal, de manière à rendre plus aisée la dérivation de langages d'application, mais est défini lui-même à partir d'un niveau plus élémentaire CONLAN PRIMITIF, ce qui nous a permis d'évaluer et de raffiner les mécanismes d'extension. On a donc le schéma suivant :



L'approche choisie s'appuie sur les recherches de ces dernières années dans les domaines des types abstraits, des types génériques, et de leur spécification formelle. Le "sucre syntaxique" associé aux extensions est décrit par un mécanisme inspiré des études sur les langages extensibles. L'effort de formalisation auquel s'est astreint le groupe de travail devrait procurer les avantages suivants :

- la définition très précise, pour le réalisateur de simulateurs, des résultats attendus dans l'interprétation des primitives.
- la possibilité, pour des descriptions d'un même système à différents niveaux d'abstraction, de valider chaque niveau par rapport au suivant.
- la possibilité de définir par des primitives du langage, des choix de réalisation matérielle en vue d'une synthèse automatique.
- une très grande souplesse de l'ensemble, qui, par extensions, pourra intégrer de nouvelles primitives issues de l'évolution technologique, ou de l'évolution des méthodes de modélisation.

Le groupe de travail s'est fixé pour objectif de définir BASE CONLAN à partir de CONLAN PRIMITIF, et à titre d'exemple deux ou trois langages d'application, couvrant des niveaux considérés comme caractéristiques. Les autres membres de cette famille extensible seront définis par les constructeurs d'outils CAO et pourront être particularisés en fonction de la technologie, notamment chez les fabricants de circuits et d'ordinateurs, pour obtenir des programmes de traitement de descriptions plus efficaces.

Deux populations d'utilisateurs de CONLAN sont donc envisagées :

- les chercheurs, concepteurs de langages et écrivains de programmes de traitement de ces langages (compilateur, simulateur, synthèse, etc ...), qui devront connaître dans le détail BASE CONLAN et tous les mécanismes d'extension.
- les concepteurs de systèmes logiques, qui ne seront utilisateurs que d' un nombre réduit de langages d'application adaptés à leur problème, et pour lesquels il sera suffisant de connaître la syntaxe et la sémantique de ces langages d'application. Par exemple, ces utilisateurs ne pourront pas manipuler les mécanismes d'extension.

A ces deux catégories d'utilisateurs, correspondent deux catégories de textes écrits dans le formalisme CONLAN, et pour lesquels une syntaxe uniforme a été élaborée :

- définition d'un langage d'application Li. C'est la construction de nouveaux types d'objets, de nouveaux opérateurs,et de modules matériels standard, à partir d'un langage de référence Lj. Le langage Li peut contenir tout Lj, ou bien ne donner accès qu'à un sous-ensemble de primitives de Lj, plus les nouvelles primitives définies dans Li.

- définition d'un système logique. C'est le modèle, écrit à l'aide des primitives d'un langage d'application Li, d'un module matériel, ou de l'interconnexion de plusieurs modules. En fait, des descriptions écrites dans des langages différents pourront être connectées au sein d'une même description globale, si leurs interfaces sont des objets appartenant à l'intersection (au sens ensembliste du terme) des différents langages.

II.2 Notions communes à tous les langages de CONLAN

a) Notations

CONLAN est défini sur le jeu de caractères standard international (ISO 646 - International Reference Version). Les mots-clés sont notés en majuscules. Certains mots-clés, suivis du caractère "@" (noté "à" dans la suite de ce texte) correspondent à des primitives réservées à la population des constructeurs de langage. De même, les identificateurs finissant par le caractère "à" ne peuvent être introduits et utilisés que dans un texte définissant un langage, et sont ultérieurement inconnus des utilisateurs de ce langage.

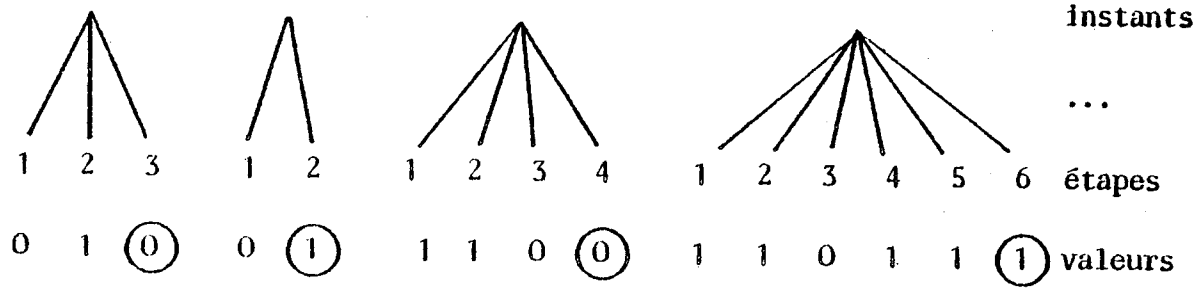
b) Modèle du temps dans CONLAN

Dans CONLAN, le temps est divisé en intervalles égaux contigüs, identifiés par les entiers positifs successifs. En général, l'unité de temps, exprimée en seconde, associée à ces intervalles, est choisie par l'utilisateur. Les "instants" d'observation du système décrit sont les frontières de ces intervalles.

En cours de simulation, plusieurs pas de calcul peuvent être nécessaires pour déterminer la valeur des porteuses de la description à chaque instant. Une deuxième subdivision existe donc, celle des intervalles en étapes de calcul, en nombre variable d'un intervalle à l'autre. Les étapes ne sont pas accessibles à l'utilisateur, mais peuvent être utilisées par le constructeur de langages pour spécifier l'algorithme d'interprétation des opérations sur les types de porteuses.

Exemple

On peut représenter par le diagramme suivant l'historique des valeurs d'une porteuse à valeurs booléennes en fonction des instants, et des étapes de calcul :



La valeur associée à chaque instant est la valeur au dernier pas de calcul.

Deux variables système, tà et sà, contiennent respectivement le numéro de l'instant et de l'étape courants.

Deux opérations, FININSTà et FINISTEPà, à définir pour chaque type de porteuse, sont automatiquement appelées par tout simulateur à la fin des calculs, de chaque instant pour la première, de chaque étape pour la seconde. Ces opérations précisent, pour les porteuses dont la valeur n'a pas été calculée au cours de l'instant ou de l'étape, quelle valeur doit leur être associée.

FININSTà et FINISTEPà définissent pour chaque type de porteuse, ses propriétés de mémorisation et ses contraintes de changement de valeur.

c) Notion de segment

Depuis la fin des années soixante, la modularité est considérée comme une propriété essentielle dans un langage de description de systèmes logiques (CASSANDRE fut l'un des premiers langages à systématiser la structuration d'un projet, grâce à la notion d'unité). Dans CONLAN, cette notion de découpage et de structuration est non seulement présente, mais étendue et unifiée entre les deux populations d'utilisateurs évoquées précédemment. Ainsi, tout texte écrit en CONLAN, qui définit un langage ou un constituant de langage, qui définit un système logique ou un sous-système, est appelé segment, dès lors qu'il forme un tout dans une optique de définition ou de traitement. La définition d'un segment :

- indique de quelle catégorie de segment il s'agit

- lui associe un identificateur
- doit préciser, pour un segment non inclus dans un autre, dans quel langage il est écrit : partie REFLAN
- est paramétrable, sauf pour une définition de langage.

On distingue 6 catégories de segments introduites par les mots-clés suivants :

TYPE : définition d'un nouveau type
CLASS : définition d'un ensemble de types
FUNCTION : opération qui renvoie un résultat
ACTIVITY : opération qui modifie un ou plusieurs objets passés en paramètres
DESCRIPTION : description d'un système, ou d'une partie de système logique
CONLAN : définition d'un nouveau langage ou d'une bibliothèque.

Tout segment s'écrit selon le schéma général suivant, les parties pouvant être absentes ou optionnelles étant indiquées entre crochets :

```
            [REFLAN identificateur-de-langage]
en-tête      <motclé> identificateur [(paramètres)]
            [ASSERT assertions ENDASSERT]
             BODY
              [définitions de segments locaux]
              [déclarations d'objets internes]
corps         [invocations d'opérations]
              [modifications syntaxiques]
             ENDidentificateur
```

<motclé> représente l'un des six mots-clés suivants :
FUNCTION, ACTIVITY, TYPE, CLASS, CONLAN, DESCRIPTION.

Nous allons à présent détailler chacune des six catégories de segments.

II.3 Fonctions et activités

Fonctions et activités sont regroupées dans CONLAN sous le terme d'opérations, en raison des nombreuses règles communes à ces deux catégories de segments. C'est en termes d'appels à des opérations qu'est décrit le comportement d'un système logique. Toutefois, ce comportement peut être représenté de manière très abstraite, ou au contraire faire apparaître des choix de réalisation, selon les porteuses déclarées et les opérations invoquées dans le corps de l'opération en cours de définition.

a) Fonctions

Une fonction est une opération qui calcule et renvoie un résultat, sans effet de bord sur son environnement. Les paramètres peuvent être des porteuses (passage par référence), ou des valeurs. Le résultat peut provenir de l'exécution d'un algorithme, ou bien du calcul d'une expression. En particulier, le concepteur de langage à accès a la forme prédicative :

THEà ident : untype WITH p (ident) ENTHE

qui renvoie l'élément du type "untype" vérifiant le prédicat p (si 0 ou plus d'un élément de "untype" vérifie p, la forme prédicative renvoie "erreur").

Il est possible d'indiquer, par extension syntaxique d'un élément non terminal de la grammaire de base, que l'appel de la fonction se fait sous une forme autre que l'écriture préfixée habituelle, par l'instruction FORMATà réservée au concepteur de langages (voir paragraphe II.6.b).

FORMATà EXTEND non-terminal règle-syntaxique
MEANS appel-préfixé ENDFORMAT

La forme générale d'une définition de fonction est la suivante, les parties optionnelles étant écrites entre crochets :

[REFLAN identificateur de langage]
FUNCTION identificateur (paramètres) : type du résultat
[ASSERT assertions sur les paramètres ENDASSERT]
[BODY
[définition de types, de fonctions et d'activités]
[déclarations de porteuses]
[invocation d'opérations]]
RETURN expression
[FORMATà définition du format d'appel]
ENDidentificateur

b) Activités

Une activité ne renvoie pas de résultat mais modifie un ou plusieurs objets de l'environnement d'appel. Cette modification peut être immédiate (rajouter

un élément à la suite des valeurs d'une porteuse au cours d'une étape), prendre un certain temps (changement d'état d'un circuit sous l'action d'un signal d'activation), ou bien être continuement effectuée (description fonctionnelle d'un circuit combinatoire). C'est dire qu'une activité regroupe les notions de procédure et de processus.

On a donc, pour une définition d'activité, la forme générale suivante :

```
[REFLAN identificateur de langage]
ACTIVITY identificateur (paramètres)
[ASSERT assertions sur les paramètres ENDASSERT]
BODY
 [définitions de types, de fonctions, d'activités]
 [déclarations de porteuses ]
 invocation d'opérations
 [FORMATà définition du format d'appel]
ENDidentificateur
```

c) Opérations statiques, opérations dynamiques

Les langages généraux de simulation (SIMULA, SIMSCRIPT ...) et certains langages de programmation (PASCAL concurrent) font la distinction entre les notions de procédure et de processus, mais curieusement offrent une seule notion de fonction. Une approche différente a été prise dans CONLAN, qui tend à moins mettre l'accent sur le mode d'invocation (instruction/expression) que sur les propriétés de nombre d'exemplaires et de mémorisation.

Une opération est dite statique si un exemplaire de l'opération est créé pour chaque occurrence textuelle d'invocation, la durée de vie de cet exemplaire et de toutes ses porteuses locales étant égale à celle du modèle tout entier.

Une opération est dite dynamique si un exemplaire de l'opération est créé à chaque activation d'une occurrence d'invocation, la durée de vie de cet exemplaire et de toutes ses porteuses locales étant limitée à l'activation en cours.

La distinction entre ces deux notions est motivée par la nécessité de faire apparaître deux points de vue. Schématiquement, le déroulement intemporel d'un algorithme sera représenté par l'invocation d'une opération dynamique; inversement, lorsque chaque invocation sera perçue comme correspondant à la mise dans le système d'un nouveau processus, auquel on ne veut pas nécessairement attribuer un identificateur, c'est une opération statique qui sera appelée.

En termes informatiques, cette dichotomie se traduit par des propriétés bien différenciées.

Opération statique :

- les objets locaux sont alloués statiquement
- les porteuses locales à mémorisation conservent leur valeur d'une invocation à la suivante
- l'exécution de l'opération peut avoir une durée non nulle
- l'opération ne peut pas être récursive
- dans le corps de l'opération, on peut trouver des invocations d'opérations statiques et d'opérations dynamiques.

Opération dynamique :

- les objets locaux sont alloués dynamiquement, à chaque invocation
- toutes les porteuses locales sont initialisées à leur valeur par défaut, à chaque invocation
- l'exécution de l'opération a une durée nulle
- l'opération peut s'invoquer récursivement
- dans le corps de l'opération, on ne peut trouver des invocations d'opérations que dynamiques.

Syntaxiquement, on distingue les opérations statiques des opérations dynamiques en préfixant les premières du mot-clé STATIC dans la définition du segment.

Applications

- Dans un langage tel que LASSO, les primitives du graphe de contrôle correspondent à des activités statiques, tandis que les procédures de la partie algorithmique correspondent à des activités dynamiques.

- Un générateur de nombres aléatoires sera défini par une fonction statique. A l'inverse, toutes les opérations booléennes et arithmétiques usuelles sont définies par des fonctions dynamiques.

- Dans une description de système logique de niveau réseau de portes, où chaque invocation d'opération entre signaux logiques correspond à l'existence d'une porte, les primitives invoquées doivent être définies comme des activités statiques.

d) Passages de paramètres

La liste de paramètres formels qui suit éventuellement le nom d'une opération est "fortement typée". Par ce qualificatif, nous voulons dire que pour chaque paramètre il faut préciser :

- son type
- son mode de liaison
- les droits d'accès.

d.1) Désignations

La définition d'opérations entrant pour une part essentielle dans la définition de tout langage de la famille CONLAN, nous avons tenu à mettre à la disposition du concepteur de langage toute la puissance synthétique possible, étendant les notions de fonctions génériques telles qu'elles apparaissent dans ALPHARD et CLU par exemple.

Ainsi, un paramètre formel peut représenter :

- un objet d'un type
- un type d'une classe
- une opération de fonctionalité fixée.

Exemple

Supposons que l'on dispose du type vecteur (t), où t désigne un type quelconque, l'indexation des éléments du vecteur se faisant par entiers successifs croissant de 1 à la taille du vecteur. Supposons de plus que "value" désigne la classe des types de valeurs. La fonctionnelle "réduction", au sens d'APL,

permettant d'appliquer de manière répétitive une opération interne sur les éléments du vecteur sera définie en CONLAN par la fonction :

```
FUNCTION reduction (ATT t : value;
    x : vecteur(t); ATT f : FUNCTION(t, t) : t) : t
RETURN
    IF taille(x) = 1 THEN x
    ELSE f(x[1], reduction(t, x[2 : taille(x)], f))
    ENDIF
ENDreduction
```

Règle de substitution uniforme

Si un identificateur apparaît plusieurs fois dans une liste de paramètres formels, toutes les occurrences de cet identificateur doivent être liées au même paramètre effectif.

Dans la fonction réduction ci-dessus toutes les occurrences de "t" identifient un même type. Des invocations particulières de cette fonction peuvent être :

```
reduction(bool,(. 0, 1, 1, 1, 0 .), and)
reduction(bool,(. 1, 0, 1, 1 .), or)
reduction(int,(. -4, -2, 0, 1, 2 .), plus)
```

Cet exemple fait apparaître trois désignations de paramètres :

value, nom de classe, désigne un type
vecteur(t), nom de type, désigne un élément
FUNCTION(t, t) : t désigne une fonction à deux paramètres de type t, à valeurs dans t.

On dispose en outre de la désignation ACTIVITY (liste de désignations) pour spécifier un paramètre formel.

d.2) Liaisons

La valeur d'un paramètre peut avoir une influence sur la structure de données interne d'une opération. Pour une opération STATIC, un tel paramètre doit avoir une valeur constante.

Nous appelons attribut un paramètre formel pour lequel une liaison statique est imposée pour toute invocation de l'opération. Un tel paramètre est précédé du mot-clé ATT. En particulier, pour des raisons de simplicité des programmes de traitement, et de clarté des définitions, CONLAN impose que les paramètres désignés par un nom de classe, par FUNCTION et par ACTIVITY soient des attributs.

Exemple

Dans la fonction réduction précédemment définie, les paramètres t et f doivent être connus statiquement; à l'inverse, x peut résulter d'une expression sur des porteuses dont le contenu varie dynamiquement, en particulier par une opération de dé-repérage (au sens d'Algol 68), selon les règles exposées en d.3).

Ainsi, si a est une porteuse dont la valeur est un vecteur de booléens, on peut écrire

réduction(bool, a, and)

mais l'invocation

réduction(bool, a, IF a[1] THEN and ELSE or ENDIF)

sera refusée car on ne peut pas déterminer statiquement si c'est une réduction par rapport à la fonction "and" ou par rapport à la fonction "or" qui doit être effectuée.

d.3) Droits d'accès

Les paramètres qui ne sont pas des attributs sont nécessairement désignés par un type. Il n'est alors pas indifférent d'employer comme désignation un type de valeur ou un type de porteuse : dans le premier cas, le paramètre effectif doit être traité comme un objet non modifiable dans le corps de l'opération; dans le second cas, le paramètre effectif est un objet pour lequel des opérations de modification de contenu sont disponibles. Il s'agit là d'une généralisation de la distinction, déjà définie en Algol 68, entre les modes t et rep(t).

Cependant, la manière de désigner les paramètres formels n'est pas suffisante pour couvrir à la fois les nécessités de vérifications de types et de restrictions de droits d'accès. En effet, on peut vouloir indiquer qu'une opération admet comme paramètre une porteuse, mais que l'opération ne modifie pas le contenu de la porteuse. C'est le cas en particulier de la fonction retard(x, i) qui délivre comme résultat la valeur que contenait la porteuse "x" il y a "i" unités de temps. On distingue donc deux catégories de paramètres parmi ceux qui ne sont pas des attributs : les paramètres modifiables et les paramètres non-modifiables :

- tout paramètre désigné par un type de valeur est non-modifiable
- tout paramètre d'une fonction désigné par un type de porteuse est non-modifiable (nous avons vu qu'une fonction ne faisait pas d'effet de bord).

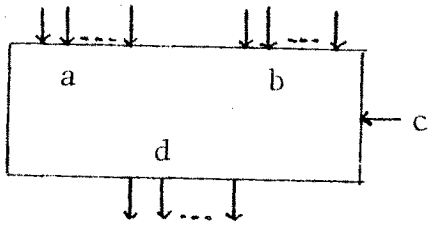
En CONLAN, un paramètre formel d'activité que l'on veut modifiable est précédé du mot-clé W ("writable"). Par défaut, un paramètre est non-modifiable.

Remarques

- Un paramètre spécifié modifiable peut n'être pas modifié : la présence du W indique à la fois un droit d'accés et une potentialité, mais non un effet certain.
- Pour chaque invocation d'une activité, l'identification des objets de son environnement qui ont été passés comme paramètres effectifs en position W détermine l'ensemble maximal des objets éventuellement modifiés. Ainsi, si une même porteuse est passée deux fois comme paramètre effectif d'une invocation d'activité, une fois en position W et une fois en position non-modifiable, cette porteuse a pu être modifiée.

Exemple

Dans un langage de niveau circuit combinatoire dans lequel SIGO est le type "fil de connexion à valeurs booléennes et valeur par défaut 0", le multiplexage entre deux chemins de données a et b de n bits, à durée de traversée i unités de temps, peut être défini par l'activité "multiplex" suivante, qui ne modifie que la sortie d. "FAN" et "%" désignent respectivement les opérateurs de "fanout" et de retard.




```
STATIC ACTIVITY multiplex
    (ATT n,i : int; a[1 : n], b[1 : n], c : SIGO;
     W d[1 : n] : SIGO)
BODY
    d .= a % i &(c FAN n) % i |
         b % i &¬(c FAN n) % i
ENDmultiplex
```

Liaison paramètre formel/paramètre effectif

Lors de l'invocation d'une opération, les paramètres effectifs qui ne sont pas des attributs sont évalués statiquement lorsque cela est possible, sinon dynamiquement au début de l'interprétation de l'opération. Le type de chaque paramètre effectif est comparé au type du paramètre formel correspondant, pour déterminer les conditions d'erreur ou les conversions éventuelles, en fonction des droits d'accés indiqués dans la définition de l'opération. Les règles, exprimées de manière informelle, sont les suivantes :

paramètre formel	paramètre effectif	mode de liaison
porteuse	porteuse	Le paramètre effectif doit appartenir au type désignant le paramètre formel, sinon une erreur est détectée. Si le paramètre est modifiable, les paramètres formel et effectif sont rendus synonymes pour la durée de l'exécution de l'opération (passage par référence). Si le paramètre est non-modifiable, une porteuse locale est créée pour le paramètre formel. On affecte à cette porteuse locale une valeur égale à celle du paramètre effectif (passage par valeur).

porteuse	valeur	Si le paramètre est modifiable, une erreur est détectée. Si le paramètre effectif n'appartient pas au type du contenu du paramètre formel, une erreur est détectée. Une porteuse locale est créée pour le paramètre formel. On lui affecte comme valeur le paramètre effectif.
valeur	porteuse	Si le type du contenu du paramètre effectif n'est pas identique au type du paramètre formel, une erreur est détectée. Le paramètre formel est rendu synonyme du contenu du paramètre effectif, pour la durée de l'exécution de l'opération (dé-repérage du paramètre effectif)
valeur	valeur	Si le paramètre effectif n'appartient pas au type du paramètre formel, une erreur est détectée. Le paramètre formel est rendu synonyme du paramètre effectif, pour la durée de l'exécution de l'opération.

Cet ensemble de règles est formalisé au sein du modèle d'évaluation exposé dans la quatrième partie de ce mémoire.

e) Le qualificatif INTERPRETERà

Certaines opérations, définies dans l'environnement d'un segment CONLAN de définition d'un nouveau langage, sont précédées du mot-clé INTERPRETERà. Ce qualificatif signifie que l'invocation de ces opérations est automatiquement effectuée au cours de l'interprétation d'une description écrite en un langage de la famille CONLAN.

Toutes ces opérations s'appliquent aux porteuses, ou à leurs historiques de valeurs. Elles assurent notamment la fonction de dérepérage (contentà), et les processus de croissance uniforme des historiques de valeurs de toutes les porteuses d'une description (shrinkà, finstepà, finintà). Le modèle d'évaluation de CONLAN suppose l'existence de ces opérations pour tous les types de porteuses de tous les langages dérivés de BASE CONLAN. Des exemples de définition de ces opérations figurent dans les types "terminal", "variable", "rtvariable" à la fin de cette troisième partie.

II.4 Les types et les classes

L'un des mécanismes fondamentaux permettant d'étendre le répertoire d'objets disponibles pour décrire des systèmes est la définition de types. Traditionnellement, les langages de description sont définis avec un répertoire limité et figé de types de valeurs et de types de porteuses. Dans CONLAN, la nécessité de définir, pour chaque niveau de modélisation, des types de valeurs et des types de porteuses appropriés, sans pour autant multiplier les catégories de segments serv nt à ces définitions, nous a conduits à unifier en un même segment l'approche mettant l'accent sur la définition de nouveaux types de valeurs, symbolisée par les ensembles énumérés de PASCAL [JeW.74], et l'approche privilégiant la définition de nouveaux constructeurs, symbolisée par la partie représentation dans ALPHARD [LSW.76a, LSW.76b] et CLU [LSA.77].

Un type est donc constitué d'un ensemble d'objets, appelé son domaine, ainsi que d'une ou plusieurs opérations définies sur ce domaine. La définition d'un type se fait à l'aide d'un segment TYPE, qui permet de lui attribuer un identificateur, et dans lequel sont regroupées la définition de son domaine et des opérations.

Un segment TYPE est paramétrable, les paramètres pouvant être des valeurs ou des types; nous disposons donc de types génériques au sens de P. Jacquet [Jac.78], le texte du segment définissant alors non pas un type unique, mais une famille de types.

Un segment TYPE a donc la structure suivante :

```
TYPE identificateur(paramètres)
BODY
  définition du domaine
  liste de définitions d'opérations
ENDidentificateur
```

a) Domaine d'un type

Le domaine d'un type est un ensemble énumérable d'objets. Pour définir un domaine on dispose en CONLAN des constructeurs et opérateurs de base de la théorie des ensembles, la définition des éléments d'un type n'étant pas éloignée, nous l'avons découvert plus tard, de la vision adoptée dans le langage

SETL [KeS.75] pour exprimer les structures de données abstraites. Un domaine peut être défini de 3 façons :

1 - Par énumération de ses éléments
{1, 2, 3, 4} {'A', 'XY'}

2 - Comme le sous ensemble de tous les éléments d'un domaine précédemment défini qui satisfont un prédicat.

Exemple :

Les réels positifs sont définis mathématiquement par $\{x \in R / x \geq 0\}$, ce qui s'exprime en syntaxe CONLAN : ALL x : real WITH x ≥ 0 ENDALL

3 - Comme identique au domaine d'un type déjà défini, éventuellement en fixant les paramètres effectifs si le segment définissant le type était paramétré. Dans ce mode de définition, l'identificateur du segment TYPE, suivi de la liste des paramètres effectifs, désigne à la fois le type et son domaine, le contexte étant toujours suffisant pour qu'aucune ambiguïté ne soit à craindre.

Exemple:

Le segment TYPE tytupleà(t : anyà)
BODY
:
END

définit le type générique tytupleà qui représente dans CONLAN toutes les suites finies d'éléments d'un même type t; anyà désigne un paramètre "type".

Dans un contexte approprié :

tytupleà(int) désigne le domaine des suites finies d'entiers
tytupleà(real) désigne le domaine des suites finies de réels.

Les modes de définition 2 et 3 peuvent se combiner, pour définir un sous-ensemble du domaine d'un type :

Exemple :

```
ALL x : tytupleà(real) WITH sizeà(x) < 5
ENDALL
```

définit l'ensemble des suites de réels de longueur inférieure à 5.

Si A et B sont deux types tels que le domaine de A est défini à partir de celui de B par l'un des modes 2 ou 3 ci-dessus, on dira que le type A dérive directement de B; on notera A <|| B.

Si un type A a son domaine défini par énumération, selon le mode 1 ci-dessus, on dira que A dérive directement du type univers : univà.

b) Opérations d'un type

Les fonctions et activités opérant sur les éléments d'un type A sont définies à partir des opérations disponibles sur le type B dont A dérive directement. Si A dérive directement de univà, les seules opérations disponibles sont les deux fonctions de comparaison à valeur booléenne "=" et "∿=".

Les fonctions et activités d'un type A sont définies dans le segment TYPE définissant A de deux façons :

b.1) Opérations transportées

Si A dérive directement de B, les opérations de B sont a priori non définies sur A; pour pouvoir en disposer sur les éléments de A, il faut les transporter explicitement, à l'aide de l'instruction :

CARRY liste_d'opérations ENDCARRY.

Chacune des opérations listées entre CARRY et ENDCARRY est automatiquement redéfinie, en conservant sa signification, de la manière suivante :

- tous les paramètres et le résultat éventuel qui étaient précédemment de type B sont typés A, les autres paramètres étant inchangés

- l'opération sur A est définie comme l'invocation de l'opération de même nom sur B, après conversion de type des paramètres appropriés (voir ci-dessous).

Toutes les opérations transportées sont exportées par le type A, c'est-à-dire connues à l'extérieur du segment TYPE qui définit A.

La directive CARRY est donc un mécanisme permettant une généricité incrémentale des opérations, au sens de [Jac.78], mais garantissant que la même opération, définie sur un ensemble de types dérivant directement les uns des autres, conserve un nombre constant de paramètres, et une signification uniforme.

b.2) Opérations définies explicitement

Si A dérive directement de B, il est possible de définir sur A de nouvelles opérations par rapport à celles de B. Ces opérations sont construites à l'aide d'opérations déjà définies sur A, ou transportées depuis B, ou bien disponibles sur B mais non sur A.

Dans cette dernière alternative, où une opération de B est invoquée avec pour paramètre un objet de A, ou bien à l'inverse où le résultat d'une opération déclaré de type B est voulu comme un élément de A, il faut procéder à une conversion de type. Ces conversions ont un but analogue aux types "rep" et "cvt" de CLU [LSA.77],et s'expriment en CONLAN par les fonctions "new" et "old".

Plus précisément, si A $<\!\!\|$ B, a $\in$ A, b $\in$ B,
new(b) convertit b en l'élément identique du domaine, considéré de type A.
old(a) convertit a en l'élément identique du domaine, considéré de type B.

Les fonctions "new" et "old" sont autorisées uniquement dans le contexte d'une définition de type, c'est-à-dire dans le corps d'un segment TYPE. Elles ne modifient pas leur paramètre, mais changent seulement son type. Elles sont applicables uniquement entre le type en cours de définition et le type dont il dérive directement. Ainsi

new(old(new(b))) = new(b)
mais new(new(b))
old(old(a)) n'ont pas de signification.

Les opérations définies explicitement dans le corps du segment TYPE peuvent être définies comme locales à ce segment TYPE : leur définition est alors précédée du mot-clé PRIVATE. Par défaut, ces opérations sont exportées par le TYPE.

b.3) Exemple

Reprenons le type générique tytupleà(t : anyà) des suites finies d'éléments d'un même type t, numérotés de 1 à la longueur de la suite. On suppose définies les fonctions :

- = : tytupleà(t) × tytupleà(t) → bool

  Deux suites sont égales si elles ont même nombre d'éléments, et leurs éléments de même numéro deux à deux identiques

- ~= : tytupleà(t) × tytupleà(t) → bool

  Inégalité : négation de l'égalité.

- sizeà : tytupleà(t) → int

  Renvoie le nombre d'éléments de la suite

- selectà : tytupleà(t) × int → t

  Sélection d'un élément de la suite.
  Cette fonction renvoie un message d'erreur si son 2° paramètre est négatif, ou s'il est supérieur à la taille du 1° paramètre.

On peut, à partir du type tytupleà(int) des suites finies d'entiers, dériver le type "range" des intervalles d'entiers, construits comme des suites finies d'entiers consécutifs, en transportant 3 des fonctions disponibles sur tytupleà(int) et en définissant de nouvelles fonctions lbound et rbound ayant pour résultat respectivement la borne gauche et la borne droite de l'intervalle.

```
TYPE range
 BODY
ALL x : tytupleà(int) WITH
   FORALLà i : int IS i ≤ 0 | i > sizeà(x)|
      selectà(x, i) = selectà(x, 1) + i - 1 ENDFOR|
   FORALLà i : int IS i ≤ 0 | i > sizeà(x)|
      selectà(x, i) = selectà(x, 1) + 1 - i ENDFOR
ENDALL
CARRY =, ∿=, sizeà ENDCARRY
FUNCTION lbound(y : range) : int
  RETURN selectà(old(y), 1) ENDlbound
FUNCTION rbound(y : range) : int
  RETURN selectà(old(y), sizeà(y)) ENDrbound
ENDrange.
```

Dans la définition de "range" ci-dessus, le domaine est défini comme le sous ensemble des suites d'entiers consécutifs ascendants (chaque élément est égal au premier de la suite augmenté de son écart au premier) ou descendants.

Dans l'instruction CARRY, les fonctions =, ∿= et sizeà définies sur tytupleà(int) sont redéfinies sur range.

Pour définir les fonctions lbound et rbound, il est nécessaire de convertir l'intervalle y en un élément de tytupleà(int) par old, puisque selectà attend un premier paramètre de ce type. Par contre, dans rbound, sizeà s'applique directement, puisqu'il a été transporté.

Le transport de la fonction sizeà est équivalent à l'écriture, dans le corps du type range, de la définition suivante :

```
FUNCTION sizeà(y : range) : int
  RETURN tytupleà(int).sizeà(old(y)) ENDsizeà
```

dans laquelle la fonction sizeà du type tytupleà(int) est explicitement invoquée, par notation pointée, avec comme paramètre le résultat de la conversion de y.

c) Relations entre types

c.1) Sous-type

Un type A est un sous-type d'un type B, ce qui est noté $A \subset B$, si

1 - le domaine de A est un sous-ensemble du domaine de B

2 - toutes les opérations définies sur B sont aussi définies sur A; dans le cas des fonctions, le type du résultat est identique au type défini pour l'opération sur B (pas de conversion).

3 - aucune opération nouvelle n'est définie sur A.

Exemple :

```
SUBTYPE pint
  BODY ALL x : int WITH x > 0 ENDALL
ENDpint
```

Ce texte définit "pint" comme le sous type de "int" constitué des entiers strictement positifs. L'ensemble des opérations arithmétiques est disponible sur pint. En particulier, le résultat de la soustraction de deux éléments de pint n'appartient pas nécessairement à pint mais est bien un élément de int (point 2 de la définition).

c.2) Type dérivé

Un type A est dérivé d'un type B, $A <| B$, si

$A = B$

ou $A <|| B$

ou il existe un ensemble fini de types $t_1, t_2, \ldots t_n$ tels que $t_1 = A$, $t_n = B$, et $\forall i \in \{2, 3, \ldots n\}$ $t_{i-1} <| t_i$

Propriétés

1 - La relation $<|$ est la fermeture transitive de la relation $<||$.
Elle confère à l'ensemble des types une structure d'ordre partiel dont le plus grand élément est univà.

2 - $A \subset B \Rightarrow A <| B$

c.3) Relation de partage de définition et dérivation canonique

Nous dirons que deux types A et B partagent leur définition si il existe un type C tel que :

$A \subset C$ et $B \subset C$

ou bien

A et B sont deux types issus d'un même segment TYPE générique.

Nous noterons $\square$ la fermeture transitive de la relation ainsi définie.

$\square$ est une relation d'équivalence.
L'ensemble quotient des types par $\square$ est en fait l'interprétation de l'ensemble des segments TYPE qui ont été définis.
La relation de dérivation canonique $\boxed{<|}$ définie sur cet ensemble quotient à partir de la relation $<|$ est aussi une relation d'ordre partiel, dont le plus grand élément est {univà}.

L'intérêt de cette nouvelle relation est que, à chaque stade de définition de CONLAN, l'arbre qu'elle détermine a un nombre fini de noeuds, contrairement à la relation $<|$.

c.4) Utilité pratique de ces relations

La distinction entre la relation de sous-type et la relation de dérivation est motivée par les vérifications de compatibilité de type qui doivent être réalisées lors de toute invocation d'opération avec des paramètres effectifs.

Un sous-type A d'un type B n'est pas considéré comme distinct de B, vis-à-vis des contraintes de compatibilité de type. Toutes les opérations invoquées sur des éléments de A sont en fait exécutées dans B. Une expression peut donc sans inconvénient être composée d'une combinaison d'éléments de A et de B; si

le contexte impose que le résultat d'une opération soit dans A, alors qu'il est a priori dans B, un test supplémentaire a lieu. On voit sans peine le gain de souplesse obtenu par une telle vérification a posteriori, par rapport aux contraintes d'un langage tel que PASCAL.

Exemple :

Si v a été déclaré variable à valeurs dans pint, et si c est une constante déclarée dans pint et égale à 12 :

```
v := (-2) * c + 100
```

est une instruction correcte, alors que

```
v := (-2) * c
```

est erronnée.

Dans les deux cas, l'expression en partie droite de l'affectation est évaluée sans erreur dans int. Dans le premier cas, le résultat étant positif, l'affectation a lieu sans erreur; dans le deuxième cas, le résultat est un entier négatif, ce qui provoque une erreur au niveau de l'opération d'affectation.

En fait, comme nous le verrons plus loin, l'affectation est définie comme un segment ACTIVITY dans CONLAN. L'exemple ci-dessus est l'illustration de ce que les erreurs de types sont toujours détectées au moment d'un passage de paramètre.

Contrairement à la relation de sous-type, la relation de dérivation n'autorise aucune combinaison d'opérandes de types différents, même si un opérateur a été transporté. Par contre, la relation de dérivation communique, pour les types de valeurs , les dénotations de constantes. Une nouvelle dénotation de constante pouvant être définie dans un type T, cette dénotation est transmise automatiquement à tous les types dérivés de T, mais ne s'applique jamais aux types dont T dérive. A toute dénotation de constante correspond donc, dans l'arbre de dérivation canonique, un nombre fini d'éléments maximaux des ensembles de types auxquelles elle s'applique. En pratique, on considèrera l'arbre de dérivation privé de son plus grand élément {univà}, afin de limiter plus encore le domaine d'application des dénotations primitives. Cette propriété est directement utilisable pour la vérification du type des constantes qui sont des paramètres effectifs d'opérateurs.

d) Notion de classe

Nous avons vu qu'un même segment TYPE, s'il est paramétré, définit une famille de types. Dans le cas d'un segment TYPE générique, on peut vouloir imposer qu'un type effectif, substitué à un paramètre formel, ne soit pas n'importe quel type, comme l'autorise la désignation "anyà", mais appartienne à un sous-ensemble des types dérivables à partir de univà.

Par exemple, dans la définition d'un langage particulier, on interdira la manipulation de pointeurs en restreignant les types de valeurs des porteuses (la plupart des langages de description des systèmes logiques sont sans pointeurs).

Cette nécessité de définir des restrictions sur les types a déjà été ressentie à propos des types génériques dans les langages de programmation. La notion de propriété [Jac.78, Ber.79] est un exemple de mécanisme permettant de regrouper des types définis indépendamment les uns des autres, mais possédant certaines fonctions liées par des structures algébriques. Cette orientation algébrique des propriétés est bien adaptée à la caractérisation des types de valeurs, mais ne permet pas d'exprimer des relations significatives sur les types de porteuses; ce concept s'est donc révélé insuffisant pour notre application.

En CONLAN, une classe est un ensemble de types sur lequel peuvent être définies une ou plusieurs fonctions. La définition d'une classe se fait à l'aide d'un segment CLASS, analogue à un segment TYPE :

```
CLASS identificateur
  BODY
   définition du domaine
   liste de définitions de fonctions
ENDidentificateur
```

La classe anyà

anyà est la classe universelle. Son domaine est constitué de tous les types définis dans tous les langages de la famille CONLAN; compte tenu du fait que deux types distincts peuvent être définis avec le même identificateur dans deux langages distincts chaque type est représenté par son nom universel, construit en préfixant son identificateur avec la suite de langages qui relie CONLAN PRIMITIF au langage dans lequel il est défini.

Ainsi, si un type t est défini dans le langage 11 directement dérivé de BASE CONLAN (voir figure du paragraphe II.1), t apparaît dans anyà par son nom universel

"pscl.bcl.11.t"

où pscl est l'identificateur de CONLAN PRIMITIF (en anglais "primitive set conlan") et bcl est l'identificateur de BASE CONLAN. Dans la suite toutefois nous n'emploierons le nom universel que si une ambiguïté est à craindre.

Trois opérateurs binaires entre éléments de anyà et à valeur dans bool sont disponibles : =, ∿= et <|. anyà joue, pour les classes, un rôle analogue à celui de univà pour les types.

Exemple : La classe des valeurs scalaires

Anticipant sur le paragraphe III.1, dans lequel sont listés les types de CONLAN PRIMITIF, nous définissons la classe des valeurs scalaires comme étant composée de tous les types qui ne dérivent ni de cellà (type des porteuses primitives) ni de tupleà (constructeur primitif) :

```
CLASS valeur_scalaire BODY
  ALL x : anyà WITH
    ∿((x <| cellà) | (x <| tupleà)) ENDALL
  CARRYALL
ENDvaleur_scalaire.
```

La directive CARRYALL rend disponibles sur la classe valeur_scalaire les opérateurs de relation de anyà : =, ∿= et <|.

Dans BASE CONLAN, le type terminal (t : anyà; di : t) définit le type des fils de connexion à valeurs dans t et valeur par défaut di (voir paragraphe III.2.e). On peut vouloir, dans un langage dérivé de BASE CONLAN, restreindre le type des valeurs d'un fil de connexion à des valeurs scalaires, ce qui s'exprime par :

```
SUBTYPE fil_scalaire(t : valeur_scalaire; di : t)
   BODY terminal(t, di)
ENDfil_scalaire.
```

L'utilisateur d'un tel langage peut définir des objets de types

fil_scalaire(bool, 1)
fil_scalaire(int, -107)
fil_scalaire(tern, 'U').

Dans le troisième exemple, on suppose que le type tern, logique à 3 valeurs, a été défini par énumération de ses éléments {0, 1, 'U'}.

Inversement, toute tentative pour définir des objets de types

fil_scalaire(fil_scalaire(bool, 1), abc)
fil_scalaire(array(bool), 0111)

sera rejetée par la détection d'une erreur de type pour le premier paramètre : fil_scalaire dérive de cellà, array(bool) dérive tupleà; aucun de ces deux types n'appartient à la classe valeur_scalaire.

II.5 Les descriptions

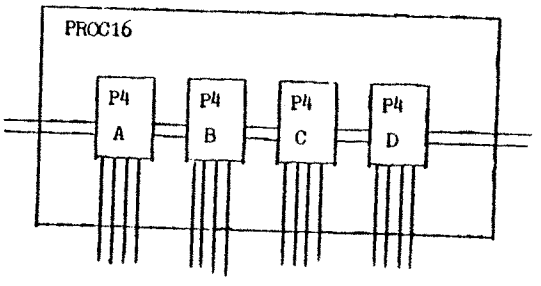
Une description doit être vue comme le modèle, écrit en CONLAN, d'un circuit logique. C'est une boîte qui communique avec l'extérieur par une interface composée de porteuses, correspondant aux broches d'entrée-sortie.

Chaque élément, ou groupe d'éléments,de l'interface est précédé d'un mot-clé qui précise le sens de transfert des informations entre la description et son environnement : IN, OUT, INOUT. Une description peut être paramétrée par une liste d'attributs, qui sont des paramètres dont la valeur doit être déterminée statiquement : dans ce cas, on a la description d'une famille de modèles. Les notions d'entité en LASSO, et de description en CONLAN sont tout à fait semblables.

Une description représente le modèle de structure et de comportement d'une classe d'objets. On peut utiliser un ou plusieurs exemplaires de ces objets dans une description englobante, en leur attribuant un nom par une déclaration. A ce titre, une description est comparable à un type de porteuse.

Exemple

Un processeur 16 bits est construit à l'aide de 4 micro-processeurs 4 bits identiques.



Ce système sera décrit par :

```
DESCRIPTION proc16 (interface de proc16)
BODY
   DESCRIPTION p4 (interface de p4)
   BODY
   :  description des objets internes et
   :  du fonctionnement de p4
   ENDp4
   USE a, b, c, d : p4 ENDUSE "/ ceci est la déclaration de 4 exemplaires
                                                          de p4 /"
        déclaration d'autres objets internes à proc16
        connexions des interfaces de a, b, c,d entre eux et avec l'inter-
        face de proc16.
ENDproc16
```

L'imbrication et l'interconnexion de descriptions permet donc de représenter la structure d'un système, alors que l'invocation de fonctions et d'activités permet d'en représenter le fonctionnement.

Une description a la forme :

REFLAN langage de référence
DESCRIPTION identificateur [(attributs)] (interface)
[ASSERT spécifications sur les attributs et les porteuses d'interface
ENDASSERT]
BODY
[définitions de types, fonctions, activités, descriptions]
[déclarations de porteuses, d'exemplaires de descriptions]
[invocation d'opérations]
ENDidentificateur

a) Les attributs

La liste d'attributs d'une description écrite en CONLAN joue un rôle analogue à la liste des paramètres d'une entité écrite en LASSO. Ce mécanisme étant essentiellement destiné à être utilisé par les concepteurs de systèmes logiques, il doit être simple à écrire, et de mise en oeuvre aisée et efficace. Aussi les attributs d'une description sont-ils restreints à des valeurs.

Lorsqu'un exemplaire de description est déclaré, tous les attributs doivent être valués. Les attributs disparaissent alors, pour toute référence ultérieure à cet exemplaire. Dans cette optique, la liste d'attributs, si elle n'est pas vide, est écrite séparément de la liste des objets d'interface.

Ainsi, lors d'une déclaration d'exemplaire, le nom de la description, suivi des attributs effectifs, joue-t-il le même rôle qu'un nom de type suivi de ses paramètres effectifs.

Exemple

La description d'une porte nand à 2 entrées et dont le temps de traversée est paramétré par un entier positif ou nul "d" s'écrit :

```
DESCRIPTION nand2(d : nnint)
      (IN e1, e2 : sig0; OUT z : sig0)
BODY
  z.=∿ (e1 & e2) % d
ENDnand2
```

Le symbole "%" est l'opérateur de retard.
La déclaration de deux exemplaires particuliers de porte nand, l'un de temps de traversée 4 unités de temps et l'autre de 5, se fait par l'instruction :

```
USE porte1 : nand2(4);
    porte2 : nand2(5)
ENDUSE
```

b) L'interface

Tout exemplaire de description étant on objet entièrement défini, le segment description définit une barrière de visibilité entre ses objets internes et son environnement. Dans le corps d'une description, on ne peut référencer aucun objet de l'environnement, et réciproquement de l'environnement d'une description on ne peut référencer aucun objet local à celle-ci. L'interface est le seul moyen de communication.

L'interface est constitué exclusivement de porteuses, qui sont à la fois connues et accessibles dans une description et dans son environnement. Plus précisément, la déclaration d'un exemplaire de description déclare implicitement tous ses objets d'interface, référençables à l'extérieur en préfixant leur nom formel par le nom de l'exemplaire.

Exemple

USE porte1 : nand2(4) ENDUSE

déclare implicitement trois signaux booléens

porte1.e1, porte1.e2 et porte1.z
de type sig0.

Droits d'accès

Les règles d'accès aux objets d'interface sont fixées par le sens de transfert des informations, spécifié pour chacun d'eux dans l'interface formel du segment DESCRIPTION.

IN indique un objet considéré comme une entrée. Il peut être référencé et modifié par l'environnement de la description; il est référençable mais non modifiable à l'intérieur de la description.

OUT indique un objet considéré comme une sortie. Il peut être référencé et modifié à l'intérieur de la description; il est référençable mais non modifiable dans l'environnement de la description.

INOUT indique un objet considéré alternativement comme une entrée et une sortie, selon les instants et en fonction de certaines conditions. Il peut être référencé et modifié par l'environnement de la description et à l'intérieur de celle-ci.

c) Les objets internes et leurs inter-communications

Les objets internes à une description sont :

- les objets (valeurs et porteuses) de types déclarés par l'instruction DECLARE
- les exemplaires de description déclarés par l'instruction USE
- les interfaces de ces exemplaires, implicitement déclarés lors du USE.

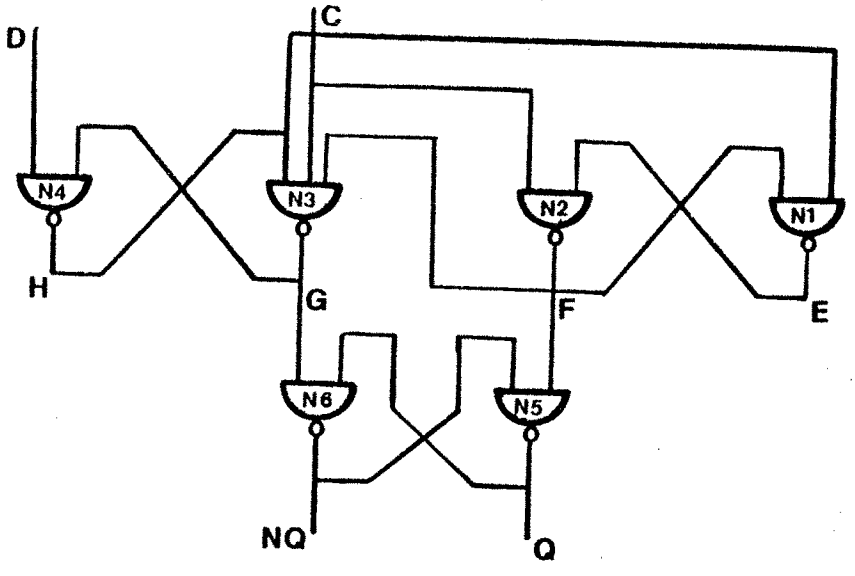
Il est fréquent, en particulier pour les descriptions de niveaux détaillés, où l'accent est mis sur la représentation structurelle d'un circuit logique, qu'un modèle soit construit comme un réseau d'exemplaires de descriptions plus simples. Dans ce cas, on lie de manière statique et permanente les entrées-sorties de ces exemplaires entre elles, ou à des chemins de données de la description englobante. Une modélisation possible consiste à considérer que deux fils soudés entre eux ne constituent plus qu'une seule et même porteuse. Une autre modélisation fait intervenir la notion de connexion permanente entre deux porteuses distinctes. Les deux modélisations ne sont fonctionnellement équivalentes que si la seconde n'induit aucun retard de modification de valeur.

Le choix de l'un ou l'autre point de vue est laissé à l'utilisateur; les deux sont descriptibles. Pour le premier type de modèle, il faut exprimer une synonymie entre porteuses : cela s'écrit dans l'instruction USE, en faisant suivre un identificateur d'exemplaire de description de la liste

des porteuses dont ses objets d'interface sont synonymes. Les correspondances se font par position, les places vides devant être explicitement marquées. Pour le second type de modèle, les communications entre interfaces de boîtes sont exprimées par l'invocation des activités appropriées, dans la partie opératoire du corps de la description.

Exemple

Le schéma ci-dessous illustre une bascule D réalisée par l'interconnexion de 6 portes NAND :



Supposons déjà définies les descriptions nand2 et nand3, respectivement à 2 et 3 entrées. La description dff1 illustre le premier point de vue : les interfaces des portes nand sont, dans l'instruction USE, déclarés comme synonymes des signaux locaux ou d'interface de dff1. Dans la description dff2, au contraire, toutes les connexions sont explicitées.

```
DESCRIPTION dff1(IN c, d : sig0;
                 OUT q, nq : sig0)
BODY
  EXTERNAL DESCRIPTION nand2, nand3 END
  DECLARE e, f, g, h : sig0 ENDDECLARE
  USE n1(h, f, e),
      n2(e, c, f),
```

```
    n4(g, d, h),
    n5(f, nq, q),
    n6(q, g, nq) : nand2(1);
    n3(f, c, h, g) : nand3(1) ENDUSE
ENDdff1

DESCRIPTION dff2(IN c, d : sig0;
                 OUT q, nq : sig0)
BODY
  EXTERNAL DESCRIPTION nand2, nand3 END
  DECLARE e, f, g, h : sig0 ENDDECLARE
  USE n1, n2, n4, n5, n6 : nand2(1);
      n3 : nand3(1) ENDUSE

n1.e1 .= h,        n5.e1 .= f,        e..= n1.z,
n1.e2 .= f,        n5.e2 .= nq,       f .= n2.z,
n2.e1 .= e,        n6.e1 .= q,        h .= n4.z,
n2.e2 .= c,        n6.e2 .= g,        q .= n5.z,
n4.e1 .= g,        n3.e1 .= f,       nq .= n6.z,
n4.e2 .= d,        n3.e2 .= c,        g .= n3.z,
                   n3.e3 .= h

ENDdff2
```

II.6 Segment définissant un langage

a) Définition des primitives

Un segment langage est une suite de définitions de types, de fonctions, d'activités, et de descriptions, qui seront considérés comme primitifs par l'utilisateur qui écrira ses descriptions avec ce segment langage comme langage de référence. Un segment langage ne contenant que des définitions de descriptions est équivalent à un catalogue de circuits.

Un nouveau langage de la famille CONLAN est défini à partir d'un langage existant : son langage de référence. Le langage de référence met à la disposition de l'auteur du nouveau langage :

- l'ensemble des types, fonctions, activités et descriptions qui le composent
- sa syntaxe.

A l'aide de ces éléments sont définis de nouveaux segments type, fonction, activité et description. Ceux du langage de référence deviennent a priori inconnus de l'utilisateur du nouveau langage, et peuvent en particulier être redéfinis. Si l'on désire que le nouveau langage contienne certains des types, opérations ou descriptions du langage de référence, on doit les transporter explicitement par

CARRY liste d'identificateurs de segments ENDCARRY

Si le nouveau langage est un sur-ensemble de son langage de référence, une écriture rapide est :

CARRYALL

Par défaut, le nouveau langage a la même syntaxe que son langage de référence. Il est cependant possible, à l'aide de l'instruction FORMATà exposée ci-dessous, de supprimer certaines productions, ou au contraire de rajouter des formes syntaxiques. L'utilisateur d'un langage particulier peut ne pas pouvoir définir de types nouveaux, par exemple (c'est en général le cas pour les langages de niveau réseau de portes). A l'inverse, on peut vouloir donner une forme infixée pour l'appel des opérations nouvellement définies, ou introduire des structures de contrôle particulières. Toutefois, les adjonctions syntaxiques sont maintenues dans des limites assez strictes, car seul un petit nombre de non-terminaux de la grammaire de base sont extensibles.

Un segment langage a la structure suivante, dans laquelle les différentes définitions et les modifications syntaxiques peuvent apparaître dans un ordre quelconque assurant l'absence de référence avant :

```
REFLAN langage de référence CONLAN identificateur
BODY
  [partie "CARRY"]
  [définitions de types]
  [définitions de fonctions et d'activités]
  [définitions de descriptions]
  [modifications syntaxiques]
ENDidentificateur
```

b) Modifications syntaxiques

Les modifications syntaxiques sont spécifiées dans un bloc FORMATà. Réservées aux concepteurs de langages, elles ont été formalisées de manière à permettre un traitement, en particulier des vérifications automatiques de cohérence d'une définition de langage. Par contre, il n'est pas question de mettre à la disposition d'un concepteur de circuits un mécanisme d'extension syntaxique. Un bloc FORMATà ne peut donc être écrit que dans un segment CONLAN, ou un segment TYPE, FUNCTION ou ACTIVITY lui même contenu, à un niveau d'imbrication quelconque, dans un segment CONLAN. L'écriture des modifications syntaxiques a été inspirée par les travaux sur les langages extensibles menés à Grenoble [JoS.70, Vid.74].

On considère que la syntaxe du langage est écrite sous une forme normale de Backus étendue, dans laquelle :

- chaque alternative d'une production est numérotée par un entier positif, la seule contrainte étant que toutes les alternatives d'une même production aient un numéro distinct
- une alternative est marquée R (removable) si elle peut être ôtée de la grammaire
- une production est marquée R si le non-terminal et toutes les alternatives de la production qui le définit peuvent être ôtés de la grammaire
- une production est marquée E (extensible) si de nouvelles alternatives peuvent lui être ajoutées.

Exemple

```
segment_description = 1 'DESCRIPTION' identificateur
      attributs_et_interface corps 'END'
attributs_et_interface =
    1 R attributs |
    2   interface |
    3 R attributs interface
interface = 1 '('sens liste_typée suite_interface ')'
E sens = 3 'IN'
         5 'OUT'
         2 'INOUT'
```

. Les productions : segment_description
interface
ne peuvent être modifiées.

. Les alternatives 1 et 3 de la production : attributs_et_interface peuvent être supprimées, mais pas l'alternative 2, ni la production toute entière.

De nouvelles alternatives peuvent être ajoutées à la production : sens.

Ecriture des modifications syntaxiques

Dans l'instruction FORMATà, on spécifie :

- les productions ou alternatives de productions que l'on veut supprimer
- les alternatives de production que l'on veut étendre, avec leur sémantique
- les alternatives de production auxquelles on veut ajouter une nouvelle signification.

La sémantique d'une alternative de production est exprimée en termes de l'état du langage tel qu'il se présente avant le traitement de l'instruction FORMATà en cours. Par exemple, la sémantique d'un opérateur infixé est le plus souvent un appel de fonction (voir les extensions dans tytupleà au paragraphe III.2.a) ou d'activité; mais la sémantique peut être aussi une expression plus complexe, voire une ou plusieurs instructions incluant des formes conditionnelles.

Une instruction FORMATà s'écrit :

```
FORMATà
  REMOVE pa1, pa2, ... pan
  EXTEND nonterminal.numéro
    marque  nonterminal = extension syntaxique
    MEANS sémantique
  EXTEND nonterminal.numéro
    marque nonterminal = extension syntaxique
    MEANS sémantique
.....
ENDFORMAT.
```

Dans l'instruction REMOVE, pa_1, ... pa_n sont soit des non-terminaux, soit des non-terminaux suffixés par un numéro d'alternative : soit la production complète, soit l'alternative dont le numéro est donné est supprimée de la grammaire.

Dans l'instruction EXTEND, on spécifie d'abord l'alternative de production, puis la marque associée (R ou E), puis la syntaxe de la nouvelle alternative, puis sa sémantique. Si aucune marque n'est indiquée, la nouvelle règle devient permanente et non modifiable pour tous les langages qui dériveront du langage en cours de définition. Si la syntaxe est absente, c'est que l'on étend la sémantique d'une production existante (par exemple on donne une nouvelle signification à un opérateur). Si la sémantique est absente, c'est que l'on est dans une règle intermédiaire servant à une production suivante.

Exemple

```
FORMATà
    REMOVE attributs_et_interface.1,
           attributs_et_interface.3
    EXTEND sens.10
      R sens = 'NONDIRECTIONNEL'
ENDFORMAT
```

II.7 Règles portant sur les segments

a) Règles d'imbrication des définitions de segments

Le tableau suivant résume la possibilité, pour une définition de segment (ligne), de contenir des définitions de segments des différentes catégories.

	CONLAN	DESCRIPTION	CLASSE TYPE	FONCTION	ACTIVITE
CONLAN	non	oui	oui	oui	oui
DESCRIPTION	non	oui	oui	oui	oui
TYPE/CLASSE	non	non	oui	oui	oui
FONCTION	non	non	oui	oui	oui
ACTIVITE	non	non	oui	oui	oui

b) Règles de visibilité

De l'intérieur d'une définition de segment, on peut ou non accéder :

1 - aux identificateurs de segments définis dans le langage de référence
2 - aux identificateurs de segments définis dans un segment englobant
3 - aux identificateurs d'objets (valeurs, porteuses et exemplaires de description) déclarés dans un segment englobant.

en plus des accès aux segments et objets locaux. Pour chaque catégorie de segment, ces possibilités sont les suivantes :

	Segments du langage de référence	Segments définis dans un segment englobant	Objets déclarés dans un segment englobant
CONLAN	tous	/	/
DESCRIPTION	tous	non	uniquement via son interface
TYPE CLASSE	types, classes, fonctions, activités	types, classes, fonctions, activités	uniquement via ses paramètres
FONCTION et ACTIVITE	tous	tous	via ses paramètres et éventuellement directement

jusqu'au segment DESCRIPTION ou CONLAN le plus proche, ce segment compris.

Les objets déclarés et les segments définis à l'intérieur d'un segment peuvent, pour certains d'entre eux, être accédés par leur identificateur, de l'extérieur du segment. Les règles d'exportation sont les suivantes :

Segment	exporte
CONLAN	tout ce qui n'est pas défini PRIVATE
DESCRIPTION	interface
TYPE, CLASSE	fonctions et activités (sauf celles qui sont définies PRIVATE)
FONCTION et ACTIVITE	rien

Les ambiguïtés de noms sont uniformément résolues en préfixant par l'identificateur du segment dans lequel le segment (ou l'objet) que l'on veut accéder est défini.

II.8 Les instructions primitives

Un texte écrit en CONLAN est une définition de segment, de l'une des 6 catégories qui viennent d'être présentées. Ce texte, dans le cas général, est constitué d'un en-tête et d'un corps, le corps étant lui-même composé des définitions de segments locaux et d'instructions.

On distingue en CONLAN quatre catégories d'instructions :

1 - Les déclarations d'objets locaux sont contenues dans les blocs USE et DECLARE, exposés au paragraphe II.5.C.

2 - Les modifications syntaxiques sont contenues dans les blocs FORMAT.

3 - Les assertions sont des expressions booléennes regroupées dans les blocs ASSERT.

4 - Les invocations d'activités, conditionnées ou non, décrivent un comportement, et ne peuvent être présentes que dans un segment FUNCTION, ACTIVITY ou DESCRIPTION.

La suite de ce paragraphe sera consacrée aux instructions des deux dernières catégories, pour lesquelles une évaluation dynamique au cours du temps simulé est l'une des applications envisagées. En CONLAN, les assertions d'une part, les invocations d'activités d'autre part, sont évaluées en parallèle. Aucun séquencement n'est pré-supposé, en dehors de l'ordre d'évaluation qui résulte de l'imbrication des opérations et, dans les expressions, des priorités des opérateurs et du parenthésage. La séquentialité des instructions doit donc être définie pour les langages de description des niveaux où cette notion est pertinente. Pour les autres langages, la séquentialité ne peut être obtenue que par une gestion explicite de conditions validant et invadant successivement les instructions.

Nous allons à présent détailler les formes prédicatives, conditionnelles et répétitives primitives dans CONLAN.

a) Les prédicats

Un prédicat est une expression booléenne, quantifiée ou non. Lorsqu'un quantificateur est présent, le type des variables liées doit être indiqué. Pour des raisons de commodité, trois quantificateurs (et non deux) sont donnés dans CONLAN.

Si p(x) est une expression booléenne, ou un prédicat dans lequel x est libre :

- FORALLà x : untype IS p(x) ENDFOR prend la valeur vrai si p(x) prend la valeur vrai pour tout élément du type "untype".

- FORSOMEà x : untype IS p(x) ENDFOR prend la valeur vrai si p(x) prend la valeur vrai pour au moins un élément du type "untype".

- FORONEà x : untype IS p(x) ENDFOR prend la valeur vrai si p(x) prend la valeur vrai pour un élément de "untype", et un seul.

Si "untype" est vide, un prédicat quantifié par FORALLà prend la valeur vrai, un prédicat quantifié par FORSOMEà ou par FORONEà prend la valeur faux.

b) Les formes conditionnelles

b.1) La forme IF

La proposition IF permet de sélectionner un parmi plusieurs

- objets de même type
- invocations de fonctions
- listes d'invocations d'activités

Si p_1, p_2, p_3 sont des prédicats, et a, b, c, d sont les objets ou invocations à sélectionner, la proposition :

IF p_1 THEN a ELIF p_2 THEN b ELIF p_3 THEN c ELSE d ENDIF

donnera comme résultat :

a si p_1 est vrai
b si $\sim p_1$ & p_2 est vrai
c si $\sim p_1$ & $\sim p_2$ & p_3 est vrai
d sinon

Le nombre de parties "ELIF" est arbitraire. La partie "ELSE" est optionnelle si la sélection opère sur des invocations d'activités.

b.2) La forme CASE

La sélection peut aussi s'effectuer sur des valeurs non booléennes, à l'aide de la proposition choix :

CASE i IS
v_1 : a_1;
v_2 : a_2;
⋮
v_n : a_n;
ELSE a_{n+1} ENDCASE

où v_1, v_2, ... , v_n sont des valeurs d'un même type deux à deux distinctes. La partie "ELSE" est optionnelle si la sélection opère sur des invocations d'activités, ou si les valeurs listées énumèrent entièrement le type de i.

c) Répétition

Des invocations multiples et concurrentes d'une même fonction ou d'une même liste d'activités sont engendrées par la proposition OVER, qui énumère un ensemble fini de valeurs d'un même type. Si "unensemble" désigne cet ensemble de valeurs, et "fa" l'invocation répétée, la répétition s'écrit :

OVER x : unensemble REPEAT
fa(paramètres) ENDOVER.

Si les valeurs énumérées sont une liste d'entiers séparés par un intervalle constant, une écriture dérivée de la "boucle pour" des langages de programmation est admise :

OVER x FROM i STEP j TO k REPEAT
fa(paramètres) ENDOVER

Dans cette seconde forme, i, j, k sont des entiers. La partie "STEP j" est optionnelle : si elle est omise, j vaut par défaut 1 si i ≤ k, -1 si i > k.

III DERIVATION DE BASE CONLAN A PARTIR DE CONLAN PRIMITIF

A titre d'illustration de l'ensemble des notions qui viennent d'être exposées, nous donnons maintenant les principales étapes de la description de BASE CONLAN. Il ne s'agit pas pour nous de reproduire l'intégralité des définitions de types et d'opérateurs, qui paraîtront dans [PBB.81], mais plutôt de guider le lecteur en explicitant la démarche suivie.

III.1 Les types et opérateurs de CONLAN PRIMITIF

Les types suivants sont donnés dans CONLAN PRIMITIF, sans définition formelle au sein du langage :

a) bool

Eléments : vrai noté 1, faux noté 0
Opérations :
& (et), | (ou) : bool × bool → bool
=, ∿=, <, =<, >, >= : bool × bool → bool
∿(non) : bool → bool

b) int

Eléments : l'ensemble des entiers, que l'on peut noter en base 2 (indiqué par B), en base 8 (indiqué par O), en base 10, et en base 16 (indiqué par H).

Exemple : -12 = -1100 B =-14O = -C H

Opérations : +, -, *, /, ↑, MOD : int × int → int
=, ∿=, <, =<, >, >= : int × int → bool

c) string

Eléments : l'ensemble des caractères du code ISO-646 [ISO.73], ordonnés selon ce code, et des chaînes de caractères de longueur finie. Ces objets sont dénotés entre apostrophes, l'apostrophe elle-même est doublée.

Exemples : '1A', 'Je l''ai vue'

Opérations : =, ^=, <, =<, >, >= : string × string : bool
orderà : string → int

Pour comparer deux chaînes de longueurs différentes, on commence par compléter la plus courte à droite par des blancs, puis la comparaison s'effectue à partir de la gauche, caractère par caractère, selon l'ordre alphabétique du code ISO.

'ABC' = 'ABC␣␣'
'ABC' < 'Z?'

La fonction orderà renvoie un entier obtenu en traitant la chaîne de caractères passée en paramètre comme un nombre codé en base 128.

orderà('x2') = 7811 * 128 + 3211 = 15410

d) cellà(t : anyà)

Eléments : objets capables de contenir au plus un élément de type t. Il n'existe pas de dénotation de cellule : une cellule doit d'abord être identifiée, dans une instruction de déclaration, avant de pouvoir être référencée. Une cellule est initialement vide. Les cellules sont les porteuses primitives, à partir desquelles toutes les porteuses des langages utilisateur sont définies.

Opérations

getà (extraction de la valeur) : cellà(t) → t
putà (modification de la valeur) : cellà(t) × t → NONE
cell_typeà (demande le type de la valeur) : cellà(t) → anyà
emptyà (demande si la cellule est vide) : cellà(t) → bool

L'opération putà est, contrairement à toutes les autres opérations de CONLAN PRIMITIF qui sont des fonctions, une activité. Elle a pour effet de modifier le contenu de son premier paramètre, à condition que le second paramètre soit du bon type. La notation cellà(t) × t → NONE signifie que l'activité putà ne renvoie pas de valeur.

e) univà

Eléments : tous les éléments de tous les types que l'on peut définir dans CONLAN, ainsi que les noms universels de ces types. C'est l'univers des objets de CONLAN.

Opérations
=, ~= : univà × univà → bool

f) tupleà

Eléments : l'ensemble des suites ordonnées d'éléments de univà, y compris la suite vide.

Opérations :

sizeà : tupleà → int	renvoie la longueur d'un tuple
selectà : tupleà × int → univà	renvoie le ième élément d'un tuple
=, ~= : tupleà × tupleà → bool	deux tuples sont égaux s'ils ont la même taille et des éléments identiques dans un ordre identique.
extendà : tupleà × univà → tupleà	renvoie un tuple ayant un élément de plus que le premier opérande, dont tous les éléments sauf le dernier sont égaux aux éléments de même rang du premier opérande, et dont le dernier élément est égal au deuxième opérande.
removeà : tupleà × int → tupleà	Soit i l'entier passé en second paramètre. Si i n'est pas compris entre 1 et la longueur du premier paramètre, le résultat est "erreur". Sinon, le résultat est un tuple ayant un élément de moins que le premier paramètre, et dont le jème élément est égal - au jème élément du premier paramètre si j < i - au j+1ème élément du premier paramètre si j ≥ i.

La notion de tuple servira à construire les objets non scalaires (structures et tableaux), et les historiques des porteuses.

g) anyà

Eléments : tous les types déjà définis, ou que l'on peut définir dans CONLAN, représentés par leur nom universel (II.4.d).
A ce stade, et sous forme abrégée :
anyà = {univà, int, char, bool, cellà, tupleà}
anyà servira à indiquer la généricité des définitions d'opérations.

Opérations : =, ∿= : anyà × anyà → bool
.< : univà × anyà → bool appartenance d'un élément à un type.
<| : anyà × anyà → bool relation de dérivation.

h) Priorité des opérateurs

Dans CONLAN, de nombreux opérateurs sont génériques. Une fois défini, un opérateur a une priorité fixée définitivement. Il y a dans CONLAN 9 niveaux de priorité; les niveaux 2 et 9 sont vides d'opérateurs en CONLAN PRIMITIF, mais contiendront respectivement :

2 : XOR (ou exclusif), EQV (équivalence)
9 : #= (concaténation)

en BASE CONLAN. On a donc à ce stade :

1 : |
2 :
3 : &
4 : <, =<, >, >=, =, ∿=, .<, <|
5 : +, -
6 : *, /, MOD
7 : ↑
8 : ∿, -, + (unaires)
9 :

III.2 Principes de construction de BASE CONLAN

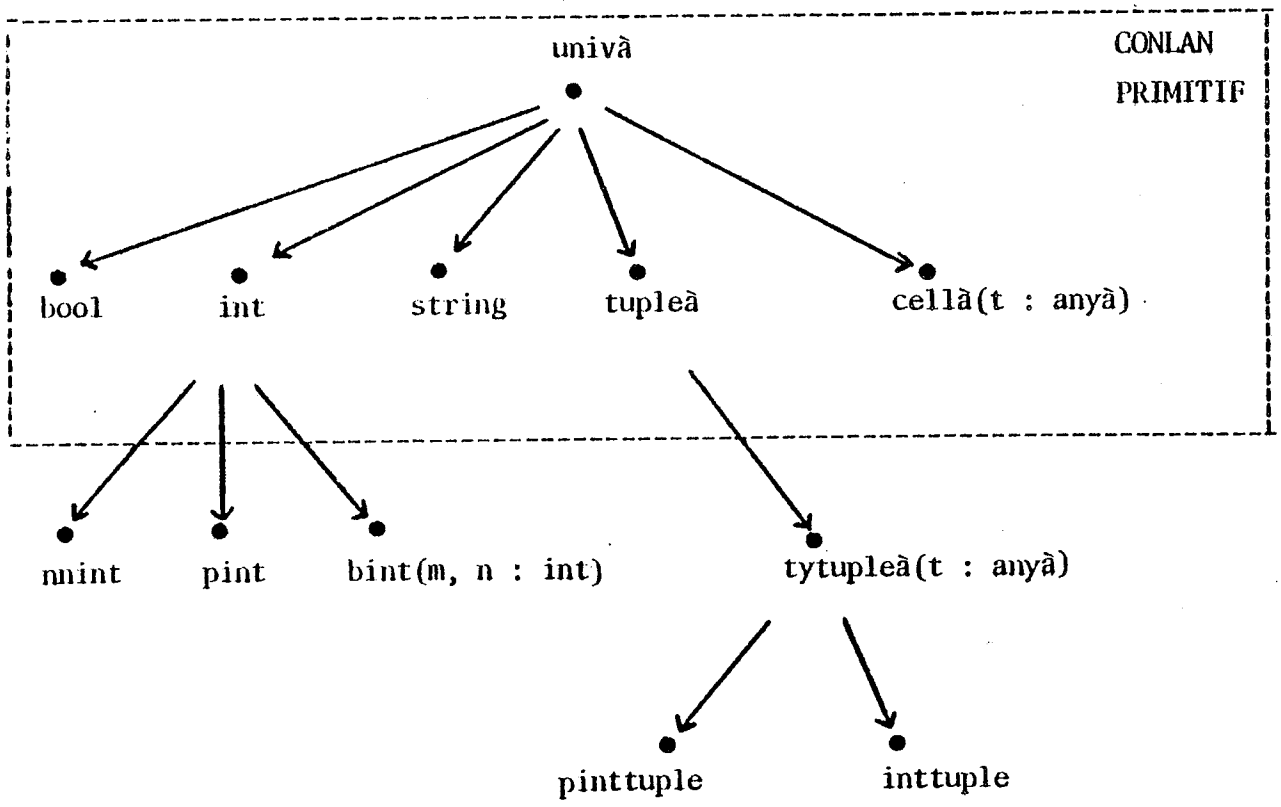
Dans CONLAN PRIMITIF, on ne dispose que de types de valeurs scalaires, de cellules, de tuples, et des univers d'objets et de types. Dans BASE CONLAN, on veut avoir des objets structurés, tableaux et enregistrements, très généraux. On veut aussi introduire la notion d'historique de valeurs au cours du temps, et de porteuses définies par un comportement temporel. L'objectif est double. En définissant tableaux, structures et historiques, nous espérons fournir des concepts suffisamment vastes pour pouvoir, par restriction et spécialisation, s'adapter aux besoins, contraintes et habitudes des utilisateurs des différents niveaux de langage prévisibles. En définissant les porteuses, nous cherchons au contraire à montrer, sur des exemples typiques et indépendants de toute technologie, comment s'écrivent en CONLAN les opérations "système" assurant valeurs par défaut, volatilité ou mémorisation, dérepérage etc ... . A partir de ces modèles, les concepteurs de langages d'application devraient être en mesure de définir les porteuses plus particulièrement adaptées à une technologie ou à un niveau de conception.

BASE CONLAN est défini comme une extension de CONLAN PRIMITIF. Ainsi, tous les types et segments que le groupe de travail s'est donné sont-ils encore disponibles pour la définition des premiers langages d'application.

a) Les types utilitaires

A partir des types de CONLAN PRIMITIF, plusieurs types utiles à la fois aux constructeurs dimensionnels et temporels sont tout d'abord définis. Le plus important d'entre eux, tytupleà(t : anyà), a déjà été présenté au paragraphe II.4.b : nous en donnerons ici la définition complète, à titre d'exemple.

L'arbre de dérivation canonique des types utilitaires à partir de CONLAN PRIMITIF est le suivant :



Le début du segment qui définit le langage BASE CONLAN, identifié en abrégé par BCL est donné ci-dessous :

```
REFLAN conlan_primitif "/ceci est un commentaire/"
CONLAN bcl
BODY
  CARRYALL "/bcl est un sur-ensemble de conlan primitif/"
     "/entiers positifs ou nuls/"
  SUBTYPE nnint BODY
    ALL a : int WITH a >= 0 ENDALL
  ENDnnint
     "/entiers strictement positifs/"
  SUBTYPE pint BODY
    ALL a : int WITH a > 0 ENDALL
  ENDpint
```

```
    "/ensemble borné d'entiers consécutifs/"
  SUBTYPE bint(m, n : int) BODY
    ALL a : int WITH
     IF m =< n THEN
        (m =< a) & (a =< n)
     ELSE(n =< a) & (a =< m) ENDIF ENDALL
  ENDbint

    "/séquences d'éléments d'un même type/"
  TYPE tytupleà(t : anyà)
   BODY
   ALL x : tupleà WITH FORALLà i: bint(1, sizeà(x)) IS selectà(x, i) .< t
          ENDFORALL ENDALL
   CARRY =, ∿=, sizeà, removeà ENDCARRY

    "/sélection d'un élément/"
  FUNCTION select(x : tytupleà(t); i : pint) : t
   RETURN THEà z : t WITH z = selectà(old(x), i)
   FORMATà
    EXTEND ref_to_declared.10
    ref_to_declared = exp19 : id1  '['exp5 : id2']'
    MEANS select(id1, id2)
   ENDFORMAT        "/on écrira x[i] /"
  ENDselect

    "/suppression du premier élément/"
  FUNCTION tail(x : tytupleà(t)) : tytupleà(t)
   ASSERT sizeà(x) > 1 ENDASSERT
   RETURN removeà(x, 1)
  ENDtail

    "/ajout d'un élément en dernière position/"
  FUNCTION extend(x : tytupleà(t); a : t) : tytupleà(t)
   RETURN new(extendà(old(x), a))
   FORMATà
    EXTEND exp9.1
    exp9 = exp9 : id1 '#' exp9 : id2
    "/on écrira x # a /"
   MEANS extend(id1, id2)
  ENDFORMATà
ENDextend
```

```
    "/concaténation de deux tytuples/"
    "/on écrira x ## y/"
  FUNCTION catenate(x, y : tytupleà(t)) : tytupleà(t)
   BODY
   RETURN THEà z : tytupleà(t) WITH
    sizeà(z) = sizeà(x) + sizeà(y) &
    FORALLà i : bint(1, sizeà(x)) IS z[i] = x[i] ENDFORALL &
    FORALLà j : bint(1, sizeà(y)) IS z[sizeà(x) + j] = y[j] ENDFORALL ENDTHE
   FORMATà EXTEND exp9.1 MEANS catenate(id1, id2) ENDFORMATà
  ENDcatenate

    "/échange 2 éléments/"
  FUNCTION exchange(x : tytupleà(t); i, j : pint) : tytupleà(t)
   ASSERT i =<  sizeà(x), j =<  sizeà(x) ENDASSERT
   RETURN THEà y : tytuplea(t) WITH
      sizeà(y) = sizeà(x) &
      FORALLà k : bint(1, sizeà(x))) IS
       IF k = i THEN y[k] = x[j] ELIF k = j THEN y[k] = x[i] ELSE y[k] = x[k] END
         ENDFORALL ENDTHE
  ENDexchange
ENDtytupleà

   "/suites non vides d'entiers strictement positifs/"
SUBTYPE pinttuple BODY
  ALL a : tytupleà(pint) WITH sizeà(a) > 0 ENDALL
ENDpinttuple
   "/suites non vides d'entiers/"
SUBTYPE inttuple BODY
  ALL a : tytupleà(int) WITH sizeà(a) > 0 ENDALL
ENDintuple
:
:
```

b) Dérivation des tableaux

Les tableaux sont des ensembles structurés d'éléments d'un même type, organisés de façon matricielle. Ils ont les propriétés suivantes :

- le nombre de dimensions n'est pas limité (mais il est fini pour chaque tableau)
- chaque dimension est indexée par une séquence d'entiers consécutifs, ascendante ou descendante
- on peut, par indexation, extraire d'un tableau
  - . un élément,
  - . une tranche,
  - . plusieurs tranches non nécessairement contiguës;

  le résultat, s'il n'est pas scalaire, est un tableau dont les dimensions sont automatiquement indexées par des entiers ascendants à partir de 1. (une telle indexation est dite normalisée)
- deux tableaux sont compatibles si ils ont le même nombre de dimensions et si leurs dimensions ont deux à deux la même étendue
- deux tableaux sont égaux s'ils sont compatibles, et si leurs éléments de même rang sont deux à deux égaux
- la transposition de deux dimensions est une opération définie sur les tableaux
- une opération binaire définie sur un type t est étendue aux tableaux à éléments de type t : elle s'applique entre tous les éléments de même rang de deux tableaux compatibles, le résultat étant un tableau compatible à dimensions normalisées.
- une opération unaire définie sur un type t est étendue aux tableaux à éléments de type t : elle s'applique sur chacun des éléments du tableau, le résultat étant un tableau compatible à dimensions normalisées
- deux tableaux compatibles peuvent être concaténés selon n'importe quelle dimension comprise entre 1 et 1 + leur nombre de dimensions; deux tableaux dont toutes les dimensions sauf une sont compatibles peuvent être concaténés selon leur dimension incompatible. Le résultat est un tableau à dimensions normalisées, et dont l'étendue de la ième dimension est égale à
  - l'étendue de la ième dimension des deux opérandes si la concaténation ne s'est pas faite selon la ième dimension;
  - la somme des étendues des ièmes dimensions des deux opérandes si i est inférieur ou égal au nombre de dimensions des deux opérandes, et la concaténation s'est faite selon la ième dimension;

- 2 si i est égal à 1 + le nombre de dimensions des deux opérandes et la concaténation s'est faite selon la ième dimension (le résultat a alors une dimension de plus que les deux tableaux dont il est la concaténation).

Remarque

Le résultat d'une opération entre tableaux est un tableau à dimensions normalisées. Ce choix offre l'avantage de permettre l'indexation directe d'expressions de tableaux, sans qu'il soit besoin d'en stocker le résultat dans une porteuse intermédiaire.

Exemples

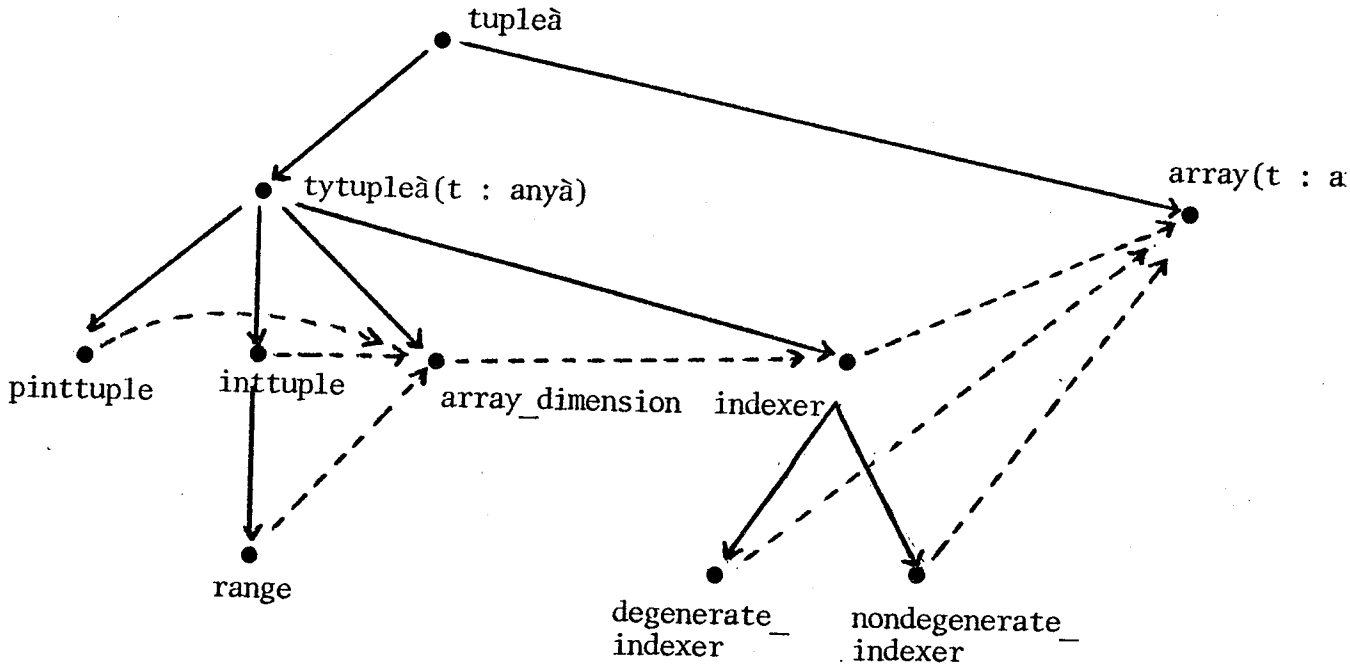
Soient a[-2 : 2], b[0 : 5; 1 : 3], c[-1 : -6; 4 : 2] trois tableaux d'entiers.

- b et c sont compatibles
- les indexations suivantes sont correctes :
  a[-1] b[0 : 2; 3] c[-1, -2, -4; 4, 2, 3] (b+c) [1 : 2; 2]
  la concaténation de b et c peut se faire selon 3 dimensions
  catenate(b, c, 1) a pour dimensions[1 : 12; 1 : 3]
  catenate(b, c, 2) a pour dimension [1 : 6; 1 : 6]
  catenate(b, c, 3) a pour dimension [1 : 6; 1 : 3; 1 : 2]

La définition des tableaux se fait à l'aide de quatre définitions de type :

- range : séquence ascendante ou descendante d'entiers consécutifs
- array_dimension : séquences de "range". C'est le type des dimensions de tableaux
- indexer : séquences de séquences d'entiers. C'est le type des objets servant à indexer les tableaux
- array(u : anyà) : tableaux d'éléments de u. Un tableau est défini comme un tuple à 2 éléments : sa partie dimension et sa partie valeur.

L'arbre de dérivation canonique des tableaux à partir des tuples est le suivant; les flèches en pointillé indiquent un ordre nécessaire dans l'écriture des définitions de types.



Les deux sous-types de indexer "degenerate_indexer" et "nondegenerate_indexer" forment une partition entre les index qui sélectionnent un élément et ceux qui sélectionnent plusieurs éléments d'un tableau.

La définition abrégée des types "range", "array_dimension", "indexer" et "array" est donnée en annexe.

c) Dérivation des enregistrements

Les enregistrements sont des ensembles structurés d'éléments de types fixés, organisés séquentiellement, et définis par une suite de champs. Ils ont les propriétés suivantes :

- deux enregistrements sont de même type si ils ont le même nombre d'éléments et si leurs descripteurs de champs sont deux à deux identiques
- chaque champ d'un enregistrement est décrit par un doublet composé de
  - . son type, élément de anyà
  - . son sélecteur, qui est une chaîne de caractère
- chaque élément d'un enregistrement doit appartenir au type spécifié dans son descripteur de champ
- on peut extraire d'un enregistrement l'un quelconque de ses éléments en concaténant à l'enregistrement (ou à son identificateur) le sélecteur de cet élément à l'aide du symbole "!"

- deux enregistrements de même type sont égaux si tous leurs éléments sont deux à deux égaux pour chaque champ.

Exemples

Soient les types d'enregistrements
t1 = record('x' : int; 'y' : bool)
t2 = record('xX' : int; 'y' : bool)
t3 = record('12AB' : cellà(int); 'y3' : bool)

Un élément de t1 et un élément de t2 ne sont pas comparables, car ils ne sont pas de même type (leurs premiers sélecteurs de champ étant différents).
Soit a un élément de t1. On a par exemple :

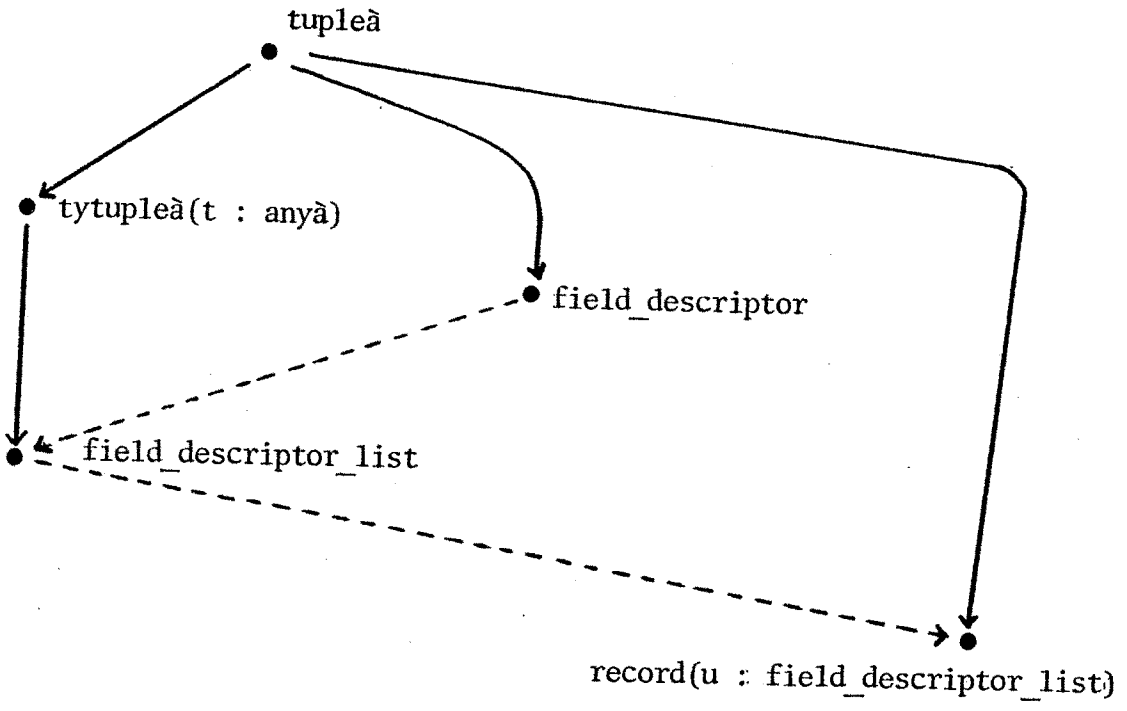
a ! 'x' = -10 a! 'y' = 1

Soit b un élément de t3. b! '12AB' est une cellule capable de contenir un entier.

La définition des enregistrements nécessite trois définitions de types :

- field_descriptor : descripteur d'un champ, défini comme un tuple à 2 éléments : une chaîne de caractères,un élément de anyà
- field_descriptor_list : liste de descripteurs de champ, définie comme une séquence de descripteurs de champ dont tous les sélecteurs sont différents
- record(u : field_descriptor_list) : enregistrements de descripteurs de champ u; un record est défini comme un tuple de même longueur que u, et dont chaque élément est du type indiqué par l'élément correspondant de u.

L'arbre de dérivation canonique des enregistrements à partir des tuples est donné par les flèches en traits pleins ci-dessous. Les flèches en pointillé indiquent l'ordre de définition des types :



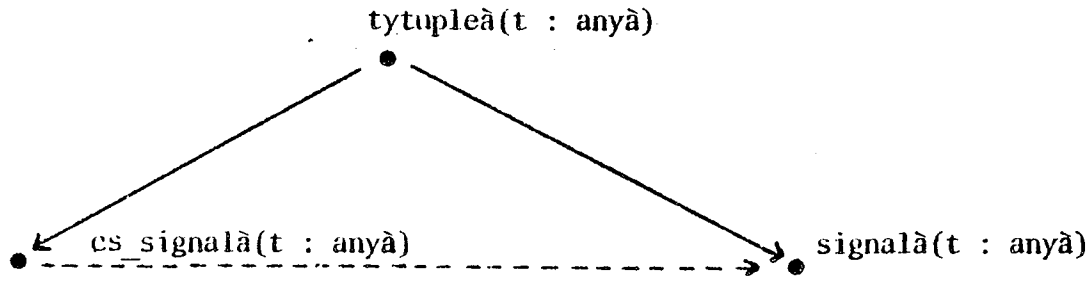
La définition abrégée des types field_descriptor, field_descriptor_list et record est donnée en annexe.

d) Dérivation des historiques de valeurs

En CONLAN, le temps est divisé en intervalles, eux-mêmes éventuellement subdivisés en pas de calcul (paragraphe II.2.b). L'historique des valeurs prises par une porteuse reflète ce modèle du temps à deux niveaux. La suite des valeurs aux différentes étapes de calcul à un instant donné est un élément du type cs_signalà. L'historique complet, défini comme la suite des cs_signaux aux instants 1, 2, ... tà, est un élément du type signalà.

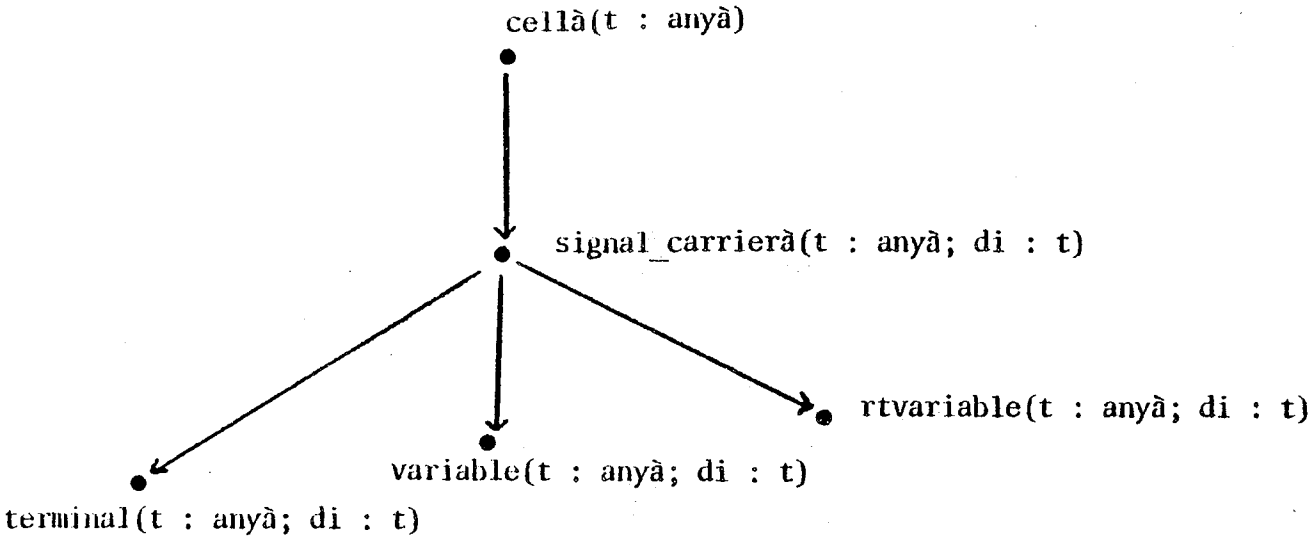
Les types signalà et cs_signalà ne sont pas manipulables par l'utilisateur. Ils ont définis dans BASE CONLAN pour permettre la définition des comportements temporels des différentes porteuses. Ces deux types dérivent de tytupleà, et fournissent des fonctions pour sélectionner une valeur, produire à partir d'un cs_signalà (resp. signalà) un autre cs_signalà (resp. signalà) ayant une valeur de plus, ou bien ayant une ou plusieurs valeurs correspondant à des étapes consécutives en moins, enfin créer un historique de valeurs toutes égales à une constante donnée.

La définition abrégée des types cs_signalà et signalà est donnée en annexe. Leur dérivation canonique à partir des tytuples est la suivante :



e) Les porteuses de BASE CONLAN

BASE CONLAN met à la disposition de l'utilisateur trois types de porteuses. Le type "terminal" modélise un fil de connexion à transmission instantannée. Le type "variable" correspond à la notion habituelle de variable dans les langages de programmation, mais peut être vu aussi comme un "latch" idéalisé. Le type "rtvariable" représente une bascule dont le délai inertiel serait de 1 unité de temps. Tous sont dérivés du type "signal_carrierà", qui définit la notion de porteuse, dont le contenu évolue au cours du temps, comme une cellule contenant un historique de valeurs.



Tous ces types génériques à partir de signal_carrierà ont deux paramètres : le premier indique le type des valeurs dont l'historique est conservé, le second donne la valeur par défaut. Ainsi, même si à l'instar d'une cellule une porteuse dérivée de signal_carrierà est initialement vide, l'interrogation de son contenu renvoie la valeur par défaut plutôt qu'une erreur.

La définition d'un certain nombre d'opérations, qualifiées INTERPRETERà, est nécessaire à l'interprétation du comportement temporel des diverses porteuses mises à la disposition de l'utilisateur pour écrire ses segments DESCRIPTION. La fonction "contentà" est la fonction de dérepérage, et fournit la valeur de la porteuse pour le pas de calcul courant de l'instant courant. L'activité "shrinkà" supprime, à l'instant courant, les valeurs correspondant à des pas de calcul locaux (voir quatrième partie). A la fin de chaque pas de calcul global, les porteuses qui ont été modifiées possèdent une valeur pour le pas de calcul suivant; pour les autres, l'invocation de l'activité "finstepà" détermine leur valeur au pas suivant. A la fin de l'évaluation de chaque intervalle de temps, l'invocation de l'activité "finintà" détermine, pour chaque porteuse, sa valeur au pas 1 de l'intervalle suivant. C'est donc la définition de finstepà et finintà, pour chaque type de porteuse, qui précise les propriétés de rémanence de sa valeur, alors que la définition de contentà spécifie la durée d'un changement de valeur.

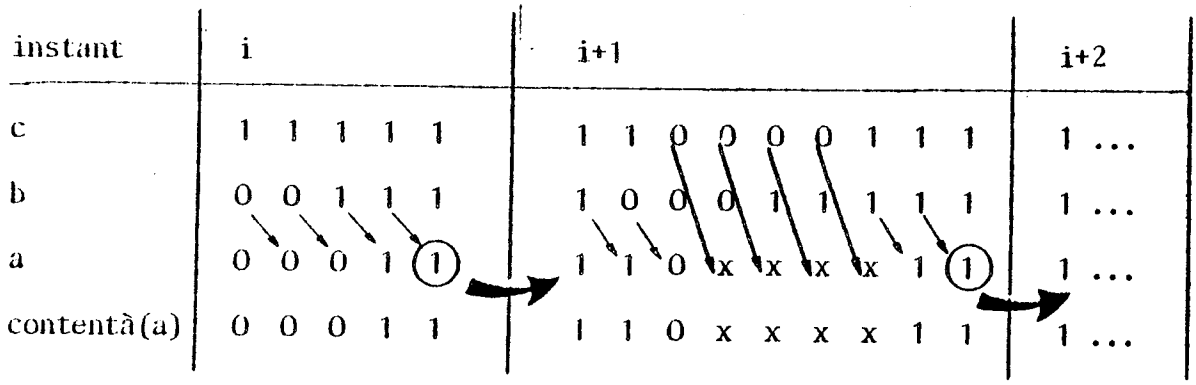
Pour chacun des types terminal, variable et rtvariable, l'utilisateur a accès à deux opérations : une activité permettant de modifier la valeur d'une porteuse (appelée selon le cas connexion, affectation ou transfert), et une fonction delay lui permettant de référencer la valeur finale de la porteuse à un instant passé.

Exemple

Soient "b" et "c" deux valeurs booléennes, "a" une porteuse à valeurs booléennes et valeur par défaut x (x désigne soit 0, soit 1). L'évolution de l'historique de a, sous l'effet d'une unique instruction conditionnelle ayant pour objet de "mettre b dans a si c vaut 1", est représentée ci-dessous, en fonction du type de a. La valeur entourée est la valeur renvoyée pour l'instant considéré, lors de l'invocation de la fonction delay.

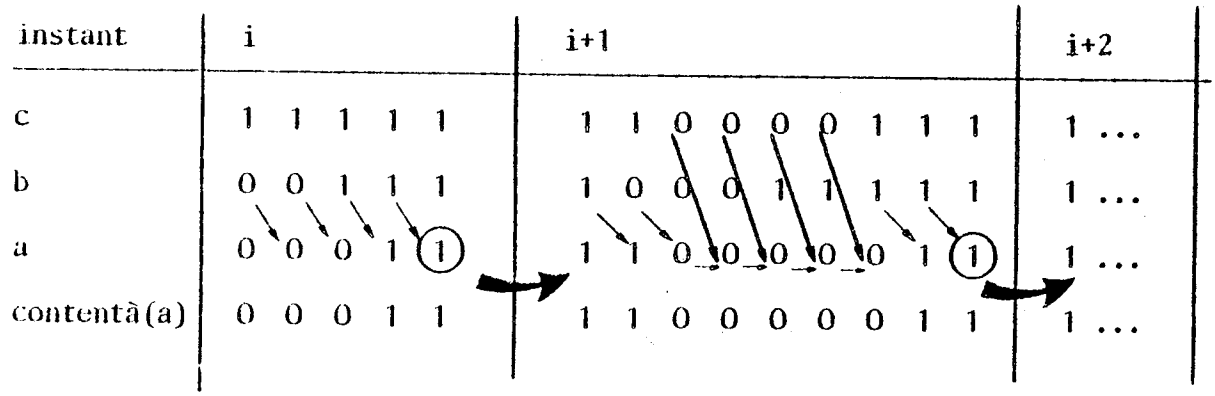
Connexion à une porteuse de type "terminal"

IF c THEN a .= b ENDIF



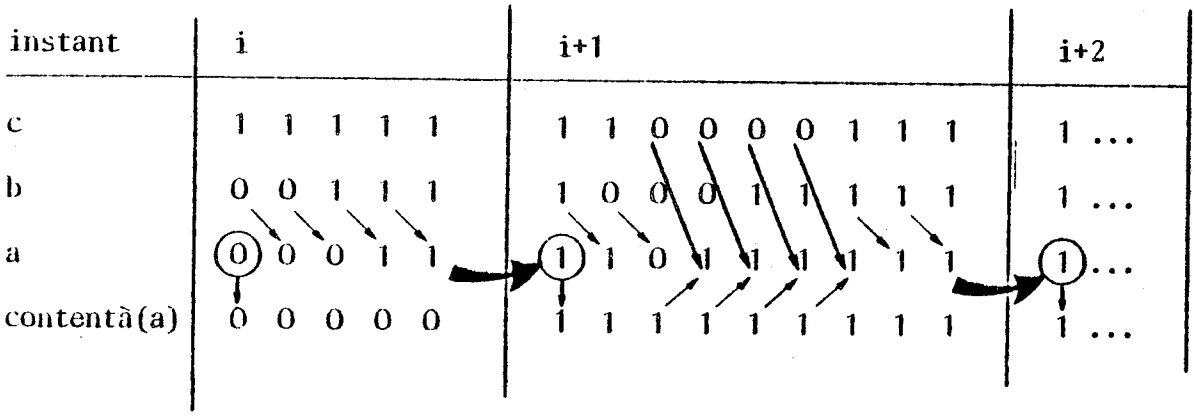
Affectation dans une porteuse de type "variable"

IF c THEN a := b ENDIF



Transfert dans une porteuse de type "rtvariable"

IF c THEN a ← b ENDIF



ANNEXE

Nous donnons ci-dessous la définition abrégée des types présentés au paragraphe III.2. Les domaines, les instructions de transport d'opérations et la fonctionnalité de toutes les opérations définies dans le corps de ces types sont explicités; par contre, nous n'avons pas détaillé les corps des opérations ni les extensions syntaxiques. Nous renvoyons le lecteur à [PBB.81] pour une définition complète.

Les intervalles

```
TYPE range BODY
 ALL a : inttuple WITH
 FORALLà i : bint(1, sizeà(a)) IS
      a[i] = a[1] + i-1 ENDFOR |
 FORALLà i : bint(1, sizeà(a)) IS
      a[i] = a[1] - i+1 ENDFOR ENDALL

 CARRY = , ∿=, sizeà, select, tail ENDCARRY

 FUNCTION lbound(x : range) : int
          "/renvoie la borne gauche/"

 FUNCTION lbound(x : range) : int
          "/renvoie la borne droite/"

 FUNCTION isin(i : int; x : range) : bool
          "/vrai si i est dans x/"

 FUNCTION position(i : int; x : range) : nnint
          "/rang de i dans x oubien 0/"

 FUNCTION normalize(x : range) : range
          "/renvoie le range allant de 1 à la taille de x/"

 FORMATà
          "/ 3 : 6 dénotera (. 3, 4, 5, 6 .)/"
ENDrange
```

Les dimensions de tableau

```
TYPE array_dimension BODY
 ALL a : tytupleà(range) WITH sizeà(a) > 0 ENDALL
 CARRY =, ∿=, sizeà, tail, select, exchange ENDCARRY

 FUNCTION isin(z : inttuple; y : array_dimension) : bool

    "/vrai si z et y ont même taille et si chaque élément de z est dans le
      "range" correspondant de y/"

 FUNCTION normalize(x : array_dimension) : array_dimension
    "/renvoie des dimensions ascendantes à partir de 1/"
```

```
 FUNCTION size_array(x : array_dimension) : pint
    "/nombre d'éléments d'un tableau ayant x comme dimensions/"
FUNCTION position(z : inttuple; y : array_dimension)
    "/place d'un élément de tableau/"
 FUNCTION ordinal_dimension(p : pinttuple) : array_dimension
    "/renvoie des dimensions normalisées, la ième dimension ayant p[i]
      éléments/"

 FORMATà
  "/ 3 : 6;-1 : 1 dénotera (. 3 : 6,-1 : 1 .) /"
ENDarray_dimension
```

Les indices de tableau

```
TYPE indexer BODY
 ALL a : tytupleà(inttuple) WITH sizeà(a) > 0 ENDALL
 CARRY =, ∿=, sizeà, tail, select, exchange ENDCARRY

 FUNCTION isin(z : indexer; y : array_dimension) : bool
    "/vrai si tous les indices sont dans les bornes déclarées du tableau/"

 FUNCTION degenerate(z : indexer) : bool
    "/vrai si l'indice sélectionne un scalaire/"

 FUNCTION élément_tuple(z : indexer) : inttuple
    "/transforme z en liste simple, si z est dégénéré/"

 FUNCTION index_map(x : inttuple; z : indexer) : inttuple
    "/indexation de z par x/"

 FUNCTION derived_dimension(z : indexer) : array_dimension
    "/renvoie les dimensions normalisées du sous-tableau indexé par z/"

 FORMATà
   "/3 : 6, 4, 1; -3 : 1, 8, 8, 4 dénotera
          (.(. 3, 4, 5, 6, 4, 1 .), (-3, -2, -1, 0, 1, 8, 8, 4 .).)/"
ENDindexer
```

Indices qui sélectionnent un seul élément

```
SUBTYPE degenerate_indexer BODY
 ALL i : indexer WITH degenerate(i) ENDALL
ENDdegenerate_indexer
```

Indices qui sélectionnent un sous-tableau

```
SUBTYPE nondegenerate_indexer BODY
 ALL i : indexer WITH ∿ degenerate(i) ENDALL
ENDnondegenerate_indexer
```

Les tableaux à éléments dans un type "u"

```
TYPE array(u : anyà) BODY
 ALL y : tupleà WITH sizeà(y) = z &
                     y[1] .<  array_dimension &
                     y[2] .<  tytupleà(u) &
                   sizeà(y[2]) = size_array(y[1])
 ENDALL

 FUNCTION dpart(x : array(u)) : array_dimension
    "/renvoie la partie dimensions/"

 FUNCTION vpart(x : array(u)) : tytupleà(u)
    "/renvoie la partie valeur/"

 FUNCTION lb(x : array(u); d : pint) : int
    "/renvoie la borne gauche de la dième dimension/"

 FUNCTION rb(x : array(u); d : pint) : int
    "/renvoie la borne droite de la dième dimension/"

 FUNCTION size_dimension(x : array(u); d : pint) : pint
    "/nombre d'éléments dans la dième dimension/"

 FUNCTION compatible(x, y : array(u)) : bool
    "/vrai si x et y ont des dimensions compatibles/"

 FUNCTION normalize(x : array(u)) : array(u)
    "/renvoie un tableau de dimensions normalisées avec même partie valeur/"

 FUNCTION select_element(x : array(u); z : degenerate_indexer) : u
    "/indexe un scalaire/"

 FUNCTION select_slice(x : array(u); z : nondegenerate_indexer) : array(u)
    "/indexe un sous-tableau/"

 FUNCTION transpose(x : array(u); i, j : pint) : array(u)
    "/transposition des dimensions i et j/"
```

```
FUNCTION equal(x, y : array(u)) : bool
   RETURN compatible(x, y) &
          vpart(x) = vpart(y)
    "/égalité de tous les éléments 2 à 2 notée = /"
ENDequal

FUNCTION not_equal(x, y : array(u)) : bool
   RETURN ∿ (x = y)
    "/inégalité d'au moins 1 couple d'éléments notée  ∿= /"
ENDnot_equal

FUNCTION eq(x, y : array(u)) : array(bool)
   "/comparaison des éléments 2 à 2,
     égalité notée EQ/"

FUNCTION neq(x, y : array(u)) : array(bool)
   "/inégalité des éléments 2 à 2,
     négation de EQ, notée NEQ/"

FUNCTION fbinà(x, y : array(u); ATT f : FUNCTION(u, u) : u) : array(u)
   "/extension d'une opération binaire interne sur u aux tableaux de u/"

FUNCTION fboolà(x, y : array(u); ATT f : FUNCTION(u, u) : bool) : array(bool)
   "/extension des opérations de comparaison sur u aux tableaux de u/"

FUNCTION fmonà(x : array(u); ATT f : FUNCTION(u) : u) : array(u)
   "/extension d'une opération unaire interne sur u aux tableaux de u/"

ACTIVITY abinà(W x : array(u); W y : array(v); ATT g : ACTIVITY(u, v))
   "/extension d'une activité à 2 opérandes scalaires aux tableaux. Sera
     utilisé pour les opérations d'affectation sur tableaux/"

FUNCTION catenate(x, y : array(u); d : pint) : array(u)
   "/concaténation selon la dième dimension/"
ENDarray
```

Tableaux à éléments de type "u" et de dimensions fixées

```
SUBTYPE fixed_array(u : anyà; d : array_dimension)
  BODY
  ALL x : array(u) WITH dpart(x) = d ENDALL
ENDfixed_array
```

Descripteur d'un champ d'enregistrement

```
TYPE field_descriptor BODY
  ALL x : tupleà WITH sizeà(x) = 2 &
                      selectà(x, 1) .<  string &
                      selectà(x, 2) .<  anyà ENDALL
  CARRY =, ∿= ENDCARRY

  FUNCTION name_partà(x : field_descriptor) : string
      "/renvoie le sélecteur de champ/"

  FUNCTION type_partà(x : field_descriptor) : anyà
      "/renvoie le type du champ/"

  FORMATà
      "/'ABC' : int dénotera (. 'ABC', int.)/"
ENDfield_descriptor
```

Liste de descripteurs de champ

```
TYPE field_descriptor_list BODY
   ALL x : tytupleà(field_descriptor) WITH
     FORALLà i : bint(1, sizeà(x)) IS
     FORALLà j : bint(1, sizeà(x)) IS
       i = j |
       name_partà(x[i]) ∿= name_partà(x[j])
     ENDFOR ENDFOR ENDALL

   CARRY =, ∿=, sizeà, select ENDCARRY

   FUNCTION field_positionà(x : field_descriptor_list; a : string) : pint
       "/renvoie le rang du sélecteur a dans x/"

   FORMATà
       "/'ABC' : int; 'dE' : bool dénotera (. 'ABC' : int, 'dE' : bool .)/"
ENDfield_descriptor_list
```

Les enregistrements

```
TYPE record(u : field_descriptor_list) BODY
   ALL x : tupleà WITH sizeà(x) = sizeà(u) &
       FORALLà i : bint(1, sizeà(u)) IS
       selectà(x, i) .< type_partà(u[i]) ENDFOR
   ENDALL
```

```
    CARRY =, ∿= ENDCARRY

    FUNCTION record_select(x : record(u); a : string) :
                type_partà(u[field_positionà(u, a)])
        "/sélection de l'élément dont le nom de champ est a. On notera x!a/"
ENDrecord
```

Séquences de valeurs au cours des pas de calcul d'un intervalle de temps

```
TYPE cs_signalà(u : anyà) BODY
 ALL a : tytupleà(u) WITH sizeà(a) > 0 ENDALL

 CARRY =, ∿=, sizeà ENDCARRY

 FUNCTION select_css(y : cs_signalà(u); i : pint) : u
     "/sélection de la valeur au ième pas de calcul/"

 FUNCTION extend_css(y : cs_signalà(u); i : pint; v : u) : cs_signalà(u)
     "/renvoie une séquence égale à
     - y augmenté de v si i = sizeà(y) + 1,
     - y si i ≤ sizeà(y) et si y[i] = v,
     sinon erreur/"

 FUNCTION reduce_css(y : cs_signalà(u); i, j : pint) : cs_signalà(u)
     "/renvoie une séquence égale à y privé des éléments de rang compris entre
       i et j, bornes exclues/"
ENDcs_signalà
```

Historique des valeurs au cours des intervalles de temps successifs

```
TYPE signalà(u : anyà) BODY
 ALL a : tytupleà(cs_signalà(u)) WITH
     sizeà(a) > 0 ENDALL

 CARRY =, ∿=, sizeà ENDCARRY

 FUNCTION select_rts(y : signalà(u); t : pint) : cs_signalà(u)
     "/sélection de la séquence de valeurs au cours de l'intervalle t/"

 FUNCTION known(y : signalà(u); t, s : pint) : bool
     "/vrai si il existe une valeur pour le pas de calcul s de l'intervalle t/"

 FUNCTION inst_value (y : signalà(u); t, s : pint) : u
     "/sélection de la valeur au pas de calcul s de l'intervalle t, notée y{t, s}
```

On notera en outre

y { } : sélection de la valeur au dernier pas de calcul du dernier intervalle

y{t,} : sélection de la valeur au dernier pas de calcul de l'intervalle t/"

```
FUNCTION extend_rts(y : signalà(u); t, s : pint; v : u) : signalà(u)
    "/renvoie un historique égal à
      - y augmenté de v si known(y, t, s) = 0,
      - y si y{t, s} = v
      sinon erreur/"

FUNCTION reduce_rts(y : signalà(u); t, i, j : pint) : signalà(u)
    "/renvoie un historique obtenu à partir de y en supprimant, pour l'intervalle t, les valeurs de rang compris entre i et j, bornes exclues/"

INTERPRETERà FUNCTION packà(y : u) : signalà(u)
    "/renvoie un historique de valeurs constantes, égales à y, ayant un élément pour les intervalles passés, et sà éléments identiques à y pour l'intervalle courant tà/"
ENDsignalà
```

Notion de porteuse

```
TYPE signal_carrierà(u : anyà; di : u) BODY
  cellà(signalà(u))
  CARRYALL

  FUNCTION dpart(y : signal_carrierà(u, di)) : u
      "/renvoie di, la valeur par défaut/"

  FUNCTION spart(y : signal_carrierà(u, di)) : signalà(u)
      "/renvoie (.(. di .).), signal initial, si la cellule est vide,
       sinon l'historique complet de y/"

  FUNCTION delay(y : signal_carrierà(u, di); d : pint) : u
      "/renvoie la valeur de l'historique au dernier pas de calcul d instants
       auparavant si l'instant courant est supérieur à d, sinon la valeur par
       défaut di.  Le retard sera noté "%"/"

  INTERPRETERà FUNCTION contentà(y : signal_carrierà(u, di)) : u
      "/fonction de dérepérage : renvoie la valeur de l'historique de y à
       l'instant courant et au pas de calcul courant, ou par défaut di/"
```

```
  INTERPRETERà ACTIVITY shrinkà(W y : signal_carrierà(u, di); i, j : pint)
    "/supprime, dans l'historique de y, les valeurs comprises entre les pas
      de calcul i et j, bornes exclues, à l'instant courant/"
ENDsignal_carrierà
```

Fil de connexion

```
TYPE terminal(u : anyà; di : u) BODY
  signal_carrierà(u, di)

  CARRY contentà, delay, getà, putà, shrinkà ENDCARRY

  ACTIVITY connect(W y : terminal(u, di); a : u)
    "/la connexion étend l'historique de y avec la valeur a.
      On notera y .= a/"

  INTERPRETERà ACTIVITY finstepà(W y : terminal(u, di))
    "/si aucune connexion n'est active à l'étape courante, on étend l'his-
      torique de y avec di/"

  INTERPRETERà ACTIVITY finintà(W y : terminal(u, di))
    "/on reporte, comme première valeur de l'instant suivant, la valeur de
      la dernière étape de calcul de l'instant courant/"
ENDterminal
```

Variable

```
TYPE variable(u : anyà; di : u) BODY
  signal_carrierà(u, di)

  CARRY contentà, delay, getà, putà, shrinkà ENDCARRY

  ACTIVITY assign(W y : variable(u, di); a : u)
    "/l'affectation étend l'historique de y avec la valeur a. On notera y := a/"

  INTERPRETERà ACTIVITY finstepà(W y : variable(u, di))
    "/si aucune affectation n'est active à l'étape courante, on étend l'his-
      torique de y avec la valeur courante/"

  INTERPRETERà ACTIVITY finintà(W y : variable(u, di))
    "/on reporte, comme première valeur de l'instant suivant, la valeur à la
      dernière étape de calcul de l'instant courant/"
ENDvariable
```

Bascule à durée de chargement 1 unité de temps

```
TYPE rtvariable(u : anyà; di : u) BODY
  signal_carrierà(u, di)

  CARRY getà, putà, shrinkà ENDCARRY

  INTERPRETERà FUNCTION contentà(y : rtvariable(u, di)) : u
    "/le contenu est, pendant tout l'intervalle de temps courant, la valeur
      au premier pas de calcul de cet intervalle/"

  ACTIVITY transfer(W y : rtvariable(u, di); a : u)
    "/on étend l'historique de y avec la valeur a. On notera y ← a/"

  FUNCTION delay(y : rtvariable(u, di); d : pint) : u
    "/renvoie la valeur au 1° pas de calcul de l'instant tà-d si tà > d,
      sinon di/"

  INTERPRETERà ACTIVITY finstepà(W y : rtvariable(u, di))
    "/si aucun transfert n'est actif à l'étape courante, on étend l'historique
      de y avec la valeur au 1° pas de calcul de l'instant courant/"

  INTERPRETERà ACTIVITY finintà(W y : rtvariable(u, di))
    "/on reporte, comme première valeur de l'instant suivant, la valeur à la
      dernière étape de calcul de l'instant courant/"
ENDrtvariable.
```

QUATRIEME PARTIE

UN MODELE D'EVALUATION POUR LES LANGAGES DE DESCRIPTION DE SYSTEMES LOGIQUES

I INTRODUCTION

Le modèle qui est présenté ici est directement issu du modèle d'évaluation que nous avons élaboré pour le projet CONLAN. Dans le cadre de la définition d'un langage de base devant servir de support syntaxique et sémantique à une famille de langages de description de systèmes logiques, la nécessité de disposer d'un modèle d'évaluation est progressivement apparue clairement au groupe. En voici les raisons essentielles :

- Au cours des années, les malentendus au sein du groupe ont été nombreux, dûs principalement au fait que, venant de spécialités différentes, les participants mettaient sous les mêmes mots des concepts différents. Expliquer chaque construction du langage en termes de sa traduction dans un modèle suffisamment simple s'est avéré le seul moyen de mettre au jour, et de lever, de telles incompréhensions dans le groupe; et nous espérons que cet exercice évitera les incompréhensions entre le groupe et la communauté scientifique.

- L'explication d'un certain nombre de mécanismes primitifs en anglais n'était jamais totalement précise, quand elle existait. Ainsi, il a fallu attendre qu'un modèle d'évaluation des mécanismes de protection,de passages de paramètres et d'allocations d'objets soit proposé pour que chacun se rende compte que l'effet obtenu ... n'était pas du tout l'effet désiré. Pourtant, rien dans le rapport de définition sous son état d'alors ne permettait d'exclure une telle interprétation (ni plusieurs autres) puisque par exemple les règles d'allocation d'objets locaux à un segment n'avaient jamais été abordées.

- La construction de CONLAN de Base à partir de CONLAN Primitif était rigoureuse et formelle pour ce qui était des nouveaux types introduits. Mais la gestion du temps simulé et la notion de stabilisation n'étaient pas réductibles à des définitions de types ou d'opérations. Il était donc nécessaire de disposer d'un outil formel complémentaire pour exposer ces concepts, faute de quoi une partie fondamentale de la sémantique de CONLAN de Base ne recevait aucune assise.

Le choix d'un modèle d'évaluation pour les langages de description de systèmes logiques doit tenir compte des particularités de ces langages, des différents niveaux de détail à couvrir, et des objectifs de leurs utilisateurs. Ainsi, il doit permettre de représenter :

- le parallélisme au niveau des instructions et le fait que dans cette hypothèse l'ordre dans lequel les instructions sont écrites n'a aucune influence sur le résultat;
- la durée des actions, et le fait que l'on puisse référencer les valeurs de certains objets à des instants antérieurs;
- l'évaluation des valeurs de toutes les porteuses d'une description au cours du temps, en particulier lorsque celles-ci n'ont pas été explicitement modifiées;

en plus des notions habituelles dans les langages de programmation :

- déclarations et portées des déclarations;
- invocations d'opérations et passages de paramètres;
- définitions de nouveaux types d'objets, de nouvelles opérations.

Par ailleurs, l'un des objectifs essentiels est que le modèle d'évaluation soit une spécification précise de l'interprétation à donner à un texte écrit dans le langage considéré, que cette interprétation soit faite par un lecteur ou par une machine. Dans le cas d'un lecteur, qui très probablement voit derrière la plupart des porteuses et des opérateurs de certains niveaux de langage un objet matériel déterminé, un modèle trop ésotérique, faisant appel à des notions trop éloignées des concepts habituels en conception de systèmes, ne l'aiderait aucunement à comprendre la sémantique du langage.

Cet ensemble de critères rend inadéquates les méthodes d'expression de la sémantique les plus connues.

Les approches opérationnelles [Liv.78][Weg.72] ont été développées pour des programmes séquentiels, et deviennent très artificielles lorsque l'on veut exprimer le parallélisme. L'approche dénotationnelle [Don.79, Sto.77] est tout aussi inadaptée en pareil cas. Les méthodes axiomatiques [Hoa.69], qui ont pour but de conduire à des preuves de correction de programmes, n'ont pas donné de résultats significatifs dès lors que l'on veut prendre en compte les aspects temporels. Une des rares tentatives pour décrire des systèmes

logiques par une approche fonctionnelle [FrS.79], inspirée par le célèbre article de Bacchus [Bac.78], se gardait bien de modéliser durées et retards, ni même des objets sans propriété de mémorisation. Enfin, les modèles basés sur les graphes de contrôle et de données [Bae.73, Mil.73] s'attachent essentiellement à décrire les interactions entre des processus séquentiels concurrents, en vue de démontrer ou vérifier certaines propriétés des systèmes logiques distribués. A ce titre, ces modèles, et les langages qu'ils ont inspirés, relèvent eux-mêmes de la description des systèmes logiques, plutôt que d'un modèle sémantique pour les langages de description.

En définitive, c'est un modèle pour la résolution de problèmes, en intelligence artificielle, qui nous a semblé le plus riche en concepts utilisables. Le modèle des acteurs développé par Carl Hewitt [Hew.77] a précisément pour objet de représenter un ensemble d'experts travaillant de manière simultanée et parallèle à l'accomplissement d'une tâche. L'accent est mis sur la décentralisation des décisions, et un mode unique d'interaction par échange de messages. Le fait que ce modèle ait été mis en oeuvre sur ordinateur, via le langage PLASMA, est aussi un point attractif, puisque nous cherchons en fait à spécifier un interpréteur de nos langages de description. Un modèle opérationnel, adapté à la spécification du parallélisme, nous semble être le moyen le plus direct et le plus compréhensible pour parvenir à cet objectif. Cependant, reprendre le modèle des acteurs tel quel pour notre application présentait un certain nombre de difficultés :

- les cellules, seuls acteurs ayant un état interne, ne sont pas typées, et sont en fait des objets trop primitifs
- il n'y a pas de différence entre définition et création d'un acteur, localement à un acteur englobant
- le fait que la réponse à une requête puisse être envoyée à un autre acteur que l'émetteur du message initial est sans doute une optimisation pour l'exécution de textes écrits en PLASMA, mais complique excessivement les échanges, pour qui cherche avant tout un modèle d'exécution.

Nous proposons donc un nouveau modèle d'évaluation, qui permet de mieux décrire les mécanismes typiquement "langage" tels que passages de paramètres, accès aux objets globaux, détection dynamique d'erreurs, qui sont toujours les moins bien définis pour les langages de description de systèmes logiques. Très inspiré des principes des "systèmes d'acteurs", notre modèle conserve les notions d'experts travaillant en parallèle, et communiquant par messages.

Cependant, nous particularisons ces experts en différentes catégories, ayant des comportements différents, et caractérisés par les messages qu'ils sont capables de comprendre et d'émettre. Dans notre modèle, les messages deviennent des objets passifs, et sont classés en un nombre fixe de catégories, reconnaissables par un mot-clé. Enfin, nous introduisons la notion de recrutement : on admet l'existence d'un réservoir inépuisable d'experts, qui sont explicitement acquis et renvoyés, ce qui permet de modéliser différentes politiques d'allocation d'objets.

Le résultat étant un modèle significativement différent de celui de Hewitt, nous n'emploierons pas le mot "acteur", afin de ne pas orienter le lecteur vers de fausses analogies.

VOCABULAIRE

Dans la suite, un certain nombre de mots seront employés indifféremment :

- nom et identificateur
- ordre, commande, requête
- réponse, acquittement, compte-rendu.

II PRESENTATION DU MODELE

L'évaluation d'une description est, intuitivement, comparable à une tâche complexe confiée à une équipe. Un chef d'équipe est responsable du démarrage du travail, et de l'obtention du résultat. La tâche globale est décomposée en tâches plus simples, elles-mêmes décomposées jusqu'à obtenir des tâches élémentaires qu'une personne seule peut accomplir.

Une tâche, qu'elle soit simple ou complexe, consiste essentiellement à produire des informations, et éventuellement à changer les informations qui sont contenues dans des tiroirs. Un tiroir est susceptible de contenir 0 ou 1 information d'un certain type. S'il contient une information, celle-ci peut être remplacée par une autre information de même type, sous certaines conditions, mais le tiroir ne peut plus redevenir vide.

II.1 Les agents

Nous appellerons agent chaque membre de l'équipe, y compris le chef. Un agent est associé à chaque tâche à accomplir, et à chaque tiroir. L'équipe est donc constituée d'agents ouvriers (responsables des tâches) et d'agents magasiniers (gardiens des tiroirs). Ce sont là les deux principales propriétés bien spécifiques.

Ouvriers et magasiniers peuvent être embauchés et renvoyés à tout moment, selon les besoins. Nous admettrons l'existence d'un agent recruteur, extérieur à l'équipe, mais que tout le monde connaît, et à qui n'importe quel agent ouvrier peut s'adresser pour obtenir des ouvriers et des magasiniers qui seront placés sous sa tutelle.

Ainsi, la structure de l'équipe est hiérarchique. Si l'on ne considère que les ouvriers, il s'agit même d'une arborescence, qui correspond à la décomposition de la tâche principale en sous-tâches. Mais deux sous-tâches peuvent se servir du même tiroir, ce qui fait qu'un magasinier peut dépendre de plusieurs ouvriers. En revanche, un ouvrier ne dépend jamais d'un magasinier. Le chef d'équipe est donc toujours un ouvrier.

Un ouvrier démarre sa tâche quand il en reçoit l'ordre de son supérieur hiérarchique immédiat; lorsque la tâche est terminée, il rend compte à celui-ci. De même, un magasinier consulte ou remplace le contenu de son tiroir sur ordre d'un ouvrier (un ouvrier n'accède jamais directement à un tiroir). Cependant, tous les tiroirs ne sont pas accessibles à tous les ouvriers : de façon générale, un tiroir porte une ou plusieurs marques, et seuls les ouvriers portant aussi l'une de ces marques peuvent y accéder, en fournissant cette marque au magasinier qui procède aux vérifications préalables. En outre, le passage de ce premier barrage permet seulement de consulter le contenu du tiroir, dans la mesure où toute modification a pu être temporairement interdite par quelqu'un d'autre. Dans tous les cas, le magasinier rendra compte à l'ouvrier demandeur de la satisfaction ou du refus de sa requête. Après que son compte-rendu ait été reçu, un agent (ouvrier ou magasinier) est selon les cas renvoyé ou conservé inactif.

Lorsqu'un ouvrier responsable d'une tâche non primitive reçoit l'ordre d'accomplir sa tâche, il peut disposer déjà d'une équipe inactive, ou être seul.

Dans le premier cas, il met ses ouvriers au travail. Sinon, il commence par embaucher ses équipiers, puis il donne l'ordre aux ouvriers d'effectuer leur tâche, et les renvoie après avoir reçu leur compte-rendu. Certaines tâches sont toujours accomplies par des ouvriers permanents, d'autres sont toujours confiées à des intérimaires.

Les agents sont très spécialisés, et ne peuvent qu'accomplir la tâche ou la gestion du tiroir pour laquelle ils ont été embauchés. Un agent ne peut satisfaire qu'une requête à la fois. Si un agent reçoit plusieurs requêtes simultanément, il en sélectionne une, les autres étant conservées pour être satisfaites ultérieurement. Lorsque toutes les requêtes ont été sélectionnées, et acquittées, l'agent redevient inactif, et en attente de requête.

Enfin, tous les agents ont un nom unique.

II.2 Les messages

Les requêtes arrivent sous forme de messages, et chaque message porte le nom de son expéditeur. Toute requête obtient une réponse en sens inverse, cette réponse étant faite elle aussi par message. Une requête n'est jamais considérée par un agent avant que la réponse à la requête précédemment prise en compte par cet agent n'ai été envoyée.

L'envoi de messages est l'unique mode de communication entre agents. De plus, pour qu'un agent communique avec un autre agent, il doit au préalable le connaître : les messages sont toujours nominatifs.

Un agent A connaît un agent B si :

- A est supérieur hiérarchique immédiat de B
- B est supérieur hiérarchique immédiat de A
- le nom de B a été écrit sur un message adressé à A
- B est l'agent recruteur.

La relation "connaît" définie sur les agents n'est ni transitive, ni symétrique, ni anti-symétrique.

a) Arrivée des messages

Le déroulement de la tâche globale s'effectue donc par échange de messages. Certains messages sont ordonnés dans le temps : le message qui commande une tâche précède toujours le message qui en annonce l'achèvement. D'autres messages, relatifs à des tâches "soeurs" dans l'arborescence de décomposition, peuvent arriver dans un ordre quelconque. Nous considérerons que seules nous intéressent les arrivées des messages à leur destinataire, et négligerons

- le temps de transit d'un message entre son émission et sa réception
- le temps mis pour accomplir une tâche.

Nous supposons seulement que ces deux temps sont finis, et que tout message reçu est pris en compte au bout d'un temps fini.

L'accomplissement d'une tâche est donc caractérisée par un pré-ordre partiel sur les instants d'arrivée des messages à leurs destinataires. En termes informatiques, ce pré-ordre caractérise la quantité maximale de parallélisme utilisable pour réaliser l'évaluation d'une description sur un multiprocesseur : deux sous-tâches démarrées par des messages non ordonnés peuvent être exécutées en parallèle sur des processeurs différents; mais elles peuvent aussi être exécutées dans un ordre quelconque sur un mono-processeur. Le modèle n'impose donc pas une mise en oeuvre particulière.

b) Forme des messages

Il y a dans le modèle un petit nombre, fixé a priori, de catégories de messages. Un mot-clé identifie chaque catégorie, et chaque message contient le mot-clé qui le rattache à sa catégorie.

Une catégorie de message caractérise :

- l'action qui est demandée à l'agent récepteur
- l'acquittement d'une requête qui a été satisfaite
- l'acquittement d'une requête qui n'a pas pu être satisfaite.

Un agent ne peut pas recevoir ni émettre n'importe quel message. En particulier, les messages susceptibles d'être acceptés par les ouvriers et par les magasiniers sont dans des catégories différentes. Ainsi, un ouvrier reconnaîtra un message lui demandant d'effectuer sa tâche, complètement ou en partie; par contre, un magasinier reconnaîtra un message lui demandant si son tiroir est vide, ou lui demandant de modifier le contenu de son tiroir. Inversement, les réponses doivent aussi correspondre aux requêtes, lorsque celles-ci ont été satisfaites. Par contre, toutes les réponses à des requêtes refusées, et toutes les réponses qui correspondent à un déroulement anormal de la tâche ordonnée appartiennent à une même catégorie ERROR.

L'ensemble des messages qu'un agent peut émettre et recevoir caractérise le comportement de cet agent.

En fonction de sa catégorie, un message peut comporter 0, 1 ou une liste de paramètres. Ainsi, pour modifier le contenu d'un tiroir, il faut fournir en paramètre le nouveau contenu. Si la modification a été menée à bien, le message d'acquittement en réponse n'a pas de paramètre. Si la modification a été refusée, le message d'erreur a un paramètre, qui est un texte expliquant les raisons du refus.

Les messages qui sont des requêtes portent aussi une marque, appelée contexte. Enfin, tous les messages portent le nom de leur expéditeur.

II.3 Notion de contexte

Considérons l'équipe d'agents limitée aux seuls ouvriers. Cette équipe a une structure d'arborescence,dans notre modèle. Si nous admettons que le chef d'équipe est le plus grand élément de la relation d'ordre partiel définie par l'arborescence, nous dirons que :

- un agent A est chef d'un agent B si $A \neq B$ et $A \geq B$
- un agent A commande un agent B si . A est chef de B et
  . il n'existe pas d'agent C tel que A est chef de C et C est chef de B.

Tous les ouvriers de l'arborescence ont, en plus de leur nom unique, une marque appelée contexte. Le contexte d'un ouvrier est :

- soit son propre nom
- soit le nom d'un ouvrier qui est son chef.

Seuls certains ouvriers ont le privilège de donner leur nom à un contexte. Ces ouvriers là ont toujours leur propre nom comme contexte. C'est en particulier le cas du chef d'équipe.

Le contexte est attribué à un ouvrier lors de son embauche en suivant la règle ci-dessous :

> Soient A et B deux ouvriers tels que A commande B. Si B n'est pas lui-même son propre contexte, alors B a le même contexte que A.

Un ouvrier inscrit toujours son contexte sur les messages de type requête qu'il émet. Ce contexte sert en particulier à vérifier ses droits d'accés aux tiroirs, et à embaucher d'autres ouvriers ayant le même contexte que lui. En fait, l'ensemble des ouvriers qui ont le même contexte partagent un ensemble de tiroirs dont les magasiniers ont été embauchés par leur chef commun qui a fourni son nom au contexte.

II.4 Présentation intuitive du fonctionnement du modèle

a) Correspondance entre éléments d'une description de système logique et éléments du modèle

Les agents ouvriers du modèle expriment la sémantique des segments du langage : DESCRIPTION, FUNCTION, ACTIVITY. Les agents magasiniers correspondent aux porteuses. Le chef d'équipe est donc associé au segment englobant, qui est toujours un exemplaire de DESCRIPTION. L'équipe complète contient, à tout moment, un agent par :

- exemplaire de DESCRIPTION
- invocation de segment opération STATIC
- invocation active de segment opération : FUNCTION, ACTIVITY
- invocation active d'instruction
- porteuse déclarée dans chaque segment, pour chaque exemplaire ou invocation active.

Ainsi, si dans une description "d1" on trouve 3 invocations inconditionnelles d'une ACTIVITY "abc", l'évaluation de d1 sera faite par une équipe contenant 3 agents distincts identiques pour abc.

Le nombre total d'agents de l'équipe peut être modifié en cours d'évaluation. Seul un noyau d'agents est présent pendant toute la vie du modèle : ce sont les agents associés aux exemplaires de description, aux opérations statiques, et à leurs porteuses locales. Les agents qui sont embauchés et renvoyés dynamiquement en cours d'évaluation sont associés aux opérations non statiques et à leurs porteuses locales.

Les tiroirs ont volontairement été choisis comme des objets relativement complexes, dotés de capacités de verrouillage et de contrôle d'accés. Ces mécanismes sont nécessaires pour décrire :

- le partage de porteuses, à l'interface des exemplaires de description
- le sens des interfaces : IN, OUT, INOUT
- les passages de paramètre par référence ou par valeur.

Enfin, la définition de contextes permet de décrire les règles de portées des déclarations.

b) Modes de transmission des messages

Lorsqu'un concepteur décrit un système logique, il définit un segment DESCRIPTION. La compilation de ce segment correspond dans notre modèle à la création d'une tâche pour cette description et pour tout segment défini localement, et à l'embauche de tous les agents dont la durée de vie est égale à celle du segment englobant, avec établissement des liens hiérarchiques exposés précédemment.

Les échanges de messages se produisent à partir du moment où le concepteur demande une évaluation de sa description. Nous admettrons l'existence d'un environnement, servant d'interface et de superviseur de communication entre le concepteur et le modèle. C'est l'environnement qui envoie le premier message, au chef d'équipe, pour démarrer l'évaluation. Ce premier message provoque l'émission par le chef d'équipe :

- d'un ensemble de messages vers l'agent recruteur pour embaucher ses ouvriers temporaires, pour chaque invocation d'activité non statique.
- quand tous les ouvriers temporaires sont arrivés, un autre ensemble de messages, un vers chaque ouvrier qu'il commande, pour qu'il démarre sa tâche.

Le processus est alors répété récursivement, tout au long de l'arborescence des ouvriers. Chaque ouvrier qui reçoit un message de démarrage commence par procéder aux embauches nécessaires, puis envoie un message de démarrage aux ouvriers qu'il commande, jusqu'à ce que soient démarrés les ouvriers chargés de tâches primitives.

Une fois qu'il a démarré les ouvriers qu'il commande, un ouvrier non primitif attend que tous les acquittements de ses messages soient arrivés.

Si un acquittement au moins est un message d'erreur, un message d'erreur et répondu à celui qui a commandé l'évaluation. Sinon, après d'éventuelles vérifications complémentaires, l'ouvrier renvoie ses agents intérimaires et répond que sa tâche est bien terminée.

Les acquittements remontent donc l'arborescence des ouvriers en sens inverse des messages ordonnant les démarrages de tâche.

En fait, à chaque niveau, un message de démarrage de tâche comporte en général des paramètres, dont la reconnaissance peut nécessiter l'embauche d'ouvriers supplémentaires, et l'émission de messages ordonnant des calculs préalables au démarrage de la tâche proprement dite. C'est en particulier au cours de cette phase intermédiaire de reconnaissance de paramètres, qui sera développée en détails plus loin, que sont établis de nouveaux liens de dépendance hiérarchique entre les ouvriers et les magasiniers. Nous pourrons cependant admettre en première approximation que :

- tout message de démarrage qui parvient à un ouvrier O_i chargé d'une tâche non primitive a pour effet :

  . de provoquer, quand cela est utile, l'extension de l'équipe, par l'embauche d'agents temporaires que commande O_i
  . de causer la création de nouveaux messages de démarrage, envoyés par O_i aux ouvriers qu'il commande

- tout message de démarrage qui parvient à un ouvrier O_j chargé d'une tâche primitive provoque l'exécution par O_j de sa tâche, et l'envoi d'un compte-rendu à son demandeur.
- un ouvrier O_i, chargé d'une tâche non primitive, une fois qu'il a reçu de tous les ouvriers qu'il commande un compte-rendu de fin de tâche, renvoie ses intérimaires, et envoie un compte-rendu unique de fin de tâche.

Définition

Un pas d'évaluation est l'ensemble des échanges de messages (bornes comprises) entre les requêtes de démarrage d'exécution de tâche envoyées par le chef d'équipe, et les réponses à ces requêtes.

A la fin d'un pas d'évaluation, le chef d'équipe décide de procéder à un nouveau pas d'évaluation, ou bien envoie directement un acquittement à l'environnement, en fonction de la sémantique du langage dans lequel la description modélisée est écrite, et de "conditions de stabilité" qui seront exposées plus loin.

Règle

Un pas d'évaluation est sans erreur si et seulement si il ne contient aucun message d'erreur.

Un énoncé équivalent de cette règle est :

Un pas d'évaluation est sans erreur si et seulement si le chef d'équipe ne reçoit aucun message d'erreur pour ce pas.

III LE MODELE D'EVALUATION

Le modèle d'évaluation par équipe d'agents est constitué :

- d'un univers d'objets prédéfinis ou construits
- d'un ensemble d'agents, organisés en équipe hiérarchisée, et classés en catégories
- de 2 agents particuliers, différents de tous les autres et hors de l'équipe, appelés environnement et agent recruteur
- d'un ensemble de messages échangés par les agents.

Le choix de l'univers d'objets pré-définis et des catégories d'agents disponibles pour constituer une équipe particulière détermine la sémantique du langage de description à évaluer.

III.1 Les objets

Les objets du modèle sont tous les éléments de tous les types de valeurs prédéfinis ou construits dont on dispose pour décrire un système logique dans un langage donné.

Dans les langages de description, on a coutume d'accepter :

- plusieurs dénotations pour la même valeur (10, AH, 1010B sont trois écritures, dans des bases différentes, pour représenter le 11ème entier naturel).
- une même dénotation pour des valeurs différentes (0 peut signifier le booléen "faux", le ternaire "faux", l'entier "zéro").

Dans l'univers des objets du modèle, tous les objets sont distincts, et reconnaissables hors contexte par une dénotation unique. Nous considérerons que l'univers contient, quel que soit le langage modélisé :

- les entiers relatifs, notés en base 10 et soulignés (ex : -10, 0, 12)
- les booléens notés vrai, faux.

Nous admettrons que tous les agents connaissant tous les objets de l'univers, et la partition de cet univers en types.

III.2 Les messages

Un message est un quadruplet dont le second et le quatrième composant peuvent être vides, selon la catégorie du message :
(nom-d'agent, contexte, mot-clé, liste de paramètres)

- . nom-d'agent est le nom de l'expéditeur du message
- . contexte est le contexte de l'expéditeur du message
- . mot-clé détermine à quelle catégorie le message appartient.

Nous noterons, pour plus de lisibilité :
nom-d'agent : contexte, mot-clé (liste de paramètres)
un message ayant ses 4 composants;
nom-d'agent : mot-clé (liste de paramètres)
un message pour lequel la partie contexte est vide;
nom-d'agent : mot-clé
un message sans contexte ni paramètre.

a) Catégories de messages

Soit "a" un agent de l'équipe, "c" son contexte, "recruteur" l'agent recruteur.

Les messages échangés pendant un pas d'évaluation appartiennent à l'une des catégories suivantes :

- a : c, EVAL(p1, p2, ... pn)
  Requête adressée à un agent ouvrier, lui demandant d'évaluer ses paramètres effectifs

- a : EVALUATED
  Réponse d'un agent ouvrier qui a évalué avec succès tous ses paramètres effectifs.

- a : c, START
  Requête adressée à un agent ouvrier ayant évalué avec succès tous ses paramètres, afin qu'il accomplisse sa tâche. Pas de résultat attendu en retour.

- a : TYPE
  Requête adressée à un agent magasinier, lui demandant le type de son contenu.

- a : GET
  Requête adressée à un agent magasinier, lui demandant son contenu.

- a : EMPTY
  Requête adressée à un agent magasinier, lui demandant s'il est vide.

- a : SET
  Requête adressée à un agent magasinier pour bloquer temporairement toute modification de son contenu.

- a : RESET
  Requête adressée à un agent magasinier pour autoriser à nouveau des modifications de son contenu.

- a : c, PUT(obj)
  Requête adressée à un agent magasinier lui demandant que "obj", un objet de l'univers du modèle, devienne son nouveau contenu.

- a : c, COMPUTE (p1, p2, ... pm)
  Requête adressée à un agent ouvrier, lui demandant d'évaluer ses paramètres effectifs et d'accomplir sa tâche. Un résultat est attendu en retour.

- a : DONE
  Réponse sans résultat d'un agent qui a pu satisfaire avec succès une requête de catégorie START, SET, RESET, PUT.

- a : RESULT(res)
  Réponse, avec résultat "res", d'un agent qui a pu satisfaire avec succès une requête de catégorie TYPE, GET, EMPTY, COMPUTE.

- a : ERROR(explication)
  Réponse d'un agent qui n'a pas pu satisfaire la requête qu'il avait reçue. Un message de cette catégorie peut remplacer un message de catégorie EVALUATED, DONE, RESULT.
  "explication" est une chaîne de caractères qui explique les raisons de l'échec.

- a : c, SEND(désignation de tâche)
  Requête adressée à l'agent recruteur, pour embaucher un agent ouvrier capable d'effectuer la tâche spécifiée.

- a : c, SEND(désignation de type)
  Requête adressée à l'agent recruteur, pour embaucher un agent magasinier, gérant un tiroir pour des valeurs du type spécifié.

- recruteur : RECEIVE(nom-d'agent)
  Réponse de l'agent recruteur à une requête d'embauche, fournissant le nom de l'agent demandé. Dans le cas de l'embauche d'un magasinier, son tiroir est initialement vide.

Dans les messages de catégorie EVAL et COMPUTE, les pi sont les paramètres effectifs que l'ouvrier doit évaluer. Un paramètre effectif peut être :

- un objet de l'univers
- un nom d'agent
- une expression en notation polonaise préfixée.

b) Représentation de l'arrivée des messages

Soient a1, a2, ... an des agents. L'arrivée d'un message à son destinataire sera notée avec une flèche pointant vers le récepteur.

message1 → a1 se lit "a1 reçoit message1"
a2 ← message2 se lit "a2 reçoit message2".

On préfèrera la première forme pour les requêtes, la seconde pour les réponses.

Pour représenter un ensemble d'arrivées de messages, on distinguera les messages non ordonnés dans le temps des messages séquentiels. La notation suivante est adoptée pour des messages non ordonnés :

message1 → a1
message2 → a2
.....
messagen → an

tandis que les messages séquentiels sont notés :

message1 → a1
+
message2 → a2
+
message3 → a3
+
:
:
messagen → an

Ces deux notations peuvent être étendues, en se combinant, à des groupes de messages; ainsi, la forme suivante indique deux groupes de messages tels que les messages soient séquentiels dans chaque groupe, mais les groupes ne sont pas ordonnés :

```
(messagei1 → ai1
   +
messagei2 → ai2
   +
......
messageip → aip)
(messagej1 → aj1
   +
messagej2 → aj2
   +
....
messagejq → ajq)
```

III.3 Définition du comportement d'un agent

Les interactions d'un agent avec les autres agents de l'équipe qu'il connaît se font par acceptation de messages et envoi de messages. L'agent qui reçoit un message réagit en fonction de la tâche dont il est chargé. Nous présentons dans ce paragraphe un langage simple permettant de décrire ces tâches, c'est-à-dire essentiellement les messages acceptables, les décisions à prendre en fonction des messages reçus, et les messages à envoyer.

a) Squelette de message

Une tâche peut être pré-définie au modèle d'évaluation, et de surcroît être exécutée en de multiples exemplaires par des agents différents d'une même équipe. Aussi, la description des messages acceptables est-elle forcément paramétrée, par exemple par le nom de son expéditeur. Un squelette de message est une telle description. C'est un quadruplet, de mêmes composants qu'un message, et écrit selon une syntaxe identique, dans lequel tous les composants sauf le mot-clé sont des éléments formels.

Les éléments formels d'un squelette de message sont des identificateurs dont le premier caractère est un "$". Les éléments formels correspondant aux composants nom-d'agent et contexte sont entièrement déterminés par position. Les éléments formels correspondant :

- à la liste de paramètres des messages de catégorie EVAL et COMPUTE,
- au paramètre des messages de catégorie PUT

doivent être typés, comme cela est fait au niveau des langages de description pour les paramètres formels des définitions de segment.

Cas particuliers

- les noms d'agent "recruteur" et "environnement" peuvent apparaître explicitement, au lieu d'un élément formel
- les noms formels "$moi" et "$mien" sont prédéfinis, et désignent toujours l'agent exécutant la tâche et son propre contexte.

Exemples

$a : $c, EVAL($p1 : int, $p2 : bool)
$2 : EVALUATED
recruteur : RECEIVE($007)
$moi : $mien, SEND(int.=)
$moi : TYPE

Règles

La portée d'un nom formel est limitée à la tâche dans laquelle il apparaît.

Dans une tâche, toutes les occurrences d'un même nom formel designent un même agent ou objet.

b) Evaluation des noms formels

Lorsqu'un message est reçu par un agent, et que le mot-clé a été identifié avec celui d'un squelette de message prévu, les noms formels du squelette sont évalués.

b.1) Noms d'agents

Les noms formels qui correspondent à des noms d'agents (expéditeur, contexte, paramètre d'un RECEIVE) sont directement remplacés par le composant correspondant du message reçu.

b.2) Paramètres typés

Les paramètres effectifs sont comparés en nombre et en type aux paramètres formels; dans le cas où un paramètre effectif est une expression ou un nom d'agent, se reporter au paragraphe III.3.6.

Si le nombre de paramètres effectifs est incorrect, ou si un paramètre effectif après calculs éventuels, n'est pas du bon type, le message est refusé. Sinon, chaque nom formel est remplacé par le paramètre effectif reconnu en position correspondante.

Lorsqu'un nom formel a été identifié, toutes les occurrences de ce nom formel dans la tâche sont remplacées par l'objet ou l'agent qu'il désigne, pour la durée de l'exécution en cours.

c) Réception et émission

La spécification d'un message acceptable, dans une description de tâche, s'écrit

** squelette de message

"**" se lit "recevoir".

La spécification d'un message à envoyer, dans une description de tâche, s'écrit

squelette de message =>récepteur

ou bien

récepteur<= squelette de message

"<=" ou "=>" se lit "envoyer".

"récepteur" est soit un nom formel, soit l'agent recruteur, soit l'environnement.

Spécifications d'émission et de réception de message peuvent être ordonnées ou non ordonnées, individuellement ou par groupe, en utilisant les mêmes conventions que celles adoptées pour les arrivées de messages effectifs à leurs destinataires effectifs (voir paragraphe III.2.2).

Exemple

```
(1)    ** $1 : $2, COMPUTE($p : int)
              +
(2)       $moi : $mien, SEND(int.*) =>recruteur
              +
(3)       ** recruteur : RECEIVE($mult)
              +
(4)          $moi : $mien, COMPUTE($p, 2) =>$mult
              +
(5)       ** $mult : RESULT($2foisp)
              +
(6) $1<= $moi : RESULT($2foisp)
```

Le texte ci-dessus est la description d'une tâche de multiplication par 2. Cette tâche est exprimée par 6 émissions ou réceptions séquentielles de messages, qui se lisent comme suit :

(1) Recevoir de $1 avec le contexte $2 un message COMPUTE à 1 paramètre de type entier.
(2) Recruter un agent pour effectuer la multiplication sur les entiers.
(3) Recevoir cet agent, appelé $mult.
(4) Envoyer à $mult un message COMPUTE avec les paramètres $p et 2.
(5) Recevoir de $mult le résultat de la multiplication.
(6) Envoyer ce résultat à $1.

Soit "foisdeux" le nom d'un agent chargé de cette tâche; "a" et "b" les noms de l'agent qui le commande et de son contexte; "m" le nom d'un agent chargé de la multiplication. Une trace d'exécution de la tâche que nous venons de décrire, si 10 est le paramètre fourni au message initial, est :

```
a : b, COMPUTE(10) → foisdeux
        +
    foisdeux : b, SEND(int.*) → recruteur
        +
    foisdeux ← recruteur : RECEIVE(m)
        +
    foisdeux :  b, COMPUTE(10, 2) → m
        +
    foisdeux ← m : RESULT(20)
        +
a ← foisdeux : RESULT(20)
```

d) Formes conditionnelles

Comme dans PLASMA, nous adoptons deux formes conditionnelles, de syntaxe assez proche. La première permet de tester la valeur attribuée à un nom formel, la seconde teste un message reçu. Le résultat du test, dans les deux cas, permet de poursuivre l'exécution de la tâche en sélectionnant une sous-tâche parmi plusieurs alternatives.

L'écriture adoptée s'apparente à celle de l'instruction "CASE".

d.1) Test d'un nom formel

Le test d'un nom formel s'écrit :

```
SELECT nom-formel
 (valeur1 : sous-tâche1)
 (valeur2 : sous-tâche2)
   :
 (valeurn : sous-tâchen)
 (OTHER : sous-tâchen+1)
ENDSELECT
```

où valeur1, valeur2, ... valeurn sont des objets de l'univers différents deux à deux. Si le nom-formel, après évaluation, est identique à l'une des n valeurs listée, la sous-tâche correspondante est sélectionnée. Sinon, la sous-tâche préfixée par OTHER est sélectionnée.

d.2) Test d'un message

Le test d'un message s'écrit :

```
DECIDE
  (** squelette1 + sous-tâche1)
  (** squelette2 + sous-tâche2)
      :
  (** squelettem + sous-tâchem)
  (** OTHER + sous-tâchem+1)
ENDDECIDE
```

où
squelette1, squelette2, ... squelettem sont des squelettes de messages de catégories deux à deux différentes. Si le message reçu a le même mot-clé que celui de l'un des m squelettes, les noms formels du squelette sont évalués en fonction du message reçu, et si le message est accepté la sous-tâche correspondante est sélectionnée. Si le message reçu a un mot-clé différent, la sous-tâche préfixée par le squelette OTHER est sélectionnée.

d.3) Règles communes aux deux formes

- Les sous-tâches correspondent à une ou plusieurs alternatives peuvent être vides.
- L'exécution d'une forme conditionnelle est terminée quand l'exécution de la sous-tâche sélectionnée est terminée.
- L'alternative préfixée par OTHER peut être absente, en particulier lorsque tous les cas ont été explicitement prévus.
- Si l'alternative OTHER est absente, et que le nom formel prend une valeur non prévue (cas du SELECT) ou que le message reçu ne correspond avec aucun squelette prévu (cas du DECIDE), la forme conditionnelle ne peut pas être exécutée, et une erreur est détectée. Une condition d'erreur est positionnée (voir III.3.6).

e) Instructions spéciales

e.1) Initialisation

Certaines tâches, pour être entièrement définies, supposent que l'embauche de certains agents soit réalisée avant toute exécution de la tâche proprement dite. Les agents ainsi recrutés ont une durée de vie égale à celle de l'agent chargé de la tâche, et sont invoqués à toutes les exécutions de la tâche.

L'ensemble des échanges de messages nécessaires à cette initialisation est parenthésé par les mots-clé INITIAL et ENDINITIAL. Ces échanges ont lieu une fois et une seule, avant toute reconnaissance de message reçu.

e.2) Renvoi d'agents

Les agents recrutés en cours d'exécution d'une tâche ont une durée de vie qui n'excède pas la durée de l'exécution pendant laquelle ils ont été recrutés. Leur renvoi explicite s'écrit :

FIRE(liste de noms formels)

Cette instruction a pour effet de faire perdre le bénéfice de l'évaluation préalable des noms formels, qui avaient été identifiés avec des noms d'agents à la réception des messages de catégorie RECEIVE. Après exécution d'un FIRE, les noms des agents renvoyés ne sont plus accessibles.

e.3) Arrêt de la tâche

La fin normale de l'exécution d'une tâche a lieu quand tous les échanges de messages prévus ont été exécutés.

La rencontre de l'instruction STOP a pour effet de provoquer l'arrêt de l'exécution en cours. Cette instruction est en fait une facilité d'écriture, permettant d'éviter de trop nombreuses imbrications de formes conditionnelles.

e.4) Définition de macro

Une autre facilité d'écriture, permettant de réduire la taille des textes de spécification de tâche, est donnée par la possibilité de nommer un ensemble d'instructions, avec des paramètres éventuels. La définition d'une macro s'écrit :

```
MACRO identificateur(liste de noms formels)
  BODY
     liste d'instructions
ENDMACRO
```

L'utilisation dans une tâche de l'identificateur de macro n'implique pas d'embaucher un agent pour exécuter les instructions. Il s'agit simplement d'une expansion textuelle; les paramètres formels de la macro, s'il y en a, peuvent donc être remplacés par n'importe quel identificateur ou nom formel de la tâche.

Exemple

On peut définir par macro l'action d'embaucher un agent, qui représente l'émission suivie de la réception d'un message. Cette macro sera couramment employée dans la suite de ce chapitre.

```
MACRO embaucher($1, $2)
  BODY
  $moi : $mien, SEND($2) =>recruteur
         +
  ** recruteur : RECEIVE($1)
ENDMACRO
```

e.5) Définition de tâche

La définition d'une tâche associe un identificateur à un ensemble d'instructions, dont les propriétés sont les suivantes :

- l'exécution de ces instructions ne peut s'effectuer qu'après l'embauche d'un agent qui en sera chargé : la désignation de la tâche est fournie en paramètre d'un message de catégorie SEND envoyé à l'agent recruteur.
- la définition de tâche délimite la portée des noms formels présents dans les instructions qui la composent.

Une définition de tâche s'écrit

```
TASK identificateur(paramètres formels "typés")
  BODY
     liste d'instructions
ENDTASK
```

Une définition de tâche peut être paramétrée. Dans ce cas, tous les paramètres doivent être "typés". Un paramètre peut désigner un objet d'un type, un type de l'univers, ou une tâche. Tous les paramètres doivent être fixés lors de l'embauche d'un agent chargé de la tâche : le paramètre du SEND est alors l'identificateur de la tâche, suivi de la liste des paramètres effectifs.

Exemple

La tâche ci-dessous compte le nombre de fois où elle est exécutée, et renvoie ce nombre en résultat.

```
TASK compteur
  BODY
  INITIAL
     embaucher($c, cell(int))        embauche d'une cellule à valeurs entières
        +
     $moi : $mien, PUT(0)=> $c       la cellule est initialisée à 0.
        +
     ** $c : DONE
  ENDINITIAL
```

```
** $1 : $2, COMPUTE ( )                    réception d'un message
      +                                     COMPUTE sans paramètre
  embaucher($plus, int.+)                   embauche d'un agent pour faire
      +                                     l'addition
  $moi : $mien, COMPUTE ($c, 1) =>$plus     demande calcul de c+1
      +
  ** $plus : RESULT($cplus1)
      +
  $moi : $mien, PUT($cplus1) =>$c           modifie contenu de c
      +
  ** $c : DONE
      +
  FIRE($plus)                               renvoie l'agent addition
$1<= $moi : RESULT($cplus1)                 répond le nouveau contenu du compteu
ENDTASK
```

f) Exécution d'une tâche

f.1) Reconnaissance d'un message reçu

Lorsqu'un agent reçoit un message, il le compare au(x) message(s) acceptable(s), compte tenu du point où il en est dans l'exécution de tâche. La reconnaissance commence par le test du mot-clé.

- Si le mot-clé ne correspond à aucun message acceptable et n'est pas ERROR, l'agent renvoie à l'expéditeur la réponse $moi : ERROR('message inacceptable') et l'état d'avancement de la tâche reste inchangé : le message n'est pas reconnu.
- Si le mot-clé ne correspond à aucun message acceptable et est ERROR, le message reçu remplace un message de catégorie RESULT, EVALUATED ou DONE attendu. L'expéditeur du message ERROR sert à identifier le message ainsi remplacé, au cas où plusieurs messages étaient attendus, et le message est reconnu. Une condition d'erreur est positionnée (voir plus bas).
- Si le mot-clé correspond à celui d'au moins un message acceptable, les 1° et 2° composants du message servent à identifier le squelette auquel il correspond, quand plusieurs messages sont attendus.

Une fois le squelette de message déterminé, les noms formels du squelette sont évalués (voir III.2.2).

Dans le cas d'un message de catégorie EVAL ou COMPUTE, les paramètres du message reçu peuvent être de nombre et de types variables, et une vérification est faite. Si le nombre de paramètres est différent de celui du squelette,

une condition d'erreur est positionnéee. Sinon, chaque paramètre est individuellement examiné, en vue de son acceptation immédiate, d'une action de liaison, ou de son rejet (voir III.4.b.2).

1. Si le paramètre est un objet de l'univers ou un magasinier, cet examen peut avoir lieu directement.

2. Si le paramètre est une expression, un ouvrier est embauché pour l'évaluer. Si le résultat de l'évaluation est un objet ou un magasinier, se reporter au point 1. Si la réponse est un message ERROR une condition d'erreur est positionnée. Si tous les paramètres ont été reconnus de bon type, les noms formels du squelette sont évalués et le message est reconnu; il n'y a pas de condition d'erreur. Sinon, le message est quand même reconnu, mais une condition d'erreur est positionnée, et l'évaluation des noms formels du squelette de même rang que les paramètres effectifs erronés n'est pas faite.

La reconnaissance du message reçu fait avancer l'état d'accomplissement de la tâche. Si un seul message était attendu, on passe à l'instruction suivante. Si plusieurs messages étaient attendus, celui qui a été reconnu est marqué comme reçu, et retiré du lot des messages attendus.

f.2) Traitement des erreurs

Chaque définition de tâche déclare implicitement un indicateur d'erreur local pour la tâche, référençable par $error. Cet indicateur est de type booléen. Il est automatiquement positionné.

- à faux au début de chaque exécution de la tâche
- à vrai lorsque
  - un paramètre de message reçu est erroné
  - un message de catégorie ERROR a été reçu à la place d'un autre message d'acquittement attendu.

L'indicateur d'erreur peut être explicitement testé dans le corps de la tâche, dans le but de provoquer un arrêt anticipé de la tâche en cas d'erreur. Si tel n'est pas le cas, une tâche pour laquelle seuls les messages attendus sont indiqués est normalement exécutée jusqu'à ce qu'une instruction d'acquittement

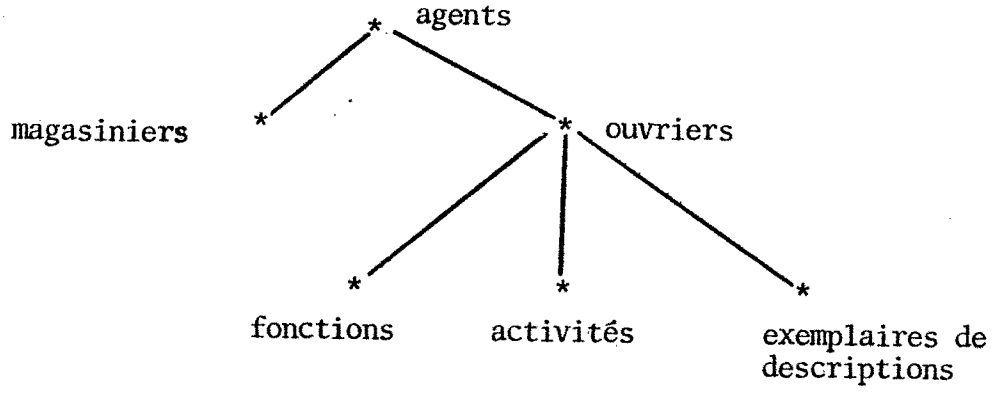
soit rencontrée. Avant qu'un message de catégorie RESULT, DONE ou EVALUATED soit adressé à l'agent expéditeur du message qui a démarré la tâche, $error est testé. Si sa valeur est faux, le message prévu est envoyé. Dans le cas contraire, le message prévu est remplacé par un message de catégorie ERROR, expliquant les raisons de l'échec de la tâche.

Ainsi, toute erreur détectée dans l'exécution d'une tâche est toujours répercutée vers celui qui a commandé l'exécution de cette tâche. De proche en proche, la connaissance de cette erreur est transmise en remontant la hiérarchie des agents d'un modèle, jusqu'à l'agent environnement.

III.4 Les catégories d'agents et de tâches

Dans la présentation du modèle, nous avons distingué magasiniers et ouvriers, les premiers gérant les tiroirs, les seconds exécutant les opérations. En fait, il existe plusieurs sortes d'ouvriers, car nous distinguons plusieurs sortes d'opérations, qui se traduisent en des tâches ayant des échanges de messages bien différenciés.

Dans la suite de ce paragraphe, nous décrirons en fait les différentes catégories de tâches; un agent étant toujours embauché pour une tâche bien déterminée, les catégories d'agents seront parfois confondues avec les catégories de tâches. En ce qui concerne les agents, la partition est donnée par le schéma ci-dessous :



a) Gestion de tiroirs et magasiniers

La tâche de gestion de tiroir consiste à répondre à un nombre fixé de messages ayant pour but de tester ou de modifier l'état du tiroir. Cet état est constitué de 4 composants :

1 - Le type du tiroir
Un tiroir ne peut contenir que des objets d'un type déterminé. Le type (des objets) du tiroir est fixé par l'agent recruteur quand il procède à l'embauche d'un magasinier pour la gestion d'un tiroir demandé. Ce type n'est pas modifiable. Un magasinier renvoie le type de son tiroir en réponse à un message TYPE.

2 - Le contenu du tiroir
Ce contenu est vide initialement. Un magasinier répond le contenu de son tiroir, s'il n'est pas vide, quand il reçoit le message GET. Un magasinier modifie le contenu de son tiroir, sous certaines conditions d'autorisation, quand il reçoit le message PUT. Un tiroir non vide ne peut plus être vidé.

3 - Une capabilité
La capabilité est un mécanisme de protection, limitant le droit de modifier le contenu du tiroir aux seuls agents dont le contexte est nommé dans la capabilité. C'est un ensemble de noms d'agents, qui peut contenir l'environnement.
La capabilité est déterminée par l'agent recruteur quand il procède à l'embauche d'un magasinier. Par défaut, la capabilité contient :

- le seul contexte de l'agent qui a demandé l'embauche, si cet agent est une fonction ou une activité.
- le nom de l'agent qui a demandé l'embauche et l'environnement, si cet agent est une description. Une description est reconnaissable au fait que son contexte et son nom sont identiques (voir paragraphe III.4.d).

La capabilité peut être modifiée uniquement par un agent-description.
A la réception du message PERMIT, le magasinier rajoute le nom de l'agent-description dans la capabilité. A la réception du message FORBID, le magasinier ôte le nom de l'agent-description de la capabilité.

4 - Un verrou

Le verrou est un indicateur booléen qui peut être dynamiquement positionné à vrai ou à faux par n'importe quel agent. Il est initialement à faux. S'il est à vrai, toute modification du contenu du tiroir est interdite. Verrouillage et déverrouillage sont effectués à la réception des messages SET et RESET respectivement.

La tâche de gestion de tiroir est définie par le texte ci-dessous, dans lequel les actions et tests effectués sur l'état interne sont décrits de manière informelle.

```
TASK tiroir ($type : TYPE)
  BODY
  DECIDE
  (** $1 : EMPTY                                   "on demande si le tiroir est
       +                                            vide"
     SELECT $contenu
     (nonvide : $1<= $moi  : RESULT(faux))
     (vide : $1<= $moi : RESULT(vrai))
     ENDSELECT)
  (** $1 : TYPE                                    "on demande le type du tiroir"
       +
     $1<= $moi : RESULT($type))
  (** $1 : GET                                     "on veut lire le contenu du
       +                                            tiroir"
     SELECT $contenu
     (nonvide : $1<= $moi : RESULT($contenu))
     (vide : $1<= $moi : ERROR('tiroir vide'))
     ENDSELECT)
  (** $1 : $context, PUT($objet)                   "on veut modifier le contenu
       +                                            du tiroir"
     SELECT $verrou
     (vrai : $1<= $moi : ERROR('tiroir verrouillé'))
     (faux : SELECT $context
       (∉ $capabilité : $1<= $moi : ERROR('écriture interdite'))
       (∈ $capabilité : SELECT $objet
       (∉ $type :
         $1<= $moi : ERROR('erreur de type'))
       (∈$type :                                   "$objet devient le nouveau
         $1<= $moi : DONE)                          contenu du tiroir"
         ENDSELECT)
       ENDSELECT)
     ENDSELECT)
```

```
  (** $1 : SET                          "demande de verrouillage"
      +
     "$verrou est mis à vrai"
     $1<= $moi : DONE)
  (** $1 : RESET                        "demande de déverrouillage"
      +
     "$verrou est mis à faux"
     $1<= $moi : DONE)
  (** $1 : $context, PERMIT             "demande d'ajout dans la capabilité"
      +
     SELECT $context
     ($1 :                              "$1 est rajouté dans la capabilité"
        $1<= $moi : DONE)
     (OTHER : $1<= $moi : ERROR('modification de la capabilité interdite'))
      ENDSELECT)
  (** $1: $context, FORBID             "demande d'effacement de la capabilité"
      +
     SELECT  $context
     ($1 :                              "$1 est supprimé de la capabilité"
        $1<= $moi : DONE)
     (OTHER : $1<= $moi : ERROR('modification de la capabilité interdite'))
     ENDSELECT)
  ENDDECIDE
ENDTASK
```

b) Tâche d'évaluation d'une fonction

Une tâche d'évaluation de fonction est exécutée quand l'ouvrier associé reçoit un message COMPUTE avec ses paramètres effectifs. Elle se termine avec l'émission d'un message RESULT ou d'un message ERROR vers l'agent qui a commandé l'exécution. Le contexte du message COMPUTE reçu doit être égal au contexte local à la fonction.

Une tâche d'évaluation de fonction se décompose en 7 phases :

1 - initialisation
2 - liaison des paramètres
3 - vérification des assertions sur les paramètres
4 - embauche des magasiniers locaux
5 - évaluation du corps de fonction
6 - évaluation du résultat
7 - renvoi des agents locaux intérimaires et réponse.

Les phases 2 à 7 se succèdent dans l'ordre de leur numérotation. Seules les phases 2, 6, 7 sont obligatoirement présentes. La phase d'initialisation, si elle existe, est effectuée avant toute exécution de la tâche.

Si,au cours d'une phase i d'exécution de la tâche, l'indicateur $error prend la valeur vrai, les phases i+1 à 6 sont sautées. Seule est alors exécutée la phase 7, qui a pour effet de renvoyer les agents intérimaires embauchés pour l'exécution interrompue, et d'émettre un compte-rendu à destination de l'ouvrier qui avait expédié le message COMPUTE.

b.1) Initialisation

La phase d'initialisation contient l'embauche des agents suivants :

- un magasinier pour chaque paramètre formel de la fonction, déclaré porteuse non modifiable.
- dans le cas d'une fonction statique uniquement :

    . un magasinier pour chaque porteuse locale à la fonction
    . un ouvrier pour chaque invocation d'opération statique dans le corps de la fonction.

b.2) Evaluation et liaison des paramètres

A la réception du message COMPUTE, les noms formels et les paramètres de ce message sont évalués. Si l'évaluation est sans erreur, chaque paramètre $pi du message COMPUTE est examiné en regard du type spécifié pour ce paramètre.

Si le paramètre est un objet de l'univers qui appartient au type spécifié, le paramètre est accepté. Si le paramètre est un objet de l'univers qui n'appartient pas au type spécifié :

- si ce type est un type de valeur, le paramètre est refusé, et $error est mis à vrai
- si ce type est un type de porteuse et qu'une action de liaison a été prévue, cette action est exécutée; sinon $error est mis à vrai.

Si le paramètre $pi est un magasinier, il y a selon les cas :

- acceptation directe si le type du paramètre formel est une porteuse de même type de valeur que le type du contenu de $pi, et que l'effet recherché est un passage par référence (porteuse W en PSCL)
- dérepérage, si le type du paramètre formel est égal au type du contenu de $pi :

```
$moi : $mien, GET =>$pi
          +
** $pi : RESULT($vpi)
```

- exécution d'une action de liaison si le type du paramètre formel est une porteuse de même type de valeur que le type du contenu de $pi, et qu'un passage par valeur est voulu (porteuse non modifiable en PSCL)
- exécution d'autres actions de liaison éventuelles, sinon $error est mis à vrai.

Nous donnons ci-dessous, à titre d'exemple, les actions de liaisons qui ont lieu pour PSCL, quand le paramètre formel est une porteuse non modifiable. Si $pi désigne le paramètre effectif du message COMPUTE reçu, et $i le magasinier embauché à l'initialisation pour ce paramètre, deux possibilités sont prévues.

Si $pi est une valeur, dont le type est identique au type du contenu de $i, la macro "lier_valeur" est exécutée : on met $pi dans $i, puis on verrouille $i. Ainsi, toute tentative pour modifier le contenu de $i au cours de l'exécution de la fonction sera détectée.

```
MACRO lier_valeur($pi, $i)
  BODY
  $moi : $mien, PUT($pi)=> $i
         +          "on met $pi dans $i"
  ** $i : DONE
         +
  $moi : SET=> $i
         +          "on verrouille $i"
  ** $i : DONE
ENDMACRO
```

Si $pi est un magasinier de même type que le paramètre formel, la macro "copier_contenu" est exécutée. Elle a pour effet de recopier le contenu de $pi dans $i, si $pi n'est pas vide, puis de verrouiller $i.

```
MACRO copier_contenu($pi, $i, $ri, $ci)
  BODY
  $moi  : EMPTY =>$pi
        +
  ** $pi : RESULT($ri)
        +
  SELECT $ri
    (vrai : ) "si $pi est vide, on ne fait rien"
    (faux : $moi : GET =>$pi
              +
            ** $pi : RESULT($ci)
               + "sinon on copie le contenu de $pi dans $i"
            $moi : $mien, PUT($ci) =>$i
               +
            ** $i : DONE)
  ENDSELECT
        +  "puis on verrouille $i"
  $moi : SET =>$i
        +
  ** $i : DONE
ENDMACRO
```

b.3) Vérification d'assertions sur les paramètres

Si des assertions sur les paramètres ont été écrites lors de la définition de la fonction, cette phase n'est pas vide. Un ouvrier est embauché pour chaque assertion. Un message COMPUTE est envoyé à chacun, tous ces messages sont émis en parallèle. Si l'une des assertions renvoie comme résultat faux, $error est mis à vrai.

b.4) Embauche des magasiniers locaux

Dans le cas d'une fonction dynamique uniquement, si le corps de la fonction n'est pas vide et contient des déclarations de porteuses locales, un magasinier est embauché pour chacune d'elles.

b.5) Evaluation du corps de fonction

Cette phase n'est pas vide si le corps de la fonction contient au moins une invocation d'activité. Elle contient la séquence suivante :

1 - l'embauche d'un ouvrier pour chaque invocation d'activité dynamique dans le corps de la fonction.
2 - l'envoi, en parallèle, d'un message EVAL à chaque ouvrier chargé d'une invocation d'activité, avec ses paramètres effectifs (cet envoi peut être conditionné par un test SELECT).
3 - la réception de tous les messages EVALUATED correspondants. Si $error est positionné par la réception d'un message ERROR, sauter au point 6.
4 - l'envoi, en parallèle, d'un message START à tous les ouvriers activés en 2.
5 - la réception de tous les messages DONE correspondants.
6 - le renvoi des agents embauchés en 1.
7 - si le corps de la fonction contient des assertions, embauche d'un ouvrier pour chaque assertion, et lancement en parallèle de toutes leurs évaluations, selon un schéma identique à la phase b.3).

b.6) Evaluation du résultat

Dans le cas général, le résultat nécessite l'évaluation d'une expression. Cette phase contient donc l'embauche d'un ouvrier pour cette expression, l'envoi à cet ouvrier d'un message COMPUTE avec ses paramètres effectifs, et la réception de son message RESULT. Le paramètre du RESULT est le résultat de la fonction complète.

Cette évaluation du résultat est simplifiée quand l'expression se réduit à :
- une constante
- un contenu de porteuse locale.

b.7) Renvoi des agents locaux intérimaires et réponse

Tous les agents embauchés pendant les phases 3, 4, 5 et 6 sont renvoyés. Les magasiniers qui avaient été verrouillés à la phase 2 par l'exécution de la macro "lier_valeur" ou "copier_contenu" sont déverrouillés par exécution de la macro "déverrouiller".

```
MACRO déverrouiller($i)
  BODY
  $moi : RESET =>$i
       +
  ** $i : DONE
ENDMACRO
```

Si $error a la valeur vrai, un message ERROR est envoyé à l'agent expéditeur du message COMPUTE qui a déclenché l'exécution. Sinon, le message RESULT, avec en paramètre le résultat calculé en phase 6, lui est envoyé.

b.8) Exemple

La fonction max ci-dessous est écrite en CONLAN PRIMITIF.

```
FUNCTION max(x, y : int) : int
   RETURN IF x > y THEN x ELSE y ENDIF
ENDmax
```

Cette fonction est dynamique. Elle a pour paramètres des valeurs, et aucune assertion n'est écrite. La tâche d'évaluation qui lui correspond contient seulement les phases 2, 6, 7. Anticipant un peu sur le paragraphe IV.1, cette tâche s'écrit :

```
TASK max
  BODY
  "Evaluation des paramètres"
  ** $appelant : $mien, COMPUTE($p1 : int, $p2 : int)
          +
  "Evaluation du résultat"
  embaucher($eif, if) "embauche d'un ouvrier pour l'expression IF"
          +
  $moi : $mien, COMPUTE(int.> ($p1, $p2), $p1, $p2) =>$eif
          +
  ** $eif : RESULT($plusgrand)
          +
  "Renvoi des agents locaux et réponse"
  FIRE($eif)
  $appelant<= $moi : RESULT($plusgrand)
ENDTASK
```

c) Tâche d'évaluation d'une activité

Contrairement à une tâche d'évaluation de fonction, qui en l'absence d'erreur est entièrement exécutée après réception du message COMPUTE, une tâche d'évaluation d'activité nécessite deux échanges de messages avec l'ouvrier qui en a demandé l'exécution. Le premier échange est réalisé par le couple de messages EVAL-EVALUATED, et concerne uniquement les actions de passage et de vérification des paramètres effectifs. Le deuxième échange est réalisé par le

couple de messages START-DONE, et correspond au déclenchement-acquittement de l'évaluation du corps de l'activité. Le contexte des messages EVAL et START doit être identique à celui de l'activité réceptrice.

Cette décomposition en deux étapes bien différenciées est le mécanisme qui rend possible la modélisation du caractère parallèle et non procédural des invocations d'activités : pendant la première étape, on extrait des porteuses leurs valeurs avant évaluation des activités; pendant la deuxième étape, les porteuses sont éventuellement modifiées. Le résultat est indépendant de l'ordre lexical des invocations.

Une tâche d'évaluation d'activité se décompose en 6 phases :

1 - initialisation
2 - liaison des paramètres
3 - vérification des assertions sur les paramètres
4 - embauche des magasiniers locaux
5 - évaluation du corps d'activité
6 - renvoi des agents locaux intérimaires et réponse.

La phase d'initialisation est identique à celle d'une fonction. Les phases 2 à 5 sont constituées des mêmes actions que les phases 2 à 5 correspondantes des fonctions; seul leur démarrage ou leur terminaison diffère. Enfin, la distinction entre activité statique et dynamique se reflète de manière identique au cours des différentes phases.

c.1) Etape EVAL-EVALUATED

Cette étape regroupe les phases 2 et 3. A la réception du message EVAL, les noms formels et les paramètres de ce message sont évalués. Si l'évaluation est sans erreur, on procède à la liaison des porteuses non modifiables. La phase de vérification des assertions a ensuite lieu, si elle est non vide. Lorsque le résultat de toutes les assertions a été reçu, les ouvriers, embauchés pour les évaluer sont renvoyés.

Si $error a la valeur vrai, les magasiniers qui avaient été verrouillés lors de la liaison des paramètres, en phase 2, sont déverrouillés; un message ERROR est émis vers l'agent qui a commandé l'exécution, et la tâche n'est pas poursuivie. Sinon, un message EVALUATED est émis, et l'activité se met en attente du message START.

c.2) Etape START-DONE

Cette étape regroupe les phases 4, 5, 6. A la réception du message START, qui doit provenir du même expéditeur que le message EVAL de l'étape précédente, les phases 4 et 5 sont exécutées séquentiellement, comme pour une évaluation de fonction. Puis la phase 6 est exécutée.

La phase 6 comprend le renvoi de tous les agents embauchés pendant les phases 4 et 5. En parallèle, tous les magasiniers qui avaient été verrouillés en phase 2 sont déverrouillés. Puis $error est testé. S'il a la valeur vrai, un message ERROR est envoyé à l'agent qui a commandé l'exécution, sinon un message DONE lui est envoyé.

c.3) Schéma général d'une tâche d'évaluation d'activité

Une tâche d'évaluation d'activité obéit au schéma général suivant :

```
TASK eval_activité
  BODY
  INITIAL
    embaucher($1, t1)                          "embauche des magasiniers et
    embaucher($2, t2)                           ouvriers permanents, s'il y
        :                                       a lieu"
    embaucher($n, tn)
  ENDINITIAL
  "étape d'évaluation des paramètres"
  ** $appelant : $mien, EVAL($p1, $p2, ... $pi)
       +
     (lier_valeur($pi1, $i1)                   "liaison des porteuses non
         :                                      modifiables"
     lier_valeur($pik, $ik)
     copier_contenu($pij, $ij)
         :
     copier_contenu($pil, $il))
       +
     (embaucher($f1, nf1)                      "embauche des ouvriers pour
         :                                      évaluer les assertions"
     (embaucher($fm, nfm))
       +
     ($moi : $mien, COMPUTE(...) =>$f1         "demandes d'évaluation des
         :                                      assertions"
     ($moi : $mien, COMPUTE(...) =>$fm)
       +
```

```
   (** $f1 : RESULT(vrai)                          "si une assertion ne renvoie
        :                                          pas vrai, $error sera positionné"
    ** $fm : RESULT(vrai))
         +
        FIRE($f1, ... $fm)                         "renvoi  des ouvriers embauchés
         +                                         pour évaluer les assertions"
   SELECT $error
   (vrai : $appelant<= $moi : ERROR('paramètre...erroné')
           déverrouiller($il)
               :
           déverrouiller($il)
           STOP)
   (faux : $appelant<= $moi : EVALUATED)
ENDSELECT
        +
"étape d'évaluation du corps de l'activité"
** $appelant : $mien, START
        +
(embaucher($m1, tm1)                               "embauche des magasiniers locaux
      :                                            s'il y a lieu"
embaucher($ml, tml))
        +
(embaucher($a1, na1)                               "embauche des activités dynami-
      :                                            ques locales, s'il y a lieu"
embaucher($ap, nap))
        +
($moi : $mien, EVAL(...) =>$a1                     "démarrage en parallèle de l'éta-
     :                                             pe 1 des activités internes"
$moi : $mien, EVAL(...) =>$aq)
        +
(** $a1 : EVALUATED                                "réception des compte-rendus"
        :
** $aq : EVALUATED)
        +
SELECT $error
(faux :                                            "s'il n'y a pas eu d'erreur,
 ($moi : $mien, START =>$a1                        démarrage en parallèle de l'éta-
        :                                          pe 2 des activités internes et
  $moi : $mien, START =>$aq)                       réception des compte-rendus"
        +
 (** $a1 : DONE
        :
 ** $aq : DONE))
(vrai : )
ENDSELECT
        +
FIRE(a1, a2, ... ap)                               "renvoi des activités dynamiques
        +                                          locales"
```

```
  SELECT $error
  (faux : ... "évaluation des assertions du corps de l'activité
              idem à ci-dessus")
  (vrai :     "on ne fait rien")
 ENDSELECT
     +
  FIRE(m1, ... ml)                    "renvoi des magasiniers locaux"
     +
  (déverrouiller($i1)
       :
       :
  déverrouillez($il))                 "déverrouillage des porteuses non
     +                                 modifiables"
  $appelant<= $moi : DONE
ENDTASK
```

d) Tâche d'évaluation d'une description

Une description représente un modèle de système digital dont la structure est fixe. Une tâche d'évaluation de description a donc toujours une phase d'initialisation non vide, qui consiste à acquérir tous les objets locaux et d'interface. Une description peut être décomposée en un nombre arbitraire d'exemplaires de descriptions plus simples : cette structure arborescente se retrouve dans le fait qu'en phase d'initialisation un ouvrier est embauché pour chaque exemplaire. Enfin, la propriété essentielle d'une tâche d'évaluation de description est de transmettre les messages d'activation et de compte-rendu le long de l'arborescence des ouvriers, en fixant comme contexte de ces messages son propre nom; ces messages sont, comme pour une tâche d'évaluation d'activité, des couples d'échanges de messages EVAL-EVALUATED et START-DONE.

Une tâche d'évaluation de description se décompose en 3 étapes principales :

1 - initialisation
2 - étape EVAL-EVALUATED
3 - étape START-DONE

d.1) Initialisation

L'étape d'initialisation contient quatre phases successives (2 et 4 peuvent être vides).

1 - La réception,de la part de l'agent qui commande l'exemplaire de description présent, de tous les magasiniers correspondant aux porteuses d'interface. Pour chaque élément spécifié, dans la déclaration d'interface, comme modifiable à l'intérieur de la description (en CONLAN : INOUT et OUT), un message PERMIT est envoyé.

2 - Si la description contient des exemplaires de description plus simples, l'embauche d'un ouvrier pour chaque exemplaire.

3 - L'embauche d'un ouvrier pour chaque invocation d'opération statique dans le corps de la description.
L'embauche d'un magasinier pour chaque porteuse déclarée dans le corps de la description, y compris les interfaces des exemplaires de description de niveau immédiatement inférieur, s'il y en a. A ce stade, on tient compte des synonymies éventuelles entre objets déclarés, notamment entre objets d'interface d'exemplaires de description englobés.
Pour chaque élément d'interface d'un exemplaire de description englobé, qui ne peut pas être modifié à l'extérieur de cet exemplaire (en CONLAN : OUT), un message FORBID est envoyé au magasinier correspondant.

4 - Si 2- est non vide, l'envoi, à chaque ouvrier embauché pour un exemplaire de description englobé, de tous les magasiniers embauchés pour leurs porteuses d'interface.

Remarques

. Le fait que les magasiniers correspondant aux porteuses d'interface soient embauchés par l'environnement d'un exemplaire de description correspond au lieu où sont déclarées les synonymies éventuelles, et donc où est connu le nombre de porteuses distinctes.

. La capabilité d'un tiroir contient a priori le nom de la description qui a procédé à l'embauche. Les messages PERMIT et FORBID envoyés aux magasiniers ont pour objet de :

- rajouter à cette capabilité le nom de tous les exemplaires de description englobés qui ont le droit de modifier le contenu du tiroir
- supprimer de cette capabilité le nom de l'agent qui a fait l'embauche, s'il n'a pas le droit de modifier le contenu du tiroir.

Ces messages ne sont permis que dans le bloc d'initialisation d'une description : les droits sont établis statiquement.

La phase d'initialisation d'une description s'écrit selon le schéma général suivant :

```
INITIAL
  ** $e : RECEIVE($i1)                    "réception des éléments d'interface dans
          +                                l'ordre de leur déclaration"
  ** $e : RECEIVE($i2)
          +
     :
     :
  ** $e : RECEIVE($in)
          +
  (($moi : $moi, PERMIT =>$ik             "modification des capabilités des éléments
          +                                d'interface OUT et INOUT"
    ** $ik : DONE)
         :
         :
  ($moi : $moi, PERMIT =>$il
          +
    ** $il : DONE))
          +
  (embaucher($exd1, d1)                   "embauche des exemplaires de description
   embaucher($exd2, d2)                    de niveau inférieur"
         :
         :
   embaucher($exdm, dm))
          +
  (embaucher($a1, op1)                    "embauche des ouvriers pour les opérations
         :                                statiques"
         :
  embaucher($aj, opj)
  embaucher($m1, t1)                      "embauche des magasiniers locaux"
         :
         :
  embaucher($mx, tx))
          +
  (($moi : $moi, FORBID =>$mv             "modification des capabilités des éléments
          +                                d'interface OUT des descriptions englobées"
   ** $mv : DONE
         :
         :
  ($moi : $moi, FORBID => $mw
          +
  ** $mw : DONE))
          +
  ($moi : RECEIVE($md11) =>$exd1          "expédition de leurs éléments d'interface
          +                                à tous les exemplaires de description en-
  $moi : RECEIVE($md12) =>$exd1            globés"
          +
          :
          :
  $moi : RECEIVE($md1a1=>$exd1)
          :
          :
```

```
(Smoi : RECEIVE(Smdm1) =>Sexdm
          +
          :
 Smoi : RECEIVE(Smdmam) =>Sexdm)
ENDINITIAL
```

d.2) Etape EVAL-EVALUATED

La tâche d'évaluation d'une description est démarrée par la réception d'un message EVAL sans paramètre. Un ouvrier est embauché pour chaque invocation d'activité dynamique dans le corps de la description. Puis des messages EVAL, avec pour contexte Smoi, sont envoyés en parallèle, à raison de :

- un message, avec les paramètres effectifs, à chaque activité invoquée
- un message sans paramètre à chaque exemplaire de description englobé.

La tâche se met en attente de tous les EVALUATED correspondants. Quand tous les acquittements ont été reçus, Serror est testé.

Si Serror a la valeur vrai, les ouvriers qui avaient été embauchés au cours de cette étape sont renvoyés, un message ERROR est répondu à l'agent qui avait envoyé l'ordre EVAL, et la tâche n'est pas poursuivie. Sinon, un message EVALUATED est émis, et la description se met en attente du message START.

d.3) Etape START-DONE

A la réception du message START, un message START est envoyé en parallèle, avec Smoi comme contexte, à tous les agents associés à une activité invoquée ou à un exemplaire de description englobé. La tâche se met en attente de tous les DONE correspondants. Quand tous les acquittements ont été reçus, les ouvriers embauchés à l'étape précédente pour évaluer les activités dynamiques sont envoyés. Si Serror a la valeur vrai, un message ERROR est répondu à l'agent qui avait envoyé l'ordre START, sinon un message DONE lui est répondu.

Remarque

Dans le cas de l'évaluation de la description englobante, les communications EVAL-EVALUATED, START-DONE, et RECEIVE ont lieu avec l'agent "environnement".

IV APPLICATION A CONLAN

Pour définir la sémantique d'un langage en termes du modèle d'évaluation par équipe d'agents, il faut :

- choisir les objets du modèle qui correspondent aux types de valeurs primitifs du langage;
- définir le comportement des agents qui correspondent aux porteuses et aux opérateurs pré-définis dans le langage;
- spécifier comment les instructions et les segments du langage sont traduits dans le modèle.

Nous allons à présent indiquer les grandes lignes d'une telle définition sémantique pour CONLAN.

IV.1 Les objets et agents primitifs

a) L'univers des objets

L'univers des objets du modèle est constitué :

- des entiers relatifs, notés en base 10
- des booléens, notés vrai, faux
- des chaînes de caractères, notées entre apostrophes
- des noms universels de types
- des suites finies d'éléments de l'univers, parenthésées par "(." et ".)".

Toutes les dénotations d'objets du modèle sont soulignées, afin de les distinguer des dénotations des objets de CONLAN PRIMITIF qu'ils représentent. Ainsi, -10 dénote l'objet du modèle qui correspond aux dénotations -10, -AH, et -1101B en CONLAN PRIMITIF.

b) Les cellules et les opérations primitives sur les cellules

Les cellules, éléments du type générique cellà(t : anyà) de CONLAN PRIMITIF, sont représentées dans le modèle par les magasiniers, dont le comportement a été exposé au paragraphe III.4.a).

Aux quatre opérations définies sur le type cellà correspondent dans le modèle quatre tâches pré-définies, évaluables par des agents ouvriers.

b.1) Tâche associée à la fonction getà

```
TASK cellà.getà
  BODY
   "Evaluation du paramètre d'appel"
  ** $appelant : $mien, COMPUTE($p1 : tiroir)
          +
     "Demande de lecture au magasinier"
     $moi : GET =>$p1
          +
     "Réception du contenu et réponse"
     ** $p1 : RESULT($contenu)
          +
  $appelant<= $moi : RESULT($contenu)
ENDTASK
```

b.2) Tâche associée à la fonction cell_typeà

```
TASK cellà.cell_typeà
  BODY
   "Evaluation du paramètre d'appel"
  ** $appelant : $mien, COMPUTE($p1 : tiroir)
          +
     " Demande au magasinier le type du tiroir"
     $moi :  TYPE =>$p1
          +
     "Réception et transmission de la réponse"
     ** $p1 : RESULT($t)
          +
  $appelant<= $moi : RESULT($t)
ENDTASK
```

b.3) Tâche associée à la fonction emptyà

```
TASK cellà.emptyà
  BODY
   "Evaluation du paramètre d'appel"
  ** $appelant : $mien, COMPUTE($p1 : tiroir)
          +
```

```
    "Demande au magasinier si le tiroir est vide"
    $moi : EMPTY =>$p1
       +
    "Réception et transmission de la réponse"
    ** $p1 : RESULT($v)
       +
  $appelant<= $moi : RESULT($v)
ENDTASK
```

b.4) Tâche associée à l'activité putà

```
TASK cellà.putà
  BODY
   "Evaluation des paramètres d'appel"
  ** $appelant : $mien, EVAL($p1 : tiroir, $p2)
        +
   "Réponse à la suite de l'évaluation des paramètres"
   SELECT $error
    (vrai : $appelant<= $moi : ERROR('paramètre erronné')
            STOP)
    (faux : $appelant<= $moi : EVALUATED)
 ENDSELECT
       +
  "Etape d'exécution : on envoie PUT au magasinier"
** $appelant : $mien, START
       +
   $moi : $mien, PUT($p2) =>$p1
       +
  "Réception et transmission du compte-rendu"
  ** $p1 : DONE
       +
 $appelant<= $moi : DONE
ENDTASK
```

c) Les fonctions sur les types primitifs autres que cellà

A chacun des opérateurs et des fonctions définis sur les types int, bool, string, tupleà et la classe anyà correspond dans le modèle une tâche ayant pour nom le symbole ou l'identificateur de la fonction préfixé par le type auquel elle se rapporte. Ces tâches sont de la catégorie "évaluation de fonction", exposée au paragraphe III.4.b, et sont caractérisées vis-à-vis de l'agent qui en demande l'évaluation par un seul échange de message COMPUTE + RESULT, ou en cas d'erreur COMPUTE + ERROR.

Exemples

L'invocation de
'ABC' = 'ABD'
dans un texte écrit en CONLAN PRIMITIF donne lieu à l'échange de messages suivant en cours d'évaluation :

a : ca, COMPUTE('ABC', 'ABD') → f_string.=
a ← f_string.= : RESULT(faux)

où "a" est l'agent correspondant au segment où est invoquée la comparaison, "ca" est le contexte de a, et "f_string.=" est le nom d'un ouvrier associé à la tâche "string.=" d'évaluation de l'égalité sur les chaînes de caractères.

Utilisant une notation analogue pour identifier les agents, l'invocation de

selectà((. 1, 3, 5, 7 .), 6)

donne lieu, en cours d'évaluation, à l'échange :

a : ca, COMPUTE((. 1, 3, 5, 7), 6) → f_tupleà.selectà
a ← f_tupleà.selectà : ERROR('2ème paramètre supérieur à la longueur du 1er')

d) Quantificateurs et sélecteur

Les quantificateurs FORALLà, FORSOMEà et FORONEà sont chacun traduits dans le modèle par une tâche primitive de la catégorie évaluation de fonction, et paramétrée par le type sur lequel porte la propriété à vérifier.

A l'expression de PSCL :

FORALLà x : untype IS p(x) ENDFOR

correspond donc la tâche primitive :

```
TASK forallà($t : TYPE)
  BODY
  ** $appelant : $mien, COMPUTE($p1, $p2 : bool)
   :  "-vérification que $p2 est une expression dont le résultat est booléen
   :   -vérification que $p2 prend toujours la valeur vrai quand $p1 parcourt
   :    $t; si oui $result a la valeur vrai, sinon faux"
  $appelant<= $moi :RESULT($result)
ENDTASK
```

L'invocation de FORALLà donne lieu à :

- l'embauche d'un ouvrier "oforallà" pour la tâche forallà(untype)
- l'échange de messages

  a : ca, COMPUTE(x, p(x)) → oforallà
  +
  a ← oforallà(RESULT, vrai)

  si untype est vide, ou si p(x) prend la valeur vrai pour tous les éléments de untype. Dans le cas contraire, et en l'absence d'erreur, l'échange est :

  a : ca, COMPUTE(x, p(x)) → oforallà
  +
  a ← oforallà(RESULT, faux).

La définition des tâches foroneà et forsomeà,correspondant respectivement aux quantificateurs FORONEà et FORSOMEà, est tout à fait analogue.

Le sélecteur THEà renvoie l'élément unique de "untype" qui vérifie un prédicat, si un tel élément existe, sinon erreur; cette expression s'écrit en PSCL :

```
THEà x : untype WITH p(x) ENDTHE
```

A ce sélecteur correspond une tâche primitive de la catégorie évaluation de fonction, paramétrée par le type sur lequel porte la sélection, et définie de manière analogue à celles qui sont associées aux quantificateurs :

```
TASK theà($t : TYPE)
  :
ENDTASK
```

e) Expressions conditionnelles

Soit p une expression booléenne,a et b deux expressions. A l'expression conditionnelle de PSCL : IF p THEN a ELSE b ENDIF correspond dans le modèle d'évaluation une tâche de catégorie évaluation de fonction :

```
TASK if
  BODY
  ** $appelant : $mien, COMPUTE($p1 : bool, $p2, $p3)
        +
    SELECT $p1
      (vrai : $appelant<= $moi : RESULT($p2))
      (faux : $appelant<= $moi : RESULT($p3))
    ENDSELECT
ENDTASK
```

Une expression conditionnelle avec parties "ELIF" :

IF p1 THEN a1 ELIF p2 THEN a2 ELIF ... ELSE an ENDIF

est transformée en imbrications d'expressions IF élémentaires :

```
IF p1 THEN a1 ELSE
    IF p2 THEN a2 ELSE ... ELSE an ENDIF ... ENDIF
```

avant d'être traduite dans le modèle en suivant la règle générale des compositions de fonctions.

Soit v une expression à valeurs dans un type donné, v1, v2 ... vn n valeurs différentes de ce type, a1, a2, ... an, an+1 n+1 expressions. A l'expression conditionnelle de PSCL

```
CASE v IS
  v1 : a1;
  v2 : a2;
  :
  :
  vn : an;
ELSE an+1 ENDCASE
```

correspond dans le modèle d'évaluation une tâche de catégorie évaluation de fonction, paramétrée par le type et le nombre de valeurs sur lesquels s'opère le choix :

```
TASK case($t : TYPE, $n : pint)
  BODY
  ** appelant : $mien, COMPUTE($v : $t, $v1 : $t, $v2 : $t, ... $vn : $t,
                    $a1, $a2, ... $an, $an+1)
       +
     SELECT $v
      ($v1 : $appelant<= $moi : RESULT($a1))
      ($v2 : $appelant<= $moi : RESULT($a2))
        :
        :
      ($vn : $appelant<= $moi : RESULT($an))
      (OTHER : $appelant<= $moi : RESULT($an+1)
  ENDSELECT
ENDTASK
```

IV.2 Traduction dans le modèle d'un segment écrit en CONLAN PRIMITIF

Les objets et tâches, énumérées au paragraphe IV.1 précédent, pré-existent à toute écriture d'un segment en PSCL. La définition d'un nouveau segment se traduit par la création d'une tâche pour ce segment s'il s'agit d'une fonction, description ou activité, ou par des adjonctions/suppressions d'objets ou de relations s'il s'agit d'un langage, d'une classe ou d'un type. Dans la suite, nous considérerons toujours que toutes les expressions et les invocations d'activités écrites dans le corps d'un segment FUNCTION, ACTIVITY ou DESCRIPTION sont mises sous forme préfixée préalablement à toute traduction dans le modèle d'agents.

a) Traduction d'un segment CONLAN

Un segment CONLAN sert à définir un nouveau langage. C'est une liste de définitions de segments TYPE, DESCRIPTION, FUNCTION, ACTIVITY et CLASS. La traduction d'un tel segment consiste donc à établir l'univers des objets disponibles et des tâches pré-définies pour tous les utilisateurs du nouveau langage.

b) Traduction d'un segment CLASS

Un segment CLASS sert à établir des regroupements de types, à des fins de vérifications statiques. La traduction d'un segment CLASS dans notre modèle est donc la création d'une fonction φ_c de l'univers d'objets dans {0, 1}, la caractéristique de la classe c, définie par :

$\varphi_c(x) = 1$ si x est le nom universel d'un type qui est élément de la classe c

$\varphi_c(x) = 0$ sinon

c) Traduction d'un segment TYPE

Un segment TYPE est traduit dans le modèle par un enrichissement de l'univers :

- S'il s'agit d'un type de valeurs, l'univers est augmenté de l'ensemble des dénotations des éléments du type.
- Le nom universel du type est rajouté à l'univers.
- Pour toutes les classes existantes, la valeur de la caractéristique de chaque classe, appliquée au nom universel du nouveau type, est déterminée.
- Une tâche est créée pour chaque opération définie ou transportée dans le segment TYPE. Cette tâche est identifiée par le nom (ou le symbole) de l'opération, préfixé par le nom universel du type.

Si le segment TYPE est paramétré, les actions que nous venons d'exposer représentent la fermeture, par rapport à la relation de partage de définition, des adjonctions à l'univers d'objets et de tâches, pour toutes les valeurs possibles des paramètres du segment.

d) Traitement des attributs d'un segment DESCRIPTION, FUNCTION ou ACTIVITY

Les attributs sont des paramètres qui peuvent apparaître dans l'en-tête d'une définition de segment DESCRIPTION, FUNCTION, ACTIVITY. Lors d'une référence ultérieure au segment, la valeur des attributs doit être connue statiquement. Les attributs sont traduits dans le modèle par les paramètres des tâches :

ces paramètres doivent toujours être fournis lors de l'embauche d'un agent pour la tâche.

Au cours des échanges de messages, les attributs sont donc présents comme paramètres du paramètre du message SEND envoyé au recruteur. Par contre, une fois l'embauche de l'agent effectuée, aucun message envoyé à cet agent pour lui demander d'accomplir sa tâche ne contient de paramètre correspondant aux attributs.

e) Traduction d'un segment FUNCTION ou ACTIVITY

Un segment FUNCTION (resp. ACTIVITY) est traduit en une tâche d'évaluation de fonction (resp. d'activité) dont les caractéristiques ont été décrites au paragraphe III.4.b (resp. III.4.c).

Si l'opération est déclarée STATIC, la phase d'initialisation de la tâche associée contient l'embauche d'un magasinier pour toutes les porteuses locales, et d'un ouvrier pour toutes les invocations d'opérations statiques dans le corps du segment. Que l'opération soit ou non statique, un magasinier est embauché pour chaque paramètre déclaré d'un type porteuse dans le cas d'un segment FUNCTION, pour chaque paramètre déclaré d'un type porteuse non W dans le cas d'un segment ACTIVITY.

Puis, en fonction du texte du segment, les squelettes de messages sont produits, correspondant aux différentes phases de la tâche.

Le message COMPUTE (resp. EVAL) attendu contient autant de paramètres que l'opération a de paramètres qui ne sont pas des attributs. Pour tout paramètre de type porteuse non modifiable est produite l'action de liaison, sous forme d'un choix entre les macros "lier_valeur" et "copier_contenu".

Dans les blocs assertions, le corps du segment, et pour une fonction l'expression RETURN, toutes les invocations de fonctions sont traduites en l'émission de messages COMPUTE vers les ouvriers appropriés, suivie de la réception des messages RESULT correspondants. Dans le corps de l'opération, toutes les invocations d'activités sont traduites en la succession des deux étapes EVAL + EVALUATED et START + DONE, l'émission de tous les messages à l'intérieur de chaque étape se faisant en parallèle. Les invocations conditionnelles d'activités, par une instruction IF ou CASE, sont traduites en deux tests SELECT identiques, l'un conditionnant l'envoi des messages EVAL, l'autre conditionnant l'envoi des messages START.

Production des paramètres

Les paramètres des squelettes de messages EVAL et COMPUTE, évoqués ci-dessus, émis par la tâche, sont produits en comparant, dans le segment écrit en PSCL, le type du paramètre effectif d'appel de l'opération et le type du paramètre formel de cette opération. Les combinaisons autorisées, et leur traduction dans le modèle, sont données dans le tableau suivant :

Type du paramètre formel t	Paramètre effectif x	Paramètre du message EVAL ou COMPUTE
valeur	constante de type t	x, objet de l'univers de type t correspondant
valeur	porteuse déclarée de type cellà(t)	getà($x) où $x est le nom symbolique du magasinier embauché pour x
porteuse	porteuse déclarée de type t	$x
porteuse t = cellà(tc)	élément du type tc	x si tc est un type de valeur $x si tc est un type de porteuse
t quelconque	expression dont le résultat est de type t	expression dont les opérandes ont subi l'une des transformations précédentes

Toute autre combinaison est interdite, et provoque une erreur et un arrêt de la construction de la tâche.

f) Traduction d'un segment DESCRIPTION

Un segment DESCRIPTION est traduit en une tâche d'évaluation de description dont les caractéristiques ont été décrites au paragraphe III.4.d. Les règles d'acquisition des éléments d'interface et de réajustement des capabilités pour tenir compte du sens de transfert des informations sont celles qui ont été exposées en III.4.d.1. Si le corps de la description contient des invocations d'activités, leur traduction en squelettes de messages EVAL + EVALUATED et START + DONE, et la production des paramètres du message EVAL, obéissent aux mêmes règles que celles qui s'appliquent aux segments ACTIVITY.

Exemple

La description d1 ci-dessous, écrite en PSCL, contient un exemplaire "inner" d'une description externe d2.

```
REFLAN pscl
DESCRIPTION d2(IN a : cellà(int); OUT b : cellà(int))
  BODY
  IF emptyà(a) THEN putà(b, 0) ELSE putà(b, a+1) ENDIF
ENDd2

REFLAN pscl
DESCRIPTION d1(IN start : cellà(bool))
  BODY
  EXTERNAL DESCRIPTION d2 END
  DECLARE x, y, z : cellà(int) ENDDECLARE
  USE inner : d2 ENDUSE
  putà(inner.a, x),
  putà(x, max(2*x, 8)),
  putà(y, inner.b),
  IF start THEN putà(z, z+y) ENDIF
ENDd1
```

La tâche associée à d1, dans le modèle d'évaluation par équipe d'agents, est donnée ci-dessous. Pour faciliter la lecture, les noms symboliques utilisés dans la tâche reproduisent les noms des objets du segment DESCRIPTION.

```
TASK d1
  BODY
  INITIAL
  ** $e : RECEIVE($start)
        +
  embaucher($inner, d2))
        +
  (embaucher($a, tiroir(int))              "porteuses locales à d1"
  embaucher($b, tiroir(int))
  embaucher($x, tiroir(int))
  embaucher($y, tiroir(int))
  embaucher($z, tiroir(int)))
        +
```

```
   $moi : $moi, FORBID =>$b
        +
   ** $b : DONE
        +
   $moi : RECEIVE($a) =>$inner                    "transmission à inner de
        +                                          son interface"
   $moi : RECEIVE($b) =>$inner
   ENDINITIAL
 ** $e : $e, EVAL
        +
   (embaucher($01, cellà.putà)
   embaucher($02, cellà.putà)                     "embauche des activités et
   embaucher($03, cellà.putà)                     fonctions internes"
   embaucher($04, cellà.getà)
   embaucher($05, cellà.putà))
        +
   $moi : $moi, COMPUTE($start) =>$04
        +
   ** $04 : RESULT($cond)                         "contenu de $start"
        +
   ((($moi : $moi, EVAL =>$inner
        +
    **  $inner : EVALUATED)
   ($moi : $moi, EVAL($a, getà($x)) =>$01
        +
    ** $01 : EVALUATED)
   ($moi : $moi, EVAL($x, max(int.*(2, getà($x)), 8)) =>$02
        +
    ** $02 : EVALUATED)
        +
   ($moi : $moi, EVAL($y, getà($b)) =>$03
        +
    ** $03 : EVALUATED)
   SELECT $cond                                   "traduction du IF"
   (vrai : $moi : $moi, EVAL($z, int.+(getà($z), getà($y)) =>$05
                +

           ** $05 : EVALUATED)
   (faux :  )
   ENDSELECT
        +                                         "fin d'étape EVAL + EVALUATED
   SELECT $error                                  On teste si une erreur a eu lieu
   (vrai :                                        "en cas d'erreur, on le signale
          FIRE($01, $02, $03, $04, $05)           et la tâche est arrêtée"
          $e<= $moi : ERROR('erreur dans l'évaluation des paramètres')
          STOP)
   (faux : $e<= $moi : EVALUATED)
   ENDSELECT
        +
```

```
**$e : $e, START                                    "étape START + DONE"
     +
(($moi : $moi, START =>$inner
     +
**   $inner : DONE)
($moi : $moi, START =>$O1
     +
** $O1 : DONE)
($moi : $moi, START =>$O2
     +
**   $O2 : DONE)
($moi : $moi, START =>$O3
     +
**   $O3 : DONE)
SELECT $cond
  (vrai : $moi : $moi, START =>$O5
                  +
            $O5 : DONE)
  (faux :  )
ENDSELECT
     +
"fin de l'étape START + DONE"
  FIRE($O1, $O2, $O3, $O4, $O5)
$e<= $moi : DONE
ENDTASK
```

IV.3 Traduction dans le modèle d'un segment écrit en BASE CONLAN

La différence essentielle entre PSCL et BCL réside dans l'introduction d'un modèle du temps dans BCL, ce qui nécessite d'adjoindre au modèle d'évaluation quelques mécanismes pour la prise en compte de la dimension temporelle d'une description. Seuls sont donc concernées les définitions de segments DESCRIPTION, FUNCTION et ACTIVITY qui font l'objet d'une évaluation dynamique.

a) Modèle du temps et stabilisation

En PSCL, les porteuses contiennent une valeur d'un type donné. L'évaluation d'une description consiste à effectuer une fois la tâche qui lui est associée, et par conséquent effectuer une fois en parallèle toutes les tâches invoquées par celle-ci à l'aide des échanges EVAL + EVALUATED + START + DONE qui viennent d'être exposés. Plusieurs évaluations successives d'une description écrite en PSCL sont possibles : elles répondent alors à une demande expresse de l'utilisateur, qui communique avec l'équipe d'agents par l'intermédiaire de l'environnement.

Ce n'est qu'avec la définition de BASE CONLAN qu'est modélisé le temps. La notion d'historique de valeurs est introduite avec les types cs_signalà et signalà (voir paragraphe III.2.d et Annexe de la troisième partie). Les porteuses, en BCL, contiennent un historique de valeurs d'un type donné. A chaque instant, l'historique contient une suite de valeurs, qui correspondent à des valeurs transitoires dues à la propagation des changements de valeurs dans le système décrit. Ces valeurs transitoires sont invisibles pour l'utilisateur : seules sont observables les valeurs finales aux instants successifs.

Nous appelons stabilisation le mécanisme qui permet d'obtenir,pour un instant déterminé, les valeurs finales dans toutes les porteuses d'un segment DESCRIPTION, FUNCTION ou ACTIVITY écrit en BASE CONLAN, ou en un langage dérivé de BASE CONLAN (à moins que le segment n'oscille, ce qui est détecté comme une erreur). Un ou plusieurs pas d'évaluation sont exécutés sur ce segment; à chaque pas, la suite des valeurs transitoires, de type cs_signalà, pour l'instant courant est étendue avec une valeur supplémentaire. Une porteuse est localement stable quand la nouvelle valeur ajoutée est égale à la précédente. Le segment est stable quand toutes ses porteuses sont localement stables simultanément.

Dans le modèle d'évaluation, la stabilisation d'un segment DESCRIPTION passe, à chaque pas d'évaluation, par la stabilisation locale de tous les segments FUNCTION et ACTIVITY invoqués dans le corps de la description, et par la croissance au même rythme des suites de valeurs de toutes les porteuses de la description. Cet effet est obtenu par l'invocation, au cours de l'évaluation, des opérations qualifiées INTERPRETERà dans la définition de BCL.

b) Compteurs globaux et locaux

b.1) Compteurs globaux de temps et de pas d'évaluations

Deux tiroirs de type int,"tà"et"sà", ont l'agent environnement pour seul élément dans leur capabilité. En phase d'initialisation d'une tâche d'évaluation de description est prévue la réception de ces deux tiroirs : l'ouvrier attaché à la description la plus englobante reçoit tà et sà de l'environnement, et les transmet, avec les éléments d'interface, à tous les exemplaires de description qu'il contient. Tous les exemplaires de description

ont donc la connaissance de tà et sà, mais ne peuvent les accéder qu'en lecture.

En phase d'initialisation d'une tâche d'évaluation de fonction ou d'activité est prévue la réception de tà seulement. Quand un ouvrier est embauché pour une fonction ou activité, il reçoit tà de l'ouvrier qui le commande, et obtient donc lui aussi un accés en lecture.

b.2) Compteurs locaux de pas d'évaluation

Chaque tâche d'évaluation de segment FUNCTION ou ACTIVITY écrit en BCL contient, en phase d'initialisation, l'embauche d'un tiroir de type int, de nom symbolique $lsà, destiné à compter le nombre de pas d'évaluation locaux pour une invocation de l'opération. Ce compteur local remplace, pour l'ouvrier qui évalue l'opération, le compteur de pas global sà connu des exemplaires de description.

Le compteur local $lsà est initialisé avec la valeur, fournie en paramètre supplémentaire du message EVAL ou COMPUTE, du compteur de pas d'évaluation connu de l'ouvrier qui invoque l'opération. Ce dernier paramètre est connu, dans l'opération invoquée, sous le nom symbolique $esà.

c) Production des paramètres des opérations

Les squelettes de messages EVAL et COMPUTE ont, comme nous venons de le voir, un paramètre de plus que n'en possèdent les invocations des segments ACTIVITY et FUNCTION auxquels ils correspondent, une fois éliminés les attributs (dont le traitement en BCL est identique à celui qui a été exposé pour PSCL).

Pour tous les paramètres non attribut, la traduction dépend, comme pour PSCL, d'une compatibilité entre le type du paramètre formel de l'opération et le type du paramètre effectif fourni. Toutefois les porteuses de BCL ne sont plus des éléments du type cellà(t), mais des éléments d'un type dérivé de signal_carrierà(t), qui se traduisent dans le modèle d'évaluation par des tiroirs dont le contenu est un historique de valeurs de t, élément du type signalà(t). Les combinaisons autorisées entre paramètres formels et effectifs, et leur traduction dans le modèle, sont donc modifiées par rapport à PSCL, et sont données dans le tableau ci-dessous :

Type du paramètre formel t	Paramètre effectif x	Paramètre du message EVAL ou COMPUTE
valeur	constante du type t	x objet de l'univers correspondant
signal t = signalà(tc)	élément de t	x
porteuse t <|signal_carrierà (tc)	porteuse déclarée de type t	$x où $x est le nom symbolique du magasinier embauché pour x
valeur	porteuse à valeurs dans t	contentà($x)
t = signalà(tc)	porteuse à valeurs dans tc	getà($x)
porteuse t<|signal_carrierà (tc)	signal de type signalà(tc)	x
porteuse t <|signal_carrierà (tc)	constante du type tc	packà(x)

Toute autre combinaison est interdite, et provoque une erreur et un arrêt de la construction de la tâche.

d) Traduction d'un segment FUNCTION ou ACTIVITY

Un segment FUNCTION (resp. ACTIVITY) est traduit en une tâche d'évaluation de fonction (resp. d'activité) qui est un sur-ensemble de la tâche produite pour évaluer une même opération en PSCL. Renvoyant le lecteur aux paragraphes III.4.b, III.4.c et IV.2.e précédents, nous ne soulignerons ici que les différences au cours des différentes phases.

Phase 1 - Initialisation

. pour chaque paramètre formel déclaré porteuse non modifiable à valeurs dans un type tc, le magasinier embauché pour ce paramètre correspond à un élément du type variable (tc).
. embauche de $1sà et réception de tà
. pour une opération statique, les embauches supplémentaires sont identiques à celles qui sont prévues pour PSCL, au type des porteuses locales près.

Phase 2 - Evaluation et liaison des paramètres

Semblable à ce qui est fait dans PSCL. Les mêmes actions de liaison, en cas de paramètre de type non modifiable s'appliquent, sachant que le paramètre $pi de la macro "lier_valeur" est maintenant de type signalà(tc). $1sà est initialisé avec le dernier paramètre $esà, valeur du compteur de pas d'évaluation de la tâche appelante.

Phase 3 - Vérification d'assertions sur les paramètres : identique à PSCL.

Phase 4 - Embauche et initialisation des magasiniers locaux.

. Dans le cas d'une opération dynamique, si des porteuses locales sont déclarées, un magasinier est embauché pour chacune d'elle. Puis le tiroir est initialisé avec un historique de valeurs constantes égales à la valeur par défaut, par invocation de la fonction packà. Pour une porteuse, la séquence de squelettes de messages produite est la suivante, où ti et di sont le type et la valeur par défaut de la porteuse.

```
embaucher($mi, ti)
    +
embaucher($fi,  packà)
    +
$moi : $mien, RECEIVE(tà) =>$fi
    +
$moi : $mien, COMPUTE(di, $1sà) =>$fi
    +
** $fi : RESULT($hdi)
    +
$moi : $mien, PUT($hdi) =>$mi
    +
** $mi : DONE
```

. Dans le cas d'une opération statique, si des porteuses locales sont déclarées, il faut éventuellement réajuster la longueur de leur historique, en fonction des valeurs de tà et $lsà, et du type de la porteuse. En effet, un magasinier a bien été embauché à l'initialisation, mais il se peut que l'opération n'ait pas été invoquée pendant plusieurs intervalles de temps. Ce rattrapage est effectué en embauchant un ouvrier pour l'activité stretchà définie sur le type de la porteuse, et invoquant cette activité. Pour une porteuse $mi de type ti, la séquence de squelettes de messages produite est la suivante :

```
embaucher($ai, ti.stretchà)
    +
$moi : $mien, RECEIVE(tà) =>$ai
    +
$moi : $mien, EVAL($mi, $lsà) =>$ai
    +
** $ai : EVALUATED
    +
$moi : $mien, START
    +
** $ai : DONE
```

Phase 5 - Evaluation du corps de l'opération.

Si le corps n'est pas vide, cette phase contient :

1. l'embauche d'un ouvrier pour chaque invocation d'activité dynamique dans le corps de l'opération, suivie de la transmission du compteur tà à chaque ouvrier.
2. l'envoi en parallèle de tous les messages EVAL aux activités invoquées dans le corps de l'opération.
3. la réception de tous les messages EVALUATED correspondants. Si $error est positionné à vrai, sauter au point 6.
4. l'envoi en parallèle d'un message START à tous les ouvriers activés en 2.
5. la réception de tous les messages DONE correspondants
6. le renvoi des ouvriers embauchés en 1. Ceci constitue un pas d'évaluation de l'opération. Si $error est positionné à vrai, sauter au point 11.
7. déverrouiller tous les paramètres porteuses non modifiables. Pour chaque porteuse de l'opération (locale ou d'interface) embaucher un ouvrier pour l'activité finstepà définie sur son

type, et démarrer tous ces ouvrier, en fournissant à chacun tà, $lsà et une porteuse, et en produisant l'ensemble des séquences EVAL + EVALUATED + START + DONE.
Après réception de tous les messages DONE, renvoyer tous les ouvriers finstepà, et verrouiller à nouveau tous les paramètres porteuses non modifiables. Toutes les porteuses connues de l'opération ont à ce point des historiques de même longueur, avec $lsà+1 valeurs à l'instant courant. Incrémenter $lsà de 1.

8. Tester la stabilité de toutes les porteuses, en évaluant pour chaque porteuse $mi l'expression :

$$getà(\$mi)\ \{tà,\ \$lsà\} = getà(\$mi)\ \{tà,\ \$lsà - 1\}$$

pour laquelle les ouvriers adéquats sont embauchés et leur tâche évaluée.
Si toutes les porteuses sont stables, passer au point 9. Sinon, $lsà est comparé au nombre limite de pas d'évaluation (fonction du nombre de porteuses locales).Si $lsà n'a pas atteint cette limite, recommencer l'évaluation de cette phase au point 1; sinon errorà est positionné à vrai, et sauter au point 11.
9. Si le corps de l'opération contient des assertions, vérification de celles-ci identique à ce qui se fait en PSCL.
10. Si l'opération est une fonction, l'évaluation du résultat se fait comme pour PSCL. Si l'opération est une activité, ce point est sauté.
11. Déverrouiller tous les paramètres porteuses non modifiables.
Pour chaque porteuse locale ou en paramètre (opération statique), ou pour chaque paramètre de type porteuse modifiable (opération dynamique), embaucher un ouvrier pour exécuter l'activité shrinkà et échanger avec chacun la séquence de messages EVAL + EVALUATED + START + DONE. Ce dernier point à pour effet de supprimer vis-à-vis de l'environnement d'appel, et pour les invocations ultérieures, toutes les valeurs intermédiaires dûes à la stabilisation locale de l'opération.
Renvoyer tous les ouvriers embauchés pendant les phases 3, 4, 5 et émettre un compte-rendu d'évaluation en direction de l'appelant.

e) Evaluation d'une description écrite en BASE CONLAN

Mis à part le fait qu'un segment DESCRIPTION écrit en BCL connaît deux compteurs supplémentaires pré-définis tà et sà, son comportement et par conséquent sa traduction dans le modèle d'agents, est identique en BCL à celui qu'il aurait en PSCL. Un exemplaire de description se contente de transmettre les messages le long de l'arborescence des ouvriers, en modifiant le contexte attaché aux messages. Une description complète est stabilisée globalement, sous le contrôle de l'environnement. Pour chaque description, un nombre limite de pas d'évaluation est établi, connu de l'environnement.

Une fois embauché l'ouvrier associé à l'exemplaire de description englobant, ce qui provoque l'embauche de tous les agents permanents de l'équipe, par évaluation des blocs d'initialisation, l'évaluation dynamique de la description est démarrée, avec tà = 1 et sà = 1. Elle se décompose en 3 phases.

Phase 1

L'environnement envoie un message EVAL à l'ouvrier chef de l'équipe d'agents (soit "descr" son nom). descr à son tour envoie des messages EVAL, à chacun de ses exemplaires de description locaux et aux activités invoquées dans son corps; et le processus est répété tout au long de l'arbre des imbrications. Quand descr a reçu en retour tous les messages EVALUATED, un message EVALUATED est transmis à l'environnement.

L'environnement envoie alors à descr un message START; le processus est répété pour les couples de message START + DONE.

Si l'environnement reçoit un message DONE, c'est le signal de la fin d'un pas d'évaluation pour la description globale. Avancer en phase 2.

Si au lieu du message EVALUATED ou DONE un message ERROR est transmis à l'environnement, celui-ci signale l'erreur à l'utilisateur, et la suite des opérations peut dépendre de la volonté de l'utilisateur (arrêt, ou reprise en début de phase 1 après la modification de certaines porteuses de la description).

Phase 2

L'environnement invoque finstepà sur toutes les porteuses de la description et des exemplaires de description englobés, jusqu'aux feuilles de l'arborescence des imbrications d'exemplaires de description. L'environnement, nous l'avons vu, connaît ces porteuses et a le droit de les modifier (contrairement aux porteuses locales aux fonctions et activités). Le compteur sà est incrémenté de 1.

Puis l'environnement teste la stabilité de toutes les porteuses qu'il connaît. Si une porteuse au moins n'est pas stable, et sà est inférieur à la limite fixée, reprendre au début de la phase 1. Si sà a atteint sa limite, signaler une erreur d'oscillation à l'utilisateur.

Si toutes les porteuses sont stables, avancer en phase 3.

Phase 3

Si tà a atteint la durée à simuler, le signaler à l'utilisateur et attendre sa prochaine commande.

Dans le cas contraire, l'environnement invoque finintà sur toutes les porteuses de la description qu'il connaît. Puis il incrémente tà de 1, il remet sà à 1, et l'évaluation reprend au début de la phase 1.

C I N Q U I E M E P A R T I E

APPLICATIONS

I INTRODUCTION

Il est souhaitable qu'une réflexion théorique, justifiée par des considérations pratiques, ait des applications directes. C'est ce que nous voulons montrer dans ce dernier chapitre. La méthode CONLAN, et le modèle d'évaluation qui l'accompagne, n'ont d'intérêt que s'ils permettent des définitions de langages de description d'envergure suffisante pour être utiles aux concepteurs, et de niveaux d'abstraction variés.

Dans cette optique, nous avons défini en CONLAN la sémantique des langages de description développés à l'I.M.A.G. : CASSANDRE, LASCAR et LASSO. La définition des types et opérateurs primitifs de ces langages s'est révélée très aisée. Nous avons été aussi capables de définir certaines des particularités syntaxiques de ces langages, en particulier les différentes sortes de conditions et de contrôle. Par contre, des incompatibilités syntaxiques existaient dès le départ entre ces langages et CONLAN, notamment dans l'écriture des paramètres, interfaces et déclarations. Sur le plan sémantique, BASE CONLAN, d'où nous sommes partis, offrait par rapport aux trois langages cités des possibilités plus étendues, en particulier les définitions de types, que nous n'avons pas jugé utile de supprimer, et les indexations.

Le résultat obtenu est donc trois langages de la famille CONLAN de niveaux CASSANDRE, LASCAR et LASSO, offrant à l'utilisateur toutes les primitives de ces langages, lui offrant en outre quelques possibilités supplémentaires. Le lecteur trouvera ces trois définitions dans les paragraphes suivants, sous forme de segments CONLAN.

La numérotation des lignes, introduite par logiciel, ne fait pas partie du texte écrit en CONLAN. Elle nous servira, dans le commentaire de présentation, à référencer des portions de la définition formelle qu'il précède.

II DEFINITION EN CONLAN D'UN LANGAGE DE NIVEAU LASCAR

II.1 Présentation

Le langage LASCAR2 est défini à partir de BASE CONLAN (lignes 9, 10).

Un certain nombre de types de BASE CONLAN sont transportés dans LASCAR2 : les booléens, les entiers et entiers positifs, les intervalles, les dimensions et indices de tableaux, les tableaux de dimensions fixées, les historiques (lignes 12-14). Inversement, d'autres types, tels les chaînes de caractères et les structures sont inaccessibles pour l'utilisateur de LASCAR2.

LES TYPES DE LASCAR2

Puis les types propres à LASCAR sont définis : les impulsions (lignes 21-43), les états de l'automate de contrôle (lignes 47-79), les registres chargés sous condition d'horloge (lignes 83-94), les compteurs (lignes 111-127). Parmi les porteuses de BASE CONLAN, seuls les fils de connexion à valeurs booléennes (ligne 105), entières (ligne 107) et impulsion (ligne 99), redéfinis comme SUBTYPE, sont transmis à LASCAR2 : en particulier, une horloge est définie comme un fil de connexion à valeurs dans le type impulsion, volontairement distingué du type booléen pour éviter des confusions (ligne 99). Les registre chargés sous condition d'impulsion et à valeurs booléennes (ligne 101), entières (ligne 109) ou d'état de l'automate (ligne 103), sont les registres primitifs de LASCAR, et reçoivent une désignation abrégée par une définition de SUBTYPE.

Plusieurs chargements de registre et de compteurs conditionnés par la même impulsion, peuvent être regroupés, et donner lieu à l'écriture simplifiée (lignes 138-145) :

<impulsion> chargement 1, ..., chargement i;

Les formats infixés des différents chargements sont aussi définis (lignes 146-163) :

chargement de registre : r <= expression
incrémentation de compteur : PI(C)
décrémentation de compteur : MI(C)
transition d'état de
l'automate : => valeur_état ou CHARGER valeur_état

L'ensemble des instructions qui dépendent d'un état de l'automate sont regroupées dans une instruction conditionnelle d'un format particulier (lignes 197-207) :

: nom_d_état : liste_d'instructions;

LES OPERATIONS DE LASCAR2

Le reste de la définition est consacré aux opérations non inclues dans des définitions de types.

Les fonctions reg_rise (lignes169-174) et sig_rise (lignes 176-179) produisent une impulsion aux fronts montants du signal booléen contenu respectivement dans un registre et un fil de connexion. L'opérateur unaire de valeur absolue, absent de BASE CONLAN, est défini dans LASCAR2 (ligne 181-189).

Les fonctions privées extend_indexer (lignes 214-227) et extend_array_dim (lignes 230-241) servent à rajouter un intervalle comme première dimension. Elles sont ultérieurement utilisées dans la définition des conversions entre entiers et vecteurs booléens. De même, trois types utilitaires, non transmis à l'utilisateur, sont définis pour rendre plus aisée la définition de ces conversions : selecteurà (lignes 249-252) regroupe l'ensemble des indices qui permettent d'indexer un élément d'un tableau; vectorà (lignes 259-262) et nonvectorà (lignes 264-267) forment une partition des tableaux d'un type donné, entre les vecteurs ligne et les autres (nous admettrons que la première dimension parcourt les lignes d'un tableau). Ainsi

a[-5 : 0, 1 : 1; 0 : 0] est un vecteur ligne
b[1 : 1; -5 : 0; 0 : 0] n'est pas un vecteur ligne
c[1 : 4] est un vecteur ligne

La fonction normalize_vectorà supprime toutes les dimensions dégénérées d'un vecteur ligne, et numérote la première dimension en indices croissants à partir de 1 (lignes 274-283) :

normalize_vectorà (a) a pour dimension : [1 : 6]

L'opération de réduction par rapport à un opérateur binaire transforme un vecteur ligne en scalaire (lignes 305-317), et un autre tableau en un tableau ayant la première dimension en moins (lignes 319-335). Un format d'appel par les opérateurs unaires /& et /| est défini pour la réduction par rapport au "et" et au "ou" booléens, les seules réductions utilisées en pratique.

A l'exception de l'égalité et de l'inégalité, déjà traitées dans le type array, l'écriture sous forme infixée de tous les opérateurs unaires et binaires définis sur les scalaires est étendue aux tableaux de dimensions compatibles, les opérations binaires s'effectuant entre éléments de même rang de leurs opérandes (lignes 346-436). Ces extensions sont rendues extrêmement concises par l'utilisation des fonctions génériques fbinà, fmonà, fboolà et de l'activité générique abinà définies sur le type array.

La fonction générique ext_monadic (lignes 420-435) n'avait pas été prévue dans BASE CONLAN. Elle permet d'appliquer à tous les éléments d'un tableau de type array (u) un opérateur unaire de u dans v, produisant un tableau de type array (v) et de dimensions normalisées compatibles. Cette fonction est ensuite utilisée pour étendre aux tableaux les fonctions valeur-absolue (ligne 430), front montant de valeur dans un registre (ligne 431) et dans un fil de connexion (ligne 432), et toutes les fonctions contentà de dé-repérage (lignes 442-472).

L'activité array_load étend aux tableaux de registres le chargement sous condition d'impulsion (lignes 479-490). Ici, l'un des opérandes, l'impulsion, reste un scalaire.

CODAGE ET DECODAGE DES ENTIERS EN VECTEURS BOOLEENS

Parmi les nombreuses possibilités de considérer un vecteur booléen comme le codage d'un entier, et réciproquement de coder un entier par un vecteur booléen, nous avons défini les deux interprétations les plus fréquemment utilisées :

- codage en base 2 d'un entier positif,
- codage en complément à 2 d'un entier, la première position déterminant le signe.

Pour chaque codage, nous avons dû distinguer entre entier scalaire et tableau dans un sens, entre vecteur ligne booléen et autre tableau booléen en sens inverse, la transformation d'entier à booléen induisant une transformation dimensionnelle. Nous avons donc défini les fonctions suivantes :

valn1 (lignes 507-515) : conversion de booléen scalaire en entier positif
valn2 (lignes 525-538) : conversion de vecteur booléen en entier positif
valn3 (lignes 540-555) : conversion de tableau booléen en tableau d'entiers positifs, ayant une dimension de moins
valn4 (lignes 560-571) : conversion de vecteur booléen en entier relatif (codage en complément à 2)
valn5 (lignes 574-587) : conversion de tableau booléen en tableau d'entiers relatifs, ayant une dimension de moins.

Les fonctions valn1, valn2, valn3 sont notées par l'opérateur unaire $, les fonctions valn4, valn5 sont notées par l'opérateur unaire INTVAL.

valb1 (lignes 518-521) : conversion des entiers 0 et 1 en booléen
valb2 (lignes 592-606) : conversion d'entier scalaire positif en vecteur booléen codé en base 2 sur i bits
valb3 (lignes 608-621) : conversion de tableau d'entiers positifs en tableau de booléens, ayant une première dimension supplémentaire de i bits
valb4 (lignes 627-639) : conversion d'entier scalaire en vecteur booléen, codé en complément à 2 sur i bits
valb5 (lignes 641-654) : conversion de tableau d'entiers en tableau de booléens, ayant une première dimension supplémentaire de i bits codée en complément à 2.

Pour les fonctions valb2, valb3, valb4, valb5, si le nombre de bits indiqué en paramètre ne suffit pas à coder les entiers fournis, il y a troncature des poids forts et émission d'un message d'avertissement. Les fonctions valb2, valb3 sont notées par l'opérateur !/i/ préfixant le paramètre à convertir, les fonctions valb4 et valb5 sont notées BINVAL /i/, où i représente la longueur du vecteur booléen résultant du codage d'un entier.

INSTRUCTIONS IMPLICITES

Dans les langages LASCAR et CASSANDRE, toute description contenant un automate de contrôle possède un registre implicite qui contient à tout instant l'état courant de l'automate. De même, les noms des états sont implicitement déclarés lors de leur apparition dans le texte de la description. En CONLAN "pur", au contraire, aucune déclaration implicite n'est admise.

Les définitions qui suivent, de LASCAR2 et CASSANDRE2, supposent que soient écrites dans chaque description, soit par l'utilisateur, soit par un pré-processeur,

- la déclaration du registre d'état de l'automate :

  DECLARE regetat : etat ENDDECLARE

  L'identificateur "regetat" est donc réservé.

- la déclaration de tous les états de l'automate :

  DECLARE ..... : valeur_etat END

- une instruction inconditionnelle, qui valide à chaque étape de calcul l'état courant :

  VALIDER(regetat)

II.2 Définition formelle de LASCAR2

```
 9  REFLAN bcl
10  CONLAN lascar2  BODY

12  CARRY bool, int, nnint, pint, range, array_dimension, indexer,
13        fixed_array, cs_signal@, signal@
14  ENDCARRY

16  "/*****************************************************************************
17    Definition des types de valeurs et des types de porteuses de LASCAR qui
18     n'existent pas dans BASE CONLAN
19  *****************************************************************************"/

21  TYPE impulsion
22  BODY
24     {'HIGH', 'LOW'}
25     CARRY =, ~= ENDCARRY

27        "/ Union de 2 impulsions  /"
28     FUNCTION or (x,y: impulsion) : impulsion
29     RETURN
31        IF x = 'HIGH' | y = 'HIGH' THEN 'HIGH' ELSE 'LOW' ENDIF
32        FORMAT@ EXTEND expl.2 MEANS or(id1, id2) ENDFORMAT
34     ENDor

36        "/ Impulsion haute sur front montant d'un historique de valeurs
37           Booleennes /"
38     FUNCTION rise (x: signal@(bool)) : impulsion
39     RETURN
40        IF x{t@-1,} = 0 & x{t@, s@} = 1 THEN 'HIGH' ELSE 'LOW' ENDIF
41     ENDrise
43  ENDimpulsion

45    "/ Definition des etats de l'automate de controle  /"

47  TYPE valeur_etat
48  BODY
50     signal_carrier@(bool,0)
51     CARRY content@, get@, put@, shrink@ ENDCARRY

53     ACTIVITY valider(W y: valeur_etat)
54     BODY
56        put@(    y , extend_rts(spart(old(y)), t@, s@+1, 1))
57        FORMAT@
58        EXTEND activity_invocation.10
59        activity_invocation = 'VALIDER' ref_to_declared:id1
60              MEANS valider(id1) ENDFORMAT
62     ENDvalider

64      "/ Tout etat non mis à 1 pendant une etape de calcul est mis à 0/"

66     INTERPRETER@ ACTIVITY finstep@(W y: valeur_etat)
67     BODY
68        IF ~known(spart(old(y)), t@, s@+1) THEN
69        put@(old(y), extend_rts(spart(old(y), t@, s@+1, 0))
70     ENDfinstep@
```

```
72      "/ Tous les etats sont invalides en debut d'intervalle de temps /"

74      INTERPRETER@ ACTIVITY finint@(W y: valeur_etat)
75      BODY
76      put@(old(y), extend_rts(spart(old(y)), t@+1, 1, Ø))
77      ENDfinint@
79    ENDvaleur_etat

81      "/ Definition des registres charges sous condition d'impulsion /"

83    TYPE clocked_reg(x: any@; init:x)
84    BODY
86      rtvariable(x, init)
87      CARRY content@, delay, finstep@, finint@, get@, put@, shrink@ END

89      ACTIVITY load (W y:clocked_reg(x, init); a: x; c:impulsion)
90      BODY
91        IF c='HIGH' THEN old(y) <- a ENDIF
92      ENDload
94    ENDclocked_reg

96    "/ Definition des noms des types de porteuses primitives dans LASCAR.
97    La valeur initiale par defaut en debut de simulation est precisee. /"

99    SUBTYPE horloge BODY  terminal(impulsion, 'LOW') ENDhorloge

101   SUBTYPE registre BODY clocked_reg(bool, Ø) ENDregistre

103   SUBTYPE etat BODY clocked_reg(valeur_etat, error@) ENDetat

105   SUBTYPE signal BODY terminal(bool,Ø) ENDsignal

107   SUBTYPE entsig BODY terminal(int, Ø) ENDentsig

109   SUBTYPE entreg BODY clocked_reg(int, Ø) ENDentreg

111   TYPE compteur
112   BODY
114     entreg
115     CARRYALL

117     ACTIVITY plusun(W x: compteur; c: impulsion)
118     BODY
119       load(old(x), x+1, c)
120     ENDplusun

122     ACTIVITY moinsun (W x: compteur; c : impulsion)
123     BODY
124       load(old(x), x-1, c)
125     ENDmoinsun
127   ENDcompteur


131   "/*************************************************************************
132     Format d'appel des chargements de registre, et des operations
133     sur les compteurs, qui dependent d'une condition impulsionnelle
134   *************************************************************************/"
```

```
136   FORMAT@
138      EXTEND activity_invocation.11
139         R activity_invocation = loading
140      EXTEND loading.1
141         R loading = '<' expression:id3 '>' suitload ';'
142      EXTEND suitload.1
143         R suitload = oneload ',' suitload
144      EXTEND suitload.2
145         R suitload = oneload
146      EXTEND oneload.1
147         R oneload = ref_to_declared:id1 '<=' expression:id2
148         MEANS load(id1, id2, id3)
149      EXTEND oneload.2
150         R oneload = 'P1' '(' ref_to_declared:id5 ')'
151         MEANS plusun(id5, id3)
152      EXTEND oneload.3
153         R oneload = 'M1' '(' ref_to_declared:id6 ')'
154         MEANS moinsun(id6, id3)
155      EXTEND oneload.4
156         R oneload = transition expression:id4
157         MEANS load(regetat@, id4, id3)
158      EXTEND transition.1
159         R transition = '=>'
160      EXTEND transition.2
161         R transition = 'CHARGER'
163   ENDFORMAT

165   "/*****************************************/"
166   "/ Fonctions non inclues dans les types /"
167   "/*****************************************/"

169   FUNCTION reg_rise (x: registre) : impulsion
170   RETURN IF ~(x%1) & x THEN 'HIGH' ELSE 'LOW' ENDIF
171      FORMAT@ EXTEND exp8.8
172               R exp8 = '^' exp8:id1 MEANS reg_rise(id1)
173      ENDFORMAT
174   ENDreg_rise

176   FUNCTION sig_rise (x: signal) : impulsion
177   RETURN IF ~(x%1) & x THEN 'HIGH' ELSE 'LOW' ENDIF
178      FORMAT@ EXTEND exp8.8 MEANS sig_rise(id1)  ENDFORMAT
179   ENDsig_rise

181   FUNCTION abs (x: int) : nnint
182   RETURN
184      IF x >= 0 THEN x ELSE -x ENDIF
185      FORMAT@ EXTEND exp8.5
186               R exp8 = 'ABS' exp8:id1 MEANS abs(id1)
187      ENDFORMAT
189   ENDabs

192   "/*************************************************************************
193      Definition du format d'ecriture des instructions sous etat
194      de l'automate de controle
195   *************************************************************************/"

197   FORMAT@
199      EXTEND conditional_invocation.3
```

```
200         R conditional_invocation = state_cond operation_invocations
201                                    endstate
202      EXTEND state_cond.1
203         R state_cond = ':' identifier;valeur_etat:idl ':'
204         MEANS IF idl = 1 THEN
205      EXTEND endstate.1
206         R endstate = ';' MEANS ENDIF
208   ENDFORMAT

210   "/*************************************************************************/"
211   "/ 2 fonctions utilitaires permettant d'etendre un index
212      et un ensemble de dimensions  par une premiere dimension /"
213   "/*************************************************************************/"

215   PRIVATE FUNCTION extend_indexer(x: range; y: indexer) : indexer
216   RETURN
218      THE@ z: indexer WITH
220         size@(z) = size@(y) + 1  &
221         tail(z) = y              &
222         size@(z[1]) = size@(x)   &
223         FORALL@ i: bint(1, size@(x)) IS
224                  z[1][i] = x[i] ENDFOR
226      ENDTHE
228   ENDextend_indexer


231   PRIVATE FUNCTION extend_array_dim (x: range; y: array_dimension) :
232                                          array_dimension
233   RETURN THE@ z: array_dimension WITH
235         size@(z) = size@(y) + 1  &
236         tail(z) = y              &
237         z [1] = x
239      ENDTHE
241   ENDextend_array_dim

243   "/*************************************************************************/"
244   "/ types utilitaires, non connus de l'utilisateur /"
245   "/*************************************************************************/"

247   "/ ensemble des index permettant de selectionner un element d'un tableau /"

249   SUBTYPE selecteur@ (x: array(u: any@))
250   BODY
251      ALL z : degenerate_indexer WITH  isin(z, dpart(x)) ENDALL
252   ENDselecteur@


255   "/ differenciation entre vecteur ligne et autre tableau ayant
256      2 dimensions au moins; cette distinction sert a construire le langage,
257      elle est transparente pour l'utilisateur /"

259   SUBTYPE vector@(u: any@)  BODY
260      ALL x: array(u) WITH size@(dpart(x)) = 1 |
261                           size_array(tail(dpart(x))) = 1 ENDALL
262   ENDvector@

264   SUBTYPE nonvector@(u: any@)  BODY
265      ALL x: array(u) WITH size@(dpart(x)) > 1 &
```

```
266                                  size_array(tail(dpart(x))) > 1 ENDALL
267   ENDnonvector@

269   "/ normalisation d'un vecteur ligne:
270      toutes les dimensions degenerees, sauf la lere, sont supprimees
271      la borne inf. de la lere dimension est mise a 1
272      la borne sup. de la lere dimension est mise a la taille du vecteur  /"

274   FUNCTION normalize_vector@ (x: vector@(u: any@)) : vector@(u)
276      RETURN THE@ y: vector@(u) WITH
278              dpart(y) = ( 1 : size_dimension(x,1) ) &
279              vpart(y) = vpart(x)
281      ENDTHE
283   ENDnormalize_vector@


286   "/*************************************************************************/"
287   "/ Definition de l'operation de reduction d'un vecteur ou d'un tableau
288       par rapport à un operateur binaire:
289           Un vecteur ligne est transforme en un scalaire
290           Un autre tableau est transforme en un tableau
291             ayant une dimension de moins /"
292   "/*************************************************************************/"

294   PRIVATE FUNCTION repeat_eval (u: any@; x: tytuple@(u);
295                                              f: FUNCTION(u, u) :u) : u
297      RETURN IF size@(x) = 1 THEN x[1]
298             ELSE f( x[1], repeat_eval(u, tail(x), f) ) ENDIF
300   ENDrepeat_eval

302   "/ Un operateur unaire est defini pour le format d'appel de la reduction
303        par rapport au 'et' et au 'ou' booleens /"

305   FUNCTION reduction1(u: any@; x: vector@(u); f: FUNCTION(u, u):u ) : u
307      RETURN  repeat_eval(vpart(x))
308      FORMAT@
310         EXTEND exp8.6
311            exp8 = '/|' exp8:id1 MEANS reduction1(bool, id1, |)
312         EXTEND exp8.7
313            exp8 = '/&' exp8:id1 MEANS reduction1(bool, id1, &)
315      ENDFORMAT
317   ENDreduction1

319   FUNCTION reduction2(u: any@; x: nonvector@(u); f: FUNCTION(u, u):u): array(u)
321      RETURN normalize (THE@ y: array(u) WITH
323              dpart(y) = tail(dpart(x)) &
324              FORALL@ p: selecteur@(y) IS
325                      y[p] = reduction1(u, x[ extend_indexer(dpart(x)[1], p) ], f)
326              ENDFOR ENDTHE)
328      FORMAT@
330         EXTEND exp8.6 MEANS reduction2(bool, id1, |)
331         EXTEND exp8.7 MEANS reduction2(bool, id1, &)
333      ENDFORMAT
335   ENDreduction2


338   "/*************************************************************************/"
339   "/Extension des operateurs sur les scalaires à des operateurs sur tableaux
```

```
340     Ces extensions font appel aux fonctions generiques definies sur le
341     type array, qui evitent une redefinition de chaque operation.
342     Seul le format d'appel doit etre precisé.
343     On conserve la meme notation infixee, et les memes priorites  /"
344  "/*************************************************************************/"

346     "/ operations binaires internes sur les booleens /"

348  FUNCTION array_dyadic_bool(x,y: array(bool); f: FUNCTION(bool, bool):bool)
349                                : array(bool)
351     RETURN fbin@(x, y, f)
352     FORMAT@
354        EXTEND exp1.2 MEANS array_dyadic_bool (id1, id2, |)
355        EXTEND exp2.2 MEANS array_dyadic_bool (id1, id2, xor)
356        EXTEND exp2.3 MEANS array_dyadic_bool (id1, id2, eqv)
357        EXTEND exp3.2 MEANS array_dyadic_bool (id1, id2, &)
358        EXTEND exp4.4 MEANS array_dyadic_bool (id1, id2, <)
359        EXTEND exp4.5 MEANS array_dyadic_bool (id1, id2, =<)
360        EXTEND exp4.6 MEANS array_dyadic_bool (id1, id2, >)
361        EXTEND exp4.7 MEANS array_dyadic_bool (id1, id2, >=)
363     ENDFORMAT
365  ENDarray_dyadic_bool

367     "/ operations binaires internes sur les entiers /"

369  FUNCTION array_dyadic_int (x, y: array(int); f: FUNCTION(int, int):int)
370                               : array(int)
372     RETURN fbin@(x, y, f)
373     FORMAT@
375        EXTEND exp5.2 MEANS array_dyadic_int (id1, id2, +)
376        EXTEND exp5.3 MEANS array_dyadic_int (id1, id2, -)
377        EXTEND exp6.2 MEANS array_dyadic_int (id1, id2, *)
378        EXTEND exp6.3 MEANS array_dyadic_int (id1, id2, /)
379        EXTEND exp6.4 MEANS array_dyadic_int (id1, id2, MOD)
380        EXTEND exp7.2 MEANS array_dyadic_int (id1, id2, ^)
382     ENDFORMAT
384  ENDarray_dyadic_int

386     "/ operations unaires internes /"

388  FUNCTION array_monadic(u: any@; y: array(u); f: FUNCTION(u):u): array(u)
390     RETURN fmon@(y, f)
391     FORMAT@
393        EXTEND exp8.2 MEANS array_monadic(int, id1, +)
394        EXTEND exp8.3 MEANS array_monadic(int, id1, -)
395        EXTEND exp8.4 MEANS array_monadic(bool, id1, ~)
397     ENDFORMAT
399  ENDarray_monadic

401     "/ operations de comparaison sur les entiers /"

403  FUNCTION array_compare_int(x,y:array(int); f: FUNCTION(int, int):bool)
404                                : array(bool)
406     RETURN fbool@(x, y, f)
407     FORMAT@
409        EXTEND exp4.4 MEANS array_compare_int(id1, id2, <)
410        EXTEND exp4.5 MEANS array_compare_int(id1, id2, =<)
411        EXTEND exp4.6 MEANS array_compare_int(id1, id2, >)
```

```
412         EXTEND exp4.7 MEANS array_compare_int(id1, id2, >=)
414       ENDFORMAT
416   ENDarray_compare_int

418       "/ operations unaires externes /"

420   FUNCTION ext_monadic(u,v: any@; x: array(u); f: FUNCTION(u):v) : array(v)
422       RETURN
424          normalize ( THE@ z: array(v) WITH
425             dpart(z) = dpart(x)  &
426             FORALL@ p: selecteur@(x) IS z[p] = f (x[p])  ENDFOR
427          ENDTHE
429       FORMAT@
430          EXTEND exp8.5 MEANS ext_monadic(int, nnint, id1, abs)
431          EXTEND exp8.8 MEANS ext_monadic(registre, impulsion, id1, reg_rise)
432          EXTEND exp8.8 MEANS ext_monadic(signal, impulsion, id1, sig_rise)
433       ENDFORMAT
435   ENDext_monadic


438   "/*************************************************************************
439      Definition du contenu d'un tableau de porteuses
440   *************************************************************************/"

442   INTERPRETER@ FUNCTION content@(x: array(horloge)) : array(impulsion)
443      RETURN ext_monadic(horloge, impulsion, x, horloge.content@)
444   ENDcontent@

446   INTERPRETER@ FUNCTION content@(x: array(registre)) : array(bool)
447      RETURN ext_monadic(registre, bool, x, registre.content@)
448   ENDcontent@

450   INTERPRETER@ FUNCTION content@(x: array(signal)) : array(bool)
451      RETURN ext_monadic(signal, bool, x, signal.content@)
452   ENDcontent@

454   INTERPRETER@ FUNCTION content@(x: array(entreg)) : array(int)
455      RETURN ext_monadic(entreg, int, x, entreg.content@)
456   ENDcontent@

458   INTERPRETER@ FUNCTION content@(x: array(entsig)) : array(int)
459      RETURN ext_monadic(entsig, int, x, entsig.content@)
460   ENDcontent@

462   INTERPRETER@ FUNCTION content@(x: array(etat)) : array(valeur_etat)
463      RETURN ext_monadic(etat, valeur_etat, x, etat.content@)
464   ENDcontent@

466   INTERPRETER@ FUNCTION content@(x: array(variable(int, 0))) : array(int)
467      RETURN ext_monadic(variable(int,0), int, x, variable(int,0).content@)
468   ENDcontent@

470   INTERPRETER@ FUNCTION content@(x: array(variable(bool, 0))) : array(bool)
471      RETURN ext_monadic(variable(bool,0), bool, x, variable(bool,0).content@)
472   ENDcontent@

474   "/*************************************************************************
475      Operations de modification de valeur des porteuses scalaires
```

```
476      etendues aux tableaux de porteuses.
477   ***************************************************************************/"

479   ACTIVITY array_load (W y: array(clocked_reg(x: any@; init: x));
480                        a: array(x); c:impulsion)
481   ASSERT compatible(y, a) ENDASSERT
482   BODY
484      OVER i:bint(1, size_array(dpart(y))) REPEAT
485           <c>  vpart(y)[i] <= vpart(a)[i]  ENDOVER
486      FORMAT@
487         EXTEND oneload.1 MEANS  array_load(id1, id2, id3)
488      ENDFORMAT
490   ENDarray_load

492   FORMAT@
494      EXTEND activity_invocation.2 MEANS abin@(id1, id2, connect)
495      EXTEND activity_invocation.3 MEANS abin@(id1, id2, assign)
497   ENDFORMAT

499   "/ fin de l'extension des operations scalaires aux tableaux /"

501   "/***************************************************************************
502      Operations de codage et de decodage des entiers en vecteurs booleens
503   ***************************************************************************/"

505   "/ conversions de type  de 0 et 1 /"

507   FUNCTION valn1 (x: bool) : nnint
508   RETURN
510      IF x = 0 THEN 0 ELSE 1 ENDIF
511      FORMAT@ EXTEND exp8.9
512              R exp8 = '$' exp8:id1 MEANS valn1(id1)
513      ENDFORMAT
515   ENDvaln1


518   FUNCTION valb1 (x: nnint) : bool
519   RETURN
520      IF x = 0 THEN 0 ELIF x = 1 THEN 1 ELSE error@ ENDIF
521   ENDvalb1

523   "/ traduction de vecteur booleen en entier positif /"

525   FUNCTION valn2(x: vector@(bool)) : nnint
526   BODY
528      DECLARE nbel : variable(int, 0) ENDDECLARE

530      nbel := size_dimension(x, 1)
531      RETURN
533         IF nbel = 1 THEN valn1(normalize_vector@(x)[1])
534         ELSE valn1(normalize_vector@(x)[1])*2^(nbel-1) +
535              valn2(normalize_vector@(x)[2:nbel]) ENDIF
536         FORMAT@ EXTEND exp8.9 MEANS valn2(id1) ENDFORMAT
538   ENDvaln2

540   FUNCTION valn3 (x: nonvector@(bool)) : array(nnint)
541   RETURN
543      normalize ( THE@ y: array(nnint) WITH
```

```
545         dpart(y) = tail(dpart(x)) &
546         FORALL@ p: selecteur@(y) IS
548            y[p] = valn2 ( x[extend_indexer(dpart(x)[1], p)] )
550         ENDFOR
552      ENDTHE )
553      FORMAT@ EXTEND exp8.9 MEANS valn3(id1) ENDFORMAT
555   ENDvaln3

557   "/ traduction de vecteur booleen en entier positif ou negatif.
558      codage en complement a 2, le premier bit etant un bit de signe /"

560   FUNCTION valn4 ( x: vector@(bool)) : int
561   BODY
563      DECLARE nbel : variable(int, 0) ENDDECLARE
564      nbel := size_dimension(x, 1)
565      RETURN
566         IF vpart(x)[1] = 0 THEN valn2(x) ELSE valn2(x) -2^nbel ENDIF
567      FORMAT@ EXTEND exp8.10
568            R exp8 = 'INTVAL' exp8:id1 MEANS valn4(id1)
569      ENDFORMAT
571   ENDvaln4


574   FUNCTION valn5 (x: nonvector@(bool)) : array(int)
575   RETURN
577      normalize ( THE@ y : array(int) WITH
579         dpart(y) = tail(dpart(x))  &
580         FORALL@ p : selecteur@(y) IS
581            y[p] = valn4 ( x[extend_indexer(dpart(x)[1], p)] )
582         ENDFOR
584      ENDTHE )
585      FORMAT@ EXTEND exp8.10 MEANS valn5(id1) ENDFORMAT
587   ENDvaln5

589   "/ traduction d'un entier positif en vecteur booleen de longueur i /"
590   "/ si la longueur est insuffisante, troncature et emission d'un message /"

592   FUNCTION valb2 (x:nnint; i:pint) : vector@(bool)
593   BODY
595      DECLARE y[i:1] : variable(bool, 0) ENDDECLARE

597      IF x >= 2^i THEN warning@ ENDIF
598      OVER j FROM 1 TO i REPEAT
599            y[j] := valb1( (x MOD 2^j) / 2^(j-1) ) ENDOVER
600      RETURN  content@(y)
601      FORMAT@ EXTEND exp8.11
602            R exp8 = '!' '/' constant_denotation:id2 '/' exp8:id1
603            MEANS valb2(id1, id2)
604      ENDFORMAT
606   ENDvalb2

608   FUNCTION  valb3 (x: array(nnint); i: pint) ; array(bool)
609   RETURN
611      normalize (  THE@ y: array(bool) WITH
613         dpart(y) = extend_array_dim(1:i, dpart(x))  &
614         FORALL@ p: selecteur@(x) IS
615            y[extend_indexer(1:i, p)] = valb2(x[p], i)
616         ENDFOR
```

```
613       ENDTHE )
619       FORMAT@ EXTEND exp8.11 MEANS valb3(id1, id2) ENDFORMAT
621    ENDvalb3

623    "/ traduction d'un entier relatif en vecteur booleen de longueur i.
624       si la longueur est insuffisante, troncature et emission d'un message.
625       codage en complement a 2, le premier bit etant un bit de signe /"

627    FUNCTION valb4 (x: int; i: pint) : vector@(bool)
628    BODY
630       IF x >= 2^(i-1) | x < -2^(i-1) THEN warning@ ENDIF
631       RETURN
633          IF x >= Ø THEN valb2(x, i) ELSE valb2(x MOD 2^i, i) ENDIF
634       FORMAT@ EXTEND exp8.12
635               R exp8 = 'BINVAL' '/' constant_denotation:id2 '/' exp8:id1
636               MEANS valb4(id1, id2)
637       ENDFORMAT
639    ENDvalb4

641    FUNCTION valb5 (x: array(int); i: pint) : array(bool)
642    RETURN
644       normalize ( THE@ y: array(bool) WITH
646          dpart(y) = extend_array_dim(1:i, dpart(x))  &
647          FORALL@ p: selecteur@(x) IS
648             y[extend_indexer(1:i, p)] = valb4(x[p], i)
651    ENDTHE )
652    FORMAT@ EXTEND exp8.12 MEANS valb5(id1, id2) ENDFORMAT
654    ENDvalb5

656    ENDlascar2
```

II.3 Définition en CONLAN d'un langage de niveau CASSANDRE

LASCAR étant un sur-ensemble de CASSANDRE, la méthode la plus rapide est de définir CASSANDRE2 à partir de LASCAR2 (lignes 664-665). On transporte depuis LASCAR2 tous les types et toutes les opérations de CASSANDRE2 (lignes 667-687), et on ôte de la syntaxe les règles relatives au format d'appel des opérations qui n'ont pas été transportées (lignes 691, 692).

```
1   659                DEFINITION DE CASSANDRE COMME SOUS-ENSEMBLE DE LASCAR


1   664   REFLAN  lascar2
1   665   CONLAN cassandre2  BODY

1   667   CARRY
1   670                    "/ types fournis à l'utilisateur /"
1   671         bool, int, pint, nnint, range, array_dimension, indexer, fixed_array,
1   672         impulsion, valeur_etat,  horloge, registre, etat, signal,

1   675                    "/ types transmis mais inconnus de l'utilisateur /"
1   676         cs_signal@, signal@, selecteur@, vector@, nonvector@,

1   679                    "/ fonctions transmises à l'utilisateur /"
1   680         reg_rise, sig_rise, array_dyadic_bool, array_monadic, ext_monadic,
1   681         array_load, valn1, valn2, valn3, reduction1, reduction2, abs

1   684                    "/ fonctions transmises mais inconnues de l'utilisateur /"
1   685         normalize_vector@, content@
1   687   ENDCARRY

1   689   "/ restrictions syntaxiques /"

1   691   FORMAT@ REMOVE oneload.2, oneload.3, exp8.11, exp8.12
1   692   ENDFORMAT

1   694   ENDcassandre2
```

III DEFINITION EN CONLAN D'UN LANGAGE DE NIVEAU LASSO

III.1 Présentation

Le langage LASSO2 est lui aussi définissable à partir de BASE CONLAN (lignes 8, 9). Les mêmes types qui avaient été transportés de BCL dans LASCAR2 sont transportés aussi dans LASSO2; à ces types s'ajoutent les chaînes de caractères, les structures, et comme porteuses les variables (lignes 11-15).

La définition de LASSO2 contient elle aussi les types et fonctions intermédiaires (lignes 244-330) permettant de construire la réduction (lignes 335-365), l'extension aux tableaux des opérateurs définis sur les scalaires (lignes 378-492), et les opérations de conversion entre entiers et vecteurs booléens (lignes 498-648). Toutes ces définitions sont identiques à celles de LASCAR2.

LES TYPES DE LASSO2

Le type event (lignes 23-177) est l'analogue, pour LASSO2, du type impulsion pour LASCAR2. On définit sur ce type les opérateurs COMMESI, PUIS, ETDEPLUS, LAFOISNO et retard qui ont été présentés au chapitre deux; les quatre derniers sont définis par une fonction statique. D'une étape de calcul à la suivante il est nécessaire de garder le souvenir du passage à 'HIGH' du premier (pour PUIS) ou de l'un (pour ETDEPLUS) des évènements opérandes. La fonction "lafois" mémorise dans une variable interne le nombre de passages à 'HIGH' de son deuxième opérande. La fonction "retard", qui est notée par l'opérateur "%" comme tous les autres retards dans CONLAN, utilise un fil de connexion interne pour enregistrer l'historique des valeurs que prend son premier opérande, et applique à ce fil de connexion l'opérateur "%" défini sur les porteuses au niveau de BASE CONLAN. Enfin, quatre fonctions, non disponibles pour l'utilisateur, servent à construire les instructions conditionnelles de la partie spécifications : la fonction dateà, appliquée à un évènement, renvoie la date de la dernière occurrence de l'évènement, ou 0 par défaut; la fonction entreà a pour résultat 1 entre chaque occurrence de son 1er opérande et l'occurrence suivante de son 2ème opérande, et 0 en dehors; la fonction suivà a pour résultat 1 pendant les i unités de temps qui suivent chaque occurrence de son premier opérande; enfin la fonction échantillonà produit un évènement toutes les d unités de temps à partir de la date de sa première évaluation.

Le type control_signal (lignes 181-216) définit les signaux du graphe de contrôle de LASSO, et correspond pour LASSO2 au type valeur_état de LASCAR2. Il est construit à l'aide de la porteuse "variable", ce qui assure les propriétés de disponibilité immédiate de sa valeur, et de mémorisation. Toutes les fonctions automatiquement invoquées par l'interpréteur du langage sont transportées à partir du type variable. Par contre, on ne donne à l'utilisateur aucune opération pour modifier directement la valeur d'un signal de contrôle. Les opérations de modification "validerà" et "invaliderà" sont invoquées par l'intermédiaire des primitives du graphe de contrôle, et détectent une erreur si l'on chercher à valider un signal déjà à 1 ou à invalider un signal dont la valeur est 0. Les seules opérations sur le type contol signal dont l'utilisateur dispose sont les fonctions unaires rise_event (notée *1) et fall_event (notée *0) qui créent un évènement sur les fronts montant et descendant d'un signal.
Le type nomsig (ligne 221) est simplement défini comme une variable à valeurs de type control_signal.

LE GRAPHE DE CONTROLE

Toutes les transitions primitives du graphe de contrôle sont définies comme des activités statiques (lignes 692-870) qui se comportent comme des automates à 3 états : prêt, occupé, fin. Toutes ces primitives sont bâties sur le même principe. Dans l'état prêt, on teste si les conditions de déclenchement de la transition sont réunies, auquel cas on passe à l'état occupé. On reste à l'état occupé pendant le temps de traversée de la transition, exprimé par délai en LASSO. Cette primitive délai est définie à l'aide de l'activité statique waità (lignes 658-671) qui agit elle-même comme un petit automate retardant le basculement de deux variables booléennes passées en paramètre, ce qui correspond pour chaque transition au passage de l'état occupé à l'état fin. Dans l'état fin, la transition invalide les signaux d'entrée et valide les signaux de sortie correspondants : l'invocation des opérations validerà et invaliderà assure la détection des erreurs de synchronisation dues à un déclenchement de plusieurs transitions par un même signal, ou au blocage d'une transition qui avait été déclenchée.

LES PREDICATS COMPOSES

Dans LASSO2, les formes conditionnelles de la partie spécification de LASSO peuvent être éclatées entre plusieurs instructions ASSERT. L'écriture des

fenêtres de validité et des instants de validité est définie par extensions syntaxiques à partir de la production assertion (lignes 875-938). Une assertion devant toujours avoir un résultat booléen, les définitions des instants et des fenêtres fait explicitement ressortir le fait qu'en dehors des plages de validité qui indiquent qu'une vérification de prédicats est demandée, l'assertion a pour valeur 1. Lorsque l'évaluation de la liste de prédicats englobés a lieu, la valeur du prédicat composé et égale à la conjonction booléenne de ses prédicats simples.

```
5   III.2 - DEFINITION FORMELLE DU LANGAGE LASSO EN CONLAN

8   REFLAN bcl
9   CONLAN lasso2  BODY

11  CARRY bool, int, nnint, pint, range, array_dimension, indexer,
12        fixed_array, cs_signal@, signal@ , variable,
13        string, field_descriptor, field_descriptor_list, record
14  ENDCARRY

16  "/*************************************************************************
17    Definition des types de valeurs et des types de porteuses de LASSO qui
18     n'existent pas dans BASE CONLAN
19  *************************************************************************"/

21     "/ Un evenement est positionne lors du changement de valeur d'un
22        signal de controle /"
23  TYPE event
24  BODY
26     {'HIGH', 'LOW'}
27     CARRY =, ~= ENDCARRY

29        "/ Union de 2 events  /"
30     FUNCTION commesi (x,y: event) : event
31     RETURN
33        IF x = 'HIGH' | y = 'HIGH' THEN 'HIGH' ELSE 'LOW' ENDIF
34        FORMAT@ EXTEND expl.4
35                R expl = expl:id1 'COMMESI' exp2:id2
36                MEANS commesi(id1, id2) ENDFORMAT
38     ENDcommesi
```

```
40        "/ Sequence de 2 events /"
41      STATIC FUNCTION puis (x,y:event) : event
42      BODY
44         DECLARE ax: variable(bool, 1);
45                 ay: variable(bool, 0);
46                 result: variable(event, 'LOW')
47         ENDDECLARE
48         IF ax THEN result := 'LOW',
49                    If x='HIGH' THEN ax := 0, ay := 1 ENDIF
50         ENDIF,
51         IF ay THEN
52            IF y = 'HIGH' THEN ax := 1, ay := 0, result := 'HIGH'
53            ELSE result := 'LOW'
54            ENDIF
55         ENDIF
56         RETURN result
57         FORMAT@ EXTEND expl.5
58                 R expl = expl:id1 'PUIS' exp2:id2
59                 MEANS puis(id1, id2) ENDFORMAT
61      ENDpuis

63         "/ 2 events se sont produits, dans un ordre quelconque /"
64      STATIC FUNCTION etdeplus (x,y:event) : event
65      BODY
67         DECLARE init: variable(bool, 1);
68                 ax, ay: variable(bool, 0);
69                 result: variable(event, 'LOW')
70         ENDDECLARE
71         IF init THEN result := 'LOW',
72            IF x = 'HIGH' THEN ay := 1, init := 0 ENDIF,
73            IF y = 'HIGH' THEN ax := 1, init := 0 ENDIF
74         ENDIF,
75         IF ax THEN
76            IF x = 'HIGH' THEN
77               ax := 0, ay := 0, result := 'HIGH', init := 1
78               ENDIF
79         ENDIF,
80         IF ay THEN
81            IF y = 'HIGH' THEN
82               ax := 0, ay := 0, result := 'HIGH', init := 1
83               ENDIF
84         ENDIF
85         RETURN result
86         FORMAT@ EXTEND expl.6
87                 R expl = expl:id1 'ETDEPLUS' exp2:id2
88                 MEANS etdeplus(id1, id2) ENDFORMAT
90      ENDetdeplus

92         "/ Date a laquelle un evenement s'est produit pour la derniere fois.
93            S'il ne s'est jamais produit, la date vaut 0 /"

95      STATIC FUNCTION date@(x:event) : nnint
96      BODY
98         DECLARE t: variable(int, 0) ENDDECLARE
99         IF x = 'HIGH' THEN t = t@ ENDIF
100        RETURN t
102     ENDdate@
```

```
104     "/ Genere un evenement la ième fois que son 2ème parametre vaut HIGH /"

106     STATIC FUNCTION  lafois(ATT i: pint; x: event) : event
107     BODY
109        DECLARE a: variable(int, Ø) ENDDECLARE
110        IF x = 'HIGH' THEN a := a + 1 ENDIF
111        RETURN
112           IF a = i THEN x ELSE 'LOW' ENDIF
113        FORMAT@ EXTEND exp8.14
114                R exp8 = 'LAFOISNO' integer:id1 'QUE' exp8:id2
115                MEANS lafois(id1, id2) ENDFORMAT
117     ENDlafois

119     STATIC FUNCTION retard (x: event; ATT i: pint) : event
120     BODY
122        DECLARE a: terminal(event, 'LOW') ENDDECLARE
123        a := x
124        RETURN a % i
125        FORMAT@ EXTEND exp7.3 MEANS retard(id1, id2) ENDFORMAT
126        "/ notation uniforme du retard par x % i /"
128     ENDretard

130        "/ entre@ a la valeur 1 entre chaque occurrence de x et
131           l'occurrence suivante de y /"

133     STATIC FUNCTION entre@ (x, y: event) : bool
134     BODY
136        DECLARE a: variable(bool, Ø) ENDDECLARE
137        IF a THEN
138           IF y = 'HIGH' THEN a := Ø ENDIF
139        ELSE IF x = 'HIGH' THEN a := 1 ENDIF
140        ENDIF
141        RETURN a
143     ENDentre@

145        "/ suiv@ a la valeur 1 pendant i unites de temps
146           suivant chaque occurrence de x /"

148     STATIC FUNCTION suiv@ (x: event; i: nnint) : bool
149     BODY
151        DECLARE a: variable(bool, Ø);
152                c: variable(int, Ø) ENDDECLARE
153        IF ~a THEN
154           IF x = 'HIGH' THEN a := 1, c := t@ ENDIF
155        ELSE IF t@-c >= i THEN a := Ø ENDIF
156        ENDIF
157        RETURN a
159     ENDsuiv@

161        "/ Generation d'un evenement a intervalles reguliers a partir
162           de l'instant de la premiere invocation /"

164     STATIC FUNCTION echantillon@ (ATT d: pint) : event
165     BODY
167        DECLARE start: variable(int, Ø);
168                a: variable(event, 'LOW') ENDDECLARE
169        IF start = Ø THEN start := t@, a := 'HIGH'
170        ELSE IF (t@ - start) MOD d = Ø THEN a := 'HIGH'
171                                       ELSE a := 'LOW' ENDIF
172        ENDIF
173        RETURN a
175     ENDechantillon@
177  ENDevent
```

```
179     "/ Definition des signaux du graphe de controle  /"

181   TYPE control_signal
182   BODY
184       variable(bool,Ø)
185       CARRY content@, get@, put@, shrink@, finstep@, finint@ ENDCARRY

187       ACTIVITY valider@ (W x: control_signal)
188       BODY
189          IF x = Ø THEN old(x) := 1 ELSE  error@ ENDIF
190       ENDvalider@

192       ACTIVITY invalider@ (W x: control_signal)
193       BODY
194          IF x = 1 THEN old(x) := Ø ELSE error@ endif
195       ENDinvalider@

197       FUNCTION rise_event(x: control_signal) : event
198       RETURN
199          IF s@ > 1 THEN
200             IF get@(x)[t@, s@-1] = Ø & x = 1
201             THEN 'HIGH' ELSE 'LOW' ENDIF
202          ELSE 'LOW' ENDIF
203          FORMAT@ EXTEND exp8.15
204                  R exp8 = '*1' exp8:id1 MEANS rise_event(id1)
205          ENDFORMAT
206       ENDrise_event

208       FUNCTION fall_event(x: control_signal) : event
209       RETURN
210          IF old(x) % 1 = 1 & x = Ø THEN 'HIGH' ELSE 'LOW' ENDIF
211          FORMAT@ EXTEND exp8.16
212                  R exp8 = '*Ø' exp8:id1 MEANS fall_event(id1)
213          ENDFORMAT
214       ENDfall_event
216   ENDcontrol_signal

218   "/ Definition des noms des types de porteuses primitives dans LASSO.
219      La valeur initiale par defaut en debut de simulation est precisee. /"

221   SUBTYPE nomsig BODY  variable(control_signal, error@) ENDnomsig

223   "/*****************************************/"
224   "/ Fonctions non inclues dans les types /"
225   "/*****************************************/"

227   FUNCTION abs (x: int) : nnint
228   RETURN
230      IF x >= Ø THEN x ELSE -x ENDIF
231      FORMAT@ EXTEND exp8.5
232              R exp8 = 'ABS' exp8:id1 MEANS abs(id1)
233      ENDFORMAT
235   ENDabs
```

```
239  "/*************************************************************************/"
240  "/ 2 fonctions utilitaires permettant d'etendre un index
241     et un ensemble de dimensions  par une premiere dimension /"
242  "/*************************************************************************/"

244  PRIVATE FUNCTION extend_indexer(x: range; y: indexer) : indexer
245  RETURN
247     THE@ z: indexer WITH
249        size@(z) = size@(y) + 1  &
250        tail(z) = y              &
251        size@(z[1]) = size@(x)   &
252        FORALL@ i: bint(1, size@(x)) IS
253              z[1][i] = x[i] ENDFOR
255     ENDTHE
257  ENDextend_indexer

260  PRIVATE FUNCTION extend_array_dim (x: range; y: array_dimension) : array_dimensi
261  RETURN
263     THE@ z: array_dimension WITH
265        size@(z) = size@(y) + 1  &
266        tail(z) = y              &
267        z [1] = x
269     ENDTHE
271  ENDextend_array_dim

273  "/*************************************************************************/"
274  "/ types utilitaires, non connus de l'utilisateur /"
275  "/*************************************************************************/"

277  "/ ensemble des index permettant de selectionner un element d'un tableau /"

279  SUBTYPE selecteur@ (x: array(u: any@))
280  BODY
281     ALL z : degenerate_indexer WITH  isin(z, dpart(x)) ENDALL
282  ENDselecteur@


285  "/ differenciation entre vecteur ligne et autre tableau ayant
286     2 dimensions au moins; cette distinction sert a la construction du langage,
287     elle est trensparente pour l'utilisateur /"

289  SUBTYPE vector@(u: any@)  BODY
290     ALL x: array(u) WITH size@(dpart(x)) = 1 |
291                          size_array(tail(dpart(x))) = 1 ENDALL
292  ENDvector@

294  SUBTYPE nonvector@(u: any@)  BODY
295     ALL x: array(u) WITH size@(dpart(x)) > 1 &
296                          size_array(tail(dpart(x))) > 1 ENDALL
297  ENDnonvector@

299  "/ normalisation d'un vecteur ligne:
300        toutes les dimensions degenerees, sauf la lere, sont supprimees
301        la borne inf. de la lere dimension est mise a 1
302        la borne sup. de la lere dimension est mise a la taille du vecteur  /"

304  FUNCTION normalize_vector@ (x: vector@(u: any@)) : vector@(u)
306     RETURN THE@ y: vector@(u) WITH
308            dpart(y) = ( 1 : size_dimension(x,1) ) &
309            vpart(y) = vpart(x)
311     ENDTHE
313  ENDnormalize_vector@
```

```
316  "/*************************************************************************/"
317  "/ Definition de l'operation de reduction d'un vecteur ou d'un tableau
318      par rapport à un operateur binaire:
319          Un vecteur ligne est transforme en un scalaire
320          Un autre tableau est transforme en un tableau
321            ayant une dimension de moins /"
322  "/*************************************************************************/"

324  PRIVATE FUNCTION repeat_eval (u: any@; x: tytuple@(u);
325                                f: FUNCTION(u, u) :u) : u
327     RETURN IF size@(x) = 1 THEN x[1]
328            ELSE f( x[1], repeat_eval(u, tail(x), f) ) ENDIF
330  ENDrepeat_eval

332     "/ Un operateur unaire est defini pour le format d'appel de la reduction
333        par rapport au 'et' et au 'ou' booleens /"

335  FUNCTION reduction1(u: any@; x: vector@(u); f: FUNCTION(u, u):u ) : u
337     RETURN  repeat_eval(vpart(x))
338     FORMAT@
340        EXTEND exp8.6
341           exp8 = '/|' exp8:id1 MEANS reduction1(bool, id1, |)
342        EXTEND exp8.7
343           exp8 = '/&' exp8:id1 MEANS reduction1(bool, id1, &)
345     ENDFORMAT
347  ENDreduction1

349  FUNCTION reduction2(u: any@; x: nonvector@(u); f: FUNCTION(u, u):u): array(u)
351     RETURN normalize (THE@ y: array(u) WITH
353            dpart(y) = tail(dpart(x)) &
354            FORALL@ p: selecteur@(y) IS
355                    y[p] = reduction1(u, x[ extend_indexer(dpart(x)[1], p) ], f)
356            ENDFOR ENDTHE)
358     FORMAT@
360        EXTEND exp8.6 MEANS reduction2(bool, id1, |)
361        EXTEND exp8.7 MEANS reduction2(bool, id1, &)
363     ENDFORMAT
365  ENDreduction2


368  "/*************************************************************************/"
369  "/ Extension des operateurs sur les scalaires à des operateurs sur tableaux
370     Ces extensions font appel aux fonctions generiques definies sur le
371     type array, qui evitent une redefinition de chaque operation.
372     Seul le format d'appel doit etre precisé.
373     On conserve la meme notation infixee, et les memes priorites  /"
374  "/*************************************************************************/"

376     "/ operations binaires internes sur les booleens /"

378  FUNCTION array_dyadic_bool (x, y: array(bool); f: FUNCTION (bool, bool):bool)
379                              : array(bool)
381     RETURN fbin@(x, y, f)
382     FORMAT@
384        EXTEND exp1.2 MEANS array_dyadic_bool (id1, id2, |)
385        EXTEND exp2.2 MEANS array_dyadic_bool (id1, id2, xor)
386        EXTEND exp2.3 MEANS array_dyadic_bool (id1, id2, eqv)
387        EXTEND exp3.2 MEANS array_dyadic_bool (id1, id2, &)
388        EXTEND exp4.4 MEANS array_dyadic_bool (id1, id2, <)
389        EXTEND exp4.5 MEANS array_dyadic_bool (id1, id2, =<)
390        EXTEND exp4.6 MEANS array_dyadic_bool (id1, id2, >)
391        EXTEND exp4.7 MEANS array_dyadic_bool (id1, id2, >=)
393     ENDFORMAT
395  ENDarray_dyadic_bool
```

```
397      "/ operations binaires internes sur les entiers /"

399   FUNCTION array_dyadic_int (x,y: array(int); f: FUNCTION(int, int):int)
400                                 : array(int)
402      RETURN fbin@(x, y, f)
403      FORMAT@
405         EXTEND exp5.2 MEANS array_dyadic_int (id1, id2, +)
406         EXTEND exp5.3 MEANS array_dyadic_int (id1, id2, -)
407         EXTEND exp6.2 MEANS array_dyadic_int (id1, id2, *)
408         EXTEND exp6.3 MEANS array_dyadic_int (id1, id2, /)
409         EXTEND exp6.4 MEANS array_dyadic_int (id1, id2, MOD)
410         EXTEND exp7.2 MEANS array_dyadic_int (id1, id2, ^)
412      ENDFORMAT
414   ENDarray_dyadic_int

416      "/ operations unaires internes /"

418   FUNCTION array_monadic(u: any@; y: array(u); f: FUNCTION(u):u) : array(u)
420      RETURN fmon@(y, f)
421      FORMAT@
423         EXTEND exp8.2 MEANS array_monadic(int, id1, +)
424         EXTEND exp8.3 MEANS array_monadic(int, id1, -)
425         EXTEND exp8.4 MEANS array_monadic(bool, id1, ~)
427      ENDFORMAT
429   ENDarray_monadic

431      "/ operations de comparaison sur les entiers /"

433   FUNCTION array_compare_int (x, y:array(int); f: FUNCTION (int, int):bool)
434                                  : array(bool)
436      RETURN fbool@(x, y, f)
437      FORMAT@
439         EXTEND exp4.4 MEANS array_compare_int(id1, id2, <)
440         EXTEND exp4.5 MEANS array_compare_int(id1, id2, =<)
441         EXTEND exp4.6 MEANS array_compare_int(id1, id2, >)
442         EXTEND exp4.7 MEANS array_compare_int(id1, id2, >=)
444      ENDFORMAT
446   ENDarray_compare_int

448      "/ operations unaires externes /"

450   FUNCTION ext_monadic(u, v: any@; x: array(u);  f: FUNCTION(u):v) : array(v)
452      RETURN
454         normalize ( THE@ z: array(v) WITH
455            dpart(z) = dpart(x)  &
456            FORALL@ p: selecteur@(x) IS z[p] = f (x[p])  ENDFOR
457         ENDTHE
459      FORMAT@ EXTEND exp8.5 MEANS ext_monadic(int, nnint, id1, abs)
460      ENDFORMAT
462   ENDext_monadic

464   "/*************************************************************************
465      Definition du contenu d'un tableau de porteuses
466   *************************************************************************/"
```

```
468  INTERPRETER@ FUNCTION content@(x: array(control_signal)) : array(bool)
469     RETURN ext_monadic(control_signal, bool, x, control_signal.content@)
470  ENDcontent@

472  INTERPRETER@ FUNCTION content@(x: array(namsig)) : array(control_signal)
473     RETURN ext_monadic(namsig, control_signal, x, namsig.content@)
474  ENDcontent@

476  INTERPRETER@ FUNCTION content@(x: array(variable(t:any@, init:t))): array(t)
477     RETURN ext_monadic(variable(t, init), t, x, variable(t, init).content@)
478  ENDcontent@

480  "/*****************************************************************************
481     Operations de modification de valeur des porteuses scalaires
482     etendues aux tableaux de porteuses.
483  *****************************************************************************/"

485  FORMAT@
487     EXTEND activity_invocation.2 MEANS abin@(id1, id2, connect)
488     EXTEND activity_invocation.3 MEANS abin@(id1, id2, assign)
490  ENDFORMAT

492  "/ fin de l'extension des operations scalaires aux tableaux /"

494  "/*****************************************************************************
495     Operations de codage et de decodage des entiers en vecteurs booleens
496  *****************************************************************************/"

498  "/ conversions de type  de 0 et 1 /"

500  FUNCTION valnl (x: bool) : nnint
501  RETURN
503     IF x = 0 THEN 0 ELSE 1 ENDIF
504     FORMAT@ EXTEND exp8.9
505             R exp8 = '$' exp8:id1 MEANS valnl(id1)
506     ENDFORMAT
508  ENDvalnl


511  FUNCTION valbl (x: nnint) : bool
512  RETURN
513     IF x = 0 THEN 0 ELIF x = 1 THEN 1 ELSE error@ ENDIF
514  ENDvalbl

516  "/ traduction de vecteur booleen en entier positif /"

518  FUNCTION valn2(x: vector@(bool)) : nnint
519  BODY
521     DECLARE nbel : variable(int, 0) ENDDECLARE

523     nbel := size_dimension(x, 1)
524     RETURN
526        IF nbel = 1 THEN valnl(normalize_vector@(x)[1])
527        ELSE valnl(normalize_vector@(x)[1])*2^(nbel-1)
528             + valn2(normalize_vector@(x)[2:nbel])
529        ENDIF
```

```
530         FORMAT@ EXTEND exp8.9 MEANS valn2(id1) ENDFORMAT
532     ENDvaln2

534     FUNCTION valn3 (x: nonvector@(bool)) : array(nnint)
535     RETURN
537        normalize ( THE@ y: array(nnint) WITH
539           dpart(y) = tail(dpart(x)) &
540           FORALL@ p: selecteur@(y) IS
542              y[p] = valn2 ( x[extend_indexer(dpart(x)[1], p)] )
544           ENDFOR
546        ENDTHE )
547        FORMAT@ EXTEND exp8.9 MEANS valn3(id1) ENDFORMAT
549     ENDvaln3

551     "/ traduction de vecteur booleen en entier positif ou negatif.
552        codage en complement a 2, le premier bit etant un bit de signe /"

554     FUNCTION valn4 ( x: vector@(bool)) : int
555     BODY
557        DECLARE nbel : variable(int, 0) ENDDECLARE
558        nbel := size_dimension(x, 1)
559        RETURN
560           IF vpart(x)[1] = 0 THEN valn2(x) ELSE valn2(x) -2^nbel ENDIF
561        FORMAT@ EXTEND exp8.10
562                R exp8 = 'INTVAL' exp8:id1 MEANS valn4(id1)
563        ENDFORMAT
565     ENDvaln4


568     FUNCTION valn5 (x: nonvector@(bool)) : array(int)
569     RETURN
571        normalize ( THE@ y : array(int) WITH
573           dpart(y) = tail(dpart(x))  &
574           FORALL@ p : selecteur@(y) IS
575              y[p] = valn4 ( x[extend_indexer(dpart(x)[1], p)] )
576           ENDFOR
578        ENDTHE )
579        FORMAT@ EXTEND exp8.10 MEANS valn5(id1) ENDFORMAT
581     ENDvaln5

583     "/ traduction d'un entier positif en vecteur booleen de longueur i /"
584     "/ si la longueur est insuffisante, troncature et emission d'un message /"

586     FUNCTION valb2 (x:nnint; i:pint) : vector@(bool)
587     BODY
589        DECLARE y[i:1] : variable(bool, 0) ENDDECLARE

591        IF x >= 2^i THEN warning@ ENDIF
592        OVER j FROM 1 TO i REPEAT
593                     y[j] := valb1( (x MOD 2^j) / 2^(j-1) ) ENDOVER
594        RETURN  content@(y)
595        FORMAT@ EXTEND exp8.11
596                R exp8 = '!' '/' constant_denotation:id2 '/' exp8:id1
597                MEANS valb2(id1, id2)
598        ENDFORMAT
600     ENDvalb2
```

```
602  FUNCTION  valb3 (x: array(nnint); i: pint) ; array(bool)
603  RETURN
605     normalize (  THE@ y: array(bool) WITH
607        dpart(y) = extend_array_dim(1:i, dpart(x))  &
608        FORALL@ p: selecteur@(x) IS
609           y[extend_indexer(1:i, p)] = valb2(x[p], i)
610        ENDFOR
612     ENDTHE )
613     FORMAT@ EXTEND exp8.11 MEANS valb3(id1, id2) ENDFORMAT
615  ENDvalb3

617  "/ traduction d'un entier relatif en vecteur booleen de longueur i.
618     si la longueur est insuffisante, troncature et emission d'un message.
619     codage en complement a 2, le premier bit etant un bit de signe /"

621  FUNCTION valb4 (x: int; i: pint) : vector@(bool)
622  BODY
624     IF x >= 2^(i-1) | x < -2^(i-1) THEN warning@ ENDIF
625     RETURN
627        IF x >= 0 THEN valb2(x, i) ELSE valb2(x MOD 2^i, i) ENDIF
628     FORMAT@ EXTEND exp8.12
629             R exp8 = 'BINVAL' '/' constant_denotation:id2 '/' exp8:id1
630             MEANS valb4(id1, id2)
631     ENDFORMAT
633  ENDvalb4

635  FUNCTION valb5 (x: array(int); i: pint) : array(bool)
636  RETURN
638     normalize ( THE@ y: array(bool) WITH
640        dpart(y) = extend_array_dim(1:i, dpart(x))  &
641        FORALL@ p: selecteur@(x) IS
642           y[extend_indexer(1:i, p)] = valb4(x[p], i)
645  ENDTHE )
646  FORMAT@ EXTEND exp8.12 MEANS valb5(id1, id2) ENDFORMAT
648  ENDvalb5

650  "/ ***************************************************************************
651     Definition des primitives du graphe de controle,
652     ainsi que de leur format d'ecriture
653  ***************************************************************************/"

655     "/ Notion d'attente, utilisee dans toutes les primitives
656        à duree de traversee non nulle /"

658  STATIC ACTIVITY wait@ (d: nnint; W a, b: variable(bool, 0))
659  BODY
661     DECLARE start: variable(int, 0);
662             busy: variable(bool, 0)
663     ENDDECLARE

665     IF a & ~busy THEN start := t@, busy := 1 ENDIF,

667     IF busy THEN
668        IF t@ - start = d THEN a := 0, b := 1, busy := 0 ENDIF
669     ENDIF
671  ENDwait@
```

```
673      "/ Ecriture des listes de signaux en entree et en sortie de transition/"

675   FORMAT@
677      EXTEND entree_transition.1
678         R entree_transition = ref_to_declared; control_signal:id1
679      EXTEND entree_transition.2
680         R entree_transition = ref_to_declared; control_signal:id2
681                                  '&' entree_transition:id3
682         MEANS id2 # id3
683      EXTEND es_transition.1
684         R es_transition = ref_to_declared;vector@(control_signal):id4
685      EXTEND es_transition.2
686         R es_transition = ref_to_declared;vector@(control_signal):id5
687                              ',' es_transition:id6
688         MEANS id5 # id6
690   ENDFORMAT

692      "/ Definition de la transition ET /"

694   STATIC ACTIVITY transition_et (d:nnint; W x, y: vector@(control_signal))
695   BODY
697      DECLARE fin, occupe: variable(bool, 0);
698              pret: variable(bool, 1)
699      ENDDECLARE
700      IF pret THEN
701         IF (/&x) & ~(/|y) THEN pret := 0, occupe := 1 ENDIF
702      ENDIF,
703      wait@(d, occupe, fin),
704      IF fin THEN
705         OVER i FROM lb(x, 1) TO rb(x, 1) REPEAT
706            invalider@ (x[i])  ENDOVER,
707         OVER j FROM lb(y, 1) TO rb(y, 1) REPEAT
708            valider@ (y[j])  ENDOVER,
709         fin := 0, pret := 1
710      ENDIF
711      FORMAT@
713         EXTEND activity_invocation.20
714            R activity_invocation = '?' entree_transition:id2 '?'
715                     delai:id1 'VALIDER' '(' es_transition:id3 ')'
716            MEANS transition_et(id1, id2, id3)
717         EXTEND delai.1
718            R delai = 'DELAI' expression:id4 MEANS id4
719         EXTEND delai.2
720            R delai =    MEANS 0H
722      ENDFORMAT
724   ENDtransition_et

726      "/ Definition de la transition SELON /"

728   STATIC ACTIVITY transition_selon(d:nnint; p: bool;
729                                 W x, y[1:2]: control_signal)
730   BODY
732      DECLARE fin, occupe: variable(bool, 0);
733              pret: variable(bool, 1) ENDDECLARE
734      IF pret THEN
735         IF x & ~(/|y) THEN pret := 0, occupe := 1 ENDIF
```

```
736      ENDIF,
737      wait@(d, occupe, fin),
738      IF fin THEN
739         invalider@(x),
740         IF p THEN valider@(y[1]) ELSE valider@(y[2]) ENDIF,
741         fin := Ø, pret := 1
742      ENDIF
743      FORMAT@
745         EXTEND activity_invocation.21
746            R activity_invocation = '?' entree_transition:id3 '?'
747                  'SELON' expression:id2 delai:id1
748                'VALIDER' '(' es_transition:id4 ')'
749            MEANS transition_selon(id1, id2, id3, id4)
751      ENDFORMAT
752   ENDtransition_selon

754      "/ Definition de la transition INDEX /"

756   STATIC ACTIVITY transition_index(d:nnint; i: int; ATT m, n: int;
757                                    W x, y[1:2]: control_signal)
758   ASSERT i .< bint(m, n) ENDASSERT
759   BODY
761      DECLARE fin, occupe: variable(bool, Ø);
762              pret: variable(bool, 1)
763      ENDDECLARE
764      IF pret THEN
765         IF x & ~(/|y) THEN pret := Ø, occupe := 1 ENDIF
766      ENDIF,
767      wait@(d, occupe, fin),
768      IF fin THEN
769         invalider@(x),
770         valider@(y[i]),
771         fin := Ø, pret := 1
772      ENDIF
773      FORMAT@
775         EXTEND activity_invocation.22
776            R activity_invocation = '?' entree_transition:id3 '?'
777                    'INDEX' expression:id2 bornel:id5 borne2:id6
778                     delai:id1 'VALIDER' '(' es_transition:id4 ')'
779            MEANS transition_index(id1, id2, id5, id6, id3, id4)
780         EXTEND bornel.1
781            R bornel = 'DE' expression:id7 MEANS id7
782         EXTEND bornel.2
783            R bornel =    MEANS Ø1
784         EXTEND borne2.1
785            R borne2 = 'A' expression:id8 MEANS id8
786         EXTEND borne2.2
787            R borne2 =    MEANS size_dimension(id4, 1) -1
789      ENDFORMAT
791   ENDtransition_index

793      "/ Definition de la transition CHOIX /"

795   STATIC ACTIVITY transition_choix(d: nnint; W e: control_signal;
796                                    W x, y: vector@(control_signal))
797   ASSERT size_dimension(x, 1) = size_dimension(y, 1) ENDASSERT
```

```
798   BODY
800      DECLARE fin, occupe: variable(bool, Ø);
801              pret: variable(bool, 1);
802              numero: variable(int, Ø)
803      ENDDECLARE
804      IF pret THEN
805         IF e & ~(/|y) & (/|x) THEN
806            numero := THE@ i:bint(1, size_dimension(x, 1)) WITH
807                           vpart(x)[i] = 1 ENDTHE,  "/ erreur si plus d'un /"
808           pret := Ø, occupe := 1
809         ENDIF
810      ENDIF,
811      wait@(d, occupe, fin),
812      IF fin THEN
813         invalider@(e),
814         invalider@(vpart(x)[numero]),
815         valider@(vpart(y)[numero],
816         fin := Ø, pret := 1
817      ENDIF
818      FORMAT@
820         EXTEND activity_invocation.23
821            R activity_invocation = '?' entree_transition:id2 '?'
822                      'CHOIX' '(' es_transition:id3 ')'
823                       delai:id1 'VALIDER' '(' es_transition:id4 ')'
824            MEANS transition_choix(id1, id2, id3, id4)
826      ENDFORMAT
828   ENDtransition_choix

830      "/ Definition de la transition CHOIXP /"

832   STATIC ACTIVITY transition_choixp(d: nnint; W e: control_signal;
833                                W x, y: vector@(control_signal))
834   ASSERT size_dimension(x, 1) = size_dimension(y, 1) ENDASSERT
835   BODY
837      DECLARE fin, occupe: variable(bool, Ø);
838              pret: variable(bool, 1);
839              numero: variable(int, Ø)
840      ENDDECLARE
841      IF pret THEN
842         IF e & ~(/|y) & (/|x) THEN
843            numero := IF vpart(x)[1] = 1 THEN 1
844                      ELSE THE@ i:bint(2, size_dimension(x, 1)) WITH
845                                vpart(x)[i] = 1
846                             & FORALL@ j: bint(1, i-1) IS
847                                          vpart(x)[j] = Ø ENDFOR
848                           ENDTHE
849                       ENDIF
850           pret := Ø, occupe := 1
851         ENDIF
852      ENDIF,
853      wait@(d, occupe, fin),
854      IF fin THEN
855         invalider@(e),
856         invalider@(vpart(x)[numero]),
857         valider@(vpart(y)[numero],
858         fin := Ø, pret := 1
```

```
859      ENDIF
860      FORMAT@
862         EXTEND activity_invocation.24
863            R activity_invocation = '?' entree_transition:id2 '?'
864                     'CHOIXP' '(' es_transition:id3 ')'
865                      delai:id1 'VALIDER' '(' es_transition:id4 ')'
866            MEANS transition_choixp(id1, id2, id3, id4)
868      ENDFORMAT
870   ENDtransition_choixp

872      "/ Definition des predicats composes /"
873      "/ ***********************************/"

875   FORMAT@
877      EXTEND assertion.2
878         E R assertion = fenetre liste_instants_validite 'END'
879      EXTEND assertion.3
880         E R assertion = instant_validite
881      EXTEND liste_instants_validite.1
882         E R liste_instants_validite = instant_validite:i1
883      EXTEND liste_instants_validite.2
884         E R liste_instants_validite = instant_validite:i2 ','
885              liste_instants_validite:i3       MEANS i2 & i3
886      EXTEND liste_instants_validite.3
887         E R liste_instants_validite = expression;bool:i4
888      EXTEND liste_instants_validite.4
889         E R liste_instants_validite = expression;bool:i5 ','
890             liste_instants_validite:i6       MEANS i5 & i6
891      EXTEND instant_validite.1
892         E R instant_validite = instant liste_predicats 'END'
893      EXTEND liste_predicats.1
894         E R liste_predicats = expression;bool
895      EXTEND liste_predicats.2
896         E R liste_predicats = expression;bool:i7 ',' liste_predicats:i8
897             MEANS i7 & i8
898      EXTEND fenetre.1
899         E R fenetre = 'DEPUIS' date:id1 'ALORS'
900             MEANS IF t@ < id1 | id1 = Ø THEN 1 ELSE
901      EXTEND fenetre.2
902         E R fenetre = 'JUSQUA' date:id2 'ALORS'
903             MEANS IF t@ >= id2 & id2 > Ø THEN 1 ELSE
904      EXTEND fenetre.3
905         E R fenetre = 'DEPUIS date:id3 'JUSQUA' date:id4 'ALORS'
906             MEANS IF t@ < id3 | id3 = Ø THEN 1
907                   ELIF t@ >= id4 & id4 > Ø THEN 1 ELSE
908      EXTEND fenetre.4
909         E R fenetre = 'DEPUIS' date:id5 'PENDANT' integer;pint:id6 'ALORS'
910             MEANS IF t@ < id5 | id5 = Ø THEN 1
911                   ELIF t@ > id5 + id6 THEN 1 ELSE
912      EXTEND fenetre.5
913         E R fenetre = 'REPETER' 'DEPUIS' expression;event:e1
914                   'JUSQUA' expression;event:e2 'ALORS'
915             MEANS IF ~entre@(e1, e2) THEN 1 ELSE
916      EXTEND fenetre.6
917         E R fenetre = 'REPETER' 'DEPUIS' expression;event:e3
918                   'PENDANT' integer;pint:id7 'ALORS'
```

```
919             MEANS IF ~suiv@(e3, id7) THEN 1 ELSE
920      EXTEND date.1
921         E R date = integer;nnint
922      EXTEND date.2
923         E R date = expression;event:e4
924             MEANS date@(e4)
925      EXTEND instant.1
926         E R instant = 'EN' date:id8 'ALORS'
927             MEANS IF t@ ~= id8 THEN 1 ELSE
928      EXTEND instant.2
929         E R instant = 'CHAQUE' expression;event:e5 'ALORS'
930             MEANS IF e5 = 'LOW' THEN 1 ELSE
931      EXTEND instant.3
932         E R instant = 'DESQUE' expression;event:e6 'ALORS'
933             MEANS EN lafois(1, e6) ALORS
934      EXTEND instant.4
935         E R instant = 'TOUSLES' integer;pint:id9 'ALORS'
936             MEANS CHAQUE echantillon@(id9) ALORS
938   ENDFORMAT

940   ENDlasso2
```

IV UNE AUTRE METHODE DE DERIVATION

Dans les paragraphes précédents, LASCAR2 et LASSO2 ont été directement définis à partir de BCL comme langage de référence. A ce point l'arbre de dérivation des langages de la famille CONLAN qui sont définis est donné par la figure 1 :

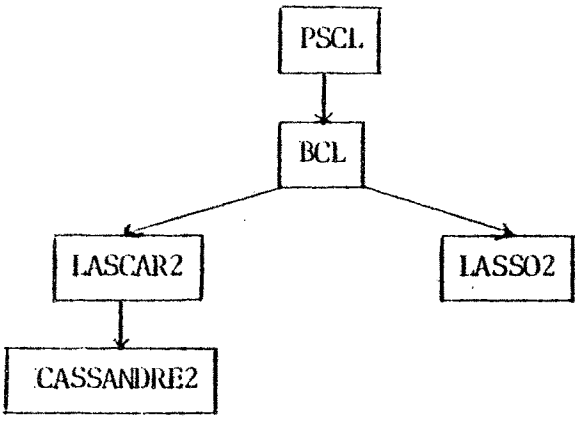


Figure 1

Cette méthode de dérivation, quoique parfaitement correcte et légitime, a l'inconvénient de ne pas mettre clairement en évidence le fait que LASCAR2 et LASSO2 ont en commun un grand nombre de définitions de types et d'opérations. Une optique plus "intégrante" de ces définitions consiste à définir un niveau de langage supplémentaire, comme le montre la figure 2 :

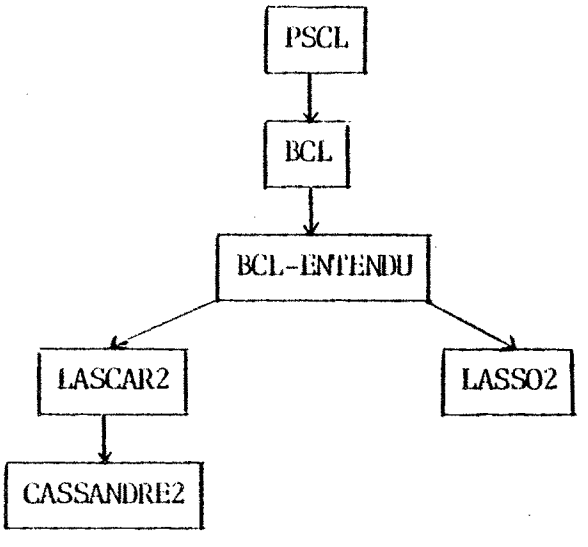


Figure 2

IV.1 BCL-ETENDU

BCL-ETENDU est défini comme un sur-ensemble de BCL, ce qui est indiqué par l'instruction CARRYALL (ligne 12). Il contient toutes les définitions de types utilitaires (sélecteurà, vectorà, non_vectorà) et de fonctions utilitaires (extend_indexer, extend_array_dim, normalize_vectorà, repeat_eval) communes à LASCAR2 et LASSO2, mais dont la connaissance n'était pas donnée aux utilisateurs.

Ces types et fonctions ne sont pas destinés à être transmis aux autres langages, mais peuvent être utilisés pour leur définition.

BCL-ENTENDU contient aussi les définitions des opérations de réduction, de conversion entre entiers et vecteurs booléens, de valeur absolue, et toutes les extensions des opérateurs scalaires à des opérateurs tableaux. Toutes ces opérations, fournies aux utilisateurs, et pour lesquelles un format infixé est défini, sont destinées à être transportées vers les langages dérivés de BCL-ETENDU.

IV.2 LASSO2 et LASCAR2

Les définitions de ces deux langages sont maintenant considérablement raccourcies. Toutes les définitions de BCL-ETENDU sont ôtées de l'un et l'autre. Les noms des opérations non utilitaires de BCL-ETENDU apparaissent dans la liste des opérations transportées.

La comparaison des listes CARRY de LASSO2 et LASCAR2 fait apparaître plus nettement au lecteur l'ensemble des types et opérations communs aux deux langages. Les définitions de types et opérations présentes dans chacun des deux segments soulignent au contraire ce qui les distingue.

Il est à remarquer que l'adoption de cette nouvelle méthode de dérivation n'a aucune influence sur la définition de CASSANDRE2.

IV.3 Définitions formelles

Nous donnons ci-dessous la définition complète de BCL-ETENDU, et le début des définition de LASSO2 et LASCAR2 prenant BCL-ETENDU comme langage de référence.

```
5                                   EXTENSION DE BASE CONLAN

9   REFLAN bcl
10  CONLAN bcl_etendu  BODY

12  CARRYALL

14  "/*************************************************************/"
15  "/ Fonction valeur absolue, non inclue dans le type int /"
16  "/*************************************************************/"

18  FUNCTION abs (x: int) : nnint
19  RETURN
21     IF x >= 0 THEN x ELSE -x ENDIF
22     FORMAT@ EXTEND exp8.5
23              R exp8 = 'ABS' exp8:id1 MEANS abs(id1)
24     ENDFORMAT
26  ENDabs


30  "/*************************************************************************/"
31  "/ 2 fonctions utilitaires permettant d'etendre un index
32     et un ensemble de dimensions  par une premiere dimension /"
33  "/*************************************************************************/"

35  FUNCTION extend_indexer(x: range; y: indexer) : indexer
36  RETURN
38     THE@ z: indexer WITH
40        size@(z) = size@(y) + 1  &
41        tail(z) = y              &
42        size@(z[1]) = size@(x)   &
43        FORALL@ i: bint(1, size@(x)) IS
44              z[1][i] = x[i] ENDFOR
46     ENDTHE
48  ENDextend_indexer


51  FUNCTION extend_array_dim(x: range; y: array_dimension): array_dimension
52  RETURN
54     THE@ z: array_dimension WITH
56        size@(z) = size@(y) + 1  &
57        tail(z) = y              &
58        z [1] = x
60     ENDTHE
62  ENDextend_array_dim

64  "/*************************************************************************/"
65  "/ types utilitaires, non connus de l'utilisateur /"
66  "/*************************************************************************/"

68  "/ ensemble des index permettant de selectionner un element de tableau /"

70  SUBTYPE selecteur@ (x: array(u: any@))
71  BODY
72     ALL z : degenerate_indexer WITH  isin(z, dpart(x)) ENDALL
73  ENDselecteur@
```

```
"/ differenciation entre vecteur ligne et autre tableau ayant au moins
   2 dimensions ; cette distinction sert a la construction du langage,
   elle est trensparente pour l'utilisateur /"

SUBTYPE vector@(u: any@)  BODY
   ALL x: array(u) WITH size@(dpart(x)) = 1 |
                        size_array(tail(dpart(x))) = 1 ENDALL
ENDvector@

SUBTYPE nonvector@(u: any@)  BODY
   ALL x: array(u) WITH size@(dpart(x)) > 1 &
                        size_array(tail(dpart(x))) > 1 ENDALL
ENDnonvector@

"/ normalisation d'un vecteur ligne:
   toutes les dimensions degenerees, sauf la lere, sont supprimees
   la borne inf. de la lere dimension est mise a 1
   la borne sup. de la lere dimension est mise a la taille du vecteur  /"

FUNCTION normalize_vector@ (x: vector@(u: any@)) : vector@(u)
   RETURN THE@ y: vector@(u) WITH
          dpart(y) = ( 1 : size_dimension(x,1) ) &
          vpart(y) = vpart(x)
   ENDTHE
ENDnormalize_vector@


"/*************************************************************************/"
"/ Definition de l'operation de reduction d'un vecteur ou d'un tableau
    par rapport à un operateur binaire:
        Un vecteur ligne est transforme en un scalaire
        Un autre tableau est transforme en un tableau
          ayant une dimension de moins /"
"/*************************************************************************/"

PRIVATE FUNCTION repeat_eval (u: any@; x: tytuple@(u);
                                       f: FUNCTION(u, u) :u) : u
   RETURN IF size@(x) = 1 THEN x[1]
          ELSE f( x[1], repeat_eval(u, tail(x), f) ) ENDIF
ENDrepeat_eval

"/ Un operateur unaire est defini pour le format d'appel de la reduction
     par rapport au 'et' et au 'ou' booleens /"

FUNCTION reductionl(u: any@; x: vector@(u); f: FUNCTION(u, u):u ) : u
   RETURN  repeat_eval(vpart(x))
   FORMAT@
      EXTEND exp8.6
         exp8 = '/|' exp8:idl MEANS reductionl(bool, idl, |)
      EXTEND exp8.7
         exp8 = '/&' exp8:idl MEANS reductionl(bool, idl, &)
   ENDFORMAT
ENDreductionl
```

```
140  FUNCTION reduction2(u: any@; x: nonvector@(u); f: FUNCTION(u,u):u): array(u)
142     RETURN normalize (THE@ y: array(u) WITH
144             dpart(y) = tail(dpart(x)) &
145             FORALL@ p: selecteur@(y) IS
146                 y[p] = reduction1(u, x[ extend_indexer(dpart(x)[1], p) ], f)
147             ENDFOR ENDTHE)
149     FORMAT@
151        EXTEND exp8.6 MEANS reduction2(bool, id1, |)
152        EXTEND exp8.7 MEANS reduction2(bool, id1, &)
154     ENDFORMAT
156  ENDreduction2


159  "/*************************************************************************/"
160  "/ Extension des operateurs sur les scalaires à des operateurs sur tableaux
161     Ces extensions font appel aux fonctions generiques definies sur le
162     type array, qui evitent une redefinition de chaque operation.
163     Seul le format d'appel doit etre precisé.
164     On conserve la meme notation infixee, et les memes priorites  /"
165  "/*************************************************************************/"

167     "/ operations binaires internes sur les booleens /"

169  FUNCTION array_dyadic_bool(x, y: array(bool); f: FUNCTION(bool,bool):bool)
170                             : array(bool)
172     RETURN fbin@(x, y, f)
173     FORMAT@
175        EXTEND exp1.2 MEANS array_dyadic_bool (id1, id2, |)
176        EXTEND exp2.2 MEANS array_dyadic_bool (id1, id2, xor)
177        EXTEND exp2.3 MEANS array_dyadic_bool (id1, id2, eqv)
178        EXTEND exp3.2 MEANS array_dyadic_bool (id1, id2, &)
179        EXTEND exp4.4 MEANS array_dyadic_bool (id1, id2, <)
180        EXTEND exp4.5 MEANS array_dyadic_bool (id1, id2, =<)
181        EXTEND exp4.6 MEANS array_dyadic_bool (id1, id2, >)
182        EXTEND exp4.7 MEANS array_dyadic_bool (id1, id2, >=)
184     ENDFORMAT
186  ENDarray_dyadic_bool

188     "/ operations binaires internes sur les entiers /"

190  FUNCTION array_dyadic_int (x, y: array(int); f: FUNCTION(int, int):int)
191                                : array(int)
193     RETURN fbin@(x, y, f)
194     FORMAT@
196        EXTEND exp5.2 MEANS array_dyadic_int (id1, id2, +)
197        EXTEND exp5.3 MEANS array_dyadic_int (id1, id2, -)
198        EXTEND exp6.2 MEANS array_dyadic_int (id1, id2, *)
199        EXTEND exp6.3 MEANS array_dyadic_int (id1, id2, /)
200        EXTEND exp6.4 MEANS array_dyadic_int (id1, id2, MOD)
201        EXTEND exp7.2 MEANS array_dyadic_int (id1, id2, ^)
203     ENDFORMAT
205  ENDarray_dyadic_int

207     "/ operations unaires internes /"
```

```
209   FUNCTION array_monadic(u: any@; y: array(u); f: FUNCTION(u):u): array(u)
211      RETURN fmon@(y, f)
212      FORMAT@
214         EXTEND exp8.2 MEANS array_monadic(int, idl, +)
215         EXTEND exp8.3 MEANS array_monadic(int, idl, -)
216         EXTEND exp8.4 MEANS array_monadic(bool, idl, ~)
218      ENDFORMAT
220   ENDarray_monadic

222      "/ operations de comparaison sur les entiers /"

224   FUNCTION array_compare_int(x,y:array(int); f: FUNCTION(int,int):bool)
225                                    : array(bool)
227      RETURN fbool@(x, y, f)
228      FORMAT@
230         EXTEND exp4.4 MEANS array_compare_int(idl, id2, <)
231         EXTEND exp4.5 MEANS array_compare_int(idl, id2, =<)
232         EXTEND exp4.6 MEANS array_compare_int(idl, id2, >)
233         EXTEND exp4.7 MEANS array_compare_int(idl, id2, >=)
235      ENDFORMAT
237   ENDarray_compare_int

239      "/ operations unaires externes /"

241   FUNCTION ext_monadic(u,v: any@; x: array(u); f: FUNCTION(u):v): array(v)
243      RETURN
245         normalize ( THE@ z: array(v) WITH
246            dpart(z) = dpart(x)  &
247            FORALL@ p: selecteur@(x) IS z[p] = f (x[p])  ENDFOR
248         ENDTHE
250      FORMAT@ EXTEND exp8.5 MEANS ext_monadic(int, nnint, idl, abs)
251      ENDFORMAT
253   ENDext_monadic

255   "/*************************************************************************
256      Definition du contenu d'un tableau de porteuses
257   *************************************************************************/"

259   INTERPRETER@ FUNCTION content@(x: array(variable(t:any@,init:t))): array(t)
260      RETURN ext_monadic(variable(t,init), t, x, variable(t,init).content@)
261   ENDcontent@

263   INTERPRETER@ FUNCTION content@(x: array(terminal(t:any@,init:t))): array(t)
264      RETURN ext_monadic(terminal(t,init), t, x, terminal(t,init).content@)
265   ENDcontent@

267   INTERPRETER@ FUNCTION content@(x: array(rtvariable(t:any@,init:t))): array(t)
268      RETURN ext_monadic(rtvariable(t,init), t, x, rtvariable(t,init).content@)
269   ENDcontent@

271   "/*************************************************************************
272      Operations de modification de valeur des porteuses scalaires
273      etendues aux tableaux de porteuses.
274   *************************************************************************/"
```

```
276   FORMAT@
278      EXTEND activity_invocation.2 MEANS abin@(id1, id2, connect)
279      EXTEND activity_invocation.3 MEANS abin@(id1, id2, assign)
280      EXTEND activity_invocation.4 MEANS abin@(id1, id2, transfer)
282   ENDFORMAT

284   "/ fin de l'extension des operations scalaires aux tableaux /"

286   "/*************************************************************************
287      Operations de codage et de decodage des entiers en vecteurs booleens
289   *************************************************************************/"

290   "/ conversions de type  de 0 et 1 /"

292   FUNCTION valn1 (x: bool) : nnint
293   RETURN
295      IF x = 0 THEN 0 ELSE 1 ENDIF
296      FORMAT@ EXTEND exp8.9
297               R exp8 = '$' exp8:id1 MEANS valn1(id1)
298      ENDFORMAT
300   ENDvaln1


303   FUNCTION valb1 (x: nnint) : bool
304   RETURN
305      IF x = 0 THEN 0 ELIF x = 1 THEN 1 ELSE error@ ENDIF
306   ENDvalb1

308   "/ traduction de vecteur booleen en entier positif /"

310   FUNCTION valn2(x: vector@(bool)) : nnint
311   BODY
313      DECLARE nbel : variable(int, 0) ENDDECLARE

315      nbel := size_dimension(x, 1)
316      RETURN
318         IF nbel = 1 THEN valn1(normalize_vector@(x)[1])
319         ELSE valn1(normalize_vector@(x)[1])*2^(nbel-1)
320              + valn2(normalize_vector@(x)[2:nbel])
321         ENDIF
322         FORMAT@ EXTEND exp8.9 MEANS valn2(id1) ENDFORMAT
324   ENDvaln2

326   FUNCTION valn3 (x: nonvector@(bool)) : array(nnint)
327   RETURN
329      normalize ( THE@ y: array(nnint) WITH
331         dpart(y) = tail(dpart(x)) &
332         FORALL@ p: selecteur@(y) IS
334            y[p] = valn2 ( x[extend_indexer(dpart(x)[1], p)] )
336         ENDFOR
338      ENDTHE )
339      FORMAT@ EXTEND exp8.9 MEANS valn3(id1) ENDFORMAT
341   ENDvaln3

343   "/ traduction de vecteur booleen en entier positif ou negatif.
344      codage en complement a 2, le premier bit etant un bit de signe /"
```

```
346   FUNCTION valn4 ( x: vector@(bool)) : int
347   BODY
349      DECLARE nbel : variable(int, Ø) ENDDECLARE
350      nbel := size_dimension(x, 1)
351      RETURN
352         IF vpart(x)[1] = Ø THEN valn2(x) ELSE valn2(x) -2^nbel ENDIF
353      FORMAT@ EXTEND exp8.1Ø
354              R exp8 = 'INTVAL' exp8:id1 MEANS valn4(id1)
355      ENDFORMAT
357   ENDvaln4


360   FUNCTION valn5 (x: nonvector@(bool)) : array(int)
361   RETURN
363      normalize ( THE@ y : array(int) WITH
365         dpart(y) = tail(dpart(x))  &
366         FORALL@ p : selecteur@(y) IS
367            y[p] = valn4 ( x[extend_indexer(dpart(x)[1], p)] )
368         ENDFOR
370      ENDTHE )
371      FORMAT@ EXTEND exp8.1Ø MEANS valn5(id1) ENDFORMAT
373   ENDvaln5

375   "/ traduction d'un entier positif en vecteur booleen de longueur i /"
376   "/ si la longueur est insuffisante, troncature et emission d'un message /"

378   FUNCTION valb2 (x:nnint; i:pint) : vector@(bool)
379   BODY
381      DECLARE y[i:1] : variable(bool, Ø) ENDDECLARE

383      IF x >= 2^i THEN warning@ ENDIF
384      OVER j FROM 1 TO i REPEAT
385                  y[j] := valb1( (x MOD 2^j) / 2^(j-1) ) ENDOVER
386      RETURN   content@(y)
387      FORMAT@ EXTEND exp8.11
388              R exp8 = '!' '/' constant_denotation:id2 '/' exp8:id1
389              MEANS valb2(id1, id2)
390      ENDFORMAT
392   ENDvalb2

394   FUNCTION  valb3 (x: array(nnint); i: pint) ; array(bool)
395   RETURN
397      normalize (  THE@ y: array(bool) WITH
399         dpart(y) = extend_array_dim(1:i, dpart(x))  &
400         FORALL@ p: selecteur@(x) IS
401            y[extend_indexer(1:i, p)] = valb2(x[p], i)
402         ENDFOR
404      ENDTHE )
405      FORMAT@ EXTEND exp8.11 MEANS valb3(id1, id2) ENDFORMAT
407   ENDvalb3

409   "/ traduction d'un entier relatif en vecteur booleen de longueur i.
410      si la longueur est insuffisante, troncature et emission d'un message.
411      codage en complement a 2, le premier bit etant un bit de signe /"
```

```
413   FUNCTION valb4 (x: int; i: pint) : vector@(bool)
414   BODY
416      IF x >= 2^(i-1) | x < -2^(i-1) THEN warning@ ENDIF
417      RETURN
419         IF x >= Ø THEN valb2(x, i) ELSE valb2(x MOD 2^i, i) ENDIF
420      FORMAT@ EXTEND exp8.12
421              R exp8 = 'BINVAL' '/' constant_denotation:id2 '/' exp8:id1
422              MEANS valb4(id1, id2)
423      ENDFORMAT
425   ENDvalb4

427   FUNCTION valb5 (x: array(int); i: pint) : array(bool)
428   RETURN
430      normalize ( THE@ y: array(bool) WITH
432         dpart(y) = extend_array_dim(1:i, dpart(x))  &
433         FORALL@ p: selecteur@(x) IS
434            y[extend_indexer(1:i, p)] = valb4(x[p], i)
437   ENDTHE )
438   FORMAT@ EXTEND exp8.12 MEANS valb5(id1, id2) ENDFORMAT
440   ENDvalb5


443   ENDbcl_etendu
```

```
  5                  DEFINITION DU LANGAGE LASCAR EN TERMES DE CONLAN

  9   REFLAN bcl_etendu
 10   CONLAN lascar2  BODY

 12   CARRY bool, int, nnint, pint, range, array_dimension, indexer,
 13         fixed_array, cs_signal@, signal@ ,
 14        abs, reduction1, reduction2, array_dyadic_bool, array_dyadic_int,
 15        array_monadic, array_compare_int, ext_monadic, content@,
 16        valn1, valn2, valn3, valn4, valn5, valb1, valb2, valb3, valb4, valb5
 17   ENDCARRY

 19   "/*****************************************************************************
 20     Definition des types de valeurs et des types de porteuses de LASCAR qui
 21      n'existent pas dans BASE CONLAN
 22   *****************************************************************************"/

 24   TYPE impulsion
 25      .....
 26   ENDimpulsion

 28     "/ Definition des etats de l'automate de controle  /"

 30   TYPE valeur_etat
 31      .....
 32   ENDvaleur_etat

 34      "/ Definition des registres charges sous condition d'impulsion /"

 36   TYPE clocked_reg(x: any@; init:x)
 37      .....
 38   ENDclocked_reg
```

```
40  "/ Definition des noms des types de porteuses primitives dans LASCAR.
41     La valeur initiale par defaut en debut de simulation est precisee. /"

43  SUBTYPE horloge BODY  terminal(impulsion, 'LOW') ENDhorloge

45  SUBTYPE registre BODY clocked_reg(bool, Ø) ENDregistre

47  SUBTYPE etat BODY clocked_reg(valeur_etat, error@) ENDetat

49  SUBTYPE signal BODY terminal(bool,Ø) ENDsignal

51  SUBTYPE entsig BODY terminal(int, Ø) ENDentsig

53  SUBTYPE entreg BODY clocked_reg(int, Ø) ENDentreg

55  TYPE compteur
56     .....
57  ENDcompteur


61  "/*************************************************************************
62     Format d'appel des chargements de registre, et des operations
63     sur les compteurs, qui dependent d'une condition impulsionnelle
64  *************************************************************************/"

66  FORMAT@
68     EXTEND activity_invocation.11
69        .....
71  ENDFORMAT

73  "/*************************************/"
74  "/ Fonctions non inclues dans les types /"
75  "/*************************************/"

77  FUNCTION reg_rise (x: registre) : impulsion
78     .....
79  ENDreg_rise

81  FUNCTION sig_rise (x: signal) : impulsion
82     .....
83  ENDsig_rise


87  "/*************************************************************************
88     Definition du format d'ecriture des instructions sous etat
89     de l'automate de controle
Ø0  *************************************************************************/"

92  FORMAT@
94     EXTEND conditional_invocation.3
95        .....
97  ENDFORMAT

99  ENDlascar2
```

```
5                    DEFINITION FORMELLE DU LANGAGE LASSO EN CONLAN

9   REFLAN bcl_etendu
10  CONLAN lasso2  BODY

12  CARRY bool, int, nnint, pint, range, array_dimension, indexer,
13        fixed_array, cs_signal@, signal@ , variable,
14        string, field_descriptor, field_descriptor_list, record,
15        abs, reduction1, reduction2, array_dyadic_bool, array_dyadic_int,
16        array_monadic, array_compare_int, ext_monadic, content@,
17        valn1, valn2, valn3, valn4, valn5, valb1, valb2, valb3, valb4, valb5
18  ENDCARRY

20  "/*************************************************************************
21    Definition des types de valeurs et des types de porteuses de LASSO qui
22     n'existent pas dans BASE CONLAN
23  *************************************************************************"/

25     "/ Un evenement est positionne lors du changement de valeur d'un
26        signal de controle /"
27  TYPE event
28     .....
29  ENDevent

31    "/ Definition des signaux du graphe de controle  /"

33  TYPE control_signal
34     .....
35  ENDcontrol_signal

37  SUBTYPE nomsig BODY  variable(control_signal, error@) ENDnomsig


40  "/ ************************************************************************
41     Definition des primitives du graphe de controle,
42     ainsi que de leur format d'ecriture
43  ************************************************************************/"

45     "/ Notion d'attente, utilisee dans toutes les primitives
46        à duree de traversee non nulle /"

48  STATIC ACTIVITY wait@ (d: nnint; W a, b: variable(bool, 0))
49     .....
50  ENDwait@

52     "/ Ecriture des listes de signaux en entree et en sortie de transition/"

54  FORMAT@
56     EXTEND entree_transition.1
58     .....
59  ENDFORMAT

61  STATIC ACTIVITY transition_et (d:nnint; W x, y: vector@(control_signal))
62     .....
63  ENDtransition_et
```

```
65   STATIC ACTIVITY transition_selon(d:nnint; p: bool;
66                                W x, y[1:2]: control_signal)
67      .....
68   ENDtransition_selon

70   STATIC ACTIVITY transition_index(d:nnint; i: int; ATT m, n: int;
71                                W x, y[1:2]: control_signal)
72      .....
73   ENDtransition_index

75   STATIC ACTIVITY transition_choix(d: nnint; W e: control_signal;
76                                W x, y: vector@(control_signal))
77      .....
78   ENDtransition_choix

80   STATIC ACTIVITY transition_choixp(d: nnint; W e: control_signal;
81                                W x, y: vector@(control_signal))
82      .....
83   ENDtransition_choixp

85   ENDlasso2
```

CONCLUSION

Au terme d'une recherche de plusieurs années, une idée directrice unique a sous-tendu tous nos travaux : définir un ensemble cohérent de langages de description de systèmes logiques, et mettre en oeuvre les logiciels de traitement associés, afin de permettre aux concepteurs de représenter et de vérifier tout ou partie d'un système à différents niveaux de détail. L'ambition de cet objectif explique que nous n'ayons pas directement attaqué le problème dans toute sa complexité. Tel le randonneur qui s'entraîne pour le sommet qu'il vise en commençant par des courses moins ardues, nous avons d'abord étendu un langage existant pour en élargir le domaine d'application (définition de LASCAR à partir de CASSANDRE), puis développé un nouveau langage pour couvrir un niveau de description alors peu abordé (LASSO) et pour lequel l'expérimentation de nouveaux concepts s'est révélée utile, avant de parvenir à dégager, au sein du projet CONLAN, les principes fondamentaux à partir desquels les langages de description de systèmes logiques peuvent être définis.

L'initiateur du projet CONLAN, le Professeur Lipovski, avait voulu que le groupe de travail fît un langage multi-niveaux pour servir de moyen de communication à la communauté des concepteurs, une espèce d'espéranto de la description des systèmes digitaux. Devant la nécessité de procéder au préalable à une réflexion théorique pour établir le noyau des notions primitives communes à tous les niveaux, le groupe de travail dévia le projet de ce but immédiat. CONLAN aujourd'hui est donc moins un langage qu'une base syntaxique et sémantique accompagnée d'une méthode de définition formelle et d'un modèle d'interprétation. C'est à notre avis un résultat beaucoup plus significatif que celui qui était initialement visé, et pour lequel on pouvait craindre le même sort que celui qui échut à l'espéranto, les mêmes causes produisant souvent les mêmes effets.

L'étude que nous avons menée, et les conclusions auquelles nous avons abouti ont déjà eu des retombées directes dans notre Laboratoire, à la fois dans la recherche et dans l'enseignement.

Le système intégré d'aide à la conception des circuits logiques, dont nous avons montré la nécessité dans l'introduction de cette thèse, est un projet en cours de réalisation au sein de l'équipe "Communication Graphique et Méthodologie de la CAO". Notre contribution à ce projet a consisté précisément à

définir le langage de description multi-niveaux de ce système. Ce langage n'est pas du "pur" CONLAN, dont la syntaxe a été jugée trop lourde par les utilisateurs de nos précédents outils, et insuffisamment "naturelle" (c'est-à-dire trop différente des conventions d'écriture dont ils avaient l'habitude). A ces différences syntaxiques doit être ajouté le fait que le système intégré couvrira aussi le niveau électrique, et que la grammaire du langage de description a été étendue pour permettre d'exprimer des modèles à ce niveau. Pour le reste, tous les principaux concepts de CONLAN, et tous les segments et constructeurs destinés aux utilisateurs sont présents dans le langage. La définition en CONLAN de LASCAR2 et de LASSO2 qui occupe la cinquième partie de cet ouvrage est la définition sémantique de deux des niveaux couverts dans le système intégré. D'autres niveaux suivront,en particulier pour permettre la manipulation d'algèbres à plus de 3 valeurs dans des descriptions de niveau réseau de portes : certaines seront pré-définies dans le système, mais l'utilisateur aura la possibilité d'ajouter ses propres tables de vérité, par des définitions de types.

Les segments DESCRIPTION, FUNCTION et ACTIVITY se retrouvent aussi tels quels dans ce langage; toutefois, le premier prototype limitera l'utilisateur à des définitions d'opérations statiques. Enfin, le système possèdera l'une des propriétés qui nous semblent les plus fondamentales : fournir des outils de vérification capables de traiter des descriptions multi-niveaux.

Notre approche des langages de description, partie d'une réalisation utilisée en milieu industriel, trouve donc son application dans une nouvelle réalisation beaucoup plus ambitieuse, destinée elle aussi à être confiée aux concepteurs de systèmes logiques, tant dans les laboratoires publics que dans diverses entreprises. Il eut été regrettable que cette réflexion méthodologique fît l'objet de séminaires chez les professionnels, et non d'un enseignement auprès des étudiants formés à l'I.M.A.G. C'est donc en nous appuyant sur l'expérience acquise que nous avons créé un cours sur les techniques de modélisation des systèmes logiques. Nous pensons que cet effort devrait non seulement être poursuivi, mais élargi afin de s'intégrer à l'enseignement de la structure des ordinateurs, comme cela se fait dans diverses universités étrangères. Dans notre esprit, il s'agit de mettre en place une nouvelle pédagogie s'appuyant sur le symbolisme des langages de description : l'enseignant disposerait d'un moyen extrêmement précis de présentation des différentes architectures de machines digitales, dont il pourrait rendre plus perceptibles les propriétés et les performances, en montrant la simulation de leur

fonctionnement; l'expérimentation logicielle offrant l'avantage d'un temps de réponse très court et d'un coût raisonnable, l'étudiant apprendrait son métier de concepteur par la pratique de la conception sur des études de cas plus nombreuses, et à des niveaux plus variés. Nous espérons que les moyens nous seront donnés d'intégrer à la formation initiale des futurs praticiens de l'informatique une technique dont l'intérêt n'est aujourd'hui plus à démontrer.

BIBLIOGRAPHIE

[AFM.79] Anlauff H., Funk P., Meinen P.
'PHPL - A New Computer Hardware Description Language'
Proc. 4th International Symposium on Computer Hardware Description Languages, Palo Alto, October 8-9, 1979

[And.79] Andrews Dorothy
'Using Executable Assertions for Testing and Fault Tolerance'
Proc. FTCS-9, 1979

[AsW.77] Ashcroft E. A., Wadge W. W.
'LUCID: a Nonprocedural Language with Iteration'
CACM, Vol. 20, Nr. 7, July 1977

[BaA.78] Bayegan H. M., Aas E.
'An Integrated System for Interactive Editing of Schematics, Logic Simulation and PCB Layout Design'
in [DAC.78]

[Bac.78] Backus John
'Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs'
CACM, Vol.21, Nr. 8, August 1978

[Bae.73] Baer J. L.
'A Survey of some Theoretical Aspects of Multiprocessing'
Computing Surveys, Vol. 5, Nr 1, March 1973

[BaL.79] Bailly J.L., Leroux J.J.
'Definition et implementation d'un langage de prise de mesures automatiques sur le langage LASSO'
Projet de 3eme annee E.N.S.I.M.A.G., Grenoble, Juin 1979

[BaN.78] Barbacci Mario R., Nagle Andrew W.
'An ISPS Simulator'
Dept. of Computer Science, Carnegie-Mellon University, March 1978

[Bar.75] Barbacci Mario
'A Comparison of Register Transfer Languages for Describing Computers and Digital Systems'
IEEE Trans. on Computers, Vol. C24, Nr 2, February 1975

[BaS.75] Barbacci Mario R., Siewiorek Daniel P.
'The CMU RT-CAD System: An Innovative Approach to Computer Aided Design'
Dept. of Computer Science, Carnegie-Mellon University, June 1975

[BBC.78] Barbacci M. R., Barnes G. E., Cattell R. G., Siewiorek D. P.
'The ISPS Computer Description Language'
Dept. of Computer Science, Carnegie-Mellon University, March 1978

[BBG.77] Borrione D., Bressy Y., Grabowiecki J.F., Mermet J.
'Methodes et Langages pour la Modelisation et l'Aide a la Conception des Systemes Logiques'
Proc. AFCET Congress : 'Modelisation et Maitrise des systemes techniques, economiques et sociaux', Versailles, November 1977.

[BDF.75] Bressy Y., David B., Fantino Y., Mermet J.
'A Hardware Compiler for Interactive Realization of Logical Systems described in CASSANDRE'
Proc. International Symposium on Computer Hardware Description Languages and their Applications, New York, September 1975.

[BeJ.77] Bert D., Jacquet P.
'Generic Abstract Data Types'
Proc. of the 5th Annual III Conference, Guidel, France, May 1977, pp. 71-82.

[BeN.70] Bell C. G., Newell A.
'The PMS and ISP descriptive systems for computer structures'
Spring Joint Computer Conference, 1970, pp 351-374

[Ber.79] Bert Didier
'La Programmation Generique'
These d'Etat, U.S.M.G., 26 Juin 1979

[BKS.74] Becker M., Klar R., Spies P.
'The Erlangen Computer Design Language ERES'
Proc. Workshop on Computer Hardware Description Languages, Darmstadt, July 31-August 2, 1974

[BMU.80] Bressy Y., Mermet J., Uvietta P.
'Mecanismes de simulation discrete et continue lies aux modeles de systemes dynamiques'
Rapport de Recherche IMAG No. 206, Juillet 1980

[BoG.79] Borrione D., Grabowiecki J.F.
'Informal Introduction to LASSO : a Language for Asynchronous Systems Specification and Simulation'
Proc. EURO IFIP 79, London, September 1979

[BoG.80] Borrione D., Grabowiecki J.F.
'Realisation d'un prototype du compilateur et du simulateur du langage LASSO:
Tome 1: Le Compilateur Prototype
Tome 2: Le Simulateur Prototype
Tome 3: Manuel d'Utilisation '
Rapport final du contrat SESORI no. 78067, Aout 1980

[Bor.75] Borrione D.
'LASCAR : a Language for Simulation of Computer Architecture'
Proc. International Symposium on Computer Hardware Description Languages and their Applications, New York, September 1975.

[Bor.76] Borrione D.
'LASCAR : un langage pour la simulation et l'evaluation des architectures d'ordinateurs'
These de troisieme Cycle, INPG, Grenoble, France, April 16, 1976.

[Bor.78] Borrione D.
'Definition d'un langage prefixe utilisable pour exprimer les specifications temporelles dans LASSO'
Note interne equipe CAO, Novembre 1978

[BoV.77] Bobas A., Valihora J.
'A Design Automation System for PCB Assemblies'
in [DAC.77]

[Bra.75] Bradshaw Franklyn T.
'Directed Graph Models for Hardware/Software Design'
Proc. International Symposium on Computer Hardware Description Languages and their Applications, New York, Sept. 3-5 1975

[Bre.75] Bressy Y.
'Version Asynchrone du systeme CASSANDRE'
Seminaire IMAG, Grenoble, 7 Novembre 1975

[Bri.75] Brinch Hansen Per
'The Programming Language Concurrent Pascal'
IEEE Trans. on Software Engineering, Vol. 1, No. 2, 1975

[Bri.76] Brinch Hansen Per
'The Solo Operating System:
- A Concurrent Pascal Program
- Job Interface
- Processes, Monitors and Classes '
Software - Practice and Experience, Vol. 6, No. 2, 1976

[CAD.1] Commission des Communautes Europeennes
Proc. Symposium on Computer Aided Design of Digital Electronic Circuits and Systems
Bruxelles, Belgique, 27-29 Novembre 1978

[CAD.2] Equipe Communication Graphique et Methodologie de la CAO
'Systeme de CAO Integre pour Ensembles Electroniques et Logiques'
Rapport, IMAG, Grenoble, Fevrier 1981

[CHDL.1]
Special Issue on 'Hardware Description Languages'
Computer, Vol. 7, Nr. 12, December 1974

[CHDL.2]
Special Issue on 'Hardware Description Language Applications'
Computer, Vol. 10, No. 6, June 1977

[CHDL.3]
Proc. International Workshop on Computer Hardware Description Languages,
Technische Hochschule Darmstadt, R.F.A., July 31-August 2, 1974

[CHDL.4]
Proc. International Symposium on Computer Hardware Description Languages and their Applications,
New York City, September 3-5, 1975

[CHDL.5]
Proc. Fourth International Symposium on Computer Hardware Description Languages
Palo Alto, California, October 8-9, 1979

[Chu.65] Chu Y.
'An ALGOL-like Computer Design Language'
CACM, Vol. 8, No. 10, October 1965

[Chu.69] Chu Yaohan
'A Computer Design Language'
Ecole d'ete de l'OTAN, Hyeres, Aout 1969

[Chu.72] Chu Yaohan
'Computer Organization and Microprogramming'
Ed. Prentice Hall Inc., Englewood Cliffs, N.J. (USA), 1972

[Chu.74] Chu Y.
'Why Do We Need Computer Hardware Description Languages?'
in [CHDL.1]

[CJB.79] Carter W. C., Joyner W. H., Brand D.
'Symbolic Simulation for Correct Machine Design'
in [DAC.79]

[CoV.80] Cory W. E., vanCleemput W. M.
'Developments in Verification of Design Correctness'
in [DAC.80]

[DAC.75] ACM-IEEE
Proc. 12th Design Automation Conference
Boston, Ma, June 23-25, 1975. IEEE Catalog No. 75CH0980-3C

[DAC.77] ACM-IEEE
Proc. 14th Design Automation Conference
New Orleans, Louisiana, June 20-22, 1977. IEEE Catalog No. 77CH1216-1C

[DAC.78] ACM-IEEE
Proc. 15th Design Automation Conference
Las Vegas, Nevada, June 19-21, 1978. IEEE Catalog No. 78CH1363-1C

[DAC.79] ACM-IEEE
Proc. 16th Design Automation Conference
San Diego, Ca, June 25-27, 1979. IEEE Catalog No. 79CH1427-4C

[DAC.80] ACM-IEEE
Proc. 17th Design Automation Conference
Minneapolis, Minnesota, June 23-25, 1980, IEEE Catalog No. 80CH1550-3

[Dar.79] Darringer J. A.
'The Application of Program Verification Techniques to Hardware Verification'
in [DAC.79]

[Dar.80] Darringer J. A.
'A New Look at Logic Synthesis'
International Workshop Problem Areas in Digital Systems Description and Design Tools, La Baule, May 1980

[DeK.74] DeRemer Frank, Kron Hans
'Programming in the large versus Programming in the small'
IFIP WG 2.4, working paper, September 1974

[Die.78] Dietmeyer Donald L.
'Logic Design of Digital Systems'
Second Edition, Ed. Allyn and Bacon Inc., Boston (USA), 1978

[DMN.68] Dahl D., Myhrhaug B., Nygaard K.
'The SIMULA 67 Common Base Language'
Norwegian Computing Center, Oslo, 1968.

[Don.79] Donzeau-Gouge Veronique
'Denotational Definition of Properties of Program Computations'
IRIA, Rapport de Recherche Nr. 349, Avril 1979

[Dub.79] Dubois D.
'Evaluation du langage LASSO pour la description fonctionnelle de circuits integres'
Rapport de Recherche IMAG No. 180, Decembre 1979

[DuD.68] Duley James R., Dietmeyer Donald D.
'A Digital System Design Language (DDL)'
IEEE Trans. on Computers, Vol C17, No. 9, September 1968

[EGO.77] Evangelisti C. J., Goertzel G., Ofek H.
'Designing with LCD: Language for Computer Design'
in [DAC.77]

[Est.78] Estrin G.
'A Methodology for Design of Digital Systems - Supported by SARA at the age of One'
Proc. National Computer Conference, 1978, pp. 313-324

[Fab.74] Fabry R. S.
'Capability-based Adressing'
CACM, Vol.17, Nr. 7, July 1974

[Flo.75] Flon L.
'Program Design with Abstract Data Types'
Department of Computer Science, Carnegie-Mellon University, June 1975

[FrS.79] Frankel Robert E., Simoliar Stephen W.
'Beyond Register Transfer: an Algebraic Approach for Architectural Description'
Proc. 4th International Symposium on Computer Hardware Description Languages, Palo Alto, October 8-9, 1979

[Gou.75] Gould I. H.
'A Multi-level Digital Simulator'
Proc. SIMULATION 75, Zurich, June 1975

[Gra.81] Grabowiecki Jean François
'LASSO: un langage pour la description et la simulation de systemes informatiques - Mise en Oeuvre -'
Memoire d'Ingenieur CNAM, Grenoble, 31 Mars 1981

[HeA.77] Hewitt Carl, Atkinson Russel
'Synchronization in Actor Systems'
Fourth ACM Symposium on Principles of Programming Languages. January 17-19, 1977. Santa Monica, Ca.

[HeB.77a] Hewitt Carl, Baker Henry
'Actors and Continuous Functionals'
IFIP Working Conference on Formal Description of Programming Concepts. St. Andrews, New Brunswick, Canada, August 1-5, 1977.

[HeB.77b] Hewitt Carl, Baker Henry
'Laws for Communicating Parallel Processes'
Proc. IFIP-77, Toronto, Canada, August 8-12, 1977.

[Hew.77] Hewitt Carl.
'Viewing Control Structures as Patterns of Passing Messages'
Artificial Intelligence, Vol. 8 No. 3 (June 1977), 323-364.

[HFH.78] Hilfinger P., et al.
'An Informal Definition of Alphard'
Department of Computer Science, Carnegie-Mellon University, February 12, 1978.

[Hil.79] Hill Dwight D.
'ADLIB: A Modular, Strongly-Typed Computer Design Language'
Proc. 4th International Symposium on Computer Hardware Description Languages, Palo Alto, October 8-9, 1979

[HiP.78] Hill Frederick J., Peterson Gerald R.
'Digital Systems: Hardware Organization and Design'
Second Edition, Ed. John Wiley & Sons, New York (USA), 1978

[Hiv.79] Hill D. D., vanCleemput W. M.
'SABLE: A Tool for Generating Structured Multi-Level Simulations'
in [DAC.79]

[Hoa.69] Hoare C. A. R.
'An Axiomatic Basis for Computer Programming'
CACM, vol 12, No. 10, October 1969

[Hoa.74] Hoare C.A.R.
'Monitors: An Operating System Structuring Concept'
CACM Vol.17, No. 10, October 1974

[HoP.75] Hoehne Harold, Piloty Robert
'Design Verification at the Register Transfer Language Level'
IEEE Trans. on Computers, Vol. C24, No. 9, September 1975

[Ish.80] Ishihara K.
'Problems around Logic Design and Design Languages'
International Workshop Problem Areas in Digital Systems Description and Design Tools, La Baule, May 1980

[ISO.73] International Organisation for Standardization
'7 bit coded character set for information processing interchange'
Document No. ISO 646-1973(E), 1973

[Jac.78] Jacquet Paul
'Les types generiques : Propositions pour un mecanisme d' Abstraction dans les langages de programmation'
These de Troisieme Cycle, USMG, Grenoble, France, September 25, 1978.

[JeW.74] Jensen K., Wirth N.
'PASCAL - User Manual and Report'
Springer Verlag, 1974.

[JoS.70] Jorrand P., Schuman S.
'Definition Mechanisms in Extensible Programming Languages'
Proc. AFIPS Fall Joint Conference, Houston, Texas, 1970.

[Kah.79] Edited by Kahn Gilles
'Semantics of Concurrent Computation'
Proc. of the International Symposium Evian, France, July 2-4, 1979 in 'Lecture Notes in Computer Science', No. 70, Springer-Verlag

[KeS.75] Kennedy K., Schwartz J.
'An introduction to the set theoretical language SETL'
Computer and Mathematics with Applications, Vol.1 No. 1, pp 97-119, 1975

[KLM.75] Krakowiak S., Lucas M., Montuelle J., Mossiere J.
'A Modular Approach to the Structured Design of Operating Systems'
Rapport de Recherche IMAG No. 3, Mai 1975

[Kos.76] Koster C.H.
'Visibility and Types'
Proc. of the Conference on Data : Abstraction, Definition and Structure, Salt Lake City, March 1976, Published in ACM SIGPLAN Notices, Vol. 8, Nr 2, 1976.

[Kov.78] Kovats T. A.
'Program Readability, Closing Keywords and Prefix-Style Intermediate Keywords'
ACM SIGPLAN Notices, Vol. 13, Nr. 11, Nov. 1978

[KSM.79] Kawato N., Saito T., Maruyama F., Uehara T.
'Design and Verification of Large Scale Computers by Using DDL'
in [DAC.79]

[LaS.76] Lampson B. W., Sturgis H. E.
'Reflexions on an Operating System Design'
CACM, Vol. 19, Nr. 5, May 1976

[Lef.74] Le Faou Claude
'Un Programme General de Simulation de Circuits Electroniques: IMAG 3'
Electronique et Micro-electronique Industrielle, Octobre 1974, pp39-43

[LeR.77] Le Faou Claude, Reynaud Jean-Claude
'Functional Simulation of Complex Electronic Circuits'
ESSCIRC 77 Conference, Ulm, 19-21 Sept 1977

[LET.78] LETI
'EPISODE : Manuel de Reference'
Rapport LETI, Janvier 1978

[Leu.79] Leung Clement K. C.
'ADL: An Architecture Description Language for Packet Communication Systems'
Proc. 4th International Symposium on Computer Hardware Description Languages, Palo Alto, October 8-9, 1979

[Lip.77] Lipovski G. J.
'Hardware Description Languages: Voices from the Tower of Babel'
Computer, Vol. 1Ø, Nr. 6, June 1977

[Liv.78] Livercy C.
'Theorie des programmes'
Dunod Ed., Paris, 1978.

[LiZ.74] Liskov B., Zilles S.
'Programming with Abstract Data Types'
SIGPLAN Notices 9, pp. 5Ø-59, April 1974.

[LiZ.75] Liskov B., Zilles S.
'Specification Techniques for Data Abstractions'
IEEE Trans. on Software Engineering, SE-1, pp. 7-19, 1975.

[LSA.77] Liskov B., Snyder A., Atkinson R., Schaffert C.
'Abstraction Mechanisms in CLU'
Computation Structures Group, Memo 144-1, MIT, January 1977

[LSW.76a] London R.L., Shaw M., Wulf W.A.
'Abstraction and Verification in Alphard'
Department of Computer Science, Carnegie-Mellon University, March 1976

[LSW.76b] London R.L., Shaw M., Wulf W.A.
'Abstraction and Verification in Alphard :
-Design and Verification of a tree handler (June 1976)
-Iteration and generators (August 1976)
-A symbol table example (December 1976) '
Department of Computer Science, Carnegie-Mellon University, 1976

[Mer.73] Mermet Jean
'Etude methodologique de la Conception Assistee par Ordinateur des systemes logiques : CASSANDRE'
These d'Etat, USMG, Grenoble, France, April 1973

[Mer.78] Mermet J.
'Definition d'un langage prefixe pour exprimer les conditions logiques et les evenements'
Rapport de recherche IMAG No. 143, Decembre 1978

[Mil.73] Miller R. E.
'A comparison of some theoretical models of parallel computation'
IEEE Trans. on Computers, Vol C22, No 8, August 1973

[MoS.77] Moalla M., Sifakis J.
'A Design Methodology for Complex Logical Systems"
Proc. 2nd IFAC - International Symposium on Discrete Systems, Dresde, RFA, March 1977

[MSZ.75] Moalla M., Sifakis J., Zachariades M.
'Un langage d'aide a la conception et a la simulation de systemes complexes'
Rapport de Recherche No. 16, IMAG, Octobre 1975

[MWW.77] McWilliams Thomas M., Widdoes Lawrence C., Wood Lowell L.
'Advance Digital Processor Technology Base Development for Navy Applications: the S-1 Project'
Lawrence Livermore Laboratory Report, UCID-17705, Sept 1977

[NoN.73] Noe J. D., Nutt G. J.
'Macro E-Nets for Representation of Parallel Systems'
IEEE Trans. on Computers, Vol. C22, Nr 8, August 1973

[NSH.77] Navabi Z., Swanson R., Hill F. J.
'User Manual for AHPL Simulator (HPSIM) / AHPL Compiler (HPCOM)'
Dept of Electrical Engineering, University of Arizona, 15 Sept. 1977

[Oak.79] Oakley John D.
'Symbolic Execution of Formal Machine Descriptions'
PhD, Carnegie-Mellon University, CMU-CS-79-117, April 1979

[PBB.80a] Piloty R., Barbacci M., Borrione D., Dietmeyer D., Hill F., Skelly P.
'CONLAN - A formal construction method for hardware description languages: basic principles'
1980 National Computer Conference, AFIPS Conference Proceedings, Vol. 49

[PBB.80b] Piloty R., Barbacci M., Borrione D., Dietmeyer D., Hill F., Skelly P.
'CONLAN - A formal construction method for hardware description languages: language derivation'
1980 National Computer Conference, AFIPS Conference Proceedings, Vol. 49

[PBB.80c] Piloty R., Barbacci M., Borrione D., Dietmeyer D., Hill F., Skelly P.
'CONLAN - A formal construction method for hardware description languages: language application'
1980 National Computer Conference, AFIPS Conference Proceedings, Vol. 49

[PBB.80d] Piloty R., Barbacci M., Borrione D., Dietmeyer D., Hill F., Skelly P.
'An overview of CONLAN - A formal construction method for hardware description languages'
Proc. IFIP Congress 1980, Tokyo & Melbourne, October 1980

[PBB.81] Piloty R., Barbacci M., Borrione D., Dietmeyer D., Hill F., Skelly P.
'Report on CONLAN'
a paraitre

[Pil.75] Piloty Robert
'Functional and Structural Segmentation in RTS III, A Register Transfer Language'
Technische Hochschule Damstadt, RFA, RO 75/2, Mars 1975

[Ram.75] Rammig F. J.
'DIGITEST II: An Integrated Structural and Behavioral Language'
Proc. International Symposium on Computer Hardware Description Languages and their Applications, New York, Sept. 3-5 1975

[Ram.79] Rammig Franz J.
'The Implementation of the Computer Hardware Description Language CAP and its Applications'

Proc. 4th International Symposium on Computer Hardware Description Languages, Palo Alto, October 8-9, 1979

[Rat.75] Ratcliff B.
'A General Approach to Definition and Simulation of Digital Systems'
Proc. SIMULATION 75, Zurich, June 1975

[RoA.75] Rose C. W., Albarran M.
'Modeling and Design Description of Hierarchical Hardware/Software Systems'
in [DAC.75]

[Ros.72] Rose C. W.
'LOGOS and the Software Engineer'
Proc. Fall Joint Computer Conference, 1972, pp. 311-323

[Rot.77] Roth J. Paul
'Hardware Verification'
IEEE Trans. on Computers, Vol. C26, No. 12, December 1977

[Sch.77] Schwandt J.
'An Approach to Use Evaluation Nets for the Performance Evaluation of Transaction-Oriented Business Computer Systems'
Computer Performance, K.M. Chandy and M. Reiser (eds.), North Holland Publishing Company, 1977

[Shi.79] Shiva Sajjan G.
'Computer Hardware Description Languages - A Tutorial'
Proceedings of the IEEE, Vol. 67, No. 12, December 1979

[Sif.77] Sifakis Joseph
'Etude du comportement permanent des reseaux de Petri temporises'
Rapport de Recherche IMAG Nr. 68, Fevrier 1977

[Sto.77] Stoy J. E.
'Denotational Semantics: The Scott-Strachey approach to programming language theory'
MIT Press, 1977

[Sue.80] Suelzle J.
'Using a CODASYL Database for Integration: The Design and Manufacturing Tools for Electronic Circuits'
International Workshop Problem Areas in Digital Systems Description and Design Tools, La Baule, May 1980

[SzT.76] Szygenda Stephen A., Thompson Edward W.
'Modeling and Digital Simulation for Design Verification and Diagnosis'

IEEE TRANS. on Computers, Vol. C25, No. 12, Decembre 1976

[Ten.76] Tennent R. D.
'The Denotational Semantics of Programming Languages'
CACM, Vol. 19, Nr. 8, August 1976

[vaC.77] vanCleemput W. M.
'An Hierarchical Language for the Structural Description of Digital Systems'
in [DAC.77]

[vaC.78] van Cleemput W. M.
'The Role of Design Automation in the Design of Digital Systems'
in Computer-Aided Design Tools for Digital Systems, W. M. vanCleemput Ed., IEEE Catalog No. EHO 132-1, 1978

[VaW.76] Van Wijngaarden A. et al.
'Revised Report on the Algorithmic Language ALGOL 68'
Springer Verlag, New York, 1976

[Vid.74] Vidart Jorge
'Extensions syntaxiques dans un contexte LL(1)'
These de Troisieme Cycle, USMG, Grenoble, France, September 28, 1974.

[Wag.77] Wagner Todd J.
'Hardware Verification'
PhD Thesis, Standford University, Computer Science DEpt. Report No. STAN-CS-77-632, September 1977

[WaP.79] Wallace John J., Parker Alice C.
'SLIDE: An I/O Hardware Descriptive Language'
Proc. 4th International Symposium on Computer Hardware Description Languages, Palo Alto, October 8-9, 1979

[Weg.72] Wegner Peter
'The Vienna Definition Language'
Computing Surveys, Vol. 4, No. 1, March 1972

[Wir.71] Wirth N.
'The Programming Language PASCAL'
Acta Informatica, Vol. 1, No. 1, pp.35-63, 1971

[Wir.77] Wirth N.
'MODULA: A language for modular multiprogramming'
Software - Practice and Experience, Vol. 7, No. 1, January 1977

[WoB.79] Wong S., Bristol W. A.
'A Computer Aided Design Data Base'
in [DAC.79]